12-2018

Automatic Performance Optimization on Heterogeneous Computer Systems using Manycore Coprocessors

Chenggang Lai
University of Arkansas, Fayetteville

Automatic Performance Optimization on Heterogeneous Computer Systems using Manycore Coprocessors

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy in Computer Engineering

by

Chenggang Lai
Shandong University
Bachelor of Science in Electronic Engineering, 2012
University of Arkansas
Master of Science in Computer Engineering, 2014

December 2018
University of Arkansas

This dissertation is approved for recommendation to the Graduate Council

Miaoqing Huang, Ph.D.
Dissertation Director

John Gauch, Ph.D.
Committee Member

Merwin Gordon Beavers, Ph.D.
Committee Member

Xuan Shi, Ph.D
Committee Member

Abstract

Emerging computer architectures and advanced computing technologies, such as Intel's Many Integrated Core (MIC) Architecture and graphics processing units (GPU), provide a promising solution to employ parallelism for achieving high performance, scalability and low power consumption. As a result, accelerators have become a crucial part in developing supercomputers. Accelerators usually equip with different types of cores and memory. It will compel application developers to reach challenging performance goals. The added complexity has led to the development of task-based runtime systems, which allow complex computations to be expressed as task graphs, and rely on scheduling algorithms to perform load balancing between all resources of the platforms. Developing good scheduling algorithms, even on a single node, and analyzing them can thus have a very high impact on the performance of current HPC systems. Load balancing strategies, at different levels, will be critical to obtain an effective usage of the heterogeneous hardware and to reduce the impact of communication on energy and performance. Implementing efficient load balancing algorithms, able to manage heterogeneous hardware, can be a challenging task, especially when a parallel programming model for distributed memory architecture.

In this paper, we presents several novel runtime approaches to determine the optimal data and task partition on heterogeneous platforms, targeting the Intel Xeon Phi accelerated heterogeneous systems.

Acknowledgements

I would like to extend my thanks to my thesis committee, especially my advisor Dr. Miaoqing Huang for providing the tools necessary for my research.

Contents

List of Figures

List of Tables

Terms and Definitions

MIC Many Integrated Core architecture. A manycore processor architecture by Intel.

CPU Central Processing Unit.

GPU Graphic Processing Unit.

MPI Message-passing Interface. A library for parallel programming on computer clusters.

OpenMP Open Multi-Processing. A library that supports multi-platform shared memory multi-processing programming.

CUDA Compute Unified Device Architecture. A parallel programming language on GPU.

List of Published Papers

This dissertation is based on the following four papers:

Chapter 3 **Comparison of Parallel Programming Models on Intel MIC Computer Cluster**
Chenggang Lai, Zhijun Hao, Miaoqing Huang, Xuan Shi, and Haihang You
in Proceedings of Fourth International Workshop on Accelerators and Hybrid Exascale Systems (**AsHES**) as part of IPDPS, May 2014.

Chapter 4 **Towards Optimal Task Distribution on Computer Clusters with Intel MIC Coprocessors**
Chenggang Lai, Miaoqing Huang, and Genlang Chen
in Proceedings of 2015 International Conferences on High Performance Computing and Communications (**HPCC**), August 2015.

Chapter 5 **Performance Optimization on Intel Xeon Phi Through Load Balancing**
Chenggang Lai, Xuan Shi, and Miaoqing Huang
in Proceedings of 24th International Conference on Parallel and Distributed Processing Techniques and Applications (**PDPTA**), July 2018.

Chapter 6 **Performance Improvement on Heterogeneous Platforms: A Machine Learning Based Approach**
Chenggang Lai,Yirong Chen, Xuan Shi, Miaoqing Huang, Genlang Chen
International Conference on Computational Science and Computational Intelligence, Dec. 2018.

Chapter 1

Introduction

High-performance computing is critical to process large volumes of data in the big data area. With the advancement of technologies, high-resolution data become available. For example, satellites can generate high-quality images with a resolution less than 0.5 meter. However, high-resolution data bring lots of challenges and complexities. Traditional desktop-based software have become inefficient and impossible to process large-scale data. While the computer provides a large memory for data processing, some software has limitations to usage of computer memory.

People has to partition the source data into many parts for analyzing. Given the growing quantity of available data and finite resources to analyze them, it is expected that data can be exploited effectively for timely delivery of accurate information and for knowledge discovery. High performance computing (HPC) can allow people to solve complicated big data problems in various regions, such as engineering, science and business. Many top supercomputers are hybrid systems including both multicore CPUs and accelerators. Performance optimization mechanisms are critical for large-scale applications to achieve the best performance on these hybrid systems. These techniques include optimal workload distribution between the host processors and the accelerators, overlapping computation and communication to reduce the communication overhead, among others. As a part of high-performance computing, accelerators are becoming popular. Compared with traditional CPUs, accelerators can provide an orders-of-magnitude improvement in performance. Many computer architectures have been implemented. They provide good platforms to employ parallelism for achieving performance and scalability.

Emerging computer architectures and advanced computing technologies, such as Intel's Many Integrated Core (MIC) Architecture [5] (brand-named Xeon Phi) and graphics processing units (GPUs) [28], provide a promising solution to employ parallelism for achieving high performance,

scalability and low power consumption. The combination of host processors and accelerators[1], such as GPUs and Intel Xeon Phi coprocessors, has been applied in many cases to achieve orders of magnitude performance improvement.

With the development of graphic processing unit (GPU), it is becoming normal to use GPU as a modified form of stream processor for general purposes. GPU can get several orders of magnitude higher performance than CPU when processing massive vector operations. Therefore, high performance computers that are based on GPUs become a significant role in large scale modeling. GPUs are typically used as accelerators in high-performance computer clusters. However, it should be clear now that GPUs are designed as numeric computing engines, and they will not perform well on some tasks on which CPUs are designed to perform well. If different threads in a warp(parallel units) need to do different things, all threads will compute a logical predicate and several predicated instructions. This is called warp divergence. All threads execute conditional branches, so execution cost is sum of both branches. Warp divergence can lead to a big loss of parallel efficiency. Another problem are arising in learning and development on domain-specific-languages (DSLs) of GPU. For example, CUDA, implemented by NVIDIA, is a DSL for parallel programming on GPU. A $40\times$ speedup can be achieved in comparison to CPU solutions, but a $11\times$ learning curve is needed on CUDA study.

Intel MIC provides another option for augmenting the computer clusters for high performance and low power consumption. The MIC has demonstrated the high performance, the scalability, and the high memory bandwidth. The current Intel MIC architecture has up to 61 processing cores. These cores are connected through a high-speed ring bus. Because every core is a low-weight classic processor, the MIC can support traditional parallel programming models, such as OpenMP and MPI. The Xeon Phi coprocessors typically co-exist with multicore CPUs, such as Intel Xeon processors, in a heterogeneous computer platform. One of classic programming models on such multicore/manycore heterogeneous architectures is to use host processors to manage the execution context while the computation is offloaded to the accelerators. Effectively leverag-

[1]In this work, we use accelerator and coprocessor interchangeably.

ing such platforms not only achieves high performance and good scalability, but also increases the energy efficiency. Although it is easy to implement application on MIC, traditional parallel programming models has their own bottlenecks.

The heterogeneous platform provides the potential for high performance and energy efficiency, but the classic offload model on GPU or MIC platforms leaves the host processors unutilized. This means this approach does not take advantage of the computing capacity of the host processors and is likely to give away too much performance potential of the whole system. Asynchronous data transfer and computation have been proposed as a solution to decrease the host-device[2] communication cost and to increase the utilization of host processors [19, 34]. In asynchronous mode, the host processor sends workload to the accelerator. Then the host processor continues the execution of other workload until it is requested to wait for a kernel running on the accelerator to finish. In this case, both host processors and accelerators can work in parallel to undertake a computation task.

Ideally, the scheduler should partition work between host processors and accelerators automatically and efficiently without any input from the application developer. However, it is hard to determine the right data partition and task parallelism on heterogeneous platforms given a new application. There are some evidences showing that choosing the right configurations, i.e., the number of for-loops needed to be parallelized and the number of concurrent tasks in for-loops, has a significant impact on the application's performance on Xeon Phi coprocessors [23, 14, 33]. However, exhaustive manual search would be ineffective to find the optimal workload partition between host CPU and the accelerator and the optimal task distribution on accelerators, because the range of the possible configurations is huge. Therefore, we need to design a technique that is capable of automatically determining the optimal configurations for any application in an efficient manner. This thesis focuses on asynchronous calculation and efficiently utilizing all available resources to achieve performance improvement, targeting the Xeon Phi coprocessor. Besides, a novel runtime approaches to determining the optimal data and task partition automatically would be proposed in

[2]host: host processor; device: accelerator/coprocessor.

this work.

The thesis consists of 7 chapters. It is organized based on the specific accelerators that are used to accelerate some applications. The first two chapters give an introduction and related work to high-performance computing and accelerators. A detailed study of parallel programming models are discussed in Chapter 3. Chapter 4 and Chapter 5 demonstrate performance optimization through load balancing by using different methods. A novel runtime approach to determining the optimal data and task partition automatically is discussed in Chapter 6. Chapter 7 concludes the whole thesis.

Chapter 2

Background

2.1 Parallel Programming

Traditional software code, such as C and C++, is written for sequential computation. It is normal for people to break a problem and solve it step by step. Only one instruction is executed at a particular moment and those instructions are executed in a sequence [8]. Figure 2.1 shows a simple serial process.

Nowadays, parallelism is becoming ubiquitous, and parallel programming is becoming mainstream in the programming world. Parallelism at multiple levels is the driving force of architecture design. There are two fundamental types of parallelism in applications: Task parallelism and Data parallelism Task parallelism arises when there are many tasks or functions that can be operated independently and largely in parallel. Task parallelism focuses on distributing functions across multiple cores. Data parallelism arises when there are many data items that can be operated on at the same time. Data parallelism focuses on distributing the data across multiple cores.

In parallel computing, multiple pieces of data will be processed simultaneously using different processing resources. It means that the problem will be partitioned to several parts and these parts can be executed concurrently. Figure 2.2 demonstrates a parallel processing scenario. When using parallel model to break down a problem, it is necessary to consider the accuracy of result. Sometimes, processors need to share results among each other, therefore introducing communications among processors.

2.1.1 MPI

Message Passing is a parallel programming model where communication between processes is done by interchanging messages. This is a natural mode for a distributed memory system,

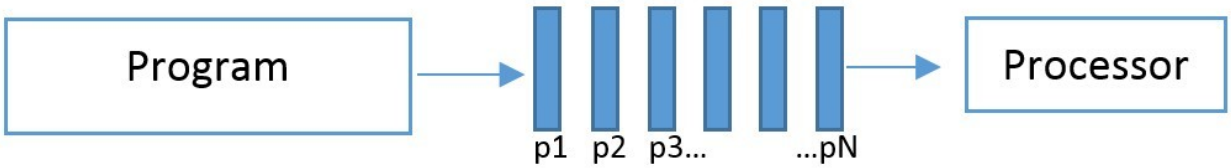


Figure 2.1: Serial Processing.

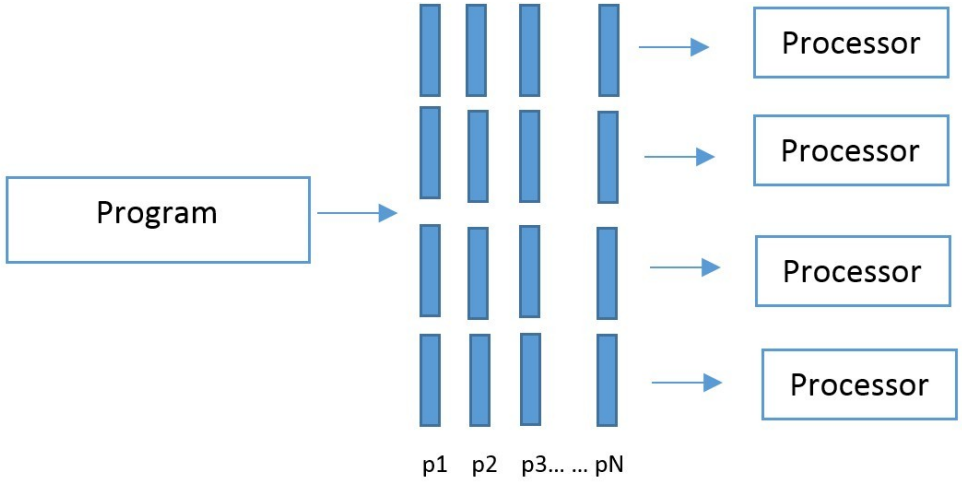


Figure 2.2: Parallel Processing.

where communication cannot be achieved by sharing variables. Message Passing Interface (MPI) processes executed in parallel have separate memory address spaces. The same program runs on all processes (Single Program Multiple Data, or SPMD). This is no restriction compared to the more general MPMD (Multiple Program Multiple Data) model as all processes taking part in a parallel calculation can be distinguished by a unique identifier.

The program is written in a sequential language like Fortran, C or C++. Data exchange, i.e., sending and receiving of messages, is done via calls to an appropriate library. Communication occurs when part of the address space of one process is copied into the address space of another process. This operation is cooperative and occurs only when the first process executes a send operation and the second process executes a receive operation.

The workload partitioning and task mapping have to be done by the programmer. The programmer need to know how to utilize hardware resource efficiently for performance and scalability to unveil any problems connected to parallelization. Besides, some issues still need to be addressed,

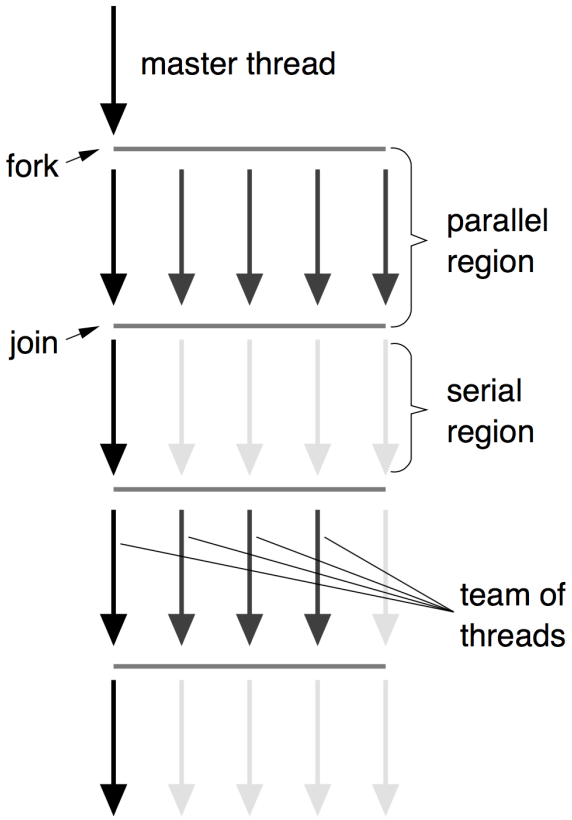


Figure 2.3: Parallel Processing.

such as serial execution (Amdahls Law), load imbalance, unnecessary synchronization, and other effects that impact all parallel performance.

In summary, MPI is well suited for applications where portability, both in space (across different systems existing now) and in time (across generations of computers), is important. MPI is also an excellent choice for task-parallel computations and for applications where the data structures are dynamic.

2.1.2 OpenMP

OpenMP is a shared memory application programming interface (API) [15] whose aim is to ease shared memory parallel programming. Shared memory opens the possibility to have immediate access to all data from all processors without explicit communication. Unfortunately, POSIX

threads are not a comfortable parallel programming model for most scientific software, which is typically loop-centric. For this reason, a joint effort was made by compiler vendors to establish a standard in this field, called OpenMP.

The OpenMP multithreading interface is specifically designed to support high performance computing (HPC) programs. It is also portable across shared memory architectures. OpenMP is a set of directives to a compiler. When a compiler recognizes OpenMP directives, then the directives are interpreted to give direction on how to create parallel tasks in order to speed execution of a program through parallelism.

In any OpenMP program, a single thread, the master thread, runs immediately after startup. Truly parallel execution happens inside parallel regions, of which an arbitrary number can exist in a program. Between two parallel regions, no thread except the master thread executes any code. This is also called the fork-join model, as shown in Figure 2.3.

Inside a parallel region, a team of threads executes instruction streams concurrently. The number of threads in a team may vary among parallel regions [46, 47]. For example, omp_get_thread_num() can fork a specified number of threads and system can allocate these threads to a task. OpenMP can assign the number of threads on environment variable, or can use function of OpenMP to assign threads' number at the code. Each thread has an ID. Every ID is integer type and the ID of master thread is 0. These threads can execute concurrently. For example, there is a "for" loop for addition operations (sum[i]=a[i]+b[i]). Each thread can do a part of addition at the same time.Working-sharing constructs can allocate part of task to different threads so that they can execute the work concurrently. Each thread cannot be disturbed by others. So if the code is not independent, there will be a problem when using working-sharing constructs. When the execution of parallelized code is done, these threads join back to master thread. And the master thread will continue executing the rest of program until it meets next parallel section or the end of program.

2.2 CUDA

CUDA stands for Compute Unified Device Architecture. It is a specific parallel programming language implemented by NVIDIA. CUDA is a parallel programming model and computing platform. When using CUDA for programming, the developers can access the memory of computational elements, such as global memory, shared memory and local memory. Like OpenCL, CUDA has its own application programming interfaces. This approach is known as Stream Processing. GPU has a different architecture than CPU. It contains hundreds to thousands of processing cores for parallel processing. CUDA supports both C/C++ and Fortran. CUDA also supports other computing interfaces such as OpenCL and OpenGL. CUDA provides two levels of API, low-level API and high-level API. Usually it is enough for programmers to only use high-level API to allocate memory of GPU and launch a kernel to GPU. When you need more specific function to your program, you need to use low-level API to allocate and run your program. Basically, CUDA supports most of GPUs provided by NVIDIA, such as GeForce, Quadro and Tesla series. CUDA is supported on multiple operating systems, such as Windows and Linux system.

CUDA programming is especially well-suited to address problems that can be expressed as data parallel computations. Many applications that process large data sets can use a data-parallel model to speed up the computations. Data-parallel processing maps data elements to parallel threads.

2.3 Parallel Computing Hardware

2.3.1 Multi-core Processor

A multi-core processor is a processor with multiple independent processing cores. With the development of computer architecture, central processing unit has changed a lot, such as design technology and the implementation of CPU. However, the basic operation keeps much the same. Most computers have multi-core processors, such as 8-core CPU and 16-core CPU. It is better for

multi-core processor to use different cores to deal with different processing. Multi-core processor can run faster and bring a better performance to users. Programmers can use parallel library such as OpenMP and MPI to take full advantage of all cores and get a better performance. Of course, not all of computing system only depend on multi-core processor. A number of hardware accelerators are provided, such as GPU, FPGA and MIC in the distributed architecture, but multi-core processor still plays an important role.

2.3.2 GPU

GPU architecture has been developed for many years and different companies have gone through many generations. For example, NVIDIA generates different architecture of GPU, such as G80→GT200→Fermi→Kepler→Pascal. With the development of graphic processing unit, it is becoming normal to use GPU as a modified form of stream processor for general purposes. This concept changes GPU from a modern graphics accelerators into a general purpose accelerator.

Therefore, high performance computers that are based on GPUs become a significant role in large scale modeling [49]. Nowadays, the two major GPU designers are NVIDIA and AMD. NVIDIA develops CUDA to support GPU programming.

OpenCL [58] is also supported by NVIDIA's GPU. OpenCL is designed to work for architectures of multiple types, such as CPUs, GPU and DSP. Both programming languages allow a program to launch a kernel on GPU and run the parallel program on its stream processors. And programmer can make a decision about which part is running on GPU or CPU. It can take advantage of the ability of GPU and CPU to perform their own appropriate work.

2.3.3 MIC

Intel demonstrated a new hardware architecture called Many Integrated Core(MIC) [4] as accelerators for high-performance computing domain. It provides another option for augmenting the computer clusters for high performance and low power consumption [42, 57, 26]. The MIC

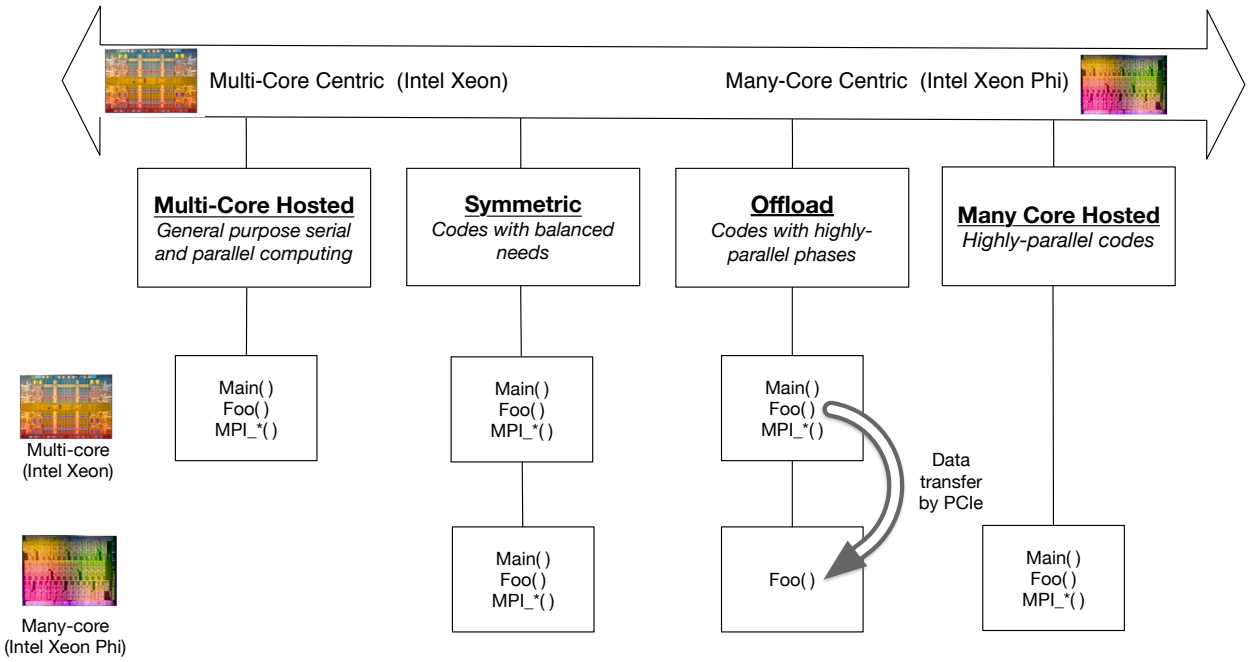


Figure 2.4: Spectrum of Programming Models.

has demonstrated the high performance, the scalability, and the high memory bandwidth. An evaluation of the scalability on the Intel MIC based graph algorithms shows that MIC can be programmed easily and scaled gracefully on graph algorithm [54].

The commercially available Intel coprocessor based on the MIC architecture is Xeon Phi. Xeon Phi contains up to 61 scalar processing cores with vector processing units. Further, each core can execute four threads in parallel. The communications between the cores can be realized through the shared memory programming models, e.g. OpenMP. In addition, each core can run MPI to realize communication. Direct communication between MIC processors across different nodes is also supported through MPI. And Intel coprocessors supports several programming models to meet application needs, as shown in Figure 2.4.

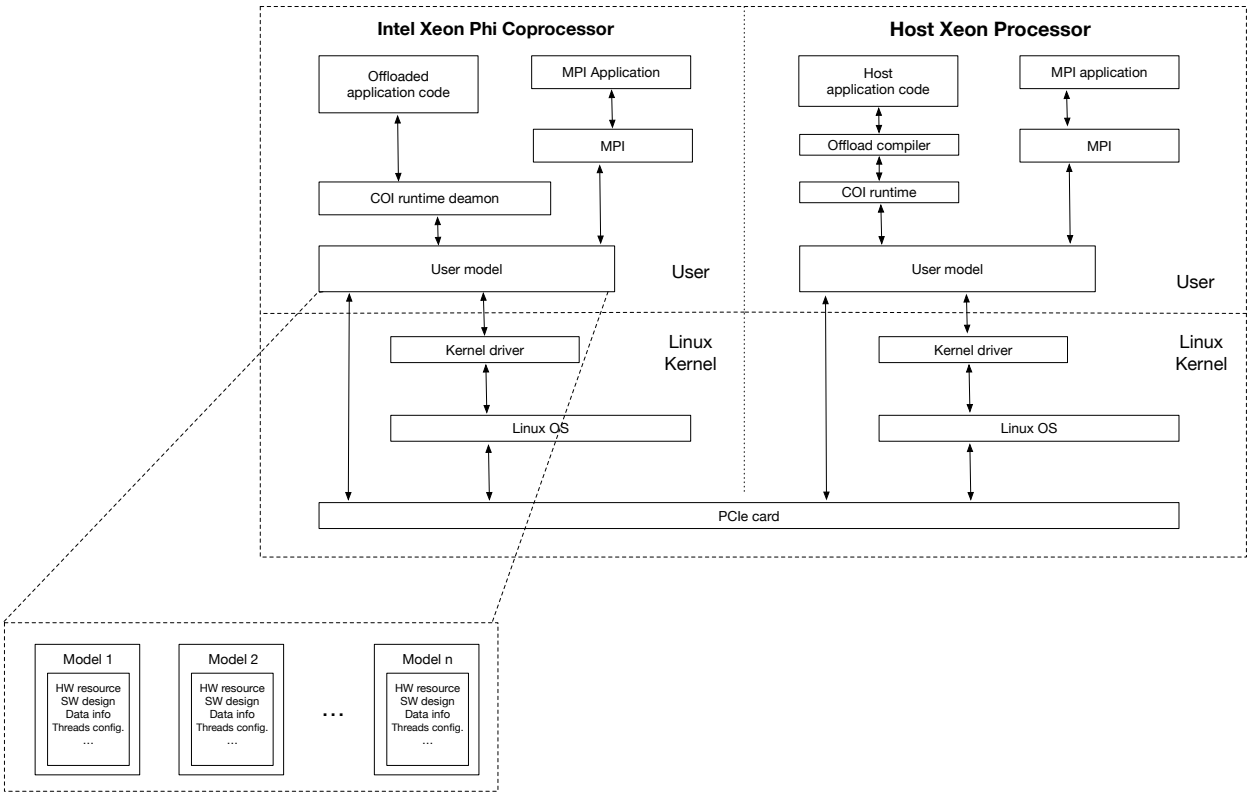


Figure 2.5: The software architecture of Intel Xeon Phi coprocessor.

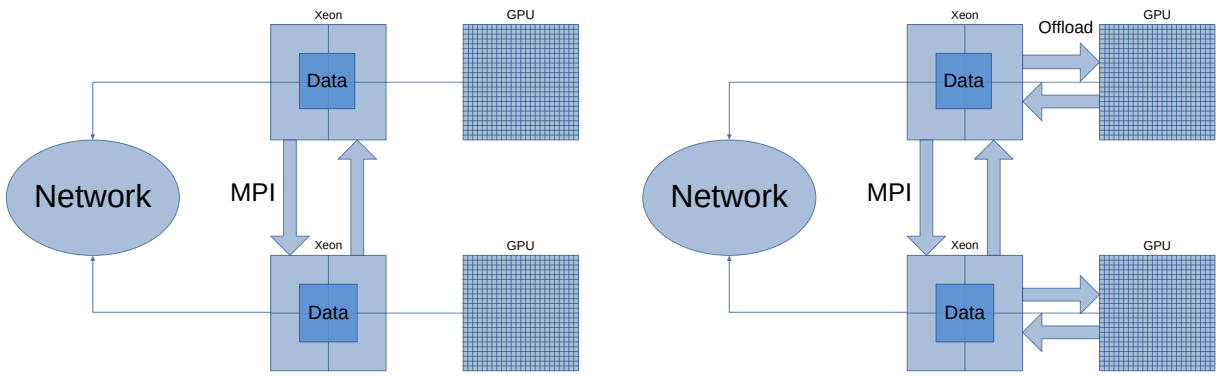
2.4 Programming models

2.4.1 MPI+CPU

MPI-based parallel implementation is shown in Figure 2.6(a). The Intel Xeon CPU works for data processing. The resource used on CPU is a single-thread process. Each MPI process runs on a single CPU. If m MPI processes are established in the parallel application, m CPU processors are used. A sample code is shown in Algorithm 1.

