12-2014

Study of Parallel Programming Models on Computer Clusters with Accelerators

Chenggang Lai
University of Arkansas, Fayetteville

Study of Parallel Programming Models on Computer Clusters with Accelerators

Study of Parallel Programming Models on Computer Clusters with Accelerators

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science 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 2014
University of Arkansas

This thesis is approved for recommendation to the Graduate Council

Dr. Miaoqing Huang.
Thesis Director

Dr. John Gauch.
Committee Member

Dr. Wing Ning Li.
Committee Member

Abstract

In order to reach exascale computing capability, accelerators have become a crucial part in developing supercomputers. This work examines the potential of two latest acceleration technologies, Intel Many Integrated Core (MIC) Architecture and Graphics Processing Units (GPUs). This thesis applies three benchmarks under 3 different configurations, MPI+CPU, MPI+GPU, and MPI+MIC. The benchmarks include intensely communicating application, loosely communicating application, and embarrassingly parallel application. This thesis also carries out a detailed study on the scalability and performance of MIC processors under two programming models, i.e., offload model and native model, on the Beacon computer cluster.

According to different benchmarks, the results demonstrate different performance and scalability between GPU and MIC. (1) For embarrassingly parallel case, GPU-based parallel implementation on Keeneland computer cluster has a better performance than other accelerators. However, MIC-based parallel implementation shows a better scalability than the implementation on GPU. The performances of native model and offload model on MIC are very close. (2) For loosely communicating case, the performances on GPU and MIC are very close. The MIC-based parallel implementation still demonstrates a strong scalability when using 120 MIC processors in computation. (3) For the intensely communicating case, the MPI implementations on CPUs and GPUs both have a strong scalability. GPUs can consistently outperform other accelerators. However, the MIC-based implementation cannot scale quite well. The performance of different models on MIC is different from the performance of embarrassingly parallel case. Native model can consistently outperform the offload model by $\sim 10\times$. And there is not much performance gain when allocating more MIC processors. The increase of communication cost will offset the performance gain from the reduced workload on each MIC core. This work also tests the performance capabilities and scalability by changing the number of threads on each MIC card form 10 to 60. When using different number of threads for the intensely communicating case, it shows different capabilities of the MIC based offload model. The scalability can hold when the number of threads increases

from 10 to 30, and the computation time reduces with a smaller rate from 30 threads to 50 threads. When using 60 threads, the computation time will increase. The reason is that the communication overhead will offset the performance gain when 60 threads are deployed on a single MIC card.

Acknowledgments

This thesis is partially supported by funds from the National Science Foundation and the National Aeronautics and Space Administration. This thesis uses resources of the Keeneland Computer Cluster at the Georgia Institute of Technology and the Beacon Computer Cluster at the National Institute for Computational Sciences. The author thanks Nvidia Corporation for GPU donations.

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.

OpenCL Open Computing Language. A framework for programs executing on heterogeneous platforms.

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

Table of Contents

List of Figures

List of Tables

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. People have to seek more powerful machines to deal with these data because traditional software and desktops have become inefficient or impossible to process big data. when the applications require low latency and high bandwidth network, and high computing capability, high-performance computing 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, such as GIS 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 order-of-magnitude improvement in performance. Many computer architectures have been implemented. They provide good platforms to employ parallelism for achieving performance and scalability. Graphics Processing Units (GPUs) have become popular hardware to accelerate applications. Various performance optimization techniques have been developed over the past several years. A couple of programming languages support GPU, such as Nvidia's CUDA and OpenCL. Another new architecture of accelerator is Intel's Many Integrated Core (MIC) Architecture. Compared with traditional computer clusters that contain only CPUs, the programming models on MIC are more complicated. There are three programming models for MIC, including the offload mode in which the computation is offloaded from host CPUs to accelerators, the native mode in which all MPI processes are directly running on accelerators, and the hybrid mode in which the MPI processes are running on both host CPUs and accelerators. It is important to use the proper programming model to implement a specific application depending on its intrinsic parallelism. Every model has its own features. According to different models, programmers can adjust the code to achieve the best performance and scalability. In both high-performance computing and embedded computing, saving power is a very important issue. A balanced workload distribution between the host processors and the accelerators is necessary to achieve both the high performance and the energy efficiency.

High-performance computing with accelerators is an exciting area. Vendors are producing new technologies at a quick pace. New programming models and optimization techniques need to be developed to address the new features and functions provided by new devices. How to keep improving the performance of large-scale hybrid computer clusters under reasonable power budget is also a challenge to be solved. Given the recent interest in accelerators and new challenges, this thesis addresses some of the issues with accelerator architectures, programming models, applications and performance evaluation. The thesis consists of 6 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. The accelerators architectures, including Intel MIC, Nvidia Kepler and Fermi GPU, and their programming models and languages are discussed in Chapter 3. The applications of different parallel cases, including Kriging Interpolation, Iterative Self-Organizing Data Analysis Technique Algorithm (ISODATA) and Cellular Automata are illustrated in Chapter 4. Chapter 5 demonstrates the platforms this thesis used and the results of every algorithm based on different accelerating architectures. Chapter 6 concludes the whole thesis.

Chapter 2

Related Work

The scale of high-performance computing is growing rapidly. The rapid growth of high-performance computing systems provides the potential of performance improvement and the capability to deal with the increased problem scale. The development of engineering and science applications, such as biology [16], climate [24], and physics [23], is becoming increasingly complex when these applications need more compute cores, memory and disk storage. The experience with applications of high-performance computing can provide the confidence that performance and the capacity of supercomputers will continue to grow.

As a part of high-performance computing, accelerators are becoming more important for performance and scalability. The accelerators are designed with the goal of achieving higher performance. GPU has become a potential accelerator in high-performance computing domain. There are many successful implementations of applications on GPU clusters, such as N-body [19] and WRF [18]. The heterogeneous workload of cooperative CPU and GPU [4] is studied to optimize linear algebra libraries. For achieving a better performance, an architecture using Dynamic Memory Management, such as GPUdmm, is introduced in [13].

Recently, Intel demonstrated a new hardware architecture called Many Integrated Core(MIC) as accelerators for high-performance computing domain. 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 [20]. As the popular accelerator architectures, researchers pay more attention to the MIC and Kepler GPU. Some comparisons of Kepler GPU and MIC based specific benchmarks are introduced in [6]. The MIC is the accelerator architecture based on conventional CPUs. Therefore, some researchers designed optimal MPI library for MIC, such as MPI Broadcast and Allreduce functions for InfiniBand Clusters [12], and the MPI Alltoall and Allgather design for InfiniBand Clusters [27].

Chapter 3

Background

3.1 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 [5]. Figure 3.1 shows a simple serial process.

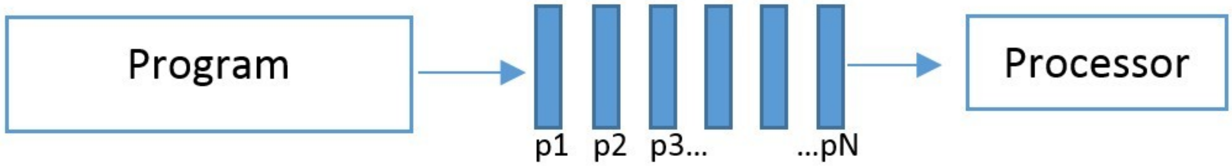


Figure 3.1: Serial Processing.

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 3.2 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.

3.2 Parallel programming models and languages

3.2.1 MPI

3.2.1.1 Overview

MPI stands for message-passing interface. It supports independent language communications protocol so that it is possible to use multiple computers for parallel programming. MPI program-

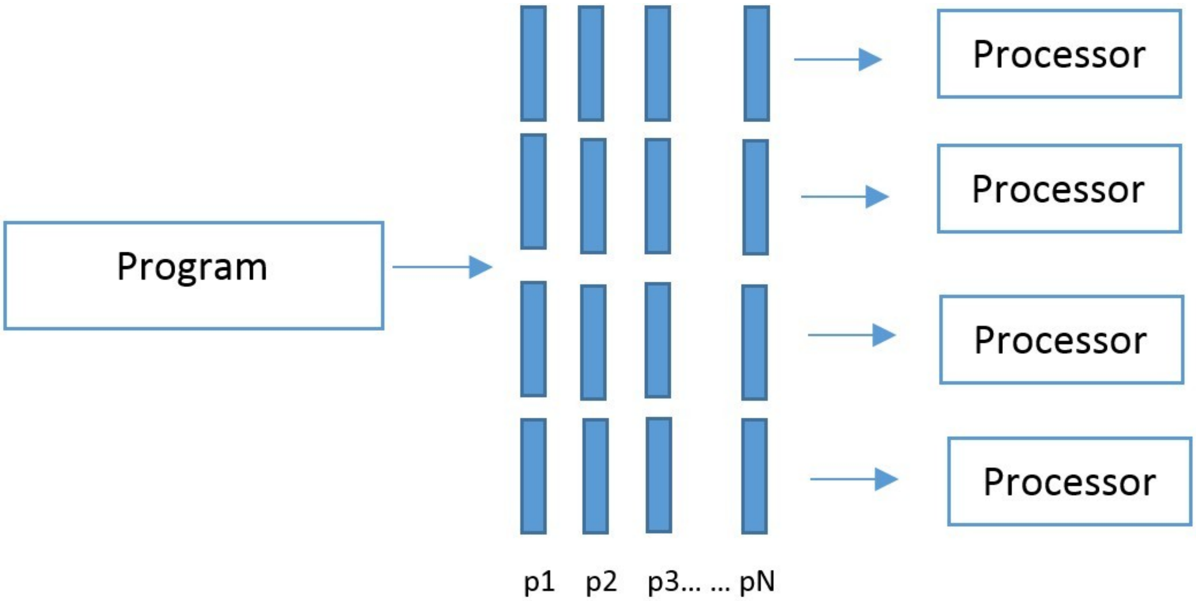


Figure 3.2: Parallel Processing.

ming can use most of hardware resources and it can bring high performance, portability and scalability. MPI has lots of fundamental functions, such as MPI__Send, MPI__Rec and MPI__Allreduce. These commands are not only for buffered receives and sends, but also they can transfer data. MPI is good at communication and provides a set of collective communication functions, such as communicators, point-to-point communication and collective. The MPI has two standards. The first MPI standard was MPI-1, which completed in 1994. The second standard called MP-2 was completed in 1998.

3.2.1.2 Communicators

Communicator becomes an important part of MPI library. It connects groups of processes. Every communicator gives contained process an identifier. Every process that belongs to the communicator is arranged in an ordered topology. Processes that belong to different communicators cannot send and receive messages from each other. MPI has definite groups in which MPI can organize and reorganize groups of processes before other communicators are made. In MPI-1, the operations of single group are most popular. However, bilateral operations became common in MPI-2 because MPI-2 includes dynamic collective communication.

