20 Years of Teaching Parallel Processing to Computer Science Seniors

Jie Liu
Computer Science Division
Western Oregon University
Monmouth, Oregon, USA
liuj@wou.edu

Abstract— **In this paper, we present our Concurrent Systems class, where parallel programming and parallel and distributed computing (PDC) concepts have been taught for more than 20 years. Despite several rounds of changes in hardware, the class maintains its goals of allowing students to learn parallel computer organizations, studying parallel algorithms, and writing code to be able to run on parallel and distributed platforms. We discuss the benefits of such a class, reveal the key elements in developing this class and receiving funding to replace outdated hardware. We will also share our activities in attracting more students to be interested in PDC and related topics.**

Keywords—parallel processing; multicore processing; curriculum development; Computer Science education; parallel programming

I. Introduction

Despite the fact that nowadays a multi-core "parallel computer" is the only choice in the PC market, schools still teach many Computer Science concepts with some form of implication that processes need to follow some necessary sequence. For example, in the definition of "algorithm," we use expressions such as "a set of ordered steps" [3], "step-by-step", or "a detailed sequence" [4]. They all imply that some form of sequence must be maintained in completing the task on hand.

In the traditional Computer Science curriculum, almost all algorithms introduced to our students in their first few years are sequential. It is true that many steps of an algorithm have to be executed in a certain order. Still, for many students, they have to wait until their senior year to see and implement a parallel algorithm, if at all. For students who never have a parallel programming class in their college education, we can only hope that they are exposed to PDC concepts somewhere else soon.

We at Western Oregon University (WOU) firmly believe that teaching PDC is a must for any Computer Science programs. Clearly, the speed of a single core computer has not increased much lately. Instead, the number of cores on a single CPU, sometimes in the form of a GPU and a co-processor, has been increasing steadily. Intel has recently announced that its new Knights Landing chip has 72 cores and can reach a double-precision speed exceeding 3 teraflops [9]. We believe that is schools' job to introduce to our students PDC concepts so they know to look at the currency as a possible solution to solve performance related issues.

EduHPC2016; Salt Lake City, Utah, USA; November 2016
978-1-5090-3827-5/16/$31.00 ©2016 IEEE.

WOU has been offering a senior level parallel programming class since 1994. Currently, the class is named Concurrent Systems and is counted as a Software Engineering track elective. In this paper, we will present the history of the course and some of the lessons we have learned through the teaching and development of the course.

WOU is a liberal arts school with 7,000 undergraduate students and less than 1,000 graduate students. Our Computer Science division has about 150 students majoring in the Computer Science program. At its peak, the Concurrent Systems class serves about 20 seniors, which is about 1/3 of our graduating class.

Students can easily agree that it is a good thing for them to learn the theory in parallel processing and be able to implement some of the not so trivial parallel algorithms using some real programming languages on a real parallel computer. However, making them genuinely interested in the subject is a very different story. We are happy to report that most of our students enjoy the class and are happy that they selected the course.

II. About our Concurrent Systems class

Program programming's perspective, our Concurrent Systems class is really focusing on parallel processing at the current stage because we are using multi-core computers to satisfy the hardware requirement. Throughout the years, however, students in the class had written parallel programs on UMA parallel computers, Beowulf clusters, and GPUs. We emphasize on coding because we believe that only through programming, students can gain firsthand experience on concepts such as process communication, data dependency, load balancing, locking, synchronization, and the performance effect of granularity. The class also covers many PDC proposed topics, such as parallel computer organizations (mesh, hyper-tree, butterfly, hypercube, hypercube ring, shuffle-exchange networks, etc.), parallel algorithms on PRAM (finding max in O(1), fan-in, parallel prefix sum, list ranking, parallel merging, etc.), parallel programming concepts (shared memory vs. distributed memory, message passing, speedup, cost of a parallel algorithm, NC and P-Complete classes, Amdahl's law and Gustafson's Law, barrier and semaphore synchronizations, data parallelism vs. control parallelism, and Brent Theorem etc.), and parallel sorting algorithms (Bitonic sort, parallel quick sort,

random sampling, etc.). In addition, students have term projects and, every two weeks, a programming assignment to practice many of the key concepts and algorithms we discussed in class. Currently, we are using Microsoft's Visual Studio Task Parallel Library and C# for our programming assignments. The class is a 3 credit hours one, meaning we have total of 30 lecture hours in 10 weeks.

