ACCELERATING AGENT BASED MODELING FOR THE SIMULATION OF INFORMATION DIFFUSION USING GRAPHICS PROCESSING UNIT AND INTEL'S XEON PHIS

Xuan Shi, Ph.D.

Assistant Professor of GIScience, Department of Geosciences

Adjunct Faculty Member, Dept. of Computer Science and Computer Engineering

Core member, Institute for Advanced Data Analytics (IADA)

University of Arkansas, Fayetteville, AR 72701

Overview

- **Background**
- **Problems with GPU**
- **Hybrid solutions by MICs**
- **Result and conclusion**

Background

- In the process of information diffusion over social media network, each user has a probability to propagate message to its followers.
- People who have great influence on others are called opinion leaders while other people are called normal users.
- To demonstrate the information propagation over a real social network, an important thing is to find best probabilities for both opinion leaders and normal users through simulation via agent based modeling (ABM).
- However, this procedure is time-consuming because the algorithm needs to try many combinations of two different parameters to find the best parameter pair.
- For this reason, one goal of our team at UARK in year 3 of the IBSS project is to accelerate the ABM simulation of information diffusion using graphics processing unit (GPU) and intel's Xeon Phis with many-integrated cores (MICs).

Simulating information diffusion by ABM

- The ABM simulation involves multiple steps as
  1) define and generate a network (including nodes and links);
  2) detect any communities in the generated network;
  3) define diffusion parameters;
  4) perform simulation and observe proceedings; and
  5) visualize observed trends.

Defining diffusion parameters

1. The user specifies the number of seed nodes and the number of nodes of opinion leaders in each community (or as percentages)
   - Initially, set the number of seed nodes and select seed nodes with the specific algorithm chosen from a list of available ones in the tool.
   - Set the number of opinion leaders, or as a percentage of all nodes, inside each community.
   - In addition, select nodes from each community as opinion leaders.
   - Set (and select) nodes serving as the bridge nodes between communities.
2. Users specify the probability of a meme being diffused (retweet) from an opinion leader to all (or just a portion) of the nodes that follow the opinion leader. In addition, users need to specify the probability of a non-opinion leader node diffusing information. In addition, users to specify the probability of a node becomes active due to outside effects, i.e., information from outside the network, including TV, newspaper, and so on.
3. At each simulation step, observe the following
   - The percentage of the number of nodes (versus all nodes) what have seen the meme.
   - The number of steps taken to reach full coverage of all nodes, 95% of all nodes, 90%, 85%, 80%, 75%, 70%, 65%, 60%, 55%, 50%, 45%, 40%, 35%, 30%, 25% , 20%, 15%, 10%, and 5% of all nodes.

Implementation by Python (1)

Parallel function

```
 for i in range(lenParameterPair ):
            valueMatrix[i] = Diffusion(Nodes, seedNodes, opinionLeader, 
parametersList[i][0], parametersList[i][1])


def Diffusion(Nodes, seedNodes, opinionLeader, p_op_leader, p_n):
    activeNodes = set()
    nodetoActive = set(seedNodes.copy())
    while len(nodetoActive) > 0:
            v = nodetoActive.pop()
            activeNodes.add(v)
            ActiveNeighbors(v, Nodes, nodetoActive, activeNodes, 
opinionLeader, p_op_leader, p_n)
    return len(activeNodes)
```

Implementation by Python (2)

```
def ActiveNeighbors(v, Nodes, nodeToActive, activeNodes,
opinionLeader, p_op_leader, p_n):
for i in range(len(lstnbrs)):
        adoptedLeader = []
        adoptedNormal = []
        for n in Nodes[lstnbrs[i]]:
            if n in activeNodes:
                if n in opinionLeader:
                    adoptedLeader.append(n)
                else:
                    adoptedNormal.append(n)

if random.Random().uniform(0, 1) < (len(adoptedLeader) * p_op_leader
+ len(adoptedNormal) * p_n) / (len(adoptedLeader) +
len(adoptedNormal)):
         s.append(lstnbrs[i])
```

Corresponding CUDA solution (1)

```
 __global__ void Socialnet( int *valueMatrix,int *NodesA, int *lenPatameterPair, int *loops, int *seedNodes, int *opinionLeader, double
*parametersList, int *lenopinionLeader )
    {
        int i=blockIdx.x*blockDim.x+threadIdx.x;
        int j=blockIdx.y*blockDim.y+threadIdx.y;

        if( i < opinionLeaderLength && j<loopsLength)
        {
            valueMatrix[i*loopsLength+j] = Diffusion(i, NodesA, seedNodes, opinionLeader, parametersList[i*2+0],
parametersList[i*2+1],len_opinionLeader);
        }

    }
 __device__ int Diffusion(int index, int *NodesA, int *seeNodes, int *opinionLeader, double p_op_leader, double p_n, int len_opinionLeader)
    {
        int activeNodes[1200]={0};
        int nodetoActive[1200]={0};
        nodetoActive[0]=seeNodes[0];nodetoActive[1]=seeNodes[1];
            while (lenNodetoActive) >0)
            {
                int v = nodetoActive[lenNodetoActive -1];
                lenNodetoActive --;
                activeNodes[lenActiveNodes++]=v;
                ActiveNeighbors( index, NodesA,v,nodetoActive,activeNodes,opinionLeader,p_op_leader,p_n, lenNodetoActive,
lenNodetoActive, len_opinionLeader);
            }
        return lenNodetoActive;
    }
```

