Teaching Parallel Thinking to the Next Generation of Programmers

Brian Ague
Computer Science, Weber State University
Ogden, UT 84408, USA

ABSTRACT

Most collegiate programs in Computer Science offer an introductory course in programming, primarily devoted to communicating the foundational principles of software design and development. 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 multi-core 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. This paper presents a classroom strategy for introducing the core skills that reinforce parallel thinking. A computerized visualization tool is described that immediately connects the student with the unique challenges associated with parallel program conceptualization. This instructional approach has the potential to make parallel code design principles available to students at the very beginning of their Computer Science education.

Keywords: parallel, programming, computer science, computational, instruction.

1. INTRODUCTION

Integrated circuit manufacturers like Intel, AMD, and Sun are currently producing multi-core 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 multi-core computer architectures continues to move forward, though many challenging issues still remain [5][8][11]. Davis [7] constructed a top ten list of the key problems confronting multi-core development; programming methodology was listed as the number one issue.

The role of the application developer in the context of the multi-core environment is far from settled, but two 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. [9]

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

For some perspective on point (2) above, Pacheco [17] noted in his book published in 1997 that "most colleges and universities introduce parallel programming in upper-division classes on computer architectures or algorithms." Yet Wilkinson and Allen [24] asserted that selected topics from the first part of their 2005 textbook helped to introduce their "first-year students to parallel programming."

Indeed, concerted efforts have been made the last ten years to introduce parallel computing issues earlier in the undergraduate CS curriculum [1][6][12][16]. 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 mechanisms. During this class, important foundational computing issues such as modularization and basic code structure can be addressed from a high conceptual viewpoint.

This 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.

This paper presents a CS1 classroom strategy for introducing the core skills that reinforce parallel thinking, primarily based on a computerized visualization tool that immediately connects the student with the unique challenges associated with parallel program conceptualization.

2. CS1 APPROPRIATE PERFORMANCE ANALYSIS

Various measures can be used to evaluate parallel programs[18][24]. 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 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. Although speedup is a useful measure which encourages the student programmer to increase the percentage of 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. 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[20] 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 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):

$$PQ(p,n) = \frac{t_{comp}}{t_{comm}} \qquad (1)$$

Note that both PQ and the speedup factor are both functions of p and n. The execution 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 is 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.

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.

3. PARALLEL ANALYSIS TOOL (PAT)

Several effective visualization tools geared towards streamlining the development of parallel programs have been designed, such as those described by Stasko[22], Kurtz[13] and Carr[4]. Carr states, "There are few pedagogical tools for teaching threaded programming." Indeed, threads currently represent the de facto primitive for programmers delving into parallel application code.

However, 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 [21] and more efficient execution techniques in processor design, most notably extending superscalar performance into fine-grained temporal multithreading and simultaneous multithreading[23].

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 multi-threaded program execution and the various synchronization objects and monitors may be of great benefit to the Senior-level CS stu-

dent, there is a finite likelihood that first year CS student programmers would become overwhelmed and possibly discouraged by this level of detail. Lee[14] has stated outright that the thread model may not be best for parallel application design. Currently, there is also an active investigation into transactional memory (TM), which aims to make the programming model simpler, freeing the developer from lock management tasks[2][19]. The recent Rock processor from Sun offers TM support.

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 effectively utilized by the Parallel Analysis Tool (PAT) proposed here. The PAT provides the student a visualization perspective motivated by the UML Activity diagrams typically employed in software engineering.

A good, concise treatment of these diagrams is given in Fowler[10]. Essentially, the Activity diagram provides a "flow-chart" view of a computation 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, 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.

A simple example that illustrates how parallel actions can be expressed in Activity diagrams is given in Figure 1.

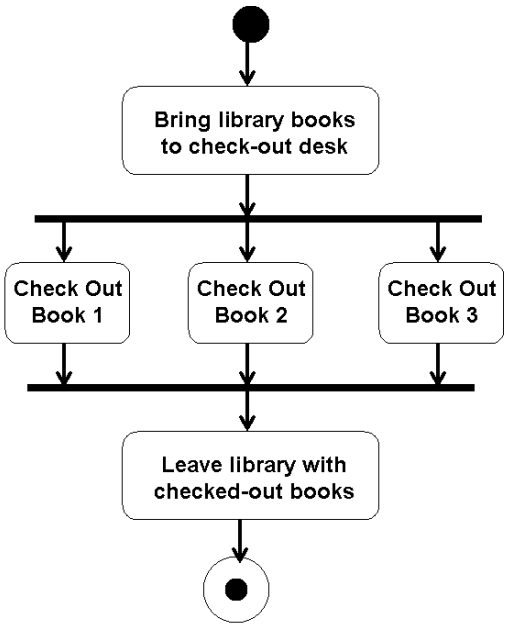

Figure 1: Library Book Check-Out

The "Check Out Book p" activities positioned between the fork and join represent parallel execution of a similar action. These may be more economically expressed as shown in Figure 2.

