ECE 420 Lab 4 Report

Authors: Noah Batiuk, Steven Jiao, Justin Boileau

Description of Implementation

Objective of Lab

In this lab, we implemented a distributed routine to calculate page rank values for a list of input nodes, each representing a webpage.

Implementation (main.c)

Distribution Strategy

Our program distributes the page rank calculation for n nodes across m machines as follows: $nodes_i = \frac{n}{m},\ i \in [0,\ m)$ where $nodes_i$ is the number of nodes for the process running on machine i.

Setup

Our program runs on multiple machines in a distributed fashion, using the Message Passing Interface (MPI). We copy the executable to each worker machine and denote each worker IP in the “hosts” file on the master machine. Next we use “mpirun” on the master machine to start the program on it and all workers.

Program

For this explanation, assume our program runs on 4 total machines, including a single master machine and zero to 3 worker machines, via the setup procedure above.

First, the program runs a startup sequence that begins with the initialization of MPI and fetching the total number of processes and rank of the current process. Next, the input file is read and used to initialize a webpage graph, the total nodes per process, and the assigned nodes for each process. Next, three rank vectors are initialized: the global rank vector for the current iteration $r_i(t+1)$, the global rank vector for the previous iteration $r_i(t)$, and the local rank vector $r_{chunk}(t+1)$ for only the nodes assigned to the current process. After this, two vectors are initialized which will hold the damped contribution value D_i to each node from its incoming neighbors, as well as D_{chunk} which holds just the update contribution values calculated with the data from the chunk. Lastly, the initial values for the rank vectors are calculated and the startup sequence is complete.

Next, the program starts tracking its runtime and begins a cycle of: copying the current rank vector into the previous rank vector, computing the next rank update for the chunk

of nodes assigned to the current process, consolidating the new rank values into the root process and updating the contribution values to all other processes, and computing the relative error in this new rank value from the true value. These steps are repeated until the error is less than the epsilon value defined in the manual, $1x10E\text{-}5$. Once this error value is achieved, a broadcast from the root process tells other processes to finish computation and the program finishes timing and the master node (process rank 0) will save an output file containing the page rank values of each node.

Performance Discussion

Results

We were unable to obtain data for running the PageRank algorithm using 32 processes on a single computer for 10000 nodes as this caused the cluster to crash. Thus, we do not have quantitative data for this point.