Corresponding CUDA solution (2)

```
__device__ void ActiveNeighbors(int index,int *NodesA, int v,  int
*nodeToActive, int *activeNodes, int
*opinionLeader,double  p_op_leader,double  p_n, int &lenA, int *lenN, int
len_opinionLeader)
	{
		int *lstnbrs=&NodesA[v*500+0];
		int adoptedLeader=0;
		int adoptedNormal=0;
		int sInActiveNodes=0;
		int count=0;
		int opinionLeader_select=0;
		int nodeToActive_select=0;
		for(int i=1;i<lstnbrs[0]+1;i++){
			count=0;
			int *index=&NodesA[500*lstnbrs[i]];
			for(int k=1;k<index[0]+1;k++)
				for(int j=0;j<lenA;j++)
					if(index[k]==activeNodes[j])
						count++;
						opinionLeader_select=0;
						for(int m=0; m< len_opinionLeader; m++)
					if(index[k]==opinionLeader[m])
							opinionLeader_select=1;
						if(opinionLeader_select==1)
							adoptedLeader++;
						else
							adoptedNormal++;
if(((adoptedLeader * p_op_leader + adoptedNormal * p_n) /
(adoptedLeader + adoptedNormal)) > 0.9)
			sInActiveNodes=0;
			for(int mm=0;mm<lenA;mm++)
				if(lstnbrs[i]==activeNodes[mm])
					sInActiveNodes=1;
		if(sInActiveNodes==0)
			nodeToActive_select=0;
			for(int mk=0;mk<lenN_d;mk++)
				if(lstnbrs[i]==nodeToActive[mk])
					nodeToActive_select=1;
				if(nodeToActive_select ==0)
					nodeToActive[lenN_d]=lstnbrs[i];
					lenNodetoActive++;
  adoptedLeader=0;
  adoptedNormal=0;
  }
}
```

GPU Warp divergence

- **Warp divergence Threads are executed in warps of 32, with all threads in the warp executing the same instruction at the same time.**
- **What happens if different threads in a warp need to do different things?**

