Paper ID #44705

Preparing Students to Master Hybrid and Co-Processing Methods for High Performance Computing

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 worked in the computer engineering industry for twenty-four years before starting an academic career in 2012. Dr. Siewert spent half of this time on NASA astronautics and deep space exploration programs and the next half on commercial product development for high performance networking and storage systems. In 2020, Dr. Siewert joined California State University Chico to teach computer science as full-time faculty and he continues in an adjunct professor role at University of Colorado Boulder. Research interests include real-time systems, interactive systems, machine vision and machine learning applied to sensor networks, sensor fusion, and instrumentation. Dr. Siewert is a co-founder of the Embedded Systems Engineering graduate program at the University of Colorado and is a graduate curriculum committee chair at California State Chico.

Preparing Students to Master Hybrid and Co-Processing Methods for High Performance Computing

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

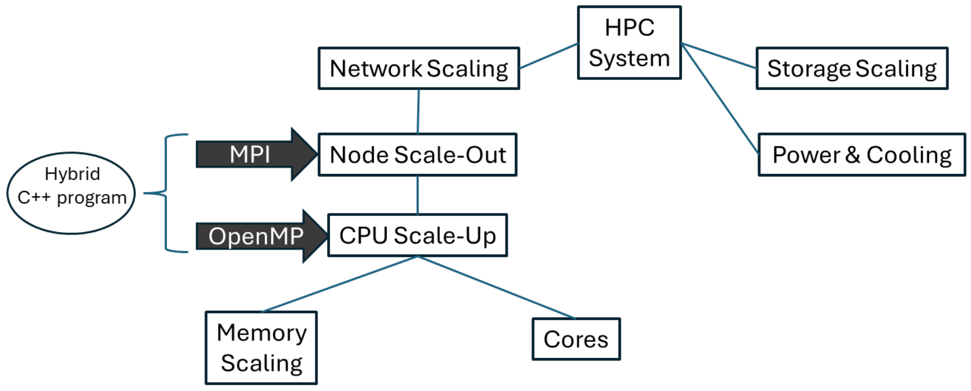
Abstract

Students learning methods of parallel programming have a unique opportunity to develop a distinguished career in scalable scientific computing by learning scale-up, scale-out, and co-processing computer architecture. Teaching a course that covers traditional parallel programming methods for scalable high-performance computing has included scale-up shared memory methods such as OpenMP, scale-out distributed memory methods such as MPI (Message Passing Interface) and more recently co-processing methods such as CUDA (Compute Unified Device Architecture). Learning these three major methods is challenging for senior year undergraduate or first year graduate students, but now, quantum computing, and specifically quantum co-processing has emerged as another challenge for future high-performance computing software developers. It is accepted that quantum computing will show advantages in specific areas such as cryptanalysis, optimization, and quantum simulation, but will not soon replace traditional digital logic high performance computing anytime soon, if ever. Instead, much like a GP-GPU (General Purpose Graphics Processing Unit) acts as a co-processor for specific parallel work off-load, so will quantum computers, providing a QPU (Quantum Processing Unit). In this paper the methods for helping students deal with the triple challenge of learning parallel programming, writing correct code for any scale, and verifying scalability are presented along with new methods to incorporate quantum computing as another hybrid option. The paper provides details of how the triple challenge can be extended to include the fourth challenge of advanced co-processing methods. Techniques used are problem-based learning for mastery and contract-based projects where students demonstrate their achievement of key learning objectives.