2.4.2 MPI+GPU

GPU-based parallel implementation on high performance system is shown in Figure 2.6(b). Each MPI process runs on Intel Xeon CPU. Every Xeon CPU offloads data to one GPU processor. If m MPI processes are used in application, m CPU processors and m GPU processors are allocated.



(a) CPU parallel implementation. (b) GPU and CPU parallel implementation.

Figure 2.6: MPI parallel implementation on Keeneland

ALGORITHM 1: MPI+CPU Programming model

Function Transition(*Array A*, *Array B*)

```
MPI__Init();
MPI__Comm__rank();
MPI__Comm__size();
for i = 0 → k − 1 do
    A[i] ← B[i];
MPI__Finalize();
```

The host CPU works for the MPI communication and collecting results. The GPU is responsible for data processing. A sample code is shown in Algorithm 2.

2.4.3 MIC Native mode

The native model runs the calculation procedures entirely on an Intel Xeon Phi coprocessor. The Intel Xeon Phi coprocessor has its own operation system, such as Linux, IP address, a high-performance network connection and memory domain. The coprocessor is an x86-based SMP-on-a-chip with over many cores. Some MICs have 59 cores, and others may have more than 60 cores. Each MIC core has multiple hardware threads, and 512-bit SIMD instructions. The Intel Xeon Phi looks like an independent compute node. Users can log into any Xeon Phi installed in production system by a terminal window and compile programs with the mmic switch to target launch and

ALGORITHM 2: MPI+GPU Programming model

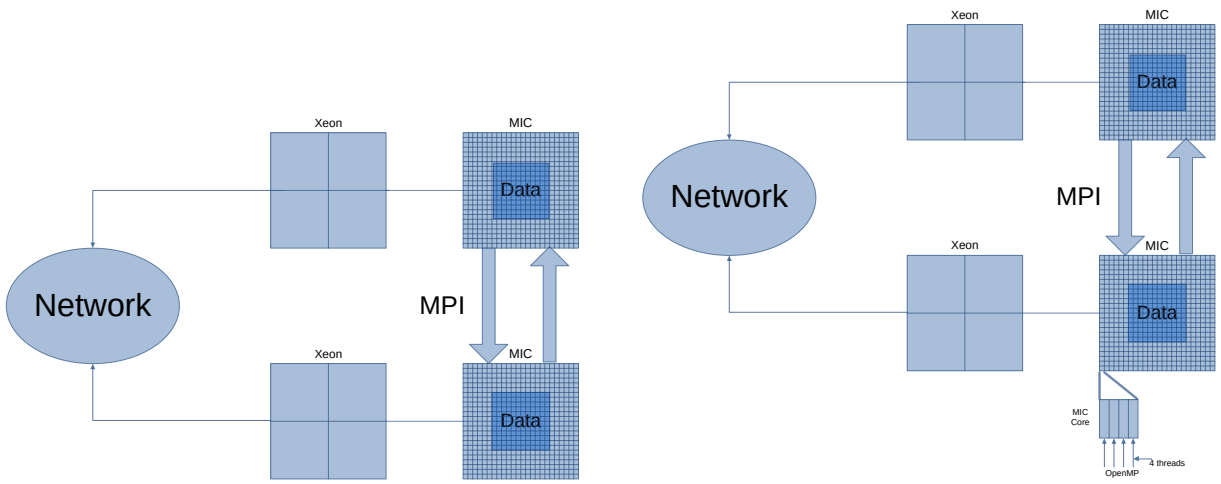
```
Function Transition(Array A, Array B)
    MPI__Init();
    MPI__Comm__rank();
    MPI__Comm__size();
    Allocate GA and GB on GPU memory;
    Copy A and B to GA and GB;
    Launch GPU kernel;
    for i = 0 → k − 1 do
        A[i] ← B[i];
    Copy GA back to A;
    MPI__Finalize();
```

ALGORITHM 3: MPI@MIC+OpenMP Programming model

```
Function Transition(Array A, Array B)
    MPI__Init();
    MPI__Comm__rank();
    MPI__Comm__size();
    #pragma omp parallel for omp__set__num__threads(4);
    for i = 0 → k − 1 do
        A[i] ← B[i];
    MPI__Finalize();
```

execution directly on the coprocessors, as shown in Figure 2.5.

Supercomputer provides a heterogeneous environment, including host Xeon CPUs and Xeon Phi coprocessors. MIC-based parallel implementation on high performance system is shown in Figure 2.7. Applications that are already implemented by MPI can use this model by distributing MPI ranks across the coprocessors natively. In the native model, MPI can be run natively on the coprocessors without any modification on the original source code. Each MIC core directly hosts n (up to 4) MPI processes. Therefore, if m Xeon Phi coprocessors are used, m n 60 MPI processes are created in the parallel implementation. However, no job is dispatched on the host Xeon CPU. Besides, each MPI process create 4 threads to run OpenMP program as shown in Algorithm 3.



(a) MPI native parallel implementation.

(b) MPI and OpenMP native parallel implementation.

Figure 2.7: MIC native parallel implementation on Beacon

2.4.4 MIC offload mode

The offload mode, as shown in Figure 2.8, provides an alternative approach to utilize the MIC coprocessors. In this case, a MPI program running on the host CPU can optionally launch part of work to a MIC coprocessor on the same platform. The developer just identifies lines or sections of code that are best suited for the many cores on MIC coprocessor by inserting commands to invoke the parallel capability.

The offload model uses the keyword pragma to specify code sections and to offload data to the MIC.In this model, the application starts on the host CPU. When an offload region is encountered, the offload region and data are transferred to run on the target device (MIC).The MPI processes are allocated on the host CPU cores, while the data and computation are dispatched to the MIC coprocessors. The MPI process specifies the number of threads to the MIC that uses OpenMP to handle data and calculation.

The code is just compiled for the host processor. When offload commands are encountered and the coprocessor is running and available, the required data and code is automatically transferred between the host and coprocessor as needed. If no MIC coprocessor is running or available, the

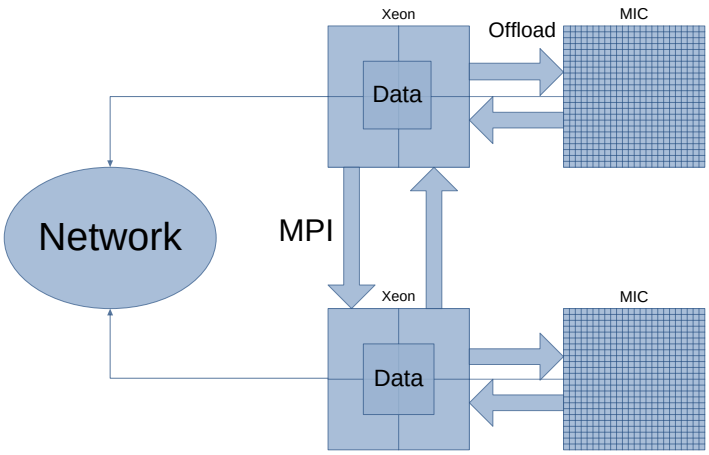


Figure 2.8: Offload parallel implementation on Beacon

command line or block of code will be executed on the host. This means that even though MIC may not work properly (such as connection problem between MIC and host CPU), the program still works on the host.

Offloading could simply be thought of an inline code that may be run on a coprocessor, as shown in Algorithm 4. The program executes the first pragma offload command to initialize all MIC devices. This initialization will load the MIC program on to each device, set up a data transfer between CPU and the device, and create a MIC thread to handle offload requests from the CPU thread. The host CPU processor and MIC coprocessors do not share the same system memory. As a result, the variables used by the code must be duplicated so that distinct copies exist on both the host processor and coprocessor. As shown in the following example, the pragma command uses specifiers to define the variables to copy between the host processor and coprocessor. The in specifier defines a specific variable as an input to the coprocessor. The value is not copied back to the host processor. The out specifier defines a specific variable as an output of the coprocessor. The host processor does not copy the variable to the coprocessor. The inout specifier defines a specific variable that is both copied from the host processor to the coprocessor and back from the coprocessor to the host processor.

ALGORITHM 4: MPI@MIC+OpenMP Programming model

Function Transition(*Array A*, *Array B*)

```
MPI__Init();
MPI__Comm__rank();
MPI__Comm__size();
#pragma offload target(mic) in(var) out(Output1 : length(Output1.size)
  inout(Output2 : length(Output2.size);
#pragma omp parallel for omp__set__num__threads(4);
for i = 0 → k − 1 do
    A[i] ← B[i];
Memory copy back to host;
MPI__Finalize();
```

Chapter 3

Study of parallel programming models with Intel MIC coprocessors

In this chapter, we conduct a detailed study regarding the performance and scalability of 5 execution modes on Intel MIC processors. In the first mode, the MPI process is directly run on each MIC core. In the second mode, we try to take advantage of the internal processing parallelism on each MIC core. Therefore, we launch 4 threads in each MPI process using OpenMP. Each MPI process is still run on a MIC core. In the third mode, only one MPI process is issued onto each MIC processor. Then OpenMP is used to launch threads to MIC cores. In the fourth mode, the MPI processes are run on the CPUs. The data processing is offloaded to the MIC processors using OpenMP. Only one thread is scheduled to one MIC core. The fifth mode is a variant of the fourth one. Four threads are scheduled to one MIC core in the fifth mode. We use two geospatial applications, i.e., Kriging interpolation and Cellular Automata, to test the performance and scalability of a single MIC processor and a computer cluster with hybrid nodes.

3.1 Introduction

The Intel MIC architecture contains many low-weight processing cores, as shown in Figure 3.1. These cores are connected through a high-speed ring bus. Each core can run 4 threads in parallel. Because each core alone is a classic processor, traditional parallel programming models, such as MPI and OpenMP, are supported on each core. The MIC processors typically co-exist with multicore CPUs, such as Intel Xeon CPU, in a hybrid computer node as coprocessors/accelerators. In the remainder of this chapter, a single MIC card or device will be called a MIC processor or MIC coprocessor. The constituent processing core on a MIC card will be called a MIC core.

Through this study, we have the following findings. (1) The native MPI programming model on the MIC processors is typically better than the offload programming model, which offloads the

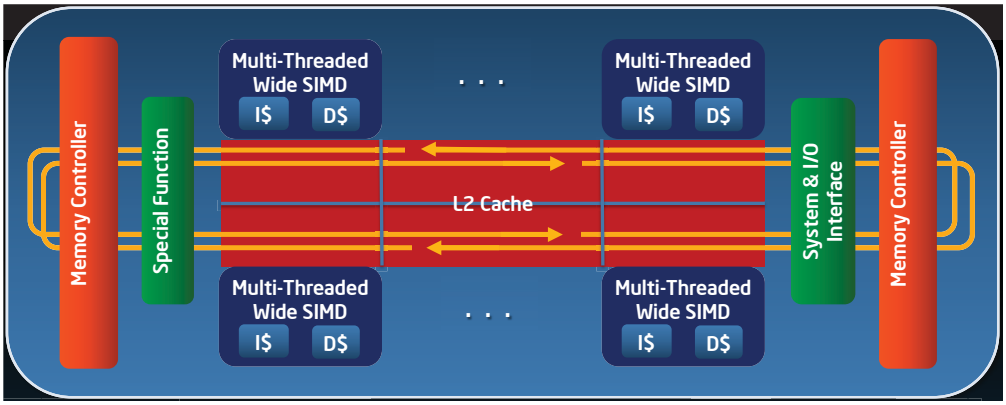


Figure 3.1: The architecture of Intel Xeon Phi coprocessor (MIC) [5].

workload to MIC cores using OpenMP. (2) On top of the native MPI programming model, multi-threading inside each MPI process can further improve the performance for parallel applications on computer clusters with MIC coprocessors. (3) Given a fixed number of MPI processes, it is a good strategy to schedule these MPI processes to as few MIC processors as possible to reduce the cross-processor communication overhead when the capacity of the on-board memory is not a limiting factor. (4) We also evaluate a hybrid MPI programming model, which is not officially supported by the Intel MPI compiler. In this hybrid model, the data processing is distributed to both the MIC cores and the CPU cores. The benchmarking results show that the hybrid model outperforms the native model.

The remainder of this chapter is organized as follows. The Intel MIC architecture and the two major programming models are discussed in Section 3.2. We discuss the details of the benchmarks and the experiment platform in Section 3.3. In Section 3.4, we show the experiment results on a single MIC device. We also compare the performance of a single MIC device with a single Xeon CPU and the latest GPUs. Then we expand the experiment on two geospatial benchmarks to the Beacon cluster using many computer nodes in Section 3.5. We discuss some related work in Section 3.6. Finally, we give the concluding remarks in Section 3.7.

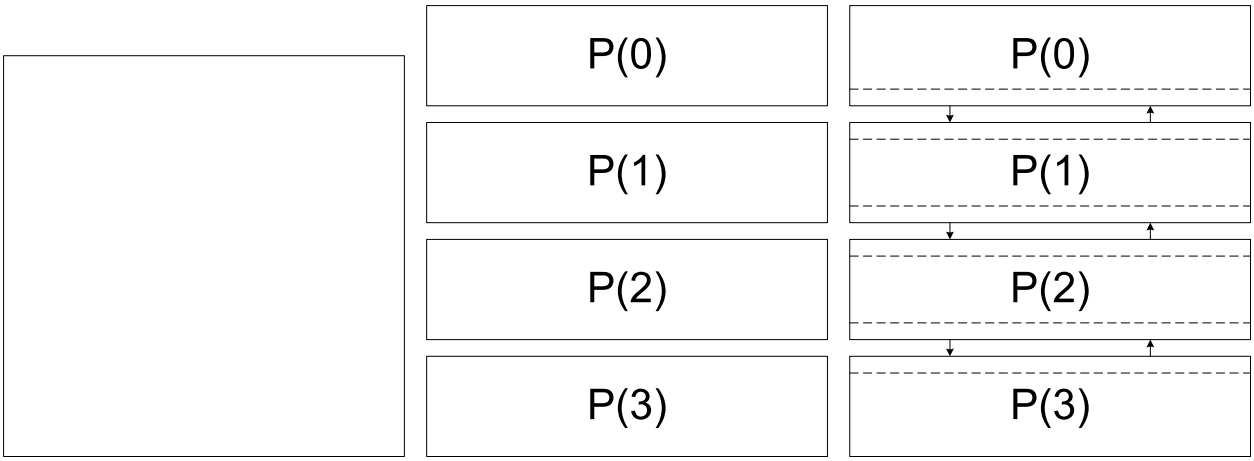
3.2 Intel MIC architecture and programming models

The commercially available Intel coprocessor based on Many Integrated Core architecture is Xeon Phi, as shown in Figure 3.1. Xeon Phi contains more than 50 scalar processing cores with vector processing units. These cores are connected through a high-speed bi-directional, 1024-bit-wide ring bus (512 bits in each direction). In addition to the scalar unit inside each core, there is a vector processing unit to support wide vector processing operations. Further, each core can execute 4 threads in parallel. The communications between the cores can be realized through the shared memory programming models, e.g., OpenMP. Additionally, each core can run MPI to realize communication. Direct communication between MIC processors across different nodes is also supported through MPI.

This work uses two approaches to parallelizing applications on computer clusters equipped with MIC processors. The first approach is the native model. In this model, the MPI processes directly run on the MIC processors. There are two variants under this model. (1) Let each MIC core directly host one MPI process. In this way, the 60 cores on the Xeon Phi 5110P, which is used in this work, are treated as 60 independent processors while sharing the 8 GB on-board memory. (2) Only issue one MPI process on one MIC card. This single MPI process then spawns threads running on many cores using OpenMP. The second approach is to treat the MIC processors as clients to the host CPUs. The MPI processes will be hosted by CPUs, which will offload the computation to the MIC processors. Multithreading programming models such as OpenMP can be used to allocate many MIC cores for data processing in the offload model.

In this work there are 5 different parallel implementations in these 2 models as follows.

- Native model: In this model, MPI processes directly execute on MIC processors. There are further 3 implementations.
  - Native-1 (N-1): Issue one MPI process onto each MIC core. If n MIC cores are allocated, then n MPI processes are issued. Each MPI process contains only one thread.



(a) Problem space. (b) Kriging interpolation. (c) Game of Life.

Figure 3.2: Data partition and communication in two benchmarks. In Kriging interpolation there is no communication among MPI processes (i.e., P(I) in the figure) during computation. In Game of Life, MPI processes need to communicate with each other in the computation.

  - Native-2 (N-2): Issue one MPI process onto each MIC core. Each MPI process contains 4 threads.

  - Native-3 (N-3): Issue only one MPI process onto each MIC card. Then allocate many MIC cores using OpenMP. On each MIC core, issue 4 threads.

- Offload model: In this model, the CPU offloads the work to the MIC processor using OpenMP. There are further 2 implementations.

  - Offload-1 (O-1): Issue one thread onto each MIC core. If n MIC cores are allocated, then n threads are issued.

  - Offload-2 (O-2): Issue 4 threads onto each MIC core. If n MIC cores are allocated, then $4 \times n$ threads are issued.

3.3 Experiment setup

3.3.1 Benchmarks

Two geospatial applications are chosen to represent two types of benchmarks in high-performance computing: the embarrassingly parallel case and the intense communication case.

Embarrassingly parallel case – Kriging Interpolation

Kriging is a geostatistical estimator that infers the value of a random field at an unobserved location [24]. Kriging is based on the idea that the value at an unknown point should be the average of the known values of its neighbors.

Kriging can be viewed as a point interpolation that reads input point data and returns a raster grid with calculated estimations for each cell. Each input point is in the form (x_i, y_i, Z_i) where x_i and y_i are the coordinates and Z_i is the value. The estimated values in the output raster grid are calculated as a weighted sum of input point values as in (3.1).

$$\hat{Z}(x,y) = \sum_{i=1}^{k} w_i Z_i, \tag{3.1}$$

where w_i is the weight of the i-th input point. Theoretically the estimation can be calculated by the summation through all input points. In general, users can specify a number k so that the summation is over k nearest neighbors of the estimated point in terms of distance. This decrease of computation is due to the fact that the farther the sampled point is from the estimated point, the less impact it has in the summation. For example, the commercial software ArcGIS [2] uses the 12 nearest points (i.e., $k = 12$) in the Kriging calculation by default. In this benchmark, embarrassing parallelism can be realized since the interpolation calculation over each cell has no dependency on the others.

In the Kriging interpolation benchmark, the problem space as shown in Figure 5.8(a) is evenly

```
for(each data set in the 4 data sets) {
  /*The following for loop can be parallelized*/
  for(each point in the 1,440x720 output grid) {
    Scan the whole data set to find the 10 closest
      sampled points;
    Use Equation (1) to estimate the value of the
      unsampled point;
  }
}
```

Figure 3.3: Pseudocode of Kriging interpolation. The inner `for` loop can be parallelized while the 4 data sets in the out `for` loop are processed in sequence.

partitioned among all MPI processes as shown in Figure 5.8(b), in which we use 4 processes as an example. The computation in each MPI process is purely local, i.e., there is no cross-process communication.

The input size of this benchmark is 171 MB, consisting of 4 data sets with the respective sizes of 29 MB, 37 MB, 48 MB, and 57 MB. Each data set has 2,191, 4,596, 6,941, and 9,817 sample points, respectively. The output raster grid for each data set has a consistent dimension of 1,440×720. In other words, each data set will generate a 1,440×720 grid. The value of each point in the output grid needs to be estimated using those sample points in the corresponding input data set. In our experiments, the value of an unsampled point will be estimated using the values of the 10 closest sample points, i.e., $k = 10$. These 4 data sets are processed in a sequence. For each data set, the generation of its corresponding output grid is evenly distributed among all MPI processes. In order to generate the value of a point in the output grid, all the sampled points in the data set need to be scanned to find the 10 closest sample points. The pseudocode of Kriging interpolation is illustrated in Figure 3.3.

Intense communication case – Cellular Automata

Cellular Automata (CA) are the foundation for geospatial modeling and simulation. Game of Life (GOL) [22], invented by British mathematician John Conway, is a well-known generic Cellular Automaton. It consists of a collection of cells that can live, die or multiply based on a few mathematical rules.

```
for(iteration=0;iteration<100;iteration++) {
  /*The following for loop can be parallelized*/
  for(all cells in the universe) {
    Update the status of cell[i,j] based on the
      statuses of cell[i,j] and its 8 neighbors;
  }
}
```

Figure 3.4: Pseudocode of Game of Life.

The universe of the Game of Life is a two-dimensional square grid of cells, each of which is in one of two possible states, alive ('1') or dead ('0'). Every cell interacts with its eight neighbors, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

- Any live cell with fewer than two live neighbors dies, as if caused by under-population.
- Any live cell with two or three live neighbors lives on to the next generation.
- Any live cell with more than three live neighbors dies, as if by overcrowding.
- Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

In this benchmark, the status of each cell in the grid will be updated for 100 iterations. In each iteration, the statuses of all cells are updated simultaneously. The pseudocode is illustrated in Figure 3.4. In order to parallelize the updating process, the cells in the square grid are partitioned into stripes along the row-wise order. Each stripe is handled by one MPI process. At the beginning of each iteration, each MPI process needs to send the statuses of the cells along the boundaries of each stripe to its neighbor MPI processes and receive the statuses of the cells of two adjacent rows as shown in Figure 5.8(c).

3.3.2 Experiment Platform

We conduct our experiments on the NSF sponsored Beacon supercomputer [3] hosted at the National Institute for Computational Sciences (NICS), University of Tennessee.

The Beacon system (a Cray CS300-AC Cluster Supercomputer) offers access to 48 compute

nodes and 6 I/O nodes joined by FDR InfiniBand interconnect, which provides a bi-directional bandwidth of 56 Gb/s. Each compute node is equipped with 2 Intel Xeon E5-2670 8-core 2.6-GHz processors, 4 Intel Xeon Phi (MIC) 5110P coprocessors, 256 GB of RAM, and 960 GB of SSD storage. Each I/O node provides access to an additional 4.8 TB of SSD storage. Each Xeon Phi 5110P coprocessor contains 60 1.053-GHz MIC cores and 8-GB GDDR5 on-board memory. Altogether Beacon contains 768 conventional cores and 11,520 accelerator cores that provide over 210 TFLOP/s of combined computational performance, 12 TB of system memory, 1.5 TB of coprocessor memory, and over 73 TB of SSD storage.

The compiler used in this work is Intel 64 Compiler XE, Version 14.0.0.080 Build 20130728, which supports OpenMP. The MPI library is intel-mpi 4.1.0.024.

3.4 Experiments and results on a single device

Since a single Intel Xeon Phi 5110P processor is a 60-core processor, it is worthwhile to investigate the performance and scalability of a single MIC processor alone.

3.4.1 Scalability on a single MIC processor

When MPI programming model is used to implement the Kriging interpolation application, the workload is evenly distributed among MPI processes. In this benchmark, there are 4 data sets. For each data set, the output is a 1,440$\times$720 raster grid. In the MPI implementation, we increase the number of MPI processes from 10 to 60 with a stride of 10 processes. The computation of 720 columns of the output grid is evenly distributed. The 50-process configuration is skipped because 720 columns cannot be distributed among 50 processes equally. For the offload programming model, we use OpenMP to parallelize the `for` loops in the program. The OpenMP APIs will automatically distribute workload to the MIC cores evenly.

The detailed execution times of the Kriging interpolation benchmark under both program-

Table 3.1: Performance of Kriging interpolation on a single MIC processor (unit: second).

Execution mode: Native-1						
	Number of MIC cores					
	10	20	30	40	50	60
Read	0.65	0.60	0.66	0.72	NA*	0.79
Interpolation	2734.45	1353.48	921.76	664.74		455.34
Write	9.44	9.21	11.04	8.04		7.95
Total	2744.54	1363.30	933.46	673.50		464.09
Execution mode: Offload-1						
	Number of MIC cores					
	10	20	30	40	50	60
Read	0.04	0.05	0.04	0.04	0.04	0.04
Interpolation	2758.22	1570.75	1040.44	784.30	632.65	548.15
Write	1.77	1.99	1.65	1.44	1.45	1.57
Total	2760.03	1572.78	1042.12	785.78	634.14	549.75

*The workload could not be distributed among 50 cores evenly.

ming models while each MIC core hosts only one thread are listed in Table 3.1. By looking at the time curves in Figure 3.5, we can find that both models show a good strong scalability for this application. Their performance in terms of interpolation time is very close too. The reason we do not include the write time in Figure 3.5 is that the write time may become dramatically lengthy when the number of MPI processes increases. In the Kriging interpolation application, each output raster grid is written into a file. When many MPI processes try to write to the same file, their writes need to be serialized. Further, the arbitration takes a lot of time. This effect is not very significant when one MIC processor is used. Later, we will find that the write time can become extremely significant when many MIC processors are allocated.

For Game of Life three different grid sizes are tested, i.e., 8,192×8,192, 16,384×16,384, and 32,768×32,768. However, we encounter either out-of-memory error or runtime error for the 32,768×32,768 case when only one MIC processor is used. From the results in Table 3.2, it can be found that the native model consistently outperforms the offload model for this intense communication case. By looking at the performance curves in Figure 3.6, we can find that both programming

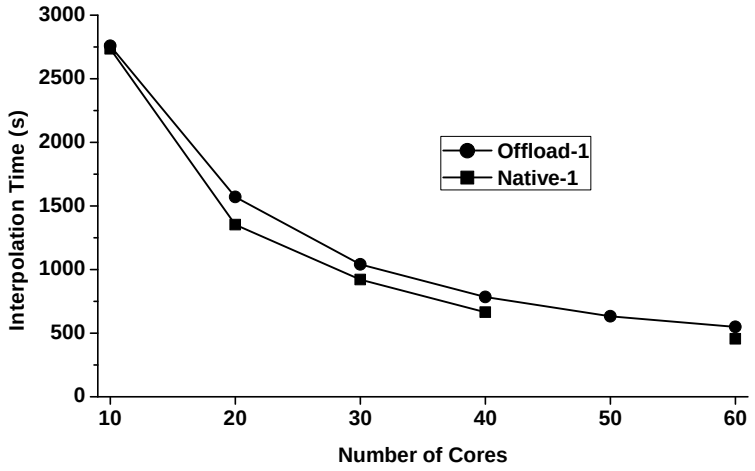
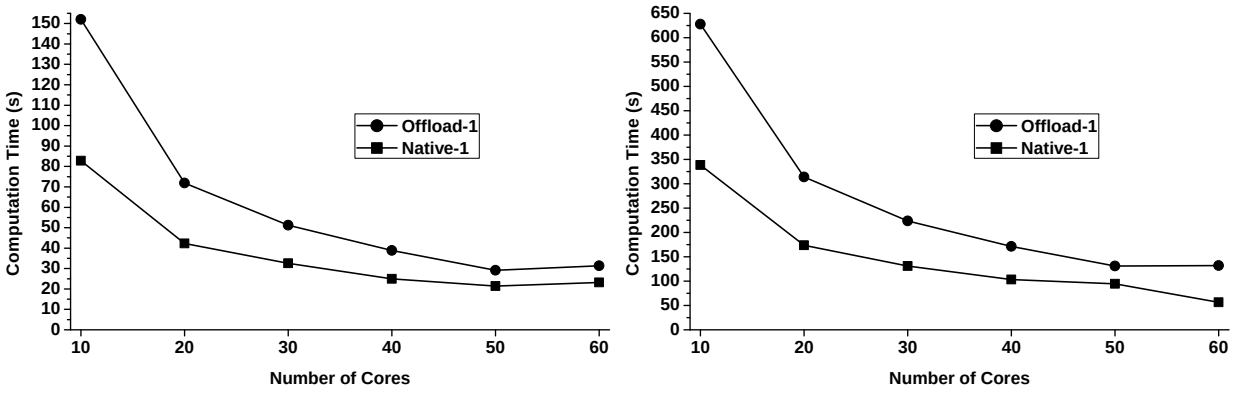


Figure 3.5: Performance of Kriging interpolation on a single MIC processor. For both implementation, only one thread runs on a MIC core. The native implementation outperforms the offload one with a small margin.

