Efficient utilization of multi-core processors and many-core co-processors on supercomputer beacon for scalable geocomputation and geo-simulation over big earth data

Chenggang Lai\textsuperscript{a}, Xuan Shi\textsuperscript{b} and Miaoqing Huang\textsuperscript{a}

\textsuperscript{a}Department of Computer Science and Computer Engineering, University of Arkansas, Fayetteville, AR, USA; \textsuperscript{b}Department of Geosciences, University of Arkansas, Fayetteville, AR, USA

**ABSTRACT**

digital earth science data originated from sensors aboard satellites and platforms such as airplane, UAV, and mobile systems are increasingly available with high spectral, spatial, vertical, and temporal resolution data. When such big earth science data are processed and analyzed via geocomputation solutions, or utilized in geospatial simulation or modeling, considerable computing power and resources are necessary to complete the tasks. While classic computer clusters equipped by central processing units (CPUs) and the new computing resources of graphics processing units (GPUs) have been deployed in handling big earth data, coprocessors based on the Intel's Many Integrated Core (MIC) Architecture are emerging and adopted in many high-performance computer clusters. This paper introduces how to efficiently utilize Intel's Xeon Phi multicore processors and MIC coprocessors for scalable geocomputation and geo-simulation by implementing two algorithms, Maximum Likelihood Classification (MLC) and Cellular Automata (CA), on supercomputer Beacon, a cluster of MICs. Four different programming models are examined, including (1) the native model, (2) the offload model, (3) the symmetric model, and (4) the hybrid-offload model. It can be concluded that while different kinds of parallel programming models can enable big data handling efficiently, the hybrid-offload model can achieve the best performance and scalability. These different programming models can be applied and extended to other types of geocomputation to handle big earth data.

1. **Introduction**

The emerging advanced sensors have generated high spectral, spatial, vertical, and temporal resolution data, such as IKONOS, QuickBird, and WorldView-2,3, as well as UAVs and mobile systems, which can provide sub-meter pixel image products (Boschetti, Boschetti, Oliveri, Casati, & Canova, 2007; Buddenbaum, Schlerf, & Hill, 2005; Clark, Roberts, & Clark, 2005; Greenberg, Dobrowski, & Vanderbilt, 2009; Martin, Newman, Aber, & Congalton, 1998;...
Sridharan & Qiu, 2013; Thenkabail, Enclona, Ashton, Legg, & De Dieu, 2004; Xiao, Ustin, & McPherson, 2004). Light Detection and Ranging (LiDAR) sensor can offer high-vertical resolution of geometry and allows the direct collection of \(x\), \(y\), and \(z\) coordinates of ground objects, which makes possible automatic detection of elevated features and construction of three-dimensional (3D) models of ground surface (Rottensteiner, Trinder, Clode, & Kubik, 2005). The temporal resolution is increased because of the shortening of revisit frequency, or the simultaneous availability of multiple sensors. Inevitably, the volume, velocity, and variety of big earth data pose great challenge to existing geospatial software when analyzing the big data because the scale of data and computation are well beyond the capacity of PC-based software due to PC’s limited storage, memory, and computing power. New computing infrastructure and system are required in response to such big data challenge.

Among different computer architectures and advanced computing technologies, Intel’s Many Integrated Core (MIC) Architecture provides a promising solution to employ massive parallelism to handle big data with better performance, scalability and efficiency (Duran & Klemm, 2012; Huang, Lai, Shi, Hao, & You, 2015; Jeffers & Reinders, 2013; Kandalla et al., 2013; Saule & Çatalyürek, 2012; Spentzouris, Amundson, & Macridin, 2014). For example, supercomputer Beacon contains 192 MIC-based Intel Xeon Phi 5110P coprocessors. The MIC processors typically co-exist with multicore CPUs, such as Intel Xeon E5, in a hybrid computer node as coprocessors/accelerators. The constituent processing core on a MIC card will be called a MIC core. These MIC cores are connected through a high-speed ring bus. Each core can run four threads in parallel. Because each core alone is a classic processor, traditional parallel programming models, such as MPI and OpenMP, are supported. For this reason, many conventional MPI solutions can be efficiently applied on MIC systems without the need of significant transformation like the CUDA solutions applicable on NVIDIA GPU clusters.

MIC has been mainly utilized on supercomputers, while individual researchers may not be able to afford such expensive hardware and the necessary compilers. In our prior work (Shi, Lai, Huang, & You, 2014), it was summarized that embarrassingly parallelism could be the simplest way in parallel geocomputation. In more complicated situation, intensive data exchange and communication have to be handled appropriately in the parallel and distributed computing environment. When multiple iterations are involved, synchronization has to be achieved in such parallel computing environment. In recent years, we conduct some pilot studies to understand how to utilize MIC clusters to conduct geocomputation over big data-sets (Shi et al., 2014; Shi, Huang, You, Lai, & Chen, 2014). These prior works mainly applied the native or offload model to utilize MICs on Beacon. This paper will introduce four different programming models for the utilization of cluster of MICs to conduct different kinds of geocomputation. One is to implement Maximum Likelihood Classification (MLC) algorithm to process big imagery data for supervised classification. The other is to implement cellular automata (CA)-based simulation as CA is a very popular solution in geo-simulation. MLC is a typical case of embarrassingly parallelism, while CA has intensive data exchange and communication issues involved in the distributed and parallel computing environment.

