Teaching Parallel Programming with Active Learning

Mohammad Amin Kuhail, Spencer Cook, Joshua W. Neustrom, Praveen Rao
School of Computing and Engineering
University of Missouri-Kansas City
Kansas City, MO, US
e-mail: {kuhailm, slcc2c, jwnf7b, raopr}@umkc.edu

Abstract—**Today parallel computing is essential for the success of many real-world applications and software systems. Nonetheless, most computer science undergraduate courses teach students how to think and program sequentially. Further, software professionals have complained about the computer science curriculum's lag behind industry in their failing to cover modern programming technologies such as parallel programming. The emphasis on parallel programming has become even more important due to the increasing adoption of horizontal scaling approaches to cope with massive datasets. In order to help students coming from a serial curriculum comprehend parallel concepts, we used an innovative approach that utilized active learning, visualizations, examples, discussions, and practical exercises. Further, we conducted an experiment to examine the effect of active learning on students' understanding of parallel programming. Results indicate that the students that were actively engaged with the material performed better in terms of understanding parallel programming concepts than other students.**

Keywords: Parallel programming, teaching, OpenMP, data structures, visualizations, active learning.

I. INTRODUCTION

Industry leaders have frequently complained about the inadequately equipped computer science graduates for solving world problems using emerging technologies. Daniel Gelernter [1] in the Wall Street Journal stated his intentions to no longer hire new graduates from computer science programs is due in part to the few classes in the curriculum related to what the company was looking for.

An example of a relevant topic for modern software development is parallel programming, which involves executing code simultaneously with multiple processors. As computing speeds far exceed the requirements of many algorithms and the advent of multicore CPUs, parallel programming provides a method for improved utilization of hardware, thereby saving money through efficiency in power and space. Applications such as web servers, which handle high volumes of requests, further increase the demand for operations to happen in parallel with high reliability.

Today companies are aggressively investing in technologies such as Apache Hadoop [2] and Apache Spark [3] for storing, processing, and analyzing massive amounts of data, or big data, to gain competitive advantage over others. Shared-nothing architectures have become the norm for horizontal scaling over massive datasets. Data are distributed across a shared-nothing cluster and the workload is parallelized on the partitioned data to gain significant speedup. Given the growing demand for skilled workforce in big data analytics, there is a pressing need to incorporate parallel programming concepts in an undergraduate computer science curriculum.

Given the rising demand for parallel computing, many universities have been scrambling to fill the gap in computer science curriculum with new coursework. Cameron University worked towards increasing student interaction with industry experts by having students visit the University of Oklahoma Supercomputing Center and inviting parallel process professionals onto campus [4]. Universidad Nacional de Río Cuarto expanded their material from an elective course into a Data Structures II course using OpenMP [5]. Undergraduate students at Stanford University are being taught MapReduce [6] in a cloud environment [7]. These attempts and many others to integrate parallel programming into computer science undergraduate curriculum are very insightful, but there remains a need to explore different teaching methods and approaches to teaching parallel programming.

At the University of Missouri-Kansas City (UMKC), the entry-level Data Structures course had an introduction to parallel programming along with a code demonstration of parallel Merge Sort. In order to help students transition from sequential to parallel algorithms, the course material interactivity was increased using algorithm visualizations, practical exercises, discussions, and test environments. Furthermore, we wanted to test the effect of *active learning*, a teaching method that involves students more directly in the learning process, on students' understanding of parallelism. We split the students into two sections: One of the sections participated in increased active learning whereas the students of the other section did not benefit from active learning. We measured the results of our experiment by reviewing the results of quizzes and surveys. Results show that students engaged in active learning performed better in terms of understanding parallel programming concepts than others.

II. BACKGROUND

Several attempts have been made at various institutions to introduce parallel programming to undergraduate students. Some authors argue that it is important to engage the students with hands-on real-life examples [8]. Other authors emphasize that students need to be introduced to think parallel by showing them what can be parallelized and what cannot [9]. Some educators think it is important to teach college students parallelism early on [10]. Other educators advocate that parallelism needs to be introduced at the right level depending on the course [11]. It was also suggested that assigning projects and asking students to write research papers on their findings helps improve students' learning [12]. Like our fellow educators, we wanted to make sure that we engage the students with hands-on tasks and introduce parallelism at the right level.

