Parallel Programming HW7 Write Up
Due Monday March 15, 2021, 11:59 pm

Connor Osborne
A01880782

Sunday March 14, 2021

1 Description

For homework 7, I wrote an MPI program that performs sender-initiated distributed random dynamic load balancing. The basic unit of work performed by each process, a task, is represented by an integer i in the range [1-1024], is to increment an integer i^2 times. Each process has a queue in which it places tasks to perform.
Each process performs the following actions:

- Check to see if any new work has arrived
- If the number of tasks in the queue exceeds the threshold of 16, then 2 tasks are sent to random destinations
- Perform some work
- For half the processes, if the number of tasks generated is less than a random number [1024-2048], generate 1-3 new tasks, and place them at the end of the process task queue.
- For the other half of the processes, generate no new tasks.

The program also implements a dual-pass ring termination algorithm, so that the system terminates normally.

2 Implementation

The implementation for this program started with a small function called doTask() that accepts the task integer and increments a number i^2 times.

```
    void doTask(int num){
    int incrementor = 0;
    for (int i = 0; i < (num * num); ++i){
        incrementor++;
    }
}
```

The next step was the variable creation and initialization of the MPI session. Besides the normal variables created for MPI. I made a totalTasks variable that the even numbered processes would use to know how many tasks to create. I also made a tasksMade variable so the process would know when to stop making tasks and a tasksCompleted variable so that each process would know how many tasks they had completed during the running of the program.

The next variables to be made were the booleans: processBlack, tokenBlack, hasToken; and terminate. The variable processBlack keeps track of whether the process is active and has sent a tasks to a lower ranked process, while tokenBlack and hasToken keeps track of what color the token currently is and whether or not the process currently possesses the token. The terminate variable is to be the kill code that will be sent by process zero when it's time to end all processes. A vector called taskQueue is also made so each process can keep track of the tasks they have. After all of this a random number seed is made with srand() so each process will be able to send tasks to random processes later.

```
int main(int argc, char** argv){
    int totalTasks;
    int tasksMade = 0;
    int tasksCompleted = 0;
    int rank, size;

    bool processBlack = false;
    bool tokenBlack = false;
    bool hasToken = false;
    bool terminate = false;
    vector<int> taskQueue;

    MPI_Status mystatus;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MCW, &rank);
    MPI_Comm_size(MCW, &size);

    srand(time(NULL) + rank);
```

After the standard variables were made I started the timer to keep track of how long the program will run for by making a starting time t1 to use at the end of the run. Then I had rank 0 start out with the token by setting it's hasToken to true. Next i set up a task variable so that all of the processes will have a variable for sending and receiveing tasks from their queue.

```
   std::chrono::high_resolution_clock::time_point t1 = std::chrono::high_resolution_clock::now();

    if(rank == 0){
        hasToken = true;
    }

    int task;
```

As stated in the introduction the even and odd number processes will work exactly the same other than the even numbered processes will be generating all of the tasks for the system.
I used a simple if statement to differentiate whether a process was even or odd. Then I had each of the even processes set the value of totalTasks by picking a random number between 1024 and 2048. Then they each print out how many tasks they will be generating before moving on.
At the beginning of the while loop the processes will be checking to see if they received the token in case the program is beginning to come to an end. A variable, tokenFlag is used with MPI_Iprobe with tag 2 to see if the token was sent to the process. If the process is receiving a token hasToken is set to true and then it checks to see if it is process 0. If the process receiving the token is process 0 and tokenBlack is false, then then terminate is set to true and process 0 will send the termination signal to all other processes using tag 3. If the tokenBlack was true then process 0 will set it to false and the system will continue for at least another round.
Next all processes besides 0 will check to see if the termination signal was sent using terminateFlag and MPI_Iprobe. If terminate becomes true for them then each process will print out how many tasks they completed and process 0 will also calculate the total time the program took to complete and print it to the