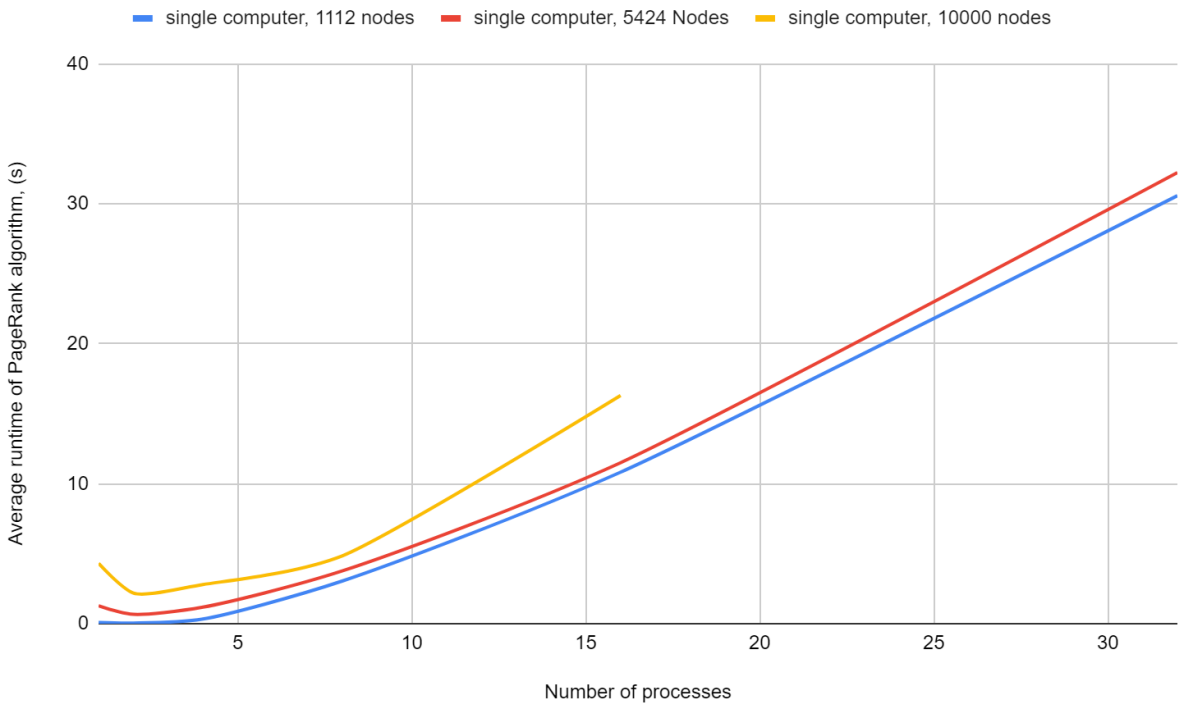

Figure 1. The average runtimes of the PageRank algorithm for single computers of varying problem sizes from 1112 to 10000 nodes.

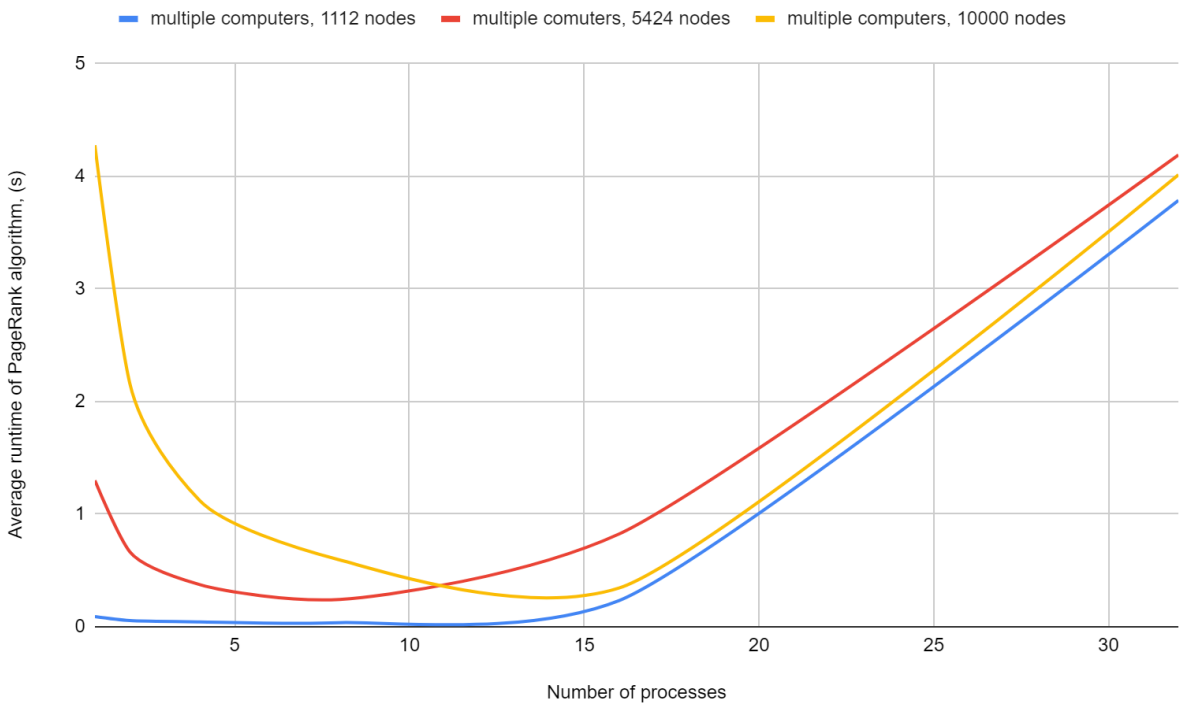

Figure 2. The average runtimes of the Page Rank algorithm for multiple computers of varying problem sizes from 1112 to 10000 nodes.