MLC and CA are implemented to understand the deployment of MIC processors by different approaches. In the native model, the MPI process is directly run on each MIC core. In the symmetric model, we try to take advantage of the host CPU computing capability. Therefore, we launch MPI processes both on host CPU and Xeon Phi coprocessors. In the offload model, only one MPI process is issued onto each MIC processor. Then OpenMP is
used to launch threads to MIC cores. In the hybrid-offload model, the MPI processes are run on the CPUs. The part of data processing is offloaded to the MIC processors using OpenMP. On the host CPU, we also use OpenMP to spawn multiple threads for parallel processing.

Through this pilot study, it can be concluded that (1) the native MPI programming model on the MIC processors is typically better than the offload programming model, which offloads the workload to MIC cores using OpenMP; (2) the hybrid-offload model can achieve about 1.7x performance speedup in comparison to the offload programming models; (3) The symmetric model only achieves about 1.3x performance speedup in comparison the native model because there is a load-balanced problem.

In the remaining sections of this paper, we first review the algorithms of MLC and CA to be implemented in this pilot study. We will then introduce the four different kinds of programming models applicable on MIC and how each model is used to implement MLC and CA. The result of this pilot study will be summarized followed by a brief conclusion. The solutions explored in this research can be extended to other types of geocomputation to handle big earth data.

2. Overview of the algorithms for geospatial computation and simulation

While many geocomputation can be done by embarrassingly parallel solutions over computer clusters, in this paper, we apply Maximum Likelihood Classification (MLC) for supervised image classification to process big imagery data. While implementing Cellular Automata (CA) for simulation needs to handle intensive data exchange and communication in distributed and parallel computing environment, it could be a perfect benchmark to explore and validate different programming models applicable on MICs. In this pilot study, Game of Life, a famous CA model, is implemented on supercomputer Beacon by four types of programming models applicable on cluster of MLCs.

2.1. MLC for supervised image classification

Maximum Likelihood Classification (MLC) is a supervised image classification method based on Bayes theorem. The algorithm was discussed in details in a prior work (Shi & Xue, 2017). In summary, MLC measures the probability that a pixel with feature vector \( x \) belongs to a class and assigns the pixel to the most possible class. The following equation (Richards & Jia, 1999) calculates the probability \( p(\omega_i|\omega) \) that pixel \( x \) belongs to class \( \omega_i \).

\[
p(\omega_i|x) = \frac{p(x|\omega) \cdot p(\omega_i)}{p(x)}
\]

where \( x \) represents a pixel, if the image contains multiple bands, then \( x \) would be a vector. \( p(\omega_i) \) is the probability that class \( \omega_i \) occurs in the image. It is estimated from the training data, be arbitrarily assigned, or assumed to be equal across all classes. It is also possible to specify a probability threshold below which the pixel won’t be assigned to any classes. \( p(x|\omega_i) \), the probability of observing \( x \) in class \( \omega_i \) can be derived from the training data. Thus,

\[
x \in \omega_i \text{ if } p(x|\omega) \cdot p(\omega) > p(x|\omega_j) \cdot p(\omega), \text{ for all } j \neq i
\]
MLC assumes that the distribution of pixel digital values obeys multivariate normal distribution, that is, 
\[ p(\mathbf{x} | \omega) \sim N(\mu_i, \Sigma_i) \]
which leads to maximum likelihood estimation of 
\[ p(\omega | x) = 2\pi^{-N/2} |\Sigma_i|^{-1/2} e^{-\frac{1}{2}(x-m_i)^T \Sigma_i^{-1} (x-m_i)} \]

Taking log on both sides, the above equation is simplified as,

\[ g_i(x) = \ln p(\omega_i) - \frac{1}{2} \ln |\Sigma_i| - \frac{1}{2} (x - m_i)^T \Sigma_i^{-1} (x - m_i) \]

Equation (1) is referred to as the determinant function as it determines the class label of pixel \( x \). \( C_i \) and \( m_i \) are sample estimates of covariance and mean in class \( \omega_i \). Although MLC is a linear complexity algorithm, the time cost for processing a large remote sensing image could be prohibitive. For example, the sample data used in this study are a series of 3-band aerial photos that have a spatial resolution of 0.5 m. Each tile of the image has a total of \( 80,000 \times 80,000 \times 3 = 19,200,000,000 \) pixels.

The following pseudo code (Shi & Xue, 2017) illustrates the implementation of MLC by serial C program. Within this code, \( M \) is the class mean values derived from the training data; \( C \) is the class covariance derived from the training data; \( G \) is the input remote sensing image; \( D \) refers to training data; \( N \) is the number of pixels in \( G \); \( L \) is a vector of class labels; \( P \) is a vector of class priori probabilities; \( O \) is the output classification map.

**Algorithm 1. Maximum likelihood Classification**

```plaintext
1: M = ComputeClassMean(G, D)
2: C = ComputeClassCovariance(G, D)
3: for i in 0 to N-1
4:     dMax = -Inf //dMax is a variable and initialized to a minimum value
5:     y = -1
6:     for j in 0 to L.size - 1
7:         d = Determinant(P[j], M[j], C[j], G[i]) //based on Equation 1
8:         if (d > dMax)
9:             d = dMax
10:            y = j
11:     O[i] = L[y]
```

This work computes class covariance and class mean, respectively, and iteratively computes the class label for every pixel in the input image. Within the iteration, it computes the determinant value for each class and determines the class label for a pixel. ESRI’s ArcMap software was used to define the training data, generate sample covariance and mean values, validate the quality of both serial C program and parallelized solutions for MLC, and serve as the base for performance comparison. Since the time used on processing training data is not significant, we only implement MLC for the classification task, which is time consuming over big imagery data.
2.2. Game of life as a cellular automata-based simulation

Cellular Automata (CA) has been a classic method widely used in geospatial simulation and modeling. Game of Life (GOL) (Gardner, 1970), invented by British mathematician John Conway, is a well-known classical CA model. According to the transition rules, a cell can live or die based on the condition of its $3 \times 3$ neighborhood. To implement the simulation, the status of each cell in the grid will be updated for 100 iterations. In each iteration, the status of all cells is updated accordingly. The pseudocode is illustrated in Algorithm 2.

