Experiences on Teaching Parallel and Distributed Computing for Undergraduates

Erik Saule
Computer Science
UNC Charlotte
Charlotte, NC, USA
Email: esaule@uncc.edu

***Abstract*—The recent increase in interest on big data and data intensive computing makes it important for CS undergraduate students to receive education in Parallel and Distributed Computing. The increase in scope and popularity of a CS education often causes the majority undergraduate students to take a whole four year degree to really perceive modern computing challenges; it therefore poses new challenges in teaching parallel computing. At UNC Charlotte, the Parallel and Distributed Computing class is a required class for the Systems, Software, and Network concentration of the BS in Computer Science. As such, it poses particular challenges because the class sees high enrollment, a diverse body of student and is the last opportunity to ensure basic computing skills as all later classes are electives.**

This paper presents the design choices of this class which focuses on teaching parallelism as opposed to performance through analysis of parallel algorithms, parallel programming in different models, and scalability testing. The paper also presents a set of scaffolded assignments that leverage a PBS cluster for testing. We present feedback from teaching the class during the Fall 2017 semester. In particular we introduced a simple tool to help extracting dependencies on algorithms and compute critical path. And we present student suggestions of assignments that would lead to higher engagement.

***Keywords*-Parallel Computing; Undergraduate Education; Scaffolded Assignments**

I. Introduction

The rise of the Big Data era has increased the interest in parallel computing and high performance computing. While in the past only few scientists would run Big Compute jobs, nowadays most science and business is actually done by running either Big Compute or Big Data jobs. As such, the interest in educating students to the usage and fundamentals of parallel computing has increased. This drove the inclusion of parallel computing topics in the 2013 ACM/IEEE-CS curriculum guidelines for undergraduate CS degrees [TAI13]. Also the CDER center for parallel computing education proposed a detailed curriculum for integrating parallel computing earlier into undergraduate curriculum [NSF12].

Bringing parallel computing into the early years of a bachelor curriculum is seen as a difficult problem. There are few models on how to properly do the integration and there are few examples in the community of how parallel computing topics can be broken down in non-classical ways. For many institutions, the instructors or early classes may not be fully trained in parallel computing. Therefore there is a need for good exemplar of how to teach parallel computing and how to design good assignments.

This paper introduces the material used at UNC Charlotte to teach ITCS 3145 Parallel and Distributed Computing. While these topics are taught in a single junior level class, the class serves as the last required classes for many students. As such the class is designed to bring up to speed students that are lagging behind, such as the students that transferred from a different institution or the ones that retained less knowledge than is ideal from previous classes. Also the course comes after a combined data structure/algorithm class and a combined operating system/networking class. As such the algorithmic, low-level programming, and system skills of the students are not fully fleshed out and ITCS 3145 serves as a practical OS class.

Therefore, parts of this class could be used to integrate parallel computing topics into an OS class or into an algorithm class. Some parts could also be integrated earlier into a CS1 class. The material provided to the students during Fall 2017 for ITCS 3145 are available online [1] and the full material is available by request. This paper describes the rational upon which the class is built, how assignments are scaffolded, and lessons learned from teaching the class in Fall 2017.

Section II presents the typical population of students in the CS degree at UNC Charlotte and explains how ITCS 3145 integrates in the curriculum and the development of the knowledge and skills of the students. Section III details the structure of the class and how the different parts articulate to give the class a practical Operating System flavor. Section IV talks about the rational in using a PBS cluster for teaching the class and how the assignments are scaffolded as well as the rational for the type of scaffolding provided. Section V reflects on the class run in Fall 2017 and derives some insight about how the class could be improved. In particular a piece of software has been introduced to help teach dependency graph representation and algorithm engineering to extract more parallelism. Also students were encouraged to provide assignment suggestion and a few engaging suggestions are described. Section VI relates some previous works to our

[1] https://webpages.uncc.edu/esaule/classes/2017_08_ITCS3145

efforts. Section VII summarizes the contribution of the paper and provides some directions for future efforts.

