Umur Gogebakan - 21702705

Implementation Details

Parallel

First, in the master process I read the input file, source vertex and output file as arguments from the commandline. Using the "util" file, I read the input in the master process and calculate how the partition is going to take place. In order to do that, I allocate an array called vertex_arr_sizes to hold each process' vertex array sizes. Also, I calculate the displs_vertices array to use scatterv. Then, I do the same for edges instead of vertices. At this point, the master process has all the information to scatter the data to other processes. So, after reading the isDebug boolean variable from commandline whether to print partition or not, I broadcast the source vertex that has been read from commandline and total number number of all vertices from root (master process) to all processes in order to all processes have that information. Then I also broadcast displs vectors for both edges and vertices for further use. Then using MPI_SCATTER, I scatter the vertices and edge array sizes to indicate each process how many elements to get from input files in total. Then, in each process I dynamically allocate memory to hold vertices (offsets) and edges. This step is followed by scattering the whole vertices and edges using MPI_SCATTERV with relative displs and size values. Then, I allocate memory for distance memory and initiate all values with -1 followed by placing 0 to distance array corresponding to source vertex. Then, in each process I set the value of level as 1 and calculate buffer size by ceiling division of total number of vertices and number of processors. This value is the maximum number of possible buffer sizes that can be sent. And in each process I initiate a stack to hold frontier vertex values using another file to implement stack under the name of "stack.c" and relative header file. Then, I find the owner of the source vertex and initiate a queue counter to hold the size of the stack in each process. Then, the owner process of the source vertex, pushes source to stack and sets distance value of source as 0. Then, I dynamically allocate send and receive buffers as 2D contagious arrays each with buffer size. I also allocate memory to hold the sizes for each process of buffers, indicating how many elements to write/read. I initiate a size array with 0 and buffers with -1. This step is followed by while all frontier stacks are not empty, and while each process' stack is not empty; for all elements in the stack I find the number of total neighbors for the current vertex and for each neighbor I find the owner processes. If the neighbor does not exist in the buffers, I place it in the buffer and increment corresponding elements of the buff_sizes array. Then, with MPI_Alltoall communication, I send each buffer to another process and receive them in the receive buffers. Here each process gets which elements to work on from all other processes. I also send all processes' buff sizes using MPI_Alltoall again to send the information, which process should read how many elements from the received buffer. Then, in each process, for each element received, if the value of the distance array received is -1, I set it to the current level and push it to the frontier stack of the computing process. This means the neighbor has not been visited and will be traversed in the next iteration. Then I increment the level and reset the buff_sizes array for the next iteration. Finally, I use all reduce with MPI_reduce operation with MPI_MAX to reach the final result distance array by reducing each processes' distance arrays with max function for each element. So basically, each process traverses its own responsible vertices in breadth first manner and communicates with each other when they come across with a neighbor that should be processed by another process.

Serial

First, I define a struct called entry to put into a queue. The queue I used is TAILQ from sys/queue.h. Then in the main function, I start with reading command line arguments. First one for input file, second one for source vertex and third one for output file. Then, I read the input file with the given "util" file and call the BFS function. In the BFS function, I start with creating an entry to add to the queue with source vertex. To hold the size of the queue. I also use another variable called queue_counter. Then, I dynamically allocate a distance array where size is the number of total vertices and initialize each value with -1. Then, I set the distance value of the source vertex as 0 and initialize the level variable as 1 as we are starting to traverse the graph. Then, while the queue is not empty, for each element in the queue I dequeue an element and calculate the number of neighbors for the current vertex. Then, for each neighbor of the current vertex, if the neighbor distance value is -1, I set it to current level and add the neighbor to the queue. So, I traverse the graph in breadth first manner and set distance array accordingly where each value indicates the shortest number of vertices to traverse to go from source to each vertex.

