An Automated Approach to Multidimensional Benchmarking on Large-Scale Systems

Undergraduate Petascale Education Program

Samuel Leeman-Munk
Earlham College
801 National Road West
Richmond, Indiana 47374
leemasa@earlham.edu

Aaron Weeden
Earlham College
801 National Road West
Richmond, Indiana 47374
amweeden06@earlham.edu

ABSTRACT

High performance computing raises the bar for benchmarking. Popular benchmarking applications such as Linpack[4] measure raw power of a computer in one dimension, but in the myriad architectures of high performance cluster computing an algorithm may show excellent performance on one cluster while on another cluster of the same benchmark it performs poorly.

For a year a group of student researchers at Earlham College in Richmond, Indiana worked through the Undergraduate Petascale Education Program (UPEP) on an improved, multidimensional benchmarking technique that would more precisely capture the appropriateness of a cluster resource to a given algorithm. We planned to measure cluster effectiveness according to the thirteen dwarfs of computing as published in Berkeley's parallel computing research paper [1]. To accomplish this we created PetaKit, a software stack for building and running programs on cluster computers.

Although not yet the benchmarking suite of thirteen programs that its creators set out to make, PetaKit has become a framework for benchmarking any command-line based program. In PetaKit's construction learned about the challenges of running programs on shared HPC systems, developing techniques to simplify moving software to a new cluster. In addition, we learned important time management skills, specifically by managing our time between classes and our PetaKit work. These skills and accomplishments have been of tremendous benefit to us in our post-baccalaureate careers, and we expect they will continue to be so.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—*computer science education*; D.2.8 [**Software Engineering**]: Metrics—*performance measures, benchmarking*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Copyright © JOCSE, a supported publication of the Shodor Education Foundation Inc.

General Terms

Parallel, Distributed Memory, Shared Memory, Benchmarks, Human Factors

Keywords

Blue Waters Undergraduate Petascale Internship, Parallel Computing, Parallel Benchmarking, Benchmarking

1. INTRODUCTION

In 2011, the University of Illinois will be home to what will likely be among the fastest publicly-accessible supercomputers in the world. This computer, Blue Waters, will reach sustained operation speeds measured in PetaFLOPS, 10^{15} floating point operations per second. Supercomputers like Bluewaters use a high degree of parallelism to achieve this level of performance, combining the power of huge numbers of computers to do what one computer could not. The catch is that such a high degree of parallelism creates an entirely new set of issues for software designers and requires an approach to programming that for many people is entirely new and foreign.

Despite its tremendous importance, this approach to programming is sorely undertaught in typical undergraduate curricula. Few students are taught to think in parallel, and even fewer are trained in the development of large-scale parallel applications.

The Blue Waters Undergraduate Petascale Education Program, led by Shodor, a non-profit computational science education foundation based in Durham, North Carolina, aims to teach young programmers the skills necessary to harness the power of High Performance Computing (HPC), and therefore Blue Waters and computers like it. For a year, an accepted student works full time during the summer and part time during the school year on a project relevant to HPC.

This paper describes our project and experiences as two of the six interns nationwide to participate in the first year of the program.

2. RELEVANT WORK

Since the late '80s the HPC community has been aware of shortcomings in the popular methods of benchmarking as applied to HPC. The Linpack benchmark, currently used to

rank supercomputers in the Top500 supercomputers list[5], has been repeatedly criticized for imprecisely representing cluster abilities with one flat metric[2][3][8]. Network bandwidth, latency, memory, number of cores on a chip, number of chips on a node, number of nodes on a blade, blades on a rack, all these factors have significant effect on program performance, and certain programs are affected more than others. Linpack, a linear algebra suite, is good for predicting how linear algebra-heavy programs will run on a cluster, but translates more poorly to other programs.

