Hybrid Programming and Automated Programming: Current and Future Programmer Responsibilities

a reading study

Helen Parks
CSE 262
March 16, 2013

This project aims to explore hybrid and automated programming models, including an analysis of their advantages and disadvantages from a programmers perspective, as a step toward answering the question, Will automated programming models someday replace hybrid models as the norm on large-scale and heterogeneous architectures? This paper examines several hybrid and several automated models generally as well as the specific hybrid and automated models available for Intels new Xeon Phi coprocessors (often referred to as MIC for many integrated core). The new MIC coprocessors as well as Intels marketing for them represent a small but significant departure from the architectures and programming style around which many hybrid programming models are designed, making it interesting to consider how coprocessors may affect the future evolution of programming models.

1 Introduction

The increasing hybridization and heterogeneity of computer architectures, from multicore processors in the average laptop to multi-node and multi-GPU supercomputers, fundamentally drives the development of both hybrid programming and automated programming models. Several architectural considerations can motivate the use of a hybrid programming model. The heterogeneity in a hybrid architecture means that while one component may be best or most easily programmed with a certain model, another component could require the use of a different model. For instance, OpenMP provides a simple interface for parallel programming on a shared memory architecture, but when several shared memory nodes of a supercomputer need to communicate it must be done with a message-passing model like MPI. Hybrid models can also be motivated by the shortcomings of a particular programming model, like the lack of locality control in MPI programming. In a large, heterogeneous architecture, a programs performance can suffer from MPI processes that
are near each other in the algorithms abstraction executing on nodes which are physically far from one another which slows their communication.

Hybrid programming attempts to meld multiple programming models so that their hybridization works well on a given hybrid architecture. This solution naturally requires more of the programmer who becomes responsible for knowing multiple programming models as well as knowing enough about the implementation architecture to use multiple models effectively. Conversely, automated programming attempts to lessen programmer burden and increase programmer efficiency by automating more of the process. Development of automated programming models is grounded in the beliefs that computer architectures will continue to grow in complexity and heterogeneity and that it is unreasonable to expect programmers to continue to be experts in architectural concerns and multiple programming models as well as their problem domains. Increased heterogeneity may also lead to an increasingly diverse range of architectures on which a given code needs to run. Performance portability concerns also motivate many automated models.

In 2013, Intel released the Xeon Phi coprocessor, the first commercial release of Intels many integrated core (MIC) architecture model, essentially Intels answer to GPGPs. The coprocessors are built with x86 instruction cores, and Intels marketing emphasizes a resultant ease of programming. Intels marketing also seems to target a wider audience than GPGPUs, from HPC users currently using GPGPUs to more traditional consumers likely to need some software help in using the coprocessors. Thus, the new coprocessors provide an example of a new architecture whose design is in part motivated by ease of programming, making them an interesting case study for exploring programming models.

2 Hybrid Programming Models

As mentioned earlier, hybrid programming models are motivated by heterogeneous architectures and the ways in which those architectures interact with given single programming models. In this paper we examine models for programming supercomputers consisting of many shared-memory nodes connected over an interconnect and computing systems in which an accelerator device, such as a GPU or an FPGA, is connected to a host computing device, possibly at each of many connected nodes. When programming a network of shared memory nodes, the programming model must allow for inter-node communication over the network. The Message Passing Interface (MPI) is the most common and established model for doing this. All of the models we consider for programming a network-of-nodes architecture consist of MPI plus an additional programming model. For host-device architectures, we consider the cases of GPUs, FPGAs. We will consider hybrid programming on MIC separately below.
2.1 Distributed Networks

This section describes models for programming distributed memory networks with hybrid models involving MPI.

2.1.1 MPI Overview

MPI allows for single program, multiple data programming of distributed memory systems like the network-of-nodes architecture discussed above through the use of concurrently running program instances called processes. MPI automatically assigns each process a unique process id number, beginning at 0. Typically each MPI process runs on its own core or its own node. MPI defines a library of functions accessible from C, C++, and Fortran programs that allow separate MPI processes to communicate with one another. These communications may be point-to-point, involving only two processes, or collective, involving multiple processes. [16]

If an MPI program runs processes on multiple nodes, the data used by the program must be partitioned and divided among the nodes. MPI leaves the abstract partitioning of data among different process IDs entirely to the programmer. Hence, an MPI programmer has complete control over abstract data distribution and communication patterns. However, MPI presents a flat network topology to the programmer and handles distribution of processes across the compute hardware itself. Thus, a programs performance may vary from run to run due to the programs communication patterns operating over different hardware topologies.

