ECE 420 Lab 4 Report
March 26th 2019
Riley Dixon, Nathan Liebrecht

Description of Implementation

The implementation of a parallel solution to the PageRank algorithm in MPI was broken down into 4 main tasks. The first part is to initialize and count the number of nodes and edges that exist in the graph. Afterwards each process initializes their own copy of the input graph. Secondly, the PageRank algorithm actually begins. This is split into two parts, the calculation step, and a update step. The calculation step runs one iteration of the PageRank algorithm on a contiguous partition of the nodes that have been assigned to that process from the first step. After each step is done calculating one step, the processes wait until each other is finished before sending each other their partition of the array that has each node's PageRank value to each other process. The MPI synchronization function MPI_Allgatherv(...) was selected as the best candidate for processes to communicate with one another. Once every process has the full updated array, the calculation step begins again until the values converge at which point the algorithm is finished. Lastly, the final results and the processing time are then saved into an output file on the machine that is running the master process.

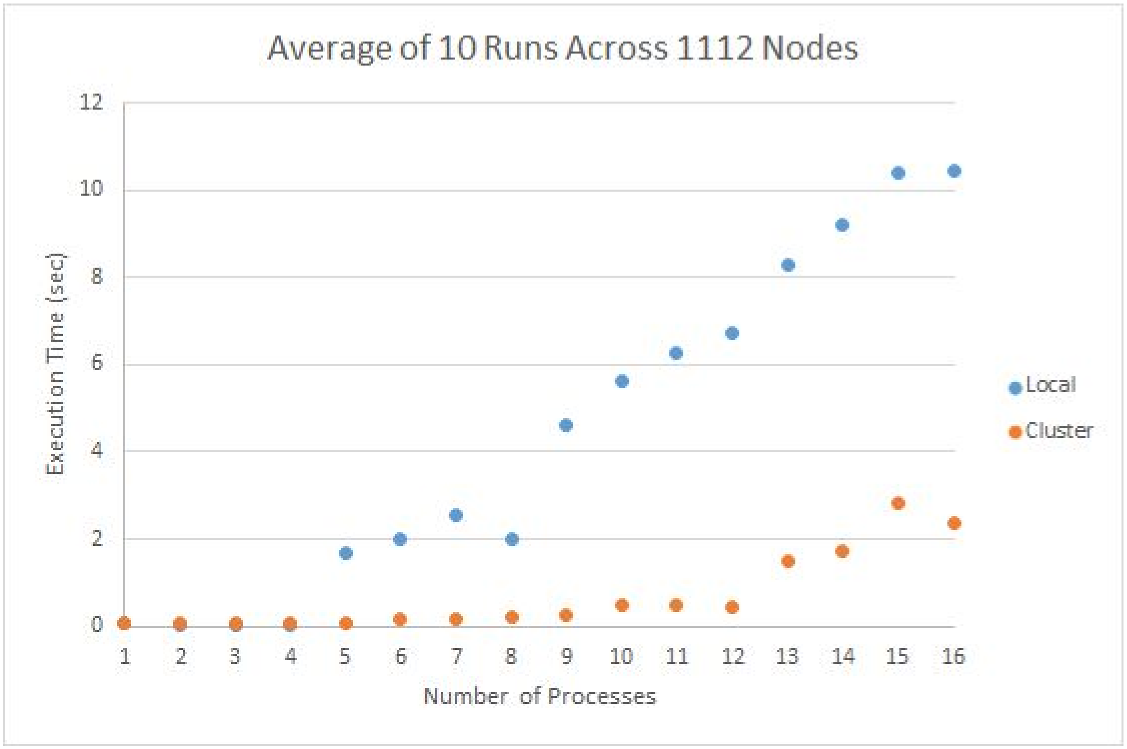
Partitioning, Communication, and other Issues

To make the number of iterations across all processes even in the page rank algorithm, that is to minimize the amount of time spent waiting for other processes to finish calculations, the array of nodes is split so that the total number of edges to process on each iteration is as even as possible across each process. This split is done by the master thread before the PageRank algorithm begins and the start and end positions of each partition is sent to every process in the team. Each partition is continuous in the full PageRank array. Then, each process calculates the r(i+1)th step for the data points included in its designated partition.