A key component of our Concurrent Systems classes is the required term project. Right before the midterm, each student is assigned a project related to one or more PDC concepts. Students are allowed to find their own suitable projects. The projects help students' learning in many ways. First, students need to conduct research either to learn some new tools or to find/develop parallel solutions for problems that have well define sequential solutions. Second, most students have to implement the parallel algorithms or to experiment with new tools. Third, students have to give presentations, which not only sharpen their communication skills, but also allow them to learn from each other on a wide variety of topics.

Some of the topics students worked on in Winter term of 2016 were GPU Programming, J-Sort, Parallel Programming using F#, Parallel Processing in the Cloud, Parallel Sorting by Regular Sampling, Parallel Gaussian Elimination, and False Coloring. Outstanding projects have been selected to give presentations in our school's Academic Excellence Showcase, a school sponsored event to reward students' outstanding academic achievement and to demonstrate the abilities of our students to their peers, professors of other disciplines, and school administrators.

III. A Short history of WOU's teaching of parallel and distributed computing

In 1990, our campus received two Sequent Symmetry UMA parallel computers as a donation from Sequent Computer Systems that would have costed us millions. The machines each had twenty 386-16 processors. Sadly, the school could not afford the supporting contract of about $40,000 per machine per year, so the computers were largely maintained by our staff members and students. Right after year 2000, when the computer science student enrollment was down drastically, the school requested us to shut down these two parallel computers completely because, with the money we spent on electricity for the air conditioning units, the school could have purchased several desktop computers that would be many times faster than the parallel computers. The fact that our Sequent computers were parallel computers was not important.

For the next five years, we used a 12-processor Sequent Symmetry retired from school computing services as our parallel computer to cover the hardware needs for our parallel processing class. The computer was turned on only when the class was in session. Unfortunately, we had used all the spare parts and were forced to look for alternatives.

Teaching any parallel programming class without a parallel computer is not a good situation to be in. Programming on a parallel computer should be a major component of a parallel programming class. There are many important and hard to master skills and concepts, such as partitioning of tasks, understanding and handling communication and synchronizations, understanding the single program multiple data (SPMD) approach of parallel programming, and debugging parallel programs. These concepts and skills cannot be comprehended easily without actually programming on a parallel computer. Fortunately, Linux Beowulf clusters were popular then and provided an economic solution to our problem of needing a real parallel computer, so we requested and received funding to build a Beowulf.

After the initial trial and error, we mastered the skills to systematically configure an eight-node Beowulf in an hour from a dedicated rack of nodes, so for several years, creating a Beowulf was students' first lab. Students used MPI and implemented a few distributed parallel algorithms. However, due to communication overhead and lacking of suitable problems, students were not overly impressed with the performance gain through the use of multi-computers.

In 2009, the multi-core computers were readily available, so we switch again to use multi-core computers with Jibu's parallel program library and C#. In 2010, Microsoft introduced its Task Parallel Library (TPL) with .Net Framework 4, so we switched to teaching our concurrent systems class using multi-core computers and Microsoft's Visual Studio supported C# and TPL. One major benefit of using multi-core computers now is that every student's computer is already a multi-core, so satisfying the hardware requirement becomes the easiest part. For our students, C# is also a programming language they are using in other senior level classes, so we can focus on PDC concepts instead of programming languages.

It looks like we are switching our hardware again. We have been following the development on NVIDIA's Tesla GPU and Intel's Knights Corner and Knights Landing co-processors. We believe it is time for our students to experience a new architecture for parallel programming again, so we have just secured a Dell PowerEdge T630 server with an Intel Xeon Phi 3130A co-processor. We are looking forward to having our students programming on this Xeon Phi 3130A's 60 plus cores.

IV. Nurturing Support from School administrations and fellow professors

In 1990, parallel processing was not yet a class offered by many small schools similar to WOU. We managed to win our division chair, the dean, and the provost's support to develop such a course mostly because of the following reasons. First, we had the hardware. In 1990, our neighboring school spent couple of million dollars to acquire a parallel computer. If we had to purchase a parallel computer to start offering a parallel processing class, we would have most likely been unable to receive the approval from our school, or we had to look for alternatives to meet the hardware requirement. Second, during later 80's and early 90's, parallel processing was a popular research topic in Oregon schools and in many of the top universities worldwide. So, a proposal of developing a course to cover these topics was relatively easy to win the support of our fellow professors and then administrators, especially when no new funding was requested. Third, we just hired a new professor whose major research area in graduate school was parallel processing. The professor was very excited about teaching a parallel processing class and completed the development of the

course quickly. Fourth, the new professor was successful in convincing the school administrators and fellow professors that offing a parallel programming class was not only a must, but also would be beneficial to students in many ways that no other classes could match. He used our senior level compiler class to make his point.