3.2.1.3 Point-to-Point

Point-to-point communication is a crucial function of MPI library. A process can receive or send messages to another specific process. For example, the MPI__Send can allow one specific process to send a message to another specific process. When program has lots of share data, point-to-point communication become very useful. Every process can send share data to the specific process. This communication is better for master-slave architecture in which the master process sends task data to slaves whenever the prior task is done.

3.2.1.4 Collective

Unlike point-to-point communication, Collective can receive or send messages to all processes in a process group. For example, MPI__Reduce collects data from all processes in a group, and then stores data to master thread. MPI__Reduce is usually used at the start and end of distributed calculation. Each process only carries out a part of data and combines them to final result. A reverse operation is MPI__Bcast. The operation sends data from one process to all processes in process group.

3.2.2 OpenMP

OpenMP is an application programming interface (API) that supports multi-threads programming. It uses fork technology to decide how many threads will be used. 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.

3.2.3 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 has been used to accelerate many applications in different areas, such as geospatial computing, biology, cryptography and other fields.

3.3 Accelerators Architectures

3.3.1 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.

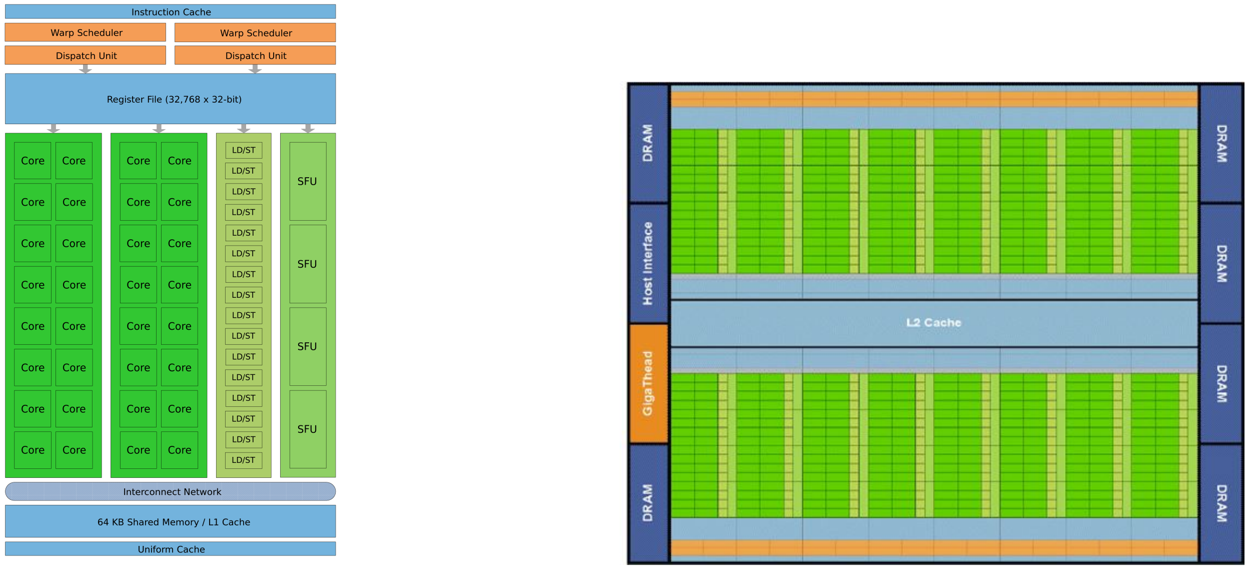
3.3.2 GPU

3.3.2.1 Overview

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. 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. 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. Nowadays, the two major GPU designers are Nvidia and AMD. Nvidia develops CUDA to support GPU programming. OpenCL is also supported by Nvidia's GPU. OpenCL is designed to work for architectures of multiple types, such as CPUs, GPU and DSP. Both 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 parts on GPU or CPU. It can take advantage of the ability of GPU and CPU to perform their own appropriate work.

3.3.2.2 Fermi Architecture

The Fermi architecture is the previous architecture of Nvidia's GPU. It is implemented with 3.0 billion transistors and 512 CUDA cores. A core can support both integer or floating-point operations. In the new model of Fermi architecture, the core also supports double-precision floating-point operations, such as Tesla C2075. Basically, the GPU has 16 stream multiprocessors, as shown in Figure 3.3(a). Each core has a fully pipelined integer arithmetic logic unit and floating-



(a) Fermi Streaming Mulitiprocessor.

(b) Fermi Parallel Processing.

Figure 3.3: Nvidia's Fermi GPU architecture [7].

point unit. The GPU has six 64-bit memory partitions, supporting up to a total of 6 GB of GDDR5 DRAM memory. The interface connects the GPU to the CPU via PCI-Express. The GigaThread global scheduler distributes thread blocks to stream multiprocessors thread schedulers [7]. In this architecture, 16 stream multiprocessors are allocated around L2 cache, as shown in Figure 3.3(b).

Fermi architecture of GPU brings researchers into a new era in high-performance computing. Based on the hybrid computing models, CPU and GPU can work together to solve large-scale computational problems, such as embarrassingly parallel case, loosely communicating and intensely communicating case. Nvidia Fermi GPUs have supported some of the fastest supercomputers, such as Keeneland supercomputer sponsored by NSF [1].

3.3.2.3 Kepler Architecture

Nvidia's latest Kepler GPU architecture contains 15 streaming multiprocessors (SMX) as shown in Figure 3.4, which consists of 192 single-precision cores and 64 double-precision cores. It is designed to maximize the performance of computation with power efficiency. Innovations of Kepler architecture can make hybrid computing more accessible and easier. The architecture supports

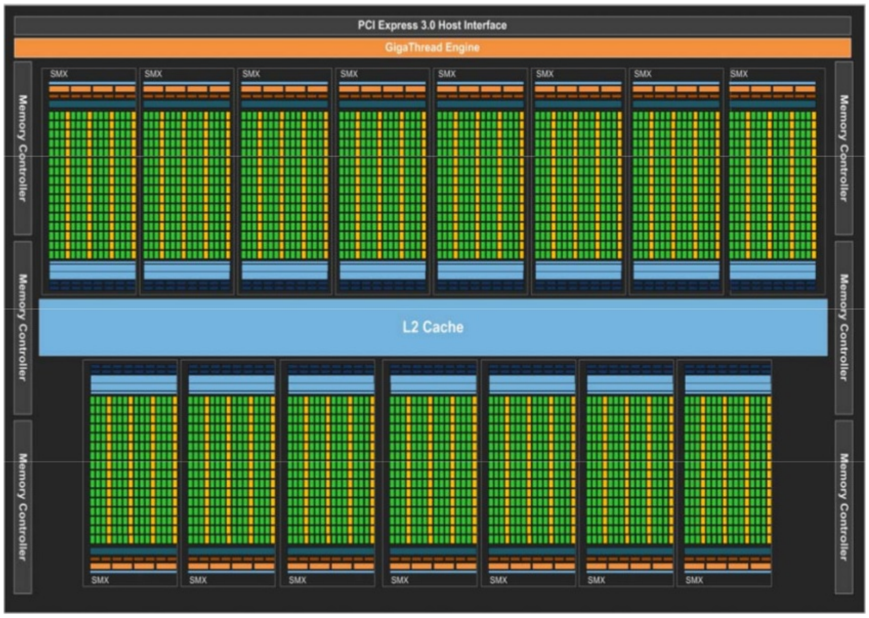


Figure 3.4: Kepler architecture.

integer, single precision and double precision operations. It also has higher memory bandwidth. There are three advanced features. (1) Efficiently share the GPU resources among multiple host threads/processes, such as Hyper-Q; (2) flexibly create new kernels on GPU (i.e., Dynamic Parallelism); and (3) reduce communication overhead across GPUs through GPUDirect.

SMX is the next generation streaming multiprocessor as shown in Figure 3.5. The SMX of Kepler architecture pays more attention to double precision performance because double precision arithmetic is used in many high-performance applications.

Dynamic parallelism is one of the most advanced technologies in Kepler architecture as shown in Figure 3.6. New Dynamic Parallelism can enable GPU to allocate threads dynamically without going back to CPU. As a result, it can decrease the overhead time and can reduce programmer's work of revising code from CPU to GPU. It maintains the similar syntax for GPU workloads as CPU kernel launches. When application is made of small or medium sized parallel parts, this feature can take its advantages for achieving a good performance.

Hyper-Q can take full advantage of GPU resources. It can enable multiple cores to launch work simultaneously on a single GPU as shown in Figure 3.7. As a result, it can improve utilization of GPU and decrease the idle time of CPU. It is a flexible solution that has a connection between

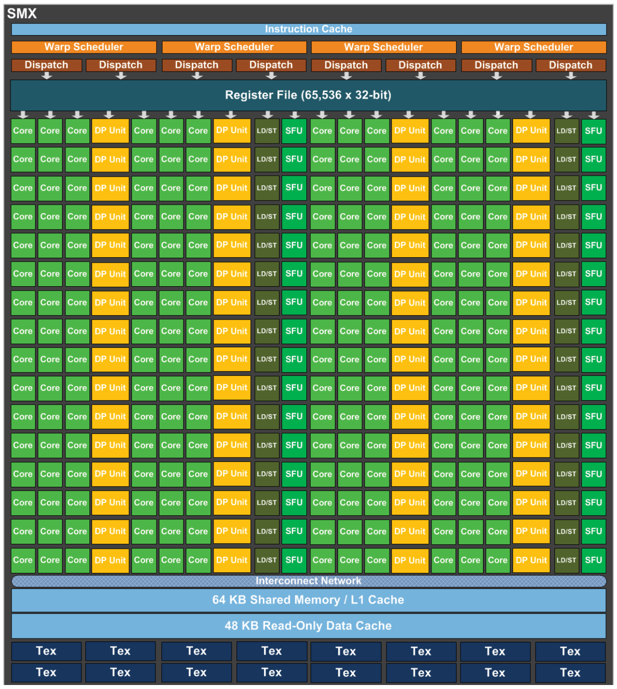

SMX
Instruction Cache
Warp Scheduler
Dispatch
Register File (65,536 x 32-bit)
Core
DP Unit
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
48 KB Read-Only Data Cache
Tex

Figure 3.5: SMX architecture.

