A CS1 PEDAGOGICAL APPROACH TO PARALLEL THINKING

by

Brian William Rague

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

August 2010

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Brian William** [redacted]

has been approved by the following supervisory committee members:

[redacted]	, Chair	**5/5/2010**
Robert Kessler	, Member	**5/5/2010**
Clif Drew	, Member	**5/5/2010**
Matthew Flatt	, Member	**5/5/2010**
John [redacted]	, Member	**5/5/2010**

and by [redacted], Director of the School of [redacted]

and by Charles A. Wight, Dean of The Graduate School.

ABSTRACT

Almost all collegiate programs in Computer Science offer an introductory course in programming primarily devoted to communicating the foundational principles of software design and development. The ACM designates this introduction to computer programming course for first-year students as CS1, during which methodologies for solving problems within a discrete computational context are presented. Logical thinking is highlighted, guided primarily by a sequential approach to algorithm development and made manifest by typically using the latest, commercially successful programming language.

In response to the most recent developments in accessible multicore computers, instructors of these introductory classes may wish to include training on how to design workable parallel code. Novel issues arise when programming concurrent applications which can make teaching these concepts to beginning programmers a seemingly formidable task. Student comprehension of design strategies related to parallel systems should be monitored to ensure an effective classroom experience.

This research investigated the feasibility of integrating parallel computing concepts into the first-year CS classroom. To quantitatively assess student comprehension of parallel computing, an experimental educational study using a two-factor mixed group design was conducted to evaluate two instructional interventions in addition to a control group: (1) topic lecture only, and (2) topic lecture with laboratory work using a software visualization Parallel Analysis Tool (PAT) specifically designed for this project. A new evaluation instrument developed for this study, the Perceptions of Parallelism Survey (PoPS), was used to measure student learning regarding parallel systems.

The results from this educational study show a statistically significant main effect among the repeated measures, implying that student comprehension levels of parallel concepts as measured by the PoPS improve immediately after the delivery of *any* initial three-week CS1 level module when compared with student comprehension levels just prior to starting the course. Survey results measured during the ninth week of the course reveal that performance levels remained high compared to pre-course performance scores. A second result produced by this study reveals no statistically significant interaction effect between the intervention method and student performance as measured by the evaluation instrument over three separate testing periods. However, visual inspection of survey score trends and the low p-value generated by the interaction analysis (0.062) indicate that further studies may verify improved concept retention levels for the lecture w/PAT group.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Many individuals have supported my lifelong interest in learning, and more specifically my endeavor to complete a doctoral degree in my late 40s. Thanks to all my teachers and mentors throughout these years that I do not mention by name in this short section.

My gratitude goes back to my master's degree graduate advisors at MIT, Prof. Larry Young and Prof. Charles Oman of the Man-Vehicle Lab, who showed confidence in me and encouraged me to pursue a doctoral degree during a time in which personal challenges prevented me from doing so. More recently, thanks to each member of my committee: Dr. Joe Zachary, Dr. Robert Kessler, Dr. Clif Drew, Dr. Matthew Flatt, and Dr. John Armstrong, my colleague at Weber State. Thanks for your service and support in the completion of this dissertation and your investment in my success. Special thanks to Clif for directing me through the fog of statistical analysis, Joe for guiding me through the overall process and always being responsive, and John for the privilege of working with him on many projects in addition to this one.

My appreciation extends to Prof. Greg Anderson, chair of the Weber State Computer Science Dept., who accommodated my requests for specific classes so that I could complete my educational research. Also thanks to Warren Hill, Dean of the College of Applied Science and Technology at Weber State, who supported my pursuit of a terminal degree. Thanks to Prof. Nicole Anderson, who preceded me as a graduate student in the School of Computing, understood first-hand the challenges that awaited me, and offered timely words of encouragement.

I would like to thank Karen Feinauer for helping me to stay on track with the unwieldy administrative tasks associated with getting a degree.

All my thanks and love to my children: Justin, Maria, Roanna, and Alexis. I know they would like to see Dad away from his computer for possibly a *whole day*. I hope my attaining a degree at this point in my life teaches you that goals are important no matter what they may be or when they are realized.

Lastly, my heartfelt love and thanks to my wife, Gina, the single best human example of a parallel processor. Thank you for juggling all of the daily responsibilities with me, being by my side all these years, and supporting me while I pursued my degree. I will love you always.

CHAPTER 1

INTRODUCTION

The research and development effort described in this dissertation expands both the experimental knowledge base and available classroom resources that support investigations into three important areas of Computer Science:

1. Pedagogical Techniques - An educational study was performed to assess the viability of two separate instructional strategies for introducing core skill sets that reinforce parallel thinking early in the Computer Science curriculum.

2. Content Inventory - A unique, original evaluation instrument was designed to effectively measure student thinking and perspectives regarding parallel system analysis and design.

3. Instructional Tool Development - A novel software visualization tool was created for the classroom to help connect students with the special challenges associated with conceptualizing parallel programs and identifying concurrency in code.

For this work, the three areas of exploration described above form an interdependent whole in that the survey listed in (2) is the evaluation instrument used to generate metrics in the educational study described in (1), and the visualization tool noted in (3) represents one mode of intervention employed in this study.

1.1 Overview

An experimental educational study was performed (Chapter 3) involving three separate CS1-level classes during the 2009-2010 academic year at Weber State University.

The subjects in the study were undergraduates enrolled in the CS1400 *Fundamentals of Programming* course offered by the Computer Science department, where the author/experimenter holds a full-time faculty position.

The experimental method was a two-factor mixed group design in which the classroom intervention mode (control, lecture only, and lecture with visualization tool) is the independent variable, and student comprehension of parallel programming concepts as measured by a customized assessment is the response variable. The same assessment was administered three separate times during the course (Weeks 1, 3, and 9) to each intervention group, providing the repeated non-independent factor in the research method. A summary diagram of the experimental study is given in Figure 1.1.

In preparation for the study, a completely new evaluation instrument called the Perceptions of Parallelism Survey (PoPS) was developed to serve as the formative, customized assessment of student comprehension about parallel concepts (Chapter 4). The PoPS was modeled after the Force Content Inventory (FCI) used for many years in undergraduate Physics education.

Students were given 60 minutes to complete the PoPS, which includes two parts composed of 1) 30 multiple-choice questions and 2) one extensive design essay question. The multiple-choice questions are grouped together into eight separate tasks, each of which targets specific parallel design fundamentals as listed in Table 1.1. The table also provides an overview of the situation/context of each task. Although each problem setting in the PoPS may not represent an everyday occurrence, the intent of the PoPS is to couch questions in a real-world context, allowing the student to rapidly grasp the gist and scope of the problem statement. The PoPS was designed with the beginning CS1 student in mind such that no prior specific programming knowledge was required in order to complete the survey.

To support the intervention level that utilizes an instructional software tool in

addition to traditional lecture, a novel visualization environment geared toward real-time parallel program analysis was written in Java and specially created for this study (Chapter 5). At the visual interface level, this Parallel Analysis Tool (PAT) displays editable Java source code and generates a corresponding UML Activity diagram [26], an artifact used frequently in software engineering. More importantly, the PAT provides students an experimental environment in which immediate feedback in the form of real-time measurements of program speedup help to improve the student's ability to recognize optimal locations for introducing concurrency into code.

All components of the research project described above were developed, managed, and implemented by the author. The three course sections involved in the experimental study were all taught by the author, including in-class administration of the evaluation instrument to the student participants. Despite the oversight of a single individual in the experimental setup, materials, and operation, the results are entirely objective. The PoPS multiple choice questions were graded electronically using standard spreadsheet functions provided by SPSS. To achieve integrated reliability and unbiased scores for the written design question responses, two graders other than the instructor independently evaluated these submissions.

The results of this experiment verify a statistically significant main effect such that student comprehension levels regarding parallel programming concepts as measured by the PoPS improve after the delivery of any CS1 three-week course module when compared with corresponding comprehension levels just prior to the three-week course module (Chapter 6). Specifically, the comparison of PoPS scores between Week 3 and Week 1 test administrations yields a p-value < 0.001, and the comparison of PoPS scores between Week 9 and Week 1 test administrations yields a p-Value $= 0.011$. Figure 1.2 gives the average scores grouped by the test administration repeated measure.

Although the interaction effect between the instructional intervention mode and

the repeated measures did not show statistical significance, the resulting p-value of 0.062 generated by this analysis came very close to the research hypothesis significance level α of 0.05. This outcome encourages at least a visual inspection of the score trends exhibited by each group, as depicted in Figure 1.3.

Pairwise comparisons of these data indicate no significant change in the Lecture Only scores across the three test administrations. However, as revealed in Figure 1.3, significant changes exist between the following three pairs: 1) the pretest and posttest of the Control group, 2) the pretest and posttest of the Lecture w/PAT group, and 3) the pretest and recap-test of the Lecture w/PAT group.

Nonparametric analysis of the written essay design question confirms the parametric results. Namely, a main effect analysis using a Friedman test performed on the repeated measures aggregated from all intervention groups indicates a statistically significant improvement in scores from pretest to posttest. The rise in scores from pretest to recap-test generated a p-value of 0.067.

The low p-value produced by the parametric analysis of the interaction effect suggests that some dependency on instructional intervention technique may be discovered in future work (Chapter 7). The future investigations stimulated by this research project include: 1) a duplicate study to confirm or disprove the results reported here, 2) similar experiments in which the parallel module length and parallel module delivery time within the course is adjusted, 3) a long-term longitudinal study of the student subjects participating in this research, 4) evolution and refinement of the Perceptions of Parallelism Survey into a widely-accepted Parallel Content Inventory, and 5) expansion of the classroom usage and functional features of the Parallel Analysis Tool software.

1.2 Motivation

Integrated circuit manufacturers like Intel, AMD, and IBM are currently producing multicore chips for commodity computational devices. Current advertising campaigns for desktops and laptops emphasize their multiprocessing features. Smaller hand-held and embedded machines also benefit from this technology.

Research in multicore computer architectures continues to move forward, though many challenging issues still remain [16][19][41]. Davis [20] constructed a top ten list of the key problems confronting multicore development; programming methodology was listed as the number one issue. Although the paucity of tools and environments customized for parallel application development presents a challenge for software engineers attempting to produce reliable and efficient parallel programs, it is generally recognized that most programmers do not have the necessary background in concurrency design to fully utilize the current multicore architectures [24].

Coincident with the increased accessibility of multiprocessor platforms is the emerging realization from educators in computer science that "teaching concurrent programming is hard" [8]. There are various factors that contribute to this observation such as the complexity of the topic, the background of the students, and the course sequence of the curriculum. In many CS programs, parallel and distributed computing topics are offered as part of advanced courses at the Junior/Senior level. Textbooks on parallel programming typically target the experienced developer. This late exposure to concurrency may exacerbate the instructional challenge since upper level students tend to focus on tools and solutions (programming constructs/system primitives) rather than the conceptual underpinnings of robust parallel design. In addition, Junior/Senior undergraduates may be somewhat entrenched in sequential ways of thinking as they approach novel parallel programming design problems.

A potential alternative approach proposed by this research is to provide instruction on parallelism early in the undergraduate curriculum, emphasizing conceptual

design rather than implementation issues. Beginning CS students exhibit a sufficient degree of openness and flexibility to fresh ideas prior to adopting specific strategies for tackling program design. The short and long term impact of this early-stage instruction can be monitored using an assessment tool discussed in this dissertation and developed specifically for this purpose: the Perceptions of Parallelism Survey (PoPS). As mentioned in the prior section, the PoPS was modeled after the Force Content Inventory (FCI) used for many years in undergraduate Physics education [33].

Classroom interventions used to clarify computational parallelism for the novice CS student can take many forms. This research focuses on two possible and generally accepted methods of instructing this topic: (1) traditional lecture emphasizing fundamental principles of parallel systems and software design, and (2) traditional lecture coupled with an interactive visualization tool in which the student can experiment with the real-time effects of parallelization on working programs. It should be emphasized early that the intent of this research is not to prove the superiority of one teaching method over the other, nor to confirm the position that more intervention modes (e.g., lecture plus software tool as described in (2) above) are necessarily better.

This research targets the question of whether *any* intervention highlighting parallel computation will enhance student understanding of this important topic at the first-year stage of CS education. Essentially, can the "high-level" conceptual areas of this important subject matter be grasped, processed, and retained by beginning CS students? If one or both of the proposed teaching interventions prove ineffective in reaching students, the question still remains open as to whether a more refined or alternative pedagogical approach would be more productive in communicating parallel concepts to first-year CS students.

Since the CS1 course in most computer science curricula represents the student's first exposure to useful programming strategies, instructors are currently confronted

with reconciling the ubiquitous multicore platforms with the skill set of budding software developers. The role of the application developer in the context of the multicore environment is far from settled, but three points can be stated with some certainty:

1. Parallel and High Performance Computing (HPC) as implemented on modern supercomputing platforms is not new, going back to vector computer systems in the latter half of the 1970s [21].

2. The relevance of concurrency is growing, a recent trend recognized in the ACM CS2008 Curriculum Update, which states "the increased emphasis on concurrency will not be a passing fashion but rather it represents a fundamental shift towards greater attention to concurrency matters" [15].

3. Many computer science programs have traditionally viewed parallel programming as an advanced topic best suited for either graduate students or Senior-level undergraduate students.

To provide focus and perspective on the third point above, Pacheco noted in his 1997 book that "most colleges and universities introduce parallel programming in upper-division classes on computer architectures or algorithms" [55]. Yet Wilkinson and Allen asserted that selected topics from the first part of their 2005 textbook helped to introduce their "first-year students to parallel programming" [71]. This research is motivated by the growing accessibility of concurrent systems and the associated questions of how and when it is most effective for computer science faculty to teach the principles of parallel design and programming, and whether this new pedagogical approach fosters the student's ability to design proper concurrent code solutions later in her academic career.

1.3 Research Objectives

This research presents an educational study to examine the efficacy of introducing fundamental skills that reinforce parallel thinking early in the Computer Science curriculum. As part of this study, a customized evaluation instrument has been designed to measure student comprehension of parallel concepts, and a novel software visualization tool has been created to provide students immediate performance feedback regarding decisions related to parallelizing specific sections of code.

In summary, the core research components include:

1. Design and administration of an educational study using a two-factor mixed design in which the classroom intervention mode (control, lecture only, and lecture with visualization tool) is the independent variable, and student comprehension of parallel programming concepts as measured through customized assessments is the repeated measure.

2. Development of an assessment instrument used to monitor the short and long term impact of the early-stage instruction on student comprehension of parallel concepts. High-level understanding of core parallel computation topics are emphasized in the design of this Perceptions of Parallelism Survey (PoPS).

3. Design and implementation of a new software Parallel Analysis Tool (PAT) to assist CS1 undergraduates to better visualize and understand concepts related to parallel programming.

CS1 as defined by the ACM is an introduction to computer programming course offered to first-year CS students. The primary objective of the class is to sharpen the problem solving skills of prospective developers using software engineering strategies and programming tools. The class is centered mostly on refining and shaping the thinking of these new students rather than stressing detailed computational mecha-

nisms. During this class, important foundational computing issues such as modularization and basic code structure can be addressed from a high conceptual viewpoint.

This research targets students in the early stages of the CS program because the distinctive learning context of the CS1 class gives the instructor an opportunity to present a variety of ideas before any particular one is adopted by the student. Specifically, the student can more readily absorb the perspectives related to parallel thinking before getting "locked in" to a consistently sequential mode of software analysis.

The assumption that students embarking on a CS program will be more open to learning about "parallel thinking" than advanced students who may be in some sense entrenched in previously adopted sequential programming strategies elicits the question of how best to expose beginning CS students to the increasingly important concepts of parallel programming design and implementation. Traditional lecture and/or instructional technology offers some alternatives for presenting these concepts in class.

The two classroom interventions utilized in this project support the main research question, ***"What pedagogical approach introduced in a CS1 undergraduate classroom produces measurable student comprehension of parallel programming concepts?"*** The work presented here will also help to answer a broader question, ***"How can undergraduate programs in Computer Science improve or modify the curriculum to produce better programmers of parallel platforms?"***

1.4 Organization

Chapter 1 - Introduction provides an overview of the research project and a summary of the key results. The current technology and pedagogical challenges related to parallel computation are discussed. The motivation for pursuing the educational

research described in this dissertation is discussed and a summary of the research objectives is listed.

Chapter 2 - Background and Rationale briefly introduces the prevalent strategies traditionally used to teach parallel concepts in the undergraduate classroom, emphasizing both curriculum and instructional technologies, such as software visual analysis tools. Technology-specific topics typically included in upper-level parallel programming undergraduate courses are examined in relation to the concepts more suitable for a lower-level CS1 course on parallel system design and utilization. A rationale for introducing parallel thinking early in the undergraduate program is presented within the context of where parallelism resides in the current standard CS curriculum.

Chapter 3 - Experimental Educational Study gives a detailed description of the motivation, design, and primary components of the experimental educational study. The specifics regarding hypothesis, participants, setting, two factor mixed design, and measurement instrument are discussed. The two interventions employed in this study are highlighted, including information on course content and delivery methods.

Chapter 4 - Evaluation Instrument describes the origins and development of the Perceptions of Parallelism Survey (PoPS), designed to provide metrics on student performance for the experimental educational study. The parallel concepts targeted by each task within the PoPS are covered in detail.

Chapter 5 - Classroom Interventions to Promote Parallel Thinking provides detailed information about the two primary interventions employed in the experimental educational study described in Chapter 3. The instructional interventions used during classroom presentation of the three-week module on parallel computing are outlined. This chapter also presents the detailed design and implementation of the visual tool intervention used in this study called the Parallel Analysis Tool (PAT). Concepts related to recognizing concurrency are revisited in light of the PAT real-time measurements of program speedup. The purpose and operation of the PAT are

investigated.

Chapter 6 - Results furnishes a description and presentation of the experimental study data. Statistical techniques are applied in order to discern trends in the information acquired from the administration of the evaluation instrument. These results are examined to make accurate assessments of student comprehension levels of key parallel concepts for the purpose of evaluating the intervention strategies utilized in the experimental study.

Chapter 7 - Conclusions provides an analysis of the educational study results, primarily focusing on information gathered from the study that might be used to benefit further research into pedagogical approaches and appropriate curriculum placement for courses presenting parallel concepts to CS undergraduates. This chapter also summarizes the research presented in this dissertation, situates this work in the context of computer science education, and suggests future extensions to this research.

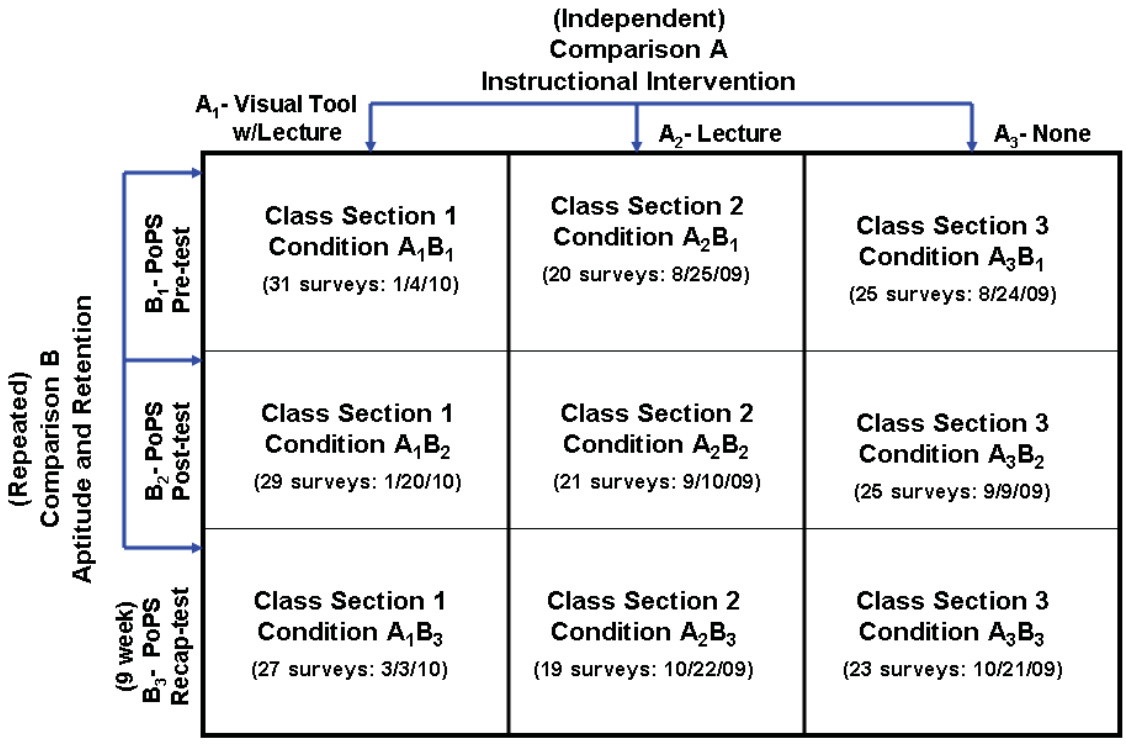
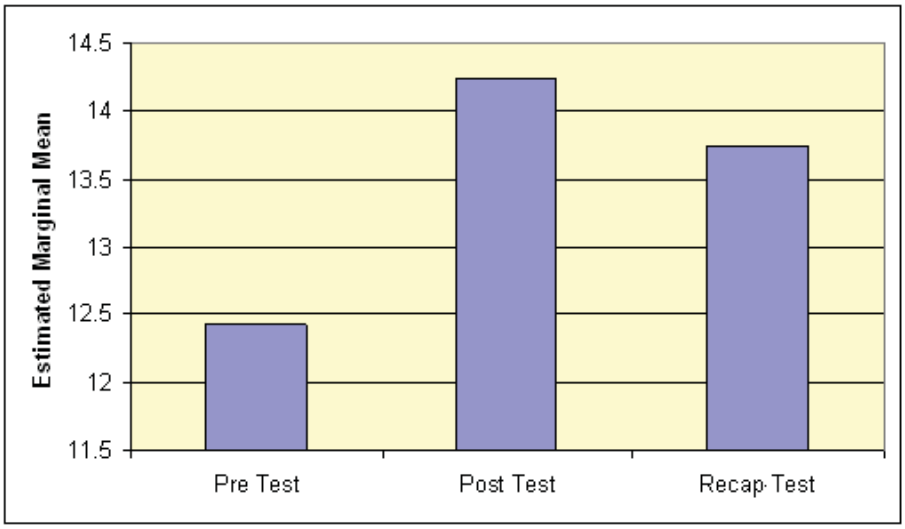


Figure 1.1: Two-Factor Mixed-Group Design



Estimates

Measure:MEASURE_1

Test	Mean	Std. Error	95% Confidence Interval	
			Lower Bound	Upper Bound
1	12.422	.404	11.618	13.226
2	14.245	.494	13.263	15.228
3	13.738	.451	12.840	14.636

Figure 1.2: Repeated Measure Means

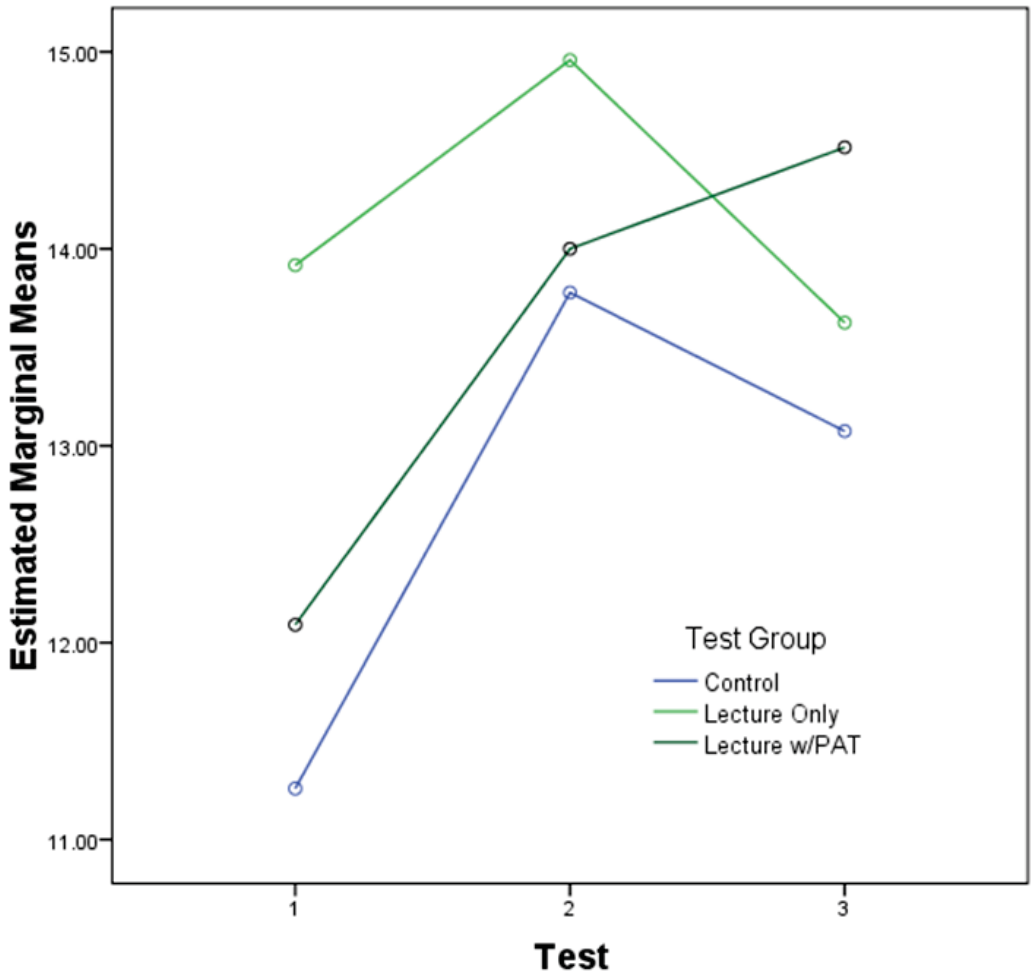


Figure 1.3: Profile Plot (Test vs Experimental Group)

Table 1.1: PoPS Concept Coverage and Scenarios

Task	**Parallel Concepts**	**Scenario/Setting**
I	Serial vs. Parallel Computation/Temporal Dependency	Sequence of Arithmetic Operations
II	Multitasking/ Context Switching	Student texting while attending lecture
III	Resource Management/ Efficiency	Check out books from library
IV	Monte Carlo Simulation/ Time vs. CPU Tradeoff	Darts thrown at a collection of dartboards
V	Master-Worker Configuation/ Communication	Alphabetizing individual words
VI	Mandelbrot example/ Load Balancing	Waiting in line for antique appraisal
VII	Lateral communication/ Data Decomposition	Card game
VIII	Application of Amdahl/ Gustafson Laws	Theoretical ball delivery mechanism
Design Question	Grouping Tasks/ Ordering Tasks	Crime Lab Photo Recognition System

CHAPTER 2

BACKGROUND AND RATIONALE

Since the middle of the 20th century, dedicated research groups and computer companies have been actively investigating the design and realization of proprietary parallel computing solutions. However, it was the early 1990s that marked the introduction of parallelization into mainstream computing strategies, and it has only been fairly recent that the prevalence of multi-processor platforms has compelled software developers to ultimately confront the potential and prospects of "parallel thinking".

The *Sourcebook of Parallel Computing* states that "the Center for Research on Parallel Computation (CRPC) was founded in 1989 with the goal of making parallel programming easy enough so that it would be accessible to ordinary scientists" [21]. One would assume this statement excludes computer scientists from that group since it is application and system developers who are charged with inventing the tools and devising programming strategies that will allow high-performance multicore architectures to be fully utilized by the "ordinary scientist," even as these hardware platforms rapidly evolve in an effort to breathe life into Moore's law.

Advances in networking have allowed inexpensive computer clusters to emerge in small laboratories [37], and scientists from all disciplines are often buried in volumes of raw information while data-starved CPUs idle away on recently purchased or constructed parallel systems. The science of parallel computation envisioned by CRPC will be realized not only when message passing, I/O interfaces, and parallel algorithm libraries are standardized, but also when software application developers hone a sharper sense for recognizing the benefits and tradeoffs of concurrency to the

point in which parallel thinking becomes second nature. Acquiring this skill set is no different than learning and applying Edsger Dijkstra's "divide and conquer" strategy to modularizing programs [18], an approach commonly taught in the first year of a computer science program and which all professional programmers are expected to know.

This chapter explores when instruction on parallel computing typically occurs in the current CS curriculum and some of the core concepts highlighted in these courses. This discussion provides the context for proposing areas of parallel design that might be suitable for a lower-level CS1 course on the topic. The instructional tools that support advanced investigations of parallel programming are contrasted with analysis tools that might best serve first year undergraduates. Throughout, a rationale for introducing parallel thinking early in the CS undergraduate program is developed with the objective of answering to some degree the following question: *At what points during an undergraduate program will students most readily accept, absorb, and critically assess the techniques particular to parallel design?*

2.1 Parallelism and the CS Curriculum

Most collegiate programs in Computer Science offer an introductory course in programming, primarily devoted to communicating the foundational principles of software design and development. This first-year course typically presents widely accepted software development methodologies for solving problems within a discrete computational context. Logical thinking is highlighted, guided primarily by a sequential approach to algorithm development and, in most cases, illustrated and put into practice by using the latest, commercially successful programming language.

At this early stage of the CS curriculum, students are arguably very receptive to instruction about diverse analysis and design methods, despite their limited capabilities in the area of implementation. In the programming fundamentals course,

the first notions about problem solving and algorithmic thinking within a computer science context begin to take root, with little interference from any highly developed coding biases or habits that typically accompany the thinking of seasoned programmers. In response to the most recent developments in accessible multicore computers, instructors of these introductory classes may wish to include some broad discussion and training about how to design workable parallel code to effectively utilize the processing power of these newer architectures. Various strategies exist for integrating concurrency into the CS1 course, including a high-level event-driven programming approach to threads and object interaction [12] [66], data parallelism [23], and background timers [60]. A decade ago, a specialized course examining the inherent challenges of concurrency like deadlock and mutual exclusion was proposed for high school students [5].

The immediate response of some CS faculty to this early undergraduate exposure to parallel concepts may be predominantly guided by the current conception that parallel programming is inherently difficult and requires extensive knowledge of system level primitives, specialized library routines, or dedicated parallel programming languages. Indeed, teaching concurrency to Junior/Senior level students can often be a formidable task simply because launching a workable concurrent program demands a solid background in computing fundamentals. Yaodong Bi admits as much in his observation: "Concurrency and inter-thread (and inter-process) synchronization have become an integral part of today's computing sciences curriculum. However, teaching concurrency and inter-thread synchronization is difficult" [8].

Table 2.1 shows the course titles and descriptions of parallel computing classes offered by five major CS programs and the associated pre-requisites needed to enroll in these courses. Table 2.1 is illustrative of the level of expertise required for the student to gain some exposure to parallel computing, either from an architectural perspective, through application development, or from some combination of the two.

A search of the MIT course list using the keyword *parallel* uncovered only the course listed in Table 2.1. Note that MIT's 6.046J is an algorithms class that includes parallel computing as one of many advanced topics. In addition, as denoted by the course description, parallel computing may not even be covered, a decision assumed to be at the instructor's discretion. Still, some basic prerequisites concerning Introductory algorithms and Discrete Math are required before the undergraduate is allowed to take this course.

Stanford offers a basic undergraduate course in the software systems area dedicated to understanding and working with parallel platforms, with a direct acknowledgment in the course description that "most new computer architectures are parallel." The course provides a comprehensive treatment of the topic, and warns of "significant parallel programming assignments." No specific parallel language is mentioned, but with references to SPMD and message passing, one can assume that some variation of MPI used in conjunction with C/C++ will constitute the development API. The core conceptual underpinnings of parallelism is addressed in topics such as locality, implicit vs. explicit parallelism, and shared vs. non-shared memory. Most of the other areas of study listed in the course description are tilted heavily toward implementation issues and require some solid CS background from the enrolled student, as revealed explicitly by the extensive list of prerequisites.

CMU provides an upper division elective course combining both the hardware and software considerations involved in developing parallel solutions. As with Stanford's parallel computing course, synchronization mechanisms and threads are listed as an important topic of discussion. As indicated by the prerequisite list, CMU students must still have a foundation in programming prior to enrolling in this course. UC Berkeley places a significant emphasis on the hardware components of parallel processors, requiring students to be well established in the area of computer architecture, to the extent that a Graduate level course on this topic is a prerequisite.

The University of Utah's initial undergraduate treatment of parallel computing is through the 3000-level Computer Organization class. The emphasis is clearly on the architecture of parallel machines and how these contrast to or overlap with other hardware configurations. Some programming skills and background foundation in computer systems are required, and, as with the MIT algorithms course, several prerequisites exist that prepare the student to better appreciate parallel architectures. The remaining two offerings at the University of Utah are graduate level courses, which essentially presupposes student attendees relatively well versed in the CS discipline. The Formal Methods class utilizes parallel processor memory models primarily as a system design example to which the student may apply predicate calculus and state enumeration techniques. The Parallel Computing and High Performance Computing class provides the most overt, dedicated treatment of the subject, albeit with relatively experienced students who have a vested interest in the topic. The University of Utah also offers a special topics class that may periodically delve into parallel computing topics.

A prevalent theme that emerges when reviewing this small sample of course offerings on parallel computation is the marked interdependence between hardware and software issues. None of the classes above dedicated solely to parallel systems could be neatly categorized either as a purely software or as a purely hardware course. The title of CMU's upper division course, "Parallel Computer Architecture and Programming," concisely expresses this reality. A reasonable conclusion to be drawn is that *proper academic treatment of parallel computing needs to include discussion of both underlying architecture and program development.* In the pedagogical approach to parallel systems, as in real life, hardware leads software and architecture drives algorithms.

Despite the fact that for many years software developers have been relatively free to ignore substantial improvements in computer hardware, from the fabrication tech-

nologies up through the mid-80s to the architectural advances since that time [31], this liberty is now being tempered by the increased accessibility of parallel platforms. Programmers can no longer write platform-agnostic code, insulated by virtual machines or API standards. A parallel mode of thinking necessarily requires cognizance of the target machine, primarily in the number of processors available and the hardware configuration of those processors.

Conversely, low-level techniques used to exploit parallelism in microprocessors are not directly applicable to writing robust parallel programs. This stark distinction is expressed best by Hennessy and Patterson [31]: "...the important qualitative distinction is that [thread-level] parallelism is identified at a high level by the software system and that the threads consist of hundreds to millions of instructions that may be executed in parallel. In contrast, instruction-level parallelism (ILP) is identified primarily by the hardware, although with software help in some cases, and is found and exploited one instruction at a time" (p. 530).

A good example of this disparity in development strategies is the widely-accepted philosophy in computer design to make the common case fast. Whenever possible, the system designer should target frequent cases for performance improvement. This approach does not necessarily apply in all situations when deciding which high-level programming statements should be distributed among a bank of parallel processors. Specifically, initializing memory locations in an array is a fairly common occurrence in programming circles. Yet, because of the intrinsic overhead of establishing and utilizing thread-level parallelism, there would be significant diminishing returns if these extremely fast initialization statements were executed in parallel.

Given the prior discussion, it would appear that the depth of knowledge required of students to grasp the intricacies of parallel computing would preclude this topic from being addressed in an introductory CS programming course. In fact, this research study is part of an effort to determine if this initial course is the optimum

venue to promote an appreciation for architectural resources and allow students the opportunity to contemplate the benefits and drawbacks of parallelizing code within this context. Instead of tacitly supporting an implicit divide between hardware and software considerations, the introductory programming course can be an occasion for the instructor to forge a connection between these two sides of CS that is absolutely essential for understanding proper parallel design.

2.2 Tools for Parallel Instruction

Regardless of student expertise, educators in Computer Science generally agree that teaching concurrency can be difficult. As described above, in many CS programs, parallel and distributed computing topics are offered as part of advanced courses at the undergraduate upper division or graduate level. Textbooks on parallel programming typically assume some experience with programming and/or computer systems.

Concerted efforts have been made since the late 1990's to integrate parallel computing topics in the undergraduate CS curriculum [1][42][54]. Concurrency is often taught in conjunction with distributed computing, and undergraduate students enrolled in this kind of course are typically in their final year of study and are expected to have some "previous exposure to classical concurrency issues at the operating system level and basic knowledge of computer networks" [17]. The implication is that students are not equipped to comprehend or correctly manage parallel processes until sufficiently conversant with system level software.

To assist in the instruction of parallel concepts, various visual tools such as those described by Stasko [65], Kurtz [43], Carr [13], and Bi [8] have been used to investigate the behavior of threads and processes in running programs. However, these tools are designed primarily for classes beyond CS1, not for the beginning student. Carr states, "There are few pedagogical tools for teaching threaded programming" [13]. Indeed, threads currently represent the de facto primitive for programmers delving

into parallel application code, and, as a consequence, either relegate instruction on concurrency to the latter stages of the CS undergraduate program or awkwardly impose specialized modules about threaded programming techniques into courses early in the curriculum.

In this section, two of the visual tools mentioned above will be evaluated in terms of their efficacy in teaching parallel (multithreading) concepts and their appropriate placement within the standard CS curriculum. Although admittedly not an exhaustive analysis of all available research or commercial tools focused on parallelism, this discussion does address two examples that are strongly representative of the state of instructional technology that currently supports the teaching of concurrency. These discussions will also shed light on the assertion that such tools necessarily target students with some background in computing systems and programming, and therefore are not suitable for novice students just beginning a CS program.

In their paper "A Visual Tool for Teaching Multithreading in Java," Yaodong Bi and John Beidler describe a visual software tool that pictorially represents thread behavior in a running Java program [8]. They suggest that allowing students to visually observe executing threads will further their understanding of concurrent mechanisms. Graphically charting the progress of underlying threads or processes certainly offers a perspective which can bolster student comprehension of thread behavior and interaction. This point of view is supported by 1) the recognition that students exhibit mixtures of different learning styles (e.g. visual, kinesthetic, and auditory) and 2) the general acceptance of the UML standard, which effectively provides visual diagrams as a means of grasping complex software engineering design strategies.

Indeed, Bi refers to displaying thread execution "like a UML sequence diagram." The element of time inherent in multithreaded program execution would logically lead to drawing vertical timelines, but the similarity to the UML sequence diagrams ends here. The UML sequence diagrams depict the interaction of objects during the execu-

tion of one or more use case scenarios, with object lifelines representing the evolution of each participating object. Threads and semaphores may be considered objects, but primarily as low-level implementation constructs. Therefore, it would be more accurate to refer to the visual display of this tool as a "thread timing and interaction diagram." However, the proposition that relevant modeling diagrams from a software engineering context, specifically the UML, can be used to support analysis of a program's concurrent behavior is a worthwhile consideration when designing visual tools. Coupling code with a corresponding diagram clearly outlining the program structure and purpose *within the same view* can significantly assist the learning process.

The paper describes the classical producer/consumer problem using terminology related to low-level concurrency and programming primitives, e.g., semaphores, mutexes, threads, deadlock, runnable state, and synchronization. The tool's reliance on such concepts would require students utilizing the tool to be initially instructed on the meaning of and interplay among these constructs; otherwise, the tool's pedagogical value diminishes. One statement from the paper is especially telling about the assumed expertise level of the student user: "To help students in understanding the execution of multithreading programs, the system visually displays for each thread the information related to the creation and termination, the state changes, name changes and priority changes, and the inter-thread synchronization."

The graphical display, shown in Figure 2.1 depicts the producer/consumer example given in the paper. The figure offers the student a low-level perspective of thread communication, with the producer thread represented by the right hand vertical line and the consumer thread depicted by the left hand vertical line. The ability to replay events of interest in slow motion can assist the instructor in providing real-world evidence of the consequences and impact of employing multiple threads in code, and allow the student to see the program unfold. As a result, the student will better appreciate the code in the producer and consumer implementations. This

experience of threads-in-action would also hopefully solidify the students conceptual understanding of why threads even exist, which is the first hurdle to overcome when instructing upper division undergraduates about parallel processing.

Apart from the operation or functional features of the tool, Figure 2.1 provides a fairly typical view of the standard thread monitoring interface densely packed with information. The color coding of the numerous events and states listed in the graph legend, coupled with the small-font annotations (e.g., *semaAcquireCompleted@27854*) require some very focused parsing from the student/learner in order to make sense of the considerable amount of information contained in a single screenshot. The degree of intuition and immediate data recognition a student achieves with this brand of thread visualization is highly dependent on the student's familiarity with the concept and behavior of threads. In fact, the authors of this tool have recently proposed a "threads early" approach by introducing dedicated modules about Java threads into their CS 2 and CS 3 courses at the University of Scranton, presumably to sufficiently prepare students for assignments utilizing this visual software [9].

Once the displayed information is deciphered, students can gain a deeper appreciation and understanding of thread states, how these states are interdependent, and the allocation of CPU cycles during the execution of a program. Students will also have an opportunity to investigate more fully the role of mutexes and semaphores in the execution of the program. Overall, the tool provides sufficient flexibility to be able to dissect and examine the various parts of a standard multithreading example and could be used not only in a CS 2/CS 3 course supplemented by a detailed treatment of Java threads as proposed by the authors, but also in a course on Operating Systems.

The pedagogical value of the tool is compromised, however, by the non-deterministic nature of multithreaded programs. For example, if the instructor wanted to illustrate deadlock using the tool, it is not clear how this situation could be purposefully gener-

ated, other than by rewriting the source code incorporating significant sleep periods or other artificial delays to essentially "force" a deadlock. Even after applying these manual strategies at causing a deadlock, there is still the finite probability that the program will avoid the deadlock because of slight system variations.

Figure 2.2 reveals how including one additional concurrency primitive, a mutex represented by the center vertical timeline, can increase the complexity and adversely affect the overall readability of the visual interface.

As the number of threads and synchronization objects increase, the instructional value of the visualization will most likely decrease due primarily to the increasing number of timing bars displayed on the graphical interface. Figures 2.1 and 2.2 reveal that the labeling of events, although informative regarding target object and time of occurrence, can potentially clutter the main display if there are more than a few thread or synchronization objects of interest. The predisposition toward information overload is fairly common among visual multithreading instructional and analysis tools, though not often discussed.

Providing user options to toggle labeling text or select specific threads on the main display, although not described in the paper, would help alleviate this problem. At the CS2 and CS3 level of instruction, a simplified interface will have more overall educational impact given the fundamental complexity of multithreading. Utilizing straightforward multithreading scenarios is sufficient in helping beginning CS students understand the essential concepts. Also, it should be noted that increasing the number of observed threads and synchronization objects may negatively affect the performance of the tool depending on the memory and CPU capabilities of the underlying system. As with most graphical systems, increased load may degrade response times to user input or information retrieval and display.

