Paper ID #35221

Improving Student Outcomes with Final Parallel Program Mastery Approach for Numerical Methods

Dr. Sam B Siewert, California State University, Chico

Dr. Sam Siewert has a B.S. in Aerospace and Mechanical Engineering from University of Notre Dame and M.S., Ph.D. in Computer Science from University of Colorado. He has worked in the computer engineering industry for twenty-four years before starting an academic career in 2012. Half of his time was spent on NASA space exploration programs and the other half of that time on commercial product development for high performance networking and storage systems. In 2020 Dr. Siewert joined the California State University to teach computer science and software engineering as full time faculty and retains an adjunct professor role at University of Colorado Boulder and Embry Riddle Aeronautical University. Research interests include real-time systems, machine vision and machine learning. Dr. Siewert was a co-founder of the Embedded Systems Engineering program at University of Colorado.

Improving Student Outcomes with Final Parallel Program Mastery Approach for Numerical Methods

Sam Siewert
California State University
400 W. First St.
Chico, CA 95929-0410
530-898-4342
sbsiewert@csuchico.edu

Abstract

Student final parallel programs for "Numerical and Parallel Programming", a course taught at California State University Chico, stress ability to put theory into practice. The course normally includes parallel programming, allowing students to assess their success with verifiable numerical methods (e.g., using MATLAB) and speed-up compared to Amdahl's law. The pandemic presented new challenges for final parallel programs, and this course had the extra challenge of proprietary cluster software, now only remotely accessible. The course has a reputation for being difficult based upon pre-course surveys. Given the limitations, it would have been easiest to simply eliminate the complex parallel programs and focus on simpler exercises and assessments.

However, based upon fall 2020 student mid-term surveys, it was clear that students preferred a final parallel program over online assessment. Further, the COVID-19 remote learning constraints presented an opportunity for change. This remote scenario along with large class size of 40 students prompted the goal to allow all students to share their work with their cohort in a new way. The final parallel programs for this course focus on numerical methods commonly used in science and engineering, traditionally summarized and/or presented one-on-one with faculty as individual efforts. Students are expected to use divide-and-conquer approaches to design parallel programs for speed-up using well-known numerical methods from calculus combined with algorithms learned in class. To maintain course learning objectives and improve upon them while overcoming the new pandemic limitations, three specific parallel programming modifications have been made:

1) All final shared memory parallel programs can now be completed on a home system using equipment that costs less than one hundred dollars or can be completed on the existing CSU cluster. Distributed memory parallel programs must be developed and tested on the CSU cluster using MPI based on cost of the system (thousands of dollars).
2) Final parallel programs can now focus on mastery of a prior problem given as an exercise with emphasis on a detailed code walk-through for parallel design.
3) All final parallel programs are required to include a brief report, but also a 30-minute video of the build, run, and code design completed with a detailed walk-through shareable with the CSU learning management system.

Longer term, CSU Chico is investigating a remotely accessible cluster built using the same at-home hardware but scaled to 60 nodes. Much like a musician mastering their art would want their own instrument and access to more exotic orchestra instruments, students mastering

programming benefit from their own home Linux systems that support parallel programming and multi-core Linux clusters provided by the university. The goal is to support not only POSIX threads and OpenMP shared memory scaling[1], but also distributed memory MPI (Message Passing Interface)[2,3,] and shared memory CUDA (Compute Unified Device Architecture)[4]. In general, the at-home use of single node versions showed no new issues compared to use of the proprietary software.

Based on fall 2020 results, the final parallel program video appears to have positively motivated students and gave them multiple options to engage and complete the course. While not as ideal as all students presenting to each other, the new approach scales well for a large class, and allows students to share their experience. The pedagogical experiment introduced by this new final parallel program for mastery shows promise, but results are preliminary. The fall 2020 results from pre-course, mid-course, and post-course surveys are summarized in this paper along with instructor lessons learned and plans to repeat in spring 2021.

1. Introduction

The Computer Science course, Numerical and Parallel Processing, is a senior year course required of all undergraduate majors and meets key program learning outcome goals for mathematics and computer science. Specifically, for mathematics, the course requires students to apply calculus, linear algebra, and discrete math skills. For the computer science learning objectives, the focus is on comparison of algorithms and design for shared memory and distributed memory speed-up of those algorithms.

The students consider this class to be a significant challenge based upon the complexity of assignments and the difficulty of combining knowledge from three areas:

1) Mathematics, with application of numerical methods for computation.
2) Programming, with iterative and recursive methods and knowledge of complexity.
3) Concurrency, with methods of threading in software and knowledge of hardware provided parallel execution.

The course overall is intended to reinforce knowledge of mathematics, specifically calculus and discrete math, while introducing students to vector, matrix methods for linear systems and data processing. At the same time, students are expected to use programming skills from prior coursework and combine that with new methods of parallel programming introduced in the course.

