Evaluating the Support of MTC Applications on Intel Xeon Phi Many-Core Accelerators

Karl Stough, Serapheim Dimitropoulos, Poornima Nookala
Illinois Institute of Technology

***Abstract*—As Many-Task Computing (MTC) is becoming commonplace on clusters, grids, and supercomputers, research that aims to take advantage of the new advances in hardware for MTC workloads becomes more relevant. A good example is the design of frameworks like GeMTC that incorporate general purpose GPU hardware to improve the concurrency of executing tasks. For this project we will attempt to support MTC workloads on the Intel Xeon Phi. Our plan is to develop two frameworks that will achieve that goal. One based on OpenMP and the other one based on Intels Symmetric Communication Interface (SCIF) provided for Many-Integrated Core (MIC) accelerators like the Xeon Phi. Both frameworks aim to provide an identical interface to the one found in the GeMTCs API. Ideally the two implementations will be suitable drop-in replacements of the GeMTC framework for the Xeon Phi coprocessor. Our end-goal is to present how programming many-core computing processors can be made easier and more productive using OpenMP or SCIF, and enable the execution of MTC workloads on this relatively new coprocessor design.**

***Keywords*-Many-task computing; Accelerators; Intel Xeon Phi; Coprocessor; Programming models; Execution models.**

I. Background Information

The Intel Xeon Phi is a hardware coprocessor from Intel. It is a PCI device with roughly 60 cores and over 240 hardware threads. Its design makes it ideal for application that are performance critical and need large levels of parallelism. Moreover, the fact that it implements x86 for its instruction set architecture, makes its integration with existing systems simpler than the integration of other accelerators like General Purpose GPUs (GPGPUs). For these reasons, it is worthwhile to note that the Xeon Phis are considered valuable additions for clusters, grids and supercomputers.

Many-Task Computing (MTC) has been an emerging paradigm and area of research for some years now. Therefore, considering embedding the capabilities of the Xeon Phi in systems that support MTC workloads is considered a relatively new ground. That is the ground that this paper attempts to cover. A MTC workload consists of task that run uninterrupted from start to completion. Rather than retaining state like MPI processes, they resemble the simple input-process-output model of procedures. The task duration may be highly variable, ranging from subseconds to minutes. Their dependency and data-passing characteristics may range from many similar tasks to complex, and possibly dynamically determined, dependency patterns. GeMTC is a CUDA based GPU framework which allows Many-Task Computing workloads to run efficiently on NVIDIA GPUs [3].

GeMTC's architecture involves queueing incoming jobs that will be executed on the GPU and return their results back to the host. The novelty in the design of this framework is that the different jobs that are running in parallel, are also isolated from each other and therefore the utilization of the GPU is almost maximized. Moreover, the whole architecture provides a relatively user-friendly and very high-level API that abstracts many details but still allows the user to have a lot of control over the GPU. For example, a user can call gemtcMalloc() that behaves exactly like the conventional malloc, while the GPU-related details are hidden by a sub-allocator built on top of the GPU's memory allocator. Unfortunately, the framework's implementation and architecture is very closely-tied to the architecture and the conventions of GPUs.

GPUs have a very restrictive programming model, but provide at least an order of magnitude better throughput for applications painstakingly coded to that model [5]. To program GPUs, typically there is a need to learn another programming language such as CUDA (NVIDIA) or OpenCL (AMD). As a result, existing vendors must spend extra time and effort to modify or rewrite parts of their codebase to take advantage of the new capabilities provided by General Purpose GPUs (GPGPUs). Besides that, barely rewriting an application just to offload computations to a GPU rarely works well. Because of the architecture of most GPUs out there, applications must be tailored from the ground up to follow the rules of the restrictive programming model of GPUs, otherwise they may suffer from severe performance penalties. Because of that, interested vendors cannot afford to go through the effort involved. Finally, while GPUs are great for massively parallel applications with thread-switching that comes almost at no cost, their performance can take a large hit when executing programs with complex logic (like complicated branching and looping for example). Therefore they may be unsuitable for certain applications of MTC.