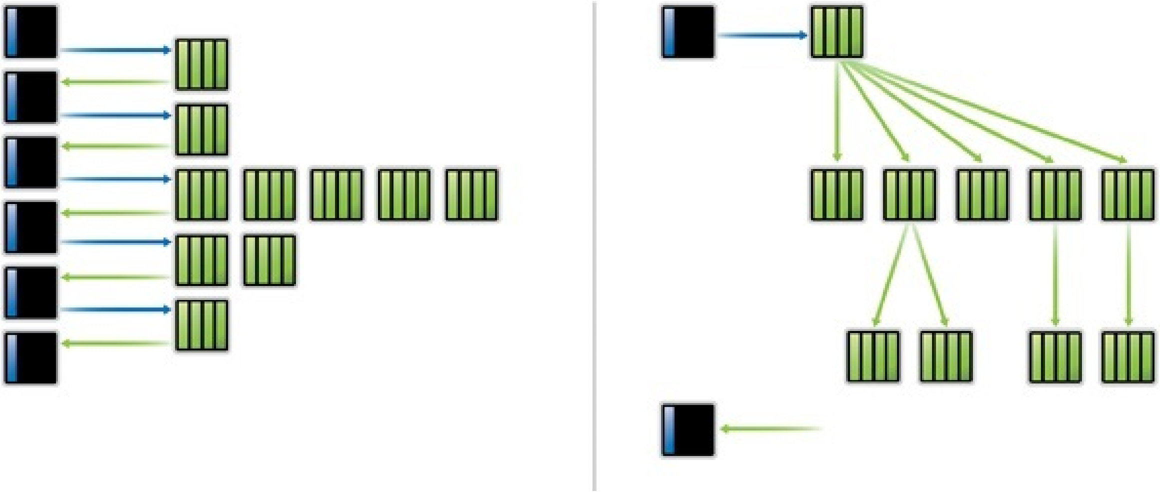


Figure 3.6: Dynamic Parallelism [17].

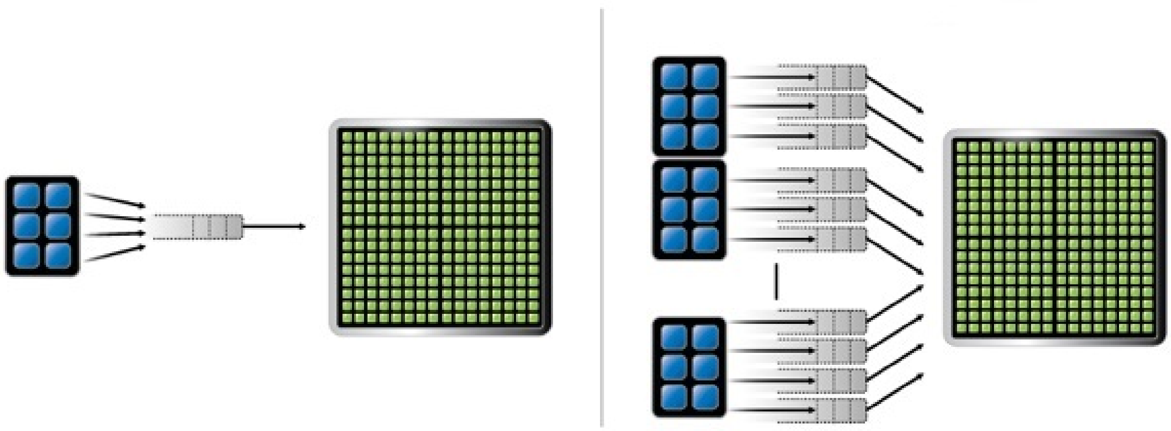


Figure 3.7: HYPER-Q [17].

CUDA streams and MPI.

3.3.3 MIC

MIC stands for Many Integrated Core architecture provided by Intel. It 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, as shown in Figure 3.8. These cores are connected through a high-speed ring bus and each core supports 4 threads. Because every core is a low-weight classic processor, the MIC can support traditional parallel programming models, such as OpenMP and MPI. The MIC has two programming models to parallelize application, native model and offload model.

In the offload model, MPI runs on the host CPU, which offloads the computation to the MIC processors using OpenMP, as shown in Figure 3.9(a). In this case, the MIC processor behaves like a typical accelerator. In the native model, MPI directly runs on each MIC cores as shown in Figure 3.9(b). In this case, each MIC core behaves like a stand-alone processor that runs its own OS and MPI.

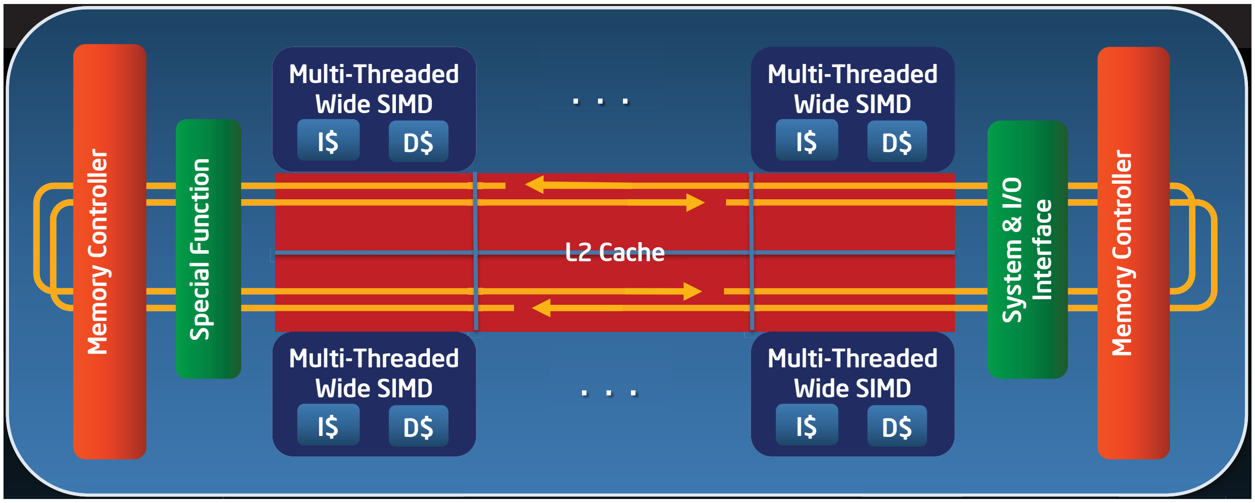
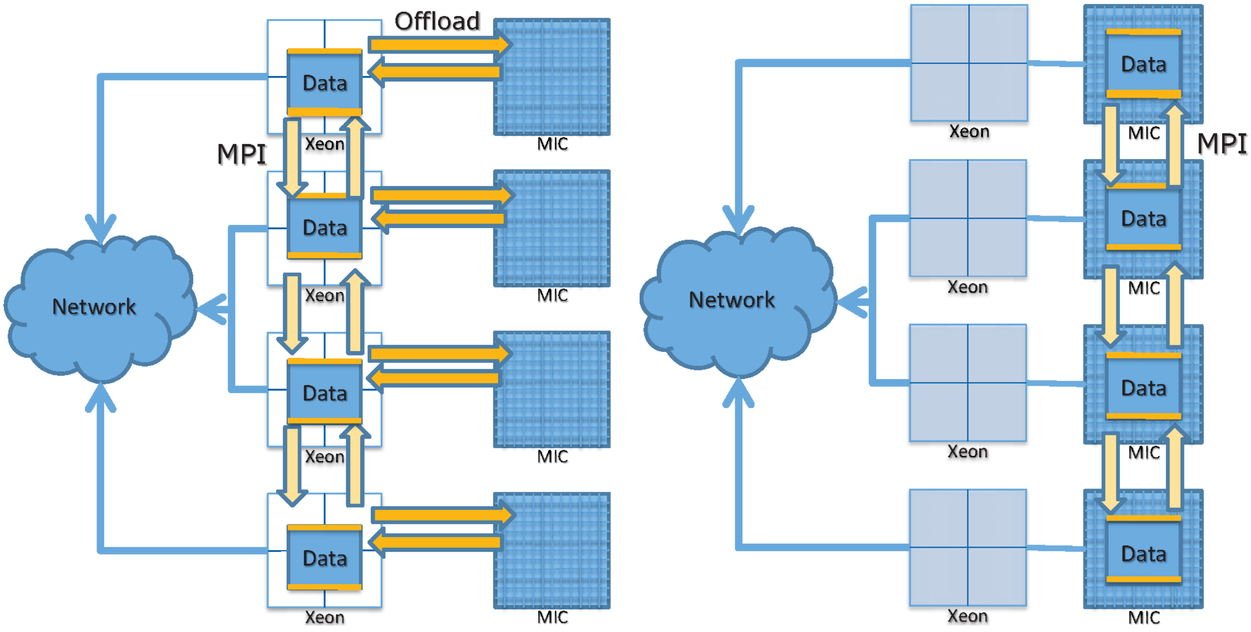


Figure 3.8: The architecture of Intel Xeon Phi coprocessor (MIC) [3].



(a) MIC Offload Model.

(b) MIC Native Model.

Figure 3.9: Two different approaches to implementing parallelism on MIC cluster.

Chapter 4

Benchmarks

4.1 Overview

This work selects three algorithms as benchmarks to examine the scalability of the computational solution over the hybrid architectures. Considering the complexity of algorithm, the Kriging Interpolation is selected as an embarrassingly parallel case. The Iterative Self-Organizing Data Analysis classification (ISODATA) [11] is selected as a loosely communicating case. Game of life is a intensely communicating case.

4.2 ISODATA

The classification of image has two methods, supervised image classification and unsupervised image classification. Supervised image classification cannot do classification automatically. Researchers have to define samples for every class. Based on the features of different classes, program will classify the whole image. Unsupervised image classification will classify the image automatically given the number of classes. Unsupervised classification is also a foundation for other classification approaches [11], such as supervised classification and object oriented classification [28, 25, 26].

The Iterative Self-Organizing Data Analysis Technique Algorithm (ISODATA) [11] is one of the most frequently used algorithms for unsupervised image classification. Compared with the K-mean algorithm, the ISODATA has some future refinements by merging and splitting of classes. Classes can be merged if either the number of pixels in a class is less than a certain threshold or if the clusters of two classes are closer than a certain threshold. Classes are split into two different clusters if the cluster standard deviation exceeds a predefined value and the number of pixels is twice the threshold for the minimum number of pixels. The time complexity of ISODADA algorithm is O(M*N). M is the maximum number of iterations and N is the number of objects. When using large-scale images as original data, the serial implementation on a desktop computer will spend a long time. A parallel implementation on high performance computer can provide a promising solution that can shorten the time of computation. This work uses the Geospatial Data Abstraction Library (GDAL) application programming interface [10] to read original data from files. GDAL is convenient for reading high-resolution image in parallelism.