Communication overhead is kept to minimum by updating the array once per iteration step instead of other potential methods. Even though a large amount of data has to be sent each iteration step, it is sent all at once to minimize the overhead of setting up communication. Paired with how the data has been partitioned, this also minimizes the amount of time spent waiting for other processes to be ready to receive data as well. Some points of the program, notably during the initialization step also use barriers to not begin processing data until the program has been setup. As mentioned in the implementation section, MPI_Allgatherv(...) was selected as the function used to update the data after each iteration. This was chosen because each process sends its partition of the data, which is then pieced together and the entire section of the data is then returned as a whole. Additionally the vectorization of the MPI_Allgatherv(...) function allows us to have unequal partition lengths in each process. As such, this offered the simplest, as well as the most efficient way for us to share data between the processes after each calculation step.

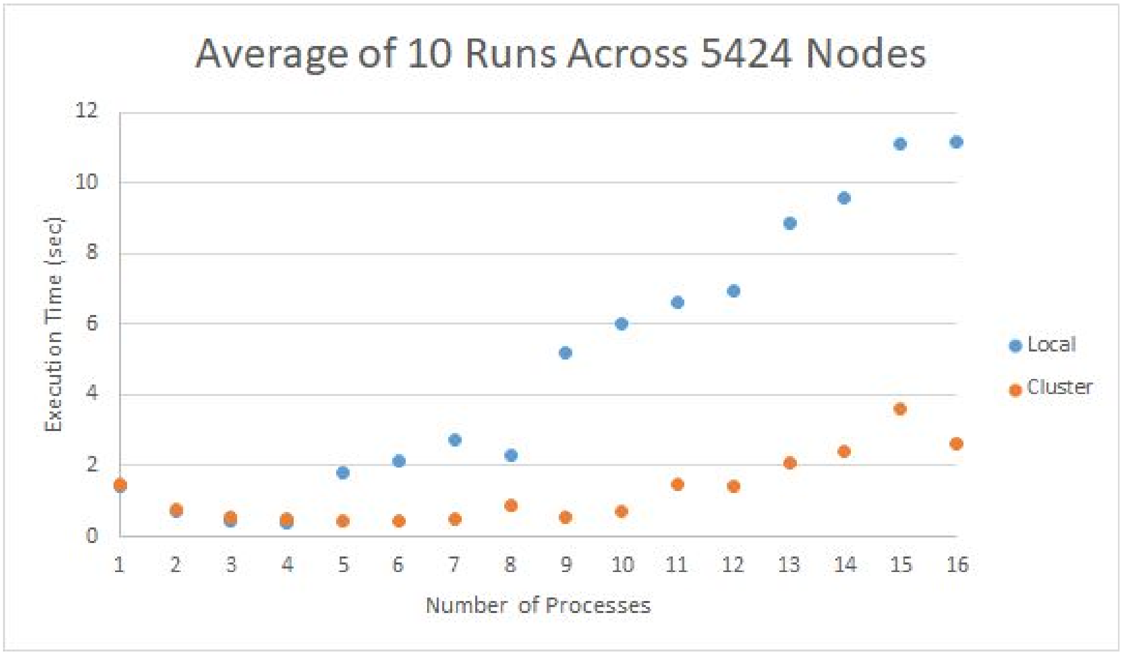
Overall the program runs better on the cluster of machines in the cloud over executing on a single machine in the cloud. The reason behind this is that having the additional independent processors calculating the PageRank of each node outweighs the amount of communication overhead during the update step. Executing the program with 4 processes or less yields nearly identical results for both local (1 machine) execution and cluster (4 machines) execution. However as soon as more than 4 processes are used, the disparity between the local and cluster configurations grows quickly, with the cluster configuration outperforming the local configuration.