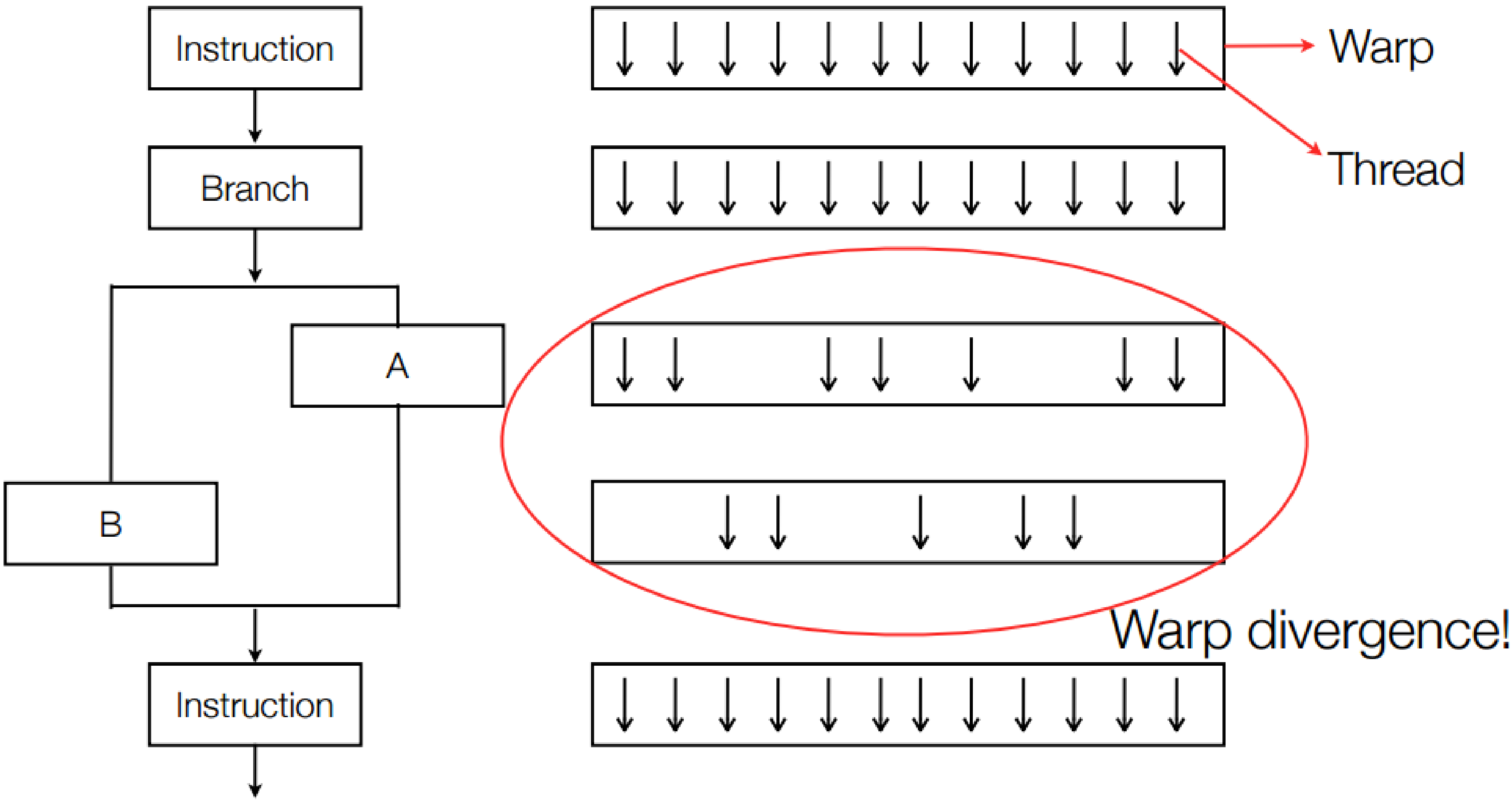
Control flow

- If statement
  - Threads are executed in warps
  - Within a warp, the hardware is not capable of executing if and else statements at the same time!

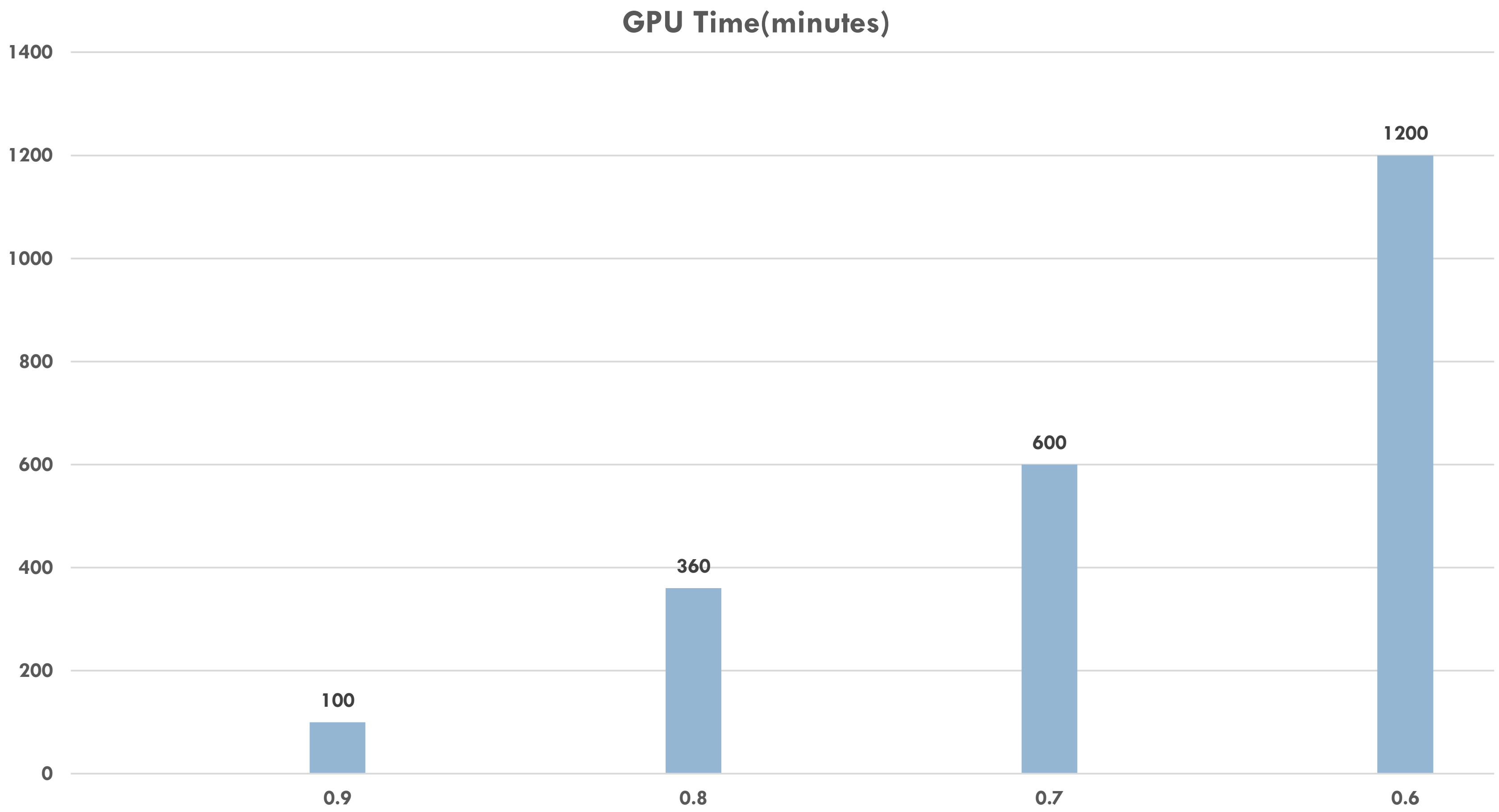
```
__global__void function();
{
        ....
        if(condition)
        {   ...
        }
        else
        {   ...
        }
}
```