1. Introduction

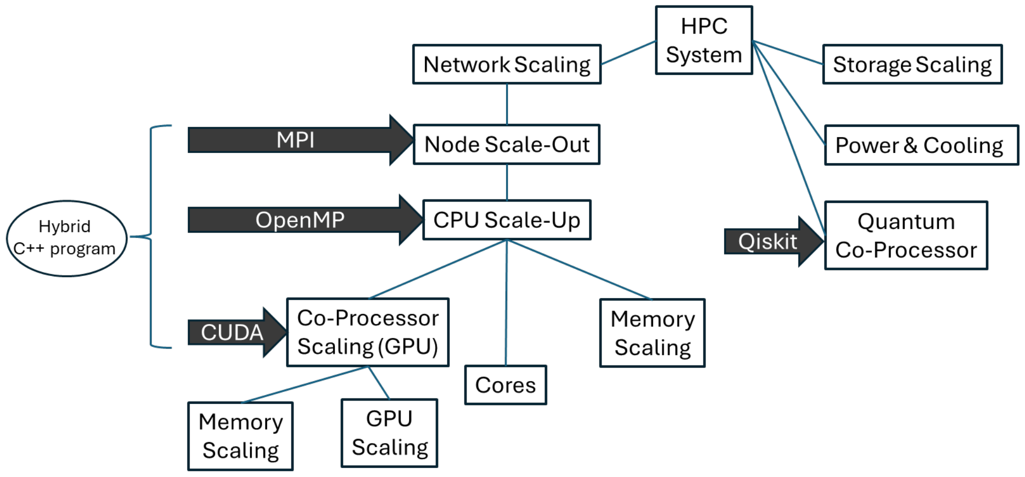
Based upon prior work to improve Numerical and Parallel Programming taught in the department of computer science at CSU (California State University) Chico, anecdotal feedback and instructor perception is that students learned more by completing a final parallel program (project) rather than a final exam as determined by success to actually scale and speed up a working program for which both sequential and parallel runs verify based on mathematical principles[1]. The anecdotal evidence based upon instructor student interaction was never formally verified by surveys or other methods of assessment, however, the instructor has seen a significant increase in student success meeting department criteria for "C" or better pass rates. Given this significant improvement to the outcome as perceived by the instructor for students who are all required to take CSCI 551, Numerical and Parallel Programming, the goal of this paper is to develop the formal assessment survey questions to be asked in the future and to design two additional methods of speeding up algorithms using co-processing. Figure 1 shows the abstracted hardware scaling architecture used by CSCI 551 to enable students to learn shared memory parallel programming (with OpenMP or Pthreads) and to learn distributed memory parallel programming with MPI (Message Passing Interface). Given a network cluster of scaled-up (multi-core) compute nodes, a student can not only write a shared memory program or a distributed memory program that scales and speeds up computation, but can also create a hybrid scale-out and scale-up program using MPI and OpenMP in a single C++ program to benefit from both types of scaling and speed-up as has become common practice in industry.

Figure 1. Scale-Out of Scaled-Up Compute Nodes for Hybrid MPI + OpenMP Programs



The prior improvements to shift focus away from benchmarks and to parallel applications for real world problem as described in prior work, has allowed students to gain an interest in hybrid C++ programs for simulation, computer vision, machine learning, cryptanalysis, and many more complex parallel programs[1]. While the systems and hybrid OpenMP with MPI programs are current with best practices for industry, lately, many new systems have also started to scale further with co-processing such as GP-GPU (General Purpose Graphics Processing Unit) methods as shown in Figure 2.

Figure 2. Scale-Out of Scale-Up Nodes with Co-Processing



While network and storage resource scaling go beyond the scope of the current course, simple methods to work around these potential bottlenecks are covered when they come up[2]. Given an understanding of multiple parallel methods, students can develop single programs in C++ that include MPI with OpenMP and CUDA kernels such that one hybrid program can take advantage of all three methods of scaling. Based upon a workforce initiative grant, California State University has started a system wide effort to integrate Quantum Computing topics into existing classes as well as adding courses dedicated to quantum topics. While a QPU (Quantum Processing Unity) can be imagined to conceptually provide similar speed-up to specific sections of existing programs where there is a "quantum advantage," such as cryptanalysis using Shor's algorithm, any sort of quantum hybrid program will have to be two distinct programs. As shown in Figure 2, a quantum sub-problem for a program today would have to be sent to a cloud-based quantum computer (e.g., using Python and Qiskit) with results combined with a hybrid MPI + OpenMP + CUDA program through a file system for example to combine all four methods (shared, distributed, co-processor, quantum cloud co-processor).

While the goal to cover four distinct methods of parallel programming in one undergraduate course is ambitious, CUDA for GP-GPU co-processing has already been successfully added to ensure all students have significant challenge. The addition of Qiskit adds the most challenge since it requires creation of a hybrid C++ and Python program. While this concept is being pursued in industry by innovative companies and researchers, it is typically not taught in the context of parallel programming[3]. However, based on industry announcements, industry views Quantum Computing as a form of HPC and therefore students should be prepared for quantum programming[4].

2. An Overview of Course Structure and Challenges

All senior year computer science students at CSU Chico state university are required to complete CSCI 551, Numerical and Parallel Processing, which meets key program learning outcome goals for mathematics and computer science. Assignments include hands-on programming challenges that are called the "triple challenge" as they all require:

1) Mathematics, with the application of numerical methods for computation and verification of correctness (via self-check, proof, math fact, or comparison with a math tool such as MATLAB or Mathematica).
2) Programming, with iterative, dynamic and recursive methods and knowledge of complexity theory.
3) Parallel Execution, with methods of shared memory scale-up, distributed memory scale-out, and finally methods of co-processing covered (GP-GPU so far, with QPU planned as an option).

While the triple challenge is what students must learn to do using programming methods and practices, they must also understand theory of speed-up and scaling, both scale-up and scale-out as well as co-processor scaling. 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, based on parallel hardware and parallel programming methods used[5,6]. Modern systems hardware can make selection of the appropriate value for scaling factor used in Amdahl's law, "S," a non-trivial decision. Amdahl's law and a very convenient algebraic equivalence I call Siewert's law makes analysis simple since speed-up (SU) can be measured with time-stamps and P computed if S is well understood.