This work implements the ISODATA program in three steps by using parallel programming. (1) Calculate the initial mean vector of every class, (2) classify each pixel to the nearest class, and (3) calculate the new class means based on all pixels in one class. The second and third steps will not stop until difference value between two iterations is small enough. In the ISODATA classification, some parameters need to be determined. We define that the number of classes is 15. The convergence threshold, i.e., the ratio of unchanged pixels for classification between two contiguous iterations, is 0.95. The maximum number of iterations is 15.

In each iteration, the pixels will be compared to the means of new class and assigned to the nearest class. The initial means of classes are decided by the statics of original data sets. The changes between two consecutive iterations can be achieved by parameter of label so that it can calculate the ratio of pixels that are not changing. The process of iteration will stop when the number of iterations becomes maximum or the ratio is over convergence threshold.

Dhodhi [8] implemented a distributed ISODATA application over a cluster of eight workstations to process a 12-spectral band image for a size of 512×512 pixels in each band. For the parallel programming, it is normal to use master-slave model to calculate. Master core will read the original data and distribute the different date to the slave cores. However, it is not a good solution for big data because master thread has to spend a long time on reading original data. Consider the supercomputing capability of Keeneland cluster and Beacon cluster, this work uses each core to read a segment of data simultaneously and all core will do the same jobs in parallelism. After each iteration, it can use MPI__Allreduce function to request information from each cores and then broadcast to all cores for next iteration.

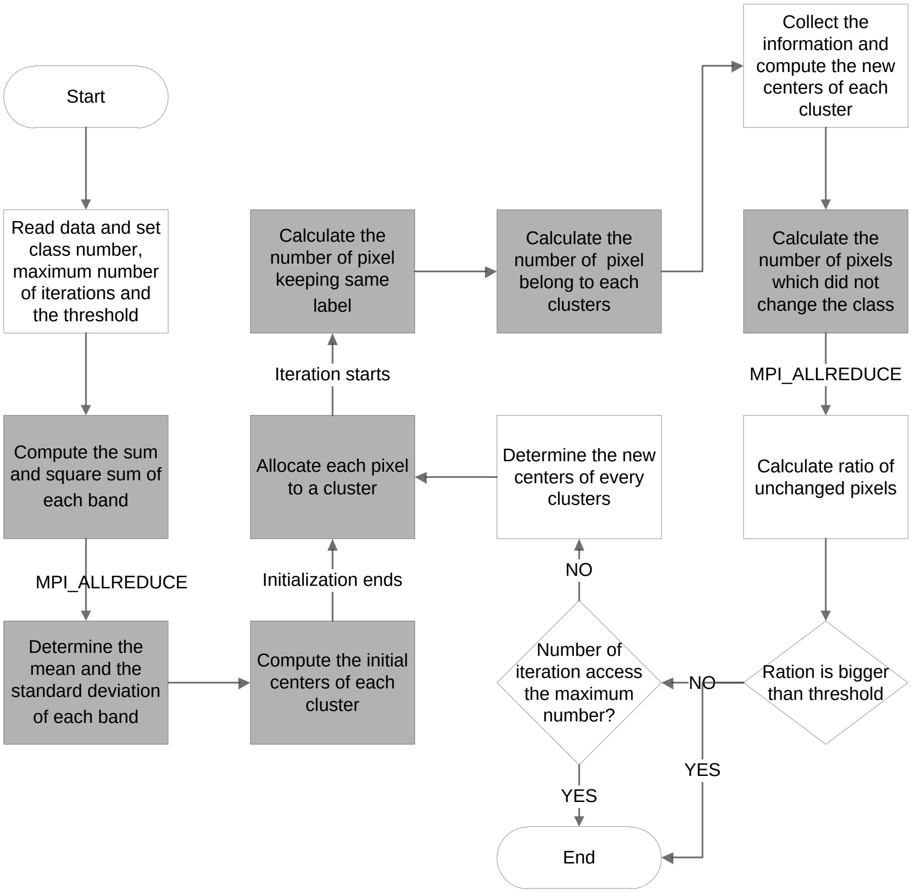
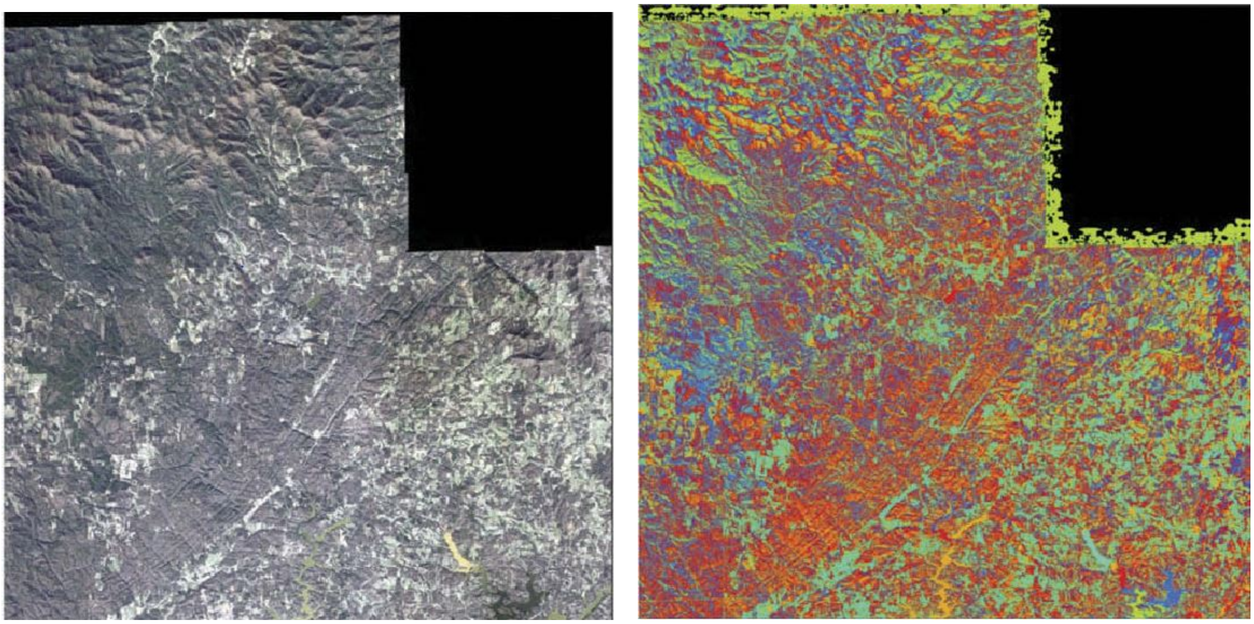

Start
Read data and set class number, maximum number of iterations and the threshold
Compute the sum and square sum of each band
MPI_ALLREDUCE
Determine the mean and the standard deviation of each band
Compute the initial centers of each cluster
Initialization ends
Allocate each pixel to a cluster
Iteration starts
Calculate the number of pixel keeping same label
Calculate the number of pixel belong to each clusters
Collect the information and compute the new centers of each cluster
Calculate the number of pixels which did not change the class
MPI_ALLREDUCE
Calculate ratio of unchanged pixels
Ration is bigger than threshold
NO
YES
Number of iteration access the maximum number?
NO
YES
Determine the new centers of every clusters
End

Figure 4.1: ISODATA Dataflow for CPU [21].

This work uses cores to virtually divide and read original data partially through GDAL library. The first MPI process will deal with all residual pixels in the classification calculation process. During every iteration of classification processing, information of all pixels will be collected by MPI__Allreduce function. The work flow of implementing ISODATA using MPI is shown in Figure 4.1. The implementation will set three required parameters, i.e., the number of classes, convergence threshold and maximum number of iterations. When each processors read its own data from image file, the sum and square sum of each band are calculated to decide the initial centers of each class. When pixels are assigned to different classes, which are based on the Euclidean distance, some calculations are implemented to decide the amount of data remaining with the same class labels and get the ratio of unchanged pixel. If the ratio is larger than the threshold or the



(a) A 18-GB image data for ISODATA.

(b) Result of ISODATA.

Figure 4.2: Source data and Classification result [21].

number of iteration exceeds the maximum number, the classification process will end. Otherwise, the program begins a new iteration to calculate the new centers and reassign each pixel to classes.

This work also implements GPU to optimize the ISODATA. The hybrid MPI + GPU programs needs to take some extra procedures to copy the data back and forth between CPUs and GPUs as shown in Figure 4.1. The GPU parts will be demonstrated by gray background. Figure 4.2 demonstrates the original image of 18 GB high-resolution and the result of classification.

4.3 Kriging Interpolation

Kriging Interpolation is based on statistical models, which stands for the statistical relationships among the measured points. It can provide some measures of the accuracy of predictions. The Kriging Interpolation assumes the direction or distance between points can reflects a spatial correlation. It uses a specified numbers of points or all points from a fixed radius to decide the output value of locations, such as temperature and atmospheric pressure value. When data includes the information about spatially correlated distance or directional bias, Kriging Interpolation can be

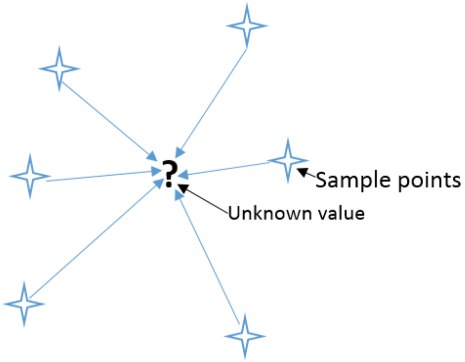


Figure 4.3: Kriging Interpolation Sample

more appropriate to deal with them.

Kriging Interpolation can be treated 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 and w_i is the weight of the i-th input point. in (4.1).

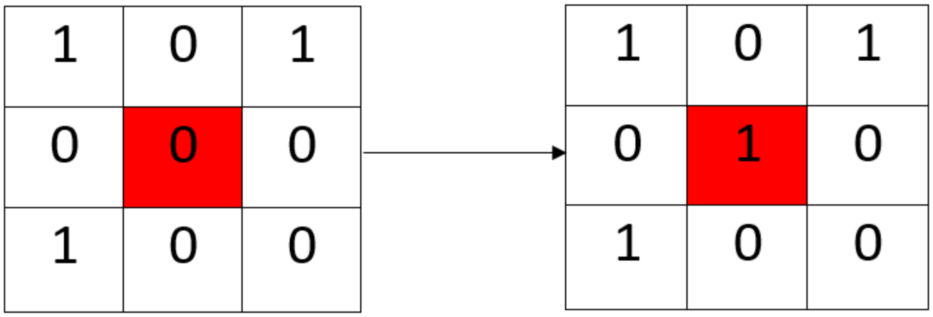
$$\hat{Z}(x,y) = \sum_{i=1}^{k} w_i Z_i, \tag{4.1}$$

In general, users can define a number k and sum over k nearest neighbors of the estimated point. If faring from the estimated point, the sample has less impact on the estimated point. The value at an unknown point should be calculated by k nearest known neighbors as shown in Figure 4.3.

