University of Texas at Austin

Project Report

EE382N Computer Architecture: Parallelism and Locality

Distributed Parallel PageRank using MPI

Authors:
Ashwini Kamath
Gurbinder Gill

Supervisor:
Dr. Mattan Erez

August 10, 2015

1 Introduction

In this project we implemented the **PageRank** algorithm on **Distributed** platforms using **MPI** for communication.

The **PageRank** algorithm for determining the "importance" or "Rank" of Web pages forms the core component of Google's search technology. As the Web graph is very large, containing over a billion nodes, page rank calculation is a very computation and memory intensive task which is mostly done offline. In order to store the graphs with billion nodes, one can have a heavy end system with large memory capacity and sufficient processing power, but these dedicated systems can be very expensive to built. Another approach to such a problem is to go distributed, using large number of machines and distributing graph nodes across these machines and communicating between them.
In this project we implemented **Parallel Bulk Synchronous Distributed PageRank** for 2 different platforms, viz.

- **NVIDIA's Tesla K20 GPU's**
- **Intel Xeon Phi Coprocessors**

We used **MPI**, specific to each platform, for communication. We studied scaling of the algorithm with respect to the number of hosts and tested it with the biggest graphs we could find (*Twitter-ICWSM10* with more than 51 million nodes and 2 billion edges).
Page Rank is a numeric value that represents the importance of a page present on the web. When one page links to another page, it is effectively casting a vote for the other page. More votes implies more importance. A web page is important if it is pointed to by other important web pages. Google calculates a page's importance from the votes cast for it. It matters because it is one of the factors that determines a page's ranking in the search results.

2 Page Rank Algorithm

Page Rank algorithm is illustrated with an example which is composed of five webpages and the links between them. The connections between pages is represented by a graph. A node represents a webpage and an arrow or an edge from page A to page B means that there is a link from page A to page B.

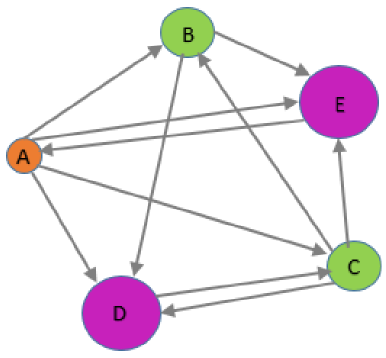


Figure 1: Page Rank Graph

In the example, Page A contains a link to page B, a link to page C, a link to page D and a link to page E. Page B contains links to page D and page E. Page C points to pages B, D and E, page D points to page C and page E points to page A.

$DampingFactor$:The PageRank theory holds that an imaginary surfer who is randomly clicking on links will eventually stop clicking. The probability, at any step, that the person will continue is a damping factor alpha.

There are many algorithms to calculate Pagerank. We looked at the bulk synchronous push based Pagerank algorithm primarily becuase it has more parallelism to exploit and is well suited for both the platforms we targeted.

2.1 The basic push based bulk synchronous Pagerank works as follows:

Each **Node** in a directed graph consists of :

- **Rank**: rank of each node
- **Residual**: sum of delta values collected from incoming neighbors

```
Result: PageRank Calculation
Update the Node PageRannk;
for N iterations or till it converges do
    for all Nodes (on a host) do
        Node.rank += Node.residual;
        if Node.outDegree > 0 then
            delta = Node.residual*ALPHA/Node.outDegree;
            for all Node.outNeighbors do
                neighbor.residual += delta;
            end
        else
            continue;
        end
    end
end
```

Algorithm 1: Push Based Bulk Synchronous PageRank

Where ALPHA is the damping factor.

Therefore,

- A page with more incoming links is more important than a page with less incoming links
- A page with a link from a page which is known to be of high importance is also important.

2.2 Our implementation

Both of the platforms we used (GPU and XeonPhi Coprocessors) provide large number of threads to work with, so we changed the above algorithm 1 to take advantage of these threads.