Most notable of the previous work done in multidimensional benchmarking, or judging a cluster with many different statistics rather than just one, is the High Performance Computing Challenge suite (HPCC). HPCC uses seven programs, selected so as to measure important factors of HPC machines such as steepness of memory hierarchy, network latency, and network bandwidth[8]. These make for detailed analyses for directly comparing cluster computers. A computational scientist with relatively little HPC experience looking at all these factors, however, would likely have trouble translating such statisics into an estimate of the performance of his or her own program. Application benchmarks, such as those suggested by the numerical aerodynamic simulation benchmarks project(NAS) and the performance evaluation for cost-effective transformations activity (the Perfect benchmarks), would use programs closer to real science applications to benchmark the cluster, generating benchmark data more relevant to real research[2][3].

Although common scientific programs have clear advantages as meaningful benchmarks, they suffer from issues of portability. Scientific software used in professional research tends to be large and demands careful tailoring to a specific system for maximum optimization. Devoting a large research task to porting scientific software generally costs more time and money than the resulting benchmark is worth [2]. NAS specifically mentions the absence of automatic tools to ease porting scientific applications. Today such tools, although far from trivializing the porting process, do make the prospect of application benchmarking much more affordable.

3. RESEARCH PLAN

Our goal was to build a framework that made application benchmarking feasible for evaluation of clusters' appropriateness for particular sets of scientific software. Admittedly, automated portability for arbitrary programs is not yet a reality, but we could make the benchmarking process much easier nonetheless. With our system we would be able to easily make a set of applications into a set of application benchmarks.What set of applications to use was not so simple. For the purpose of predicting performance on clusters, programs can be judged by their patterns of communication and ratios of communication to computation. A group of Berkeley researchers analyzed communication patterns among parallel programs and wrote a paper describing thirteen "dwarfs," or computing paradigms. According to Berkeley, any conceivable program may be described as a combination of one or more of these dwarfs[1]. Our system would take representatives of each of these thirteen dwarfs and collect performance data on various clusters. Then a user would identify his or her program using these paradigms and have the opportunity to make an educated decision of what cluster to use for his or her research.

Our system, PetaKit, would handle the entire process, from deployment and compilation on the remote host to visualization of the performance statistics in web-browser format. After decompressing the PetaKit file on the cluster of choice, the user proceeds into the generated folder and runs ./config to build all the necessary files with GNU Autotools, a system for managing compilation on various different systems. Then the user runs the performance data harvester perl script, stat.pl, which takes various parameters including style of parallelization, problem size[1], number of threads or processes to spawn and processors on which to spawn them. Stat.pl then runs these programs and sends various information, including number of threads, problem size, and wall time to a PostgreSQL database on a computer at Earlham College. A browser-based visualization tool converts user selections of the data into a graph.

To build the framework, we started with representatives of two of the thirteen dwarfs[1]. These are area-under-curve (AUC) – a definite integral approximation program based on the Reimann sum and the MapReduce dwarf, and GalaxSee – a galaxy simulation that serves as an n-body problem. AUC gives each thread (in some cases each thread is in its own process) a fraction of the area under the curve to estimate, they work and each sends its answer to the main thread, which sums the answers up and prints the total area under the curve. Computation increases linearly over problem size while communication remains constant. GalaxSee, on the other hand, splits the stars among the threads and each calculates the new positions based on the positions of all the other stars. Then the new positions are collected and redistributed and the next set of positions is calculated. Communication and computation both increase at a rate of $O(n^2)$ where n is the problem size. Each of these programs was split into four "styles" of parallelism: serial, shared memory (OpenMP), distributed memory (Message Passing Interface, MPI), and hybrid shared and distributed memory. The idea of having all four styles instead of just the most effective one was to confirm that hybrid was the most effective, and could outperform MPI under the right circumstances. In addition to resolving that issue, our expectation was that at the end of this project PetaKit's graphs would teach us something about the difference in scaling between MapReduce and n-body problems on various clusters.

4. CHALLENGES AND OPPORTUNITIES

Of all the challenges our group faced, most pervasive was the difference in the cluster environments that we came across when attempting to build and run our code. Each of about five clusters proved to be unique in its combination of compiler, network interconnect, linker, file system, scheduler, and software stack. Each error that occurred during building or running of our code had to be pinpointed as a problem with our code (either in its functionality or our fundamental

[1]Problem size is the variable in a program that has the most significant impact on the program's runtime. A Reimann sum algorithm, for example, uses the number of segments into which it splits the area under the curve as its problem size, and a galaxy simulation uses the number of stars in the simulation.

