Overview:

By chunk we mean sections of data, one chunk means the data was processed by one thread.

The raw data we collected is in the slurm files, and they should have an output description at the top and bottom of each run. The runtime data is recorded and graphed in the data.xlsx file.

In our first implementation of this project we had all the programs in one directory, and we ran all the batch files with the testall.c program. My partner and I then looked at the documentation and realized that each program needs to be in it's own subdirectory. The current project is split into sub directories (one for each program) and is SUPPOSED to use one make file that does a single run to demonstrate that each program is working. However, we were unable to get the makefiles to run `module load` to be able to load foss and OpenMP to run the programs. Instead, we'll provide the instructions to run each program below. To run the batches of the program, you can run the testall program in the main directory.

<u>To compile and run Pthreads:</u>

gcc -fopenmp Pthreads.c -o Pthreads

./Pthreads 1 1

<u>To compile and run OpenMP:</u>

gcc -fopenmp OpenMP.c -o OpenMP

./OpenMP 1 1

<u>To compile and run MPI:</u>

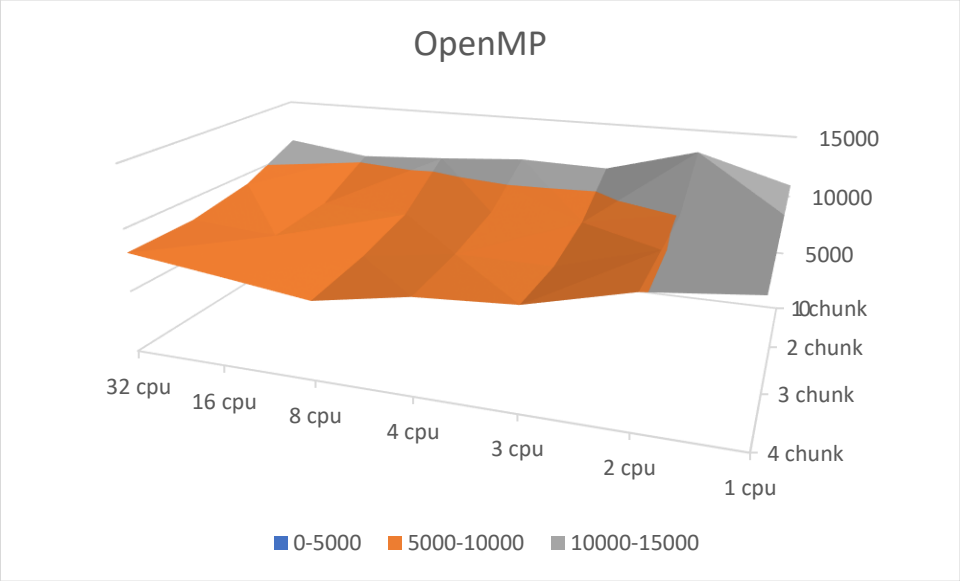
module load foss

mpicc MPI.c -o MPI

mpirun -np 1 MPI 1

OpenMP

Graph:



High: 13144.577 at 2cpu and 1 chunk

Low: 7201.84 at 8cpu and 4 chunks

Are there any race conditions? There are no race conditions. The data output is correct to the best of our knowledge.

How do you handle synchronization between processes? Give each thread it's own section to work on, so that none of the threads are in conflict.

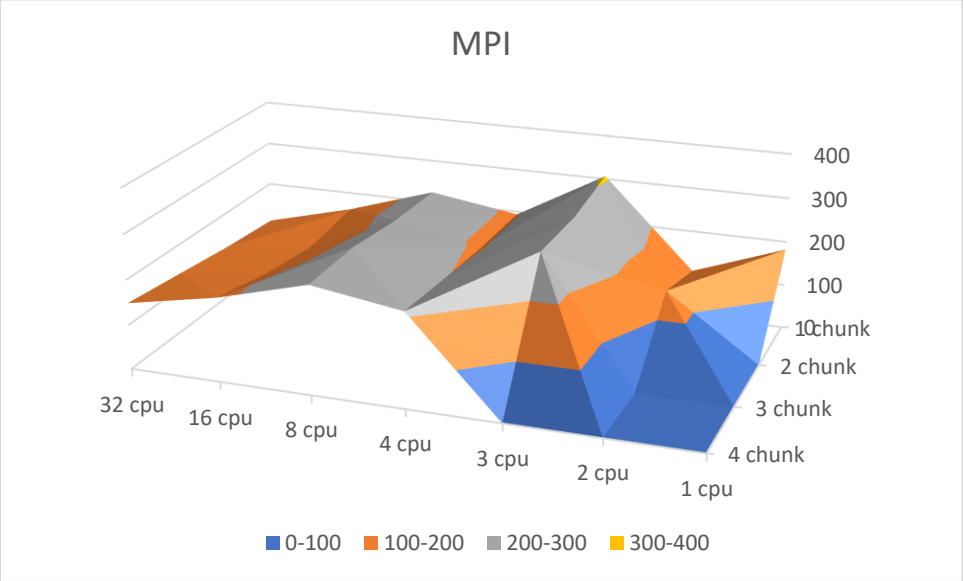
```
startPos = myID * (ARRAY_SIZE / NUM_THREADS);
endPos = startPos + (ARRAY_SIZE / NUM_THREADS);
```