screen.

```
  if(rank%2 == 0){
      totalTasks = rand()%(2048 - 1024) + 1024;
      cout << rank << " will produce " << totalTasks << " tasks." << endl;
      while(1){
          //Check to see if the token was sent to the process
          int tokenFlag;
          MPI_Iprobe(MPI_ANY_SOURCE, 2, MCW, &tokenFlag, &mystatus);
          if(tokenFlag){
              MPI_Recv(&tokenBlack, 1, MPI_C_BOOL, MPI_ANY_SOURCE, 2, MCW, &mystatus);
              hasToken = true;
              if (rank == 0 && tokenBlack == false){
                  terminate = true;
                  for (int i = 1; i < size; ++i){
                      MPI_Send(&terminate, 1, MPI_C_BOOL, i, 3, MCW);
                  }
              }
              else if(rank == 0 && tokenBlack == true){
                  tokenBlack = false;
              }
          }

          //Check to see if the program is ending
          if(rank != 0){
              int terminateFlag;
              MPI_Iprobe(0, 3, MCW, &terminateFlag, &mystatus);
              if(terminateFlag){
                  MPI_Recv(&terminate, 1, MPI_C_BOOL, 0, 3, MCW, &mystatus);
              }
          }
          if(terminate){
              cout << rank << " completed " << tasksCompleted << " tasks." << endl;
              if (rank == 0){
                  std::chrono::high_resolution_clock::time_point t2 =
                       std::chrono::high_resolution_clock::now();
                  std::chrono::duration<double> time_span =
                       std::chrono::duration_cast<std::chrono::duration<double>>(t2 - t1);
                  sleep(1);
                  cout << "Program Completed in " << time_span.count() << "seconds." << endl;
              }
              break;
          }
```

If the program didn't terminate then the processes move on and check to see if any work has been sent to them using the variable flag and an MPI_Iprobe with tag 0. If tasks have been sent then the process enters a while loop to receive all tasks sent to it and adds them to its taskQueue. After receiving all sent tasks the process then checks to make sure that the size of the task queue is not greater than 16, if it is the the process will send the last two tasks in its queue to two random processes in the system.

```
          //Check to see if tasks have been sent to the process and add them to the queue
          int flag;
          MPI_Iprobe(MPI_ANY_SOURCE, 0, MCW, &flag, &mystatus);
          while(flag){
              MPI_Recv(&task, 1, MPI_INT, MPI_ANY_SOURCE, 0, MCW, &mystatus);
              taskQueue.push_back(task);
              MPI_Iprobe(MPI_ANY_SOURCE, 0, MCW, &flag, &mystatus);
          }
```

```
        // Check to see if the process' queue has too many tasks, if so send 2 tasks to random
            processes.
        if(taskQueue.size() > 16){
            int sendTo1;
            int sendTo2;
            do{
                sendTo1 = rand() % size;
                sendTo2 = rand() % size;
            }while(sendTo1 != rank && sendTo2 != rank);

            if(rank > 0){
                if(sendTo1 < rank || sendTo2 < rank){
                    processBlack = true;
                }
            }

            int sendingTask1 = taskQueue[(taskQueue.size()-1)];
            taskQueue.pop_back();
            MPI_Send(&sendingTask1, 1, MPI_INT, sendTo1, 0, MCW);

            int sendingTask2 = taskQueue[(taskQueue.size()-1)];
            taskQueue.pop_back();
            MPI_Send(&sendingTask2, 1, MPI_INT, sendTo2, 0, MCW);
        }
```

Next each process will use doTask() to complete the first task in the taskQueue and then remove the task from the queue. Then if the process is even numbered it will check to see if it has made enough tasks. If not it will create 3 new tasks and add them to its queue.

```
// If the quota of tasks is not fulfilled then make more tasks and update the number of tasks
    that have been made.
        for(int i = 0; i < 3; ++i){
            if(tasksMade < totalTasks){
                int newTask = rand() % 1025 + 1;
                tasksMade += 1;
                taskQueue.push_back(newTask);
            }
        }
```