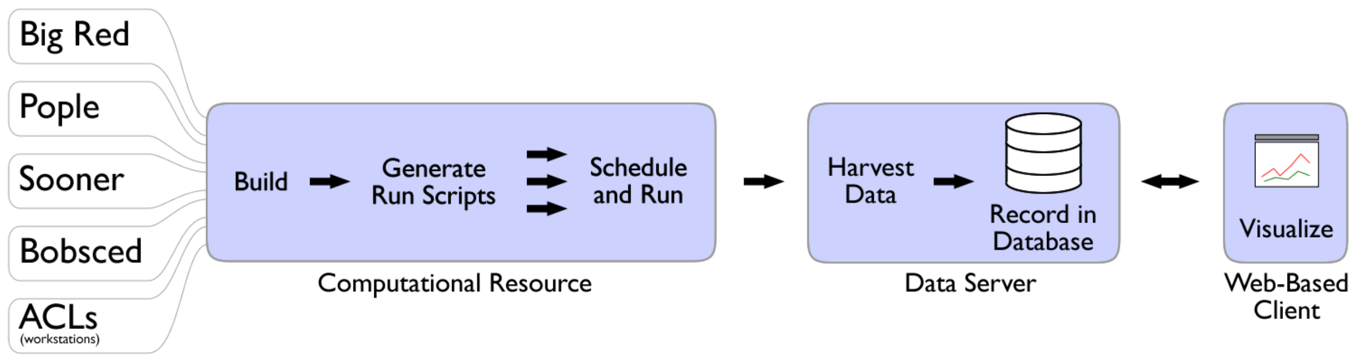

Figure 1: PetaKit's flow of data[6].

understanding of it), the compiler options we were attempting to use, a typo in the configure script, or a myriad of other possibilities. New, unrecognized errors forced us to perform a great deal of trial and error until we were able to find clues as to the sources of the errors. As we encountered more errors, however, we became more adept at recognizing common points of failure and remedying them. We were also able to make our package more robust by guarding against possible error scenarios. In this way, then, we learned a great lesson in working in new cluster environments, that there are many errors that can arise on the new system, and one must use the skills one has learned in the past but also have the courage to attempt new solutions.

A particular challenge in working on a shared cluster as opposed to a personal device is the scheduler. On a high-traffic cluster machine an automated scheduler, a program that distributes many tasks among many more processors, is the only way to share resources effectively. Because PetaKit is a benchmarking suite with an emphasis on the effect of increasing parallelism, it has to run the same program many dozens of times to collect all its data. In order to minimize the time each separate run of this program would have to wait before executing, PetaKit submits each run as its own job. In this manner, rather than demanding that the scheduler find one contiguous span of time for all the runs to complete, many small runs submitted to a scheduler could be run one by one on whatever processors become available. Incidentally this also takes advantage of the inherent parallelizability of benchmarking. Each individual instance of a program has no dependencies on or need to communicate with other instances, making it relatively easy to run them in any order or at the same time. The only caveat of running them in parallel is that the visualization program cannot rely on data coming in in any particular order.

Of course, like everything else, schedulers vary significantly by cluster. Designing PetaKit to work successfully with these various schedulers sometimes proved to be more than trivial syntax changes. For instance, The University of Oklahoma's supercomputer "Sooner" runs a Load Sharing Facility (LSF) scheduler, which, unlike PetaKit's other supported schedulers, Portable Batch System (PBS) and LoadLeveler, prints helpful information to standard output as it runs. The issue was that stat.pl collected its data from standard output, and LSF's messages to standard output were confusing it. We eventually bordered the meaningful output with a distinctive string on either side to distinguish it from the rest of the output.