The strategy has been a balance of coverage of the three topic areas, with emphasis on applications that benefit from the combination of the three. Students are shown the value of this approach to scientific computing, high performance computing and emergent application areas such as computer vision. Through a series of challenging exercises (six in total), they are introduced to combined numerical, parallel problems and methods to speed-up programs. The exercises include simple problems, but each problem set also has a final problem that is a significant challenge to implement and realize speed-up. Students are challenged at the end of the course to pick a program to design or re-design and to show significant speed-up, comparing results to Amdahl's law[5,6], based on parallel hardware used. Modern systems hardware can make selection of the appropriate value for scaling factor used in Amdahl's law, "S", a non-trivial decision. The value for S can be based on SMT (Simultaneous Multi-Threading), super-scalar multi-instruction features, vector instructions, and the potential to use co-processor cores. Generally, in this course the goal is to keep the scaling factor S simple, with focus on the number of cores per node (ignoring SMT) and the number of nodes in a cluster. However, this simplification can be pessimistic given micro-parallel features most CPU systems now incorporate by default[7,8,9]. Based upon micro-parallel features and node scaling with MPI where memory and cache are scaled in addition to number of cores, some students see super-linear results, which exceed the speed-up expected. More time is needed by students to master their understanding of speed-up obtained.

To deal with this challenge and to ensure that distance requirements of COVID-19 do not dilute this final parallel programming effort, students were asked to produce a short 30-minute video, explaining their design, their code, and the speed-up attained. The video presentation allows all students to share their final parallel program work in good detail, to view each other's outcomes, and for the instructor to hear their story as well as reading a report. This paper shares this experience and presents preliminary results that characterize the pedagogical challenge of this course and potential advantages and disadvantages of recorded presentations compared to alternatives (written, presentation, traditional final exam).

2. Course Structure and Challenges

Numerical and Parallel Programming is designed to leverage student programming skills obtained in three previous courses, to build upon calculus and discrete mathematics, and to introduced significantly new numerical methods and parallel programming for shared memory and distributed memory systems. The class is a challenge well suited to interactive teaching methods with problem-based and active learning approaches that enhance student engagement[10,11].

Students are shown two shared memory parallel programming methods. The first is OpenMP, a method that uses compiler pragmas (directives) to generate concurrent threading programs for loop bodies and functions, with a high level of abstraction. While this approach makes adaptation of sequential algorithms to parallel simpler, it is however opaque, and exactly how the compiler generates the parallel code is not directly studied. This approach allows students to think of parallelism at a block level, e.g., one thread per loop iteration, or multiple threads executing the same block or function with different parameters, that must be later combined. The second method is POSIX threading, known as "Pthreads"[12], which requires students to more directly determine how threads will divide work, execute in parallel, and combine results, with very explicit thread creation, control, and synchronization. Both methods allow for shared memory scaling, where scaling is determined by the parallel hardware (processor cores and thread microarchitecture) on one computing node.

Students are also introduced to cluster scaling, and programing for distributed memory parallel processing using MPI (Message Passing Interface). The MPI programs can execute on a single node in distinct process address spaces as well as on physically distinct address spaces on different computer nodes that are networked together. A key learning objective is to clearly understand the differences, advantages and disadvantages of distributed memory scaling compared to shared memory. Students are provided strategies and methods to time code sections, to analyze code tracing and profiles, and to determine the speed-up that can be achieved for a range of algorithms that vary from easy to make parallel (embarrassingly parallel, e.g., thread gridded image transformation) to very difficult to make parallel (algorithms with data dependencies and requirements for data access and resource locks). The distributed approach has an advantage of node scaling, where the scaling factor includes memory, cache, and processing for each additional node, but with overhead of message passing. By comparison, shared memory has lower overhead for data sharing, but more potential pitfalls associated with race conditions and data integrity.

The calculus and discrete math skills required have been previously learned by students in prerequisite courses and linear algebra is introduced in the course, but leverages common algebra, and furthers knowledge with vector, matrix approaches to linear systems. The calculus focuses on numerical methods of integration, finite difference methods, and comparison to exact solutions as well as simulation of systems where exact solutions are not known. For both calculus and linear systems, tools such as MATLAB are used[13], and students are encouraged to check their work done by hand and implemented as programs with MATLAB and other online "cloud-based" math tools[14].

For all problems, students start with a sequential numeric program and verify correctness by comparison to known and sometimes exact solutions. For example, the area under a sine function of amplitude one over the 0 to π interval is known to be 2.0 exactly, provable geometrically and known as a calculus antiderivative. Methods such as a simple Riemann sum, trapezoidal approximation, Simpson's rule, and others are then used to show that the numerical methods for approximation can be compared to definite integral solutions to gain insight into the accuracy and precision of these methods[15]. Students should be familiar with definite integrals such as the area of sin(x) over the interval x=0… π.

$$\int_0^{\pi} Sin(x)dx = -Cos(x)]_0^{\pi} = 2.0$$

The above can also be graphically proven by considering the velocity of a shadow cast vertically (vertical projection) on the diameter of a unit circle, where-by the velocity of the projection as it traces a unit circle is the integration of sin(x) over the interval x=0… π, and the distance from one side of the unit circle to the other is exactly 2.0. In other words, students are encouraged to use mathematically observable facts to verify numerical methods and to assess accuracy of numerical methods for any precision (e.g., double precision floating point).

While calculus and discreate math are previously studied, linear systems are known from algebra and solutions for simultaneous equations used in calculus and physics. A major goal for the numerical learning objectives is demonstration of the value of approximations and numerical methods for applied math uses such as image processing and solutions for simulations of realworld problems from physics and engineering. The mathematics introduced in earlier courses are reinforced and emphasized with simulation and applications such as image transformation. These objectives require critical thinking, where students not only recall mathematical methods and programming methods, but must also apply them, evaluate methods of making the programs parallel, and create an overall solution.