In 2003, Steve Carr, et al. designed a multiplatform pedagogical tool for multithreaded programming named ThreadMentor [13]. Similar to the visual tool just

described, ThreadMentor provides a visualization system giving detailed information about every thread and every synchronization primitive employed during execution of a multithreaded program. Unlike Bi's software tool, this system is targeted for students in an Operating Systems or Concurrent Programming course.

In the early part of the paper describing ThreadMentor, Carr makes some general, but germane observations about parallelism using threads: "(1) multithreaded programming requires a new mindset; (2) the behavior of a multithreaded program is dynamic, making debugging very difficult; (3) proper synchronization is more difficult than anticipated; and (4) programming interfaces are usually more complex than necessary, causing students to spend time in learning the system details rather than the fundamentals." In the remainder of the paper, Carr proceeds to tackle each of these challenges head-on by proposing a "coherent and unified environment" which captures the dynamic behavior of a threaded program and allows for event playback so students might better grasp the intricacies of synchronization primitives.

One of the key points expressed by both Carr and Bi is that their respective visual systems do not require instrumenting the original program with extra statements or directives. The philosophy is to allow the code to execute "as is," unencumbered by additional bindings to the monitoring tool which might interfere with the behavior or performance of the program. This perspective suggests an analogue in the often cited observer effect in physics, in which the act of observation is recognized to have some altering influence on generated experimental results. Indeed, for pedagogical reasons alone, it is beneficial for student comprehension of thread mechanics or parallel behavior if additional programming commands unrelated to the code's original objectives are avoided. In the case of a CS1 level student learning a new programming language, it is especially critical that only native statements from that language occupy the learner's attention, even when parallel analysis is being applied to the codebase by a software tool either dynamically behind the scenes or offline to a storage location

in order to be replayed later.

Consequently, the designers of ThreadMentor attempted to insulate students from the system details related to multithreading by employing a class library that uses textbook syntax. Although ThreadMentor supports C/C++, the small sample code in Figure 2.3 shows the use of a Thread class implementation, which closely models the approach employed in Java. Indeed, many of the methods defined for this Thread class mirror those in the Java Thread class, e.g., `Begin()` instead of `start()`.

This effort to minimize possible distractions to the student's learning due to unnecessary system level constructs is noteworthy. If the code used in the visual tool corresponds closely with textbook examples, then the narratives in the book and the presentations in the classroom more readily reinforce the course objectives. A possible approach would be to use straightforward, unobtrusive code annotations that trigger appropriate parallel processing and analysis from the supporting visual software tool.

Carr draws an important distinction between thread visualization systems developed for debugging and performance purposes and those pedagogical tools designed to clearly illustrate low-level thread behavior and interaction. At the time of publication, Carr claims in his paper that "ThreadMentor is perhaps the only comprehensive pedagogical system available for teaching and learning multithreaded programming." Commercial thread debuggers like Borland's older Optimizeit system presupposes that the user is knowledgeable about thread structure and mechanics. Pedagogical tools do not have the advantage of a user experienced in system level concepts, and must provide an interface that not only reports thread actions but also packages the information so that it can be rapidly understood with a reasonable learning curve. As an addendum to Carr's comment above, it is also difficult to find multithreaded pedagogical tools whose instructional impact has been measured using an educational study, as is proposed in this research.

Whereas Bi's monitoring tool only provides offline analysis of thread interactions

after the target program has executed, ThreadMentor is designed to also generate information on-the-fly during code execution. However, this attempt at real-time analysis comes with measurable communication delays incurred because of the portability of the class library and the asynchronous message queue latencies between the class library and the visualization system. The instructional benefits obtained by allowing students to observe threading behavior as it happens are marginal when compared to the effort required to design a dynamic visual tool truly synchronized with underlying system activities.

Offline analysis (studying thread interactions after execution has completed) is more than sufficient to achieve a thorough understanding of thread behavior for that executing process. Even if the target application has already completed, nothing is lost in program analysis if students are given the capacity to "see" the code's specific instructions in a graphical representation and, if necessary, to immediately replay the program execution in slow motion. Classroom examples of multithreading tend to have short run times, so students will not have an inordinate wait to view the results of the program execution. In addition, offline investigations of multithreaded code sufficiently compartmentalizes the program development cycle from the visualization/analysis step. Since the conceptual demands required of these two important activities are distinctly different, this offline strategy encourages the student to focus on each task with the appropriate mindset. For example, a student receiving a run-time error while immersed in observing the details of thread/mutex communications will be abruptly diverted from the analysis activity and, as a result, will ultimately need to refocus her efforts back on the program development and logic. One significant drawback to offline analysis is that a forced halt of the program would be required to view data in the case of deadlock.

ThreadMentor's visualization system is event-driven. Figure 2.4 shows ThreadMentor's main window when the visualization system is activated by a user pro-

gram. For a currently running program, the student/user can essentially drill down to more detailed levels of information about the supported synchronization primitives listed on buttons along the right-hand side of the main window. These primitives include mutex locks, semaphores, monitors, readers-writers locks, barriers, and synchronous/asynchronous channels. For example, selecting *Semaphore* will show in the large white display area all semaphores created so far in the executing program. The student can then select a specific semaphore from the list to obtain more detailed information about that primitive.

The lower right portion of the main window allows the user to control the execution speed of the running program, to pause and resume that program, and to step through the thread management and synchronization activities. Actual visualizations of thread lifecycles are furnished by the thread management functions in the lower left of the main window.

Figure 2.5 shows a snapshot of the Thread Status window while a qucksort program is executing. Threads are assigned symbolic names by the user and the current state of each thread is indicated using an appropriate icon. Note that the current visualization indicates the lowest-level threads are currently executing and performing the required partitioning activities associated with the quicksort algorithm.

Figure 2.6 shows the execution history of all participating threads. Colors along the history bar are used to indicate the thread state, either running, joining, or blocked by a synchronization primitive. Tags along the history bar represent synchronization events. Clicking a tag produces the source code location in which the synchronization primitive originates. Carr claims that this view is "perhaps the most commonly used ThreadMentor window because, for every thread, it provides the state, the relative time of execution and description of a synchronization event with history bar tags, and a link between a tag and the corresponding source statement."

This last feature is especially important since student programmers must develop

skills that help them relate the effects of their program statement choices on the hidden world of thread management and parallel processes. Debugging parallel code is made difficult by the relative disconnect between high-level statements and the associated threading events. Students should have at least a first-order notion of the consequences of source directives for the primary purpose of parsing out execution scenarios realized at the system level. However, this modest level of student proficiency in developing multithreaded programs cannot prevent the occasional deadlock or unpredictable result. Indeed, even with the insight provided by these visual tools, constructing 100% reliable thread-safe programs is a difficult venture primarily because of the inherently nondeterministic nature of thread primitives.

A comprehensive and compelling argument regarding the inadequacies of threads as the core primitive for developing parallel code was put forth by Edward Lee in his May 2006 article, "The Problem with Threads" [45]. Indeed, either because of the predominant textbook content on the subject or simply the lack of alternative models, an association has been forged in the minds of many computer science faculty that teaching parallel/distributed design must include a thorough treatment of threads and threading mechanisms. In the middle part of this past decade, software engineers forced to contemplate how to fully harness the performance potential emerging from multicore platforms turned reflexively to the only programming tool ostensibly available. This prevailing bias for threads and the associated cautionary tone that developers should place a high priority on polishing their multithreading skills are concisely captured in the following quotation from a May 2005 software periodical article on the subject: "It's clear that the future of all desktop software development is threaded. So, if the advent of hyper-threading didn't get you to think about using threads in your client applications, this is surely the right time to dip your toe into programming for parallel processing" [10].

This unfortunate implicit connection between parallel computing and threads

leads to unnecessarily telescoping a fairly broad topic to the minutiae of system level implementation. An equally important consideration is that threads may not even be the best tool for development of parallel programs. The real utility of the thread concept can be traced to OS design and improved hardware implementations. Threads motivated the development of a more flexible Unix kernel [67] and more efficient execution techniques in processor design, most notably extending superscalar performance into fine-grained temporal multithreading and simultaneous multithreading [70].

Threads, with their associated contention and synchronization concerns, present a fairly complex picture of parallel coding to the beginning CS1 programmer. Although tools which support visualization of multithreaded program execution and the various synchronization objects and monitors may be of great benefit to the Junior/Senior CS student, there is a finite likelihood that first year CS student programmers would become overwhelmed and possibly discouraged by this level of detail. Lee has stated outright that because of the inherent unreliability of thread behavior between program executions, the thread model may not be best for parallel application design: "Threads are seriously flawed as a computation model because they are wildly nondeterministic." Lee issues a broader indictment regarding designing multithreaded code: "A folk definition of insanity is to do the same thing over and over again and expect the results to be different. By this definition, we in fact require that programmers of multithreaded systems be insane." An alternative to parallelization through threading is the current investigation into transactional memory (TM), which aims to make the programming model simpler, freeing the developer from lock management tasks [11][62]. Sun's recent Rock processor offers TM support.

After essentially dismantling the utility of threads in creating parallel programs, Lee proposes that the answer to this unruly nondeterministic problem is to focus programming language development on coordination languages that introduce new syntax orthogonal to the more widely established programming languages. Unlike

threads-style programming, which attempts to corral a wildly nondeterministic interaction mechanism, coordination languages would afford the programmer the opportunity to initially model deterministic processes and then judiciously and explicitly introduce nondeterminism only where needed. Examples in his article include a rendezvous-based coordination language with a visual syntax. Lee envisions a visual style of programming emphasizing process coordination overlaying the more finely-grained computation expressed in more conventional languages. The acceptance of a visual environment, he states, is made possible by programmer endorsement of the "UML–properly viewed as a family of languages, each with a visual syntax–[which] is routinely combined with C++ and Java." The final word on threads is best expressed by Lee: "They must be relegated to the engine room of computing, to be suffered only by expert technology providers."

2.3 Early Exposure to Parallel Concepts

The previous section details the applicability and popularity of rendering specific thread behavior and general parallel functionality in a graphic visual environment. This is a key discovery derived from past research and development that should not be cast aside when considering instructional strategies for teaching parallel concepts to CS1-level undergraduates. The subject of what constitutes an appropriate course in parallelism and the classroom tools that promote learning on this topic is still up for debate. The current educational challenge is best summarized by David Patterson's observation: "I wake up almost every day shocked that the hardware industry has bet its future that we will finally solve one of the hardest problems computer science has ever faced, which is figuring out how to make it easy to write parallel programs that run correctly. . . . ***We don't even know for sure what we should be teaching***, but we know we should be changing what we're teaching"[56].

The CS2008 Curriculum update recommends a Parallel Computation class that

cuts across several advanced courses and is a suitable prerequisite to a capstone project class focusing on creating a complex parallel system [15]. Alternatively, if a treatment of parallel concepts were moved to the very beginning of a student's CS program, then the content and delivery of that course or module requires careful thought and planning. Novel issues arise when programming concurrent applications which can make teaching these concepts to beginning programmers a seemingly formidable task. The instructional challenge is to make parallel code design principles not only accessible, but comprehensible to students at the very beginning of their Computer Science education.

As mentioned in Section 2.1, various strategies have been devised to integrate concurrency into a CS1 course. One such approach implemented at Williams College and later at Pomona College by Bruce, Danyluk, and Murtaugh emphasizes a high-level event-driven programming treatment of threads and object interaction [12]. This course introduces Java event-driven programming on the first day of class and makes extensive use of graphics and animations. Libraries are provided to shield beginning students from a few unfamiliar aspects of the Java language such as exceptions. Using these libraries, students are writing programs that create and execute separate processes by the fourth week of the term.

Students then begin to work on a simple version of the Frogger^{TM}game, in which a player uses mouse clicks to safely guide a frog across a four-lane highway. The car, frog, and lane objects are each controlled by separate threads, which the authors claim is a natural fit with the student's intuition about independent interactive entities. Conversely, Bruce et al. state, "If we avoid introducing concurrency early, we force students to learn to sequentialize naturally concurrent processes" [12]. Thread usage and implementation becomes more prevalent as the course progresses, necessitated by the emphasis on the Java programming concurrency model. Apart from the immediate visual rewards obtained through this game development approach,

beginning CS1 students are not positioned to fully appreciate the complexities and non-determinism inherent in threads.

The success of instructing concurrency concepts at this introductory level was measured informally by asking students to rank the lab activities on educational value, difficulty, and fun using a scale of 1 to 5. Laboratory projects involving concurrency were rated higher in all three categories than other projects, with noticeable differences in the ratings for difficulty and fun.

Over 10 years ago, Lynn Andrea Stein from MIT suggested that CS1 instruction should more diligently adhere to the "computation as interaction" model represented by real-world software implementations like spreadsheets, video games, and web applications. She noted that "we teach our students a single-thread-of-control static problem-solving view of the role of the computer program: computation as calculation" [66]. The author's focus is on establishing interaction patterns rooted in concurrency as the next level of abstraction beyond traditional functions and objects.

Stein contends that providing students an appreciation for interacting processes at the CS1 level pays dividends later in the curriculum: "For example, an operating systems course no longer needs to teach both the idea of concurrency and the mechanisms by which it is implemented; students will have been living in a concurrent computational world from the very beginning."

Like Bruce's approach described above, Stein's CS1 course relies on a GUI framework that allows students to visualize the behavior of their Java code. In the beginning of the course, students write an entity to control one end of a virtual seesaw. As their knowledge base of Java functionality is expanded, students are guided through a program emphasizing the internal structure of a computational entity, how entities are tied together, and ultimately culminating in a treatment of a variety of canonical architectures for distributed systems including RMI.

For both pedagogical approaches, concurrency concepts are interwoven through-

out the entire course in conjunction with learning a programming language. Stein targets a more high-level system interaction model whereas Bruce focuses more on the low-level thread mechanisms that underlie concurrent behavior. Stein's assessment of student learning is limited to observations like "course evaluations have been positive," and "students going on to subsequent course work report that their experience with this material was extremely helpful."

A constructivist view of learning theory asserts that students will either assimilate or accommodate new concepts based on their past experiences. Prior knowledge is typically structured according to working models or "thinking" frameworks. Bain states that if instructors do not provide meaningful frameworks to students, they will attempt to form their own (with uncertain results) [4]. In conjunction with colleagues, the author has investigated various manifestations of "thinking" frameworks that infuse computer science, namely mathematical, abstract, and computational thinking.[50] These frameworks are motivated by commentary from computer scientists that attempt to distinguish the discipline's theoretical and design strategies from the programming task, and are built on previous research investigating different ways to teach discrete mathematics to CS students [51][52].

The question of where to place the topic of parallelism within a CS curriculum naturally leads to identifying at what points during an undergraduate program students are most ready to accept, absorb, and critically assess the techniques particular to parallel design and development. As students progress toward the baccalaureate degree, it is reasonable to assume that strategies related to solving problems in the discipline are gradually culled and solidified. Assisting the student in establishing an effective set of tools is the intended outcome of all academic departments, and results in the core skill set that allows the graduating student to embark on a professional career.

Specifically, students in upper level CS courses have adopted ways of abstracting,

modeling, and implementing programming solutions from requirement documents based primarily on the methodologies taught in their initial fundamentals classes. CS1 as defined by the ACM is an introduction to computer programming course offered to first-year CS students. Different models exist for the early introductory courses (e.g., imperative first, objects first) [14], but the primary objective of the CS1 class is to sharpen the problem solving skills of prospective developers using software engineering strategies and program design tools. The class is centered mostly on refining and shaping the thinking of these new students rather than stressing detailed computational mechanisms. During this class, important foundational computing issues such as modularization and basic code structure can be addressed from a high conceptual viewpoint. The ACM Computer Science Curriculum 2008 recommendations do not list concurrency as a core topic within the Programming Fundamentals (PF) knowledge area. A six-hour core unit on concurrency does appear under the Operating Systems knowledge area, primarily directed at managing resources at the system level [15].

The author's personal experience in teaching undergraduate CS1 courses has fostered the belief that students in this initial class are especially suited to not only understanding, but also adopting the design principles associated with developing programs for parallel platforms. Admittedly, the CS1 class is not the best time for a detailed exposition and analysis of the various primitives and mechanisms that support parallel program execution, but an overview that helps students appreciate simultaneous processes at an early stage in their education will most certainly pay dividends when they later confront program development tasks for the increasingly prevalent multicore architectures.

The CS1 class effectively guides students on how to think about a computational problem and how to abstract the essential elements of the problem. Although mapping these concepts to code represents a secondary goal, this process allows the student

to confirm whether her models can be realized through the power of programming tools. Given that fundamental perspectives about programming paradigms are being molded at this early phase of the CS curriculum, it is reasonable to assume that students just beginning a computer science program will be more open to learning about "parallel thinking" than advanced students who are in some sense entrenched in previously adopted sequential or object-oriented programming strategies.

The distinctive learning context of the CS1 class gives the instructor an opportunity to present a variety of ideas before any particular one is summarily adopted or rejected by the student. Specifically, the student can more readily absorb the perspectives related to parallel thinking before getting locked in to some other mode of software analysis. At this unique juncture in the CS curriculum, more emphasis should be placed on high concepts particular to parallelism rather than concurrency implementation mechanisms, so that beginning CS undergraduates who might lack background in operating systems, networks, or advanced programming constructs are not disadvantaged.

The discussion of when the minds of CS undergraduates are ready for parallelism remains theoretical until a valid and reliable assessment tool is developed that consistently measures a student's grasp of fundamental patterns of parallel design. In a series of papers, Mattson, Sanders, and Massingill have developed a pattern language for parallel programming, in which one of the benefits is to "disseminate the experience of experts by providing a catalog of good solutions to important problems, an expanded vocabulary, and a methodology for the design of parallel programs." The pattern language described in the Mattson, et al., publications provides a roadmap for conceptualizing and implementing parallel programs. The target audience for this language, as with most pattern collections, is primarily the code designer/developer. The authors confirm this focus by stating that this pattern language is intended to "guide the programmer through the entire process of developing a parallel pro-

gram." [6] Most notable in this particular pattern language is the first design space, *Finding Concurrency*, which is "concerned with structuring the problem to expose exploitable concurrency. The designer working at this level focuses on high-level algorithmic issues and reasons about the problem to expose potential concurrency" [47].

From the above descriptions, it would appear that the ability to reason, abstract, and conceptualize parallelism is the crucial initial step in devising workable solutions to concurrency problems. In fact, Mattson, et al. claim that identifying exploitable concurrency is an acquired and improvable skill such that experienced parallel designers may ultimately be able to move directly to the second design space. It's also interesting to note that the authors address process/thread management only in their fourth and final design space, *Implementation Mechanisms*.

A student learning about parallel programming for the first time can benefit from the categorizations and strategies described in the *Finding Concurrency* design space, and, to a more limited extent, the *Algorithm Structure* design space. Determining how to exploit concurrency is one of the key goals of parallel programming. Most textbooks on parallel programming address the partitioning and grouping process in one of the early chapters [34][57][71], so the *Decomposition* and *Dependency Analysis* patterns in the *Finding Concurrency* design space help to structure the discovery process for a student new to parallel software development. In fact, the authors state in their 2000 paper that the patterns in the *Finding Concurrency* design space "form the starting point for novice parallel programmers" [7]. In the same paper, they reiterate the instructional nature of the *Finding Concurrency* design space by noting one of its main functions is to "help the programmer select an appropriate pattern in the *Algorithm Structure* design space." Immediately following, they state that "experienced designers" might already know how to make this selection.

An especially powerful pattern that instructors can emphasize during class discussions is that of *Design Evaluation*, described in detail in the 2000 paper. The

intent of the pattern clearly establishes the iterative nature of the design phase: "In this pattern, we evaluate the design so far, and decide whether to revisit the design or move on to the next design space." This iterative approach may already be familiar to students with backgrounds in Object-Oriented or even Structured analysis and design. To help guide students in evaluating the initial parallel program design, this pattern provides metrics based on suitability, quality, and preparation. A set of specific questions are proffered in this pattern, which can help the students generate enough detailed evidence to critically evaluate their design as well as stimulate classroom discussion about what constitutes good parallel program analysis.

The value of the *Algorithm Structure* design space in education is primarily through its elaboration of some classic parallel programming problems and solutions. In the 1999 paper, the authors present three mature patterns: *EmbarrassinglyParallel*, *SeparableDependencies*, and *GeometricDecomposition*. In an introductory parallel programming class, students can begin to appreciate these patterns almost immediately either through class discussion or, based on the coding skills of the students, some fairly simple programming assignments. For example, the *EmbarrassinglyParallel* pattern requires no special techniques to effect parallelization, and a programming example like an affine transformation of individual pixels in a bitmap can be used to illustrate the pattern.

Although the pattern language does provide support for the educational process through the *Finding Concurrency* design space and some fundamental patterns in the *Algorithm Structure* design space, there is a substantial preliminary analysis assumed by the language that most students will need significant time to foster and hone. The core questions of this initial analysis are repeated in several of the papers: 1) is the problem sufficiently large; 2) are results sufficiently significant; 3) are key features and data elements within the problem well understood; and, most importantly, 4) which parts of the problem are most computationally intensive? Overall, before embarking

on a parallel programming project, students must be able to determine whether the computational demands of the algorithm justify pursuing a parallel solution. This insight points toward a crucial rule of conduct in software engineering quality assurance, that of doing the right thing rather than simply doing things right. Therefore, instructors would need to devote sufficient time to these issues to ensure that students are prepared to appropriately apply the pattern language for parallel programming.

One of the ways in which students can be trained in the process of parallel thinking is through the use of analysis tools. As suggested at the beginning of this section, the hands-on nature of visual analysis tools can play an important role in conveying key concepts about parallel design to the student. Computationally intensive portions of the code can be identified using a tool that provides a rough measure comparing computation time with communication time. The tool should reinforce the skills representative of the *Finding Concurrency* design space by helping the student identify which code sections are good candidate of parallelization. Immediate experimental feedback on running programs will either confirm or refute the student's choice. The tool could also provide a sense of the scalability of the application, since parallel programs should be able to reliably handle growing data sets. Here, as with education, the pattern language can inform the design and development of these tools.

At a minimum, a parallel analysis/design tool should have an awareness of the resources currently available on a specific platform, primarily the number of processing elements. Because of this implicit connection with runtime architecture, the design space that would offer the best guidelines for developing this tool would be *Algorithm Structure*. This design space focuses on major organizing principles. A tool that allows for the identification and visualization of tasks, datasets and computational resources would be enormously helpful to a programmer attempting to design a parallel algorithm. Similar to designing organizational charts or processes for business, a drag-and-drop tool capability would allow a developer to experiment with different

configurations of program elements, along with the potential for feedback to assist in optimizing the design.

The *Algorithm Structure* design space also addresses data flow considerations, which can be suitably represented in graphics/icon format. Both the pipeline and event-based coordination pattern lend themselves to pictorial representations or architectures, which can be more easily understood and assessed when compared with simply scanning source code. In iterative fashion, these visual configurations can enlighten and refine the task grouping, task ordering, and data sharing activities contained in the Finding Concurrency design space, allowing experienced designers who have initially skipped these initial patterns to refine and update their design.

Currently, the Weber State CS department (where the author holds a faculty position) does not expose students to concurrency issues until upper division courses (3000 level and above). Two such courses that have been taught by the author include (1) CS3100 Operating Systems in which low-level multithreaded constructs like mutexes, semaphores, and critical sections are presented and (2) CS3230 Java Programming in which approximately one week is devoted to how Java supports concurrency through the Thread class and the synchronized keyword. This approach to instructing parallel concepts appears to be standard operating procedure for most CS programs as discussed earlier in this chapter. Instead of expanding the scope of multithreading in upper division courses, or creating an entirely new Senior-level course on concurrent/distributed processing, this research effort maintains that the existing treatment of parallel computing concepts later in the curriculum can have more impact and value if appropriate seeds about parallel program design are planted in the minds of students during the initial CS1 programming fundamentals class.

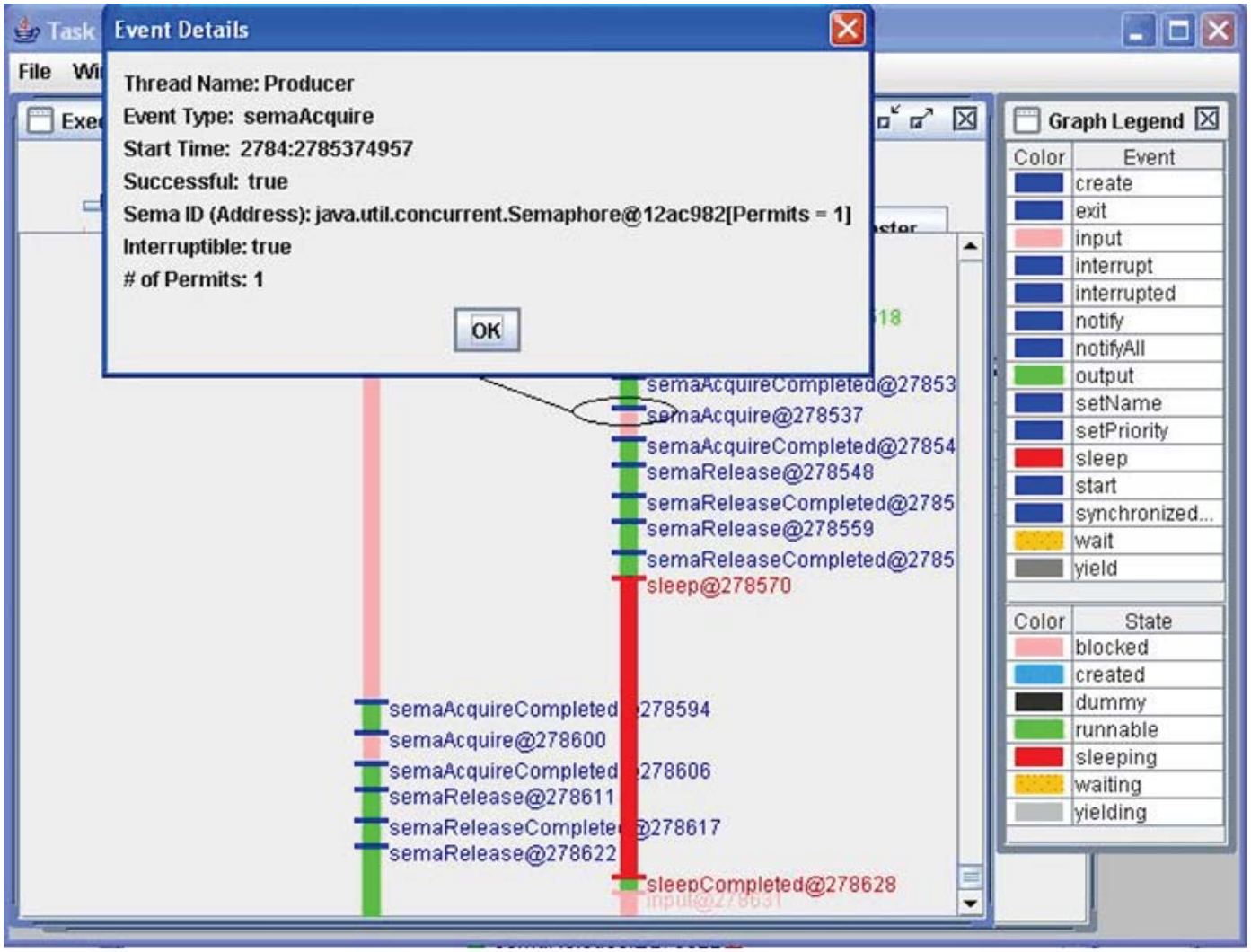


Figure 2.1: Multithreading Visual Tool

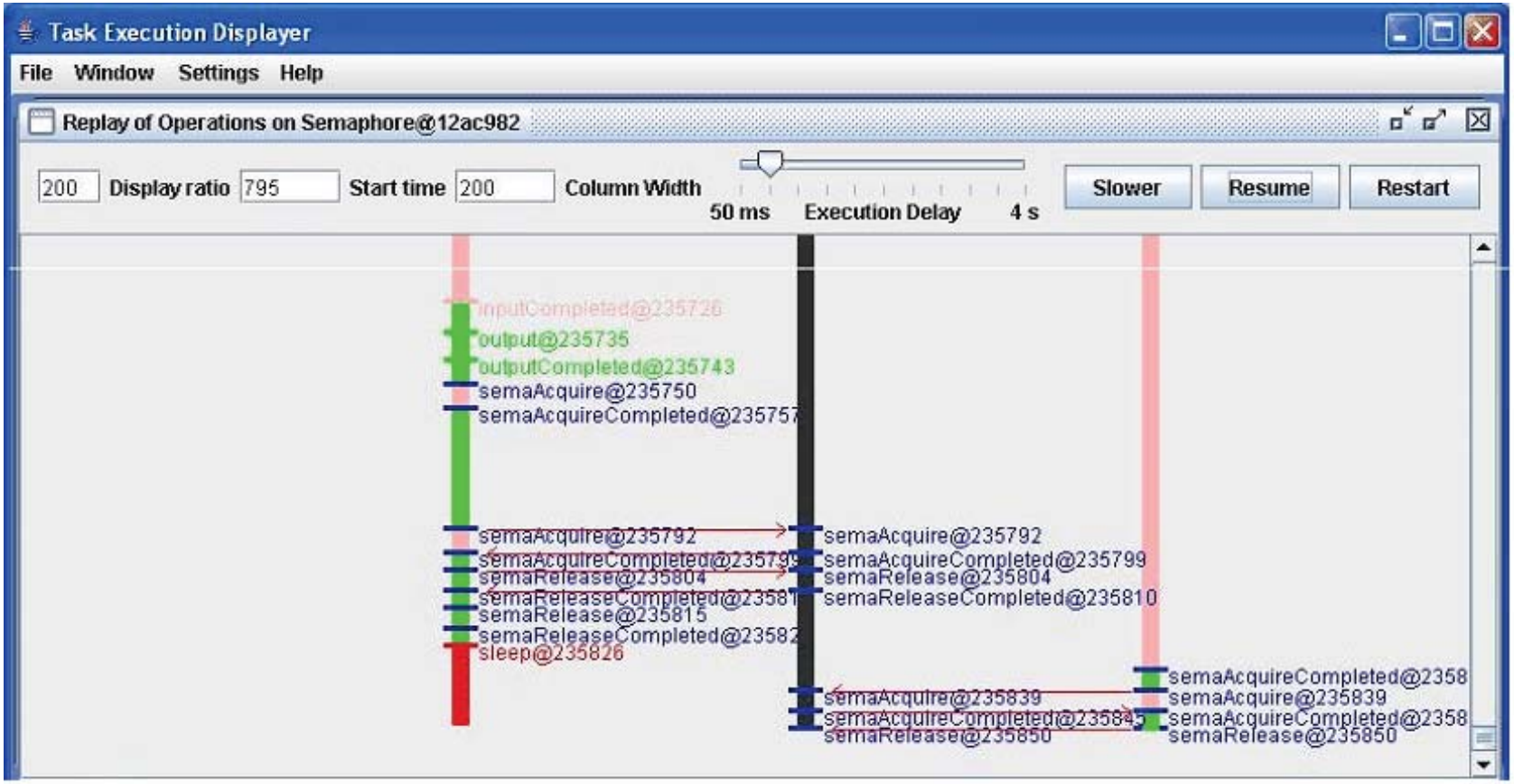


Figure 2.2: Visual Tool with Mutex

```
#include "ThreadClass.h"

class QuickSortThread : public Thread
{
     public:
          QuickSortThread(int Lower, int Upper, int Input[]);
     private:
          void ThreadFunc();  // thread body
          int  lower, upper;  // lower & upper bounds
          int  *a;            // pointing to array to be sorted
};
```


http://doi.acm.org/10.1145/958795.958796

Figure 2.3: Using the Thread Class in ThreadMentor

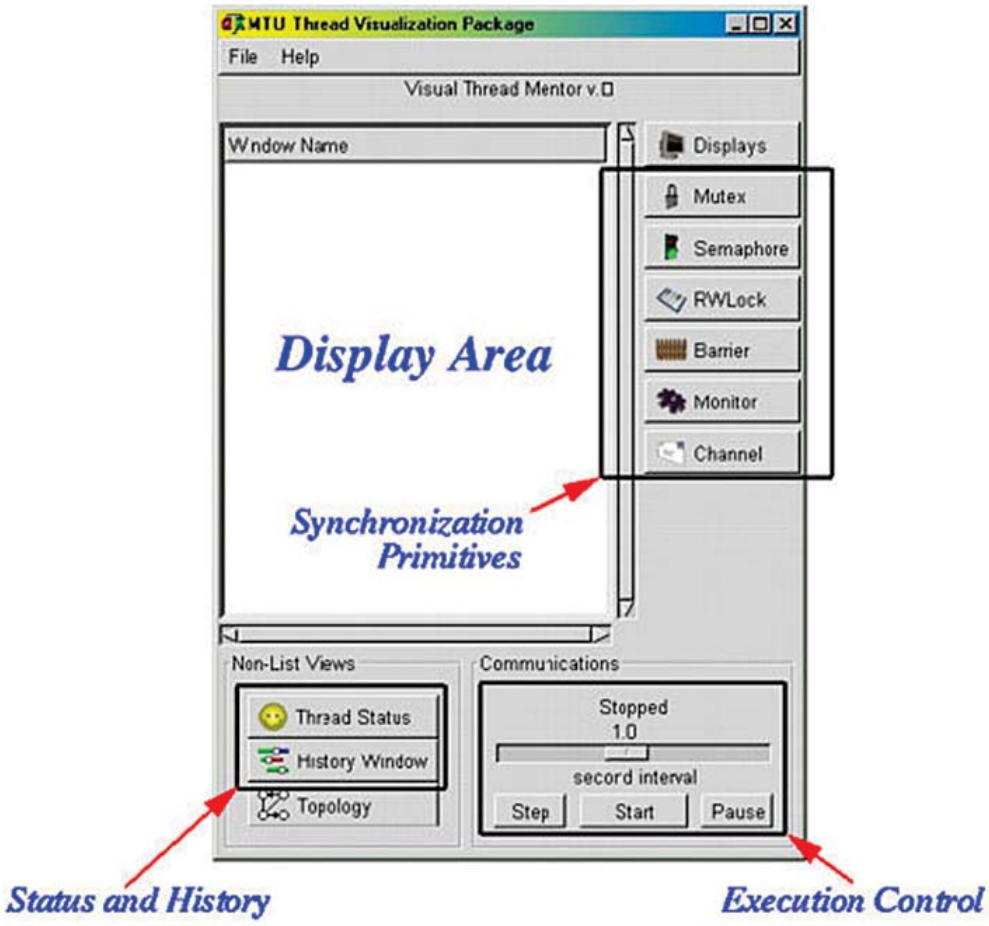


Figure 2.4: ThreadMentor's Main Window

Figure 2.5: ThreadMentor's Thread Status Window

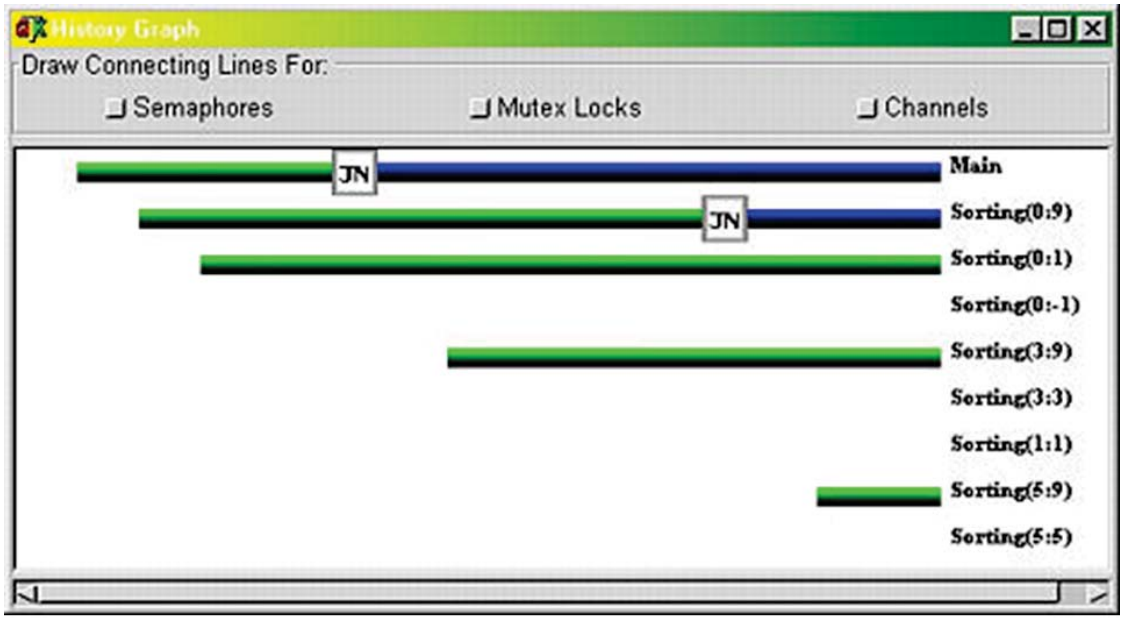


Figure 2.6: ThreadMentor's History Graph Window

Table 2.1: Example Courses on Parallel Computing

MIT:

6.046J Design and Analysis of Algorithms Techniques for the design and analysis of efficient algorithms, emphasizing methods useful in practice. Topics include sorting; search trees, heaps, and hashing; divide-and-conquer; dynamic programming; greedy algorithms; amortized analysis; graph algorithms; and shortest paths. Advanced topics may include network flow; computational geometry; number-theoretic algorithms; polynomial and matrix calculations; caching; and ***parallel*** computing.

Pre-reqs:
6.006 Introduction to Algorithms
6.042 Mathematics for Computer Science
6.01 Introduction to EECS I

Stanford University:

CS149 Parallel Computing Course is an introduction to parallelism and parallel programming. Most new computer architectures are parallel; programming these machines requires knowledge of the basic issues of and techniques for writing parallel software. Topics: varieties of parallelism in current hardware (e.g., fast networks, multicore, accelerators such as GPUs, vector instruction sets), importance of locality, implicit vs. explicit parallelism, shared vs. non-shared memory, synchronization mechanisms (locking, atomicity, transactions, barriers), and parallel programming models (threads, data parallel/streaming, futures, SPMD, message passing, SIMT, transactions, and nested parallelism). Significant parallel programming assignments will be given as homework.

Pre-reqs:
CS140 Operating Systems and Systems Programming
CS110 Principles of Computer Systems
CS107 Programming Paradigms
CS106A Programming Methodology
CS106B Programming Abstractions
CS103 Discrete Mathematics for Computer Science

Carnegie Mellon University:

15-418 Parallel Computer Architecture and Programming The fundamental principles and engineering tradeoffs involved in designing modern parallel computers, as well as the programming techniques to effectively utilize these machines. Topics include naming shared data, synchronizing threads, and the latency and bandwidth associated with communication. Case studies on shared-memory, message-passing, data-parallel and dataflow machines will be used to illustrate these techniques and tradeoffs. Programming assignments will be performed on one or more commercial multiprocessors, and there will be a significant course project.

Pre-reqs:
15-213 Introduction to Computer Systems
15-123 Effective Programming in C and UNIX
15-110 Introduction to Programming

Table 2.1 continued

UC Berkeley:

258 Parallel Processors In-depth study of the design, engineering, and evaluation of modern parallel computers. Fundamental design: naming, synchronization, latency, and bandwidth. Architectural evolution and technological driving forces. Parallel programming models, communication primitives, programming and compilation techniques, multiprogramming workloads and methodology for quantitative evaluation. Latency avoidance through replication in small-scale and large-scale shared memory designs; cache-coherency, protocols, directories, and memory consistency models. Message passing: protocols, storage management, and deadlock. Efficient network interface, protection, events, active messages, and coprocessors in large-scale designs. Latency tolerance through prefetching, multithreading, dynamic instruction scheduling, and software techniques. Network design: topology, packaging, k-ary n-cubes, performance under contention. Synchronization: global operations, mutual exclusion, and events. Alternative architectures: dataflow, SIMD, systolic arrays

Pre-reqs:
252 Graduate Computer Architecture
152 Computer Architecture and Engineering
61C Machine Structures
61B Data Structures and Programming Methodology
61A The Structure and Interpretation of Computer Programs

University of Utah:

3810 Computer Organization An in-depth study of computer architecture and design, including topics such as RISC and CISC instruction set architectures, CPU organizations, pipelining, memory systems, input/output, and **parallel machines**. Emphasis is placed on performance measures and compilation issues.

Pre-reqs:
2420 Introduction to Computer Science II
1410 Introduction to Computer Science I
1010 Introduction to Unix

6110 Formal Methods for System Design Study of methods for formally specifying and verifying computing systems. Specific techniques include explicit state enumeration, implicit state enumeration, automated decision procedures for first-order logic, and automated theorem proving. Examples selected from the areas of superscalar CPU design, **parallel processor memory models**, and synchronization and coordination protocols.

Graduate Level Course

6230 Parallel Computing and High Performance Computing Overview of parallel computing; processors, communications topologies and languages. Use of workstation network as parallel computers. Design of parallel programs: data composition, load balancing, communications and synchronization. Distributed memory and shared memory programming modules; MPI, PVM, threads. Performance models and practical performance analysis. Case studies of parallel applications.

Graduate Level Course

CHAPTER 3

EXPERIMENTAL EDUCATIONAL STUDY

This chapter describes the experimental method employed in the educational study that was carried out as a core part of this research. The fundamental research questions targeted by this study are enumerated, establishing the proper context for an examination of results in Chapter 6.

The experimental setting, student subjects, and representative participant population are addressed. Relevant information about the "laboratory" for this experiment, the CS1400 *Fundamentals of Programming* course, is provided. Institutional Review Board approval for this study was granted from both the University of Utah and Weber State University.

The chapter details the two-factor mixed-group research design utilized in this study, in which the classroom intervention mode (control, lecture only, and lecture with PAT visualization tool) is the independent variable, and student comprehension of parallel programming concepts as measured by the customized PoPS assessment is the response variable. The same assessment was administered three separate times during the CS1400 course (Weeks 1, 3, and 9) to each intervention group, providing the repeated non-independent factor in the research method.

Survey administration procedure is covered, and the grading of both the multiple choice and written design questions is explained. Both the parametric and nonparametric statistical tests applied to the survey scores are discussed, as well as the method employed in generating and interpreting the results of the experimental study. This chapter essentially provides a profile of the overall structure and detailed components

of the experimental educational study employed in this research.

3.1 Problem Statement

The objective of this study is to quantify the impact of two distinct modes of CS1-level classroom interventions on student comprehension of parallel computing concepts.

3.2 Research Hypotheses

The specific instructional interventions used in this study are categorized as 1) None (Control), 2) Lecture Only, and 3) Lecture with PAT software analysis tool. Each of these is described in more detail in Chapter 5. Given the application of these two levels of intervention across independent groups of students and the repeated measure factor within a given group, the main research objective stated in Section 1.2 can be resolved further into the following research hypotheses:

1. With significance level $\alpha = 0.05$, CS1 students exposed to a three-week "lecture-only" course module on parallel design concepts will exhibit statistically significant comprehension levels about this subject matter *after* the delivery of the course module when compared to students exposed to traditional CS1-level topics during the same time period.