In addition to variations in schedulers, we also had to work with a completely different staff team for every remote cluster we used. Nearly every time we tried to port our code to a new system we would have to find out little details such as the location or version of MPI libraries. First we would look to the cluster's relevant website for the information. When that didn't work, we'd get in touch with the cluster's system administrators or help staff. Generally, we received prompt, helpful replies to our questions, which greatly improved our productivity.

Besides technical challenges, an issue we faced on a deeper level was finding a way to balance the work of the project with our other course work. This was particularly a challenge because of the looser deadline structure associated with the project. Whereas classes were meeting two or three times weekly, our group's meeting to discuss the project only occurred once per week, and even then there were seldom hard deliverables. We had to develop a new level of focus and motivation for the project to be able to stay on top of it. Thus, we learned new ways of motivating ourselves, for example splitting the project into small pieces, working on small goals throughout the week rather than letting the greater end become daunting. Through constructing our own schedules and picking our own goals we were able to find personal meaning in the tasks we completed, and we made more efficient use of the time we could devote to the project.

Throughout the program our mentor Dr. Charlie Peck, associate professor of Computer Science at Earlham, was helpful and available to answer questions. During our weekly meetings we discussed our progress and where we'd be heading next. Dr. Peck would look at our data and point out places it did not make sense, such as when he helped us notice an issue in a graph that eventually led us to a set of old data that was being erroneously combined with recent data and leading to bad graphs.

5. RESULTS

At the writing of this paper, the PetaKit framework exists in its entirety. The system is not yet public, but the steps for a developer to benchmark a supported program on a given cluster are as follows[2] (see Figure 1):

1. Build (Computational Resource)
   (a) Generate a compressed folder from the latest source and send it to a supported cluster
   (b) Decompress the folder
   (c) run `./configure --with-mpi --with-omp` (Autotools) in the folder
2. Run StatKit (Computational Resource)
   (a) Run StatKit, specifying relevant options, such as an identifying tag (see Figure 5)
   (b) StatKit takes the specified parameters over which to observe scaling and iterates over them, building a script file for each combination of parameters
   (c) As it is built, each script is submitted to the scheduler once for each repetiton requested
3. Scripts run (Computational Resource)
   (a) Run program, collect output
   (b) Pipe output through secure shell to centralized data server
4. Data Collection (Data Server)
   (a) Parse meaningful performance data from raw output
   (b) Access PostgreSQL database and insert performance data
5. Data Display (Web-Based Client)
   (a) User selects data by tag
   (b) User selects the independent and dependent data to observe
   (c) User selects a variable over which to "split" the data into multiple lines
   (d) PHP backend accesses database, averages repeated runs, and creates a graph
   (e) Page displays graph image file

PetaKit developers can generate a compressed folder from the latest source and send it to a supported cluster, where Autotools is run to build the files. Then StatKit takes its parameters and generates a set of shell scripts, one for each combination. StatKit submits these scripts to the scheduler, resubmitting for each repetition, and when each is finished it pipes its output through ssh to the parser.pl script on a computer at Earlham College. Parser.pl parses the output into data which it sends to a PostgreSQL relational database.

[2]This is one possible flow of data chosen to show every piece of the project. By default the harvester saves its data to a text file that can be directly visualized with the accompanying plotting program PlotKit.

The data remains in the database where it can be accessed via our web browser application which allows the user to pick dependent and independent variables and compare the scaling of one setup to another(see Figure 1). The most common use of the graph is to compare OpenMP performance to MPI performance to hybrid performance by plotting the three threads vs. walltime.

