Is Teaching Parallel Algorithmic Thinking to High School Students Possible? One Teacher's Experience

Shane Torbert
Thomas Jefferson High School
for Science and Technology
Fairfax County Public Schools
Fairfax County, VA
smtorbert@fcps.edu

Uzi Vishkin
The University of Maryland
Institute for Advanced
Computer Studies (UMIACS)
College Park, MD 20742
vishkin@umd.edu

Ron Tzur
University of Colorado Denver
ron.tzur@ucdenver.edu

David J. Ellison
Indiana University
Bloomington
djelliso@indiana.edu

ABSTRACT

All students at our high school are required to take at least one course in Computer Science prior to their junior year. They are also required to complete a year-long senior project associated with a specific in-house laboratory, one of which is the Computer Systems Lab. To prepare students for this experience the lab offers elective courses at the post-AP Computer Science level. Since the early 1990s one of these electives has focused on parallel computing. The course enrolls approximately 40 students each year for two semesters of instruction. The lead programming language is C and topics include a wide array of industry-standard and experimental tools. Since the 2007-2008 school year we have included a unit on parallel algorithmic thinking (PAT) using the Explicit Multi-Threading (XMT) system [11, 12]. We describe our experiences using this system after self-studying the approach from a publicly available tutorial. Overall, this article provides significant evidence regarding the unique teachability of the XMT PAT approach, and advocates using it broadly in Computer Science education.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—*computer science education*

General Terms

Design, Human Factors, Algorithms

Keywords

Parallel Algorithmic Thinking, High School, XMT, PRAM Algorithms

SIGCSE'10, March 10–13, 2010, Milwaukee, Wisconsin, USA.

1. INTRODUCTION

Thomas Jefferson High School for Science and Technology is a member of the National Consortium for Specialized Secondary Schools of Mathematics, Science, and Technology. Admission is competitive and students are drawn from a region whose total population is over one million people. Through a national contest in the early 1990s the Computer Systems Lab won an ETA supercomputer and subsequently created a one-semester elective course in Supercomputing Applications.

The original ETA was damaged by a leaky roof and later replaced with a cluster of Linux workstations running PVM and MPI. In 2003 the ETA was exchanged for a Cray SV1 which is still housed in the lab, but the main environment for computing continues to be approximately 50 mixed-use Linux workstations, a few dedicated high-performance Linux and Solaris servers, and a recently acquired eight-node cluster. System administration and maintentance are handled by a small group of the high school students themselves.

The recognition that parallel programming is becoming mainstream Computer Science knowledge is supported by texts such as [4] that wrote: "for the first time in history, no one is building a much faster sequential processor. If you want your program to run significantly faster..., you're going to have to parallelize your program". In 2007 this evolution of the field led us to expand the offering from one semester to two semesters and change the name of the course to Parallel Computing.

2. STUDENT POPULATION

The students in our course have either already completed or are currently enrolled in AP Computer Science and many have also taken two-semesters of another elective in Artificial Intelligence. Thus they typically come into the Parallel Computing course with two years experience in Java and one year in Python, as well as exposure to Big-Oh analysis and data structures including queues, heaps, trees, and graphs. Mathematically they are at least through Algebra II and many are in Calculus, so they are familiar with matrices, exponentials, and logarithms. They are highly motivated in this area and many will eventually become undergraduate Computer Science majors.

3. GOALS OF THE COURSE

Essentially all aspects of the course have changed from the original conception and continue to change as needed.

3.1 Language and Tools

At various times the course has been taught in C, C++, and Fortran. We currently teach the course in C and one major goal is to expose students to aspects of C (e.g., pointers) that they would not have seen in either Java or Python. This choice also facilitates discussions of system-level architecture which play a minor but important role in the course.

We have long abandoned PVM but still spend a semester covering MPI at the level of send/receives, focusing first on the Manager-Worker paradigm for the embarrassingly parallel problems and then later on more complicated communication schemes. In the second semester we cover XMT, pthreads, OpenMP, sockets, and Nvidia's CUDA [2].

3.2 Principles

We begin with embarrassingly parallel problems such as parameter search, fractal generation, and cellular automata, using 2-D OpenGL graphics (students have previously used Java's Swing) to visualize speedup when possible. Then we move to coupled problems such as heat transfer and orbital mechanics, before returning again to such problems as iterative matrix solvers, image compression, and ray tracing.

We want students to appreciate the breadth of parallel computing's current landscape while at the same time see some depth in the context of classic problems, their solutions in parallel, and a generalizable approach to writing parallel code. Formal analysis, which has long been an integral part of our serial algorithmic thinking courses, has only been included in the parallel course since our introduction of XMT.

