Programming Heterogeneous Computers and Improving Inter-Node Communication across Xeon Phis

Chris Feilbach, Adam Sperling, Eftychios Sifakis, Mark D. Hill

Computer Sciences Department, University of Wisconsin – Madison
{crf, sperling, sifakis, markhill}@cs.wisc.edu

Abstract

Scientific computing workloads are well suited to parallel accelerators such as GPGPUs and the Intel Xeon Phi. While these accelerators can provide greater performance than traditional CPUs due to their parallel architectures and greater memory bandwidth, their maximum workload size is limited by relatively small memory capacity. To solve this problem, data can be split across multiple accelerators to utilize the combined memory capacity as well as increased compute capability. Combining multiple accelerators into heterogeneous systems introduces a new bottleneck. Communication bandwidth between accelerators over the PCIe interconnect is much slower than internal memory bandwidth.

This project examines the inter-node bandwidth bottleneck using the Intel Xeon Phi in the context of scientific applications. We show the limitations of traditional MPI programming paradigms, and leverage Intel's Xeon Phi-specific SCIF communication API to achieve increased inter-node memory bandwidth. While small messages still incur communication overhead penalties, messages larger than 512KB are able to saturate the PCIe bus and achieve bandwidth utilization close to 90% of the theoretical maximum. This project also attempts to address the complexities of programming systems of multiple accelerators. We introduce a software interface wrapper over SCIF that coalesces groups of small messages into larger ones. This new interface eases the programming experience and provides greater interconnect bandwidth from coalescing.

1. Introduction

Modern scientific computing applications, including many graphics applications, often contain complex but highly parallel mathematical kernels involving matrix and vector algebra. Tools such as numerical differential equation solvers, solvers for large linear systems, and graphical renderers for fluid simulations make use of these kernels [1]. The parallel nature of these workloads makes them good candidates for accelerators such as general purpose GPUs (GPGPUs) and Intel's Xeon Phi, with the latter as the focus of this project.

Scientific applications and data parallel applications place different requirements on computer hardware than traditional sequential programs. They are likely to be memory bandwidth bound with a high ratio of memory accesses to computation. As problems sizes grow, the need to share more data between accelerators also grows. These workloads are also iterative, with each calculation depending on data from a previous iteration, and often have large amounts of time between data reuse.

Workload demands in graphics and scientific computing are always growing, and while newer hardware is providing more computer power, memory capacity has failed to scale at the same rate. Since accelerators have limited amounts of memory compared to their host processors, programmers can combine multiple accelerators into heterogeneous systems in order to utilize their aggregate processing power and memory bandwidth. However, these heterogeneous systems introduce a new bottleneck not seen when using a single host: communication bandwidth between accelerators.

This project utilizes the Xeon Phi accelerator, which is based on Intel's Many Integrated Cores (MIC) architecture. Like a GPGPU, the Xeon Phi provides very high internal memory bandwidth with a highly parallel architecture. Unlike a GPGPU, the Xeon Phi is able to boot and run a version of Linux and can be programmed without accelerator-specific languages. Common parallel programming paradigms such as OpenMP and MPI are supported, which allows parallel host applications to run on the Xeon Phi with very little extra work. The Intel compiler can build MIC applications with only the addition of a single compiler flag [7].

While MPI can be straightforward to implement on the Xeon Phi, it does not saturate the bandwidth available between multiple accelerators. Intel provides a software API called the Symmetric Communication Interface (SCIF), which provides a lower level interface to the PCIe interconnect. SCIF can be used to set up remote memory windows that act as a shared memory region for all connected accelerators. Users can then directly read from or write to these shared regions. Our results show inter-node unidirectional bandwidth over

SCIF is higher for large message sizes than MPI alone. However, only a fraction of available bandwidth is utilized when smaller messages are communicated [8].

This project aims to improve bandwidth utilization for small messages in systems using multiple accelerators, where there is a large disparity between local DRAM bandwidth and interconnection network bandwidth. It also attempts to provide a uniform programming model and guidelines to minimize CPU run time for these workloads. We present a data structure that can group many smaller messages into fewer larger messages and retain interconnect bandwidth performance, giving the illusion that the accelerator has the ability to send small messages with high bandwidth utilization. Moreover, the underlying SCIF interface is invisible to the user, leading to more efficient programming and easier to understand code.

