Journal of Parallel and Distributed Computing 157 (2021) 201–219

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

www.elsevier.com/locate/jpdc

Evolving PDC curriculum and tools: A study in responding to technological change

Joel C. Adams

Department of Computer Science, Calvin University, Grand Rapids, MI, USA

ARTICLE INFO

Article history:
Received 19 November 2020
Received in revised form 9 June 2021
Accepted 3 July 2021
Available online 15 July 2021

Keywords:
Beowulf cluster
Education
HPC
PDC
Supercomputing

ABSTRACT

Much has changed about parallel and distributed computing (PDC) since the author began teaching the topic in the late 1990s. This paper reviews some of the key changes to the field and describes their impacts on his work as a PDC educator. Such changes include: the availability of free implementations of the message passing interface (MPI) for distributed-memory multiprocessors; the development of the Beowulf cluster; the advent of multicore architectures; the development of free multithreading languages and libraries such as OpenMP; the availability of (relatively) inexpensive manycore accelerator devices (e.g., GPUs); the availability of free software platforms like CUDA, OpenACC, OpenCL, and OpenMP for using accelerators; the development of inexpensive single board computers (SBCs) like the Raspberry Pi, and other changes. The paper details the evolution of PDC education at the author's institution in response to these changes, including curriculum changes, seven different Beowulf cluster designs, and the development of pedagogical tools and techniques specifically for PDC education. The paper also surveys many of the hardware and software infrastructure options available to PDC educators, provides a strategy for choosing among them, and provides practical advice for PDC pedagogy. Through these discussions, the reader may see how much PDC education has changed over the past two decades, identify some areas of PDC that have remained stable during this same time period, and so gain new insight into how to efficiently invest one's time as a PDC educator.

1. Introduction

In 1996-97, the author was a young faculty member in the Department of Computer Science (CS) at Calvin University. At that time, the department had no course on parallel computing and decided that this was a significant gap in Calvin's CS curriculum. The author's dissertation topic was in the area of distributed systems, which was conceptually closer to parallel computing than that of anyone else in the department, so the author was tasked with designing a course to fill this gap. With no direct training in parallel computing, the author had no curricular models to draw on, which was disconcerting.

Fortunately, Chris Nevison from Colgate University and Nan Schaller from the Rochester Institute of Technology led a week-long NSF-sponsored faculty development workshop on parallel computing at Colgate during the summer of 1997. This workshop included an overview of different parallel and distributed computing (PDC) hardware platforms, an introduction to parallel algorithms, and hands-on practice using PDC software platforms. The workshop leaders also provided the syllabi of their parallel computing courses, sample assignments, PowerPoint presentations, and other useful materials. In short, this workshop provided the author with an excellent introduction to the subject, and a good set of resources for developing his own course.

The rest of this paper reviews how parallel computing education has changed since that summer, particularly the impact of technologies such as MPI, Beowulf clusters, multithreading and multicore systems, accelerator devices such as GPUs and software platforms (e.g., CUDA, OpenCL) to program them, and similar changes. The paper is organized as roughly five-year intervals: Section 2 provides background for the rest of the paper, describing the author's early experiences in the late 1990s. Section 3 describes the impact of technology innovations in the early 2000s. Section 4 describes the significant changes that began to occur as single-core processors began to disappear after 2005. Section 5 describes the impact of new technologies of the early 2010s, and Section 6 describes how PDC technology has continued to change since 2015. Section 7 discusses various insights regarding PDC infrastructure and pedagogy, and Section 8 provides some concluding remarks.

E-mail address: adams@calvin.edu.

https://doi.org/10.1016/j.jpdc.2021.07.003
0743-7315/

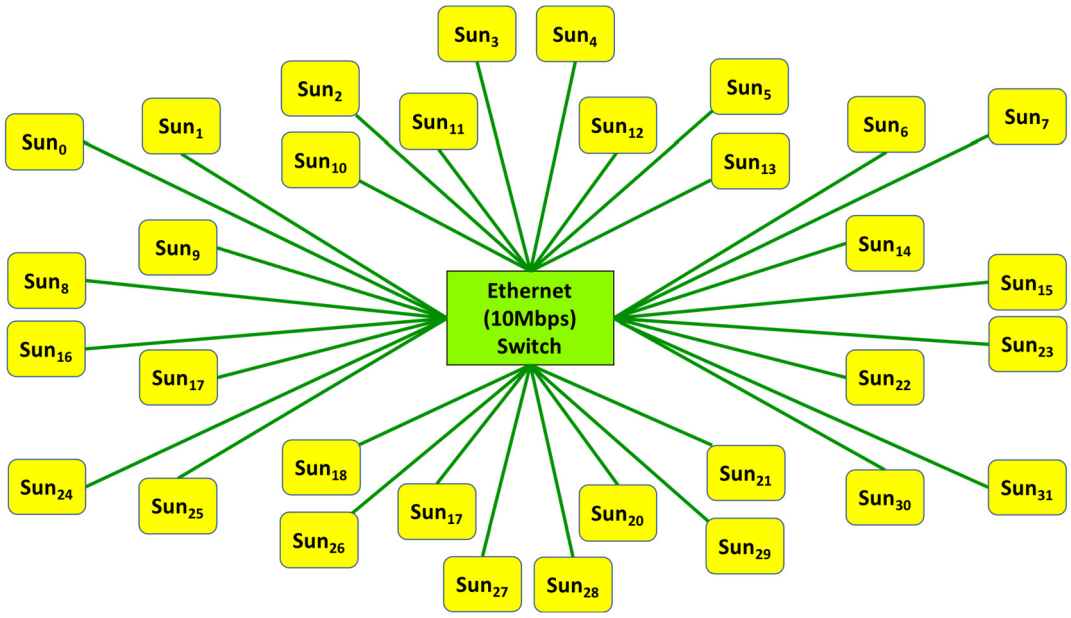


Fig. 1. Calvin CS Lab as a Star-Topology NoW Multiprocessor. (For interpretation of the colors in the figure(s) and table, the reader is referred to the web version of this article.)

2. Background

Through 2021, Calvin University has followed a "4-1-4" calendar in which an academic year consists of a normal four-month Fall semester, a one-month January term (J-term), and a normal four-month Spring semester. During the J-term, a student takes one course that meets three hours per day for fifteen days. By "compressing" a fifteen-week semester-course into fifteen days, the J-term provides an ideal opportunity to test-drive experimental courses that explore topics outside of the normal curriculum. If successful, such courses can eventually be turned into regular courses that meet during a normal semester.

2.1. 1998: parallel computing, iteration 1

Encouraged by his positive experience at the Colgate workshop, the author offered an initial *Parallel Computing* course during Calvin's 1998 J-term. 15 students enrolled in the course. The course content was heavily based on material from the Colgate workshop; it included exposure to parallel computing concepts, with experiential learning through hands-on activities and six homework projects. The course text was Pacheco's book *Parallel Programming with MPI* [23], and the course met in the CS department's computer lab of 32 Sun Sparcstations running Solaris, connected with Ethernet. The conceptual material included coverage of the Flynn Taxonomy; parallel hardware organization and network topologies; parallel algorithms (e.g., odd-even transposition sort, parallel matrix multiplication, parallel Eratosthenes Sieve); analysis of parallel complexity (speedup, parallel efficiency, scalability and Amdahl's Law); and parallel computing history.

One goal of the course was for students to experience two different models of parallelism: SIMD and MIMD computing. For SIMD computing, the author installed Parallaxis-III [14] on each lab workstation. Parallaxis is a machine-independent framework for defining virtual SIMD architectures, specifying parallel algorithms (in Modula-2), and running a specified algorithm on a defined architecture. To support MIMD computing, the author installed MPICH [20], a free, open source implementation of the message passing interface (MPI), on each of the lab's workstations, thus turning the lab into a star-topology network-of-workstations (NoW) multiprocessor. Fig. 1 presents a schematic of this multiprocessor.

Students spent the first half of the course using Parallaxis to explore SIMD computing. Parallaxis includes a simulator that lets a student run their parallel algorithm on their workstation to experience pseudo-parallel execution. Students used Parallaxis to complete three SIMD projects of increasing difficulty levels.

In the second half of the course, the students explored MIMD computing using MPICH. With the lab configured as a NoW multiprocessor, students could use it to run hands-on MPI active-learning activities, plus three MPI projects of increasing difficulty levels.

Using the lab as a NoW multiprocessor provided students with hands-on experience using MPI, but unless there were few other students in the lab, this NoW did not provide them with a realistic experience of the benefits of parallel computing. More precisely, students were unable to experience consistent speedup because the remote MPI processes they were launching were competing with the other students' processes for the NoW's limited unicore CPU resources. Students who were willing to come to the lab very early in the morning (i.e., when no one else was present) could have all of the workstations to themselves and experience consistent parallel speedup, but this was not a general solution to the problem. The only good solution to this problem was to somehow acquire a dedicated multiprocessor for students to use, and a small university like Calvin lacked the budget to buy a commercial multiprocessor.

2.2. 1999: a first Beowulf cluster

One of the parallel hardware topics covered in the course was the *Beowulf cluster*, created by Thomas Sterling and Donald Becker at NASA [13]. Their Beowulf cluster was a dedicated multiprocessor workstation, built from commodity off-the-shelf PC hardware, connected with a standard network (e.g., Ethernet), and running open source software (e.g., Linux, MPI, etc.).

In January of 1999, the author submitted a proposal to the U.S. National Science Foundation's Major Research Instrumentation (NSF-MRI) program, requesting funding to build a Beowulf cluster for research and education at Calvin. This proposal was not

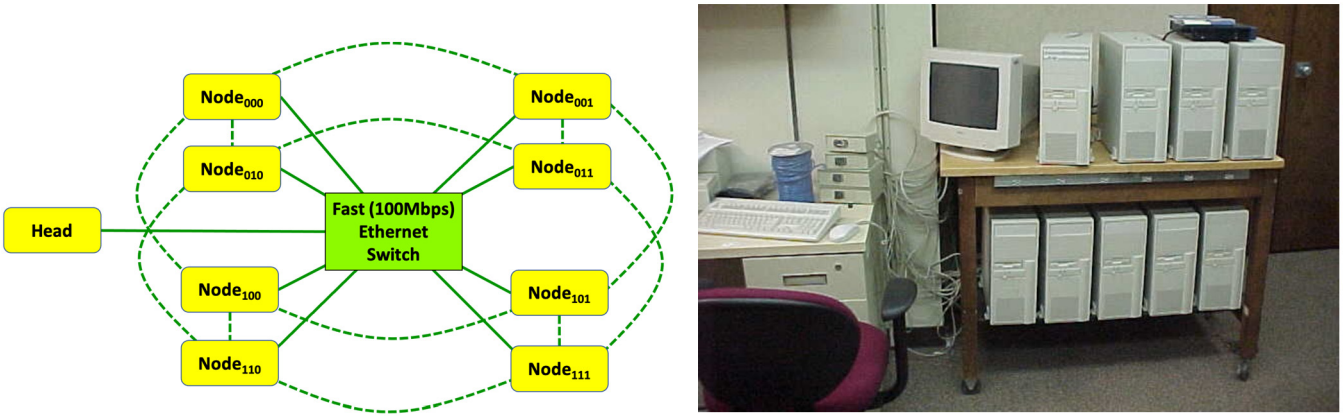


Fig. 2. MBH'99.

funded, and a key reason was a reviewer's skepticism that an undergraduate institution could build such a cluster. At the time, every cluster listed on the *Beowulf.org* website was either built by PhD researchers at government labs or by graduate students from university research labs.

However, one of the students in the author's 1998 course—Mark Ryken—was fascinated by *Loki*, a Beowulf cluster built by Dr. Michael Warren at Los Alamos National Labs [18]. Loki consisted of sixteen Pentium-200 PCs connected with Fast Ethernet in a star+hypercube (4D) topology. Ryken decided to build a similar cluster as his senior capstone project. Ryken worked part-time in Calvin's Information Technology group and leveraged his connections there to acquire a dozen castoff 25-Mhz Intel-486 PCs. From these, he was able to assemble nine working PCs, providing a head node plus eight compute nodes. An internal Calvin grant provided funds for a Fast Ethernet switch and enough network interface cards to connect the PCs into a star+hypercube (3D) topology. The result was *MBH'99*, a name that at various times stood for either "Mark's Beowulf Hypercube" or "Mark's Big Headache" depending on how the project was going. Fig. 2 presents a schematic and picture for MBH'99.

In Fig. 2, the solid lines are the star-topology links; the dashed lines are the hypercube links. Each node in a star+hypercube topology cluster requires N+1 network adaptors, where N is the dimension of the hypercube; the extra adaptor connects the node to the network switch at the center of the star. With a star+hypercube (3D) topology, each node of MBH'99 required 4 adaptors.

Shortly after MBH'99 was built, the author learned about "Amdahl's Other Law" [10], which defines a "balanced" computing system as one in which the hertz of CPU performance, the bytes of main memory, and the bits per second of I/O bandwidth are all at least roughly the same. In using nodes of 25-MHz I-486 CPUs connected with "Fast" (100Mbps) Ethernet, MBH'99 was CPU-bound, not balanced. MBH'99 was thus too slow to be of practical use in solving real-world problems, but it served to demonstrate for the first time that an undergraduate student could build a working Beowulf cluster [6].

3. The early 2000s

Calvin discourages its faculty members from offering the same J-term course in consecutive years, so it was January 2000 before the author's parallel computing course was offered again.

3.1. 2000: parallel computing, iteration 2

During Calvin's 2000 J-term, the author offered the *Parallel Computing* course a second time, with an enrollment of 16 students. This second offering followed the same syllabus as the 1998 course, with minor adjustments to fix issues that arose in the previous offering of the course.