Table 3.2: Performance of Game of Life on a single MIC processor (unit: second).

Execution mode: Native-1						
Problem Size	Number of MIC cores					
	10	20	30	40	50	60
8192×8192	82.85	42.27	32.56	24.91	21.37	23.15
16384×16384	338.57	173.57	131.10	103.30	94.41	56.31
Execution mode: Offload-1						
Problem Size	Number of MIC cores					
	10	20	30	40	50	60
8192×8192	152.06	71.9	51.23	38.88	29.1	31.33
16384×16384	627.94	313.88	223.54	171.33	131.14	131.72

models show strong scalability when the number of cores increases from 10 to 20. Beyond that, both models lose the strong scalability although the total computation time still decreases. For both problem sizes, the reduction of workload is gradually offset by the increase of communication overhead when the number of cores increases. Further, when more cores are allocated, the memory access demand increases as well. Eventually, the communication and memory bandwidth become the limiting factors for the performance.



(a) 8,192×8,192. (b) 16,384×16,384.

Figure 3.6: Performance of Game of Life on a single MIC processor. The native model outperforms the offload model with a big margin when only a few cores are used. The performance gap decreases as more cores are allocated.

Table 3.3: Performance of Kriging interpolation on single devices (unit: second).

	MIC (60 cores)					CPU (Xeon E5-2670)		Nvidia GPU	
	N-1	N-2	N-3	O-1	O-2	8-thread	16-thread	C2075	K20
Read	0.79	1.03	0.45	0.04	0.42	0.01	0.01	0.01	0.01
Intp.	455.34	173.89	5147.95	548.15	225.93	330.11	182.60	23.87	10.90
Write	7.95	8.57	16.71	1.57	1.38	9.85	10.27	1.68	1.68
Total	464.09	183.49	5165.11	549.75	227.72	339.96	192.86	25.55	11.77

3.4.2 Performance comparison of single devices

As an emerging new technology, it is worthwhile to compare the performance of the Intel MIC processor with the other popular accelerator, i.e., GPU. Furthermore, it is a routine to include very powerful multicore CPUs in supercomputers. Therefore, we conduct a comparison among these three technologies at the full capacity of a single device. For Intel Xeon Phi 5110P, we use all 60 cores under two programming models for 5 different implementations. For the 8-core Xeon E5-2670 CPU on Beacon cluster, we use OpenMP to issue either 8 threads or 16 threads. For GPU, we test two devices, the Nvidia Tesla C2075 based on Fermi architecture [43] and the Tesla K20 based on Kepler architecture [44]. The CUDA version is 5.5.

The execution times of Kriging interpolation on various devices are listed in Table 3.3. It

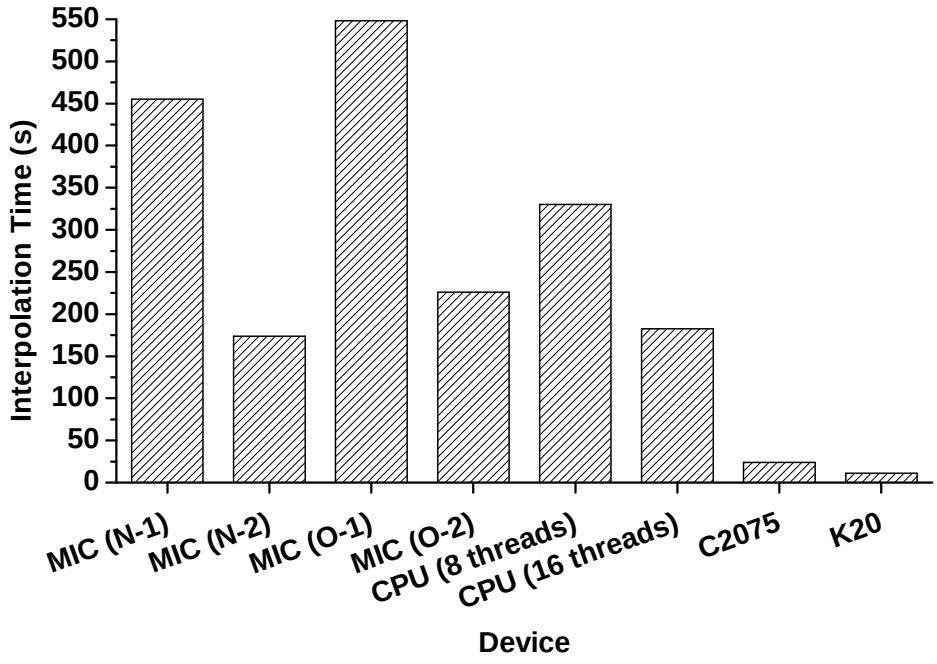


Figure 3.7: Performance of Kriging interpolation on single devices (excluding Native-3 implementation on Intel MIC device).

Table 3.4: Performance of Game of Life on single devices (unit: second).

	MIC (60 cores)					CPU (Xeon E5-2670)		Nvidia GPU	
	N-1	N-2	N-3	O-1	O-2	8-thread	16-thread	C2075	K20
8192^2	23.15	18.22	11.23	31.33	19.53	12.03	8.13	15.36	3.25
16384^2	56.31	82.66	41.12	131.72	79.93	48.22	32.65	58.44	12.58
32768^2	NA					217.33	114.98	274.03	46.99

can be found that the performance of the MIC processor and the CPU is at the same order of magnitude. When running at the full capacity, the performance of the Intel Xeon Phi 5110P is equivalent to the Xeon E5-2670. By increasing the number of threads in an MPI process to 4, the Native-2 implementation is able to improve the performance by 3 times compared with Native-1 implementation. However, the Native-3 implementation, i.e., one MPI process with 240 threads, has the much worse performance. We varied the number of threads in the MPI process and found that the performance did not change significantly. We speculate that the OpenMP library does not work well with the Kriging interpolation under Native-3 programming model. For Xeon CPU the 16-thread CPU implementation is almost 2 times faster than the 8-thread implementation because

Table 3.5: Performance of Kriging interpolation under various execution modes on multiple MIC processors(unit: second).

Number of	Native-1				Native-2				Native-3			
Processors	Read	Inter.*	Write	**Total**	Read	Inter.[1]	Write	**Total**	Read	Inter.[1]	Write	**Total**
2	1.24	232.43	12.24	245.90	0.57	60.43	8.82	69.82	0.33	2563.07	12.53	2575.97
4	1.27	116.34	16.44	134.05	0.51	36.54	122.53	159.59	0.33	1284.93	10.04	1305.35
8	1.23	61.48†	54.43	117.14	0.50	20.43[2]	240.33	261.26	0.33	730.58	9.37	740.29
16	1.31	36.74[2]	300.23	338.28	0.52	12.33[2]	210.45	223.30	0.34	377.95	9.10	387.39

Number of	Offload-1				Offload-2			
Processors	Read	Interpolation*	Write	**Total**	Read	Interpolation*	Write	**Total**
2	0.18	280.83	1.60	282.61	0.39	91.65	1.88	95.79
4	0.04	141.03	1.27	142.33	System does not return result.			
8	0.04	74.30	1.19	75.53				
16	0.04	38.54	5.94	44.51				

*The interpolation time includes both the time spent on data processing and the time spent on communication.

†Only 360 or 720 MIC cores are used in the computation with 8 or 16 processors, respectively.

each CPU core can execute two threads simultaneously. Both GPUs are able to improve the performance by one order of magnitude. Further, K20 is more than 2 times faster than C2075, as shown in Figure 3.7.

The performance results of Game of Life on three different types of processors are listed in Table 3.4. The performance of both models on the MIC processor is at the same order of magnitude as the implementations on the CPU and the C2075. All 5 implementations work quite well on MIC and the native model is typically better than offload model. The Native-3 implementation has the best performance compared with other 4 implementations on MIC. Overall the K20 implementation is generally one order of magnitude better in terms of performance compared with other implementations.

Table 3.6: Performance of Game of Life under various execution modes on multiple MIC processors (unit: second).

Number of	8,192×8,192					16,384×16,384					32,768×32,768				
Processors	N-1	N-2	N-3	O-1	O-2	N-1	N-2	N-3	O-1	O-2	N-1	N-2	N-3	O-1	O-2
2	14.56	7.99	92.94	20.40	13.66	48.39	33.11	275.87	78.71	48.59	194.15	149.43	964.62	308.01	184.72
4	11.63	8.04	44.41	11.57	8.58	46.31	24.06	172.04	42.65	26.31	169.54	104.14	544.44	155.99	96.75
8	7.84	9.28	23.26	12.32	8.08	39.78	22.98	108.01	42.08	28.86	157.73	106.24	317.24	154.56	99.26
16	7.18	8.74	21.46	13.52	9.39	35.30	23.60	107.01	47.91	34.19	128.40	110.99	300.68	176.73	105.82

3.5 Experiments and results using multiple MIC processors

We also conduct the experiments using multiple MIC processors to demonstrate the scalability of the parallel implementations for those two geospatial applications. For both benchmarks we have 5 parallel implementations on the Beacon computer cluster using multiple nodes.

We want to show the strong scalability of the parallel implementations as the case on the single device. Therefore, the problem size is fixed for each benchmark while the number of participating MPI processes is increased.

3.5.1 Comparison among five execution modes

Kriging Interpolation

We allocate 2, 4, 8, and 16 MIC processors for 4 different implementation cases. For the Native-1 and the Native-2 implementations, $m \times 60$ MPI processes are created if m MIC processors are used. For the Native-3, Offload-1, and Offload-2 execution modes, m MPI processes are created if m MIC processors are used. As mentioned before, for each output raster grid, the generation of the 720 columns is evenly distributed among the MPI processes. Therefore, only 360 or 720 MPI processes, which execute on 360 or 720 MIC cores, are created when 8 or 16 MIC processors are allocated, respectively, for both Native-1 and Native-2 cases.

The detailed results of the five execution modes for Kriging interpolation are listed in Table 3.5. It is noticed that the system does not return results when more than 2 MIC processors are

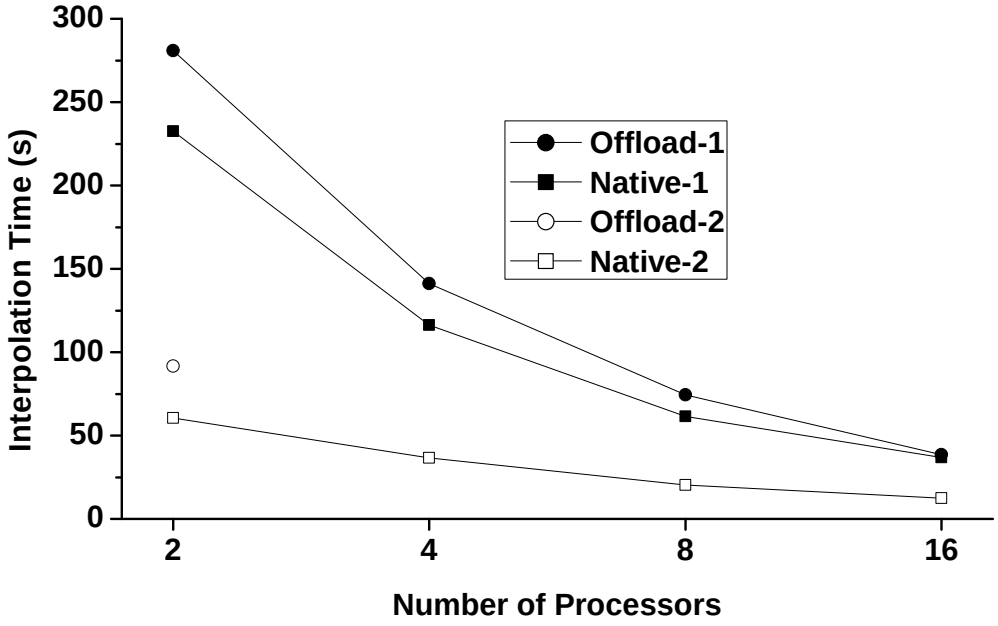
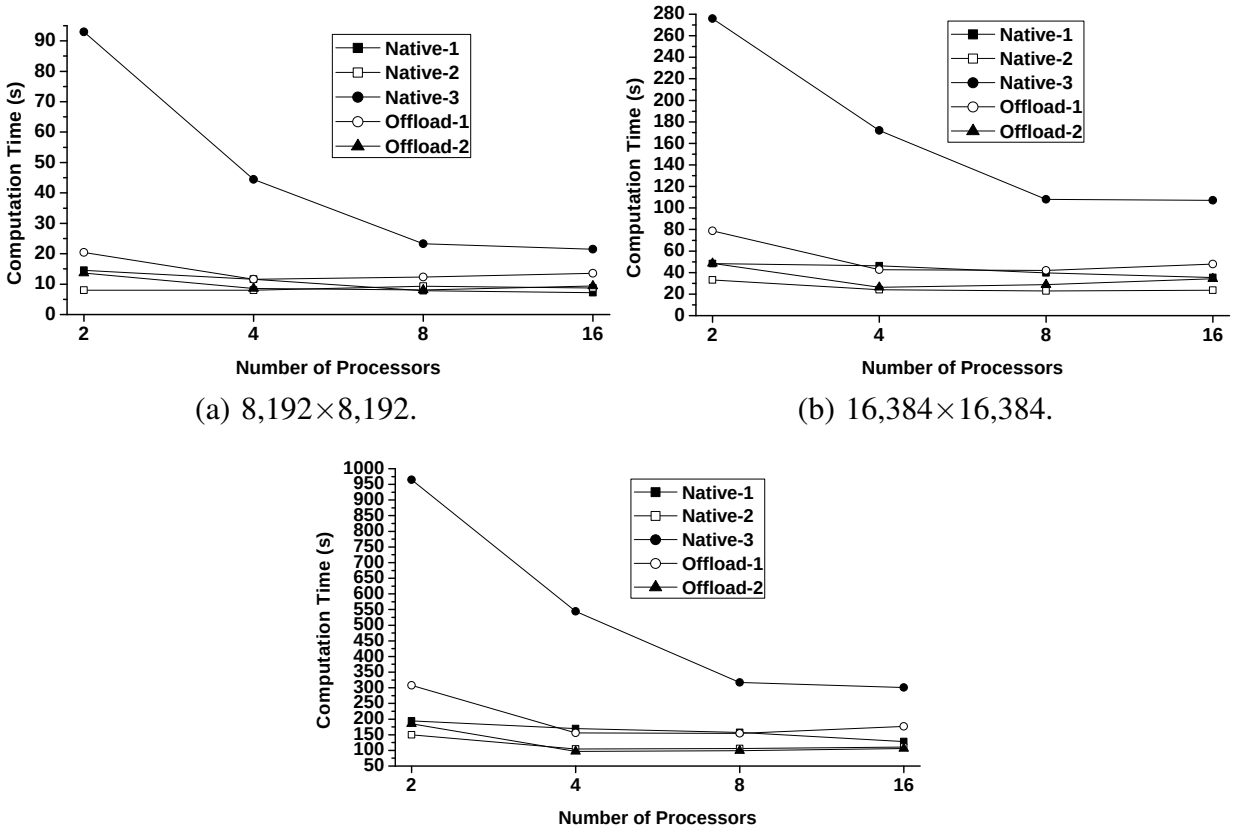


Figure 3.8: Performance of Kriging interpolation under various execution modes on multiple MIC processors (excluding Native-3 execution mode).

used for Offload-2 execution mode. We can find that the write time grows dramatically when more MIC processors are used for both Native-1 and Native-2 execution modes. As mentioned before, the serialization of the write and the arbitration among the numerous MPI processes contribute to the lengthy write process. Therefore, we only include the interpolation time, which includes both the time spent on data processing and the time spent on cross-processor communication, when comparing the performance of the four execution modes in Figure 3.8. We do not include Native-3 in Figure 3.8 because its interpolation time is significantly larger than other execution modes although it obeys the strong scalability. It can be found that the Native-1 and the Offload-1 execution modes have the very close performance for this benchmark. When the multithreading is applied in each MPI process on the native MPI programming model, the performance can be improved by roughly 3 times. This case shows that it is not enough to only parallelize application to all the cores on MIC processors. It is equally important to increase the parallelism on each MIC core to further improve the performance.



(a) 8,192×8,192. (b) 16,384×16,384.

(c) 32,768×32,768.

Figure 3.9: Performance of Game of Life under various execution modes on multiple MIC processors.

Conway's Game of Life

For Game of Life on multiple MIC processors, three different grid sizes are tested, i.e., 8,192×8,192, 16,384×16,384, and 32,768×32,768. By observing the performance results in Table 3.6 and Figure 5.10, it can be found that the behavior is quite different from the performance behavior of Kriging interpolation. First, the strong scalability does not hold for all five execution modes. Although the offload execution modes are still able to reduce the computation time to half when moving from 2-processor implementation to 4-processor implementation, the performance plateaus afterwards. For Native-1 and Native-2 execution modes, it almost stops scaling when more processors are allocated. Apparently, for this communication dense application, there is not much performance gain when increasing the number of MIC processors from 4 to 8 and 16. When

Table 3.7: Performance of Game of Life using MPI@MIC_Core+OpenMP execution mode (unit: second).

Number of	8,192×8,192		16,384×16,384		32,768×32,768	
Processors	4 threads	8 threads	4 threads	8 threads	4 threads	8 threads
2	7.99	10.94	33.11	32.92	149.43	110.37
4	8.04	9.03	24.06	27.94	104.14	109.79
8	9.28	8.39	22.98	25.69	106.24	100.79
16	8.74	10.77	23.60	27.11	110.99	110.67

the grid is partitioned into $m \times 60$ MPI processes on m MIC processors, the performance gain from the reduced workload on each MIC core is easily offset by the increase of the communication cost among the cores. Therefore, it is critical to keep a balance between computation and communication for achieving the best performance. For Native-3 execution mode, there is a big increase of computation time from one-processor implementation to multiple-processor implementation. Native-3 execution mode is not officially mentioned in the programming guide on Beacon computer cluster. Therefore, we speculate that the library support for Native-3 execution mode on multiple devices is premature at this moment.

3.5.2 Experiments on the MPI@MIC_Core+OpenMP execution mode

For the implementations using the Native-2 execution mode in Section 3.2, the number of threads running on each MIC core is 4, which is the number of threads a MIC core can physically execute in parallel. We also want to check the potential of performance improvement by running more threads on a single core. Therefore, in addition to the case of 4, we double the number of threads to 8 for the Game of Life benchmark. The results are listed in Table 3.7. It can be found that the benefit of adding more threads to MIC cores is very marginal. For small problem sizes, e.g., 8,192×8,192, the 8-thread OpenMP implementation actually has a worse performance than the 4-thread OpenMP implementation for most cases. For this communication-intensive benchmark, partitioning the computation into more threads introduces more cross-thread communication overhead. For large problem sizes, it is still possible to achieve some performance benefit if each MPI

Table 3.8: Performance of Game of Life (32,768×32,768) using Offload-1 execution mode (unit: second).

Number of	# of OpenMP threads offloaded to each MIC processor					
Processors	10	20	30	40	50	60
2	1,375.37	730.96	478.81	382.15	317.84	308.01
4	709.70	382.15	258.39	196.05	158.40	155.99
8	687.71	351.86	240.56	184.40	149.45	154.56
16	689.78	367.11	244.26	193.79	160.14	176.73

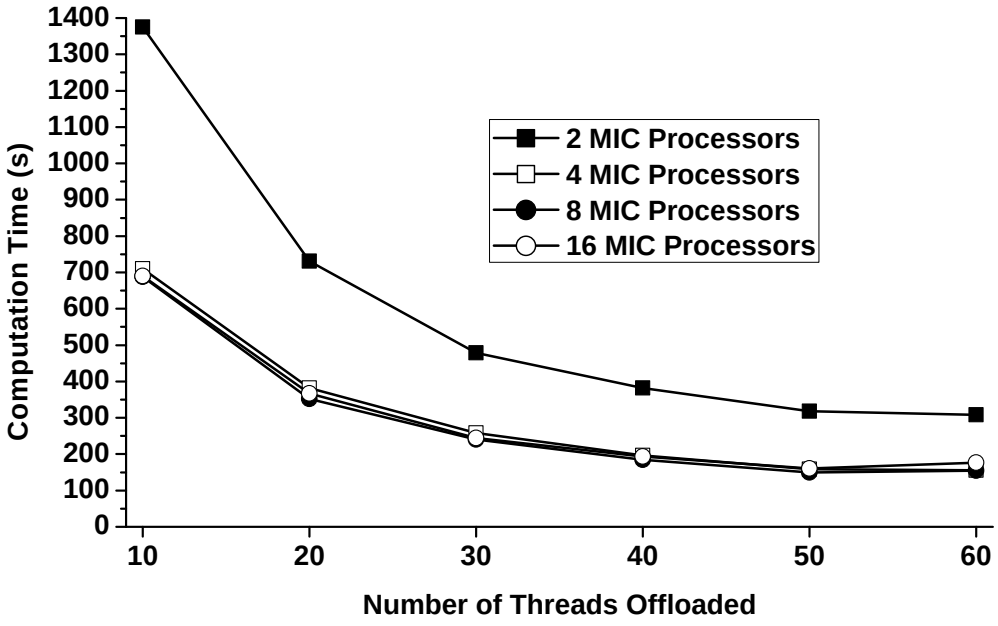


Figure 3.10: Performance of Game of Life (32,768×32,768) using Offload-1 execution mode. The number of threads on a MIC processor is increased from 10 to 60.

process is given a relatively large amount of data, e.g., 32,768×32,768 partitioned into 120 MIC cores.

3.5.3 Experiments on the Offload-1 execution mode

For the implementations using the Offload-1 execution mode in Section 3.2, the number of OpenMP threads offloaded to the a MIC processor by an MPI process, which runs on the CPU, is 60, i.e., one OpenMP thread per MIC core. In this experiment, we change the number of threads offloaded to the MIC processor by the MPI process from 10 to 60, as shown in Table 3.8 and

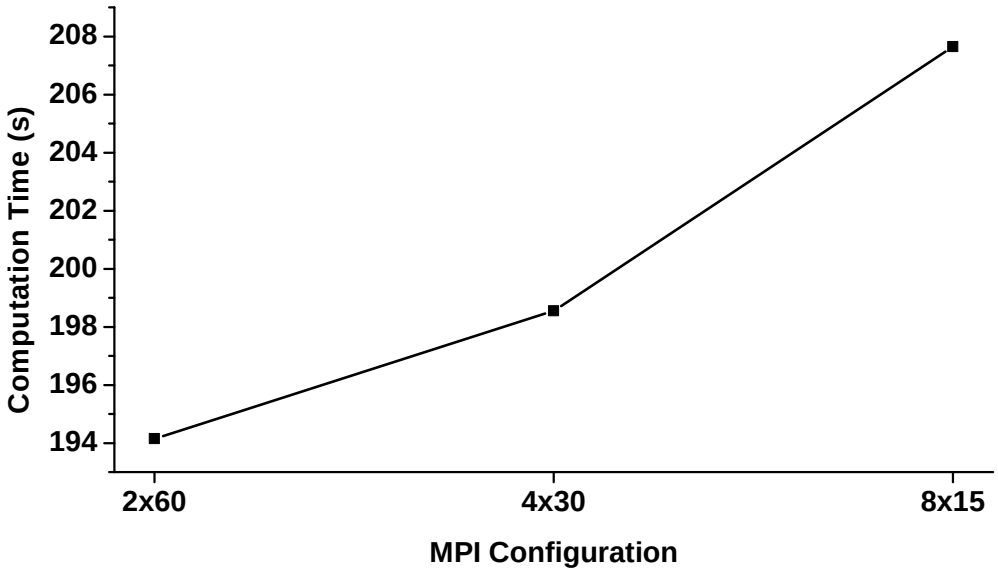


Figure 3.11: Performance of Game of Life (32,768×32,768) under different MPI configurations using Native-1 execution mode. Given 120 MPI processes, 2×60 means that 120 processes are distributed in 2 MIC cards, each of which hosts 60 processes. The less number of MIC cards are used, the better the performance.

Figure 3.10. In each case, when the number of threads increases from 10 to 30, the scalability holds. When more threads are scheduled, the computation time decreases, however, at a much smaller rate. For most cases, the computation time actually grows when the number of offloaded threads is increased from 50 to 60. This performance degradation may be due to the increased inter-thread communication overhead.

We also can find that the performances are almost the same for implementations using more than 2 MIC processors. Apparently when 4 or more MIC processors are allocated, the cross-processor communication overhead becomes dominant in the computation process so that adding more processors will not increase the overall performance.

3.5.4 Experiments on the distribution of MPI processes

When an MPI parallel application runs on a computer cluster with nodes consisting of many-core processors such as Xeon Phi, the distribution of MPI processes is not uniform. Some MPI processes are scheduled to the cores on the same processor. The others are scheduled to differ-

ent processors. Two MPI processes on the same processor are physically close to each other. On the other hand, two MPI processes on two separate processors are distant. The difference of the distance between two MPI processes will cause the disparity of the inter-MPI communication time.

We design a simple benchmark consisting of only 2 MPI processes using native programming model. In this benchmark, MPI process_A sends 500 MB data to MPI process_B. Then MPI process_B returns the 500 MB data back to MPI process_A. We have two options to run the benchmark. In Implementation_1, both MPI processes are scheduled to the same MIC processor. In Implementation_2, these two MPI processes are scheduled to two separate MIC processors. It turns out that Implementation_1 and Implementation_2 take 1.59 seconds and 2.81 seconds, respectively. Apparently, the longer distance between the two MPI processes in Implementation_2 contributes to the more time spent on communication.

The location difference of MPI processes can result in the performance disparity of an application when it is executed under different MPI configurations while the total number of MPI processes is the same. Figure 3.11 illustrates the different performances of the Game of Life benchmark using the various configurations when 120 MPI processes run on 120 MIC cores. Each MPI process contains only one thread. In the 2×60 configuration, 2 MIC processors are allocated, each of which hosts 60 MPI processes. When the number of MIC processors doubles, the number of MPI processes on a processor is halved. The more processors are allocated, the more cross-processor communication, which brings down the performance. Therefore, when the capacity of the on-board memory is not a limiting factor, it is typically a good strategy to schedule as many MPI processes to a single MPI processor as possible to minimize the cross-board communication overhead.

3.5.5 Hybrid MPI vs native MPI

Another programming/execution model that is not officially supported on Beacon computer cluster is the Native MPI@Hybrid CPU/MIC, i.e., the MPI processes run on both CPUs and MIC

processors. The results in Section 3.4.2 already demonstrate the impressive performance of the latest multicore CPUs. Therefore, it is necessary to use both processors in the applications. We first implement the Kriging interpolation on the 57 MB data set using 16 MPI processes on a single Xeon E5-2670 CPU, which support 16 parallel threads. The total execution time is 46.02 seconds. Then we implement the same application using a 16+14 hybrid MPI model, i.e., 16 MPI processes on a single Xeon CPU and 14 MPI processes on 14 MIC cores of a single card[1], the total execution time is 24.75 seconds, an almost 2× speedup. Again, each MPI process contains only one thread in this sub-study.