Active learning has been suggested by educators to promote student participation and learning in the classroom [13]. A plethora of studies have shown the effectiveness of active learning in many fields. For example, an analysis of 225 studies that compares conventional lectures to active learning in math, science, and engineering courses found that active learning improves student performance on course evaluation [14]. A promising form of active learning is collaborative learning groups where students are assigned in groups, and they are given tasks to work on together [15]. Given the potential of active learning in the classroom, we also wanted to see its effect on students' understanding of the materials.

III. Method

The objective was to teach parallel programming with great effectiveness using the following principles: First, we gradually switched from what is known to unknown. Second, we engaged the students by means of visualizations, role play, examples, discussions, and hands-on tasks. Third, to make our teaching practical and beneficial, we demonstrated all the examples using a development environment. Further, the source code was shared with the students. Fourth, in addition to using traditional teaching methods, we tested the effect of active learning in the form of collaborative learning groups on students' understanding of the materials. Therefore, we split the students into two sections: The second section of students had collaborative learning groups, worked together on parallel programming tasks, and presented their solutions to the tasks to other students. The first section of students did not benefit from collaborative groups. Instead, the solutions to the tasks were presented to them in class. Apart from the active learning experiment, both sections were taught with the teaching methods and materials. We measured the results of our experiment by reviewing the results of quizzes and surveys.

Initially we taught the parallel programming concepts in general, but we also used OpenMP to implement parallel programming, because it is widely adopted and well integrated with C++, a programming language that the students are familiar with. Further, OpenMP is relatively simple to use as it adds a layer of abstraction that hides complex details [16].

The environment utilized was the Data Structures course at UMKC during the spring semester in 2017.

The parallel-programming materials were taught in five lectures. Each lecture lasted 1 hour and 15 minutes. The total teaching time was 6 hours and a half. The materials of this study can be found at [17]. Table I shows the teaching activities the process follows.

TABLE I. TEACHING ACTIVITIES

No.	Activity
1.	Pre-Quiz.
2.	Introducing parallelism and the fork-join model.
3.	Basic parallel code example that shows multiple threads.
4.	Explaining parallelizing a for loop with OpenMP.
5.	Introducing race conditions, and giving examples on resolving race conditions with synchronization constructs such as atomic, critical, locks.
6.	Giving three tasks on writing parallel code with OpenMP parallel for.
7.	Introducing OpenMP sections with an example.
8.	Giving a task on OpenMP sections.
9.	Interacting with the students with a visualization of serial and parallel merge sort.
10.	Giving the students a task on writing code for parallel merge sort.
11.	Giving an advanced task on parallelism.
12.	Active Learning Experiment Section 1: Teacher presenting solutions to selected parallelism task. Section 2: Students presenting solutions to selected parallelism task to other students.
13.	Post-Quiz
14.	Survey

First, the students were given a quiz in order to obtain a benchmark of past effectiveness in teaching methods. The details of the quiz can be found at the evaluation section. After the quiz, preliminary course material on parallel programming was covered. The material covered a brief introduction to parallel programming. The students were also shown the running time difference between serial Merge Sort and a parallel Merge Sort that sorted a ten-million item unsorted array. The details of Merge Sort were not explained yet.

Due to the time limitation, the coursework focused on the core topics of parallel programming. In particular, we focused on parallelizing serial for loops, creating multiple threads, race conditions, and creating sections. Using the OpenMP library, students were taught how to configure a C++-based OpenMP program in Visual Studio 2015. We demonstrated the code with a laptop, which has a multi-core Intel core i7 processor.

A. *Poor Parallel Programming*

In order to show the risk of poor parallel programming, students were given two algorithms that were supposed to count to ten million using a simple for loop (Algorithm I and Algorithm II). The students were also given a third algorithm (Algorithm III) that initialized an array to have multiples of two. The algorithms worked correctly in serial, but failed in parallel.

(1) Algorithm I did not parallelize the loop correctly (Figure 1) since it created a team of threads that all ran the loop ten million times. x resulted in the

expected value (ten million) since x is private for each thread:

```
int MAX=10000000;
#pragma omp parallel shared(MAX)
   {
          int x = 0;
          for (int i = 0; i < MAX; i++)
          {
                 x = x + 1;
          }

   }
```

Figure 1. Algorithm I.

