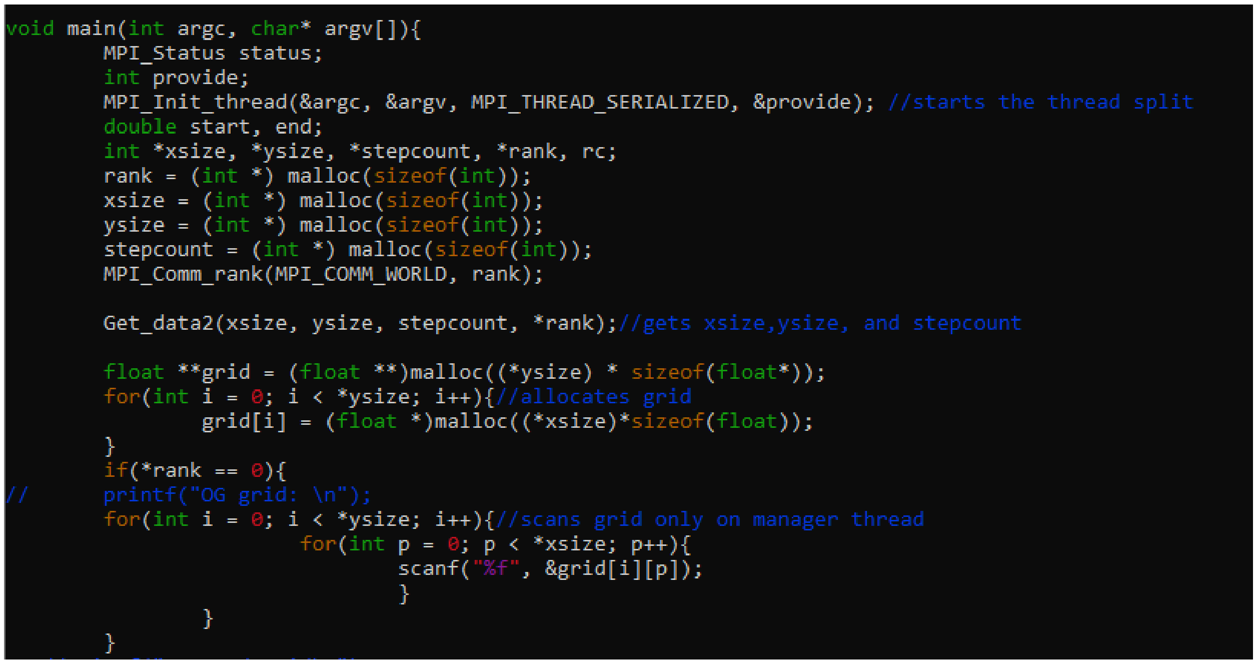
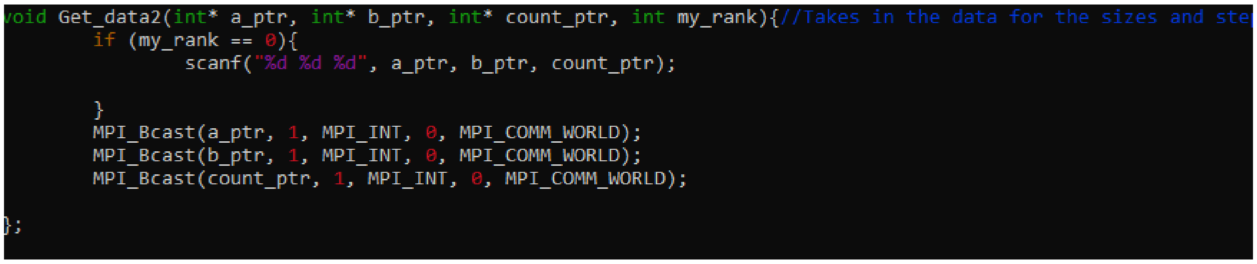
Michael Gibson

This assignment is much like assignment 2, in which we use RB switching to simulate temperature in an ocean. This time, however, MPI is used. We start of by initializing MPI