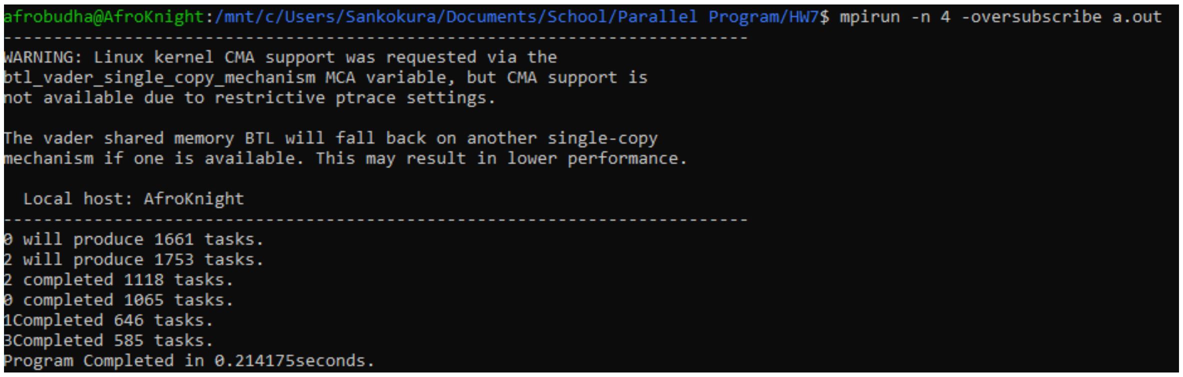
After all of this the processes will check to see if their queue is empty. If it is empty then it will also check to see if is has the token. If the process has the token and the process' processBlack is true and tokenBlack is false, then processBlack becomes false and tokenBlack become true. Then the process will decide a destination to send the token to. If the process is the last token in line it will sen the token to process 0 other wise it will send it to the next process in line. Then the process' hasToken becomes false.

```
        //Check to see if queue is empty, if it is and the process has the token make necessary
            changes and send token to next process.
        if(taskQueue.size() == 0){
            if(hasToken == true){
                // cout << rank << " has gone idle and will send the token to " << rank+1 <<
                    endl;
                if(processBlack == true && tokenBlack == false){
                    processBlack = false;
                    tokenBlack = true;
                }
                int dest;
                if (rank == size -1){
                    dest = 0;
                }
                else{
                    dest = rank + 1;
                }
                MPI_Send(&tokenBlack, 1, MPI_C_BOOL, dest, 2, MCW);
                hasToken = false;
            }
        }
    }
}
```