How much communication do you do, and are you making any attempts to optimize this? Why or why not? We do not worry about communication, minus the echo statements that will output what test ran. For example in the main we pass the program arguments to print out the number of threads and the number of cpus it ran on. We did not see a need to optimize this, thus we did not attempt it.

```
cpu_num = strtol(argv[1], NULL, 10);
printf("cpu_num: %d\n", cpu_num);
NUM_THREADS = strtol(argv[2], NULL, 10);
printf("num_threads: %d\n", NUM_THREADS);
```

MPI

Graph:



High: 306.507 at 3cpus and 1 chunk

Low: 107.068 at 2 cpus and 1 chunk

Note: the blue area is recorded as a zero in the graph, we found out those resource combinations were not possible and got an error in allocating when it was attempted on beocat.

Are there any race conditions? No there are no race conditions.

How do you handle synchronization between processes?

I believe we use what is called a barrier to catch all the threads and sync up before continuing.

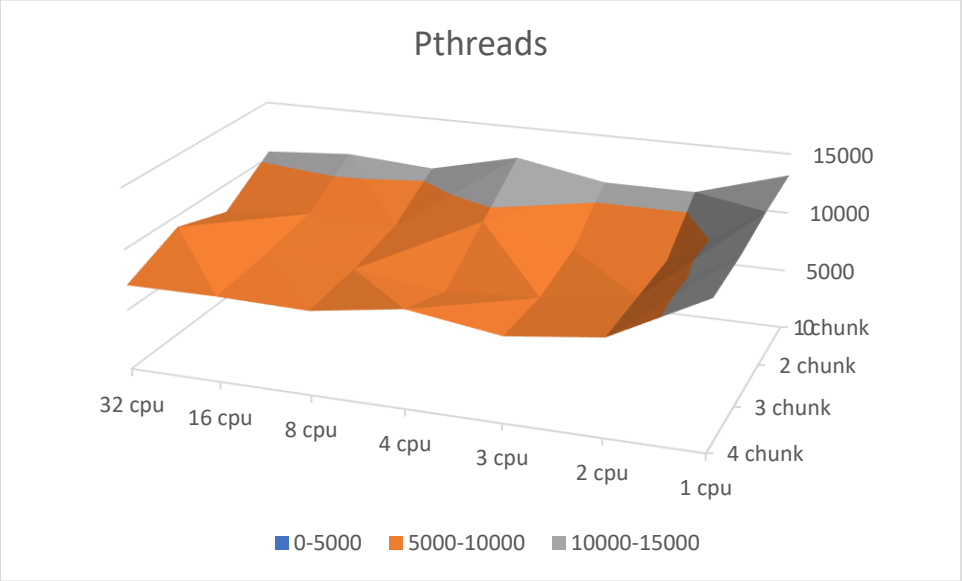
```
MPI_Reduce(local_line_avg, line_avg, ARRAY_SIZE, MPI_INT, MPI_SUM, 0, MPI_CO
MM_WORLD);
```

We also use the if rank == 0 check, to make the first thread initialize the arrays and do some set up work, and result printing. (so that the results are not printed per thread) In this case it appears that MPI is easier to implement and significantly faster than the other two implementations.

How much communication do you do, and are you making any attempts to optimize this? Why or why not? We pass arguments to the program to set the number of cpus. (used for the cpu efficiency calculation, which turned out to be 1 over the number of cpus) We did not see a need to optimize this thus, we did not attempt it.

Pthreads

Graph:



High: 13202.816 at 1cpu and 1 chunk (the graph is tilted to better view the valley for the low)

Low: the valley at 6370.543 at 4 cpus and 3 chunks

Are there any race conditions?

There are no race conditions that we are aware of.

How do you handle synchronization between processes?

We catch each thread, and use the pthread_join function.

```
for (j = 0; j < NUM_THREADS; j++)
        pthread_join(threads[j], NULL);
```

How much communication do you do, and are you making any attempts to optimize this? Why or why not? We pass in the number of cpus and the number of threads for each test. We did not see a need to optimize the communication, thus we did not attempt it.