Future systems will continue to have this communication bottleneck unless it is addressed by systems architects. While heterogeneous systems enable larger datasets with much greater internal memory bandwidth for scientific workloads, they are still limited by interconnect bandwidth. New on-chip memory technologies such as High Bandwidth Memory (HBM) [9] and improvements on traditional DRAM buses such as GDDR5X [2] will increase local memory bandwidth. Interconnection network bandwidth is not scaling at the same rate, even in newer interconnects such as NVIDIA's NVLink [10].

In section 2 we present a detailed background on scientific workloads, the MIC architecture, and parallel programming models. In section 3 we outline the evolution of our own parallel programming methods used throughout the project. Section 4 provides a detailed description of our solution and software deliverables. Our experimental results are presented in section 5 and discussed in section 6. We provide an overview of related work in section 7 and present final conclusions in section 8.

2. Background

2.1 Workload Characteristics

Modern scientific applications present several workload characteristics which lend themselves well to both heterogeneous parallel computing. Workloads such as differential equation solvers, solvers for large linear systems, and fluid and smoke simulations all make use of complex but parallelizable mathematical kernels involving vector or matrix algebra. Since the work is highly parallel, these kernels are good candidates for GPGPUs or accelerators such as the Intel Xeon Phi. However, there are other characteristics that challenge the parallel model and can cause new bottlenecks in heterogeneous systems.

Workload size has been a limiting constraint when using accelerators. Efficient computation requires data to be resident in the accelerator's local memory, ideally with limited copies from host memory. This prevents the total workload size from exceeding the total memory capacity of the accelerator. Both modern, discrete GPGPUs and the Xeon Phi contain around 8GB of DRAM, with 240 GB/s DRAM bandwidth. Host CPUs can be connected to terabytes of DRAM, with 38 GB/s of theoretical DRAM bandwidth. Accelerators are often used to speed up parallel workloads due to their higher memory bandwidth, but a lack of memory capacity disallows larger and/or more accurate simulations (a priority over speed in scientific applications). In order to meet the size constraints of modern scientific workloads, multiple accelerators can be connected in a heterogeneous system to overcome the memory capacity limitation of a single accelerator.

Both intra- and inter-node bandwidth is needed to alleviate bottlenecks in communication. As grid sizes grow, the computation to communication ratio becomes larger. However, communication bandwidth still remains the primary bottleneck throughout the system.

The workloads we target have a few important properties. The workloads are largely iterative. Each calculation depends on data from the previous iteration, and therefore data must be synchronized across all compute units every iteration. However, the iterative nature of the workload allows for a large synchronization window. There can be millions of cycles in each iteration, enough to hide latency for inter-process communication. Communication between multiple accelerators requires more sophisticated methods to increase bandwidth (see section 3 for details.)

2.2 The MIC Architecture

The Xeon Phi accelerator, based on Intel's Many Integrated Core Architecture (MIC), provides hardware and software features useful for accelerating scientific workloads. A single Xeon Phi accelerator can provide up to 61 x86 processing cores with four threads each. While each core contains a relatively simple in-order, dual issue pipeline based on the Intel Pentium-Pro, it also contains a separate 512-bit vector pipeline. This allows the vector unit to execute 16 single-precision or eight double-precision operations per cycle. The accelerator is connected to the host and other accelerators via PCIe 2.0 x16 interconnect [11].

Unlike a GPGPU, the Xeon Phi runs a platform specific version of the Linux kernel which is user accessible via SSH connection from the host CPU. This allows the Xeon Phi to run programs directly as it would be done on a regular machine. In addition to the coprocessor only execution mode, programs can be run symmetrically on both the host and MIC with data shared between them. Hosts can also offload specific tasks to the MIC, just as GPGPU kernels require data to be offloaded from the host.

2.3 Common Parallel Programming Models

The Xeon Phi allows programmers to use standard parallel processing solutions such as OpenMP and MPI without any code modification. Therefore, it is simple to port existing parallel code to the Phi and see performance gains if the program is sufficiently parallel. Unlike a GPGPU, there is no specific language in which a kernel must be written. OpenMP allows for simple fork-join parallelism and makes loop-level parallelism relatively trivial. However, only the latest versions of OpenMP support task-based parallelism with coarse grain control. It is also impossible to run programs on multiple accelerators with OpenMP alone.

MPI enables multiple processes to run the same program, and also enables multiple accelerators to share data. Messages must be sent and received explicitly, which can add to code complexity. Users can also take advantage of offloading in order to move data to and from the host. In practice, utilizing OpenMP parallelism within a single accelerator and sharing data with other accelerators via MPI provides a straightforward programming paradigm for heterogeneous systems. Users are no longer bound by the memory limitations of a single accelerator. Rather, the bottleneck becomes communication bandwidth between accelerators.