Umur Gogebakan - 21702705

Plots

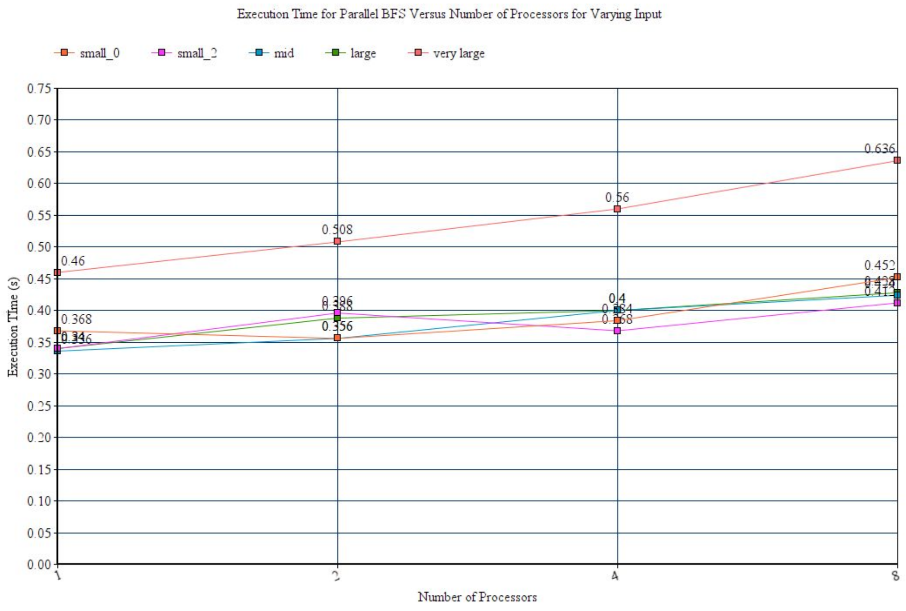

Figure 1: This figure shows the relation between execution time for Parallel BFS implementation in seconds and number of processors for different inputs.

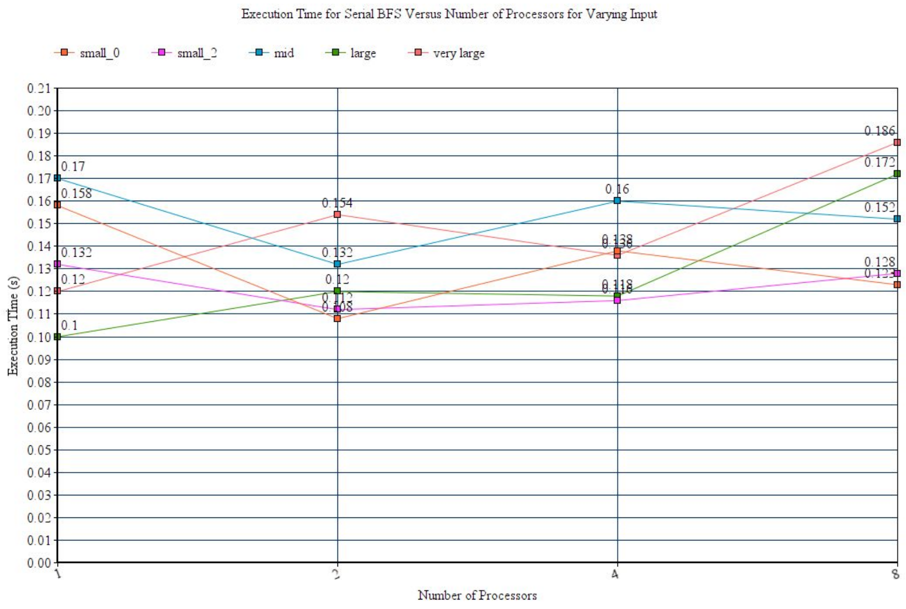

Figure 2: This figure shows the relation between execution time for Serial BFS implementation in seconds and number of processors for different inputs.

Umur Gogebakan - 21702705

Observations

From Figure 1, it is seen that as the number of processes increase, total execution of time also increases. This increase is observed as linear for all input sizes. Especially for the very large sized input, this linear increase trend is clearly observable. Yet, as expected the largest input results in the longest execution time. The reason for this linearly increasing observation can be the communication overhead between the processors.

From Figure 2, it is seen that as the number of processes increases, total execution of time does not follow a regular trend. Although for large and very large input, it is seen that execution time is increased and reaches its maximum at 8 processors, overall fashion of all executions cannot be regarded as linear unlike parallel implementation. Especially the small input with source vertex chosen as 2 shows the irregularity of the algorithm with respect to number of processors.

When we compare both of the implementations both figures indicate that serial implementation runs faster for all inputs invariant of the number of the processors. It is observed that the longest execution time of the serial algorithm is still faster than the shortest execution time of the parallel algorithm. Probably the reason for this is the communication bottleneck, that is caused by different processing nodes communicating between each other frequently.

First option as an improvement to this problem can be holding a unified frontier stack for storing elements to process. Holding a shared stack reduces the number of communicative steps in each iteration of the algorithm since we do not need to ask for information at each iteration whether their frontier stacks are empty or not. Second option would be changing the decomposition of the input data. 2D decomposition could have been also used since it captures a lower communication cost. With 2D communication, k^{th} frontier can be represented as a vector x_k. Third, changing the connection structure of nodes can change the results positively. Changing the topology of the network may cause speedups especially for all to all communication instructions.