II. Context and Goals of ITCS 3145

UNC Charlotte is an urban university and the largest in the greater Charlotte region. The Bachelor of Science degree in Computer Science is common to the entire College of Computing and Informatics. There were over 1800 majors in Computer Science at UNC Charlotte in Fall 2017. About a quarter of the students transferred from a community college. About 13% are African American, 7% are Hispanic, and 17% are female. Over 80% of the students work full-time or part-time.

All students in the Systems, Software and Network concentration are required to take ITCS 3145: Parallel and Distributed Computing. Systems, Software and Network is the most popular concentration which means that over 100 students take this class every year. This class is the last required class for all the students of the concentration; all further classes are electives. As such this class is tasked with making sure that outgoing students have solid programming and systems skills. (An other required class covers networking aspects.) Also this class comes after ITSC 3181 Computer Architecture and after ITSC 3146 Operating Systems and Networking.

Because early classes are taught in Java, students enrolling in ITCS 3145 Parallel and Distributed Computing often only had a single class in C or C++; which means that their low-level programming skills are basic. Also ITSC 3146 combines networking concepts with operating systems concepts. So the students only have developed a basic understanding of operating systems concepts and are still uncomfortable with using UNIX systems in practice.

As such, this class is tasked with 1) improving low level (operating system level) programming skills; 2) making sure students can survive in a UNIX environment; 3) belaboring concepts of programming abstraction; 4) introducing a parallel thinking mindset; 5) enabling the leveraging of parallel computing resource for simple strategies.

In particular, this class is not an High Performance Computing class where the finer details of code, algorithms, and processors and buses will be discussed; these topics are covered in a later elective class. This is also not a Parallel Algorithms (PRAM like), or even a Distributed Systems (consensus, election) class.

The author is not aware of a textbook that provides ready to run lectures and assignments that would match these objectives (and would be glad to have one). As such, the class has been custom built with a set of lectures and scaffolded assignments.

III. Structure of ITCS 3145

To accomplish the goals of the class, the class has to be assignment driven; so the students will do 9 assignments in the span of 16 weeks. In-class time is set aside to talk about the assignments in details and work through some of the simpler problems. Content-wise, the class is structured in four parts which are now described.

A. Getting Started

The first part is introductory and kick starts the students in their programming and in the usage of the PBS cluster used in the class. At that time, the students write a sequential numerical integration code. This code will serve as an example that will be followed in most of the class. Doing this first part sequentially enables catching up students whose C++ programming skills are not up to par; and students who are not used to using UNIX systems. It also builds up confidence in a simple code that will be made parallel later.

Using numerical integration as the first assignment is not an arbitrary choice. It is a computation that most students understand easily because they almost all have studied calculus by the time they reach ITCS 3145. It is a kernel that can easily be made more or less computationally expensive; but more importantly it is a compute-bound kernel, rather than a memory-bound kernel. Using a compute-bound kernel removes problems of memory contention that could cause speedup curves not to be linear and would trigger confusion in the mind of students. Memory contention is a processor design issue that is explored in detail in a later classes and is only briefly discussed in the middle of ITCS 3145 when memory bound kernel are used.

B. Algorithms in Parallel

The second part of the class is essentially an introduction to parallel algorithms. In particular, it is about representation of parallel algorithms. The students learn how to extract tasks and dependencies from a piece of sequential code. This is achieve by explicitly choosing tasks and analyzing read, and write dependencies to the variables of the application. Note that these steps are the same ones that would be covered in a compiler class for code analysis and instruction reordering. However it is unlikely students have taken a compilation course at this point of their curriculum. But it allows to give them an hint at the complexity of compilers. A similar process is also used in OpenMP when using the tasking clause `depend`.