Many schools, including WOU, offer a senior level class in compiler. We do so not because we expect our students to build compilers in their career, but for the reasons that the compiler class synthesizes many important computer science concepts. The parallel processing class could be very much like the compiler class in synthesizing important concepts in hardware, software, operating systems, algorithms, data structures, and theory of computation etc. In addition, students would be much more likely to actually use the knowledge and skills learned in the parallel class later on.

The Concurrent Systems class has been a small one from enrollment's point of view. However, it always attracts good students who are smart, curious, strong in math and coding, and willing to invest their efforts to master the materials. Most of the students who completed the class enjoyed it and gave it very high scores during our school's end of the term class evaluations, which we will show in a later section of this paper.

The course survived three rounds of necessary hardware changes can be attributed to two main reasons. First, since only our top students enroll in the class and they have been providing very positive feedbacks even after graduation, professors who support the class feel good themselves and have been more than willing to voice their support whenever necessary. Second, we have been very careful in selecting the new technology used to replace the outgoing ones to make sure the new technology is suitable and, more importantly, affordable.

V. Making Learning Parallel Processing Interesting to students

Even for important subjects, such as parallel processing, we still have to make the learning interesting so students are willing to invest in time and effort to learn. In the past 20 some years, we have developed a series of methods to make our class interesting to our students and to uphold their curiosity. Also, we selected Dr. Quinn's Parallel Computing: Theory and Practice, listed as [1] in our references, as the main textbook. The book is easy to read, provides a good coverage of the topics, and is well-liked by our students.

A. *Start with impossible*

Our students' first programming assignment is to find the largest element of a given array in parallel and measure the performance improvement, through the calculation of speedups, between the parallel version and sequential version. Naturally, we would be discussing parallel algorithms for finding the largest element on PRAM in our lectures at the same time. Students could easily understand a theoretical O(log n) algorithm using n/2 processors (the fan-in approach), or a concrete solution using a fixed number of *P* processors. Now, the question "could you do better if you had unlimited number of processors?" encourages students to think hard. We then present the algorithm listed in Figure 1. The same notations are used in [1] and can be easily deduced with some programming knowledge. The simple algorithm has a complexity of O(1) on PRAM with n^2 processors that support concurrent read and write and "common write" as the writing conflict resolution.

For most students, this is the first non-trivial algorithm with a O(1) complexity. The fact that, the execution time does not increase at all even if we double or quadruple the problem size excites students a great deal. They all work hard to understand the algorithm and try to find the tricky part.

```
FindingMax(arrA[0..n-1])
{
    for all  P_{i,0}  where 0 <= i < n-1
             arrB[i] = 1
    for all P_{i,j}   where 0 <= i, j < n-1
             if (arrA[i] < arrA[j])
                 arrB[i] = 0
    for all  P_{i,0}   where 0 <= i < n-1
             if (arrB[i] = 1)
                 print arrA[i]
}
```

Figure 1. Finding the max of an array in O(1) time

This O(1) algorithm takes some thinking to understand fully. Once understood, students quickly started to point out that it is not possible to have n^2 processors in reality. However, for a PRAM, we can assume we have whatever the number of processors needed. Once so often, we could have one or two students pointing out that activating n^2 processors on PRAM actually takes O(log n) time.

This is the best time to introduce the concept of the cost of a parallel algorithm, which is defined as the product of number of processors and the execution time [1]. For this case, the sequential algorithm is much more cost effective than the parallel algorithm, which is just very fast. This way of introducing the concept of cost of a parallel algorithm makes a very deep impression to our students. Hardly any student would make mistakes around this and other related concepts for the rest of the class.

After discussing such an algorithm, students learned that critical section could be an important opportunity in designing parallel algorithms and also had a good understanding about many aspects of PRAM. Such an unusual algorithm is an example where "thinking outside the box" is strongly demonstrated. Elegant algorithms like this really draw student's attention. After students are told that there will be many algorithms just as clever as this one, many are eager to embrace more mind exercises for the rest of the term.