run	1 process	2 processes	4 processes	8 processes	16 processes	32 processes
1	0.092864	0.04963	0.199651	3.142445	10.31401	31.093575
2	0.084069	0.043449	0.175996	3.228657	10.266462	30.401636
3	0.079508	0.047634	0.92548	3.344013	12.317575	30.503987
4	0.092142	0.047043	0.028183	2.845159	10.551059	31.254123
5	0.091914	0.046797	0.044415	3.253988	11.397863	28.900678
6	0.075902	0.04774	0.027989	2.940235	10.984941	32.563412
7	0.082629	0.047685	0.797431	3.076419	10.829061	27.450987
8	0.078746	0.047499	0.786281	2.879856	10.306241	33.786532
9	0.080297	0.047787	0.114955	3.022407	10.489988	30.500345
10	0.076426	0.047772	0.222161	2.731604	10.807986	29.321908
average	0.0834	0.0473	0.3323	3.0465	10.8265	30.5777

Table 1. Testing data on a single computer with 1112 nodes

run	1 process	2 processes	4 processes	8 processes	16 processes	32 processes
1	0.092938	0.056907	0.046661	0.036056	0.12368	3.659896
2	0.092862	0.0505	0.04475	0.038356	0.376754	4.251435
3	0.075734	0.056958	0.043936	0.03691	0.078293	3.796995
4	0.092828	0.056843	0.044345	0.037987	0.390955	3.975906
5	0.09349	0.056666	0.044294	0.037918	0.085548	3.534801
6	0.092938	0.056584	0.044559	0.038466	0.140659	3.785586
7	0.093685	0.05739	0.044921	0.038027	0.078944	4.025882
8	0.092941	0.051072	0.044899	0.03864	0.120594	3.150322
9	0.09296	0.056695	0.045105	0.040173	0.216436	3.675538
10	0.092729	0.060627	0.045247	0.038986	0.731019	3.993495
average	0.0913	0.0560	0.0449	0.0382	0.2343	3.7850

Table 2. Testing data on multiple computers with 1112 nodes.

run	1 process	2 processes	4 processes	8 processes	16 processes	32 processes
1	1.280899	0.647074	1.157287	3.73768	11.728287	30.57619
2	1.281405	0.74009	1.473141	4.119008	11.507988	35.219834
3	1.279689	0.645387	0.972625	4.076684	12.25842	31.51423
4	1.279988	0.64488	0.647872	4.183878	11.234532	28.654321
5	1.279884	0.643946	1.330989	3.925468	12.543545	32.987654
6	1.27952	0.673155	1.505763	3.452233	11.059347	31.345678
7	1.285076	0.685218	0.896689	3.931916	11.354345	29.765432
8	1.282549	0.657237	1.297114	3.34298	11.276378	34.123456
9	1.280005	0.768432	1.127465	3.406111	10.366744	33.901234
10	1.334508	0.645651	1.420672	3.524029	11.666543	34.145075
average	1.2864	0.6751	1.1830	3.7700	11.4996	32.2233

Table 3. Testing data on a single computer with 5424 nodes.

run	1 process	2 processes	4 processes	8 processes	16 processes	32 processes
1	1.28092	0.664211	0.356261	0.248545	1.328858	4.382466
2	1.279471	0.658692	0.352286	0.262245	1.071535	3.729963
3	1.317325	0.659132	0.351501	0.256258	0.523241	4.112254
4	1.400001	0.657748	0.40947	0.196978	0.709745	4.375499
5	1.318356	0.67256	0.350764	0.249735	0.84423	4.064786
6	1.279029	0.657566	0.376521	0.242449	0.673229	4.244766
7	1.282627	0.703727	0.369755	0.198827	0.574317	4.598236
8	1.281314	0.658433	0.418116	0.271155	0.94808	3.930746
9	1.280207	0.67008	0.415015	0.26733	0.625113	4.426179
10	1.280762	0.657648	0.350276	0.246708	0.954322	4.016336
average	1.3000	0.6660	0.3750	0.2440	0.8253	4.1881

Table 4. Testing data on multiple computers with 5424 nodes.

run	1 process	2 processes	4 processes	8 processes	16 processes
1	4.271649	2.244307	3.096418	4.233747	26.195464
2	4.275106	2.145128	3.083182	5.081217	12.050896
3	4.370949	2.143219	2.405396	4.284739	14.732236
4	4.386684	2.18782	2.540049	4.631239	12.640431
5	4.282608	2.233442	3.524451	4.902926	18.235412
6	4.278072	2.250968	3.085695	5.248326	14.756894
7	4.289273	2.22551	2.771158	5.36331	16.678901
8	4.287891	2.1597	2.688071	4.524487	13.450983
9	4.288106	2.153017	2.423837	5.187483	17.986539
10	4.365712	2.143443	2.377364	5.015669	16.300345
average	4.3096	2.1887	2.7996	4.8473	16.3028