Then tasks and dependencies can be analyzed to compute the work, the width, and the critical path of an application [BJK+95]. These metrics are popular in the field as they are used extensively by software such as Intel Cilk Plus [Int]. They enable discussing "how parallel" an algorithm is; especially when the analysis of the algorithm can be made for arbitrarily sized problems. This can help students get a practical understanding of why some codes do not scale and also serves as a reinforcement of Ahmdahl's law. The students also learn how to build a schedule out of a dependency graph and how greedy algorithms such as List

Scheduling [Gra66] or LPT [Gra69] can be used, and are used in practice.

Building dependency graph and schedule help students understand that some codes are inherently more parallel than others. More importantly, it gives the students the tools to understand why a piece of code is not parallel. Often a variable causes synchronization of many of the tasks. The students can then be introduced to some techniques to make algorithms more parallel, for instance by decoupling some of the operations in a way that is similar to loop splitting. Tasks that could happen in different orders but not at the same time can be used to tie back to mutual exclusion many students used in a previous operating system class. Reductions, divide and conquer algorithms, dynamic programming are all good examples to build the students' skills.

C. Shared-Memory Programming

The third part of the class presents shared-memory programming. Here we first present the `pthread` library. The purpose of talking about `pthread` is to remove the "magical" aspect of parallel programming. Since all programming models are essentially built upon `pthread` or a library of the sort, students learn the system aspects that relate to parallel computing. The students build loop schedulers (both static and dynamic) using `pthread` primitives. They use the numerical integration example to test their loop schedulers. This enables discussions about the granularity of parallel decomposition, but also locking granularity in the reduction of the calculation and the related overheads. One can also ask students to extract logs out of their scheduler to introduce system instrumentation.

Then a higher level shared memory programming model is introduced. OpenMP is a good candidate because of its availability in most compiler tool-chains. Also OpenMP supports both loop decomposition that the students built in `pthread`, but also support task decomposition that map to what the students learned in the algorithmic part of the class. Presenting OpenMP enables to talk about middleware in general and how compilers' API can help exposing simpler more abstract parallel programming interfaces. The students write an OpenMP version of the numerical integration problem and reflect on the simplicity of the writing compared to the `pthread` implementations. The existence of loop interfaces and tasking interfaces in OpenMP also allows to show tradeoffs between complexity of the programming models, expressible algorithms, and performance obtained using a single tool. This reduces confusions that would be introduced by having the student use multiple parallel programming middlewares. Also the simplicity of using OpenMP compared to `pthread` enables more challenging and realistic parallel computing problems such as parallel sorting or dynamic programmings.

D. Distributed-Memory Programming

The fourth part of the class is about distributed-memory computing. First, introducing explicitly communication cost is necessary to make a theoretical difference with shared-memory systems. This enables discussing different machine topologies and algorithms for classic problems such as reductions. Data distribution is emphasized using problems such as stencils operations, and matrix multiplication.

Then, MPI is introduced in a fairly typical way. Starting from point-to-point communication, the students will solve the numerical integration problem using both a static partitioning and a dynamic partitioning by building a master-worker model. This enables the student to observe speedups that were not possible in the shared memory system. Then, collectives are introduced and more advanced problems are solved such as stencil and matrix multiplication.

Finally, Map Reduce operations [DG08] are introduced using the MR-MPI library [PD11]. MR-MPI[2] provides Map Reduce features inside an MPI code. This new programming model and execution environment provides an other example of what middlewares can do to improve programmer productivity. At the same time, the students become familiar with a programming model that is popular in the industry.

IV. Parallel computing assignments and scaffolding

The core part of the ITCS 3145 class at UNC Charlotte revolves around students figuring out assignments. Programming exercises are often seen as way to engage students more in parallel computing [Bog17]. As such, the archive we released online contains the scaffolded assignments for other instructors to use.