2.4 Intel's Symmetric Communication Interface

Intel provides a proprietary communication API for the Xeon Phi called the Symmetric Communication Interface (SCIF) [12]. SCIF presents a socket-like interface that allows for communication between accelerators as well as communication with the host. SCIF sits directly on top of the PCIe interconnect, which can provide higher bandwidth for large messages.

In order to use SCIF, each accelerator and host (called *nodes*) must be bound to a specific port. One or many *endpoints* can be bound to specific nodes, and all operations are carried out on endpoints. Once endpoint connections are created, devices may communicate using two distinct paradigms. A simple point-to-point messaging layer is provided via the *scif_send* and *scif_recv* functions. These functions specify an endpoint, message buffer, message size, and flags. Like MPI, these functions can be either blocking or non-blocking. Despite relative simplicity and similarity with MPI, the messaging layer is only intended for messages smaller than 4KB; performance will degrade with larger messages. High performance bulk transfers larger than 4KB are achieved through a process called memory registration and subsequent remote memory access (RMA) operations. Memory registration allows nodes to expose regions of memory and mark them for sharing. Other nodes can then use the *scif_writeto* or *scif_readfrom* functions to access these shared regions directly. Using these functions, we can achieve near optimal bandwidth for large messages, as shown in Section 7.

3. Methods

3.1 Measuring Performance

The majority of our work revolves around trying to find the best means of making use of the limited amount of bandwidth provided by the PCIe interconnect. For evaluating the performance of our microbenchmarks and our streaming Laplacian kernel, we measure the number of bytes sent across the bus and measure the time it takes for the transfer to complete. We assume that the rate at which bits are pushed across the PCIe link is constant for the entire transmission, and then divide the message size by the time transmission took.

This rate allows us to compare different methods for data transfer against each other and allows us to compare it against the theoretical bandwidth of the PCIe interconnect. The closer to the theoretical max bandwidth, the better we consider that algorithm, because it is able to transfer more data in less time, holding data size constant.

3.2 Workloads

Three main workloads were used to investigate this problem and collect results.

Streaming Laplacian Kernel

A streaming Laplacian kernel is used for many of our initial investigations into how well bandwidth was utilized over the PCIe interconnect. Our kernel operates over a 2D or 3D grid and performs a Jacobi iteration. It allocates two grids in memory, X and Y, and initializes X with random data, with the borders of the grid being set to some constant. Each iteration, a Laplacian is taken over X and the results are stored into Y, and then the results from Y are scaled by a constant and added to X. This produces a smoothing effect over time, with the grid approaching the values at the borders.

Each node receives a partition of the X and Y grids and computes this subsection of the grid locally. Nodes then shared data at the end of each iteration. Each node stalls at a barrier until every node completes the current iteration.

Substantial optimization was done to exploit the high amount of thread level parallelism that the Xeon Phi provides. OpenMP was used to spawn threads to compute over a subgrid, and also to hint to the compiler to vectorize the compute kernel. Several run time constants were converted to compile time constants to allow the Intel C Compiler to produce more optimized binaries. Part of this effort was to ensure that the compute section of our kernel approached the architecture's performance ceilings. Our final result was able to utilize nearly 90% of local memory bandwidth both on the Host CPU and on the Xeon Intel Phi cards.

This workload evolved several times over the course of the project. An initial MPI version was used to get baseline bandwidth utilization numbers. It was then converted to SCIF, to explore if this library allowed for better use of PCIe interconnect bandwidth.

Microbenchmarks

With the kernel described above, it was difficult to decompose exactly how much time was spent in computation, communication, and synchronization. Ultimately we moved towards using microbenchmarks, with similar characteristics to our streaming Laplacian kernel. These microbenchmarks aimed to isolated communication patterns in scientific workloads. Random, streaming, and repeated access patterns were tested.

Communication Cache

Our final deliverable was an attempt to create a user-level software library that groups cache line sized messages into larger messages that then could be sent across the PCIe interconnect. With this, we were able to create L1 sized in-memory caches, and use these to create messages that were several tens of kilobytes.

4. Software Deliverables

For this project we developed several small to medium sized software programs and tools. The primary deliverable is a C++ streaming Laplacian kernel which can solve 2D or 3D grids, and partition and distribute grids across various nodes, where nodes can be a generic CPU or Intel Xeon Phi cards.