B. *Let student see the benefit of parallel processing early on*

Almost every student has a multi-core computer nowadays, so we ask them to check their computers' CPU utilization. Not surprised, most students have a quad-core CPU, and the CPU utilization is less than 50% at most because most of the applications are developed for single processor and cannot utilize the multi-core.

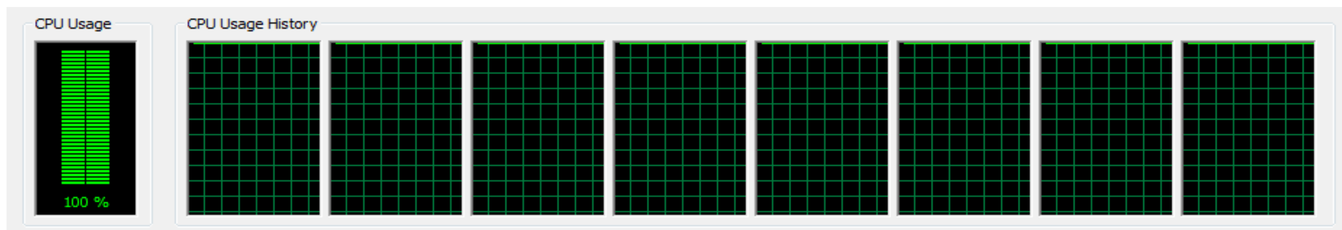

Figure 2. CPU utilization on parallel matrix multiplication -- all cores are at 100% for several minutes

With Microsoft's Visual Studio, we can easily fire up many cores using the "parallel for" structure. Once a program, such as matrix-multiplication, starts as shown in Figure 2, CPU utilization can quickly reach 100%, and having 99% of overall CPU time is dedicated to the problem. The figure shows that every core has been running at 100% for some times already. For many students, this is the first time they have seen any computer working this hard. At this point, most students are very eager to try their hands on coding some parallel algorithms and start to think about all kinds of problems that could benefit from this newly mastered skill.

C. Inspect a known problem from a different angle

In parallel programming classes, we often discuss developing parallel algorithms for a well-known problem. However, if we look a problem from a different angle, we may reach different solutions, possibly parallel solutions. For example, merging two sorted array of size *n* is a classical data structure problem with a complexity of O(n). On PRAM with n processors, we can solve the problem in O(log n) as showing in Figure 3.

```
MergeArray(A[1..n])

{ int x, i, low, high, index
   for all     where 1 <= i <= n
      // The lower half search the upper half,
      // the upper half search for the lower half
    { high = 1    // assuming it is the upper half
      low = n/2
      if i <= (n/2)
    { high = n
      low = (n/2) + 1}
      x = A[i]
      Repeat // perform binary search
      {  index = floor((low + high)/2)
         If x < A[index]
                 high = index – 1
         else
                 low = index + 1
      }
      until low > high
      A[high + i – n/2] = x
    }
 }
```

Figure 3. Merging an array of *n* elements with two sorted halves in parallel on PRAM

Assuming two sorted arrays are stored in the two halves of a larger array, the outline of the algorithm is as follow. For a given element a_i in the first half of the array, we can find out the number of elements in the second half that are smaller than it (denoted as POS) by using binary search in O(log n) time as if we were to insert a_i into the second half. We know there are i elements that are smaller than a_i in the first half, so there are total of i + POS elements that are smaller than a_i in the entire collection. We can then just copy a_i into in the final merged array's slot at i + POS. Using n processors to merge the two half arrays (of n/2 elements each) takes O(log n) time because each processor works independently and concurrently. Figure 4 attempts to illustrate the algorithm with a real example.

In Figure 4, let's assume the top array is the first half of the array needs to be merged. Let's use A[1] (zero based), which has a value of 4, as an example. Since its index is 1, it has 1 element smaller than it in its half of the array. If we were to insert A[1] into the second half, it would have taken the slot of the fourth element, that is POS = 3. That is, there are three elements smaller than A[1] in the second half of the array. This can be determined in O(log n) time using the binary search algorithm. So in total, there are four elements smaller than A[1]. Therefore, we can just copy A[1] into the fifth element of the final array, as showing in Figure 4.

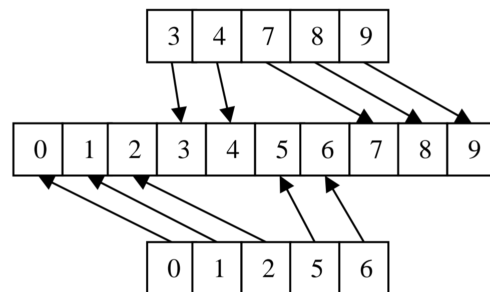

Figure 4. Merging an array with two sorted halves.