This system has collected and visualized data on over five clusters:

- Indiana University's Big Red, a BladeCenter JS21 Teragrid resource with PPC 970 processors and a LoadLeveler scheduler
- Pittsburgh Supercomputing Center's Pople, an SGI Altix 4700 Teragrid resource with Itanium 2 Montvale 9130M processors accessed via a PBS scheduler
- Univerity of Oklahoma's Sooner, a PowerEdge 1950 Teragrid resource with Xeon E54xx processors and an LSF scheduler
- Earlham College's Bobsced, a decomissioned cluster with Intel Core2 Quad processors and a PBS scheduler
- Earlham's Advanced Computing Lab Computers (ACLs), a group of Dell commodity workstations with Pentium D processors and no scheduler

These visualizations have been presented at TeraGrid and SuperComputing in 2009, and at Earlham College in its 2009 research conference[7]. The most dramatic discovery of our benchmarking software was that the particular programs we were using to test it had serious bugs and fell short of the scaling one would expect of n-body and MapReduce problems, even at large problem sizes. AUC behaved as expected except for its hybrid version. Hybrid's failure to outperform MPI on BigRed was understandable for AUC, a Map-Reduce dwarf with low communication overhead. On Sooner, however, Hybrid AUC took over eight times as long to run as its MPI counterpart even with heavy parallelization and large problem size(see Figure 2). Even accounting for the additional overhead of a hybrid program, such a performance penalty suggests an error. As for GalaxSee, we were still having significant trouble running it and had not yet gotten results. Fortunately, we realized that the tools we had developed to get our results were actually much more important than the results themselves. A poster describing PetaKit as a performance data collecting framework was presented at SIGCSE and the Earlham College Annual Research Conference in 2010[6].

In the summer UPEP workshop at the National Center for Supercomputing Application (NCSA) we presented PetaKit to the new group of UPEP interns. We explained it to the students and guided them through a lab where they equipped their programs to output PetaKit-compatible statistics (using supplied header files) and collected data on it. We had modularized the harvester such that it could output data into a text file to be graphed via gnuplot, but StatKit was still a long way from being truly user friendly. Some students accomplished this with relative convenience, but many

had trouble using a PetaKit interface that was not yet designed for benchmarking arbitrary programs. By the next presentation at the University of Oklahoma (OU) the harvester gained improved versatility via a "template system" where the command line into which parameters are dropped was no longer hardcoded but defined by the user at runtime with the command `--cl`. Since then, scheduler submission scripts have also been templatized, allowing users to make a "script template" for a given system and instruct StatKit to use it with `-t`. Users can send their template scripts to us, and we will incorporate them, expanding PetaKit's cluster compatibility. For further user-friendliness we added a feature to PetaKit,`--proxy-output`, to collect minimal data from programs that have not had PetaKit output added to their code. To minimize visualization hassle we now include a supplementary program, PlotKit, another Perl script that automatically averages repeated runs, organizes the data, and creates and runs a gnuplot script to make a visualization according to a few simple parameters the user gives.

Using this improved PetaKit system it became simple for us to try many different programs without wasting our time reprogramming either them or PetaKit. With our new system we quickly tried a few different versions of GalaxSee until we found one that effectively used MPI to perform faster(see Figure 4). Now we can compare the results of AUC and GalaxSee on Sooner. While AUC shows a relatively smooth asymptotic curve, GalaxSee's heavy communication load makes it more sensitive to architecture changes. We see this in the graphs' bumps, the most prevalent being in the jump from eight to nine cores where GalaxSee added an extra node and started sending messages over a network.

6. IMPACTS

With the information we learned from our project, we will continue to improve PetaKit as both a software scaling evaluation tool and a cluster evaluation tool. In terms of the cluster evaluation our first step will be to debug GalaxSee and hybrid AUC. Then we'll add the other eleven dwarfs, making sure to check the efficacy of each before committing it. The UPEP workshop gave us the opportunity to show other people PetaKit, people who will likely be programming parallel algorithms that could very well find their way into PetaKit itself. Collecting pre-made programs will be a much more effective way of building PetaKit's dwarf array than building each one from scratch, so these contacts will be extremely useful.

Once we've shown the efficacy of hybrid, continuing to make four versions of each program is unweildy and unnecessary. Instead we will design a program to make the best use of the resources available, for example a hybrid MPI-OpenMP program that, when run on a dual-core laptop computer behaves as plain OpenMP. At the UPEP workshop we became acutely aware of the growing popularity of the Graphics Processing Unit for General Purpose Computation (GPGPU), so another version of each program may be introduced where it is appropriate to support GPGPU, or other accelerators as they become common on cluster systems.