II. Proposed Work

The Intel Xeon Phi follows an alternative programming model that, although may not provide the same level of parallelism, provides more flexibility and therefore can be more suitable for certain application of MTC that GPUs are not suited for. The reason is that the Xeon Phi has x86 cores that are more capable (can handle complex branching and looping) than most GPU cores. Another advantage of having x86 cores is that programming the coprocessor minimizes the amount of work that needs to be done in order to integrate a Xeon Phi to an existing system. That is because the Phi does not require being programmed in any specific framework and it can natively run applications written in C with Pthreads or OpenMP.

All of the above facts were enough to motivate us to work on this project. This project aims to implement two frameworks, one based on OpenMP and one based on Intel's Symmetric Communication Interface (SCIF), that provide a functionality to the Phi which is similar to the one provided by GeMTC for GPUs. By doing that we believe that the Xeon Phi will be ready to be integrated to existing systems and ultimately be able to undertake MTC workloads in an efficient manner.

The outline of our contributions for this project is the following:

1) Design, analyze and implement two frameworks, one in OpenMP and one in SCIF, to provide functionality identical to GeMTC and to allow MTC workloads to run on Intel Xeon Phi accelerators. The team shall attempt to duplicate GeMTCs API for easy integration with systems and software that already use GeMTC like the Swift/T framework for example.
2) Evaluate the performance of running concurrent homogenous and heterogeneous tasks across the Phis 60-core architecture (240 hardware threads).

III. Related Work

The GeMTC framework and its API are limited exclusively to NVIDIA GPUs since it is developed in CUDA. Even though we will be working with OpenMP and SCIF on the Xeon Phi, some ideas from the CUDA version of GeMTC will still be applicable. For example using a buffer in the hosts memory that will flush tasks periodically to the device seems like a useful concept to reduce communication overhead. The actual implementation of course will have to be made from scratch since there are differences between the architecture of the Xeon Phi and the GPUs that GeMTC was initially designed for.

Moreover, there have been some preliminary results for MTC workloads on the Xeon Phi [4]. In order for these results to be acquired at that time a similar framework to GeMTC was made that used SCIF. The issue with that implementation is that it does not provide the whole feature set of GeMTC and therefore is not suitable for MTC use. Nevertheless, that initial implementation gave signs that using SCIF for direct communication between the host and the accelerator via the PCI Express bus minimized certain overheads over a similar OpenMP implementation.

More specifically now for OpenMP applications on the Xeon Phi, there have been results of performance comparisons[7]. The results show that the overhead of the standard OpenMP constructs which use synchronization is smaller than on big SMP machines, which makes the approach very promising for many MTC applications using OpenMP. The overhead of the offload pragma used in the language extension (LEO) is also quite low, so that it will not limit the scalability. This results alone compared to the easiness of just specifying pragma directives to enable parallelism motivated us to give OpenMP a try.

As for SCIF compared to OpenMP there exist results that show high performance wins (80% in throughput) for tasks with sizes equal or less than 4KBs when SCIF uses its socket-like API [8]. Unfortunately 4KBs is not a significant size and there are no results indicating what is the performance win of SCIF using its Remote Memory Access API over OpenMP if any.

IV. Architecture

Due to the foundations in Intel architecture, the coprocessor can be programmed in several different ways. For the OpenMP we used offloading so the code run on the host can offload the specific computation to the Phi. For the SCIF implementation, on the other hand, we implemented to run natively on the Phi while accepting jobs from clients running on the host.

There are several advantages and disadvantages between the two methods. The major advantage of native execution coupled with SCIF over offloading is that the developer gets more control overall in the configuration and the architecture of their design in order to maximize performance. Computation does not necessarily have to be transferred back to to the CPU. In addition, different MIC cards can communicate dirrectly with each other basically making certain designs more efficient. Finally, frameworks that use offloading mode, do not necessarily take advantage of the DMA-features of the hardware they run on while on SCIF you are guaranteed that if you are using Remote Memory Access (RMA) [11] [12].