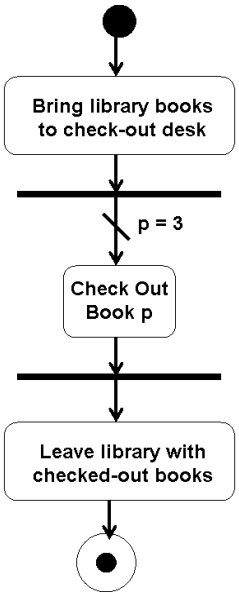

Figure 2: Library Book Check-Out (Revised)

CS1 programming students will be confronted with more code-specific examples, such as the following loop parallelization:

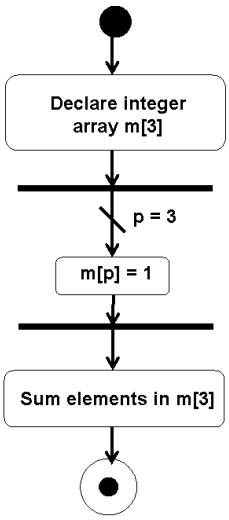

Figure 3: Initializing and Summing Array Elements

The array initialization is an inherently parallel operation typically performed iteratively. The code corresponding to Figure 3 is given here:

```
int m[3], i, sum;

/* Initialization Loop */
for ( i = 0; i < 3; i++)
    m[i] = 1;

/* Summing Loop */
sum = 0;
for( i = 0; i < 3; i++)
    sum += m[i];
```

The initialization loop lends itself to concurrent behavior because of the embarrassingly parallel nature of the operation in the body of the loop. The summing loop presents a possible contention issue since a common resource, the memory location represented by `sum`, is shared by each execution of the loop operation.

PAT Basic Operation

When the PAT is started, the student is presented with a main window divided vertically into two panels: the left panel is the Program Editor, and the right panel is the Process View. At the very top of the PAT is a value indicating how many processors are available on the current machine.

The student can enter information into either panel. The Program Editor handles source code, and the Process View displays the corresponding activity flow. Figure 4 shows the user interface on PAT initialization.

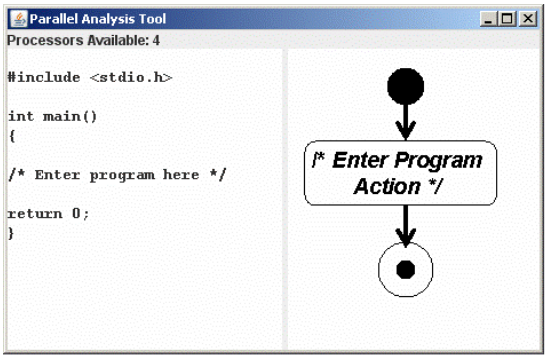

Figure 4: PAT Start-Up Screen

The student can enter relevant descriptions and annotations into activity icons in the Process View. As the program grows, the student will be able to generate additional activity icons to help conceptually modularize code behavior, as shown in Figure 5.

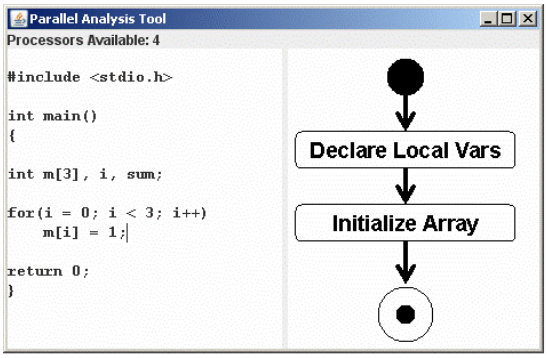

Figure 5: Modularizing Activities

The student can initiate parallel analysis by selecting portions of code that the student identifies as good candidates for parallelization. These choices rely solely on the student's knowledge and intuition about parallel design, gathered from prior presentations and discussions of the topic by the instructor.

Once a code portion is selected, and parallel analysis has been activated, the process view will be updated to show the results of the analysis. For clarity, Figure 6 shows an expanded portion of the interface.

