Accelerating Agent Based Modeling for the Simulation of Information Diffusion Using Graphics Processing Unit and Intel's Xeon Phis

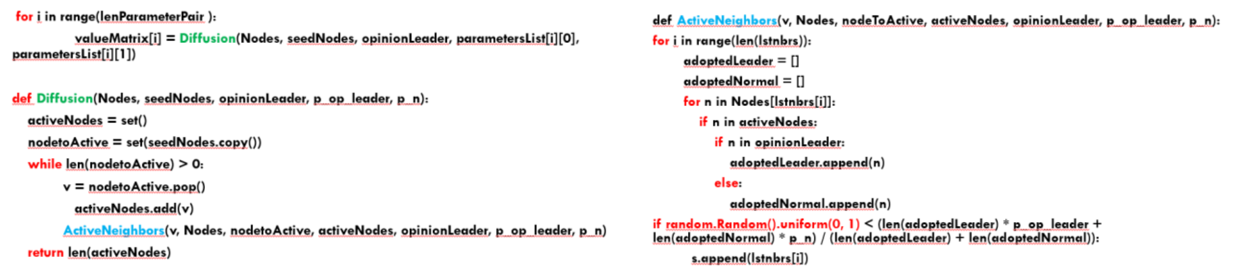
Xuan Shi, Ph.D.
Assistant Professor of GIScience, Department of Geosciences
Adjunct Faculty Member, Dept. of Computer Science and Computer Engineering
Core member, Institute for Advanced Data Analytics (IADA)
University of Arkansas (UARK), Fayetteville, AR 72701

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).

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. The process of defining diffusion parameters are the most time-consuming part and include multiple steps:

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.

The original ABM was encoded as serial Python scripts. The following graphics display the sections of Python scripts that need to be parallelized for acceleration:

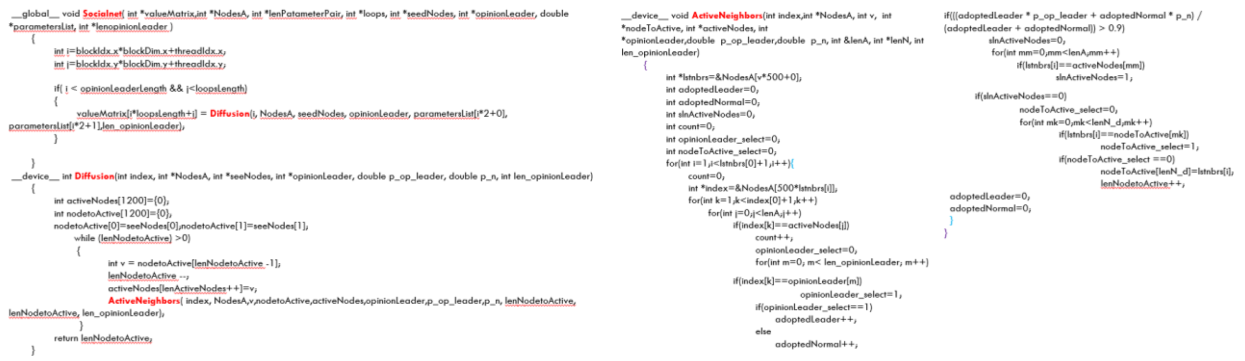


```
 for i in range(lenParameterPair ):
            valueMatrix[i] = Diffusion(Nodes, seedNodes, opinionLeader, parametersList[i][0],
parametersList[i][1])


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)
```

```
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])
```

First of all, we tried to transform the serial Python scripts into GPU solutions. Both pyCUDA and generic C CUDA programs were developed, for example, the corresponding C CUDA program is displayed in the following graphics:



```
 __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;
      }
```

```
__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 slnActiveNodes=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)
            slnActiveNodes=0;
             for(int mm=0;mm<lenA;mm++)
                  if(lstnbrs[i]==activeNodes[mm])
                        slnActiveNodes=1;
      if(slnActiveNodes==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;
  }
}
```

Unfortunately, the sequential Python program contain randomized procedures that devaluate the functionality of GPU. A typical CUDA-capable GPU is organized into an array of highly threaded streaming multiprocessors (SMs). Within the SM, computing threads are grouped into block, which is then managed by a grid structure. Within a block of threads, the threads are executed in groups of 32 called a warp. In the case of the random procedures, if different threads in a warp need to do different things, all threads will compute a logical predicate and several predicated instructions. This is called warp divergence. When all threads execute conditional branches differently, the execution cost could be the sum of both branches. Warp divergence can lead to a big loss of parallel efficiency. Thus the performance of GPU solution was even worse.

As a result, we developed four types of parallel solution using the MICs on supercomputer Beacon. Both the *native model* and the *offload model* only utilize the Xeon Phi coprocessor, while the host (Xeon) CPU is not efficiently used, or not used, in the ABM calculation. Two kinds of *hybrid solutions* were explored to optimally utilize both the Xeon CPU and MIC coprocessors. For example, to extend the native model, we create MPI ranks that reside on the host CPU and the MIC coprocessors. If *m* MIC (Xeon Phi) coprocessors and *n* host CPU processors are used, $m \times 60 + n$ MPI processes are created in the parallel implementation. In the case of offload model, 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, we also use OpenMP 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. In general, the hybrid-offload solution could be more flexible and efficient. When *8 MICs* were used, the hybrid-offload solution reduce the time from *96 hours* by Python serial program to *1 hour*. It is expected that the time can be further reduced to *30 minutes* when *16 MICs* could be used.