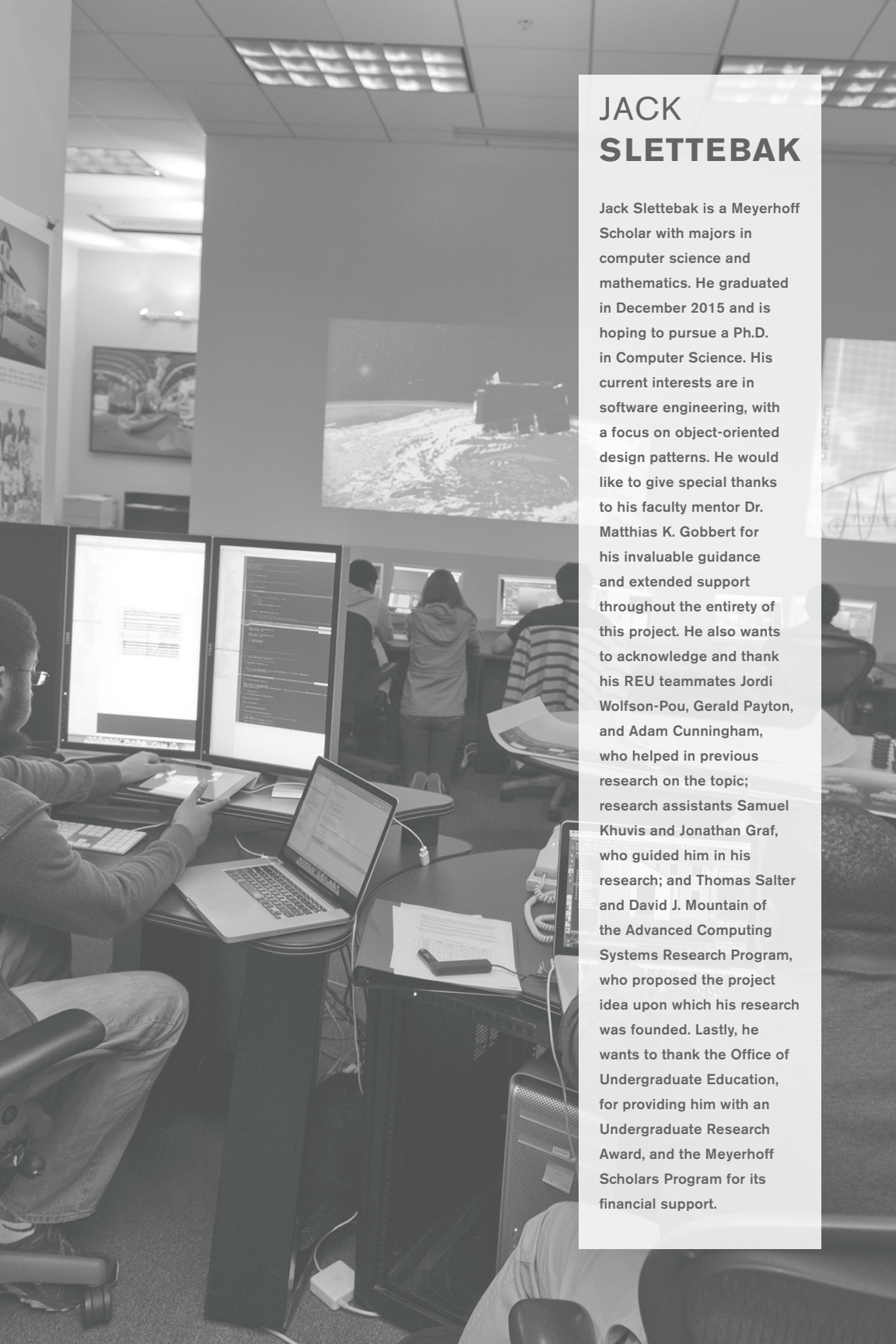

JACK **SLETTEBAK**

Jack Slettebak is a Meyerhoff Scholar with majors in computer science and mathematics. He graduated in December 2015 and is hoping to pursue a Ph.D. in Computer Science. His current interests are in software engineering, with a focus on object-oriented design patterns. **He would like to give special thanks to his faculty mentor Dr. Matthias K. Gobbert for his invaluable guidance and extended support throughout the entirety of this project. He also wants to acknowledge and thank his REU teammates Jordi Wolfson-Pou, Gerald Payton, and Adam Cunningham, who helped in previous research on the topic; research assistants Samuel Khuvis and Jonathan Graf, who guided him in his research; and Thomas Salter and David J. Mountain of the Advanced Computing Systems Research Program, who proposed the project idea upon which his research was founded. Lastly, he wants to thank the Office of Undergraduate Education, for providing him with an Undergraduate Research Award, and the Meyerhoff Scholars Program for its financial support.**