We decomposed the pageRank into 3 steps:

```
Result: PageRank Calculation: step 1
for all Nodes (on a host) on different threads do
    Node.rank += Node.residual;
    if Node.outDegree > 0 then
        delta = Node.residual*ALPHA/Node.outDegree;
        for all Node.outNeighbors do
            AtomicAdd(SendBuffer[neighbor.ID], delta);
        end
    else
        continue;
    end
end
```

Algorithm 2: PageRank Computation Phase

```
Result: Communication: step 2
for all hosts do
    MPI_All_to_All(SendBuffer, ReceiverBuffer);
end
```

Algorithm 3: Communication Phase

```
Result: Apply Residual: step 3
for all Nodes (on a host) on different threads do
    Node.residual += ReceiverBuffer[Node.ID];
end
```

Algorithm 4: Apply Residual Phase

We have seperated the communication phase and hence we have to buffer all the updates that need to be sent out in this phase. This is one extreme solution where we are buffering all the updates. The other extreme is to send messages along with the computation. However, the best policy would be somewhere in the middle, to buffer some updates and send them simultaneously with computation (overlapping communication and computation). But in this project we just used the first appraoch and we expect the communication to become bottleneck as we scale.

2.2.1 Graph Construction

Distributed graph is constructed from binary files (gr format). Required portion of the binary file is read on each host and stored as vector of nodes. Each node is a struct with **rank**, **residual** and **list of out going neighbors** to push updates. We operate over the vector of nodes.

2.2.2 Initialization Phase

In this phase, the PageRank of each node is initialized to a constant value (0.85 in our case) and one iteration of pagerank algorithm is run to initialize node residuals before running pagerank iterations.

3 Distributed Architecture

Systems using distributed memory, are essentially a collection of serial computers (nodes) working together to compute. Each node has rapid access to its own local memory and access to the memory of other nodes via some sort of communications network, usually a proprietary high-speed communications network. Data is exchanged between nodes as messages over the network. Unlike shared memory based parallel architecture, which are bound by the bandwith of the memory bus connecting processors, distributed systems are generally network bound.

In this project we have used both these architectures to increase the throughput and thereby decrease the compute time. We used the General Purpose GPU's from NVIDIA and Intel. The NVIDIA GPU is the Tesla K20 which has which has 13 SM's and 2496 cores(ALU's) and the Intel GPU is the Xeon-Phi which has 61 cores. Message Passing Interface(MPI) is used to communicate between two or more of these nodes.

3.1 MPI for Communication

MPI (Message Passing Interface) is a language-independent communications protocol used to program parallel computers. Both point-to-point and collective communication between hosts is supported.

In this project we have utilized one of the common collective communication routine `MPI_Alltoall` which rearranges n items of data such that the nth node gets the nth item of data from each.

$MPI\_Alltoall(constvoid^{*}sendbuf, intsendcount, MPI\_Datatypesendtype,$
$void^{*}recvbuf, intrecvcount, MPI\_Datatyperecvtype, MPI\_Commcomm);$

An example for the `MPI_Alltoall` between 4 nodes which are sending 8 integers each is as shown below.

```
MPI_Alltoall ( u, 2, MPI_INT, v, 2, MPI_INT, MPI_WORLD_COMM);
```

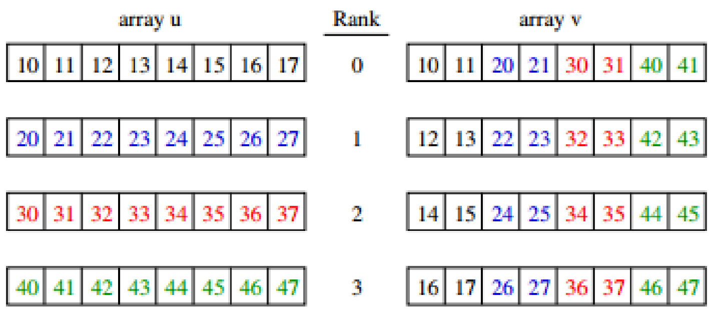


Figure 2: $MPI\_Alltoall$

3.2 GPUs from NVIDIA

We ran all our experiments on **stampede** supercomputer where each node consists of NVIDIA's Tesla K20. Each K20 has 13 SMs and 2496 cores(ALU's).