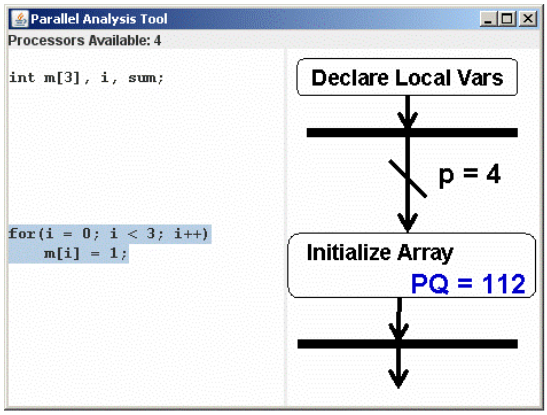

Figure 6: Activating Parallel Analysis

As shown in the figure, the PQ number will be displayed in the corresponding activity icon as a measure of the impact of parallelization on the performance of the code. Students will have the option of determining how many processors p are used to compute the PQ number. This control will allow the student to observe the influence of p on communication time, and illustrate the possibility that program performance may actually *decrease* as p increases.

In the design of the PAT, a special emphasis is placed on interface simplicity, thereby allowing the student programmer to focus on the broad results of parallelizing code. In addition, the feedback afforded by the PAT will aid the student in recognizing code sections that will benefit from parallelization.

4. SUMMARY

The educational needs of beginning programmers are influenced by current technological developments. The wide availability of multi-core devices suggests that CS students develop an early appreciation of parallel programming design. To avoid the danger of overwhelming beginning programmers with details of thread mechanics, this paper presents a metric and visualization tool that focuses on honing the core recognition and analysis skills that promote parallel thinking.

In future research, a formal experimental educational study of first year CS students will be performed to determine the effectiveness of the tool in teaching parallel design.

5. 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] 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.

[3] Bryant, Randall E., and O'Hallaron, David R., **Computer Systems – A Programmer's Perspective**. Prentice Hall, New Jersey, 2003.

[4] 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.

[5] Constantinou, Theofanis, et al. "Performance implications of single thread migration on a chip multi-core." **ACM SIGARCH Computer Architecture News**, pp. 80-91, 2005.

[6] Cunha, Jose C., et al. "An Integrated Course on Parallel and Distributed Processing." **SIGCSE 1998.**

[7] Davis, Al. "Top Ten List of Multi-Core Problems." Slides by permission. **Multi-Core Discussion Colloquium**. University of Utah, November, 2007.

[8] Darringer, John A., "Multi-core design automation challenges." **Proceedings of the annual conference on Design Automation**, pp. 760-764, 2007.

[9] Dongarra, Jack, Ed. **Sourcebook of Parallel Computing**. Morgan Kaufmann, San Francisco, CA, 2003.

[10] Fowler, Martin. **UML Distilled** 3rd Ed. Addison-Wesley, 2004.

[11] 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.

[12] Kurtz, Barry L., et al. "Parallel Computing in the Undergraduate Curriculum." **SIGCSE 1998.**

[13] Kurtz, Barry L., et al. "A Concurrency Simulator Designed for Sophomore-level Instruction." **SIGCSE 1998**.

[14] Lee, Edward A., **The Problem with Threads**. Computer, May 2006.

[15] Leonard, Patrick, "Concurrent Computing for C++ Applications - Achieving scalability on multi-core processors." White paper, Rogue Wave Software, 2007.

[16] Neeman, Henry, et al, "Analogies for Teaching Parallel Computing to Inexperienced Programmers" **SIGCSE Bulletin**, Vol. 38, No. 4, December, 2006.

[17] Pacheco, Peter S., **Parallel Programming with MPI**. Morgan Kaufmann, San Francisco, CA, 1997.

[18] Quinn, Michael J., **Parallel Programming in C with MPI and OpenMP**. McGraw-Hill, New York, 2004.

[19] 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.

[20] Sivasubramaniam, Anand, et al. "An Approach to Scalability Study of Shared Memory Parallel Systems." **Technical Report GIT-CC-93/62**, October, 1993.

[21] Sun Microsystems. "Multithreading in the Solaris Operating Environment." White paper, 2002.

[22] Stasko, J.T. "The PARADE environment for visualizing parallel program executions: a progress report." **Technical Report GIT-GVU-95-03**, 1995.

[23] Tullsen, Dean, et al, "Simultaneous Multithreading: Maximizing On-Chip Parallelism" **Proceedings of the Annual International Symposium on Computer Architecture**, June, 1995.

[24] Wilkinson, Barry, & Allen, Michael, **Parallel Programming** 2nd ed.. Pearson Prentice Hall, New Jersey, 2005.