Equation 1: Amdahl's law is $SU = \frac{1}{(1-P)+\frac{P}{S}} = T_{sequential} / T_{parallel}$

Equation 2: Siewert's equivalent is $P = \frac{S\left(1-\frac{1}{SU}\right)}{S-1}$, a simple algebraic derivation from Equation 1

Normally, students measure $T_{sequential}$ and $T_{parallel}$ by running the same code with 1 worker, and then with $N_{workers} \leq S$.

The value for S can be hard to determine for hybrid programs that use multiple nodes with multiple cores with co-processors. In this course the goal is to keep the scaling factor S simple, with focus on the number of nodes used as a product with the number of cores per node for hybrid OpenMP + MPI for example. However, this simplification can be pessimistic given micro-parallel features most CPU systems now incorporate by default[7,8,9].

Further hybridization of programs to including GPU and QPU co-processing makes the S scaling factor even more tricky to determine with the use of programming methods like CUDA or a Qiskit for a QPU. Fundamentally, parallel programming without a QPU has an advantage over sequential machines clocked at the same rate based on the ability to transform most algorithms that are polynomial in time to become polylogarithmic. The reduction in complexity from polynomial to polylogarithmic comes from parallel divide and conquer transformation, or in more challenging scenarios, map and reduce. So, as shown in equation 3, we have the following time complexity reduction:

Equation 3: Log(n) or O(log n) < **Poly(log n) or (log n)^k** < **Poly(n)** or $2^{O(\log n)}$

Poly(log n) which is based upon division of work (or mapping) and simultaneous computation prior to reduction (combining results) has an advantage that is inherently limited by hardware scaling, the S factor in Amdahl's law. The parallel advantage is theoretically not as significant as quantum computing since it is limited by how big a scale-up chip can be fabricated and less limited by how many nodes can be interconnected for scale-out, but still physically limited in a domain much larger and limited than quantum scaling. Quantum scaling is possible by increasing the number of qubits in a single quantum computer (scale-up) and as of now, is not scalable with scale-out[2,10].

Students are able to learn the parallel advantage and compose code to take advantage of it for a wide variety of problems ranging from cryptanalysis to image processing, machine learning, linear systems, simulation, and many other challenges where dimensions and degree of the fundamental math is high, but still polynomial bounded. A key aspect that they learn is that the code must be efficient in the mapping and reduction phases so that most of the work is done in parallel. When the mapping and reduction phases, done sequentially, become diminishingly small compared to the parallel work done, the speed up approaches linear where S=SU, which is often explained by Gustafson's law. Since Poly(log n) < Poly(n), we can see from Equation 4 that this is useful for any high dimension and high degree problem.

Equation 4: **Poly(log n) < Poly(n);** and therefore $O((\log n)^k) < O(n^k)$ … $O((\log n)^3) < O(n^3)$ … $O((\log n)^2) < O(n^2)$ … $O((\log n)) < O(n)$

The class of algorithms that benefit from being transformed from sequential to parallel are sometimes referred to as Nick's class for Nick Pippenger at Harvey Mudd. While limited compared to the full landscape of complexity that is bonded by exponential time as shown in Equation 5, parallel scale-up and scale-out have provided very practical value to allow for solution to be found for problems in Poly(n) completed in human time, that otherwise might take longer than a human lifetime.

Equation 5: Poly(n) < EXP linear or $2^{O(n)}$ < EXP or $2^{\mathrm{Poly}(n)}$

Using measured speed-up and iterative analysis, S can be estimated and then refined and the assertion is that this may also be possible with a QPU. A QPU has a quantum advantage that is known as BQP as shown in Equation 6. A QPU has been shown to be able to transform a problem that is linear exponential (e.g., semi-prime factorization into component primes) in Poly(n). The most famous example is Shor's algorithm. However, as a new co-processing learning module for CSCI 551, students can explore the use of a QPU using Qiskit for semi-prime factoring, machine learning, QUBO (Quadratic Unconstrained Binary Optimization), and a host of other applications to learn about BQP.

Equation 6: BQP = Poly(n) < EXP linear or $2^{O(n)}$ < EXP or $2^{\mathrm{Poly}(n)}$

In theory, a hybrid program might that includes EXP linear problems combined with Poly(n) could be solved with time complexity that is less than the sum of polylogarithmic + BQP.

This is an interesting and realistic assignment for new graduates entering the HPC industry or going on to graduate school to study HPC related disciplines. The class is a challenge well suited to interactive teaching methods with problem-based and active learning approaches that enhance student engagement1[1,11,12].