A. System

The assignments are all to be done in C or C++ and to be run on a PBS cluster for testing. The university is running an educational PBS cluster separate from the research cluster in order to prevent educational users disrupting regular research activities. It is indeed common that students run code that crashes the head node of the cluster or bring compute nodes down by over allocating memory. The cluster is composed of 12 nodes, each equipped with 2 Intel Xeon processors featuring 8 cores each; for a total of 16 cores per node; and 128GB of memory. The nodes are interconnected using InfiniBand technology. Two of the nodes are equipped with an NVIDIA graphics card providing 4 programmable CUDA devices across the cluster. To minimize management overhead, the educational cluster uses the same software stack as the research cluster, which provides a realistic environment while consuming little additional manpower.

Pedagogically, having such a system enables students to be exposed to a real cluster environment. This bolsters their

[2]available online at http://mapreduce.sandia.gov/

UNIX skills which are important in today's industry. Also having 16-core nodes enables studying complex algorithms at a high core count. For instance, parallel prefix sum has an efficiency of $\frac{1}{2}$ which makes it hard to test and observe practical speedup on a student laptop which often only has one or two cores. Also, all students benchmarking their code on the same machine enables the comparison of speedup that they get without having to worry about discrepancies coming from the system they are using. Also for the distributed-memory part of the class, the machine is sufficient to solve problems that do not fit in the memory of a single node and to see speedup on simple problems.

The students are encouraged to solve first the assignments on their own machine. An Ubuntu Virtual Machine is provided to them with all the required software to run the class: a compiler with OpenMP 4.5, an MPI library, and MR-MPI. The students are encouraged to communicate code and results between the cluster and their VM using git. This has two advantages: 1) it teaches git to the students that have not been exposed to it; 2) when a problem occurs, it enables the student to point the instructor and Teaching Assistant to a complete version of the code, rather than providing a code snippet that may not exhibit the problems they are having.

While we understand that not all institutions have access to such computing resources, computing centers have been opening their clusters for educational purposes. Also, CDER makes available a SLURM cluster for educational purposes[3].

B. Scaffolding

The assignments themselves are scaffolded so that the students can focus on the tasks that matter to their understanding of parallel computing rather than on the tasks that are necessary and keep them busy but do not further their parallel computing skills. The assignments define command line parameters and output format. Often the answer to the problem is to be written on `stdout`, while the time it took to compute the answer is to be written on `stderr`. This enables the scaffolding to provide a lot of help so that the students only have to write the application code. Benchmarking is done by scripts that are provided to them. This frees the students from having to look deeply into designing and scripting complex scalability experiments, which is an error-prone and time consuming process (albeit educational). And the students can focus purely on parallel computing issues.

But more than just running the code, the scaffolding also automatically plots the charts that are meaningful for the students to look at. This also ensures that any student sees all the effects that are to be seen on a particular problem. Otherwise students might observe one of the strange effect, assume that is all there is to see and move on.

Additionally, the scaffolding enables automatically testing the codes for correctness. Provided computing resources are limited, it is helpful to be able to check whether a code is correct before a student runs a massive benchmark. Some of the benchmarks can take a couple of hours to run, and since the machine is small compared to the number of users it is preferable to avoid running incorrect code. Indeed, 12 nodes are supporting about 60 students in a typical semester. If each students waste 2 hours of computation (and they often waste more), it adds up to the equivalent of the cluster being unavailable for 10 hours the week the assignments are due.

The assignment scaffoldings are built so that students do not have access to some functions, they only receive a static library that will be compiled against. It enables the instructor to build functions that students can not change (easily). In particular, for the numerical integration assignment, the functions to be integrated are provided that way. The functions take two parameters `x` where the function is evaluated, and a parameter `intensity` which enables to change how long it takes to evaluate the function. Providing the function in a binary form rather than asking the students to come up with them enables to write unit tests, and to ensure that the implementation of intensity does not get compiled out. Indeed depending how the feature is implemented, the compiler could realize what is happening and simply optimize out parts of the function, so having a robust implementation is necessary.