2. With significance level $\alpha = 0.05$, CS1 students exposed to a three-week "lecture with software visual tool" course module on parallel design concepts will exhibit statistically significant comprehension levels about this subject matter *after* the delivery of the course module when compared to students exposed to traditional CS1-level topics during the same time period.

3. If there is no detectable interaction between the experimental factors, then with significance level $\alpha = 0.05$, CS1 students will exhibit statistically significant

comprehension levels about this subject matter *after* the delivery of any CS1 three-week course module when compared to comprehension levels just prior to the three-week course module.

For each hypothesis above, comprehension levels are measured by a valid test instrument described in Chapter 4. The third hypothesis tests for the repeated measure main effect, and is only applicable if there is no factor interaction. The statistical analysis used in this study follows the conventional approach of quantifying the probability of a Type I error, i.e. incorrectly rejecting the associated null hypotheses. The choice to focus analysis on rejecting the null hypotheses is suggested by Drew, et al.[22]: "For statistical testing and problem distillation purposes, the null hypothesis works very well and is probably used more frequently by practicing researchers than the directional hypothesis."

3.3 Research Design

The educational study employs a two-factor mixed group design with two experimental variables. Figure 3.1 provides the core structure of the research design. As shown in the figure, the interventions listed horizontally represent the independent group measure and the longitudinal testing within groups along the vertical axis represents the repeated measure. The three levels along each factor result in a total of nine conditions or treatments. The response variable for each condition is the survey score obtained from each student in the designated population using the test instrument administered on the day indicated in the figure. The diagram also shows the number of surveys collected for each treatment.

The horizontal axis in the figure denotes the comparison among groups across content delivery techniques: A_1) the class section exposed to presentation of parallel concepts with the support of the PAT visual analysis tool, A_2) the class section exposed only to presentation of parallel concepts, and A_3) the control group that

receives only traditional CS1 instruction. For Groups A_1 and A_2 above, identical learning objectives involving similar depth and breadth of topic content are defined. The instructional intervention for Groups A_1 and A_2 also includes a single homework assignment that exercises the student's ability to find concurrency. The key difference between these two levels is that Group A_1 students have hands-on access to the PAT visual analysis tool for in-class demonstrations of parallel program behavior and for solving problems in the homework assignment. Consequently, the homework assignment for Group A_1 is designed differently than the homework assignment for Group A_2: the Group A_1 exercises direct students to use the PAT to analyze the effects of parallelizing specific program sections, whereas the Group A_2 exercises instruct students to analyze code for possible parallelization from a strictly theoretical context.

The participant population for this study can be generally characterized as first-year/second-year undergraduates attending a CS1-level course at a university that awards primarily baccalaureate degrees. The actual subjects in this study were students enrolled in three different 15-week sections of the *Fundamentals of Programming* (CS1400) course offered at Weber State University in Ogden, Utah. The CS1400 class is a requirement for both the AAS and BS computer science degrees. The following is the course description for CS1400 as it appeared in the 2009-2010 Weber State University catalog:

CS SI1400. Fundamentals of Programming (4)
This course covers basic operating system operation and components of the development environment. The majority of the course covers basic problem solving and program design of a software application using a selected language. Topics presented and discussed depending on selected language include: thinking logically to solve problems, working with input/output devices, compilation and library use, structured programming and modularity concepts, conditional and iterative structures including recursion, data types and structures, and pointers. Prerequisite: CS 1030.

The class information based on research intervention is given in Table 3.1. Although the selection process for this study's participants cannot be strictly characterized as random sampling, the inherently indiscriminate nature of university course

registration provides the sampling mechanism that assigns students to the independent groups. The size of the groups was sufficiently similar that the researcher assumed orthogonality for the experiment [61]. Prior to enrolling in the classes listed in Table 3.1, students were not aware that course content would either include a discussion of parallel concepts (in the case of Groups A_1 and A_2) or involve an evaluation instrument. On attending the first day of class, students were informed of the activities related to the study and directed to the appropriate IRB information. No students reported dropping or transferring from the classes because of the experimental study; any subsequent drops or transfers were due to reasons unrelated to the study. The enrollment values listed in Table 3.1 reflect the "stable" roster numbers typically recorded three weeks after the term begins. All reported results were acquired from students enrolled in the class on the day the evaluation instrument was administered. As with any class session, some individuals in the class may not have been present on the day of a test evaluation.

Given these considerations, this research assumes the participant pool is representative of the population described above, and that any variation among participants related to age, occupation, gender, or socioeconomic level did not significantly influence the outcome of the study. Although participants in this study are in a sense "self-selected," the factors influencing the self-selection process are related to the normal course enrollment process (e.g., time-of-day, course conflicts) and are entirely orthogonal to the target objectives of the study. The research also assumes that all participants enrolled in this CS1-level course possess equivalent computing expertise and technical capability. This assumption is strengthened by the department policy that allows students to place out of the CS1400 class and earn credit by taking an advanced standing examination prior to enrolling in the course. As a result, it is assumed that only those students who truly require the instruction offered by the CS1 class will enroll. Finally, it is assumed that participants have no extensive prior

experience with parallel computing, which is bolstered by the signature introductory nature of the CS1 class.

The information in Table 3.2 gives some limited profile of the participant demographics by class. Figure 3.1 shows the study sample sizes for a given treatment range from 19 for Condition A_2B_3 to 31 for Condition A_1B_1. This range edges across the $n = 30$ inflection point, typically considered the dividing line distinguishing between large and small samples [63]. Intuitively, larger samples generally provide more information about the target population than do smaller samples. Given that only one sample size in this study exceeds the 30 sample threshold, the test results for each treatment are assumed to satisfy the requirements of small sample analysis: 1) all nine treatment population probability distributions are normal, 2) the nine treatment population variances are equal, and 3) test scores obtained for each treatment represent random, independent samples.

Since "control is the essential element in sound experimental design,"[22] this study attempts to minimize any potentially counterproductive classroom variations and influences among the three independent test groups A_1, A_2, and A_3 that can be reasonably curtailed. Instruction to all groups is administered by the same teacher to remove any potential differences in knowledge-base or instructor effectiveness than can exist between two different teachers. To maintain initial uniformity among the experimental and control groups, the study intervention was conducted in the first three weeks of the class. This will minimize the phenomenon that typically occurs late in the term by which students acclimate to teaching methods, solidify opinions about teacher effectiveness, and generally adjust motivation levels based on nonacademic factors unrelated to course content.

As regards test instrument administration, the vertical axis in Figure 3.1 represents the repeated measure experimental variable defined by three successive evaluation conditions named Pretest, posttest, and Recap-test. The term PoPS in the

figure refers to the Perceptions of Parallelism Survey (PoPS) instrument specially designed for this study and described in more detail in the next chapter. The PoPS is a formative assessment instrument to determine student comprehension and retention of parallel programming concepts at each level of the repeated measure factor. This study is quantitative in the sense that numerical results from the assessment instrument will be used as the criterion measure (the dependent variable).

On the dates listed in the figure for each treatment, the PoPS written exam was administered to each student in that group. Students are given no more than one hour to finish the assessment. Incomplete exams are accepted. The survey tests the knowledge-based, reasoning, and problem-solving capabilities of the students as they relate to comprehension of parallel programming design concepts. The PoPS survey targets the student's grasp of high-level concepts and thinking, and requires abstraction of the course material delivered through assignments and class time participation during the initial three week period dedicated to parallel programming topics.

The PoPS survey includes two parts composed of 1) 30 multiple-choice questions and 2) one design "essay" question. Each part will be graded separately. The single answer multiple-choice questions help remove the possibility of unintentional bias which can sometimes occur when correcting essay-type questions, whereas the design question will provide some measure of the student's deductive reasoning and exposition abilities. The multiple-choice questions impart high reliability to the assessment instrument, similar to a collection of math problems whose measure "would be very likely to show consistency across different times and observers"[22]. Regarding validity, the assessment questions will target the learning objectives clearly outlined for the instructional portion of the study. Given that students in the experimental groups will be exposed to identical content as dictated by those objectives and all students will be assessed under relatively similar conditions in both time and place, the instrument attains a sufficient level of validity in measuring student comprehension.

As mentioned above, the criterion measure (response variable) is the individual numerical results of the graded exams. A grade for the multiple choice exam questions is assigned by the instructor using a straightforward right/wrong correction scheme; the resulting measurement is the number of questions answered correctly by the student.

A numerical value is applied to the design (essay) question using a six-point Likert scale. Assessments of the written design question focus on the following areas of aptitude: 1) the overall readability of the student's response, 2) the level of reasoning applied by the student, and 3) the student's mastery of key concepts. The purpose of the design "essay" question is to provide additional support for the conclusions derived from the analysis of the multiple-choice part, and to furnish explicit written examples of the student's expertise in parallel programming design.

To achieve integrated reliability, two objective graders other than the instructor independently evaluated all the submitted design question responses. The graders did a blind evaluation of each design question, with no knowledge of the identity of the student. The instructor invested sufficient time with each grader to explain the required design question response, and to train each grader according to the assessment rubric (Figure 3.2) and correction sheet (Figure 3.3) specifically constructed to ensure systematic evaluation of the design question. The "chart" referenced in Figure 3.2 is the correction sheet of Figure 3.3.

Because the two graders did not exchange communication or influence each other's student subject ratings, a Mann-Whitney U Test is applied across the two separate independent sets of scores from each grader to determine if they are statistically different. If no difference is indicated, then a single score will be calculated for every design question response by taking the average of the two scores from each grader.

Because of the assumption stated above that that the numerical test results from the multiple-choice portion of the assessment instrument derive from an underlying

normal probability distribution with equal variance for each condition (treatment group) given in Figure 3.1, a parametric two-way ANOVA for mixed comparisons test can be used for data analysis. This experiment is a complete 3 x 3 factorial design since samples exist for all nine possible factor-level combinations. The two-way ANOVA will provide information about the probability of an interaction effect between the two experimental factors. For the situation in which there is no statistically significant factor interaction, the differences in the mean levels of the mutually independent main effects (the choice of intervention instructional method and the repeated measure) can then be investigated An F test will be used to ascertain the validity of the research hypotheses listed in Section 3.2.

Since no assumptions can be made about the underlying probabilities of the design question scores, nonparametric tests are used for statistical analysis of these measures. Specifically, a Kruskal-Wallis test can be applied for comparisons between the experimental and control groups, and a Friedman matched group test can be applied to the repeated measures within each group. Consistency between the parametric and nonparametric analysis will strengthen inductive conclusions derived from the study.

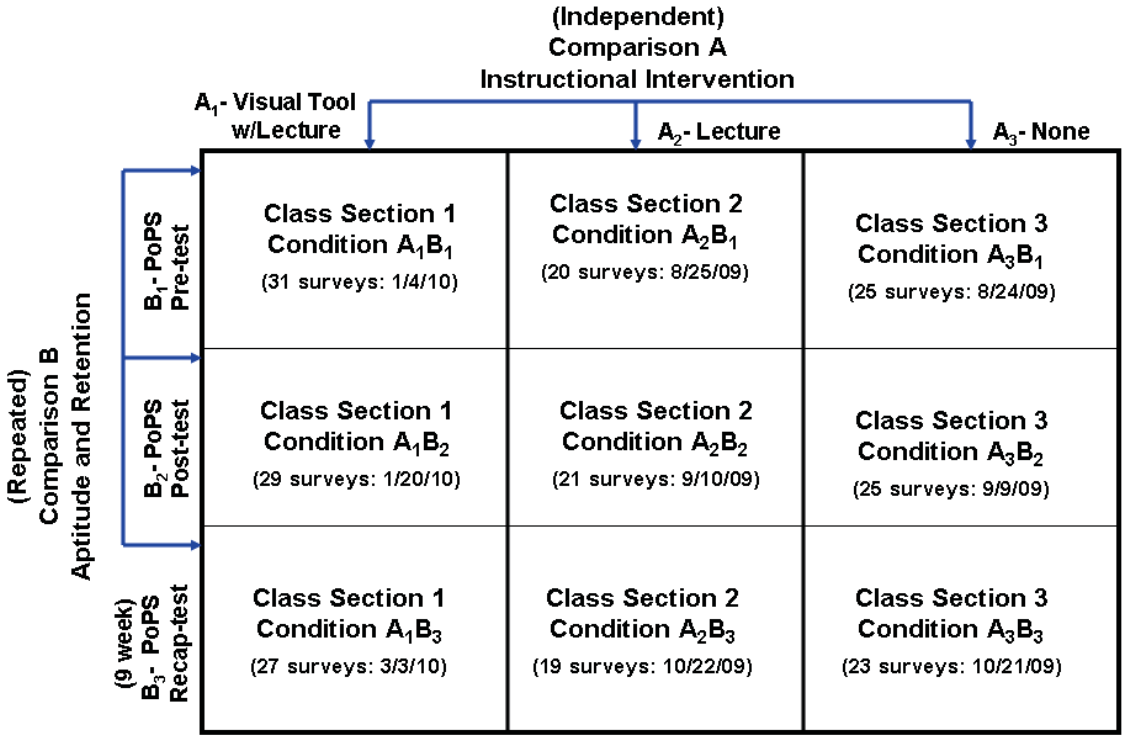


Figure 3.1: Two-Factor Mixed-Group Design

Six Point Likert Scale:

6: Excellent Reasoning/Correct Solution
5: Good Reasoning/Accurate Solution
4: Average Reasoning/Practicable Solution
3: Fair Reasoning/Marginal Solution
2: Poor Reasoning/Incorrect Solution
1: Minimal Reasoning and Solution

* The chart below provides a gradation in the criteria used to assess the design question.
* Points should be awarded beginning from Criterion #1 and proceeding in order to Criterion #6.
* The evaluation should conclude when a "No" answer is assigned at a specific criterion – all subsequent criterion then receive a "No" answer.
* The design question final score is the total number of "Yes" responses in the rubric.

Figure 3.2: Design Question Grading Rubric

Student:_____ Class Code:_____
Response submitted: Yes_____ No_____

Criterion	Assessment
1. Solution submitted.	Yes / No
2. Narrative description comprehensible and concise/ Diagram clearly drawn.	Yes / No
3. Narrative/diagram provides complete solution with some reference to processors and/or process flow.	Yes / No
4. Solution provides specific description of processor utilization and/or process flow that provides a broadly workable solution. All processors should be utilized.	Yes / No
5. Solution integrates reduction operations (maximizations) and communication paths to correctly solve the problem. Diagram (if it exists) Supports narrative.	Yes / No
6. Solution Narrative/Diagram closely approximates the actual proposed solution.	Yes / No
Total Score (Sum the number of "Yes" responses)	

Figure 3.3: Design Question Grading Chart

Table 3.1: Experimental Study Independent Groups

Intervention Group	Course Time	Enrolled Students
Lecture w/Visual Tool	Spring 2010: MW 9:30-11:30	31
Lecture Only	Fall 2009: TR 11:30-1:30	22
Control	Fall 2009: MW 9:30-11:30	27

Table 3.2: Student Subject Majors and Gender

	Student Major					Gender	
Intervention Group	CS	CNS	Eng	GS	Other	Male	Female
Lecture w/Visual Tool	21	0	3	4	3	25	6
Lecture Only	14	2	4	1	1	20	2
Control	18	1	3	1	4	23	4

CS = Computer Science
CNS = Computer and Network Security
Eng = Computer and Electronics Engineering Technology or Pre-Engineering
GS = General Studies

CHAPTER 4

EVALUATION INSTRUMENT

Student comprehension of parallel design concepts is measured using a Perceptions of Parallelism Survey (PoPS) specifically developed for this research study by the author. The survey focuses on monitoring a student's high-level reasoning strategies about finding concurrency in familiar situations rather than testing the student's detailed knowledge of terminology, taxonomies, and implementation mechanisms used in classifying and creating parallel programs and systems. The PoPS is modeled after the Force Content Inventory (FCI) used for many years in undergraduate Physics education.

4.1 Parallel Concept Inventory

To inform and clarify the discussion on parallel computing instruction, an assessment tool that consistently measures a student's grasp of fundamental patterns of parallel design is needed. Section 2.3 introduced a series of papers by Mattson, et al. about developing a pattern language for parallel programming and highlighted the relation between the first two design spaces and the effort to provide instruction on parallel concepts early in the CS curriculum [47]. Recall that the first design space, *Finding Concurrency*, is "concerned with structuring the problem to **expose exploitable concurrency**." CS1-level instruction of parallelism and the subsequent assessment of a student's comprehension of parallel design should target this foundational skill. Sharpening the student's intellectual awareness to identify potential

concurrency wherever it might exist is the vital first step in developing complex parallel computing systems.

Undergraduates beginning a program in computer science may enter with underdeveloped or outright erroneous intuitions about how parallelism works. Indeed, the ideas on parallelism they bring with them may be flawed and may potentially hinder their ability to design proper concurrent code solutions later in their academic careers. Misperceptions of vital core concepts are not unique to computer science students. Efforts by educational researchers and faculty in Physics have been focused for many years on the flawed concepts that students bring with them to the first-year course on Newtonian mechanics. Halloun and Hestenes examined the "common sense beliefs of students" about basic physical phenomena, and how some of these conceptions, no matter how incompatible they may be with established scientific theory, are not easily discarded by students even after a full course in which these notions are explicitly disproved through in-class demonstrations [29][30].

Physics students drew conclusions about physical movement predominantly from a mixture of flawed Aristotelian and Impetus theories, and some correct Newtonian conceptions. The extent to which students adhere to inaccurate views of the physical world is illustrated in the account by the psychologist Michael McCloskey [49], whose 1980 experiment showed that strongly held intuitive models are not always correct. A group of undergraduates were shown a picture of a thin curved metal tube and were then asked about the resulting trajectory of a metal ball shot through the tube. Of the students who studied no physics, 49% thought the ball would continue along a *curved* path as suggested by the curvature of the tube, thus displaying adherence to the medieval impetus theory formalized by Jean Buridan in the 14th century. As explained by Halloun and Hestenes, impetus theory proposes "that when an object is thrown, the active agent imparts to the object a certain immaterial motive power which sustains the body's motion until it has been dissipated due to resistance by the

medium" [30]. More notable is that 19% of the students with college-level physics predicted a curved path for the ball. For whatever reason, student concepts of motion as interpreted through direct experience with the physical world did not provide enough convincing evidence to displace the erroneous impetus theory from their thinking.

Concern for these strongly-held but incorrect student perceptions led Hestenes, Wells, and Swackhamer to develop the Force Content Inventory (FCI), an assessment instrument intended to "provide a clear, detailed picture of the problem of commonsense misconceptions in introductory physics" [33]. In contrast to the extensively researched design of the FCI over many years, the development of a validated computer science concept inventory that measures a CS1 student's grasp of fundamental programming concepts is still in its very early stages [69]. Whereas the substance and matter of the FCI has been examined by a vast number of content experts and refined through many iterations of the assessment, the definition of the knowledge base a CS1 student should possess remains an open research question, and the proper evaluation of this knowledge is hampered by the rapidly changing programming languages and paradigms utilized in the CS1 classroom. One such change is the emerging importance of concurrent computation.

Whereas instructors in Physics are confronted with a lifetime of student exposure to motion and force, Computer Science faculty do not necessarily have to contend with a vast student experiential history in parallel processes. However, the recent explosion of electronic communication devices may easily blur the conceptual line between classical single CPU multitasking and true simultaneous processes. Given the recent introduction of multicore and many-core computing and the popular definition of multitasking as the performance of multiple tasks at the same time, it is vital that future CS graduates be able to readily distinguish single CPU multitasking from true simultaneous processes, and to take a proper accounting of computational resources in order to correctly assess and identify actual levels of concurrency in computing

systems.

One notable popular misconception that beginning CS students bring to the classroom is that multitasking can be accomplished without the cost of protracted execution time, possible deteriorating performance, and context switching. Research has confirmed the myth of performing demanding cognitive tasks simultaneously [59]. The brain can be a useful instructional metaphor when discussing parallel processing systems with single or multiple CPUs, but only if students appreciate the inherent limitations of computational devices.

The motivation behind the design and development of the Perceptions of Parallelism Survey (PoPS) springs from the same general objectives and intentions as articulated for the FCI: measure and address student misconceptions about parallelism early in the educational process and encourage the proper perspectives about concurrency that will serve each student throughout her career in computer science. Parallelism is a concept well worth monitoring given the growing demand for software developers who appropriately exercise "parallel thinking."

4.2 Perceptions of Parallelism Survey (PoPS)

Similar to practitioners in other disciplines, computer scientists must cultivate and refine a core set of essential skills. Problem solving is regarded as a mainstay in computer science education and is often highlighted during the CS1-level course. Kramer addresses the question whether a CS student's ability to apply the powers of abstraction can be linked to success in producing elegant computational models and designs [39]. In the same article, Kramer asks if abstraction is teachable and measurable, and explicitly calls for tests that "examine different forms of abstraction, different levels of abstraction, and different purposes for those abstractions." In this way, effective means for teaching abstraction, whether by updating course content or labs, can be identified and utilized.

Parallel design is a specific area that relies heavily on abstraction. The models and systems that support concurrency cannot necessarily be seen and felt. A student may be shown a cluster of several networked computers or a sophisticated rack-mounted tightly-coupled physical collection of parallel processors, but capitalizing on the inherent computational power and speed of that computing system requires that the student possess some command of abstract concepts. A student's intuition about parallelism, much like a student's ideas about the concept of force in the physical world, may not be intrinsically sufficient or correct.

The Perceptions of Parallelism Survey (complete listing in Appendix A) provides an instrument for measuring a student's abstraction of parallel processes. The current version of the survey targets fundamental design patterns closely allied with the task of finding concurrency. The questions in the survey do not rely on a student's extensive knowledge of detailed implementation mechanisms, which are typically learned in an advanced course on parallel and distributed computing. Instead, problems are posed such that important patterns and practices in parallel design are reworked into commonplace scenarios using everyday objects that all students can immediately recognize. These objects include dartboards, waiting lines, books, and playing cards. Use of familiar objects and situations removes obstacles to student comprehension typically associated with CS-specific concepts or technical jargon. The situations and scenarios performed in each survey task are also familiar; examples include checking out a library book, performing addition, and appraising antiques.

In identifying general student misconceptions about programming, Kaczmarczyk cites students who inadvertently apply real world semantic understanding to code abstractions [36]. The PoPS attempts to narrow this divide between common student experiences and computational strategy, thereby allowing the student to focus primarily on the parallel characteristics intrinsic in the PoPS scenario. Although each problem setting in the PoPS may not represent an everyday occurrence, the intent

of the PoPS is to couch questions in a real-world context, allowing the student to rapidly grasp the gist and scope of the problem statement.

The PoPS includes two parts composed of 1) 30 multiple-choice questions and 2) one extensive design essay question. The multiple-choice questions are grouped together into eight separate tasks, each of which targets specific parallel design fundamentals as listed in Table 1.1. The table also provides an overview of the situation/context of each task.

Though the overall length of the PoPS (see Appendix A) and the associated sustained focus required from the student subjects may initially appear too demanding for a standard 60 minute content assessment, the design and structure of the survey was influenced by three primary factors related to the author's objectives, experiences, and observations:

1. The PoPS closely followed the test format of the Force Content Inventory (FCI) used in Physics. There were 30 questions on the FCI, with similar depth and detail. When I (the author) first took the FCI, I thought that it required an inordinate amount of time to complete. But after taking the same test a second and third time, I was able to more efficiently grasp the concepts and finished the test within a reasonable amount of time. Since the PoPS is administered to the same groups three separate times, I ultimately latched onto the idea that the survey really needed to stretch the students *initially* so that improvements in concept comprehension could be more readily noticed and measured.

   The other extreme would be to give a more manageable, "easier" test. But the main concern here is to avoid the ceiling effect, the situation in which many students do well on the initial test, thereby leaving very little room for measurable improvement in the subsequent repeated trials.

2. In length and content, the PoPS is about 110% of what my students typically

get on a midterm or final. Anecdotally, I have designed exams for Weber State students that I thought should take a full 60 minutes, and about 85% of them walk out of the examination room before 35 minutes have elapsed. (I am still amazed at this, but less so as time goes on.) I have observed that only a very few students take the fully allotted test time or go back to review their answers. So, although the ideal hope is that students carefully think through each question, the reality is that they answer primarily through "intuition", which is fine since the PoPS attempts to probe the student's *conceptual* grounding.

3. This point is related to item 2 above. The primary intent of each survey question is that a student's correct answer arises from a combination of correct intuition and applied conceptual thinking. I believe each question forces the student to do a little computational work even if they think they "know" the answer immediately. This necessarily will lengthen the exam because they will probably need to put a pencil to paper to compute optimum times, do some addition, or sketch some rough task management/organization diagram.

The principal features of the PoPS overall construction and design include:

1. Questions of moderate difficulty and length that require a modest level of calculation. Because students will be retested with the PoPS several times (see Chapter 3), the problems should challenge the students initially so that improvements in the comprehension of parallel concepts can be more readily monitored. Conversely, a survey with relatively easy, quick-response questions may result in many students doing extremely well on the initial test, thereby leaving little room for measurable improvement in the repeated trials. Ideally, a student's correct answer should arise from a combination of accurate intuition and applied conceptual thinking. Students may think they know the answer immediately, but each question by design forces the student to do a little computational work

to verify these initial hunches.

2. An emphasis on processing time. One of the primary measures of parallel computation is speedup, a performance analysis metric initially formulated by Amdahl [2], then expanded on using alternate scalability concepts by Gustafson, Barsis, Singh, Hennessy, and Gupta [71]. For early undergraduates, one of the most easily recognizable benefits of parallel solutions is that, for a given problem, processing time should be decreased when compared with the serial solution. These students may initially embrace the idealist view of linear speedup: by simply applying p processors to an existing sequential program, the run time is reduced by a factor of p. The diminishing returns afforded by this philosophy are addressed in the PoPS questions, which require students to reckon with the time cost of communication, and the limitation of Amdahl's Law. Numerous small and possibly duplicate problems that can be rapidly calculated may not be good candidates for parallelization. The calculations mentioned in feature 1 above are primarily devoted to determining the duration of processes, and how these values may be positively or negatively influenced by the addition of more processing elements (PEs).

3. Architectural considerations. Parallelism is not strictly an issue of correct or efficient programming. An awareness of underlying architecture is necessary for the software developer to write effective parallel programs. Both scaling and speedup have an architectural as well as an algorithmic component. As suggested in feature 2, CS1 students can develop an immediate appreciation for this interplay between hardware and software by carefully considering why a sequential program does not simply run p times faster on a machine with p processors. Sivasubramaniam aptly describes interaction overhead, a component of parallel architectures that degrades linear speedup, as involving communi-

cation issues like contention, latency, synchronization, resource management, and cache effects [64]. The PoPS questions consistently examine the computation/communication tradeoffs inherent in all parallel configurations.

Following is an overview of more specific patterns in parallel thinking targeted by the PoPS:

4.2.1 Serial/Parallel Computation

To find concurrency within a series of operations, students should be able to identify those steps that must be performed in sequence, and those steps that exhibit the potential to be run in parallel. Executing a binary addition operation on two separate pairs of independent operands is a straightforward example of calculations that can be made concurrent.

4.2.2 Temporal Dependencies

Even within tasks that can execute concurrently, there may be constraints placed on the order in which a series of steps occurs. Calculating the time evolution in a 3-D N-body problem illustrates a temporal or sequential dependency.

4.2.3 Multitasking and Context Switching

Students should gain an appreciation for the classical conceptualization of multitasking, which implies thread/process swapping on a single CPU. This contrasts with pure parallelism, in which processing elements are dedicated exclusively to a single task. Real-world parallel computations typically involve a combination of these two strategies.

4.2.4 Resource Management/Synchronization

Efficiently managing a program's assets is critical in order to attain the highest performance from a parallel problem. Proper communication among computational

units must be confronted and resolved. A subclass of resource management is data sharing, involving both global information and exchange of task-local data.

4.2.5 Understanding Amdahl's Law

The motivation here is not to present CS1 students with an exhaustive exploration of the origins of constant problem size scaling, nor to simply have them plug numbers into a formula to derive hypothetical speedup values. The key idea to recognize is that simply throwing more processors at a parallel problem does not constitute a viable strategy. In fact, performance may actually deteriorate, and a regard for other scalability concerns (i.e., Gustafson's Law) should be integrated into the student's thinking.

4.2.6 Master/Worker Configuration and Delegation

Different processor topologies can be employed in solving parallel problems, and students must recognize that only a select few of these types of problems can be solved by a collection of independent processors in an embarrassingly parallel way. Here again, issues of communication surface as well as the likelihood that for some problems one processor may be dedicated only to delegating tasks and possibly to a single reduction operation.

4.2.7 Load Balancing

The seminal example of the influence of load balancing on the efficiency of parallel computation is the generation of the fractal image of the Mandelbrot Set. The central part of the image that contains members of the set requires substantially more computation than the top or bottom portions. If the student applies data decomposition in a conventional way by simply dividing the image calculation into evenly-sized horizontal sections, then program performance is compromised. Students must consider proper types of fixed or dynamic scheduling in order to achieve sufficient levels of

load balancing.

4.3 Survey Task Objectives

To provide some additional perspective on the scope and intent of each PoPS task, this section provides detailed descriptions highlighting the survey task objectives. As revealed by a quick scan of the PoPS, a student's full understanding of each task scenario does not demand an extensive background in mathematics, software development, or networking. Refer to Appendix A for the specific task problem and associated questions.

4.3.1 Task I - Arithmetic Operation Sequence

This task is a straightforward exercise in recognizing data dependencies within a series of arithmetic calculations and applying the appropriate sequential constraints on these operations. From the problem description, instruction III exhibits a Read after Write (RAW) true dependency. The data hazard arises if access to the contents of variable c in instruction III is performed before instruction I saves its results to the same memory location. Instruction II exhibits no data dependencies with the other two instructions.

This task requires students to detect the limitations of parallelization embedded in the execution of these arithmetic operations, ignoring the effects of the overhead of establishing parallel processes. Question 1 verifies the student's basic knowledge of serial computation, and Question 2 examines the student's ability to identify concurrent sections of code.

4.3.2 Task II - Multitasking

This task challenges the student's tacit acceptance of the popular notion of multitasking, and illustrates the concept by using a common example derived from the student's direct experience with the traditional classroom environment. As described

in Section 4.1, students typically do not attribute a cost to performing several tasks at the same time, and may have misplaced ideas regarding the semantic difference between multitasking and simultaneous processes.

This task requires students to recognize that cognitive *and* computational multitasking requires the overhead of context switching. Pure parallelization assumes more than one dedicated processing element with no interaction among those elements. Question 3 demonstrates the nondeterminism of preemptive scheduling on a single CPU system, and Question 4 turns the spotlight on the efficiency drawbacks of multitasking among activities that demand focused attention.

4.3.3 Task III - Resource Management

Often software developers are confronted with limited resources, and sharing those resources requires specialized programming strategies like thread synchronization. Fundamentally, this task explores the appropriate mapping of processes to processing elements (PEs), and the effect of selecting specific mappings on system performance. In the task description persons represent processes, and the library book check-out machines denote our resources, the processing elements. As with all other PoPS tasks, computational efficiency is strictly a function of time.

Question 5 emphasizes the compromise in performance if the number of threads does not necessarily equal the number of PEs. None of the options for this question allow for the optimum situation of one thread per processor, so the student must reconcile this mapping mismatch to determine the solution that exhibits the best performance. Question 6 introduces the communication cost of distributing tasks among the PEs, with a special focus on how best to initiate and execute parallel processes. When compared with scenario (C), the slight modification introduced in scenario (D) increases the fraction of time spent on sequential operations, and negates the effort to capitalize on the efficiency of parallel computation. Scenario (D) is the

focus in Question 7, which essentially confirms that only a purely sequential single CPU computational strategy would have worse performance than scenario (D).

4.3.4 Task IV - Monte Carlo Simulation

This task is a recasting of a Monte Carlo Simulation using a setting and situation which should be familiar to all students. The topology of distributed, parallel computing units is highlighted as well as the reduction pattern and communication considerations. All requisite information is presented in the problem such that any individual with a mild inclination toward things technical would be able to grasp the problem, apply some reasoning, and settle upon an answer to each of the questions accompanying the task within a reasonable amount of time.

As suggested by the list of different costs for performing a single experiment, the task spotlights the ever-present tradeoffs that pervade all designs of parallel systems. The associated questions encourage the student to weigh the pros and cons of different choices related to materials, computation, and communication, effectively performing a kind of cost minimization analysis on the fly. Question 8 ensures the student can identify the true cost of quasi-simultaneous transmissions. Question 9 focuses on pure calculation cost without direct reference to FLOPS. Question 10 allows the student to exercise abstraction in order to reinforce the fundamental idea that one thread per processing unit is the optimum, assuming access to an unlimited number of processors and no hardware cost.

For Questions 11 and 12, material cost is now a consideration and equated with computation cost to simplify calculations and bring the CPU/Time tradeoff issue to the forefront. The intent with this question is that students can intuitively discern a “middle ground” solution when confronted with a specific hardware cost, rather than favoring either extreme of purchasing large quantities of expensive processors or falling back to the less efficient single CPU approach. Question 13 focuses solely

on the hardware cost, and relies on the simple deduction that procuring the fewest processors leads to minimizing this metric. Question 14 challenges the student's notions regarding the overall effect on task distribution and computation time when applying one more processor to a parallel computing problem. This question requires some implicit understanding of barrier synchronization.

4.3.5 Task V - Master-Worker Configuration and Communication

The objective of this task is to examine the student's understanding of the communication cost incurred by interacting parallel processes. The problem is for five persons to alphabetize sixty words, with each word printed on a separate card. Details about the time cost for information exchange between processors (represented as persons) is provided. The reduction time to condense alphabetized sub-stacks is also given as significantly less than the time required to perform initial sorting on a set of un-alphabetized cards.

This task challenges the student to recognize the most efficient system configuration, and addresses essential concepts of parallel design such as master/worker topology, communication, delegation, process integration, sequential dependencies, algorithm design, and reduction. Students are confronted with the shared-memory issues that arise when two or more threads attempt to perform a single operation.

Question 15 emphasizes the importance of maximizing computation time and minimizing communication time in parallel process design. Students should recognize the drawbacks of scenarios that propose an inordinate amount of idle processing cycles. Question 16 employs a simple visualization of a communication strategy using directed graph notation, a diagrammatic representation of network configuration and process flow that CS1 student's will learn about in a subsequent Discrete Math course. Question 17 attempts to solidify the liabilities of exchanging information in parallel computing by increasing the communication cost six-fold. Students should identify

that the scenario with the most stack exchanges will absorb the biggest performance hit because of the increased communication cost.

Question 18 highlights the synchronization penalties incurred when P processors attempt to sort the same set of N items simultaneously. This is reflected in the problem as two or more (P) persons alphabetizing the same stack of N cards, and the overall cost of this strategy is expressed as a time cost of $P * N$. Students are encouraged to limit task sharing when delegating the individual activities that constitute a parallel computing problem, and to assign raw input data set components to single processors whenever possible. Question 19 revisits the drawbacks of communication cost in the context of shared tasks.

4.3.6 Task VI - Mandelbrot Example

One of the classic examples of applying parallel techniques to image processing is generating the Mandelbrot Set, a fractal object published by an IBM mathematician of the same name in 1977 [46]. The Mandelbrot Set is a set of points in two dimensional space defined as follows: for each point (x, y), compute a sequence of points (a_i, b_i), such that:

$$a_0 = 0$$

$$b_0 = 0$$

$$a_{i+1} = a_i^2 - b_i^2 + x$$

$$b_{i+1} = 2a_i b_i + y$$

In Figure 4.1 points in the Mandelbrot Set are black. Points not in the Mandelbrot set are often rendered as a range of colors. The point (x, y) is a member of the Mandelbrot Set if the magnitude of each point in the sequence (a_i, b_i) remains finite, where $i = 1, 2, 3, \ldots$. If the magnitude of the sequence of points (a_i, b_i) tends toward

infinity, then the originating point (x, y) is not a member of the Mandelbrot Set. Computer programs that calculate membership of the Mandelbrot Set must rely on the proof that if the magnitude of the point (a_i, b_i) in a sequence ever exceeds 2, then that sequence will inevitably head towards infinity. Given the above, the algorithm for determining if point (x, y) is a member of the Mandelbrot Set can be expressed as follows:

Initially assume that (x, y) *is in the Mandelbrot Set*
for $i = 1$ *to* N
    calculate (a_i, b_i)
    if(magnitude of $(a_i, b_i) > 2$*)*
    then (x, y) *is not in the Mandelbrot Set*
end for

Other than the fact that calculations of adjacent points are independent, the strategy listed above illustrates the key feature of Mandelbrot Set processing highlighted by this PoPS task: identifying a point in the Mandelbrot Set is more computationally intensive than identifying a point outside the Mandelbrot Set. Essentially, to get the best performance from a parallel solution, proper delegation of data items and load balancing must be taken into consideration. As revealed in Figure 4.1, a processor operating on a horizontal band of points from the middle of the figure will require significantly more computation time than a processor operating on a horizontal band of points taken from either the top or bottom of the figure.

This PoPS task replicates this computational challenge by initially placing all of the time-intensive antique appraisals in one of three queues, and by introducing the oversight of a floor manager in assigning patrons to specific queues based on the order of arrival. Question 20 depicts the primary load balancing problem for

the student's consideration. Question 21 examines delegation strategies when only one computationally intensive task needs to be processed. Question 22 introduces overhead associated with task delegation and communication and asks the student to account for this influence on performance. Question 23 examines the student's ability to judiciously allocate tasks to processors using basic load balancing strategies.

4.3.7 Task VII - Lateral Communication

This task is predicated on finite difference equation approximations typically employed in scientific computing to model and measure physical processes. A common example often translated to a parallel algorithm is the one-dimensional diffusion equation:

$$\frac{\partial \theta}{\partial t} = \kappa \frac{\partial^2 \theta}{\partial x^2}$$

If θ represents temperature and κ is interpreted as thermal diffusivity, then the above second order partial differential equation becomes the heat equation which describes the variation of temperature along a single spatial axis over time. Discretized parallel computing implementations of this type of differential equation rely on the sharing of information between adjacent points in space, as illustrated by this C++ function constructed by Karniadakis and Kirby[38] that incorporates a central finite difference scheme with an Euler-forward time integration scheme:

```
void Diffusion_EF_CentralDifference( int N, double DN,
                                     double *uold, double *unew)
{
 for(int i = 1; i < N-1; i++)
    unew[i] = uold[i] + DN * (uold[i+1] - 2.0*uold[i] + uold[i-1]);


 return;
}
```

Note in this function that for each time step, the computation of location `unew[i]` depends on previous values of not only the target point, but also the two flanking points. Realizations of this code on message passing parallel platforms require that each processing node send and receive information to and from neighboring nodes.

The card game described in this PoPS task highlights the required information exchange and time expenditures when performing lateral communication. Time steps are represented by a single round of the card game, and the structure of the game simulates the initial value and boundary value problems that constrain the solutions to scientific models: each player is dealt a single card to start the first round and players at both ends of the linear seating arrangement are influenced by only one neighboring player. The order in which an internal node retrieves information about its adjacent nodes is set down explicitly in order to approximate the synchronization of send/receive commands typical of a message passing paradigm.

Question 24 tests the student's baseline knowledge of the game rules and the minimum time required for a node (player) to gather information from its adjoining nodes in order to determine its current value. Question 25 simply switches the send/receive order of the middle nodes, and consequently reduces the communication time required to establish a winner of the game. Question 26 demonstrates the contrast in efficiency between the two-way request/response "pull" strategy employed by the previous two questions and an alternative one-way scattering "push" strategy represented by the sharing action of adjoining players. Question 27 verifies the students understanding of the communication cost of the "push" approach described in the previous question for a single iteration (round) of information exchange.

4.3.8 Task VIII - Application of Amdahl/Gustafson Laws

Amdahl's Law is founded on the assumption that the computational problem to be solved is of fixed size and the scalability of his parallel analysis amounted to

applying more processors to this problem. However, Amdahl maintained that a finite percentage of the original problem could not be parallelized, which resulted in his generally pessimistic conclusions about the increase in speedup as a function of the number of processors [2].

John Gustafson countered with his own assumptions, stating that as the number of processors increased, real-world analysts would naturally wish to process problems of larger size. Essentially, the problem size selected frequently depends on the number of available processors. Gustafson maintained that the execution time should be fixed and that "it is the parallel or vector part of a program that scales with the problem size" [27].

The objective of this PoPS task is to focus the student's attention on both potential benefits of parallelization, speedup and problem size. Often the primary emphasis when initially exploring parallel concepts is to design faster programs. However, the goal of many parallel computing applications is to provide more accurate results, which can be realized by processing a larger quantity of higher-resolution data using more processing elements. Using a theoretical ball delivery mechanism to demonstrate the advantages of scaling up problem size, this PoPS task represents processors as output tubes and data items as balls.

Question 28 tests the student's basic understanding of the delivery mechanism and how the functionality of the mechanism translates to execution time. Question 29 examines the student's ability to recognize the optimum number of processors for the given problem size. Question 30 illustrates the fixed execution time constraint with a given number of processors, and asks students to scale up the problem size in order to take full advantage of the "computational" power of the delivery mechanism.

4.3.9 Design Question - Grouping and Ordering Tasks

The design "essay" question examines the student's ability to organize a series of tasks both sequentially in time and distributed among a fixed number of processing elements. The structure of the solution adheres to agenda parallelism, in which there is an series of tasks which must be performed and which may exhibit sequential dependencies [37]. The student is also asked to recognize and describe a reduction step, in which a global maximum must be computed to generate the final result.

The context of the problem is to determine the identity of a thief using archived photographs from a crime lab. Students must devise the most efficient strategy to analyze 1000 photos using 250 available processors. The definition and delegation of tasks follow this basic outline:

- Task 1: Compare robber photograph with potential suspect photograph and compute "similarity" score. All 250 processors can be used, resulting in 4 comparisons per processor.

- Task 2: Locally compare the 4 resulting similarity scores on each of the 250 processors, and output the highest (maximum) score. (This minimizes communication overhead.)

- Task 3: Each of 249 processors sends its maximum score output to the remaining processor, which compares its local maximum with those from the other processors, producing a high score from all 1000 comparisons and the most likely suspect.

A diagram depicting the above processing strategy is given in Appendix B.

4.4 Survey Role in Educational Study

The PoPS should be considered a formative rather than a summative assessment, and should be employed repetitively as a diagnostic test of student comprehension

of parallel design concepts. The intent of the PoPS is not only to help students progress in the growing field of parallel computing, but also to modify and validate instructional timeliness and pedagogical techniques related to this important topic.