- **Computation** : Each thread worked on a node in the graph belonging to it's respective host.
- **Communication**: Since we are combining MPI and CUDA, we wanted a direct communication between GPUS's. CUDA-Aware MPI makes it easy to communicate between hosts running application on GPU's by allowing to direclty exchange device

pointer, therefore one doesn't have to copy between host and device to communicate with other devices. Buffers can be directly copied between the memories of two GPU's in the same system. Without CUDA-aware MPI, you need to stage GPU buffers through host memory, using cudaMemcpy.

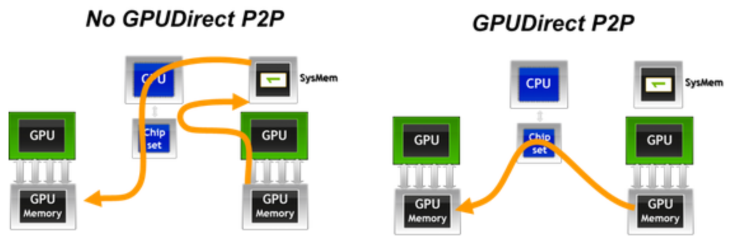


Figure 3: Traditional MPI vs CUDA Aware MPI

We had to install $mvapich2-2.1$ with $cuda\_enable$ option, as it was not enabled in the existing MPI installation on stampede. To run jobs using our custom installed MPI: We have to set ENV variable $MPICH\_HOME = "/work/02982/ggill0/modules/mvapich2-2.1-install/"$ And $MV2\_USE\_CUDA = 1$

3.3 XeonPhi Coprocessors from Intel

The most common execution models for Xeon can be broadly classified as

- **Offload Execution Mode:** Also known as heterogeneous programming mode, here the host system offloads part or all of the computation from one or multiple processes or threads running on host. The application starts execution on the host. As the computation proceeds it can decide to send data to the coprocessor and let that work on it and the host and the coprocessor may or may not work in parallel. This is the common execution model in other coprocessor operating environments.

- **Symmetric Execution Mode:** In this case the application processes run on both the host and the Intel Xeon Phi coprocessor. They usually communicate through some sort of message passing interface like MPI. This execution environment treats Xeon Phi card as another node in a cluster in a heterogeneous cluster environment.

- **Coprocessor Native Execution Mode:**An Intel Xeon Phi hosts a Linux micro OS in it and can appear as another machine connected to the host like another node in a cluster. This execution environment allows the users to view the coprocessor as another compute node.

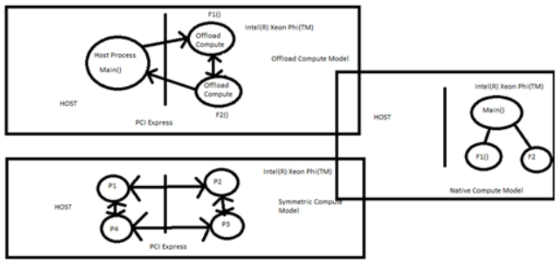


Figure 4: Intel Xeon Phi Execution Models

We chose the Coprocessor Native Execution Mode since our algorithm is communication and computation intensive. Offload Execution mode would result in unnecessary communication between the host processor and the coprocessor. In case of Symmetric Execution Mode, the host would also be counted as a node, but it would not provide the throughput capability of an Intel Xeon Phi Coprocessor. Hence the best performance could be gained from Coprocessor Native Execution mode, where every node will have balanced throughput.

4 Results and Analysis

We experimented with graphs with varying sizes as listed in table 1 and tried to use huge graphs with biggest being **Twitter-ICWM10** graph with more than **51 million** nodes and **2 billion** edges. Twitter-ICWN10 is the actual twitter graph from twitter.com and rmats are the random generated graphs.

