ECE 420 Lab 4

Implementing PageRank with MPI

Section H21

By Ang Li, Dillon Allan, and Ryan Brooks

April 1, 2021

Description of implementation

In Lab 4, the PageRank algorithm was parallelized using Message Passing Interface (MPI) as shown in main.c. The file contained two functions: *main()* and *pagerank()*. The driver code for the program was kept in *main()*, which began by declaring local variables and initializing a team of MPI processes using *MPI_Init()* (lines 20-29). Next, the graph was read from the files data_input_meta and data_input_link and each node in the graph was initialized using the library function *node_init()* found in Lab4_IO.c (lines 34-47). The algorithm was then run by having all processes execute *pagerank()*, which populated an array of ranks corresponding to the rank of each node in the graph (Line 48). The final ranks and latency were saved to a file named data_output using the library function *Lab4_saveoutput()* before the program exited (lines 51-61).

The *pagerank()* function contained a parallelized implementation of the PageRank algorithm. First, the function allocated heap memory for the ranks calculated in the previous iteration and for the contributions to the current rank due to a node's incoming links. Additionally, a local ranks array was allocated so that each process running the function could calculate a partition of all ranks (line 101). The size of each partition was equal to the floor of the node count divided by the process count, except for when the node count was not a multiple of the process count. In that case, the partition size for the last process was extended to also include the remainder (line 96). After the root process set the initial ranks and contributions, each process entered a loop that contained the core PageRank calculation (lines 114-143). Inside the loop, the root process used *memcpy()* to save the previous ranks before sharing them with the other processes using *MPI_Bcast()*. Next, the current ranks were partitioned among the processors using *MPI_Scatter()*. Each process updated their set of local ranks, which were then recombined into the main array of ranks using *MPI_Allgatherv()*. The updated ranks were then used to set the contributions for the next iteration. This process continued until the relative error between the current and previous ranks was less than 0.001% (set by the macro EPSILON).

Performance discussion

Table 1 below shows the top average and median PageRank latencies for serial, single machine, and cluster setups for increasing graph sizes. Table 2 shows the average PageRank latency of the single machine setup for increasing process counts. Finally, Table 3 shows the average PageRank latency of the cluster setup for increasing process counts. Additionally, each table is visualized with a corresponding bar graph in Figures 1, 2, and 3.

Is the performance of your program better on a single machine setup or on a multiple machines setup? What is the reason?

The performance of our program was best when using multiple machines for problem sizes larger than 1112 nodes (we have a latency of 0.3s for multiple machines and 0.911s for a single machine when testing with 10000 nodes). When using smaller problem sizes, it is better to use a single machine (0.0253s for a single machine as compared to 0.033s for multiple machines). This was likely due to multiple machines having a larger amount of cores to run the problem on than with a single machine.

This larger amount of cores was enough to overshadow the overhead of communicating between machines for larger problem sizes, which led to an almost 3x gain in speedup when compared to the average latencies for each node size (5424, 10000) and running the program on one machine and on multiple machines. However, at lower problem sizes (1112 nodes) the overhead of communicating between multiple machines was too large to make parallelizing over multiple machines faster than using a single machine. To illustrate this speedup, consider the values at 10000 nodes for the single machine and multiple machine setups from Table 1. For the single machine we have a value of 0.911s and for the multiple machine value we have 0.3s. To see the 3x speedup we can compare single machine parallel and multiple machine parallel by dividing 0.911 by 0.3 to get 3x speedup when using multiple machines over a single machine. For smaller sizes we see a speedup of 0.033/0.0253 = 1.3x for a single machine compared to the multiple machine setup.

What is the best number of processes that should be used in your program, respective to the different problem sizes? How does the granularity affect the running time of your program and why?

It was better for problem sizes under 1112 to choose 4 processes within the same machine since the MPI communication overhead significantly contributed to the overall latency (especially among different clusters). For problem sizes beyond 1112, it was better to choose 16 processes among 4 machines (i.e., one process per CPU with 4 CPUs per cluster node) since the problem size became the main factor restricting our program's performance. If we allocated more processors, they were responsible for a smaller number of nodes which meant the average computation time would decrease. But it also introduced a larger overhead for MPI initiation and communication. In contrast, fewer processors reduced the above overhead, but each processor had to handle more nodes.

How did you partition your data? How did you partition the graph among the processes?

Data (i.e., nodes) were partitioned according to the number of processors. Each partition size was equal to the floor of the node count divided by the process count, except for the last partition which also included the remainder (if one existed). Each processor would load the entire graph using local copies of data_input_meta and data_input_link. See Description of Implementation for more details.

What communication mechanisms are used in your program? What is the advantage of your specific choices in terms of communication overhead and running time?