(2) We discussed that parallelizing a `for` loop in OpenMP can be simply done via the `parallel for` construct. However, despite using `parallel for`, algorithm II did not parallelize the loop correctly (Figure 2). Algorithm II created a team of threads and divided the work amongst them to run the loop ten million times. Multiple loops were modifying the variable x at the same time. It was explained to the students that this situation is called a *race condition*. It was also explained that synchronization is needed when multiple threads are trying to update the same variable at the same time. We discussed several approaches to resolving race conditions with OpenMP such as the atomic and critical constructs as well as reduction and locks. Further, we discussed the pros and cons of the different approaches. Figure 3 shows an example for resolving race conditions with reduction.

```
int MAX=10000000;
  #pragma omp parallel for shared(MAX)
  for (int i = 0; i < MAX; ++i)
   x++;
```

Figure 2. Algorithm II

```
int MAX=10000000;
  #pragma    omp    parallel    for    shared(MAX)
reduction(+:x)
  for (int i = 0; i < MAX; ++i)
   x++;
```

Figure 3. Resolving race conditions for Algorithm II

(3) Algorithm III is not possible to parallelize since the iterations are dependent on each other.

```
int a[10];a[0]=2;
for (int i = 1; i < 10; i++)
 {
   a[i] = 2 * a[i - 1];
 }
```

Figure 4. Algorithm III.

B. OpenMP Sections

We discussed with the students that despite the usefulness of `parallel for` to parallelize for loops, it has limitations. For instance, one cannot break out of a parallelized for loop. Further, serial recursive functions cannot be parallelized with `parallel for`.

OpenMP sections provides a flexible solution to parallelizing serial algorithms. The `sections` construct assigns multiple threads to work on different blocks of code independently. To start with a simple example, we discussed how we can use OpenMP sections to allow two threads to search for an item in an unsorted array. Thread 1 works uses linear search to search the first half whereas thread 2 searches the other half (Figures 5 and 6). To illustrate the example, we showed a visualization with MS PowerPoint to illustrate the example.

```
void find_item(int low, int high, int target,
vector<int>& vec, int& index){
  for (int i = low; i < high ; i++){
    if (index != -1)
      break;
   if (vec[i] == target){
     index = i;
     break;
   }
 }
}
```

Figure 5. Serial linear search for finding an item in an array.

```
int find_item_sections(vector<int>& vec, int
target){
 int index = -1;
 #pragma omp parallel sections shared(index)
 {
 #pragma omp section /** thread 1**/
 find_item(0, vec.size()/ 2, target,vec,
index);

 #pragma omp section /** thread 2 **/
find_item(vec.size()/2,vec.size(),target,vec,
index);
 }
return index;
}
```

Figure 6. Algorithm IV: Using Sections to parallelize linear search

C. Merge Sort Activity

Serial merge sort had been taught to the students earlier in the semester. However, to refresh students' memory, the algorithm was quickly explained by means of an animated visualization. The visualization showed students step by step how the algorithm sorts an array of integers by dividing, conquering, and merging partially sorted sub-arrays.

After the visualization, each student was asked to use serial merge sort to sort the same array. The students were given index cards that represented integers in an array. The students placed the cards on their desks, and engaged in the activity. This role play activity only took a few minutes.

Similar to serial merge sort, parallel merge sort with two threads was explained to the students using a visualization (Figure 7). After the illustration, each student was asked to team up with a neighboring student to work on parallel merge sort with two threads. Furthermore, we illustrated merge sort with four threads with a visualization. Afterwards, we asked every four students to team up to work on parallel merge sort. Many students reflected that adding more threads (classmates) to work on parallel merge sort resulted in faster processing time, but ultimately it would make a more substantial difference if the array was large. However, the speedup factor was explained earlier. The students were made aware that increasing the number of processers and threads does not always result in speeding up the performance

D. Tasks

We gave the students different kinds of tasks to engage them and reinforce the ideas that we presented to them. We checked on the students as they were working on the tasks. We present some examples of the tasks here:

Task 1 (easy): Task 1 required the students to write parallel code that finds the minimum element in a vector. The students were given ten minutes to work on the task. Later, the students were presented with the solution, which was an application of parallel for and reduction.

Task 2 (medium): The students were given serial code for merge sort, and were asked to parallelize it. They were first asked to use two threads, and then generalize it to accommodate any even number of threads.

The students were given fifteen minutes to work on that task. We observed that most students successfully used OpenMP sections for the solution; one section for half the array, and another for the other half.