We also carry out the hybrid MPI programming model on a separate workstation, which contains one Xeon E5-2620 CPU and two Xeon Phi 5110P cards. On this platform, we use the Game of Life (16,384×16,384) as the benchmark. The native MPI implementation of 120 MPI processes on two MIC cards takes 30 seconds. The 12+120 hybrid MPI implementation in which the additional 12 MPI processes run on the single CPU takes 27.42 seconds. The 1.1× speedup aligns with the ratio of number of MPI processes between the hybrid model and the native model.

3.6 Related work

MPI and OpenMP are two popular parallel programming APIs and libraries. MPI is primarily for inter-node programming on computer clusters. On the other hand, OpenMP is mainly used for parallelizing a program on a single device. Krawezik compared MPI and three openMP programming styles on shared memory multiprocessors using a subset of the NAS benchmark (CG, MG, FT, LU) [32]. Experimental results demonstrate that OpenMP provides competitive performance compared to MPI for a large set of experimental conditions. However the price of this performance is a strong programming effort on data set adaptation and inter-thread communications. Numerous benchmarks have been used to evaluate the performance of supercomputers. For example, the HPC Challenge (HPCC) benchmark suite and the Intel MPI Benchmark (IMB) are used to compare and

[1]When we allocate more than 14 MPI processes on the MIC card, the result is incorrect.

evaluate the combined performance of processor, memory subsystem and interconnect fabric of five leading supercomputers - SGI Altix BX2, Cray XI, Cray Opteron Cluster, Dell Xeon cluster, and NEC SX-8 [51, 52]. The portability and performance efficiency of radio astronomy algorithms are discussed in [41]. Derivative calculations for radial basis function were accelerated on one Intel MIC card [20]. We use two representative geospatial applications with different communication patterns for benchmarking purpose. Although they are both domain specific applications, many applications in other domains share the same internal communication patterns as these two cases.

Schmidl et al compared a Xeon-based two-socket compute node with the Xeon Phi stand-alone in scalability and performance using OpenMP codes [55]. Their results show significant differences in absolute application performance and scalability. The work in [53] evaluated the single node performance of an SGI Rackable computer that has Intel Xeon Phi coprocessors. NAS parallel benchmarks and CFD applications are used for testing four programming models, i.e., offload, processor native, coprocessor native and symmetric (processor plus coprocessor). They also measured the latency and memory bandwidth of L1, L2 caches, and the main memory of Phi; measured the performance of intra-node MPI functions (point-to-point, one-to-many, many-to-one, and all-to-all); and measured and compared the overhead of OpenMP constructs. Compared with [53], our work in this paper presents the results on single MIC device as well as on multiple MIC cards. Further, we discuss multiple variances of the native models and the offload models.

3.7 Conclusions

In this work, we conduct a detailed study regarding the performance and scalability of the Intel MIC processors under different parallel programming models. Between the two programming models, i.e., native MPIs on MIC processors and the offload to MIC processors, the native MPI programming model typically outperforms the offload model. It is very important to further improve the parallelism inside each MPI process running on a MIC core for a better performance. For embarrassingly parallel benchmarks such as Kriging interpolation, the multithreading inside

each MPI process can achieve 3 times speedup compared with the single-thread MPI implementation. Due to the fact that the physical distance between two MPI processes may be different under various MPI distributions, it is typically a good strategy to schedule MPI processes to as few MIC processors as possible to reduce the cross-processor communication overhead given the same number of MPI processes. Finally, we evaluate the hybrid MPI programming model, which is not officially supported by the Intel MPI compiler. Through benchmarking, it is found that the hybrid MPI programming model in which both CPU and MIC are used for processing is able to outperform the native MPI programming model. In the future work, we would test the performance when all of threads on the MIC are launched by MPI, which means that if m MIC are used, $m \times n \times 60$ single-threaded MPI processes are created in the parallel implementation.

Chapter 4

Towards Optimal Task Distribution on Computer Clusters with Intel MIC Coprocessors

In the chapter 3, we conduct a detailed study regarding different modes on Intel MIC processors. Given the benchmarks with regular kernels, they achieve a good performance and scalabilities. However, irregular kernel may have a complicated case. It may have the problem of load imbalance, which leads to an unsatisfactory performance.

In this chapter, we propose a dynamic distribution mechanism to resolve the imbalance of workload among many cores in the offload mode. In order to achieve the dynamic task distribution, all tasks will form a task pool. Through this manner, All cores will be kept busy in the whole computation process. We apply two additional optimization techniques to further improve the performance of applications on clusters with Intel MIC coprocessors. First, we design hybrid implementations to distribute tasks to both the CPUs and MICs. Second, we apply multiple-level parallelism technique to realize the concurrency among the N tasks as well as the concurrency in each task.

4.1 Introduction

Computer clusters with coprocessors/accelerators are typically leveraged to parallelize applications for reducing computation time. Given N parallel tasks and M processing cores, the typical strategy is to statically distribute those N tasks among M cores so that each core receives $\frac{N}{M}$ tasks. For example, given 1,024 tasks and 4 processing cores, core_0 takes tasks 0-255, core_1 takes tasks 256-511, etc. This kind of static task distribution is fine for applications in which each task requires almost the same amount of processing time. However, for many sophisticated applications such as sparse coding, it will be shown later that those parallel tasks require different amounts of processing times. In other words, some tasks will take longer time than others. The

static distribution will cause the cores with light tasks to wait for the cores with the heavy tasks, resulting in an imbalance in task distribution and the nonminimal overall processing time for the application. In such case, the workloads distributed to those cores will become uneven. In the previous example, if tasks 0-255 are quite heavy and tasks 256-511 are relatively light, then core_1 will take less time to process its tasks and then stays idle while waiting for core_0 to finish.

In order to resolve the imbalance of workload among those cores, it is better to distribute those tasks to the participating cores dynamically. In order to achieve the dynamic task distribution, all tasks will form a task pool. Every time a core will request an available task from the pool. Once it finishes the current task, it will go to request a new task from the pool. In this way, all cores will work together to finish the tasks in the pool and spend more or less the same amount of time on data processing.

Computer clusters typically contain both powerful multicore CPUs and massively parallel manycore coprocessors/accelerators. Therefore it is desired to distribute workload to both the host CPUs and the coprocessors to take advantages of both types of processors. Our results show that the hybrid implementation in which the workload is evenly divided into CPUs and MICs can almost double the performance compared with the implementation in which only the MICs are used.

We choose the sparse coding application as the benchmark in this work. Sparse coding is a class of unsupervised methods for learning bases to represent data efficiently. The aim of sparse coding is to find a set of basis vectors such that an input vector can be represented as a linear combination of these basis vectors. Unlike some other unsupervised learning techniques such as PCA [60], sparse coding can be applied to learn over-complete basis sets, in which the number of basis vectors is greater than the input dimension [38]. However, the high computational cost has seriously hindered its applications. Many times the size of the dictionary and the training example data have to be restricted due to concerns on the execution time of sparse coding algorithms. Especially, when it is applied on image processing, it may have millions of free parameters and

face the big data problem. Due to the complexity of the learning model, it may take weeks to learn the parameters using a single CPU [50]. Thus the sparse coding-based methods are good candidate benchmark for parallel processing on computer clusters. The results show that the dynamic task distribution can improve the performance by 25% compared with the static one. Further, the hybrid mode implementation involving both the host CPUs and the MICs can outperform the basic offload mode implementation by 40%.

The rest of the chapter is organized as follows. In Section 4.2 and 4.3 we introduce the sparse coding benchmark and various parallel implementations based on MIC. The experimental results and evaluation using the AUVs seafloor image data set are presented in Section 4.4. We conclude this work in Section 4.5.

4.2 The Benchmark: Sparse coding

Sparse coding is an algorithm for constructing succinct representations of input vectors such as images using the basis vectors in a dictionary [45]. Given a set of n 2D images, each image can be represented by a 1D array $\overrightarrow{x^{(i)}}$, $i = 1 \ldots n$. For example, if an image contains 32×32 pixels, the corresponding $\overrightarrow{x^{(i)}}$ will consist of 1,024 elements. The aim of sparse coding is to find a set of basis vectors $\overrightarrow{d^{(j)}}$ ($j = 1 \ldots k$) such that we can represent an input vector $\overrightarrow{x^{(i)}}$ as a linear combination of these basis vectors:

$$\overrightarrow{x^{(i)}} = \sum_{j=1}^{k} a_j^{(i)} \overrightarrow{d^{(j)}}, \tag{4.1}$$

in which each $a_j^{(i)}$ is a scalar coefficient. If we use D and $\overrightarrow{a^{(i)}}$ to represent $[\overrightarrow{d^{(1)}}, \overrightarrow{d^{(2)}}, \ldots \overrightarrow{d^{(k)}}]$ and $[a_1^{(i)}, a_2^{(i)}, \ldots a_k^{(i)}]^T$, respectively, we can re-write Equation (4.1) as Equation (4.2) .

$$\overrightarrow{x^{(i)}} = D\overrightarrow{a^{(i)}}. \tag{4.2}$$

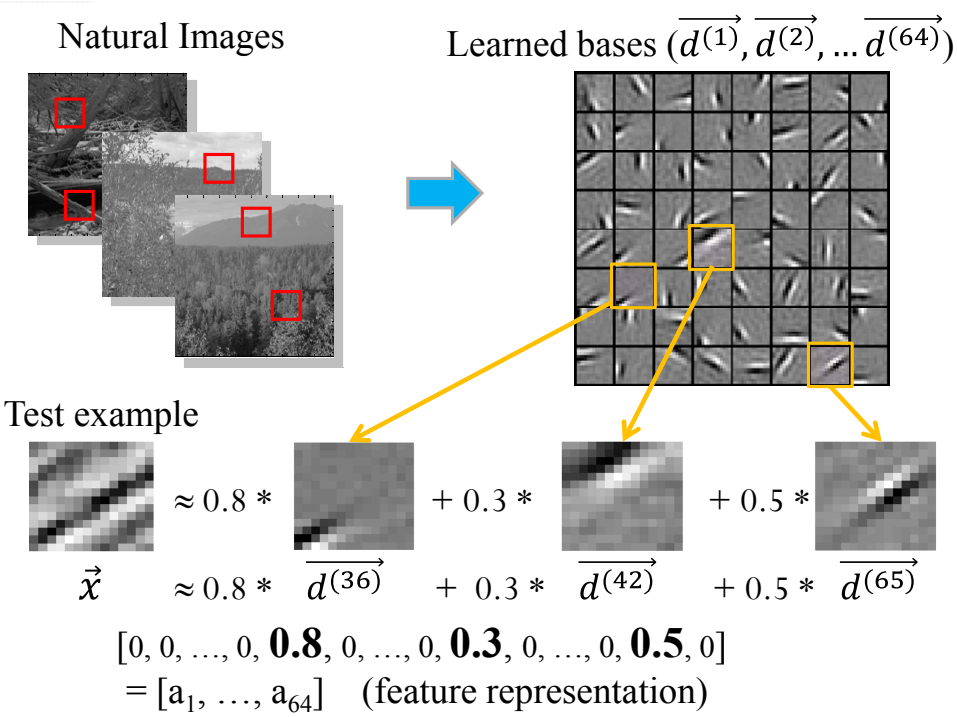


Figure 4.1: Illustration of sparse coding applied to natural images [37]. In the upper left, small red squares represent 14×14 image patches randomly sampled from natural images. The upper right shows the learned bases from natural images via sparse coding. The below illustrates how sparse coding decomposes a new 14×14 image patch into a linear combination of a few basis vectors. The resulting sparse coefficients can be used as features representing 14×14 pixels. (Note: although illustrated in 2D images, all the dictionary bases and test images are represented as 1D vectors in sparse coding algorithm.)

Further, if we use X and A to represent $[\overrightarrow{x^{(1)}}, \overrightarrow{x^{(2)}}, \ldots \overrightarrow{x^{(n)}}]$ and $[\overrightarrow{a^{(1)}}, \overrightarrow{a^{(2)}}, \ldots \overrightarrow{a^{(n)}}]$, respectively, we want to represent X as

$$X = DA. \tag{4.3}$$

Each column of D (i.e., $\overrightarrow{d^{(j)}}$) is regarded as a basis in the directory. Each column of A (i.e., $\overrightarrow{a^{(i)}}$) is the sparse representation of the corresponding input vector $\overrightarrow{x^{(i)}}$ according to the dictionary. Most components of $\overrightarrow{a^{(i)}}$ should be zero's. One example illustrating how to represent the input image using the bases in the dictionary is shown in Figure 4.1.

The basic idea of sparse coding is defined as finding two matrices D and A and aims to solve

Table 4.1: Computation time of four steps in a sequential implementation on an Intel Xeon E5606 2.13-GHz CPU.

Size of Image Patch	Step	Computation Time (s)
32×32	Load image	5.93
	Optimizing A	14,811.29
	Optimizing D	208.32
	Write results	1.8

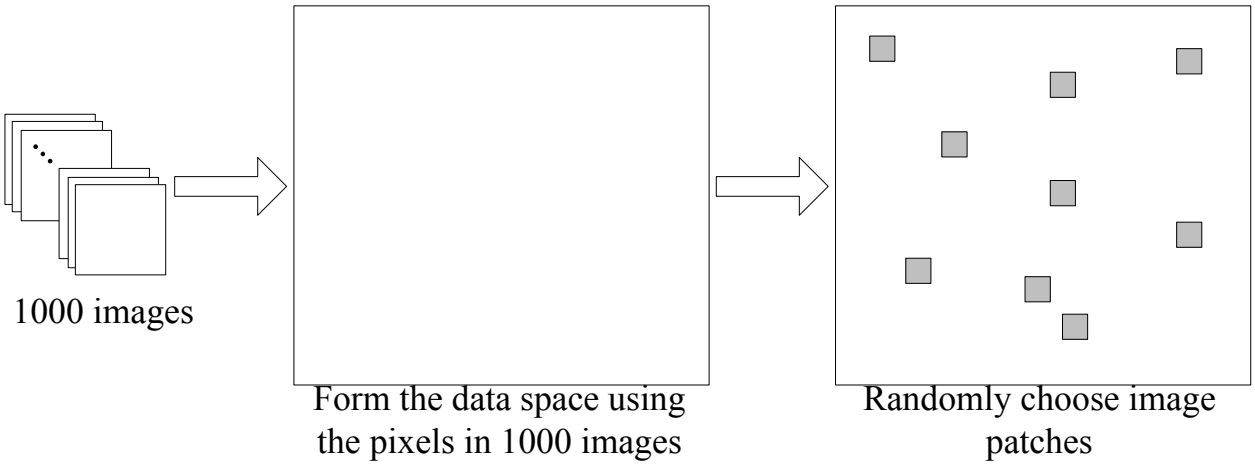


Figure 4.2: The process to form the source data space and randomly choose several data sets.

the following optimization problem:

$$\min_{A,D} \sum_{i=1}^{n} \left(\left\| \overrightarrow{x^{(i)}} - \sum_{j=1}^{k} a_j^{(i)} \overrightarrow{d^{(j)}} \right\|^2 + \lambda \sum_{j=1}^{k} |a_j^{(i)}| \right) \tag{4.4}$$

It has been shown that the optimization problem is not jointly convex in both A and D, but it is convex in either A or D if the other one is kept fixed. An alternating minimization algorithm has bee proposed in [38], as shown in Algorithm 5. When facing large numbers of images and dictionary bases, the step of optimizing over A is particularly time consuming since it involves an uncertain objective function. The overwhelmingly dominant computational effort is spent on optimizing A by solving an L_1 regularized least squares problem. The entire learning process is divided into four steps, including Load image, Optimizing A, Optimizing D, and Write results.

ALGORITHM 5: Learning the dictionary in sparse coding.

```
Transfer a large number of images into global memory;
Select a group of data sets randomly;
Initialize the dictionary D randomly;
while (convergence criterion is not satisfied) do
    for (each data set in the group) do
        Set A ← D^T X, and normalize A;
        Keep D fixed, optimize over A by solving an L1 regularized least squares problem;
        Keep A fixed, optimize over D by using convex optimization techniques;
```

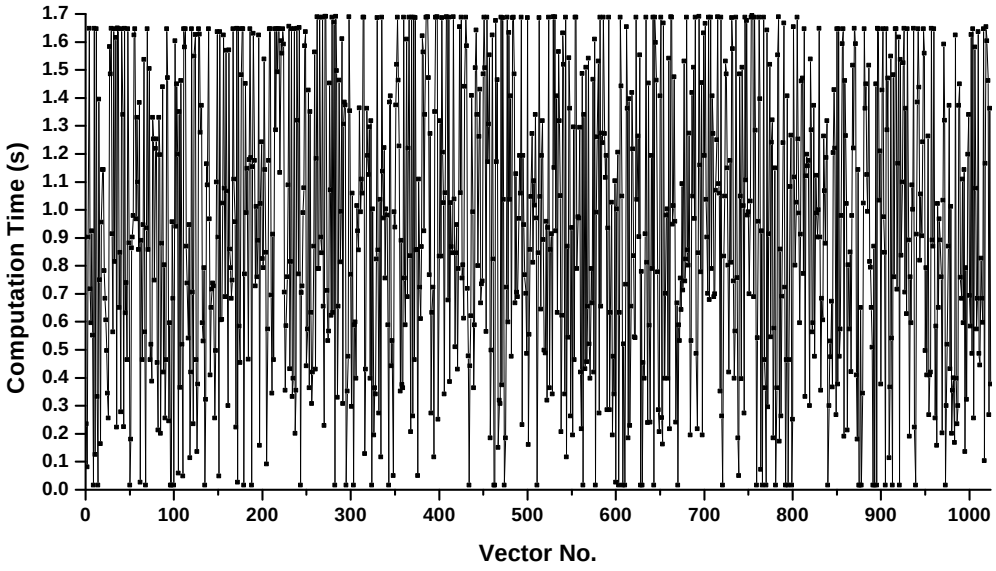


Figure 4.3: The computation times of 1,024 coefficient vectors in the sequential implementation on CPU.

We conducted a simple test to check the computation times spent on each step. The original data set is comprised of 14 dive missions conducted by the AUV Sirius off the southeast coast of Tasmania in October 2008 [10, 9]. It contains over 100,000 stereo pairs of images. Marine scientists used the CPCe software package [31] to label 50 random points on each image with different class labels, such as biological species (including coral, algae and others), abiotic elements (sand, gravel, rock, shell, etc.), and other unknown data types. In this test, we chose 1000 1,024×1,024 images from the original data set. Then we randomly chose 5 sets of image patches. Each set contains 1,024 patches, each of which is of 32×32. The process to form the source data space and choose the data sets is illustrated in Figure 4.2. Once the 5 sets of image

patches were generated, we applied the optimization on them using Algorithm 5. In this simple test, we only carried out 3 iterations of the `while` loop in Algorithm 5. This implementation was a single-thread implementation written in C and executed on an Intel Xeon E5606 2.13-GHz CPU. The computation times on all steps are shown in Table 4.1. It is obvious that the step of optimizing A is the most time-consuming step.

Raina et al. have presented a method to optimize the computation in the Equation (4.4) [50]. Let $\overrightarrow{d_j}$ (i.e., $\overrightarrow{d^{(j)}}$) be the j^{th} column of the dictionary D and set $r_j = \overrightarrow{d_j}^T \overrightarrow{d_j}$, the new optimal value $a_j^{(i)}$ for input $\overrightarrow{x^{(i)}}$ can be calculated as follows.

$$a_j^{(i)} = \begin{cases} 0 & \text{if } |g_j - r_j a_j^{(i)}| \leqslant \beta \\ (-g_j + r_j a_j^{(i)} + \beta)/r_j & \text{if } g_j - r_j a_j^{(i)} > \beta \\ (-g_j + r_j a_j^{(i)} - \beta)/r_j & \text{if } g_j - r_j a_j^{(i)} < -\beta \end{cases} \tag{4.5}$$

where g_j is the j^{th} component of vector $\overrightarrow{g}$, which is as follows.

$$\overrightarrow{g} = \nabla_a \frac{1}{2} \left\| \overrightarrow{x^{(i)}} - \sum_{j=1}^{k} a_j^{(i)} \overrightarrow{d^{(j)}} \right\|^2 = D^T D \overrightarrow{a^{(i)}} - D^T \overrightarrow{x^{(i)}} \tag{4.6}$$

In our implementation, each $\overrightarrow{a^{(i)}}$ will be updated for 150 times in one iteration of the optimization of A while D is fixed.

Based on the algorithm, there are two levels of parallelism we can take advantages of.

In the first level (**level-1**) of parallelism, it can be found that all the columns of A, i.e., $\overrightarrow{a^{(i)}}$'s, can be optimized independently. From Equation (4.6), it can be found that only $\overrightarrow{x^{(i)}}$ is used to update $\overrightarrow{a^{(i)}}$. Therefore, the updating of different $\overrightarrow{a^{(i)}}$'s can be handled by different processing cores. However, the computation times of those coefficient vectors vary quite significantly. Figure 4.3 shows the computation times of 1,024 vectors in our benchmark. The times are in the range of $[0.016s, 1.694s]$. A static distribution of those 1,024 vectors will result in an imbalance of computation among participating processing cores.

In the second level (**level-2**) of parallelism, the updating of 1,024 components in each $\overrightarrow{a^{(i)}}$ can be parallelized as well. In our implementation, the updating of one $\overrightarrow{a^{(i)}}$ is a combination of sequential stages and parallel stages. Those parallel stages can be carried out by multiple cores.

4.3 Parallelization using MIC

The task of learning dictionary is a cycle of alternating process. Given a set of images, they will be transferred into the global memory. To ensure higher levels of concurrency, we split the basis learning and propose the parallel sparse coding using both native and offload parallel programming models on the Intel MIC accelerated computer clusters. The computation can be parallelized where each MIC device works on a single subset inputs, and each thread in the device works on a single subtask. Finally, the result will be gathered over all devices. The parallel sparse coding can be implemented with the following execution modes:

- **Native-1**: i.e., the native model. In this implementation, the MPI process is directly executed on each MIC core. For a group of N work items, if we allocate M MIC cards, each card is assigned the same number of work items, i.e., $\frac{N}{M}$. These $\frac{N}{M}$ work items are scheduled to the same number of MIC cores, each of which only executes a single-thread MPI process. For sparse coding, only the level-1 parallelism is realized.
- **Native-2**: On top of Native-1 execution mode, we try to take advantage of the internal processing parallelism on each MIC core. Therefore, we launch 4 threads in each MPI process using OpenMP. Each MPI process is still run on a MIC core. In this mode, the work items are assigned to MIC cards at first. Just like Native-1 mode, each card will execute 60 MPI processes. Further, the subtasks in each work item will be parallelized by launching 4 threads in each MPI process. Therefore, this mode will provide two levels of parallelism: parallel MPI processes for the level-1 parallelism and OpenMP threads in each MPI process for the level-2 parallelism.

- **Offload**: In this mode, the MPI processes are hosted by the CPU cores, which offload the computation including data to the MIC processors. The communication among MPI processes is handled by CPUs. The host MPI process on CPU issues multiple threads to the MIC card using OpenMP so that each thread works on one or more coefficient vectors depending on the number of participating MIC cards. If several coefficient vectors are assigned to one thread, they will be processed in a sequence. Then we apply a second level of parallelism, i.e., spawning multiple threads in each thread. Given the 240 threads that can be physically executed in parallel on the Xeon Phi 5110P device, various configurations can be applied. If we use M and N to represent the level-1 and level-2 threads, respectively, different combinations of (M,N) can be adopted subject to $M \times N \leqslant 240$. This mode can be further categorized into two sub-modes, i.e., **Offload-S** for static task distribution and **Offload-D** for dynamic task distribution.

- **Hybrid**: In this mode, both CPUs and MICs are allocated for data processing. First the workload is distributed to CPUs through MPI. Then a host CPU will offload part of the workload to a MIC card using OpenMP. On the host CPU, we also use OpenMP to spawn multiple threads for parallel processing. The peak performance of an Intel MIC 5110P is around 2,022 GFLOP/s, which is about 4 times the performance of a single thread on the host CPU (i.e., Intel Xeon E5-2670). Therefore we run 4 threads on the host CPU and evenly divide the workload between a host CPU and its corresponding MIC processor. The hybrid mode also has two variants, **Hybrid-S** and **Hybrid-D**.

4.4 Results and discussion

We conduct the experiments to use multiple MIC processors to demonstrate the scalability and performance of the learning dictionary process, which is the most time-consuming part in sparse coding methods. In our experiments, we mainly evaluate the following three aspects: the performance scalability of the native modes, and the performance improvement due to dynamic

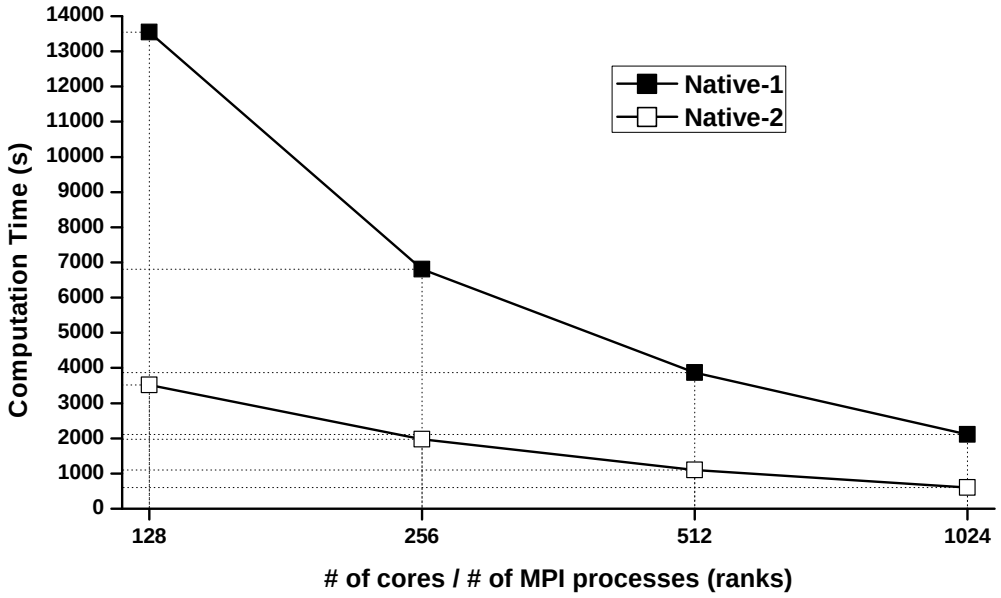


Figure 4.4: Performance scalability under the native execution modes. 60 MPI processes (ranks) are scheduled to one MIC card.

task distribution on a single MIC card as well as on multiple MIC cards.

4.4.1 Performance scalability of the native modes

We implemented two native parallel execution modes, i.e., Native-1 and Native-2, on the Beacon computer cluster using multiple nodes. The image size is 1,024×1,024. Then we randomly chose 5 sets of data as shown in Figure 4.2, and applied the optimization on them using Algorithm 5. Only 3 iterations are carried out in the experiment.

In order to show the strong scalability of the parallel implementations, the problem size is fixed for each implementation while the number of participating cores is increased. For the native modes, each MIC will host one MPI process. We schedule 60 MPI processes to a MIC card.

From Figure 4.4 we can see that both implementations in native mode keep the strong scalability, i.e., the computation time halves when the number of processing cores doubles. Because the Native-2 implementation uses 4 times of threads as the Native-1 implementation, its performance is approximately 4 times better than the Native-1 mode.