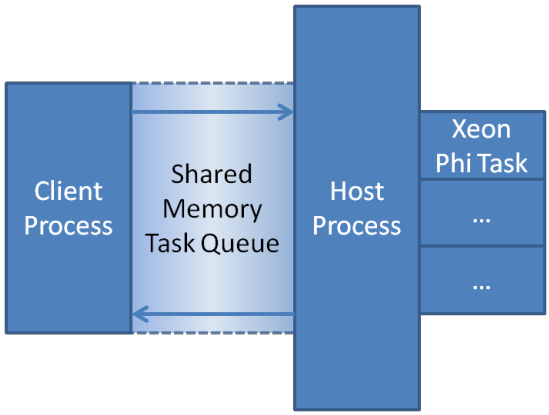
That is not to say that OpenMP does not come with any advantages over SCIF. Quite the opposite, the advantages of offloading are pretty significant for the framework that was implemented for this project. The low-level C code needed for the SCIF implementation is relatively a lot more complex when compared with pragma directives provided by OpenMP. In addition, using SCIF implies that the framework must have at least one of its parts running natively on the Phi

as the endpoint. In order to do that the developer needs to set up an application to run natively on the Phi and involves a lot of configuration. Using OpenMP with the offloading capabilities provided by the MIC, all this configuration is taken cared for you [9].

A. OpenMP Implementation

The OpenMP version of the framework mimics GeMTC by utilizing OpenMP asynchronous offloading capabilities. We have employed a Producer-Consumer architecture which communicates using shared memory for IPC. The Consumer side hosts the framework which runs as a single thread and launches as many master processes on the Xeon Phi as specified by the user. The master processes use the shared memory space as a queue structure, continuously accepting new tasks from producer processes. Likewise, the producer acts as a client process which submits tasks to the queue via this shared memory pool.

For testing this framework, we have implemented two different types of applications: Sleep and Matrix Multiplication. Both were developed and tested using the OpenMP approach of offloading tasks on to Xeon Phi. The master processes in our framework read the tasks from shared memory location and based on the type of task, offload the computation part to Xeon Phi. Asynchronous offloading is used to allow the framework to continue accepting tasks while other tasks are running. The Phi sends a signal back to the master processes after job execution has completed. At this point the output is sent back to the Client. This approach was chosen to provide the same feature set as GeMTC while taking advantage of asynchronous offloading capabilities of OpenMP.



B. SCIF Implementation

The SCIF implementation is a complete port of the GeMTC framework and the source code is available online [1]. Of course it comes with some important differences which are unavoidable since the underlying hardware is different (GPGPU vs Xeon Phi architecture).

The core architecture of GeMTC is actually completely rewritten in pure C from CUDA and abstracted out into a shared library that we named **libmtcq**. The library includes all the main functionality of GeMTC (handling queues, pushing jobs, distribute tasks to workers, etc.) and can be compiled against any application that wants to use it. It is also completely parametrizable in terms of queue sizes, worker threads, and application threads. In addition, other applications can be easily compliled into the shared library, so the library itself can provide the functionality of these applications to external clients who want to use it.

Since the Xeon Phi does not have the hierarchical architecture of SMXs and Warps nor the concept of application kernels that you generally see in GPGPUs, everything is implemented with stardard threading - Pthreads to be more specific (note that Pthreads are also used internally for OpenMP applications [13]). There is a parametrizable number of master threads that dequeues tasks from the incoming queue. If the task is a parallel application, which is the case most of the time, then the master thread will assign the task to the specified number of worker threads. Else if it is sequential only one thread will be assigned and the master thread will go back to dequeue more jobs. Each queue is implemetented as a finite buffer from the Producer-Consumer model which means that it uses a single mutex and two semaphores[17] to ensure that no deadlocks or data-races arise. This is an actual improvement over the GeMTC framework which just uses a single mutex and only ensures that no deadlocks occur.