Control flow

- How does the hardware deal with an if statement?



Result

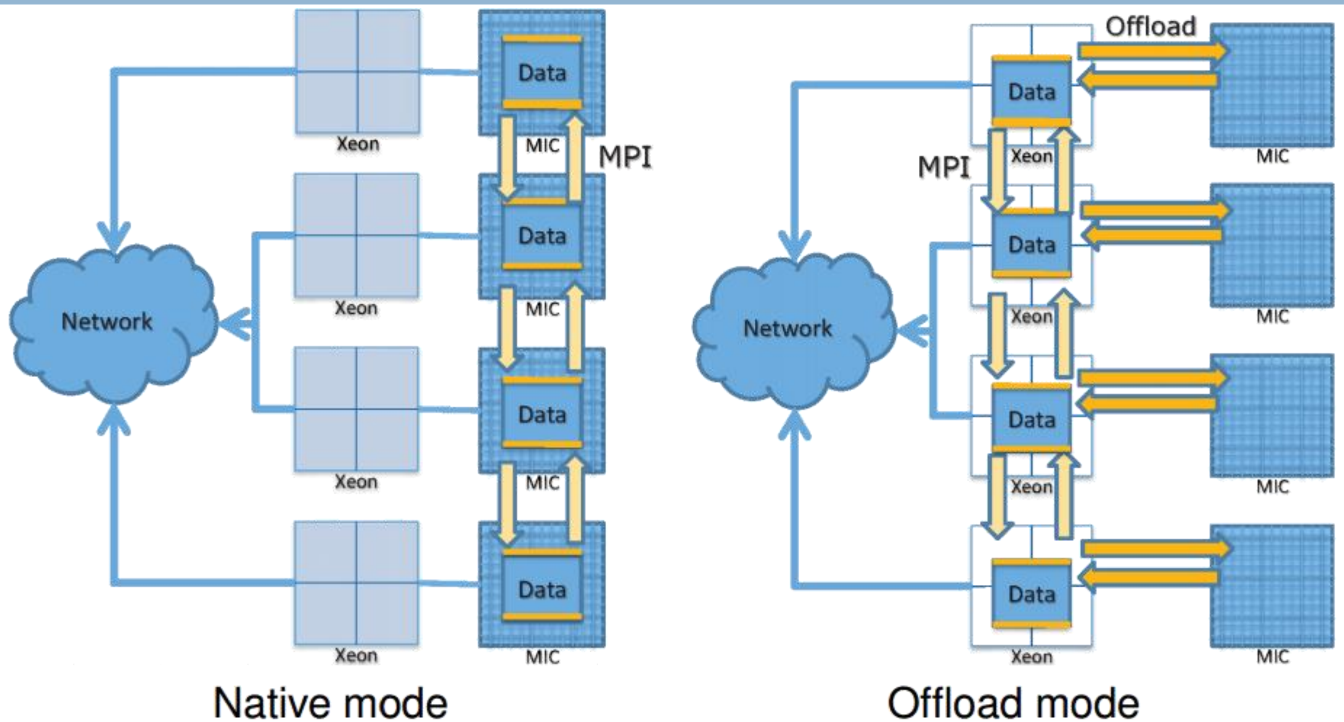


Pre-defined numbers used in the test

Deploying MICs on Supercomputer Beacon

- **MIC models**
- **Four parallel programming models**
- **Performance of four models**

MIC programming models



- MPI directly on MIC cores

- MPI on CPUs
- Offload to MIC by OpenMP

❖ In both models, Xeon CPU is not efficiently utilized

MPI + OpenMP Solution

- CPU is better to deal with an if statement than GPU.
- MPI uses distributed memory model on distributed network.
- OpenMP uses shared memory model on multi-core processors.