3.2. 2000: a high performance Beowulf cluster

Also in January 2000, the author submitted a revised NSF-MRI proposal to build a Beowulf cluster at Calvin, using MBH'99 as a prior-work prototype to demonstrate the feasibility of the project. This proposal was approved and about $60,000 was awarded in August 2000. During Fall 2000, the author designed the new cluster, purchased the components, and recruited students to help assemble and configure it. The new cluster was dubbed *Ohm* (for Our Hypercube Multiprocessor); it became operational in early 2001 [5]. Ohm consisted of 16 worker nodes, two head nodes (a primary and a backup, which could serve as a 17^{th} compute node when not in use), and an NFS-mounted RAID-array/file server node through which all nodes could access their users' home directories. To create Ohm as a "balanced" cluster, each node contained a 1-GHz AMD Athlon-64 CPU, 1GB of RAM, and the nodes were connected using Gigabit Ethernet. Using HP-Linpack, Ohm was benchmarked at 10.4 GFlops (R_{Max}), which was reasonably "high performance" at the time. Fig. 3 shows a schematic of Ohm.

For research purposes, the author designed Ohm's interconnect as a star+hypercube (4D) topology. Using software written by student Kevin DeGraff, the cluster's working topology could be varied on-the-fly between a star (the solid green links in Fig. 3), a hypercube (the dashed blue links), or a star+hypercube (all of the links). This let the author conduct a cost-benefit analysis, which revealed that for an Amdahl's-other-law "balanced" cluster, the star topology was the most cost-efficient of these three topologies, even for communication-intensive applications.

The star+hypercube interconnect required each node to have five Gigabit Ethernet adaptors. These first-generation GigE adaptors generated significant heat, so each node was housed in a 4U rackmount case to provide adequate ventilation and cooling. Calvin upgraded the room's air conditioning and electrical power systems. Three UPS systems were also used to provide clean, uninterruptable power to the cluster, and two keyboard-video-mouse (KVM) switches were added to make it easier for a sysadmin to maintain the 19 nodes. To accommodate all of these components, three racks were needed, as shown in Fig. 4.

In 2001, the department was seeking a full-time sysadmin, so a series of upper-level students were hired and trained to administer and maintain Ohm, including David Vos, Kevin DeGraff, Matthew Post, and Matthew Koop. During summer 2002, the department hired a full-time sysadmin, and responsibility for administering Ohm was part of this new hire's job description. Initially, this new sysadmin just supervised the students in their day-to-day admin-

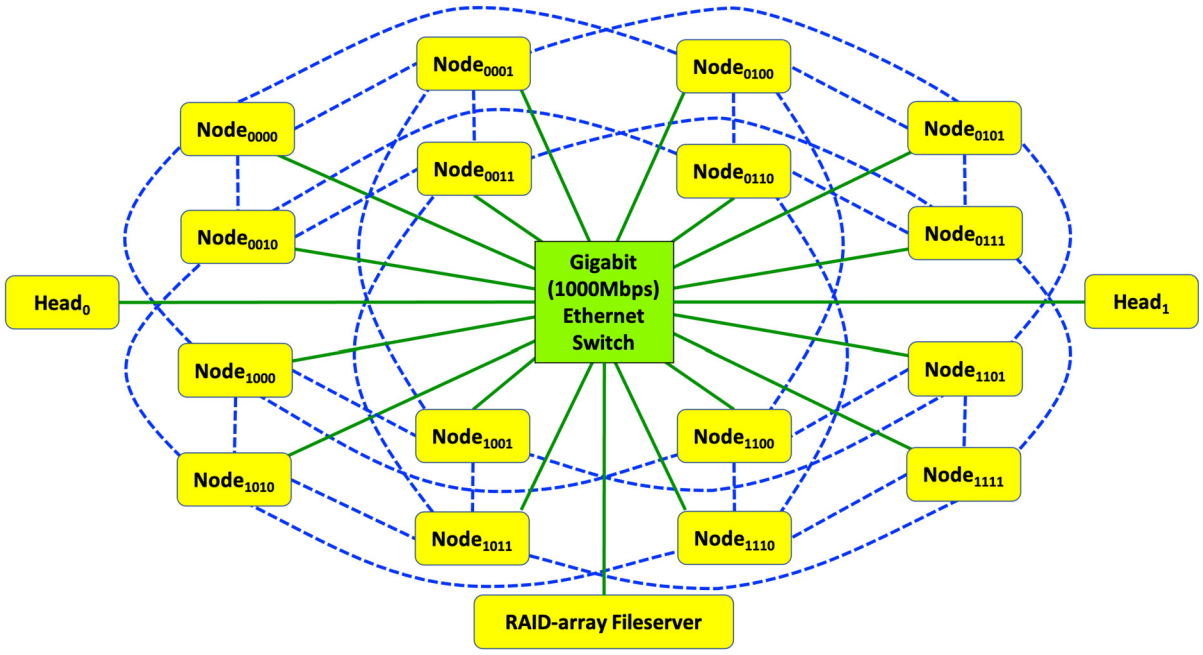


Fig. 3. Schematic of Ohm.

Fig. 4. Ohm.

istration of Ohm, but he took on a more active management role as those students graduated.

It is worth noting that Ohm's primary purpose was to support multidisciplinary research, but at Calvin, most research takes place during the summer, providing open time during the academic year to use the cluster for educational purposes.

3.3. 2002: high performance computing, iteration 1

With the acquisition of a dedicated Beowulf cluster, the 2002 J-term course was renamed *High Performance Computing* and the course was revised to utilize the new cluster. 13 students enrolled in the course. The content of this HPC course was similar to the previous courses, except that in the second (MPI) half, after using the lab NoW to develop and debug their MPI programs, students now transferred their programs to Ohm and ran them there, varying the numbers of processes to test their scalability. Having a dedicated cluster provided the students with far more accurate timing data, allowing the students to assess the scalability of their programs (up to 18 processes, beyond which performance would plateau). By running their programs via a batch queueing system on Ohm, students could experience the speedup of parallel computing, resulting in a much richer learning experience than using a NoW.

3.4. 2003: high performance computing, iteration 2

The presence of Ohm created demand for Calvin's HPC course two years in a row, so the author offered the course again during the 2003 J-term, with 15 students enrolling. This course differed from the previous courses in two significant ways.

The first change came in response to student feedback: Students in the 2002 course wanted earlier and more hands-on experience using Ohm, so the segment on MPI and distributed-memory parallelism was moved to the first half of the course.

The second change was driven by external forces: Symmetric multiprocessors (SMPs) were increasingly affordable and commonplace, so the department had acquired a two-CPU Sun SMP system. Coverage of multithreading and shared-memory parallelism was deemed to have higher priority than SIMD computing, so the second half of the HPC course was mostly devoted to SMP computing using OpenMP [22]. Multithreading thus replaced most of the SIMD computing content.

3.5. 2004: a NCSI workshop at Oklahoma

In summer 2004, the author attended a parallel computing workshop sponsored by the *National Computational Science Institute* (NCSI) and the *Shodor Education Foundation* [28], hosted at the University of Oklahoma. This workshop focused on using MPI to solve science problems, and let the attendees explore MPI-based solutions to a variety of compelling problems, such as an N-body galaxy-formation simulation, a Monte Carlo forest fire simulation, and others. The author subsequently used these examples to "refresh" the content of his HPC course (see below).

One of the many interesting sessions at this workshop was a demonstration of *LittleFe* [25], a six-node miniature Beowulf cluster. Created by Charlie Peck, Paul Gray, David Joiner, and Tom Murphy, LittleFe was small enough to fit into a checked-luggage suitcase or wheel into a classroom, but still useful for demonstrating parallel speedup. One could (for example) run the N-body galaxy-formation simulation with one process to watch a spiral galaxy form very slowly, then re-run the simulation with six processes and watch the galaxy form about six times as fast. It was clear to this author that a portable cluster like LittleFe would open up many possibilities, ranging from taking the cluster to a technical conference to live-demo parallel software, to taking the cluster to a local high school class to demonstrate how parallel computing can accelerate scientific discovery.

Following this workshop, email exchanges between the author and Charlie Peck subsequently catalyzed a series of LittleFe Buildouts at the annual Supercomputing and SIGCSE conferences. At these day-long events, each participant would arrive, receive a box of LittleFe hardware components, spend the morning assembling their own LittleFe, install and configure software (including Linux, MPI and MPI-based example applications), and then spend the rest of the day learning the basics of MPI. Thanks to sponsorship by Intel, at the end of the day, each participant could take home the LittleFe s/he had assembled. Through these LittleFe Buildout events over the next several years, dozens of U.S. colleges and universities received a free LittleFe Beowulf cluster suitable for teaching students about parallel computing.

3.6. 2005: distributed computing in Iceland

The author spent the first six months of 2005 as a Fulbright scholar in Iceland, at what is now the School of Engineering of Reykjavik University. While there, he was asked to teach a course on distributed-memory parallel computing using MPI. The university had no distributed-memory multiprocessor, but provided funds to build one, so the author and his 14 students spent the first few weeks of the course designing and building a Beowulf cluster within the provided budget. The students named their new cluster *Sleipnir*, after Odin's eight-legged horse. To keep its CPUs, memory, and network bandwidth somewhat balanced, each node contained a 3.8-GHz Intel Pentium-4 CPU and 2GB of RAM, and a Gigabit Ethernet interconnect. The nodes were connected using a simple star topology, as shown in Fig. 5.

Using HP-Linpack, Sleipnir was benchmarked at 20.25 Gflops (R_{Max}), which was quite remarkable for such a modest cluster.

Some of the students in this course were from eastern Europe and had difficulty understanding spoken English, making it a challenge for the author to explain the behaviors of the various MPI functions in ways they could understand. However, these students were adept at reading C code. After discovering this, the author replaced many of his static Powerpoint lectures on MPI with live-demoes of minimalist MPI programs, each containing just enough code to illustrate the behavior of a particular MPI function: send+receive, broadcast, reduce, scatter, gather, creating a parallel for loop, the master-worker approach, and so on. By only including the bare minimum of what was needed to illustrate a given MPI function's behavior, running the program, and then relating its output-behavior to the source code that produced that behavior, the author was able to teach these students the basics of distributed-memory multiprocessing using MPI, despite the language barrier.

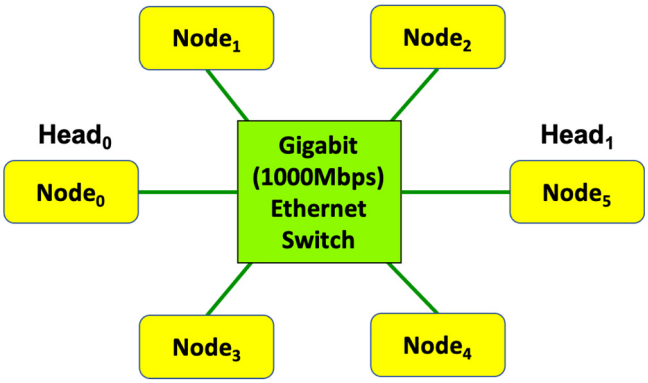


Fig. 5. Schematic of Sleipnir.

These minimalist programs were so useful for explaining the behavior of MPI functions, the author continued to use them in his courses after returning to Calvin. That proved so successful, he subsequently used this same approach to create additional minimalist programs to illustrate other parallel design patterns [19] in MPI, OpenMP, and other parallel platforms.

3.7. 2005: high performance computing, iteration 3

In 2005, the author offered the Calvin HPC course for the first time during the regular Fall semester instead of the J-term. With CS enrollments in decline nationwide, just six students enrolled. This offering was a significant revision in several ways.

One change was that a new course text was adopted: *Parallel Programming in C with MPI and OpenMP* by Quinn [27]. The syllabus from 2003 was adjusted in this 2005 course to match the presentation of material in the new text.

Another change was that science problems from the 2004 NCSI workshop (e.g., the forest fire simulation) were incorporated into this 2005 course. These replaced less-compelling examples, and so made the course more interesting without affecting its overall structure.

Another change was that the author replaced some of his Powerpoint presentations with live-demos of the minimalist MPI programs he had developed in Iceland, as well as newly created minimalist OpenMP programs to illustrate the behaviors of different OpenMP constructs. The resulting set of programs formed the basis for the *patternlets*—minimalist programs illustrating parallel design patterns—that have been shown to improve student understanding of parallel concepts [3].

With the expanded focus on HPC, another change was the addition of a lecture on how to improve program performance via compiler optimizations, introducing students to different optimization techniques and discussing how to implement them by hand in "cutting edge" languages whose compilers (or interpreters) did not support optimization.

When offered as a 15-day J-term course, students were only taking that one course, allowing them to focus on it exclusively. However, the J-term's compressed schedule did not leave much outside-the-classroom time for students to digest and internalize parallel thinking. Moving the course to the regular semester in 2005 expanded its duration to 15 weeks, which gave students more outside-of-class time to better digest and internalize this new way of thinking, and thus gain significantly more skill.

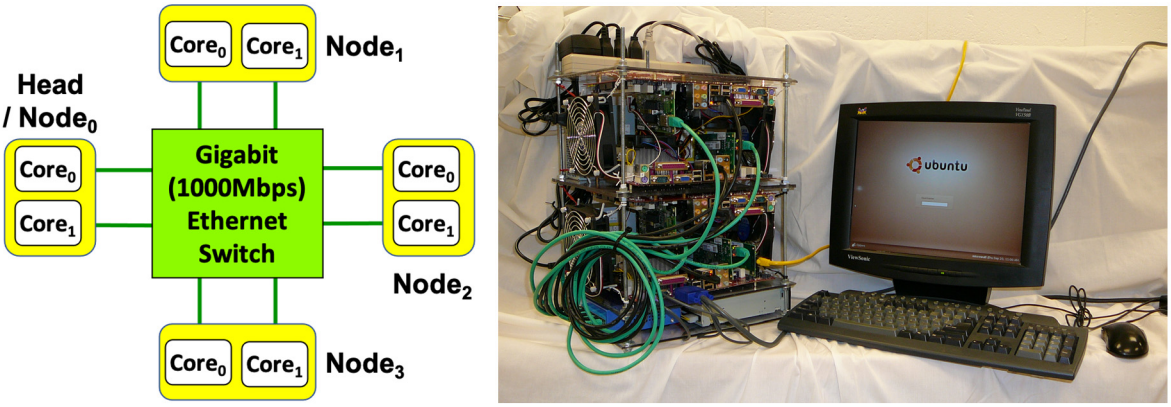


Fig. 6. Schematic and photo of Microwulf.