PUSHING THE LIMITS OF THE MAYA SUPERCOMPUTER WITH THE HPCG BENCHMARK

In summer 2014, I was fortunate enough to be selected to participate in a Research Experience for Undergraduates (REU) hosted here at UMBC. The REU focused primarily on high-performance computing and how parallel algorithms can be applied to various computational problems. All of the participants were assigned to teams and given an accelerated course on parallel algorithms. We were then exposed to several ongoing research projects that applied the techniques we learned. My group decided to pursue a topic under Dr. Thomas Salter of the Advanced Computing Systems Research Program. This involved performing a set of taxing software tests, called benchmarks, on maya, one of the two supercomputers at UMBC. By doing this, we gained valuable information on how the supercomputer functions. This knowledge can be applied to future research projects both within and outside the Department of Mathematics and Statistics. We decided to use the High Performance Conjugate Gradient (HPCG) benchmark because it is a well-known, respected benchmark that has its results published. The following paper provides the results of multiple experiments and explains what these results mean in the context of a parallel computing environment.

[LEFT] IRC, 2013. Photograph by UMBC Creative Services.

[RIGHT] Computer scientists programming punch cards, 1970s, University Archives, Special Collections, University of Maryland, Baltimore County (UMBC).

ABSTRACT

Parallel architectures and algorithms sit at the forefront of high-performance computing as ways to decrease the execution time of a computationally intense problem. Parallel architecture is hardware that has multiple cores or multiple threads, and a parallel algorithm is a software algorithm designed to be able to use multiple cores or threads simultaneously. The supercomputer maya in the UMBC High Performance Computing Facility (HPCF) is designed to provide a resource for the researchers of various disciplines who require a powerful parallel computer to solve the problems they encounter in their work. Using the newly developed High Performance Conjugate Gradient (HPCG) benchmark (www.hpcg-benchmark.org) from Sandia National Laboratories, this effort identified several runtime optimizations that allow for maximum performance on maya. These optimizations nearly doubled the reported performance of the benchmark from previous tests. A total throughput of 450 GFLOP/s was achieved using only a fourth of the hardware available on maya.

INTRODUCTION

The supercomputer maya is supported by the High Performance Computing Facility (HPCF) at the University of Maryland, Baltimore County (UMBC). It has more than 300 nodes equipped with powerful multi-core CPUs and several other cutting-edge technologies. Challenges include creating an environment that yields maximum performance and even knowing what level of performance should be expected. To address these challenges, a benchmark may be used to identify and measure an optimal runtime setup for use in other research applications. A benchmark is a portable program that runs a specified task on a system and returns a meaningful metric of the system's performance. A useful measure of performance for supercomputers is the number of FLoating-point OPerations per second (abbreviated as "FLOP/s"). Because supercomputers are capable of billions of

operations per second, our results are reported in GFLOP/s (spoken as "GigaFLOPs"). One GFLOP/s is equal to exactly 1,000,000,000 FLOP/s.

The High Performance Conjugate Gradient (HPCG) benchmark (www.hpcg-benchmark.org) from Sandia National Laboratories was used to test maya. This benchmark was recently developed to complement the more than 35-year-old High Performance LINPACK (HPL) benchmark. Both benchmarks solve large systems of linear equations. HPL solves a dense system of equations, while HPCG solves a sparse system of equations. Appropriately, the HPL benchmark uses a direct solver, while the HPCG benchmark uses a pre-conditioned iterative solver. HPCG reports computational throughput as a measure of performance. HPCG benchmark results have been reported for many of the top supercomputers in the world and are therefore appropriate for establishing comparisons to maya.