Algorithm 2. Game of Life

1: function TRANSITION(Cell, t)
2:     n = number of alive neighbors of cell at time t
3:     if cell is alive at time t then
4:         if n > 3 then
5:             Cell dies of overcrowding at time t+1
6:         end if
7:         if n < 2 then
8:             Cell dies of under-population at time t+1
9:         end if
10:        if n = 2 or n = 3 then
11:           Cell survives at time t+1
12:       end if
13:     end if
14:     if n=3 and Cell is dead at time t then
15:        Cell becomes a live cell by reproduction at time t+1
16:    end if
17: end function

When the size of the grid is big, CA is time-consuming, as evidenced in our prior work on a CA-based 50-year urban sprawl simulation (Guan, Shi, Huang, & Lai, 2016). In order to parallelize the CA calculation process, the grid has to be partitioned into stripes by the row-wise order. Each stripe is handled by one processor. Since the status of each cell is dependent upon the status of its neighboring cells, before the calculation, the data at the boundary rows have to be exchanged between different processors. After the CA calculation, a statistical calculation has to be conducted to summarize the status of all cells. For this reason, intensive data exchange and communication between distributed processors have to be handled appropriately in CA simulation.
3. Implementation of MLC and CA by four programming models on supercomputer Beacon

We conducted our experiments on supercomputer Beacon (2017). The Beacon system (Cray CS300-AC Cluster Supercomputer) offers access to 48 compute nodes and 6 I/O nodes joined by a FDR InfiniBand interconnect providing 56 Gb/s of bi-directional bandwidth. Each compute node is equipped with two Intel Xeon E5-2670 8-core 2.6 GHz processors, four Intel Xeon Phi (MIC) coprocessors 5110P, 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 coprocessor contains 60 1.053 GHz MIC cores and 8 GB GDDR5 on-board memory. As a result, Beacon provides 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.
Figure 1 displays the Many Integrated Core (MIC) architecture of Intel's Xeon Phi coprocessor. Xeon Phi contains multiple 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 up to 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 support communication between the processors. Direct communication between MIC processors across different nodes is also supported through MPI.

Figure 2 displays two conventional and generic approaches for parallel computing on MIC (Xeon Phi) coprocessors. The first approach is to treat the MIC coprocessors as clients to the host Xeon CPUs. As shown in Figure 2(a), the MPI processes will be hosted by Xeon CPUs, which will offload the computation to the MIC coprocessors. Multithreading programming models such as OpenMP can be used to allocate many cores on MIC for data processing. The second approach, as shown in Figure 2(b), is to let each core on MIC coprocessors directly host one MPI process. In this way, the 60 cores on the same die are treated as 60 independent processors while sharing the 8 GB on-board memory on the Xeon Phi 5110P.

Both offload and native models, however, only utilize the MIC coprocessors for computation, while the computing resources on Xeon CPUs are not used efficiently. For this reason, two more programming models are explored in this pilot study to efficiently utilize both the MIC coprocessors and Xeon CPUs through the hybrid deployment of both resources. Consequently four different programming models are applied in this pilot study, including the native model, the hybrid-native model (i.e. the symmetric model), the offload model, and the hybrid-offload model. Each programming model is applied to implement both MLC and CA on Beacon.

3.1. The native model (MPI-based implementation)

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 execution directly on the coprocessors.

Supercomputer Beacon provides a heterogeneous environment, including host Xeon CPUs and Xeon Phi coprocessors. Applications that are already implemented by MPI can use this model by distributing MPI ranks across the coprocessors natively. In the native model, as shown in Figure 2(b), 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, \(m \times n \times 60\) MPI processes are created in the parallel implementation. However, no job is dispatched on the host Xeon CPU.
3.1.1. **MLC by native model**

MLC is pixel-based calculation, it is a typical embarrassingly parallelism for implementation on computer clusters. The imagery data used in this study are a set of aerial photos with a high-spatial resolution of 0.5 meter captured in 2007 covering the metropolitan area of Atlanta. The file size of each tile is about 18 GB when stored in ERDASE Imagine IMG format covering 1,600 km².

Since individual MIC has limited memory, big imagery data have to be partitioned appropriately to efficiently utilize MIC resources. For this reason, each tile of the input data is divided into multiple data segments. Assume each segment has M pixels, when N processors are used, each MPI process will handle M/N pixels. The following pseudo code illustrates the MPI implementation of MLC for each segment. One segment of the data is partitioned into parts in line 5–9. $myNCell$ is the number of pixels that each MPI rank needs to process. line 11–19 shows how do calculation of classification. In the native model, the MPI process is directly executed on each MIC core. Each MIC launches a total of 240 MPI processes to do the MLC calculation.