Graph	**Number of Nodes**	**Number of edges**
RMAT 20	1,048,576	8,259,994
RMAT 21	2,097,152	16,570,170
RMAT 22	41,94,304	33,226,138
RMAT 25	33,554,432	267,096,620
TWITTER	51,161,011	1,963,212,211

Table 1: Graphs Used

Pagerank was run for 60 iterations and all the measurements are the average over 10 runs. Running pagerank for fixed number of iterations is fair since we are not comparing between different pagerank algorithm but rather we are comparing same algorithm on different platforms.
We ran 1 **MPI** task per host and we scaled upto 32 hosts on each platform.

4.1 Runs on GPUs

We fixed the *Blocksize* of 512 threads on GPUs and the *GridSize* is based on the number of nodes in a graph. For all the graphs, our pagerank algorithm scaled as we increased the number of hosts (or nodes, not to be confused with the graph nodes).

Figure 5 shows the run time of pagerank on twitter and rmat25 graphs on GPUs. Each host consists of one GPU and as we increase the number of hosts, the number of nodes being processed by each host decreases and number of threads to do the work increases significatly and hence we get good scaling. Similar trends can be observed for Rmat22 Rmat21 and Rmat20 graphs in 6. If you look closely, scaling is not very significant for smaller graphs like Rmat20 which has only 1 million nodes, since there is not enough work to share between different hosts, therefore all the resources are not being utilized efficiently.

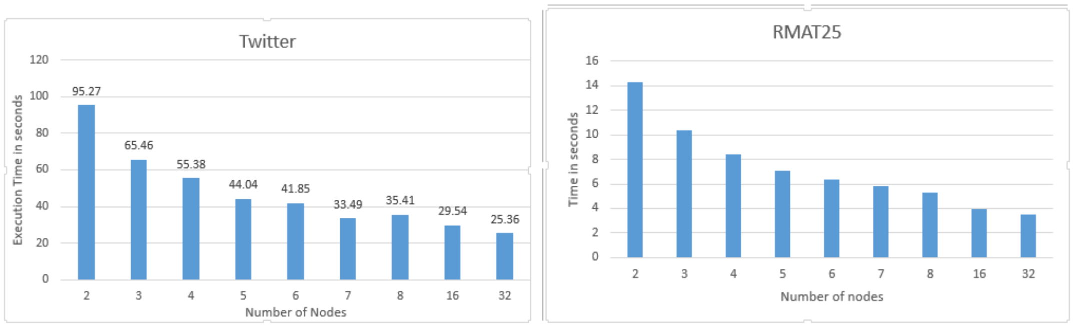


Figure 5: Run time of Twitter and RMAT 25 graph on CUDA

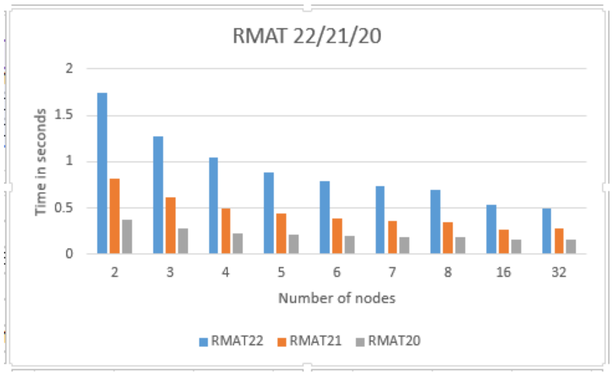


Figure 6: Run time of RMAT 22,RMAT 21 and RMAT 20 graphs on CUDA

There are two basic ways to measure the parallel performance of a given application, depending on whether or not one is cpu-bound or memory-bound. These are referred to as strong and weak scaling, respectively. In case of strong scaling, size stays fixed but the number of processing elements are increased. This is used as justification for programs that take