Chapter 2 addresses the challenge of identifying at which points during a CS education it is best to present students with the ideas and foundation of "parallel-thinking." An instrument like the PoPS can provide one measure as to whether the minds of students in the early stages of a CS program are receptive to concepts in parallelism. Monitoring survey scores of a CS1 class both *before* and *after* the presentation of a module on parallel computing would provide some baseline information as to whether core concepts were fully absorbed and understood by the attending students. In the extreme theoretical case, an individual could take the PoPS, be exposed to a few minutes clarification on specific aspects of parallelism, and then immediately retake the PoPS. The performance improvement or reduction between the two administrations of the test would reveal something of the effectiveness of the intervening instruction.

The PoPS is the assessment instrument used in the two-factor mixed design educational study of CS1-level undergraduates described in Chapter 3. The PoPS will help answer the research question about the appropriateness of presenting material about parallelism to an introductory CS class.

The validity of PoPS as an instrument for assessing a fundamental level of competency in parallel design has been discussed in detail in the previous section. The tasks and questions included in PoPS elicit information about student perceptions and beliefs on parallel behavior and student aptitude on finding concurrency. Since PoPS is a new assessment instrument, the reliability of the survey is purely descriptive and based primarily on the consistency inherent in multiple-choice tests. Like mathematics tests, the PoPS multiple-choice questions have only one answer and are likely to show consistency across the different test times and students. Integrated re-

liability for the design question is achieved by utilizing two separate objective graders and comparing their scores for consistency.

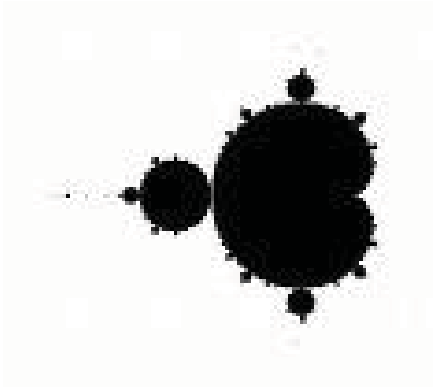


Figure 4.1: The Mandelbrot Set

CHAPTER 5

CLASSROOM INTERVENTIONS TO PROMOTE PARALLEL THINKING

The experimental educational study described in Chapter 3 employed two primary methods for introducing parallel design concepts to beginning CS1-level students. As indicated by the levels applied to the independent groups along the horizontal axis in Figure 3.1, separate sections of the CS1400 *Fundamentals of Programming* class were exposed to:

1. A three week module of lecture/presentations of parallel concepts and hands-on programming exercises applying those concepts using a software visual Parallel Analysis Tool (PAT) especially designed and written by the author for CS1-level instruction (level A_1).

2. A three week module of lecture/presentations of parallel concepts with written exercises suitable for CS1-level undergraduates (level A_2).

3. No instruction about parallel concepts. Traditional initial three week CS1-level instruction involving sequential analysis and design (the control group A_3).

This chapter delineates the module content and elucidates the pedagogical strategy utilized in this study. Regarding the overall educational study, the module was not designed such that questions on the PoPS were directly addressed in the classroom discussions. The intent of the module design was to furnish students pertinent information about parallel design so that these newly acquired skills and knowledge

could translate to better performance on the PoPS. The forthcoming discussion can be essentially categorized into two fundamental classroom interventions as outlined in the above list: 1) instructional intervention, and 2) visual analysis tool (PAT) intervention.

5.1 Instructional Intervention

The module on parallel computing was scheduled as the initial topic of investigation in the CS1400 *Fundamentals of Programming* CS1-level class, spanning roughly the first three weeks of the semester. Because CS1400 is an introductory class, upon entering the classroom students were assumed to have no significant expertise in parallel systems, computer architecture, or programming. Students did have some broad exposure to computing concepts through the CS0 prerequisite course *Foundations of Computer Science*, but certainly not enough to claim sufficient mastery of any single area in software development or computational systems.

With such limited student experience, the module was necessarily designed to target the core conceptual underpinnings of parallel computation. Any technical terminology was carefully explained as it was introduced. The module content had to be relatively easy to understand by the beginning student without extensive discussion or examples, and the topics clearly had to be restricted in scope. Nonetheless, a relatively substantial foundation about parallel computing could be established in this short amount of time. The objectives of this parallel concepts module can be concisely summarized as shown in Figure 5.1.

In an attempt to connect with the diverse learning styles of CS1 students, instructor presentation of the module content relied not only on oral or written descriptions of parallel concepts, but also on explicit visualizations of parallel design using primarily the Activity diagram from the UML and a new, original Processor-Time (P-T) diagram created by the author to be described later. A single textbook

was not required for this module. Instead, specific introductory chapters were used from Quinn's *Parallel Programming in C with MPI and OpenMP* (Chapter 1)[57] and Kaminsky's *Building Parallel Programs* (Chapters 1-3)[37]. The selected readings from these texts provided sufficient overview and descriptive explanations of parallel systems while avoiding any detailed investigation of computational strategies or mathematical performance measures too advanced for the novice CS1 student.

The weekly lesson plan and topic sequence for the parallel concepts module are given in Table 5.1. Immediately following are additional summary explanations of how some of the more substantial topics listed in the table were presented within the context of the CS1 parallel concepts module.

5.1.1 Multicore Concepts/Parallel Computing

Moore's Law is initially presented with special emphasis on how it significantly drives the cultural expectations of computer performance over time. A brief history of processor architecture is provided in order to show that a noteworthy inflection point I_{proc} occurred in the early-80's with regards to computing speed. Prior to I_{proc} improvements in processing performance were dependent on the physics and chemistry of integrated circuit technology whereas after I_{proc} it was advances in microarchitecture on recently developed microprocessors that provided the innovations for increasing performance. A subset of these optimization efforts included finding parallelism in the way low level instructions are managed and manipulated.

The objective of this discussion is to cultivate student appreciation of how parallel processing was, until recently, conveniently hidden from the mainstream software developer amid the intricacies of hardware implementations that capitalized on instruction level parallelism (ILP). In the current multicore and many-core era, programmers must be minimally aware of the number of available processing elements on the development machine, a sentiment embodied in the classroom discussion of

this topic by the expressions "Hardware drives software," and "Architecture defines the algorithm."

In addition, the role of numerical simulation as a replacement for physical experimentation within the scientific method is discussed. Some investigations are just too complex, expensive, or outright impossible to perform in real-world laboratories. Examples of applications that can benefit from parallel programming are presented, such as weather forecasting, protein sequence matching, and planet/star cluster formation simulations.

5.1.2 Dependencies in Real-World and Computational Problems

Dependencies are emphasized early in this treatment of parallel computing. The capacity to apply parallelism to computational problems is constrained by dependencies among related activities. The student's aptitude in recognizing dependencies helps her succeed in the equally important task of finding concurrency. The fundamental question posed by students engaged in parallel thinking is: "Which operations can be executed simultaneously?"

Students are acquainted with the fact that this question represents a fairly common cognitive experience, one which they have applied and developed many times prior to enrolling in a *Fundamentals of Programming* class. The popular Fox-Goose-Grain puzzle is proffered as a prime example of how dependencies heavily influence the solutions to real-world problems.

A farmer owns a single boat and needs to deliver his fox, goose, and sack of grain intact from one side of a river to the other side. The farmer can only carry one of the three in his boat during a trip across the river. However, if the fox and goose are left alone, the fox will eat the goose. If the goose and the sack of grain are left isolated, the goose will eat the sack of grain. During the class discussion of the solution of this classic puzzle, the dependencies existing among the different characters and objects

in the scenario are highlighted.

Immediately following this discussion, students are asked to design an algorithm for finding the maximum of two test score averages, each calculated from two separate, independent student populations. The objective is to have students not only recoginze the sequential dependency that exists between the averaging and reduction (maximum) operations, but also discover that the two independent averaging calculations can be done concurrently.

5.1.3 Parallel Architectures and Configurations

This topic discussion begins with the basic design of the von Neumann architecture, and shows briefly how CPUs can be duplicated to construct a symmetric multiprocessor (SMP) configuration within a single computing machine. Given a description of the SMP environment, CS1 students are able to easily make the intellectual leap necessary to abstract the basic organization of the current multicore architecture, in which two or more CPUs (cores) can be integrated on a single chip. Students are instructed that SMP and cache memory are two architectural features that influence the construction of high-level programs, reinforcing the interdependence of hardware and software.

Four principal parallel computing architectures are addressed:

1. SMP, the foundation for multicore configurations described above.

2. Clusters, a collection of separate processor nodes interconnected using Ethernet for commodity clusters like Beowulf, or InfiniBand/Myrinet for dedicated rack-mounted servers.

3. Hybrid, a cluster in which the processor nodes are SMP machines

4. Grid, a highly distributed network of independent machines working on isolated computations. Examples include SETI@home and the Great Internet Mersenne

Prime Search (GIMPS)

The Grid configuration listed above helps to introduce the concept of embarrassingly parallel problems in which communication or interaction between computing nodes is effectively absent.

5.1.4 Speedup as a Performance Measure

Quinn [57] supplies the following expression for speedup:

$$\psi(n,p) \leq \frac{\sigma(n)+\phi(n)}{\sigma(n)+\phi(n)/p+\kappa(n,p)} \tag{5.1}$$

CS1 student focus and motivation would most likely experience a sudden and dramatic decline on viewing this mathematical description of an important performance metric of parallel programs. Extracting the essential qualitative meaning of speedup from the above formula is the challenge for the CS1 instructor.

Kaminsky's [37] diagrammatic description of speedup, shown in Figure 5.2, is more suitable for the beginning CS student. A rudimentary Gantt chart indicates the time required for a computation to be executed on a single processor. Positioned immediately below the first chart, a second Gantt chart shows the time required if four processors are applied to this massively (embarrassingly) parallel problem. In this second chart, the student views four bars, each a quarter of the length of the bar drawn in the first chart. Consequently, this straightforward visualization of execution time makes explicit the conceptual definition of speedup:

$$Speedup = \frac{SequentialExecutionTime}{ParallelExecutionTime} \tag{5.2}$$

Students are forewarned that real-world parallel computing is not necessarily as clean-cut as the preceding example, but speedup is often the initial main motivation for seeking a parallel solution to a sequential problem. In conjunction with the specific

scenario just described, students are subsequently asked about the case in which the amount of data to be processed is increased correspondingly by four. The bars in the parallel Gantt chart can be extended to the same length as the single bar in the sequential Gantt chart, illustrating the main message that, using the parallel strategy, four times the information can be processed when compared with the sequential, single CPU solution. This exposition illuminates the different perspective afforded by Gustafson's Law, in which the size of the problem should be scaled to match the available computational capacity.

5.1.5 Dependencies Exhibited in Code

Small pseudocode examples taken directly from Quinn [57] are used to highlight two key examples of dependencies relative to program construction: for-loop parallelism and task parallelism. Given the CS1 student's relative inexperience with program design and syntax, the intent is not to provide an exhaustive treatment of all potential occurrences of dependencies among program statements, but to demonstrate a few obvious cases where dependencies can affect program development primarily so that students can ground the ongoing theoretical discussion to actual code implementation.

An example of a fully parallelizable for-loop is presented:

```
for i ← 0 to 99 do
    a[i] = b[i] + c[i]
endfor
```

The statement within the body of the above for-loop can be safely distributed among 100 dedicated processors without concern for data synchronization issues. Students are then confronted with:

```
a[0] ← c[0]
for i ← 1 to 99 do
```

```
    a[i] = a[i-1] + c[i]
endfor
```

The above illustrates a data dependency between subsequent executions of the statement in the body of the loop. This simple example serves to bring home the important point that proper application of parallelization requires sharpening one's ability to detect and analyze possible dependencies within code.

The issue of data dependency remains at the forefront in an example used to demonstrate how parallelization efforts should always consider the correct execution order of a sequence of tasks:

(1) a $\leftarrow$ 2

(2) b $\leftarrow$ 3

(3) m $\leftarrow$ (a + b)/2

(4) s $\leftarrow$ (a^2 + b^2)/2

(5) v $\leftarrow$ s - m^2

Instructions (1) and (2) may be performed concurrently, followed by the simultaneous execution of instructions (3) and (4), which can then be followed by instruction (5). By analyzing this code section in this way, students are engaged in applying the technique of out-of-order execution so often used in computer architecture design to exploit instruction level parallelism.

Despite the fact that applying parallelization strategies to arithmetic tasks of extremely short duration may prove counterproductive in actual practice, the objective of the above examples is for students to recognize the tradeoffs between concurrency and data dependencies within the context of computational requirements.

5.1.6 Activity Diagrams

Representing the flow of a parallel program in diagrammatic form is critical for beginning CS1 students attempting to model concurrency. As mentioned above, stu-

dents bring various learning styles into the classroom, and Activity diagrams provide visualizations of parallel concepts that will appeal to a sizable fraction of those students. Beyond the pedagogical benefits, the Activity diagram is an established tool within the Unified Modeling Language (UML), often used in industry and academia to express software engineering design and specification.

Fowler's book on the UML provides a detailed description of the Activity diagram [26]. The primary advantage of the Activity diagram and the main reason it is included in the parallel concepts module is that it has the capacity to model concurrent behavior. Typical flow charts can sufficiently represent standard control structures, but come up short when attempting to depict parallel operations. The Activity diagram includes both a *fork* and *join* synchronization bar which effectively define barriers around concurrent activities. The standard description of a *join* specifies that all flows going into the *join* must reach it before processing may continue. The Activity diagram helps the student explicitly visualize the sequential and parallel portions of an executing program.

The parallel concepts module initially presents Activity diagrams as a tool for representing real-world situations in which parallelism could potentially arise. When describing the popular conception of "human" multitasking, the **Attending Class** Activity diagram in Figure 5.3 is proposed as a representation of how students may view the multitasking experience. The accuracy of this model is examined with respect to the current research on human cognitive limitations in performing focused tasks simultaneously. Are the three tasks flanked by the the *fork* and *join* bars in Figure 5.3 being performed concurrently? This question stimulates classroom discussion regarding the difference between multitasking and pure concurrent processes, and also compels the students and instructor to clearly define how parallelism is represented in an Activity diagram.

5.1.7 Multitasking and Context Switching

The importance of this topic relative to exploring parallel concepts at the CS1 level has been discussed both in the prior subsection about Activity diagrams and more extensively in Section 4.1.

5.1.8 Data Parallelism, Functional Parallelism, and Pipelining

The subject matter in Week 3 assimilates the material from the prior two weeks by emphasizing the design, structure, and interpretation of parallel programs. The previous coverage of dependency analysis and the visualizations provided by Activity Diagrams dovetail to help clarify the concepts of data parallelism and functional parallelism. Figures 5.4a and 5.4b depict the basic structure of data parallelism and functional parallelism, respectively. A, B, C, D, and E in the figures are considered separate activities.

Data parallelism denotes the same operation performed on different data concurrently, similar to the fundamental computing strategy of vector processors and GPUs. Functional parallelism shows simultaneous execution of different operations on different data. Through these representations, students begin to recognize the important and necessary parallel programming task of appropriately partitioning data to processing elements.

Pipelining presents a unique challenge within the context of parallel program structure because Activity diagram representations of this processing strategy are not particularly good at indicating potential parallelism. In fact, despite the assembly line approach in which each stage is working on a particular *part* of the problem simultaneously, pipelining exhibits a conventional sequential flow when depicted by an Activity diagram as shown in Figure 5.5.

The data mapping is not obvious for this pipelining scenario. However, introducing this variant of parallel computation offers the student an alternative perspective on

the data dependency example given earlier:

```
a[0] ← c[0]
for i ← 1 to 99 do
    a[i] = a[i-1] + c[i]
endfor
```

If the above for-loop is unrolled, the following sequence of statements emerges:

```
a[0] ← c[0]
a[1] ← a[0] + c[1]
a[2] ← a[1] + c[2]
a[3] ← a[2] + c[3]
...
```

Now if K independent data sets of size 100 (each represented by a different `c` array) need to be processed, the benefits of the pipeline computation scheme become evident. Each individual statement in the unrolled loop still maps to an activity, but the data mapping now cuts across each corresponding element of the K input data sets.

For convenience, students are introduced to a shorthand notation which can simplify Activity diagram representations of data parallelism. As shown in Figure 5.6, a collection of simultaneous data parallel operations can be condensed to a single activity of the same name, and the input transition to the activity is annotated with the number of active parallel processors.

5.1.9 Parallel Program Structure

Parallel computing configurations are categorized in various ways, and the most noteworthy is the original classification known as Flynn's taxonomy [25]. Although this classification scheme is widely accepted, it has limited value for the CS1 level

student, who may be somewhat baffled by abstractions like data and instruction streams.

Flynn's approach focused primarily on parallel computing architecture and organization, but functionality is also implied through his systematic groupings. The von Neumann machine, represented as SISD in Flynn's classification, is essentially restricted to sequential processing. The other three categories (SIMD, MISD, and MIMD) constitute the different variations of parallel design and functionality. Instead of memorizing a pedantic, intellectually remote template for parallel machines, beginning students in parallel computing will more likely be stimulated by descriptions of parallel configurations in which real-world problems are being addressed.

Kaminsky introduces three ways to structure a parallel program, which he calls *patterns*, based primarily on computational objectives: result parallelism, agenda parallelism, and specialist parallelism [37]. As with any classification scheme of parallel design or architecture, the boundaries among the different groupings are not necessarily well defined or clear cut. However, these patterns can be immediately connected to practical problems, thereby drawing students more readily into strategies related to parallel thinking.

Problems that employ result parallelism include pixel generation for computer images, and N-body calculations like deriving the physical positions of planets or stars under both Newtonian and relativistic influences. Agenda parallelism typically targets a single result, so an example of this kind of pattern includes searching a massive DNA and protein sequence database for similarities to an input query sequence. The task appropriation and associated data scattering and gathering that characterizes the master-worker pattern utilized often in cluster configurations can be introduced when an agenda parallel problem has many more tasks than processors. The specialist pattern aligns nicely with the pipelining solution described earlier.

Conceptualizing these modes of parallelism and connecting them with case stud-

ies ultimately help students establish the intellectual foundation needed to formulate solutions to the problems posed by the Perceptions of Parallelism Survey (PoPS) evaluation instrument discussed in Chapter 4. In fact, the student's success in responding to the PoPS design "essay" question scenario relies solely on their understanding of agenda parallelism, dependencies, and reduction.

5.1.10 Mapping Parallel Problems to Activity Diagrams

This topic is integrated into the discussions on Activity diagrams, data parallelism, functional parallelism, and pipelining in the previous subsections.

5.1.11 Modeling Performance Using the Processor-Time Diagram

In the spirit of providing compact, information-dense visual renderings of the efficiency of parallel computing solutions, the author created and experimented with a novel format for representing parallel computation time and communication time. This new Processor-Time (P-T) diagram depicts the individual processors (or processes) involved in a parallel computation, along with the explicit interactions that occur among the processing elements during the execution of the parallel program.

The Processor-Time diagram is a variation on the UML Sequence diagram in which the objects displayed horizontally across the top of a Sequence diagram are replaced by processing elements, and the message arrows represent some form of information exchange between two of those processing elements. Also, the angle of the message arrow with respect to the lifeline of the processing element sending the message departs from ninety degrees in proportion to the overhead of the communication. For example, a message arrow perpendicular to the lifeline of the sending processor denotes virtually instant communication (zero overhead). Unlike Sequence diagrams, P-T diagrams have only one type of message, since their primary purpose is to give the viewer a relative sense of the time investment involved in communication. Similar to Sequence diagrams, time in P-T diagrams progresses vertically along lifelines from

top to bottom.

Figure 5.7 shows how a P-T diagram models the time evolution of the **Attending Class** multitasking activities depicted in Figure 5.3. Note that single nodes (dots) represent the processes, which are labeled as "Listen," "Surf," and "Text." In the diagram, lifelines are the vertical dotted lines. Effectively, any node drawn along a lifeline indicates that some kind of computational processing is being performed. As mentioned above, communication between processes is depicted using a solid, single-headed arrow directed from the sending node to the receiving node. Communication within P-T diagrams is an umbrella term which comprises any type of non-computational event, including latency, transmission time, barrier delays, load balancing tasks, synchronization, etc. In the P-T diagram of Figure 5.7, the communication arrows represent context switches, an overhead that adds to the overall execution time of the collection of multitasked activities.

The principal benefit of the P-T diagram is its capacity to provide the student a comprehensive snapshot of parallel program performance. Students can apply an immediate visual interpretation to a given P-T diagram based on the rule that more efficient parallel programs are rendered in P-T format as generally "shallower" and "wider."

Note that the length or duration of the vertical time axis is made shorter or "shallower" if computation eclipses communication in the parallel program implementation due to the relatively small size of the computation symbol (dot) as plotted on a lifeline when contrasted with the communication symbol (arrow). A "wider" P-T diagram implies that all available processors are being used in some capacity to solve the programming problem. This full utilization of processors, however, may be offset by increased communication activity. One of the chief purposes of the P-T is to illustrate in diagrammatic form these tradeoffs that arise in parallel design between processor usage and communication overhead.

For example, the extreme case of an embarrassingly parallel problem using five processors is illustrated by the P-T diagram in Figure 5.8. The diagram depicts the ideal situation in which independent processing units are performing simultaneous computations with no interaction. The diagram ignores communication overhead associated with initial task or data distribution to participating nodes, as well as any gathering or reduction operation on the results generated by the processing nodes. The P-T representation reveals that the entire processor pool is being utilized (the "wide" criterion) and all execution time is devoted to computation (the "shallow" criterion).

Compare the embarrassingly parallel instance with the P-T representation in Figure 5.9 of a parallel program that utilizes only two of five processing elements and involves three distinct communication operations between the two participating processors. The representation in Figure 5.9 is noticeably narrower and deeper in terms of P-T criteria. Thus, with a single glance, CS1 students can use P-T diagrams to make rough assessments regarding parallel program performance, which will help build student confidence in assigning qualitative labels to different parallel designs and also encourage students to pursue a more detailed analysis of the parallel programming strategy portrayed by the P-T diagram.

The instructional intervention described so far also includes a homework assignment that hones the student's ability to identify potential concurrency within code. The excerpts of the assignment listed in Appendix D combine the student's first exposure to Java language features with the broad concepts of parallel design discussed in class. Students are requested to go through the steps in the program development cycle using code whose performance may be improved through proper application of parallelization. For-loop parallelization and data dependencies are both highlighted in these exercises. Students are asked to simply annotate those portions of the code that they believe may benefit from parallel design strategies.

5.2 The Parallel Analysis Tool (PAT)

The current landscape of software pedagogical tools for parallel computing was explored at length in Chapter 2. Visual analysis software predominantly targets the time capture and rendering of multithreaded activities, highlighting the interaction among various synchronization objects (e.g., mutexes) and simultaneously executing threads. To be effective in the classroom environment, these tools presuppose some background in system-level thread mechanisms and primitives. Consequently, this type of instructional software is best suited for students in mid-level or advanced classes in the CS curriculum.

As mentioned in Chapter 2, threading concepts may be introduced at the CS2-level as proposed by Bi and Beidler[9] or from a high-level applied programming perspective at the CS1-level as suggested by Bruce, Danyluk, and Murtaugh [12]. However, presenting the low-level thread model to beginning CS1 students would most likely prove distracting and confusing since the rudimentary idea of a "process" in the context of programming has not yet been firmly established in the mind of the student taking *Fundamentals of Programming*. In addition, the indispensability of threads in the discussion of high-level parallel concepts has yet to be determined since different and possibly better methods of expressing concurrency may be developed as discussed by Edward Lee [45]. To program parallel systems, recall that Lee proposed a novel coordination language with a visual syntax layered above and orthogonal to an established programming language providing mundane, low-level functionality.

This section discusses the development and operation of an original pedagogical visual software tool created by the author and specifically designed for CS1 level instruction.

5.2.1 Design

It should be stated outright that the primary intent of developing this visual tool in the context of this project was to provide an additional classroom intervention dimension to the educational study. The use of instructional multimedia or software is commonplace and widely accepted in teaching environments [40][48][53]. The pedagogical principles driving the design and development of the visual analysis tool described in this study were published in an earlier paper by the author [58]. This research does not seek to prove the superiority of a software tool over traditional instruction in increasing student comprehension of parallel concepts. Because the course of interest in this study emphasizes computer programming, the visual tool simply offers an alternative hands-on approach for students to observe and experiment with the practical effects of applying concurrency to actual running code.

Regarding software tools in the classroom, the author subscribes to the utilitarian philosophy expressed succinctly by Hestenes, et al. that "technology by itself cannot improve instruction" [33], and by Jaron Lanier that "the value of a tool is its usefulness in accomplishing a task" [44].

The main objective of the design effort was to distill core information about parallel program design and performance into a compact visual interface format immediately intelligible to the CS1 student. A single CS1 appropriate performance metric that imparted the most information about the quality of the concurrency effort was sought.

Various measures can be used to evaluate parallel programs [57][71]. When comparing sequential and parallel performance for a single application or kernel, the number of primary interest is often the speedup factor $S(p, n)$, defined earlier in Equation 5.2 as the ratio of sequential execution time to parallel execution time. The variable p is the number of processors, n is a measure of the "size" of the program, and it is assumed that the "best" sequential algorithm is used for this relative analysis.

When scalability is factored in, the speedup computation assumes different forms depending on constraints such as constant problem size, constant parallel execution time, or memory availability. Although speedup is a useful measure which encourages the student programmer to increase the percentage of viable parallelizable code wherever possible, it abstracts away critical program design issues that are valid topics of study for CS1 students.

Developing parallel programs requires a working knowledge of both architecture and algorithms. As stated by Janssen and Nielsen [35], practitioners of parallel program design and implementation must consider the "major obstacles to achieving linear speedups... such as communication overhead and load imbalance." Sivasubramaniam [64] aptly describes the difference between linear speedup and real execution time as the sum of algorithmic overhead and interaction overhead. CS1 students would be well served by a performance metric or expression which opens discussion about both algorithmic overhead issues (inherently serial code, balanced task management) and interaction overhead (contention, latency, synchronization, resource management, and cache effects.)

A measure more suitable to the student and pedagogical needs of the CS1 classroom is the computation/communication ratio which will be referred to as the parallel quotient (PQ):

$$PQ(p,n) = \frac{t_{comp}(p,n)}{t_{comm}(p,n)} \tag{5.3}$$

It can be assumed that PQ is roughly proportional to speedup since increased communication overhead will tend to degrade speedup whereas increasing the time dedicated to computation will tend to improve speedup.

$$PQ(p,n) \propto S(p,n) \tag{5.4}$$

Note that PQ and the speedup factor are both functions of p and n. The execu-

tion time of a parallel program is the sum of t_{comp} and t_{comm}, with the first factor heavily dependent on algorithmic overhead and the second factor heavily dependent on interaction overhead.

The PQ number can be viewed as a measure of the quality of the parallelization effort. A high PQ would suggest that parallelization is generally effective and essentially the right course of action. A low PQ suggests possible diminishing returns through parallelization. CS1 students could benefit from the summary feedback provided by a single metric, which promptly conveys the pros or cons of a particular parallel design strategy without miring the student in detailed analysis. Essentially, code parallelization choices which increase the PQ are reinforced.

A PQ number could be determined not only for an entire application, but also for select sections of code. One of the challenges confronting the student programmer is identifying which parts of the code are good candidates for parallelization. By monitoring PQ values for code sections that possess inherent parallelization such as loops, the student will gain familiarity and confidence with the paradigm of parallel thinking.

If required, there will be ample time to introduce the student to threading concepts and detailed synchronization mechanisms such as semaphores and mutexes later in the CS program. The reductionist PQ analysis described here seeks to expose the CS1 student to important high level parallel design concepts at a crucial time in their programming careers, possibly circumventing adoption of a rigid approach in which only sequential thinking is applied to computing problems.

The challenge facing the CS1 student is one of *recognition*: given a programming model and underlying parallel architecture, which sections of code are good concurrency candidates? Any tool assisting students at this level of analysis should foster this recognition skill, reinforcing the student's parallelization choices with straightforward feedback measures. The PQ number described earlier offers a simple scalar

measurement that is generated by the Parallel Analysis Tool. The PAT provides the student a visualization format motivated by the UML Activity diagrams typically employed in software engineering.

Activity diagrams were discussed in detail in Section 5.1. In brief, the Activity diagram provides a "flow-chart" perspective of a computational process, with the important addition of synchronization bars that join or fork several actions. This visual notation helps to solidify the student's conceptualization of a set of parallel processes, and terms such as "fork" and "join" could potentially open discussion about more detailed topics related to processes and threads. Also, as suggested earlier, since it is commonly accepted that a student's primary learning modality falls into one of three categories – auditory, visual, kinesthetic – the visual component of the PAT will enhance the student's comprehension of parallel code behavior.

As a representative example adapted from Kaminsky [37] and used throughout the remainder of this discussion, CS1 students in this study are asked to model the basic flow of a program that determines if an integer is prime using the trial division algorithm. In the first version of this program listed in Figure 5.10, this prime test routine will be applied sequentially to a series of eight very large prime numbers, resulting in a relatively lengthy execution time (≈5.8 secs on a 2.59 GHz dual quad-core AMD Opteron server with 15.9 GB of RAM). Figure 5.11 shows the associated Activity diagram in which the labels for each activity correspond to the commented sections in the main method of the program.

Students examining the code example in Figure 5.10 may identify the loop that performs the prime test as a good candidate for concurrency since a significant amount of computation time is devoted to determining if a given number is prime. Similarly, students may regard the array initialization section as a reasonable place to apply parallelization because each separate statement within that section is performed independently. The PAT is designed to allow CS1 students to rapidly investigate the

validity of their parallel design decisions by using empirical execution times of both the sequential and parallel versions of the program to calculate experimental speedup and thus generate a feedback PQ value.

5.2.2 Operation

The PAT is fundamentally a source code translator and execution environment written in Java. The PAT converts a Java sequential program with specialized annotations into a Java parallel program that utilizes the openly available Parallel Java Library developed at the Rochester Institute of Technology by Alan Kaminsky. Parallel Java is described as an API and middleware for parallel programming in 100% Java on SMP parallel computers, cluster parallel computers, and hybrid SMP cluster parallel computers. As of this writing, the Parallel Java API can be downloaded from http://www.cs.rit.edu/∼ark/pj.shtml.

The startup screen shown in Figure 5.12 and the functional diagram illustrated in Figure 5.13 provide a comprehensive portrait of the component parts and features of the PAT design.

On startup, the PAT detects the number of available processors on the host platform and displays that value just underneath the title bar. This vital piece of information makes the user/student aware of the primary parameter of the underlying parallel architecture and provides important context for subsequent analysis in which program performance is measured as a function of the number of processors appropriated for the parallel computation.

The three user option functions located in the lower-left section of the PAT user interface are listed down the left hand side of Figure 5.13. The *Update Activity Diagram* function simply updates the current activity diagram in the Process View pane of the PAT. As mentioned earlier, the activity labels are obtained from the comments preceding each code section in the main method. The PAT provides basic

Java compile and run capabilities, generating program results in the Output Pane of the user interface. If specific code sections are marked by the programmer for parallelization (to be described below), the PAT *Compile & Run* option performs a translation of all these program sections to valid Parallel Java source code prior to compiling and executing this auto-generated parallel program.

The *Perform Parallel Analysis* option activates the core functional capability of the PAT, resulting in the calculation and display of PQ values. Before outlining how the PAT conducts parallel analysis, the technique by which the programmer identifies parallel code sections will be described. A straightforward annotation mechanism is used to indicate code sections to be translated to Parallel Java. This approach is motivated by the tagging strategies and metadata facility used in languages like Java and C#, and to some extent the pragmas that drive parallel compilation in OpenMP. Figure 5.14 gives an example of the annotation structure recognized by the PAT.

In this example, the prime number test loop from Figure 5.10 is selected by the programmer as a viable candidate for parallelization. Within the comment line that immediately precedes the loop, a **P=8** notation is prepended to the comment as shown in Figure 5.14. The meaning of this additional text is straightforward for the CS1 student: use a total of eight processors to parallelize the for-loop immediately following the annotated comment line. In general, the specialized PAT annotation that triggers translation of code sections to Parallel Java is $P = n$ where n is the number of processors to be employed during parallel execution of the code section.

Currently, the PAT supports two types of code structures that respond to the parallel annotation: 1) for-loop, and 2) sequence of statements delimited by curly braces. Future versions of the software can extend the translation capabilities such that other programmatic parallelizations (translations) can be realized, but the two code sections listed above were more than sufficient to allow CS1 students ample opportunities to experiment with and understand proper application of parallel con-

cepts in the context of working programs. Syntactic constraints regarding the proper placement of the annotated comment line with respect to the targeted parallel code section did not impede student progress when using the tool. Informative error messages that guide students in diagnosing annotation problems are an integral part of the PAT.

Returning to the *Perform Parallel Analysis* function depicted in Figure 5.13, the Java regular expression routines are used to parse the original source code for occurrences of parallel section annotations. If no annotations exist in the program, then an appropriate "No Parallel Sections Defined" message is displayed in the Output Pane. If one or more parallel sections are indicated, then the PAT performs the following steps for each individual section:

1. **Translate Target Code Section to Parallel Java:** Using parsing routines and the scaffolding provided by static code fragments, the PAT assembles Parallel Java source using the original source code and the number of processors specified in the parallel annotation as input. Figure 5.15 lists the Parallel Java code generated after PAT source translation of the code section given in Figure 5.14. Refer to Kaminsky for detailed descriptions of the Parallel Java library [37].

   Note that a team of eight threads is created and activated, with each thread assigned to a single execution of the program statements defined by the `run` method. If the original for loop had specified sixteen iterations rather than eight, then each of the eight threads would be assigned two tasks, which would be reflected in code as a change in the upper limit of the Parallel Java for loop from one to two: `for(int k = 0; k < 2; k++)`.

2. **Compile and Run Sequential Program:** Prior to executing the original sequential program, the PAT instruments the source code with two timing statements in order to calculate an execution time estimate for the program. Fig-

ure 5.16 shows the resulting sequential code after injecting timing commands both immediately before and after the code section of Figure 5.14. These statements utilize the Java nanosecond timer to provide a measure of the overall execution time of this code section. For this study, the accuracy of the nanosecond timer was verified in separate tests on the server hosting the PAT. The sequential execution time acquired by the PAT is used in the calculation of the PQ value.

3. **Compile and Run Parallel Program:** Note that the parallel code section in Figure 5.15 is flanked with timing statements similar to those in the sequential code in Figure 5.16. These statements utilize the Java nanosecond timer to provide a measure of the overall execution time of this parallel section. The parallel execution time acquired by the PAT is used in the calculation of the PQ value.

4. **Calculate the PQ Value:** Given both the sequential execution time and parallel execution of the code section of interest, a PQ value (speedup) can be calculated for this section using Equation 5.2.

After the previous steps are performed for each individually annotated code section, the PAT activity diagram in the Process View is updated with the calculated PQ value as shown in Figure 5.17. In this way, students can receive timely feedback on the quality of the parallelization effort using the PQ performance metric described earlier. To reinforce the classroom discussion about modeling parallel processes, note the activity diagram corresponding to the parallel section is updated using an appropriate join, fork, and the short hand notation depicted in Figure 5.6 indicating that eight processors have been recruited for the execution of this code section.

The CS1 students enrolled in the PAT w/Lecture class sessions of the educational study were assigned programming exercises which focused on performing experiments

with the PAT software. For a given code section, students were asked to adjust the number of processors and note the effect on the resulting PQ value. Through analysis of these recorded data, students were to recommend the optimum number of processors to use for the specific target code section running on the class host server. By emphasizing the broad interplay of both software and hardware in the empirical results, this exercise reinforces the importance of considering both algorithm and architecture in parallel design.

In a second exercise, students are confronted with the reality that parallelizing certain sections of code may yield disappointing results. Students must analyze the situation in which a parallel strategy is applied to a collection of very fast initialization statements. In this case, the time investment for thread setup and overhead significantly overwhelms any nominal performance gains attained by parallelizing statements with extremely short execution times. The parallel design sections of this assignment are included in Appendix F.

Figure 5.18 shows a sample output generated from PAT analysis when students parallelize code sections with short execution times (initialization statements) and long execution times (calculating prime tests). The contrast in the magnitude of the PQ number communicates the important message that indiscriminate use of concurrency is a viable concern when designing parallel programs. The extremely low PQ value that results when parallelizing eight very fast initialization statements among eight processors illustrates the subtle distinctions the student must make when practicing the art of finding concurrency in code.

Students will achieve both a comprehensive and analytical understanding of:

- **Fundamental parallel architectures and configurations**
- **Speedup: the single most intuitive performance measure of parallelism**
- **Parallel program design and structure**
- **How to recognize concurrency within code**

Figure 5.1: Objectives of Parallel Concepts Module

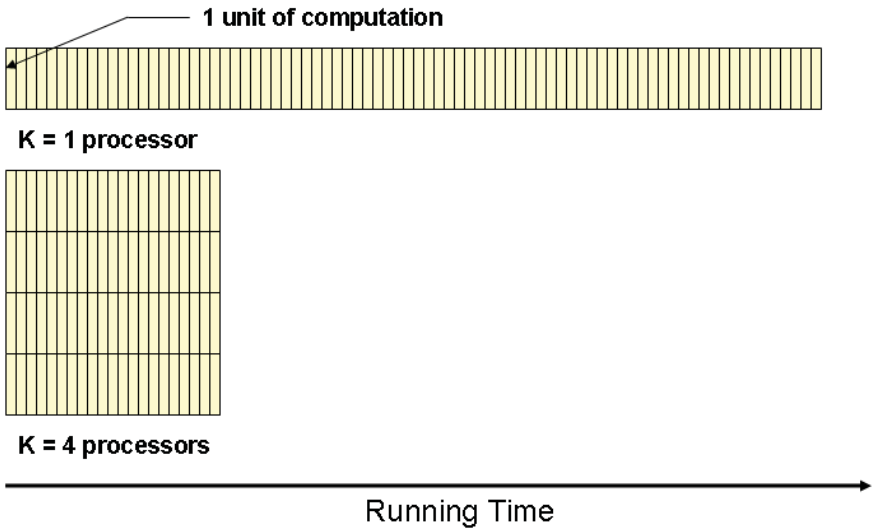


Figure 5.2: Speedup Diagram

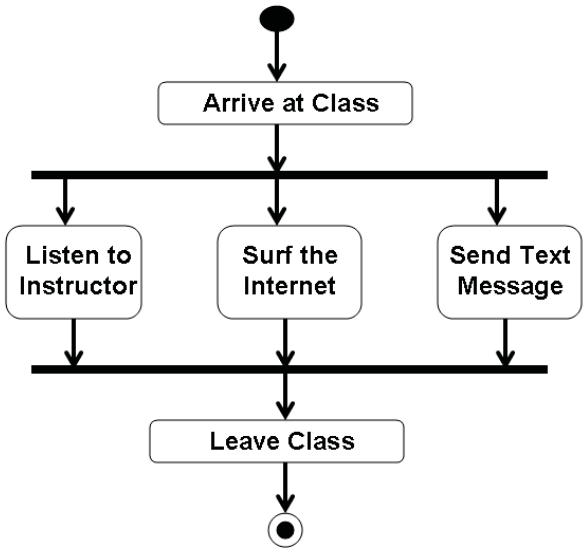


Figure 5.3: **Attending Class** Activity Diagram

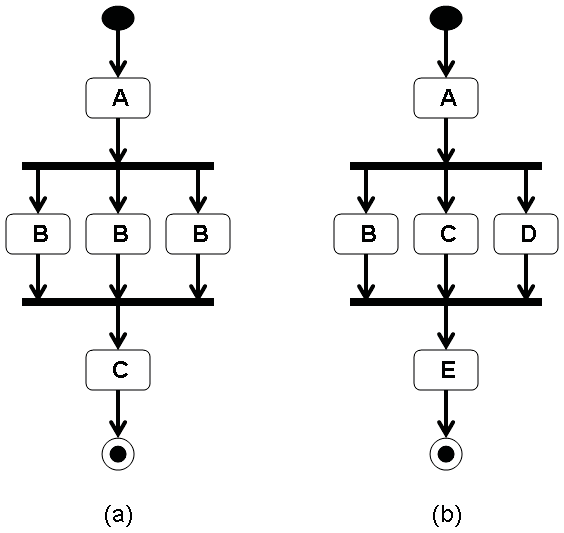


Figure 5.4: Data and Functional Parallelism

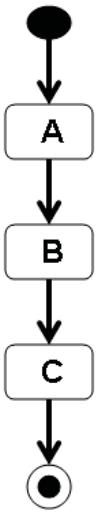


Figure 5.5: Activity Diagram Representation of Pipeline

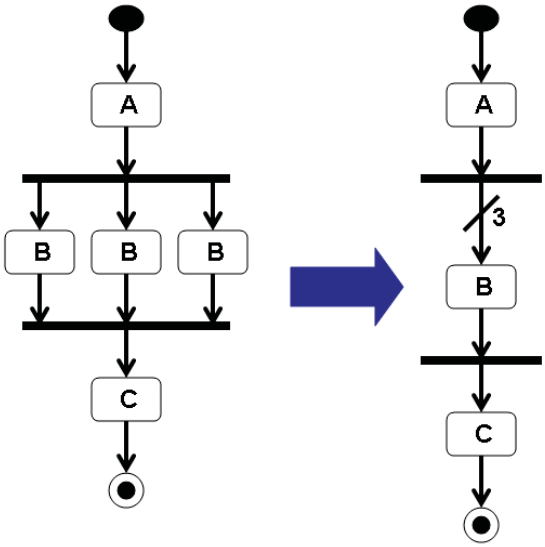


Figure 5.6: Shorthand Representation of Data Parallelism

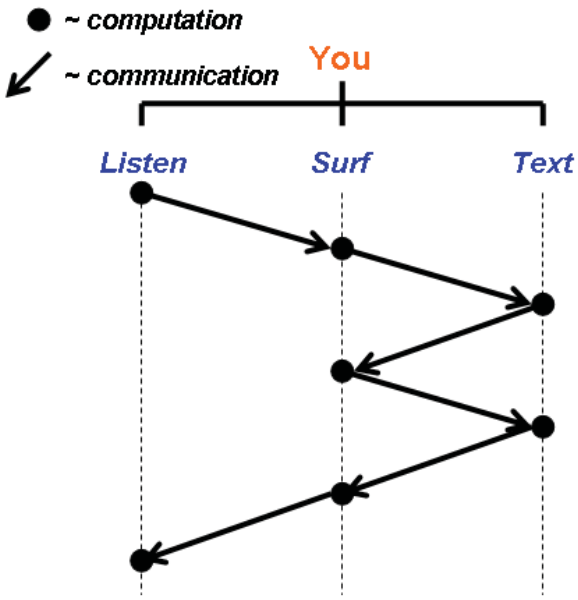


Figure 5.7: P-T Diagram of Multitasking

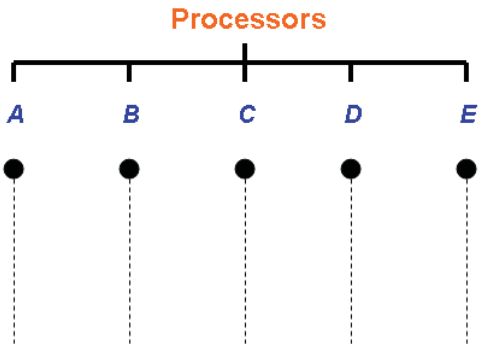


Figure 5.8: P-T Diagram of Embarrassingly Parallel Program

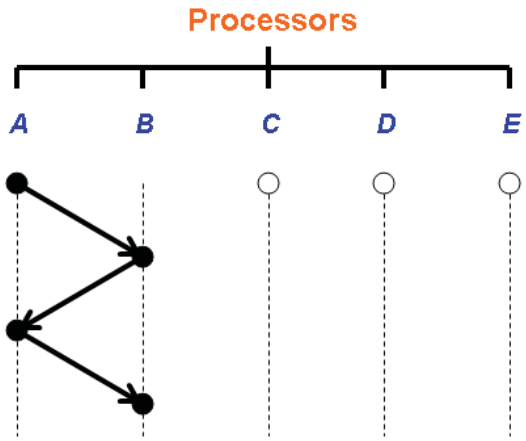


Figure 5.9: P-T Diagram of Inefficient Parallel Program

```
public class ParallelTest {

   static int n = 8; // Number of input values
   static long[] inputValues = new long[n];

   public static void main(String args[]) {

     long t1 = System.currentTimeMillis();

     /* Set Input Values */
     inputValues[0] = 1000000000000037L;
     inputValues[1] = 1000000000000091L;
     inputValues[2] = 1000000000000159L;
     inputValues[3] = 1000000000000187L;
     inputValues[4] = 1000000000000223L;
     inputValues[5] = 1000000000000241L;
     inputValues[6] = 1000000000000249L;
     inputValues[7] = 1000000000000259L;

     /* Test for Prime */
     for(int i = 0; i < n; i++)
     {
        isPrime(inputValues[i]);
     }

     /* Display program duration */
     long t2 = System.currentTimeMillis();
     System.out.println("Running Time:" + (t2-t1) + " msecs");
   }

   private static boolean isPrime(long x)
   {
        if( x % 2 == 0) return false;
        long p = 3;
        long psqr = p*p;
        while(psqr <= x)
        {
            if( x % p == 0 ) return false;
            p += 2;
            psqr = p*p;
        }
        return true;
   }
}
```