A substantial amount of work went into the development of the kernel, adopting conventions that made the kernel easier to debug and reason about, the verification of the kernel both manually and through software, and the optimization of both the compute and communication time of the kernel. The investment of time was twofold: we grew as C++ programmers and were able to improve our parallel programming skills and developed a piece of software that can be used outside of this particular research project. The class hierarchy implemented can easily be reused to support other scientific kernels.

Several microbenchmarks were also developed to measure various things were developed. A short list of these include: measuring DRAM bandwidth on the Intel Phi, measuring inter-node bandwidth for various sized messages using MPI and SCIF, and microbenchmarks implementing a Poisson solver with various levels of optimization. Besides providing the performance numbers that we desired, these microbenchmarks also served to teach us about compiler optimizations and how to write high performance C++ code. They also served as a means of checking the behavior of more complex code.

As part of our analysis, we made use of many Perl scripts for the analysis of our workloads and microbenchmarks. While these scripts are specific in nature, we believe that the industry inspired flow that we have used for analysis of our workloads may be useful for future research.

In terms of code for handling communication, three C++ classes were implemented: *MPI_COMMUNICATOR*, *SCIF_COMMUNICATOR*, and *COMMUNICATION_CACHE*. The first two were designed to allow for sending/receiving data stored in a 2D or 3D Cartesian grid. They contain functions that allow for ranges over the Cartesian grid to be sent and received, which encapsulates iterating over the grid from the compute kernel itself. These also support the use of blocking and non-blocking MPI/SCIF calls, and also perform some rudimentary routing to ensure messages end up where the kernel intends. In the case of SCIF, it contains functions to connect all available Phi and CPU nodes to each other, just as MPI does behind the scenes.

The *COMMUNICATION_CACHE* is the final deliverable, and allows for multiple processes to create a grid over a sparse virtual address space at a fixed address. Transmissions between processes can be done using virtual addresses which avoiding the several layers of translation that RMA based SCIF calls incur. Functionality is also provided to store data to be sent in an in-memory, software managed cache. Instead of sending individual cache lines across the PCIE bus, which results in poor bandwidth utilization, larger messages are sent across when a conflict miss occurs on insertion into the in memory cache. A cache way that has the highest occupancy is grouped into a single message, sent across the PCIe interconnect via an SCIF RMA, and then the data that caused the miss is inserted into the cache.

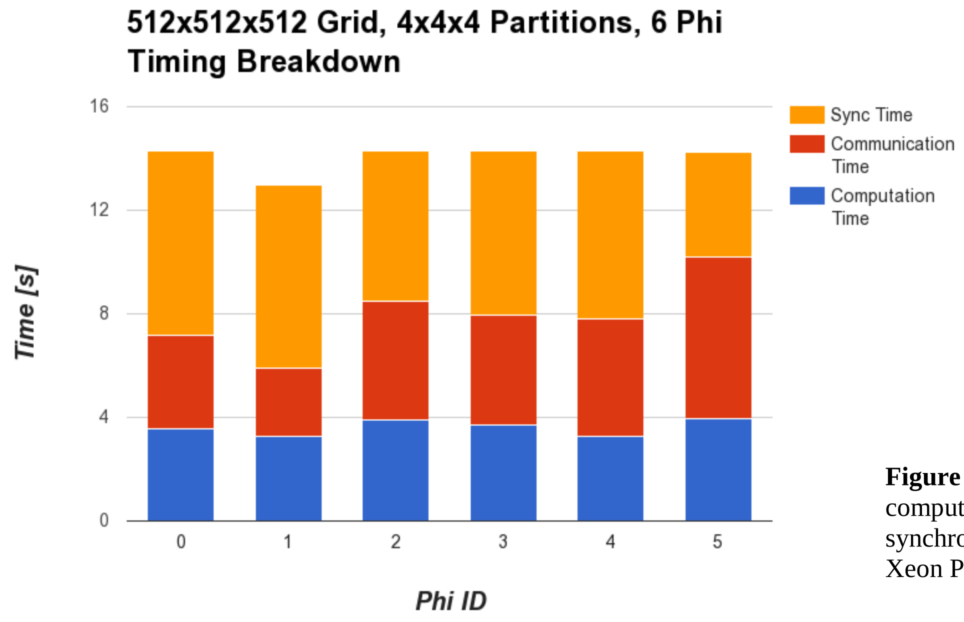

Figure 1: Time spent computing, communicating, and synchronizing across 6 Intel Xeon Phi accelerators.