2.1.2 MPI + OpenMP

OpenMP, with MP standing for multiprocessing, is an API for shared memory parallel programming. OpenMP allows for higher-level programming than MPI. A programmer can often specify with a short pragma that a certain region of code should be executed in parallel. In this case, OpenMP will spawn several new threads to execute the specified region. The programmer can but is not required to make any further specifications of how the program should run. OpenMP can take care of selecting how many threads to spawn, spawning them, dividing the work of the specified section among the threads, and joining the threads when the work is complete. Hence, OpenMP operates at a slightly higher level than MPI, which requires the programmer to be explicit about how parallelism is executed. Due to the use of pragmas, OpenMP requires compiler support. [16]

MPI and OpenMP are often used together on large network-of-nodes computers when the nodes are multicore shared memory computers. One of the major costs of running an MPI program is the cost of communication, the sheer time required to send and receive messages among processes. When an MPI program runs on a shared memory system, it artificially divides the memory into regions specific to certain processes and passes messages among processes. This can be slower than implementing the computations carried
out by all processes on the node with a shared memory model like OpenMP, especially if the communication to computation ratio of the program is high. To avoid such a potential slowdown MPI+OpenMP hybrid models often run a single MPI process per shared memory node and parallelize each process computations within the node using OpenMP. MPI programs can also easily suffer from load imbalance since the programmer bears responsibility for the load distribution and maintaining a balanced load distribution across processes can be quite difficult, especially if the distribution of work across the data structures being operated on is likely to change. OpenMPs automatic work and data distribution patterns are much less susceptible to load imbalance since they spawn a new set of threads with a new work distribution at each parallelized section. Hence an MPI program with load-imbalance issues may see performance improvement after hybridization with OpenMP.

The MPI+OpenMP hybrid model clearly has the potential to improve performance for certain codes, and it has been demonstrated that the two standards interact well together on the large-scale machines the hybrid model is most desirable for. Yet the comparison studies of all-MPI versus hybrid codes are, for the most part, inconclusive. [cite Lusk, Chan] A 2000 study of both approaches applied to the current NAS benchmarks concluded mainly that the superiority of one model over the other depends with high sensitivity on the communication patterns, memory access patterns and available parallelism of the code being implemented as well as the relative speed of the hardware nodes to the speed of their interconnect. [17]

In my groups work for assignment 3 of CSE 260 last fall, we implemented the Aliev-Panfilov stencil method using MPI, OpenMP, and a hybrid approach. We found there to be some benefit to implementing the code with OpenMP or a hybrid approach, especially from an ease of programming perspective. A single MPI process with 8 OpenMP threads gave the same performance as 8 MPI processes (on a machine with 8 cores per node). A 2011 comparison of MPI and hybrid MPI+OpenMP models applied to partial differential equation simulations on the BlueGene/P supercomputer also claims the hybrid model to be superior. However, to get the best hybrid model performance, they must manually distribute the work among OpenMP threads via a graph partitioner and more complicated OpenMP syntax than simple pragmas, destroying much of the ease-of-use advantages of OpenMP. [4]

2.1.3 MPI + Stream

The Stream programming model decomposes an algorithm into units of computation called kernels. The kernels are joined by directed point-to-point links showing the data flow through the algorithm. Essentially, the model breaks an algorithm up into chunks that send information to each other according to the data flow graph for the algorithm. Kernels dont explicitly control their communication. Results are automatically discarded or sent off following computation, and when a kernel receives new data it automatically processes it. In this way, kernels are decoupled, independent units of computation. [3]
As mentioned in the hybrid programming introduction, MPI programming does not allow the user any control over hardware resource allocation, and this can have adverse effects on a program's performance or cause unwanted variability from run to run. In [3] the authors suggest combining the Stream model with MPI as a way to potentially relieve locality issues in MPI. Supposedly, by decomposing an MPI program into chunks, the system's resource allocator has a better idea of which processes are interdependent and can allocate them with better locality. The paper presents a result in which a hybrid implementation of a financial matching engine performs better than an MPI-only implementation.