Figure 5.10: Prime Test Sequential Program (adapted from Kaminsky, 2010)

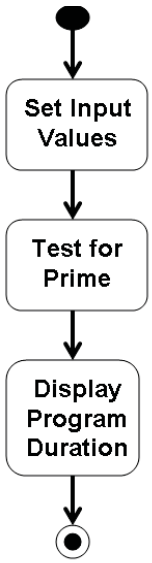


Figure 5.11: Activity Diagram for Prime Test Sequential Program

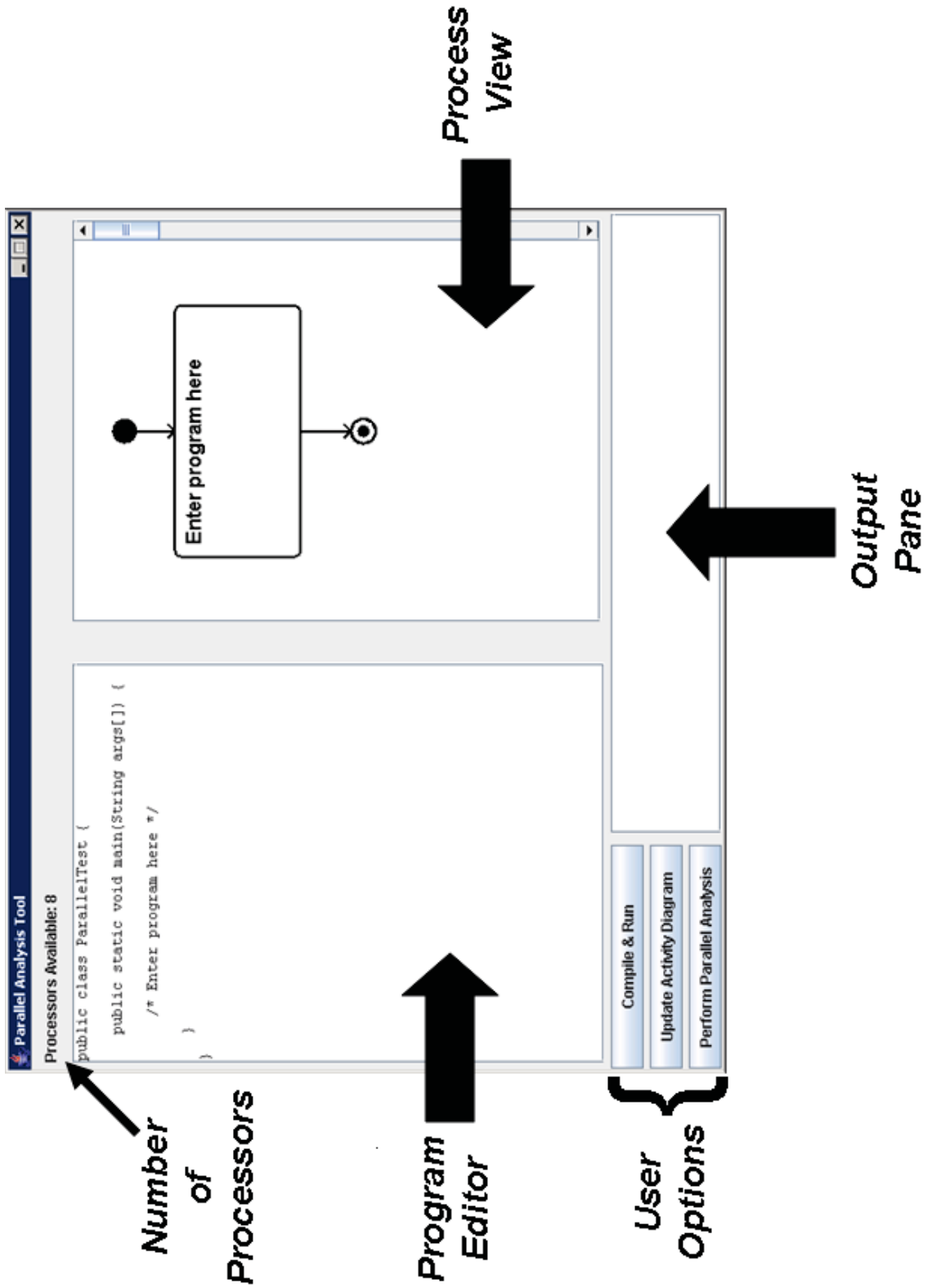


Figure 5.12: PAT User Interface

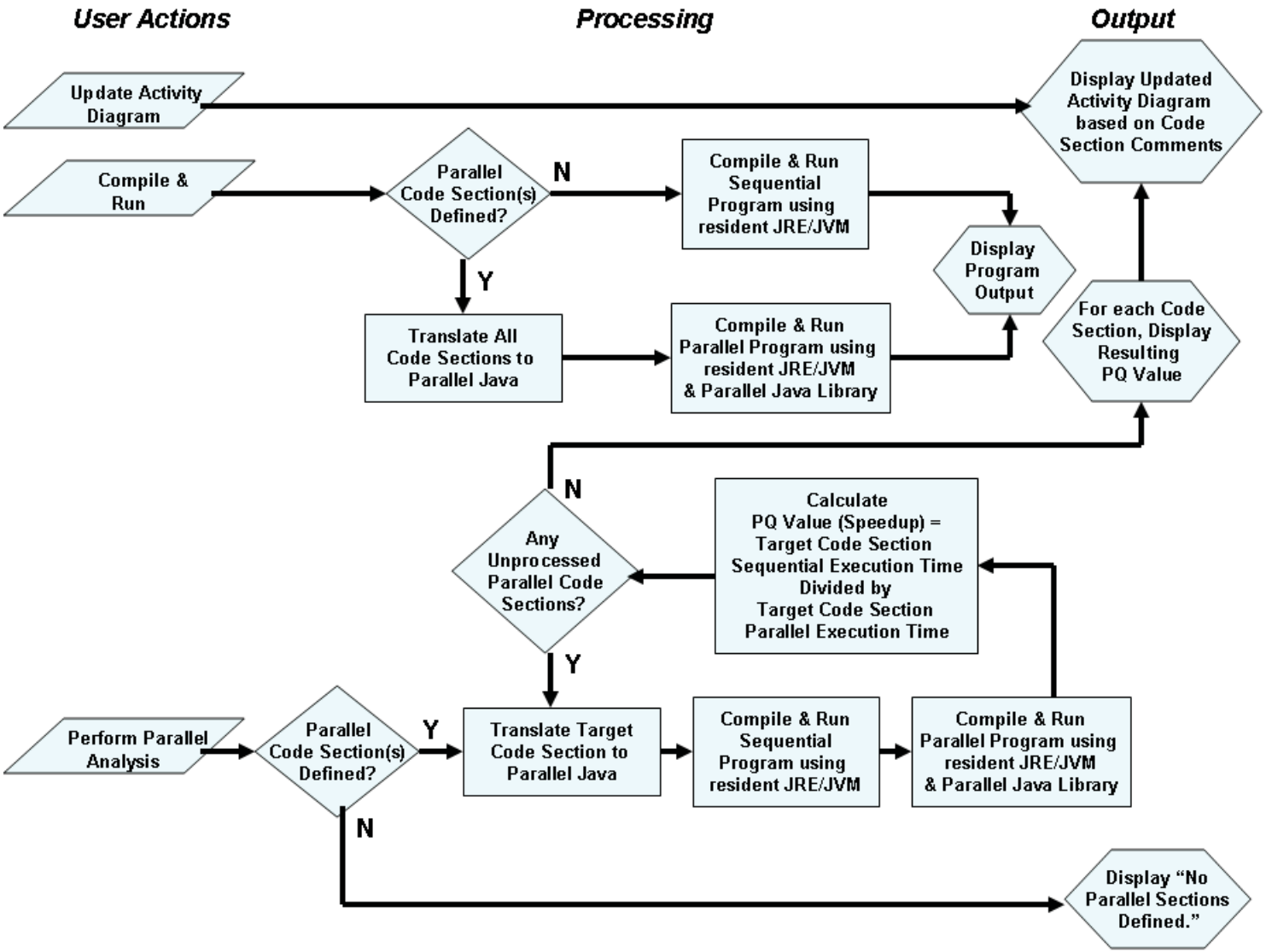


Figure 5.13: PAT Functional Diagram

```
     /* P=8 Test for Prime */
     for(int i = 0; i < n; i++)
     {
        isPrime(inputValues[i]);
     }
```

Figure 5.14: PAT Annotation to Define Parallel Code Sections

```
 // Begin program timing
SeqStartTime8061 = System.nanoTime();
     /* P=8 Test for Prime */
new ParallelTeam(8).execute (new ParallelRegion()
{
public void run()
{
int index = getThreadIndex();
for(int k = 0; k < 1; k++) {
int i = k * 8 + index;


        isPrime(inputValues[i]);
}
}
});
// End program timing
SeqStartTime8062= System.nanoTime();
```

Figure 5.15: PAT Translation of Prime Test Code Section to Parallel Java

```
// Begin program timing
SeqStartTime8061 = System.nanoTime();
     /* P=8 Test for Prime */
     for(int i = 0; i < n; i++)
     {
        isPrime(inputValues[i]);
     }
// End program timing
SeqStartTime8062= System.nanoTime();
```

Figure 5.16: PAT Code Section Instrumentation with Timing Statements

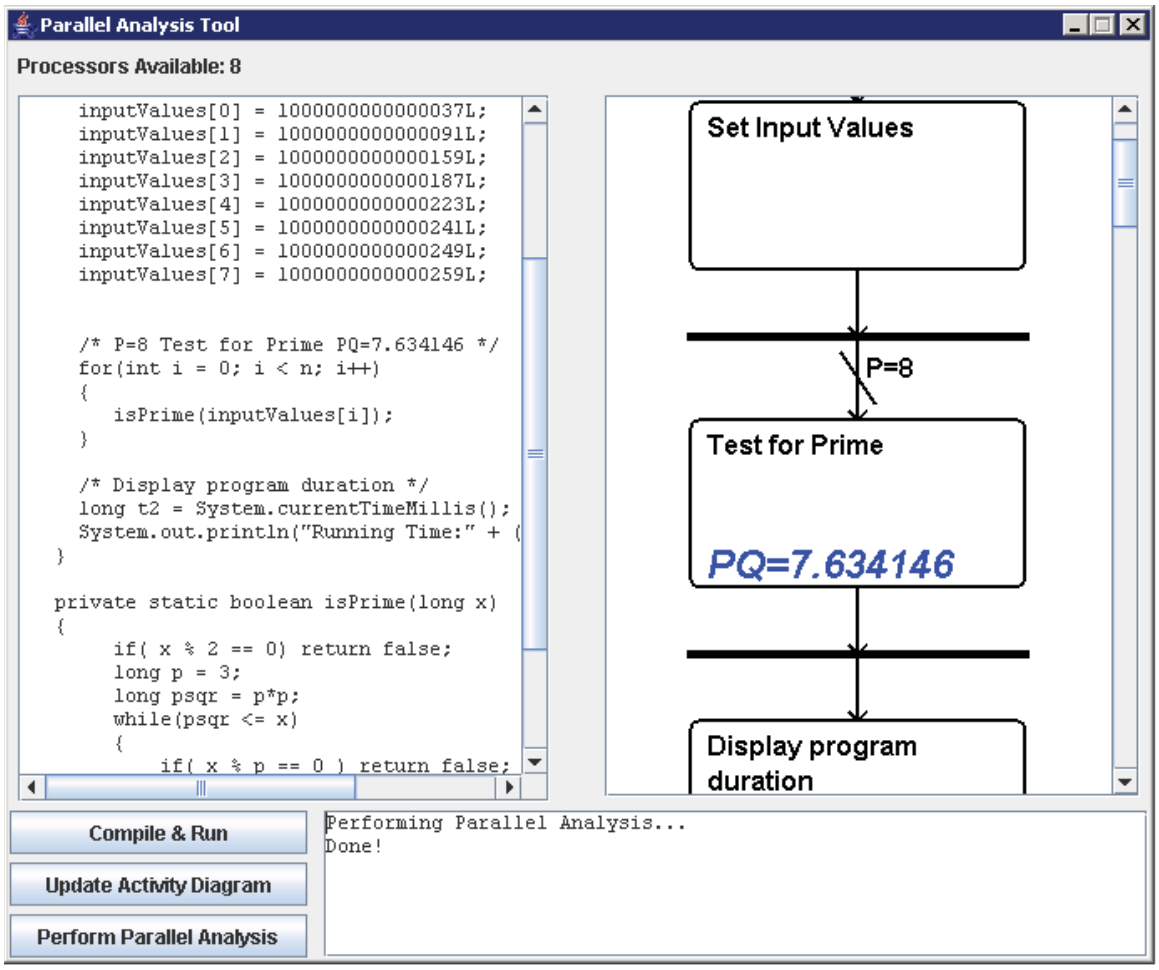


Figure 5.17: PAT after Parallel Analysis

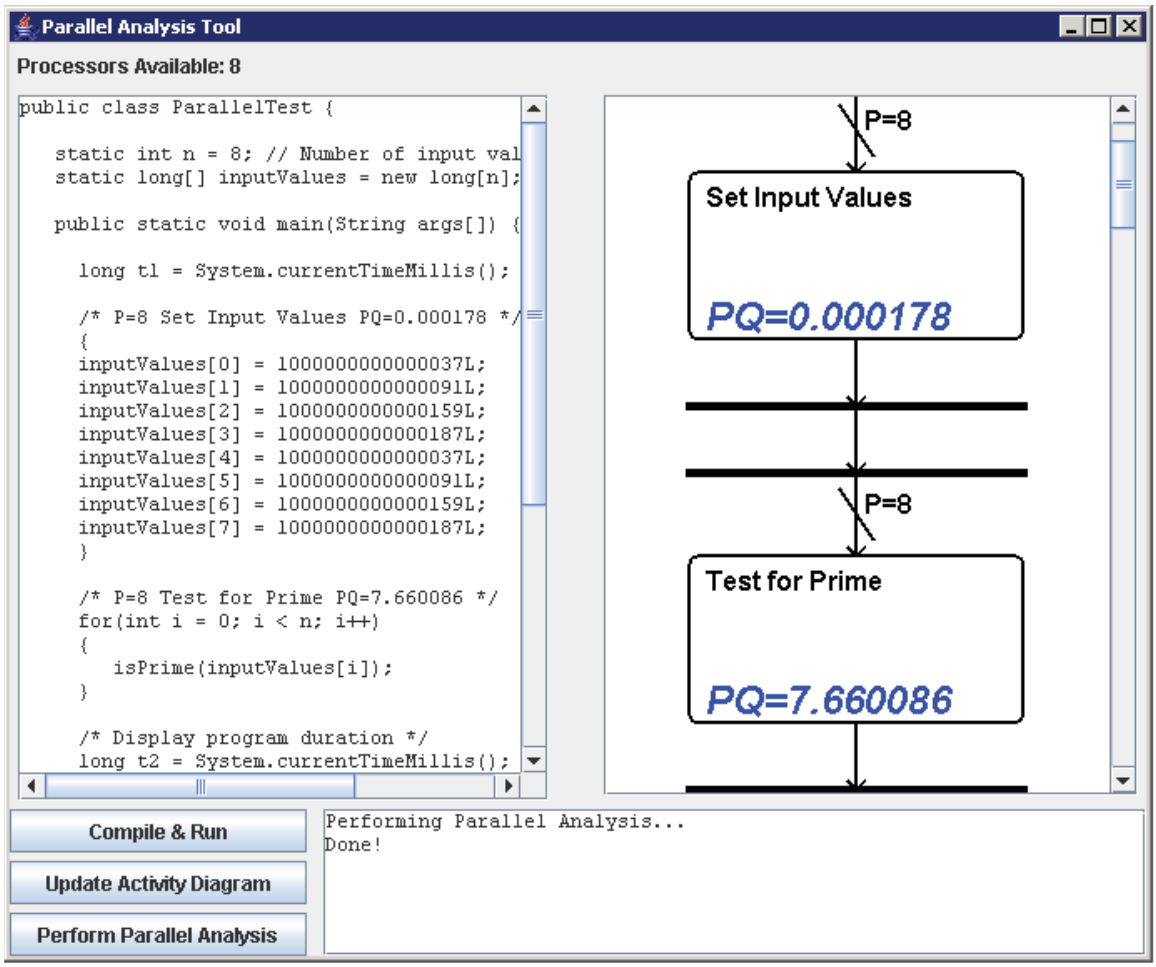


Figure 5.18: PAT after Parallelizing Initialization Statements

Table 5.1: Parallel Concepts Module Weekly Topic Coverage

Week 1	• Multicore Concepts/Parallel Computing • Top 500: http://www.top500.org/ • Dependencies in real-world and computational problems
Week 2	• Parallel Architectures and Configurations • Speedup as a Performance Measure • Dependencies exhibited in code: For-loop Parallelism & Parallelism among a Sequence of Tasks • Activity Diagrams • Multitasking and Context Switching
Week 3	• Data Parallelism, Functional Parallelism, & Pipelining • Parallel Program Structure: Result, Agenda, & Specialist Patterns • Mapping Parallel Problems to Activity Diagrams • Modeling Performance using The Processor-Time Diagram • The prospects for a widely accepted Parallel Programming Language

CHAPTER 6

DATA ANALYSIS AND RESULTS

Details regarding the research methodology employed in this educational study are given in Chapter 3. This chapter examines the results of statistical analyses applied to data gathered by administering the PoPS evaluation instrument to student subjects. As described in Chapter 4, the PoPS measures a student's comprehension of concepts related to parallel design and his/her overall capacity to recognize concurrency.

The results of this experimental study verify a statistically significant main effect such that student comprehension levels regarding parallel programming concepts as measured by the PoPS improve after the delivery of any CS1 three-week course module when compared with corresponding comprehension levels just prior to the three-week course module. Specifically, the comparison of PoPS scores between Week 3 and Week 1 test administrations yields a p-value < 0.001, and the comparison of PoPS scores between Week 9 and Week 1 test administrations yields a p-Value $= 0.011$.

Although the interaction effect between the instructional intervention mode and the repeated measures did not show statistical significance, the resulting p-value of 0.062 generated by this analysis came very close to the research hypothesis significance level α of 0.05.

Nonparametric analysis of the written essay design question confirms the parametric results. Namely, a main effect analysis using a Friedman test performed on the repeated measures aggregated from all intervention groups indicates a statistically significant improvement in scores from pretest to posttest. The rise in scores from pretest to recap-test generated a p-value of 0.067.

Figure 3.1 summarizes the research design: a 3 x 3 two-factor mixed-group approach in which the independent (between-subject) factor is defined by the instructional intervention and the repeated (within-subject) factor is defined by the time that the PoPS was administered to the student participants during the instructional process. The response variables were the total number of the PoPS 30 multiple-choice questions answered correctly for the parametric analysis, and the 1 to 6 Likert ratings assigned by two independent graders for the nonparametric analysis.

The surveys were given to students enrolled in three separate CS1400 *Fundamentals of Programming* classes during the Fall 2009 and Spring 2010 terms at Weber State University in Ogden, Utah. For the repeated measure, pretest occurred on the first scheduled day of the CS1400 class, posttest occurred during Week 3 after the initial parallel concepts module had been delivered, and the recap (retention) test occurred during Week 9. Students recorded their responses to the PoPS on a standardized paper answer sheet listed in Appendix C. Student names were requested but not required, and the anonymity of students was maintained during the correction of the design "essay" question by the study's two independent graders. The collective results from all surveys exhibited no prominent ceiling effect (measurement of student performance is limited by the survey maximum score) or floor effect (survey diffculty is too high such that most student subjects exhibit extremely poor performance). As a result, standard statistical tests that compared the data's central tendencies could be utilized.

No correlates (covariates) that could potentially affect the experimental outcome were identified for this study. Given the normal distribution of scholastic aptitude found in a typical medium-sized (15-40 students) introductory college classroom, it was assumed that each between-subject group started from an equivalent knowledge base, with no prior computational or instructional experience that may favor one independent group over any other with regard to general computer science skill sets

and, more importantly, to parallel design. This assumption was borne out by the fact that none of the study participants claimed any prior exposure to formal treatments about parallel concepts in computing.

The Statistical Package for Social Scientists (SPSS) was utilized to perform the necessary data analysis. The PoPS scores were allotted to each appropriate treatment in the 3 x 3 mixed-group design in order to derive appropriate descriptive statistics and mixed model analysis of variance (ANOVA). The differences in the mean values among the various treatments will indicate the degree of interaction between the experimental factors and/or any main effects. The repeated measure was the only main effect of interest since a comparison of the intervention groups independent of the PoPS administration time would not yield any useful information for this study. For the analysis of Likert ratings generated by grading the PoPS design question, a Friedman test that compares three matched groups is applied within subjects along the repeated measure, and a Kruskal-Wallis test comparing three or more unmatched groups is applied between subjects along the intervention factor.

6.1 Null Hypotheses

Per traditional nondirectional hypothesis testing, the research hypotheses stated in section 3.2 can be recast into the following null hypotheses:

NH1: With significance level $\alpha = 0.05$, CS1 students exposed to a three-week "lecture-only" course module on parallel design concepts will exhibit *no* statistically significant comprehension levels about this subject matter *after* the delivery of the course module when compared to students exposed to traditional CS1-level topics during the same time period.

NH2: With significance level $\alpha = 0.05$, CS1 students exposed to a three-week "lecture with software visual tool" course module on parallel design concepts will exhibit *no* statistically significant comprehension levels about this subject matter

after the delivery of the course module when compared to students exposed to traditional CS1-level topics during the same time period.

NH3: If there is no detectable interaction between the experimental factors, then with significance level $\alpha = 0.05$, CS1 students will exhibit *no* statistically significant comprehension levels about this subject matter *after* the delivery of any CS1 three-week course module when compared to comprehension levels just prior to the three-week course module.

The first two null hypotheses (NH1, NH2) posit a significant interaction effect among the two factors in the research design, essentially stating that the student subjects will show a different pattern of comprehension over time depending on which independent group of the between-subjects intervention factor they are associated with. The third null hypothesis (NH3) is considered if there is no significant interaction effect, at which point the repeated measures of all groups are evaluated over time.

6.2 Parametric Data Analysis

The response variable for the parametric data analysis was the student's number of correct answers from the 30 total multiple-choice question portion of the PoPS. In order to apply a mixed model ANOVA to these data, specific requirements and assumptions should be satisfied. These include:

Req. 1: The repeated measure variable should be interval level and the between-subject factor should be any level that defines groups.

Req. 2: The total number of subjects for a given repeated measure should be 10 plus the number of time periods making up the within-subject factor, and the minimum number in each cell should be 5 [68].

Req. 3: The response (dependent) variable is normally distributed in the population being sampled.

Req. 4: The homogeneity of variance for the between-subject factor.

Req. 5: The sphericity for the within-subject repeated measure.

Each of these requirements will be addressed below coincident with the presentation of experimental results. The significance level applied to the diagnostic tests described below (α=0.01) will be more conservative than the significance level applied to the research hypotheses (α=0.05).

Req. 1: As mentioned above, the response variable is identical to a standard test grade, which can be characterized not only as a valid interval variable, but as a ratio variable in which a zero indicates the absence of correct answers. The between-subject intervention factor is a mutually exclusive nominal (categorical) variable.

Req. 2: Descriptive statistics about the data set as generated by SPSS can be illuminating, as well as the survey numbers listed in the research design diagram of Figure 3.1. Table 6.1 shows the number of survey sample data for each treatment in this experimental study.

The descriptive statistics generated by SPSS shown in Table 6.2 verify the data sample count and provide an overview of the central tendency behavior of the data sets. Note these statistics are for the raw data; no missing data have been replaced to balance the repeated measures. The information contained in these charts clearly indicates that Req. 2 above is satisfied in that the number of cases for a given repeated measure exceeds the minimum threshold of 13, and the number of cases for any given treatment exceeds 5.

A visualization of the magnitude differences between treatment means is furnished by Figure 6.1, which shows a bar chart of the calculated averages. An initial observation made from this figure is that the relative upward trend in the Lecture/PAT

group average from posttest to recap-test differs in direction from the downward trends exhibited by the averages of the other two intervention groups from posttest to recap-test.

Another rendering of the data that provides insight into the distribution of the survey scores is given in Figure 6.2, which shows the average values and range of each treatment. A visual inspection of this figure reveals that most of the average values (marked by circles) are skewed slightly lower than the middle point of the range lines, indicating a denser concentration of scores in the lower part of the designated range.

For the Control group, the highest score was achieved by the same student subject for the pretest (16) and recap-test (20). For the Control group posttest, this student performed just one point below the high score of 19. For the Lecture Only group, the highest score was achieved by the same student subject for the pretest (22) and posttest (22). For the Lecture Only group recap-test, this student was not present. For the Lecture/PAT group, a different student registered the highest score for each of the repeated measures. The student in the Lecture/PAT group who scored highest for the posttest exhibited the following performance trend: pretest (14), posttest (26), recap-test (23).

Req. 3: A two factor mixed model ANOVA is a parametric analysis that relies on the assumption that the survey scores are normally distributed in the population being sampled. The distribution of the dependent variable can be estimated for each of the repeated measures. Normality can be substantiated if both the Skewness and Kurtosis statistical criteria fall within the range from -1.0 to 1.0. SPSS generated the descriptive statistics shown in Table 6.3 for the raw data (no missing data replacements) grouped by repeated measure.

The Skewness and Kurtosis distributions give statistics that are within the appropriate range for all three repeated measures. Therefore, the analysis does not violate the assumption of normality. Skewness measures the degree and direction of asym-

metry in the distribution, and Kurtosis is a measure of the "heaviness" of the tails of the distribution. The Valid N (listwise) value of 59 indicates the total number of student subject cases in which scores were recorded for all three tests. The mean and maximum statistic in Table 6.3 also exhibit a characteristic increase from pretest to posttest and decrease from posttest to recap-test. However, the recap-test mean and maximum values do not return to pretest levels.

For subsequent analysis, missing data are replaced by the median for the specific treatment in order to maintain orthogonality and provide a balanced data set for the within-subjects repeated factor. Table 6.4 shows the number of missing survey data for each treatment.

Req. 4: For a given repeated factor, the variance of survey scores for each of the independent intervention groups should be equal (homogeneity of variance). Essentially, all the errors (residuals) for a repeated measure must come from the same normal distribution. This is a basic requirement of a simple ANOVA but is also applied to the between-subjects factor in a mixed model ANOVA. Levene's test for homogeneity of variances is used to investigate this homoscedasticity property. SPSS generated the data shown in Table 6.5.

The significance column (p-value) in Table 6.5 reveals, for each repeated measure, that the probability associated with Levene's test for equality of error variances among the intervention groups is greater than the alpha for the diagnostic tests (0.01).

Req. 5: Sphericity means that the variances of the repeated measures are all equal, and the correlations among the repeated measures are all equal. This assumption is needed to allow for comparing the variances among the repeated measures. Violation of this assumption will require a Greenhouse-Geisser adjustment when evaluating interaction or main effects. Mauchly's test of sphericity is used to investigate data compliance to this property. SPSS generated the data shown in Table 6.6. The significance column (p-value) in Table 6.6 reveals that the probability associated

with Mauchly's test for sphericity (0.036) is slightly greater than the alpha for the diagnostic tests (0.01), so sphericity may be assumed.

Subsequent to addressing the above requirements for a two-factor mixed model ANOVA, the potential interaction effect can be analyzed. Specifically, is the evolution of survey scores from pretest to recap-test significantly different when comparing the Control, Lecture only, and Lecture w/PAT groups? To answer this question, SPSS generates the relevant data shown in Table 6.7.

Since sphericity can be assumed, the first row for the Test*group source provides the p-value of interest (0.062). Although the reported p-value is very close to the research hypothesis α of 0.05, there is not sufficient evidence from these survey data to reject the null hypothesis. When comparing the Control, Lecture only, and Lecture w/PAT groups, there is no statistically significant difference in the time evolution of the survey test scores.

Although a general interaction effect has not been established, Table 6.8 breaks down the pairwise estimated marginal means comparisons corrected using a Bonferroni adjustment for each test within an intervention group. For a given intervention group, the Bonferroni confidence interval adjustment holds the cumulative error rate of the multiple pairwise tests to the specified α=0.05. The figure shows that, for each intervention group, all differences between the repeated measures can be considered minimal *except for* 1) the pretest and posttest of the Control group, 2) the pretest and posttest of the Lecture w/PAT group, and 3) the pretest and recap-test of the Lecture w/PAT group.

Finally, Figure 6.3 shows the profile plot of the estimated marginal means for each experimental treatment. For the Test axis labels, '1' represents pretest, '2' represents posttest, and '3' represents recap-test. The significant pairwise differences listed in Table 6.8 can be seen in the graph, as well as the cross-over that occurs between the Lecture Only and Lecture w/PAT plotlines between the posttest and recap-test.

However, the patterns of change for the three groups were not sufficiently significant to register an interaction effect.

In the absence of a statistically significant interaction effect, an investigation of the main effects can be pursued. The research (null) hypotheses stated in Section 6.1 do not target the between-subjects factor main effect, primarily because comparing test subject survey scores based only on intervention group membership does not provide useful information about the time dependent impact of CS1 instruction on student comprehension of parallel concepts. "Flattening" the data to the intervention group main effect will merely indicate the relative performance of these groups independent of the test times, and no conjectures have been made in this study regarding between-group variations.

Since the primary focus of the study is to monitor student performance based on times in which the test instrument was administered during the CS1 course, the main effect of interest is the repeated measure. Are there significant differences in survey scores across the different testing periods independent of the type of classroom intervention? The Test source top section of Table 6.7 provides statistics on the within-subject main effect. Because sphericity can be assumed as mentioned earlier, the first row of the Test source section gives $p < 0.001$ for $F(2, 162) = 10.783$.

Consequently, there was a significant main effect among the repeated measures. Ignoring the intervention method, survey scores were significantly different depending on when the test instrument was administered. Figure 6.4 and the data in the associated Table 6.9 reveal the estimated marginal means for each of the testing factors used in this study.

To shed more light on the relative magnitudes among the three levels, Table 6.10 displays the pairwise comparisons for the repeated measure main effect. The table shows a statistically significant difference between the survey scores for 1) the pretest and posttest ($p < 0.001$) and 2) the pretest and recap-test ($p = 0.011$). No statistically

significant difference exists between the posttest and the recap-test.

6.3 Nonparametric Data Analysis

In addition to thirty multiple choice questions, the PoPS contained a design "essay" question that students answered by submitting a handwritten solution and diagram. The design question is at the end (**Part IX**) of the PoPS listed in Appendix A, and the processing strategy for solving the design question is given in Appendix B. As the research study methodology in Chapter 3 states, the design question assessment was performed by two independent graders using a six-point Likert scale. Each student submission was given a separate rating by the two graders based on the evaluation rubric in Figure 3.2. If the student submitted some written response or design sketch minimally related to the design question content and objectives, the grade awarded was an integer within the Likert scale. If no response was submitted, the student design question was given a zero. Graders were not made aware of student identities.

Since no assumptions can be made about the underlying probabilities of the design question scores, nonparametric tests are used for statistical analysis of these measures. Specifically, a Kruskal-Wallis test can be applied for comparisons between the experimental and control groups, and a Friedman matched group test can be applied to the repeated measures within each group.

To obtain a single design question score for each student response, a Mann-Whitney U Test was applied across the two separate sets of ratings from each grader to determine if they are statistically different. The Mann-Whitney U Test is the relevant nonparametric test because the two graders did not consult about the evaluation process not did they influence each other's student subject ratings. If no difference between the score sets was indicated, then a single score will be derived for each design question response by taking the average of the two scores from each grader.

The fundamental requirement for all nonparametric tests to be discussed in this section is that the dependent variable must be ordinally scaled at a minimum. By definition, the Likert scale used to assess the design question in this study is an ordinal measure. A total of 192 design question responses were evaluated by each of the two graders. Submissions in which there was no response were excluded from this analysis. Table 6.11 gives the results of the Mann-Whitney U Test comparing the score sets from the two graders. The p-value of 0.488 strongly suggests that there is no statistically significant difference between the Likert ratings assigned by the two graders. Thus, to perform subsequent between-group and within-group comparisons of the design question ratings, an individual student response will be assigned the average of each score given by the two graders.

Both the nonparametric and parametric analysis have the same number of missing scores, as shown in Table 6.4. Similar to the strategy used for the mixed ANOVA processing described earlier, the within-subject data sets for nonparametric tests were balanced by substituting the appropriate treatment median for missing scores. In addition, some student subjects did not submit an answer for the design question, either because of disinterest or lack of time. Whatever the underlying reason, the graders were instructed to assign a zero to an unanswered design question to indicate "nonresponse" (no data). For this nonparametric data analysis, nonresponse data was treated like missing data, with median substitution applied as described above. This approach permits a matched-groups analysis to be performed on the within-subject data.

Table 6.12 provides the number of unanswered design questions as well as the number of missing responses per treatment for the complete within-subject balanced data set. Table 6.13 gives the median and high score recorded for each treatment.

The median for all design question scores in this study was 2.0. For the Control group, three different students achieved the highest score across the repeated test

measures. The student who received the posttest high score was given the following design question ratings: pretest(2.0), posttest(5.0), and recap-test(3.5). For the Lecture Only group, the same student received the high score for the pretest (4.0) and posttest (4.0), but was not present to take the recap-test. Similarly, for the Lecture/PAT group the student who received the high score for the pretest (4.0) and posttest (5.0) was not present to take the recap-test.

For nonparametric hypothesis testing, the median serves as the best indicator of central tendency in a skewed dataset as long as there is no more than one distinct identifiable mode. Figure 6.5 provides the design question score frequencies of the raw data (missing and nonresponse substitutions excluded) grouped by the repeated test measure. The histogram shows that the scores exhibit a unimodal distribution skewed toward the low end of the Likert range used for this study. The mode (2.0) is equivalent to the overall median. The figure also shows that all of the maximum scores (5.0) were achieved either during the posttest(2 cases) or recap-test(1 case).

A different grouping across the raw data design question score frequencies is shown in Figure 6.6. In this chart, scores are grouped by intervention. A quick visual inspection of Figure 6.6 reveals that the shape of the score distributions for each intervention group appear to be roughly equivalent.

A nonparametric analysis was performed on design question data acquired from each treatment of the 3x3 mixed (repeated measure) research design described in Section 3.3. Substitutions for missing and nonresponse data were implemented as described above. The Kruskal-Wallis test was used for between-subjects analysis and the Friedman test for three or more matched groups was used for within-subject analysis. Unlike the two-factor mixed ANOVA test, no interaction effect can be ascertained from nonparametric tests.

Therefore, the analysis involves 1) three separate Kruskal-Wallis tests, one for each row of the design shown in Figure 3.1; 2) three separate Friedman tests, one

for each column of the design shown in Figure 3.1; and 3) a single Friedman test to analyze the repeated measure main effect. As with the parametric tests, an analysis of the intervention group main effect will provide no useful information relevant to the research hypotheses of this study.

The results obtained by performing a Kruskal-Wallis test across intervention groups for each of the three repeated measures is shown in Table 6.14. The table reveals that statistically significant differences exist (at $\alpha = 0.05$) among the intervention groups at the pretest (p-value = 0.026) and recap-test (p-value = 0.035) levels. The p-value for the posttest group (0.065) is close to the α significance value, but there is not sufficient evidence to reject the null hypothesis (no significant differences in ranks) at the posttest level.

Further analysis was performed on the survey scores for the pretest and recap-test levels to determine which of the three intervention groups exhibited significant differences in mean rank. For the pretest and recap-test levels, separate Mann-Whitney U tests were applied to each pair combination of the intervention groups, resulting in a total of three Mann-Whitney U tests for the repeated measure group under investigation. The output from these pairwise tests confirmed the relative differences implied by the Mean Rank column generated by the Kruskal-Wallis test.

Table 6.15 provides a summary of the comparative analysis of the between-subjects survey scores. For the recap-test, the p-value for the Mann-Whitney U test performed between the Lecture Only and Lecture w/PAT groups was 0.070 with the higher mean rank belonging to the Lecture Only group.

The results obtained by performing a within-subject matched group Friedman test across the repeated measures for each of the three intervention groups is shown in Table 6.16. The table reveals that statistically significant differences exist (at $\alpha = 0.05$) among the repeated measures for all of the intervention groups.

Further analysis was performed on the survey scores for each intervention group

to determine the pattern of performance change over time. For each intervention level, separate Wilcoxon tests were applied to each pair combination of the repeated measures, resulting in a total of three Wilcoxon pairwise tests for the intervention group of interest. Table 6.17 shows the results.

Significant differences are detected for the Control group pretest/posttest pair, the Control group pretest/recap-test pair, and the Lecture Only group pretest/posttest pair. For each of these pairs, the pretest logged a significantly greater number of lower scores over all pairwise comparisons.

Finally, a Friedman test including all repeated measures from the three intervention groups was performed. This test investigated the pattern of performance change over time independent of the type of intervention. The Friedman test results and the associated pairwise Wilcoxon results are given in Table 6.18.

The Friedman test indicates a significant difference ($p < 0.001$) in the repeated measures if all experimental matched data are considered. The test confirms a main effect supported by the Wilcoxon results, in which the posttest design question scores exceed the pretest design question scores by a statistically significant margin ($p = 0.001$). Also, the 0.067 p-value for the recap-test/pretest score comparisons is close to the significant α value of 0.05. Table 6.19 provides more details generated by this Wilcoxon pairwise analysis and confirms the directionality of the results for each of the paired score data sets.