3. Course Learning Objectives

The key program learning objectives for the course used to assess pedagogical goals include:

1) Application of computer science theory and software development fundamentals to produce computing-based solutions.
2) Use of divide-and-conquer-algorithms for computation with identification of practical examples where this can be applied.
3) Reinforcement of mathematics previously studied (calculus and discrete math) and introduction to vector, matrix mathematics applied to data processing and linear systems.

Based on Bloom's cognitive dimensions[16], this is an ambitious goal that stresses higher level reasoning to create, evaluate, analyze, and apply math and programming in a new way with the parallel programming methods introduced in this class.

The most significant practical challenge of this course is that it is taken senior year, after a sequence of three programming courses (CSCI 111 – Programming and Algorithms I, 211 – Programming and Algorithms II, 311 – Algorithms and Data Structures), completion of mathematics prerequisites, and operating systems. Most students are not in the mood for a final challenge of this magnitude their senior year. So, the strategy taken to keep students engaged is to show students the value of the knowledge and the power of these methods to solve high impact problems. Further, the course provides evidence that this knowledge is useful for industry, for scientific and high-performance computing, and emerging applications, such as computer vision.

4. Course Challenges

Given the combined application of math, programming, and newly learned parallel processing, many students are intimidated and sometimes overwhelmed. Often the class is split on challenges such as mathematics, concurrent programming methods, and programming in general. The pre-class survey includes questions on prior knowledge on a Likert scale[17], and very often the class is split as can be seen from student answers to the following questions. Table 1 shows that perhaps much of the perception that this course is hard is based upon lack of familiarity with calculus used. Table 1 shows that 50% of students disagree or strongly disagree that they are familiar with the mathematics they will need to succeed in this course. Only 37.5% believe that they are familiar with mathematical methods that will be used. These statistics indicate the value of more formative learning, allowing students more time to re-learn math skills prior to summative assessment of them. The skills must also be combined with new methods of programming. Given this extra challenge, where students must combine math learned much earlier (typically freshman and sophomore year) with advanced programming in their final semester of their senior year, it is not surprising many students worry about succeeding in this course.

Table 1. Pre-Course Student Familiarity with Calculus Used in Course

I am familiar with numerical integration and difference equations.

40 students polled	**Percent Answered**
Strongly Agree	8.3%
Agree	29.2%
Neither Agree nor Disagree	12.5%
Disagree	33.3%
Strongly Disagree	16.7%
Unanswered	0.0%

While all students have studied integration and differentiation, they are split on whether they have familiarity with numerical integration and difference equations, with more than a third agreeing that they are familiar, half in disagreement, and a small fraction undecided. This may in part be due to terminology unfamiliar to them, but perhaps also based upon prior struggles with mathematics.

5. Course Adjustment for Distance Learning (COVID-19)

Prior to the 2020/2021 academic year, taught entirely via video conference for Computer Science students at California State University Chico, Numerical and Parallel programming had the following formative and summative learning assessment strategy[18, 19, 20].

1) Seven quizzes on basic concepts and methods – 20% of total grade
2) Five major programs – 40% of total grade
3) A two-hour final exam covering all learning objectives – 30% of total grade.
4) Attendance and participation – 10% of total grade

The final exam and the challenge of the programing assignments resulted in an unofficial reputation for the course as a "difficult final hurdle for graduation", and many students appear to have delayed taking the course based on this reputation. The department encouraged re-design of the course in 2020/2021. With COVID requiring distance teaching, the re-structuring was based upon student feedback and a more formative approach for learning with programming exercises and quizzes.

With higher stakes assessments such as mid-term exams and most of the summative assessment deferred and determined by a final parallel program. This final parallel program is not a new "project" and can be refinement for mastery of an existing programming problem. For many students, they have indicated that they have not fully understood or completed exercises on their first try, earlier in the semester. The new strategy is summarized as follows:

1) Five low-stakes quizzes to provide formative attainment of key concepts and methods. Quizzes are timed, making look-up while completing difficult, questions are randomized, but answers shown, and students can re-attempt. Scores are averaged and students are expected to achieve 100% accuracy on their last try. Quizzes are given every third week during the semester – 15% of total grade.
2) Six major programming exercises, with challenge such that the last problem is not expected to be fully solved by most of the students. These are given every 2 weeks and due after non-quiz weekends – 30% of total grade.
3) One mid-term exam with focus on parallel programming methods and numerical methods at a fundamental level – 15% of total grade.
4) A high stakes final parallel program, with detailed design and speed-up analysis, presented as a design, code, and result walk-through video – 40% of total grade.

The re-design was based upon formative and active learning principles where students can attempt hard problems early. Based on their challenge experienced, they can then choose a program to re-attempt as they learn more. This re-attempt allows students to bring questions to class for problem-based learning and hands-on programming practice with the instructor. The re-design was based upon unanimous concern about online programming exams and a final exam with programming.

Another key pedagogical theory employed is based upon low stakes grading early. Students are encouraged to take on hard problems and analyze, evaluate, and create parallel programs that will provide significant improvement over sequential programs solving the same problem. Very often, students will achieve success in one or two of the three learning objectives for parallel programs:

1) Correct numerical methods with verified mathematical accuracy and precision.
2) Correct C and C++ programs that run without error.
3) Parallel programming that provides significant speed-up on parallel hardware.

