ECE 420 - Section H42
Lab 4: Parallelizing Page Rank using MPI
Due Date: April 7th, 2023

Emily Li (1375932), Hilton Truong (1615505), and Stephine Yearley (1623617)

Description of Implementation

Our implementation of parallel page rank is based off the provided serial version but has the computational work at each iteration shared across multiple processes. The code can be found in main.c and all parts of the implementation (both work division, computation, and synchronization) are handled in the main() function.

The main function first loads in the data from the "data_input_meta" file and initializes MPI. Using calls to *MPI_Comm_size*() and *MPI_Comm_rank*(), we get the number of processes and the rank of each process. With this information, each process can determine which nodes it must calculate the page rank score for. If the number of nodes, n, does not divide perfectly by the number of processes, *p*, then the first *p - 1* processes get assigned *ceil(n/p)* nodes, with the last process being assigned the remaining nodes. If *n % p = 0*, then each process gets assigned an identical number of nodes.

After all processes have the chance to allocate memory for the global arrays *r*[] (the page rank result for each node) and *contribution*[] (the contribution of each node), the core calculation can begin. This initial synchronization is achieved through a call to *MPI_Barrier*(). The core calculation is timed by the process with rank 0 (the root process).

The core calculation involves first preserving a copy of the previous iteration's result vector (so that it can be used later to determine if the relative error of our result is small enough to stop). Then, each process updates its portion of *r*[], *my_r*[], in a for loop. After this, each process updates its portion of *contribution*[], *my_contribution*[]. A call to *MPI_Gather*() updates *r*[] for the root process, and a call to *MPI_AllGather*() updates *contribution*[] for all processes. This also forces synchronization between iterations which is required to obtain correct values.

At the end of each iteration, the root process determines the relative error between the current and previous iteration. It then updates and broadcasts *cont* (the core calculation looping condition flag) accordingly, using a call to *MPI_Bcast*(). Once the relative error is small enough, the looping condition is set to false. The root process calculates the runtime of the core calculation and saves this time along with the result, *r*[], to a file.

Our choice in communications mechanisms were informed by simplicity and speed. Choosing blocking communication methods instead of non-blocking methods allowed us to keep the program simpler and more readable. This does have the disadvantage of not being able to process data while communication is queued, meaning we miss out on potential speedup in the name of simplicity. Using *MPI_Gather()* when possible instead of *MPI_AllGather()* helped us reduce communication overhead by updating global variables for only the root process. The root process then communicated to the other processes through faster means such as *MPI_Bcast()* of a flag.

Performance Discussion

The average runtime (over 10 runs) and related performance statistics about speedup and efficiency for a single machine and a cluster of 4 machines can be found in Tables S.1 and C.1, respectively. These tables provide characteristic performance data for our parallelized page rank algorithm for various problem sizes (num nodes) and process counts (num processes). Runtime data from these two tables was plotted using bar graphs in Figures S.2 and C.2.

First examining the speedup section for the single machine tests in Table S.1, we found that the optimal number of processes is 4 for all problem sizes (1112, 5424, and 10000 nodes). Looking at Figure S.1, we see that running time trends follow a curve, initially decreasing from the single process runtime, then eventually increasing once the granularity becomes too fined grained and inter-process communication and synchronization overheads begin to outweigh the benefits of parallel computation. These effects are most clearly seen for smaller problem sizes since they require fewer calculations and therefore benefit less from dividing the computational workload across multiple processes.

Examining the effects of process count on the cluster of machines yields similar but not identical results. In Table C.1, we see that for the smaller problem size of 1112 nodes, 2 processes run fastest, while for the larger problem sizes of 5424 and 10000 nodes, 5 processes provide the best speed up. In Figure C.2, we see the same trend of initially decreasing runtimes reaching a minimum at optimal granularity, followed by increasing runtimes when overhead becomes more prominent. The reason a smaller number of processes may run faster for small data on a cluster of machines is that communications overhead is even greater on multi-machine systems with distributed memory compared to single-machine shared memory systems.