In order to share data between processes, we used 4 functions: *MPI_Allgather()*, *MPI_Allgatherv()*, *MPI_Bcast()*, and *MPI_Scatter()*. We used 2 communication processes at the start of the algorithm in order to properly use *MPI_Allgatherv* and since they are small communications they should not have a large impact on the overhead. In the do/while loop construct, we used 3 more communication processes. These allowed each thread to get the new data they needed to properly compute their new local ranks, which were then sent back to the main process to check relative error. These 3 calls to communication processes minimized the amount of data sent between threads which allowed for a minimum in communication overhead, which allowed for a faster calculation time for each rank in the PageRank algorithm.

Tables and figures

Table 1: Top average and median PageRank latency for serial, single machine, and cluster runs

	Serial (s)		**Single machine (np=4) (s)**		**Cluster (np=16) (s)**	
No. nodes	**Average**	**Median**	**Average**	**Median**	**Average**	**Median**
1112	7.75E-02	8.42E-02	2.53E-02	2.53E-02	3.30E-02	3.25E-02
5424	1.19E+00	1.19E+00	3.33E-01	3.21E-01	1.32E-01	1.33E-01
10000	3.94E+00	3.94E+00	9.11E-01	9.02E-01	3.00E-01	2.85E-01

Figure 1: Bar graph of top average PageRank latency for serial, single machine, and cluster runs