a long time to run (something that is cpu-bound). The goal in this case is to find a "sweet spot" that allows the computation to complete in a reasonable amount of time, yet does not waste too many cycles due to parallel overhead. In strong scaling, a program is considered to scale linearly if the speedup (in terms of work units completed per unit time) is equal to the number of processing elements used (N). In general, it is harder to achieve good strong-scaling at larger process counts since the communication overhead for many/most algorithms increases in proportion to the number of processes used. This overhead can be seen in this implementation of the Page Rank algorithm.
As we mentioned in 2.2, our algorithm consists of 3 components (Pagerank update, MPI Communication and Apply phase), so we also measured the timing for these different components. Figure 7 shows the decomposition of the total time into above mentioned 3 components for rmat25 graph and twitter graph. It can be observed that for smaller number of hosts, computation dominates over communication since we have less resources for computation at our disposal, but as we increase the number of hosts, communication starts to become the bottleneck.

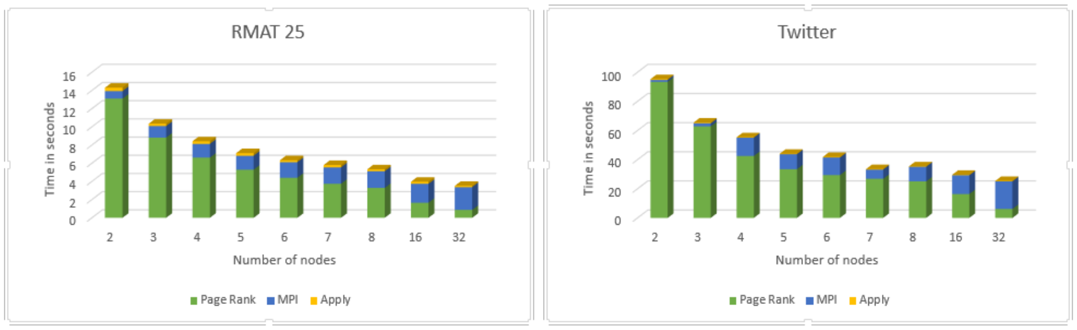


Figure 7: Decomposition of execution time of RMAT 25 graph and Twitter graph

Weak scaling is a measure for memory bound applications. Since Page Rank algorithm is communication bound and not memory bound, we did not delve more into this measurement.

The below graphs explicitly show scaling as the number of nodes increase. The scaling is measured with respect to the time taken by a system containing 2 hosts, since we were not able to load the huge graphs on 1 node.

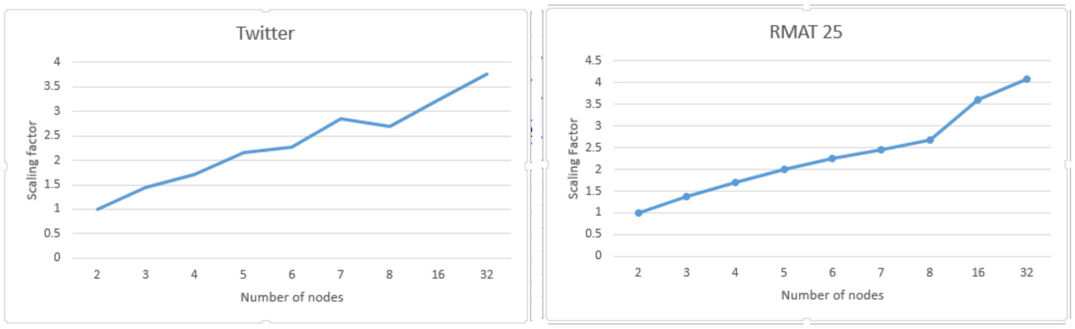


Figure 8: Scaling of Twitter and RMAT 25 graph on CUDA

4.2 Runs on XeonPhi

For XeonPhi coprocessors, we used the maximum number of threads available (240 threads on 60 cores) and used openMP to utilize all the threads. Figure 9 shows the pagerank scaling for the twitter graph on XeonPhi. We were not able to fit the twitter graph on less than 4 hosts because of insufficient memory. Figure 10 shows scaling for rmat graphs. XeonPhi shows similar trends as the pagerank on GPUs.