This algorithm generates a lot of interests among students because the parallel solution is relatively speaking easy to understand. In addition, students already know the problem well and understand the sequential algorithms used in the parallel solution well too. However, the key in the parallel solution is to look into the problem differently. Instead of finding an element to be copied into a given slot of the final array (as we do in our sequential solution), the new algorithm take the opposite approach. For a given array element, the algorithm finds the slot where the final element should be stored and copy the element

into the slot. Again, thinking outside of the box is well demonstrated here.

D. Challenge students with complex algorithms

Many parallel algorithms take time for students to understand because not only these algorithms are complex, and their underlining problem solving approach can be very different from what the students have experienced. Having a clear picture about several thousand or even millions of processors working concurrently in solving a single problem takes time to be used to. Bitonic sort is a very good example. The algorithm sorts an array of size n in $O(log^2 n)$ time using n/2 comparators (simple processors that sort two numbers) [2].

Coding such an algorithm is an excellent programming exercise, where students learn the value of careful design and analysis of algorithms. During the early years, we just assigned the lab without giving much help. We estimated that close to 50% of students were not able to find the essence of the problem and could not come up with a workable approach solving the problem. Even we later gradually provided many helps, for many students the helps came too late because they already determined that they could not complete the assignment.

Now we start by giving some general directions and discussing the two main loops of the algorithm. In addition, during the introduction of the algorithm, we give programming hints. Not to mention that our students have been trained to research solutions on the Internet whenever they encounter some difficulties. To many students, implementing Bitonic sort algorithm is the hardest individual programming assignment ever. Knowing its difficulty makes a big boost of students' self-confidence after being able to complete the assignment. The assignment for implementing Bitonic sort comes at the later part of the term when students already have some experience implementing and debugging parallel algorithms. Still, it is one of these moments where the students are highly anticipating and somewhat afraid at the same time.

The parallel version of the list-ranking algorithm finds the ranks of elements in a linked list and can appear to be simple. However, truly understanding it requires the comprehension of a couple of important concepts of parallel programming and data structures, namely data parallelism and representing a linked list using an array. In addition, the list-ranking algorithm shows once again the importance of choosing the right data structure. When using traditional "pointers", the list-ranking algorithm seems to be inherently sequential. However, when we represent the list using an array, an NC class algorithm to solve the same problem becomes available. In addition, the key part of the algorithm has only two lines, an excellent example of "simple" parallel algorithms performing complex tasks fast.

E. Learn to look at problems from very different angles

Because parallel programming is a relatively new discipline, students are exposed to many new findings. Some of which seem to contradict to each other. This benefits students by demystifying Computer Science research and teaches students that studying the same problem from different perspectives may result in very different results. Subconsciously, we hope, students develop the ability to think independently and to question well-accepted findings.

One such example of appeared contradicting results is found in the study of the speedup ψ of a parallel algorithm. Amdahl's Law states that $\Psi = 1/(f + (1 - f)/p)$, where f is the fraction of operations in the computation that must be performed sequentially, and p is the number of processors used in the parallel solution. However, Gustafson-Barsis' Law states that $\psi \leqslant p + (1 - p)/s$, where s denotes the fraction of total execution time spent in serial code in a parallel algorithm.

According to Amdahl's Law, even if we have infinitely many processors, the speedup is limited to $1/f$, i.e. $\psi \leq 1/f$. This is not the same suggested by Gustafson-Barsis' Law, which states that when the total execution time spent in serial code is very small compared to the total execution time spent in parallel code (which is generally true for applications using parallel computers), the s is significantly smaller compare to the parallel portion of the algorithm. In addition, as the problem size increases, s becomes even smaller. Therefore, the speedup is only bounded by the number of processors, i.e. $\psi \leqslant p$. Now the question: which law is correct? The answer is: BOTH!

Amdahl's Law answers the following question [1]:

> If an algorithm takes t time on a sequential computer, how long does it take for the *SAME* problem to be solved on a parallel computer with p processors?

Gustafson-Barsis' Law considers the fact that we use parallel computers to solve larger problems. It answers the following question [1]:

> If a parallel algorithm takes t time on a parallel computer with p processors to solve, how long does it take for a sequential computer with the same type of processor to solve the same problem?