Past tests showed that increasing the number of threads led to an increase in computational throughput.[3] A greater increase in throughput was observed when the number of processes was increased. Total throughput decreased, however, when the sum of the number of processes on each node and the number of threads on each node exceeded 16. This was because although we were spawning more processes, we had no available hardware to execute them. In light of this observation, future trials kept the number of cores assigned to each job equal to the sum of the number of processes and the number of threads.

It was also found that an increase in throughput could be achieved by increasing the problem size.[3] As the problem size increased, so did the density of calculations, which resulted in higher throughput and less time spent fetching memory.

HARDWARE SPECIFICATION

This report focuses on the 72 newest nodes in maya, 69 of which are compute nodes. Each of these compute nodes has two eight-core 2.6-GHz Intel® E5-2650v2 Ivy Bridge CPUs. Figure 1 shows a schematic of one of the compute nodes. Each core has dedicated 32 kB of L1 cache and 256 kB of L2 cache; 20 MB of L3 cache is shared among the eight cores of each CPU. The 64 GB of each node's memory is formed by eight 8-GB DIMMs, and each CPU is connected to four DIMMs. Two QuickPath Interconnect (QPI) links connect the two CPUs of each node, and the nodes are connected to each other through a Quad data rate (QDR) InfiniBand interconnect.

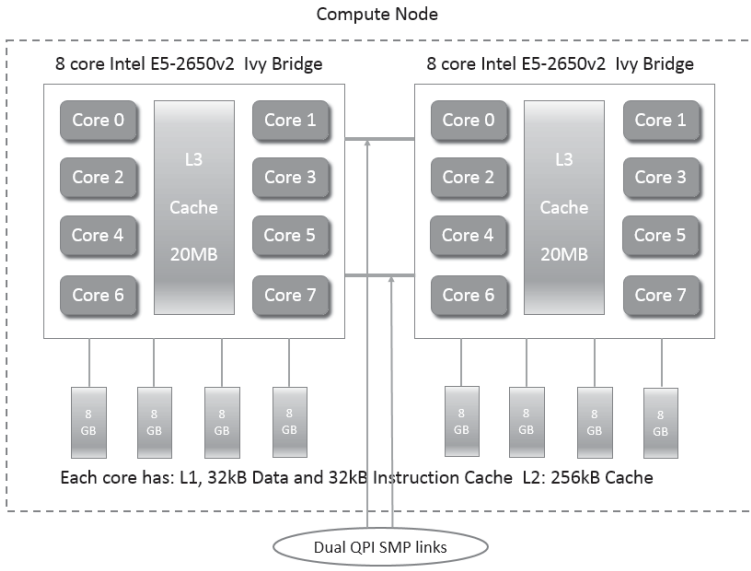

FIGURE 1. Illustration of compute node architecture with dual sockets, showing the physical cores within each socket, on-chip cache, and memory channels.

SOFTWARE

The HPCG benchmark is capable of integrating several different libraries and data structures. The options available thanks to this capability have a large impact on the execution of the benchmark and consequently its total computational throughput. In our tests, we made use of Message Passing Interface (MPI)[7] for process-level parallelism and OpenMP® for thread-level parallelism. We ran the benchmark with varying numbers of threads and processes.

In the context of this paper, a process is a task that requires computation time on a CPU and that is represented by a Process Control Block (PCB) and an independent memory space for associated data. The PCB holds all of the state variables of the process. The memory needed to maintain these features can reduce throughput by increasing the number of cache misses and the amount of input/output between the CPU and main memory. Also, because each process has its own memory space, each process must use MPI to communicate essential data.[7] While communicating with main memory or sending and receiving messages through MPI, a process is idling, which decreases total throughout.

Threads can be thought of as stripped-down processes. Like processes, threads require computation time from the CPU, but they lack several features of a process. Specifically, a thread has no PCB or independent memory space; instead, it shares the memory of its parent process with any other threads that may exist within it. Because of this, threads waste less memory than processes do, and an explicit interface is not needed to pass data between threads within a process. As a result, the computational efficiency of a thread is greater than that of a process.

The HPCG benchmark measures throughput in units of GFLOP/s (Giga FLoating-point OPerations per second). A floating-point operation is any arithmetic operation between stored numbers. An increase in GFLOP/s corresponds to an increase in performance.