A few students had trouble remembering the exact syntax. Some students had an idea about how to generalize the algorithm so that it can use any number of threads, but did not have the time to write the code.

Task 3 (advanced): Task 3 was about optimizing a given piece of code. The students were expected to write pseudo code on paper. The students were given serial code (Figure 7) that allocates eight queens on a chessboard so that they do not attack each other horizontally, vertically, or diagonally [18]. The allocations are stored in an array. To start with, each queen is placed in its own column (queen 0 in column 0, queen 1 in column 1, etc.) to eliminate vertical attacks. The algorithm then generates all possible allocations. After generating an allocation, the algorithm checks if the solution is valid (when queens cannot attack each other).

The algorithm had been explained to the students earlier in the semester. It is relatively challenging to parallelize this algorithm since there are dependencies. Each queen allocates the next queen. However, with a close look, the first queen (queen 0) does not depend on the allocation of a previous queen. Hence, the allocation of queen 0 can be parallelized.

```
bool place_queen(int i) {
if (i == 8) {
 if (is_solution()) {
  cout << "Solution # " << (++numsolutions)
  << endl;
  print_board();
  cout << endl;
  return true;
 }
 else
  return false;
}
for (int j = 0; j < 8; j++) {
 row[i] = j;
 place_queen(i + 1);
}
return false;
}
```

Figure 7

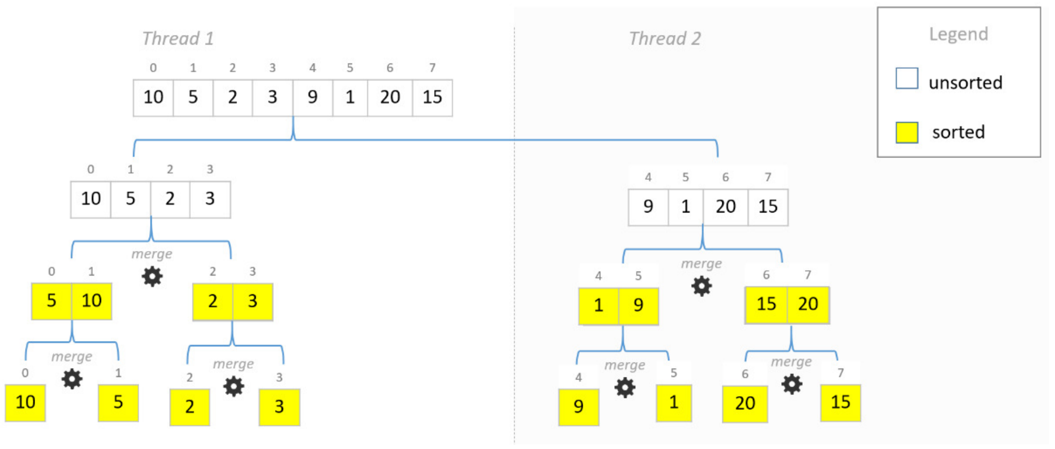

Figure 8. A snapshot of visualizing parallel merge sort.

IV. ACTIVE LEARNING EXPERIMENT

We designed an experiment to measure the effect of active learning in the form of collaborative learning groups on students' understanding of the materials. The students were split into two sections. We randomly chose the first section to be our control group and the second to be our treatment group. Other than the active learning experiment, both sections were taught with the same teaching methods and materials.

We randomly split the students in the treatment group into four groups. Each group had five or six students. The groups worked on different parallelism problems. The problems were designed to allow students to explore different aspects of parallelizing serial algorithms. The students were also asked to help and teach each other while working on the problems. Further, the students were asked to present their solutions in front of other classmates. To make sure that every student was involved in the process, we told them that any student can be asked questions about the presentation. The students managed to solve the problems successfully with very few issues. The presentations included walking the audience through the solution, and answering questions from the audience.

The tasks that the students worked on in the collaborative groups are as follows:

Task 1: The following function uses backtracking to find a path to the exit in a maze. The maze is simply a two dimensional array that contains a number of rows and columns. A cell can be part of a path, a barrier, a dead end, or a background cell.