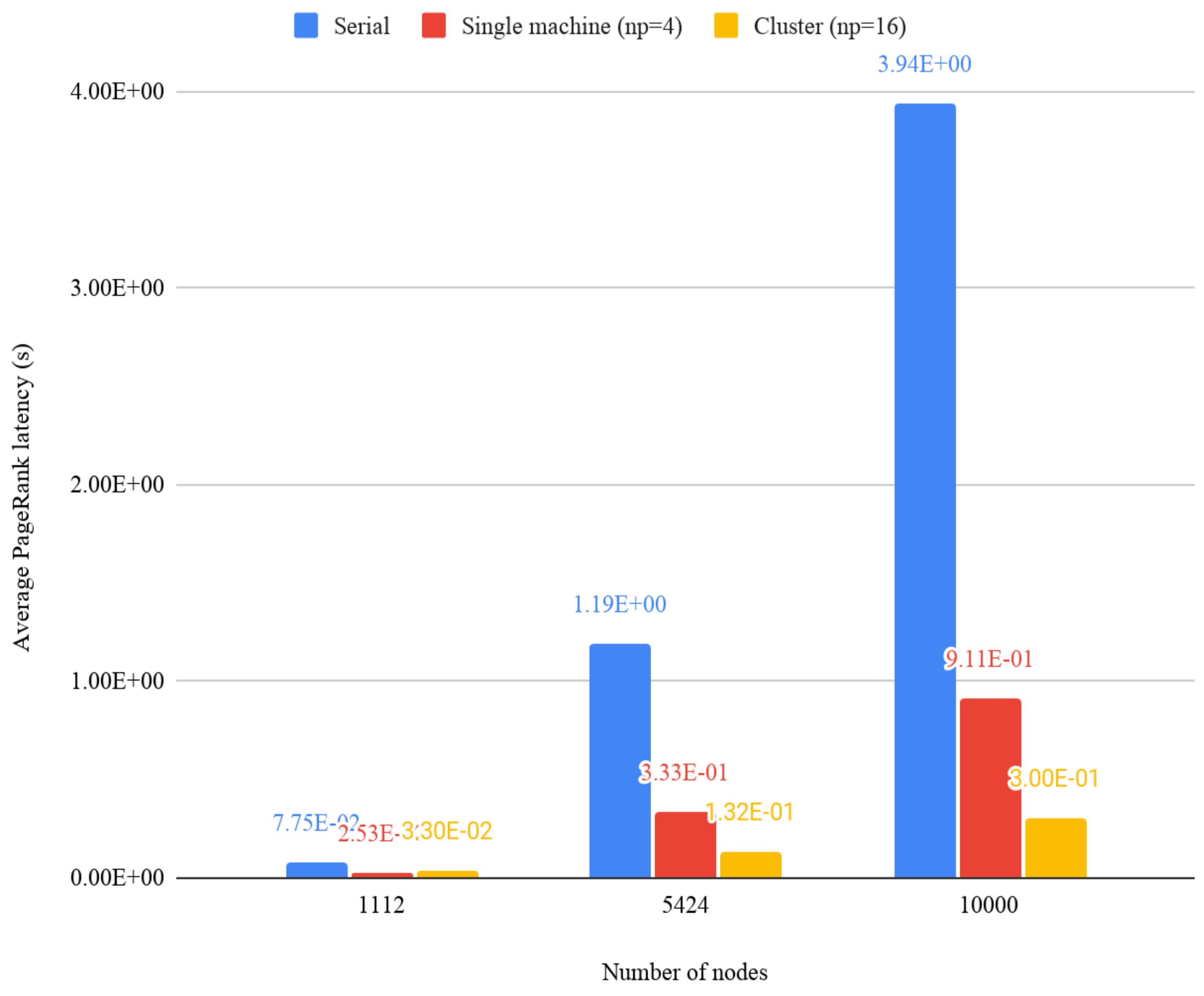


Table 2: Average single machine PageRank latency for increasing process count

	Average latency (s)		
Single machine no. processes	1112 nodes	5424 nodes	10000 nodes
2	4.92E-02	6.57E-01	1.79E+00
4	2.53E-02	3.33E-01	9.11E-01
8	1.42E+00	1.54E+00	3.57E+00
16	7.35E+00	8.25E+00	1.73E+01
20	1.14E+01	3.97E+01	3.88E+01

Figure 2: Bar graph of average single machine PageRank latency for increasing process count

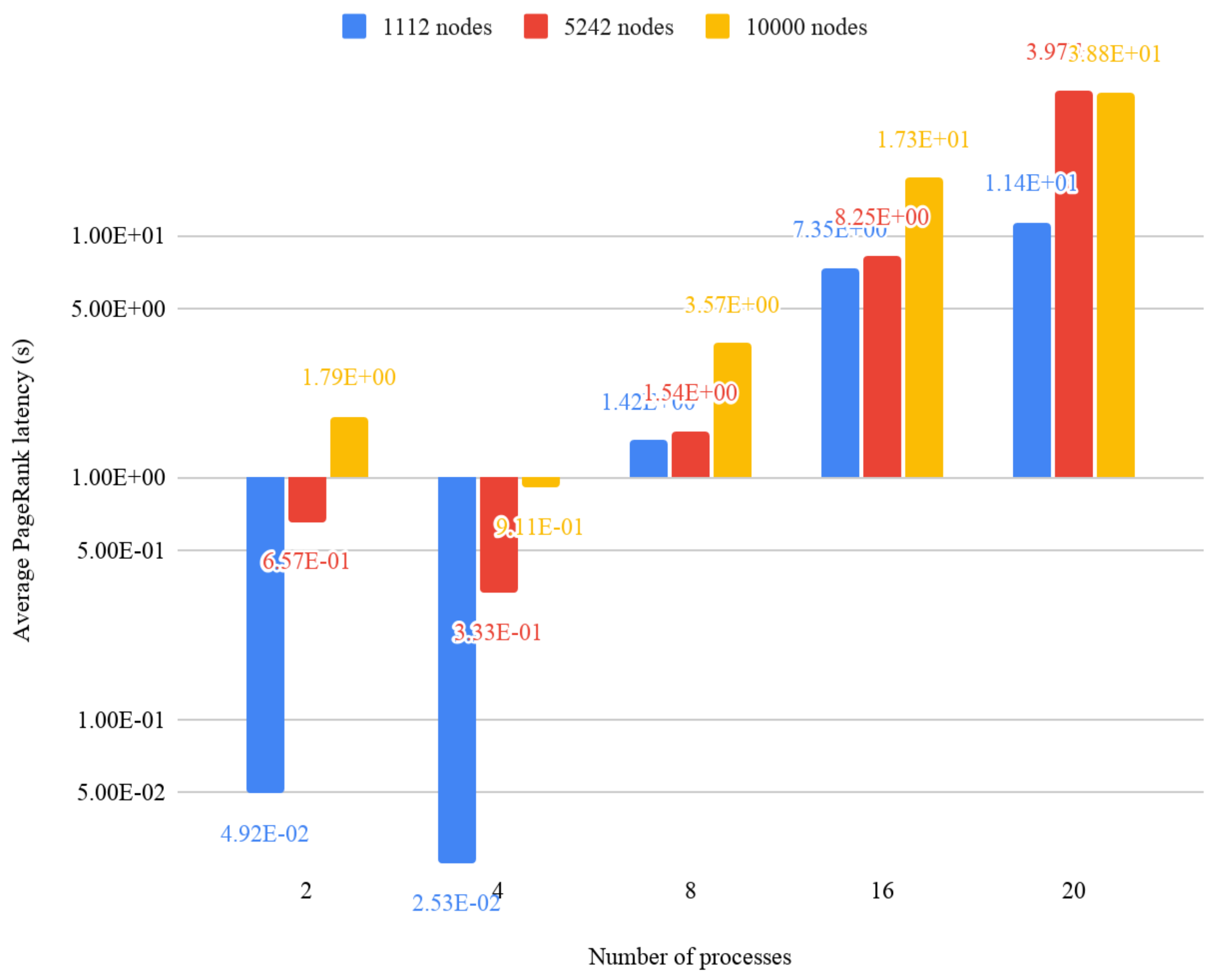
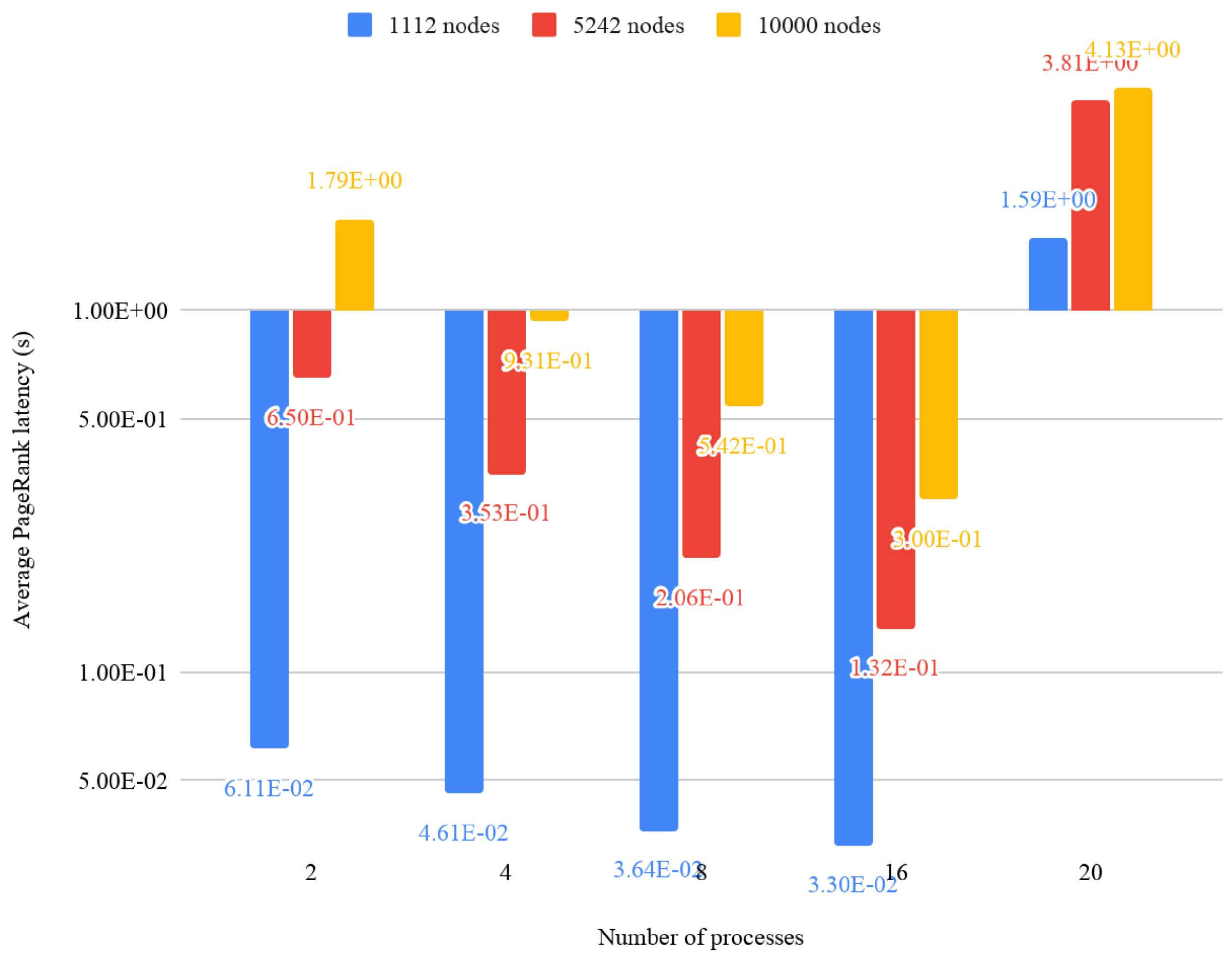


Table 3: Average cluster PageRank latency for increasing process count

	Average latency (s)		
Cluster no. processes	**1112 nodes**	**5424 nodes**	**10000 nodes**
2	6.11E-02	6.50E-01	1.79E+00
4	4.61E-02	3.53E-01	9.31E-01
8	3.64E-02	2.06E-01	5.42E-01
16	3.30E-02	1.32E-01	3.00E-01
20	1.59E+00	3.81E+00	4.13E+00

Figure 3: Bar graph of average cluster PageRank latency for increasing process count


1112 nodes
5242 nodes
10000 nodes
Average PageRank latency (s)
1.00E+00
5.00E-01
1.00E-01
5.00E-02
6.11E-02
6.50E-01
1.79E+00
4.61E-02
3.53E-01
9.31E-01
3.64E-02
2.06E-01
5.42E-01
3.30E-02
1.32E-01
3.00E-01
1.59E+00
3.81E+00
4.13E+00
Number of processes