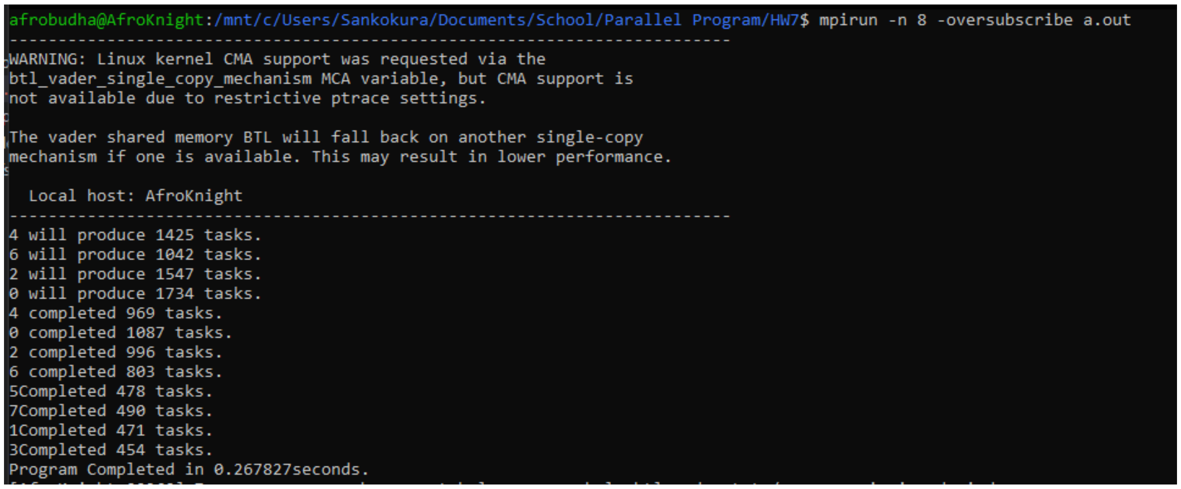
As previously stated all odd numbered processes perform the same tasks other than creating new tasks for the system.

```
   else{
    while(1){
        //Check to see if the token was sent to the process
        int tokenFlag;
        MPI_Iprobe(MPI_ANY_SOURCE, 2, MCW, &tokenFlag, &mystatus);
        if(tokenFlag){
            MPI_Recv(&tokenBlack, 1, MPI_C_BOOL, MPI_ANY_SOURCE, 2, MCW, &mystatus);
            hasToken = true;
            if (rank == 0 && tokenBlack == false){
                terminate = true;
                for (int i = 1; i < size; ++i){
                    MPI_Send(&terminate, 1, MPI_C_BOOL, i, 3, MCW);
                }
            }
        }

        //Check to see if program is endingAft
        if(rank != 0){
            int terminateFlag;
            MPI_Iprobe(0, 3, MCW, &terminateFlag, &mystatus);
            if(terminateFlag){
                MPI_Recv(&terminate, 1, MPI_C_BOOL, 0, 3, MCW, &mystatus);
            }
        }
        if(terminate){
            cout << rank << "Completed " << tasksCompleted << " tasks." << endl;
            break;
        }

        // Check to see if tasks have been sent to this process and then add the tasks to queue.
        int flag;
        MPI_Iprobe(MPI_ANY_SOURCE, 0, MCW, &flag, &mystatus);
        while(flag){
            MPI_Recv(&task, 1, MPI_INT, MPI_ANY_SOURCE, 0, MCW, &mystatus);
            taskQueue.push_back(task);
            MPI_Iprobe(MPI_ANY_SOURCE, 0, MCW, &flag, &mystatus);
        }

        // Check to see if the process' queue has too many tasks, if so send 2 tasks to random
            processes.
        if(taskQueue.size() > 16){
                int sendTo1;
                int sendTo2;
                do{
                    sendTo1 = rand() % size;
                    sendTo2 = rand() % size;
                }while(sendTo1 != rank && sendTo2 != rank);

                if(rank > 0){
                    if(sendTo1 < rank || sendTo2 < rank){
                        processBlack = true;
                    }
                }

                int sendingTask1 = taskQueue[(taskQueue.size()-1)];
                taskQueue.pop_back();
                MPI_Send(&sendingTask1, 1, MPI_INT, sendTo1, 0, MCW);

                int sendingTask2 = taskQueue[(taskQueue.size()-1)];
                taskQueue.pop_back();
                MPI_Send(&sendingTask2, 1, MPI_INT, sendTo2, 0, MCW);
            }

            // Complete a task and remove it from queue
            if(taskQueue.size() > 0){
                int workingTask = taskQueue[0];
                doTask(workingTask);
                tasksCompleted += 1;
                vector<int>::iterator deleteTask = taskQueue.begin();
                taskQueue.erase(deleteTask);
            }

            //Check to see if queue is empty, if it is and the process has the token make necessary
                changes and send token to next process.
            if(taskQueue.size() == 0){
                if(hasToken == true){
                    // cout << rank << " has gone idle and will send the token to " << rank+1 <<
                        endl;
                    if(processBlack == true && tokenBlack == false){
                        processBlack = false;
                        tokenBlack = true;
                    }
                    int dest;
                    if (rank == size -1){
                        dest = 0;
                    }
                    else{
                        dest = rank + 1;
                    }
                    MPI_Send(&tokenBlack, 1, MPI_C_BOOL, dest, 2, MCW);
                    hasToken = false;
                }
            }
        }
    }

    MPI_Finalize();
    return 0;
}
```