4. EXPLICIT MULTI-THREADING

The parallel random-access machine/model (PRAM) theory of algorithms [6, 7], developed mostly in the 1980s, provides a well-established, easy approach to parallel algorithmic thinking (PAT). The Explicit Multi-Threading (XMT) system from the University of Maryland was designed to implement PRAM-like programming. As such, XMT provides students a simple-to-use alternative to MPI. As has been noted by others [3, 5, 8, 9, 10] the complexity of coding in MPI or even OpenMP can be quite overwhelming for a beginning student. With XMT the coding overhead is instead very light.

The programming language for XMT, called XMT-C, adds only two constructs to ANSI C: `spawn` and prefix-sum. So far we have only used the spawn construct, within which the dollar-sign ($) variable acts like an MPI rank or a thread ID, and shared access to previously declared arrays makes send/receives of data unnecessary. Installation of the emulator [1] is easy and provides cycle and time estimates for comparison calculations. Most importantly from an instructional perspective, XMT has allowed us to cover a range of algorithms that we had never considered covering using MPI.

5. ALGORITHM ANALYSIS

We make extensive use of timing data to show speedups for embarrassingly parallel problems but had not ventured into formal analysis of algorithms until we began using XMT. The main reason for parallel algorithms is performance and it was important to be able to match their performance analysis with that of serial algorithms.

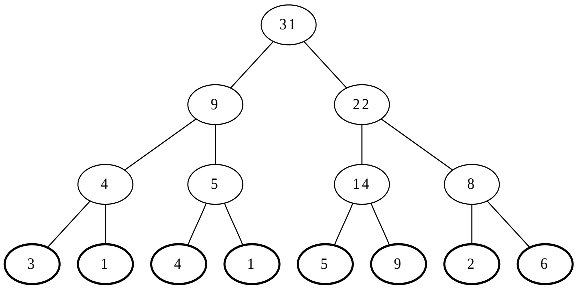

Figure 1: Tree representation of a simple sum.

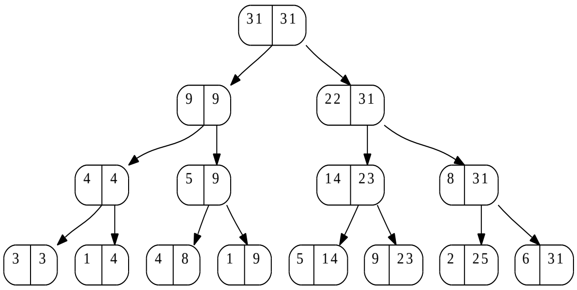

Figure 2: The widely used prefix-sum calculation.

5.1 Summation Example

We began our XMT unit with a simple example: calculating the sum of an array of integers. A serial version can be constructed easily and runs in $O(N)$ time. In parallel we take advantage of a tree structure as shown in Figure 1. The operations at each level of the tree are independent and can be done in parallel using a spawn block in $O(1)$ time. If we then run a loop over the levels we will arrive at the top node of the tree in $O(\log N)$ time. Note that the amount of work (i.e., total number of operations) is still $O(N)$ but the amount of time has been reduced drastically. A code listing is provided in the Appendix.

5.2 Prefix-Sum

We do not have the students code the simple summation example. Instead we use it as a basis for discussing the prefix-sum problem whose main idea will be used for many other algorithms. For a given position in the array the prefix-sum is the sum of all the values up to that point. Having completed a bottom-to-top pass of the tree to calculate the overall sum we then complete a top-to-bottom pass to calculate all the prefix-sums, as shown in Figure 2. Again the operations at each level can be done independently in parallel. We walked through the parallel algorithm with the students, modified the summation pseudocode to calculate these prefix-sums, and the students then wrote their own XMT-C code. This served as a first check for us as to whether or not the system was working properly and also that the students understood the basic style of approach we planned to follow.

5.3 Matrix Multiplication

At this point we were able to have a very powerful discussion about matrix multiplication. How can we multiply two $N \times N$ matrices in parallel? We began from the perspective of the N^2 elements in the resulting matrix. Each of these elements is calculated by taking the dot product of a given row and column from the input matrices. Since each of these dot products is independent they can all be done in parallel. In turn, each dot product consists of N multiplications and a summation. The multiplications are independent and each summation, as we have seen, can be done in $O(\log N)$ time and independently of the other summations. Thus the entire matrix multiplication takes the normal $O\left(N^3\right)$ work but an astonishingly low $O(\log N)$ time. The "wow" factor resulting from this discussion cannot be overstated.