Four parallel programming models

- **Native Model**
  - No job is dispatched on CPU (Xeon)
  - Each MIC core directly hosts one single-thread MPI process. Therefore, if m MIC (Xeon Phi) coprocessors are used, m × 60 MPI processes are created in the parallel implementation

- **N-hybrid Model**
  - Both CPU (Xeon) cores and MIC (Xeon Phi) cores are utilized in the calculation.
  - Pure MPI applications on CPU and MIC

- **Offload Model**
  - The MPI processes are allocated on the 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.

- **O-hybrid Model**
  - Both CPUs and MICs are utilized for data processing on Beacon.
  - The workload is first distributed to CPUs through MPI. Then a host CPU will offload part of the job to a MIC card using OpenMP.
  - On the host CPU, OpenMP is used to spawn multiple threads for parallel processing.

Implementations on Four Models

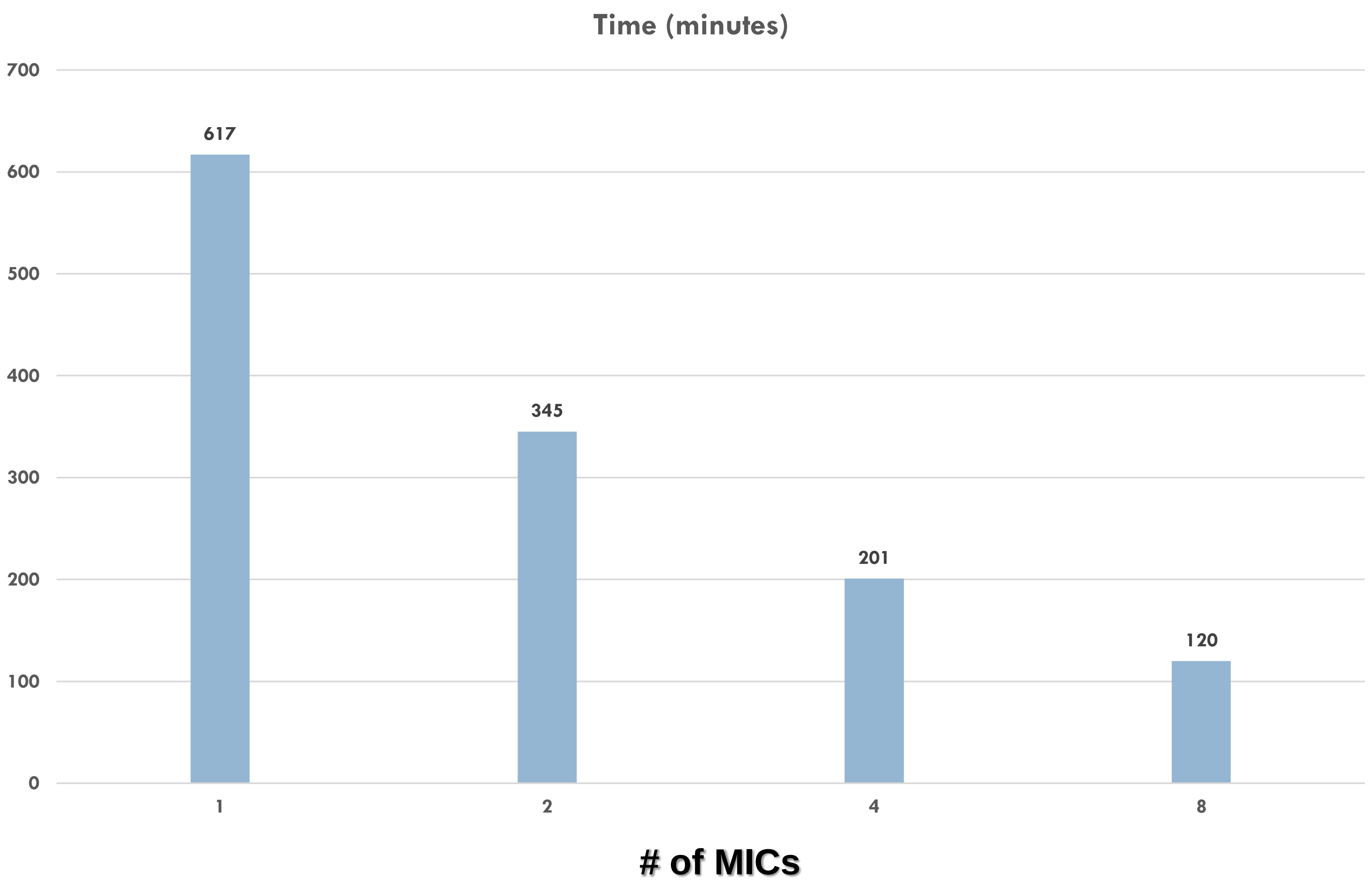
- **Native Model**
  - In this implementation, the MPI process is directly executed on each MIC core. The data is evenly distributed among MPI processes for computation
  - Each MIC emits 240 MPI processes.