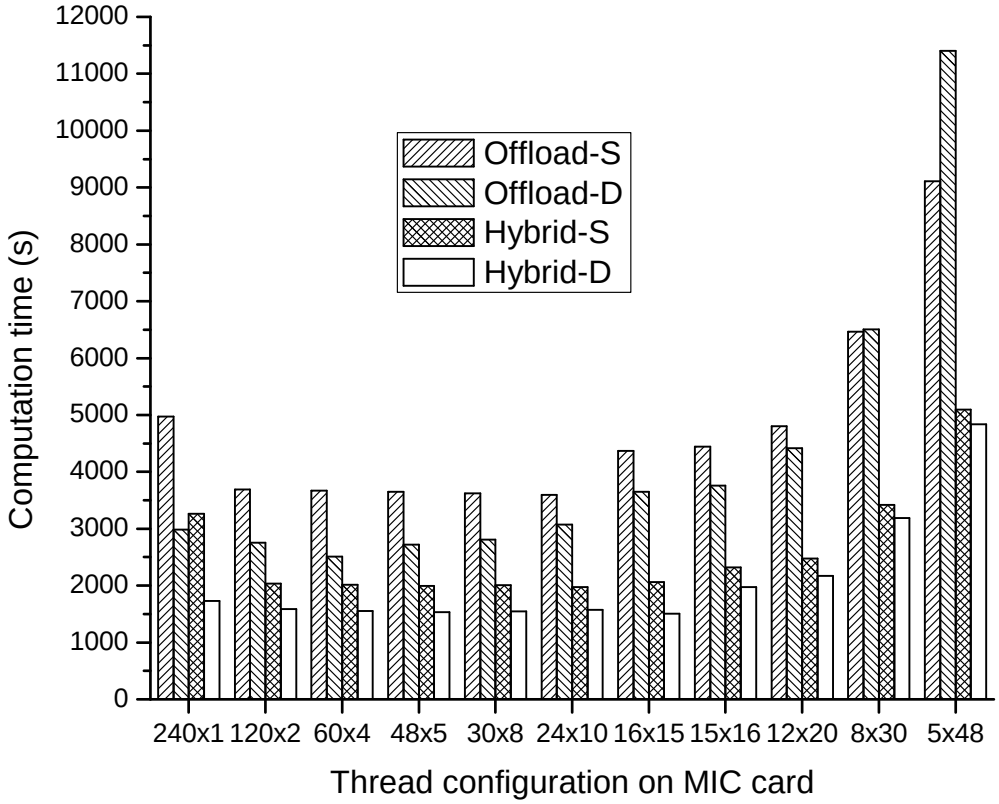


Figure 4.5: Performance comparison between static distribution and dynamic distribution as well as between offload mode and hybrid mode on a single MIC card. $M \times N$: M is the number of threads offloaded to a MIC card for realizing the level-1 parallelism, each thread further spawns N threads to achieve the level-2 parallelism. 4 threads are created on the host CPU for the level-1 parallelism.

4.4.2 Performance improvement of dynamic task distribution on a single MIC card

We first show the benefit of the dynamic task scheduling compared with the static task scheduling on a single MIC card. Because there are multiple combinations of the number of the threads scheduled to a MIC card and the number of threads spawned by each thread, we list 11 options in Figure 4.5. When there are many level-1 threads, the advantage of dynamic distribution is very evident. For example, when there are more than 30 level-1 threads, the performance improvement is more than 25%. When the number of level-1 threads decrease, the difference between the static distribution and dynamic one diminishes. There are two reasons. First each level-1 thread will receive many tasks in both distributions. This will reduce the imbalance of workload among those threads. Second, because the computation of one coefficient vector

Table 4.2: Thread configuration in multiple-MIC implementations.

Number of MIC cards	Offload-S	Offload-D	Hybrid-S	Hybrid-D
2	24×10	60×4	60×4	60×4
4	24×10	60×4	60×4	60×4
8	24×10	60×4	60×4	60×4
16	24×10	60×4	60×4	16×15
32	24×10	30×8	16×15	16×15
64	16×15	16×15	16×15	16×15
128	16×15	16×15	–	–

consists of both sequential stages and parallel stages, the sequential stages will become a limiting factor for performance improvement when a large number of level-2 threads are spawned. More level-2 threads will not further improve the performance once the number of level-2 threads reaches a certain point.

Figure 4.5 also shows that the hybrid mode can improve the performance by more than 40% compared with the offload mode. In this implementation, we artificially distribute the total workload evenly between the host CPU and the MIC coprocessor. Therefore we only issue 4 threads on the CPU for pairing with one MIC card. The host CPU, Xeon E5-2670, contains 8 cores and can execute 16 threads physically. In theory, we can schedule more work to the host CPU. This reminds us that it is very important to schedule workload to both the host CPUs and the coprocessors for achieving the best performance. The ratio of the workload between these types of processors needs to be carefully assigned for realizing the balance. Between the two variants of the hybrid mode, the comparison is similar to the case of the offload mode. The Hybrid-D mode can improve the performance by 25% compared with the Hybrid-S mode when the number of level-1 threads is abundant.

4.4.3 Performance improvement of dynamic task distribution on multiple MIC cards

We also conduct the experiment by using multiple MIC cards. The thread configurations for various implementation modes are different and listed in Table 4.2. The performance saving trend

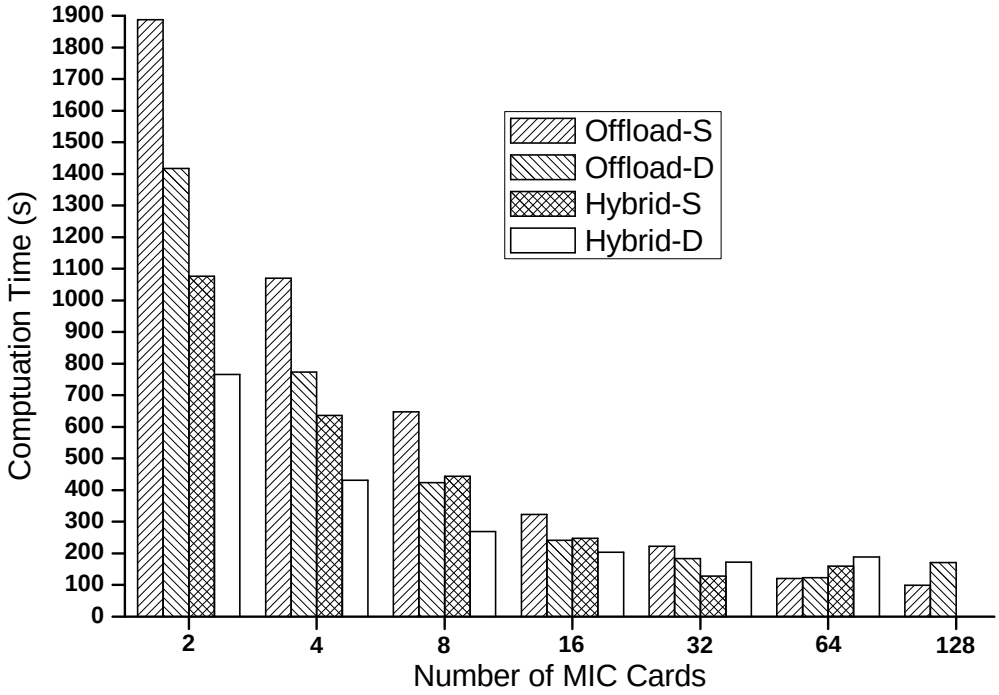


Figure 4.6: Performance comparison between static distribution and dynamic distribution as well as between offload mode and hybrid mode on multiple cards. For hybrid modes, 4 CPU threads are scheduled for one MIC card. Two MIC cards are hosted by one CPU, which will execute 8 threads. Therefore, the ratio between the number of host CPUs and the number of MIC cards is 1:2.

is very similar to the one-MIC-card case. As shown in Figure 4.6, for both offload mode and hybrid mode, the dynamic task distribution is able to improve the performance by more than 25% when the number of MIC cards is less than 16. When more MIC cards are added, the communication overhead will become dominant. Both hybrid implementations are able to outperform their offload counterparts by around 40% when there are significant amount work scheduled to each MIC card and CPU (i.e., the number of MIC cards is less than 16). Beyond that, the performance gain due to hybrid implementation starts diminishing due to the introduced communication overhead.

4.5 Conclusions

In this work we conduct a detailed study regarding the performance and scalability of various parallel execution modes on computer clusters with Intel MIC coprocessors. The tasks in an application are typically distributed to multiple nodes for parallel processing. However, for sophisticated applications such as sparse coding, those parallel tasks may require different processing times. A static task distribution among working threads may introduce an imbalance of workload among them. In order to improve the overall performance, we leverage the dynamic task distribution so that a thread will request a new task from a task pool once it finishes the processing of the current task. Experiment results show that this technique can improve the performance by 25%. In addition, we try to leverage the processing power of the host CPU by implementing the sparse coding algorithm in a hybrid mode. The results show that the hybrid implementation is able to further improve the performance by 40%.

At this moment, the dynamic task distribution is only applied through OpenMP when the host CPU offloads tasks to the MIC card and processes its own tasks. The first level task distribution through MPI is still static. As the future work, we plan to investigate how to realize the dynamic task distribution in the first level.

Chapter 5

Performance Optimization on Intel MIC Through Load Balancing

In the chapter 4, we propose a dynamic distribution mechanism to resolve the imbalance of workload among many cores. Besides, a hybrid mode, processing data on both processors and coprocessors, is proposed to evenly divide the workload between a host CPU and its corresponding MIC processor based on their peak performance. However, in parallel and distributed computation, data communication between computing nodes may be required in different scenarios. Before the computation is implemented, partial data may have to be shared or exchanged among the distributed nodes. One significant issue in data communication is the amount of data exchanged that may have an significant impact on the total performance. Therefore, we cannot simply decide the distribution of data via the peak performance of processors/coprocessors.

This chapter proposed smart data distribution model relies on the accurate performance profiling as the parameter to allocate the amount of input data to different types of processors and coprocessors based on their computing capabilities to achieve the load balance. A performance profiling is launched to find the computing capabilities of host processors and MIC coprocessors. Then we launch single-threaded MPI processes to both processors and coprocessors. An MPI process will check on which type of processor/coprocessor it is running and requests the amount of data accordingly. In other words, the amount of data an MPI process will process is directly proportional to the computing capability of the hosting processor/coprocessor.

5.1 Introduction

Emerging computer architectures and advanced computing technologies, such as Intel's Many Integrated Core (MIC) Architecture [5] and graphics processing units (GPUs) [28], provide

a promising solution to employ parallelism for achieving high performance, scalability and low power consumption. However, the heterogeneous environment may have the problem of load imbalance, which leads to an unsatisfactory performance. MPI programs may synchronize frequently. The faster ranks will idle at the synchronization point waiting for the slowest rank to finish. A heterogeneous supercomputer contains processors of various computing capabilities, such as the host Xeon CPUs and the Xeon Phi coprocessors. A single core on the Xeon CPU typically outperforms a single core on the MIC. If we run MPI programs on both Xeon CPU and MIC and distribute the data among all MPI processes (or MPI ranks) evenly, the MPI processes running on Xeon CPU will idle at the synchronization point waiting for the processes running on MIC. In order to achieve a good performance, load imbalance has to be minimized in the heterogeneous environment.

We use cellular automata (CA) as the benchmark to test the performance and scalability of different programming models on a heterogeneous computer server with MIC coprocessors. We also develop an urban sprawl simulation on MIC and do a comparison of performance between MIC and GPU.

The remainder of this chapter is organized as follows. We discuss some related work in Section 5.2. The four models based on Intel Xeon Phi and a new data distribution model(DDM) are discussed in Section 5.3 and Section 5.4. Section 5.5 introduce the details about implementation of game of life under different programming moels. Section 5.6 shows experiment results and compare the performance under different programming models. In Section 5.7, we demonstrate the benefit of Xeon Phi by presenting an urban sprawl simulation case. Finally, we give the concluding remarks in Section 5.8.

5.2 Related Work

Heterogeneous multiprocessors have been drawing increasing attentions from both the hardware and software research communities. Load balancing in heterogeneous environment is a

critical task in order to get high performance. The hardware heterogeneity makes it difficult to ensure reasonably uniform resource utilization, thus leading to performance losses due to load imbalance [13]. In the heterogeneous programming systems, static and dynamic models are used to solve the problem of load imbalance.

[63] and [64], which are based on data partitioning, needs information about the application and heterogeneous platform. This information can be gathered at both compilation time and execution time. Static methods depend on accurate performance model to predict the future execution of the application. Static load balancing can lead to very high performance only after several runs of the benchmark. This process determines the correct ratio of data distribution between the host processor and the coprocessors. Static methods are particularly useful for applications that have good data locality because they do not require data redistribution. However, when the data load changes over the time, they are unable to achieve the load balance between different types of processors/coprocessors [36].

[33] and [7], which are based on task scheduling and work stealing, balance the load by moving fine-grained tasks between processors and coprocessors during the calculation. These methods do not require the information of applications and heterogeneous platforms. Dynamic methods often use static partitioning for their initial step due to its communication cost, bounded tiny load imbalance, and smaller scheduling overhead [39]. A dynamic load balancing approach allows the users to manage unexpected performance perturbations in a transparent way. Although it may lead to significant communication overhead due to data migration, the communication overhead can be effectively hidden by a software pipelining technique, which is particularly useful for large memory-bound applications [63].

5.3 Programming models

5.3.1 Native Model (MPI-based implementation)

Native execution occurs when an application runs entirely on Intel Xeon Phi coprocessors and no job is dispatched on the host Xeon CPUs. Applications that are already implemented by MPI can use this model by distributing MPI ranks across the coprocessors natively. The MPI library is designed to support a program running on a heterogeneous set of nodes. In the native model, MPI can run on the coprocessors without modifying any source code. Each MIC core directly hosts n (up to 4) MPI processes. Therefore, if m Xeon Phi coprocessors are used, $m \times n \times 60$ single-threaded MPI processes are created in the parallel implementation.

The native model avoids the complex architectural heterogeneity on a heterogeneous supercomputer. However, the MPI program only runs on Xeon Phi coprocessors. It does not take advantage of the compute capacity of the host Xeon CPUs. It is supposed to be noticed that the number of MPI processes on a Xeon Phi coprocessor may be fewer than the maximum 240 processes it can host because of the limited memory on the coprocessor and the communication overhead among all MPI processes.

5.3.2 Symmetric Model

Native model only makes MPI programs run on the coprocessor cores. This model leaves the Intel Xeon processors unutilized, which means this approach does not take advantage of the computing capacity of the host Xeon CPUs and is likely to give up too much performance potential of the whole system. In the symmetric model, the program runs on both the processors and coprocessors. The MPI ranks reside on the host CPU and the MIC coprocessors. If m MIC (Xeon Phi) coprocessors, in which each MIC core directly hosts n (up to 4) MPI processes, are used in addition to k MPI processes on the CPUs, $m \times n \times 60 + k$ single-threaded MPI processes are created in the parallel implementation.

Many MPI programs were written with the implicit assumption that they will run on homogeneous systems in which each MPI rank computes at the same rate. These programs decompose the problem into even parts to compute so that all MPI processes can synchronize at some points without waiting for a long time. However, when some ranks are hosted on processors/coprocessors that have a stronger computing capability, the slowest rank will determine the overall computation rate. This is a typical load imbalance problem.

5.3.3 Hybrid Model

On top of the symmetric programming model, a hybrid model makes multithreading inside each MPI process on MIC. When MPI processes are scheduled to both host Xeon processors and the MIC coprocessors, a single-threaded MPI process on MIC coprocessor has a weaker performance than a single-threaded MPI process on Xeon processor. Given the same amount of data, the MPI process on MIC will take a longer time than the MPI process on the Xeon processor to finish. In order to improve the performance of MPI processes on MIC, multiple internal threads are launched. This model can solve the load imbalance problem and improve the performance for parallel applications by carefully adjusting the extent of multithreading inside the MPI processes on MIC coprocessors.

5.4 Data Distribution Model (DDM)

5.4.1 DDM Analysis

In above three models, each MPI process will receive the same amount of data no matter it is single-threaded or multiple-threaded. In the hybrid model, a careful tuning is required to have equivalent performances on processes on different processors/coprocessors.

Single-threaded MPI processes are scheduled to both the host processors and the coprocessors. Based on the computing capabilities of the hosting processors/coprocessors, a

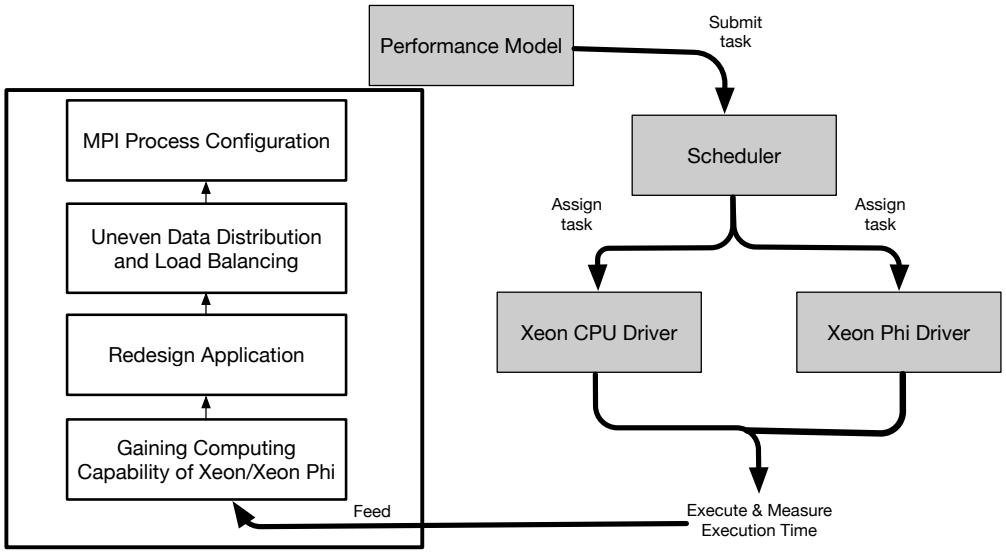


Figure 5.1: DDM workflow.

proportional amount of data will be given to an MPI process so that all MPI processes will finish the data processing in more or less the same amount of time. The high performance of parallel applications on heterogeneous platforms can be achieved by partitioning the computational load unevenly across processors/coprocessors. This model relies on accurate performance models as a reference to partition data among MPI processes with different computing capabilities, as shown in Figure 5.1. We submit the same amount of task into host processor and MIC coprocessor and measure their running time. According to their performance, we redesign the application and decompose the problem into unevenly parts so that all MPI processes can synchronize at some points without waiting for a long time.

In parallel and distributed computation, data communication between computing nodes may be required in different scenarios. Before the computation is implemented, partial data may have to be shared or exchanged among the distributed nodes. One significant issue in data communication is the amount of data exchanged that may have an significant impact on the total performance. Therefore, we cannot simply decide the distribution of data via the peak performance of processors/coprocessors. Instead, a performance profiling by running the benchmark on the real platform is required to decide the precise data distribution among various

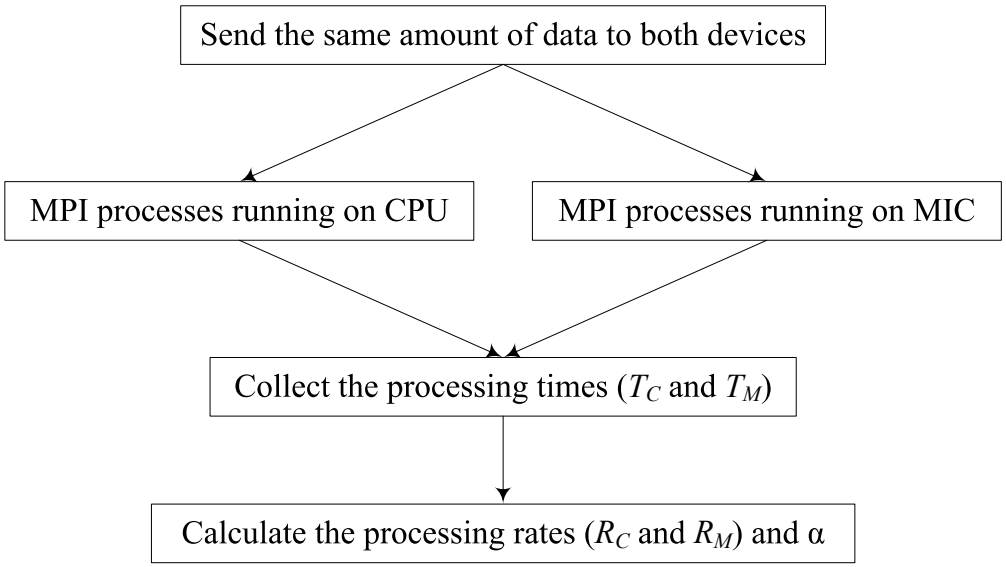


Figure 5.2: The process to calculate α in the DDM model.

types of processors/coprocessors.

For achieving a load balance among heterogeneous nodes, optimal distribution of data between heterogeneous processors is typically based on their computing capability to the kernel, which can be either pre-built or being built during the execution of the application for each processor. In this study, we use ratio α to get the optimal distribution.

Figure 5.2 shows our workflow for calculating ratio α. When pre-built application is executed for the first time over N iterations, the runtime distributes N w_p iterations to the MIC, where w_p is a ratio between 0 and 1, and measures the rate M_r at which the MIC processes the application. Concurrently, the runtime executes another N w_p iterations on the multicore CPU and computes the CPU rate C_r.

One MPI process collects and compare the two rates M_r and C_r after completing the calculation. The runtime distributes the remaining iterations to the CPU and MIC based on the calculated α to verify their rates.

Let α become the ideal ratio for distribution to the MIC, and the remaining $1 - \alpha$ to the host CPU. We can represent the rate with which MIC executes workload as M_r and CPU as C_r. The α

can be found that :

$$\frac{N\alpha}{M_r} = \frac{N(1-\alpha)}{C_r}$$

Where N is the total number of workload for pre-build model. When M_r and C_r are achieved by the first execution, α can be derived by the following formula:

$$\alpha = \frac{M_r}{M_r + C_r}$$

5.4.2 Determining profiling size

In the data distribution model, it is important to determine the right number of iterations or work items of pre-built application. If we select too many iterations or work items, it would increase the overhead of waiting time, since one MPI process may have to wait for the other processes to complete execution at synchronous point. On the other hand, if we determine too few iterations, it is hard to know the computing capabilities of devices to kernel function. Therefore, the work items of pre-build application should be large enough to fully utilize all the available MIC resource.

It is well-known that MPI-based model is very effective for exploiting parallelism in regular programs, such as operating on large vectors or matrices. These programs often contains high computational demands, exhibits extensive data parallelism and require little synchronization. A large number of algorithms from important application areas fit these criteria, including algorithms used in urban simulation and interpolation. There exists a broad base of knowledge on the efficient parallelization of these algorithms[35], and their MIC implementations can be tens of times faster than single-thread CPU version.

Figure 5.3 demonstrates the impact of different number of work items to workload distribution between processor and coprocessor, α. The kernel function is to calculate the sum of

Figure 5.3: Changes in ratio α with increasing number of work items

integers from 1 to 10 per work-item. 240 MPI processes are running on a MIC card and 8 MPI processes are running on the host CPU. As we increase the number of work items on each MPI rank, α consistently increases when the MIC takes the same amount of time to process more data and the rate stabilized at 2048 items. As a result, the MIC is not fully utilized in processing less than 2048 items. In order to determine a right execution rate, we have to ensure fully utilization of the MIC as a prerequisite.

The programs containing regular kernels are effective for parallelism. However, some problem domains employ algorithms that include irregular data structures such as trees, graphs, and priority queues. Irregular programs are more different to be parallelized and easier to generate load imbalance problem between the host processor and MIC coprocessor. Besides, MPI processes running on MIC card may have this problem, since these MPI processes have different amount of calculation and may have to wait for the other processes to complete execution at synchronous point. In the unpredictable calculation, MPI-based programming model seems to be unable to solve the problem. In the following section, we will continue discussing this problem and provide a solution based on the offload mode.

5.4.3 Asynchronous Management Controller(AMC)

Full utilization of resources requires a highly asynchronous and parallel model. Asynchronous message passing allows more parallelism and significantly hides overhead. Since a process does not block, it can do some computation while the message is in transit. A synchronous operation blocks a process till the operation completes. An asynchronous operation is non-blocking and only initiates the operation.

One limitation in MPI applications is that the standard does not force progress during asynchronous MPI calls which ultimately results in not achieving efficient overlap of communication with computation. For example, if an application posts an MPI_Irecv followed by an MPI_Isend and then performs computations before posting the corresponding MPI_Wait operation, it is often the case that communication will not be overlapped with the computation. This is because the implementation of the send operation requires the corresponding receive to be posted on the target before data transfer is initiated. Applications may alleviate this by periodically calling MPI_Iprobe or MPI_Test to make progress on asynchronous calls. However, it becomes extremely difficult to determine the optimal frequency of these calls to achieve the desired overlap. This situation is exacerbated on the Intel Xeon Phi as if one thread frequently calls into the MPI runtime, significant load imbalance can be created which can severely degrade performance.

To address this limitation, we use an AMC(Asynchronous Management Controller) to eliminate this issue by using a dedicated CPU core or MIC core that constantly attempts to make progress on asynchronous calls and thus achieves the desired overlap without creating any load imbalance, as shown in Figure 5.4.

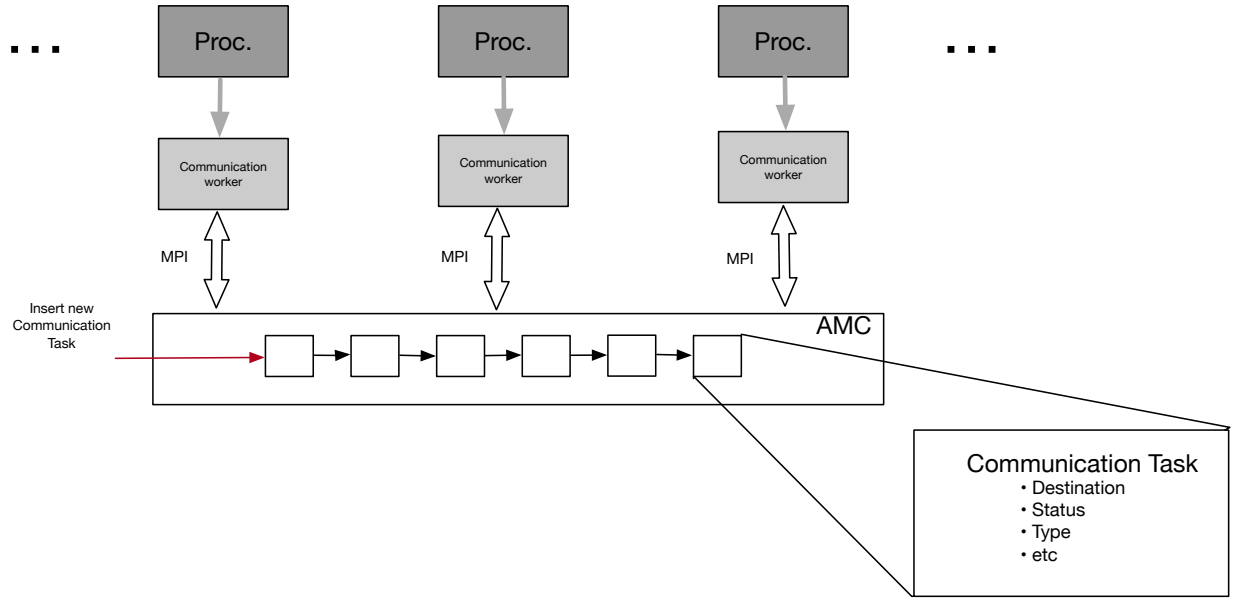


Figure 5.4: AMC Structure.