2.1.4 MPI + UPC

Unified Parallel C (UPC) is a Partitioned Global Address Space (PGAS) model. It extends the C programming language to allow for distributed data to function as shared among processors. That is, UPC provides a global address space among distributed nodes and allows any core within any node to access any data in memory without explicitly interacting with the node where the data is physically stored. UPC also provides data locality information to the programmer and allows the programmer to tune data placement. Parallel program instances in UPC are called threads. [2]

In [2] the authors propose a hybridization of MPI with UPC, motivated by concerns related to data locality and shared memory size requirements when programming in MPI alone. For some algorithms, the structure of the problem decomposition requires prohibitively large amounts of shared memory as the problem size is scaled up. The authors cite the Greens function Monte Carlo algorithm for simulating atoms as an example. They argue that a hybrid implementation with UPC would allow for the solution of larger problem sizes than ever before.

They propose three different implementation models for the overall hybrid model: flat, nested-funneled, and nested-multiple. In the flat model, each process is both a UPC thread and an MPI process. Essentially the whole machine is virtualized to shared memory by UPC with an MPI structure running on it. Both the nested-funneled and nested-multiple models have MPI as an outer structure with several instances of UPC, called UPC groups, running within the MPI implementation. In the nested-funneled model, there is a single, master UPC thread which also acts as an MPI process. In the nested-multiple model, all UPC threads within a group also act as MPI processes. In either instance, the UPC threads may communicate like shared memory threads only within their group and inter-group communication relies on MPI calls made by any thread acting as an MPI process.

The authors present results for both a random access benchmark and an n-body simulation. They work on machines with four cores per node and implement hybrid models with UPC groups of 4, 8, and 16 cores. Strangely, after motivating their work with concerns about MPI the results only compare the hybrid models to a baseline UPC implementation, and the authors specify that their benchmarks were chosen for their poor scaling in UPC implementations. The results do show increased performance for the hybrid model over
the base UPC implementation, with the 4-core groups performing best since the memory
shared in UPC is still physically shared at that point.

This was the strongest hybrid programming paper outside of MPI+OpenMP papers
that I found in my reading, but it still has some decided shortcomings. Given their mo-
tivating arguments, it would be nice to see results comparing an MPI implementation
with the hybrid implementations. It would also be nice to see results for a few additional
benchmarks not chosen explicitly for their poor scaling in non-hybrid implementations.

2.1.5 Hybrid programming on distributed networks conclusions

In reading about hybrid programming models and implementations on distributed memory
computers, it seems that MPI+OpenMP is both the most established and the most viable
hybrid model. The vast majority of papers with the keyword hybrid programming concern
MPI+OpenMP hybridization, and those that do not tend to be less impressive in terms of
results and coherent model explanation. The non-OpenMP papers often raise legitimate
concerns about the limitations of MPI programming. Their models may well develop into
more impressive and viable solutions to those concerns, but for now MPI+OpenMP, for all
its sensitivity and fickleness, remains by far the most useful hybrid model.

2.2 Host+Device programming

In general, programming for a computing system with a host CPU and an accelerator device
like a GPU or an FPGA requires the use of hybrid programming since a common language
like C or Fortran is required to program the CPU and a second language like CUDA or
OpenCL is required to interact with the device. Intel designed the Xeon Phi coprocessors
so that they can be programmed with common CPU languages rather than introducing
another new language, but they are the exception. We will discuss programming on the
coprocessors more in another section.