The rest of the framework implementation now lays on top of that shared library. It is modeled after a client-server architecture where the clients send their tasks to the Phi from the host and the server, which runs natively on the Phi, accepts the jobs. After submitting the job, the clients can request the result and the server will deliver it to them when the task has finished processing and is placed on the results queue of the framework. The whole procedure is non-blocking for the server who can handle multiple requests and submissions at the same time. That functionality is implemented with epoll() for handling connections that are later passed to threads [14] that push or dequeue tasks from the queues. The SCIF socket-like API is used for communications between the server and the clients. It is also worth noting that an optional mechanism for receiving acknowledgements from the server exists when sending a message for debugging and verification procedures, and it can be enabled by applying the debugging flags during compilation.

For this framework we made 5 sample applications that will test not only how the actual implementation performs but also how relevant optimization techniques for normal CPUs are for the coprocessor. One of these test application is a sleep-test application and the rest are different implementation of matrix multiplications. One is naive, the second one is optimized according to CS:APP2[16] to be very cache-friendly, the third one is a blocking version [16]

(with a parametrizable block/cache-size that fits exactly on a L1/L2 cache of a Xeon Phi core), and finally a parallel version using Pthreads. So the goal in test would be to use the sleep-test and parallel matrix multiplication applications for preliminary throughput testing while using the naive, optimized and blocked versions of matrix multiplication for testing if really these extra optimizations that are considered common for CPUs will have the same performance wins on the Phi.

V. Evaluation

All of our experiments were ran on the MidWay High-Performance Computing Cluster at University of Chicago. Our testing host is an Intel SandyBridge with 16 cores at 2.6 Ghz and 32 GB of RAM. It has 2 Xeon Phis attached to it. Both of them are from the 5100 series of Intel coprocessors and have 60 cores at 1.053 GHz each and 8 GB or RAM.

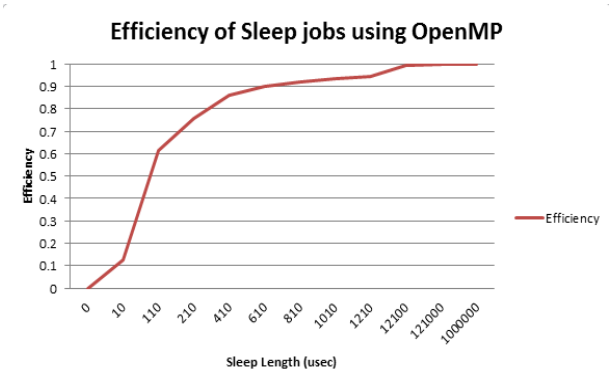
A. Preliminary Results for OpenMP

As we mentioned before, to compare the overhead of OpenMP we have analysed two different types of programs: sleep and matrix multiplication.

Synthetic Sleep Workloads

Sleep 0 jobs were used to measure the throughput of offloading tasks on to Xeon Phi. Experiments were performed by offloading sleep jobs individually and measuring the effective time spent between offloading and receiving a signal back from Xeon Phi. Our framework was able to offload 20K Sleep 0 tasks per second to Xeon Phi using OpenMP asynchronous offloading approach.

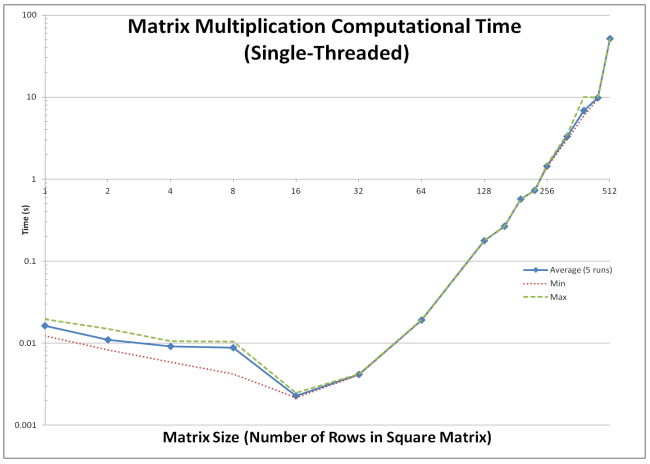
Experiments were also performed by offloading various sleep length tasks. Execution time was computed effectively by offloading 100 tasks and taking average of the time. As seen in the below figure, efficiency is $> 90\%$ for task lengths > 620 microseconds. This clearly shows the granularity of tasks and level of parallelism to be employed when utilizing OpenMP offloading capabilities.