**Algorithm 3. MLC by native model**

```plaintext
1: M = ComputeClassMean(G, D)
2: C = ComputeClassCovariance(G, D)
3: MPI_Comm_size(MPI_COMM_WORLD, &np);
4: MPI_Comm_rank(MPI_COMM_WORLD, &rank);
5: myNRow = totalRow / np
6: myBRow = myNRow * rank + 1
7: if (rank == np-1 || totalRow % np > 0)
8:     myNRow += totalRow % np
9: myNCell = myNRow * rasWidth
10: ReadDATA();
11: for i in 0 to myNCell -1
12: dMax = -Inf //dMax is a variable and initialized to a minimum value
13: y = -1
14: for j in 0 to L.size - 1
15: d = Determinant(P[j], M[j], C[j], G[i]) //based on Equation 1
16: if (d > dMax)
17: d = dMax
18: y = j
19: O[i] = L[y]
20: MPI_Finalize()
```
3.1.2. GOL by native model

In the native model, CA runs entirely on an Intel Xeon Phi coprocessor. Each MIC core directly hosts up to four single-thread MPI processes. In order to parallelize CA on MIC, the CA grid is partitioned into stripes by the row-wise order. Each stripe is handled by one MPI process. The following pseudo code illustrates the implementation of GOL by the native model. In this case, the cells in the $n \times n$ matrix grid are partitioned into 240 stripes in the row-wise order, as shown in line 3–5. Each stripe is handled by one MPI process. At the beginning of each iteration, the status of the cells along the boundaries of each stripe has to be exchanged with its neighboring processors through the MPI send and receive command in line 8–13. In the other word, each MPI process needs to send the statuses of the cells along the boundaries of each stripe to its neighboring MPI processes and receive the statuses of the cells of two adjacent rows. Consequently, each MIC runs 240 MPI to process 240 stripes. The next status of each cell would be determined by TRANSITION function, as illustrated in Algorithm 2.

Algorithm 4. GOL by native model

1: MPI_Comm_size(MPI_COMM_WORLD, &np);
2: MPI_Comm_rank(MPI_COMM_WORLD, &rank);
3: myNRow = totalRow / np;
4: if (rank == np-1 \&\& rank > 0)
5:     myNRow = totalRow \% np;
6:   initial(Grid)
7: for t in 0 to iteration
8:     if (rank \%2 ==1)
9:         MPI_Send tail and MPI_Recv head
10:        MPI_Send head and MPI_Recv tail
11:     else
12:        MPI_Recv head and MPI_Send tail
13:        MPI_Recv tail and MPI_Send head
14:     for i in 0 to Grid.size
15:        cell = Grid[t][i]
16:        Grid[t+1][i] = TRANSITION(cell, t)
17:     achieve live cell at iteration t
18: MPI_Finalize()
3.2. The Symmetric (Hybrid-Native) Model

Native model only uses the Xeon Phi coprocessors to run MPI programs, while the Xeon CPU processors are not utilized. This means the native model does not take advantage of the computing capacity of the host Xeon CPU, which is much more powerful than the cores on the MIC coprocessor. The symmetric model extends the native model by utilizing both Xeon CPU and MIC coprocessors, thus we call it the hybrid-native model. In this model, the MPI ranks reside on the host CPU and the MIC coprocessors. If \( m \) MIC coprocessors in which each MIC core directly hosts \( n \) (up to 4) MPI processes and \( k \) host CPU processors are used, \( m \times n \times 60 + k \) MPI processes are created in the parallel implementation.

The MPI standard was designed from the ground up to support a single program running on a heterogeneous set of nodes. Without any source code modification, an MPI program can be executed on both the host Xeon CPU and Xeon Phi coprocessors. However, two executable files have to be created for host CPU and MIC coprocessors. To build and run an application in symmetric mode, the following commands have to be executed to compile the source code into executables to be used on supercomputer Beacon:

```
# compile the program for MIC coprocessor (-mmic)
mpiicc -mmic -o test.MIC test.c
```

```
# compile the program for the host Xeon CPU
mpiicc -o test.host test.c
```

A running script is set up to tell computer which executable file is running on host or coprocessors. The total number of MPI processes running on the host or coprocessors can be determined in this script. Besides, the number of MPI processes running on each Xeon Phi coprocessor also can be determined because MIC has its own operation system and IP address. According to the configuration of supercomputer Beason, one node may contain multiple host CPUs and Xeon Phi coprocessors. Developer can adjust the number of MPI ranks running on the host CPU and the MIC coprocessors in the execution command. In the following sample commands, \( N \) represents the number of MPI ranks, the \# in beacon# means the node on Beacon, mic0 or mic1 specifies the specific MIC to be used.

```
#launch MPI jobs on the host and on the coprocessor mic0 and mic1
mpirun -host beacon# -n N ./test : -n N -host beacon#-mic0 /tmp/test.MIC : -n N -host beacon#-mic1 /tmp/test.MIC
```

The symmetric implementations of GOL and MLC are the same as the native implementations, as shown in Algorithm 3 and Algorithm 4. The MPI processes are executed on both MIC cores and host CPU cores. In this pilot study, each Xeon Phi coprocessor runs 240 MPI and host CPU runs 8 MPI to process GOL and MLC calculations. The sample commands of MLC on multiple MICs are shown as follows:
In these examples, \texttt{beacon001} represents node 1 and \texttt{beacon001-mic0} is the first MIC on node 1. \texttt{beacon002} represents node 2 and \texttt{beacon002-mic0} is the first MIC on node 2. The host CPU runs 8 MPI ranks, and each MIC runs 240 MPI ranks.