2.2.1 GPGPU with CUDA

GPU devices offer the opportunity for impressive speedups on workloads with a high level
of data parallelism, and the CPU+GPU model is the most common hybrid device model at
the HPC level. NVIDIA provides the CUDA compiler, libraries, and software to aid general
purpose programmers in utilizing an NVIDIA GPUs computing power without having to
rephrase their problem as a graphics calculation. The CUDA programming model extends
the C/C++ model for the host CPU to allow the declaration of kernel functions and device
functions for execution on the GPU. Kernel functions may only be called (or launched)
from the host but execute on the device. Device functions may only be called from within
a kernel function on the device to execute on the device. Device functions may not contain
indirect or recursive function calls. [15]
The CUDA model requires the programmer to explicitly allocate device memory and specify memory transfers between the host and the device. A program running on a GPU executes in an SPMD way with many threads each running the same program. The programmer organizes the threads into abstract thread blocks, specifying the number and size of the blocks to be used at kernel launch. This organization usually consists of a 2D array of blocks, with each block a 3D array of threads. Each block receives a unique block index and each thread receives a unique thread index within the block. These indices are available to the programmer to distinguish between threads within the program. The hardware organizes threads into groups of 32, called warps, which run in lock-step with one another. This requires the programmer to avoid thread divergence within a warp as a performance consideration. [15]

In [6] the authors dissect the CUDA model for GPGPU programming, pointing out ways in which the model could be improved for increased ease of use and potentially better use of the GPU resources. For the most part, the authors seem eager for finer and more dynamic control over how the program runs on the device. They point to the fact that only the host can initiate data movement and only outside of kernel functions, that the dimension and number of blocks must be set at kernel launch, that blocks must be completely independent and run in an order opaque to the programmer, and that block states are completely erased upon completion as obstacles to more refined user control of the hardware. They also wish for the ability to communicate between blocks as they run without having to do a global memory synchronization and the ability to recursively call kernels from within kernels on the device.

The authors propose a new programming model called Persistent Threads (PT) to address some of these issues. They argue that hardware threads are actually active throughout the runtime of the program. Rather than hide that aspect from the programmer by having a blocks threads seem to terminate when the block completes, they want to make ongoing threads transparent to the programmer in a model where threads remain active and each run through a work queue as the program executes. They present results comparing their proposed models style with traditional CUDA programs for several benchmarks. I would have liked more details of how they got PT-style programs to run on a GPU, and while some of the results show potential for the model, they are not overwhelmingly convincing and could have been graphed more clearly. Interestingly, many of their complaints about the GPGPU model can be addressed by using MPI or OpenMP on a MIC-type device.

2.2.2 FPGA programming

FPGAs, standing for Field Programmable Gate arrays, are accelerator boards of reconfigurable logic gates that can be configured to execute specific functions needed by the programmer. FPGAs tend to be used at lower levels of computing than HPC, since their flexibility provides an environment that is good at everything but excels at nothing. [8] Similar to [6], in [7], Andrew, Niehaus, and Ashenden argue for the development of new,
more abstract programming models for FPGAs. In [7], the authors assert that to achieve a useful reconfiguration, a programmer must possess extensive knowledge of hardware design methods and tools, making programming for FPGAs unnecessarily time-consuming and difficult. They argue for extension of the operating system across the CPU and FPGA as a way to provide synchronization and control capabilities and higher-level, more abstract programming interface better suited to simple, effective FPGA programming.

2.2.3 Host+device programming conclusions

Current host+device programming models clearly exhibit some room for improvement. On the one hand, HPC users crave even finer, more explicit control of the architectural capabilities offered by massively parallel GPU-type devices. Lower level FPGA users could benefit from increased abstraction and less intellectual involvement with the architecture. These things sound different on the surface, but really both desires push for programming models that better fit the users capabilities and goals and provide a more seamless interaction between user and machine.

3 Automated Programming Models

In this paper, automated programming models refer to any programming model or infrastructure whose aim is to transfer some of the traditional programming responsibilities to the computer to be executed in an automated fashion. Such models also find their motivation in the challenges of programming for hybrid and heterogeneous architectures. Under the assumption that architectures will continue to be increasingly hybridized in the future, advocates of automated programming believe that effective hybrid programming demands an unsustainable level of skill from the programmer. To write a well-performing hybrid implementation a programmer must have extensive knowledge of multiple programming models, how those programming models fit together, the implementation architecture, and the problem being solved. Automated programming advocates believe its unreasonable to expect that level of expertise from the average programmer. Instead, they aim to reduce the burden on future programmers by developing models in which the computers themselves do more of the work of improving performance on hybrid architectures. This section discusses several approaches to this end goal.

