Project III – OpenMP and MPI

CIS6930

Nathaniel Price

UFID: 1436-7117

Task 1 – PageRank in OpenMP

Development Environment

The code for task 1 was developed using C++ under Windows 8 using CodeBlocks. The code was compiled using MinGW with –fopenmp set in the other options in Codeblocks. The code was compiled using CodeBlocks IDE.

Implementation Methods

The PageRank algorithm was implemented using the ego-Facebook dataset. The graph data was treated as undirected and therefore each undirected link A to B was treated as two directed links from A to B and from B to A. The first step in the program is counting the number of links in the data file. The file is then read and the data is stored in two arrays of in links and out links. The stochastic matrix is initialized using values of zero and then a value of one is inserted for each link. The number of out links per node is calculated by summing over the rows. The stochastic matrix is then normalized by the number of out links and the value of d=0.85 is used to allow for random teleports in the case of spider traps. This is shown by Eq. (1).

$$M_{ij} = 0.85\left(M_{ij} / N_j^{out}\right) + 0.15 / N \tag{1}$$

After the stochastic matrix is initialized then a parallel region is spawned. The page rank array is initialized in a parallel for loop using 1/N where N is the number of nodes. The power iterations are also performed in parallel using a parallel for loop where different threads multiply different rows. In order to make the code as general as possible no specific block size and the rows are divided among the threads automatically. The convergence criteria are shown in Eq. (2)

$$\text{Break If} \begin{cases} \varepsilon \le 10^{-7} \text{ where } \varepsilon = \sqrt{\left(\mathbf{r}^{(i)} - \mathbf{r}^{(i-1)}\right)^2} & \text{Converged within tolerance} \\ i \ge 100 & \text{Maximum number of iterations exceeded.} \\ & \text{Increase number of iterations.} \end{cases} \tag{2}$$

Validation of Code

The performance of the code was checked using a toy data set. The graph data was based on an example from lecture using three nodes. Since the example from lecture used directed links an option was added to the code to treat the graph data as directed.

Performance & Conclusions

The performance was checked using the ego-Facebook dataset treating the graph data as undirected but also allowing for random teleports. The power iterations converged after 40 iterations with an error norm of 8.35e-8. The execution time was approximately 1.9 seconds. The performance was not compared to a serial version of the code. Since the graph data is treated as undirected there are no deadends or spidertraps in the graph data. However, in order to check the performance of the code using directed graph data with spider traps the calculations allow for random teleports as shown in Eq. (1).

Task 2 – Parallel Reducer in MPI

Development Environment

The code for task 2 was developed using C++ under Windows 8 using Microsoft Visual Studio Ultimate 2013. The MPI runtime libraries were obtained from the Microsoft High Performance Computing (HPC) Pack 2012 MS-MPI Redistributable Package. Visual Studio was configured by adding the MS HPI Pack include directory to additional include directories and the MS HPC Pack libraries to the additional library directory. In the linker/input section msmpi.lib was added to the additional dependencies section. The code was compiled using Visual Studio. The code was run from the command prompt using mpiexec.

Implementation Methods

An overview of the basic structure of the program is shown in Figure 1. The number of lines of keys in the file is read by all nodes. Node 0 then reads the file and stores the data in a 2-D array. Node 0 also calculates the minimum and maximum key values and broadcasts this information to the other nodes for use in later steps. The 2-D array of key value pairs is scattered from node 0 to the other nodes. The distribution of the keys is accomplished by creating a subarray data type (MPI_create_subarray) with the size equal to the number of the keys divided by the number of nodes. One block is sent to each node. If the number of keys is not equally divisible among the nodes then the block size is rounded down to the largest equally divisible size and the remaining keys are sent to the Nth processor (up to N-1 blocks).

In the first local reduction step, each node converts its 2-D array of key value pairs to a vector and the vector is sorted by key using the sort function. The sorted vector of key value pairs is then converted back to a 2-D array. Using a loop the values with the same key are added together and the results are stored in a new 2-D array that is equal in size or smaller than the original array depending on the number of local reductions. Each node now has a table of unique keys but it is possible that some keys are split across different nodes.

In the next step specific ranges of keys are sent to each node in a block distribution. Using the previously calculated minimum and maximum key value the range of keys is divided into N sets. If the range of keys is not equally divisible by the number of nodes then the block size is rounded down to the largest equally divisible value and any remaining keys are sent to the Nth node. The number of keys to be sent from each node depends on the number of local reductions in the previous step and therefore it is necessary to use an all-to-all personalized communication. Since the nodes do not know how many keys will be sent and how many keys will be received it is first necessary to use an all-to-all communication to share this information (send counts / receive counts) before performing the all-to-all personalized communication.

The second local reduction is identical to the previous local reduction. However, since now the keys are not split across nodes the key value pairs are reduced completely. In the final step a gather communication is used to gather the reduced tables on node 0. Since the size of the table on each node may be different it is necessary to use the MPI_Gatherv function instead of the MPI_Gather function. Before the keys can be gathered to node 0 it is first necessary to gather the counts to node 0 to allocate memory and set up the communication. After the key value pairs are gathered to node 0 the results are written to a comma separated variable file.