Conventionally, many MPI programs were written with the implicit assumption they will run on homogeneous systems in which each MPI rank computes at the same rate. These programs decompose the problem into evenly parts to compute so that all MPI processes can synchronize at some points without waiting a long time. However, since the host CPU cores have much more powerful computing capability than the MIC cores, in general, the host CPU cores have to wait at the synchronize point for the slowest MIC cores. That is a typical load imbalance problem in such heterogeneous environments but cannot be resolved by MPI itself. In some other cases, load imbalance problems could be solved by irregular/uneven decomposition of the data (e.g. spatial domain), or dynamic load-balancing methods such as task-farming, on traditional grid system or cluster of GPUs. On Beacon, however, there is no way to detect which processor is CPU core, or which processor is MIC core. When every processor is treated equally, the load imbalance problem is caused by the different capacity and performance of Xeon CPU cores and MIC cores, but is not caused by data partitioning.

### 3.3. The offload model (MPI on CPU, OpenMP on MIC)

The offload model, as shown in Figure 2(a), 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 \texttt{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 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.

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.

```c
#pragma offload target(mic) in(var) out(Output1: length(Output1.size) inout(Output2: length(Output2.size))
```

3.3.1. MLC by offload model

To implement MLC by the offload model, it starts from the host CPU. As shown in Algorithm 5, the input data are divided into multiple data segments. The MPI processes are allocated on the CPU cores. For each segment, given N processors and M pixels to process, each MPI process will handle M/N pixels, while the data and computation are dispatched to the MIC coprocessors. Each MPI process launches 240 threads to the MIC that further uses OpenMP to handle data and calculation. The data movement between the host processor and MIC coprocessor is defined in line 11. In this case, the *in* specifier copies variable myNCell and array P, M, C, G, L (defined in Algorithm 1) to the coprocessor. The *inout* specifier both copy array O (defined in Algorithm 1) from the host processor to the coprocessor and back from the coprocessor to the host processor. Line 12 specifies the number of threads used by MIC and divides the *for-loops* into multiple threads in line 13. *pragma omp parallel* defines a parallel region where all threads specified in line 12 will execute the following *for-loops*.
Algorithm 5. MLC by offload model

1: M = ComputeClassMean(G, D)
2: C = ComputeClassCovariance(G, D)
3: MPI_Comm_size(MPI_COMM_WORLD, &np);
4: MPI_Comm_rank(MPI_COMM_WORLD, &rank);
5: myNRow = totalRow / np;
6: myBRow = myNRow * rank + 1;
7: if (rank == np-1 && totalRow % np > 0)
8:     myNRow += totalRow % np;
9: myNCell = myNRow * rasWidth
10: ReadData();
11: #pragma offload target(mic:0) in(myNCell) in (P: length(L.size))
    in (M: length(L.size)) in (C: length(L.size)) in (G: length(myNCell))
    in (L: length(L.size)) inout(O: length(myNCell))
12: omp_set_num_threads(240)
13: #pragma omp parallel for
14: for i in 0 to myNCell -1
15:     dMax = -Inf //dMax is a variable and initialized to a minimum value
16:     y = -1
17:     for j in 0 to L.size - 1
18:         d = Determinant(P[j], M[j], C[j], G[i]) //based on Equation 1
19:         if (d > dMax)
20:             d = dMax
21:             y = j
22:     O[i] = L[y]
20: MPI_Finalize()

3.3.2. GOL by offload model

Similarly, to implement GOL by the offload model, the main procedure starts on the host platform. The MPI processes are allocated on the CPU cores, while the data and computation are dispatched to the MIC coprocessors. At the beginning of each iteration, the status of the cells along the boundaries of each stripe has to be exchanged with its neighbor through the MPI send and receive command run on the host CPU. The MPI process specifies the number of threads to the MIC that uses OpenMP to handle all data and calculation. In the GOL implementation, each MIC runs 240 threads to update the status of cells that are dependent upon their 8 neighbors, as shown in Algorithm 6.
Algorithm 6. GOL by offload model
1: MPI_Comm_size(MPI_COMM_WORLD, &np);
2: MPI_Comm_rank(MPI_COMM_WORLD, &rank);
3: myNRow = totalRow / np;
4: if (rank == np-1 & &  % np > 0)
5:     myNRow += totalRow % np;
6: initial(Grid)
7: for t in 0 to iteration
8:     if (rank %2 ==1)
9:         MPI_Send tail and MPI_Recv head
10:        MPI_Send head and MPI_Recv tail
11:     else
12:         MPI_Recv head and MPI_Send tail
13:         MPI_Recv tail and MPI_Send head
14:     n=rank%4;
15:     #pragma offload target(mic:n) in(Grid[t]: length(Grid.size ))
inout(Grid[t+1]: length(Grid.size ))
16:     omp_set_num_threads(240);
17:     #pragma omp parallel for
18:     for i in 0 to Grid.size
19:         cell = Grid[t][i]
20:         Grid[t+1][i] = TRANSITION(cell, t)
21:     achieve live cell at iteration t
22: MPI_Finalize()

In the line 14, target(mic:n) can be used to explicitly indicate which coprocessor should be
used. On supercomputer Beacon, n is represented from 0 to 3. The program first sets mic n
as a target, and then copy input data Grid[t] into MIC using in specifier. Output data Grid[t +1] would be copied back from the coprocessor to the host processor when program running
on MIC is completed. In the line 15, omp_set_num_threads() specifies the number of threads
used by MIC. #pragma omp parallel for in line 16 divides the for-loops into multiple threads,
which are run simultaneously.

When multiple MICs are utilized, MPI rank is used to determine which MIC would be used.
In this study, each node on supercomputer Beacon runs four MPI processes and each process
uses the corresponding MIC by the command: n = rank % 4.