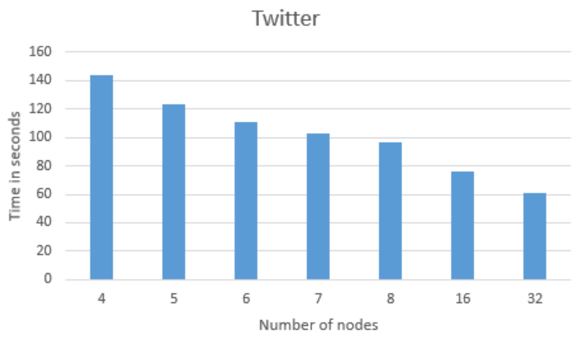
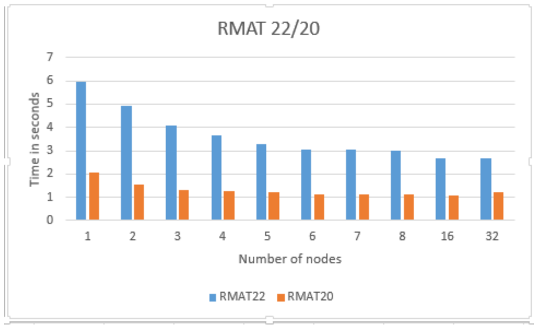


Figure 9: Run time of of Twitter graph on Xeon



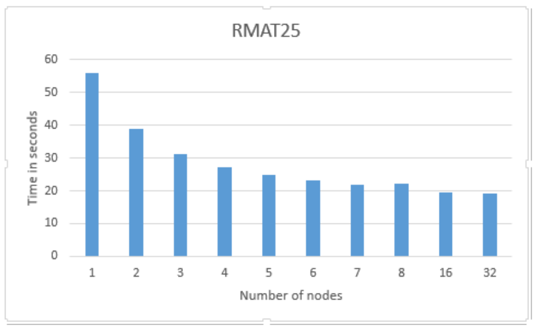
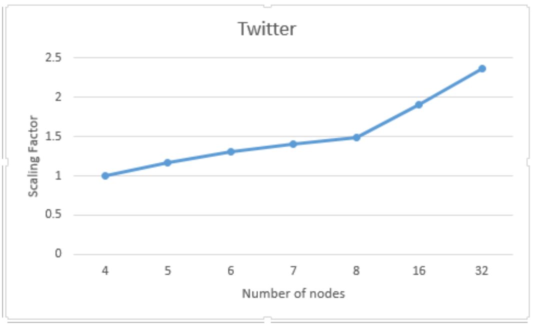


Figure 10: Run time of RMAT 25, RMAT 22 and RMAT 20 graphs on Xeon

The below graphs explicitly show scaling as the number of nodes increase. The scaling is measured with respect to the time taken by a system containing 4 hosts in case of twitter, since we were not able to load the huge graphs on 1 node.



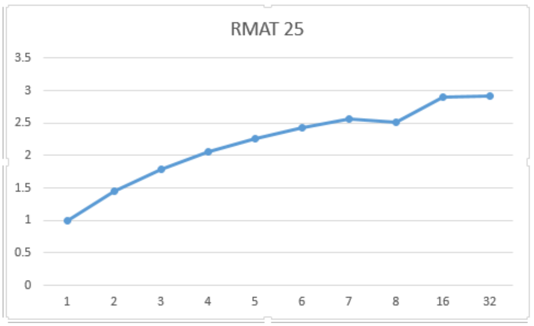


Figure 11: Scaling of Twitter AND RMAT 25 on Xeon

4.3 GPU's Vs XeonPhi

We also compared the PageRank on GPUs and XeonPhi coprocessors and observed that GPUs always win primarily beacuse of more number of resources (threads) at their disposal. Figure 12 shows the speedup of the pagerank algorithm on GPUs over XeonPhi coprocessors on twitter graph and figure 13 shows the same thing for the smallest rmat20 graph.