5. Results

5.1 Experimental Setup

Our experiments were run on a high performance host connected to 6 Intel Xeon Phi 3150P (Knights Corner) accelerators running at 1.1 GHz, with 8 GB of local DRAM. The host node contains a dual-socket Intel Xeon E5-2650 v3 at 2.30 GHz paired with 256GB of DRAM. Each Xeon Phi contains and 8GB RAMDISK, with only about 6GB usable for non-system software and data.

5.2 Initial MPI Results

Figure 1 shows a timing breakdown of our MPI Laplacian streaming kernel across 6 Intel Phis. Timing numbers were generated through instrumentation of the compute, communication, and synchronization functions with a pair of RDTSC instructions. Results were then collected per iteration and aggregated to produce a final result of total time spent in each phase of each iteration. Blocking MPI calls were used, and a generic MPI barrier call was used to provide synchronization.

For this particular instance of our workload, the compute kernel was threaded using OpenMP, but was not vectorized, giving an upper bound on the total amount of time that computation could take. Even with this, communication and synchronization dominate the overall wall time of the application.

Our communication and synchronization results are much higher than our analytic models suggest they should be. Communication time for this grid size and number of iterations, assuming maximal use of available PCIe bandwidth, should complete within roughly 1.5 seconds. Synchronization time on the other hand consists of a single *MPI_Barrier* call, and should ideally take very little time. Load imbalance may contribute to the large amount of synchronization time.

Our experiments show that for this particular grid arrangement, MPI is able to achieve roughly 300-400 MB/s of unidirectional data transfer per second between the Phi’s. This comprises between 3 to 5% of the overall theoretical bandwidth of the PCIe interconnect.

5.3 SCIF Microbenchmark Results

Before transitioning our workload to SCIF, we collected bandwidth numbers using microbenchmarks. Results for both *scif_mmap()* and *scif_send()/scif_recv()* bandwidth are on the order of a few hundred MB/s. The use of RMAs can provide substantial bandwidth utilization for larger message sizes, as shown in Figure 2.

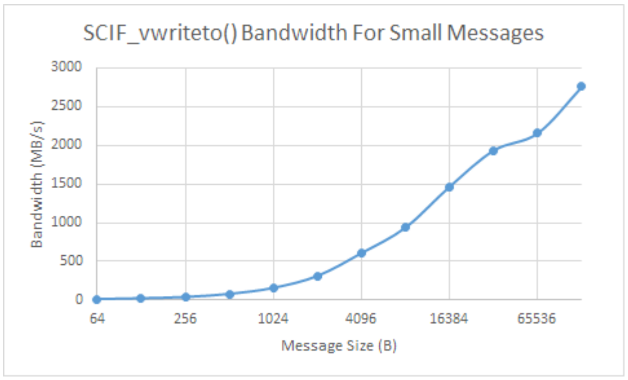

Figure 2: Bandwidth utilization for varying message sizes over an SCIF connection.

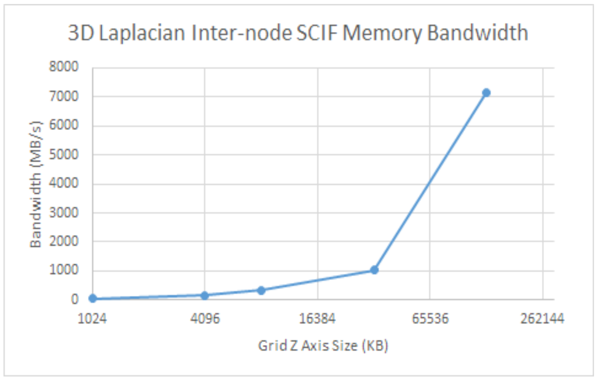

Figure 3: Average bandwidth utilization for various cube sizes across 6 Intel Xeon Phi accelerators

5.4 Transition to SCIF in Streaming Laplacian Kernel

Our transition to SCIF showed that it is possible to achieve substantially better bandwidth results over MPI. Figure 3 shows that for grids with very large minor axes, it is possible to achieve nearly 90% of the theoretical unidirectional bandwidth provided by PCIe. However, this is dependent on the requirement that data to be shared must be contiguous in memory. To actually achieve the bandwidth, the grid under simulation must look like a very narrow, lengthy, rectangular prism, instead of a cube, in order to fit within the limited DRAM of each Intel Phi.

5.5 Communication Cache

The communication cache is a software library that stores cache lines to be sent between two processes. Cache lines must be programmatically inserted into the cache via an *insert* function. We found that for varying sizes and associativity that messages on the order of 16KB-64KB could be generated. Messages are sent on conflict misses in the cache, and comprise of all the data in that cache way. We briefly investigated changing with cache placement algorithms (first free vs round robin) and found no difference in terms of average message size or bandwidth utilization.