5.4.4 Asymmetric scheduling

In this section, I would describe our asymmetric scheduling algorithm that addresses the overheads of the DDM method, and provides adaptive strategies that handle the load imbalance problem.

In the DDM method, we first assign a part of workload to host processor and MIC coprocessor and then determine their computing capabilities to the kernel function. The overhead in this method would be caused by waiting on a barrier when one of the devices completes the execution before the other. Figure 5.5 demonstrates the workflow of the asymmetric method. We submit the job into host processor. Host processor offloads a part of workload to MIC coprocessor and continues the execution of other workload until it observed that a kernel running on the accelerator has completed. According to their performance, we redesign the application and decompose the problem into unevenly parts so that all MPI processes can synchronize at some points without waiting for a long time.

Figure 5.6 shows our workflow for calculating ratio α. We set up shared pool of workload

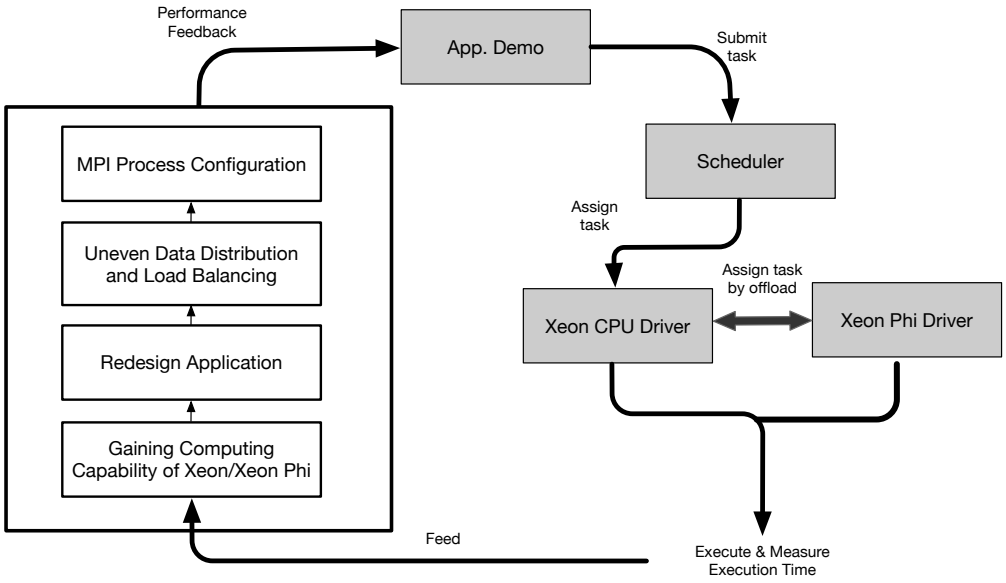


Figure 5.5: Symmetric model.

where it contains the entire the parallel iteration space. And then we select a part f_p of items and offload them into MIC coprocessors. MIC launches 240 threads to do kernel calculation though OpenMP. The host processor continues the execution of other workload and pick items from the shared global pool until MIC completes the kernel execution.

When the OpenMP threads in MIC devices finish executing kernel function, host processor would collect and compare the two rates M_r and C_r after completing the calculation. The runtime distributes the remaining iterations to the CPU and MIC based on the calculated α to verify their rates. The calculation of distribution ratio α has been discussed in Chapter **??**. It is important to note that while the MIC coprocessor is executing, the host CPU continues working from the global shared pool, thereby eliminating the overhead seen in DDM method.

In the asymmetric model, our design choice is to use a global shared pool to distribute work items to the processor and coprocessor. We do not use the work-stealing technique since it is hard to partition the work among CPU threads and MIC threads. Besides, we have to consider the overhead of transferring data between the CPU and MIC. If we find a good profiling size that keeps MIC fully utilized with the smallest amount of data, it can be seen that the asymmetric method has a less overhead than DDM method. Figure 5.7 demonstrates the overhead of DDM

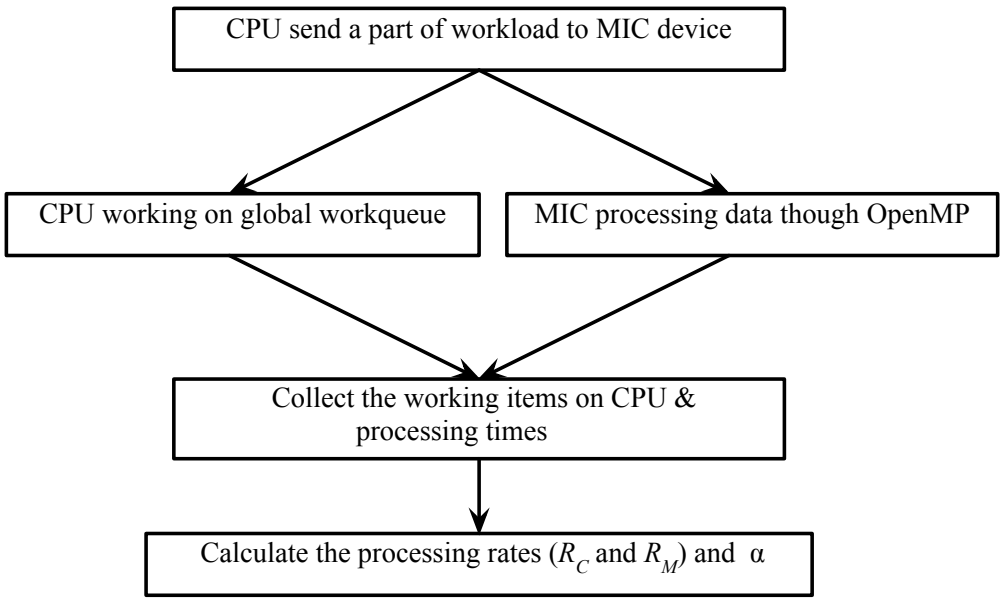


Figure 5.6: The process to calculate α in the Asymmetric model.

method and asymmetric method. Game of life with different grids, 8192×8192, 16384×16384 and 32768×32768, are used to test the both methods It can be found that the overhead of asymmetric method consistently reduces than the DDM method.

5.5 The implementation of Game of Life by different models on heterogeneous clusters with MIC coprocessors

The game of life has been disscussed in Chapter 3, as shown in Algorithm 7. We will discuss the implementation details of Game of Life using a single MIC coprocessor. In order to utilize the computing resources available in the Intel Xeon Phi coprocessor, the proposed approach employs task parallelism which utilizes more than 200 mic cores, and data parallelism which uses efficiently vector processing units to process one operation on multiple pairs of data at once. For using multiple MIC coprocessors, the implementation on a single MIC card will be replicated to other additional cards.

In the native model, Game of Life runs entirely on an Intel Xeon Phi coprocessor. No job is dispatched onto the host Xeon CPU. Each MIC core directly hosts up to 4 single-threaded MPI

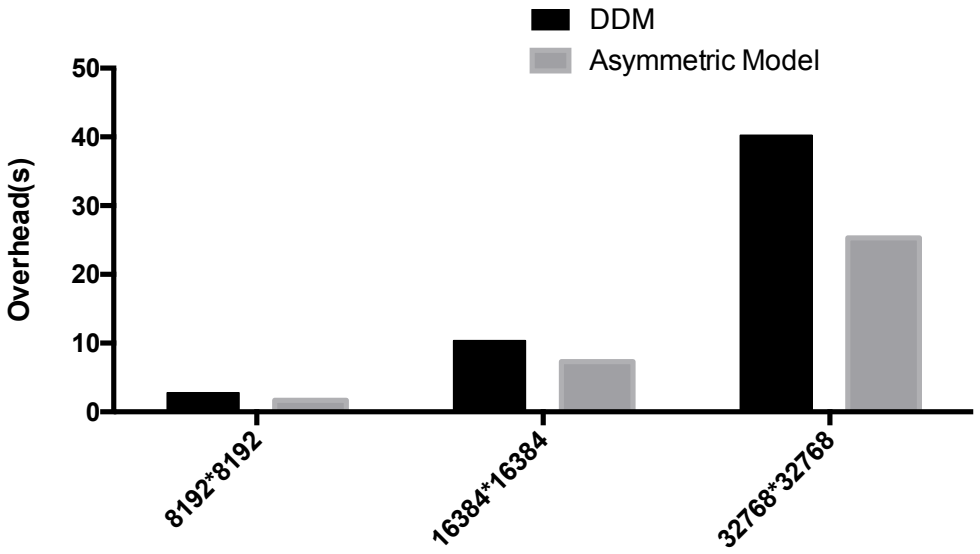


Figure 5.7: The overhead to calculate α in the DDM and Asymmetric model.

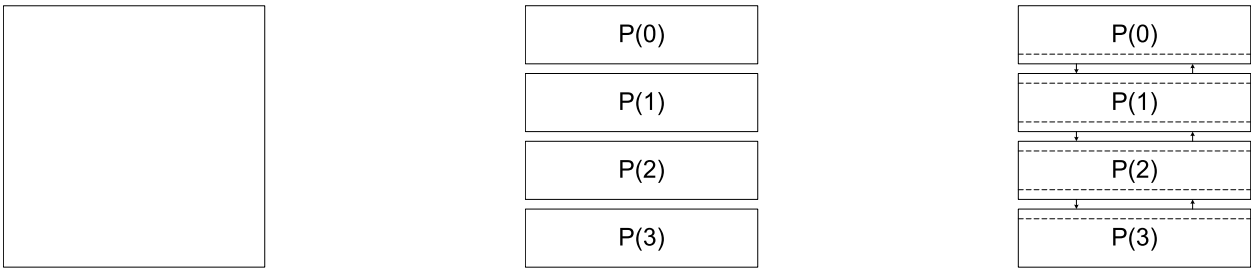


Figure 5.8: (a) The original problem size, i.e., a square matrix of cells. (b) Distribute the data among MPI processes in a row-wise order (4 processes as the example). (c) Communication among MPI processes.

processes. For each iteration, the statuses of all cells are updated simultaneously. In order to parallelize the updating process, the cells in the n×n matrix grid are partitioned into 240 stripes in the row-wise order as the example shown in Figure 5.8. Each stripe is handled by one MPI process. At the beginning of each iteration, the statuses of the cells along the boundaries of each stripe have to be exchanged with its neighbors through the MPI `send` and `receive` commands. In other words, each MPI process needs to send the statuses of the cells along the boundaries of each stripe to its neighbor MPI processes and receive the statuses of the cells of two adjacent rows. Each MIC runs 240 MPI ranks to process 240 stripes.

ALGORITHM 7: Game of Life

```
Function Transition(Cell, t)
    n = number of alive neighbors of cell at time t;
    if cell is alive at time t then
        if n > 3 then
            Cell dies of overcrowding at time t+1;
        end
        if n < 2 then
            Cell dies of under-population at time t+1;
        end
        if n = 2 or n = 3 then
            Cell survives at time t+1;
        end
    end
    if n = 3 and Cell is dead at time t then
        Cell becomes a live cell by reproduction at time t+1;
    end
end
```

For the symmetric model, the MPI processes run on both the MIC cores and the host Xeon CPU cores. One host CPU processor and one Xeon phi processor are used. On the computation node on Beacon supercomputer we use in this work, there are 2 host Xeon CPUs, running up to 32 MPI processes, and 4 Xeon Phi 7120P coprocessors. Therefore we run 8 single-threaded MPI processes to process 8 stripes on the host CPU. On the MIC coprocessor, we run 240 single-threaded MPI processes to process 240 stripes.

In the DDM, we use two iterations of Game of Life as a test to determine the computing capabilities of various processors/coprocessors. We find the computational capability of a MIC coprocessor is about $1.5\times$ times the combined performance of 8 threads on the host processor. We redesign the implementation of Game of Life based on this test. We schedule 8 single-threaded MPI processes on the host processor to process 8 stripes. A MIC coprocessor runs 240 single-threaded MPI processes to process 240 stripes.The programs decompose the problem into unevenly parts based on calculated α. An MPI process on CPU handles 20 times data compared with an MPI process on MIC. Through the uneven data distribution, all MPI processes can finish the data processing at roughly the same time.

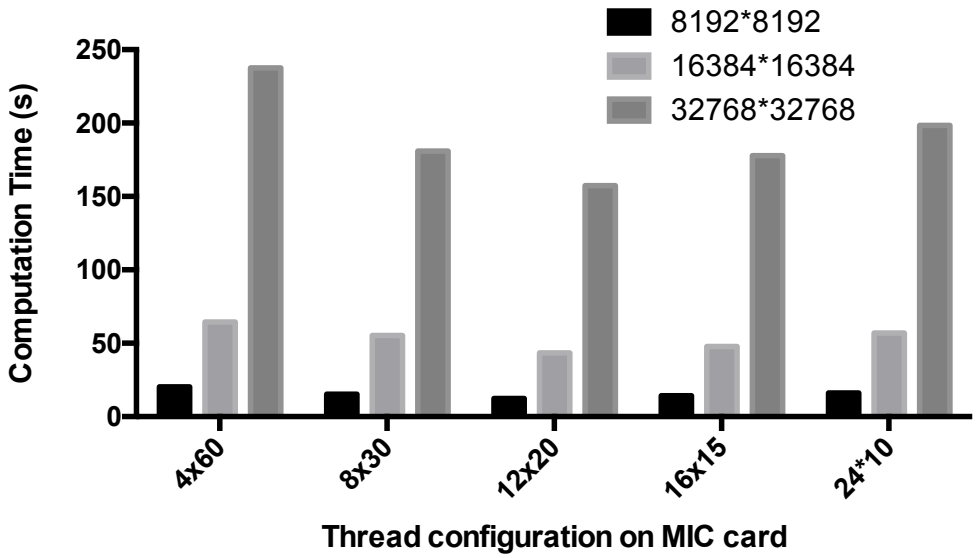
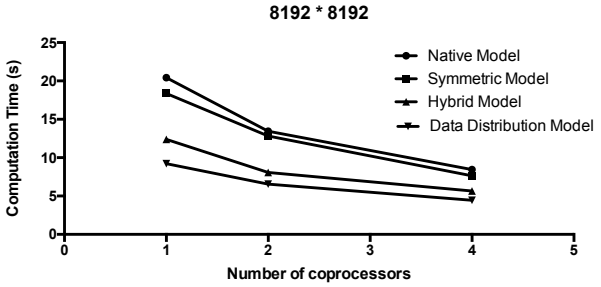
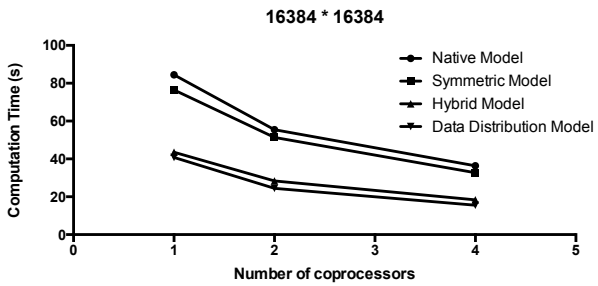


Figure 5.9: Performance comparison among different MPI/OpenMP configurations on a single MIC card. M×N: M is the number of MPI processes running on a MIC card, N is the number of OpenMP threads spawned in each MPI process.

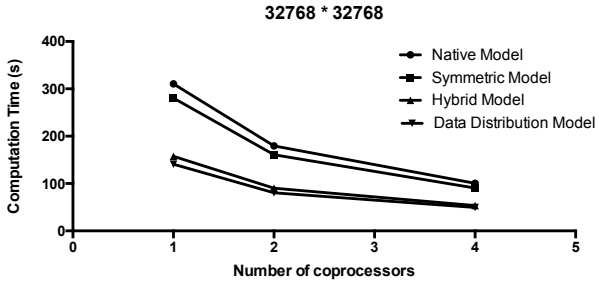
In the hybrid model, the MPI processes running on MIC do the communication and serial parts; and the OpenMP threads inside the MPI processes carry out the computation part. We schedule 8 MPI processes on host CPU and then determine the optimal combination of MPI/OpenMP on the MIC coprocessor. Because there are multiple combinations of the number of MPI processes on a MIC coprocessor and the number of threads spawned by each MPI, we list 5 options in Figure 5.9. It can be found that we can achieve the best performance when the MIC card issues 12 MPI processes and each MPI spawns 20 threads. This result agrees with the performance model we obtain in the DDM, i.e., the performance of a single thread on the host CPU is equivalent to the combined performance of 20 threads on the MIC coprocessor. Therefore, this model issues 8 single-threaded MPI processes to a CPU and 12 MPI processes to a MIC coprocessor. A single MPI process running on MIC spawns 20 threads using OpenMP.



(a) 8,192×8,192.



(b) 16,384×16,384.



(c) 32,768×32,768.

Figure 5.10: Performance of Game of Life on three different configurations.

5.6 Result and discussion

We conduct the experiments to use Xeon Phi 7120P coprocessors to demonstrate the performance of the Game of Life with different programming models. Three different grid sizes are tested, i.e. 8192×8192, 16,384×16,384, and 32,768×32,768. From the results in Table 5.1, it can be found that the hybrid model and DDM consistently outperform the native model and the symmetric model for this intense communication case. The native model does not take advantage of the compute capacity of the host Xeon CPU when the MPI program only run on Xeon Phi coprocessors. By observing the performance results of symmetric model, although the symmetric model is still able to reduce the computation time, the improvement can be ignored when the small grid size is tested. Xeon CPU core has a stronger computation capability than MIC core. But the program decomposes the problem into even parts so that the ranks executed on Xeon CPU have to wait at the synchronize point. Apparently, for this communication dense application, there is not much performance gain in the symmetric model. Although the symmetric model takes

Table 5.1: Performance of Game of Life using a single MIC coprocessor(unit: second).

	Native	Symmetric	Hybrid	DDM
8,192×8,192	20.43	18.37	12.38	9.21
16,384×16,384	84.45	76.46	43.53	40.79
32,768×32,768	310.45	280.35	157.42	140.76

Table 5.2: Performance of Game of Life (unit: second).

Number of	8,192×8,192				16,384×16,384				32,768×32,768			
MICs	N	S	H	D	N	S	H	D	N	S	H	D
1	20.43	18.37	12.38	9.21	84.45	76.46	43.53	40.79	310.45	280.35	157.42	140.76
2	13.45	12.82	8.07	6.56	55.42	51.34	28.35	24.45	179.34	160.90	90.02	80.45
4	8.44	7.63	5.66	4.45	36.34	32.67	18.36	15.56	100.34	90.45	53.47	49.45

N: Native model; S: Symmetric model; H: Hybrid model; D: DDM.

advantage of the compute capacity of the host Xeon CPUs, it demonstrates a load imbalance problem when the MPI programs are written for homogeneous systems.

Both the hybrid model and DDM also take advantage of the compute capacity of the host Xeon CPU. From the results in Table 5.1, they have a similar performance and they get about 1.7 times speedup than the native model. Table 5.2 lists the execution time of Game of Life on Beacon Supercomputer with different programming models. It can be found that the strong scalability is demonstrated for MPI and OpenMP implementations on four programming models. For this benchmark, the hybrid model and DDM can consistently outperform the native model by 1.7× based on the results in Table 5.2 and Figure 5.10.

5.7 Accelerating Urban Sprawl Simulation

5.7.1 Urban Sprawl Simulation

Among varieties of approaches to simulating urban growth, Cellular Automata (CA), an important spatiotemporal simulation approach and an effective tool for spatial optimization strategies, has been extensively applied to understand the dynamics of land use and land cover

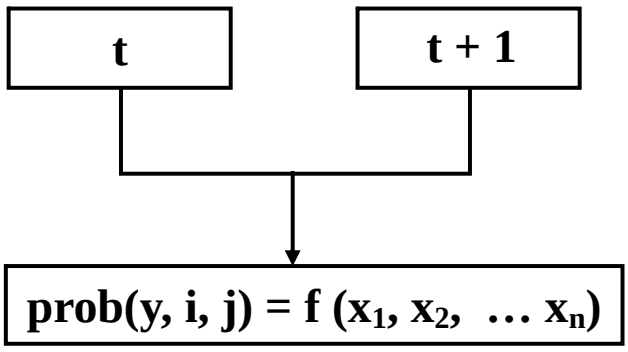


Figure 5.11: Calibration of the global probability surface from sequential land use data.

change. This study aims to utilize the MIC/CPU heterogeneous architecture to enable complex CA simulations for urban growth simulation over massive datasets to provide quick and scalable support for land allocation.

According to the original model designer [61], the stochastic CA model was implemented by a procedure that calibrates the initial global probability surface from sequential land use data and then modifies the global probability with the local probability, which is updated in each iteration of simulation. The purpose of calibration is to extract the coefficients or parameter values of the rules from the observation of land use pattern at time t and $t+1$, as shown in Figure 5.11. Mathematically, this is generalized as the estimation of the probability of particular state transition y occurring at a particular location (i, j) through a function of development factors $(x_1, x_2, \ldots, x_n)$.

Besides, two factors, the probability of site selection and the joint probability, constrain the quantity of simulated conversions to projected land demand. In the case of land use changes, for each year of simulation, a logistic model can be developed to calculate the probability, computed as (5.1), In the other word, it can make a decision whether a pixel is developed into urban land use type or remaining in the current state .

$$p_g(S_{ij} = urban) = \frac{exp(Z)}{1+exp(Z)} = \frac{1}{1+exp(-Z)}, \tag{5.1}$$

where p_g is the observed global probability y for a cell to convert to urban, ranging within [0, 1],

S_{ij} is the state of the cell (i, j). The joint probability can be calculated as the product of global probability, cell constraint, and neighborhood potentiality. Cell constraint refers to factors that exclude land development on the cells such as a body of water, a mountainous area and planning restriction zones. It is possible to use an evaluation score of land suitability instead of a binary one (suitable/unsuitable). The joint probability is stated in (5.2).

$$p_c^t = p_g con(S_{ij}^t = suitable)\Omega_{ij}^t, \tag{5.2}$$

where $con()$ converts the state of suitable land into a binary variable. Again, please note that the joint probability p_c is denoted with time t, indicating it changes along with iterations.

In this model, the neighborhood function is calculated in a conventional ad hoc way. The neighborhood potentiality of cell transition, as defined in (5.3), calculates the urban density of the cells 3×3 neighborhood at time t.

$$\Omega_{ij}^t = \frac{\sum_{3\times 3} con(s_{ij} = urban)}{3 \times 3 - 1}. \tag{5.3}$$

Based on the joint probability, a Monte Carlo process is applied in the calculation of the scaling of probability defined in (5.4).

$$p_s^t(ij) = \frac{q p_t^t(ij)}{\sum_{ij} p_t^t(ij)}, \tag{5.4}$$

where q is the number of cells to be converted according to projected land conversion at each iteration. The value of the right-hand should be limited at 1.0 to ensure that the probability is within the range of 0 to 1. This transformation in fact constitutes an additional constraint to the joint probability. As a result, the scaled probability is composed of three probabilities: (1) the probability of development measured on global factors, (2) the probability of development measured on local factors, and (3) the probability of cell selection according to the projected land

Table 5.3: Performance of Urban Sprawl Simulation (unit: second).

Platform: Single MIC and Single GPU (problem size: 10,000×10,000×7)															
No. of	1993–2002			1993–2012			1993–2022			1993–2032			1993–2042		
Proc.	Read	Comp.	**Total**	Read	Comp.	**Total**	Read	Comp.	**Total**	Read	Comp.	**Total**	Read	Comp.	**Total**
Single-MIC DDM Implementation															
7120p	6.32	9.4	**15.72**	6.31	18.85	**25.16**	6.11	28.6	**34.71**	6.23	40.45	**46.68**	6.26	49.4	**55.66**
Single-GPU Implementation															
K20	4.40	27.65	**32.05**	4.40	54.57	**58.97**	4.41	78.63	**83.04**	4.43	103.65	**108.08**	4.42	134.76	**139.18**

demand.

The grid $p_s^t(ij)$ is the urbanization status of a cell at time t, and $rand(ij)$ generates a random number with uniform distribution within the range of [0, 1]. Comparison between the grid $p_s^t(ij)$ and a random grid will decide whether the cell is to be converted at time $t+1$ as in (5.5).

$$S^{t+1}(ij) = \begin{cases} urban, & p_s^t(ij) > rand(ij) \\ rural, & p_s^t(ij) \leqslant rand(ij) \end{cases}, \tag{5.5}$$

When large scale datasets are applied, it is time consuming to complete such a complex model, or it could be an impossible task if the computer does not have sufficient memory to hold the input data, intermediate processing outcome, and the final output results.

5.7.2 Implementation and Results

The input image to this application is southern California taken in 1993. We use a part of image, a dimension of 10,000×10,000 for 7 bands, as an input. In order to project the urban growth in n years, n iterations of simulation need to be carried out. When multiple processors are used in the parallel implementation, the data partition is similar to the case of Game of Life, i.e., each image is divided into multiple stripes along the row-major order. It is also a densely communicating parallel application.

This urban sprawl simulation is implemented on a single Xeon Phi 7120P coprocessor with a host CPU (Intel Xeon E5-2603 v3) using the DDM parallel model. 240 single-thread MPI

processes are scheduled on the MIC coprocessor in addition to 6 single-thread MPI processes on the host CPU. Five different simulations are carried out from 10-year projection to 50-year projection. For this application, we also conduct a performance comparison between a single Nvidia K20 GPU [35] and a single MIC coprocessor (with the host CPU).

The results in Table 5.3 show that the implementation of DDM model using one Xeon Phi 7120P (with the host CPU) can outperform the implementation on a single Nvidia K20 GPU by 2 times. GPU solution does not get an ideal performance, because this simulation contains four kernels of CA transition. CA may lead to a problem of warp divergence on GPU because the value of each cell need to be decided by their neighbors, as shown in Algorithm 7. Besides, after completing each CA kernel, operations of *joint* and *sum* are implemented to decide the rule of next kernel. As a result, data exchange between GPU and CPU has to be done for a better performance.

5.8 Conclusion

In this work, we conduct a detailed study regarding the load imbalance problem on heterogeneous computing environment including Intel Xeon Phi coprocessors. Among the four parallel programming models, the native model only schedules the MPI processes to the MIC coprocessors, therefore wasting the computing capability of the host processors. The remaining three programming models schedule the MPI processes to both the MIC coprocessors and the host processors, which may have different computing capabilities. The symmetric model may have a load imbalance problem because the MPI processes running on different types of processors/coprocessors may spend different amount of time to process the evenly distributed work load. The hybrid model and the DDM model try to address the load imbalance problem using different ways. In the hybrid model, the work load is still evenly distributed. However, the MPI processes running on MIC coprocessors will spawn multiple OpenMP threads to match the performance of MPI processes running on the host CPUs. However, the combination of

MPI/OpenMP implementation has to be carefully adjusted to achieve the balance between the host processors and the coprocessors. In the DDM model, single-thread MPI processes are deployed on both host processors and coprocessors. However, the work load distribution among the MPI processes is proportional to the computing capabilities of the processors/coprocessors. The DDM model automatically carries out a performance profiling step to determine the computing capabilities of various processors and coprocessors in the heterogeneous environments. Based on the computing capabilities of the hosting processing cores, a proportional amount of data will be distributed to MPI processes so that they finish the data computation in roughly the same amount of time. Among the four parallel programming models, the hybrid model and DDM outperform the native model and symmetric model by about 2 times for the Game of Life benchmark.