Matrix Multiplication Results

In order to assess the real-world performance of the Xeon Phi, the team developed a matrix multiplication application to show how well it performed for various task sizes and levels of concurrency. The team also compared the performance of the Phi to CPU performance. It should be noted that the work performed is exponentially greater than the matrix size, since a naive matrix multiplication algorithm of $O(n^3)$ was used.

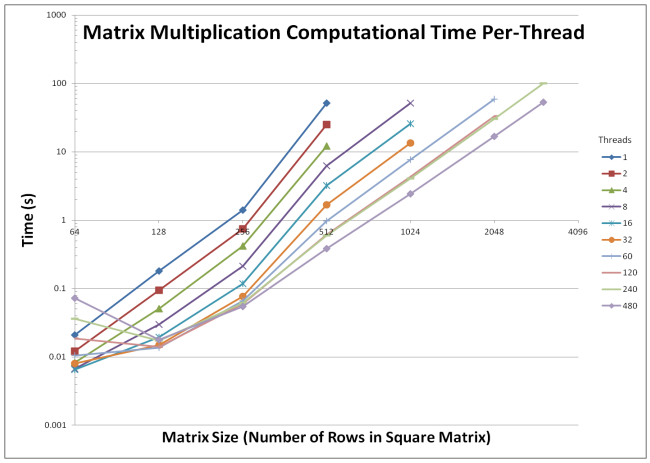
The first test performed was to analyze the performance penalty associated with offloading tasks to the Phi (see Figure below). In this test, the test load was increased while using only a single thread on the Phi. If the offloading cost was non-existent, the plot of time taken to complete the task would have been nearly linear with the amount of work performed. In reality, the overhead only becomes negligible when matrix sizes of 64x64 were tested. Before that point, the task completion time remained fairly constant. At larger matrix sizes, the time taken increases linearly with the amount of work performed.

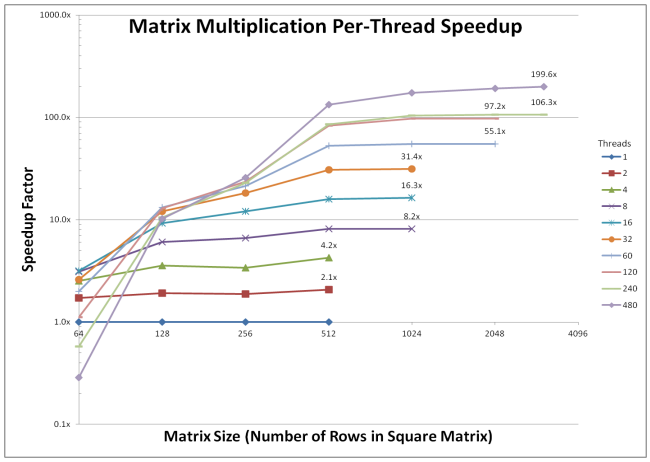


The team also analyzed the performance gain from an increase in the number of threads. The testing methodology used is as follows: The number of threads was increased from 1 to 480 as the workload also increased. Due to the exponential growth of work generated by the matrix multiplication procedure, task completion time was limited to approximately 1 minute. If tasks couldn't complete in that time, they were discarded from the test. The plot in the two figures below highlight this. Each point in the plot represents the average of 5 tests using that combination of matrix size and thread concurrency.