During each configuration for a certain number of nodes, the same input data files are used for each calculation. This will ensure that any variance in time is not associated with a change in the input files.



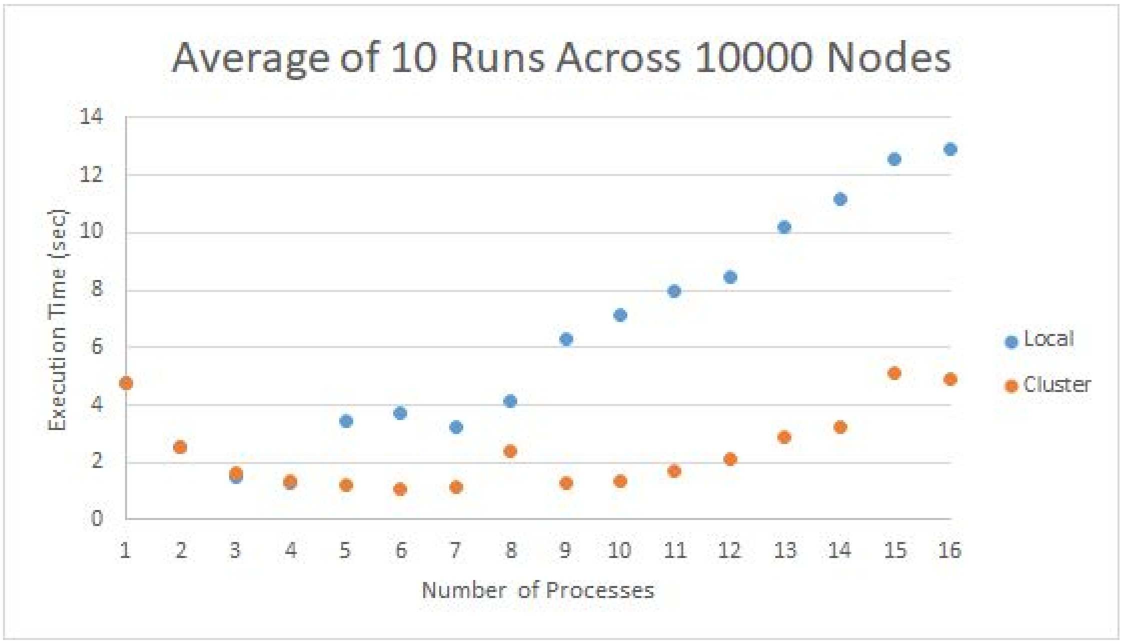
Scalability: As we can see, for 1112 nodes there is very little benefit to using the cluster since we are able to fit the data on one machine, which provides lower latency communication. We are able to achieve similar times on the local machine when compared to scaling to four nodes.

Granularity: here we can see that less or equal to four processes is ideal for both one machine and cluster approaches. However we can see that as the number of processes increases, the cluster approach wins over the local approach, thanks to the extra computing resources available.



Scalability: Compared to the previous run, we see that the cluster continues to scale up until around 7 processes thanks to the larger problem size.

Granularity: Again, the number of processes seems optimal around 4 or 5, for the local and cluster modes respectively. At this problem size we see that the cluster machine is starting to show a real ability to increase the number of processes while also increasing performance.



Scalability: Finally with the 10k nodes size we can see that the cluster scales much better that local runs, with performance improvements being had up until 6 processes.

Granularity: At 10k node size we see that adding more processes to the local mode will drastically increase the runtime when compared to the cluster mode. The optimal number of partitions seems to still be around 4 or 6 for the local and cluster modes respectively.

In conclusion, as the problem size continues to increase, we are able to see more and more advantages to using the cluster. However, the local mode comes very close to the cluster timings at these problem sizes as long as we do not exceed four processes on the local mode. Beyond four processes, the cluster mode only provides marginal performance improvements. We suspect this is due to the communication overhead outweighing the benefits of scaling out, at least at this problem size. Perhaps with a larger number of nodes the advantages of clustering would be more apparent.