In a parallel computing environment, each MPI thread can be used to calculate one cell. Since there is no dependency among MPI processors, Kriging can be a typical case of embarrassing parallelism.

4.4 Cellular Automata

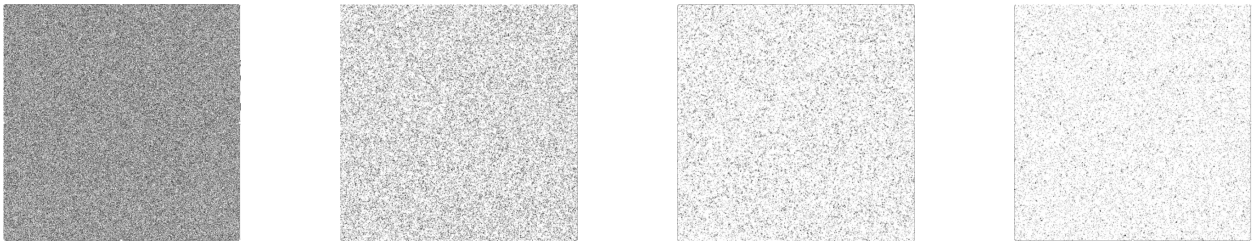
Cellular Automata includes a regular grid of cells and several states, such as “On” and “Off”. A set of cells that are adjacent to a special cell are called the neighborhood of the cell. First, an initial state is decided by assigning a state for each cell. Usually the program can use random



1	0	1
0	0	0
1	0	0

1	0	1
0	1	0
1	0	0

Figure 4.4: Cellular Automata Sample



(a) Stage 1. (b) Stage 2. (c) Stage 3. (d) Stage 4.

Figure 4.5: Game of Life Growing Process.

function to decide the state of each cell, such as 0 or 1. Number 0 means “Off” and 1 means “On”. According to some rules, program can determine the new state of each cell by the current state of theirs neighbors. Furthermore, the rule will be not changed over time and be applied to the whole grid.

The program can simulate a two dimensional cellular automaton and every square become a “cell”. Each cell has two possible states, 0 or 1, as shown in Figure 4.4. One cell has 8 neighbors next to it. The rule will decide the center cell to become 1 or 0 on the next time interval. A sample of 4 stages is shown in Figure 4.5. As the simulation proceeds, each stage of Game of Life can get fewer live cells than previous stage.

Conway’s Game of Life (GOL) is a well-known classical Cellular Automata model [9]. According to the transition rules, a cell can live or die based on the condition of its 3×3 neighborhood. According to previous work [22], the pseudocode of such transition rules can be described as algorithm 1.

Algorithm 1 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
        if $n < 2$ then
            Cell dies of under-population at time t+1
        end if
        if $n = 2$ or $n = 3$ then
            Cell survives at time t+1
        end if
    end if
    if $n = 3$ and Cell is dead at time t  then
        Cell becomes a live cell by reproduction at time t+1
    end if
end function
```

Chapter 5

Implementation and Result

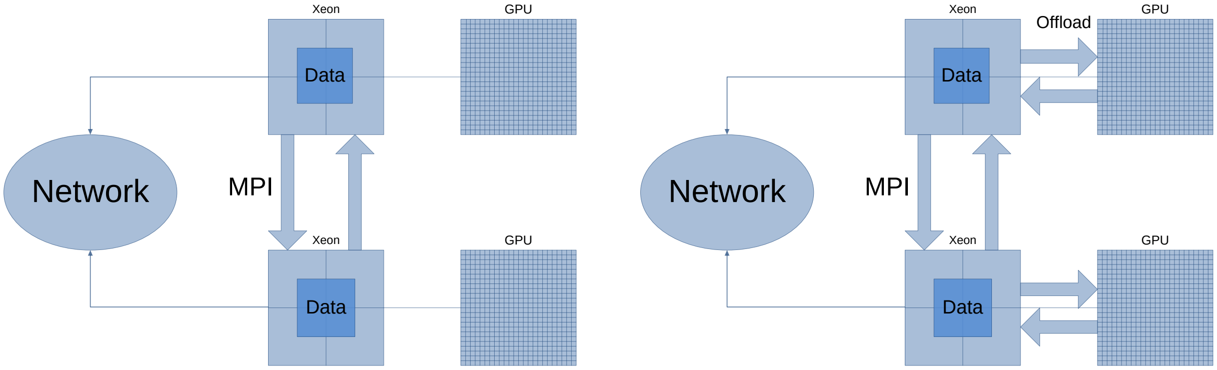
5.1 Experiment Platform

5.1.1 Tesla M2090 GPU and Keeneland

Keeneland [1] is a hybrid supercomputing cluster which was developed by the Georgia Institute of Technology, the Oak Ridge National Laboratory and the University of Tennessee at Knoxville sponsored by the NSF. Keeneland has two subcluster, Keeneland Kids and full-scale Keeneland. The KIDS system is a 120-node HP SL390 system with 240 Intel Xeon X5660 CPUs and 360 Nvidia Fermi M2090 GPUs, with the nodes connected by a QDR InfiniBand network. Each node has 2 6-core Xeon CPUs and 3 Tesla M2090 GPUs. Each M2090 GPU contains 512 CUDA cores and 6 GB GDDR5 on-board memory.

5.1.2 MIC and Beacon

Beacon is a supercomputing system based on Intel MIC accelerators. It can offer access to 6 I/O nodes and 48 compute nodes by FDR InfiniBand interconnect providing 56 GB/s of bi-directional bandwidth [2]. Each compute node has 4 Intel Xeon Phi coprocessors 5110P, two Intel Xeon E5-2670 8-cores CPUs, 256 GB of RAM and 960 GB of SSD storage. Each I/O node offers access to an additional 4.8 TB of SSD storage. Each Intel Xeon Phi coprocessor 5110P includes 60 1.053 GHz MIC cores and 8 GB on-board memory. Beacon provides 768 conventional cores and 11,520 accelerator cores with 12 TB of system memory, 1.5 TB of coprocessor memory and over 210 TFLOP/s of combined computational performance.



(a) CPU parallel implementation.

(b) GPU and CPU parallel implementation.

Figure 5.1: MPI parallel implementation on Keeneland

5.2 Programming models

This work uses 5 parallel programming models on two clusters as follows.

(1) MPI@CPU or MPI+CPU:

MPI-based parallel implementation on Keeneland is shown in Figure 5.1(a). The Intel Xeon E5 8-core 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 2

Algorithm 2 MPI@CPU or MPI+CPU Programming model

```
function ASSIGNMENT($Array\ A, Array\ B$)
    $MPI\_\_Init()$
    $MPI\_\_Comm\_\_rank()$
    $MPI\_\_Comm\_\_size()$
    for $i = 0 \rightarrow k-1$ do
        $A[i] \leftarrow B[i]$
    end for
    $MPI\_\_Finalize()$
end function
```

(2) MPI+GPU:

MPI-based parallel implementation on Keeneland is shown in Figure 5.1(b). Each MPI process runs on Intel Xeon CPU. Every Xeon CPU offloads data to one GPU processor. Keeneland provides Nvidia M2090 GPUs, which are based on Fermi architecture. If m MPI processes are used in application, m CPU processors and m GPU processors are allocated. 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 3.

Algorithm 3 MPI+GPU Programming model

```
function ASSIGNMENT($Array\ A, Array\ B$)
    $MPI\_\_Init()$
    $MPI\_\_Comm\_\_rank()$
    $MPI\_\_Comm\_\_size()$
    Allocate GA and GB on GPU memory
    Copy A and B to GA and GB
    Launch GPU kernel
    for $i = 0 \to k - 1$ do
        $GA[i] \leftarrow GB[i]$
    end for
    Copy GA back to A
    $MPI\_\_Finalize()$
end function
```

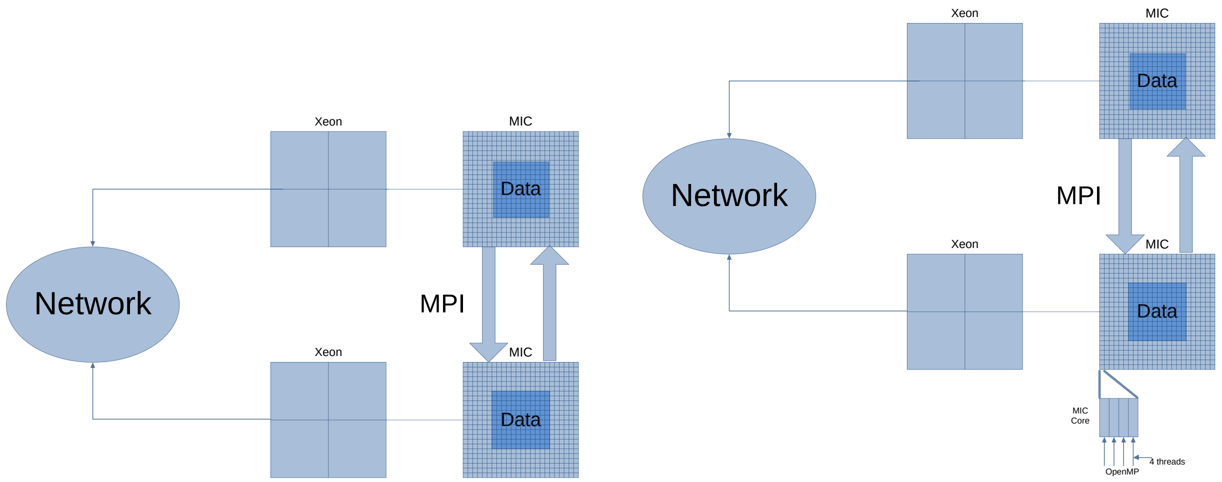
(3) MPI@MIC:

MPI-based parallel implementation on Beacon is shown in Figure 5.2(b). The Intel Xeon Phi 5100P is used for data processing. Totally, there are 60 cores on each MIC card. Each core runs one MPI process. Therefore, 60 MPI processes can be created on each MIC card in the parallel implementation. The parallel MPI code can be directly complied and executed on systems with the Intel Xeon Phi 5100P. Sample code is the same to MPI+CPU programming model as shown in Algorithm 2

(4)MPI@MIC+OpenMP:

MPI-based parallel implementation on Beacon is shown in Figure 5.2(a). Each MIC core on Intel Xeon Phi 5110P can support up to 4 threads in parallel. Each MPI process create 4 threads to run OpenMP program as shown in Algorithm 4.

(5) MPI@CPU+offload:



(a) MPI native parallel implementation.

(b) MPI and OpenMP native parallel implementation.

Figure 5.2: MIC native parallel implementation on Beacon

MPI-based parallel implementation on Beacon. The MPI processes will be running on the CPU, which can offload data to MIC through OpenMP, as the case shown in Figure 5.3. A sample code is shown in Algorithm 5.

Algorithm 4 MPI@MIC+OpenMP Programming model