- **N-hybrid Model**
  - Both CPU (Xeon) cores and MIC (Xeon Phi) cores are utilized in the calculation.
  - Each MIC emits 240 MPI processes and host CPU emits 8 MPI processes.

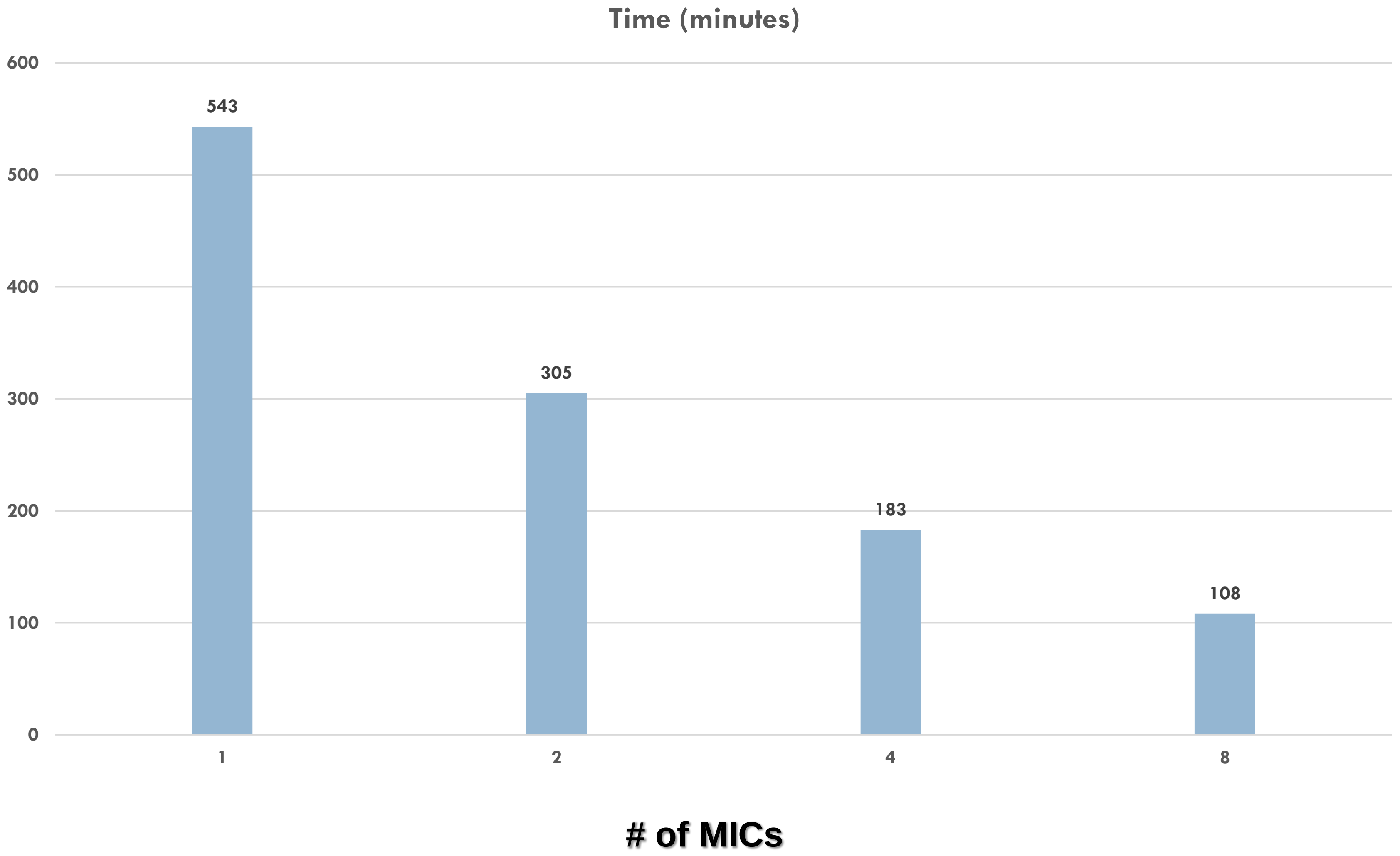
- **Offload Model**
  - The MPI processes are allocated on the host CPU cores
  - The MPI process specifies 240 threads to the MIC that uses OpenMP to handle data and calculation.

- **O-hybrid Model**
  - The workload is first distributed to CPUs through MPI. Then a host CPU will offload a half job to a MIC card using OpenMP.
  - On the host CPU, 8 OpenMP is used to spawn multiple threads for rest of the workload.