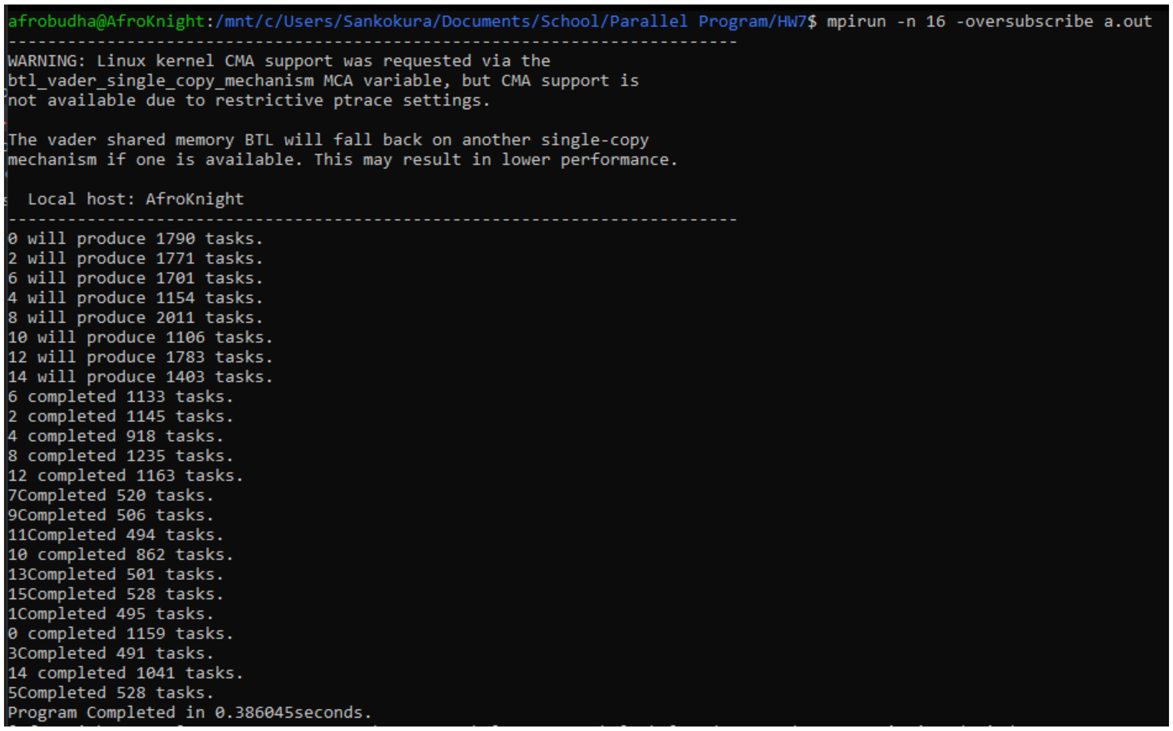
3 Input and Output with 4, 8, and 16 Processes



```
afrobudha@AfroKnight:/mnt/c/Users/Sankokura/Documents/School/Parallel Program/HW7$ mpirun -n 4 -oversubscribe a.out
--------------------------------------------------------------------------
WARNING: Linux kernel CMA support was requested via the
btl_vader_single_copy_mechanism MCA variable, but CMA support is
not available due to restrictive ptrace settings.

The vader shared memory BTL will fall back on another single-copy
mechanism if one is available. This may result in lower performance.

  Local host: AfroKnight
--------------------------------------------------------------------------
0 will produce 1661 tasks.
2 will produce 1753 tasks.
2 completed 1118 tasks.
0 completed 1065 tasks.
1Completed 646 tasks.
3Completed 585 tasks.
Program Completed in 0.214175seconds.
```



```
afrobudha@AfroKnight:/mnt/c/Users/Sankokura/Documents/School/Parallel Program/HW7$ mpirun -n 8 -oversubscribe a.out
--------------------------------------------------------------------------
WARNING: Linux kernel CMA support was requested via the
btl_vader_single_copy_mechanism MCA variable, but CMA support is
not available due to restrictive ptrace settings.

The vader shared memory BTL will fall back on another single-copy
mechanism if one is available. This may result in lower performance.

  Local host: AfroKnight
--------------------------------------------------------------------------
4 will produce 1425 tasks.
6 will produce 1042 tasks.
2 will produce 1547 tasks.
0 will produce 1734 tasks.
4 completed 969 tasks.
0 completed 1087 tasks.
2 completed 996 tasks.
6 completed 803 tasks.
5Completed 478 tasks.
7Completed 490 tasks.
1Completed 471 tasks.
3Completed 454 tasks.
Program Completed in 0.267827seconds.
```



```
afrobudha@AfroKnight:/mnt/c/Users/Sankokura/Documents/School/Parallel Program/HW7$ mpirun -n 16 -oversubscribe a.out
--------------------------------------------------------------------------
WARNING: Linux kernel CMA support was requested via the
btl_vader_single_copy_mechanism MCA variable, but CMA support is
not available due to restrictive ptrace settings.

The vader shared memory BTL will fall back on another single-copy
mechanism if one is available. This may result in lower performance.

  Local host: AfroKnight
--------------------------------------------------------------------------
0 will produce 1790 tasks.
2 will produce 1771 tasks.
6 will produce 1701 tasks.
4 will produce 1154 tasks.
8 will produce 2011 tasks.
10 will produce 1106 tasks.
12 will produce 1783 tasks.
14 will produce 1403 tasks.
6 completed 1133 tasks.
2 completed 1145 tasks.
4 completed 918 tasks.
8 completed 1235 tasks.
12 completed 1163 tasks.
7Completed 520 tasks.
9Completed 506 tasks.
11Completed 494 tasks.
10 completed 862 tasks.
13Completed 501 tasks.
15Completed 528 tasks.
1Completed 495 tasks.
0 completed 1159 tasks.
3Completed 491 tasks.
14 completed 1041 tasks.
5Completed 528 tasks.
Program Completed in 0.386045seconds.
```