When looking at the same problem from different angles, we sometimes reach different conclusions. The take home here for the students is that new discoveries may be waiting for you if you are willing to turn some rocks and look the problem from different perspectives.

VI. PLAN ON INTRODUCING PDC CONCEPTS TO MORE STUDENTS

We plan to introduce PDC concepts in three other classes: freshman year programming classes, second year data structure classes, and third year Operating Systems classes. The plan is to just show some simple programs for the freshman year programming classes. For example, at one of the lectures I gave to some first year Computer Science students, I showed them using Monte Carlo simulation to calculate π sequentially, then in parallel and compared the performance difference. The implementation can be done in many programming languages.

The UMA version of parallel matrix multiplication would be a good example to show to the data structure classes for students to see the effect parallel processing and granularity on performance. For Operating Systems, threading would be one of natural topics to expend more PDC related concepts. The main purpose of introducing the PDC concepts early is to stir some interests toward the capabilities of concurrency.

VII. IMPROVEMENT WE PLAN TO MAKE IN THE NEAR FUTURE

There are a few areas we plan to make changes. The first is in the hardware. In the past we have been limiting ourselves to one platform, mostly due to the cost considerations. We plan to add a programming assignment to use instances in the cloud to build a virtual parallel computer and offer students an opportunity to experience using MPI for high performance computing again, without the limitation of hardware infrastructure [7]. A programming assignments of this type expose students to different architectures, paradigms, libraries, and possibly programming languages. The assignment will surely drive home the concept of Amdahl effect, where the performance improves with the increasing of problem size.

We have been using C# and Microsoft's Visual Studio for about 10 years. Microsoft's Task Parallel Library greatly simplifies the development of parallel algorithms, especially data parallel algorithms, so students can develop both fine-grain and coarse-grain algorithms that are scalable to utilize all cores without having to work directly with threads. We'd like to expose our students with some low level concepts with a lab using Java 8's Threads and compare the performance with C# and Microsoft's support of asynchronous operations using its System.Threading.Tasks.

VIII. FROM STUDENT EVALUATIONS TO PEEK INTO THE EFFECTIVENESS OF THE COURSE

Figure 5 shows our most recent school survey result on our course. It is a screen dump form our school's professor evaluation web page. Since surveys are voluntary and our class only offered once every year, it has not been easy to collect survey data with more than 30% of student participation. In Winter term of 2016, 41% of students answered the school's survey, the highest for the class ever.

	Response Rate	Mean	Median	Std Dev
Effective instructor	41%	4.4	4.0	0.5
Used time effectively	41%	4.9	5.0	0.3
Helped me to learn	41%	3.9	4.0	0.6
Assignments helped learning	41%	4.6	5.0	0.5
Responded respectfully	41%	4.3	4.0	0.7
Communicates well	41%	4.1	4.0	0.6
Clear grading criteria	41%	4.7	5.0	0.4
Timely feedback	41%	4.6	5.0	0.5
Available office hours	41%	3.6	3.0	0.8
Would recommend course	41%	3.7	4.0	1.0

Scales:
5.0 - Strongly Agree; 4.0 - Agree; 3.0 - Neutral; 2.0 - Disagree;
1.0 - Strongly Disagree; -1.0 - Doesn't Apply;

Figure 5. Survey questions and student's feedbacks.

It seems to me that students who answered the survey were overall very positive about the class and their experiences during the term. Figure 6 offers a high level view on how we are doing comparing with the division and university. Since our Computer Science (CS) division does not have any department, the *Department mean* in Figure 6 represents all the classes offered by the division. Generally speaking, students give CS classes lower scores because our classes are considered as hard, and students receive lower grades than that from most of other divisions. This is clearly reflected in Figure 6 – the Computer Science division classes received lower scores than the university mean in every question! However, except one question, this class scored better than that of the division. In addition, this class scored better than the University mean in exactly 50% of questions.