Table 5. Testing data on a single computer with 10000 nodes.
Data for 32 processes weren't able to be obtained as it exceeds the amount of resources the cluster can utilize.

run	1 process	2 processes	4 processes	8 processes	16 processes	32 processes
1	4.316025	2.159046	1.138882	0.62699	0.329957	4.393191
2	4.283734	2.152554	1.108578	0.575495	0.392433	4.026904
3	4.267379	2.150996	1.147538	0.574174	0.323904	3.626984
4	4.265823	2.152228	1.107421	0.573761	0.350979	3.895602
5	4.27716	2.153309	1.107081	0.577233	0.343226	4.418056
6	4.265744	2.153689	1.105674	0.574633	0.324946	4.16794
7	4.264427	2.195165	1.107012	0.583323	0.360638	3.945246
8	4.273969	2.156325	1.134598	0.617037	0.345424	3.757876
9	4.265653	2.158166	1.107359	0.662691	0.328115	3.858157
10	4.265652	2.161762	1.143093	0.578188	0.351517	4.035103
average	4.2746	2.1593	1.1207	0.5944	0.3451	4.0125

Table 6. Testing data on multiple computers with 10000 nodes.

Discussion

As we can see, using multiple machines for computation gives just as good results as using a single machine for a smaller number of processes, or much better when the problem sizes grow and the number of processes increases. The reason is because the work can be distributed across multiple machines as opposed to being assigned to a single machine, and thus there is a larger pool of resources that can be used where the work can be split and worked on asynchronously.

For the problem size of 1112 nodes, the greatest speedup achieved for was $s = \frac{0.0834s}{0.0382s} = 2.1818$ for using multiple computers at 8 processes. For 5424 nodes, it was $s = \frac{1.2864s}{0.2440s} = 5.2716$ also using multiple computers at 8 processes, and for 10000 nodes, it was $s = \frac{4.3096ss}{033451s} = 12.4875$ using multiple computers at 16 processes. We can see that for smaller problem sizes, a smaller number of processes spread across multiple computers does better than using more processes as the overhead of communication increases. However, we can see that with a very large problem size of 10000 nodes, the cost of overhead is diminished by the amount of work that can be done by spreading the workload over more processes on more computers and thus we see the greatest speedup at 16 processes. However, if we increase the number of nodes to 32 processes for 10000 nodes on multiple machines, we can see that the average runtime drops significantly to the same as processing 10000 nodes on a single computer with a single process, or just serially. In terms of granularity of our program, we can see that increasing the number of processes doesn't always decrease the runtime as the more processes we spawn, the more communication overhead there is and there requires a balancing of the number of processes to be used for optimizing this problem.

For partitioning the graph and splitting the workload, we chose to split the problem size equally amongst the number of processes we are given which was $nodes\ per\ process\ = \frac{nodecount}{processes}$. Each process would have access to a global D_i that is always updated after every iteration by all processes, and a global $r_i(t+1)$ and $r_i(t)$ to update with their own chunks of processed data. For each process, they will also have a $nodes\ per\ process$-sized $r_{chunk}(t+1)$ and D_{chunk} array that is their own calculated portions of data to update the global arrays with.
Since only D_i is the only array that can have data shared between other processes, this is updated globally between all processes using MPI_Allgather(). For updating each process's $r_{chunk}(t+1)$ into $r_i(t+1)$, we only update the data structure in the master process 0 for calculating the relative error between $r_i(t+1)$ and $r_i(t)$ so a MPI_Gather() into the root process is called. The advantages of this is that we are able to tighten the communication overhead for updating $r_i(t+1)$ using MPI_Gather() instead of MPI_Allgather() such that each process has the necessary data to perform computation on their own chunks of data (even though the rest of each processes $r_i(t+1)$ is incorrect aside from their chunks). Lastly, a MPI_Bcast is called by the root process so that it can update other processes once a satisfactory relative error is calculated and the processes can end the program.