The HPCG benchmark is a program that uses a conjugate gradient solver on a 3-D chimney domain. It was written in portable C++ code and was designed to be able to run on any number of processors. HPCG allows the user to specify sub-block sizes on each processor and then generates a 27-point finite difference matrix.[4]

The problem generated by this benchmark is essentially a single-degree stationary heat diffusion model with Dirichlet boundary values of zero. Its global domain dimensions are $N_x \times N_y \times N_z$ where $N_x = n_x p_x$, $N_y = n_y p_y$, and $N_z = n_z p_z$. The numbers n_x, n_y, and n_z are the sub-block dimensions in the x-, y-, and z-dimensions, respectively. The benchmark uses a total number of MPI processes p_N, which is factored into three dimensions $p_x \times p_y \times p_z$. Each MPI process is then assigned a sub-block.

In the setup phase, a sparse linear system is created by constructing a 27-point stencil at each grid point in the 3-D domain.[5] The three-dimensional stencil at an interior mesh point is sketched in Figure 2. The equation at a given point relies on the values at its specific location as well as the values at the other 26 points. The setup is weakly diagonally dominant for the interior points on the domain, while the setup for the boundary points is strongly diagonally dominant. The setup for this matrix implements the synthetic conservation principle for the interior points and illustrates the impact on the boundary equations of having Dirichlet boundary values of zero.

The properties of the linear system include all initial guesses that contain at least one value of 0, a matching right-hand-side vector, and a solution vector that equals 1. The system matrix is a non-singular, positive-definite matrix. It has 27 nonzero entries per equation for interior points and 7 to 18 nonzero entries per equation

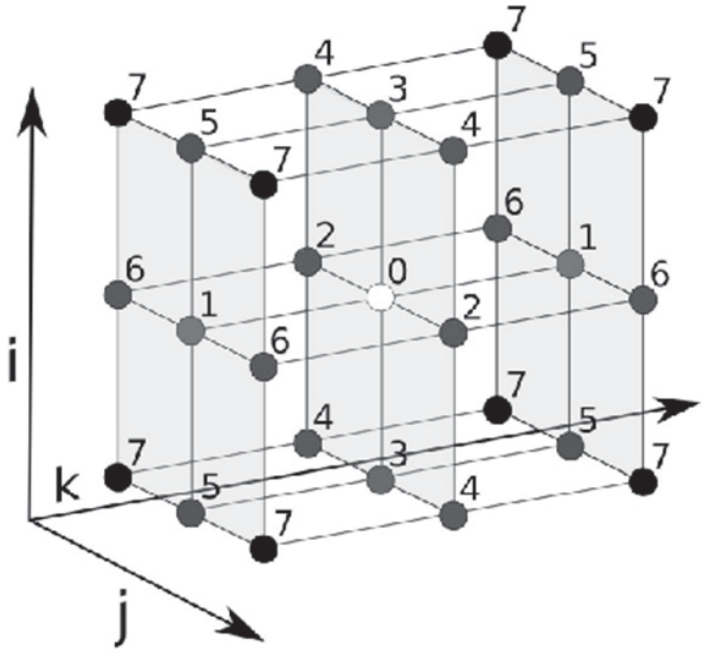

FIGURE 2. Diagram of the 27-point stencil created at each point in the mesh. Only the interior mesh points have the symmetry shown here.

for the boundary points. Each mesh point has internal symmetry, as shown in Figure 2. The values at the mesh points that neighbor an interior point are symmetric, as shown by the numbers in Figure 2. These nonzero values form 27 bands of nonzero entries in the system matrix of the discretized equation.

RESULTS INVOLVING DIFFERENT RUNTIME SETUPS

A previous technical report[3] includes several useful insights regarding the internal algorithm and problem setup in the HPCG benchmark. Most importantly, it was found that larger sub-block sizes ($n_x \times n_y \times n_z$) resulted in greater throughputs than smaller sub-block sizes. This is because each larger sub-block created more unknowns in its system of equations, so the ratio of computation to communication was higher for each sub-block. The sub-block size was set to have the maximum dimensions that would fit within the 64 GB of available memory. These dimensions were calculated to be $n_x \times n_y \times n_z = 160 \times 160 \times 160$.