It was found that while the single-threaded tasks scaled fairly linearly with the workload, the many-threaded tasks didn't achieve optimal scalability until much larger matrices were tested. While this is shown to some extent in the figure below, the speedup in figure after it better demonstrates this.

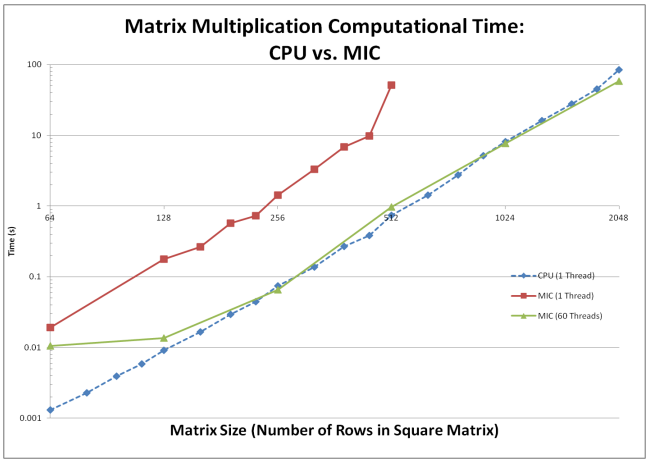
Speedup was calculated based on the task completion time of the single-threaded matrix multiplication. Due to the task duration for single-threaded multiplications, speedup for 8 or more threads at matrix sizes beyond 512x512 elements was extrapolated from the single-threaded test assuming a proportional time was taken. This gives actual speedup which was very close to the ideal case where the maximum speedup would have been equal to the number of threads used.





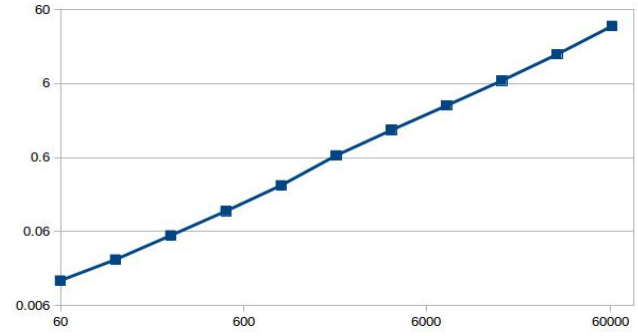
The final test aimed to compare the performance of the Phi vs. running the task natively on the CPU. In this test, tasks bypassed the offloading procedure and were run directly on the CPU. It should be noted that the team was only able to run tasks using one thread on the CPU due to issues getting OpenMP running natively on the CPU. Given more time, the team would have liked to troubleshoot this issue to get a better test.

The last figure of this subsection shows the runtime comparison between 1 thread on the CPU compared to both 1 and 60 threads on the Xeon Phi. It is expected that 1 CPU thread wouldn't equate to 1 thread on the Phi, but it is surprising to see that a full 60 threads were required to match the CPU's performance. This is especially surprising considering that the CPU had 31 additional threads that could have also been used to improve performance. It is likely that there are performance improvements that could be made to the framework. Offloading is likely not implemented in the most efficient manner and further compiler optimizations should also be investigated. Additionally, the test application was written with the CPU in mind, so some operations may be optimized or replaced with better methods if running in offloaded mode. This is left as future work.



B. *Preliminary Results for SCIF*

Again as for the evaluation of OpenMP we implemented two types of applications: sleep and matrix-multiplication jobs. Unfortunately, the SCIF implementation started failing after matrix sizes of 32 by 32 elements due to the fact that our SCIF implementation used the socket-like API which is not made for bulk I/O. Therefore, we do not have any results on the performance of matrix multiplications nor the comparison between the different implementations that use different optimizations each. We could add support for bulk I/O usign the RMA API but unfortunately we got our hardware very late and as a result we don't have enought time to enable proper support for this case (see Appendix). Thus we leave it as future work.