One final observation from this 2005 course was that students noticed that they could get faster performance testing and debugging their MPI programs in our lab than they could running them on Ohm. The workstations in our lab were replaced every three years; thanks to Dennard Scaling, they now contained 3.2-GHz AMD Athlon-64 CPUs, which were much speedier than Ohm's 1.0-GHz Athlon-64 CPUs. Just four years old, Ohm was already starting to show its age.

4. 2006-2010: the multicore era begins

Early in 2006, the author submitted a follow-up proposal to the NSF-MRI program for funding to replace Ohm with a new cluster. This proposal was not funded, but the program officer encouraged the author to revise and resubmit the next year.

In 2005, AMD released their dual-core Athlon-64 X2 CPU, and in 2006, Intel followed with their Core-2 Duo CPU, marking the beginning of the multicore era. During the same period, Gigabit Ethernet (1000Mbps) replaced "Fast" Ethernet (100Mbps) as the standard on-board networking hardware on most computer motherboards. These two changes—multicore CPUs and standard Gigabit Ethernet—opened up the possibility of building a portable Beowulf cluster with far more computational density (and correspondingly higher performance) than was previously possible.

4.1. 2006: Microwulf: a personal, portable HPC cluster

During the 2006-2007 academic year, the author designed and helped Calvin senior Tim Brom assemble *Microwulf*, a personal, portable, computationally-dense Beowulf cluster. Fig. 6 shows a schematic and photo.

One goal of the Microwulf design was to achieve as much computational performance as possible within a $2500 budget, while maintaining portability. By using microATX motherboards, 3.8-GHz AMD Athlon-64 X2 (dual core) CPUs in each of four nodes, and connecting those nodes with Gigabit Ethernet, Microwulf achieved 26.25 Gflops (R_{MAX}) on the HP-Linpack benchmark for just $2,470, making it the first Beowulf cluster to break the $100/Gflop price barrier [4]. (By August 2007, the price of Microwulf's components had decreased to $1,255, improving its price/performance ratio to less than $48/Gflop.) At $11 \times 12 \times 17$ inches and weighing just 31 pounds, Microwulf was small and light enough to fit on a desktop or in a checked-luggage suitcase, thus maintaining portability.

Recall that the original Beowulf cluster was designed as a multiprocessor *workstation*. Over the years, this vision had been lost as people (this author included) built clusters that were shared by many remote users in a centralized fashion. Just as the *microcomputer* enabled a paradigm shift from centralized mainframe or minicomputer computing to personal computing, Microwulf embodied the concept of a *microcluster*, shifting cluster-computing away from centralized Beowulf clusters shared by many users, back towards Sterling and Becker's original vision of personal Beowulf clusters operated by individuals.

4.2. 2007: high performance computing, iteration 4

Early in 2007, the author submitted a revised proposal to the NSF-MRI program for funding to replace Ohm. This proposal was funded late in the summer of 2007, providing about $206,000 for a new centralized Beowulf cluster for multidisciplinary research (and education) at Calvin.

In Fall 2007, the HPC course was offered again, with 12 students enrolling. The funding for the new cluster was awarded too late to be used in the Fall 2007 semester. However, with four-node Microwulf being over 2.5 times faster than 18-node Ohm, the author revised his Fall-2007 HPC course to use Microwulf instead of Ohm for the MPI segment of the course, and to use one of Microwulf's dual-core nodes instead of the department's comparatively slow Sun SMP system for the OpenMP segment.

Another change was that this 2007 course incorporated coverage of new MPI-2 and OpenMP 2.0 features into the MPI and OpenMP segments of the course. For example, since sequential I/O has historically been a key bottleneck in many parallel computations, MPI-2 added a new parallel I/O mechanism to address this bottleneck, so coverage of parallel I/O was incorporated into the 2007 course. Likewise, OpenMP-2.0 contained a new mechanism for user-defined reduction operations, so coverage of that mechanism was added to the course. Both MPI and OpenMP continue to evolve, so these standards must be regularly reviewed and course materials updated as appropriate. To make room for these changes, the coverage of the history of parallel computing was slightly compressed/reduced.

By replacing Ohm with Microwulf, the lab NoW was no longer faster than the course's Beowulf cluster, but with just eight cores, Microwulf's scalability was limited. More precisely, Microwulf was faster than the lab's NoW using 1-8 MPI processes, but the two multiprocessors performed about the same with 16 processes. If a student had the lab to herself, the NoW's slower but more numerous CPUs could outperform Microwulf's eight faster cores at 32 or more processes.

Ohm continued to be used as a research multiprocessor by Calvin's scientists during Fall 2007, but with funding having been acquired for a new cluster, its time was clearly coming to an end.

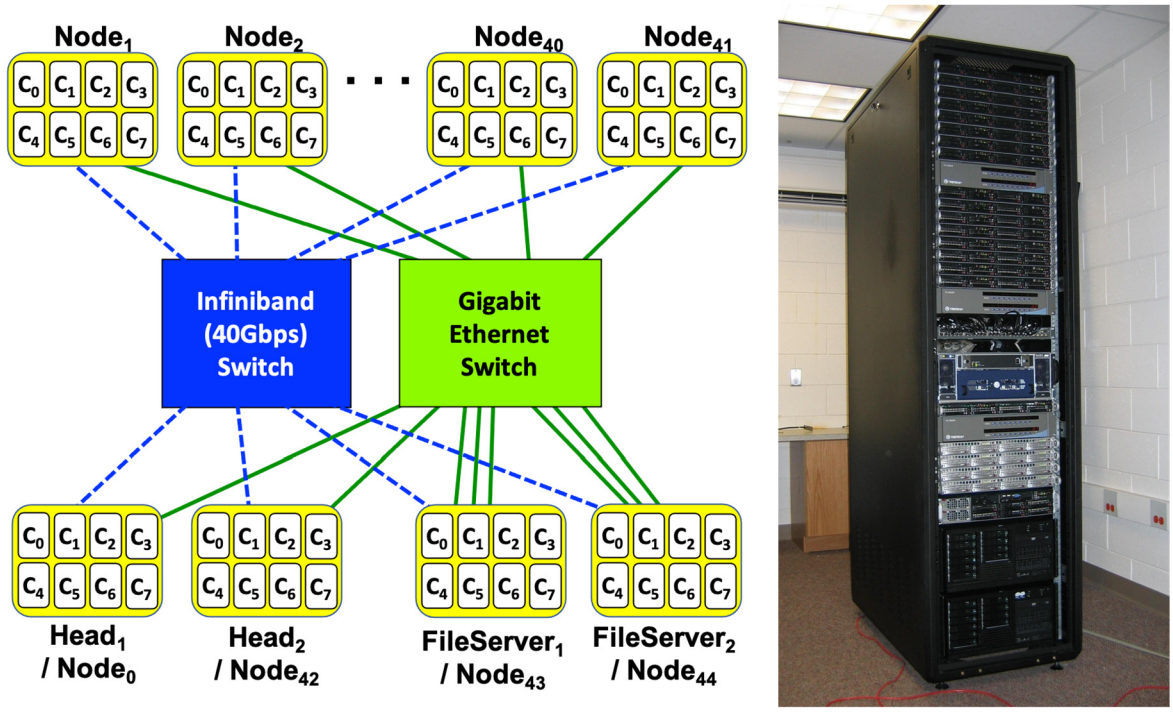


Fig. 7. Schematic and photo of Dahl.

4.3. 2008: Dahl, a new HPC Beowulf cluster

The author designed the new cluster during Fall 2007, purchased its components in December of that year, assembled it in early 2008, and configured it during spring 2008. Since this new cluster would be heavily used for running simulations, it was named *Dahl* in honor of Ole Johan Dahl, the Norwegian computer scientist who co-invented the Simula programming language. With the help of students Kathy Hoogeboom and Jon Walz, Dahl was fully operational by May 2008, at which point Ohm was decommissioned. Fig. 7 shows a schematic and photo of Dahl.

Ohm had required three racks to store 18 nodes. Thanks to Moore's law increasing computational density, Dahl needed just one rack to store 45 nodes plus 40Gbps Infiniband and Gigabit Ethernet networks for data and administrative traffic, respectively. Each node contained at least two Intel 2.2-GHz Xeon quad core CPUs and 4GB of RAM; providing over 360 cores and 3.7 Tflops (R_{PEAK}) of performance [7]. As a Tflops-scale machine, Dahl was a major upgrade from Ohm and Microwulf; it greatly accelerated several Calvin scientists' research programs, facilitating over 20 research publications.

4.4. 2008: parallel computing in CS2

By 2008, dual- and quad-core CPUs were commonplace, and it was apparent that in the near future, virtually all software would be running on multicore processors. In order for software performance to improve as it was ported from devices with fewer cores to devices with more cores, it would have to be designed and implemented with parallelism in mind. To the author, it was evident that *all* CS majors needed experience in parallel computing, especially shared-memory parallelism via multithreading. As department chair, the author saw two options for ensuring that all CS majors gained such parallel experience:

1. Add a new course on parallelism to the CS *core* curriculum (ensuring that all students took it), or
2. Incorporate parallel topics into existing courses in the *core* CS curriculum, as appropriate.

The CS curriculum is quite full, and there are other topics competing for coverage (e.g., databases, security), making option #1 problematic. Option #1 also only provides a single (concentrated) exposure to parallelism, which seemed unlikely to change students' software development skills, if they only saw and used sequential computing in the rest of their CS coursework.

For option #2, the biggest problem seemed to be getting faculty members to agree to incorporate parallelism into their core courses; some faculty members were hoping that parallelism was just a fad that would go away. However, option #2 had the clear advantage of ensuring that students would receive consistent and repeated exposure to parallelism throughout the core CS curriculum. Recent studies such as [17] and [29] indicate that introducing students to parallelism early is beneficial.

The author regularly taught Calvin's CS2 course (*Data Structures*, implemented in C++), so as a first step toward option #2, he added a week on parallelism and multithreading to each section of his CS2 course during the 2008-9 academic year. To make room for this new material, a week's worth of coverage of graphs was shifted from CS2 to CS3.

Thanks to Calvin's use of C++ in CS2, it was fairly easy to introduce multithreading using OpenMP. The author gave three lectures, in which he used OpenMP patternlets to live-demo fork-join multithreading, parallel loops, race conditions, synchronization, and reductions. He also created a lab exercise in which students: (i) timed sequential Matrix addition and transpose operations, (ii) used OpenMP to parallelize those operations, (iii) timed the parallel versions using 1, 2, 4, 6, and 8 threads, and then (iv) used a spreadsheet line-chart to visualize the changing performance. The author also had the students parallelize the other Matrix operations as homework and added four questions related to multithreading to the final exam.

The author also regularly taught Calvin's *Operating Systems & Networking* course, a core CS course in which all majors are introduced to shared-memory and distributed-memory concurrency.

It was easy to expand the shared-memory coverage with material on thread performance, software issues specific to multicore cache performance (e.g., false sharing), and so on.

With these changes, the author began to ensure that all Calvin CS majors learned about shared-memory parallelism.

4.5. 2009: high performance computing, iteration 5

In Fall 2009, the author offered Calvin's HPC course again, with an enrollment of 7 students. The course content was stable: once again, the first half covered distributed-memory parallelism using MPI and the second half covered shared-memory parallelism using OpenMP, but with Dahl replacing Microwulf. Since each of Dahl's nodes now had at least 8 cores (some had 16), students could use it to better test their programs' scalability in both the MPI and OpenMP segments of the course.

As in previous HPC offerings, students developed each program using the department's lab workstations and then moved that program to Dahl to empirically measure the program's scalability. The 2009 lab workstations contained 3.6-GHz Intel i7 (quad core) CPUs that were over 50% faster than the dual 2.2-GHz Intel Xeon (quad core) CPUs in each of Dahl's compute nodes. Because of this clock-speed difference, the lab NoW multiprocessor would typically outperform Dahl for MPI computations using 1, 2, 4, and 8 processes. Beyond that point, the performance of lab NoW multiprocessor would plateau due to the NoW's slower network and contention with other students, but Dahl's performance would continue to improve through 16, 32, 64, 128, and 256 processes. The performance cross-over point where Dahl would match the lab NoW was usually about 16 processes. Beyond about 400 processes, Dahl's performance would plateau or degrade, letting the students directly experience how hardware limitations ultimately constrain a program's scalability.

Similarly, for OpenMP computations, the lab workstations would generally outperform Dahl when using 1, 2, and 4 threads; but Dahl's nodes would prevail using 8-16 threads. The 2009 HPC course thus let students directly experience the strengths and weaknesses of NoW multiprocessors vs. Beowulf clusters, shared-memory vs. distributed-memory, and so on.

5. 2010-2015: accelerating parallelism

In 2010, the author used the last of that NSF-MRI grant's funds to add an "accelerator" node to Dahl, containing an eight-core Intel Xeon CPU, an Nvidia GeForce GTX 470 GPU, with CUDA and OpenCL installed. This and other developments set the stage for expanding Calvin's coverage of parallel topics to include accelerators.

5.1. 2010: the TCPP early adopter program

In 2010, the NSF/IEEE TCPP group began their *Early Adopter Program* [21], which offered mini-grants for CS faculty or departments to adopt their Curriculum Recommendations [26] and incorporate parallelism into CS courses. As department chair, the author applied for and received one of these mini-grants, which he then used as to provide financial incentives for his faculty members to incorporate a week's worth of parallel computing content into CS core courses, specifically:

- In Calvin's *Algorithms and Advanced Data Structures* course, the instructor added coverage of parallel algorithm design, select parallel algorithms, distributed graph algorithms, and parallel asymptotic analysis.
- In Calvin's *Programming Language Concepts* course, the instructor expanded coverage of shared memory parallelism, including a new lab exercise in which students explored the parallel mechanisms of Ada, C++ with OpenMP, and Ruby, measuring and comparing each program's scalability (or lack thereof).
- In Calvin's *Intro to Computer Architecture* course, the instructor expanded the coverage of multicore processor and cache organization, and the implications for bus bandwidth, main memory, memory controllers, and so on.
- In Calvin's *Software Engineering* course, the instructor expanded the coverage of how to create distributed computing apps for mobile devices that access cloud-computing services via those services' APIs.