Comparing between the cluster and single machine trials, we can see that for the smallest problem size of 1112 nodes, the single machine can provide a speedup of 3.15 whereas the cluster can provide a speedup of only 1.33. This can be attributed to the fact that a cluster has greater communication overheads than a single machine. For 5424 nodes, the single machine at optimal granularity provided a speedup of 3.39, while the cluster performed better, providing a speedup of 3.60. For 10000 nodes, the cluster once again outperformed the single machine delivering a speedup of 4.23 compared to 3.71. A clear trend is visible here: as problem size increases, we are able to more fully exploit the computational power provided by the cluster. For smaller data sets, the communication overhead between processes on different machines dominates and we no longer see the benefits of cluster computing.

Single Machine Performance Data								
	Num Processes:	**1**	**2**	**3**	**4**	**5**	**7**	**15**
Runtime (s) by num of nodes	**1112**	0.063	0.030	0.024	0.020	1.326	2.033	10.544
	5424	1.104	0.564	0.405	0.326	1.431	2.287	10.879
	10000	3.572	1.798	1.256	0.964	2.636	3.005	12.447
Speedup by num nodes	**1112**	1.00	2.07	2.62	3.15	0.05	0.03	0.01
	5424	1.00	1.96	2.73	3.39	0.77	0.48	0.10
	10000	1.00	1.99	2.84	3.71	1.35	1.19	0.29
Efficiency by num nodes	**1112**	1.00	*1.04	0.87	0.79	0.01	0.00	0.00
	5424	1.00	0.98	0.91	0.85	0.15	0.07	0.01
	10000	1.00	0.99	0.95	0.93	0.27	0.17	0.02

Table S.1. Performance statistics of parallelized page rank algorithm running on single machine for various problem sizes and number of processes

*While an efficiency value greater than 1 is theoretically impossible, since these results are based of averages of multiple runs on different randomly generated data sets, it is possible that the 2-process 1112-node test got "easier" data to process for its trails. We can assume that the actual efficiency is slightly less than 1.

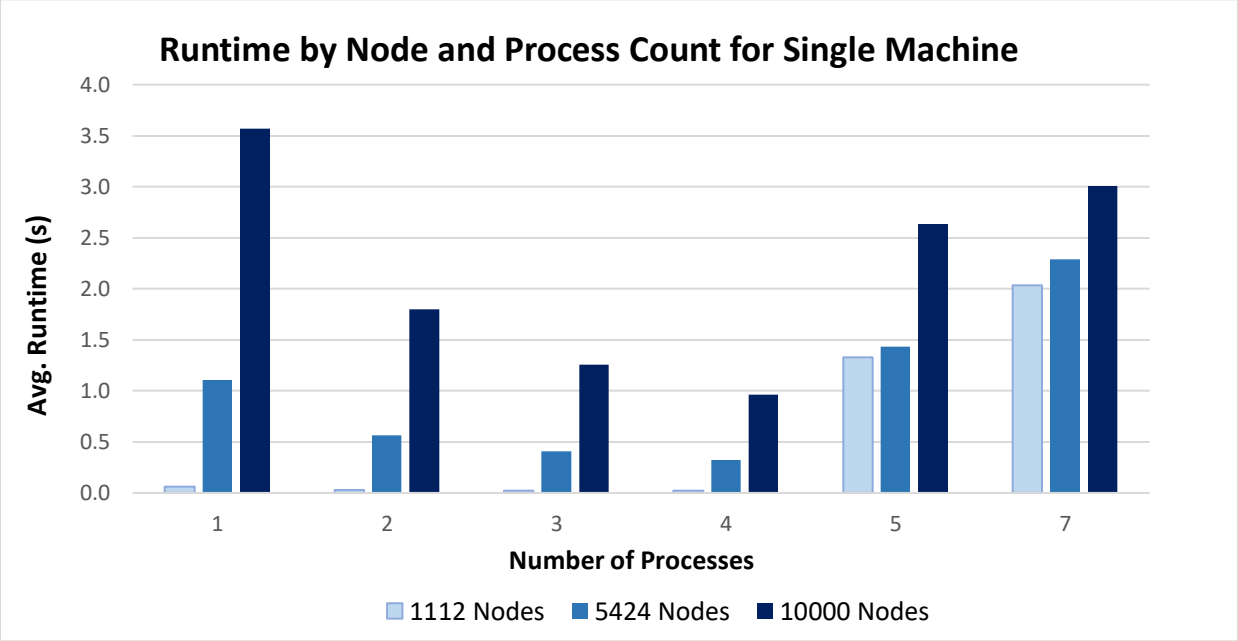

Figure S.2. Average runtime for parallelized page rank algorithm running on single machine for various problem sizes and number of processes.