The value of the course re-design to de-emphasize the exam-based programming frees up time for students to repeat a programming problem with greater mastery (speed-up achieved, numerical accuracy and precision, algorithmic correctness, and efficiency). The value of this approach is reflected in their mid-course survey from fall 2020. Students had a negative reaction to the word "project" and preferred the idea of more singular pursuit of mastery of a program, not a semester-long project.

The mastery approach allows students to re-attempt a problem from earlier exercises, completed, and explained more fully than time allowed during their first attempt. The elimination of a final exam and/or final project, frees up time as well at the end of the semester. Time regained has been used for more formative learning exercises such as the quizzes. Based upon polling in the fall 2020 semester, students found the quizzes useful by clear majority, summarized in Table 2.

Table 2. Mid-Course Student Sentiment Regrading Quiz Style Used

I have found the take 3x and average quizzes to be helpful for achieving learning objectives.

40 students polled	**Percent Answered**
Strongly Agree	58.8%
Agree	29.4%
Neither Agree nor Disagree	5.9%
Disagree	0.00%
Strongly Disagree	0.00%
Unanswered	5.8%

Table 3 shows that while formative exercises are time consuming and challenging for students, the majority found them useful for achievement of course learning objectives.

Table 3. Student Perceived Value of Assigned Exercises to Learning Objectives

I have found the assignments to be interesting and they have helped me to learn programming methods (e.g., MPI, Pthreads, OpenMP) as well as numerical methods and processing.

40 students polled	**Percent Answered**
Strongly Agree	29.4%
Agree	29.4%
Neither Agree nor Disagree	29.4%
Disagree	11.8%
Strongly Disagree	0.0%
Unanswered	0.0%

Finally, Table 4 shows that a strong majority of students "liked" the idea of re-attempting a problem previously encountered. While some students liked re-attempting a parallel program thinking that this is easier than a new project, the final program analysis and requirement to explain fully with a video walk-through increases challenge. The 30-minute video seems long, but this affords students ample time to walk through and explain every line of their code in good detail. The ability to explain code in good detail helps to substantiate that the students wrote the code, given their ability to fully describe it. Ideally these videos would not only be made available to other students for viewing, but also for rating and providing peer reviews. This was not done due to lack of time in 2020.

Table 4. Student Perceived Value of Final Parallel Program

I like the idea that the final project is my choice in terms of topic and that I can re-do and improve a prior assignment problem OR choose any creative programming problem I wish that requires parallel programming, demonstrates speed-up, and allows me to explain my code in good detail.

40 students polled	**Percent Answered**
Strongly Agree	52.9%
Agree	35.3%
Neither Agree nor Disagree	5.8%
Disagree	5.8%
Strongly Disagree	0.0%
Unanswered	0.0%

6. Course Re-Design Hypothesis

The hypothesis for the course re-design with no final exam and/or final project, is to instead require mastery of a final parallel program - an idea that was well received by students, with unanimous agreement by ZOOM interactive poll in fall 2020. Students felt that an exam was not a good final assessment of learning, and the majority in fall 2020 supported having a choice between re-doing and mastering a prior problem or choosing a creative programming problem. As this approach was determined by a ZOOM poll, these questions were again repeated in a new documented Likert survey poll in spring 2021. The pedagogical theory is that by delaying final assessment with a high stakes event announced way in advance, students will achieve learning objectives earlier, with more challenging problem-based learning first, and that they will do better overall in attainment of learning outcomes at the end.

To ensure that the high-stakes final parallel program is in fact mastered, students are required to create a video to be shared with the instructor and all other students via CSU's learning management system (Kaltura and Blackboard). Emphasis is placed upon design quality, code quality with a required walk-through of the design, code, and speed-up results.

Students are asked to analyze their design and program performance two ways. The first way is to measure speed-up, using scaling factor "S" based on number of nodes and cores per node, and then solve for "P", the parallel portion of the program, and "(1-P)", the sequential portion of the program. This approach using measured speed-up is an indirect method to estimate how much parallelization their design provides. Time is measured for the parallel and sequential version of the program to determine Speed-up actual below.

1) Speed-up actual = $T_{\text{sequential}} / T_{\text{parallel}}$, e.g., 3.0, with S=4 for a 4-core system
2) $\frac{1}{(1-P)+\frac{P}{4}} = 3.0$, so $3.0\left[(1-P)+\frac{P}{4}\right] = 1.0$, or $3.0 - \frac{9P}{4} = 1.0$, $\frac{9P}{4} = 2.0$

3) $P = \frac{8}{9}$ = 88.88%, (1-P)=11.11%

Second, they are asked to examine their code and estimate P and (1-P) based upon the threaded section of code and non-threaded using compiler assembly output instruction count. Based upon P and (1-P), the expected speed-up can be computed, given S and Amdahl's law.

1) Amdahl's Law = $\frac{1}{(1-P)+\frac{P}{S}}$, if P=0.8, S=4, (1-P)=0.2

2) Speed-up = $\frac{1}{0.2+\frac{0.8}{4}}$ =2.5x

Students observing anything that was above linear speed-up, super-linear, must justify their measurements based upon memory, cache, and other node scaling advantages for MPI. Having previously developed and tested programs, most students can leverage sequential numerical programs from prior exercise work and focus intently on methods of parallelization. Making an existing program faster with parallel programming is a major course learning objective along with methods to determine the accuracy and precision of results.