```
function ASSIGNMENT($Array\ A, Array\ B$)
    $MPI\_\_Init()$
    $MPI\_\_Comm\_\_rank()$
    $MPI\_\_Comm\_\_size()$

    $\#pragma\ omp\ parallel\ for\ omp\_\_set\_\_num\_\_threads(4)$
    for $i = 0 \to k-1$ do
        $A[i] \leftarrow B[i]$
    end for

    $MPI\_\_Finalize()$
end function
```

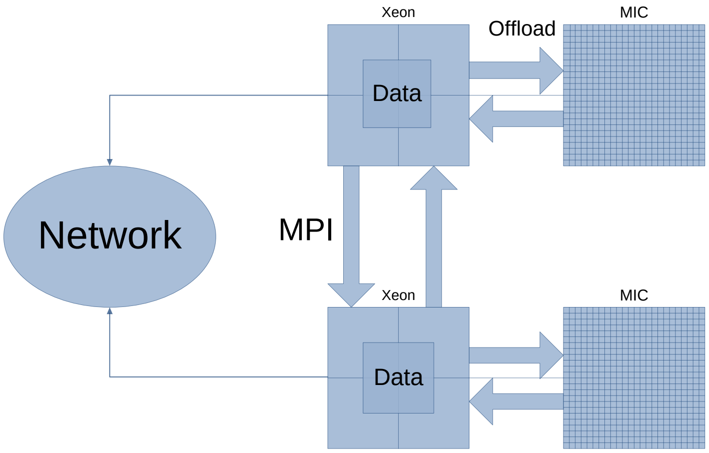


Figure 5.3: Offload parallel implementation on Beacon

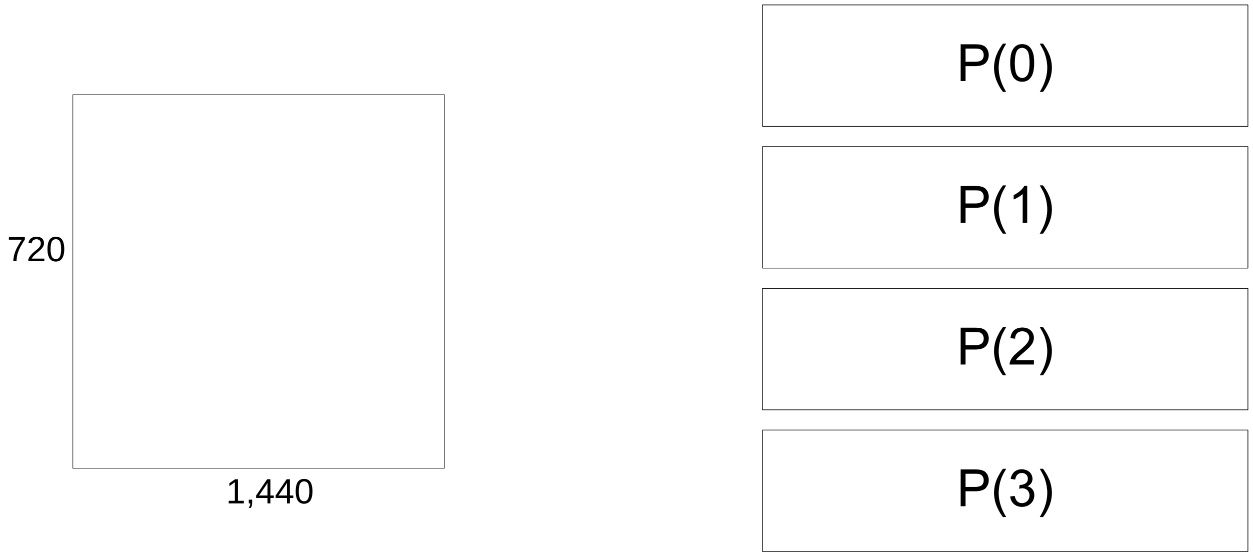
Algorithm 5 MPI@CPU+offload Programming model

```
function ASSIGNMENT($Array\ A, Array\ B$)
    $MPI\_Init()$
    $MPI\_Comm\_rank()$
    $MPI\_Comm\_size()$

    $Memory\ copy\ from\ host\ to\ MIC$
    $\#pragma\ omp\ parallel\ for\ omp\_set\_num\_threads(4)$
    for $i = 0 \rightarrow k - 1$ do
        $A[i] \leftarrow B[i]$
    end for
    $Memory\ copy\ back\ to\ host$