All of these are core CS courses at Calvin, so the TCPP Early Adopter program played a key role in providing the incentives needed to get other Calvin faculty to incorporate parallelism into their courses, providing at least a week's coverage of PDC topics in nearly all of Calvin's core CS courses.

5.2. 2010: CSinParallel, iteration 1

In summer 2010, the author attended the ITiCSE conference in Ankara, Turkey. There he participated in a working group organized by Drs. Richard Brown and Elizabeth Shoop who had launched *CSinParallel.org*, an NSF-funded web-repository for modular parallel computing course materials that had been tested at multiple institutions [15]. One of the outcomes of this working group was an influential paper on how to incorporate parallel computing topics into the CS curriculum [16]. Another outcome was that the author was invited to join the CSinParallel project, laying the groundwork for a follow-up NSF proposal.

5.3. 2011: high performance computing, iteration 6

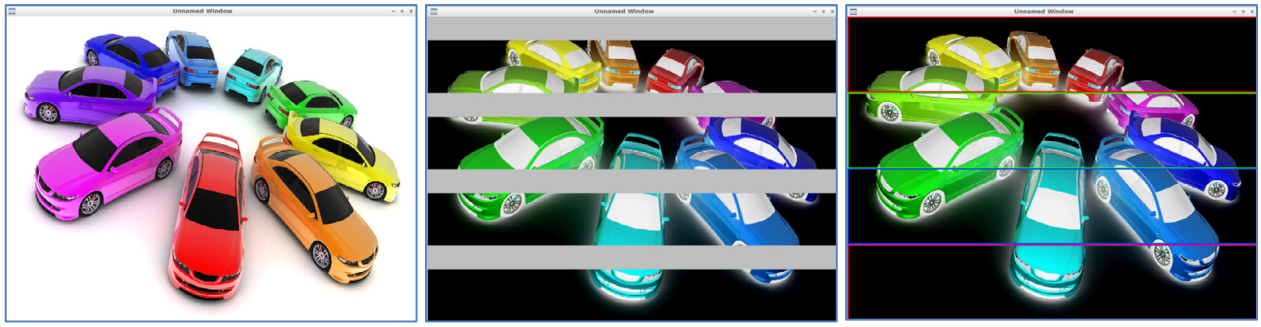
The author offered Calvin's HPC course again in Fall 2011, with 12 students enrolling in the course. The major change in this iteration was a new module on accelerators, specifically CUDA and OpenCL on GPUs. Several lectures about CUDA and OpenCL were added, plus lab exercises and projects for both of these frameworks, using Dahl's new accelerator node.

To make room for this new material, some of the OpenMP content was trimmed from the course. The topics removed were now receiving significant coverage in Calvin's core CS courses, making their coverage in the HPC course redundant.

Most students had no problems completing the CUDA material, but nearly all found OpenCL to be very challenging. OpenCL's added complexity (e.g., discovering platforms and devices, setting up queues for each, compiling the kernel for each, and so on) seemed to cross a cognitive threshold for these students; many just could not implement good OpenCL solutions in a timely fashion. The author found this very disappointing, as unlike CUDA, OpenCL was an open, non-proprietary standard.

5.4. 2012: CSinParallel, iteration 2

In 2012, the CSinParallel group applied for and received NSF TUES-2 funding to develop new parallel computing modules, expand the *CSinParallel.org* website with new interfaces, and hold faculty development workshops to help CS faculty start incorporating parallel topics into existing CS courses. Over the next few years, this group was highly productive, developing 15 new course modules; creating new search-interfaces for *CSinParallel.org* based on course, software platform, or hardware platform; hosting 19 conference or summer faculty development workshops; organizing five conference special sessions or panels; and giving numerous presentations, all promoting PDC education.



(a) Original Image (b) In-Progress Inversion (c) Completed Inversion

Fig. 8. Image inversion (4 threads).

5.5. 2013: ACM/IEEE CS curriculum 2013

In 2013, the *ACM/IEEE CS 2013 Curriculum Recommendations* (CS2013) were released [2]. Thanks to the active involvement of NSF/IEEE TCPP representatives, CS2013 included PDC as a new knowledge area and recommended that all CS majors receive significant exposure to PDC, especially the shared-memory parallel techniques needed to make efficient use of the multicore CPUs present in most devices. Thanks to the changes noted in Section 5.1, Calvin's core CS courses satisfied the CS2013 PDC Core recommendations; our HPC course covered the majority of the CS 2013 PDC Elective recommendations.

5.6. 2012-2013: coprocessors

In 2012, Adapteva released the *Parallella*, a single-board computer (SBC) that provided a 1-GHz dual-core ARM processor plus a 16-core coprocessor, for $99. (Note: With 18 1-GHz cores, the Parallella's R_{PEAK} performance was about the same as that of Ohm, Calvin's first Beowulf cluster, but where Ohm occupied 3 racks, a Parallella was the size of a credit card!) That same year, Intel released the *Xeon Phi*, a family of coprocessors offering 57 or 61 Intel-x86 cores, depending on the model purchased. (Later models offered 64, 68, and 72 cores.) In 2013, China's Tianhe-2 system burst onto the scene as the world's fastest supercomputer through its use of 48,000 Xeon Phi coprocessors. Coprocessors appeared to be the future of HPC.

During summer 2013, Calvin acquired a Xeon Phi 3120 coprocessor—a PCI card that was added to Dahl's accelerator node. Each of the Phi's 57 1.1-GHz cores contained four hardware threads, providing substantial computational power. To support communication between the cores, the Phi contained a sophisticated internal network that allowed the Phi to be used in either of two ways: (i) as a cluster-on-a-chip with MPI, or (ii) as a shared-memory manycore processor using OpenMP.

5.7. 2013: high performance computing, iteration 7

In Fall 2013, the author offered Calvin's HPC course again, with 23 students enrolling. This iteration was a significant overhaul of the course, with (i) the adoption of a new textbook, *An Introduction to Parallel Programming* by Pacheco [24]; and (ii) the use of parallel design patterns [19] as a unifying theme throughout the course. The general structure of the course remained similar to the 2011 course: the first half used MPI to explore distributed-memory parallelism; a second segment explored shared-memory parallelism using OpenMP; followed by a final segment on accelerators. However, parallel design patterns were used to provide a consistent thematic structure throughout the course, across all three segments.

The new textbook included a unit on Pthreads, so the author added a new week on Pthreads to the shared-memory segment, before the OpenMP material. After having to complete a programming project using Pthreads' explicit multithreading, the students were far more appreciative of OpenMP's implicit multithreading!

Another change was the addition of a week on heterogeneous architectures between the shared-memory segment and the accelerator segment, with lectures plus an MPI+OpenMP lab exercise and project.

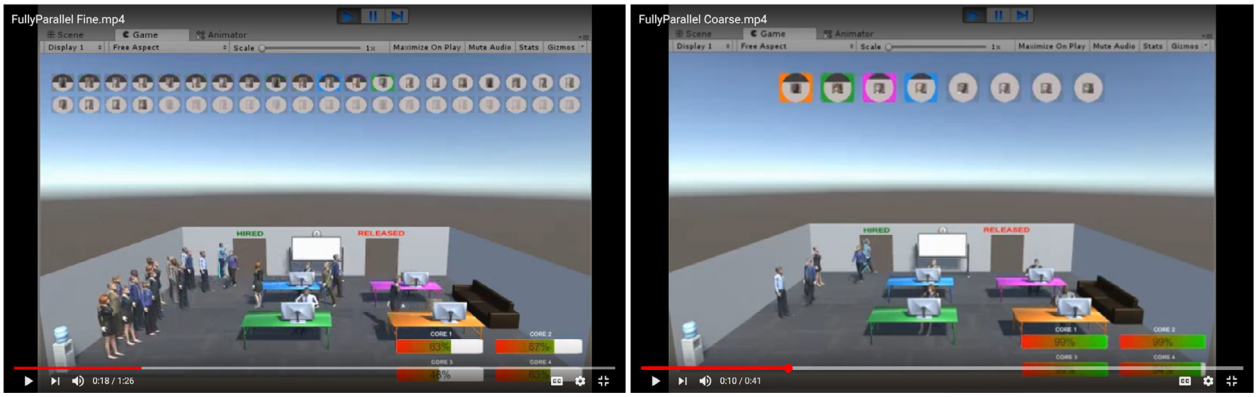
A final change was that, where the 2011 course's accelerator module only explored GPUs, the 2013 course added a week's worth of content on coprocessors. This included lectures on the Parallella and Xeon Phi architectures, plus a lab exercise and assignment to provide hands-on experience using the Xeon Phi. To make room for this content, the OpenCL content was reduced to a single lecture, leaving the CUDA content as the primary GPU component.

5.8. 2014-15: seeing parallelism

As part of his work with the CSinParallel group, the author began work in 2014 on real-time visualizations of shared-memory parallel algorithms. Shared-memory parallelism was a logical starting point, since parallel entities would need to draw on a shared canvas. This ultimately became a multiyear project that has been worked on by students Ian Adams, Zachary Chin, Patrick Crain, Christopher Dilley, Samuel Haileselassie, Elizabeth Koning, Christiaan Hazlett, Mark Vander Stel, and Ryan Vreeke.

Since existing graphics libraries were not thread-safe, the first step was to create a library that was. The result was the *Thread Safe Graphics Library* (TSGL),[1] an object-oriented C++11 library that allows multiple threads to safely draw to the same canvas in near real-time. In Spring 2015, the author ran an experiment, in which half of his CS2 (*Data Structures*) students did the existing lab exercise in which they parallelized Matrix operations, while the other half of the students did an experimental lab exercise in which they parallelized an image-processing operation. Both groups timed the operations and measured the parallel speedup, but the latter group could see the parallelization occurring in real-time, thanks to TSGL. The results of this controlled experiment provided strong evidence that the visualization improved student understanding of OpenMP's parallel for abstraction [8]. As a result, the author replaced the Matrix-processing exercise with the visual image-processing exercise in all subsequent offerings of his CS2 course. Fig. 8 shows (a) the original image, (b) the partially completed inverse of that image using 4 threads, and (c) the completed inverse of the image, with each thread's contribution outlined in a unique color.

[1] TSGL is freely available from its github repository: https://github.com/Calvin-CS/TSGL.



(a) More small tasks = more waiting, context switching (b) Fewer large tasks = less waiting, context switching

Fig. 9. Nasser Giacaman's office analogy: visualizing one-thread-per-task parallelism.

Relatedly, in May 2015, the author met Dr. Nasser Giacaman (University of Auckland) at the 2015 EduPar workshop of the *International Parallel and Distributed Processing Symposium* (IPDPS). Dr. Giacaman was developing a visual analogy of a multicore CPU, using a workplace office with four desks to represent processor cores, people to represent threads, and paperwork to represent the tasks to be performed [1]. Giacaman's visualizations allowed a user to choose: (a) a computation's *workload* (many fine-grained tasks, fewer coarse-grained tasks, or a mixture of fine- and coarse-grained tasks), and (b) its *thread policy* (sequential, one thread per task, one thread-per-core static, one thread-per-core dynamic), and then see an animation of the computation's behavior. Different choices produced markedly different behaviors. For example, if the user specified one thread per task and many fine-grained tasks, then many people (threads) would stand around waiting and more time was spent context-switching, reducing efficiency (see Fig. 9a). If the user specified a thread per task and fewer coarse-grained tasks, then fewer people (threads) stood idle and less time was lost context-switching, increasing efficiency (see Fig. 9b).

With a shared interest in visualizing parallel computing concepts, the author and Dr. Giacaman became good friends and research collaborators. Dr. Giacaman spent part of a 2018 sabbatical at Calvin; the images in Fig. 9 stem from that work.

5.9. 2015: high performance computing, iteration 8

In Fall 2015, the author offered Calvin's HPC course again, with 28 students enrolling. One change in this iteration was that the parallel visualizations created using TSGL were incorporated into the course, allowing students to see the behaviors of specific parallel algorithms and patterns (e.g., the parallel loop).