Our initial bandwidth findings show that for caches with a large numbers of sets, average message size increases. For caches that send on average more than 16 KB per message, we see bandwidth ranging from roughly 500 - 3000 MB/s.

6. Discussion

6.1 Debugging Parallel Scientific Kernels

The design and debugging of parallel C++ code is challenging, especially when a message passing paradigm is used. Substantial amounts of time were spent debugging MPI related bugs. These bugs sometimes caused deadlock. Even when these bugs allowed for the program to complete execution we found often that the final result of the simulation was incorrect. Sometimes this was caused by messages modifying data in an improper subgrid, but having the exact same size as the message the node expected, which meant that the MPI library was unable to detect that this was incorrect. Other times, it was caused by data that should have been sent elsewhere.

To facilitate easier debugging, in particular, to ensure that the proper messages are sent to the proper nodes, we made use of C++ visibility modifiers, and wrapper classes, to create a model MPI system. A class called *COORDINATED_ENTITY* was created to represent a node in an MPI environment. This node makes use of private member variables to provide the X and Y grids, and the compute kernel was modified to only make use of these member variables. This, in effect, gives the illusion of disparate address spaces. Functions were created to allow for the entities to communication with each other. Communication was facilitated by a singleton class called *COORDINATOR*, which each *COORDINATED_ENTITY* subscribed to. The *COORDINATOR* provided an interface through which messages could be sent and received, in a manner similar to *MPI_Send* and *MPI_Recv* calls.

This model is not suitable to find all MPI bugs. It still leaves bugs that can be caused by variations in the global order in which messages are sent. However, it does allow for algorithmic bugs to be found, and also bugs in which the wrong message is sent to the wrong node, and ultimately, the implementation and use of this model saved us time.

6.2 Thoughts On Debugging Future Kernels

For MPI debugging, it was rare that the ability to correctly send and receive messages was the cause of a bug. While some bugs were focused on getting the ordering of blocking *MPI_Send* and *MPI_Recv* calls correct, most of the errors had to do with the contents of messages. On top of that, we often found that our contents sent were generally valid, that is, they were part of the data in a grid, but the location we believed them to be from, or where in the grid they were being written to, was incorrect.

For this, we propose that any future work have a wrapper class around the data object being sent for debugging purposes. This wrapper class would include the datum and metadata, such as the coordinates in the grid that it was from, and the coordinates in the grid to where it should be placed. Other data, such as who sent it and where it was to be sent, would also have been helpful. Instead of instantiating grids of floats, grids of these metadata objects would be instantiated and transmitted.

Lastly, we note that a more efficient method for debugging MPI programs needs to be developed. We made use of gdb, and inserted while loops spinning on a volatile int flag which we modified the value of with gdb, and then started running that process. This is how some MPI library authors debug the MPI library itself, and while it works, it is rather hacky and takes a large amount of time [3].

6.3 Memory Ordering Impacts Communication Time

With MPI and SCIF it is convenient and sometimes required that messages must be contiguous in memory. On the Intel Phi, we found that lexicographical ordering incurs substantial slowdown due to ordering of the memory. Within our Laplacian kernel, some sides of our grid were entirely contiguous in memory, others had contiguous regions of memory, but the worst case was when there was one float per cache line. This degenerate case added substantial latency to memory copies (roughly 16x), because the Phi fetches cache lines from DRAM, as opposed to individual floats.

Possible solutions for this were to extend the borders of each cube to be 16 thick instead of 1 thick (so each border element shared is a cache line, as opposed to a float), or to use an ordering layout that better correlates geometry distance with spatial locality.

6.4 Limitations of SCIF and the Intel Xeon Phi

SCIF provides a reasonable API to send messages with higher bandwidth utilization than MPI does, across the devices that support it. However, it has several limitations. Our claim for this was a code review of the kernel module encapsulating SCIF functionality and through the results collected from microbenchmarks.

The most glaring limitation that we encountered was the use of locking within the kernel, which severely limits the use of SCIF in parallel workloads. Even non-blocking SCIF calls are sequentially ordered across each Intel Phi by the kernel. This is primarily driven by the need to protect kernel data structures, and to protect the memory space for DMA descriptors, which are used to drive the DMA engine to perform memory copies.

Given how “wimpy” an individual Xeon Phi core is, the latency incurred for setting up a DMA transfer is likely substantial, which leads to SCIF communication being a very expensive call, especially for small message sizes. Previous work done on the SHRIMP multicomputer also cite communication call latency as a problem in their system [5].