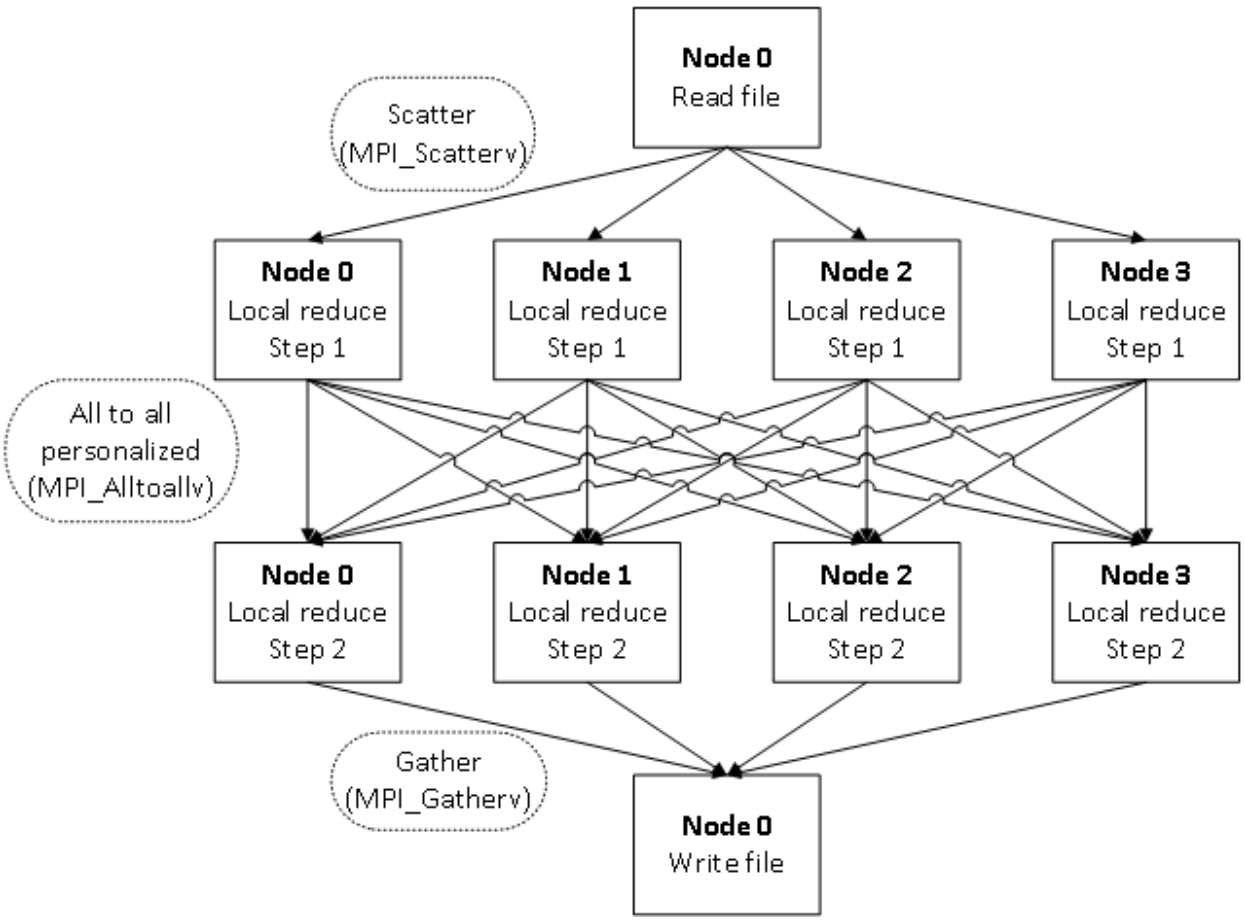

Figure 1 - Node 0 reads the file and scatters the data to the other nodes. The nodes sort and reduce the set of keys that was sent to them. Using an all-to-all personalized communication the keys are then distributed using a block distribution to all nodes. A second sorting and reduction is performed. The keys are gathered to node 0 to write the output file. (For visualization only)

Validation of Code

The performance of the code was checked using a toy data set. The data set consisted of 13 key value pairs with 6 unique keys ranging from 1 to 10. The code was checked and the solution was verified using 4, 5, and 8 nodes. It is possible to run the code with any number of nodes but when running with only 1 or 2 nodes some errors are encountered. These errors were not fully debugged since it was not a requirement for the code to run on a user specified number of nodes.

Performance & Conclusions

Throughout the program only bulk communications are used in order to improve the efficiency by reducing the communication overhead. The data is sorted and reduced locally which is more efficient than sorting the data on the master node before scattering the data. In addition, the code is scalable to large size datasets because the data is distributed over all nodes. For example, after the first reduction an all-to-all personalized communication is used to avoid gathering all the data to one node before distributing to the nodes again. In this project it was necessary to collect the data on one node to read and write the data. If scaled to larger datasets it would be better to eliminate these steps. For example, each node could write a separate output file without collecting the data on one node in the final step.

The performance suffers when the number of keys is not equally distributable over the number of nodes. In this program the size of the global key value pair table is divided by the number of nodes and then rounded down to the largest equally divisible size and the remaining keys are sent to the Nth processor (up to N-1 blocks). However, if the block size was rounded up instead of down then it would be even worse because in some instances the keys would be equally distributed across N-1 nodes and the last node would not receive any key value pairs. Instead of having equal distribution on N-1 nodes and moving all remaining keys to the Nth node it would be better to try to distribute the keys in a more uniform manner. It may be better to use a cyclic distribution but it would be slightly more difficult to keep track of the counts being sent to each node and the data offsets in the global matrix.