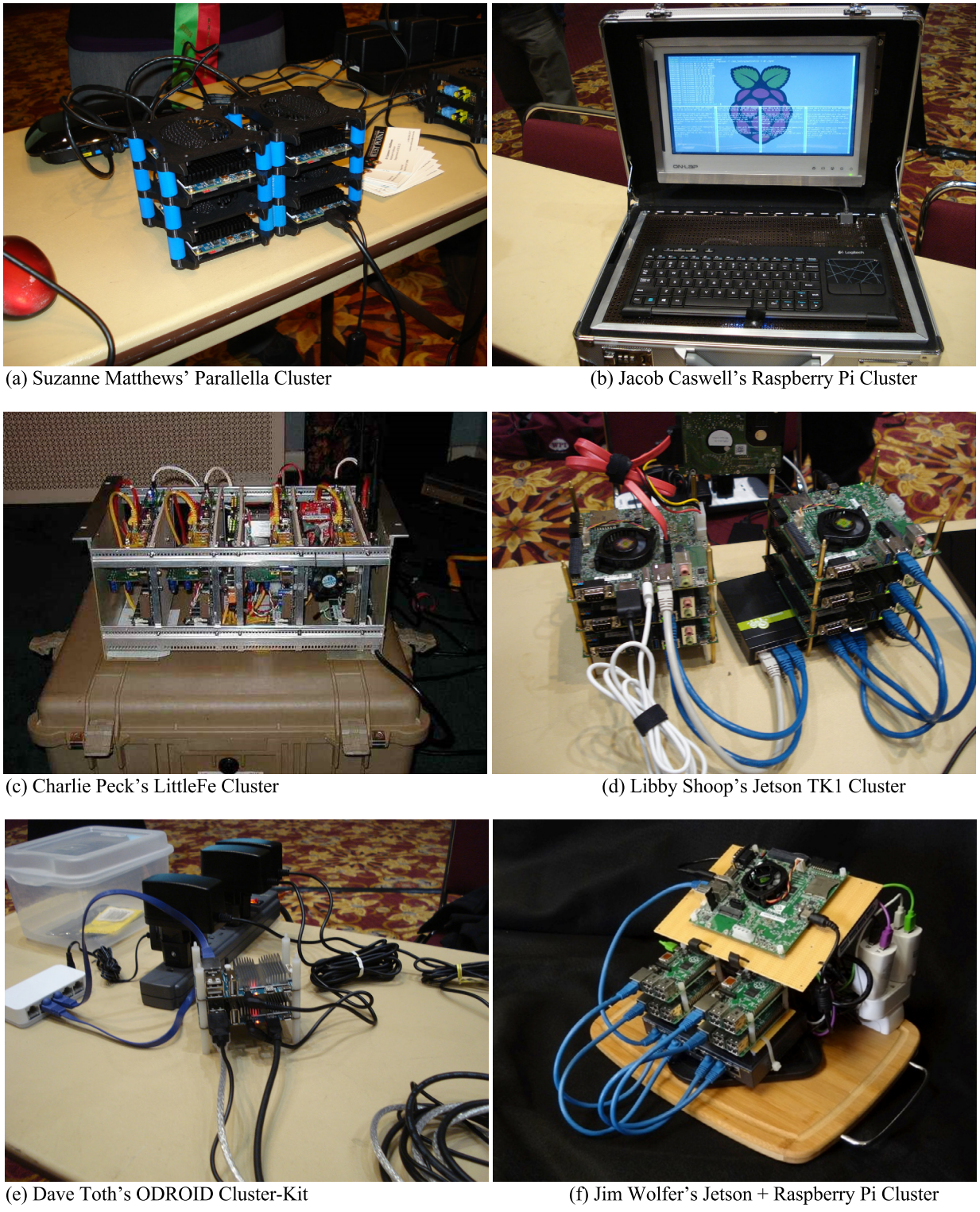
Another change was catalyzed by CSinParallel: By summer 2015, this group had authored several modules using *Apache Hadoop* [11] and its *Hadoop Distributed File System* (HDFS) to explore the *MapReduce* framework. With HDFS providing a distributed file system to spread very large data sets (i.e., those too big to fit into a single computer's memory) across a cluster, plus Hadoop for processing those data sets in parallel, this technology was interesting and useful for students to experience. To introduce students to this technology, a lecture was added explaining the MapReduce conceptual framework, plus a lab exercise to provide hands-on experience using HDFS and Hadoop. To make room for this content, the coverage of HPC history was compressed to a single lecture.

One other observation from this iteration was that Dahl was now 7 years old. The CS lab machines had been replaced twice since Dahl was created, and their improved CPU performance made it increasingly difficult for Dahl to surpass the performance of the CS lab NoW multiprocessor. Dahl's nodes also began failing, indicating it was nearing its end.

5.10. 2015-16: SBC microclusters

In 2014, the author began hearing from different individuals who were building Beowulf microclusters from single board computers (SBCs), and using those clusters as PDC learning platforms. Some of these people included:

- Jacob Caswell (St. Olaf College). An undergraduate student from St. Olaf College, Jacob had built a 5-node Beowulf cluster using Raspberry Pi SBCs, which he was using to learn about MPI distributed multiprocessing. The nodes, monitor, and keyboard (with integrated touchpad) were all housed *inside a briefcase*, at a cost of just $240.
- Dr. Suzanne Matthews (West Point). For her course, Dr. Matthews had built a cluster whose nodes were Adapteva Parallella SBCs, each with a 2-core CPU and a 16-core Epiphany coprocessor. The Parallella's co-processor was "a cluster on a chip", so the Parallella supported learning about multithreading and MPI distributed multiprocessing. Matthews was also experimenting with power-efficient clusters using Raspberry Pi SBCs as the nodes. About this time, Dr. Matthews joined the CSinParallel team, bringing new ideas and energy to the team.
- Dr. Elizabeth Shoop (Macalester College). Dr. Shoop had participated in a LittleFe Buildout and eventually decided to design and build her own 6-node microcluster using Nvidia Jetson TK1 SBCs as the nodes. Each Jetson SBC offered a quad-core ARM CPU plus a 192 CUDA-core GPU, providing a single hardware platform on which Shoop could teach her students about OpenMP multithreading, MPI distributed multiprocessing, and CUDA GPU computing.
- Dr. David Toth (Centre College). Instead of having his PDC students purchase a textbook, Dr. Toth had them purchase a kit containing two of Hardkernel's ODROID multicore SBCs, the needed power supplies and cables, plus a Tupperware container in which to store all of these. The resulting "kit" was about half the size of a shoebox—easily small enough for student to carry to and from class in their backpacks. During the first part of a semester, Toth taught his students about multithreading using one of the SBCs. During the second part, he taught his students to turn the pair of SBCs into a microcluster, which they then used as a platform for learning about MPI distributed multiprocessing. Toth thus used these inexpensive SBC kits to provide each student with their own personal, portable Beowulf cluster.



(a) Suzanne Matthews' Parallella Cluster

(b) Jacob Caswell's Raspberry Pi Cluster

(c) Charlie Peck's LittleFe Cluster

(d) Libby Shoop's Jetson TK1 Cluster

(e) Dave Toth's ODROID Cluster-Kit

(f) Jim Wolfer's Jetson + Raspberry Pi Cluster

Fig. 10. Microcluster showcase clusters.

- Dr. James Wolfer (Indiana University, South Bend). Dr. Wolfer had built a heterogeneous microcluster with an Nvidia Jetson TK1 head node and four Raspberry Pi worker nodes. The head node provided a platform for his students to learn about shared-memory multithreading and CUDA-GPU computing, while the head node plus the worker nodes let his students learn about MPI distributed-memory multiprocessing.

The author thought it would be interesting to get all of these people (along with LittleFe's Charlie Peck) into the same room at the same time, so at SIGCSE 2015 and 2016, he organized two Special Sessions titled "Budget Beowulfs" and "The Microcluster Showcase", respectively. Each participant brought their microcluster to the conference and assembled it prior to the session. Each of these Special Sessions was 90 minutes, divided into three 30-minute thirds:

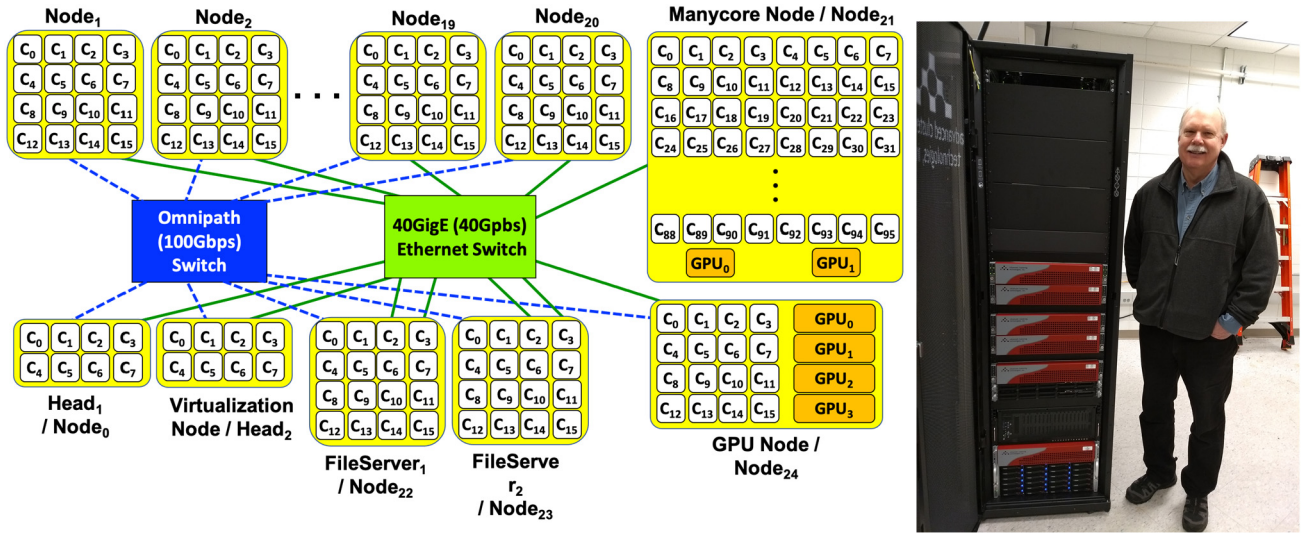


Fig. 11. Schematic and photo of Borg with the author.

1. In the first third, each person gave a brief presentation introducing their cluster and how they used it.
2. The middle third consisted of a panel discussion with the audience directing questions to the group as a whole.
3. In the final third, each participant moved to their cluster where they live-demoed it and answered individual's questions.

These Special Sessions proved to be fertile venues for sharing and cross-pollenating ideas. Participants and audience members discussed Beowulf cluster design decisions, the tradeoffs involved in different designs, funding sources, teaching strategies, and a variety of other topics related to using these devices as hardware platforms for teaching PDC. Fig. 10 shows some of the microclusters shown at these sessions.

6. 2016-present: always reforming

Parallel technology has continued to evolve. Inexpensive single board computers (e.g., the Raspberry Pi) have gained multicore CPUs, new languages have appeared with built-in parallel computing support, and some older languages have enhanced their support for parallelism. As the parallel computing landscape continues to evolve, PDC educators must also adapt by keeping their courses up to date.

6.1. 2016-18: a Beowulf cluster for data science

In 2016-17, Dahl was deteriorating too much to support the needs of local research projects. At the same time, the author chaired a Calvin committee to design a new Data Science B.S. program. To support Calvin research and this new program (i.e., to store and process large data sets), the author wrote an NSF-MRI proposal for a new multidisciplinary research cluster costing about $260,000. This proposal was funded late in the summer of 2017. The author spent the Fall of 2017 designing and then purchasing the new cluster, which was named *Borg* in remembrance of computer scientist Anita Borg.

To process data, Borg was configured with:

- A head node with two 3.6-GHz Xeon Gold (4-core) CPUs and 96 GB of RAM.
- Twenty compute nodes, each with two 3.2-GHz Xeon Gold (8-core) CPUs and 96GB of RAM.
- A virtualization node for running the web services needed by different research projects. This node was hardware identical to the head node, so that it could serve as a failover replacement for the head node if necessary.
- A GPU/high-memory node containing the same CPU configuration as the compute nodes, but with 768 GB of RAM, and four Nvidia Titan V graphics cards. Each of these cards has 5120 CUDA cores for GPU computations plus 640 Tensor cores for machine learning projects.
- A many-core node, containing two 2.2-GHz AMD EPYC (48-core) CPUs, 512 GB of RAM, and two Nvidia Quadro P2200 GPUs, each with 1280 CUDA cores and 5GB VRAM.
- Two file-server nodes (one as primary, one as backup), each with 100TB of tiered RAID storage, equipped with NVMe hardware and extra network links to support parallel access.

The nodes were connected using a 100Gbps Omnipath network for data and a 40Gbps Ethernet network for administrative traffic. Fig. 11 shows a schematic and photo.

Borg became operational in May 2018; it was immediately put to heavy use by Calvin researchers.

6.2. 2017: high performance computing, iteration 9

In Fall 2017, Calvin's HPC course ran again, enrolling 21 students. Because Borg was still being designed that semester, Dahl was used a final time. This course was thus almost the same as the 2015 course; one minor difference was that some of the lecture material on MapReduce and Hadoop was revised to explore the newer, better-performing *Apache Spark* [12] technology.

6.3. 2017-18: Crayowulf

During the 2017-18 academic year, the author designed and supervised a senior capstone project to build *Crayowulf*, a $2500 microcluster commemorating the Cray-1 supercomputer with a cylindrical tower case and liquid-cooling system [9]. Crayowulf has five nodes, each having an Nvidia Jetson TX-2 SBC (a 2-GHz six-core ARM CPU-complex plus a 256 CUDA-core GPU), connected with Gigabit Ethernet and NFS-mounted shared solid-state storage. Fig. 12 is a schematic and photos of the system.

Thanks to the computational density of the Jetson SBCs, Crayowulf has a total of 30 CPU cores and 1280 CUDA cores.

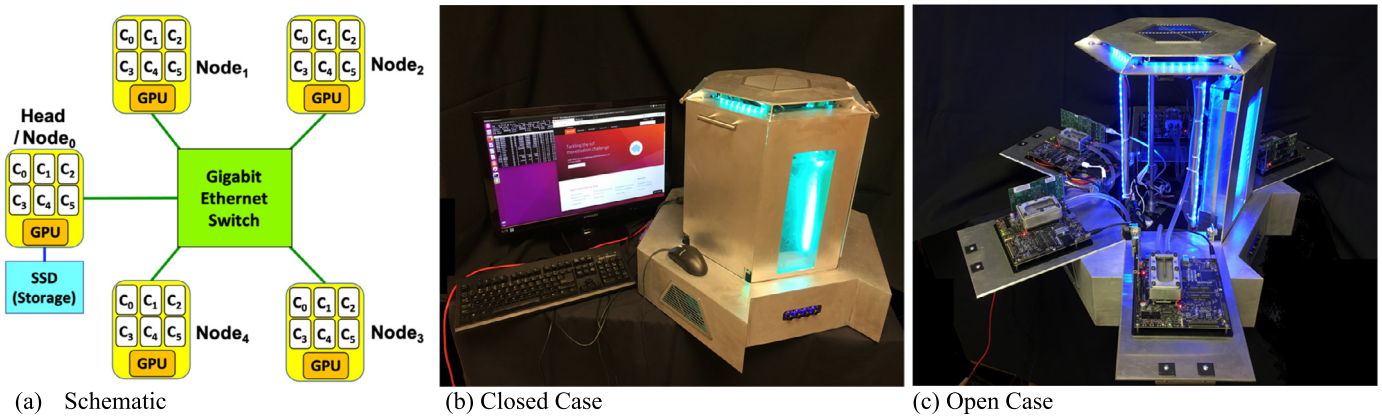


Fig. 12. Crayowulf.