5.4 Other Examples

The students next solved a series of problems related to prefix-sum. We did not reveal any solutions until the students had a chance to attempt the problems themselves, which included prefix-min, compaction, nearest-one, and segmented prefix-sum. Even without fully knowing where all of this was leading the students became very engaged in these problems. One student even coined the term "micro-parallelism" to describe the radically different approach we were taking with XMT as compared to MPI. Not only were they successful at solving the problems but the students also realized immediately that this bottom-up approach to building algorithms in parallel was of a fundamentally different nature from what they had previously seen.

5.5 Communication Schemes

No one wanted to try coding these algorithms in MPI with send/receives and the careful up-and-down tree-based communication that would be needed to pass data, manage tasks, and avoid deadlock. Since communication is a major difficulty in writing MPI code we wait as long as possible to attempt even simple schemes like nearest-neighbor and round-robin. The ease of implementing tree communication is a major difference between XMT and MPI.

5.6 Student Work

Not only was the language overhead far less with XMT than it had been with MPI but it was small enough that upon returning from a week-long winter break students required essentially no review before resuming productive work. In addition, the algorithmic problems considered sparked real creativity. It was no longer the case that everyone in the lab was chasing the same canonical solution but instead students were actually inventing different methods for solving these problems.

6. RANKING AND MERGING

6.1 Big-Oh Analysis

Once the students had gained some experience coding in XMT we attempted a much more ambitious problem but one whose big-picture usefulness they could immediately grasp. When the students were introduced to Big-Oh analysis in the AP Computer Science course, the first application they saw was sorting. That course considers the selection, insertion, merge, quick, and heap sorts. This made writing a mergesort for XMT a natural fit and a powerful example of how to evaluate the quality of a parallel algorithm. Our goal was to find an algorithm that balances a small amount of work with a small amount of time.

Table 1: Big-Oh comparison for rank-merge.

	Serial	Binary Search	Partitioning
Work	$O(N)$	$O(N \log N)$	$O(N)$
Time	$O(N)$	$O(\log N)$	$O(\log N)$

6.2 Statement of the Problem

The problem was presented as follows: given two sorted arrays merge them into one sorted array. Due to their prior experience learning the mergesort the students knew that a parallel solution to this problem would lead to a technique for sorting in parallel. We discussed the idea of determining for each element from one list where it would fall in the other list, and that once we knew where it fell in both lists (we already knew where it fell in its own list) we could calculate its ultimate location in the sorted list. We could also move everything into place with a single spawn block in $O(N)$ work and only $O(1)$ time. This concept, known as ranking, tells us how many elements are smaller than we are, which in turn tells us our index in the merged array.

6.3 A First Attempt

We can calculate the rank of each element using a binary search on the other list. This takes $O(\log N)$ work for each of N elements for a total of $O(N \log N)$ work. The binary searches are all independent so they require only $O(\log N)$ time. Already students were impressed by this result because if a mergesort requires $O(\log N)$ levels of recursion then we have just described a parallel sort that runs in a mere $O\left(\log^2 N\right)$ time! But we can still improve the amount of work being done; a serial merge takes only $O(N)$ work which our first parallel attempt has increased by a factor of $\log N$.

6.4 Big Idea: Partitioning

We consider breaking the rank-merge problem into pieces in order to reduce the asymptotic behavior of the work without changing the time. We partition each of the sorted input arrays into $O(N/\log N)$ pieces of size $O(\log N)$ and perform the binary search ranking process on only the endpoints of these partitions. This still takes $O(\log N)$ time but now only $O(\log N \cdot (N/\log N)) = O(N)$ work. To rank the remaining elements we perform $O(N/\log N)$ serial-style merges, all of which are independent and can be done in parallel.

Each serial-merge is linear in both work and time. That is, the serial-merge is linear in terms of the amount of data being merged by this particular instance of the serial-merge code. We are not merging all the data at once but rather in small pieces, the partitions. Since the partitions have been carefully constructed so that their sizes are $O(\log N)$ this amounts to only $O(\log N)$ work and time for each merge instance.

This critical point must be explained with great care or only the best students will catch it. Then, multiplying by the $O(N/\log N)$ number of partitions gives us a total of $O(N)$ work. Since we're still only using $O(\log N)$ time this means the parallel mergesort will be as good as serial in terms of work, $O(N \log N)$, for only $O\left(\log^2 N\right)$ time.