Based on COVID-19 restrictions, all students have had to either access the California State University Chico cluster remotely and/or use home equipment. Home systems can be used for shared memory parallel programming (OpenMP and Pthreads) and even at-home clusters using Raspberry-Pi[21] to support students who want to better understand the hardware. This has been largely successful, and while students can just use the remotely accessible university hardware, most have interest in building their own at-home solutions too.

In fact, the Computer Science department is adding two new Raspberry Pi clusters for remote access and a Jetson Nano cluster as well to replicated what students can build at home. Open-source tools for parallel computing are widely available, with OpenMP and Pthreads supported by GNU tools and Linux by default, and MPI supported with open-source tools for small clusters[22], including the newer OpenHPC tools[23].

7. Course Re-Design Formative Problems

Up front hard problems are a key to the hypothesis that problem-based learning with a high-stakes assessment at the end for refinement and improvement of solutions has advantage over timed exams. The course has six programming exercises with the following summary of challenge by learning objective noted in Table 5. Students have more time during office hours to get help with exercises they found most challenging with the re-do for mastery approach. Many of the problems involve simulation and algorithms that are not easy to make parallel. The challenge most often is due to data sharing and data dependencies inherent in the problems such as physical simulation. Tackling simulation and more involved numerical applications rather than exercises allows students to understand that not all problems are "embarrassingly parallel". The challenge of the problems that students attempt and can later re-do for mastery include programming, numerical methods, and parallel programming for each problem described in Table 5 is significant.

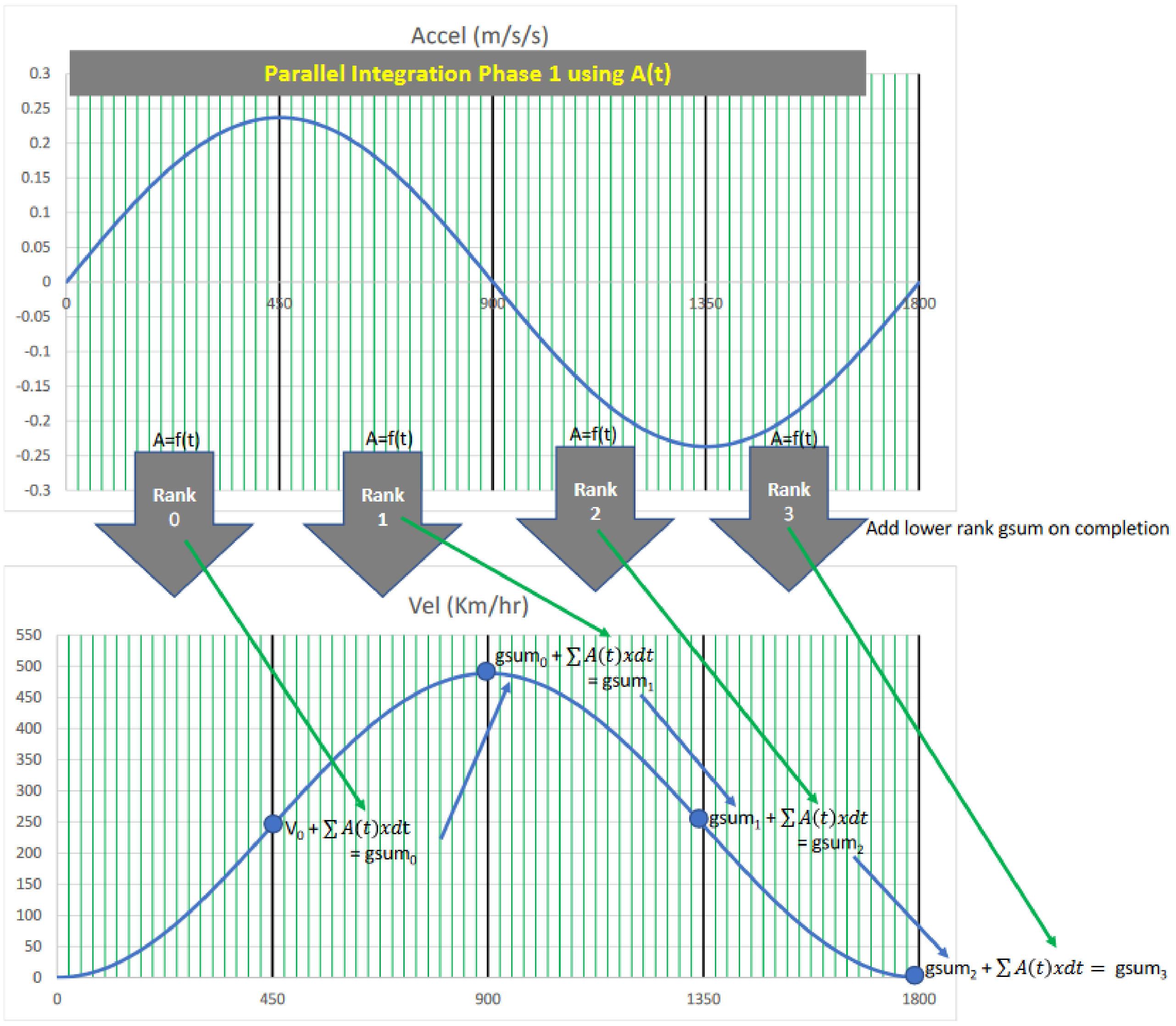
Table 5. – Six Parallel Programming Exercises that Deal with "Hard" Problems