Chapter 6

Automatic Performance Improvement on Heterogeneous Platforms: A Machine Learning Based Approach

In the previous chapters, we demonstrate different techniques regarding to the problem of load imbalance in different scenarios. Chapter 4 focuses on the distribution of task in XeonPhi and Chapter 5 demonstrates a smart data distribution model balancing the distribution of workload between the processors and coprocessors. Given these results, the imbalanced data and task partition can seriously hurt the performance. However, the number of possible options regarding data and task distribution between host processors and coprocessors is huge. It would take a large amount of time to determine the right data partition and task parallelism on heterogeneous platforms given a new application.

In this chapter, we presents a novel runtime approach to determining the optimal data and task partition automatically on heterogeneous platforms, targeting the Intel Xeon Phi accelerated heterogeneous systems. We employ machine learning techniques to train a predictive model off-line and then use the trained model to predict the data partition and task granularity for any unseen programs at runtime. We apply our approach to 21 representative parallel applications and evaluate it on a Xeon-Xeon Phi mixed heterogeneous platform.

6.1 Introduction

A supercomputer may provide a heterogeneous environment including the host processors and accelerators. As a consequence, programmers can execute applications on both host processors and accelerators. To achieve the combined performance potential, it requires software to effectively partition the the workload of parallel applications to maximize the computation overlap between host processors and accelerators. Offload mode, a classic programming mode in

the heterogeneous system, is to use host processors to manage the execution context while the computation is offloaded to the accelerators. Effectively leveraging such platforms not only achieves high performance and good scalability, but also increases the energy efficiency.

While the heterogeneous platform provides the potential for high performance and energy efficiency, the classic offload model leaves the host processors unutilized. This means this approach does not take advantage of the computing capacity of the host processors and is likely to give away too much performance potential of the whole system. Asynchronous data transfer and computation have been proposed as a solution to decrease the host-device[1] communication cost and to increase the utilization of host processors [19, 34]. In asynchronous mode, the host processor sends workload to the accelerator. Then the host processor continues the execution of other workload until it is requested to wait for a kernel running on the accelerator to finish. In this case, both host processors and accelerators can work in parallel to undertake a computation task.

However, it is hard to determine the right data partition and task parallelism on heterogeneous platforms given a new application. The number of possible options regarding data partition between host processors and accelerators is huge. The imbalanced data and task partition can seriously hurt the performance. In this work, we provide a novel approach to determine the optimal distribution of data and threads for any unseen programs.

The remainder of this chapter is organized as follows. We discuss some related work and problem scope in Section 6.2. The predictive model is discussed in Section 6.3. In Section 6.4, we demonstrate the experiment platform and the benefit of our new method and compare the performance with the non-asynchronous method. Finally, we give the concluding remarks in Section 6.5.

[1]host: host processor; device: accelerator/coprocessor.

6.2 Background and Overview

Supporting data parallelism in a heterogeneous system requires two parts. First, executable versions of a kernel must be compiled for all types of processors. Second, a scheduling mechanism is required to determine how to execute parallel sections (e.g., parallel iterations) of applications. There are different approaches to scheduling parallel sections across multiple processors of a heterogeneous system in order to minimize execution time.

This section presents a novel runtime approach to determining the optimal data and task partition on heterogeneous platforms, targeting the Intel Xeon Phi architecture. We do so by employing machine learning techniques to train a predictive model that can automatically decide at runtime the optimal configuration for any application. Our predictor is first trained off-line. Based on the code and runtime features of a program, the model predicts the best configuration for a new and unseen program. We apply our approach to 21 representative benchmarks, and evaluate the predictive model on a heterogeneous many-core platform that contains general purposed multi-core CPUs and 61-core Intel Xeon Phi coprocessors. Our approach achieves, on average, a 1.6× and 1.9× speedup with 1MIC+1CPU and 1MIC+2CPUs, respectively, over the optimized, non-asynchronous code.

6.2.1 Related work

Online profiling

In online approaches, the decision of where to execute parallel iterations is made at runtime. The programmer does not have to train the system on representative inputs for each target platform. Work-stealing and profiling-based scheduling are popular methods to solve load imbalanced problems. Work-stealing [11] is to address load imbalance in multi-core execution. Profiling-based scheduling is an online profiling based approach [25]. The application kernel is dynamically profiled in order to determine the distribution of workload between the host

processors and the accelerators.

Offline profiling

In offline approaches, the application is first executed using a training data set and profiled. The profiling data is used to select the scheduling policy for subsequent runs of the same application against real data. Qilin [40] performs an offline analysis to measure the kernel's execution rate on each device (CPU and GPU). These rates are used to decide the distribution of work for each device. Qilin uses a linear performance model to chose the scheduling policy based on the size of the input data set. In fact, many factors determine the performance of each device. If only depending on the kernel's execution rate of training data, the scheduling policy is likely to be suboptimal when the training data differs significantly than the actual data used in subsequent runs. In this paper, we present a new method based on machine learning algorithms to decide the distribution of work among processors (including both host processors and accelerators).

6.2.2 Problem Scope

Our work aims to improve the performance of parallel applications on heterogeneous platforms. The performance improvement would be achieved by determining at runtime how to distribute workloads among host processors and accelerators. In this work, we target the Intel Xeon Phi architecture. But our methodology is generally applicable and can be extended to other architectures including GPGPUs and FPGAs.

Figure 6.1 demonstrates a simplified code example written with Intel's asynchronous APIs running on a platform containing one host processor and one Xeon Phi coprocesosr. At line 3 we initialize the asynchronous execution by using an $offload$ pragma with a *signal* clause and we set the number of workload working on Xeon Phi. This initialization process transfers data between the host processor and Xeon Phi coprocessor and determines how much data can run on the coprocessor. In the $for\ loop$(lines 9-14) we decide how many concurrent threads are

```
1 // setting the workload size and enable asynchronous data \
2 transfer and computation by directives with signal and wait
3 #pragma offload target(mic) in(input[0:workload_mic]) \
4 out(out[0:workload_mic]) singal(signal_value)
5 {
6     // setting OMP nestd parallelism
7     omp_set_nested(true);
8     // setting the number of concurrent threads in level 1
9     #pragma omp parallel for num_threads(threadsNum_1) ...
10    for(...){
11        // setting the number of concurrent threads in level 2
12        #pragma omp parallel for num_threads(threads_Num_2) ...
13        for(...){}
14    }
15 }
16
17 // host CPU executes a part of computation work
18 #pragma omp parallel for ...
19 for(...){
20    for(...){}
21 }
22 // blocks execution until an asynchronous data transfer\
23 or computation is complete
24 #pragma offload_wait target(mic) wait(signal_value);
```

Figure 6.1: Asynchronous code example.

employed in level 1 and level 2 (need to explain what are level 1 and level 2). The nested parallelism is set by $omp\_set\_nested()$ API.

When Xeon Phi is working on computation task, the host CPU continues execution line 17-23 until it is requested to wait for a kernel to complete as shown in line 24. In this way, computation between host processor and Xeon Phi can be overlapped during execution. Because host CPU usually has a limited number of concurrent threads, we only employ parallelism in level 1. A performance improvement is going to be achieved when the kernels running on the host processor and Xeon Phi are completed at the same time. Our predictive model determines the ratio of workload running on Xeon Phi and the number of concurrent tasks in level 1 and level 2 before invoking asynchronous initialization.

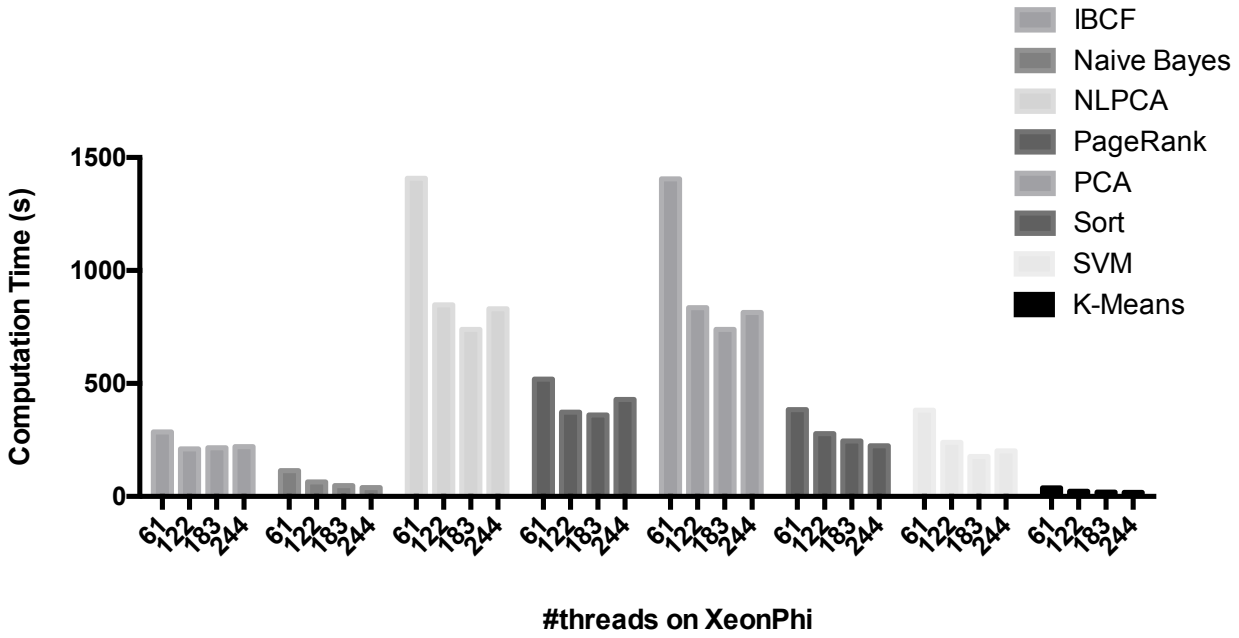
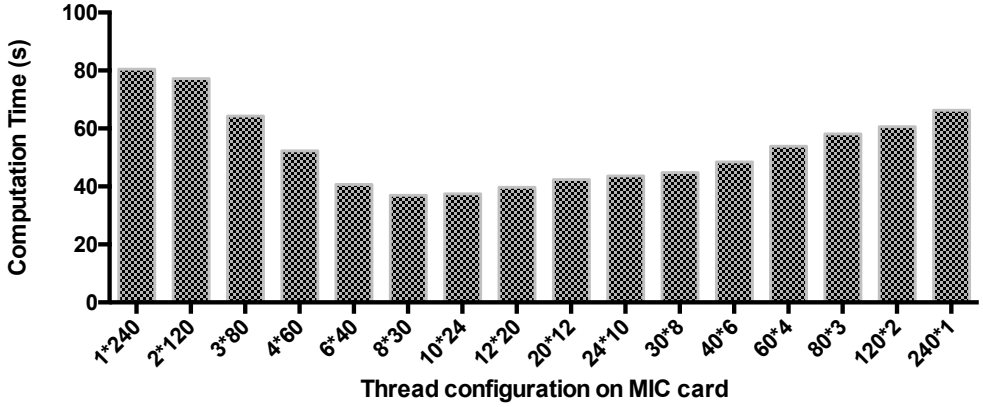


Figure 6.2: Running times for different numbers of threads [62].

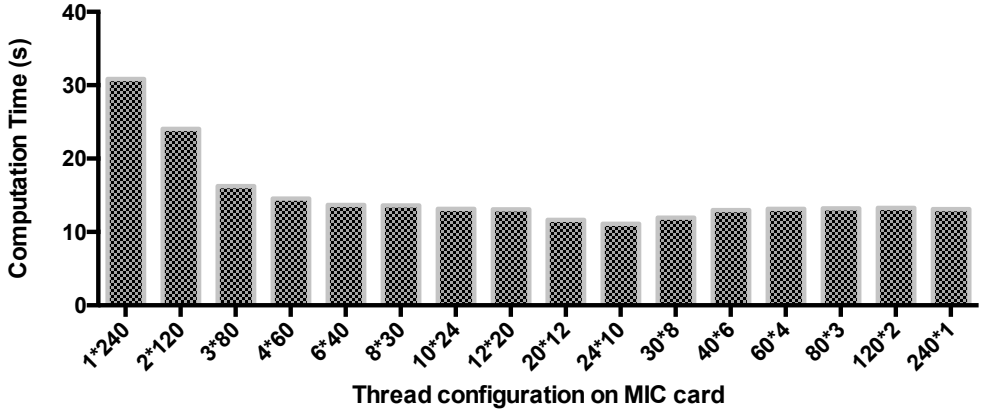
6.2.3 Motivating Examples

Impact of number of threads

The work in [62] investigates the impact of affinity and scheduling policy on performance. The number of threads are set to 61, 122, 183 and 244, so that the capability of manycore Xeon Phi can be fully utilized. Figure 6.2 demonstrates the impact of number of threads on performance. It can be found that some benchmarks, such as Naive Bayes, Sort and K-means, achieve the best performance when thread counts are set to 244, while NLPCA, PageRank, PCA and SVM have the best performance when using 183 threads and IBCF uses 122 threads to get the best performance. Therefore, we cannot simply determine the best performance by using all hardware resources.



(a) Stencil.

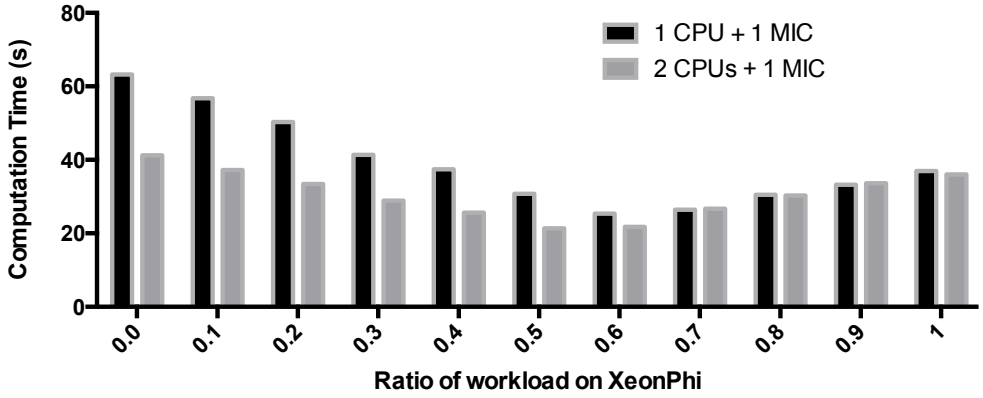


(b) FFT2D.

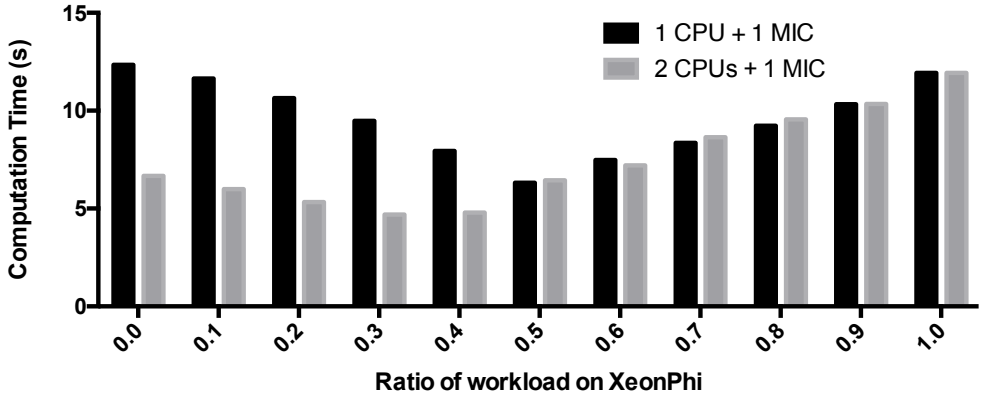
Figure 6.3: Computation time of Stencil and FFT2D under different thread configurations. $M \times N$: M is the number of threads offloaded to a MIC card for realizing the level-1 parallelism, each thread further spawns N threads to achieve the level-2 parallelism

Impact of configuration of threads

Figure 6.3 shows the computation times based on MIC offload mode with different configurations of threads for two applications (Stencil and FFT2D) on a 61-core Intel Xeon Phi system. In this experiment, 240 threads are scheduled to one Intel Xeon Phi device under various configurations. It can be observed from these benchmarks that it is critical to adjust the configuration between level 1 and level 2 parallelism to achieve the best performance. As can be seen from the diagrams, the search space of thread configurations is huge and good configurations are sparse. The performance varies significantly over thread configurations. The optimal thread



(a) Stencil.



(b) FFT2D.

Figure 6.4: Performance based on different ratios of workload distribution. 1 CPU + 1 MIC: 6 threads are created on one host CPU and 240 threads are created on one MIC. 2 CPUs + 1 MIC : 12 threads are created on two host CPUs and 240 threads are created on one MIC.

configurations for *Stencil* are 6×40, 8×30, 10×24 and 12×20. In contrast to *Stencil*, $FFT2D$ benefits from thread configurations of 20×12, 24×10 and 30×8. As a result, the optimal thread configurations vary for different applications.

Impact of workload distribution

Figure 6.4 shows the computation times with different ratios of workload distribution and different configurations of platforms. As can be seen from the figure, the best distribution of workload can vary for different benchmarks. The optimal workload distribution of *Stencil* is to transfer 60%-70% workload into Xeon Phi device for 1 CPU + 1 MIC configuration and

50%-60% workload to into Xeon Phi device for 2 CPUs + 1 MIC model. In the case of $FFT2D$, the amounts of workload sent to Xeon Phi device are 50%-60% and 30%-40% for 1 CPU + 1 MIC configuration and 2 CPUs + 1 MIC configuration, respectively.

These three examples demonstrate that choosing the right configuration has a great impact on the performance improvement and the best configuration must be determined for each unique program. Attempting to find the optimal configuration through means of an exhaustive search would be ineffective. The overhead involved would be far bigger than the potential benefits. Therefore we use machine learning to automatically construct a predictive model that can directly predict the best configuration. The predictive model introduces the minimal runtime overhead and has little development overhead when targeting new architectures.

6.2.4 Overview

To determine the optimal workload distribution and thread configurations, we use a set of features to capture the characteristics of the benchmarks. We use the MAQAO [18] performance tool for performing static analysis of binary code and use perf's tools [17] to collect runtime information at execution time. Given the outputs of MAQAO and perf's tool, we develop a Python program to capture features for training process and predicting process.

Because the profiling process also contributes to the final program output and the method has a low overhead, there is no big computation cycle to be wasted. At prediction process, a predictive model, which is trained offline, is used to predict the optimal configuration of workload distribution and threads by feature values.

6.3 Predictive Modeling

In this study, Support Vector Machine (SVM) with a Gaussian function kernel [56] is used to determine the best workload distribution and thread configuration. The model is implemented

Table 6.1: Benchmarks list.

Benchmarks to generate training (including cross validation) data			
Package	Benchmark Name		
NVIDIA SDK	convolutionSeparable convolutionFFT2d MonteCarlo	fwt vectorAdd dotProduct	matVecMul transpose
AMD SDK	binomial	dct	BlackScholes
Benchmarks to generate testing data			
Package	Benchmark Name		
Parboil	tpacf lbm mri-gridding	sgemm histo stencil	cutcp mri-q spmv

using scikit-learn library [48]. Besides, we have evaluated a number of machine learning models, including random forest, K-nearest neighbor (KNN), decision trees, and the artificial neural network (ANN). We chose SVM to train predictive model because it generates the best performance and can model both linear and non-linear problems. The model takes in feature values and predict labels for the optimal configuration.

Our approach follows four steps: (1) generate training data, (2) features analysis, (3) train a predictive model, and (4) predict the optimal configuration of an unseen application.

6.3.1 Generating training data

As currently there exist very few programs written with Intel's asynchronous API, we manually translated 20 applications from the commonly used benchmark suites [6, 1, 59]. Table 6.1 gives the full list of these benchmarks. In this work, we use over 15 different input dataset for each training benchmark and totally the training dataset contains more than 1,500 samples.

These 20 benchmarks are divided into two groups. 11 benchmarks in NVIDIA SDK and AMD SDK are used for generating the training data. The training data is further divided into multiple folds to carry out cross validation. We did not include the prefixSum benchmark in the AMD SDK because the cost of transferring data from host processor to MIC coprocessor is far

Table 6.2: Nested Thread Configuration.

#threads	Config.	Config.	Config.	Config.	Config.	Config.
60	1×60 2×30	3×20 4×15	5×12 6×10	10×6 12×5	15×4 20×3	30×2 60×1
80	1×80 2×40	4×20 5×16	8×10 10×8	16×5 20×4	40×2	80×1
100	1×100 2×50	4×25 5×20	10×10 20×5	25×4	50×2	100×1
120	1×120 2×60 3×40	4×30 5×24 6×20	8×15 10×12 12×10	15×8 20×6 24×5	30×4 40×3	60×2 120×1
140	1×140 2×70	4×35 5×28	7×20 10×14	14×10 20×7	28×5 35×4	70×2 140×1
160	1×160 2×80	4×40 5×32	8×20 10×16	16×10 20×8	32×5 40×4	80×2 160×1
180	1×180 2×90 3×60	4×45 5×36 6×30	90×20 10×18 12×15	15×12 18×10 20×9	30×6 36×5 45×4	60×3 90×2 180×1
200	1×200 2×100	4×50 5×40	8×25 10×20	20×10 25×8	40×5 50×4	100×2 200×1
220	1×220 2×110	4×55 5×44	10×22 11×20	20×11 22×10	44×5 55×4	110×2 220×1
240	1×240 2×120 3×80	4×60 6×40 8×30	10×24 12×20 20×12	24×10 30×8 40×6	60×4 80×3	120×2 240×1

larger than calculation time. The remaining 9 benchmarks in the Parboil package are used to generate the testing data for the performance predictive model.

We run each program under a specific configuration multiple times and report the geometric mean of the runtimes. A set of programs are executed on the CPU and the Xeon Phi many times with different configurations to determine the best workload and threads distribution. We first run each benchmark on Xeon Phi to determine the best thread configuration and achieve the best performance and then decide the workload distribution between the processor and coprocessor. Besides, we execute each training program with different amount of dataset across all of our considered workload and threads configurations, and record the performance of each. Specifically, we profile the program using the α, which represents the percentage of workload distribution to coprocessors, ranging from 0 to 1 with the step of 0.1 and the number of threads ranging from 10 to 240 with the step of 10. In the nested applications, we additionally profile the thread configurations, as shown in Table 6.2.

Next, we record the best performing configuration for each program and dataset, keeping a label of each. Finally, we extract the values of our selected set of features from each program and dataset.

6.3.2 Features

Feature Selection

In order to apply machine learning techniques to any decision-making problems, we need to select a set of relevant features that comprehensively represent a set of problem instances. These features would have an significant effect on the training model process. Code features are extracted from the program source code using static analysis tools. Dynamic features are collected using hardware performance counters during the initial profiling run of the target application. In order to collect the static and dynamic features, we run our benchmarks both on host CPUs and MIC with MAQAO and perf-tools. Perf-tools are commonly available on modern processors and provide low-overhead access to a wealth of detailed performance information. MAQAO [18] is a performance tuning tool that performs both static and dynamic analysis of the binary code and it is able to capture values in the code. Besides, MAQAO can be used to trace memory accesses, count loop iterations and capture function parameters. By using above tools, we are able to achieve more choices of code and runtime features and this approach also can be migrated to other architectures. We considered 28 candidate features in this work. Some features were chosen based on previous work [21, 29], and others were chosen from our intuition that these factors can affect the performance such as the overhead of data transferring and the times of data transfer API calls.

Feature Importance

We used the forward feature selection and random forest [16] to select the most important features. Forward feature selection begins training a separate model for each single feature. In the

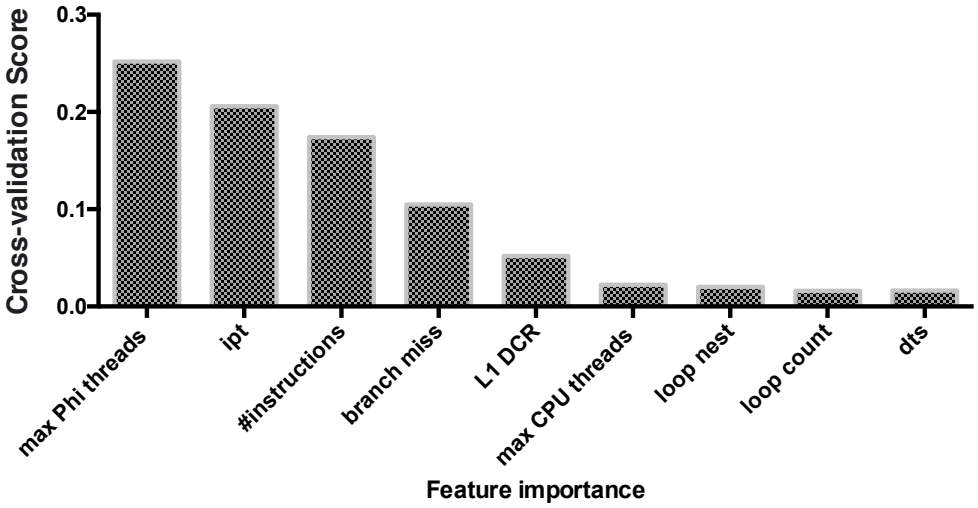


Figure 6.5: Feature importance according to Forward Feature Selection and Random Forest

Table 6.3: Seleted Features.

Feature	Description
max Phi threads	the maximum number of xeon Phi threads
ipt	number of items to be processed for each thread
#instructions	the total number of instructions of the kernel
branch miss	branch miss rate
L1 DCR	L1 Data cache miss rate
max CPU threads	the maximum number of CPU threads
loop nest	number of levels the loop can be parallelized
loop count	number of the parallel loop iterations
dts	total host-device transfer size

other word, we start by measuring the cross-validation score of the one feature subsets so that we can find the best individual feature and add it into the set of selected features. And then forward selection finds the best subset containing two features by measuring the cross-validation score of each remaining feature and the already selected features. This step will repeat until adding another feature will not further improve the score. Random forest is a classification method, but it also provides feature importance [12]. Its basic idea is as follows. A forest contains many decision trees, each of which is constructed by instances with randomly sampled features. The prediction is by a majority vote of decision trees. In order to obtain feature importance, the method would split the training sets into two parts. One is used to train and the other is used to predict so that we can obtain an accuracy value. For the i_{th} feature, we randomly permute its

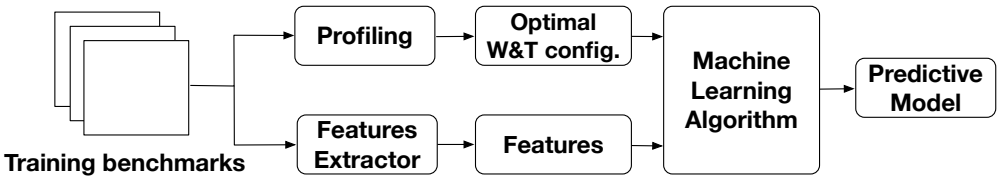


Figure 6.6: Training process (W&T: workload distribution and thread configuration).

values in the second set and obtain another accuracy. The difference between the two numbers can indicate the importance of the i_{th} feature. According to the results of two feature selection methods, we find out 9 important features, as shown in Table 6.3. Figure 6.5 also demonstrates average cross validation scores. In these 9 features, features that capture the parallelism (e.g., max Phi threads), workload on each thread (e.g., items per thread), and computation (e.g., instructions) are found to be the most important. Other features such as L1 DCR and loop nest are useful, but are less important compared with others.

6.3.3 Training model

Our method for model training is shown in Figure 6.6. Based on the training patterns, we build a model with inputs for each feature and outputs that predict the optimal workload distribution and thread configuration for an unseen application. The features extracted from performance tuning tools and the corresponding configuration labels are passed to a machine learning algorithm. The output of our learning algorithm is a SVM model where the weights of the model are determined from the training data. The program generating machine learning model is developed based on Python and scikit-learn library. Besides, we use the parameter tuning tool provided by scikit-learn to determine the kernel parameters. Parameter search is performed on the training dataset using cross-validation.