6.5 Comparison of Algorithms

A summary of these three approaches is shown in Table 1. This kind of development of a non-trivial algorithm, drawing from previous experience and refining with subtle ideas, was in no way a part of our program prior to using XMT. Students continue to be impressed by the timing data speedups of a big MPI run but now we can also appeal to their more sophisticated abilities of anaylsis, to extend those skills in a parallel environment, and to build general purpose techniques in a highly-coupled context.

7. TEACHER PREPARATION

Prior to the emulator being available for download a pilot was conducted during the spring semester of 2008 as a proof-of-concept. After XMT proved to be a viable tool the instructor self-taught himself the material only by reviewing a series of online video lectures in preparation for the 2008-2009 year. He was in regular contact with the XMT team before, during, and after the five-week instructional unit.

8. CONCLUSIONS

The students in this course would be classified as very good in Computer Science even if they weren't studying parallel computing. Our challenge is to present them a broad array of meaningful, inspiring, real-world, eye-opening experiences. Our use of XMT as described here has provided them with a level of insight that MPI just couldn't do given their backgrounds and the instructional timeframe we have to work with. It has earned itself a permanent place in our curriculum and we look forward to improving our presentation of this material each year. Specifically, we plan on extending our instructional unit to include the selection problem (that will require use of the prefix-sum construct) and the general technique of accelerating cascades. For the 2009-2010 school year this will begin the first week of February at the start of the second semester and continue until the first week of March, covering five weeks of instruction.

Compared to OpenMP, which requires a laundry list of preprocessor directives that are also hardware specific, we found the XMT interface was easier for students to pick up and use. Compared to CUDA, we found the additional syntax features to be far more intuitive in XMT with the added benefit that no specialized graphics cards had to be purchased, installed, and configured in order to run student programs.

As a high school we have the luxury of assuming that an undergraduate education will follow anything we do with our students. Thus, our end-product goal is not a ready professional heading into the job market but rather a motivated, activated, and stimulated teenager entering a university setting prepared for many more years of learning. While our presentations do not always contain the same level of abstraction that might be found in a college-level lecture, wherever possible we want to challenge our students with the most rigorous problems they can reasonably be expected to solve. The XMT system has allowed us to expand these offerings with both a new style of question and a user-friendly environment for writing parallel code.

Overall, this article provides significant evidence regarding the unique teachability of the XMT PAT approach, and advocates using it broadly in Computer Science education.

9. POSTSCRIPT

In response to SIGCSE reviews, we describe also experiences and strategies teaching more typical groups of K-12 and a freshman course to college students.

9.1 K-12 Settings

D. Ellison, a K-12 mathematics teacher, taught more typical students with minimal or no programming experience at:

1. Baltimore Polytechnic Institute, a majority African-American high school. The class met bi-weekly over two months and was offered to 11th grade students who happened to take AP Chemistry as an alternative to their Lego Mindstorms robot programming class.

2. Montgomery Co. Public Schools, middle-school summer camp for underrepresented students. The class met nine mornings from 8:30 AM until 12:30 PM.

9.2 Pedagogy

Consistent with a constructivist theory of learning and reform education methods at this elementary level we posed problem solving tasks designed to prompt student construction of algorithms, generally in small groups of 3-4. During class discussions we considered students' proposed algorithms especially in light of the following three questions designed to promote an understanding of parallelism:

- How can I introduce parallelism into my algorithm?
- How can I measure the benefits of parallelism?
- How can I justify when parallelization is exhausted?

Quantifying operations in algorithms proved useful as we guided students to an understanding of a PRAM computer model and more general notions of work and depth. Again we posed questions:

- What is this serial algorithm doing for us? What is enabling it in the computer?
- What is this parallel algorithm doing for us? What is enabling it in the computer?

9.3 Findings

Under these methods we find typical students, even at the middle school level, are capable of viewing the computer as PRAM and are able to gain deeper understandings of their algorithms including time complexities. We found students' initial struggling with C coding can be rather tedious. After some problem solving activities described in detail below we supplied them with nearly completed XMT-C code to speed things along. With help from the XMT team at the University of Maryland we produced elegant and understandable code representing both serial and parallel algorithms.

The next phase of the curricula included compiling and executing the XMT-C code during class. We encouraged students to compare serial and parallel time cycles and number of operations by running progressively larger data sets. Students analyzed their results via spreadsheets. The purpose of examining the data in this way was to help students quantify the benefit of the parallelism over serial, to form and test their conjectures regarding the PRAM model, to test the limit of parallelism's benefit, and to help them develop a sense of the time complexity of their algorithms. These activities strengthened the impact of previous discussions.