All students are shown two shared memory parallel programming methods: 1) the OpenMP method that uses compiler pragmas (directives) and 2) POSIX threading, known as "Pthreads"[13], as well as programing for distributed memory parallel processing using MPI (Message Passing Interface). Adding in learning modules for GPU and QPU will allow students interested in hybrid parallel programming to explore a wide variety of hybrid combinations:

1. Shared memory + distributed scale-out: e.g., OpenMP + MPI
2. Distributed scale-out with co-processing on each node: e.g., MPI + CUDA
3. Shared memory + quantum: e.g., a host with cloud access to a quantum computer as a QPU

Most of the applications studied in CSCI 551 are rooted in well-known mathematic challenges that have large dimensions (degrees of freedom), high degree (non-linear), and significant overall computational complexity, or may in fact be NP-hard such as optimization problems. For calculus-based and linear system problems, tools such as MATLAB are used[14], and students are encouraged to check their work done by hand and implemented as programs with MATLAB and other online "cloud-based" math tools[15]. 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.

4. Course Learning Objectives

In addition to the practical goal to teach four distinct options to speed up a program (algorithm) including:

1) Shared memory parallel – OpenMP or Pthreads
2) Distributed memory parallel – MPI
3) Co-processing parallel – CUDA
4) Quantum advantage for distinct algorithms (e.g., Shor's algorithm[16], QUBO[17], and Grover's algorithm[18]) - Qiskit

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.
4) Introduction to advanced concepts beyond divide and conquer such as map and reduce.
5) Challenging problems associated with adaptation of sequential programs to parallel execution such as loop-carried dependencies.
6) Selection of the best method to attain speed-up form the four studied or identification of a hybrid solution.

Based on Bloom's cognitive dimensions[19], 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. While it would therefore seem risky to increase topics covered and to create the potential for even more challenge by promoting hybrid programs (e.g., MPI + OpenMP + CUDA) and two-program solutions combining parallel methods such as MPI with Quantum computing, an increase in student success appears to be coupled to endless challenge (with some simpler options for students with more modest goals).

5. General Course Challenges and Successful Outcomes

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, parallel programming methods, and programming in general. To assess if the new co-processor modules and course teaching methods are effective, the author has designed a pre-class survey that will include questions on prior knowledge on a Likert scale[20]. Once approved by the CSU Chico IRB (Institutional Review Board) the survey is planned to be administered in fall 2024 and spring 2025 to measure student sentiment and achievement of learning objectives.

Figure 3. Planned Survey on Pre-Course Student Familiarity with Calculus Used in Course

I am familiar with numerical integration and equations.

Response	# of respondents	percent
Strongly Agree		
Agree		
Neither Agree nor Disagree		
Disagree		
Strongly Disagree		
Not Applicable		
No Answer		

While it is believed that students have the perception that the math will be hard it is also hypothesized that they will become comfortable with the math quickly through the use of problem-based learning where numerical and algorithmic methods focus on programming solutions rather than proofs. Traditional math facts, already known proofs, tools such as Mathematica and MATLAB are then used to help students verify the correctness of their programs. A major hypothesis of the PBL (Problem Based Learning) approach is that the practice using methods students are most comfortable with (programming) to solve novel math problems will result in a quick gain of confidence since they do have knowledge via pre-requisites and programming plays to their strengths. To verify the prior assertions, students will be polled to assess their confidence again mid-course and at the end of the course as shown in Figure 4.

Figure 4. Mid-Course Student Confidence in Numerical Method Math Skills

I have made progress learning numerical methods and programming methods to solve math problems including calculus, vector/matrix transformations, linear algebra, and discrete math.

Response	# of respondents	percent
Strongly Agree		
Agree		
Neither Agree nor Disagree		
Disagree		
Strongly Disagree		
Not Applicable		
No Answer		

Student confidence in programming is expected to be high coming into the course since most have had four or more programming courses using the C and C++ programming languages. For this study, the will be confirmed using a survey question as shown in Figure 5. For computer science, it is hypothesized that programming is not an obstacle while learning parallel programming concepts and practices.

Figure 5. Pre-Course Student Confidence in Programming Skills

I am comfortable with C/C++ programming, using Makefiles, and Linux based code development.

Response	# of respondents	percent
Strongly Agree		
Agree		
Neither Agree nor Disagree		
Disagree		
Strongly Disagree		
Not Applicable		
No Answer		

It is further believed that students will have lower confidence writing parallel programs (shared, distributed memory and co-processing) compared to their general confidence in sequential programming skills. This hypothesis will be assessed using a survey as shown in Figure 6.

Figure 6. Pre-Course Student Confidence in Divide and Conquer with Threading

I have knowledge of how to create shared memory, distributed memory and co-processing parallel programs to speed-up sequential programs.

Response	# of respondents	percent
Strongly Agree		
Agree		
Neither Agree nor Disagree		
Disagree		
Strongly Disagree		
Not Applicable		
No Answer		

Once again, by mid-semester students it is believed that students will be feeling much more comfortable with shared memory, distributed memory and co-processor parallel programming by the middle of the semester as will be assessed with the survey in Figure 7.