Another problem that was briefly encountered was that SCIF methods for communication have non-uniform timing. For the *scif_writeto* and *scif_vwriteto* functions, if several small messages are sent in quick succession, that the latency grows from several thousand cycles to several hundred thousand cycles. The current hypothesis is that the remote Phi has not completed the DMA transfer from previous calls, and that the local Phi must wait on an acknowledgement to confirm it is safe to send new data. While it is well known that latency for messages is dependent on interconnection network load [19], this range of latency is huge and makes it hard for programmers to reason about how expensive a transfer is.

One issue we faced was the ability to try and modify the SCIF kernel code on the Intel Phi. We were unable to find a way to do this, despite several different attempts at it, which hindered our ability to investigate potential solutions to the bandwidth utilization problem we faced with small messages. Latency may be able to be brought down by optimizing out portions of SCIF that our workloads do not require. In particular, SCIF always ensured that it was safe to send data, and that the card receiving it was operational. While safety matters for correctness of our workload, if we allowed SCIF to fail sometimes as a tradeoff for faster latency, we could envision higher bandwidth utilization most of the time as opposed to slow bandwidth utilization all the time.

6.5 Address Translation and Virtual Addresses

While we do not propose a solution to better use PCIe interconnect bandwidth, we briefly investigated whether physical to physical address translation could be used to make communication between Intel Phi cards easier.

Our literature search suggests that address translation was an important part of NUMA systems that provide hardware mechanisms for shared virtual address spaces across multiple compute nodes. Both Wildfire and SHRIMP [4][5] made use of address translation for this purpose. Using the same partitioning scheme as our Laplacian kernel, we found that the vast majority of 4KB pages contained both shared and private data, implying that nearly every page must undergo translation when the shared data is communicated between nodes. This held regardless of cube size and was verified using a microbenchmark.

Our preliminary work into this subject, and several favorable workload characteristics, imply to the authors that a virtual coherence system for the grids being computed over may be a potential solution. The authors note that in memory bandwidth bound workloads, address translation causes will cause a performance decrease because it requires several memory accesses per translation. The authors also note that POSIX system calls like *mmap()* can specify a starting virtual address for a memory allocation and that synonyms over the virtual address range of the grid are highly unlikely [20]. Previous work on virtual coherence and virtual caching cite synonyms as being one of the primary problems for designing these systems.

7. Related Work

The area of accelerating scientific workloads has a wide variety of related work. Inspiration was drawn from several different areas of research over the course of this project.

Ultimately, we postulated that being able to emulate a shared memory system over multiple Phis was the correct choice for developing scientific workloads. Within that, we aimed to be able to send small message sizes while effectively using bandwidth provided between the interconnects. To start, we investigated large parallel systems: soNUMA, Wildfire, and the SHRIMP multicomputer [4][5][6].

SCIF allowed for us to perform Remote Memory Access operations between multiple Intel Phis, which gave the appearance of a NUMA system to our software. In particular, we looked towards these systems for insight into how other systems sent messages across different address spaces. Both Wildfire and SHRIMP made extensive use of address translation and virtual memory to give the impression of a unified virtual memory space, for small messages.

Several other authors have concluded that MPI and SCIF has poor bandwidth utilization when small message sizes are used and much higher bandwidth utilization when large messages are sent between multiple Xeon Phis. Potluri et. al. extended the MVAPICH2 MPI library to support SCIF, and are able to achieve over 7 GB/s intra-node bandwidth for large messages [13]. Si and Ishikawa proposed DFCA-MPI, enabling direct communication over InfiniBand [14] and corroborated that small message sizes do not achieve good bandwidth utilization. Noack et. al. developed HAM-Offload in order to streamline the offload programming model [15]. Jose et. al. presented a design for OpenSHMEM on the Phi [16]. Like this project, Haidar et. al. examined scaling effects across multiple accelerators in linear algebra workloads, and the communication penalties associated with them [17]. Neuwirth et. al. investigated Xeon Phi performance when connected to the EXTOLL interconnect [18]. The same authors later compared previous communication models in a meta-analysis, finding MPI performance to be less than desirable [8].

8. Conclusion

This project examined the communication bottleneck in systems with multiple Xeon Phi accelerators. We corroborated results found in previous research showing poor interconnect bandwidth utilization for small messages. We also showed how SCIF can improve message bandwidth for larger sizes, and improve overall performance in scientific applications over MPI. In addition, we present the communication cache - a simple interface for programmers that also alleviates the communication overhead penalty of small messages. We provided insights into the results that we received and push for the adoption of hardware acceleration for inter-node communication for accelerators.