As for our results sending sleep jobs to the framework, we

have the figure above with some very preliminary data. The Y-axis represents the number of seconds while the X-axis represents the number of jobs. The complete cycle of submit-execute-dequeue result for 60 'sleep 0' jobs took 0.013 seconds (lowest point in graph) while on the other end it took 36.27 seconds for a complete cycle of 61440 'sleep 0' jobs. Even though on the graph it seems like the framework scales very well the difference between the actual CPU time and the ideal is getting larger after 15360 submitted jobs.

VI. Conclusion and Future Work

Enabling MTC workloads on the Xeon Phi is relatively new ground, and we were glad to have the opportunity to learn about this new technology with this project. To achieve this, we designed a framework that that not only sends and executes tasks to the Xeon Phi but also ensures that these tasks are isolated from each other and can run in parallel. This way the Xeon Phi can be utilized in an efficient and correct way. Our work built upon the existing functionality of GeMTC and in the future would allow for an identical interface which could be dropped into Swift/T. Based on our research and analysis, we chose to implement both a OpenMP framework as well as a SCIF-based framework and were able to run applications on the Xeon Phi.

Our preliminary evaluation data are encouraging and should provide enough motivation for future. The evaluation data from the OpenMP-based framework show some interesting results that emphasize some of the Phi's strengths. For the SCIF implementation unfortunately the preliminary data only show us that it still scales pretty well but we haven't pushed it to the point that scalability degrades. Also, we due to the lack of time towards our deadline we couldn't enable its support for bulk I/O that would allow us to compare our matrix mutliplication benchmarks between the two frameworks.

Our future work includes:

1) Run more evaluation experiments with the OpenMP framework and explore the interesting parts pointed out (See evaluation section)
2) Enable support for bulk I/O in the SCIF implementation through the use of RMA
3) Evaluate our designs and compare them to one another
4) Compare our designs to the actual GeMTC framework to draw a clear line between the types of tasks that are more suitable for GPGPUs and the tasks that are more suitable for the Xeon Phi.

Besides the clear goals for our future work above there are also further improvements that are more tied to the implementation of our designs. For example, a possible performance improvement would be the use of spinlocks instead of traditional mutexes that are in use in the shared library. Another improvement that could be a "killer-feature" in the framework would be the dynamic loading of external applications by the framework. The point is that now, since the prototypes exist and they are functional, more features can be added allowing future work of any direction.

Finally, there is future work that involves the integration of our designs into bigger systems. For example, someone could build on top of our work for the Phi and the GeMTC work for GPUs and combine them in a transparent manner. This way when a task is submitted the system will decide whether to execute the task on the CPU, the Phi or the GPU, based on the task's type and system load in order to maximize performance. Another example would be to integrate our prototypes with the Swift/T framework and improving the performance of MTC workloads by spliting work to multiple nodes.

VII. Acknowledgements

We would like to express that we are very thankful to Professor Ioan Raicu (Illinois Institute of Technology) for suggesting such an interesting project and for his feedback on our progress. We are also very grateful to Michael Wilde (Argonne National Laboratory) for letting us the MidWay cluster at University of Chicago to run our experiments. Finally we would also like to thank Andy Wettstein and Devon Compton from University of Chicago that gave us resources and helped us troubleshoot problems on the MidWay cluster.

VIII. Appendix

Individual Contributions

Poornima Nookala designed and implemented a framework similar to GeMTC using OpenMP. She implemented her OpenMP framework in offloading mode and native mode. After that she ran some preliminary tests to decide which approach is the best and it turned out to be offloading mode. Her work also includes merging the shared memory implementation and the matrix multiplication code with the rest of the OpenMP framework. She evaluated the performance of sleep tasks and their throughput on the Xeon Phi.

Serapheim Dimitropoulos designed and implemented the shared library **libmtcq** from the groundup. The shared library was used as the basis for both frameworks. The OpenMP framework by Poornima and Karl added OpenMP directives to libmtcq to implement their framework. Serapheimalso implemented the SCIF framework on top of libmtcq by adding the client-server part. He implemented 5 clients for the SCIF framework and did the evaluation part for the SCIF framework.