Parallelize the `findMazePath`. For instance, imagine that you have multiple threads searching at the same time, but with different strategies (e.g. one strategy is to go up,down,left,right another is to go left,right,up,down, etc.). As soon as a thread finds the exist, other threads should stop.

```
bool findMazePath(color grid[ROW_SIZE][COL_SIZE],
int r, int c) {
  if (r < 0 || c < 0 || r >= ROW_SIZE || c >=
COL_SIZE)
     return false;      // Cell is out of bounds.
  else if (grid[r][c] != BACKGROUND)
     return false; //Cell was visited or dead end.
  else if(r == ROW_SIZE - 1 && c == COL_SIZE - 1){
    grid[r][c] = PATH;          // Cell is on path
    return true;
   }
  else {
   grid[r][c] = PATH;
   if (findMazePath(grid, r - 1, c)||
findMazePath(grid, r + 1, c) || findMazePath(grid,
r, c - 1)|| findMazePath(grid, r, c + 1)){
    return true;
  else{
   grid[r][c] = TEMPORARY;  // Dead end.
   return false;
 }
}
}
```

Figure 9, adapted from [19]

Task 2: Parallelize the following algorithm which finds the prime numbers between 2 and a given max number.

```
vector<int> calc_primes(const int max){
 vector<int> primes;
 for (int i = 2; i < max; i++){
  primes.push_back(i);
 }
 for (int i = 0; i < primes.size(); i++){
  //get the value
  int v = primes[i];
  if (v != 0) {
  //remove all multiples of the value
  for (int x = i + v; x < primes.size();x = x + v)
{
   primes[x] = 0;
  }
 }
}
return primes;
}
```

Figure 10

Task 3: Write parallel code for the Odd-Even sorting algorithm. Test your solution with a randomly-generated array that has a 1,000,000 items. When you demonstrate your code, compare the time difference between a serial Odd-Even and parallel Odd-Even.

Task 4: Write parallel code that merges two binary search trees. The idea is to convert each search tree into a sorted vector, and then merge the sorted vectors, and finally convert the sorted vector into a binary search tree. Can you scale the code so that it merges four or eight trees?

V. EVALUATION

We used a quiz and a survey in order to evaluate the effectiveness of our teaching method and collect students' opinions.

A. *Quiz Evaluation*

We gave students of Section 1 and Section 2 the same quiz on parallelism with OpenMP to measure the effectiveness of our teaching method, and to also see if active learning in section 2 made a difference on students' understanding. The twenty-minute quiz was given at the beginning and end of teaching parallelism with OpenMP. To keep track of improvement, we asked students to write their names on both quizzes. However, the quiz did not have credit that counts towards the students' grades. To make sure our results are reliable, we discarded papers coming from students who only took either the pre-quiz or post-quiz but not both.

The quiz included multiple choice, short answers, and brief coding. The questions focused on the following:

Quiz Questions: We asked several conceptual questions to test the students' knowledge on core concepts of parallelism. Here are some examples of the questions:

Question I (7 points): multiple-choice questions: the questions asked about advantages and disadvantages of parallel programming in relation to traditional serial methods, and in what situations would parallel code work best/worst.

- Question II (2 points): What is parallel programming? What are some programming APIs developers can use to write parallel code?
- Question III (2 points): Would it be better to use parallel programming within Insertion Sort or Merge Sort? Explain.
- Question IV (3 points): Write parallel code to calculate the average of a vector of integers. The question tests for the students' knowledge of synchronization.
- Question V (3 points): Write parallel code to search for a value in a binary tree.
- Question VI (3 points): Parallelize the given serial code for the quick sort algorithm.
- Question VII (3 points): What is a race condition? Mention an example. What's the solution? Coding Questions:

B. *Survey Questions*

To get the students' opinions on our teaching method, we conducted a survey at the end of the teaching activity. The survey contained quantitative and qualitative questions.

Quantitative Questions: We asked the students to rate various aspects of our teaching method on a scale from one to ten. The questions were as follows:

- Q I: How would you rate the section of the course on parallel programming?
- Q II: How useful was the material (slides and code)?
- Q III: How enjoyable were the activities (tasks and merge sort activity)?
- Q IV: How much did you learn from the merge sort and coding activities?
- Qualitative Questions: We asked the student two open-ended questions to get their feedback in more detail.
- Q V: What did you like about the section of the course on parallel processing?
- Q VI: What did you not like about the section of the course on parallel processing?
- Q VII: Do you have suggestions for improvement?