A multidisciplinary team of students worked on the project, with the following primary responsibilities:

- Philip Holmes, a mechanical engineering major, built the hexagonal aluminum tower enclosure, which was designed to have a unique "flower petal" structure to provide easy access to the interior of the cluster, as shown in Fig. 12c.
- Benjamin Kastner, a CS major, installed and configured the operating system, network services, and software libraries (e.g., MPI, CUDA) needed to run homogeneous MIMD or SIMD applications, or a heterogeneous mixture of the two.
- Peter Oostema, a CS + electrical engineering double major, built the power system, the lighting system, and wrote specialized application software.
- Noah Pirrotta, a mechanical engineering major, built a custom closed-loop liquid-cooling system for the Jetson SBCs, using de-ionized water as coolant. Testing revealed that this kept the system about 9 °C cooler than air-cooling.

The project was a resounding success. Crayowulf's unique "flower petal" design lets students see and touch a multiprocessor's parallel hardware components, and thus build accurate mental models of the hardware on which a parallel computation runs. From the time it was completed until it was shut down for the 2020 pandemic crisis, Crayowulf was used for student MPI and CUDA projects, with spare cycles devoted to the SETI@Home project.

6.4. 2018: CSinParallel, iteration 3

In 2018, the CSinParallel group applied for and received an NSF-DUE Level 2 grant to fund a project titled "*Seeing, Hearing, and Touching Parallel Computing*." Building on the previous NSF-TUES 2 work, the author continued to work on TSGL to explore new ways to help students to *see* parallel algorithmic behavior. However, since visualization offers limited benefits for students with visual impairments, the author also began exploring *sonification*—the sonic equivalent of visualization—by which students can *hear* algorithmic behavior. To support the creation of such sonifications, the author and student Mark Wissink have created the *Thread Safe Audio Library* (TSAL[2]), by which a program may be annotated with method-calls that generate sounds that provide insight into the program's algorithmic behavior. To illustrate, imagine that a sorting algorithm plays a tone whose frequency is scaled to the value the algorithm is currently processing. Figs. 13 and 14 present spectrograms of what one hears when InsertionSort sorts 150 values vs. QuickSort sorts 10,000 values (so that both take about the same time).

As can be seen in Figs. 13 and 14, these algorithms *sound* very different as they run! The descending arcs in Fig. 13 occur as InsertionSort iterates backwards through the sequence, seeking the insertion point for the current value. The clumps seen in Fig. 14 are QuickSort's recursive partitioning of a sequence into subsequences using pivot values; the curve that gradually ascends from left to right reflects how the pitch increases as the recursively-sorted subsequences are concatenated.

For contrast, Fig. 15 shows the spectrogram of parallel MergeSort on 25,000 values using 16 threads. Once again, this sonification reveals a "sonic signature" that is distinct from that of the other sorting algorithms. Since each algorithm sounds distinctly different from the others, such sonifications have the potential to help students—especially those with visual impairments—better understand algorithmic behavior, whether sequential or parallel.

Others in the CSinParallel group are exploring ways to let students experience parallel algorithmic behavior by *touch*. For example, Dr. Suzanne Matthews is curating a variety of "unplugged" activities for PDC concepts,[3] Dr. Richard Brown has been examining the benefits of constructing microclusters using single board computers, and other ideas involving tactile exploration.

6.5. 2019: high performance computing, iteration 10

In Fall 2019, the author offered Calvin's HPC course again, with 33 students enrolling, a record high enrollment. This iteration was similar to the 2017 offering, aside from two changes.

One change was that Borg was integrated into the course, replacing Dahl. At this time, the CS lab workstations had 3.6-GHz Intel i7 CPUs, compared to the 3.2-GHz Xeon Gold CPUs in Borg. However, with 33 students taking the course (i.e., saturating the lab), Borg's scalability and faster network generally gave it the edge, especially on communication-intensive computations.

The other change stemmed from two 2017 developments: (i) Intel's announcement that it was halting further work on its "Knights Hill" Xeon Phi family, and (ii) Adapteva's halting its production of the Parallella SBC. Taken together, these developments cast serious doubt on the future of coprocessor-based parallel computing. As a result, the author replaced the coprocessor content and Xeon Phi materials from the 2017 course with expanded coverage of GPU computing, to provide students with additional depth and hands-on experience using CUDA.

[2] TSAL is freely available from its github repository; https://github.com/Calvin-CS/TSAL.

[3] https://www.pdcunplugged.org.

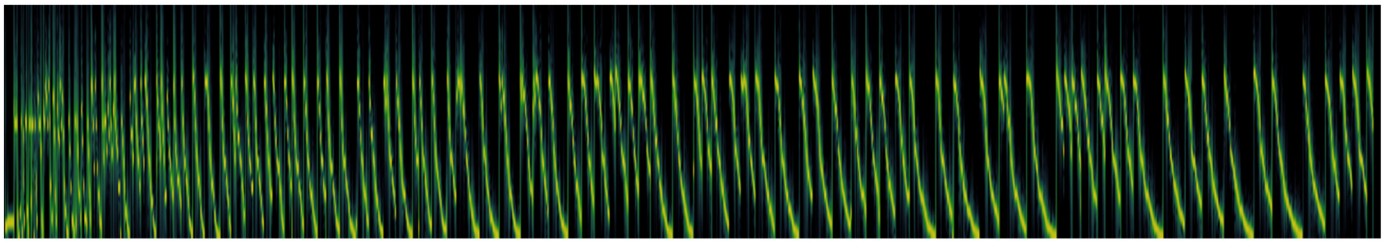


Fig. 13. Spectrogram of InsertionSort (150 values).

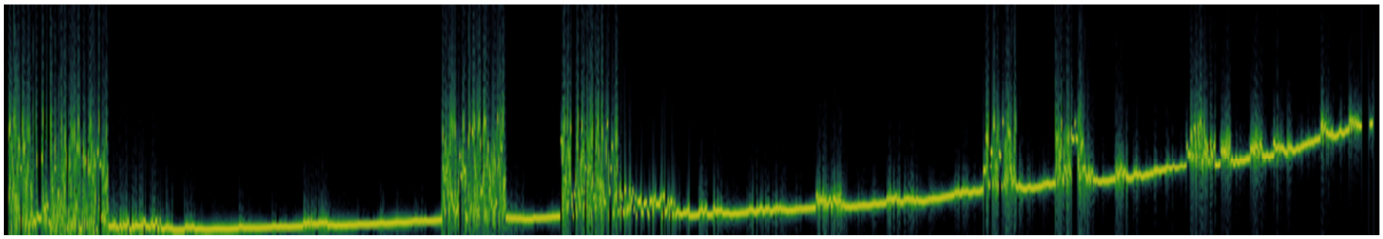


Fig. 14. Spectrogram of QuickSort (10,000 values).

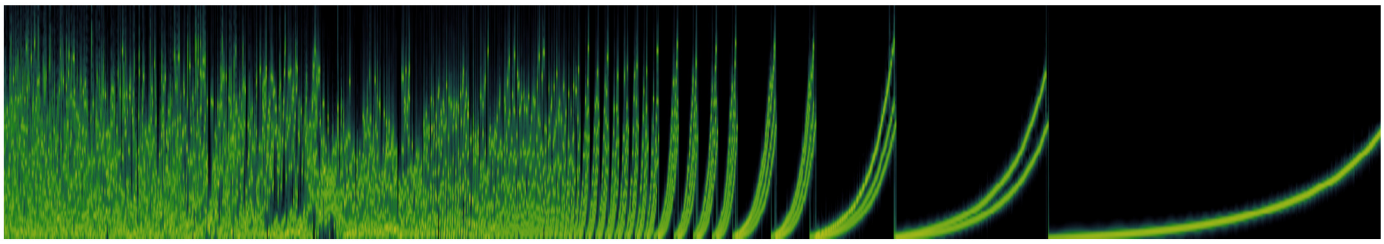


Fig. 15. Spectrogram of parallel MergeSort (25,000 values), 16 threads.

Table 1 summarizes the coverage of specific technologies and topics in the author's Parallel Computing / HPC course, from its inception in 1998 to the 2019 offering, with the significant platform changes in a given year highlighted in red.

It is worth noting that those parts of the course that have remained the most stable through the years are those based on the open standards MPI and OpenMP, as opposed to proprietary technologies.

7. Discussion

The changes to PDC education over the past 20+ years have been breathtaking. When the author began his journey in 1996, a Cray T3D-256 supercomputer that could achieve 25.3 Gflops cost millions of dollars. In 2007 (a decade later), Microwulf could achieve 26.25 Gflops for less than $2,500. Today, a single graphics card can outperform either one. This section explores insights the author has gained through his years of teaching PDC, with respect to infrastructure and pedagogical issues.

7.1. Infrastructure for teaching PDC

When it comes to PDC infrastructure, this is a golden age, as PDC educators have many options from which they may choose. In this author's opinion, the infrastructure option(s) a PDC educator should choose depends on two interrelated factors:

- The instructor's desired PDC *student learning outcomes* (SLOs), including the Bloom levels of those outcomes.
- The *budget* available for the infrastructure.

To illustrate, consider the following scenarios:

Table 1
Parallel / high performance computing at Calvin, 1998-2019.

Week	1998	2000	2002	2003	2005	2007	2009	2011	2013	2015	2017	2019
1	Parallaxis	Parallaxis	Parallaxis	MPI	MPI	MPI	MPI	MPI	MPI	MPI	MPI	MPI
2	Parallaxis	Parallaxis	Parallaxis	MPI	MPI	MPI	MPI	MPI	MPI	MPI	MPI	MPI
3	Parallaxis	Parallaxis	Parallaxis	MPI	MPI	MPI	MPI	MPI	MPI	MPI	MPI	MPI
4	Parallaxis	Parallaxis	Parallaxis	MPI	MPI	MPI	MPI	MPI	MPI	MPI	MPI	MPI
5	Parallaxis	Parallaxis	Parallaxis	MPI	MPI	MPI	MPI	MPI	MPI	MPI	MPI	MPI
6	MPI	MPI	MPI	MPI	MPI	MPI	MPI	MPI	MPI	MPI	MPI	MPI
7	MPI	MPI	MPI	OpenMP	OpenMP	OpenMP	OpenMP	OpenMP	Pthreads	Pthreads	Pthreads	Pthreads
8	MPI	MPI	MPI	OpenMP	OpenMP	OpenMP	OpenMP	OpenMP	OpenMP	OpenMP	OpenMP	OpenMP
9	MPI	MPI	MPI	OpenMP	OpenMP	OpenMP	OpenMP	OpenMP	OpenMP	OpenMP	OpenMP	OpenMP
10	MPI	MPI	MPI	OpenMP	OpenMP	OpenMP	OpenMP	OpenMP	Hetero	Hetero	Hetero	Hetero
11	MPI	MPI	MPI	OpenMP	OpenMP	OpenMP	OpenMP	CUDA	CUDA	CUDA	CUDA	CUDA
12	History	History	History	History	History	History	History	OpenCL	XeonPhi	XeonPhi	XeonPhi	CUDA
13	History	History	History	History	History	History	History	History	History	Big Data	Big Data	Big Data

Scenario 1. Suppose an instructor's SLO is that students will "describe" or "explain" PDC concepts (i.e., at the lower Bloom levels). Then no hands-on training is required to achieve the SLO; such SLOs can be achieved via traditional lectures or active-learning exercises such as those available on Suzanne Matthews' *PDC Unplugged* website.[4] Without an SLO that requires students to at least *apply* PDC concepts, no special infrastructure or budget are needed.

Scenario 2. Suppose an instructor has an SLO that students will write multithreaded programs and analyze their scalability on shared-memory multiprocessors. One way to achieve this is to have students write multithreaded programs and then time their executions on a shared-memory multiprocessor using different thread counts. If a student records those execution times in a spreadsheet, s/he can create charts, compute and compare the programs' speedups, and so analyze their programs' scalability.

For software to achieve this SLO, any language or library that supports multithreading can be used. A few of these options are: C/C++ with POSIX threads, C/C++ with OpenMP, C++11 threads, Go, Java, Rust, Scala, and many others.

In terms of hardware, there are a number of infrastructure options available, depending on the available budget:

- If there is no budget, but there is a lab of multicore workstations available, then students can use those workstations to experience scalability up to the number of cores it provides.
- If there is no budget or lab, but the students own laptops, virtually every current laptop has a multicore CPU, so students can use their laptops to experience scalability up to the number of cores available on their laptops.
- If there is a modest budget available, an alternative approach is to equip each student with an inexpensive SBC such as the Raspberry Pi. (This can also be accomplished with no budget via a course fee.) The vast majority of today's SBCs have multicore CPUs, so a student can use their SBC to experience scalability up to the number of cores on the SBC.
- If there is a rich budget available, yet another approach is to purchase a manycore system (e.g., a 128-core server). With so many cores at their disposal, students will be able to experience the scalability of their programs up to the number of cores on the system—a far greater degree than on more common multicore machines. In this case, a reservation system or batch queueing system is helpful to prevent the students' computations from interfering with one another.

Scenario 3. Suppose an instructor has an SLO that students will write message passing programs and measure their scalability on a distributed-memory multiprocessor. One way this SLO can be achieved is by having students write parallel programs that communicate via message passing and then time their executions using different numbers of processes. If students record those times in a spreadsheet, they can easily compare those execution times, create charts, compute their speedups, and so on.

For software to achieve this SLO, any language or library that supports message passing might be used. C/C++ with MPI, Erlang, Julia, and Scala all support message passing, so any of them might be used to achieve this SLO.