Using these results[3] and results from a senior thesis[9] as a foundation, we sought to optimize the runtime environment at both the intra-node and inter-node levels. This involved trying different ways of assigning threads to cores and processes to CPUs at runtime. It also involved optimizing the environment variables within the operating system to achieve maximum throughput.

TABLE 1. Observed GFLOP/s for local sub-block dimensions $n_x \times n_y \times n_z = 160 \times 160 \times 160$ using combinations of alternating and socketfill process configurations and compact and one-core thread configurations. The variable p_N represents the number of MPI processes per node and n_t represents the number of threads per MPI process. The notation ET indicates that the case took excessive time and was not run to completion.

(a) Alternating Process, One-Core Thread with OFA MPI					
	$p_N = 1$	$p_N = 2$	$p_N = 4$	$p_N = 8$	$p_N = 16$
	$n_t = 16$	$n_t = 8$	$n_t = 4$	$n_t = 2$	$n_t = 1$
16 nodes	12.79	49.71	68.82	99.89	121.38
32 nodes	35.18	58.83	114.14	195.03	207.40
64 nodes	42.44	125.74	225.44	392.35	ET
(b) Alternating Process, Compact Thread with OFA MPI					
	$p_N = 1$	$p_N = 2$	$p_N = 4$	$p_N = 8$	$p_N = 16$
	$n_t = 16$	$n_t = 8$	$n_t = 4$	$n_t = 2$	$n_t = 1$
16 nodes	18.07	48.25	66.72	99.67	121.56
32 nodes	35.05	60.62	116.34	194.64	196.66
64 nodes	55.56	122.16	232.43	392.80	52.59
(c) Socketfill Process, One-Core Thread with OFA MPI					
	$p_N = 1$	$p_N = 2$	$p_N = 4$	$p_N = 8$	$p_N = 16$
	$n_t = 16$	$n_t = 8$	$n_t = 4$	$n_t = 2$	$n_t = 1$
16 nodes	16.24	35.55	48.04	60.82	121.97
32 nodes	39.91	58.63	93.81	121.68	224.62
64 nodes	65.51	116.57	186.22	242.40	51.28
(d) Socketfill Process, Compact Thread with OFA MPI					
	$p_N = 1$	$p_N = 2$	$p_N = 4$	$p_N = 8$	$p_N = 16$
	$n_t = 16$	$n_t = 8$	$n_t = 4$	$n_t = 2$	$n_t = 1$
16 nodes	16.22	35.47	47.96	61.91	120.53
32 nodes	40.09	58.67	92.61	122.36	180.42
64 nodes	65.98	116.16	186.33	243.51	47.14

EXPERIMENTAL DESIGN

Table 1 reports the intra-node results and holds constant the flags and environment variables that would affect MPI and other inter-node communications. Modifications were made to the job scheduler known as Slurm (Simple Linux Utility for Resource Management), which is used to allocate system resources within maya, and to OpenMP environment variables, which control the number of threads spawned during execution. These configurations allowed us to explore several different allocation schemes for processes and threads:

a. **Alternating Process, One-Core Thread:** alternates sockets when placing MPI processes and places all threads onto a single core.
b. **Alternating Process, Compact Thread:** alternates sockets when placing MPI processes and distributes threads to the closest vacant cores.
c. **Socketfill Process, One-Core Thread:** fills one socket before adding processes to the other socket and places all threads onto a single core.
d. **Socketfill Process, Compact Thread:** fills one socket before adding processes to the other socket and distributes threads to the closest vacant cores.

Because hyperthreading is disabled, we are forced to use only the physical cores within each socket. This means that each node on maya is capable of running up to 16 processes in parallel. Therefore, to maximize parallelization and avoid context switches, we fix the product of the number of threads (n_t) and the number of processes (p_N) assigned to each test to $(p_N)(n_t) = 16$.

DISCUSSION OF RESULTS

- **Effect of Increasing Threads/Processes:** GFLOP/s tend to increase with number of processes (left to right in Table 1). This result makes sense given how the HPCG benchmark sets up the test problem. Each process is assigned a sub-block so the global grid expands as processes are added. The result is a greater number of calculations. By contrast, threads are only used to distribute calculations within existing sub-blocks, and although this increases the speed at which calculations are done, it does not increase the volume of the domain.