4 Computer Cluster Performance Data								
	Num Processes:	**1**	**2**	**4**	**5**	**6**	**7**	**15**
Runtime (s) by num nodes	**1112**	0.065	0.049	0.052	0.080	0.134	0.810	1.618
	5424	1.074	0.604	0.373	0.298	0.343	0.637	2.351
	10000	3.569	1.864	0.998	0.844	0.858	0.954	2.945
Speedup by num nodes	**1112**	1.00	1.33	1.26	0.82	0.49	0.08	0.04
	5424	1.00	1.78	2.88	3.60	3.13	1.69	0.46
	10000	1.00	1.91	3.58	4.23	4.16	3.74	1.21
Efficiency by num nodes	**1112**	1.00	0.67	0.32	0.16	0.08	0.01	0.00
	5424	1.00	0.89	0.72	0.72	0.52	0.24	0.03
	10000	1.00	0.96	0.89	0.85	0.69	0.53	0.08

Table C.1. Performance statistics of parallelized page rank algorithm running on a cluster of 4 machines for various problem sizes and number of processes

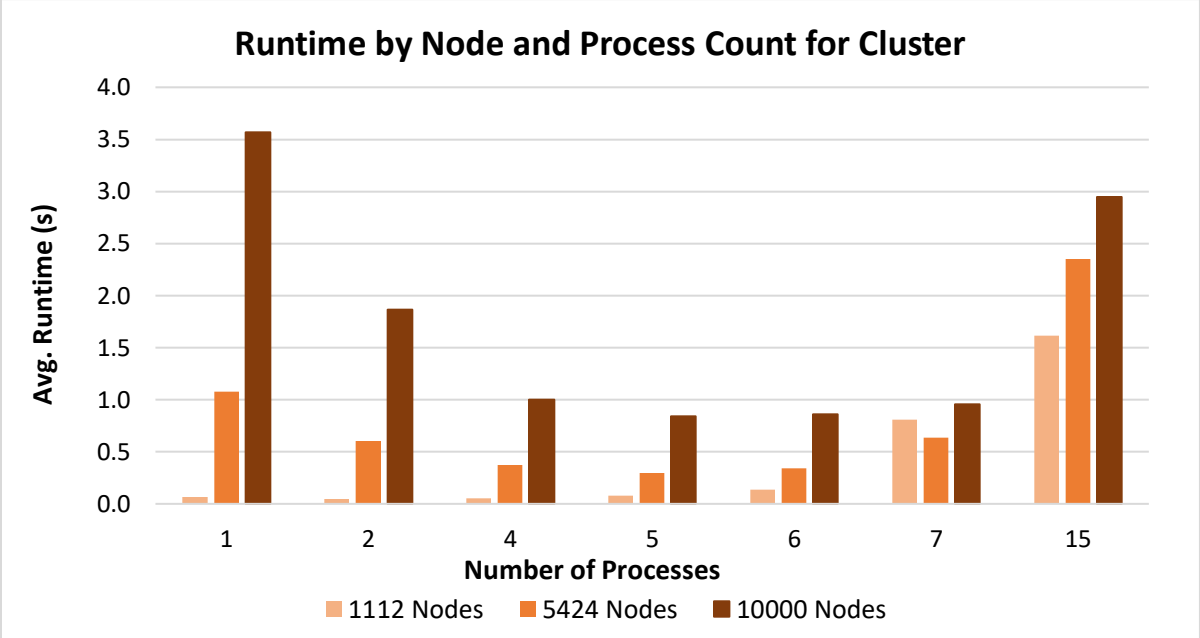

Figure C.2. Average runtime for parallelized page rank algorithm running on a cluster of 4 machines for various problem sizes and number of processes.