VI. EVALUATION RESULTS

Table I shows the number of students who participated in the evaluation. The study was conducted in the spring semester in 2017. We discarded participation from students who only took either the pre-quiz or post-quiz but not both.

TABLE II. STUDENT PARTICIPATION IN EVALUATION

Section	Number Of Students
1 (control)	15
2 (treatment)	21

Figure 11 shows a box plot showing the results of the quiz before and after the teaching activity in Section 1 and Section 2. Tables III, IV, V, and VI show the details of the results. The quiz was out of 23 points. The tables show how students scored in the individual questions as well as the total quiz score. The results show only know very little about how to write parallel code prior to the teaching activity. Overall, our interactive teaching method points to a significant improvement in all areas.

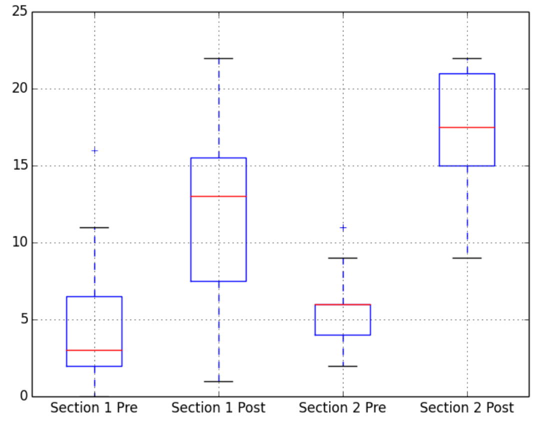

Figure 11: Box Plot of Quiz Results

TABLE III. PRE-QUIZ RESULTS FOR SECTION 1

Questions	Mean	STD	Max	Min
Q I (out of 7)	2.8	1.42	6	0
QII (out of 2)	0.53	0.83	2	0
QIII (out of 2)	0.47	0.74	2	0
QIV (out of 3)	0.17	0.65	2.5	0
QV (out of 3)	0.13	0.52	2	0
QVI (out of 3)	0.1	0.39	1.5	0
QVII (out of 3)	0.4	1.06	3	0
Total (out of 23)	**4.6**	**4.34**	**16**	**0**

TABLE IV. PRE-QUIZ RESULTS FOR SECTION 2

Questions	Mean	STD	Max	Min
Q I (out of 7)	3.71	1.31	7	1.31
QII (out of 2)	1.14	0.79	2	0
QIII (out of 2)	0.76	0.89	2	0
QIV (out of 3)	0	0	0	0
QV (out of 3)	0	0	0	0
QVI (out of 3)	0	0	0	0
QVII (out of 3)	0.24	0.77	3	0
Total (out of 23)	**5.86**	**2.39**	**11**	**2**

A. *Student Learning Improvement*

Table VII shows how students in section 1 and 2 improved. On average, students of section 1 improved 6.87 points out of 23 points (29.87%) whereas students of section 2 improved 11.55 points out of 23 points (50.27%).

A few students reported that if they had more time working on the post-quiz, they would have given better answers to some questions, especially the coding questions.

TABLE V. POST-QUIZ RESULTS FOR SECTION 1

Questions	Mean	STD	Max	Min
Q I (out of 7)	3.93	1.22	6	1
QII (out of 2)	1.47	0.92	2	0
QIII (out of 2)	1.37	0.89	2	0
QIV (out of 3)	1.27	1.22	3	0
QV (out of 3)	0.4	0.91	3	0
QVI (out of 3)	1.5	1.27	3	0
QVII (out of 3)	1.53	1.51	3	0
Total (out of 23)	**11.47**	**6.02**	**22**	**1**

TABLE VI. POST-QUIZ RESULTS FOR SECTION 2

Questions	Mean	STD	Max	Min
Q I (out of 7)	5.24	1.04	7	1.04
QII (out of 2)	1.86	0.36	2	0.36
QIII (out of 2)	1.81	0.51	2	0
QIV (out of 3)	2.24	1.04	3	0
QV (out of 3)	1.4	1.39	3	0
QVI (out of 3)	2.19	1.249	3	0
QVII (out of 3)	2.59	1.02	3	0
Total (out of 23)	**17.40**	**4.11**	**22**	**9**

TABLE VII. IMPROVEMENT COMPARISON

	Section 1	Section 2
Improvement Mean	6.87 points	11.55 points
Improvement STD	4.76 points	3.86 points

To show that the improvement was meaningful from before to after the quiz, we ran a pairwise t-test on the before and after quiz data for each section of the class. The improvements in section 1 and 2 had kurtosis/skew of -1.77/.09 and -0.40/-0.50 respectively. This is well within the range to indicate that both improvements are normally distributed and the tests will be meaningful.