3.1 SEEC Framework

One of the most interesting and successful automated programming models examined in this study is the SEEC (SElf-awarE Computing) model presented in [?]. The model introduces a decision engine rooted in control theory capable of dynamically assessing program performance at runtime and manipulating hardware resource allocation to meet performance goals. The model uses the Application Heartbeats API previously developed by
some of the same authors. This automation approach aims mainly to liberate the programmer from hardware resource allocation concerns, providing better and more consistent performance from run to run and machine to machine without requiring the programmer to rewrite any of the code. Thus, the SEEC framework addresses performance portability, a growing concern for programming on an increasingly diverse array of architectures.

To use the SEEC framework, the programmer must use the Heartbeats API to insert heartbeats into their code at critical moments and specify application performance goals in terms of the heartbeats. This could be in terms of a desired heartrate or a desired time between heartbeats. The decision engine monitors the heartbeats emitted by the running program and decides whether a speedup is needed to better meet the programmers specified goals. The framework assumes the involvement of a third party systems developer who is familiar with the hardware on which the application and the framework run. The systems developer provides the SEEC framework with a set of actions as well as their associated speedups and instructions to execute them. Then when the SEEC frameworks control system decides a speedup is necessary, it executes some combination of the available actions to achieve the speedup. These actions could be the allocation of cores on a multicore system, adjustment of core clock frequencies, manipulation of memory controllers, etc.

The paper provides a robust set of results, demonstrating the frameworks ability to predictably achieve a particular percentage of maximal performance (maximal performance having been determined manually), to optimize performance per watt of power consumption, and to maintain performance in the event of simulated hardware failures. They achieve their most impressive results in the study of maximizing performance per watt of power usage where they observe performance per watt for a video encoder acting on fifteen different videos. The study first determines the best possible performance per watt for each video if resources are statically allocated then runs the video encoder for each video with the SEEC framework. The SEEC implementation performs as well or better than the best static implementation for all videos.

Clearly, the framework has great potential, especially in the case of applications like a video encoder which are likely to have a highly variable workload from run to run and whose performance can easily be measured using heartbeats. (The video encoder emits a heartbeat after each frame encoded, and its performance goal is a desired frame-rate, easily measured as a heart rate.) Many other applications fail to fit so neatly into the heartbeat framework for specifying goals and measuring performance, but for applications suited to SEECs structure it works impressively well.

### 3.2 Translators: Mint and Physis

The last example set out to liberate the programmer from hardware resource allocation concerns using a decision engine framework, but the programmer was still responsible for all code generation. Source-to-source translators instead try to reduce the amount of hardware-specialized code a programmer has to write. Both the Mint and Physis translators
examined here try to reduce CUDA code generation to pragmas and annotation for the developer, with the translator taking care of the actual CUDA source code generation.

The Mint translator described in [18] is aimed at non-expert programmers and designed for stencil method computations on GPUs. Reducing the application space to stencil methods allows the translator to apply more sophisticated and effective optimizations and achieve better end performance. The use of stencil methods in a wide array of scientific applications ensures that the translator is still broadly useful. The Mint model as presented in [cite mint] focuses on a single host + accelerator computing system.

The Mint model has the programmer write a traditional C program for the application and then annotate the program with just five simple pragmas. The pragmas are inspired by those in OpenMP with the aim of making it easy for OpenMP users to adapt quickly to Mint. Mint focuses on parallelizing the nested for loops inherent to stencil computations and running them on the GPU. After an initial translation into legal CUDA code, Mint applies several optimizations to the generated kernel. The end results presented in [18] show Mints optimizations to be highly effective, with optimized Mint-generated code achieving roughly 80% of the performance of carefully hand-optimized CUDA code for each of five common stencil kernels tested. For more advanced users, the ease of using Mint probably will not outweigh the 20% loss of performance over hand optimization. However, the less advanced users Mint targets would likely come closer to the performance of Mints initial translation codes if coding by hand. With that assumption, those users would likely see a 2x improvement by using Mint.