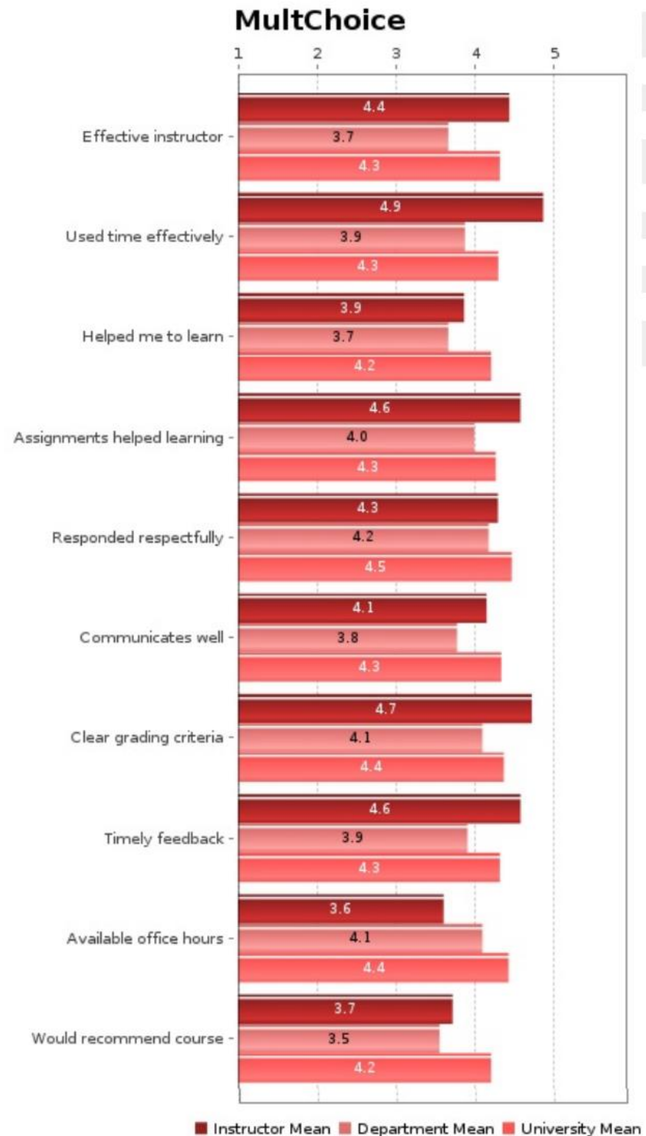

Figure 6. Comparing our class survey result with the division and university.

IX. CONCLUSIONS

Parallel programming classes not only introduce many innovative ways of solving well known problems, they also provide excellent opportunities for students to review and expend concepts in many Computer Science subject areas. We

believe that teaching PDC concepts to Computer Science undergraduates is both necessary and can be extremely beneficial to students' future growth. We have shared several of our approaches that make the class much more interesting to hope that more and more students can enjoy taking such a class. We also have presented much details of our Concurrent Systems class, where many PDC concepts are taught for more than 20 years.

With the right approaches, we managed to find resources to support several rounds of changes in hardware. We believe the class allows students to learn parallel computer organizations, study parallel algorithms, and write code to be able to run on parallel and distributed platforms. We also shared our plan for attracting more students to become interested in learning PDC concepts and parallel programming.

ACKNOWLEDGMENT

We would like to thank the WOU's Faculty Development Committee, NSF, and the division of Computer Science at WOU for supporting our efforts. We also would like to let our anonymous reviewers to know that we deeply appreciate their comments, corrections, and suggestions.

REFERENCES

[1] Quinn, Michael J. *Parallel Computing: Theory and Practice*. 2d ed. New York: McGraw-Hill, 1994

[2] Liu J. and F. Liu, Teaching Parallel Programming with Multi-core Computers, The 2010 Intl. Conf. on Frontiers in Education: Computer Science and Computer Engineering, July 12-15, 2010

[3] Brookshear, Glenn J. *Computer Science – and overview,* 10ed Addison-Wesley, 2009.

[4] www.dictionary.com

[5] http://hothardware.com/news/intels-72-core-knights-landing-xeon-phi-supercomputer-chip-cleared-for-takeoff

[6] https://msdn.microsoft.com/en-us/library/dd460693(v=vs.110).aspx

[7] Hurtgen , *Alyssa. High Performance Computing Cluster in a Cloud Environment*, https://support.rackspace.com/how-to/high-performance-computing-cluster-in-a-cloud-environment/, June, 2016

[8] Barney, Blaise. *Introduction to Parallel Computing* https://computing.llnl.gov/tutorials/parallel_comp/

[9] Morgan, Timothy. *Intel Knights Landing Yields Big Bang For The Buck Jump*, http://www.nextplatform.com/2016/06/20/intel-knights-landing-yields-big-bang-buck-jump/, June 20, 2016