The HPCG Benchmark for Cluster Computing

Jack Slettebak

Department of Mathematics and Statistics, University of Maryland, Baltimore County

Abstract

Parallel algorithms, algorithms that use multiple cores/threads, and architectures sit at the forefront of high performance computing as a means to decrease the execution time of a computationally intense problem. The maya computing cluster at the University of Maryland, Baltimore County (UMBC) High Performance Computing Facility (HPCF) is a machine designed to take advantage of these algorithms and provide a resource for the various researchers who require a powerful computer to solve the problems they encounter in their research. To ensure that the entire system runs at maximum performance, we plan to test each component of the cluster with the newly developed Sandia High Performance Conjugate Gradient (HPCG) benchmark. It is our hope that by using a consistent piece of software we can test components on the system incrementally and compare their performances.

1 Introduction

The University of Maryland, Baltimore County (UMBC) High Performance Computing Facility (HPCF) offers the 240-node maya cluster, which comes equipped with several cutting-edge technologies. This includes the Intel Xeon Phi 5110P Coprocessor and the high-end NVIDIA K20 graphics processing unit (GPU) for heterogenous computing. Unfortunately, while all of these technologies can be leveraged for parallel computing, they all have different architectures and associated programming models. There is therefore a need to benchmark each of these hardwares to better explore their potential.[1]

To test the maya cluster, I used the Sandia High Performance Conjugate Gradient (HPCG) benchmark (www.hpcg-benchmark.org), which was recently developed to complement the now thirty-five-year old High Performance LINPACK (HPL) benchmark. Both solve a large system of linear equations, however for HPL benchmark the system is dense while in HPCG the system is sparse. Accordingly and appropriately, the HPL benchmark uses a direct solver, while the HPCG benchmark uses a pre-conditioned iterative solver. The benchmark itself has been used by many of the top supercomputing clusters in the world, and is convenient in that it offers a lot flexibiltiy in its implementation. As a result of this flexibility, it provides a solid framework to run and optimize a meaningful computational test on each device. These results can then be reviewed and compared allowing for a better understanding of the corresponding architectures.

In past tests, it was found that an increase in total MPI processes was followed closely by an increase in computational throughput [2]. This trend was also true for threads, although the throughput didn't scale quite as significantly. However, when the product of our total processes and threads assigned to processes on each node exceeded 16, we found that total throughput began to decrease. This was simply because, although we were spawning more software processes, we had no available hardware to execute them. In response to this

[1] A benchmark is a portable program that runs a specified task on a system and returns a meaningful metric of the system's performance.

observation, we made sure to keep a one-to-one correspondence between the total number of software processes and total cores assigned to a job.

It was also found that an increase in nodes and total problem size was linked to an increase in total performance. The former is self- explanatory, as you would expect more throughput with more hardware, but the latter was a result of the underlying algorithm. Namely, that as the problem size increases the density of calculations per node was increased greatly when compared to the total communications, and the result was a greater total number of calculations and a lower number of memory stalls.

The report begins in Sections 2 and 3 with a description of the hardware and software libraries that were used to run the benchmark, and we define some basic terms. Section 3 includes an explanation of the underlying algorithm of the HPCG benchmark for insight on the limiting factors and application of the algorithm. After describing the background and methodology, the results of our tests with the benchmark are presented in Section 4, and provide our analysis on why these results were observed and what they mean in the context of the maya cluster. To conclude, there is a short summary of the results and how they can be used in Section 5.

2 Hardware Specification

2.1 Homogeneous Computing

Figure 1 shows a schematic of one of the compute nodes that is made up of two eight-core 2.6 GHz Intel E5-2650v2 Ivy Bridge CPUs. Each core of each CPU has dedicated 32 kB of L1 and 256 kB of L2 cache. All eight cores of each CPU share 20 MB of L3 cache. The 64 GB of the node's memory is formed by eight 8 GB DIMMs, four of which are connected to each CPU. The two CPUs of a node are connected to each other by two QPI (quick path interconnect) links. The nodes in maya 2013 are connected by a quad-data rate (QDR) InfiniBand interconnect.