Although Dr. Peck always made a point of addressing parallel processing in his computer science classes, we went into our UPEP internship with only the most basic understanding. Some of us were under the impression that we knew everything there was to know about the Unix command line, but writing Perl and shell scripts solving the complex problems PetaKit presented we learned to harness Unix's true capabilities, far beyond what in simple academics we had taken for granted.

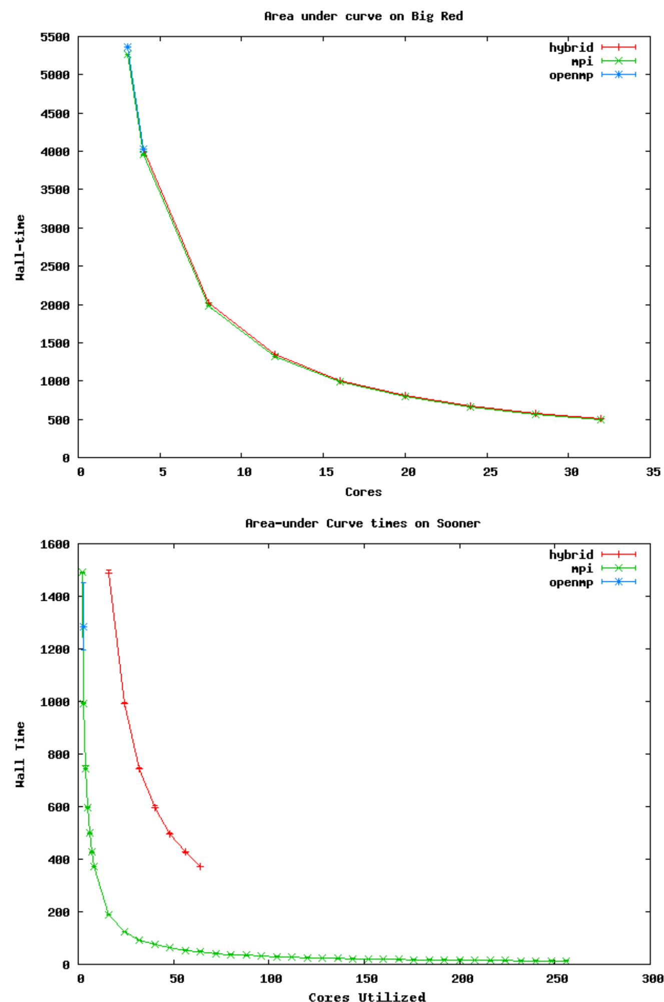

Figure 2: PetaKit visualization of AUC scaling on Sooner and Big Red[6].

The skills we learned in UPEP translated easily to much of our classwork. After building the statistics harvester for PetaKit, the test automation portion of software engineering class came as second nature to Sam. He ended up spending most of his time helping his classmates with their automation projects. The time management skills learned during the internship helped Aaron in his senior seminar and independent study. Deadlines were much more spread out than in normal classes, making a structured working schedule essential for success. Where other students struggled to schedule their time such that they were consistently doing work, Aaron felt comfortable staying focused and motivated under the looser schedule of the seminar. Both of us learned the power of source code control, something that seldom comes up in undergraduate computer science classes. Aaron made

```
perl stat.pl
-t al-salam.sh --scheduler pbs
--cl 'mpirun -np $processes
 ~/area-under-curve/area-mpi -s $problem_size '
--processes 2-4-16 --problem_size
 1000000000,100000000000 --proxy-output
```