9.4 Activities

From our presentation at the 2009 CS4HS workshop [11]:

1. Introduction to algorithms (via problem solving activities from the CSTA Curriculum Statement, 2007)
2. MS LOGO activities, especially at the middle school level as a warm-up to coding algorithms in XMT-C
3. The Exchange Problem (introduces the use of variables and exchanging array values in serial and parallel)
4. The Bill Gates Problem (Mr. Gates completes morning activities illustrating that independent tasks can be conducted simultaneously under parallelism)
5. Vector + Vector addition (naive parallelism compared to serial for constant time operations)
6. Parallel addition (the binary tree technique supported students understanding of algorithms and strengthened their mathematical notions of the binary exponential and its logarithm)
7. Matrix Multiplication (and exploring the potential of log time using nested spawn commands)
8. Merge Ranking (middle school students actually formed themselves into two lines and performed the ranking)

9.5 College Freshman Setting

U. Vishkin taught an elective freshman course at the University of Maryland in Spring 2009. Most of the 19 students were not Computer Science or Computer Engineering majors. Programming assignments included parallel algorithms for radix sort, finding the median, sample sort (an extension of Quicksort), and merge sort. Similar assignments would be acceptable in a serial programming course for freshmen. Pedagogy: Revisiting the performance (complexity) analysis of an assignment after completion was helpful towards the next assignment. Findings: We found that nearly all students were able to produce correct working code that achieved satisfactory speed-ups for each of the assignments.

10. ACKNOWLEDGMENTS

Help by the XMT team, and support by National Science Foundation grants 0325393, 0811504, and 0834373, are gratefully acknowledged.

11. REFERENCES

[1] Emulator for xmt-c. sourceforge.net/projects/xmtc.

[2] Toolkit for cuda. nvidia.com/cuda.

[3] D. Ernst, B. Wittman, B. Harvey, T. Murphy, and M. Wrinn. Preparing students for ubiquitous parallelism. *SIGCSE '09: Proceedings of the Fortieth SIGCSE Technical Symposium on Computer Science Education*, pages 136–137, 2009.

[4] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A quantitative approach, 4th edition.* Morgan Kauffman, 2007.

[5] C. Jacobsen and M. Jadud. Towards concrete concurrency: occam-pi on the lego mindstorms. *SIGCSE '05: Proceedings of the Thirty-Sixth SIGCSE Technical Symposium on Computer Science Education*, pages 431–435, 2005.

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

[7] J. Keller, C. Keller, and J. Traff. *Practical PRAM Programming.* Wiley-Interscience, 2000.

[8] A. Kimball, S. Michels-Slettvet, and C. Bisciglia. Cluster computing for web-scale data processing. *SIGCSE '08: Proceedings of the Thirty-Ninth SIGCSE Technical Symposium on Computer Science Education*, pages 116–120, 2008.

[9] J. Paul, M. Kouril, and K. Berman. A template library to facilitate teaching message passing parallel computing. *SIGCSE '06: Proceedings of the Thirty-Seventh SIGCSE Technical Symposium on Computer Science Education*, pages 464–468, 2006.

[10] C. Pheatt. An easy to use distributed computing framework. *SIGCSE '07: Proceedings of the Thirty-Eighth SIGCSE Technical Symposium on Computer Science Education*, pages 571–575, 2007.

[11] U. Vishkin. umiacs.umd.edu/users/vishkin/xmt.html.

[12] U. Vishkin. Using simple abstraction to guide the reinvention of computing for parallelism. *Comm. ACM*, to appear.

APPENDIX

A. CODE LISTING FOR SIMPLE SUM

```
#include <xmtc.h>
#include <xmtio.h>
#define n 8
#define log_n 3
int main()
{
   int h,p,B[log_n+1][n+1];
   int A[n+1]={0,3,1,4,1,5,9,2,6};

   // copying elements of an array to be summed A
   // into the leaves of a balanced binary tree B
   spawn(1,n)           // for i, 1<=i<=n pardo
   {
      int i;            // index for left-to-right
      i=$;              // XMT-C uses dollar sign
      B[0][i]=A[i];
   }

   // at each point in time p=2^h and n/p gives the
   // number of nodes at level h of the binary tree
   h=1;
   for(p=2;p<=n;p*=2)   // move up the tree
   {
      spawn(1,n/p)
      {
         B[h][$]=B[h-1][2*$-1]+B[h-1][2*$];
      }
      h+=1;           // h goes from 1 to log_n
   }

} // output uses printf and is not shown here
```