Other use of static libraries as part of the scaffolding enables to provide instances of problems for which the solution is known and easily checked. For instance, in a sorting problem, giving a random permutation of all the integers between 0 and n enables a simple test for correctness and that the instance provided is not a worst case instance of the tested algorithm. Choosing the tested instances also helps with debugging codes. Picking the instance of a parallel prefix sum to be $1, -1, 2, -2, 3, -3, \ldots$ gives a very strong structure to the solutions which helps figuring out quickly where the students' code are incorrect.

C. List of assignments

Assignment 1 is a purely sequential assignment where the students learn to use the PBS cluster and solve the numerical integration problem.

Assignment 2 is an dependency extraction assignment where the student work through transform, reduce, find first, prefix sum, and merge sort.

Assignment 3 is a pthread assignment where the students implement a static and dynamic loop scheduler. The schedulers are used to solve the numerical integration problem.

Assignment 4 makes the students use OpenMP parallel loop construct to solve problems of reduction, numerical integration, prefix sum, and merge sort.

Assignment 5 also uses OpenMP but this time uses the tasking constructs to do reductions, merge sort, longest common subsequence, and bubble sort.

[3]https://grid.cs.gsu.edu/~tcpp/curriculum/?q=node/21615

Assignment 6 is a distributed memory algorithm assignment that makes the students think about data placement and network topologies implication by looking at problems of reductions, 1D heat equations, and matrix multiplication.

Assignment 7 uses MPI to perform numerical integration using a static work partitioning and a master-worker model.

Assignment 8 looks at more complex MPI problems with 2D heat equations and matrix multiplication.

Assignment 9 is an MR-MPI assignment which makes the students look at word counting.

V. Lessons Learned the Hard Way

A. On parallel data structures

The first version of the class included a lecture regarding parallel data structures. That lecture covered the basic aspects of tradeoffs between locking granularity and performance and went through the details of lock-free and wait-free data structures [FH07]. Understanding these more advanced concepts requires understanding advanced system issues such as instruction reordering by the compiler, out-of-order execution of assembly instructions, and memory barriers. These issues are just too complex for students to understand in a short amount of time. As such, covering lock-free and wait-free algorithm is judged counter-productive and should be reserved to a more advanced class.

B. On class management

Starting an assignment is typically hard for students. So students often realize late the assignment requires some preparation work that can not be slammed in an hour. The basic strategy was to use in-class time to make sure that students are properly set up. However, once students are late, they tend to use in-class time to work on their late assignments rather than on the assignment-of-the-day. Accepting late assignment is useful to struggling students and not accepting late assignment would mostly cause at-risk students to fail. A strategy being tested is to make sure most (if not all) assignments have a preliminary part which mostly consists in getting the simplest use-case running and is due the day after the lecture introduced it. Of course, no deadline extension are given on preliminary. That way, students that try to catch the preliminary deadline; even if they do not make it; will realize that there is more than meet the eyes and will be seeking help before the assignment deadline.

Teaching a large class that heavily relies on assignments means that students will need additional help. Even with two Teaching Assistants, students often will rely on email to seek help. While personalized communication is sometimes necessary, this model is not scalable on a 50+ student class. The instructor gets swamped in many emails often relating to the same topics. Now the class is taught in a way that encourages asking questions on the class discussion board (UNC Charlotte uses Canvas) and any questions asked by email that does not pertain to a particular student or which public discussion would violate FERPA laws is redirected to the discussion board.

C. On scaffolding

Scaffolding is great... when it works. Indeed, students really appreciated the scaffolding being able to tell them whether their code returned the correct solutions and being able to get plots automatically without having to think about it too much. However, more attention to the design of the scaffolding is necessary. In particular, it is necessary to make sure that unit tests actually catch all incorrect codes. If the test returns "OK", students will assume the code is good to be benchmarked. For instance, the testing method was not checking the format of time and many students added extra labels that broke the plotting scripts. Separating benchmarking scripts from plotting scripts can also enable student to only rerun the plotting part of the scaffolding without having to rerun the entire code.