Figure 7. Mid-Course Student Confidence in Divide and Conquer with Threading

I have made progress on learning how to use "divide and conquer" design methods for shared memory, distributed memory, and co-processor parallel programs to speed-up existing sequential programs.

Response	# of respondents	percent
Strongly Agree		
Agree		
Neither Agree nor Disagree		
Disagree		
Strongly Disagree		
Not Applicable		
No Answer		

The most important metric to be measured is however how students feel about their own learning outcomes at the end of the course after completion of a final program that they proposed with contract grading. They choose the program (algorithm) to make parallel that is of most interest to them and the method(s) to be used including hybrid combinations to attain maximum scaling and speed-up.

To date, the use of shared-memory (OpenMP and Pthreads) for multi-core computers and distributed-memory (MPI) for clusters as well as CUDA for GPU co-processing has largely been successful based upon the diversity and number of final programs completed since fall 2020. While the GPU co-processing is an optional final parallel program for students (not required in practice), success has been high based upon instructor demonstration and support for CUDA coding and GPU testing as well as readily available equipment.

The inclusion of GPU co-processing as an option first suggests that it can now be made a required practice. One of the major hypotheses of this paper is that introduction of quantum computing as a co-processor with a QPU will allow for students to learn about quantum advantage and classes of algorithms that can benefit from it. It is of course not clear that this pattern can be repeated and quantum has extra challenges compared to GPU co-processing (fewer known algorithms, limited qubit scaling, access to quantum computers), but it will be safe to add it as an optional component.

All students will be exposed to QPU concepts and get some practice with Qiskit. Given that quantum computing is largely seen as HPC, this knowledge should serve students well in industry or for graduation studies, and can emphasized over time based on growing support and understanding for the use of a QPU in HPC.

Assessment of how students feel about what they learned overall will be assessed with a survey question that simply asks them how much they learned as shown in Figure 8.

Figure 8. Final Outcomes for Students – How much was learned.

Based upon overall experience with practice exercises and work on my final parallel program – how much have you learned?

Response	**# of respondents**	**percent**
5 – Significant learning		
4 – Good amount		
3 – Some learning		
2 – Little		
1 – Not much at all		
Not Applicable		
No Answer		

In summary, the course re-design[1] presented in prior work to the Pacific Southwest ASEE conference has been successful. Student feedback has improved year over year and employers hiring our graduates for HPC jobs have noted that they have the right skills to directly enter their HPC groups. The next challenge is to improve coverage of the fourth method of parallelism (co-processing) with more practice problems and to add quantum co-processing to meet state workforce goals[21, 22, 23]. Adding a second advanced quantum computing course compared to simply adding a module in CSCI 551 moderates the risk of not knowing how valuable quantum computing knowledge and experience will be for our students. All students will simply have a broader understanding of the key learning objectives for parallel advantage and now quantum advantage and how hybrid methods can benefit classes of problems.

6. A Quick Review of Prior 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 and simply requires them to fill out a brief proposal as shown in Figure 9.

Figure 9. Proposal Rubric with List of Required Aspects for Contract

	Question	**Answer**	**Notes**	**Point Value**
1a	What program (algorithm) will you start with?			10
1b	Have you attached a working, running sequential starter program?			10
1c	How will you verify the sequential program? (Math, compare to tool, self-checking, other?)			10

1d	Have you included sequential program testing and verification?			10
2	What method(s) do you plan to use to make it run in parallel?			10
3	What mathematics (numerical) method is involved?			10
4	What machine will you test on and have you already timed the sequential starter program?			10
5	What speed-up do you expect to achieve and how will you demonstrate?			10
6	**Why is this of interest to you?**			10
7	**What do you see as your biggest challenge to complete this?**			10
Proposal Score (must be 70 or above to be accepted)				**100**

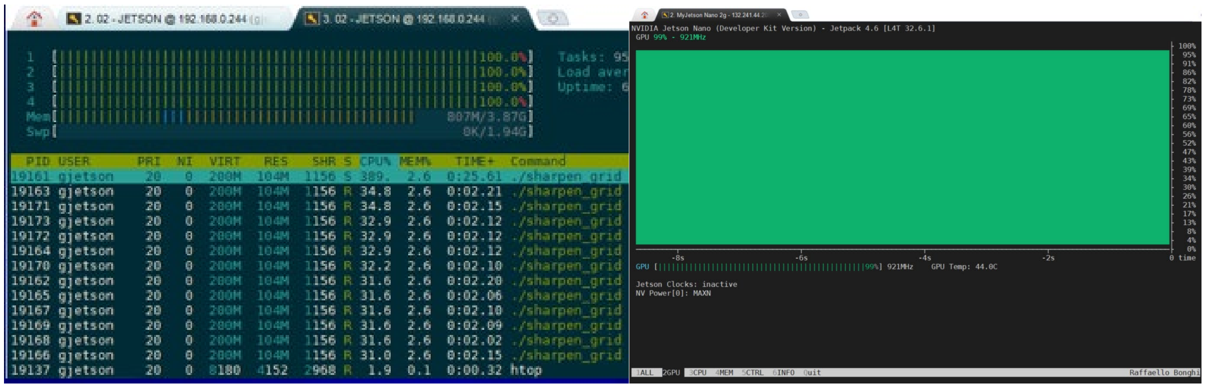
To summarize, students understand they must address the triple challenge (math, programming, and making algorithms parallel) and the methods they must use, and make sure that they can demonstrate mastery fo these methods by completing their own problem. The quality of these final programs has been high, with many hybrid OpenMP + MPI programs, alternative CUDA problems, and even hybrid MPI + CUDA solutions.