3.4. The hybrid-offload model

As discussed in the above section, the offload model provides a mechanism that transfers
data from the host Xeon CPU to the Xeon Phi coprocessors, performs the computation on
the coprocessor, and returns the results back to the host. In this case, the computing power
of the Xeon CPU is not used after the job is dispatched onto the MIC. The hybrid-offload
model provides an asynchronous mechanism to efficiently utilize the computing power of both Xeon CPU and MIC coprocessors. The workload is first distributed to host CPUs through MPI and then a host CPU will offload a part of the job to MIC coprocessor using OpenMP. On the host CPU, OpenMP is used to spawn multiple threads for parallel processing. Asynchronous offload allows overlap of data transfer and compute. The host initiates an offload to be performed asynchronously and can proceed to next statement after starting this computation. Consequently, an offload_wait pragma is used to wait for completion of the offload activity.

The signal and wait clauses are used within pragmas to asynchronous data transfer and computation. An offload pragma with a signal clause begins an asynchronous computation on the MIC coprocessors. The host begins the offload execution on the coprocessor while computation on the host continues in parallel. An offload_wait pragma with a wait clause blocks execution. The host arrives at a pragma statement written with a wait clause and blocks until an asynchronous data computation on the coprocessor is complete. The host initializes an offload to be performed asynchronously, as shown in following sample commands, and can proceed to the next statement after starting this computation process. Later in the code, an offload_wait pragma is used to wait for completion of the offload activity running on both host CPU and MIC coprocessors. The wait and signal clauses are associated with each other through a unique value. In the following commands, the wait clause is waiting for activity signaled by signal_value to be completed.

```c
// Decide which MIC (e.g. mic0, mic1, mic2 and mic3) to be used
n = rank % 4.

// Initiate asynchronous computation
#pragma offload target(mic:n) signal(signal_value)
{
    Computation on MIC;
}

Host statement;

// wait for completion of the offload activity
#pragma offload_wait target(mic:n) wait(signal_value)
```

### 3.4.1. MLC by Hybrid-offload model

To implement MLC by the hybrid-offload model, the input data are divided into multiple data segments. When running at the full capacity, the performance of each Intel Xeon Phi 5110P is equivalent to one core of Xeon E5–2670. Therefore, MIC runs 240 threads to do the calculation and it processes a half data. Each MPI process running on the host CPU specifies eight threads that use OpenMP handle the rest of data and calculation.

The following pseudo code illustrates the implementation of MLC by the hybrid-offload model. The signal and wait clauses are used to enable asynchronous data transfer and computation in line 11 through 31. The host CPU continues to proceed to next statement (line 21) after MIC coprocessors start the kernel computation (line 11), which means the host CPU processors and MIC coprocessor do the MLC calculation (lines 23–30 and 11–22) simultaneously.
Algorithm 7. MLC by Hybrid-offload model

1: M = ComputeClassMean(G,D)
2: C = ComputeClassCovariance(G,D)
3: MPI_Comm_size(MPI_COMM_WORLD, &np);
4: MPI_Comm_rank(MPI_COMM_WORLD, &rank);
5: myNRow = totalRow / np;
6: myBRow = myNRow * rank + 1;
7: if (rank == np-1 || totalRow % np > 0)
8: myNRow += totalRow % np;
9: myNCell = myNRow * rasWidth
10: ReadData();
11: n=rank%4;
12: #pragma offload target(mic:n) in(myNCell) in (P: length(L.size)) in (M: length(L.size)) in (C: length(L.size)) in (G: length(myNCell/2)) in (L: length(L.size)) inout(O: length(myNCell/2)) signal(0)
13: omp_set_num_threads(240);
14: #pragma omp parallel for
15: for i in 0 to myNCell/2
16: dMax = -Inf //dMax is a variable and initialized to a minimum value
17: y = -1
18: for j in 0 to L.size - 1
19: d = Determinant(P[j], M[j], C[j], G[i]) //based on Equation
20: if (d > dMax)
21: d = dMax
22: y = j
23: O[i] = L[y]
24: #pragma omp parallel for num_threads(8)
25: for i in myNCell/2 to myNCell-1
26: dMax = -Inf //dMax is a variable and initialized to a minimum value
27: y = -1
28: for j in 0 to L.size - 1
29: d = Determinant(P[j], M[j], C[j], G[i]) //based on Equation
30: if (d > dMax)
31: d = dMax
32: y = j
33: O[i] = L[y]
34: #pragma offload_wait target(mic:0) wait(0)
35: MPI_Finalize()
3.4.2. GOL by hybrid-offload model

To implement GOL by the hybrid-offload model, the MPI process is executed on each host CPU and offloads a part of data to Xeon Phi coprocessor that uses OpenMP to do calculation. The host CPU continues to proceed to next statement after the MIC coprocessor starts this computation. In this case, each MIC still runs 240 threads to do the calculation but Xeon Phi coprocessors only process a half data. Each MPI process running on host CPU specifies eight threads that use OpenMP handle the rest of data and calculation. When running at the full capacity, the performance of each Intel Xeon Phi 5110P is equivalent to one core of Xeon E5-2670. Therefore, the workload is evenly distributed to the host CPU and Xeon Phi on each Beacon node.

The following pseudo code illustrates the hybrid-offload implementation of GOL. The `signal` and `wait` clauses are used to asynchronous data transfer and computation in line 14 and 24. Offload `pragma` called with a `signal` in line 14 begins an asynchronous computation on the MIC coprocessors. The output `Grid[t + 1]` is used to associate with `signal` and `wait` clauses. Lines 14–19 show that the host CPU begins the offload execution on the MIC coprocessor. Concurrently, the computation on the host CPU continues in parallel in lines 20–23. The host CPU arrives at line 24 written with a `wait` clause and waits for completion of the offload computation on MIC coprocessor.