Students often run code on the head node of the cluster which tends to crash the head node and makes the cluster unusable. Introducing checks in the scaffolding that queries the name of the machine and does not allow running on the head node can prevent many cluster crashes.

D. Extracting parallelism with par_graph_lib

Extracting dependencies and exposing more parallelism is definitely the hardest and most confusing part of the class for students. More detailed slides during the lecture were introduced to give a clear recipe of the steps to follow to go from code to a dependency graph. An additional in-class example is unrolled that scrupulously follow the steps of the recipe. Also many students still have some difficulties thinking recursively as the students at UNC Charlotte get a combined data structure and algorithm class and may not have enough experience with recursive algorithms.

To help students figuring out dependency structures, a library was developed and introduced. par_graph_lib[4] provides a simple interface to extend an existing sequential code to define tasks. Essentially the developer is responsible for defining where a task starts and what its name is. She also chooses a processing time for the tasks and explicitly declares variables accessed and whether the access is a Read, Write, or ReadWrite access. Based on that information, the software extracts the dependency graph. The graph is visualized, using an interactive web visualization tool designed for data structures classes called BRIDGES [BMG+16], and highlights the critical path of the application. An example of how the algorithm code looks like and the related visualization is displayed in Figure 1.

It is expected that students will study by working out a problem and confirming their intuition by implementing the algorithm and inserting the instructions necessary to generate

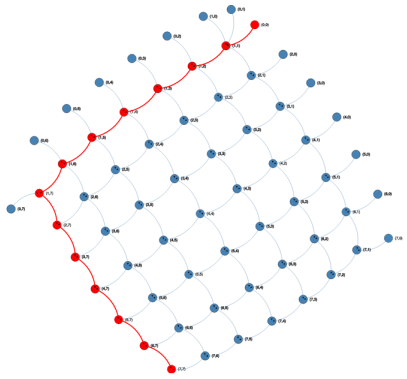
[4]https://github.com/esaule/par_graph_lib

```
stringstream ss;

for (int a=1; a <= m; ++a) {
  for (int b=1; b <= n; ++b){
    ss.str(""); ss<<"("<<a<<","<<b<<")"; newtask (ss.str());
    ss.str(""); ss<<"X["<<a-1<<"]"; read (ss.str());
    ss.str(""); ss<<"Y["<<b-1<<"]"; read (ss.str());
    ss.str(""); ss<<"C["<<a<<"]["<<b<<"]"; write (ss.str());
    ss.str(""); ss<<"C["<<a-1<<"]["<<b-1<<"]"; read (ss.str());
    ss.str(""); ss<<"C["<<a<<"]["<<b-1<<"]"; read (ss.str());
    ss.str(""); ss<<"C["<<a-1<<"]["<<b<<"]"; read (ss.str());

    if (X[a-1] == Y[b-1]) {
      C[a][b] = C[a-1][b-1] + 1;
    } else {
      C[a][b] = max(C[a][b-1], C[a-1][b]);
    }
  }
 }
```

(a) Sample code for solving the Longest Common Subsequence problem using dynamic programming



(b) Sample visualization of the Longest Common Subsequence problem

Figure 1. Sample code that show how a Longest Common Subsequence code is annotated and the graph generated by the BRIDGES. Only the main part of the code that builds the dynamic programming matrix C by comparing string X and Y are shown. Complete code available at https://github.com/esaule/par_graph_lib/blob/master/samples/lcs.cpp. The visualization is available at http://bridges-cs.herokuapp.com/assignments/102/esaule

the graph. A few examples are also provided with the library to be basic examples that the students can study.

E. On engaging assignments

While gathering feedback from students during the course, it appeared that engagement was something that could be improved. Indeed, many students reported that integrating the same function multiple ways was not particularly engaging.