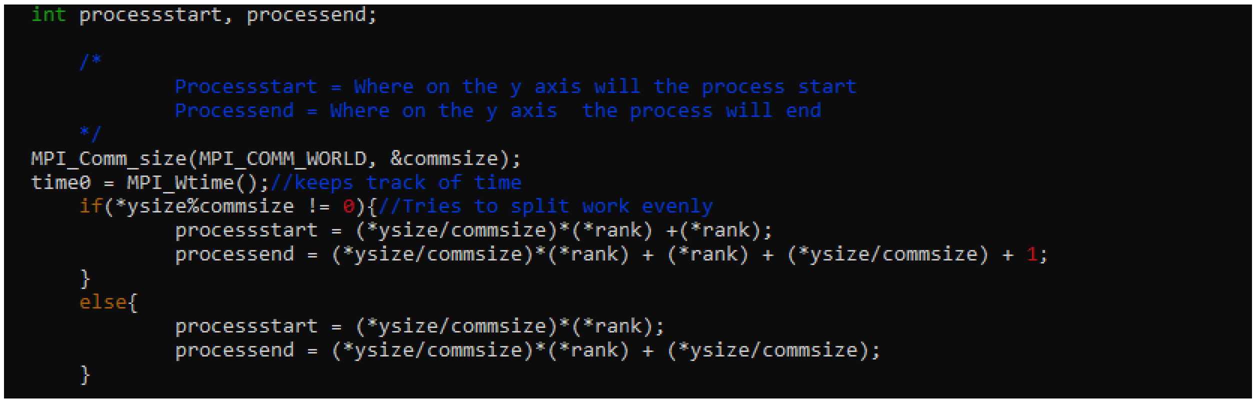
```
void main(int argc, char* argv[]){
        MPI_Status status;
        int provide;
        MPI_Init_thread(&argc, &argv, MPI_THREAD_SERIALIZED, &provide); //starts the thread split
        double start, end;
        int *xsize, *ysize, *stepcount, *rank, rc;
        rank = (int *) malloc(sizeof(int));
        xsize = (int *) malloc(sizeof(int));
        ysize = (int *) malloc(sizeof(int));
        stepcount = (int *) malloc(sizeof(int));
        MPI_Comm_rank(MPI_COMM_WORLD, rank);

        Get_data2(xsize, ysize, stepcount, *rank);//gets xsize,ysize, and stepcount

        float **grid = (float **)malloc((*ysize) * sizeof(float*));
        for(int i = 0; i < *ysize; i++){//allocates grid
                grid[i] = (float *)malloc((*xsize)*sizeof(float));
        }
        if(*rank == 0){
//      printf("OG grid: \n");
        for(int i = 0; i < *ysize; i++){//scans grid only on manager thread
                        for(int p = 0; p < *xsize; p++){
                                scanf("%f", &grid[i][p]);
                                }
                }
        }
```

We also allocate for all our values we are receiving. Get_data2 is what retrieves the x and y size, as well as step count.



