Dan Levy

My final project was to work on the page rank algorithm using MPI and a sparse matrix. The goal was to minimize communication by using sends() and receives() instead of allgathers(). I used the same web page "web" that was used for Assignment 3 (this did not require load balancing because it was designed to be evenly split up).

In order to use MPI for this task, the first step was to split up the sparse matrix and all other vectors. We have to do this because MPI uses the distributed memory model. This splitting step was not needed for the OpenMP implementation because it uses a shared memory model. In order to explain the process of splitting, let us assume that that there are eight pages and four processes being used. When splitting up the sparse rank matrix, we must realize that process 0 (the first process) only needs $\left(\frac{numPages}{numProcs} * 2\right) - 1 = 3$ elements, but that the other processes need $\left(\frac{numPages}{numProcs} * 2\right) = 4$ elements. This is due to the fact that row 0 only has 1 link, whereas all other rows have 2 links (this can be seen by looking at the dense rank matrix). The other trick is getting the oddball link in the correct location (the oddball link being 1.0). Regardless of the number of pages and the number of processes, the 1.0 must be the third-to-last element in the sparse rank matrix. In this example of eight pages and four processes, it is easy to put it in the correct location because it fits into process three's sparse rank matrix. This becomes a problem when there are, say, eight pages and eight processes (the 1.0 will have to go in the second-to-last process' sparse rank matrix). When splitting up the column index vector, we must take into account the fact that process zero has a column index vector with length one less than all other processes. The other trick with the column index vector is computing the correct indices. This can be done by getting the equivalent dense rank matrix's diagonal column index using the equation $\left(\frac{numPages}{numProcs} * myID\right) + currentRowIndex$. We can then get the correct indices to the left and right of the diagonal. There are two edge cases, however. The first is that row zero only has a link to the right of the diagonal. The second edge case is that the last row (row 7 for this example) has a link at column 0 and column numPages – 2 (column 6 for this example). Splitting up the row index vector is not quite as straightforward. Each process needs a row index vector that is indexes into its local sparse matrix. In our example of 8 pages, row 8 of the "global" row index vector has an index of 15. None of the processes have a "local" sparse rank matrix of

length greater than 4, so we need to accommodate. The first process starts with a row index of 0 (the first dense row), then has a row index of 1 (the second dense row), and then increments by 2 (3, 5, 7, …). For our example, process 0 only need row indices 0, 1, and 3. Even though process 0 only computes rows 0 and 1 of the dense rank matrix, we need row 3 as a "pseudo row" in order to tell the matvec loop to stop. Notice that we end up with odd numbers (1, 3, 5, 7, …) except for row 0. The rest of the processes (processes 1, 2, and 3 in our example) start with a row index of 0 and continue incrementing by 2 (2 links per dense rank matrix row). Notice that we end up with even numbers (0, 2, 4, …). This turns out to be an easy computation because, in our example, process 0 is hard-coded with a 0 and a 1, and then increments by 2, and processes 1 through 3 all start with 0 and increment by 2. The Y (page rank) vector can be split up into $\frac{numPages}{numProcs}$ elements per process (in this case, 2 elements per process because each process computes 2 pages). The X (multiply) vector is the exception to the rule here. Because of the complications that arise when splitting up the X vector, we give each process a "global" X vector.

The second step is what makes this problem so interesting. Rather than using the MPIAllgather() function in MPI, we use MPISend() and MPIRecv() functions in order to minimize communication between processes. This is due to the fact that the processes do not use every element in each of their X vectors in order to do their part of the page rank computation. Sticking with our example of 8 pages and 4 processes, we come to realize that most processes only need the two page ranks that they compute, as well as the page rank of the page just before it and just after it. The two edge cases here are with process 0 and process numPages – 1 (process 3 in our example). Process 0 only needs to get the page rank of the page after it. Process 3 needs the page rank of page 0 and the page just before it. In this case, most processes only call 2 sends and 2 receives. If we were to use MPI_Allgather(), we would essentially be doing 2 sends and 6 receives per process (in our example). The trick here is making sure that the we send off the right page ranks and put the received page ranks in the correct index locations in the X vector.

Because we have the luxury of using the same web of pages as we used in assignment 3, we can compare speeds of the sparse matrix using OpenMP to that of the sparse matrix using MPI. Note that we include speedup studies for OpenMP, the MPISend/Recv() code, as well as the MPIAllgather() version of our code. As is shown in the speedup studies below, all 3 versions of our code had terrible speedups when using only 16 pages. This is simply due the fact that the

required overhead to spin up multiple processes is not worth it. When moving to 1600 pages, however, the results are quite different. While none of the programs had great speedups with 16 processes, we did see decent speedups with the MPISend/Recv() version of our code. OpenMP performed poorly, which was likely due to the fact that the process creation overhead was too high. We have to remember that OpenMP uses a shared memory model, so all processes have access to the global data (no communication needed). The MPIAllgather() code saw very little speedup. This is likely due to the communication overhead. Interestingly, though, it had much better speedups than the OpenMP code (something that we didn't expect).

With this data in mind, let us take a look at the pros and cons of programming with each of the three methods we used. OpenMP is quite easy to use because it uses the shared memory model (no need to split up data). We also don't need to deal with the communication overhead. There are potentially some major scalability problems with this model, however. 1600 pages doesn't come *anywhere* near the number of actual web pages on the WWW. If we wanted to be able to compute the page rank of every page on the WWW, we probably wouldn't consider using shared memory. With that in mind, let us take a look at the coding challenges associated with MPIAllgather() and MPISend/Recv() code. Using MPIAllgather() is easier because it is one command that handles potentially many sends and receives. The problem with MPIAllgather(), however, is that we might not need every single value to be moved around. While sends and receives communicate just the right amount of information, they are significantly more complex to code. In the end, the extra coding effort is most definitely worth it for this type of program, especially if you start to compute very large numbers of page ranks.

16 Pages

OpenMP Sparse Speedup

Procs	Runtime (sec.)	Speedup
1	0.001707	1.000000
2	0.002311	0.738641
4	0.006632	0.257388
8	0.081032	0.021066
16	0.009028	0.189078

MPI Allgather Sparse Speedup

Procs	Runtime (sec.)	Speedup
1	0.001035	1.000000
2	0.003723	0.278002
4	0.003423	0.302366
8	0.015487	0.066830
16	4.587890	0.000226

MPI Send/Recv Sparse Speedup

Procs	Runtime (sec.)	Speedup
1	0.000898	1.000000
2	0.002835	0.316755
4	0.012240	0.073366
8	0.014347	0.062591
16	14.734795	0.000061

1600 Pages

OpenMP Sparse Speedup

Procs	Runtime (sec.)	Speedup
1	0.081346	1.000000
2	0.086240	0.943251
4	0.145172	0.560342
8	0.147048	0.553194
16	0.134218	0.606074

MPI Allgather Sparse Speedup

Procs	Runtime (sec.)	Speedup
1	0.072373	1.000000
2	0.068496	1.056602
4	0.061127	1.183978
8	0.056276	1.286037
16	31.336738	0.002310

MPI Send/Recv Sparse Speedup

Procs	Runtime (sec.)	Speedup
1	0.073745	1.000000
2	0.053150	1.387488
4	0.036749	2.006721
8	0.016838	4.379677
16	2.824470	0.026109