    $MPI\_Finalize()$
end function
```

5.3 Implementation Result

5.3.1 Kriging Interpolation

Kriging Interpolation belongs to embarrassingly parallel case [15]. It is based on an idea that determines the value of an unknown point by using known values as its neighbors. For each input dateset, Kriging Interpolation will generate a $1,440 \times 720$ matrix as a result. This matrix will be evenly partitioned among all processors along the row-major order as shown in Figure 5.4(b). There is no any communication among MPI processes. In order to calculate one item in the output



(a) Result data.

(b) Kriging Interpolation.

Figure 5.4: Data partition and communication on Kriging Interpolation [21].

matrix, the whole dataset will be searched to find the 10 nearest neighbors. Each MPI process only computes its own results and shares the same source data. The input data includes 4 datasets with the respective size of 29 MB, 37 MB, 48 MB and 57 MB. Totally, the size of original dataset is 171 MB and each dataset has 2,191, 4596, 6941, and 9817 sample points, respectively. The values of unsampled location will be calculated by values of the 10 closest sample points. The program will process 4 datasets sequentially and process each datasets in parallelism. On Keeneland cluster, m processors means m CPUs or m GPUs are allocated. On Beacon cluster, each MIC core can be an MPI process. For the output raster grid, 720 columns is evenly distributed among processors. So 360 or 720 processors will be used when 8 or 16 MIC are allocated for the computation.

In order to learn the difference between Kepler architecture and Fermi architecture of GPU, this work compares the performance of single GPU implementation of Kriging interpolation on Kepler K20 and Fermi M2090. The results are shown in Table 5.1. The K20 can achieve a 2.39x speedup without any change for the benchmark. If using specific technology of Kepler architecture, such as dynamic parallelism, the K20 may achieve a better performance. Figure 5.5 demonstrates the performance of 4 different implementations for Kriging Interpolation. The performance of K20 GPU cluster is projected based on the data of single K20 GPU because we have no access to a

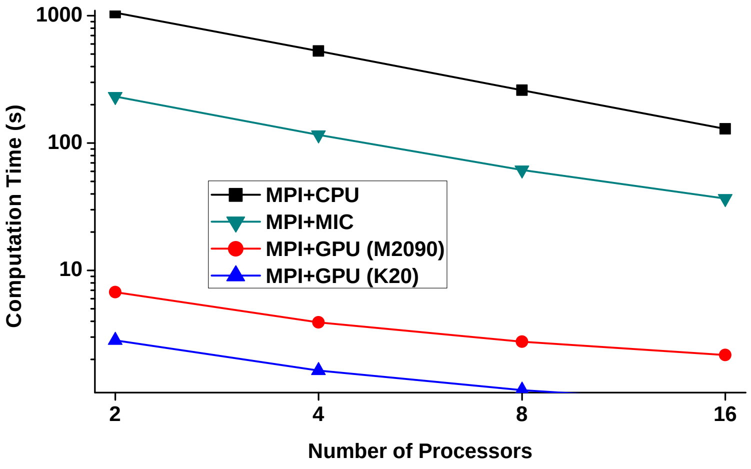


Figure 5.5: Performance of Kriging Interpolation on different configurations [15].

GPU	Kriging Interpolation
M2090	24.64
K20	10.32
Speedup	2.39

Table 5.1: Kriging Performance on M2090 and K20

computer cluster equipped by K20. The input/output are not considered because both times cannot become stable on Keeneland and Beacon cluster for three benchmarks. By looking at the time curves in Figure 5.5, it can be found that the implementations of MPI@CPU and MPI@MIC show a good strong scalability for this application. Although the implementation of MPI+GPU is still able to reduce the computation to half when moving from 2-processor to 4-processor, the performance cannot be better afterwards. Apparently, for this embarrassingly parallel case, there is not much speedup gain when increasing the number of processors on implementation of MPI@GPU. When the grid is partitioned into m processes on GPU processors, performance gain from the reduced parallel workload on each processor cannot become significant part. The speedup of this application is limited by the time needed for the sequential part.

In order to learn the performance and scalability of different programming models on MIC, this work allocates 2, 4, 8, and 16 MIC processors for Kriging Interpolation. The processors of three models mean the numbers of MIC nodes. For example, if m MIC processors are allocated, m MPI processes are created. As mentioned before, for the output raster grid, 720 columns is evenly

Table 5.2: Performance of Kriging Interpolation Based on Beacon Models(unit:second) [14]

Number of	MPI@MIC				MPI@MIC+OpenMP(4 threads)				MPI@CPU+offload			
Processors	Read	Interpolation[*]	Write	**Total**	Read	Interpolation[1]	Write	**Total**	Read	Interpolation[1]	Write	**Total**
2	1.24	232.43	12.24	245.90	0.57	60.43	8.82	69.82	0.18	280.83	1.60	282.61
4	1.27	116.34	16.44	134.05	0.51	36.54	122.53	159.59	0.04	141.03	1.27	142.33
8	1.23	61.48[†]	54.43	117.14	0.50	20.43[2]	240.33	261.26	0.04	74.30	1.19	75.53
16	1.31	36.74[2]	300.23	338.28	0.52	12.33[2]	210.45	223.30	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.

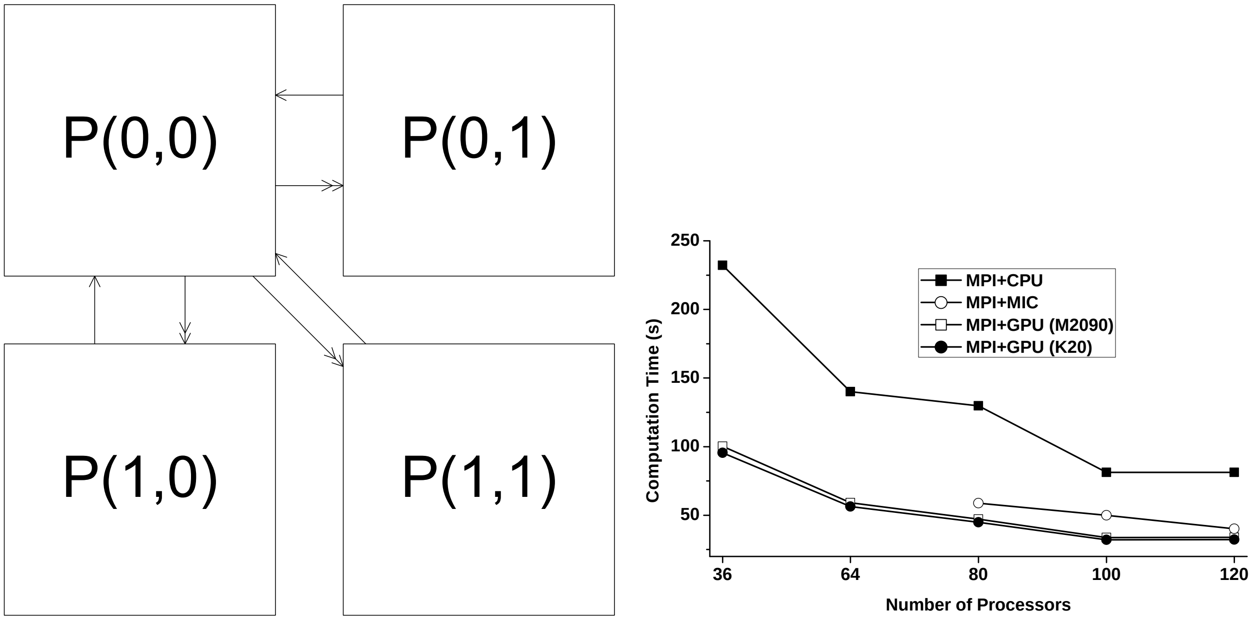
distributed among processors. So 360 or 720 processors will be used when 8 or 16 MIC cards are allocated for the computation. The detailed results of the three programming models for Kriging interpolation are listed in Table 5.2.

When comparing the performance of three programming models, it can be found that the gap between MPI@MIC (native model) and MPI@CPU+offload (offload model) are small. When using the same source of hardware, there is no obvious difference between the two models of MIC for this application. However, when using multithread programming in each MPI process on the native programming model, it can be found that it has a better performance and can be improved by roughly 3 times. Therefore, it is not enough to only parallelize and partition the whole application to all MIC cores. It is still significant to increase parallelism inside each MIC core to further improve the performance.

5.3.2 ISODATA

For ISODATA algorithm, this work only uses MPI@GPU, MPI@CPU and MPI@MIC. It cannot use MPI@MIC+openmp and MPI@MIC+offload since the GDAL library on beacon became not available. The program uses a high-resolution image as input data in ISODATA with a dimension of 80,000x80, 000 for three band [15]. And it defines 15 classes to classify the image. In the parallel implementation, the whole image will be partitioned for parallel computation. We use 4 processes as an example, shown in Figure 5.6(a).

The classification of ISODATA will take many iteration until meet either of two conditions, (1) access the maximum number of iterations (i.e., 15) or (2) the ratio that pixels do not change classes



(a) Data partition and communication on ISODATA.

(b) Performance of ISODATA Interpolation on different configurations.

Figure 5.6: Data partition and Performance on ISODATA [15].

between two consecutive iteration is above the threshold (i.e., 0.95).

During every iteration of classification, each MPI process calculates the local means of each class and then sends the mean value to the main MPI process (i.e., P(0,0)) as shown in Figure 5.6(a). The main MPI process calculates the global means of every class and returns the results to each MPI process. Then next iteration will begin. Performance of ISODATA benchmark is shown in Table 5.3.

When using Beacon to run native programming mode, MIC cannot handle 18 GB image by 36 or 64 processors because the program consumes a lot of memory and leaves not much space for data. Therefore, this work assigns more MIC processors, such as 80, 100 and 120 processors. From the result in Table 5.4, the performance of K20 is better than that of M2090 with a 1.05 speedup. The K20 cannot show the advantages of Kepler architectures because some sequential operations have to run on the GPU with a single thread. In fact, CPU has a good capability to sequential parts. However, it will increase the communicating overhead when transferring data from GPU to CPU frequently. The performance comparison is shown in Figure 5.6(b). By looking at the

Table 5.3: Performance of ISODATA of Three Parallel Implementations (unit:second) [15].

Number of Processors	Keeneland Kids						Beacon		
	MPI+CPU			MPI+GPU			MPI+MIC		
	Read	Computation	Total	Read	Computation	Total	Read	Computation	Total
36	6.04	232.22	238.26	3.91	100.26	104.17	NA*		
64	12.06	140.00	152.59	3.51	59.18	62.69	NA*		
80	15.03	129.72	144.74	21.11	47.12	68.24	41.27	58.75	100.02
100	1.29	81.31	82.59	36.35	33.81	70.16	27.01	50.014	77.02
120	0.98	81.34	82.39	22.29	33.95	56.24	32.32	40.08	72.40

*:The implementation on Beacon cluster cannot be done with 36 and 64 MIC processors because of the limited memory on board.

Table 5.4: ISODATA Performance on M2090 and K20

GPU	ISODATA
M2090	49.19
K20	46.85
Speedup	1.05

*The image size is reduced to 10,000×10,000×3 to fit the on-board memory.

time curves in Figure 5.6(b), it can be found that the performance of MPI@MIC and MIC@GPU are very close. One reason is that FDR InfiniBand network on Beacon cluster has much higher bandwidth than the QDR InfiniBand network on Keeneland KIDS. When the number of processors increases from 100 to 120, the benefits of efficient communication network on Beacon can be further demonstrated. Although the implementation of MPI+GPU is still able to reduce the computation to half when moving from 36 processors to 64 processors, the performance will slow down afterwards. The computation time are almost the same when using 100 processors and 120 processors. The reason is that the performance gain from the reduced workload on each GPU is offset by the increase in the communication time. On Beacon, it still has a strong scalability when using 120 MIC processors in computation.

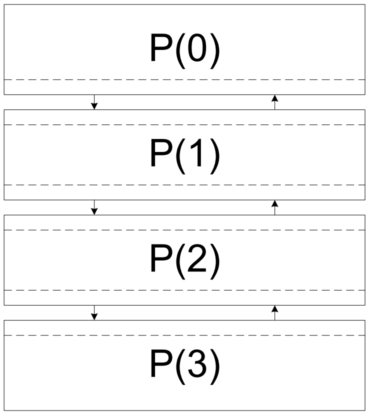


Figure 5.7: Game of Life segmentation [15]

5.3.3 Game of Life

For Game of Life algorithm, I use MPI@GPU, MPI@CPU and MPI@MIC, MPI@MIC+openmp and MPI@MIC+offload architectures.

Game of Life is a generic Cellular Automata program and the status of each cell will be determined upon its eight neighbors. The program sets 100 iteration to update the status of each cell in the grid [15]. The cells are partitioned into stripes along row-wise order for parallelizing the updating process. Statuses of all cells will be updated simultaneously in each iteration. This work uses three datasets as inputs, i.e., 8,192×8,192, 16,384×16,384, and 32,768×32,768. In MPI-based parallel implementation, the boundary rows have to be exchanged before a simulation can be implemented when the two dimension matrix is divided into several segments. At the beginning of each iteration, each MPI will send the data of boundary to its adjacent MPI processes and receive the data from neighbor processes at the same time as shown in Figure 5.7. After each iteration, the total living cells will be aggregated and one of the MPI processors calculates local values into a global value. Therefore, the communication and data transfer among processors will be implemented, which is a typical case of intense communication.

The results demonstrate the strong scalability for MPI implementations on both CPUs and GPUs. The performance of K20 is better than that of M2090 with a 4x speedup based on the results in Table 5.5. Compared with Kriging Interpolation, the performance of Game of Life on

Table 5.5: Game of Life Performance on M2090 and K20

GPU	Game of Life		
	8,192×8,192	16,384×16,384	32,768×32,768
M2090	12.86	51.92	204.99
K20	3.25	12.58	46.99
Speedup	3.96	4.08	4.36

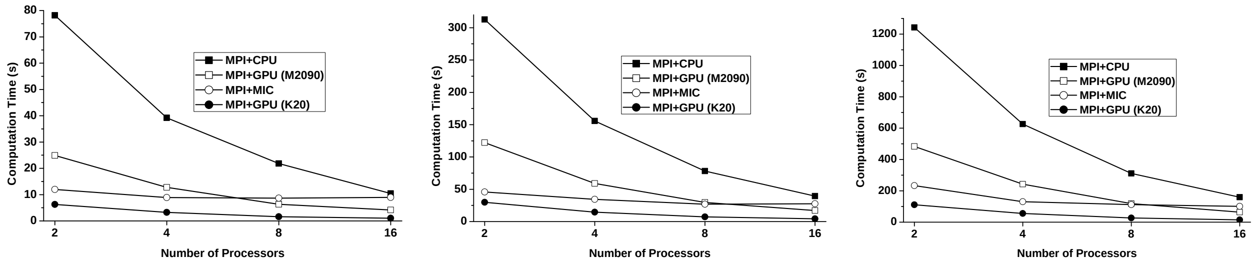
Table 5.6: Performance of Game of Life Under Various Programming Models(unit:second) [15]

Number of Processors	8,192×8,192			16,384×16,384			32,768×32,768		
	MPI+CPU	MPI+GPU	MPI+MIC	MPI+CPU	MPI+GPU	MPI+MIC	MPI+CPU	MPI+GPU	MPI+MIC
2	78.15	24.92	14.56	312.69	122.19	48.39	1242.19	483.58	194.15
4	39.20	12.79	11.63	155.64	59.14	46.31	625.92	242.43	169.54
8	21.82	6.30	7.84	78.14	29.66	39.78	311.34	118.31	157.73
16	10.41	4.15	7.18	39.35	17.20	35.30	159.05	63.83	128.40

Kepler GPU is better than that on Fermi GPU because Game of Life has a smaller fraction of sequential code. As a result, the advantage of Kepler architecture is further demonstrated.