**Algorithm 8. GOL by Hybrid-offload model**

```c
1: MPI_Comm_size(MPI_COMM_WORLD, &np);
2: MPI_Comm_rank(MPI_COMM_WORLD, &rank);
3: myNRow = totalRow / np;
4: if (rank == np-1 && np > 0)
5:     myNRow = totalRow % np;
6: initial(Grid)
7: for t in 0 to iteration
8:     if (rank % 2 ==1)
9:         MPI_Send tail and MPI_Recv head
10:     MPI_Send head and MPI_Recv tail
11: else
12:     MPI_Recv head and MPI_Send tail
13:     MPI_Recv tail and MPI_Send head
14:     n=rank%4;
15: #pragma offload target(mic:n) in(Grid[t]; length(Grid.size/2+1)) inout(Grid[t+1]; length(Grid.size/2 )) signal(Grid[t+1])
16: #pragma set_num_threads(240);
17: #pragma omp parallel for
18: for i in 0 to Grid.size/2
19:     cell = Grid[t][i]
20:     Grid[t+1][i] = TRANSITION(cell, t)
21: #pragma omp parallel for num_threads(8)
22: for i in Grid.size/2 to Grid.size
23:     cell = Grid[t][i]
24:     Grid[t+1][i] = TRANSITION(cell, t)
25: #pragma offload_wait target(mic:0) wait(Grid[t+1])
26:     achieve live cell at iteration t
27: MPI_Finalize()
```
4. Result and discussion

We conduct the experiments to implement MLC and GOL for CA-based simulation by four different programming models on supercomputer Beacon. Table 1 displays the execution time of MLC over one tile of 18 GB image on a single Xeon Phi coprocessor by different programming models. The performance of serial C program is from (Shi & Xue, 2017). Four programming models demonstrate better performance than sequential result. Besides, it can be found that the native and symmetric model have a good performance than the offload and hybrid-offload model for this embarrassingly communication case. The hybrid-offload model has about 1.8x speedup than offload model and symmetric model has about 1.15x speedup than native model.

Table 2 displays the results when multiple MICs were utilized to implement MLC. The hybrid-offload approach also shows a strong scalability than the symmetric model. As a result, hybrid-offload model almost realizes a 2× speedup in comparison to the symmetric model.

In the case of CA, two different grid sizes are tested, i.e. 8192 × 8192, 16,384 × 16,384. Table 3 displays the result when one MIC is utilized. In this case, the symmetric model shows the best performance and achieves about 5x speedup in comparison to the serial C program.
to implement GOL. When one MIC is utilized, among the four different programming models, the native model consistently outperforms the offload model for this intense communication case on single device. In the two hybrid programming models, the hybrid-offload model has more improvement than the symmetric model. The hybrid-offload model has about 1.75x speedup than offload model and symmetric model has about 1.2x speedup than native model.

Table 4 displays the results of implementing GOL on multiple MICs. Although the native and symmetric models are still able to reduce the computation time when moving from 1-processor implementation to 2-processor implementation, the performance hangs afterwards. For the native and symmetric programming models, it almost stops scaling when more processors are allocated. There are two reasons for the performance scaling to stop. (1) GOL is a communication dense application. When the problem size is fixed, double the number of processors will double the number of MPI processes, which will significantly increase the communication volume between 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. (2) The native mode is not supported very well on the Beacon supercomputer based on the comments from the Beacon administration staff. The communication efficiency between MIC cores will decrease when more cores (i.e. more MPI processes) are allocated. This could be a specific issue associated with the Beacon system, but it is unclear whether such a problem exists on other supercomputers. However, the offload and hybrid-offload models show a strong scalability when 2 or 4 MICs are used on the Beacon system.

In summary, the hybrid programming models take the advantage of the compute capacity of the host Xeon CPU, then could achieve better performance than its original models. In the native model and symmetric model, programs decompose the problem into evenly parts to compute so that all MPI processes can synchronize at some points without waiting a long time. However, the ranks running on the host CPU have a stronger computing capability than that running on Xeon Phi coprocessors. As a result, the slowest rank running on the coprocessors determines the computation time and final performance. The ranks running on the host CPU have to wait at the synchronize point for the slowest one. That is the typical load imbalance problem. The hybrid-offload approach seems to be able to improve the load balance and performance issue through asynchronous job dispatch routine.

5. Conclusion

Processing and analyzing big earth data will be a significant challenge to earth scientists and geoscientists when high resolution and volume of data are increasingly available. The emerging new hardware architecture and computational solutions are inevitable to resolve the big data problem. In this pilot study, we explored different parallel programming models to efficiently utilize a cluster of Intel’s MIC hosted at supercomputer Beacon to implement two kinds of geocomputation algorithms that are typical in geocomputation and geo-simulation. It can be concluded that the native programming model typically outperforms the offload model on a single MIC device. In both MLC and CA-based simulation, the hybrid-offload model achieves better performance in comparison to the offload programming models, while the hybrid-native, or symmetric, model achieves better performance in comparison
the native model when both host CPU and MIC coprocessors are utilized. Moreover, the hybrid-offload model shows a strong scalability and demonstrates the best performance when four MICs are used.

Both maximum likelihood classification (MLC) and Cellular Automata-based simulation could be highly suitable to Xeon Phi architecture, which employs massive amount of low-weight CPU cores and wide bandwidths on the Xeon Phi. However, when handling big data, one potential bottleneck for geocomputation on a single Xeon Phi is the limit of memory on the host CPU and MIC coprocessors. Clusters of heterogeneous host CPUs and MICs have to be efficiently utilized. For this reason, the four different programming models explored in this study could be extended to other types of geocomputation and geo-simulation in the future. Since repeatedly allocating and releasing memory on such a cluster would cause additional time and lower down the performance, further solutions need to be explored to develop appropriate strategy to transfer data between the host CPU and Xeon Phi coprocessor.

