ECE 420
Lab 4 Report

Cynthia Li, Michael Kwok, Samuel Lojpur

Description of Implementation

In order for the given PageRank algorithm to work, several variables must be set prior to starting. Firstly, node count is provided by user input to determine the number of nodes we must process in the algorithm. This value is used to initialize the nodes linked list (returns a node head), along with several node count length arrays: r, r_pre, and contribution. The r and contribution arrays have its values initialized to 1/node count and (1/node count) / (the number of outgoing links from the node * the damping factor) respectively. This damping factor is a constant set to 0.85, to ensure the algorithm will reach an end. Following this initial setup is the setup for MPI, where an iteration counter is initialized to 0, worker ranks and total works are initialized via MPI::COMM_WORLD.Get_rank() and MP::COMM_WORLD.Get_size(). Lastly, the work to be done on each local worker is defined by splitting the number of nodes to process (calculate node counts for each node) and calculate the displacement of where the local workers' nodes to process start in the node linked list.

Each thread is given its own number of nodes to handle, divided as evenly as possible. The number of these each thread has is stored as 'local_nodecount'. Each thread is given an 'r' and 'contribution' array with length equal to its local_nodecount. Each local contribution is initialized to equal the corresponding section of the global contribution array,

The core calculation is done within a do/while loop. The r array is copied to an 'r_pre' value, to be compared against later. Each thread takes a number of nodes equal to its local_nodecount and, for each node, adds up the contributions of each node that is linked to that node, using the global 'contributions' array, along with a damping constant. This can be thought of as the website's popularity. These form the new local r array. Next the local_contribution arrays are modified to be equal to the local_r (times a damping factor) for the node divided by the number of links out from the node. This can be thought of as the website's links to other websites, where each site it links to getting an even share of the site's popularity (with a 15% chance that the user stays on the page).

Once the contributions are calculated, EACH thread sends ALL of its updates to r and contributions to EACH other thread. R is required to be updated globally because the program uses the difference between each r and the previous r to determine whether to terminate the loop, and each thread uses the contributions of all nodes linked to a node in order to calculate its own local r. These linked nodes could be outside of the thread's set partition, so it's important that they are updated each loop. This sending is performed using the MPI::Allgatherv command. The allgatherv command uses a 'displacements' array, which contains a 'start point' for each thread to put its array so they all fit together into the full array.

Finally, the r and r_pre are compared, and if the difference is small enough, the loop terminates. The program is timed from before starting the loop to just after the loop ends. Upon completion, all dynamically allocated memory (for the linked list of nodes as well as the calculation arrays) is freed. The result array r and the time it took to run is saved by the worker with rank 0 before exiting.

Performance Discussion

For each of the given problem sizes: 1112, 5424, and 10000 nodes; the program was run with a different number of threads: 1, 2, 3, 4, 5, 10, 20. For each of these cases, an average runtime was

calculated through 10 runs. It is clear to see that increasing the number of nodes, or the problem size, will directly increase the length of time required to run the algorithm. Increasing the number of processes concurrently processing the algorithm will decrease the running time, since a single process must do less work. However,this only applies up to a certain point. Past this, the overhead of needing to gather the information for every process exceeds the speedup of parallelism, and causes an increase in the average time taken to run the program.

Processes\Nodes	1112 nodes	5424 nodes	10000 nodes
1	0.02037250	0.34540440	1.11997890
2	0.02093000	0.19434540	0.57418340
4	0.02217860	0.11918140	0.31354360
5	0.02853690	0.13356320	0.28108020
10	0.03163020	0.11574850	0.19701300
20	2.60666470	2.93578150	3.47433260

Table 1: The averaged run times of 10 runs for each thread/node count, distributed on multiple machines