Exercise	Description	Programming	Numerical	Parallel
1	Image transformation with PSF convolution and Discrete Cosine Transform	3D color image arrays, addressing and application of pixel neighborhood operations	Convolution, transcendental functions, inverse, and forward transformation (lossy, lossless)	OpenMP and Pthreads, thread gridding and thread blocks
2	Image transformation (rotation) and prime number theorem	Vector, matrix operations, large numbers, primality testing, and search	Vector matrix operations and prime number theorem	OpenMP and Pthreads, thread gridding and locks for critical sections
3	Simulation of a train trajectory given **linear** acceleration profile	Simulation and analysis of physical problems	Numerical integration, accuracy verification, and precision	Pthreads shared memory scaling and MPI distributed memory node scaling
4	Simulation of a train trajectory given **non-linear** acceleration profile	Simulation and analysis of physical problems	Numerical integration, comparing methods, accuracy verification, and precision	Pthreads shared memory scaling and MPI distributed memory node scaling
5	Linear systems solvers for engineering systems	Iterative (e.g., Gauss-Seidel and vector, matrix row operations for linear systems solving (LU decomposition)	Iteration and automation of LU decomposition with analysis of accuracy and issues of truncation and cancellation	OpenMP and MPI scaling and speed-up
6	Numerical integration with Root Solving to find intersection (e.g., times when trains pass each other)	Numerical integration and methods of bisection, and intervals with slope, to precisely determine roots of error equations	Methods of bisection, Newton-Raphson to precisely determine roots of error equations	OpenMP and MPI scaling and speed-up

The train problems, from Table 5, are harder than most students expect, with challenges based upon data dependencies in integration and programming to abstract functions for features such as root solving. To provide an idea of design complexity students face, Figures 1 and 2 show diagrams provided by the instructor to document one potential strategy for speed-up using MPI. Students can re-attempt this problem improving their MPI solution or can try a new method of parallelization such as OpenMP, or even use of more advanced methods such as CUDA and GP-GPU co-processors.

Figure 1 shows the first parallel pass to integrate an acceleration profile, which can use a function generator or look-up table with interpolation (complicating the problem).

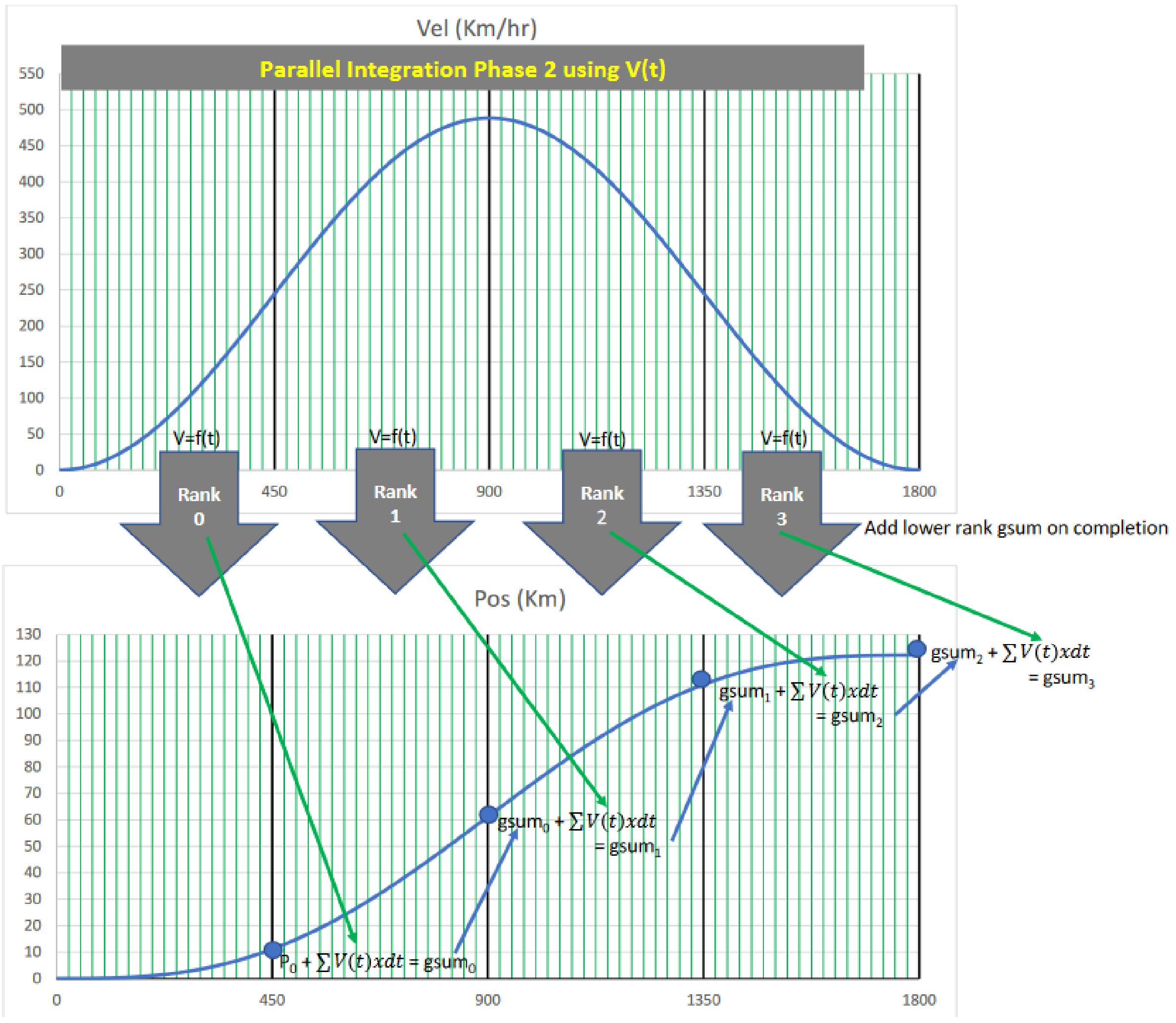
Figure 1. Program Design for Parallelization of Acceleration Profile Integration