The students got offered the opportunity to propose their own assignment for the class and to complete the assignment they proposed. We now report on a few of the suggestions that were submitted by students.

The first project is about implementing a textual similarity query engine that interfaces with Twitter. Twitter exposes a fairly simple API that enables implementing a Twitter bot easily. The bot would listen to the message received by the associated Twitter account which also follows particular twitter handles or hashtags. Then the user could submit a query to the bot by directly addressing the bot's account. The bot would reply by returning the tweet it stores that is the *closest* to the query. The definition of *closest* could be left for the students to figure out, in order to make the assignment more exploratory. Or some simple measures such as the cosine similarity of word occurence vectors could be used if students need more directions. Students are at this point unlikely to think of complex spatial algorithms such as generalization of quad-trees. The expected implementation would identify the tweet the most similar to the query by using a simple loop over all the stored tweets. This could be parallelized easily using an OpenMP for-loop construct. Or a distributed memory assignment could be constructed by distributing the stored tweets to different nodes.

Role playing games can have complex dice throwing methods which can be hard to analyze on paper. An assignment was proposed to obtain a probability density function of a particular dice combination. Some games use dice of different number of faces (most common dice have 6 faces, but some 4 faces, 10 faces, or 20 faces dice are also often used). Also some games have "exploding" dice which when the die get its highest number is re-rolled and the number is added. Similarly, rolling the lowest value on the die causes the die to be re-rolled negatively. Meanwhile if a die gets its highest value while an other die in the same roll get its lowest value, neither dice "explodes". This process is recursive and there are many rare events that can occur. Obtaining a probability distribution that accurately takes into account rare events requires a high number of Monte-Carlo iterations which makes the problem computationally expensive. On the parallel computing side, this assignment poses problems of generating independent random number in parallel and could be done either in shared memory (rembering that `rand` is not reentrant) or distributed memory.

Classifying text is a common Natural Language Processing task for which a popular dataset has been made publicly

available, the 20 newsgroup dataset [5], which contains about 20,000 texts extracted from 20 different newsgroups. A simple way to cluster the texts can be obtained by first computing the cosine similarity on word occurrence vector of every pair of texts; and then running a traditional nearest neighbor algorithm to derive clusters. The computation of the similarity is likely expensive as it is quadratic in the number of texts and on the 20 newsgroup dataset requires computing 200 million text cosine similarity of high-dimensional sparse vectors. Of course, better algorithms exist [AK15], but students are unlikely to know them. It is also easy to scale the size of the problem by ignoring some of the texts.

Other interesting submissions of assignments but that seemed harder to deploy in class included a raytracer, computing covariance of the columns of a large database, a sudoku solver, a password cracker, an image distance calculation using color histograms, and collision computations in a video game.

VI. Related Works

An instructor at TACC suggested MPI should be taught through mental models [Eij16]. The idea is mostly that the classical way of teaching MPI from point-to-point communication and talking about deadlocks and inefficiencies before talking about collective is an historically oriented way of teaching MPI. But really, it makes more sense to teach process symmetry first and introduce collective as they maintain process symmetry. Then harder concepts that break the symmetry can be introduced like rooted collectives, and point-to-point communications. The rational of [Eij16] appears to make sense and ITCS 3145 will probably see its distributed memory part revamped to include this development. It should be noted that MR-MPI does not break process symmetry.

Rice University pioneered tools to teach parallel computing [GAC+17]. In particular their Habanero system provides an auto-grader for parallel codes. Essentially, the students are completely abstracted from the system which runs the students code. Habanero checks correctness of the codes but also automatically compute performance charts. The idea is to provide automated feedback and grading of students assignments. The Habanero system is certainly valuable, however the system seems fairly complicated to deploy and completely shields the user from system issues which may not be best in all context. Yet the idea is laudable, and similar ideas percolated in ITCS 3145 under the (simpler) form of unit test and scaffolded benchmarks.