9. Acknowledgements

We thank Nathan Mitchell, Haixiang Liu, Tim Czerwonka, the Multifacet Group, and the UW Graphics Group for their insightful comments and feedback on this work. This work is supported in part by the US National Science Foundation (CNS-1302260, CCF-1423064, CCF-1438992, CCF-1533885, and IIS-1253598) and the University of Wisconsin-Madison (Named professorship to Hill). Hill has a significant financial interest in AMD.

10. References

[1] A. McAdams, E. Sifakis and J. Teran, "A parallel multigrid Poisson solver for fluids simulation on large grids", *Eurographics/ACM SIGGRAPH Symposium on Computer Animation*, 2010.

[2] G*RAPHICS DOUBLE DATA RATE (GDDR5X) SGRAM STANDARD*, 1st ed. 2016.

[3] "FAQ:Debugging applications in parallel", *Open-mpi.org*, 2016. [Online]. Available: https://www.open-mpi.org/faq/?category=debugging.

[4] E. Hagersten and M. Koster, "WildFire: A Scalable Path for SMPs", *High Performance Computer Architecture*, 1999.

[5] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten and J. Sandberg, "Virtual memory mapped network interface for the SHRIMP multicomputer", *ACM SIGARCH Computer Architecture News*, vol. 22, no. 2, pp. 142-153, 1994.

[6] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi and B. Grot, "Scale-out NUMA", *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems - ASPLOS '14*, 2014.

[7] "Building a Native Application for Intel® Xeon Phi™ Coprocessors", *Intel Corporation*, 2014. [Online]. Available: https://software.intel.com/en-us/articles/building-a-native-application-for-intel-xeon-phi-coprocessor.

[8] S. Neuwirth, D. Frey and U. Bruening, "Communication Models for Distributed Intel Xeon Phi Coprocessors", *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*, 2015.

[9] "What is High Bandwidth Memory (HBM Technology)?", *Amd.com*, 2016. [Online]. Available: http://www.amd.com/en-us/innovations/software-technologies/hbm.

[10] "NVIDIA NVLink High-Speed Interconnect | NVIDIA", *Nvidia.com*, 2016. [Online]. Available: http://www.nvidia.com/object/nvlink.html.

[11] "Intel® Xeon Phi™ X100 Family Coprocessor - the Architecture | Intel® Developer Zone", *Software.intel.com*, 2012. [Online]. Available: https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner.

[12] "Intel® Xeon Phi™ X100 Family Coprocessor - the Architecture | Intel® Developer Zone", *Software.intel.com*, 2012. [Online]. Available: https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner.

[13] S. Potluri, A. Venkatesh, D. Bureddy, K. Kandalla and D. K. Panda, "Efficient Intra-node Communication on Intel-MIC Clusters," *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, Delft, 2013, pp. 128-135. doi: 10.1109/CCGrid.2013.86

[14] M. Si, Y. Ishikawa and M. Tatagi, "Direct MPI Library for Intel Xeon Phi Co-Processors," *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, Cambridge, MA, 2013, pp. 816-824. doi: 10.1109/IPDPSW.2013.179

[15] M. Noack, F. Wende, T. Steinke and F. Cordes, "A Unified Programming Model for Intra- and Inter-Node Offloading on Xeon Phi Clusters," *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, New Orleans, LA, 2014, pp. 203-214. doi: 10.1109/SC.2014.22

[16] J. Jose *et al.*, "High performance OpenSHMEM for Xeon Phi clusters: Extensions, runtime designs and application co-design," *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, Madrid, 2014, pp. 10-18. doi: 10.1109/CLUSTER.2014.6968754

[17] A. Haidar, P. Luszczek, S. Tomov and J. Dongarra, "Heterogenous Acceleration for Linear Algebra in Multi-coprocessor Environments", Lecture Notes in Computer Science, pp. 31-42, 2015.

[18] S. Neuwirth, D. Frey, M. Nuessle and U. Bruening, "Scalable communication architecture for network-attached accelerators," *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Burlingame, CA, 2015, pp. 627-638. doi: 10.1109/HPCA.2015.7056068

[19] N. Jerger and L. Peh, "On-Chip Networks", Synthesis Lectures on Computer Architecture, vol. 4, no. 1, pp. 1-141, 2009.

[20] Personal conversation with Eftychios Sifakis.