Accel (m/s/s)
Parallel Integration Phase 1 using A(t)
A=f(t)
Rank 0
Rank 1
Rank 2
Rank 3
Add lower rank gsum on completion
Vel (Km/hr)
$V_0 + \sum A(t)xdt = gsum_0$
$gsum_0 + \sum A(t)xdt = gsum_1$
$gsum_1 + \sum A(t)xdt = gsum_2$
$gsum_2 + \sum A(t)xdt = gsum_3$

Figure 2 shows the second parallel pass to integrate a velocity profile, derived from the first pass integration of acceleration, which must be used in a second look-up table with interpolation (for the most general use) or a second function generator. While students have most likely seen linear interpolation before, they have perhaps not used it with integration and to model functions for physical simulation.

Figure 2. Program Design for Parallelization of Acceleration Profile Integration


Vel (Km/hr)
Parallel Integration Phase 2 using V(t)
V=f(t)
Rank 0
Rank 1
Rank 2
Rank 3
Add lower rank gsum on completion
Pos (Km)
$P_0 + \sum V(t)xdt = gsum_0$
$gsum_0 + \sum V(t)xdt = gsum_1$
$gsum_1 + \sum V(t)xdt = gsum_2$
$gsum_2 + \sum V(t)xdt = gsum_3$

The MPI programming requires use of advanced features to compute global sums that are synchronized to divide up the work over a large number of MPI nodes (up to 31 in the CSU cluster). Making this simulation parallel and testing it for any acceleration profile, linear or non-linear, is not trivial. For students wanting more challenge, the problem can be extended to model more physical details (e.g., friction, over longer periods or time, etc.).

Students may design a creative problem of their own interest, or simply re-do one of the major problems with a goal for mastery. The improvements for mastery must be significant and clearly demonstrate speed-up in the final reporting and video overview. Whether a new creative program is proposed, or an old problem is re-done for mastery, all students must clearly analyze speed-up and present a clear design and code walk-through to demonstrate what they learned. The videos created take students significant time after completion of their programming, analysis, and testing (at least 3 to 4 hours) and force them to reflect upon the correctness and clarity of their work. Often students think of ways to improve code while explaining, and many indicated they had to re-do their videos.

8. Course Final Parallel Program Results

Several students achieved speed-up higher than expected by Amdahl's law and linear scaling based upon number of cores and nodes used. Most students achieve speed-up that can be explained by Amdahl's law with credible parallel portions in the 80% or better range. Students who explain their speed-up comparing to Amdahl's law with their estimated parallel and sequential code portions with accurate and precise results have mastered programs that gave them difficulty when the first attempted them. Only a few students were below expectation in fall 2020, and while some did drop the course based upon COVID-19 rules at California State University Chico, the majority performed above expectation. The results are summarized below.

Table 6. – Final Parallel Program Outcomes for Students

Above Expectation for Learning Outcomes	Expected Learning Outcomes	Below Expected	Not Completed
60.6%	23.7%	2.7%	13.0%

Final overall course grades were not quite as successful. Some students had difficulty with quizzes, exercises, and the mid-term exam, perhaps based upon workload and time challenges, or just based upon learning and making mistakes on the first try. Overall, the outcomes for the course for fall semester can be summarized as follows.

Table 7. – Final Course Grade Outcomes for Students

Above Expectation for Learning Outcomes	Expected Learning Outcomes	Below Expected	Not Completed
23.7%	60.5%	15.8%	0.0%

9. Future Work

Based on success and student interest in this approach, the persistence of COVID-19 and need for distance learning this spring, the new design has been repeated in spring 2021. An effort to collect a second sampling of pre-course, mid-course, and post-course data has been repeated to determine if results are repeatable. The original design was taught by another instructor, so the only way to obtain a control comparison would be reversion to the older approach, away from this new approach described here. However, this only seems ethical and viable if students support this approach and it is unlikely that this course will ever be offered online only again. It is the author's opinion that the re-design should instead be adapted to in-person instruction and fine-tuned, perhaps to better scaffold the final parallel program work and to provide some earlier assessment of this final high-stakes exercise. With 40% of the grade all assessed by one final report and a video, for more than fifty students, is a serious challenge for the instructor to grade the last week of the semester. Ideas to have students help with peer assessment and other methods such as automated program testing and assessment may also be investigated in the future. Overall, the strategy appears to be successful enough to warrant more study and refinement.