Karl Stough assisted in the design and implementation of the OpenMP offloading framework. He also implemented the shared memory interface for IPC and the matrix multiplication test application for that same framework. His contributions also include debugging and testing

the offloading procedures that used OpenMP and finally the performance analysis of the matrix multiplication application.

A Final Note of our Work

Our work may seem as incomplete from some aspects but that is with reason. We got our hardware 3 weeks before our deadline and therefore we did not have enough time to test our prototypes and evaluate them. Before we got our hardware we tried our best to implement as much as possible from our local machines. Poornima and Karl were experimenting with OpenMP while Serapheim implemented his shared library and the framework on top of it with traditional UNIX sockets. All three of us wanted to have something ready before we get our hardware so we don't waste any time. When we eventually got the hardware, 3 weeks before our deadline, Poornima and Karl put their MIC offloading directives in use while Serapheim switched all the code from sockets to SCIF. Of course the transition was not easy and many errors had to be fixed leaving us with less time to do our evaluations. Thus, even if we got some very valuable feedback and suggestions during our presentation, we were not able to apply all of them in our final submission.

REFERENCES

[1] Serapheim Dimitropoulos, *"GeMTC-SCIF Source Code Repository"*, https://github.com/sdimitro/scif-modules/tree/master/scif-sc.

[2] Poornima Nookala, Karl Stough, *"GeMTC-OpenMP Source Code Repository"*, https://github.com/pnookala/MIC_OpenMP_GeMTC.

[3] S. Krieder, J. Wozniak, T. Armstrong, M. Wilde, D. Katz, B. Grimmer, I. Foster and I. Raicu, *"Design and Evaluation of the GeMTC Framework for GPU-enabled Many-Task Computing"*, ACM HPDC, 2014.

[4] J. Johnson, S. Krieder, B. Grimmer, J. Wozniak, M. Wilde and I. Raicu, *"Understanding the Costs of Many-Task Computing Workloads on Intel Xeon Phi Coprocessors"*, GCASR, 2013.

[5] NVIDIA Inc. , *"CUDA C Programming Guide v6.5, Section 5.1-5.4, Performance Guidelines"*, 2014.

[6] J. Fang, H. Sips, L. Zhang, C. Xu, Y. Che, A. Varbanescu, *"Test-Driving Intel Xeon Phi"*, ACM Digital Library, 2014.

[7] T. Cramer, D. Schmidl, M. Klemm, D. Mey, *"OpenMP Programming on Intel Xeon Phi Coprocessors: An Early Performance Comparison"*, 2013.

[8] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu and Y. Wang, *"High-Performance Computing on the Intel Xeon Phi: How to Fully Exploit MIC Architectures"*, Springer, 2014, pp. 3-30.

[9] Rezaur Rahman, *"Intel Xeon Phi Coprocessor Architecture and Tools"*, Apress, 2013.

[10] Jim Jeffers, James Reinders, *"Intel Xeon Phi Coprocessor High Performance Programming"*, Apress, 2013.

[11] Intel, *"Intel Xeon Phi Coprocessor System Software Developers Guide"*, 2014.

[12] Intel, *"Intel Many Integrated Core Symmetric Communications Interface (SCIF) User Guide"*, 2012.

[13] David R. Butenhof, *"Programming with POSIX Threads"*, 1997.

[14] Robert Love, *"Linux System Programming"*, 2013.

[15] Bryant, O'Hallaron, *"Computer Systems: A Programmer's Perspective"*, 2011.

[16] Bryant, O'Hallaron, *"Using Blocking to Increase Temporal Locality"*, http://csapp.cs.cmu.edu/2e/waside/waside-blocking.pdf.

[17] Allen B. Downey, *"The Little Book of Semaphores"*, http://greenteapress.com/semaphores/.