Figure 3: An example StatKit command line

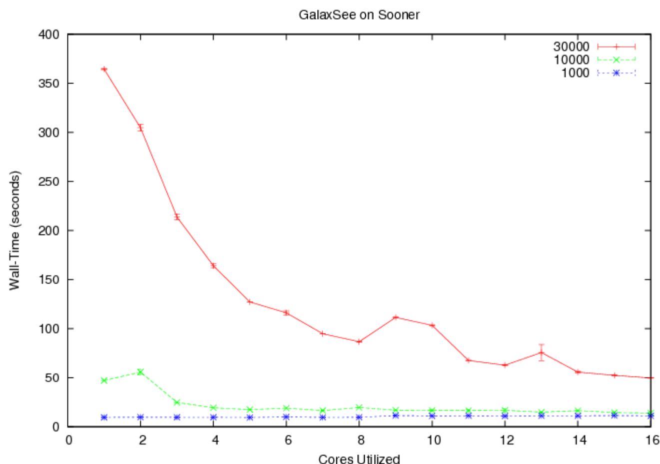

Figure 4: StatKit and PlotKit visualization of various problem sizes of an MPI GalaxSee program.

heavy use of Github in his senior year independent study and Sam was quick to suggest collaboration via Github to his group in robotics class.

Of course, besides everything else we learned, UPEP's goal was to teach us parallel thinking, and we have. Now the second question after "How would we solve this problem algorithmically" is always "How would we do it in parallel?" Far from being daunting and byzantine, parallel programming is just another tool in our repertoire. We're now interested in helping other people who find parallel programming intimidating to learn how much they really can do with it if they just give it a chance.

7. RECOMMENDATIONS

The UPEP program was a powerful experience. For many of us, it was unlike any before. Between talented mentors, open-ended project assignments, and access to a vast array of HPC professionals and educators, scholarly HPC literature, and of course full-scale shared cluster systems that we shared with genuine high performance scientific models, it taught us much more than a classroom environment could.

Certainly, UPEP's success, at least for us, was in significant portion due to Dr. Peck, our motivated, high-energy mentor who was not only happy to lecture us and other members of our small Earlham College Cluster Computing group in our weekly meetings, but gladly volunteered his time to give each and every one the individual help he or she needed. Even as we attempted to solve problems which no one had before attempted to tackle, Dr. Peck guided us with a calm, confident demeanor, not allowing a shred of doubt that success was assured.

For the most part, Dr. Peck took a non-interfering role in mentoring our internships. He would send us off with general tasks at first, then later in the internship he stepped back, remaining available for help, asking during meetings only whether we had enough work to keep us going. Given the opportunity to explore the project on our own terms, we developed a sense both for how to structure our work and what areas within HPC interested us most, whether they be analyzing and rewriting parallel code or constructing an effective, user-friendly interface for generating and sending out large sets of performance tests. As a result, we stayed motivated, working on what we found engaging at our own pace: challenging, but not exhausting. Our self-defined relationship with the project allowed us to connect with it more closely and to feel more accomplished with the results. Rather than some job we were doing for Dr. Peck, the projects became our own. Encouraging future UPEP interns to similarly take the lead in planning their own projects should help them to establish a more meaningful and personalized relationship with HPC.

As we carried out our personalized projects, we did so on actual high-end clusters that provided tools, libraries, and challenges for developing and running parallel code. Such access allowed us to put theory into practice and to experience the development of real parallel software on actual computers. Continuing to provide access to high end clusters through resources such as the TeraGrid and Blue Waters will help interns gain real hands-on experience with developing high end parallel code. Though much of working with clusters and parallel code appears daunting at first, the inherent challenges, more than simply justified by the corresponding educational benefit, are an integral part of the education themselves. To paraphrase HPC educator and system administrator Henry Neeman, "People should learn in education to do what they will do in real life." [1]

Continuing the theme of real-life connection, Dr. Peck provided us with scholarly work in the field, and encouraged us to communicate with experts who deal with high performance computing in their work, whether in industry or academica. Although practical experience working with clusters and parallel code clearly belongs at the center of UPEP, the encounters we had with professional-level HPC people and materials – the Berkely view paper [1], our work with Dr. Peck, and the people we met at conferences – served as important supplements to our education, teaching us where our work fits in the greater practice of HPC. Armed with our understanding of our context, we could improve the quality and depth of our research, for instance using Berkeley's thirteen dwarfs as guides for our benchmarks. Future interns will benefit from greater networking and connection to HPC researchers and workers.