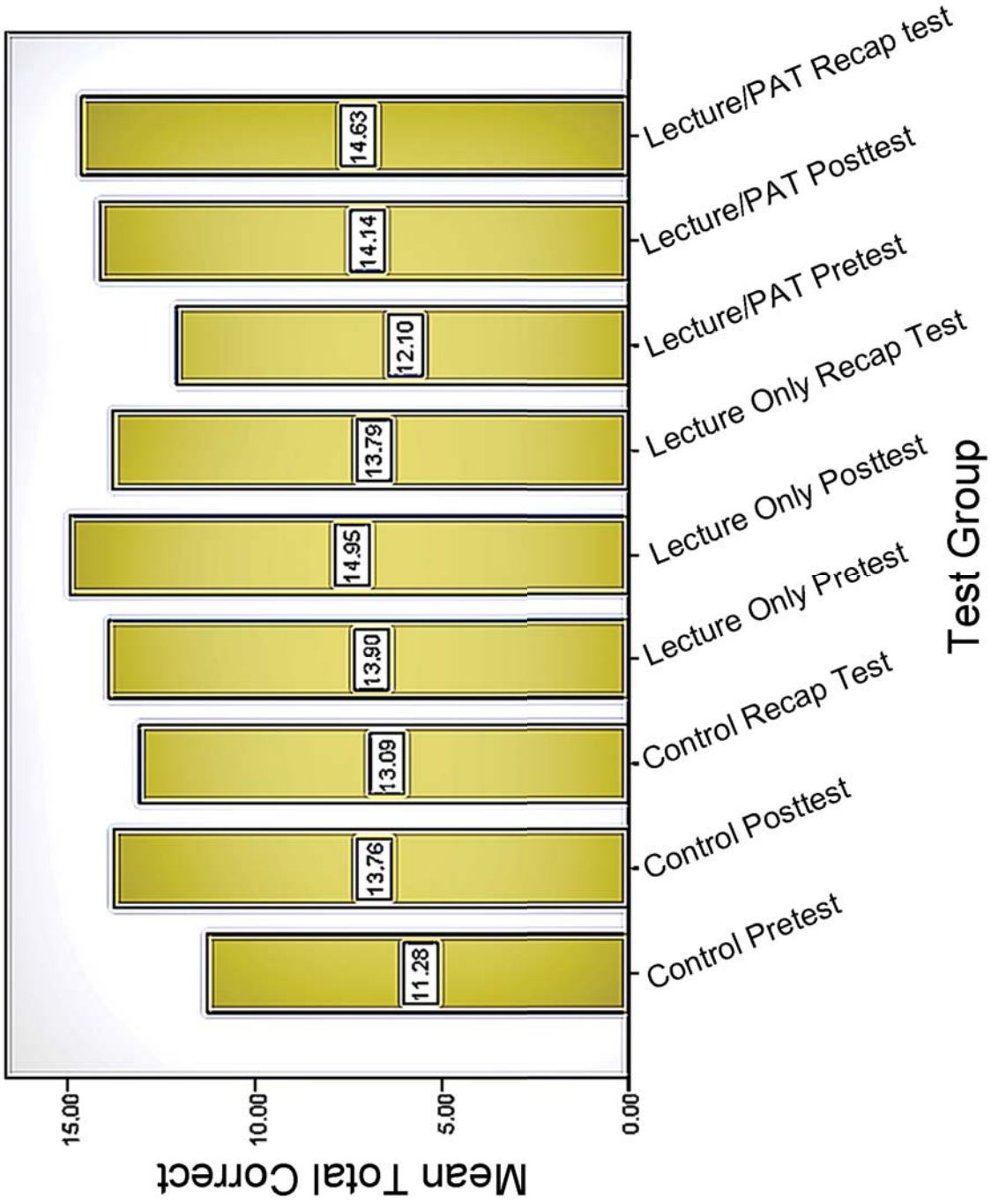


Figure 6.1: Mean Total Correct

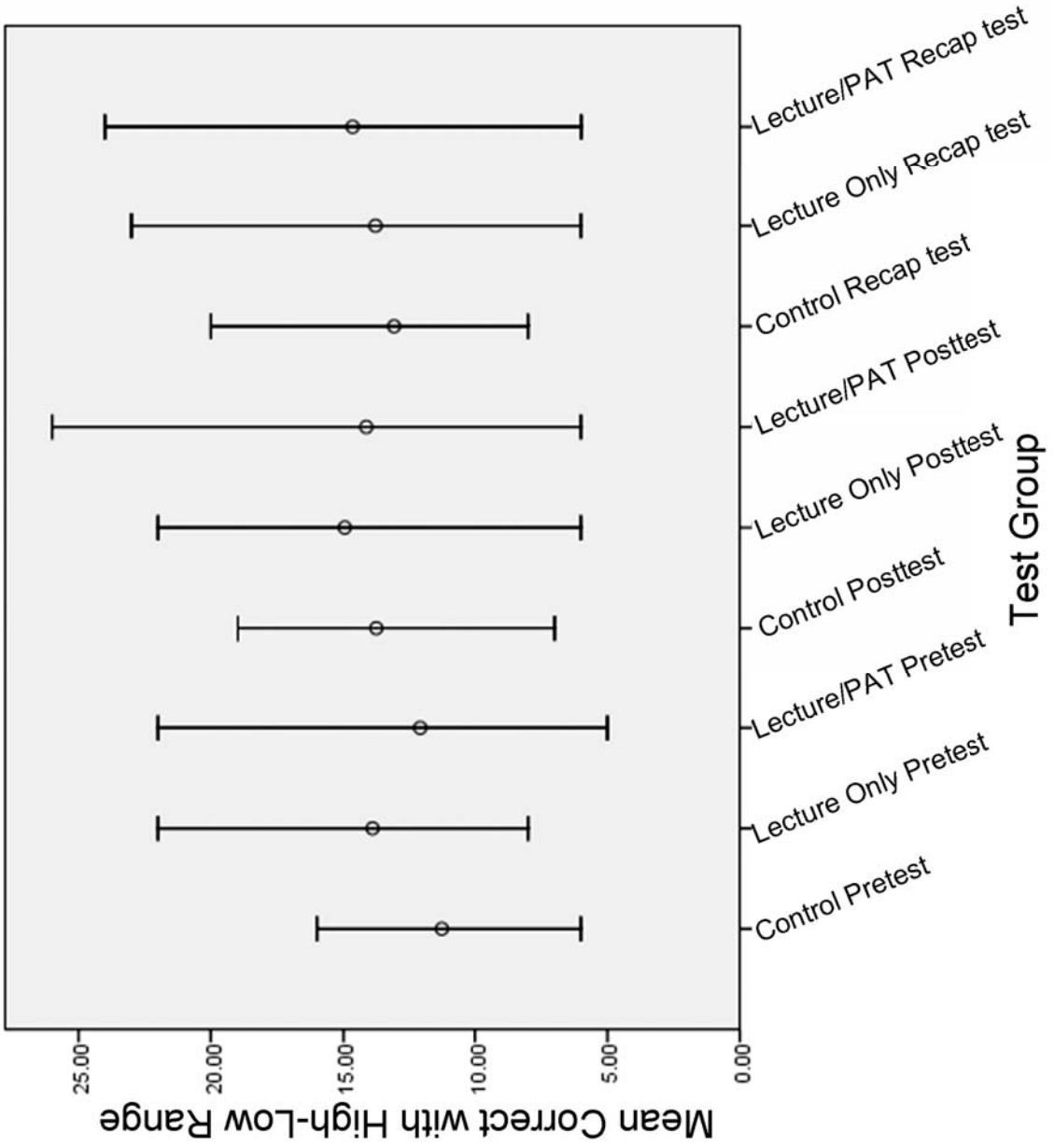


Figure 6.2: Mean Total Correct with Range

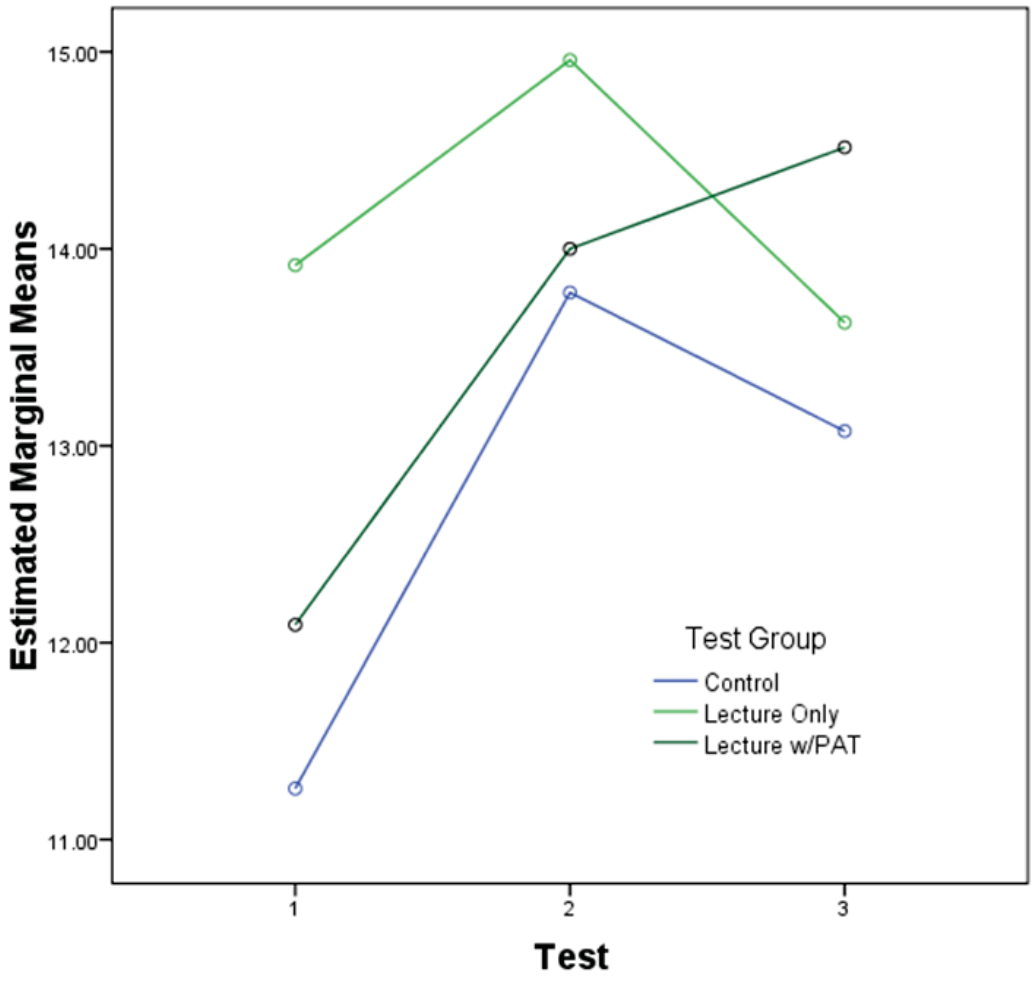


Figure 6.3: Profile Plot (Test vs Experimental Group)

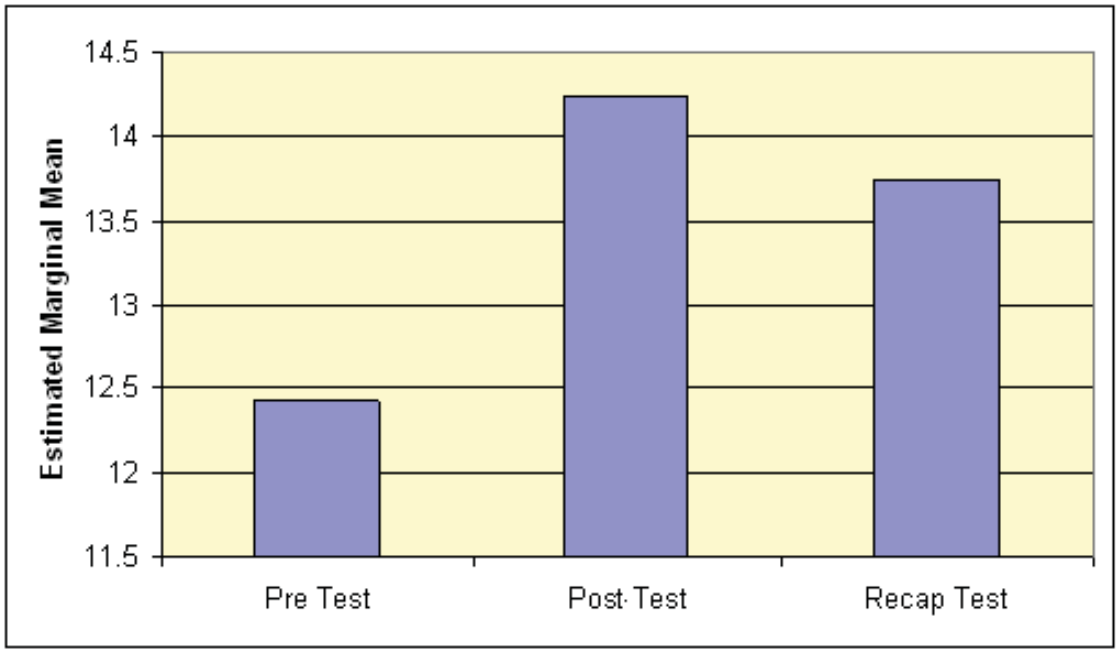


Figure 6.4: Repeated Measure Means

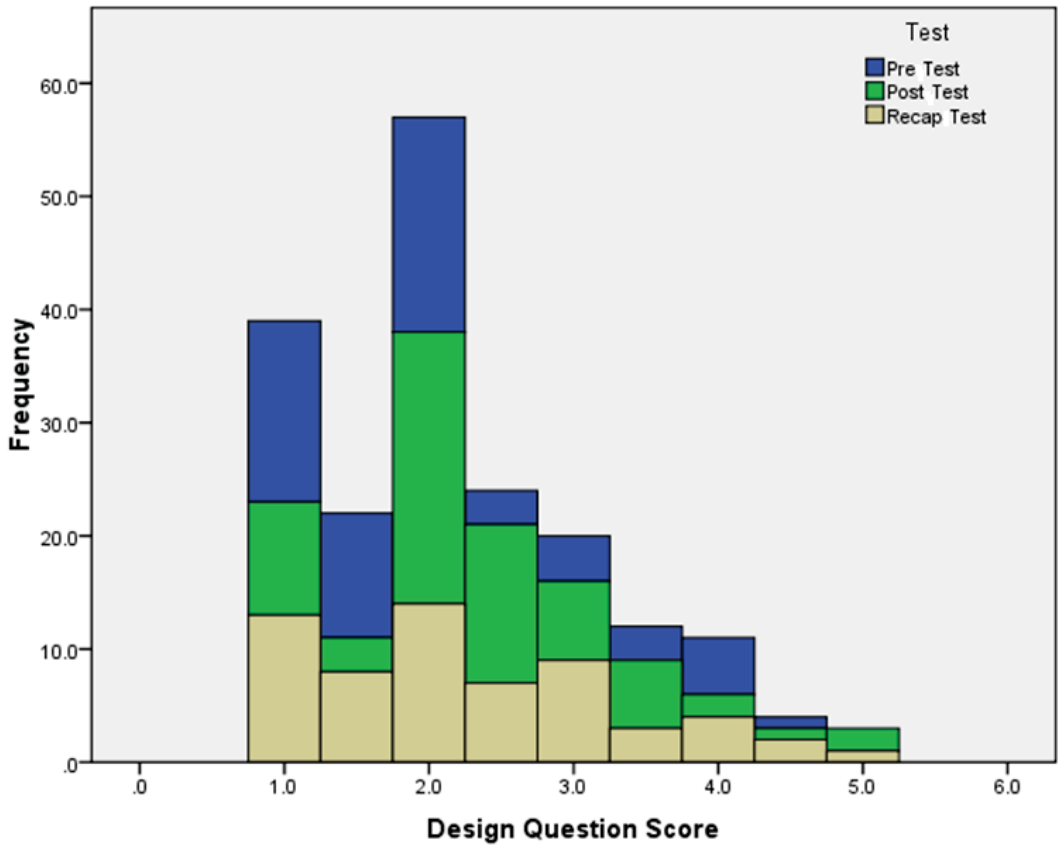


Figure 6.5: Histogram of Design Question Scores by Test

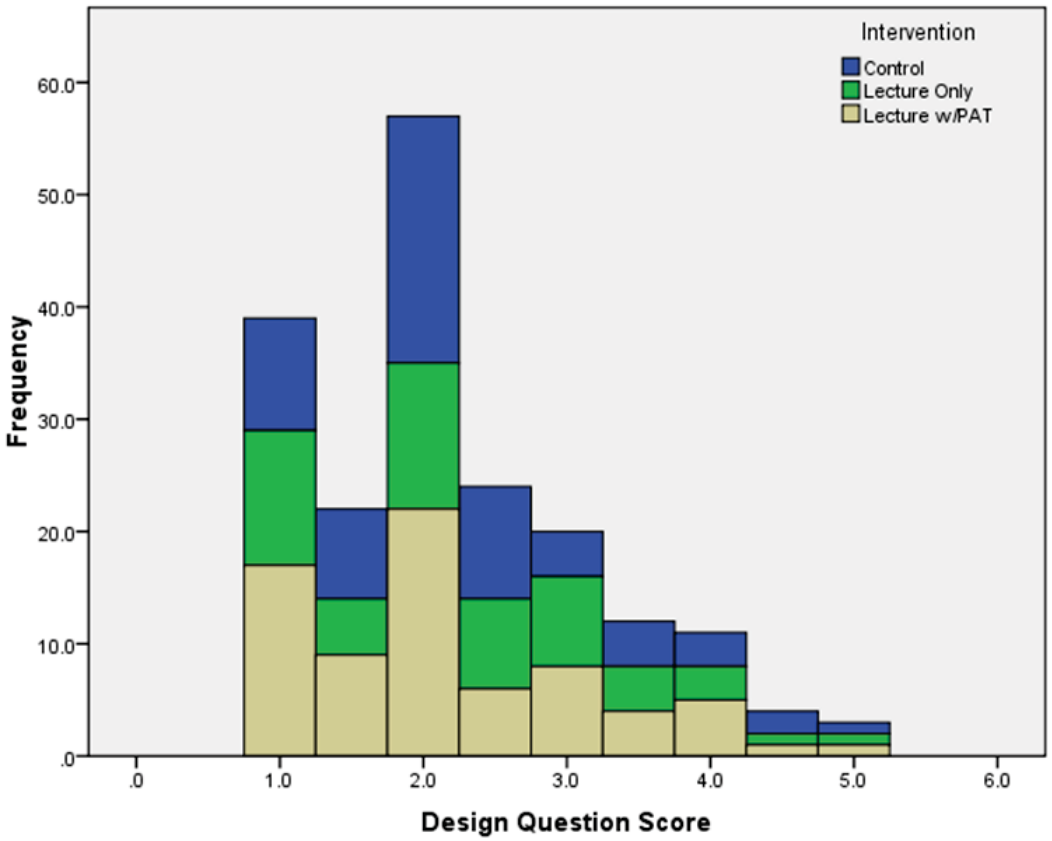


Figure 6.6: Histogram of Design Question Scores by Intervention

Table 6.1: Experimental Study Survey Cases

	Intervention Measure		
	Control	**Lecture Only**	**Lecture/PAT**
Pre Test	25	20	31
Post Test	25	21	29
Recap Test	23	19	27

Table 6.2: Descriptive Statistics for each Experimental Treatment (Total Correct)

Test Group	N	Mean	Median	Std. Deviation	Maximum	Minimum
Control: Pre Test	25	11.2800	11.0000	2.99333	16.00	6.00
Control: Post Test	25	13.7600	14.0000	3.34515	19.00	7.00
Control: Retention Test	23	13.0870	13.0000	3.50212	20.00	8.00
Lecture Only: Pre Test	20	13.9000	14.0000	4.47096	22.00	8.00
Lecture Only: Post Test	21	14.9524	15.0000	4.82158	22.00	6.00
Lecture Only: Retention Test	19	13.7895	13.0000	4.61373	23.00	6.00
Lecture/PAT: Pre Test	31	12.0968	12.0000	4.06903	22.00	5.00
Lecture/PAT: Post Test	29	14.1379	13.0000	5.64247	26.00	6.00
Lecture/PAT: Retention Test	27	14.6296	14.0000	5.20464	24.00	6.00
Total	220	13.4591	13.0000	4.46121	26.00	5.00

Table 6.3: Descriptive Statistics for Repeated Measures

	N	Minimum	Maximum	Mean	Std. Deviation	Skewness		Kurtosis	
	Statistic	Statistic	Statistic	Statistic	Statistic	Statistic	Std. Error	Statistic	Std. Error
Pre Test Score	76	5.00	22.00	12.3026	3.94975	.538	.276	.005	.545
Post Test Score	75	6.00	26.00	14.2400	4.71008	.138	.277	-.556	.548
Recap Test Score	69	6.00	24.00	13.8841	4.51643	.419	.289	-.307	.570
Valid N (listwise)	59								

Table 6.4: Number of Missing Surveys

	Intervention Measure		
	Control	**Lecture Only**	**Lecture/PAT**
Pre Test	2 of 27	4 of 24	2 of 33
Post Test	2 of 27	3 of 24	4 of 33
Recap Test	4 of 27	5 of 24	6 of 33

Table 6.5: Levene's Test of Equality

	F	df1	df2	Sig.
Pre Test Score	.690	2	81	.504
Post Test Score	3.171	2	81	.047
Recap Test Score	1.503	2	81	.229

Tests the null hypothesis that the error variance of the dependent variable is equal across groups.

a. Design: Intercept + group
Within Subjects Design: Test

Table 6.6: Mauchly's Test of Sphericity

Within Subjects Effect	Mauchly's W	Approx. Chi-Square	df	Sig.	Epsilon[a]		
					Greenhouse-Geisser	Huynh-Feldt	Lower-bound
Test	.920	6.629	2	.036	.926	.971	.500

Tests the null hypothesis that the error covariance matrix of the orthonormalized transformed dependent variables is proportional to an identity matrix.

a. May be used to adjust the degrees of freedom for the averaged tests of significance. Corrected tests are displayed in the Tests of Within-Subjects Effects table.

b. Design: Intercept + group
Within Subjects Design: Test

Table 6.7: Two Factor Mixed Model ANOVA

Source		Type III Sum of Squares	df	Mean Square	F	Sig.	Partial Eta Squared	Noncent. Parameter	Observed Power[a]
Test	Sphericity Assumed	146.202	2	73.101	10.783	.000	.117	21.566	.989
	Greenhouse-Geisser	146.202	1.853	78.914	10.783	.000	.117	19.977	.985
	Huynh-Feldt	146.202	1.941	75.323	10.783	.000	.117	20.930	.988
	Lower-bound	146.202	1.000	146.202	10.783	.002	.117	10.783	.901
Test * group	Sphericity Assumed	62.203	4	15.551	2.294	.062	.054	9.175	.659
	Greenhouse-Geisser	62.203	3.705	16.787	2.294	.067	.054	8.500	.633
	Huynh-Feldt	62.203	3.882	16.023	2.294	.064	.054	8.905	.649
	Lower-bound	62.203	2.000	31.101	2.294	.107	.054	4.588	.453
Error(Test)	Sphericity Assumed	1098.242	162	6.779					
	Greenhouse-Geisser	1098.242	150.067	7.318					
	Huynh-Feldt	1098.242	157.221	6.985					
	Lower-bound	1098.242	81.000	13.559					

a. Computed using alpha = .05

Table 6.8: Pairwise Test Comparisons per Intervention Group

Test Group	(I) Test	(J) Test	Mean Difference (I-J)	Std. Error	Sig.[a]	95% Confidence Interval for Difference[a]	
						Lower Bound	Upper Bound
Control	1	2	-2.519*	.742	.003	-4.333	-.704
		3	-1.815	.770	.063	-3.698	.068
	2	1	2.519*	.742	.003	.704	4.333
		3	.704	.602	.738	-.769	2.176
	3	1	1.815	.770	.063	-.068	3.698
		2	-.704	.602	.738	-2.176	.769
Lecture Only	1	2	-1.042	.787	.568	-2.966	.883
		3	.292	.817	1.000	-1.705	2.289
	2	1	1.042	.787	.568	-.883	2.966
		3	1.333	.639	.120	-.228	2.895
	3	1	-.292	.817	1.000	-2.289	1.705
		2	-1.333	.639	.120	-2.895	.228
Lecture w/PAT	1	2	-1.909*	.671	.017	-3.550	-.268
		3	-2.424*	.697	.002	-4.127	-.721
	2	1	1.909*	.671	.017	.268	3.550
		3	-.515	.545	1.000	-1.847	.817
	3	1	2.424*	.697	.002	.721	4.127
		2	.515	.545	1.000	-.817	1.847

Based on estimated marginal means

*. The mean difference is significant at the .05 level.

a. Adjustment for multiple comparisons: Bonferroni.

Table 6.9: Repeated Measure Means

Test	Mean	Std. Error	95% Confidence Interval	
			Lower Bound	Upper Bound
1	12.422	.404	11.618	13.226
2	14.245	.494	13.263	15.228
3	13.738	.451	12.840	14.636

Table 6.10: Main Effect Pairwise Test Comparisons

(I) Test	(J) Test	Mean Difference (I-J)	Std. Error	Sig.[a]	95% Confidence Interval for Difference[a]	
					Lower Bound	Upper Bound
1	2	-1.823*	.424	.000	-2.861	-.786
	3	-1.316*	.440	.011	-2.392	-.239
2	1	1.823*	.424	.000	.786	2.861
	3	.507	.344	.434	-.335	1.349
3	1	1.316*	.440	.011	.239	2.392
	2	-.507	.344	.434	-1.349	.335

Based on estimated marginal means

*. The mean difference is significant at the .05 level.

a. Adjustment for multiple comparisons: Bonferroni.

Table 6.11: Mann-Whitney U Test between Evaluator Scores

	Evaluator Group	N	Mean Rank	Sum of Ranks
Design Question Evaluation	Evaluator A	192	196.23	37677.00
	Evaluator B	192	188.77	36243.00
	Total	384		

	Design Question Evaluation
Mann-Whitney U	17715.000
Wilcoxon W	36243.000
Z	-.693
Asymp. Sig. (2-tailed)	.488

a. Grouping Variable: Evaluator Group

Table 6.12: Missing and Unanswered Design Questions

	Intervention Measure								
	Control			**Lecture Only**			**Lecture/PAT**		
	miss	unansw	total	miss	unansw	total	miss	unansw	total
Pre Test	2	5	27	4	1	24	2	8	33
Post Test	2	2	27	3	0	24	4	4	33
Recap Test	4	2	27	5	4	24	6	2	33

Table 6.13: Design Question Median and High Scores

	Intervention Measure					
	Control		**Lecture Only**		**Lecture/PAT**	
	Median	High	Median	High	Median	High
Pre Test	1.5	4.5	1.5	4.0	2.0	4.0
Post Test	2.0	5.0	2.5	4.0	2.0	5.0
Recap Test	2.0	4.5	2.0	5.0	1.5	4.0

Table 6.14: Kruskal-Wallis Test Results per Repeated Measure

Ranks

	Test Group	N	Mean Rank
Pre-Test Score	Control	27	36.93
	Lecture Only	24	36.90
	Lecture w/PAT	33	51.14
	Total	84	

Test Statistics[a,b]

Pre-Test Group

	Pre-Test Score
Chi-Square	7.317
df	2
Asymp. Sig.	.026

a. Kruskal Wallis Test

b. Grouping Variable: Test Group

Ranks

	Test Group	N	Mean Rank
Post-Test Score	Control	27	41.33
	Lecture Only	24	51.44
	Lecture w/PAT	33	36.95
	Total	84	

Test Statistics[a,b]

Post-Test Group

	Post-Test Score
Chi-Square	5.480
df	2
Asymp. Sig.	.065

a. Kruskal Wallis Test

b. Grouping Variable: Test Group

Ranks

	Test Group	N	Mean Rank
Recap-Test Score	Control	27	49.28
	Lecture Only	24	46.13
	Lecture w/PAT	33	34.32
	Total	84	

Test Statistics[a,b]

Recap-Test Group

	Recap-Test Score
Chi-Square	6.686
df	2
Asymp. Sig.	.035

a. Kruskal Wallis Test

b. Grouping Variable: Test Group

Table 6.15: Relative Differences in Mean Rank

	Intervention Measure					
	Control		**Lecture Only**		**Lecture/PAT**	
Compared To	Lect	Lect/PAT	Contr	Lect/PAT	Contr	Lect
Pre Test	None	Lower	None	Lower	Higher	Higher
Post Test	None	None	None	None	None	None
Recap Test	None	Higher	None	None	Lower	None

Table 6.16: Friedman Test Results per Intervention Group

Ranks

	Mean Rank
Pre Test Score	1.46
Post Test Score	2.30
Recap Test Score	2.24

Test Statistics[a]

N	27
Chi-Square	13.326
df	2
Asymp. Sig.	.001

a. Friedman Test

Control Group

Ranks

	Mean Rank
Pre Test Score	1.56
Post Test Score	2.38
Recap Test Score	2.06

Test Statistics[a]

N	24
Chi-Square	9.214
df	2
Asymp. Sig.	.010

a. Friedman Test

Lecture Only Group

Ranks

	Mean Rank
Pre Test Score	2.08
Post Test Score	2.26
Recap Test Score	1.67

Test Statistics[a]

N	33
Chi-Square	8.489
df	2
Asymp. Sig.	.014

a. Friedman Test

Lecture w/PAT Group

Table 6.17: Wilcoxon Pairwise Test Results per Intervention Group

	Post Test Score - Pre Test Score	Recap Test Score - Pre Test Score	Recap Test Score - Post Test Score
Z	-2.572[a]	-2.598[a]	-.366[a]
Asymp. Sig. (2-tailed)	.010	.009	.714

a. Based on negative ranks.

b. Wilcoxon Signed Ranks Test

Control Group

	Post Test Score - Pre Test Score	Recap Test Score - Pre Test Score	Recap Test Score - Post Test Score
Z	-2.907[a]	-1.461[a]	-1.357[b]
Asymp. Sig. (2-tailed)	.004	.144	.175

a. Based on negative ranks.

b. Based on positive ranks.

c. Wilcoxon Signed Ranks Test

Lecture Only Group

	Post Test Score - Pre Test Score	Recap Test Score - Pre Test Score	Recap Test Score - Post Test Score
Z	.000[a]	-1.503[b]	-1.516[b]
Asymp. Sig. (2-tailed)	1.000	.133	.130

a. The sum of negative ranks equals the sum of positive ranks.

b. Based on positive ranks.

c. Wilcoxon Signed Ranks Test

Lecture w/PAT Group

Table 6.18: Repeated Measure Main Effect Design Question Analysis

Ranks

	Mean Rank
Pre Test Score	1.73
Post Test Score	2.30
Recap Test Score	1.96

Test Statistics[a]

N	84
Chi-Square	17.077
df	2
Asymp. Sig.	.000

a. Friedman Test

Test Statistics[c]

	Post Test Score - Pre Test Score	Recap Test Score - Pre Test Score	Recap Test Score - Post Test Score
Z	-3.357[a]	-1.832[a]	-1.270[b]
Asymp. Sig. (2-tailed)	.001	.067	.204

a. Based on negative ranks.

b. Based on positive ranks.

c. Wilcoxon Signed Ranks Test

Table 6.19: Wilcoxon Repeated Measure Main Effect Details

		N	Mean Rank	Sum of Ranks
Post Test Score - Pre Test Score	Negative Ranks	16[a]	31.59	505.50
	Positive Ranks	46[b]	31.47	1447.50
	Ties	22[c]		
	Total	84		
Recap Test Score - Pre Test Score	Negative Ranks	27[d]	33.57	906.50
	Positive Ranks	42[e]	35.92	1508.50
	Ties	15[f]		
	Total	84		
Recap Test Score - Post Test Score	Negative Ranks	42[g]	28.37	1191.50
	Positive Ranks	21[h]	39.26	824.50
	Ties	21[i]		
	Total	84		

a. Post Test Score < Pre Test Score

b. Post Test Score > Pre Test Score

c. Post Test Score = Pre Test Score

d. Recap Test Score < Pre Test Score

e. Recap Test Score > Pre Test Score

f. Recap Test Score = Pre Test Score

g. Recap Test Score < Post Test Score

h. Recap Test Score > Post Test Score

i. Recap Test Score = Post Test Score

CHAPTER 7

CONCLUSIONS

A student participant in this study walked up to me after class and mentioned that he had been speaking with a friend who had a CS degree about the inclusion of parallel concepts in our CS1 course. His friend responded, "Why are they teaching parallel programming in an introductory CS class?" The perception that parallel computing most naturally belongs in the later stages of the CS curriculum is generally accepted by many within the discipline.

The research reported here represents an initial empirical step toward answering the "why" question put forth above by directly addressing the "how" or methodology question. Specifically, which pedagogical strategies will most efficiently introduce beginning CS students to parallel thinking such that these students will realize a significant, long-term benefit throughout their academic and professional careers? The evaluation instrument, visualization software tool, and data analysis presented in this dissertation provide some of the requisite groundwork to inform the discussion regarding the proper instruction and placement of parallel concepts in the CS program.

The Perceptions of Parallelism Survey (PoPS) was developed specifically for this study by the author and represents a first version of an evaluation instrument which provides the foundation for the development of a full-fledged parallel concept inventory. Assessment instrument development for any academic topic is a lengthy, exacting process as set down in the standard educational test development guidelines [3]. To account for the CS-specific issue of programming language independence, Tew and

Guzdial [69] injected an additional step into the traditional test instrument development process:

1. Define Conceptual Content
2. Expert Review of Test Specification
3. Build Test Bank of Questions
4. Verify Language Independence
5. Pilot Questions
6. Establish Validity
7. Establish Reliability

The first step in assessment development may be driven by input from content experts, topical outlines from widely adopted texts on the subject matter of interest, or student misconceptions as was the case for the Force Concept Inventory used in Physics [33]. It is generally recognized that a validated and reliable concept inventory within the computer sciences is several years away, though efforts toward developing a CS1 Computing Fundamentals and Digital Logic assessment are just beginning [32][36][69]. What students should really know about these topics remains an open research question.

The purpose and definition of the PoPS utilized in this study (step 1 above) was to provide a measure of a student's grasp of concepts related to parallel design. As described in Chapter 4, the main topic categories targeted by the PoPS was compiled from the author's exposure to parallel programming projects, his experience in teaching the topic, and his reading and analysis of various textbooks about parallel computation. To a limited extent, the PoPS accomplishes steps 3 and 4 above, since an experimental test bank of questions was developed with no ties to a specific

programming language. Each of the PoPS questions was framed in a real-world context to provide some degree of familiarity for the CS1 student; no prior experience in programming was required.

Extensive validity/reliability requirements were not demonstrated, nor was a comprehensive review of test specification and pilot questions from a diverse panel of subject matter experts (steps 2, 5, 6 and 7) fully undertaken in the development of the PoPS. Consequently, no claim is being made that the PoPS used in this study represents a rigorously developed content inventory in the traditional sense. Validity of the test questions can be enhanced using a "think-aloud" approach, in which students are asked to verbally express what they are thinking about when solving the assessment problems. This strategy would help to clarify student misconceptions about parallel thinking and would help identify and refine poorly constructed questions in which the student knew the core concept but misunderstood the question or the answer choices. Similarly, reliability can be established using split-half testing, in which the test instrument is divided into two equivalent tests and administered to the same students at one to two week intervals. Correlations performed between the two separate test scores provides a consistency measure for the test. Finally, whereas most formal content inventories are designed for a test time of 30 minutes, the PoPS in this study was administered over a one hour period.

Notwithstanding the disparities described above, this early version of the PoPS proved an effective assessment tool for the modest scope of this investigation. The PoPS avoided both a ceiling and floor effect, in which important information about student comprehension is lost because the assessment is either too easy or too hard. The scores generated by the PoPS followed a normal distribution as discussed in Chapter 6 and confirmed by Table 6.3, thereby supporting the subsequent parametric data analysis. In addition, except for the design essay question, the multiple choice format of the PoPS eliminated any potential grading bias, allowed for more efficient

administration/scoring, and has been shown, when properly constructed, to provide the same information about conceptual knowledge as short answer or open response questions [28].

The Parallel Analysis Tool (PAT) described in Chapter 5 offered a unique variation on the traditional development environment used in introductory programming classrooms. The primary intention of this software tool was to cultivate and hone the student's ability to find concurrency directly within Java source code. Student users could also experiment with parallel performance in real time by directly utilizing multicore resources in the underlying platform. Students were able to specify the number of processors to be recruited for a parallel task, and were able to rapidly assess the quality of their concurrency strategies through a speedup metric called the PQ value. By transforming specific code sections from sequential to parallel and examining the influence on overall program performance, students were able to experience firsthand a shared-memory model process called *incremental parallelization*, by which a sequential program is converted into a viable parallel program one block of code at a time.

The PAT provided translation of traditional Java code to Parallel Java code [37] through straightforward comment annotations, thereby insulating students from the complexities of parallel programming primitives, threads, and other low-level implementation details that might distract the student from fully exploring and experiencing the *Finding Concurrency* parallel programming design pattern [47]. In addition, the PAT generates a corrresponding UML Activity Diagram of the main program steps, offering the student a visualization of the source code design including appropriate forks and joins to indicate parallel sections.

Some typical pedagogical software tools that supplement instruction of parallel concepts were described in detail in Chapter 2. Most of these visualization environments are geared toward advanced level courses in multithreading, though there are

a few instances in which thread-centric tools have been designed for CS2 (data structures) courses [8]. Much more rare are investigations into teaching concurrency at the CS1 level. Though these classroom strategies may involve libraries that simplify the development of graphics and animations, the underlying objective is to expose students to the implementation of threads [12].

In contrast, the PAT draws the student's focus away from threads toward a more high-level conceptual view of concurrency. As suggested by Edward Lee, despite the de facto inclusion of thread primitives in many popular programming languages, and their seemingly broad acceptance by working programmers, threads may *not* be the best concurrency mechanism for designing and developing parallel applications [45]. In fact, Lee proposes a more visually oriented form of coding parallel programs closely allied with the representations proffered by the UML.

The PAT was designed with this potential future direction of parallel programming in mind. CS1 students do not necessarily need to know how concurrency is implemented, but they most certainly will need to build an awareness of "why parallelism?" and "where parallelism?" The scope of the "where" question is broad, embracing both the architecture of the computing device and the proper location of parallelism in code. Understanding these aspects of software design is an integral part of becoming a successful parallel programmer.

The PAT as currently constructed can be expanded, both in usage and functionality. Students in this particular study were exposed to the tool at the start of the CS1 course for a relatively short three-week period. Exercises requested that students work with pre-existing code rather than develop their own. The opportunity for students to experiment with the PAT was limited, which could easily be remedied by extending the module on parallel programming by one to two weeks, or using the PAT toward the end of the course when student programming proficiency has significantly improved.

Regarding specific program structure analysis, at present the PAT targets two types of parallel code constructs: for-loop parallelism, and explicit task (functional) parallelism. These two programming structures encompass a wide variety of potential entry points in which parallelism may be introduced into code. The PAT can be enhanced to also include shared task lists or other resources, and to allow for the identification of critical sections. The migration of the PAT to message passing systems has yet to be explored.

The PAT was introduced into this research to provide another distinct level of classroom intervention, apart from the traditional lecture/homework strategy employed in most educational experimental studies. Software tools are an integral and expected part of the CS student experience, and thus the PAT fit naturally as a candidate for instructional enhancement in this study. However, no a priori assumptions were made in this work about the effectiveness and impact of the PAT in communicating key parallel concepts to CS1 students. The results of the experimental educational study were designed to offer some hard evidence regarding the future development and use of the PAT.

7.1 Statistical Inferences

The parametric results reported in Section 6.2 support the following conclusions about the research and associated null hypotheses for this study:

Concl. 1: With significance level $\alpha = 0.05$, there is insufficient evidence (p-value = 0.062) to reject the null hypothesis that CS1 students exposed to a three-week "lecture-only" course module on parallel design concepts will exhibit *no* statistically significant comprehension levels about this subject matter *after* the delivery of the course module when compared to students exposed to traditional CS1-level topics during the same time period.

Concl. 2: With significance level $\alpha = 0.05$, there is insufficient evidence (p-value = 0.062)

to reject the null hypothesis that CS1 students exposed to a three-week "lecture with software visual tool" course module on parallel design concepts will exhibit *no* statistically significant comprehension levels about this subject matter *after* the delivery of the course module when compared to students exposed to traditional CS1-level topics during the same time period.

Concl. 3: If there is no detectable interaction between the experimental factors, then with significance level $\alpha = 0.05$, there is sufficient evidence ($p < 0.001$ and $p=0.011$) to reject the null hypothesis that CS1 students will exhibit *no* statistically significant comprehension levels about this subject matter *after* the delivery of any CS1 three-week course module when compared to comprehension levels just prior to the three-week course module.

The first two conclusions address the interaction effect between the intervention strategy and repeated test measures. The third conclusion resolves the main effect exhibited by the repeated test measures. The nonparametric results discussed in Section 6.3 support Conclusions 1 and 3, with some qualifications to be clarified shortly regarding Conclusion 2.

Although the parametric analysis revealed no statistically significant interaction effect between the two factors, the relatively low p-value (0.062) suggests some qualitative differences may be observed, which is borne out by the pairwise comparisons in Table 6.8 and the estimated marginal means profile plot in Figure 6.3. These figures offer some insights into the evolution of survey scores for each intevention group.

Between the pretest and posttest, Figure 6.3 displays a distinct increase or "bump" for each intervention group. In fact, the pairwise comparisons register statistically significant pretest/posttest changes in survey scores for both the Control group ($p=0.003$) and the Lecture w/PAT group ($p=0.017$). Between the posttest and recap-test, only the scores for the Lecture w/PAT maintain an upward trend, verified by the associated statistically significant pretest/recap-test pairwise comparison ($p=0.002$).

A true absence of interaction would typically imply that the Lecture w/PAT scores would decrease modestly between posttest and recap-test similar to the other two interventions. However, students in the Lecture w/PAT group at least qualitatively demonstrate a slightly improved performance in the recap-test when compared to the posttest. The magnitude of the Lecture w/PAT mean increase (0.515) is less than the magnitudes of the mean decreases of the other two interventions (0.704 and 1.333), but the difference in directionality is noteworthy. The almost two point posttest/recap-test swing between the Lecture and Lecture w/PAT groups is made evident by the crossing plotlines in Figure 6.3. A visual nonstatistical analysis of this figure would seem to suggest that students in the Lecture w/PAT group appeared to retain information about parallel concepts at the nine-week mark slightly better than the student subjects in the other two groups. Only by duplicating this study would this assertion be verified.

The main within-subject effect was much more pronounced and was confirmed statistically (see Conclusion 3 above). The repeated measure means independent of intervention shown in Figure 6.4 and the pairwise comparisons in Table 6.10 both support the existence of a pretest/posttest "bump" in all scores described earlier. For the main effect, there is a statistically significant difference between pretest/posttest ($p < 0.001$) and pretest/recap-test ($p = 0.011$). Assuming the PoPS evaluation instrument accurately measures student comprehension of parallel design concepts, these results indicate that *any* mode of CS1 instruction improves a student's powers of parallel thinking, both immediately after the module has been delivered and six weeks beyond that.

Given that the interventions in this study were introduced during the initial three weeks of the CS1 course, it is possible that all three content deliveries and instructional strategies for this first module benefited somewhat by a general "bump" in student focus and attitude that may naturally occur between day 1 and day 6 of

any given class. In addition, since the same assessment is administered for all three repeated measures, students may acquire a familiarity with survey questions and format that could lead to improved test-taking efficiency and performance. However, in the absence of any studies correlating a general improvement in student performance with course week, Conclusion 3 above regarding CS1 specific instruction must remain as stated. Further studies that introduce the parallel module at different times during the semester would help to determine if this influence were real and measurable.

In Table 7.1, the 30 multiple choice question results were grouped by PoPS task to determine if students noticeably improved in any specific areas of parallel design. Recall that the targeted skills and concept coverage for each PoPS task are listed in Table 1.1. First note that from posttest to recap-test, the Lecture/PAT group student subject performance improved in six (Tasks I-VI) of eight areas, whereas the Control group (Tasks III, VI, VIII) and Lecture Only group (Tasks I, V, VIII) each improved in only three areas. This supports the upward inflection in the Lecture w/PAT plotline from posttest to pretest shown in Figure 6.4 and discussed earlier. Also, the Lecture Only/Post Test value for Task I (76.2%) may be considered anomalous since 16 out of 21 students received full credit for all questions in the task whereas the remaining 5 students received no credit for any question in the task. No other entry in Table 7.1 exhibited that type of bimodal behavior.

From pretest to posttest, substantial improvements occurred for Task II (Total increase = 29.2%), Task VI (Total increase = 10.5%), and Task VIII ((Total increase = 8.2%). The lecture portion of the instructional intervention specifically highlighted the main learning points covered by Task II, which dealt with the formal definition and conceptualization of multitasking in a single processor environment. The notable increase from pretest to posttest in each intervention group implies a marked correction to an initial student misconception about the proper meaning of multitasking and the computational cost of a context switch.

For Task II, although the Control group shows an increase from a very low pretest performance, note that the posttest score for the Control group is in the neighborhood of the pretest scores for the other two experimental groups, whose corresponding posttest scores exhibit an appreciable absolute success rate (83.3% and 69%). This points toward the conclusion that the students in the Lecture and Lecture w/PAT groups gained appropriate knowledge of multitasking concepts through the materials presented in the module on parallel concepts.