- **Comparing Process Configurations:** When we compare sub-tables a and b to sub-tables c and d (alternating vs. socketfill), we see that throughput is much more strongly influenced by process assignment than by thread configuration. The alternating process distribution produces significantly better maximum throughputs than the socketfill algorithm.

  Comparing like rows in these sub-tables, we see a much more gradual increase in performance in the socketfill algorithm as p_N increases, although performance evens out when $p_N = 16$. This result may be connected to how L3 cache is shared on a socket. As the number of processes goes up, the amount of cache available to each process goes down. As a result, a socketfill scheme has much less cache than an alternating scheme does, so the benchmark experiences more cache misses and lower throughput. When the process count reaches the number of cores on a node, the allocation scheme is irrelevant because the sockets received the same number of cores, so the reported throughput remains constant.

- **Comparing Thread Configurations:** When we compare sub-tables a and c to sub-tables b and d (one-core vs. compact), we see that compact assignment can yield better results. Exceptions to this are found at higher process counts and lower thread counts, where the difference in thread configuration is less drastic. The improvement associated with compact assignment is more pronounced at a thread count of 16. This is because with compact assignment, each thread is guaranteed its own core on which to perform calculations. With a scheme that places all threads on a single core, however, there are more software threads but there is no increase in the amount of hardware used. As a result, threads are executed sequentially or the operating system performs several context switches among queued threads.

An alternating process scheme with compact thread assignment produces the highest throughput (392.80 GFLOP/s) and should therefore be used when running the benchmark.

RESULTS WITH TAG MATCHING INTERFACE

We also optimize the inter-node environment by modifying the communication environment used by MPI. In our intra-node tests, we set the network to use a shared memory fabric with OFED (Open Fabrics Enterprise Distribution) verbs for MPI. Using Tag Matching Interface (TMI) for MPI communications substantially improved scaling and performance on nodes. These results are recorded in Table 2 and can be directly compared to the results in Table 1b.

COMPARING MPI ENVIRONMENT CONFIGURATIONS

With 64 nodes, 16 processes per node, and one thread per process, the throughput with TMI was over eight times the throughput without TMI. This improvement dwarfs the differences between each intra-node setup. There were also noticeable improvements at lower numbers of nodes and processes.

MAXIMUM THROUGHPUT

After optimizing both the intra-node and inter-node levels, we can see that maya is capable of reaching nearly 450 GFLOP/s (with 64 nodes, 16 processes per node, and 1 thread per process) on 1,024 total cores. This result is 84% greater than the result of 240 GFLOP/s recorded in previous tests.[3] Our changes were made without optimizing the underlying algorithm. The performance of maya is very consistent with results reported in the June 2015 edition of the HPCG benchmark:[6] The supercomputer Bifrost at Linköping University in Sweden reports 4,500 GFLOP/s with 10,256 cores and slightly better hardware. It makes sense that Bifrost achieved approximately ten times the throughput of our tests because it used approximately ten times as many cores. Our results were obtained using less than a fourth of the total hardware on maya, indicating the possibility of achieving much higher throughputs. If an optimized benchmark were expanded to run on all of maya, we would expect a throughput far greater than 450 GFLOP/s.

TABLE 2. Observed GFLOP/s for local sub-block dimensions $n_x \times n_y \times n_z = 160 \times 160 \times 160$ using an alternating process configuration and a compact thread configuration. The variable p_N represents the number of MPI processes per node and n_t represents the number of threads per MPI process.

Alternating Process, Compact Thread with TMI MPI					
	$p_N = 1$	$p_N = 2$	$p_N = 4$	$p_N = 8$	$p_N = 16$
	$n_t = 16$	$n_t = 8$	$n_t = 4$	$n_t = 2$	$n_t = 1$
16 nodes	15.95	46.72	63.69	96.50	111.11
32 nodes	37.92	75.30	125.60	194.36	222.63
64 nodes	62.12	145.92	244.28	377.71	442.83

CONCLUSIONS AND REFLECTIONS

This work addresses an ongoing need for benchmarking on cutting-edge supercomputers such as maya. These benchmarks provide both a standard for comprehensive testing and a way to explore the computational limits of hardware. The main goal of this paper was to identify how to reach these limits with modifications that can be performed on any supercomputer. This paper also shed light on the internal workings of maya with the hope that the optimizations that were uncovered could be used in other areas of computational research that are performed on maya.