For hardware, there are a variety of distributed multiprocessor options available, depending on one's budget:

- If there is no budget but a lab of multicore workstations is available, these workstations can be configured as a NoW multiprocessor. As noted earlier, a modern NoW can serve as a powerful distributed multiprocessor, because a typical student working at a workstation is using just one of its cores. If that workstation is a quad-core machine, that leaves three other cores free to host the processes of distributed computations.
  In Calvin's NoW, MPI is configured to run one process per node and we provide students with a script to generate randomly-ordered MPI host files. When students run their MPI computations using those host files, their computations are distributed randomly across the NoW, spreading the demand for CPU resources across the lab. This provides the students with reasonably accurate scalability results until the NoW becomes saturated with processes, which is often sufficient for teaching students about parallel scalability.
- If there is no budget for building a distributed multiprocessor and no lab, but students have laptops, then those laptops can be used as front ends to a cluster. There are several cluster options, including:
  - If the instructor's institution replaces its computers on a regular basis, then a local Beowulf cluster can be built from old but serviceable PCs. Since Dennard scaling ended in 2005, today's "old" PCs (i.e., 3-to-5 years old) may be nearly as fast as a new PC. By having students help build a local Beowulf cluster from such PCs, students can gain valuable hands-on experience building the cluster, enjoy the pride of "owning" it, and then use it to experience the scalability of their distributed programs until the point at which their processes saturate the cluster.
  - If building a local cluster from old PCs is not an option, a cloud-based research cluster may be an option. In the U.S., research clusters such as Chameleon[5] and XSEDE[6] offer free grants of time for educational use. These are supercomputers with thousands of nodes and tens of thousands of cores, allowing students to experience the scalability of their programs far beyond what is possible on most local clusters. This has the advantage of providing students with the authentic experience of using a real-world supercomputer.

  Note that in both of these options, students should run their computations using a batch scheduler, to keep them from interfering with other users' computations.
- If there is a budget available for acquiring a new Beowulf cluster, then acquiring such a cluster and having the students run their distributed computations on it is another option. Some of the ways this can be done include:
  - *A centralized local cluster*. In this approach, one's university uses institutional funds, corporate support, or a government grant to either acquire a new Beowulf cluster from a vendor, or purchase the components for a new Beowulf cluster and build it in-house. The cost of such a cluster is the sum of the features one chooses (number of nodes, processor models, amount of memory, amount of storage, network technology, backup system, and so on). In addition, the university must provide a secure space to house the cluster, with adequate climate control, electrical power, and (ideally) an uninterruptable power supply. The university should also commit to pay a person to administer and maintain the cluster, on at least a part-time basis. Students can then run their message passing programs on the cluster (ideally by submitting them to a batch queueing system, to avoid conflicts), and experience their programs' scalability up to the number of cores on the cluster.
    One disadvantage of this approach are the costs: the up-front cost to purchase the cluster and the on-going costs to maintain the cluster. Another disadvantage is that such

[4] http://www.pdcunplugged.org/.

[5] https://www.chameleoncloud.org.

[6] https://www.xsede.org.

hardware historically has a relatively short lifetime before it becomes obsolete (e.g., 5-8 years in this author's experience). With Dennard scaling having ended and Moore's Law slowing down, it is possible that this will change, but heavily used equipment wears out, sooner or later.

– *Per-student SBC clusters*: In this approach, the instructor provides each student with a kit containing two or more SBCs, cables, and any other hardware needed to assemble the SBCs into a microcluster. Using a disk image and following directions provided by their instructor, each student creates a personal Beowulf cluster from their kit. If a secure lab space is available, students may keep their clusters there between classes, to minimize setup and teardown time. Otherwise, thanks to the modest size of an SBC, the instructor can provide each student with a small plastic container to store and protect their cluster; students transport their cluster to/from class in their backpack. Each student can write and run their message passing programs on their personal cluster, and experience scalability up to the number of cores on their cluster.

  Depending on the SBC chosen and the other hardware provided in the kit, this approach can cost less than $100 per student, which can be funded via institutional funds, or by charging the students a course fee. An alternative is for the instructor to provide each student with one SBC and have the students work in groups.

  An advantage of this approach is that SBCs like the Raspberry Pi, Hardkernel's ODROID devices, and Nvidia's Jetson devices make it possible to build such clusters at a variety of price points—in some cases, for less than the price of a student textbook. Another advantage is that students can run their message passing programs directly on their personal clusters, without going through a batch queueing system. Another is that the students own their clusters at the end of the course; the university is not left with an expensive and depreciating cluster to maintain. The main disadvantage is that the instructor must spend significant time at the outset organizing and purchasing the kits, configuring the SBC storage images, creating the directions the students follow to build their clusters, and so on.

Scenario 4. Suppose an instructor has an SLO that students will write and run a program on a distributed-memory heterogeneous multiprocessor. Some of the options to accomplish this include:

- If there is a NoW or a Beowulf cluster available where each workstation or node has a multicore CPU, then one way to achieve this SLO is by having the students write and run programs that use MPI+OpenMP. In this approach, MPI is used to launch one process on each workstation or node and for communication between those processes; each process uses OpenMP to spawn threads that utilize all of the CPU-cores on that node to solve that process's part of the problem.
- If there is a NoW or a Beowulf cluster available where each workstation or node has a GPU, then some ways to achieve this SLO are by having the students write and run programs that uses MPI+X, where X is CUDA, OpenACC, OpenCL, or OpenMP if the GPUs are made by Nvidia; and X is OpenACC, OpenCL, or OpenMP if the GPUs are not by Nvidia. As before, MPI is used to launch a process onto each node and for interprocess communication; each process uses CUDA, OpenACC, OpenCL, or OpenMP to run a kernel on the GPU that solves that process's piece of the problem.

Summary. As the four preceding scenarios indicate, when it comes to infrastructure, this is a golden age for PDC instruction, as there are free software resources and many hardware options at a variety of price points. In making infrastructure decisions, an instructor's SLOs, their Bloom levels, and the available budget are all key factors that should drive the decisions. Fig. 16 presents a decision-tree for identifying the hardware needed to achieve shared-memory vs. distributed-memory SLOs.

Regardless of one's local situation, the key steps are to: (i) specify the desired SLOs at the outset, (ii) determine the infrastructure

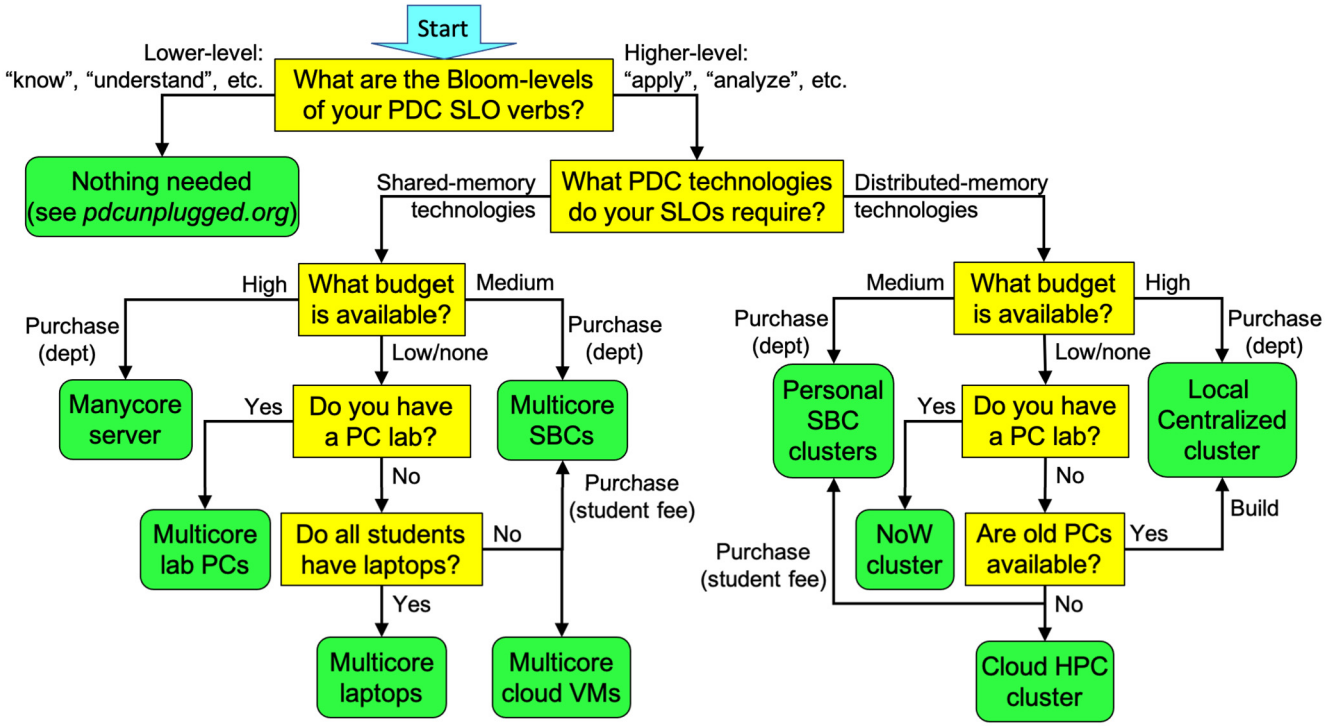


Fig. 16. A decision tree for hardware infrastructure to achieve select PDC SLOs.

resources needed to achieve those SLOs, and (iii) seek out those resources (or the funding needed to acquire them).

If a given SLO cannot be achieved with one's local situation, it may be necessary to revise that SLO or delay it until that situation can be changed. To illustrate, in the first two offerings of his parallel computing course (1998 and 2000), the author's SLOs including having his students (i) write parallel programs using MPI, and (ii) measure the speedup of those parallel programs. The first SLO could be readily achieved by configuring the department lab as a NoW, but the second SLO could not be achieved using that NoW, since each workstation had a single core, the network was slow, and the lab was heavily used. That second SLO had to be deferred until the 2002 course, when NSF-MRI funding allowed a dedicated cluster to be acquired, which in turn facilitated the achievement of that SLO.

7.2. PDC pedagogy

The primary task of PDC educators is to serve their students. To this author, one aspect of *serving* students is to help them acquire useful knowledge and skills, and not teach them ephemera. In this section, we explore how we can best serve our students through *what* we teach, and *how* we teach it.

Technologies with open standards. In this author's experience, technologies that have official, open standards tend to be fairly stable (i.e., *not* ephemeral). Such standards will naturally evolve over time in response to user feedback, but such evolutionary changes are usually backward-compatible, providing the helpful stability. The time a PDC educator invests in gaining knowledge and skill with such technologies is time well-invested, since s/he can continue to draw on such knowledge and skills for many years. To illustrate, the time this author invested in learning MPI, OpenMP, POSIX multithreading, and similar open technologies was time well-spent, since he could utilize that knowledge over many years and in different courses.

As a corollary, PDC educators should be cautious about building their courses around proprietary technologies, as the company that develops such a technology may (for business reasons) decide to drop it with little warning. If that happens, the time the educator spent learning about that technology has been wasted, since the knowledge and skills are now obsolete. This author made that mistake with Intel's Xeon Phi co-processor, though thankfully that was only one week of his HPC course. PDC educators should be careful when adopting "hot, cutting edge" technologies unless they are willing to "bleed" a bit.

Proprietary technologies. However, PDC educators cannot entirely ignore proprietary technologies and serve their students well: If a proprietary technology dominates its market, that technology may become the *de facto* standard for that market. CUDA is a current example of this: Nvidia dominates the GPU market and CUDA dominates the GPU-software market as the technology for writing the most efficient data-parallel programs for Nvidia GPUs. As a result, PDC educators are (in this author's opinion) remiss if they do not at least introduce their students to CUDA. Thankfully, Nvidia makes CUDA freely available to educators and developers, even though it is not fully open-source software.

The reader may be asking, "What about the 'hot' technologies in which my students are interested? Do we just ignore such technologies?" This author's approach is to end one's course with an open-ended final project, in which the students must write about and/or present a particular "hot" technology of their choice. Turn them loose! Students have amazing energy, creativity, and enthusiasm, and such end-of-course projects provide positive channels for their interests and energies. Such projects can also open interesting doorways, as Mark Ryken's fascination with Loki did for this author over 20 years ago.

Parallel design patterns. Another way PDC educators can serve their students well is by teaching them the best practices identified by professional parallel software developers over decades of experience, also known as *parallel design patterns*. Having passed the test of time, these patterns are a stable body of knowledge amidst the turbulence of PDC technologies. Thanks to that stability, these parallel patterns are well worth mastering and teaching to our students, as they have excellent potential to remain useful throughout our and our students' careers.

As a simple illustration, if a student has completed a traditional CS1 course and is then given a matrix operation to implement, we would expect that student to know to use two nested for loops to process the matrix—nested loops are a common sequential pattern for processing 2D structures. If a student has completed a PDC course and is given the same problem, then we would expect the student to know to use a *parallel loop*—a common parallel pattern—for the operation's outer loop, to speed up the operation. Depending on the PDC course, we might also expect the student to think of the matrix operation as a *dataflow problem*—a different parallel pattern—and consider solving it by using a kernel that runs on the GPU.

The more we can get our students to think in terms of these parallel patterns, the more closely their thinking will resemble that of a professional parallel software developer, and so the better we will be serving the students. The above-mentioned *patternlets* provide an excellent way to introduce students to these parallel design patterns and the syntax needed to implement them.

PDC early and often. If a student sees only sequential computing during most of their university program and never sees parallelism until the final year, their early-year experiences will almost inevitably impose "sequential fetters" on their algorithmic thinking. It is very difficult for a single, upper-level PDC course to free a student's mind from these "sequential fetters."