Task VI emphasizes proper load balancing, specifically within the context of a Master/Worker configuration in which jobs of various computation length are delegated to selected processors. Through this example, students are given a sequence of jobs and charged with determining the best distribution policy to minimize computation cost. The Control group demonstrated a substantial 18% performance increase in this task from pretest to posttest, whereas the Lecture/PAT group performance increased by 8.7%. Absolute posttest performance levels for each of the groups were confined to a narrow range (45.2% to 48%). Although Master-Worker patterns were addressed briefly in the materials provided to the Lecture Only and Lecture w/PAT groups, no direct example of load balancing was presented or assigned. Since each group had developed a short Java program by the time of the posttest, the improvement among all three groups for this task may be attributed to a heightened sense of the time dimension in computation, namely, some programs will take noticeably more time depending on the computational requirements.

Task VIII provided a physical model of Amdahl's and Gustafson's Laws, proposing problems of both fixed and scalable size. Students must recognize that simply adding an arbitrary number of processors to a parallel solution may not always be the correct approach. Conversely, when given a specific number of processors, students should consider increasing the problem size to match the corresponding computational power of the system. As with Task VI, there were pretest to posttest performance

increases for all three groups, with the highest belonging to the Lecture Only group (12.1%). Presentations for the Lecture Only and Lecture w/PAT group emphasized the common case in scientific computing whereby larger systems typically invite and accommodate proportionately larger input data sets. Weather forecasting and remote sensing were used as examples.

For both Task VI and Task VIII, applying a "think-aloud" approach while students answered the survey questions would be illuminating, especially to define more clearly the fundamental concepts employed by the Control group student subjects as they were reasoning through the problem and arriving at a solution. It may be discovered that the basic problem solving capabilities of all student participants had improved enough over the three week period to account for the performance increase for these specific tasks.

The nonparametric analysis of the single PoPS written design question directly supports Conclusions 1 and 3 above. The results from the Kruskal-Wallis between-subjects tests and the associated pairwise Mann-Whitney U tests indicate statistically significant differences in the following design question Likert ratings: 1) Control pretest vs. Lect/PAT pretest, 2) Lecture Only pretest vs. Lect/PAT pretest, and 3) Control recap-test vs. Lecture/PAT recap-test. Detailed results are reported in Table 6.14 and Table 6.15 shows the relative differences in mean rank among the treatments. A concise snapshot comparing the design question performance levels for each treatment is given in Figure 7.1, which plots the calculated medians from Table 6.13.

Conclusion 1 is supported by the nonparametric analysis since there are no significant differences between the Control and Lecture Only group for any of the repeated measures. Conclusion 3 is strengthened by the repeated measure main effect analysis using the Friedman test and associated pairwise Wilcoxon tests as reported in Table 6.18. These tests indicate a significant difference ($p < 0.001$) among the re-

peated measures if matched data from all three intervention groups are combined. Further investigation reveals the statistically significant difference (p=0.001) between the posttest and pretest scores, verifying an associated "bump" in design question student performance independent of the classroom intervention. The 0.067 p-value between the recap-test scores and pretest scores is close to the 0.05 significance level, indicating that student scores at the nine-week mark in the course were still noticeably better than pretest scores.

Figure 7.1 presents an interesting symmetry in the median scores for the Control group and Lecture/PAT group. The Control median rises from 1.5 to 2.0 between pretest and posttest, and then remains at 2.0 for the recap-test. Conversely, the Lecture/PAT group stays consistent at 2.0 for both the pretest and posttest, and then drops to 1.5 for the recap-test. This decrease in the Lecture/PAT group median from posttest to recap-test in the nonparametric analysis counters the upward trend between posttest and recap-test for the same group in the parametric analysis as shown in the estimated marginal mean profile plot of Figure 6.4. This would indicate that at the retention level, the multiple choice section of the PoPS and the design question section of the PoPS are each monitoring fundamentally different student aptitudes about parallel concepts.

The overall shape of the Lecture/PAT performance curve in Figure 7.1 also confounds any support for Conclusion 2 that may have been derived from the nonparametric analysis. In fact, the relative measures given in Table 6.15 show that, at the retention level, performance of students in the Lecture/PAT group is worse than student performance in the Control group. Student interest or motivation in taking the survey during Week 9 could be postulated as a factor in the Lecture/PAT score decrease, but this study does not attempt to monitor these influences. If a correlation is assumed between collective student interest and the number of unanswered design questions, then the data from Table 6.12 reveal that for the retention test, student

subjects in the Lecture/PAT group (2 unanswered out of 33) demonstrated equivalent "motivation" to take the survey as the Control group (2 unanswered out of 27). Additional data regarding the long-term influence of the Lecture/PAT intervention on student perceptions of parallelism are necessary to clarify this result.

7.2 Future Work

This research acquired preliminary data and performed a detailed statistical analysis to determine the effect of specific CS1 classroom interventions on student comprehension of concepts related to parallel computing. Because of the recent rapid emergence of multicore, many-core, and other parallel architectures, it is vital that computer science education establish training regimens in the area of concurrent systems appropriate to the needs of the student community and industry partners. Studies such as the one described here will help inform computer science curriculum designers, instructors, and administrators about proper parallel programming course content and placement within a given baccalaureate program.

Future investigations into this important area may benefit from the following enhancements to this study:

1. Establish the validity and reliability of the Perceptions of Parallelism Survey (PoPS) evaluation instrument according to the American standard educational test development guidelines as described earlier in this chapter. Take appropriate measures to evolve the current PoPS into a widely accepted Parallel Concept Inventory that is useful to a broad spectrum of computer science programs, from large research universities to small liberal arts colleges.

2. Related to the previous item, specifically apply the "think-aloud" approach while students answer the PoPS questions to more clearly define the fundamental concepts employed by student subjects as they reason through the problem

and arrive at a solution. This strategy will open a window into student parallel thinking and allow for the refinement of PoPS questions to better target student misconceptions.

3. Expand the usage and functionality of the Parallel Analysis Tool (PAT) designed specficially for this research. Allow students to work more extensively with the tool, either through an additional assignment or in-class demonstrations. Enhance the PAT to help students identify shared resources and critical sections.

4. Perform a duplicate study to confirm or disprove the results generated by the current investigation. The parametric analysis for this study came very close to establishing a statistically significant interaction effect between the independent and repeated measure factors. In addition, the relatively low design question recap-test score for the Lecture/PAT group can be verified.

5. Perform similar studies in which the parallel module length and location within the course is modified. A CS1 course which can accommodate a longer treatment of this topic may reveal different posttest and retention test performance from the student subjects. Also, test results may be influenced by the placement of the module within the course. Student comprehension levels might be different in response to a parallel module occurring during the last three weeks of the course when compared to the same module occurring during the first three weeks of the course, as was done in this study.

6. Evaluate the current written design question for future versions of the PoPS. This may involve eliminating the design question, increasing the overall number of "essay" type questions, or refining the current design question to target a more specific student aptitude in parallel design.

7. When evaluating student performance, provide separate, independent admin-

istration times for the multiple choice section and the design question section. Determine if the placement of the single design question at the end of the current PoPS has any influence on student motivation and performance.

8. Implement a long-term longitudinal study of the student subjects participating in this research. As students progress through upper-division and capstone courses in the CS program, administer the PoPS periodically to monitor each student's comprehension of key parallel computing concepts. Tracking the evolution of PoPS performance levels throughout a student's educational career supports the accurate assessment of overall student retention of parallel concepts and assists in the development of an appropriate parallel programming curriculum.

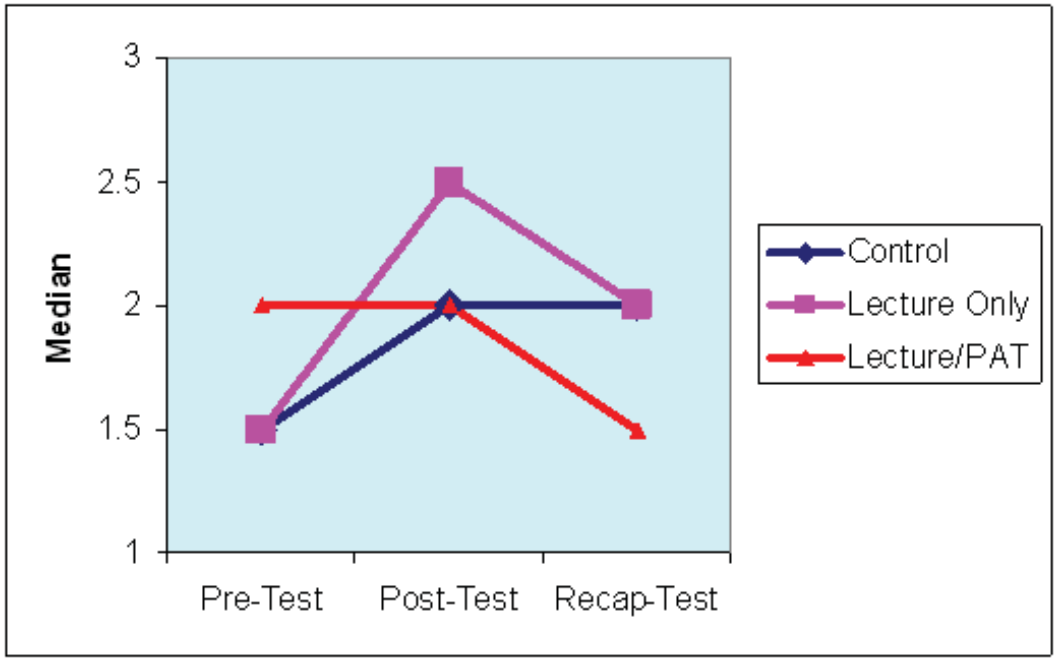


Figure 7.1: Design Question Medians

Table 7.1: Student Subject Performance by Task (% Correct)

		Pretest	**Posttest**	**Recap-Test**
	Control	82.0%	86.0%	80.4%
Task I	Lecture Only	90.0%	76.2%	86.8%
	Lecture/PAT	75.8%	82.8%	88.9%
	Total	81.6%	82.0%	85.5%
	Control	26.0%	54.0%	34.8%
Task II	Lecture Only	52.5%	83.3%	73.7%
	Lecture/PAT	40.3%	69.0%	70.4%
	Total	38.8%	68.0%	59.4%
	Control	57.3%	53.3%	56.5%
Task III	Lecture Only	61.7%	61.9%	50.9%
	Lecture/PAT	43.0%	43.7%	45.7%
	Total	52.6%	52.0%	50.7%
	Control	30.3%	38.3%	37.3%
Task IV	Lecture Only	40.7%	44.9%	40.6%
	Lecture/PAT	35.5%	37.4%	43.4%
	Total	35.2%	39.8%	40.6%
	Control	33.6%	40.0%	33.9%
Task V	Lecture Only	35.0%	40.0%	41.1%
	Lecture/PAT	36.8%	42.8%	45.9%
	Total	35.3%	41.1%	40.6%
	Control	30.0%	48.0%	51.1%
Task VI	Lecture Only	41.3%	45.2%	36.8%
	Lecture/PAT	38.7%	47.4%	48.1%
	Total	36.5%	47.0%	46.0%
	Control	31.0%	34.0%	32.6%
Task VII	Lecture Only	40.0%	31.0%	23.7%
	Lecture/PAT	30.6%	39.7%	33.3%
	Total	33.2%	35.3%	30.4%
	Control	38.7%	46.7%	47.8%
Task VIII	Lecture Only	45.0%	57.1%	57.9%
	Lecture/PAT	46.2%	51.7%	49.4%
	Total	43.4%	51.6%	51.2%

APPENDIX A

PERCEPTIONS OF PARALLELISM SURVEY

Perceptions of Parallelism Survey

by

Brian Rague

August 2009

Perceptions of Parallelism Survey

The *Perceptions of Parallelism Survey* (PoPS) is a multiple-choice "test" designed to assess student understanding of the *most basic* concepts associated with parallel processes. The PoPS attempts to measure both student recognition and analysis skills in the design and execution of scenarios that may involve simultaneous actions which are often duplicated and repetitive.

Perceptions of Parallelism Survey

Please:

*Do **not** write anything on this questionnaire.*

*Mark your answers clearly on the Answer Sheet provided. Give **only one** answer per item.*

*Do **not** skip any question.*

*Avoid guessing. Your answers should reflect what **you** personally think.*

On the Answer Sheet:

Fill in your name and section (course number and meeting time).

Plan to finish this questionnaire in 60 minutes.

Thank you for your cooperation.

(I) USE THE STATEMENT BELOW TO ANSWER THE NEXT TWO QUESTIONS (1 and 2).

A computer must calculate the three arithmetic statements given below. *Assume a, b, c, d, e, f, i, j are all integers.* The time required to calculate each individual statement is 1 nsec.

(I) `c = a + b;`
(II) `d = e + f;`
(III) `j = i - c;`

1. Assume one processing unit is available to perform the computation. What is the minimum amount of time required to calculate all three statements?
   (A) 1 nsec
   (B) 2 nsec
   (C) 3 nsec
   (D) 4 nsec
   (E) 6 nsec

2. Assume two processing units are available to perform the computation. What is the minimum amount of time required to calculate all three statements?
   (A) 1 nsec
   (B) 2 nsec
   (C) 3 nsec
   (D) 4 nsec
   (E) 6 nsec

(II) USE THE STATEMENT BELOW TO ANSWER THE NEXT TWO QUESTIONS (3 and 4).

While attending a lecture in a classroom equipped with computers at each seat, you decide to browse the Internet. The lecture topic is important so you periodically take notes while still viewing the browser. You discover a website with information you would like to pass to a friend, so you now decide to "multitask" the following:

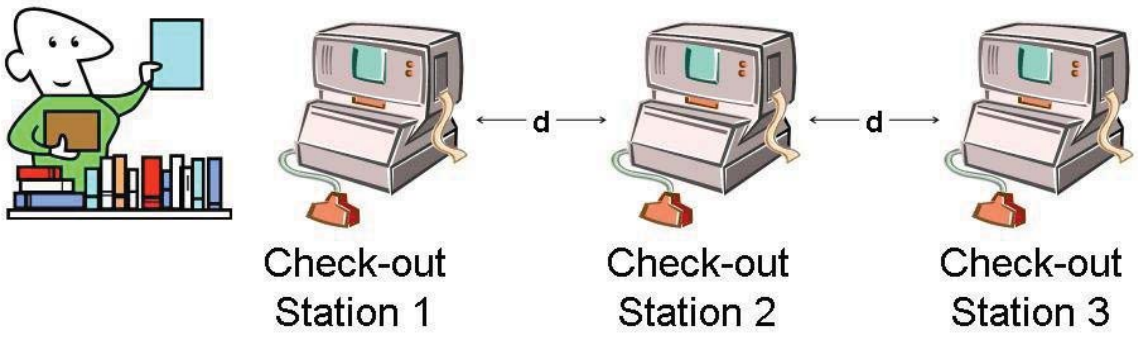
(I) texting your friend on your cell phone
(II) clicking to and viewing a *new* website
(III) listening to the lecture and taking notes

3. Choose the best description of how you are *actually* performing the three tasks.
   (A) Tasks I, II, and III are being executed simultaneously
   (B) Switching occurs among the tasks in the following order with no delay between tasks: Task I, Task II, Task III, Task I, Task II, Task III, Task I, Task II, Task III, …
   (C) Switching occurs among the tasks in no particular order with no delay between tasks.
   (D) Switching occurs among the tasks in the following order with a measurable delay between tasks: Task I, Task II, Task III, Task I, Task II, Task III, Task I, Task II, Task III, …
   (E) Switching occurs among the tasks in no particular order with a measurable delay between tasks.

4. Choose the best description of the time benefits or deficits of multitasking.
   (A) It takes more time to complete the three tasks using multitasking when compared to focusing on and completing each individual task in succession.
   (B) It takes less time to complete the three tasks using multitasking when compared to focusing on and completing each individual task in succession.
   (C) It takes the same time to complete the three tasks using multitasking when compared to focusing on and completing each individual task in succession.
   (D) It takes the same time to complete the three tasks using multitasking when compared to focusing on and completing each individual task in succession. However, for six or more tasks, multitasking would save time.
   (E) It takes more time to complete the three tasks using multitasking when compared to focusing on and completing each individual task in succession. However, for six or more tasks, multitasking would save time.

(III) USE THE STATEMENT AND FIGURE BELOW TO ANSWER THE NEXT THREE QUESTIONS (5, 6, and 7).

You would like to check out five books from the library. There are currently three automated check-out machines available as shown in the accompanying figure. To operate the check-out machine, the book is placed on the machine and a "start" button is pressed, at which point the process is automated. After 10 seconds elapses, the book has been successfully checked-out. The machine can be left unattended after initiating the check-out process, but the book must remain on the machine during the entire 10 second period. Assume it takes 0 seconds to initiate the check-out process.



5. If one person is allowed to use only a single machine, which of the following changes to the above resources would allow you to take the *least* time to check out the five books?
   (A) A fourth check-out machine becomes available.
   (B) A fourth check-out machine becomes available, and three friends show up and check out one book each.
   (C) Only two machines are now available, but four friends show up and check out one book each.
   (D) A fourth and fifth check-out machine becomes available, and two friends show up and check out one book each.
   (E) No changes.

6. In the figure, assume it takes 5 seconds to travel the distance *d* between each check-out station. Also assume there is a 2 second delay between checking out successive books at a single station. You are located at check-out station 1. Which strategy requires the *least* time for you to check out three books (Book One, Book Two, and Book Three)?

   (A) Check out Book One at Station 1; move to Station 2; check out Book Two at Station 2; move to Station 3; check out Book Three at Station 3.
   (B) Check out Book One at Station 1; move to Station 2; check out Books Two and Three at Station 2.
   (C) Start check out process for Book One at Station 1; move to Station 2; start check out process for Book Two at Station 2; move to Station 3; start check out process for Book Three at Station 3; move to Station 1; retrieve Book One; move to Station 2; retrieve Book Two; move to Station 3; retrieve Book Three.

(D) Start check out process for Book One at Station 1; move to Station 2; start check out process for Book Two at Station 2; move to Station 3; check out Book Three at Station 3; move to Station 1; retrieve Book One; move to Station 2; retrieve Book Two

(E) Check out all three books from check-out station 1

7. What modification(s) in scenario (D) in question 6 would *increase* the time it takes to check out three books (Book One, Book Two, and Book Three) when compared with the original scenario? Actions indicated with strikethroughs are not performed; bold, italicized actions have been added.

(A) ~~Start check out process for~~ ***Check out*** Book One at Station 1; move to Station 2; start check out process for Book Two at Station 2; move to Station 3; check out Book Three at Station 3; ~~move to Station 1; retrieve Book One;~~ move to Station 2; retrieve Book Two

(B) Start check out process for Book One at Station 1; move to Station 2; start check out process for Book Two at Station 2; move to Station 3; check out Book Three at Station 3; ~~move to Station 1; retrieve Book One; move to Station 2; retrieve Book Two~~ ***move to Station 2; retrieve Book Two; move to Station 1; retrieve Book One***

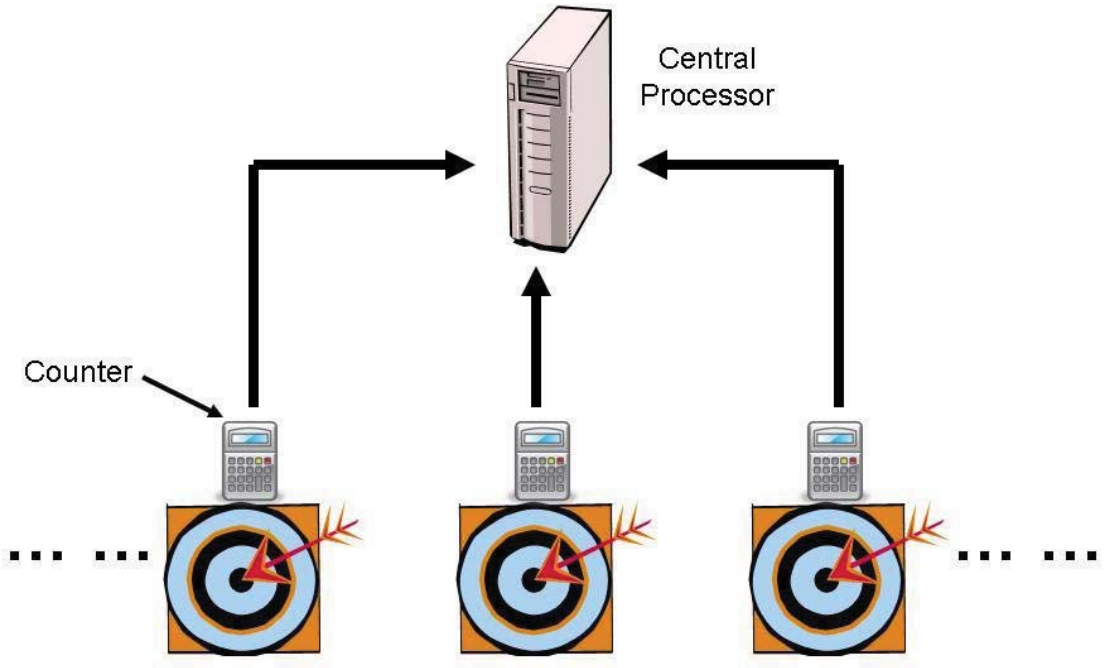
(C) ~~Start check out process for~~ ***Check out*** Book One at Station 1; move to Station 2; ~~Start check out process for~~ ***Check out*** Book Two at Station 2; move to Station 3; check out Book Three at Station 3; ~~move to Station 1; retrieve Book One; move to Station 2; retrieve Book Two~~

(D) Start check out process for Book One at Station 1; move to Station 2; Start check out process for Book Two at Station 2; ~~move to Station 3; check out Book Three at Station 3;~~ move to Station 1; retrieve Book One; move to Station 2; retrieve Book Two; ***move to Station 3; check out Book Three at Station 3***

(E) Start check out process for Book One at Station 1; ~~move to Station 2; start check out process for Book Two at Station 2;~~ move to Station 3; check out Book Three at Station 3; ***move to Station 2; start check out process for Book Two at Station 2;*** move to Station 1; retrieve Book One; move to Station 2; retrieve Book Two

(IV) USE THE STATEMENT AND FIGURE BELOW TO ANSWER THE NEXT SEVEN QUESTIONS (8 through 14).

The accompanying figure depicts a configuration for counting the number of darts thrown within a specific target region on N dartboards. Attached to the back of each dartboard is a counter which keeps track of the number of "hits" and "misses" for that dartboard. The individual hits/misses count from each dartboard is then routed to a central processor which sums the total number of hits and misses and provides a final value equal to hits/(hits+misses).



N Dartboards

The location of each dart thrown at any individual dartboard is considered to be randomly generated with probability evenly distributed across the face of the dartboard. Darts must be thrown successively at a single dartboard; they cannot be thrown simultaneously at a single dartboard. Assume the time required to throw a single dart is fixed. An *experiment* is defined as throwing a total of 500 darts at a total of N dartboards.

Analysts have determined that the overall *cost* of performing a single experiment can be attributed to the following:

(I) N, the total number of dartboards
(II) T, the time required to throw the darts
(III) C, the communication time between the individual counters and the central processor
(IV) R, the reduction time required for the central processor to compute the final value.

8. Which of the following statements is the *most accurate* when describing C, the communication time cost for a given experiment?

   (A) C will be the minimum time it takes any dartboard counter to send its hits/misses total to the central processor
   (B) C will be the maximum time it takes any dartboard counter to send its hits/misses total to the central processor
   (C) C will be the sum of all the times it takes each dartboard counter to send its hits/misses total to the central processor
   (D) C will be the average of all the times it takes any dartboard counter to send its hits/misses total to the central processor
   (E) C will be the median of all the times it takes any dartboard counter to send its hits/misses total to the central processor

9. Which of the following statements is the *most accurate* when describing R, the reduction time cost for a given experiment?

   (A) R increases as C increases
   (B) R increases as C decreases
   (C) R increases as N increases
   (D) R increases as N decreases
   (E) R increases as N stays constant but the number of darts increases

10. Assume the costs N, C, and R are negligible. Indicate the best strategy to minimize T given an experiment of 500 darts.

   (A) Use one dartboard – 500 darts/board (N = 1)
   (B) Use ten dartboards – 50 darts/board (N = 10)
   (C) Use fifty dartboards – 10 darts/board (N = 50)
   (D) Use five hundred dartboards – 1 dart/board (N = 500)
   (E) Use one thousand dartboards – 0.5 darts/board (N = 1000)

11. Assume the costs C and R are negligible, and assume that each dartboard has a cost equivalent to throwing five darts in succession. Which of the following choices has the *least* overall cost for an experiment of 500 darts?

   (A) Use one dartboard – 500 darts/board (N = 1)
   (B) Use ten dartboards – 50 darts/board (N = 10)
   (C) Use fifty dartboards – 10 darts/board (N = 50)
   (D) Use two hundred fifty dartboards – 2 darts/board (N = 250)
   (E) Use five hundred dartboards – 1 dart/board (N = 500)

12. Assume the costs C and R are negligible, and assume that each dartboard has a cost equivalent to throwing two darts in succession. Which of the following choices has the *least* overall cost for an experiment of 500 darts?

   (A) Use one dartboard – 500 darts/board (N = 1)
   (B) Use ten dartboards – 50 darts/board (N = 10)

(C) Use fifty dartboards – 10 darts/board (N = 50)
(D) Use two hundred fifty dartboards – 2 darts/board (N = 250)
(E) Use five hundred dartboards – 1 dart/board (N = 500)

13. Assume the costs T, C, and R are negligible. Indicate the best strategy to minimize N given an experiment of 500 darts.

    (A) Use one dartboard – 500 darts/board
    (B) Use ten dartboards – 50 darts/board
    (C) Use fifty dartboards – 10 darts/board
    (D) Use five hundred dartboards – 1 dart/board
    (E) Use one thousand dartboards – 0.5 darts/board

14. You perform an experiment of 500 darts with the number of dartboards N = 40. For a second subsequent experiment of 500 darts, you are provided with an additional dartboard such that N = 41. How does the cost T vary from the first experiment to the second experiment?

    (A) T doubles
    (B) T increases slightly but does not double
    (C) T stays the same
    (D) T decreases slightly but is not halved
    (E) T is halved

(V) USE THE STATEMENT BELOW TO ANSWER THE NEXT FIVE QUESTIONS (15 through 19).

An experimental setup involves 5 persons (Ann, Beth, Carl, Dee, and Ed) and a stack of 60 cards, with each card containing a single word in capital letters. The persons are asked to arrange the cards in alphabetical order. Assume it takes a single individual N seconds to sort N cards. An *exchange* occurs if one person passes a single stack of one or more cards to another person.

For example, *two* separate exchanges have taken place if Ann passes a three card stack to Beth, and then passes a ten card stack to Beth. *One* exchange takes place if Ann passes a single thirteen card stack to Beth.

Assume an exchange requires 2 seconds, independent of the size of the stack being transferred.

Finally, assume if a person in the experiment receives K separate *alphabetized* stacks, it requires only K seconds to produce a correctly sorted single stack.

15. Which strategy offers the *most* time-efficient approach?

(A) Ann sorts all 60 cards while the others observe.

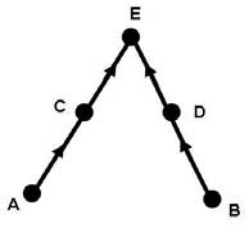
(B) Ann, Beth, Carl, and Dee sort 15 cards each. After sorting, each of the four passes his/her sorted stack to Ed (1 exchange/person). Ed then does a final sort on the 4 stacks.

(C) Ann sorts 30 cards and Beth sorts 30 cards while the others observe. After sorting, Beth passes her stack to Ann in a single exchange. Ann does a final sort on the 2 stacks.

(D) Ann, Beth, Carl, Dee, and Ed sort 12 cards each. After sorting, Ann passes her stack to Beth in a single exchange. Beth sorts the two stacks. After sorting, Beth passes her resulting stack to Carl in a single exchange. Carl sorts the two stacks. After sorting, Carl passes his resulting stack to Dee. Dee sorts the two stacks. After sorting, Dee passes his resulting stack to Ed. Ed does the final sort on the two stacks.

(E) Ann, Beth, Carl, and Dee sort 15 cards each. After sorting, Ann passes her sorted stack to Carl, and Beth passes her sorted stack to Dee. Both Carl and Dee perform sorts on their two stacks. After sorting, Carl and Dee each pass his sorted stack to Ed (1 exchange/person). Ed then does a final sort on the 2 stacks.

16. Choose which strategy in Question 15 is best represented by the graph below? Dots indicate persons and arrows indicate a stack exchange.



(A) Strategy A.
(B) Strategy B.
(C) Strategy C.
(D) Strategy D.
(E) Strategy E.

17. Assume an exchange now requires 12 seconds. Which strategy in Question 15 offers the *least* time-efficient approach?

(A) Strategy A.
(B) Strategy B.
(C) Strategy C.
(D) Strategy D.
(E) Strategy E.

18. Assume that if P persons work together to sort N cards, the communication investment is such that it takes P*N seconds to perform the sort. Similarly, if P persons work together to sort K separate *alphabetized* stacks, it requires P*K seconds to produce a correctly sorted single stack. Assume an exchange takes 2 seconds as originally stated. Which strategy offers the *most* time-efficient approach?

(A) Ann sorts all 60 cards while the others observe.
(B) Ann, Beth, Carl, and Dee sort 15 cards each. After sorting, each of the four passes his/her sorted stack to Ed (1 exchange/person). All 5 persons then work together on a final sort of the 4 stacks.
(C) Ann and Carl sort 30 cards together while Beth and Dee sort 30 cards together. After sorting, Beth/Dee pass their stack to Ann/Carl in a single exchange. Ann and Carl together perform a final sort on the 2 stacks.
(D) Ann, Beth, Carl, Dee, and Ed sort 12 cards each. After sorting, Ann passes her stack to Beth in a single exchange. Ann and Beth work together to sort the two stacks. After sorting, Ann/Beth passes their resulting stack to Carl in a single exchange. Carl sorts the two stacks. After sorting, Carl passes his resulting

stack to Dee. Carl and Dee work together to sort the two stacks. After sorting, Carl/Dee passes their resulting stack to Ed. Ed does the final sort on the two stacks.

(E) Ann and Carl sort 30 cards together while Beth and Dee sort 30 cards together. After sorting, Beth/Dee pass their stack to Ed and Ann/Carl pass their stack to Ed. All 5 persons then work together on a final sort of the 2 stacks.

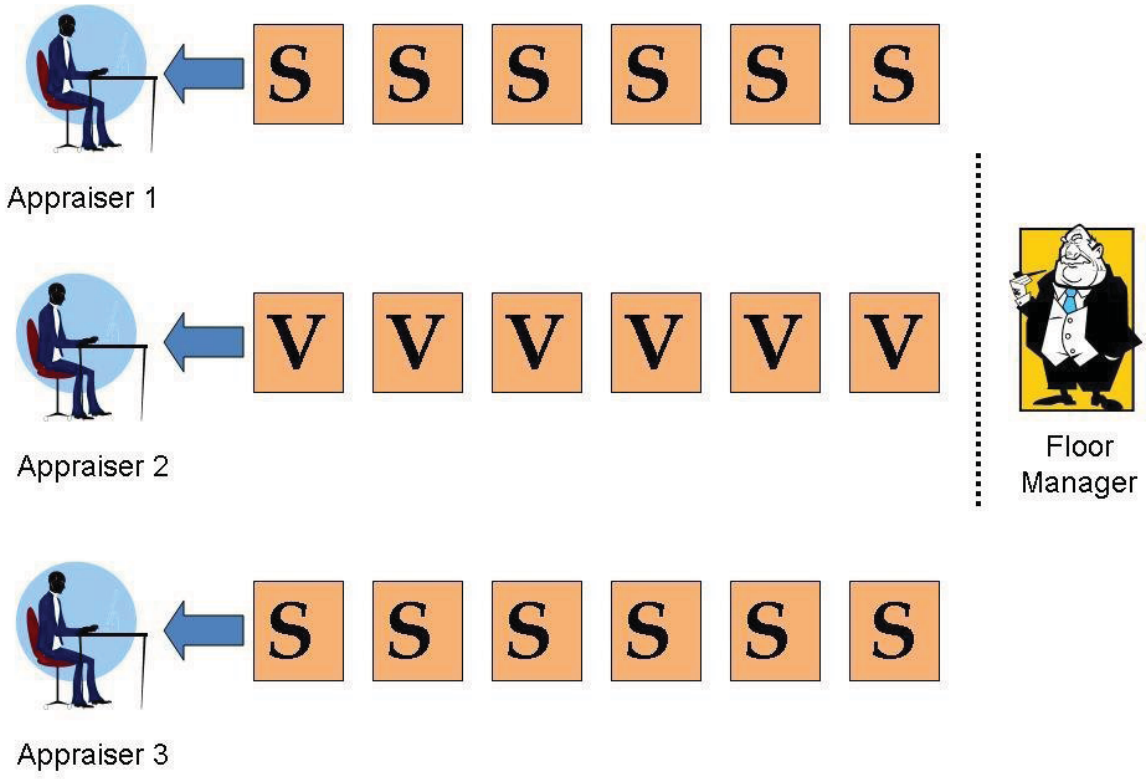
19. Assume an exchange now requires 12 seconds. Which strategy in Question 18 offers the *least* time-efficient approach?

(A) Strategy A.
(B) Strategy B.
(C) Strategy C.
(D) Strategy D.
(E) Strategy E.

(VI) USE THE STATEMENT AND FIGURE BELOW TO ANSWER THE NEXT FOUR QUESTIONS (20 through 23).

Three antique experts are seated at separate locations within a convention hall as shown in the diagram below. When they arrive, individuals who wish to get their single item appraised are asked by the floor manager to stand in one of the three lines associated with each station. Individuals are not allowed to jump lines. Floor managers are not able to accurately appraise items.

Items are categorized by the experts as valuable (V) or sentimental (S). Valuable antiques require exactly 1 minute of appraisal time to determine their worth. Sentimental antiques require exactly 10 seconds of appraisal time to be recognized as having no monetary value. Individuals in the figure are represented as squares containing either the letter 'V' or 'S' depending on the type of item they wish to be appraised.



20. Given 18 individuals arranged in the three lines as shown in the figure, what is the amount of time required to complete appraisals of all 18 items?

(A) 1 minute
(B) 2 minutes
(C) 6 minutes
(D) 7 minutes
(E) 8 minutes

21. Instead of sending an individual immediately to a specific line on arrival, the floor manager can ask all individuals to wait in a single 'holding' line (the dotted line in the figure) until an appraiser station is open, at which time the manager could choose to send one or more individuals in the 'holding' line to the open appraiser station.

Given 6 individuals arriving in order from left to right as shown above, which of the following strategies requires the *least* amount of time to complete appraisals of all 6 items?

(A) Use a holding line. Send one individual at a time to an open appraiser station.
(B) Use a holding line. Send two individuals at a time to an open appraiser station.
(C) Use a holding line. Send three individuals at a time to an open appraiser station.
(D) Use no holding line. Randomly divide the group into three lines of two individuals each.
(E) Use no holding line. Put all individuals in a single line.

22. Assume that 10 seconds are required for the floor manager to send one or more individuals from the 'holding' line to an open appraiser station. Zero seconds are required when no holding line is used, and individuals can line up at appraisal stations immediately. Given 6 individuals arriving in order from left to right as described in Question 21 above, which strategy in Question 21 requires the *least* amount of time to complete appraisals of all 6 items?

(A) Strategy A.
(B) Strategy B.
(C) Strategy C.
(D) Strategy D.
(E) Strategy E.

23. Assume zero seconds are required for the floor manager to send one or more individuals from the 'holding' line to an open appraiser station. Given 12 individuals arriving in order from left to right as shown above, which of the following strategies requires the *most* amount of time to complete appraisals of all 12 items?

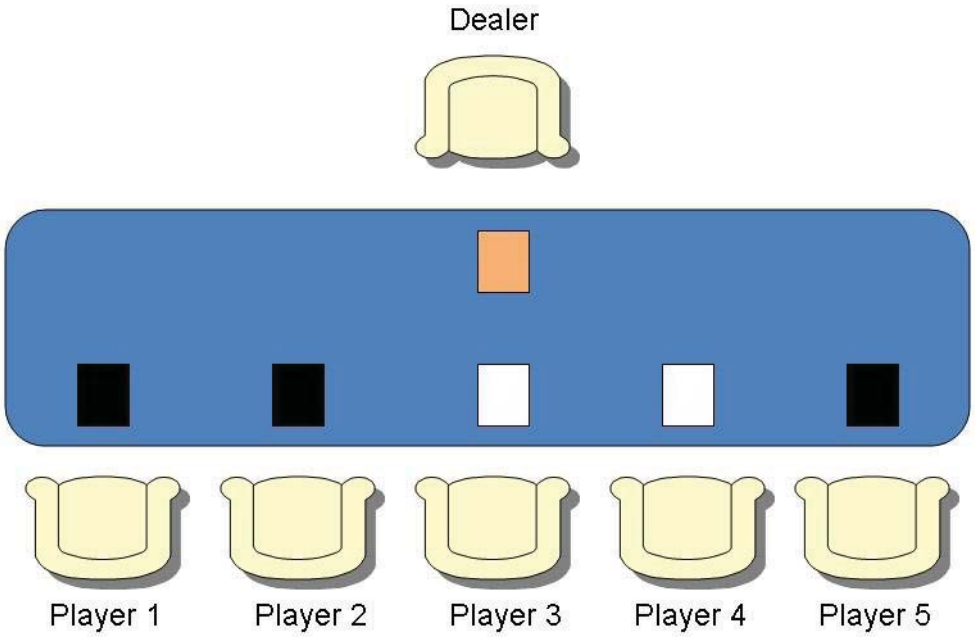
(A) Use a holding line. Send one individual at a time to an open appraiser station.
(B) Use a holding line. Send two individuals at a time to an open appraiser station.
(C) Use a holding line. Send three individuals at a time to an open appraiser station.
(D) Use a holding line. Send four individuals at a time to an open appraiser station.
(E) Use a holding line. Send six individuals at a time to an open appraiser station.

(VII) USE THE STATEMENT AND FIGURE BELOW TO ANSWER THE NEXT FOUR QUESTIONS (24 through 27).

A card game involves a deck of 52 cards in which half are black and half are white. As shown in the figure below, a single card is dealt face-up to each of five players to start the game, and all the players sit next to each other across from the dealer. The game proceeds as a series of rounds.

For a given round, each player looks at the card(s) of the player(s) seated next to him/her. If the player's card is black and at least *one* of his/her adjacent players has a white card, then the player's black card must be exchanged for a white card from the deck. A player with a white card simply keeps the card. Successive rounds are played until only one person holds a black card. That person is declared the winner.

Note that the players on the end ("end" Players 1, and 5) only have one player seated next to them. All remaining players ("middle" Players 2, 3, and 4) each have two players seated next to them. These "middle" players must first check the left adjacent player's card followed by the right adjacent player's card.



24. Assume that it takes each player 3 seconds to check the card of a person seated next to him/her. Given the starting hand as shown in the figure, indicate the future winner of the game *and* the time required to complete the game.

(A) Player 1, 3 seconds
(B) Player 1, 6 seconds
(C) Player 2, 6 seconds
(D) Player 5, 3 seconds
(E) Player 5, 6 seconds

25. Assume that it takes each player 3 seconds to check the card of a person seated next to him/her, and "middle" players now check adjacent player's cards right-to-left, rather than left-to-right. Given the starting hand as shown in the figure, indicate the future winner of the game *and* the time required to complete the game.

(A) Player 1, 3 seconds
(B) Player 1, 6 seconds
(C) Player 2, 6 seconds
(D) Player 5, 3 seconds
(E) Player 5, 6 seconds

26. Assume that each "middle" player is dealt 3 cards of the same color, and each "end" player is dealt 2 cards of the same color. When the dealer shouts "SHARE," a round is performed in which each player gives one of his/her cards to an adjacent player, a transfer that takes 2 seconds. Note that "end" players share only one card, whereas "middle" players share two cards left-to-right. Players can now use the copy of the adjacent player's card to determine their card for the next round.

Given the starting hand as shown in the figure, indicate the time required to complete the game.

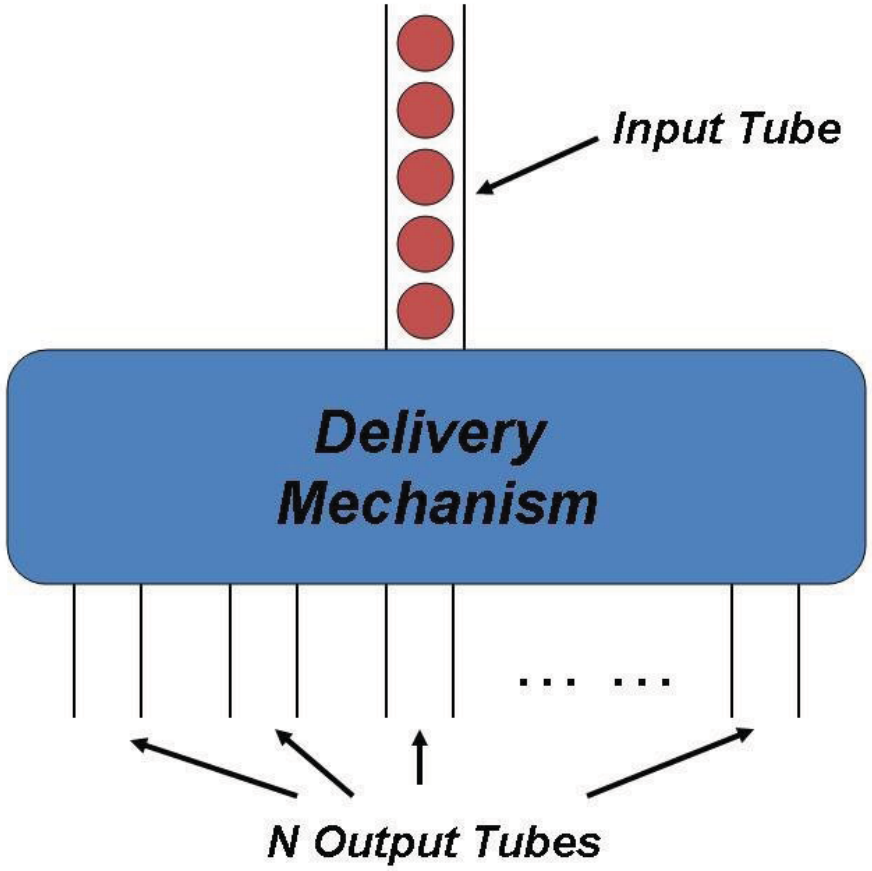
(A) 2 seconds
(B) 4 seconds
(C) 6 seconds
(D) 8 seconds
(E) 10 seconds

27. Assume a 100 player game in which only one of the 98 "middle" players can win. Using the "SHARE" round rules described in Question 26, indicate the time required to perform a single round.

(A) 2 seconds
(B) 4 seconds
(C) 50 seconds
(D) 98 seconds
(E) 100 seconds

(VIII) USE THE STATEMENT AND FIGURE BELOW TO ANSWER THE NEXT THREE QUESTIONS (28 through 30).