2.2 Heterogeneous Computing

Figure 2 shows a schematic of the Intel Phi 5110p architecture that is made up of 60 cores connected on a bidirectional ring bus. Each core has up to four threads with hyperthreading enabled, its own L2 cache, extended 512-bit vector registers, and runs at a frequency of 1.053 GHz. Each core is connected, through the ring bus, to a 6 GB cache of GDDR5 memory that it can fetch data from. The Phi also maintains a series of tag directories that log which core has which piece of data and can be queried for said information. This means that the cache on each of the Phi's 60 cores is fully coherent. A diagram of the Phi can be seen in Figure 2

The Phi also hosts its own instance of an operating system, a micro-kernel of Linux, which can be mounted with filesystems and executables to be run natively. When testing on the Phi, one core was always set aside for the use of the operating system, meaning that only 59 cores or 239 threads were actually used for computations during tests. The Intel Phi is capable of running in three seperate modes: offload, native, and symmetric/hybrid mode.

Offload mode resembles a standard heterogeneous setup, wherein the Intel Phi is used to speed up sections of code that are computationally dense. As can be seen in Figure 2, all information must be passed to the Phi through a PCI Express bus. The latency incurred by

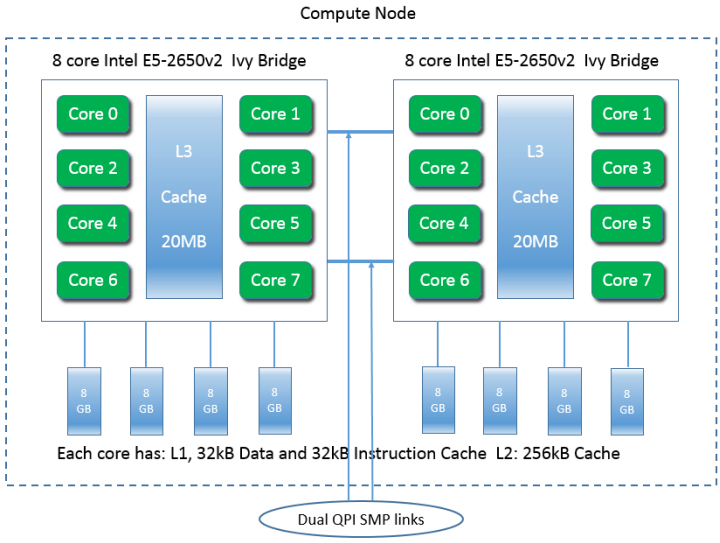

Figure 1: Illustration of node architecture with dual sockets shown as well as physical cores within each socket, on-chip cache, and respective memory channels.

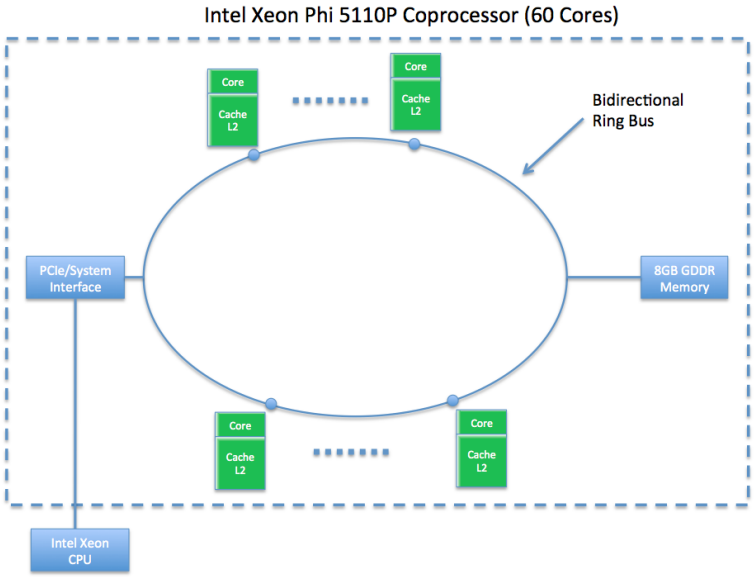

Figure 2: Illustration of Intel Phi 5110p co-processor with ring bus, attached cores, and PCI Express interface.

communicating through the PCI interface means that offloading mode should really only be utilized when there are enough calculations to justify this latency.