The Physis translator presented in [11] builds on the ideas presented in the Mint paper, departing from Mint only its focus on scalability to multi-GPU systems. It still focuses on stencil method computations and receives annotated code for translation to CUDA. Since Physis seeks to translate code for distributed, multi-node computers, the programmer provides annotated C code with MPI calls rather than annotated sequential C. Physis annotation system is not quite as simple and compact as Mints. The authors make a rough measure of the added burden to the programmer compared to an MPI implementation by comparing code lengths of MPI programs and find the lengths to be comparable in two of three tests and not unreasonable, barring a few justifications, in the third. Their results show the optimizations applied by Physis to be effective and the generated code to be fairly scalable up to 100-200 GPUs. They do not compare the translators code to hand-tuned CUDA directly like in the Mint paper, but they calculate based on performance numbers an achievement of roughly 60% of the performance of hand-tuned codes. Once again, this is far too large a performance drop to appeal to expert users, but to offer non-experts even 60% of the performance of hand-tuned CUDA on hundreds on GPUs is impressive and potentially quite useful.
3.3 Dynamic Feedback for Compiler Optimizations

In the course of my reading I came across a 1999 paper by Pedro Diniz and Martin Rinard advocating for the use of dynamic feedback in a parallelizing compiler [19]. Their compiler uses an automatic commutativity analysis, so no pragmas or annotations are required, to determine whether operations commute (roughly, whether operations are independent) and thus whether they can be executed in parallel. The compiler focuses only on parallelizing loops and uses lock-based synchronization to protect critical regions of the code. The dynamic feedback comes into the process of selecting the best synchronization optimization from among three options. The compiler generates three different versions of a program from the same source code, with each version using a different synchronization optimization. As the program runs, it goes through what they call sampling and production phases. In a sampling phase, each version of the program run for short times to determine which performs the best in the current environment. In a production phase, the best version just determined by sampling runs for a longer time. For each of three benchmarks, they run the code with each of the three possible optimizations and then with the dynamic feedback framework. Although using the dynamic feedback model introduces some overhead, the dynamic feedback runs are extremely close to the best-model runs for all three benchmarks.

This paper presents results that are interesting in their own right in the study of automated programming models and performance concerns. It also shows that the ideas of automated programming and dynamic feedback have been around for several years. The introduction to this paper also cites increasingly heterogeneous architectures as its motivation, especially in the context of programming for upcoming mobile platforms, and mentions a general trend toward adaptive computing. The existence of a continued trend toward adaptive computing in practice is debatable, but the mention of heterogeneity among mobile computing platforms points in an interesting direction for future exploration. Are smartphone architectures overly diverse in 2013? And do any of the most popular apps that need to run on all of them use dynamic feedback techniques?

3.4 Other ideas worth mentioning

Several other automation ideas came up in the course of this study that merit mention. Two papers co-authored by Dean Tullsen and other researchers from UCSD explore the idea of automated parallelization from a more hardware-oriented perspective. In Software Data-Triggered Threads, the authors discuss a model that generates parallel threads dynamically as the result of changes to memory contents rather than according to control flow. [10] The previously proposed Data-Triggered Threads (DTTs) model purports to eliminate some unnecessary computations by only generating threads when actually needed as well as to expose parallelism earlier in program runtime by allowing computations to begin as soon as their input data becomes available. Previous proposals of the model required the development of hardware support unavailable on current architectures, but this paper
successfully implements a software-based version and shows increased performance for some PARSEC benchmarks when combining DDTs with more traditional parallelism.

In Runtime Parallelization of Legacy Code on a Transactional Memory System, the authors propose a way of automatically parallelizing serial code at runtime by leveraging transactional memory support in hardware. [9] The aim is to allow legacy code to run more efficiently on newer parallel architectures without anyone having to revisit the code and hand parallelize it. Transactional memory allows for concurrent memory accesses across threads when it presents no real conflict as opposed to the more traditional model of locking shared memory during each threads access, effectively serializing memory accesses. [20] Transactional memory hardware is speculative in the sense that it is not widely available in current architectures, but it has been successfully implemented. The BlueGene/Q supercomputer has transactional memory support, and IBM released a commercial server with transactional memory in 2012 [21]. In [9] the authors only simulate their model to assess potential speedups, finding the potential for 34-36% speedups on the NAS and SPEC2000 benchmarks on a two-core system.