Native Performance



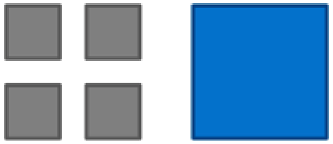
N-hybrid Performance



Native vs N-hybrid

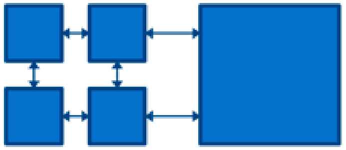
- **MPI in N-hybrid is like running on a heterogeneous cluster. Original load balanced codes may get imbalanced, because host and coprocessor computation performance are different .**

Native model



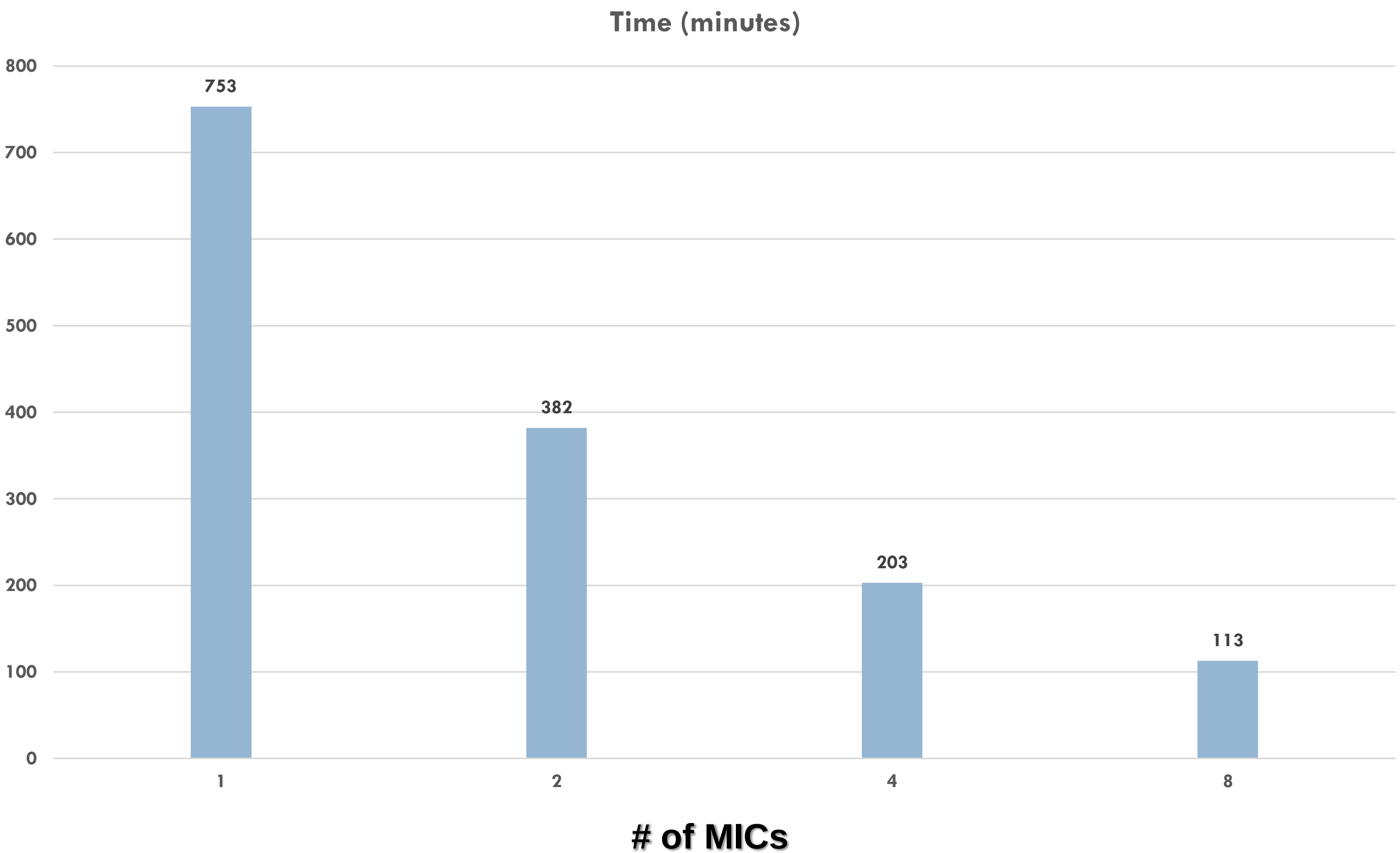
- Target Code:
  Highly parallel (threaded and vectorized) throughout.
- Potential Bottleneck:
  Serial/scalar code.

N-hybrid model



- Target Code:
  Highly parallel and performs well on both platforms.
- Potential Bottleneck:
  Load imbalance.