We conducted the experiment to test two hypotheses. Our first hypothesis is: students' mean score in both sections after the quiz is higher than the students' mean score before the quiz in both sections. In other words, our teaching method improves students' understanding of parallel programming in both sections. The second hypothesis is: Active learning improves understanding of parallel programming. In other words, the mean score of students who benefited from active learning (section 2 students) after the quiz is higher than the mean score of students in section 1 after the quiz.

The null hypothesis can be defined as: our teaching method has no effect on students' score, which means the mean score of students in both sections before the quiz is equal to the mean score in both sections after the quiz, $H_0 : \mu_{after} = \mu_{before}$. Our results show that in section 1, p=6.75*10^{-5}, and in section 2, p=3.38*10^{-8}. Thus at $\alpha = .05$ in both cases, the null hypothesis must be rejected. Another null hypothesis can be defined as: students' mean score before the quiz is higher than the students' score after the quiz in both sections, $H_0 : \mu_{after} < \mu_{before}$. This hypothesis is also rejected for each with p=3.38*10^{-5}, 1.69*10^{-8} respectively. Rejecting both null hypotheses indicates that the alternative hypothesis, our teaching method of parallel programming improves understanding, $H_A : \mu_{after} > \mu_{before}$, is more likely.

We also wanted to test our second hypothesis, active learning improves students' understanding of parallel programming. The null hypothesis can be defined as: students' mean score in both sections after the quiz is similar, $H_0 : \mu_{section\,1} = \mu_{section\,2}$. The null hypothesis is rejected with p=0.004. Additionally, the null hypothesis that the mean score in section 1 is higher than the mean score in section 2 after the quiz, $H_0 : \mu_{section\,1} > \mu_{section\,2}$ is rejected with p=0.002. Hence, it cannot be accepted that section 1 improved the same or more than section 2. Therefore, our second hypothesis, active learning improves understanding of parallel programming, $H_A : \mu_{section\,2} > \mu_{section\,1}$, is more likely.

B. Survey Results

The quantitative survey results (Table VII) show positive feedback from the students on our interactive teaching activity. The mean score for each question in both sections was great than 6, indicating the students felt it was a beneficial experience, confirming the results seen from their quiz scores. On the qualitative questions, many students commented positively on our teaching activities. In particular, some students appreciated the informative and clear notes, tasks and visualizations. Other students found it useful that they were showed how to parallelize serial algorithms. On the other hand, a common theme emerged in student's negative feedback. Students commented on the short amount of time to cover such an important subject. Further, students requested additional time and activities on parallel programming. Some students wanted the topics to be explored more deeply. Some students wanted more guidance on how to run parallel code with Visual Studio.

TABLE VIII. QUANTITATIVE QUIZ RESULTS FOR SECTION 1

Questions	Mean	STD	Max	Min
Q I (out of 10)	7.88	1.22	10	5
QII (out of 10)	8,18	1.42	10	4
QIII (out of 10)	7.71	1.96	10	3
QIV(out of 10)	7.56	1.70	10	4

TABLE IX. QUANTITATIVE QUIZ RESULTS FOR SECTION 2

Questions	**Mean**	**STD**	**Max**	**Min**
Q I (out of 10)	6.70	2.05	10	1
QII (out of 10)	7.30	2.30	10	4
QIII (out of 10)	6.61	1.96	10	1
QIV(out of 10)	6.52	2.74	10	4

VII. DISCUSSION AND CONCLUSION

In this paper, we presented an innovative approach to teaching parallel programming to undergraduate students. Our approach combined several ingredients to improve the effectiveness of learning. We engaged the students with visualizations, role play, and practical exercises. Moreover, we provided the students with informative notes and source code. We also tested the effect of active learning on students' understanding of parallel programming. Our evaluation shows a significant improvement in understanding theoretical and practical PDC topics. Further, the active learning approach seems to give students an advantage in terms of allowing them to engage in the material and learn it more effectively. To complement our efforts, we allowed the students to practice PDC topics further by asking them to use parallelism in their course projects. Overall the result was positive with a majority of students successfully implementing parallel techniques into their own code.