3.5 Automated programming conclusions

Automated programming models form an interesting subset of ongoing research on improving the programming experience and performance portability on hybrid and heterogeneous architectures. The above models all show considerable promise, but also demonstrate that it remains difficult to match hand-tuned performance. As such, these models are unlikely to receive attention from performance-focused expert users. These models are not really designed for expert programmers, though. Most of the models state the desire to appeal to non-expert programmers, but (with the exception of the prescient mention of mobile platforms and the ensuing droves of current app developers) it remains unclear whether there are enough non-expert programmers working on complicated architectures to really need automated models yet.

4 Hybrid and Automated models on MIC

Intel has been working on its Many Integrated Core (MIC) architecture for several years, with earlier experiments aimed more directly at graphics processing to compete with GPUs than at general purpose computation [13]. The Xeon Phi coprocessors, which began shipping to select partners in late 2012 and were released more widely in early 2013 mark the first commercial implementation of the MIC architecture [22]. The coprocessors are designed both to offer performance competitive with GPUs at the HPC level and to offer improved programmability and functionality over GPUs. To this end, the coprocessors are built on an x86 architecture, allowing them to run MPI and OpenMP programs, and have their own Linux operating system [23].
4.1 Hybrid Programming on MIC

As mentioned above, MIC supports programming in both MPI and OpenMP, the most common parallel programming models. As of right now, the coprocessors can run code natively using their own operating systems, but they still need to be connected to a CPU to be fully operational. At 8GB, the shared memory on the coprocessor is relatively too low for many applications to run natively, so most programmers will still program for the MIC in a host+device style with a C/C++ or Fortran programming running on the host CPU and heavy compute loads running on the coprocessor. In this model, the Intel compiler can identify certain compute-heavy operations in the code and automatically offload them to the coprocessor. This is discussed more below. However, for maximum control and performance, programmers will want to use Intels pragmas to specify when to offload data and computations to the coprocessor [24]. Thus, programming on MIC resembles a combination of the distributed-CPU and CUDA device programming styles, with MPI or OpenMP interleaved with device-control pragmas. CUDA can be used within an MPI program. The difference on the Phi coprocessors is just that MPI or OpenMP can run in the device portions of the code as well [25] [24].

4.2 Automated Programming on MIC

Intel provides several modes of automation to help make programming on the Xeon Phi coprocessors as easy as they claim it to be. They also offer an impressive range of user customization options to appeal to the widest possible range of programming abilities and styles. In an introductory guide to programming the MIC coprocessors, the author asserts that, The true power of Intel Xeon Phi is currently realized through the capabilities of Intels compilers. [cite DD intro] That article cites the compilers ability to compile as single source file to run on the host, natively on the coprocessor, or hybridly on both as the most valuable resource from a program-design perspective.

Auto-vectorization and automatic computation offloading (referred to as AO for Automatic Offload) are among the compilers other automated skills. Simple preliminary tests show that efficient vectorization is essential to reaching peak performance on the coprocessors [25]. Thus, effective compiler auto-vectorization is essential for easy, effective coprocessor programming. More advanced users may still bypass the auto-vectorization in favor of using vector intrinsics if they like. To use the AO functionality of the compiler, a programmer need only add a single support function call or set a single environment variable [26]. The compiler and library then use information about current coprocessor states to decide whether to offload any computation and how many coprocessors to use. Even within the AO environment, users can tune the exact work division, and AO can be used in conjunction with pragma offloading if desired. Currently, the AO capabilities are limited to certain MKL kernels, namely *GEMM, *SYMM, *TRMM, *TRSM, LU, QR, and Cholesky [26]. Here, * stands for either D or S in the various matrix operations.
Lastly, in an attempt to appeal even more to non-expert users, Intel has announced an array of software partnerships all aiming to make coprocessor usage as simple as possible. ScaleMP has announced software support that virtualizes the host and coprocessor memory spaces and eliminates any need for explicit memory transfers [27]. Other partnerships include one with Bright Computing promising to save the user the "tedious challenge" of configuring their new coprocessor. Instead, the coprocessor should "work out of the box" in conjunction with their software. Univas Grid Engine software similarly promises to provide users with maximal coprocessor performance and minimal work by automatically managing compute resources as best as possible. These and other partnerships are announced in Intel’s whitepaper [28] among other places [29]. For now, it remains difficult to assess how well these software abstractions work. I never found anything about them coming from a source other than Intel or the software developer. Still, their very existence makes it clear that Intel intends the coprocessor product to reach a wider audience of users than just HPC programmers.