By contrast, **native mode** is much closer to a standard homogeneous computing setup. In **native mode**, the Phi runs an executable locally on its own operating system without accessing the host CPU. Theoretically, one can run a job across multiple Phis simultaneously by accessing the InfiniBand network interface directly. However, we were unable to get this to work and are currently working on fixing the problem.

Symmetric mode or **hybrid mode**, allows code to run on both the Phi and the host CPU simultaneously. This should theoretically provide the best performance, as it utilizes the throughput of both the Phi and the CPU. A technical report from Intel on the Intel Optimized HPCG benchmark affirms this conclusion [1]. However, again, it is worth noting that **symmetric mode** has not yet been shown to work on this system.

3 Software Specification

The HPCG benchmark is capable of integrating several different libraries and data structures into its compilation. All of these options can have a large impact on the execution and, consequently, the total computational throughput of the benchmark. In our tests, we made use of two particular features, specifically MPI (**M**essage **P**assing **I**nterface [5]) for process-level parallelism and OpenMP for thread-level parallelism. In addition, we also altered the total number of processes and threads that the benchmark ran its tests on.

In software, a process is any task that requires computation time on a CPU. It is represented in software by a PCB (**P**rocess **C**ontrol **B**lock), which holds all of a process' state variables, and an independent memory space for associated data. Memory overhead related to these features can reduce total throughput by increasing cache misses and I/O with main memory. Also, as a result of each process having its own independent memory space, each process must use MPI to explicitly communicate essential data [5]. During the process of sending or receiving messages, the assigned computational core is essentially idling, decreasing total throughput.

A thread can be thought of as a lightweight process in that it too is an entity requiring computation time from the CPU, but it is stripped of many of the features included in a process. Specifically, a thread has no PCB or independent memory space, but instead shares the memory of its parent process with any other threads that exists within it. This means that a thread does not have any of the memory overhead associated with the PCB, which means less wasted cache or memory. Also, it means that a thread exists on a single-instruction, multiple-data model, although each thread does have a virtual copy of a memory address to prevent race conditions during operations. Consequently, a thread does not need an explicit interface to pass data from one thread to another, which increase its computational efficiency.

Figure 3 shows a schematic that visualizes the difference between sequential and parallel processing. The top part of the figure shows a problem being executed by multiple threads. The bottom part indicates that in parallel processing the problem is first broken into several, four in the figure, sub-problems; of the these is then solved using multiple threads again.

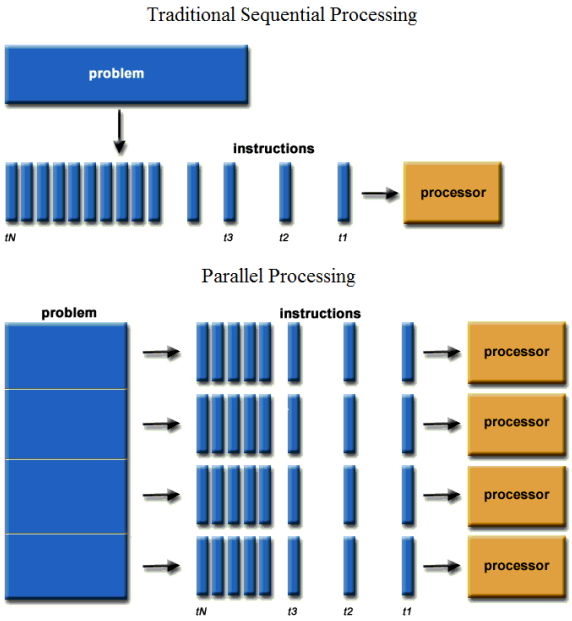

Figure 3: Diagram detailing sequential execution vs. parallel execution of a problem. The problem, visualized as a solid block, can be locally distributed amongst several different threads. In a parallel implementation, the problem can either be split up or, as is the case with HPCG, the problem size can be increased proportionally to the total number of computers/processes.