10. Summary

The outcome, where most students designed, implemented, explained, and analyzed a non-trivial parallel program for a numerical simulation or analysis at the end of the course, appears to support the hypothesis of low-stakes formative learning early and high-stakes summative assessment later in the semester along with refinement of a previously attempted problem and solution. The idea is that mastery in upper division courses is important to student success and attainment of learning objective outcomes, and that this also allows more students to complete this important course their senior year.

One hazard of this approach is that many students are not satisfied with their grades prior to the final parallel program grading, with lower grades computed up to that point. One method to repair this is to perhaps break the final parallel program into two parts, one delivered earlier than the other, perhaps as an early submission and then the final video. However, this creates more work for students while they are trying to concentrate on the formative learning.

References

[1] Chandra, Rohit, et al. Parallel programming in OpenMP. Morgan Kaufmann, 2001.
[2] Pacheco, Peter. An introduction to parallel programming. Elsevier, 2011.
[3] Quinn, Michael J. "Parallel programming." TMH CSE 526 (2003): 105.
[4] Yang, Chao-Tung, Chih-Lin Huang, and Cheng-Fang Lin. "Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters." Computer Physics Communications 182.1 (2011): 266-269.
[5] Amdahl, Gene M. "Computer architecture and Amdahl's law." Computer 46.12 (2013): 38-46.
[6] Hill, Mark D., and Michael R. Marty. "Amdahl's law in the multicore era." Computer 41.7 (2008): 33-38.
[7] Sun, Xian-He, and Yong Chen. "Reevaluating Amdahl's law in the multicore era." Journal of Parallel and distributed Computing 70.2 (2010): 183-188.
[8] Gustafson, John L. "Reevaluating Amdahl's law." Communications of the ACM 31.5 (1988): 532-533.
[9] Shi, Yuan. "Reevaluating Amdahl's law and Gustafson's law." Computer Sciences Department, Temple University (MS: 38-24) (1996).
[10] De Graaf, Erik, and Anette Kolmos. "Characteristics of problem-based learning." International Journal of Engineering Education 19.5 (2003): 657-662.
[11] Hung, Woei, David H. Jonassen, and Rude Liu. "Problem-based learning." Handbook of research on educational communications and technology 3.1 (2008): 485-506.
[12] https://computing.llnl.gov/tutorials/pthreads/
[13] Yang, Won Y., et al. Applied numerical methods using MATLAB. John Wiley & Sons, 2020.
[14] Many great cloud-based online math verification tools are available, including: https://www.desmos.com/calculator , https://www.symbolab.com , https://handymath.com/calculators.html , and https://onlinemschool.com/math/assistance/
[15] Vandergraft, James S. Introduction to numerical computations. Academic Press, 2014.
[16] Sosniak, Lauren A. Bloom's taxonomy. Ed. Lorin W. Anderson. Chicago, IL: Univ. Chicago Press, 1994.
[17] Joshi, Ankur, et al. "Likert scale: Explored and explained." *Current Journal of Applied Science and Technology* (2015): 396-403.
[18] Harlen, Wynne, and Mary James. "Assessment and learning: differences and relationships between formative and summative assessment." Assessment in education: Principles, policy & practice 4.3 (1997): 365-379.
[19] Dixson, Dante D., and Frank C. Worrell. "Formative and summative assessment in the classroom." Theory into practice 55.2 (2016): 153-159.
[20] Harlen, Wynne. "On the relationship between assessment for formative and summative purposes." Assessment and learning 2 (2006): 95-110.

[21] Alvarez, Lluc, Eduard Ayguade, and Filippo Mantovani. "Teaching HPC systems and parallel programming with small-scale clusters." 2018 IEEE/ACM Workshop on Education for High-Performance Computing (EduHPC). IEEE, 2018.
[22] https://magpi.raspberrypi.org/articles/build-a-raspberry-pi-cluster-computer
[23] Schulz, Karl W., et al. "Cluster computing with OpenHPC." (2016).
[24] Kalu, E. Eric. Numerical Methods with Applications: Abridged. Lulu. com, 2009 (http://nm.mathforcollege.com/topics/textbook_index.html)
[25] Doucet, Kevin, and Jian Zhang. "Learning cluster computing by creating a Raspberry Pi cluster." Proceedings of the Southeast Conference. 2017.
[26] Sulistyoningsih, Margaretha. "Promoting Active Learning for Increasing Students' Understanding of the Teaching Materials: A Report on Teaching Experience in Computer Science." Indonesian Journal of Information Systems 3.1 (2020): 64-74.
[27] Indriasari, Theresia Devi, Andrew Luxton-Reilly, and Paul Denny. "A Review of Peer Code Review in Higher Education." ACM Transactions on Computing Education (TOCE) 20.3 (2020): 1-25.
[28] D. Battaglia, K. Sampigethaya, A. Almagambetov, M. Andalibi, T. Groh, K. Martin, M. Pavlina, S. Siewert, A. Boettcher, "Integrating Research into Undergraduate Courses: Experiences from a Multi-Disciplinary Faculty Learning Community", ASEE Rocky Mountain Section Conference, Cedar City, Utah, October 2016.