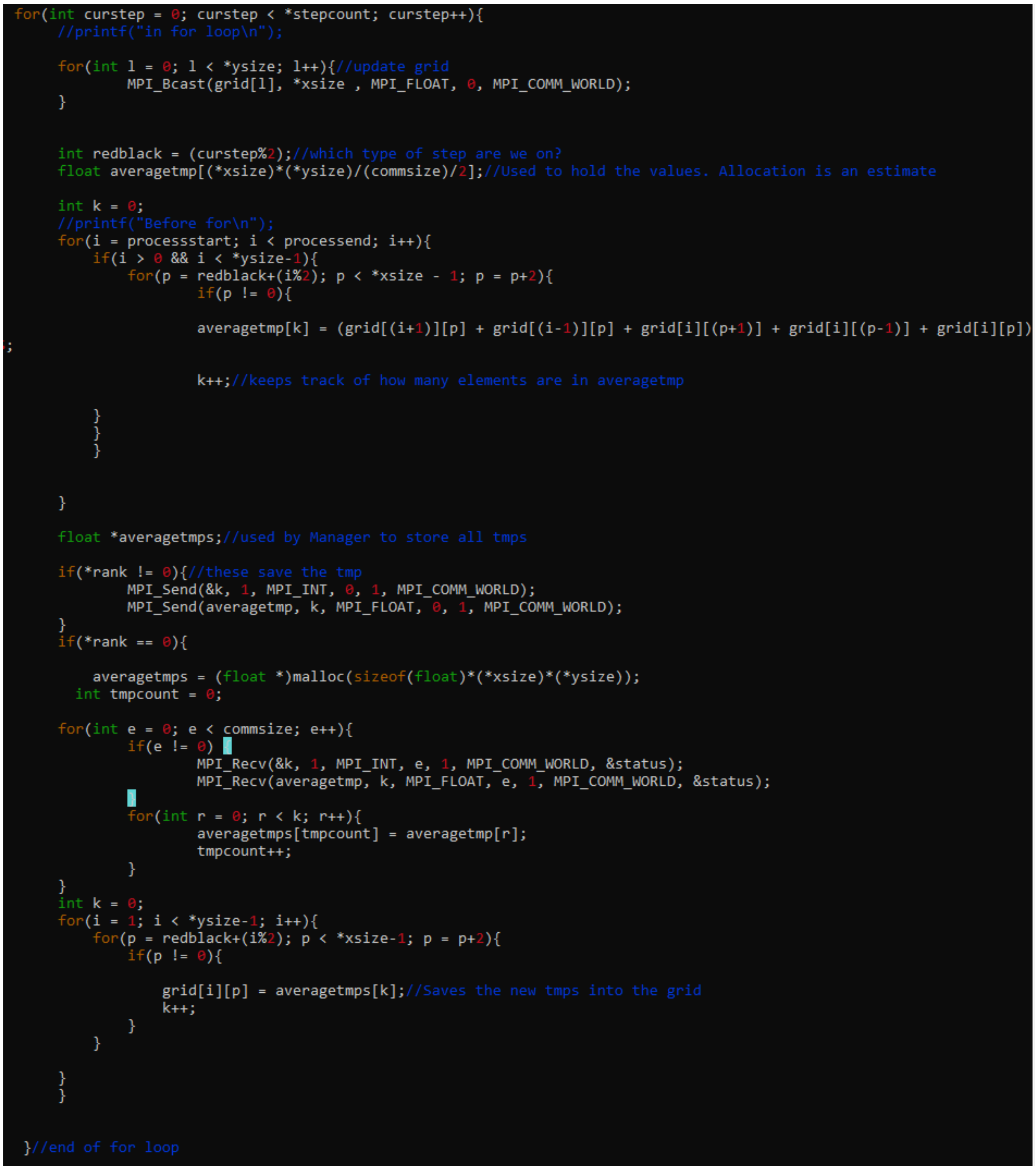
```
void Get_data2(int* a_ptr, int* b_ptr, int* count_ptr, int my_rank){//Takes in the data for the sizes and ste
        if (my_rank == 0){
                scanf("%d %d %d", a_ptr, b_ptr, count_ptr);

        }
        MPI_Bcast(a_ptr, 1, MPI_INT, 0, MPI_COMM_WORLD);
        MPI_Bcast(b_ptr, 1, MPI_INT, 0, MPI_COMM_WORLD);
        MPI_Bcast(count_ptr, 1, MPI_INT, 0, MPI_COMM_WORLD);

};
```

The grid is read in from the main function, only from the manager thread though, thread 0. Next we find Process start and end, which will tell each thread where in the grid they'll start and end at. This is determined based on whether or not the program can have mod 0 from rows to thread count.Otherwise, it increases the size of the process to get the whole grid.



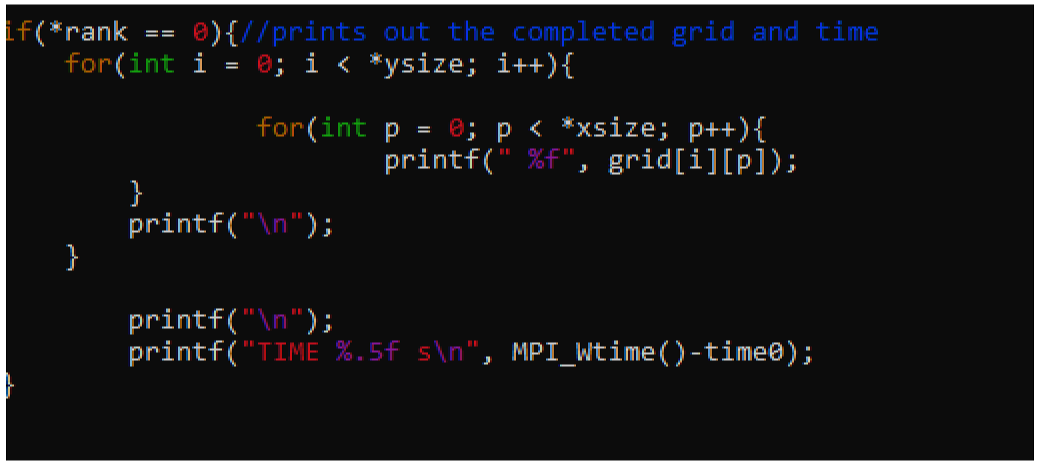
```
int processstart, processend;

    /*
            Processstart = Where on the y axis will the process start
            Processend = Where on the y axis  the process will end
    */
MPI_Comm_size(MPI_COMM_WORLD, &commsize);
time0 = MPI_Wtime();//keeps track of time
    if(*ysize%commsize != 0){//Tries to split work evenly
            processstart = (*ysize/commsize)*(*rank) +(*rank);
            processend = (*ysize/commsize)*(*rank) + (*rank) + (*ysize/commsize) + 1;
    }
    else{
            processstart = (*ysize/commsize)*(*rank);
            processend = (*ysize/commsize)*(*rank) + (*ysize/commsize);
    }
```