3.1 Sandia HPCG Benchmark

The HPCG benchmark measures performance in units of GFLOP/s (**G**iga **FL**oating-Point **OP**erations per **S**econd), a floating-point operation being any arithmetic operation between stored numbers. Thus, the total number of GFLOP/s can be thought of as the total computational throughput of the system, and an increase in total GFLOP/s is analagous to an increase in total efficiency and performance.

The HPCG benchmark is a conjugate gradient code for a 3D chimney domain meant to run on an arbitrary number of processors. This benchmark, which is written in portable C++ code, generates a 27-point finite difference matrix with sub-block sizes on each processor specified by the user. This publication from Sandia National Laboratories provides a more detailed description of the benchmark implementation [3].

The problem generated by this benchmark can be viewed as a stationary heat diffusion model of a single degree with zero Dirichlet boundary conditions, whose global domain dimensions are $N_x \times N_y \times N_z$ with $N_x = n_x p_x$, $N_y = n_y p_y$, $N_z = n_z p_z$, where $n_x \times n_y \times n_z$ are the local sub grid dimensions in the x-, y-, and z-dimensions, respectively, assigned to each MPI process with a total number of MPI processes P, which are factored into three dimensions as $p_x \times p_y \times p_z$. The division of the global $N_x \times N_y \times N_z$ mesh onto $P = p_x \times p_y \times p_z$ parallel MPI processes is sketched in Figure 4. The blowup of one process's subdomain sketches also the faces at the interface from this cube to its neighbors, whose contents need to be communicationed between the parallel processes.

In the setup phase, a sparse linear system is constructed using the 27-point stencil at

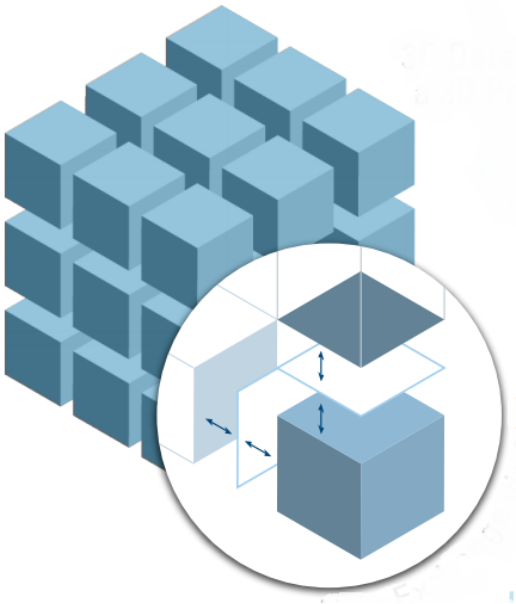

Figure 4: Three-dimensional cube of the $N_x \times N_y \times N_z$ mesh on $P = p_x \times p_y \times p_z$ parallel MPI processes. Each of the smaller cubes contains a subgrid of size $n_x \times n_y \times n_z$.

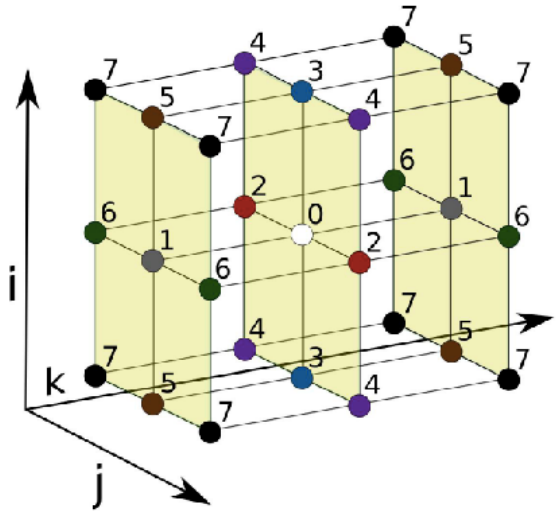

Figure 5: Diagram of the 27-point stencil that is created at each point in the mesh.

each of the grid points in the 3D domain [4]. The three-dimensional stencil at an interior mesh point is sketched in Figure 5. Therefore, the equation at a given point will rely on the values at its specific location as well as the surrounding 26 other points. For the interior points in the matrix, the setup is weakly diagonally dominant for the interior points on the domain, while the boundary points are setup to be strong diagonally dominant. The setup for this matrix implements the synthetic conservation principle for the interior points and displays the impact of no Dirichlet boundary values on the boundary equations.