The training is only performed once off-line. In this implementation, the overall training process takes less than a week on a single machine with Intel i7-5775R CPU and 16GB memory.

Figure 6.7: Predicting process (W&T: workload distribution and thread configuration).

6.3.4 Runtime Deployment

Our overall approach requires to build a model using machine learning and then predict workload distribution and thread configuration from a set of features that describe the essential characteristics of a program. Once we have built and trained our predicted model as described above, we can use it to predict the best W&T configuration for any new, unseen program as shown in Figure 6.7.

When a new application is fed into our program, the feature values of the program would be extracted by profiling the program. Once feature collection is completed, feature values would be fed to the predictive model, which will generate the outputs predicting the optimal workload and threads configuration for the target program.

6.4 Experimental results

6.4.1 Experiment setup

Experiment platform

Our evaluation platform is a server with two Intel E5-2603 v3 Xeon CPUs and an Intel Xeon Phi 7120P accelerator (61 cores). The host CPUs and the accelerator are connected through PCIe. The host environment runs CentOS 6.5. We use Intel's MPSS (v3.6) to communicate between the host and the coprocessor.

Evaluation Methodology

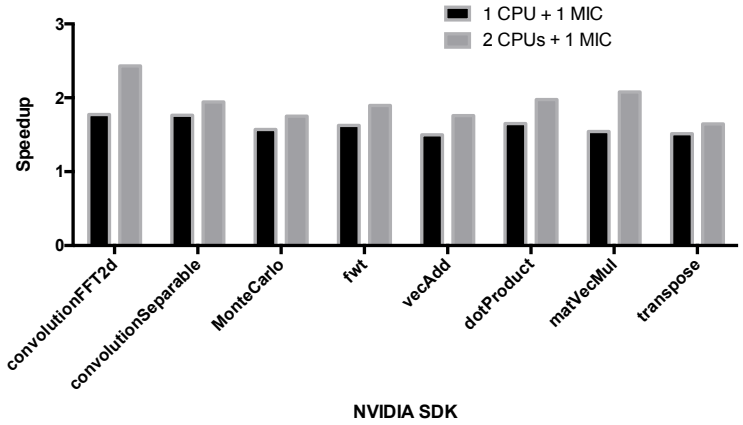
Cross validation [30] is used to evaluate our machine learning model. Cross validation is a method to evaluate predictive models by partitioning the original sample into a training set to train the model, and a validation set to evaluate it. In this work, leave-one-out cross validation are applied to AMD SDK, NVIDIA SDK suites for training the predictive model, which means one target program excluded from the training program set is kept for validation, and remaining programs are applied for training a predictive model. we then apply the learned model to predict the workload and threads configuration of the validated target program. We repeat this process over and over again until each benchmark from AMD and NVIDIA SDK is evaluated. Leave-one-out cross validation is a standard evaluation methodology, providing an estimate of the generalization ability of a machine-learning model in predicting unseen data.

6.4.2 Result

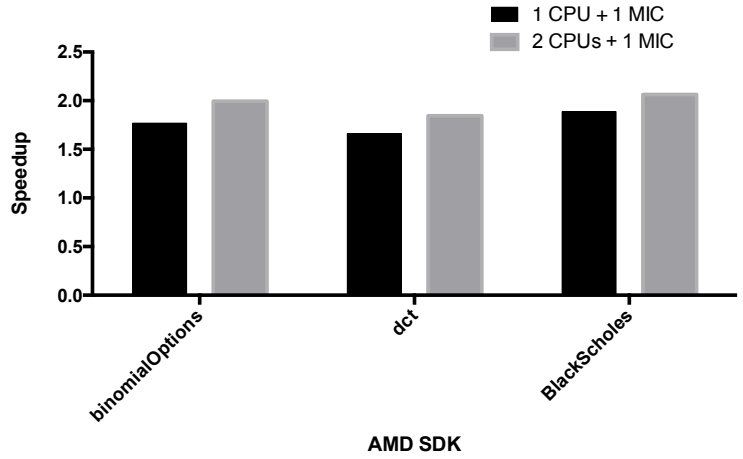
In this section, we demonstrate the overall performance of our approach with 1MIC+1CPU and 1MIC+2CPUs. We then compare our approach using the fixed number of threads with different configurations and show that a right choice of thread configuration also plays a significant role in performance improvement.

Overall Results

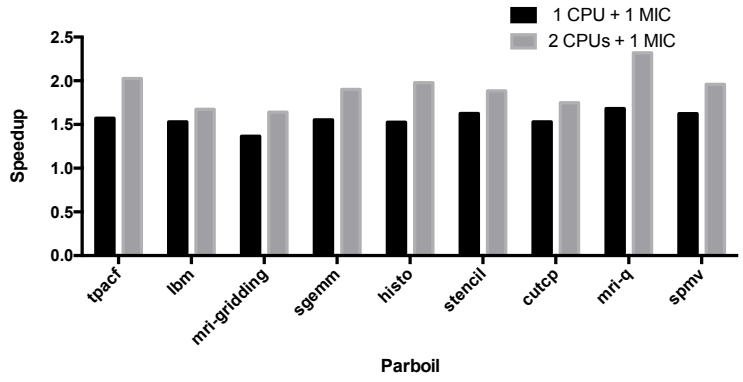
The result, shown in Figure 6.8, demonstrates the range of speedups per application across all evaluated inputs. Overall, our approach achieves an average speedup of 1.6× and 1.9× over the baseline method. The baseline method used to calculate the speedup is running the application on a single MIC without CPU processors. And the results of baseline method are the best result we achieved from optimal thread configuration. In this experiment, we exhaustively profiled each application with all possible workload and threads configurations. Our approach use



(a) Performance comparison of NVIDIA SDK between asynchronous code and non-asynchronous code.



(b) Performance comparison of AMD SDK between asynchronous code and non-asynchronous code.



(c) Performance comparison of Parboil between asynchronous code and non-asynchronous code.

Figure 6.8: Overall performance compared to a single Xeon Phi version with the optimized, non-asynchronous code. Our approach achieves an average speedup of 1.6× and 1.9× with using 1MIC+1CPU and 1MIC+2CPUs.

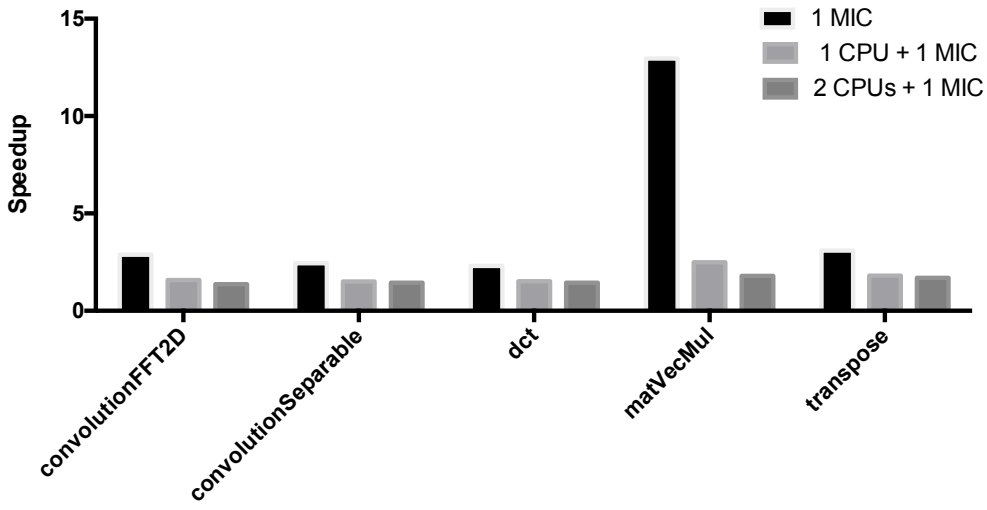


Figure 6.9: Performance difference between the worst and the best thread configurations.

1MIC+1CPU and 1MIC+2CPUs to run the entire applications. Although our model is not trained on the Parboil benchmark suite, it achieves good performance, achieving 1.5× and 1.8× speedup. This demonstrates the portability of our approach across benchmarks.

Analysis of High Speedup Cases

We found that there are several benchmarks obtain a speedup of over 2×. After having a detailed investigation, we notice that the performance of two host CPUs(12 threads) is better than that of Xeon Phi because the cost of transferring data between the host and Xeon Phi is also important to the overall performance.

Analysis of thread configurations

The Figure 6.9 demonstrates the performance difference between the worst and the best thread configurations. To quantify the benefit of kernel time reduction, we measure the kernel execution time with different threads configuration, as shown in Table 6.2, and calculate the speedup between the worst case and the best case. The best case is achieved by parallelizing a nested loop with different thread configurations. Based on the same number of threads of the best

case, we can get a thread configuration for the worse case. Compared to the worse thread configuration, the best configuration achieves an average speedup of 4.3× using a single MIC without CPU processors. Besides, the best configuration achieves an average speedup of 1.7× and 1.5× over the worse configuration using a single MIC with one CPU processor and two CPU processors. We found the final configurations of workload and threads can decrease the effect of the worse case of threads distribution.

The performance improvement for convolution, dct, matVecMul and transpose is shown in Figure 6.9. As can be seen from the diagram, choosing a good threads configuration can lead to more than 2× speedup on the kernel execution time because these benchmarks are implemented by parallelizing a nested loop. Efficient implementation of nested parallelism is difficult or complex, because it needs a thread management that can comfortably handle a very large number of threads. The parallel threads working on the inner loop will need to be created, synchronized, or destroyed for each outer loop iteration. In general, parallelizing inner loops while outer loops iterates many times would achieve a bad performance because of forking and joining overhead of threads. This threading overhead could be significant to the overall performance.

When using a better combination of threads, we essentially divide the whole outer loop iteration space into multiple smaller iteration space. This allows multiple groups of threads to be managed simultaneously, leading to a significant decrease in threading overhead and faster kernel execution time. On the other hand, we note that using too many threads on outer loop will lead to a performance decrease. This is because creating nested parallel regions adds overhead, which means overhead increases as the number of outer threads increases.

Compare to different Learning models

The average speedup achieved by different learning models are shown in Table 6.4. For each learning model, we use the same training dataset and features and follow the same training methodology to build a model. These program are implemented by scikit-learn [48]. In the

Table 6.4: Compare to the different learning models.

Learning model	Ave. Speedup (1MIC+1CPU)	Ave. Speedup (1MIC+2CPU)
RandomForest	1.55	1.8
DecisionTree	1.24	1.48
KNN	1.34	1.59
ANN	1.55	1.78
GussianSVM	1.6	1.9

process of setting parameters, we try various of parameters to find the best result. For example, we try different k values to build KNN models. In the implementation of ANN, we also try different number of hidden layers and neurons.

It can be found that all models achieve a performance over $1.2\times$ and $1.4\times$ speedup when the program uses $1MIC+1CPU$ and $1MIC+2CPUs$. The SVM model based on the Gaussian kernel achieves the best performance. This is because the Gaussian kernel function can model a non-linear relation between the features and the labels. Typically, the best possible predictive performance is better for a nonlinear kernel [27]. As a result, it predicts the best workload and threads configuration more accurate than other alternative models. Artificial neural network(ANN) is believed to train a better model if more data are added into training dataset.

Summary

The performance improvement of our approach comes from two factors. First, our approach allows effective usage of threads by predicting the right configuration. Second, compared to the single MIC execution, our approach achieve the maximum gain and resource utilization on heterogeneous computer architecture, the computation workload in an application should be distributed across accelerators and host CPUs. Besides, the ideal distribution of workload on host and accelerators achieves maximum performance improvement and it can decrease the effect of the worse case of threads configuration. As a result, the performance improvement mainly comes from the computation capability of host CPU and the optimal distribution of threads on Xeon Phi.

6.5 Conclusion

In this section, we conduct an approach based on machine learning to partition workload between host and accelerator on heterogeneous computing environment targeting Intel Xeon Phi coprocessors. We use existing machine learning methods to predict the optimal workload and threads configurations. The training process is to build our predictive model and predicting process is based on a set of features of program. Our approach is evaluated by a set of benchmarks on a heterogeneous platform containing two Xeon CPUs and one Xeon Phi. The results demonstrate that the approach achieves, on average, a 1.6× and 1.9× speedup with 1MIC+1CPU and 1MIC+2CPUs over the optimized, non-asynchronous code, respectively. When we use a single MIC without CPU processors, the best configuration of threads achieves an average speedup of 4.3× than the worse configuration. Besides, the best configuration achieves an average speedup of 1.7× and 1.5× over the worse configuration using a single MIC with one CPU processor and two CPU processors, respectively. We found the final configurations of workload and threads can decrease the effect of the worse case of threads distribution.

Chapter 7

Conclusion

7.1 Summary

This thesis shows several new approaches solve the problem of load imbalance in different levels. An important contribution of this work is to achieve the maximum gain and resource utilization on heterogeneous computer architecture automatically instead of an exhaustive manual search. Through this study, we have the following findings:

1. A static task distribution among working threads may introduce an imbalance of workload among them. In order to improve the overall performance, leveraging the dynamic task distribution is necessary so that a thread will request a new task from a task pool once it finishes the processing of the current task. Experiment results show that this technique can improve the performance by 25%. In addition, we try to leverage the processing power of the host CPU by implementing the algorithm with irregular kernels. The results show that the hybrid implementation is able to further improve the performance by 40%.

2. The hybrid model and the proposed DDM model try to address the load imbalance problem using different ways. Hybrid model has to be carefully adjusting the combination of MPI/OpenMP implementation by an exhaustive search manually to achieve the balance between the host processors and the coprocessors. The DDM model automatically carries out a performance profiling step to determine the computing capabilities of various processors and coprocessors in the heterogeneous environments. Based on the computing capabilities of the hosting processing cores, a proportional amount of data will be distributed to MPI processes so that they finish the data computation in roughly the same amount of time. Results show that the hybrid model and DDM have similar performance and outperform the native model and symmetric model by about $2\times$ for the benchmark.

3. A machine learning method is proposed to automatically construct a predictive model that can directly predict the best configuration. Instead of using an exhaustive search, an predictive model would be trained to find the optimal configurations of workload and threads. This approach is evaluated by a set of benchmarks on a heterogeneous platform containing two Xeon CPUs and one Xeon Phi. The results demonstrate that the approach achieves, on average, a 1.6× and 1.9× speedup with 1MIC+1CPU and 1MIC+2CPUs over the optimized, non-asynchronous code, respectively.

7.2 Future Work

In this thesis, we focus on asynchronous calculation and efficiently utilizing all available resources to achieve performance improvement, targeting the Xeon Phi coprocessor. A machine learning model builds a good model to predict the configuration of the unseen data. We use MAQAO and perf-tools to capture the static and dynamic code features. Through the study, a few ideas of future work are demonstrated as follows:

1. Instead of using MAQAO, a LLVM compiler can be developed to extract static code features at compile time so that the overhead of capturing static feature can be ignored. At runtime, a predictive model (that is trained offline) takes in the feature values and predicts the optimal configurations. The overhead of runtime feature collection and prediction is going to be small.

2. This thesis uses the traditional machine learning models to train a model to predict the unseen data because we only generate about 1500 samples. However, a neural network is gaining much popularity due to its accuracy when trained with huge amount of data. This work transfers 20 benchmarks to the targeted code. In the future work, a neural network would be evaluated when a huge amount of data is generated with different benchmarks.

References

[1] AMD SDK. https://developer.amd.com/tools-and-sdks/.

[2] http://www.esri.com/software/arcgis/.

[3] http://www.jics.tennessee.edu/aace/beacon.

[4] Intel many integrated core architecture. http://www.intel.com.

[5] Intel Many Integrated Core Architecture. http://www.intel.com.

[6] NVIDIA SDK. http://developer.nvidia.com.

[7] Cédric Augonnet, Samuel Thibault, and Raymond Namyst. Automatic calibration of performance models on heterogeneous multicore architectures. In Euro-Par Workshops, pages 56–65. Springer, 2009.

[8] Blaise Barney et al. Introduction to parallel computing. Lawrence Livermore National Laboratory, 6(13):10, 2010.

[9] NS Barrett, L Meyer, N Hill, and PH Walsh. Methods for the processing and scoring of auv digital imagery from south eastern tasmania. 2011.

[10] MS Bewley, B Douillard, N Nourani-Vatani, A Friedman, O Pizarro, and SB Williams. Automated species detection: An experimental approach to kelp detection from sea-floor auv images. In Proceedings of Australasian Conference on Robotics and Automation, Victoria University of Wellington, New Zealand, 2012.

[11] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. Journal of the ACM (JACM), 46(5):720–748, 1999.

[12] Leo Breiman. Random forests. Machine learning, 45(1):5–32, 2001.

[13] R Glenn Brook, Alexander Heinecke, Anthony B Costa, Paul Peltz, Vincent C Betro, Troy Baer, Michael Bader, and Pradeep Dubey. Beacon: Deployment and application of intel xeon phi coprocessors for scientific computing. Computing in Science & Engineering, 17(2):65–72, 2015.

[14] Jaroslaw Bylina and Beata Bylina. Parallelizing nested loops on the intel xeon phi on the example of the dense wz factorization. In Computer Science and Information Systems (FedCSIS), 2016 Federated Conference on, pages 655–664. IEEE, 2016.

[15] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. Using OpenMP: portable shared memory parallel programming, volume 10. MIT press, 2008.

[16] Yi-Wei Chen and Chih-Jen Lin. Combining svms with various feature selection strategies. In Feature extraction, pages 315–324. Springer, 2006.

[17] Arnaldo Carvalho De Melo. The new linuxperftools. In Slides from Linux Kongress, volume 18, 2010.

[18] Lamia Djoudi, Denis Barthou, Patrick Carribault, Christophe Lemuet, Jean-Thomas Acquaviva, William Jalby, et al. Maqao: Modular assembler quality analyzer and optimizer

for itanium 2. In The 4th Workshop on EPIC architectures and compiler technology, San Jose, 2005.

[19] Jack Dongarra, Mark Gates, Azzam Haidar, Yulu Jia, Khairul Kabir, Piotr Luszczek, and Stanimire Tomov. Hpc programming on intel many-integrated-core hardware with magma port to xeon phi. Scientific Programming, 2015:9, 2015.

[20] Gordon Erlebacher, Erik Saule, Natasha Flyer, and Evan Bollig. Acceleration of derivative calculations with application to radial basis function: Finite-differences on the Intel MIC architecture. In Proceedings of the 28th ACM International Conference on Supercomputing, ICS '14, pages 263–272, 2014.

[21] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Edwin Bonilla, John Thomson, Hugh Leather, et al. Milepost gcc: machine learning based research compiler. In GCC summit, 2008.

[22] Martin Gardner. Mathematical games - the fantastic combinations of John Conway's new solitaire game "life". Scientific American, (223):120–123, October 1970.

[23] Miaoqing Huang, Chenggang Lai, Xuan Shi, Zhijun Hao, and Haihang You. Study of parallel programming models on computer clusters with intel mic coprocessors. The International Journal of High Performance Computing Applications, page 1094342015580864, 2015.

[24] John R Jensen. Introductory Digital Image Processing: A Remote Sensing Perspective (3rd ed). Prentice Hall, Upper Saddle River, New Jersey, May 2004.

[25] Rashid Kaleem, Rajkishore Barik, Tatiana Shpeisman, Chunling Hu, Brian T Lewis, and Keshav Pingali. Adaptive heterogeneous scheduling for integrated gpus. In Parallel Architecture and Compilation Techniques (PACT), 2014 23rd International Conference on, pages 151–162. IEEE, 2014.

[26] K Kandalla, Akshay Venkatesh, Khaled Hamidouche, Sreeram Potluri, Devendar Bureddy, and Dhabaleswar K Panda. Designing optimized mpi broadcast and allreduce for many integrated core (mic) infiniband clusters. In High-Performance Interconnects (HOTI), 2013 IEEE 21st Annual Symposium on, pages 63–70. IEEE, 2013.

[27] S Sathiya Keerthi and Chih-Jen Lin. Asymptotic behaviors of support vector machines with gaussian kernel. Neural computation, 15(7):1667–1689, 2003.

[28] David B. Kirk and Wen-mei W. Hwu. Programming Massively Parallel Processors: A Hands-on Approach (2nd ed). Morgan Kaufmann, Burlington, MA, December 2012.

[29] Klaus Kofler, Ivan Grasso, Biagio Cosenza, and Thomas Fahringer. An automatic input-sensitive approach for heterogeneous task partitioning. In Proceedings of the 27th international ACM conference on International conference on supercomputing, pages 149–160. ACM, 2013.

[30] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In Ijcai, volume 14, pages 1137–1145. Montreal, Canada, 1995.

[31] Kevin E Kohler and Shaun M Gill. Coral point count with excel extensions (cpce): A visual basic program for the determination of coral and substrate coverage using random point count methodology. Computers & Geosciences, 32(9):1259–1269, 2006.

[32] Géraud Krawezik. Performance comparison of MPI and three OpenMP programming styles on shared memory multiprocessors. In Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '03, pages 118–127, 2003.

[33] Chenggang Lai, Miaoqing Huang, and Genlang Chen. Towards optimal task distribution on computer clusters with intel mic coprocessors. In High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on, pages 811–814. IEEE, 2015.

[34] Chenggang Lai, Miaoqing Huang, and Xuan Shi. Accelerating the calculation of minimum set of viewpoints for maximum coverage over digital elevation model data by hybrid computer architecture and systems. SIGSPATIAL Special, 8(3):3–4, 2017.

[35] Chenggang Lai, Miaoqing Huang, Xuan Shi, and Haihang You. Accelerating geospatial applications on hybrid architectures. algorithms, 18:19, 2013.

[36] Alexey Lastovetsky, Lukasz Szustak, and Roman Wyrzykowski. Model-based optimization of eulag kernel on intel xeon phi through load imbalancing. IEEE Transactions on Parallel and Distributed Systems, 28(3):787–797, 2017.

[37] Honglak Lee. UNSUPERVISED FEATURE LEARNING VIA SPARSE HIERARCHICAL REPRESENTATIONS. PhD thesis, Stanford University, Stanford, CA, August 2010.

[38] Honglak Lee, Alexis Battle, Rajat Raina, and Andrew Y Ng. Efficient sparse coding algorithms. In Advances in neural information processing systems, pages 801–808, 2006.

[39] Michael D Linderman, Jamison D Collins, Hong Wang, and Teresa H Meng. Merge: a programming model for heterogeneous multi-core systems. In ACM SIGOPS operating systems review, volume 42, pages 287–296. ACM, 2008.

[40] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on, pages 45–55. IEEE, 2009.

[41] Rama Malladi, Richard Dodson, and Vyacheslav Kitaeff. Intel many integrated core (MIC) architecture: Portability and performance efficiency study of radio astronomy algorithms. In Proceedings of the 2012 Workshop on High-Performance Computing for Astronomy Date, Astro-HPC '12, pages 5–6, 2012.

[42] Andres More. Intel xeon phi coprocessor high performance programming. Journal of Computer Science & Technology, 13, 2013.

[43] NVIDIA Corporation. NVIDIA's next generation CUDA compute architecture: Fermi. White paper V1.1, June 2009. Available online on http://www.nvidia.com.

[44] NVIDIA Corporation. NVIDIA's next generation CUDA compute architecture: Kepler gk110. White paper V1.0, 2012. Available online on http://www.nvidia.com.

[45] Bruno A Olshausen et al. Emergence of simple-cell receptive field properties by learning a sparse code for natural images. Nature, 381(6583):607–609, 1996.

[46] API OpenMP. Specification for parallel programming, 2010.

[47] Sethuraman Panchanathan and GR Andrews. Foundations of parallel and distributed programming. 1998.

[48] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. Journal of machine learning research, 12(Oct):2825–2830, 2011.

[49] James C Phillips, John E Stone, and Klaus Schulten. Adapting a message-driven parallel application to gpu-accelerated clusters. In High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for, pages 1–9. IEEE, 2008.

[50] Rajat Raina, Anand Madhavan, and Andrew Y Ng. Large-scale deep unsupervised learning using graphics processors. In Proceedings of the 26th annual international conference on machine learning, pages 873–880. ACM, 2009.

[51] S. Saini, R. Ciotti, B.T.N. Gunney, T.E. Spelce, A. Koniges, D. Dossa, P. Adamidis, R. Rabenseifner, S.R. Tiyyagura, M. Mueller, and R. Fatoohi. Performance evaluation of supercomputers using HPCC and IMB benchmarks. In Proc. 20th International on Parallel and Distributed Processing Symposium (IPDPS), April 2006.

[52] Subhash Saini, Robert Ciotti, Brian T. N. Gunney, Thomas E. Spelce, Alice Koniges, Don Dossa, Panagiotis Adamidis, Rolf Rabenseifner, Sunil R. Tiyyagura, and Matthias Mueller. Performance evaluation of supercomputers using HPCC and IMB benchmarks. J. Comput. Syst. Sci., 74(6):965–982, September 2008.

[53] Subhash Saini, Haoqiang Jin, Dennis Jespersen, Huiyu Feng, Jahed Djomehri, William Arasin, Robert Hood, Piyush Mehrotra, and Rupak Biswas. An early performance evaluation of many integrated core architecture based sgi rackable computing system. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '13, pages 94:1–94:12, 2013.

[54] Erik Saule and Umit V Catalyurek. An early evaluation of the scalability of graph algorithms on the intel mic architecture. In Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International, pages 1629–1639. IEEE, 2012.

[55] Dirk Schmidl, Tim Cramer, Sandra Wienke, Christian Terboven, and Matthias S. Müller. Assessing the performance of OpenM programs on the Intel Xeon Phi. In Proc. Euro-Par 2013 Parallel Processing, Lecture Notes in Computer Science Volume 8097, pages 547–558, August 2013.

[56] Bernhard Scholkopf, Kah-Kay Sung, Christopher JC Burges, Federico Girosi, Partha Niyogi, Tomaso Poggio, and Vladimir Vapnik. Comparing support vector machines with

gaussian kernels to radial basis function classifiers. IEEE transactions on Signal Processing, 45(11):2758–2765, 1997.

[57] Panagiotis Spentzouris, James Amundson, and Alexandru Macridin. High performance computing modeling advances accelerator science for high energy physics. 2014.

[58] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. Computing in science & engineering, 12(3):66–73, 2010.

[59] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Center for Reliable and High-Performance Computing, 127, 2012.

[60] Manfred K Warmuth and Dima Kuzmin. Randomized online pca algorithms with regret bounds that are logarithmic in the dimension. Journal of Machine Learning Research, 9(10), 2008.

[61] Fulong Wu. Calibration of stochastic cellular automata: the application to rural-urban land conversions. International Journal of Geographical Information Science, 16(8):795–818, 2002.

[62] Biwei Xie, Xu Liu, Sally A McKee, Jianfeng Zhan, Zhen Jia, Lei Wang, and Lixin Zhang. Understanding data analytics workloads on intel (r) xeon phi (r). In 2016 IEEE 18th International Conference on High-Performance Computing and Communications, IEEE 14th International Conference on Smart City, and IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), pages 206–215. IEEE, 2016.

[63] Canqun Yang, Feng Wang, Yunfei Du, Juan Chen, Jie Liu, Huizhan Yi, and Kai Lu. Adaptive optimization for petascale heterogeneous cpu/gpu computing. In Cluster Computing (CLUSTER), 2010 IEEE International Conference on, pages 19–28. IEEE, 2010.

[64] Ziming Zhong, Vladimir Rychkov, and Alexey Lastovetsky. Data partitioning on heterogeneous multicore platforms. In Cluster Computing (CLUSTER), 2011 IEEE International Conference on, pages 580–584. IEEE, 2011.