Punctuating our internship with the occasional conference served a dual purpose: not only to allow us to see the work of others, but to present our own work as well. Each of the presentations we made at conferences and workshops helped us to share our project and receive meaningful feedback from experts in the field. Furthermore, the preparation for these

presentations forced us to reflect on our project and shape an informative and concise defense of it. These presentations were so effective at getting us to evaluate our standing that, as mentioned before, at one point in presentation preparation we actually changed our direction, going from a display of the data collected from poorly constructed programs to focusing heavily on the sophisticated system that we were developing for collecting that data. Regular presentations will encourage interns to feel comfortable synthesizing, explaining, and defending their projects, which will lead to improved comprehension and learning.

For all its strengths, the UPEP program's first year exhibited some disorganization characteristic of a nascent program with no preexisting documentation or buearacracy. None of the issues were particularly troublesome, but we were glad to find them addressed in the first official UPEP workshop at the National Center for Supercomputing Applications at the end of our internships (and the beginning of those of the next year's interns).

The UPEP workshop was not only the first time we met the next generation of UPEP interns, but the first time we met the rest of our generation of UPEP interns. Had we been in touch with each other from the beginning we could have communicated and helped each other with our problems in our projects.

Fortunately, the workshop resolved many of the human networking issues. The workshop was extremely informative, and working so close to Blue Waters itself made for an ideal beginning of the UPEP internships. Meeting so many people with the same interests, especially well-spoken presenters, was inspiring and motivating. In general, having the previous year's interns help teach the next year's is an excellent strategy. It's a cost-effective solution that solves the networking issue while strengthening the expertise of both parties.

8. CONCLUSIONS

Although the concept of a universal application benchmark made up of an instance each of Berkeley's thirteen dwarfs is still a long way from realization, the work done on the PetaKit project has by no means gone to waste. PetaKit's daughter project StatKit has come into its own and even generated its own companion project PlotKit. StatKit and PlotKit are currently being used in UPEP parallel education curricula.

Our work in UPEP was extremely rewarding, as was developing and presenting curricula for the UPEP workshop at the end of our internship. Sam has accepted a job at Shodor, where he continues work on PetaKit and other STEM and HPC education projects. He is looking into Carnegie Mellon and University of Southern California to get his PhD in Natural Language Processing, an often computationally intense field that will benefit from effective parallel experience. Aaron hopes to pursue a PhD in computer science and to have the opportunity to use the knowledge gained at UPEP to teach and mentor others. He is currently working as a system administrator in Earlham College's Computer Science department, putting to use the skills he has obtained from working on a variety of cluster implementations.

9. REFERENCES

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks. *SC Conference*, 0:158–165, 1991.

[3] G. Cybenko, L. Kipp, L. Pointer, and D. Kuck. Supercomputer performance evaluation and the perfect benchmarks. *SIGARCH Comput. Archit. News*, 18:254–266, June 1990.

[4] J. Dongarra, P. Luszczek, and A. Petitet. The linpack benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003.

[5] J. J. Dongarra, H. W. Meuer, E. Strohmaier, J. J. Dongarra, H. W. Meuer, and E. Strohmaier. Top500 supercomputer sites. Technical report, Supercomputer, 1997.

[6] M. Edlefsen, A. F. Gibbon, B. Johnson-Stahlhut, D. Joiner, S. Leeman-Munk, G. Schuerger, A. Weeden, and C. Peck. Collecting performance statistics on various cluster implementations. Poster at ACM Special Interest Group on Computer Science Education, April 2010.

[7] M. Edlefsen, A. F. Gibbon, B. Johnson-Stahlhut, D. Joiner, S. Leeman-Munk, A. Weeden, and C. Peck. Parallel performance over different paradigms. Poster at Teragrid Conference and Supercomputing Conference, 2009.

[8] P. Luszczek, J. J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. McCalpin, D. Bailey, and D. Takahashi. Introduction to the hpc challenge benchmark suite. 2005.

Notes

[1] H. Neeman (personal communication, August 8, 2010).