The resulting properties of the generated linear system include all initial guesses with a

value of zero, a matching right-hand-side vector, and a solution vector that is equal to one. The system matrix is a symmetric positive definite, non-singular matrix with 27 nonzero entries per row for interior points and 18 to 7 entries for the boundary equations. More precisely, the values at each of the mesh points that are neighbors to an interior point on the three-dimensional mesh are symmetric, as indicated by the numbers at the neighboring mesh points shown in Figure 5. These non-zero values form then 27 bands of non-zero entries in the system matrix of the discretized equation.

3.2 Intel Optimized HPCG Benchmark

The Intel Optimized version of the HPCG benchmark has already altered all of the allowed kernels to achieve near optimum performance on the Intel Xeon CPU, achieving 95% of available throughput, and enhanced performance on the Intel Xeon Phi, nearly 67% of available throughput [6]. The benchmark is offered by Intel in the form of pre-compiled binaries, which hides any source code from the user. Despite this, a recent paper published on HPCG describes several of the methods used within the benchmark, which includes methods like block multi-color reordering [6].

4 Results

4.1 Previous Results

In a previous tech. report [2], several useful discoveries were made that guided the experimental design of this paper. Most importantly, it was found that larger submesh sizes ($n_x \times n_y \times n_z$) resulted in much higher throughputs when compared to smaller problem sizes. The increased number of unknowns in the system being solved allowed for a much larger number of computations and thus a higher total throughput. Thus, we maximized the local submesh size, being careful to make sure the resulting grid would fit within the bounds of available memory on both the Intel Phi and host node. Eventually the submesh size was chosen at $192 \times 192 \times 192$.

4.2 Tests in Homogeneous Environment using Generic HPCG

We performed our first tests with the HPCG benchmark version 2.6, without making any changes to any of its computational kernels or data organization. Our main focus was identifying the different compiler flags and process/thread combinations that would produce the highest throughput and the best scaling. The results of these tests can be seen in Table 1.

With these values held constant for our executable, we also tested four different runtime configurations through the `slurm` scheduler, which controls resource management and job submission to the cluster. These configurations involved different process and thread allocation schemes, which are listed here:

- **Alternating Process, One-Core Thread**: Alternates sockets when placing MPI processes, and places all threads onto a single core.
- **Alternating Process, Compact Thread**: Alternates sockets when placing MPI processes, and sets the `KMP_AFFINITY` environment variable to `compact`, which will distribute threads amongst any nearby vacant cores (will place all threads onto same core otherwise).

Table 1: **Sandia HPCG:** Observed GFLOP/s for local subgrid dimensions $n_x \times n_y \times n_z = 160 \times 160 \times 160$ using an alternating process assignment and compact thread assignment. Variable p_N denotes total number of MPI processes per node and n_t denotes total number of threads per MPI process.

Alternating Process, One-Core Thread					
	$p_N = 1$	$p_N = 2$	$p_N = 4$	$p_N = 8$	$p_N = 16$
	$n_t = 16$	$n_t = 8$	$n_t = 4$	$n_t = 20$	$n_t = 1$
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	N/A
Alternating Process, Compact Thread					
	$p_N = 1$	$p_N = 2$	$p_N = 4$	$p_N = 8$	$p_N = 16$
	$n_t = 16$	$n_t = 8$	$n_t = 4$	$n_t = 20$	$n_t = 1$
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	N/A
Socketfill Process, One-core Thread					
	$p_N = 1$	$p_N = 2$	$p_N = 4$	$p_N = 8$	$p_N = 16$
	$n_t = 16$	$n_t = 8$	$n_t = 4$	$n_t = 20$	$n_t = 1$
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	N/A
Socketfill Process, Compact Thread					
	$p_N = 1$	$p_N = 2$	$p_N = 4$	$p_N = 8$	$p_N = 16$
	$n_t = 16$	$n_t = 8$	$n_t = 4$	$n_t = 20$	$n_t = 1$
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	N/A

- **Socketfill Process, One-Core Thread**: Uses the `cpu_bind` option with `map_cpu` to fill a socket before adding processes to the other socket, and places all threads onto a single core.
- **Socketfill Process, Compact Thread**: Uses the `cpu_bind` option with `map_cpu` to fill a socket before adding processes to the other socket, and sets the `KMP_AFFINITY` environment variable to `compact`, which will distribute threads amongst any nearby vacant cores (will place all threads onto same core otherwise).