After that, the for loop start. This for loop has the majority of the work. Starts by updating the grid for all threads using broadcast. Then, it checks the nearby temperatures for each of its designated coordinates. It takes the average temperature and puts it in a local array. That local array is then sent to the main thread, where it all is added into one large array through MPI_Send and Recv. That large array is then inserted into the right parts of the grid, and the process then starts all over again.



```
 for(int curstep = 0; curstep < *stepcount; curstep++){
      //printf("in for loop\n");

      for(int l = 0; l < *ysize; l++){//update grid
              MPI_Bcast(grid[l], *xsize , MPI_FLOAT, 0, MPI_COMM_WORLD);
      }


      int redblack = (curstep%2);//which type of step are we on?
      float averagetmp[(*xsize)*(*ysize)/(commsize)/2];//Used to hold the values. Allocation is an estimate

      int k = 0;
      //printf("Before for\n");
      for(i = processstart; i < processend; i++){
          if(i > 0 && i < *ysize-1){
              for(p = redblack+(i%2); p < *xsize - 1; p = p+2){
                      if(p != 0){

                      averagetmp[k] = (grid[(i+1)][p] + grid[(i-1)][p] + grid[i][(p+1)] + grid[i][(p-1)] + grid[i][p])
;


                      k++;//keeps track of how many elements are in averagetmp

          }
          }
          }


      }

      float *averagetmps;//used by Manager to store all tmps

      if(*rank != 0){//these save the tmp
              MPI_Send(&k, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
              MPI_Send(averagetmp, k, MPI_FLOAT, 0, 1, MPI_COMM_WORLD);
      }
      if(*rank == 0){

          averagetmps = (float *)malloc(sizeof(float)*(*xsize)*(*ysize));
        int tmpcount = 0;

      for(int e = 0; e < commsize; e++){
              if(e != 0) {
                      MPI_Recv(&k, 1, MPI_INT, e, 1, MPI_COMM_WORLD, &status);
                      MPI_Recv(averagetmp, k, MPI_FLOAT, e, 1, MPI_COMM_WORLD, &status);
              }
              for(int r = 0; r < k; r++){
                      averagetmps[tmpcount] = averagetmp[r];
                      tmpcount++;
              }
      }
      int k = 0;
      for(i = 1; i < *ysize-1; i++){
          for(p = redblack+(i%2); p < *xsize-1; p = p+2){
              if(p != 0){

                  grid[i][p] = averagetmps[k];//Saves the new tmps into the grid
                  k++;
              }
          }

      }
      }


  }//end of for loop
```

After that, all thats left is to print out the last grid we got as well as the time.



```
if(*rank == 0){//prints out the completed grid and time
    for(int i = 0; i < *ysize; i++){

            for(int p = 0; p < *xsize; p++){
                    printf(" %f", grid[i][p]);
        }
        printf("\n");
    }

        printf("\n");
        printf("TIME %.5f s\n", MPI_Wtime()-time0);
}
```

And of course, we end it with MPI_Finalize();

How it works:

This program takes in 3 starting variables, the size of x, the size of y, and the amount of steps (time) that you want to take. Each number after that is added to the grid til it is full. I started testing by using a small text file called nums.txt. It contains a 5 by 5 grid, consisting of varying steps, the boundaries are all 10s while the inside is all 0s. I used this to do basic testing, and to compare it with my other program to make sure values were coming out correctly. Sometimes I'd change the size, specifically the y size, to make sure it wasn't just coincidence. Then, I created a large text file consisting of a matrix with 1 million elements. I created this by using another c program I wrote. This was used to test performance.

Performance:

# elements	# nodes	Time
25 elements (w/ 100 steps)	1	0.00014s
25 elements	2	0.00098s
25 elements	4	0.00190s
25 elements	8	0.01016s
1,000,000 elements (w/ 10 steps)	1	3.48792s
1,000,000 elements	2	3.91650s
1,000,000 elements	4	3.70359s
1,000,000 elements	8	4.38308s

The performance seems to be a slight decrease, even at 1 million elements. However, the ratio has shrunk. It is significantly slower to run more threads on smaller grids than larger grids in comparison to less threads.