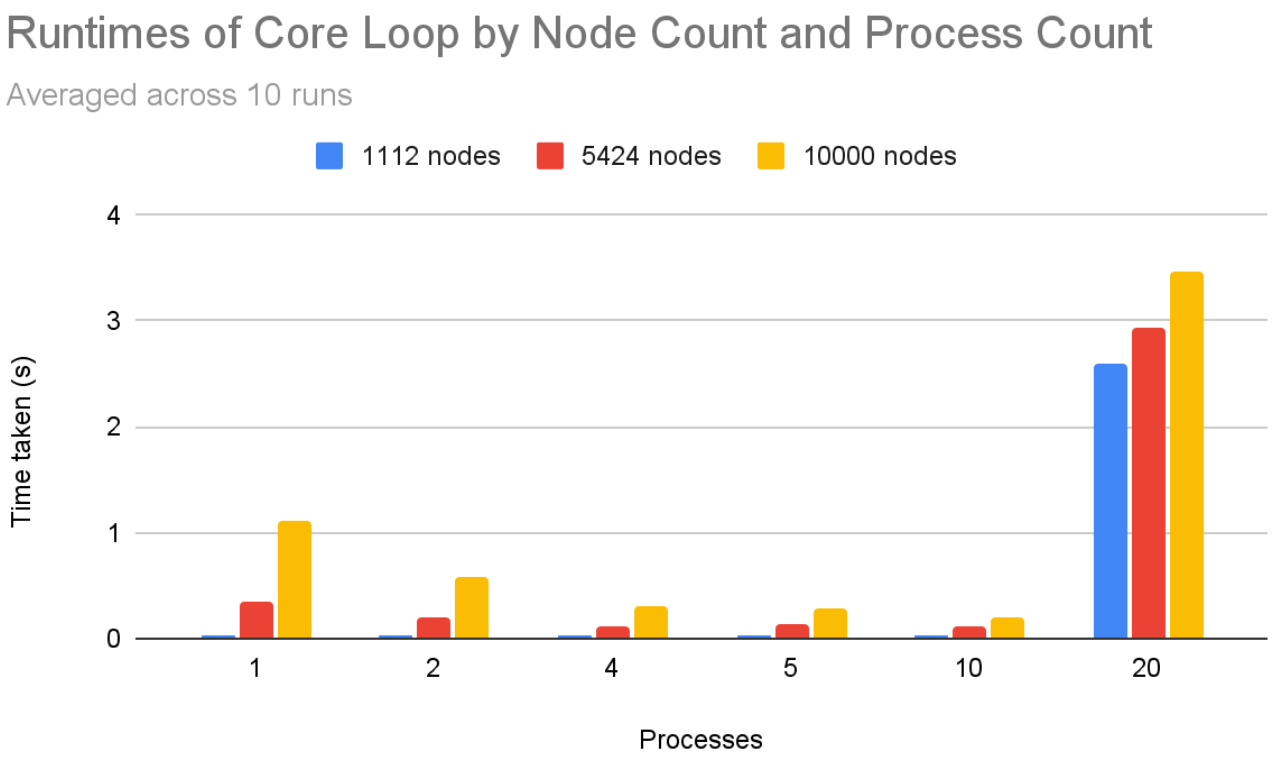

Figure 1: The average run times of different number of threads on different problem sizes (node counts) distributed on multiple machines

Processes	1112 nodes	5424 nodes	10000 nodes
1	0.0222116	0.3483461	1.1404626
2	0.0155747	0.1843806	0.5798416
4	0.0083424	0.0937707	0.2892722
5	1.6715584	1.9460368	2.2215384
10	5.3381757	5.8560476	7.0628963
20	16.2209334	17.2301067	19.2931867

Table 2: The averaged run times of 10 runs for each thread/node count, distributed on threads on one machine

When only a few processes were running, performance of our program was better on a multiple machine setup, due to use of more processes and thus more processing power to compute its share of the algorithm. As the number of processes grows beyond the number of machines, these benefits diminish, but the overhead of machine-machine communication remains and grows, making single machine setups run better for 10 and 20 process jobs.

The best number of processes that should be used in our program corresponding to the different problem sizes is 1 process for the smallest case 1112 nodes, and 10 processes for the larger 5424 and 10000 nodes. It is likely that the smallest problem takes so little time to run that the overhead introduced by communication slows it down considerably more than the benefit of additional processors. However, in other cases, this additional computing power helps more than the computing overhead hurts.

Before we run the program we place a copy of the data (data_input_link and data_input_meta) on each machine. When processes run, they load all of this data in, but they are each assigned a range within it to manage. The graph was partitioned into mostly equal segments among the processes. If the number of nodes to process was perfectly divisible among the number of processes, then every process gets the same number of nodes to process. Otherwise if the number of nodes was not evenly divisible among the number of processes, then the remaining nodes are distributed to threads one by one until there are no more nodes left.

The communication mechanism used in our program is message passing interface (MPI). We ended up using the Allgatherv function for all of our messages. Since each process should update each other process on the new node contributions, and the new R, we need Allgather, so all the data can be eventually shared to all nodes. Running a gather followed by a scatter would be much slower, since all data would need to go through the bottleneck of a single process, rather than be spread evenly by all. We use allgatherv since the amount of data may not be divisible by the number of processes. Allgatherv allows us to send different amounts of data from different processes, without padding any of them with zeros, which keeps our packet size lower than it otherwise would be. These choices help to keep our running time low.