Students know enough to self-assess scaling and speed-up while working and provide detailed code walk-throughs turned in as a video along with their final report to explain their methods. They focus is not only on speed-up measured with timestamps, but determination of S and P using multiple forms of Amdahl's law and Gustafson's law.

1) Amdahl's Law for speed-up: SU = $\frac{1}{(1-P)+\frac{P}{S}}$, S=scaling assumed, P=% code estimated to run in parallel,
2) Siewert's reorganization of Amdahl's Law to verify % of code executing in parallel where P = $\frac{S(1-(\frac{1}{SU})}{S-1}$,
3) Simple verification of Gustafson's law be verifying the resources (CPU cores, GP-GPU, and nodes) are saturated using "htop" and "jtop" or "nvtop" as shown in Figure 10.

Having clear methods to self-assess helps student confidence that they have achieved course learning objectives and allows them to debug the parallel solutions they are working on with simple methods. The simplest measurement SU can be obtained by simply prefacing a parallel run with “time” on a Linux command line and comparing to a sequential single worker run of the same program. The more complex % parallel P can be estimated with embedded timestamps used to time specific sections of code that are assumed to run sequentially (start-up mapping and final reduction) and those that run in parallel. Working on six prior practice exercises where students have used assigned problems that they had to make parallel and scale with assessment of speed-up means they are well practiced by the time they work on their own final program.

Figure 10. Use of Htop and Jtop / Nvtop for Resource Saturation Verification


2. 02 - JETSON @ 192.168.0.244
3. 02 - JETSON @ 192.168.0.244
2. MyJetson Nano 2g - 132.241.44.2

```
  1  [||||||||||||||||||||||||||||||||||||||||||||||||||||||100.0%]   Tasks: 95
  2  [||||||||||||||||||||||||||||||||||||||||||||||||||||||100.0%]   Load aver
  3  [||||||||||||||||||||||||||||||||||||||||||||||||||||||100.0%]   Uptime: 6
  4  [||||||||||||||||||||||||||||||||||||||||||||||||||||||100.0%]
  Mem[|||||||||||||||||||||||||||||||||||||||||      807M/3.87G]
  Swp[                                                 0K/1.94G]

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
19161 gjetson    20   0  200M  104M  1156 S 389.  2.6  0:25.61 ./sharpen_grid
19163 gjetson    20   0  200M  104M  1156 R 34.8  2.6  0:02.21 ./sharpen_grid
19171 gjetson    20   0  200M  104M  1156 R 34.8  2.6  0:02.15 ./sharpen_grid
19173 gjetson    20   0  200M  104M  1156 R 32.9  2.6  0:02.12 ./sharpen_grid
19172 gjetson    20   0  200M  104M  1156 R 32.9  2.6  0:02.12 ./sharpen_grid
19164 gjetson    20   0  200M  104M  1156 R 32.9  2.6  0:02.12 ./sharpen_grid
19170 gjetson    20   0  200M  104M  1156 R 32.2  2.6  0:02.10 ./sharpen_grid
19162 gjetson    20   0  200M  104M  1156 R 31.6  2.6  0:02.20 ./sharpen_grid
19165 gjetson    20   0  200M  104M  1156 R 31.6  2.6  0:02.06 ./sharpen_grid
19167 gjetson    20   0  200M  104M  1156 R 31.6  2.6  0:02.10 ./sharpen_grid
19169 gjetson    20   0  200M  104M  1156 R 31.6  2.6  0:02.09 ./sharpen_grid
19168 gjetson    20   0  200M  104M  1156 R 31.6  2.6  0:02.02 ./sharpen_grid
19166 gjetson    20   0  200M  104M  1156 R 31.0  2.6  0:02.15 ./sharpen_grid
19137 gjetson    20   0  8180  4152  2968 R  1.9  0.1  0:00.32 htop
```

```
NVIDIA Jetson Nano (Developer Kit Version) - Jetpack 4.6 [L4T 32.6.1]
 GPU 99% - 921MHz
                                                              100%
                                                               95%
                                                               91%
                                                               86%
                                                               82%
                                                               78%
                                                               73%
                                                               69%
                                                               65%
                                                               60%
                                                               56%
                                                               52%
                                                               47%
                                                               43%
                                                               39%
                                                               34%
                                                               30%
                                                               26%
                                                               21%
                                                               17%
                                                               13%
                                                                8%
                                                                4%
                                                                0%
      -8s           -6s           -4s           -2s          0 time
 GPU [|||||||||||||||||||||||||||||||||||||||||||99%] 921MHz    GPU Temp: 44.0C

 Jetson Clocks: inactive
 NV Power[0]: MAXN

 1ALL 2GPU 3CPU 4MEM 5CTRL 6INFO Quit                  Raffaello Bonghi
```