We can see that there is more than 7X speedup on GPU over XeonPhi for 32 hosts when the graph is small. But for larger sized graph, the speedup reduces to about 2.8X. This may be due to the fact that NVIDIA GPU threads have less memory available to them when compared to the Xeon Coprocessors. Hence the speedup attenuates for larger graphs.

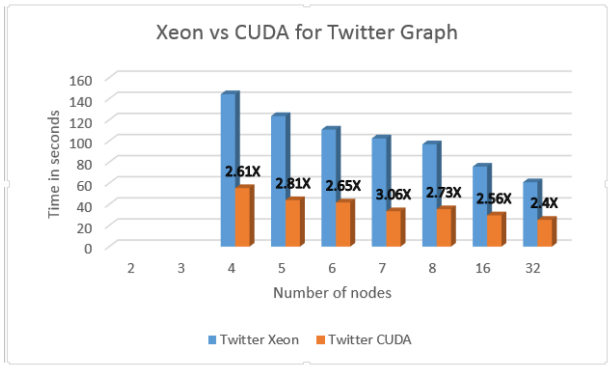


Figure 12: Comparison of run time on Xeon and CUDA Processors for Twitter graph

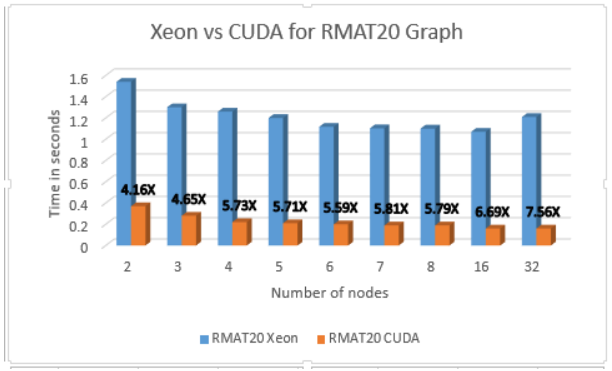


Figure 13: Comparison of run time on Xeon and CUDA Processors for RMAT 20 graph

5 Sources

1. *J.J.Whang, A.Lenharth, I.S.Dhillon , Scalable Data − driven PageRank − Algorithms, System Issues and Lessons Learned Euro − Par* 2015

2. $http : //www.math{-}cs.gordon.edu/courses/cps343/presentations/MPI_Collective.pdf$

3. *http : //devblogs.nvidia.com/parallelforall/introduction − cuda − aware − mpi*

4. *https : //software.intel.com/en − us/articles/intel − xeon − phi − core − micro − architecture*

5. *https : //software.intel.com/en − us/articles/intel − xeon − phi − programming − environment*

A Compiling and running Cuda-Aware MPI

A.1 Compilation

It requires gcc/4.7.1 and mvapich2-2.1.
We installed our own mvapich2-2.1 with cuda_enabled **ON**. Therefore to compile this code one need mvapich2 with cuda_enabled ON since only then it will be able to work with CUDA device pointers.
Our custom mvapich2 is installed in the following directory and it has required permissions for everyone to compile and execute code.:

/work/02982/ggill0/modules/mvapich2-2.1-install/

To compile, there is compile.sh script with required commands to compile this code.

source compile.sh

A.2 To Run

You need to set **MPICH_HOME** and **MV2_USE_CUDA**

```
export MPICH_HOME="/work/02982/ggill0/modules/mvapich2-2.1-install/"
MV2_USE_CUDA=1 ibrun PageRank inputgraph.gr
```

B Compiling and running XeonPhi Coprocessor MPI

B.1 Compilation

To compile this code one need to do the following:

1. **source pre_script :** to load the required modules and set environment variables.
   - module load intel/14.0.1.106
   - module load impi/4.1.3.049
2. **source compile.sh :** to compile the code

B.2 To run

1. **source pre_script** or change the **.bashrc** file so that required modules can be loaded when job is scheduled.
2. To Run $ibrun.symm - m".\/PageRank.mic..\/inputs\/rmat20.gr"c$ $".\/PageRank.out..\/inputs\/rmat20.gr"$