VII. Conclusion

Teaching parallel computing to undergraduate students has shifted from being an arcane specialty class to being required in many programs; with strides to include parallel computing constructs in classes as early as CS1. However a common barrier to a broader adoption of parallel computing in CS education is the lack of existing models and materials for teaching such a class. In this paper, we present the rational behind the design decision of ITCS 3145 which serves as a practical OS class.

The class however is fairly modular and parts could be reused in other (earlier) classes. Also to promote the reuse of assignments across institution and to build better assignments banks, we released the scaffolded assignment as they were given to the students of ITCS 3145. The design decisions behind the scaffolding are provided. And we certainly hope that other instructors will adopt our assignments and will propose new assignments using a similar model. We believe that making available more assignments and material will contribute to a wider adoption of parallel computing in various classes.

Running the class in Fall 2017 highlighted some shortcomings. In particular we plan on improving the communication of dependency graph extraction using our newly designed library. Also we plan on improving the engagement of students by investigating the use of more current assignments; we received few suggestions from students that are described in this paper.

[5] http://qwone.com/~jason/20Newsgroups/

Acknowledgment

The author would like to thank Abhishek Chandratre and Vivek Soni who were the Teaching Assistants of the ITCS 3145 class during the Fall 2017 semester. This material is based upon work supported by the National Science Foundation under Grant CCF-1652442 and DUE-1726809.

References

[AK15] David Anastasiu and George Karypis. L2knng: Fast exact K-Nearest Neighbor graph construction with L2-norm pruning. In *Proceedings of the Conference on Information and Knowledge Management (CIKM)*, 2015.

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, 1995.

[BMG+16] David Burlinson, Mihai Mehedint, Chris Grafer, Kalpathi R. Subramanian, Jamie Payton, Paula Goolkasian, Michael Youngblood, and Robert Kosara. BRIDGES: A system to enable creation of engaging data structures assignments with real-world data and visualizations. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE)*, pages 18–23, 2016.

[Bog17] Steven A. Bogaerts. One step at a time: Parallelism in an introductory programming course. *Journal of Parallel and Distributed Computing (JPDC)*, 105:4–17, 2017. Keeping up with Technology: Teaching Parallel, Distributed and High-Performance Computing.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Communication of the ACM*, 51(1):107–113, January 2008.

[Eij16] V. Eijkhout. Teaching MPI from mental models. In *2016 Workshop on Education for High-Performance Computing (EduHPC)*, pages 14–18, Nov 2016.

[FH07] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems (TOCS)*, 25(2), May 2007.

[GAC+17] Max Grossman, Maha Aziz, Heng Chi, Anant Tibrewal, Shams Imam, and Vivek Sarkar. Pedagogy and tools for teaching parallel computing at the sophomore undergraduate level. *Journal of Parallel and Distributed Computing (JPDC)*, 105:18–30, 2017.

[Gra66] Ronald L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[Gra69] Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, March 1969.

[Int] Intel. Intel Cilk Plus. https://www.cilkplus.org/.

[NSF12] NSF/IEEE-TCPP Curriculum Working Group. NSF/IEEE-TCPP curriculum initiative on parallel and distributed computing core topics for undergraduates. Technical report, IEEE-TCPP, 2012. available at http://www.cs.gsu.edu/~tcpp/curriculum/sites/default/files/NSF-TCPP-curriculum-version1.pdf.

[PD11] Steven J. Plimpton and Karen D. Devine. MapReduce in MPI for large-scale graph algorithms. *Parallel Computing (ParCo)*, 37(9):610–632, September 2011.

[TAI13] The Joint Task Force on Computing Curricula, ACM, and IEEE Computer Society. Computer science 2013: Curriculum guidelines for undergraduate programs in computer science. Technical report, ACM, 2013. available at https://www.acm.org/education/CS2013-final-report.pdf.