4.3 Programming models for MIC conclusions

All of the advertisements, reviews, and vague early results papers give the impression that the coprocessors offer excellent performance and extreme ease of programming [14] [30]. However, all of these sources appear tightly controlled by Intel, and not many papers offering raw performance numbers exist yet. Based on the experience of our class, it may not be difficult to get a program to run on MIC at all, but it can be awfully difficult to get it to perform well.

Perhaps the most interesting aspects of the new architecture from the perspective of this study are its explicit effort to be both high performance and easy to use as well as Intels marketing toward a wider audience than just the HPC users they are competing for against GPGPUs. As stated in a Computer World UK article, By introducing a widely available parallel accelerator solution with lower barriers to application migration, Intel can accelerate the adoption of explicit parallel coprocessors. [31]. Wide adoption of co-processor computing could necessitate and accelerate further development of automated programming models.

5 Overall Conclusions

This project aimed to explore the range of hybrid and automated programming models both currently available and being proposed in research settings with an eye toward these models implications for future programmers responsibilities. In the study of hybrid models, several models were found in the literature and explored, but it quickly became clear that MPI + OpenMP is the most studied, established, and viable hybrid model. Even in the studies of this most viable model, it seems impossible to make any generic statements about when hybridization works better than a flat MPI implementation. The hybrid models
performance and usefulness depends sensitively on the application, the implementation hardware, and the skill of the programmer in manipulating the two models and their interaction. Many studies show the hybrid model to be beneficial in some cases, but all state that knowing when to use the hybrid model and realizing its benefits is nontrivial. Hence, truly effective and successful use of this model remains an area for more expert programmers seeking absolute maximum performance.

The study of automated programming models led to the discovery of some interesting and exciting research. Getting computers to perform tasks typically reserved for human minds and getting the computers to do those tasks well always makes for fun reading. For all of their practical motivations, automated models remain more of a fun research area than a set of practical strategies. None of these models can best hand-tuned code in terms of raw performance, so the HPC community and anyone else in search of maximum performance is unlikely to adopt these models. The everyman programmer inspiring the models does not yet have the hardware at his disposal to really need them, but considering the general trend of computing to make increasingly powerful computers available at an increasingly large consumer scale, the average programmer could well find herself with an accelerator board in every laptop and in need of an automated model.

Intel’s wide array of software partnerships, all promising to make the new coprocessor exceedingly easy to use, indeed indicate an effort to push coprocessor capabilities and usage not only into the HPC domain but also further into the mainstream. They designed the entire coprocessor architecture to compete with GPUs in terms of performance but also to be maximally easy to program, using x86 instruction cores capable of running the most common and established serial and parallel programming models. If the coprocessors are successful, this could represent an important shift in perspective. Rather than always working to come up with better programming models to deal with new parallel architectures, the coprocessor design suggests considering how to amend those architectures to fit the programming models.

References


6 Self-Evaluation

(1) Team: Helen Parks
(2) Estimated time devoted after progress report:
Writeup: 12 hours
Planning (and reading): 21 hours
Total time for project:
Writeup: 16 hours
Planning and reading: 46 hours
(3) After refocusing and continuing to read, I decided to rename the project again from Self-Aware Computing to Automated computing because I felt that what really interested me were models in which some part of the traditional programming process was transferred to the machine to be done in an automated fashion.

After a closer reading of the 1997 paper I mentioned in my talk on Wednesday, I realized that I had gotten lost in the language and misread what the paper was about. I looked up both authors research interests and found the 1999 paper mentioned in this writeup which provides the experimental results for the 1997 papers discussion as well as a much clearer explanation of the whole model.

I'm still not completely pleased with how the project turned out. I think despite the initial refocusing at the progress report, its still too broad. I probably spent too much time searching and reading and not enough time analyzing. In retrospect, exploring a single hybrid programming model (MPI+OpenMP being the obvious candidate) and possibly running tests with it on MIC might have made for a better project. Still, my initial goal in taking this class and doing any sort of project was just to make the time to think and learn more about parallel computing and whats out there in terms of programming models and architectures, and I certainly achieved that much.