Despite the promising results, we believe there is still room for improvement. We believe more time needs to be given to covering PDC topics in more depth, and allowing the students to practice more exercises. For instance, it would be very helpful to have the students practice more exercises in the lab, and see the speedup factor. In our teaching activity, we used Intel Core i7 processor to illustrate the usage of parallelism. However, we believe allowing the students to work on clusters to process large sets of data would be more practical.

Long term, a full parallel programming course is anticipated to grow students into practical preparation for the real world.

ACKNOWLEDGEMENTS

We would like to thank NSF/IEEE-TCPP for awarding us the Early Adopters 2015 award to integrate parallelism into our computer science undergraduate curriculum at UMKC. We would also to thank professors Appie Van de Liefvoort and Brian Hare for their comments. Finally, we would like to thank the CS 303 Data Structures students from UMKC for their enthusiastic participation.

REFERENCES

[1] D. Gelernter, "Why I'm Not Looking to Hire Computer-Science Majors". Wall Street Journal, 2015.

[2] T. White. "Hadoop: The Definitive Guide". O'Reilly Media, Inc., 1st edition, 2009.

[3] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, and I. Stoica. "Spark: Cluster Computing with Working Sets". In Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10, 2010, pages 10–10,

[4] J. Dean and S. Ghemawat. "MapReduce: simplified data processing on large clusters". Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation, 2004, p 10-10,

[5] M. Estep, F. M. "Methods for Teaching a First Parallel Computing Course to Undergraduate Computer Science Students". Proceedings of the International Conference on Frontiers in Education: Computer Science and Computer Engineering (FECS), 2014.

[6] A. S. Rabkin, Charles Reiss, Randy Katz, and David Patterson. "Experiences teaching MapReduce in the cloud". In Proceedings of the 43rd ACM technical symposium on Computer Science Education (SIGCSE '12), 2012, pp. 601-606.

[7] M. Arroyo, "Teaching Parallel and Distributed Computing topics for the Undergraduate". Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), IEEE 27th Internationa, 2013.

[8] R. Keller, "Teaching parallel programming to undergrads with hands-on experience", Workshop on Parallel, Distributed, and High-Performance Computing in Undergraduate Curricula (EduPDHPC), 2013.

[9] A. Marowka., "Think Parallel: Teaching Parallel Programming Today", IEEE Computer Society, 2008.

[10] D. Johnson, D. Kotz, F. Makedon, "Teaching Parallel Computing to Freshmen", 1994, Conference on Parallel Computing for Undergraduates, Colgate University, 1994.

[11] M. Burtscher, W. Peng, A. Qasem, H. Shi, D. Tamir, H. Thiry, A Module-based "Approach to Adpoting the 2013 ACM Parallel Computing", 2013, 2015, SIGCSE '15 Proceedings of the 46th ACM Technical Symposium on Computer Science Education.

[12] Y. Pan, "Teaching Parallel Programming Using Both High-Level and Low-Level Languages", Computational Science — ICCS 2002, 2002.

[13] D. Weltman, "A Comparison of Traditional and Active Learning Methods: An Empirical Investigation Utilizing a Linear Mixed Model", PhD Thesis, The University of Texas at Arlington, 2007, p.7.

[14] Freeman, S. et al. "Active learning increases student performance in science, engineering, and mathematics". Proceedings of the National Academy of Scientists, 111(23), 8410–8415. http://dx.doi.org/10.1073/pnas.1319030111, 2014.

[15] R. Hake, "Interactive-engagement versus traditional methods: A six-thousand-student survey of mechanics test data for introductory physics courses". American Journal of Physics, 1998.

[16] B. Chapman, G. Jost, R. Van Der Pas, "Using OpenMP", The MIT Press, Cambridge, Massachusetts, 2008.

[17] M. Kuhail, J. Neustrom, "Materials of Interactive Teaching of Parallel Computing", Found at: https://www.dropbox.com/sh/b84d974mn9ruypp/AACl7i67geNgvA5xAlaX2s4Ea?dl=0 , 2017.

[18] "N-Queen Problem", Found at:

http://mathworld.wolfram.com/QueensProblem.html , 2017.

[19] E. Koffman, P. Wolfgana, "Objects, Abstractions, Data Structures and Design Using C++", Wiely 2005.