Moving forward there is more to be done. There are several reports that refer to certain strategies for optimizing software to run on the Intel Xeon Phi™, and these changes can be made to the sections of HPCG that are able to be modified.[1,2,8] These same functions can also be made to take advantage of GPU acceleration, which would allow a meaningful comparison of performance on the HPCG benchmark between the Intel Xeon Phi and NVIDIA GPUs. It would also be useful to rebuild an open-source version of the Intel Optimized High Performance Conjugate Gradient Benchmark to allow further revisions to existing code. With all of these elements, a truly comprehensive comparison could be drawn among all relevant computing architectures currently available on the maya system.

ACKNOWLEDGMENTS

This work was sponsored by an Undergraduate Research Award from the Office of Undergraduate Education. The author was also supported by the Meyerhoff Scholars Program through a contract with the National Security Agency (NSA). The work began during the REU Site: Interdisciplinary Program in High Performance Computing in the Department of Mathematics and Statistics at the University of Maryland, Baltimore County (UMBC) in summer 2014. This REU Site program was funded jointly by the National Science Foundation and the National Security Agency (NSF grant no. DMS-1156976) with additional support from UMBC, the Department of Mathematics and Statistics, the Center for Interdisciplinary Research and Consulting (CIRC), and the UMBC High Performance Computing Facility (HPCF). HPCF is supported by the U.S. National Science Foundation through the MRI program (grant nos. CNS-0821258 and CNS-1228778) and the SCREMS program (grant no. DMS-0821311) with additional substantial support from UMBC.

REFERENCES

1. Intel Corporation. Intel Optimized Technology Intel Optimized High Performance Conjugate Gradient Benchark, 2014. https://software.intel.com/en-us/articles/intel-optimized-technology-preview-for-high-performance-conjugate-gradient-benchmark, accessed on January 14, 2016.

2. Intel Corporation. Intel Optimized Technology Preview for High Performance Conjugate Gradient Benchmark, 2014. https://software.intel.com/sites/default/files/managed/1f/e8/HPCG_KB_Article-v23.pdf, accessed on January 14, 2016.

3. Adam Cunningham, Gerald Payton, Jack Slettebak, Jordi Wolfson-Pou, Jonathan Graf, Xuan Huang, Samuel Khuvis, Matthias K. Gobbert, Thomas Salter, and David J. Mountain. Pushing the Limits of the Maya Cluster. Technical Report HPCF–2014–14, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2014.

4. Jack Dongarra and Michael A. Heroux. Toward a new metric for ranking high performance computing systems. Technical Report SAND2013–4744, Sandia National Laboratories, June 2013. https://software.sandia.gov/hpcg/doc/HPCG-Benchmark.pdf, accessed on January 14, 2016.

5. Michael A. Heroux, Jack Dongarra, and Piotr Luszczek. HPCG technical specification. Technical Report SAND2013–8752, Sandia National Laboratories, October 2013. https: //software.sandia.gov/hpcg/doc/HPCG-Specification.pdf, accessed on January 14, 2016.

6. HPCG. June 2015 hpcg results, 2015. http://www.hpcg-benchmark.org/custom/index.html?lid=155&slid=279, accessed on January 14, 2016.
7. Peter S. Pacheco. Parallel Programming with MPI. Morgan Kaufmann, 1997.

8. Jongsoo Park, Mikhail Smelyanskiy, Karthikeyan Vaidyanathan, Alexander Heinecke, Dhiraj Kalamkar, Xing Lui, Md. Mosotofa Ali Patwary, Yutong Lu, and Pradeep Dubey. Efficient shared-memory implementation of high-performance conjugate gradient benchmark and its application to unstructured matrices. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2014.

9. Jack Slettebak. *The HPCG Benchmark for Cluster Computing*. Senior Thesis, Department of Mathematics and Statistics, University of Maryland, Baltimore County, 2015.