Given the growing interest in parallel programming at CSU Chico, the Computer Science department has added large scale GP-GPU servers and has added a new Raspberry Pi cluster for remote access and a Jetson Nano cluster as well in addition to an Intel NUC cluster we have been using for over a decade. 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[24], including the newer OpenHPC tools[25].

Student designed final programs include a wide range of interesting problems which must be numerical or semi-numerical. Here is a sample list of some of the more interesting and challenge programs from spring 2022 and fall 2023:

1) Ant Colony Optimization for graph traversal with OpenMP & MPI
2) Simpson’s interpolated function integration with MPI & CUDA
3) A* search with heuristic to find shortest path in graph, OpenMP + MPI.
4) Markov chain (Markov Decision Process) using MPI with OpenMP
5) De-duplication using XOR compared to MD5 and hashing with OpenMP.
6) Wavefunction collapse to generate maps with OpenMP.
7) Long duration train integration using MPI and CUDA
8) DCT2 to compress images with Huffman encoding using OpenMP.

Some students choose to use a single method they believe is ideal for their problem, whereas some use a hybrid of two methods, and finally, some have compared methods to choose the best. The problems and starting sequential programs may be based upon practice assignments done in class prior to the last three weeks of class, completed in a previous class with instructor permission, or based upon research. They proposed the project, it is assessed to determine equal challenge compared to the entire cohort in a class and approved by the instructor to form at contract for grading based on a rubric to verify proposed deliverables and target performance along with code quality and code walk-through provided. The re-design of this course follows a typical formative learning process with problem-based learning, a discussion and activity section, and low-stakes quizzes and a single mid-term exam intended to assess readiness for the second half of the semester.

7. Goals to Pursue Mastery of Hybrid and Co-processing Methods

The further refinement of CSCI 551 to support improved coverage of hybrid and co-processing methods of parallelism are based upon industry feedback. For example, a guest speaker from a fortune 500 company with significant HPC work who has hired many of our undergraduates and some straight into the HPC group has stated that they need hires with experience in MPI, OpenMP, and CUDA. That they further develop seismic data processing applications that use OpenMP with MPI and CUDA all in one program. Likewise, based upon a California State University workforce initiative to integrate Quantum computing into existing classes, the viewpoint that Quantum computing will be another co-processor (two program Python with C++) like a GP-GPU, but less integrated as shown in Figure 2, has motivated updates to the course.

In fall 2023, the class was modified to better support the learning objectives that balance hands-on low-stakes programming practice in class by adding an activity section. For this 3.0 credit course this meant that the discussion (lecture) time to cover theory (math and algorithms) went from 3 hours per week to 2 hours with 2 hours of hands-on lab time added. So, while the total instructor contact time with students went up an hour, the time to discuss theory went down by an hour. Given the math coverage and curriculum that meets a university wide requirement for “quantitative reasoning” which would otherwise require students to take yet another math class, the next proposed change is to increase time and make the course 4.0 credits. This will allow for more coverage of CUDA, which is of high interest to students and industry, and to introduce the options for students to work on hybrid Qiskit + C++ parallel solutions if they have taken our optional Quantum Computing for Computer Scientists. This provides more time for discussion (3 hours per week) and retains hands-on time (2 hours per week).

8. Future Work

The goal of formal assessment using surveys is to ensure that students agree with that the methods have efficacy based upon their perception of their success. Adding too many parallel programming and co-processing methods to the course learning objectives could overwhelm students, however instructor belief is that to date it has motivated and engaged students to have options and to practice multiple methods. Based on instructor perceived success and student interest in learning more methods of parallelism and combining them for extra challenge, the curriculum for CSCI 551 is being updated for fall 2024 to include modules on problems that benefit from combined use of Qiskit (or eventually will if quantum advantage can’t be realized yet) and MPI or OpenMP solutions. Further, while only a small number of students might be interested in the QPU (Quantum Co-Processing Unit) concept, it is expected that a much higher number of students will be interested in GP-GPU and CUDA programming. The Computer Science department has acquired an NVIDIA A100 system with 512GB of host RAM, 80GB of GPU RAM, and 16 cores for a scale-up multi-core with co-processing option for students to use recently. Similarly, the addition of a Quantum Computing class and Qiskit support will allow students wanting to learn about QPUs and alternatives. All students will be required to do at least one or more proact. exercises where they employ either type of co-processing with MPI and/or OpenMP. Overall, the problem-based learning followed by contract grading of a final program as a strategy appears to be successful enough to warrant the risk of adding more burden to students. Furthermore, given that the course will be 4.0 credit hours, this also justifies added work. Finally, industry has interest in students with co-processor experience (GP-GPU more than quantum, but this may change over time as quantum advantage is explored and demonstrated or not).