**Data availability statement**

The imagery data used in this study are a set of aerial photos with a high spatial resolution of 0.5 meter captured in 2007 covering the metropolitan area of Atlanta. Data may be available by submitting request through https://data.georgiaspatial.org/. Readers may use any other imagery data to complete the task of supervised image classification. The data used in the Game of Life are simulated. Readers may generate the data by themselves.

**Disclosure statement**

No potential conflict of interest was reported by the authors.

**Funding**

This research was partially supported by the National Science Foundation through the award SMA-1416509. This research used resources of the Beacon supercomputer, which is a Cray CS300-ACTM Cluster Supercomputer. The Beacon Project is supported by the National Science Foundation and the State of Tennessee.

**References**

Beacon. (2017). WWW document. Retrieved from [http://www.nics.tennessee.edu/beacon](http://www.nics.tennessee.edu/beacon)

Boschetti, M., Boschetti, L., Oliveri, S., Casati, L., & Canova, I. (2007). Tree species mapping with Airborne hyper-spectral MIVIS data: The Ticino Park study case. *International Journal of Remote Sensing*, 28(6), 1251–1261.

Buddenbaum, H., Schlerf, M., & Hill, J. (2005). Classification of coniferous tree species and age classes using hyperspectral data and geostatistical methods. *International Journal of Remote Sensing*, 26(24), 5453–5465.

Clark, M. L., Roberts, D. A., & Clark, D. B. (2005). Hyperspectral discrimination of tropical rain forest tree species at leaf to crown scales. *Remote Sensing of Environment*, 96(3-4), 375–398.
Duran, A., & Klemm, M. (2012). The Intel many integrated core architecture. In Proceedings of the International Conference on High Performance Computing and Simulation (HPCS) (pp. 365–366).

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

Greenberg, J., Dobrowski, S., & Vanderbilt, V. (2009). Limitations on maximum tree density using HSR remote sensing and environmental gradient analysis. Remote Sensing of Environment, 113, 94–101.

Guan, Q., Shi, X., Huang, M., & Lai, C. (2016). A hybrid parallel cellular automata model for urban growth simulation over GPU/CPU heterogeneous architectures. International Journal of Geographical Information Science, 30(3), 494–514.

Huang, M., Lai, C., Shi, X., Hao, Z., & You, H. (2015). Study of parallel programming models on computer clusters with Intel mic coprocessors. The International Journal of High Performance Computing Applications, 31(4), 303–315. doi:10.1177/1094342015580864

Kandalla, K., Venkatesh, A., Hamidouche, K., Potluri, S., Bureddy, D., & Panda, D. K. (2013). Designing optimized mpi broadcast and allreduce for many integrated core (mic) infiniband clusters. In Proceedings of the 21st IEEE Annual Symposium on High-Performance Interconnects (HOTI) (pp. 63–70).

Martin, M., Newman, S., Aber, J., & Congalton, R. (1998). Determining forest species composition using high spectral resolution remote sensing data. Remote Sensing of Environment, 65(3), 249–254.

Jeffers, J., & Reinders, J. (2013). Intel Xeon Phi coprocessor high performance programming. Burlington: Morgan Kaufmann. ISBN-13: 978-0124104143.

Richards, J. A., & Jia, X. (1999). Remote sensing digital image analysis (vol. 3). Berlin Heidelberg: Springer. doi:10.1007/978-3-662-03978-6

Rottensteiner, F., Trinder, J., Clode, S., & Kubik, K. (2005). Using the Dempster–Shafer method for the fusion of LiDAR data and multi-spectral images for building detection. Information Fusion, 6, 283–300.

Saule, E., & Çatalyürek, Ü. V. (2012). An early evaluation of the scalability of graph algorithms on the intel mic architecture. In Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW) (pp. 1629–1639).

Shi, X., & Xue, B. (2017). Parallelizing maximum likelihood classification on computer cluster and graphics processing unit for supervised image classification. International Journal of Digital Earth, 10(7), 737–748.

Shi, X., Lai, C., Huang, M., & You, H. (2014). Geocomputation over the Emerging Heterogeneous Computing Infrastructure. Transactions in GIS, 18(S1), 3–24.

Shi, X., Huang, M., You, H., Lai, C., & Chen, Z. (2014). Unsupervised image classification over supercomputers Kraken, Keeneland and Beacon. GIScience & Remote Sensing, 51(3), 321–338.

Spentzouris, P., Amundson, J., & Macridin, A. (2014). High performance computing modeling advances accelerator science for high energy physics. Computing in Science & Engineering, 16(6), 32–41.

Sridharan, H., & Qiu, F. (2013). Developing an object-based HSR image classifier with a case study using WorldView-2 Data. Photogrammetric Engineering & Remote Sensing, 79(11), 1027–1036.

Thenkabail, P. S., Enclona, E. A., Ashton, M. S., Legg, C., & De Dieu, M. J. (2004). Hyperion, IKONOS, ALI, and ETM+ sensors in the study of African rainforests. Remote Sensing of Environment, 90(1), 23–43.

Xiao, Q., Ustin, S., & McPherson, E. (2004). Using AVIRIS data and multiple-masking techniques to map urban forest tree species. International Journal of Remote Sensing, 25(24), 5637–5654.