One way to avoid this problem is to introduce parallelism as early as possible and incorporate it throughout the computing curriculum as frequently as possible. For example, a CS1 course can use active-learning exercises from *pdcunplugged.org* to introduce introductory students to parallel thinking without introducing new programming language syntax. Likewise, a *Data Structures* course can use a data structure to store a very large data set (e.g., an organism's genome) that takes a "long" time to process sequentially, providing a natural motivation for using parallelism to speed up the processing. Other courses (*Algorithms*, *Operating Systems*, *Programming Languages*, ...) also provide natural places to introduce PDC topics. We can serve our students well by introducing PDC topics early and revisiting them often.

Experiential teaching. When it comes to pedagogy, the more we can provide our students with direct, hands-on experiences with technologies in which they apply PDC concepts, the more likely they are to internalize those concepts. For example, the preceding paragraph noted that a *Data Structures* course can motivate parallelism by having the students process a large data set that is stored within a data structure. Most students have no prior experience with problems that require longer than one second to solve. (This author has had students mistakenly kill 10-second programs, assuming they were "hung.") By requiring students to solve problems that take ten or more seconds to solve sequentially, a PDC instructor can leverage a student's impatience with the sequential solution, using that impatience as motivation for the student to create a parallel solution. Watching the execution time drop as a parallel solution employs more threads or processes can be a visceral learning experience for many students.

To explore the scalability of a parallel program interactively, students need to be able to control the *degree* of parallelism—the number of processes or threads being used—when executing

the program. MPI lets the user to specify the number of processes using the `-n` switch when the program is launched from the command line. For example, to run the MPI program `mpiProgram` with eight processes, one can enter:

```
mpirun -n 8 ./mpiProgram
```

One can achieve a similar control capability in OpenMP by using C's `argc` and `argv` parameters to retrieve the desired number of threads via a command line argument. If an OpenMP program named `ompProgram` begins as follows:

```
#include <omp.h>     // omp functions
#include <stdlib.h> // atoi()

int main(int argc, char** argv) {
   int numThreads = (argc >= 2) ? atoi(argv[1]) : 1;
   omp_set_num_threads( numThreads );

   // ... rest of OpenMP program
}
```

then running the program with no command line arguments will run the program with a single thread, but to run `ompProgram` with eight threads, one can enter:

```
./ompProgram 8
```

Giving students the ability to control the degree of parallelism in their programs and then having them use it to explore how varying the degree of parallelism affects their program's execution time is one way to support their experiential learning.

Visualization. Many of the concepts in PDC education are highly abstract, making it easy for students to create incorrect mental models of those concepts. For example, suppose an array `arr` contains a large data set, a for loop is being used to process that array, and an OpenMP parallel loop is used to parallelize the iterations of the for loop:

```
#pragma omp parallel for
for (int i = 0; i < SIZE; ++i) {
   process( arr[i] );
}
```

What actually happens if two threads are used? Does thread 0 process the first half of the array and thread 1 process the second half, or does thread 0 process the even array entries and thread 1 process the odd array entries, or does thread 0 process the odd entries and thread 1 process the even entries, or perhaps something entirely different? For four semesters, the author taught his students the actual behavior of this loop through both a lecture and a hands-on lab exercise, but on their final exams, few students were able to correctly answer a multiple-choice question about this behavior.

However, after the author changed the lab exercise to the TSGL-based "visual" image-processing exercise shown in Fig. 8, (statistically) significantly more students answered that same question correctly on their final exams. By having the students vary the number of threads and then *seeing* the resulting image-parallelization behavior happening in real time, the students built a more accurate mental model of the parallel loop's behavior. The author believes this principle should guide our PDC pedagogy—PDC educators should do everything they can to create interactive visualizations that help their students build accurate mental models for the abstract concepts being taught. Nasser Giacaman's office-analogy visualizations (shown in Fig. 9) follow this principle to help students understand the interactions of several abstractions: thread scheduling policy, task granularity, context-switch overhead, and load balancing.

However, visualizations are not effective for all students (e.g., those with visual impairments). For these students, TSAL *sonifications* (see Section 6.4) have the same goal: to help students build more accurate mental models of abstract behaviors.

Energy consumption. One of the current challenges facing the HPC community is limiting the energy consumption of high-performance systems. The current strategy to address this challenge is to design high-end systems as heterogeneous clusters of nodes with manycore CPUs and/or accelerators. Accelerator-based designs are especially effective, as they reduce the overall energy consumption by reducing the physical distance a computation's data-bits must be moved in order to be processed.

A related challenge is to develop the software for such systems. To use such systems efficiently, developers must create equally heterogeneous software solutions. The current strategy is to create MPI+X hybrid applications, where MPI is used to distribute a computation across the system's nodes, and X is CUDA, OpenACC, OpenCL, or OpenMP for nodes with accelerators; or X is OpenMP for multicore nodes without accelerators.

Research on the best way to teach students how to develop such hybrid software is still in its infancy, as little work has been done on pedagogy for heterogeneous systems. Even basic concerns—such as whether this should be taught at the undergraduate or graduate level—are still open questions. Research into such questions may be the subject of a future report.

8. Conclusions

This paper has explored the evolution of PDC education between 1997 and 2020, in response to changing PDC tools and technologies. A few of these changes include: the rapid spread of Beowulf clusters in the late 1990s; the development of open software standards such as MPI, OpenMP, and others; the release of relatively inexpensive GPUs in the early 2000s and software APIs for programming them; the appearance of multicore CPUs in 2006 and languages and/or software APIs for multithreading; the proliferation of inexpensive SBCs in the early 2010s; and many others. While Dennard scaling ended in 2005, Moore's Law has remained in effect throughout this period, causing the computational density of computing systems to consistently increase. As a result of these changes, the state of the art has been ever-changing, as can be seen by comparing this author's seven Beowulf cluster designs and the six SBC clusters designed by others (see Fig. 10). It will be interesting to see what will happen when Moore's Law ends, which is predicted to occur in the near future.

When it comes to infrastructure, PDC educators have numerous options by which they can introduce their students to PDC. For hardware, these options include personal laptops and desktops, SBCs, NoWs, GPU-equipped workstations, centralized local Beowulf clusters, personal SBC-based microclusters, and cloud-based research clusters. Software options include languages with built-in multithreading capabilities (e.g., C++, Go, Java); languages with built-in message-passing capabilities (e.g., Erlang, Scala); C/C++/Fortran with standards-based libraries such as OpenMP, MPI; C/C++ with POSIX sockets and threads; C with accelerator libraries like CUDA, OpenACC, and OpenCL; newer languages like Chapel,[7] and other options. While the plethora of options may seem overwhelming, this abundance is a Good Thing, as it allows an instructor to achieve their desired student learning outcomes by introducing their students to PDC using whatever approach best fits their local environment.

[7] https://chapel-lang.org.

Curated sites such as *CSinParallel.org*,[8] the *Center for Parallel and Distributed Computing Curriculum Development and Educational Resources* (CDER),[9] and *Peachy Parallel Assignments*[10] can help an instructor by providing tested, ready-to-use resources to achieve particular SLOs. Visualization (and perhaps sonification) tools are example resources that can be used to help students build accurate mental models of PDC concepts.

Professional meetings, including the EduPar / EduHPC / EuroEduPar / EduHiPC workshops,[11] are great places to learn about new resources, present one's own ideas, and meet like-minded colleagues. It is difficult to underestimate the importance of building a professional network of colleagues with similar interests, and this author is grateful to the many people who have aided and encouraged his efforts over the years.

The author's journey as a PDC educator began with his participation in a parallel computing summer faculty development workshop. This author highly recommends such workshops, as they have the potential to introduce a person to excellent, career-changing developmental opportunities. One never knows what new doorway a workshop will open, nor where stepping through that doorway may lead!

Finally, the author hopes that this article has been beneficial to other PDC educators seeking to invest their time wisely as they strive to teach their students. If we let the past inform our work in the present, we may be grateful in the future.

CRediT authorship contribution statement

Joel C. Adams: Conceptualization, Data curation, Funding acquisition, Investigation, Methodology, Project administration, Software, Supervision, Validation, Visualization, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The author gratefully acknowledges the support of the National Science Foundation, specifically DUE-IUSE #1822486, MRI#1726260, DUE#1225739, MRI#0722819, and MRI#0079739; and the Calvin University Science Division.

References

[1] M. Abernethy, O. Sinnen, J. Adams, G. De Ruvo, N. Giacaman, ParallelAR: an augmented reality app and instructional approach for learning parallel programming scheduling concepts, in: 2018 IEEE International Parallel and Distributed Processing Symposium EduPar Workshop (IPDPSW), Vancouver, BC, Canada, May 2018, pp. 324–331.
[2] ACM/IEEE-CS Joint Task Force on Computing Curricula, Computer Science Curricula 2013, ACM Press and IEEE Computer Society Press, 2013.
[3] J. Adams, Patternlets: a teaching tool for introducing students to parallel design patterns, J. Parallel Distrib. Comput. 105 (July 2017) 31–41.
[4] J. Adams, T. Brom, Microwulf: a Beowulf cluster for every desk, in: 39th SIGCSE Technical Symposium on Computer Science Education, March 2008, pp. 121–125.
[5] J. Adams, D. Vos, Small college supercomputing: building a Beowulf cluster at a comprehensive college, in: 33rd SIGCSE Technical Symposium on Computer Science Education, Covington, KY, February 2002, pp. 411–415.
[6] J. Adams, W.D. Laverell, M. Ryken, MBH'99: a Beowulf cluster capstone project, in: 14th Annual Midwest Computer Conference, Whitewater, WI, March 2000.
[7] J. Adams, K. Hoogeboom, J. Walz, A cluster for CS education in the multicore era, in: 42nd SIGCSE Technical Symposium on Computer Science Education, March 2011, pp. 27–31.
[8] J. Adams, et al., TSGL: a tool for visualizing multithreaded behavior, J. Parallel Distrib. Comput. 118 (P1) (Aug 2018) 233–246.
[9] J. Adams, et al., Crayowulf: a multidisciplinary capstone project, in: 2020 American Society for Engineering Education (Virtual) Annual Conference. Online, https://www.jee.org/34342. (Accessed 30 August 2021).
[10] G. Amdahl, Storage and I/O parameters and systems potential, in: Proc. of the IEEE International Computer Group Conference (Memories, Terminals, and Peripherals), June 1970, pp. 371–372.
[11] Apache Software Foundation, Apache Hadoop, Online, https://hadoop.apache.org. (Accessed 30 January 2021).
[12] Apache Software Foundation, Apache Spark: lightning fast unified analytics engine, Online, https://spark.apache.org. (Accessed 30 January 2021).
[13] D.J. Becker, J. Salmon, D.F. Sevarese, T. Sterling, How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters, MIT Press, 1999.
[14] T. Braunl, Parallaxis-III: a structured data-parallel programming language, in: Algorithms and Architectures for Parallel Processing (ICA3P-95), May 1995, pp. 43–52.
[15] R. Brown, E. Shoop, CSinParallel: parallel computing in the computer science curriculum, Online, https://csinparallel.org/. (Accessed 30 January 2021).
[16] R. Brown, et al., Strategies for Preparing Computer Science Students for the Multicore World, in: Proceedings of the 2010 ITiCSE Working Group Reports, Ankara, Turkey, pp. 97–115.
[17] D. Conte, P. de Souza, G. Martins, S. Bruschi, Teaching parallel programming for beginners in computer science, in: 2020 IEEE Frontiers in Education Conference (FIE), 2020, pp. 1–9.
[18] J. Hill, M. Warren, P. Goda, I'm not going to pay a lot for this supercomputer!, Linux J. (January 1998).
[19] T. Mattson, B. Sanders, B. Massengill, Patterns for Parallel Programming, Addison-Wesley, 2004.
[20] MPICH: High Performance Portable MPI. Online, https://www.mpich.org. (Accessed 30 January 2021).
[21] NSF/IEEE TCPP Curriculum Initiative, Early adopter program, Online, http://tcpp.cs.gsu.edu/curriculum/?q=the-early-adopter-program.html. (Accessed 30 January 2021).
[22] OpenMP: Enabling HPC Since 1997. Online, https://www.openmp.org. (Accessed 30 January 2021).
[23] P. Pacheco, Parallel Programming with MPI, Morgan-Kaufmann, 1996.
[24] P. Pacheco, An Introduction to Parallel Programming, Morgan-Kaufmann, 2011.
[25] C. Peck, et al., LittleFe: parallel and cluster computing on the move, Online, http://littlefe.net. (Accessed 9 February 2020).
[26] S. Prasad, et al., NSF/IEEE-TCPP curriculum initiative on parallel and distributed computing - core topics for undergraduates, Online, http://tcpp.cs.gsu.edu/curriculum/?q=system/files/NSF-TCPP-curriculum-version1.pdf. (Accessed 30 January 2021).
[27] M. Quinn, Parallel Programming in C with MPI and OpenMP, McGraw-Hill, 2003.
[28] Shodor Education Foundation, National computational science institute, Online, http://computationalscience.org. (Accessed 30 January 2021).
[29] L. Vasconcelos, et al., Teaching parallel programming to freshmen in an undergraduate computer science program, in: 2019 IEEE Frontiers in Education Conference (FIE), 2019, pp. 1–8.

Joel Adams is professor of Computer Science at Calvin University, where he has been teaching his students about parallel and distributed computing since the late 1990s. He has been the principle architect of six Beowulf clusters including Microwulf, the first cluster to break the $100/GFLOP barrier. He is a PI on *CSinParallel.org*, an NSF-funded project to create and distribute high quality pedagogical materials for teaching students about parallel and distributed computing. His contributions to that project include the *parallel patternlets*, the *thread safe graphics library* (TSGL) and the *thread safe audio library* (TSAL). He is a two-time Fulbright scholar (Mauritius, 1998; Iceland, 2005) and is an ACM Distinguished Educator.

[8] https://csinparallel.org.
[9] https://tcpp.cs.gsu.edu/curriculum/?q=node/21183.
[10] https://tcpp.cs.gsu.edu/curriculum/?q=peachy.
[11] https://tcpp.cs.gsu.edu/curriculum/?q=node/21242.