9. Summary

Prior work to re-design CSCI 551 where students must design, implement, explain, and analyze a non-trivial parallel program for a numerical simulation or analysis at the end of the course of their choice, has not only improved the course, but opens up the potential to add new content to keep up with industry workforce and graduate school parallel computing skills students will need. 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. The added challenges with more freedom and flexibility seem to have in fact not overburdened students but have instead motivated them to excel.

One hazard of this approach is that many students would like a straight-forward final program experience that is mostly aligned with industry, so the GP-GPU option will provide this. Students considering graduate school and a career in HPC might be more inclined to try Qiskit and more challenging hybrid programs with CUDA. When the final parallel program idea was first introduced, some students procrastinated during the final three weeks of class, but a simple method to repair this was to break the final parallel program into multiple parts, proposal and final report and walk-through. The key to motivating students to pursue more hybrid solutions will be through the proposal review process and by exposing them to QPU and GPU methods during practice assignments.

References

[1] Siewert, Sam B. "Improving Student Outcomes with Final Parallel Program Mastery Approach for Numerical Methods." *2021 ASEE Pacific Southwest Conference-" Pushing Past Pandemic Pedagogy: Learning from Disruption"*. 2021.

[2] Gambetta, Jay. "IBM's roadmap for scaling quantum technology." *IBM Research Blog (September 2020)* (2020).

[3] Britt, Keith A., and Travis S. Humble. "High-performance computing with quantum processing units." *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 13.3 (2017): 1-13.

[4] Humble, Travis S., et al. "Quantum computers for high-performance computing." *IEEE Micro* 41.5 (2021): 15-23.

[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] Reilly, D. J. "Challenges in scaling-up the control interface of a quantum computer." *2019 IEEE International Electron Devices Meeting (IEDM)*. IEEE, 2019.
[11] De Graaf, Erik, and Anette Kolmos. "Characteristics of problem-based learning." International Journal of Engineering Education 19.5 (2003): 657-662.
[12] Hung, Woei, David H. Jonassen, and Rude Liu. "Problem-based learning." Handbook of research on educational communications and technology 3.1 (2008): 485-506.
[13] https://computing.llnl.gov/tutorials/pthreads/
[14] Yang, Won Y., et al. Applied numerical methods using MATLAB. John Wiley & Sons, 2020.
[15] 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/
[16] Monz, Thomas, et al. "Realization of a scalable Shor algorithm." *Science* 351.6277 (2016): 1068-1070.
[17] Li, Junde, and Swaroop Ghosh. "Quantum-soft qubo suppression for accurate object detection." *European Conference on Computer Vision*. Cham: Springer International Publishing, 2020.
[18] Grassl, Markus, et al. "Applying Grover's algorithm to AES: quantum resource estimates." *International Workshop on Post-Quantum Cryptography*. Cham: Springer International Publishing, 2016.
[19] Sosniak, Lauren A. Bloom's taxonomy. Ed. Lorin W. Anderson. Chicago, IL: Univ. Chicago Press, 1994.
[20] Joshi, Ankur, et al. "Likert scale: Explored and explained." *Current Journal of Applied Science and Technology* (2015): 396-403.
[21] Aiello, Clarice D., et al. "Achieving a quantum smart workforce." *Quantum Science and Technology* 6.3 (2021): 030501.
[22] Salehi, Özlem, Zeki Seskir, and İlknur Tepe. "A computer science-oriented approach to introduce quantum computing to a new audience." *IEEE Transactions on Education* 65.1 (2021): 1-8.
[23] Ajagekar, Akshay, Travis Humble, and Fengqi You. "Quantum computing based hybrid solution strategies for large-scale discrete-continuous optimization problems." *Computers & Chemical Engineering* 132 (2020): 106630.
[24] https://magpi.raspberrypi.org/articles/build-a-raspberry-pi-cluster-computer
[25] Schulz, Karl W., et al. "Cluster computing with OpenHPC." (2016).
[26] Siewert, Sam, and Dane Nelson. "Solid state drive applications in storage and embedded systems." *Intel Technology Journal* 13.1 (2009): 29-53.