The figure below depicts a system comprising a theoretical delivery mechanism, one input tube, and N output tubes. A certain number of balls are placed into the input tube one at a time. The figure shows 5 balls in the input tube. The diameter of each tube is just slightly larger than the diameter of an individual ball.



The delivery mechanism is designed such that it takes 0 seconds to evenly distribute (as best as possible) the input balls among the output tubes. For example, for 5 input balls and 5 output tubes, the mechanism would place one ball in each output tube. However, for 5 input balls and 4 output tubes, the mechanism would place 2 balls in one of the output tubes and one ball in each of the remaining tubes.

For any given tube, either input or output, it takes K seconds for K balls to travel through the tube. In the figure, 5 seconds are required for the 5 balls to travel through the input tube.

28. As shown in the figure, assume 5 balls are delivered to the input tube. Assume the number of output tubes N is 3. How long will it take all balls to traverse through the system?

(A) 3 seconds
(B) 4 seconds
(C) 5 seconds

(D) 6 seconds
(E) 7 seconds

29. Given the initial configuration described in question 28, indicate the smallest change to the system that will minimize the amount of time it takes the balls to traverse through the system.

(A) Remove 1 output tube.
(B) Add 1 output tube.
(C) Add 2 output tubes.
(D) Add 5 output tubes.
(E) Add 20 output tubes.

30. Assume the number of output tubes N is 50, and we are required to run the system using no more than 102 seconds. What is the maximum number of balls that can be input to the system?

(A) 5 balls
(B) 20 balls
(C) 50 balls
(D) 80 balls
(E) 100 balls

(IX) PROVIDE A BRIEF NARRATIVE DESCRIPTION AND DIAGRAM OF A PARALLEL PROCESSING STRATEGY THAT OPTIMIZES SYSTEM PERFORMANCE TO SOLVE THE FOLLOWING PROBLEM.

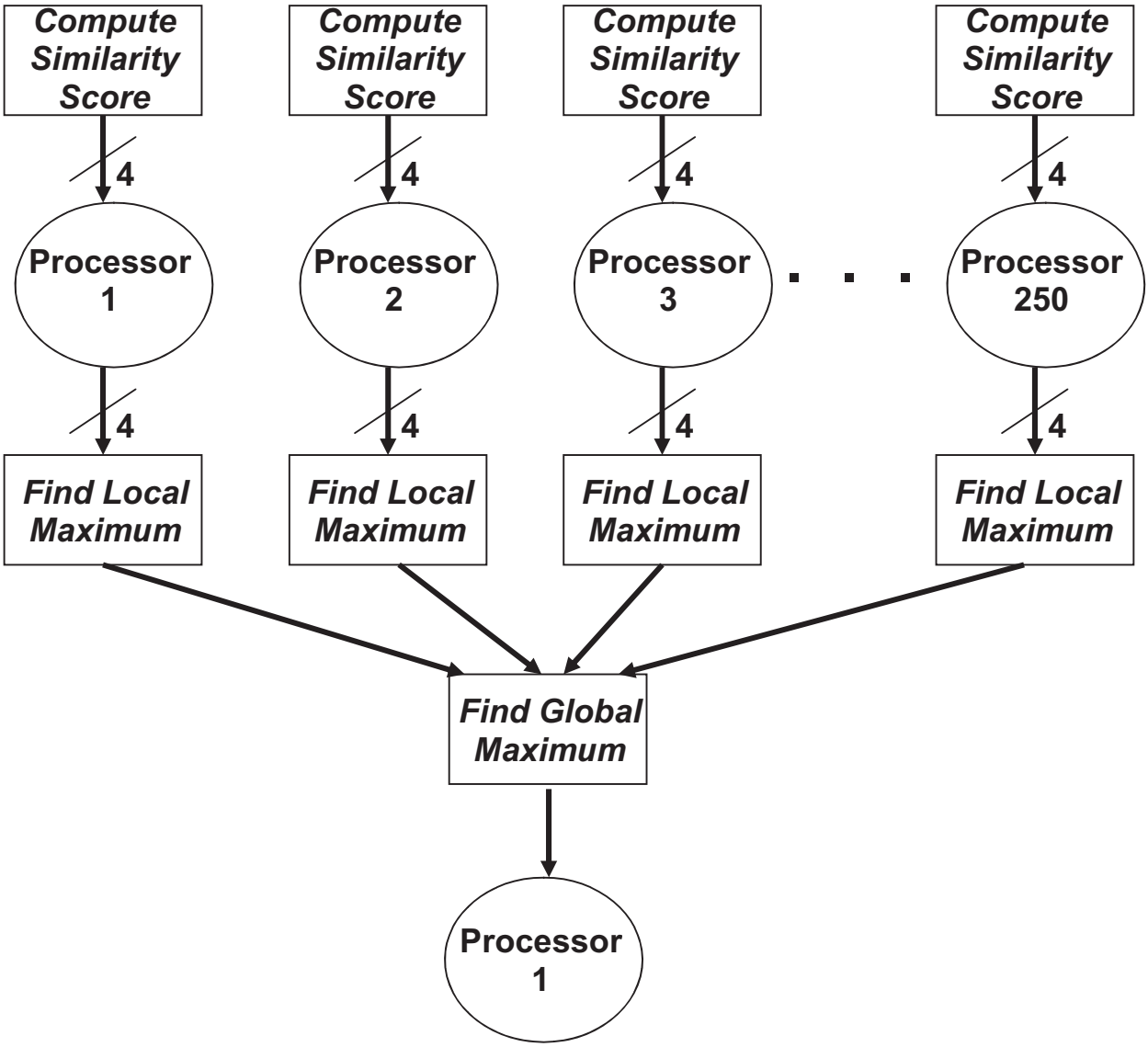
A criminal lab has recently received a grainy photograph of an individual who recently robbed a bank. The police believe the culprit is a repeat offender.

The lab database currently has 1000 photographs of suspects not currently incarcerated who have committed this type of robbery in the past. The lab has image processing software that can compare the photograph of the robber with the photograph of a potential suspect and produce what is called a “similarity score.” The higher the similarity score, the more likely the suspect is the robber. Because of photo conditioning and resolution requirements, computing a single comparison takes approximately 15 seconds.

The lab computer system has 250 available processors. Police are requesting the most likely suspect based on the lab’s analysis.

APPENDIX B

PoPS DESIGN QUESTION PROCESSING STRATEGY


Compute Similarity Score
Compute Similarity Score
Compute Similarity Score
Compute Similarity Score
4
4
4
4
Processor 1
Processor 2
Processor 3
Processor 250
4
4
4
4
Find Local Maximum
Find Local Maximum
Find Local Maximum
Find Local Maximum
Find Global Maximum
Processor 1

APPENDIX C

PoPS STUDENT ANSWER SHEET

Perceptions of Parallelism Survey

Version 1
(8/12/09)

Answer Sheet

Name: _____

Section: _____

1. ___	**6. ___**	**11. ___**	**16. ___**	**21. ___**	**26. ___**
2. ___	**7. ___**	**12. ___**	**17. ___**	**22. ___**	**27. ___**
3. ___	**8. ___**	**13. ___**	**18. ___**	**23. ___**	**28. ___**
4. ___	**9. ___**	**14. ___**	**19. ___**	**24. ___**	**29. ___**
5. ___	**10. ___**	**15. ___**	**20. ___**	**25. ___**	**30. ___**

FOR THE PROBLEM DESCRIBED IN TASK (IX), USE THE SPACE BELOW AND THE ATTACHED BLANK SHEET TO PROVIDE A BRIEF NARRATIVE DESCRIPTION AND DIAGRAM OF A PARALLEL PROCESSING STRATEGY THAT OPTIMIZES SYSTEM PERFORMANCE

APPENDIX D

LECTURE-ONLY EXPERIMENTAL GROUP ASGNT.

CS 1400
Fundamentals of Programming
Spring Semester 2010

INDIVIDUAL ASSIGNMENT **#1**

Exercise 2: Java Language Features

Coding Exercise and Recognizing Concurrency

Here is an example of a class, called `Integrator`. This will illustrate the behavior of for-loops, method calls, and some of the arithmetic operators. The program uses the trapezoidal rule to calculate the area under the curve $f(x) = x^2$ from $x = 0$ to $x = 2$. The details of the implemented algorithm are not important right now, but your ability to enter the code using correct syntax and formatting *is* important.

Enter this class into your Java development environment exactly as shown below including the comments. Compile and run the program.

```
public class Integrator {

   static int n = 1000000; // number of trapezoids
   static double[] results = new double[n];

   public static void main(String args[]) {

     /* Initialize starting Time */
     long t1 = System.currentTimeMillis();

     /* Compute Individual Trapezoid Areas */
     for(int i = 0; i < n; i++)
       results[i] = computeArea(i);

     double totalArea = 0.0;
     /* Sum up all Areas */
     for(int i = 0; i < n; i++)
       totalArea += results[i];

     /* Print out the Area. Total Area = ???? */
     System.out.println("Area = " + totalArea);

     /* Display program duration. Duration = ???? */
     long t2 = System.currentTimeMillis();
```

```
      System.out.println("Running Time:" + (t2-t1) + "
msecs");
  }

    // Uses trapezoid formula
   private static double computeArea(int section)
    {
      double height = 2.0/n;
      double leftXValue = section * height;
      double rightXValue = (section + 1) * height;
      double leftSide = leftXValue * leftXValue;
      double rightSide = rightXValue * rightXValue;
      double area = 0.5 * (leftSide + rightSide) * height;

      return area;
    }
}
```

Note that the + operator can be used inside the argument to `println()`, to construct a string from several different components at run-time.

Also note the different steps in the algorithm as indicated by each of the comments delimited by **/*…*/** in the main method. These annotations give clear descriptions of the individual task performed at each step, and provide a prominent illustration of the importance of code comments in helping other developers understand the purpose and execution of a program.

Now think about which, if any, of these individual tasks could be run more efficiently on a parallel computer. Our goal would be to reduce the time required to execute the program while still generating the correct result for the computed area. Consider only the five "commented" steps denoted in the main method.

Do the following:

→ In the source code, replace the two `????` items with the Total Area and Duration values generated by the program.
→ Indicate any steps that could potentially run faster on a parallel computer by placing the term "P=n" at the beginning of the comment associated with that step , where n is a number indicating the optimum number of processors. This term should be enclosed by the opening comment delimiter so that the program will still compile.
→ Submit the source code of `Integrator.java` for this exercise.

Exercise 3: Writing a Java Application

Java applications run "standalone:, without a driver program. A Java application consists of one or more classes, and can be as large or as small as needed. A class is frequently created for the sole purpose of encapsulating the `main()` method, necessary for the application to begin. Any class may contain the `main()` method, but there can be only one in a Java application. One common programming technique to handle this is to create a "jumping-off" class that contains the `main()` method, as well as any specialized initialization or termination routines for the application.

The signature for the `main()` method always looks like this:

```
public static void main (String args[]) {  }
```

The `public` keyword means it's available to the Java interpreter; the `static` keyword means there is only one occurrence of this function; the `void` keyword means it doesn't return a value. `args []` is the array of strings passed as arguments to the application [analogous to the `argv []` array in C and C++]. `args []` could be given another name – there is nothing special about "args". However, the first value in the array, `args [0],` is the first argument to the program, NOT the program name, as it would be in C or C++.

Since `args` is an array, it has a `length` member, indicating the number of values passed. You must test the value of `args.length` to see that your application receives an appropriate number of arguments. Java won't do it for you.

Coding Exercise and Recognizing Concurrency

Create a Java program called `AddArguments.java` that accepts exactly four command line arguments. Within the code, construct one Java statement that will add the first two arguments, and a second Java statement that will add the last two arguments. Then construct a third Java statement that will sum the two previous results. The program should display this final value as follows:

```
Final Sum: value
```

where *`value`* represents the sum of all four arguments.

Sample Session:

```
java AddArguments 8 5 3 2
Final Sum: 18
```

Hint: To convert from a Java String to an int datatype, use the method **Integer.parseInt().**

Do the following:

→ In the source code, comment the steps in the main method as shown in the previous exercise.
→ Indicate any steps that could potentially run faster on a parallel computer by placing the term "P=n" at the beginning of the comment associated with that step , where n is a number indicating the optimum number of processors. This term should be enclosed by the opening comment delimiter so that the program will still compile.
→ Submit the source code of `AddArguments.java` ***for this exercise.***

APPENDIX E

CS1-LEVEL QUIZ QUESTIONS ON PARALLEL CONCEPTS

Q. The running time of a program on a single CPU computer is 100 seconds, and the running time of the same program on a parallel computer is 25 seconds. What is the speedup value?

- 100
- 25
- ✔4
- 0.25

Q. **All** of the current top 500 fastest computers are single CPU systems

- True
- ✔False

Q. Indicate which kinds of complex problems can be solved using parallel computing. *Choose **all** that apply*

- ✔Computational Fluid Dynamics
- ✔Cosmology: Star Cluster Simulation
- ✔Climate Modeling
- ✔Protein Sequence Matching

Q. What step is **added** to the classical scientific method when employing computational systems?

- Observation
- Physical Experimentation
- ✔Numerical Simulation
- Theory

Q. What is the primary benefit of parallel computing when applied to a computational problem of fixed size?

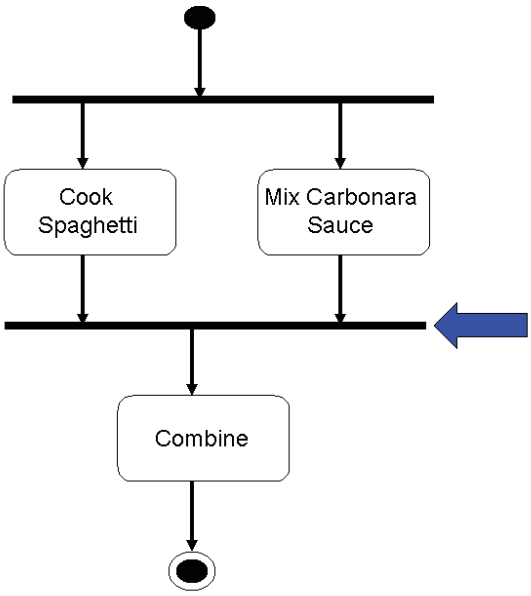
- Reduce data storage
- ✔Reduce computation time
- Reduce communication time
- Reduce system cost

Q. Which parallel computer architecture is best suited for problems where each processor produces results that are used by some or all of the other processors?

- ✔SMP
- Cluster

- Hybrid
- Grid

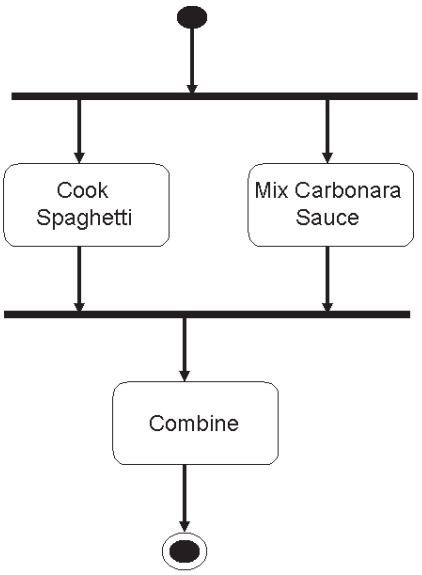
Q.



The above blue arrow points to which component of the Activity Diagram?

- transition
- activity
- fork
- stop
- ✔join

Q.



Based on the above Activity Diagram, which of the following statements is true?

- The "Cook Spaghetti" and "Mix Carbonara Sauce" activities must be performed in sequence.
- ✔The "Cook Spaghetti" and "Mix Carbonara Sauce" activities may be performed in parallel.
- The "Cook Spaghetti" and "Mix Carbonara Sauce" activities take exactly the same time.
- The "Cook Spaghetti" and "Mix Carbonara Sauce" activities must be performed simultaneously.

Q.

```
for(int i = 0; i < 100; i++)
{
        a[i] = b[i] + c[i];
}
```

In the above Java code, assume **a**, **b**, and **c** are each integer arrays of size 100. Which of the following statements is true?

- The statement in the body of the for-loop is not parallelizable.
- The statement in the body of the for-loop is parallelizable, and the optimum number of processors is 10.
- ✔The statement in the body of the for-loop is parallelizable, and the optimum number of processors is 100.
- The statement in the body of the for-loop is parallelizable, and the optimum number of processors is 1000.

Q.

```
int total = 0;
for(int i = 0; i < 100; i++)
{
total = total + a[i];
}
```

In the above Java code, assume **a** is an integer array of size 100. Which of the following statements is true?

- ✔The statement in the body of the for-loop is not parallelizable.
- The statement in the body of the for-loop is parallelizable, and the optimum number of processors is 10.
- The statement in the body of the for-loop is parallelizable, and the optimum number of processors is 100
- . The statement in the body of the for-loop is parallelizable, and the optimum number of processors is 1000.

Q.

```
(1) int a = 2;
(2) int b = 3;
(3) double m = (a+b)/2.0;
(4) double s = (a*a + b*b)/2.0;
(5) double v = s - (m*m);
```

Given the above Java statements, which of the following is true?
*Choose **all** that apply.*

- ✔Statements (1) and (2) can be executed concurrently.
- Statements (2) and (3) can be executed concurrently.
- Statements (2) and (4) can be executed concurrently.
- ✔Statements (3) and (4) can be executed concurrently.
- Statements (4) and (5) can be executed concurrently.

Q. Multitasking is the simultaneous performance of two or more tasks on a single CPU.

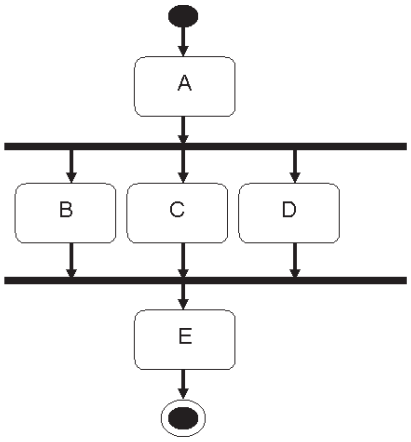
- True
- ✔False

Q. A context switch, used to manage multiple processes on a single CPU, takes 0 seconds of processing time.

- True

- ✔False

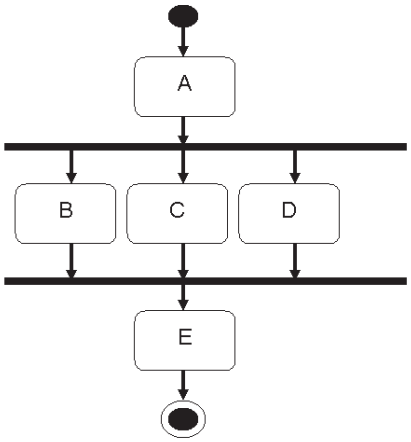
Q.



The above diagram is a representation of:

- Data Parallelism
- ✔Functional Parallelism
- Pipelining

Q.

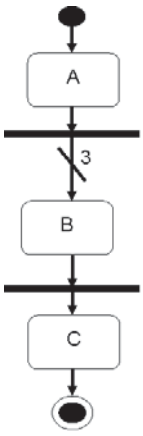


Given the above diagram, which of the following statements is true?
*Choose **all** that apply.*

- A sequential dependency exists between activity B and activity C
- A sequential dependency exists between activity C and activity D
- ✔A sequential dependency exists between activity A and activity C

- [ ] ✔A sequential dependency exists between activity D and activity E
- [ ] ✔A sequential dependency exists between activity A and activity E

Q.



The above diagram is a representation of:

- ✔Data Parallelism
- Functional Parallelism
- Pipelining

Q. Indicate the easiest, quickest, least expensive, and most popular approach to programming parallel computers.

- Extend an existing compiler
- ✔Extend an existing language
- Add a new parallel language layer on top of an existing sequential language
- Define a totally new parallel language and compiler system

Q. What is the primary obstacle to successfully introducing a new parallel programming language?

- Inventing new syntax
- Providing a new runtime system
- ✔Developers' resistance to learning a new language
- Providing a new compiler

APPENDIX F

PAT W/LECTURE EXPERIMENTAL GROUP ASGNT.

CS 1400
Fundamentals of Programming
Spring Semester 2010

INDIVIDUAL ASSIGNMENT **#1**

Important Note: ***To complete Exercises 2 & 3, the Parallel Analysis Tool (PAT) may be accessed as follows:***

A. From a CS laboratory computer:
1. **Use Remote Desktop Connection to log in to hermes.cs.weber.edu or 137.190.19.26 using your CS username and password.**
2. **Click on the Parallel Analysis Tool (PAT) icon on the desktop.**

B. From an external computer:
1. **Use Remote Desktop Connection to log in to athena.cs.weber.edu:53243 using your CS username and password.**
2. **Now use Remote Desktop Connection from athena to log in to hermes.cs.weber.edu or 137.190.19.26 using your CS username and password.**
3. **Click the Parallel Analysis Tool (PAT) icon on the desktop.**

Exercise 2: Java Language Features

Coding Exercise and Recognizing Concurrency

For this exercise, you will submit the completed answer sheet given on the next page and a single screenshot to be described later.

Below is an example of a class that performs numerical integration. This will illustrate the behavior of for-loops, method calls, and some of the arithmetic operators. The program uses the trapezoidal rule to calculate the area under the curve $f(x) = x^2$ from $x = 0$ to $x = 2$. The details of the implemented algorithm are not important right now, but your ability to enter the code using correct syntax and formatting *is* important.

Exercise 2
Answer Sheet

Step I:

Total Area: _____

Running Time: _____

Step II:

	Total Area	*Running Time*	*PQ Value*
P=2			
P=4			
P=8			
P=16			
P=32			

Summary Question: ***Based on the information in the table above, what is the optimum number of processors to use when parallelizing the computation of individual trapezoid areas? _____***

Explain your reasoning.

_____

_____

_____

_____

_____

Step I: Using the ***Parallel Analysis Tool (PAT)***, enter this class into the Program Editor panel exactly as shown below including the comments. Compile and run the program.

```
public class ParallelTest {

   static int n = 1048576; // number of trapezoids
   static double[] results = new double[n];

   public static void main(String args[]) {

     /* Initialize starting Time */
     long t1 = System.currentTimeMillis();

     /* Compute Individual Trapezoid Areas */
     for(int i = 0; i < n; i++) {
       results[i] = computeArea(i);
      }

     double totalArea = 0.0;
     /* Sum up all Areas */
     for(int i = 0; i < n; i++) {
       totalArea += results[i];
      }

     /* Print out the Area */
     System.out.println("Area = " + totalArea);

     /* Display program duration */
     long t2 = System.currentTimeMillis();
     System.out.println("Running Time:" + (t2-t1) + " msecs");
 }

   // Uses modified trapezoid formula
  private static double computeArea(int section)
   {
      int numSegments = 500;
      double height = 2.0/n;
      double leftXValue = section * height;
      double rightXValue = (section + 1) * height;
      double leftSide = (leftXValue * leftXValue)/numSegments;
     double rightSide = (rightXValue * rightXValue)/numSegments;
      double area = 0.0;
      for(int i = 0; i < numSegments; i++) {
           area += 0.5 * (leftSide + rightSide) * height;
     }

      return area;
   }
}
```

Note that the + operator can be used inside the argument to `println()`, to construct a string from several different components at run-time.

Do the following:

→ In the Step I section of the Exercise 2 Answer Sheet, enter the reported values for Area and Running Time. The Running Time value should be labeled appropriately.

Note the different steps in the algorithm as indicated by each of the comments delimited by **/*…*/** in the main method. These annotations give clear descriptions of the individual task performed at each step, and provide a prominent illustration of the importance of code comments in helping other developers understand the purpose and execution of a program. In the PAT, these comments provide activity labels in the associated activity diagram displayed in the right-hand pane.

Now think about which, if any, of these individual tasks could be run more efficiently on a parallel computer. Our goal would be to reduce the time required to execute the program while still generating the correct result for the computed area. We will now consider two of the five specific "commented" steps denoted in the main method.

Step II: Using the ***Parallel Analysis Tool (PAT)***, click on the *Update Activity Diagram* button. The right hand Process View should now reflect the sequential order of the program tasks as indicated by the code comments.

Now we will observe the real-time effect of parallelizing the portion of code that computes the large number of tiny area sections in our integrator. Locate the following comment in the PAT Program Editor:

```
/* Compute Individual Trapezoid Areas */
```

Add the following annotation indicated in bold so that the comment looks exactly as shown here:

```
/* P=2 Compute Individual Trapezoid Areas */
```

The **"P=2"** term above indicates that the collection of calculations associated with the subsequent for-loop will be divided among two processors and these two resulting groups of calculations will be executed in parallel. In general, **"P=n"** means that **n** processors will be used to parallelize the for-loop.

Click the *Compile & Run* button and note the results for Area and Running Time.

Do the following:

→ In the Exercise 2 Answer Sheet, record the reported values for Area and Running Time in the "P=2" row of the Step II section table. Label the Running Time.

Now click the *Perform Parallel Analysis* button and wait for the **"Done!"** message to appear in the Output Window at the bottom of the PAT. Observe the updated activity diagram in the Process View pane. Note the special annotation attached to the incoming transition arrow of the "Compute Individual Trapezoid Areas" activity icon indicating that two processors were used to perform this activity. The "Compute Individual Trapezoid Areas" activity icon should now also contain a ***PQ*** value displayed in blue.

The ***PQ*** value measures the *quality* of parallelizing the targeted section of code. Ideally, the ***PQ*** value should be equal to the number of processors ***n*** used in the parallelization effort, but a ***PQ*** value no less than ***n-1*** is still very good.

Very small, fractional ***PQ*** values indicate the cost of parallelizing the section of code is too high, and thus the use of multiple processors should be avoided. Under these conditions, it actually requires more time to run the targeted section of code in parallel than it would to run the section of code sequentially.

Do the following:

→ In the Exercise 2 Answer Sheet, record the reported PQ value in the "P=2" row of the Step II section table.
→ Repeat the actions described in Step II above to fill in the remaining entries of the table in the Step II section of the Exercise 2 Answer Sheet. To obtain a reliable PQ value, you may need to perform several parallel analyses for a given number of processors to arrive at a stable value.
→ Answer the Summary Question on the Exercise 2 Answer Sheet
→ Submit one screenshot of the PAT console after performing the Parallel Analysis for P=32. Make sure the PQ value for the "Compute Individual Trapezoid Areas" activity can be seen in the Process View Pane.
→ Submit the Exercise 2 Answer Sheet.

Exercise 3: Java Program Structure

Java applications run "standalone:, without a driver program. A Java application consists of one or more classes, and can be as large or as small as needed. A class is frequently created for the sole purpose of encapsulating the `main()` method, necessary for the application to begin. Any class may contain the `main()` method, but there can be only one in a Java application. One common programming technique to handle this is to create a "jumping-off" class that contains the `main()` method, as well as any specialized initialization or termination routines for the application.

The signature for the `main()` method always looks like this:

```
public static void main (String args[]) {  }
```

The `public` keyword means it's available to the Java interpreter; the `static` keyword means that this function is accessible at the class level; the `void` keyword means it doesn't return a value. `args []` is the array of strings passed as arguments to the application [analogous to the `argv []` array in C and C++]. `args []` could be given another name – there is nothing special about "args". However, the first value in the array, `args [0],` is the first argument to the program, NOT the program name, as it would be in C or C++.

Since `args` is an array, it has a `length` member, indicating the number of values passed. You must test the value of `args.length` to see that your application receives an appropriate number of arguments. Java won't do it for you.

Coding Exercise and Recognizing Concurrency

For this exercise, you will submit the completed answer sheet given on the next page and a single screenshot to be described later.

Here we will create a Java program that utilizes some of the arithmetic operators described above in a specific sequence. Since these operations will be applied to variables within our program, we must first perform proper initialization of these variables.

Note that in order to obtain proper analysis from the PAT, the set of initialization statements and the set of arithmetic operation statements are each contained in a single Java ***block***, which is delimited by curly braces.

Exercise 3
Answer Sheet

Step I:

Result: _____

Step II:

Parallel Initialization

Result: _____

PQ Value: _____

Step III:

Parallel Arithmetic Operations

Result: _____

PQ Value: _____

Summary Questions:

Does Parallel Initialization in Step II change the program result? Yes____ No____

Based on the Step II results, would you choose to parallelize variable initialization? Yes____No____

Explain your reasoning.

_____

_____

Do Parallel Arithmetic Operations in Step III change the program result? Yes____ No____

Based on the Step III results, would you choose to parallelize arithmetic operations? Yes____No____

Explain your reasoning.

_____

_____

Step I: Using the ***Parallel Analysis Tool (PAT)***, enter this class into the Program Editor panel exactly as shown below including the comments. Compile and run the program.

```
public class ParallelTest {

   static int zero, one, two, three, four, five, six, seven;
   static int resultOne, resultTwo, resultThree, resultFour;
   static int resultFive, resultSix, resultSeven, resultEight;

   public static void main(String args[]) {

     /* P=1 Initialize Variables */
     {
           zero = 0;
           one = 1;
           two = 2;
           three = 3;
           four = 4;
           five = 5;
           six = 6;
           seven = 7;
     }

     /* P=1 Perform Arithmetic Operations */
     {
           resultOne = zero + one;
           resultTwo = three - two;
           resultThree = three + zero;
           resultFour = (resultThree * resultTwo) - resultOne;
           resultFive = four * five;
           resultSix = six + seven;
           resultSeven = resultFive - resultSix;
           resultEight = resultFour * resultSeven;
     }

     /* Print out the Result */
     System.out.println("Result = " + resultEight);

 }
}
```

Do the following:

→ In the Step I section of the Exercise 3 Answer Sheet, enter the reported Result value.

Step II: Using the ***Parallel Analysis Tool (PAT)***, click on the *Update Activity Diagram* button. The right hand Process View should now reflect the sequential order of the program tasks as indicated by the code comments.

Now we will observe the effect of parallelizing the initialization statements on the generated result.

In the PAT Program Editor, change the number of processors in the "Initialize Variables" code section from 1 to 8. In other words update the "Initialize Variables" comment as follows:

```
/* P=8 Initialize Variables */
```

This change will cause each individual statement in the block immediately following the comment to execute on a separate processor. Since there are 8 statements within the initialization block and we are specifying 8 processors, then each statement will be run simultaneously on a dedicated processor.

Click the *Compile & Run* button and note the result value produced by the program.

Do the following:

→ In the Step II section of the Exercise 3 Answer Sheet, enter the reported Result value.

Now click the *Perform Parallel Analysis* button and wait for the **"Done!"** message to appear in the Output Window at the bottom of the PAT. Observe the updated activity diagram in the Process View pane. The "Initialize Variables" activity icon should now contain a ***PQ*** value displayed in blue.

Do the following:

→ In the Step II section of the Exercise 3 Answer Sheet, enter the reported PQ value.

Step III: Now we will observe the effect of parallelizing the arithmetic operations on the generated result.

In the PAT Program Editor, change the number of processors in the "Perform Arithmetic Operations" code section from 1 to 8. In other words update the "Perform Arithmetic Operations" comment as follows:

```
/* P=8 Perform Arithmetic Operations */
```

As with the initialization statements described in Step II, this change will cause each individual statement in the block immediately following the comment to execute on a separate processor. Since there are 8 statements within the arithmetic operation block

and we are specifying 8 processors, then each statement will be run simultaneously on a dedicated processor.

Click the *Compile & Run* button and note the result value produced by the program.

Do the following:

→ In the Step III section of the Exercise 3 Answer Sheet, enter the reported Result value.

Now click the *Perform Parallel Analysis* button and wait for the **"Done!"** message to appear in the Output Window at the bottom of the PAT. Observe the updated activity diagram in the Process View pane. The "Perform Arithmetic Operations" activity icon should now contain a ***PQ*** value displayed in blue.

Do the following:

→ In the Step III section of the Exercise 3 Answer Sheet, enter the reported PQ value.
→ Answer the Summary Questions on the Exercise 3 Answer Sheet
→ Submit one screenshot of the PAT console after performing Step III above. Make sure the PQ values for both the "Intialize Variables" and the "Perform Arithmetic Operations" activities can be seen in the Process View Pane.
→ Submit the Exercise 3 Answer Sheet.

REFERENCES

[1] Allen, Michael, Wilkinson, Barry, & Alley, James. "Parallel Programming for the Millenium: Integration Throughout the Undergraduate Curriculum." Second Forum on Parallel Computing Curricula, June, 1997.

[2] Amdahl, G. "Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities." Proc. AFIPS Conf. Vol. 30, 1967.

[3] American Educational Research Association, American Psychological Association, and National Council on Measurement in Education. *Standards for Educational and Psychological Testing,* Washington, D.C., 1999.

[4] Bain, Ken. *What the Best College Teachers Do*, Harvard University Press, 2004.

[5] Ben-Ari, M., and Kolikant, Y.B.-D. "Thinking Parallel: the Process of Learning Concurrency," ITiCSE '99, pp. 13-16, New York, 1999.

[6] Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. "Patterns for Parallel Application Programs," Proceedings of the Sixth Pattern Languages of Programs Workshop (PLoP 1999), 1999. http://jerry.cs.uiuc.edu/ plop/plop99/proceedings/.

[7] Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. "Patterns for Finding Concurrency for Parallel Application Programs," Proceedings of the Seventh Pattern Languages of Programs Workshop (PLoP 2000), 2000. http://jerry.cs.uiuc.edu/ plop/plop2k/proceedings/proceedings.html.

[8] Bi, Y., and Beidler, J. "A Visual Tool for Teaching Multithreading in Java." Journal of Computing Sciences in Colleges, 2007.

[9] Bi, Y., and Beidler, J. "Threads Early (Tutorial Proposal)" http://uhaweb.hartford.edu/ccscne/2010Abstracts/3004.pdf

[10] Binstock. Andrew. "Dual-Core Processors Changing Software." Software Development Times, May 15, 2005.

[11] Blundell, Colin, et al. "Making the Fast Case Common and the Uncommon Case Simple in Unbounded Transactional Memory." ACM SIGARCH Computer Architecture News, Vol. 35, Issue 2, May 2007.

[12] Bruce, Kim B., Danyluk, Andrea, and Murtaugh, Thomas. "Introducing Concurrency in CS 1," SIGCSE 2010.

[13] Carr, Steve, et al, "ThreadMentor: A Pedagogical Tool for Multithreaded Programming." ACM Journal of Educational Resources, Vol. 3, No. 1, March, 2003. pp. 1-30.

[14] *Computing Curricula 2001 Computer Science*, Joint Task Force on Computing Curricula, IEEE Computer Society, ACM. http://www.acm.org/education/curric_vols/cc2001.pdf

[15] *Computer Science Curriculum 2008: An Interim Revision of CS 2001*, Interim Review Task Force, IEEE Computer Society, ACM. http://www.acm.org/education/curricula, 2008

[16] Constantinou, Theofanis, et al. "Performance Implications of Single Thread Migration on a Chip Multi-Core." *ACM SIGARCH Computer Architecture News*, pp. 80-91, 2005.

[17] Cunha, Jose C., et al. "An Integrated Course on Parallel and Distributed Processing." SIGCSE 1998.

[18] Dahl, O.J, Dijkstra, E.W., and Hoare, C.A.R. *Structured Programming*, Academic Press, New York, 1972.

[19] Darringer, John A. "Multi-Core Design Automation Challenges." *Proceedings of the annual conference on Design Automation*, pp. 760-764, 2007.

[20] Davis, Al. "Top Ten List of Multi-Core Problems. Slides by permission. *Multi-Core Discussion Colloquium*, University of Utah, November, 2007.

[21] Dongarra, J. Ed. *Sourcebook of Parallel Computing*, Morgan Kaufmann, 2003.

[22] Drew, Clifford J., Hardman, Michael L., and Hosp, John L. *Designing and Conducting Research in Education*, Sage Publications Inc., 2007

[23] Ernst, D.J., and Stevenson, D.E. "Concurrent CS: Preparing Students for a Multicore World," ITiCSE '08, pp. 230-234, New York, 2008.

[24] Feng, Wu-chun, and Balaji, Pavan. "Tools and Environments for Multicore and Many-Core Architectures." *Computer*, pp. 26-27, December, 2009.

[25] Flynn, Michael. "Very High-Speed Computing Systems." Proceedings of the IEEE 54:1901-1909, December 1966.

[26] Fowler, Martin. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, Addison-Wesley Professional, 2003.

[27] Gustafson, J. "Reevaluating Amdahl's Law," Communications of the ACM, 31(5):532-533, May 1988.

[28] Haladyna, T.M. *Developing and Validating Multiple-Choice Test Items.* Lawrence Erlbaum Associates, Inc., 3rd ed., 2004.

[29] Halloun, I., and Hestenes, D. "The Initial Knowledge State of College Physics Students." American Journal of Physics, 53 (11), 1043-1055, 1985.

[30] Halloun. I., and Hestenes, D. "Common Sense Concepts about Motion." American Journal of Physics, 53 (11) 1056-1065, 1985

[31] Hennessy, John L., and Patterson, David A. *Computer Architecture: A Quantitative Approach*, 3rd ed. Morgan Kaufmann, 2003.

[32] Herman, Geoffrey L., Loui, Michael C., and Zilles,Craig "Creating the Digital Logic Concept Inventory", SIGCSE 2010.

[33] Hestenes, D., Wells, M., and Swackhamer, G. "Force Concept Inventory." *The Physics Teacher*, 30(3), 141-151, 1992

[34] Hughes, C., and Hughes, T. *Parallel and Distributed Programming Using C++*, Addision-Wesley, 2004

[35] Janssen, Curtis L., and Nielsen, Ida M.B. *Parallel Computing in Quantum Chemistry*, CRC Press, 2008.

[36] Kaczmarczyk, Lisa C., et al. "Identifying Student Misconceptions of Programming," SIGCSE 2010.

[37] Kaminsky, Alan. *Building Parallel Programs*, Course Technology, 2010.

[38] Karniadakis, George Em, Kirby, Robert M. *Parallel Scientific Computing in C++ and MPI* Cambridge University Press, 2003.

[39] Kramer, J. "Is Abstraction the Key to Computing?"CACM, Vol 50, No 4, 2007

[40] Kozielska, M. "Educational Computer Programs in Learning of Physics by Action", Education Media In-ternational, 37(3), pp 161-166, 2000

[41] Kumar, Rakesh et al. "Multi-Core Design I: Core Architecture Optimization for Heterogeneous Chip Multiprocessors." *Proceedings of the International conference on Parallel architectures and compilation*, pp. 23-32, 2006.

[42] Kurtz, Barry L., et al. "Parallel Computing in the Undergraduate Curriculum." SIGCSE 1998.

[43] Kurtz, Barry L., et al. "A Concurrency Simulator Designed for Sophomore-level Instruction." SIGCSE 1998.

[44] Lanier, Jaron. *You are not a Gadget*. Alfred A. Knopf, New York, 2010.

[45] Lee, Edward A., "The Problem with Threads." Computer, May 2006.

[46] Mandelbrot, Benoit. *The Fractal Geometry of Nature.* W.H. Freeman and Company, 1977.

[47] Mattson, T.G., Sanders, B.A., and Massingill, B.L. *Patterns of Parallel Programming*. Addison-Wesley, 2005

[48] Mbarika, Victor W. A. "Using a Multimedia Case Study Approach to Communicate Information Technology Concepts at the Graduate Level The Impact of Learning Driven Constructs." Journal of SMET Education, 4(1 & 2), 28-36, 2003

[49] McCloskey, M., et al. "Curvilinear Motion in the Absence of External Forces: Naive Beliefs about the Motion of Objects." Science 210(4474): 1139-1141, 1980.

[50] McMaster, K., Rague, B., Anderson, N. "Integrating Mathematical Thinking, Abstract Thinking, and Computational Thinking," Frontiers in Education (FIE) Conference, Arlington, VA, 2010.

[51] McMaster, K., Rague, B., McMaster, T., and Blake, A. "Two Gestalts for Mathematics: Logical vs. Computational." Information Systems Education Journal, 6 (20). http://isedj.org/6/20/. ISSN: 1545-679X. (Meritorious Paper), 2008.

[52] McMaster, K., Rague, B., Hadfield, S., "Two Frameworks for Discrete Mathematics", Information Systems Education Journal, 7 (68) http://isedj.org/7/68/, July, 2009.

[53] Moreno, R., and Mayer, R.E. "Learning Science in Virtual Reality Multimedia Environments: Role of Methods and Media." Journal of Educa-tional Psychology, 94(3), pp. 598-610, 2002.

[54] Neeman, Henry, et al,"Analogies for Teaching Parallel Computing to Inexperienced Programmers" SIGCSE Bulletin, Vol. 38, No. 4, December, 2006.

[55] Pacheco, P. S. *Parallel Programming with MPI*, Morgan Kaufmann, 1997

[56] Patterson. David. "We Don't Know what we should be Teaching." Software Development Times, May 1, 2008.

[57] Quinn, Michael J. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, New York, 2004.

[58] Rague, B. "Teaching Parallel Thinking to the Next Generation of Programmers." Proceedings 6th International Conference on Education and Information Systems, Technologies and Applications: EISTA. Orlando, FL, 2008.

[59] Rubinstein, Joshua S., Meyer, David E., and Evans, Jeffrey E."Executive Control of Cognitive Processes in Task Switching." Journal of Experimental Psychology: Human Perception and Performance. Vol. 27, No. 4, pp 763-797, 2001.

[60] Sanders,K.E., and van Dam, A. *Object-Oriented Programming in Java: A Graphical Approach*, Addison-Wesley, 2006.

[61] Sheskin, David. J. *Handbook of Parametric and Nonparametric Statistical Procedures*, CRC Press, New York, 1997.

[62] Shriraman, Arrvindh, et al, "An Integrated Hardware-Software Approach to Flexible Transactional Memory." Proceedings of the International Symposium on Computer Architecture, pp. 104-115, 2007.

[63] Sincich, T., Levine, D.M., and Stephan, D. *Practical Statistics by Example*. Prentice Hall, New Jersey, 2002.

[64] Sivasubramaniam, A., et al. "An Approach to Scalability Study of Shared Memory Parallel Systems." Technical Report GIT-CC-93/62, 1993.

[65] Stasko, J.T. "The PARADE Environment for Visualizing Parallel Program Executions: a Progress Report." Technical Report GIT-GVU-95-03, 1995.

[66] Stein, L.A. "What we've Swept under the Rug: Radically Rethinking CS1," *Computer Science Education*, 8(2):118-129, 1998.

[67] Sun Microsystems. "Multithreading in the Solaris Operating Environment." White paper, 2002.

[68] Tabachnick, B.G, Fidell L.S. *Using Multivariate Statistics*. 4th ed., Allyn & Bacon, 2000.

[69] Tew, Allison E., and Guzdial, M. "Developing a Validated Assessment of Fundamental CS1 Concepts", SIGCSE 2010.

[70] Tullsen, Dean, et al, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," Proceedings of the Annual International Symposium on Computer Architecture, June, 1995.

[71] Wilkinson, B., and Allen, M. *Parallel Programming*, 2nd ed., Pearson Prentice-Hall, 2005