As a result of hyperthreading being disabled, we were restricted to using only the physical cores within each socket. This means that each node on the cluster is capable of running up to 16 processes in parallel. Therefore, to maximize parallelization and avoid any unintended context-switching, we enforced a strict one-to-one ratio between the total number of threads and processes assigned to a test and the total number of cores assigned to the test. The results of all of these tests can be seen in Table 1.

4.2.1 Increasing Threads/Processes

Moving from left to right along a row in any of the subtables in Table 1, we can see that more processes generally corresponds to greater total throughput. Of course, this result is relatively unsurprising when one considers how the HPCG benchmark sets up the test problem. Each process is assigned a subgrid so, as more processes are added, the volume of the global grid expands accordingly. The result is a greater number of total calculations, each of which is distributed. By contrast, threads are only used to distribute calculations within an existing subgrid, and although this increases the speed at which calculations are done, it does nothing to increase the volume of the domain being operated upon, and thus we see a much smaller increase in performance.

4.2.2 Comparing Process Configurations

Comparing different process configurations across each of the subtables (**alternating** vs. **socketfill**), we can easily see total throughput is much more strongly influenced by process assignment. In our case, the alternating process distribution produced significantly better results than the socketfill algorithm.

Looking at the same row from both a socketfill table and an alternating table, we can see a much steadier degradation of performance in the socketfill algorithm as p_N increases, although performance evens out when $p_N = 16$. This result is most likely connected to how L3 cache is shared on a socket. As the number of processes goes up, the total proportion of cache available to each process goes down. As a result, in the socketfill scheme, the total proportion of cache is much less than in an alternating scheme, causing a higher amount of cache misses and a lower total throughput. When process count reaches the total number of cores on a node, the allocation scheme is irrelevant as both sockets received an identical number of cores, and the reported throughput remains constant.

4.2.3 Comparing Thread Configurations

Comparing different thread configurations across different subtables (**compact** vs. **one-core**), we can see that compact assignment yields generally better results, although there are a few outliers, especially at higher process and lower thread counts where the difference in

configurations is less drastic. For higher thread counts however, the improvement of compact assignment becomes more pronounced.

The reason for this is obvious when one considers how both configurations run on hardware. With compact assignment, each thread is guaranteed an exclusive core to perform calculations on. However, with a schema that places all threads on a singular core, there might be an increase of software threads but no increase in utilization of hardware. As a result, each thread is executed in serial or, even worse, the operating system performs several context switches between queued threads.

4.2.4 Maximum Throughput

Looking at our results, we can see that the maya cluster is capable of running at nearly 400 GFLOP/s with $C = 1024$ total cores, split amongst 64 nodes with $p_N = 8$ processes per node and $n_t = 2$ threads per process. Our result is 61% greater than the result of 240 GFLOP/s we had received in previous tests, and the only change was a tweak in thread assignment during run-time [2]. It is important to remember that this is still running on an unoptimized version of the HPCG benchmark, and performing optimizations would also allow for large increases in performance.

Unfortunately, because of issues with the maya cluster, we found that running on 64 nodes with $p_N = 16$ processes per node caused a steep decline in total performance. As such, the data attained with this runtime setup is unreliable and cannot be used in our study. This is indicated by N/A in the tables.

4.2.5 Compact vs. Normal Thread Assignment

One of the runtime environment variables that can be set when running the benchmark is the `KMP_AFFINITY` variable, which controls how thread are mapped during runtime. In our tests we compare the `compact` setting with the `sparse` and default settings, and comparing the results that we got from each setting.

The `compact` setting ensures that all threads in a given thread block will be bound to cores that are close together. By contrast, the `scatter` setting will spread the threads out to cores as evenly as possible or, in other words, will separate the threads as much as possible. The final setting is the default slurm setting which will assign threads and processes in a sequential manner, that is process 1 goes to CPU 0, process 2 goes to CPU 1, etc.