The result performance under various programming models is shown in Table 5.6 and Figure 5.8. By demonstrating the time curves in Figure 5.8, the MPI implementations of CPUs and GPUs have a strong scalability. Besides, K20 can consistently outperform the other accelerators. For the implementation of MPI+MIC, it is shown that the performance cannot scale quite well. The implementation can reduce the computation time when the number of processors increases from 2 to 4. With the increase of source size, computation time can be reduced to half. However, the performance will stop afterwards.

In order to get thorough understanding MIC programming models for intensely communicating case, this work uses three models on Beacon to test Game of Life [14]. The results are shown in Table 5.7. It can be found that the performance is quite different from the performance of Kriging Interpolation. Although the model of MPI@MIC+OpenMP can achieve a good performance, the strong scalability does not hold. In the three programming models, the offload model still keeps ability to reduce the computation time to half from 2 processors to 4 processors, but there is no



(a) 8,192×8,192.

(b) 16,384×16,384.

(c) 32,768×32,768.

Figure 5.8: Performance of Game of Life on four different configurations. [15]

Table 5.7: Performance of Game of Life Based on Beacon Models(unit:second) [14]

Number of Processors	8,192×8,192			16,384×16,384			32,768×32,768		
	MPI@MIC	MPI@MIC+ Openmp(4threads)	MPI@CPU+ offload	MPI@MIC	MPI@MIC+ Openmp(4threads)	MPI@CPU+ offload	MPI@MIC	MPI@MIC+ Openmp(4threads)	MPI@CPU+ offload
2	14.56	7.99	169.12	48.39	33.11	760.20	194.15	149.43	2926.34
4	11.63	8.04	80.50	46.31	24.06	405.66	169.54	104.14	1512.72
8	7.84	9.28	89.03	39.78	22.98	365.23	157.73	106.24	1502.51
16	7.18	8.74	82.51	35.30	23.60	370.65	128.40	110.99	1517.89

consistent reduction and the performance hangs afterwards. As a result, for the intensely communicating application, the performance cannot gain so much when increasing the number of MIC processors. When the source is partitioned into m×60 MPI processes, the increase of communication cost will offset the performance gain from the reduced workload on each MIC core. So it is significant to keep a balance between communication and computation for achieving the best performance.

In order to learn the potential of performance improvement, this work also uses 4 threads on each MIC core for multithreading programming based on native model. These threads can be physically executed in parallelism. This work runs more threads on each MIC core, such as 8 threads and the result is shown in Table 5.8.

It can be known that the gain of using much threads to MIC cores is very marginal. For the small problem size, such as 8,192×8,192, the 8-thread OpenMP implementation does not get a better performance than the 4-thread OpenMP implementation. There will be more cross-thread communication overhead. For the large problem size, it can still get some benefits.

In order to learn the capability and scalability of threads on each MIC card, this work changes

Table 5.8: Performance of Game of Life Using MPI@MIC+OPENMP
Programming Model (Unit:Second) [14].

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

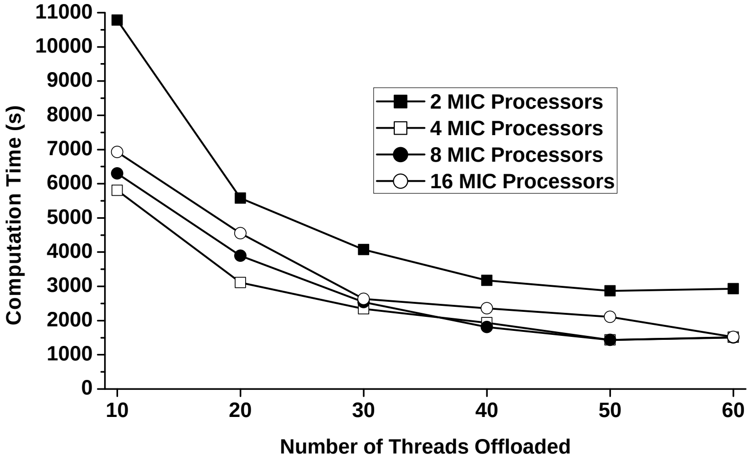


Figure 5.9: Performance of Game of Life (32,768×32,768) using MPI@CPU+offload programming model [14].

the number of threads on each MIC from 10 to 60, as shown in Figure 5.9. By demonstrating the time curves in Figure 5.9, the scalability can hold when increasing the number of threads from 10 to 30 in each case. However, the computation time reduces with a smaller rate from 30 threads to 50 threads. For most cases, the computation time will grow with the increase the number of threads from 50 to 60. This situation can be resulted from the increased inter-thread communication overhead. Therefore, it is not necessary to use all MIC cores when offload programming model is used to each MIC.

Chapter 6

Conclusion

This work shows the potential of accelerators for accelerating applications using parallel implementation on hybrid computer clusters. It also conducts a detailed study about the performance and scalability of the Intel MIC processors under various parallel programming models.

According to different benchmarks, the results demonstrate different performance and scalability under various parallel programming models.

1. For embarrassingly parallel case, GPU-based parallel implementation on Keenland computer cluster has a better performance than other accelerators. Although the implementations of GPU-based parallel implementation can reduce computation time and show a good strong scalability when moving form 2 processors to 4 processors, the performance cannot be further improved afterwards. There is no much speedup gain when increasing the number of processors on these GPU-based parallel implementations. The reason is that the speedup is limited by the time needed for the sequential part. However, MIC-based parallel implementation shows a better scalability than implementation on GPU. For the programming models of MIC on the embarrassingly parallel case, the performances of native model and offload model on MIC are very close.

2. For loosely communicating case, the performances of GPU and MIC are very close because the FDR InfiniBand network on Beacon cluster has much higher bandwidth than the QDR infiniBand betwork ob Keenekand KIDS. When using more processors, the benefits of efficient communication network can be further demonstrated. The MIC-based parallel implementation still carries out a strong scalability when using 120 MIC processors in computation.

3. For the intensely communicating case, the MPI implementations of CPUs and GPUs both have a strong scalability. And the performance of K20 is better than that of M2090 with

a 4× speedup. Compared with the Kriging Interpolation, the performance of Game of life on Kepler GPU is much better. Compared with other accelerators, GPU can consistently achieve the best performance. However, the MIC-based implementation cannot scale quite well. The performance of various models on MIC is different from that of embarrassingly parallel case. Native model can consistently outperform the offload model by 10×. And there is not much performance gain when allocating more MIC processors because the increase of communication cost will offset the performance gain from the reduced workload on each MIC core. When using different number of threads on the intensely communicating case, it shows different scalability on MIC based offload model. The scalability can hold when increasing the number of threads from 10 to 30. The computation time reduces with a smaller rate from 30 threads to 50 threads. When using all threads, the computation time will increase. The reason is that more threads will increase the inter-thread communication overhead. Therefore, it is not necessary to use all MIC cores on offload programming model when applications belong to intensely communicating cases.

The hybrid architectures of MPI@GPU and MPI@MIC can achieve significant improvement compared with the MPI+CPU parallel implementation as well as single CPU implementation. The latest Kepler GPU has a better performance than the Fermi GPU for most applications without special performance tuning. However, the communication of direct cross-GPU can outperform the Intel MIC processors when dealing with intensely communicating applications. For the test between native model and offload model, native model has a better performance than offload model, particulary for the intensely communicating applications. When using multithreading programming on native model, it achieves a better performance than native model. It becomes important to further increase parallelism inside each MPI process for achieving a better performance.

References

[1] http://keeneland.gatech.edu/.

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

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

[4] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience, 23(2):187–198, 2011.

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

[6] Massimo Bernaschi and Francesco Salvadore. Multi-kepler gpu vs. multi-intel mic: A two test case performance study. In High Performance Computing & Simulation (HPCS), 2014 International Conference on, pages 1–8. IEEE, 2014.

[7] NVIDIA Corporation. nvidia fermi compute architecture whitepaper.

[8] Muhammad K Dhodhi, John A Saghri, Imtiaz Ahmad, and Raza Ul-Mustafa. D-isodata: A distributed algorithm for unsupervised classification of remotely sensed data on network of workstations. Journal of Parallel and Distributed Computing, 59(2):280–301, 1999.

[9] Martin Gardner. Mathematical games: The fantastic combinations of john conways new solitaire game life. Scientific American, 223(4):120–123, 1970.

[10] GDAL. Geospatial data abstraction library. http://www.gdal.org.

[11] John R Jensen et al. Introductory digital image processing: a remote sensing perspective. Number Ed. 2. Prentice-Hall Inc., 1996.

[12] 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.

[13] Youngsok Kim, Jaewon Lee, Jae-Eon Jo, and Jangwoo Kim. Gpudmm: A high-performance and memory-oblivious gpu architecture using dynamic memory management. 2014.

[14] Chenggang Lai, Zhijun Hao, Miaoqing Huang, Xuan Shi, and Haihang You. Comparison of parallel programming models on intel mic computer cluster. In Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops, pages 925–932. IEEE Computer Society, 2014.

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

[16] Weiguo Liu and Bertil Schmidt. Parallel pattern-based systems for computational biology: A case study. Parallel and Distributed Systems, IEEE Transactions on, 17(8):750–763, 2006.

[17] Anupam Mahajan, Dishant Sharma, and Kunal Sachdeva. General purpose computing on graphicalprocessing unit: Extending parallel processing.

[18] John Michalakes and Manish Vachharajani. Gpu acceleration of numerical weather prediction. Parallel Processing Letters, 18(04):531–548, 2008.

[19] 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.

[20] 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.

[21] Xuan Shi, Miaoqing Huang, Haihang You, Chenggang Lai, and Zhong Chen. Unsupervised image classification over supercomputers kraken, keeneland and beacon. GIScience & Remote Sensing, 51(3):321–338, 2014.

[22] Xuan Shi, Chenggang Lai, Miaoqing Huang, and Haihang You. Geocomputation over the emerging heterogeneous computing infrastructure. Transactions in GIS, 2014.

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

[24] Peter E Strazdins, Jie Cai, Muhammad Atif, and Joseph Antony. Scientific application performance on hpc, private and public cloud resources: A case study using climate, cardiac model codes and the npb benchmark suite. In Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International, pages 1416–1424. IEEE, 2012.

[25] Yanbing Tang and Clifton W Pannell. A hybrid approach for land use/land cover classification. GIScience & Remote Sensing, 46(4):365–387, 2009.

[26] David E Tenenbaum, Yun Yang, and Weiqi Zhou. A comparison of object-oriented image classification and transect sampling methods for obtaining land cover information from digital orthophotography. GIScience & Remote Sensing, 48(1):112–129, 2011.

[27] Akshay Venkatesh, Sreeram Potluri, Raghunath Rajachandrasekar, Miao Luo, Khaled Hamidouche, and Dhabaleswar K Panda. High performance alltoall and allgather designs for infiniband mic clusters. In Parallel and Distributed Processing Symposium, 2014 IEEE 28th International, pages 637–646. IEEE, 2014.

[28] Keith T Weber and Jackie Langille. Improving classification accuracy assessments with statistical bootstrap resampling techniques. GIScience & Remote Sensing, 44(3):237–250, 2007.