Brian Rodriguez

Quiz 5

1. Continuations in parallel programming are a way of assigning work to threads that are available. The availability of work solely depends on if there is a worker available to take that work. The act of continuation stealing is hard to implement in a library, if there the library does not have compiler support. Naturally, with child stealing, the work is available before there is a worker available. This way of dividing up work is more natural to all operating systems and does not require compiler support. Clik and TBB differ in the sense of how the libraries divide and assign work to each thread worker needed. The work space complexity for both depends on their implementations of work assigning. Clik, assigns work on an availability basis which if it has N stack space and P workers then the work space complexity is Θ(PS). TBB, assigns the work first and waits for the threads to pick up the work as they finish their executions of the work they had been assigned. This results in a work space complexity of Θ(n).

2.

    a. The first 2 lines are instantiations of variables to be used as either flags, or as a way of signaling which source or destination to send the MPI call to. The first notable instantiation is MPI_Status Stat; this call is needed to make a call to MPI_Recv, it serves as a object to be populated with additional information about the receive operation after it completes. MPI_init is a function call that takes in the number of arguments and the argument vector received, this call is to be called once by the main thread and before a MPI_Finalize and it also initializes the MPI execution environment. MPI_Comm_Size determines the number of tasks that is associated with the communicator being used. MPI_Comm_rank determines the rank of the calling process associated with the communicator, this essentially gives a rank to the threads that were formed. If the thread rank is 0 then set the destination and source variable to 1 which sends the message and receives a message from thread with rank 1. MPI_Send takes in the out message which is x, the number of chars to send, the data type that is being used, in this case a char, the rank of the destination, the message tag and the communicator handle. This function sends a message to the rank thread that was specified. MPI_Recv waits for a message to be sent to the current thread. It takes in an input buffer, number of characters to receive, the data type being received, the source rank that is sending the message, the tag number, the communicator handle, and the Status of the message being received. The next if statement if if the rank is 1. These next executions are the reverse order of the rank 0. While rank 0 thread is working on sending a message, this thread is waiting to receive the message from rank 0. Then rank 1 thread is sending a message back to rank 0. After both threads have done their work, they are making a call to MPI_Get_count which takes in the status of the threads receive, the data type of the buffer, and the number of received elements. Then the next line prints off the rank, the number of elements received in the message, the source element which the message was sent from and the tag number. MPI_Finalize then ends the MPI execution environment.

b. The non-blocking code is essentially merry-go-round involving each rank that was created. The rank determines the neighbors in which to send the message to. For example, if my rank is 3 then I would send a message to 2 and 4 respectively. The receiving and sends are non-blocking so the messages can be received and sent concurrently. MPI_Request and MPI_Status sets up the statuses and request handles to be used by each thread. In this case, it seems like the statuses aren't analyzed like in the blocking case above. MPI_Init, MPI_comm_size and MPI_Comm_rank are standard when using MPI. These calls setup the MPI environment, gather the number tasks and assign ranks to each thread. MPI_Irecv is the non-blocking receive call that is called by each thread to receive the message from the previous rank and the next rank. MPI_Isend is the non-blocking receive call that is called by each thread to send their respective rank to the previous and next threads in the merry-go-round structure. MPI_Waitall, waits for all MPI requests to complete and assigns the statuses to their respective ranks. MPI_Finalize then ends the MPI execution environment.

3. At first glance, the program logic seems very complicated and I am not too sure if I am understanding the full extend of what is being done. To my understanding it looks like the master thread or thread with rank 0 is dividing up the work for the worker threads. The master thread is giving work while the other worker threads are waiting to be given work. After the threads have been given work, the master thread waits on a response back from the worker thread to report to the answer.

4. This program executes in a non-blocking fashion, unlike the previous program that was analyzed. They essentially work the exact same. The master thread divides the work to the worker threads but does not block and continues the execution. The master thread then waits for all the threads to return from their executions and continues to receive all of the data that was processed by the worker threads. It then sends more work to all of the worker threads and again waits for all of the threads to be done. The work is sent out in a non-blocking fashion and they are all waited on by the master thread. The master thread then reports all of the work done by all the threads combined at the end.