4.3 Results with the Intel Phi Accelerator using Intel's Optimized HPCG

Results attained with the Intel Phi using the Intel Optimized benchmark, which includes a number of allowed optimizations to HPCG code, are shown in Table 2. Specifically, I tested the benchmark on two of the three possible Intel Phi modes (offloading and native mode). The results from these tests can be found in Table 2. Also, graphs that compare throughput, speedup, and efficiency between my homogeneous and heterogeneous runs are shown in Figure 6, Figure 7, and Figure 8, respectively.

Offload mode did not give good performance even with the Intel optimizations in place, performing much worse than Intel's expected results (link to HPCG article). Several optimizations can be made by modifying the runtime environment, such as altering the `KMP_AFFINITY` variable to be `compact`. This has the same effect as when this mode is set on the CPU, and will make threads as close together as possible.

Table 2: $(n_x, n_y, n_z) = (192, 192, 192)$ local domain dimensions. Where p_N = number of processes, n_t = number of threads, and ϕ = number of Phis.

CPUs vs. Phis w/ Offloading and Native mode					
	Generic $p_N = 1$ $n_t = 16$	Intel $p_N = 1$ $n_t = 16$	Intel $\phi = 1$ $p_N = 1$	Intel $\phi = 2$ $p_N = 2$	Intel Native ϕ
1 node	1.10	9.41	0.68	1.21	12.34
2 nodes	2.27	18.82	1.22	2.48	
4 nodes	4.34	36.02	2.43	4.69	
8 nodes	8.61	70.75	4.75	9.31	
16 nodes	17.00	135.21	9.38	18.49	

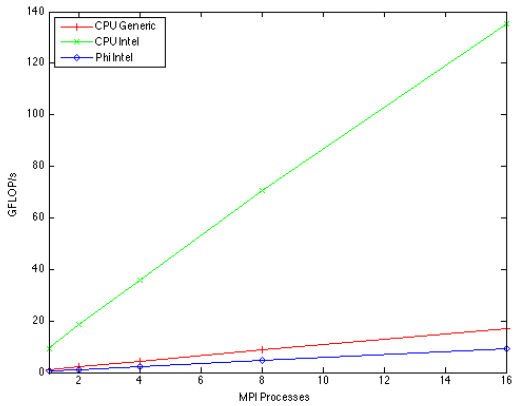

Figure 6: Reported GFLOP/s for Phi 5110P accelerator vs. Intel E5-2650v2 CPU.

However, despite the poor throughput, there was near optimal speedup and performance scaled very closely with the number of nodes running the program. This is very promising given the inherent communication overhead of running offloading on multiple nodes. We found in earlier work that the overhead of communicating through the PCI Express Bus to the host CPU, from host CPU to host CPU, and then from host CPU back to a device through the PCI Express bus limited the throughput severely because of how much time was devoted to communication (link tech report).

By contrast, native mode gave very good results compared with a homogeneous run, reporting approximately eleven times the throughput. This is not surprising as the Intel Phi has a much larger pool of threads to run computations on, and can therefore achieve a much higher degree of parallelism. Unfortunately, there is currently only results for a single Phi in native mode, but we can predict the results for the Intel Phi using information on efficiency we achieved with offloading mode. By using the efficiency as a coefficient for the throughput and multiply the throughput for one Phi in native mode by the efficiency and then the number of nodes, we can predict throughput for multiple Phis in native mode. Figure 9 shows what the predicted outcome would look like, outperforming all other setups.

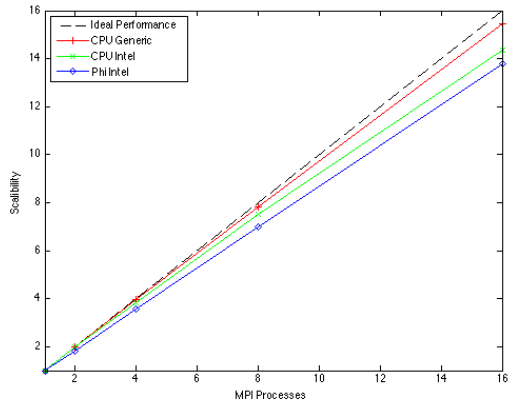

Figure 7: Scalability of Phi 5110P accelerator vs. Intel E5-2650v2 CPU.