Offload Performance



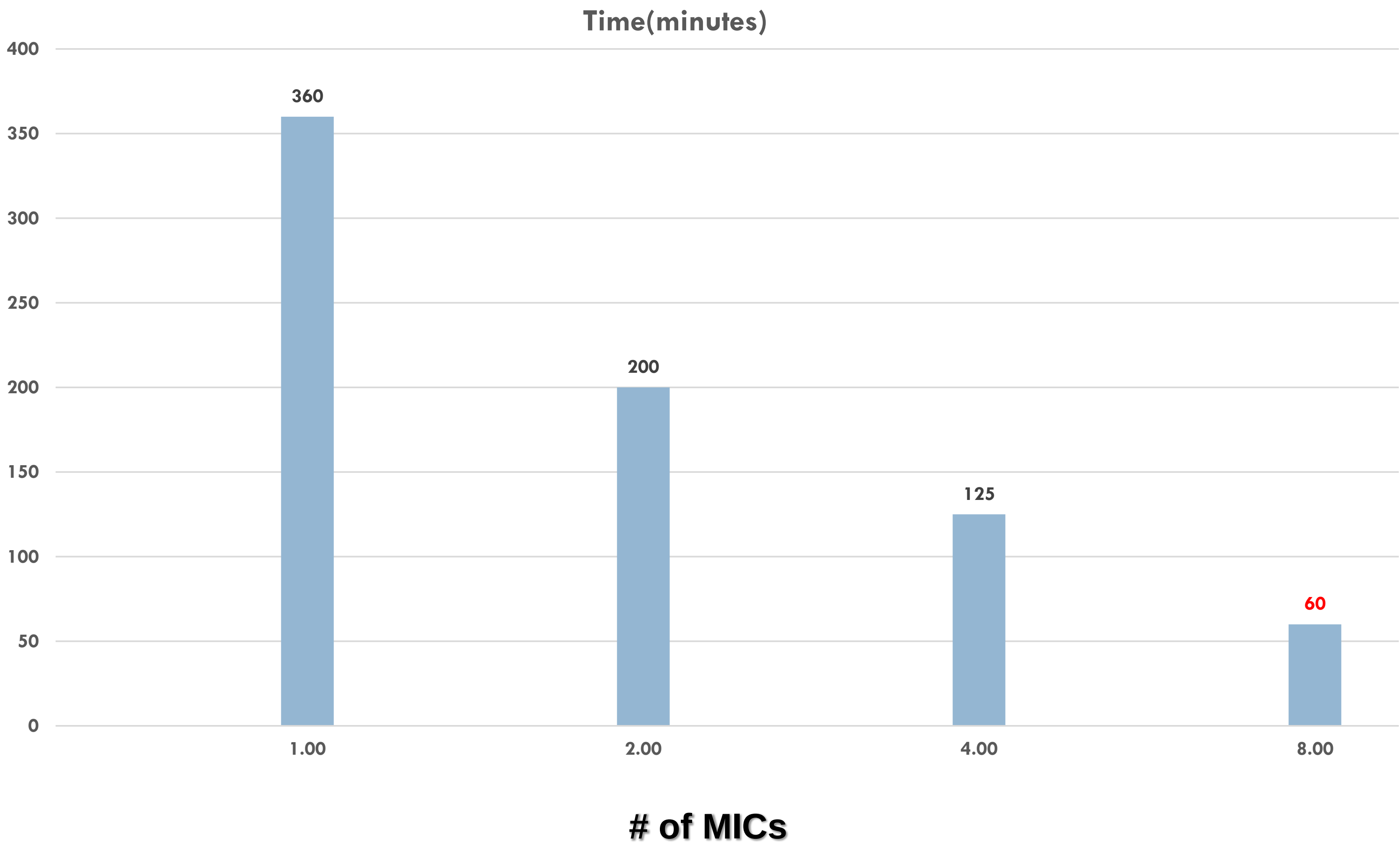
O-hybrid Coding

O-hybrid Coding Example

```
//Allocate memory on MIC  and transfer a half of data to MIC
#pragma offload target(mic:0) in(p: length( Psize/ 2  )  ) signal(p)
{
        #pragma omp for schedule(dynamic) num_threads(240)
        for(i=0;i< Psize/ 2; i++ )
                MICcalculation(p[i]) //MIC does a computation using p
}

#pragma omp for schedule(dynamic) num_threads(4)
for(k= Psize/ 2;k< Psize; k++ )
        HOSTcalculation( p[ k ]);   // Host CPU does a computation using p



#pragma offload_wait target(mic:0) wait(p)  // Do the offload only after
both MICcalculation() and HOSTcalculation() complete.
```

- In this programming model, we can decide how much data is going to be calculated in MIC or Host CPU.
- Dynamic scheduling works on a "first come, first served" basis.
- Both MICcalculation and HOSTcalculation are running simultaneously.
- An offload wait pragma is used to wait for completion of the MICcalculation() and HOSTcalculation() activitities.

O-hybrid Performance



Conclusion

- From the result, the native model and the offload model achieve very close performance for this work. Parallel implementation on O-hybrid model shows the best performance.

- O-hybrid does not have load balanced problem. We can decide how much data is going to be calculated in MIC or host CPU.

- O-hybrid model has a strong scalability.
  - If we use more MICs, such as 16 MICs, the work can be completed in 30 minutes.