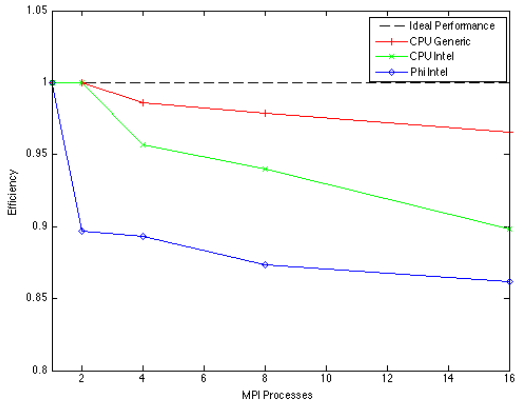

Figure 8: Efficiency of Phi 5110P accelerator vs. Intel E5-2650v2 CPU.

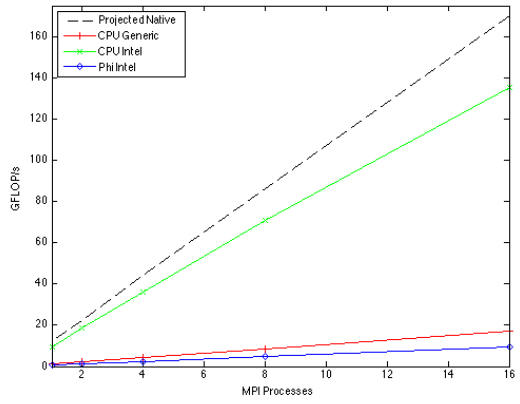

Figure 9: Reported GFLOP/s for Phi 5110P accelerator vs. Intel E5-2650v2 CPU with Predicted Native Phi.

5 Conclusions and Reflections

There is a clear and present need for benchmarking in a cutting-edge, homogeneous cluster, such as the maya cluster. Likewise, it is useful to compare and contrast the strengths of each computing component to gain insight into what workloads are best suited for a device's architecture. In these tests, we manage to get comprehensive results for Intel Xeon CPUs as well as for certain Phi modes.

In general the benchmark performs and scales well on CPUs, at least when compared to the Phi, and with Intel optimizations can perform at near optimal levels. This advantage can be further expanded on by setting up an optimal runtime evironment with appropriate mesh dimensions and thread mapping schemes. These same environment settings can also be extended to the tests on Intel Phis, but there is a clear need for further tuning of the benchmark to run optimally on the Intel Phi.

Moving forward there is a lot more work to be done. There are several papers that refer to certain strategies for optimizing software to run on the Intel Phi, and these changes can be made to the sections of HPCG that are allowed to be modified. These same functions can also be made to take advantage of GPU acceleration, which would allow for a meaningful comparison of performance on the HPCG benchmark between the Intel Phi and NVIDIA GPUs. It would also be useful to rebuild an open source version of the Intel Optimized benchmark to allow for further optimizations and revisions to existing code. With all of these elements a truly comprehensive comparison could be drawn between all relevant computing architectures currently available.

Acknowledgments

Special thanks to my mentor Dr. Matthias Gobbert of the Mathematics and Statistics Department at UMBC, as well as the TAs Samuel Khuvis, Jonathan Graf, and Xuan Huang who helped me in various capacities with my research. This work in [2] started during the REU Site: Interdisciplinary Program in High Performance Computing (www.umbc.edu/hpcreu) in the Department of Mathematics and Statistics at the University of Maryland, Baltimore County (UMBC) in Summer 2014. This program is 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. The author was supported by the Meyerhoff Scholarship Organization, through a contract with the National Security Agency (NSA). The author was also funded by the Office of Undergraduate Education, through the Undergraduate Research Award (URA) program.

References

[1] Intel Corporation. Intel optimized technology preview for 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 May 17th, 2015.

[2] 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.

[3] 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 15, 2015.

[4] 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 15, 2015.

[5] Peter S. Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann, 1997.

[6] 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.