Scripts:

```
#!/bin/bash -l
#SBATCH --mem=120G
#SBATCH --time=24:00:00
#SBATCH --job-name=MPI
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1

 module purge
 module load foss

 echo "Running MPI on $HOSTNAME"

 export OMP_NUM_THREADS=1
 time mpirun -np 1 MPI 1

 echo "Finished run on $SLURM_NTASKS $HOSTNAME cores"

 echo "Running MPI on $HOSTNAME"

 export OMP_NUM_THREADS=1
 time mpirun -np 2 MPI 1

 echo "Finished run on $SLURM_NTASKS $HOSTNAME cores"

  echo "Running MPI on $HOSTNAME"

 export OMP_NUM_THREADS=1
 time mpirun -np 3 MPI 1

 echo "Finished run on $SLURM_NTASKS $HOSTNAME cores"

 echo "Running MPI on $HOSTNAME"

 export OMP_NUM_THREADS=1
 time mpirun -np 4 MPI 1

 echo "Finished run on $SLURM_NTASKS $HOSTNAME cores"
```

This is an example of the batch script we wrote for each program. This one is marked to 1 core, and 1,2,3,4 threads. (we were not sure what OMP_NUM_THREADS is for in the example script and opted to leave it alone.)

```c
#include <stdio.h>
#include <string.h>

int main()
{
    /*
     int help;
    char command[100] =
        "sbatch sb.MPI-1; sbatch sb.MPI-2; sbatch sb.MPI-3;";// sbatch sb.MPI-
4;";
    help = system(command);
    char g2[100] =
        "sbatch sb.OpenMP-1; sbatch sb.OpenMP-2; sbatch sb.OpenMP-
3; sbatch sb.OpenMP-4;";
    help = system(g2);
    char g3[100] =
        "sbatch sb.Pthreads-1; sbatch sb.Pthreads-2; sbatch sb.Pthreads-
3; sbatch sb.Pthreads-4;";
    help = system(g3);
    */
    int help;
    char command[100] =
        "sbatch sb.MPI-8; sbatch sb.MPI-16; sbatch sb.MPI-32;";// sbatch sb.MPI-
4;";
    help = system(command);
    char g2[100] =
        "sbatch sb.OpenMP-8; sbatch sb.OpenMP-16; sbatch sb.OpenMP-32;";
    help = system(g2);
    char g3[100] =
        "sbatch sb.Pthreads-8; sbatch sb.Pthreads-16; sbatch sb.Pthreads-32;";
    help = system(g3);
    return 0;
}
```

This is the testall.c program which can be called to run all the batch programs as needed. As you can see by the comments, we started with 1,2,3,4 cores, and realized we needed to go up to 32, so at the bottom we added more scripts to handle that.

Pthreads.c

```c
#include <pthread.h>//p4 addition
```

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/resource.h>

//p4 start
#define STRING_SIZE 2001 // no lines larger than 2000 chars
//int STRING_SIZE = 2001;
//#define CPU_NUM 8
int cpu_num = 0;
#define ARRAY_SIZE 1000000//was 1000
//int ARRAY_SIZE = 0;
int NUM_THREADS = 0;
//int ARRAY_SIZE = 1000;//was 10.31pm

pthread_mutex_t mutexsum;

char line_array[ARRAY_SIZE][STRING_SIZE];
float line_avg[ARRAY_SIZE];         // count of individual characters
//char *line_array;
//float *line_avg;

//p4 stop

float find_avg(char* line, int nchars) {
   int i, j;
   float sum = 0.0;

   for ( i = 0; i < nchars; i++ ) {
      sum += ((int) line[i]);
   }

   if (nchars > 0)
        return sum / (float) nchars;
   else
        return 0.0;
}

//p4 start
void init_arrays()
{
  int i, j, err;
  FILE *fd;

   fd = fopen( "/homes/dan/625/wiki_dump.txt", "r" );
```

```
   for ( i = 0; i < ARRAY_SIZE; i++ )  {
      err = fscanf( fd, "%[^\n]\n", line_array[i]);
      if( err == EOF ) break;
   }

  for ( i = 0; i < ARRAY_SIZE; i++ ) {
   line_avg[i] = 0.0;
  }
}

void *count_array(void *myID)
{
  char theChar;
  int i, j, charLoc;
  float local_line_avg[ARRAY_SIZE];

  int startPos = ((int) myID) * (ARRAY_SIZE / NUM_THREADS);
  int endPos = startPos + (ARRAY_SIZE / NUM_THREADS);

  printf("myID = %d startPos = %d endPos = %d \n", (int) myID, startPos, endPos);

              // init local count array
  for ( i = 0; i < ARRAY_SIZE; i++ ) {
   local_line_avg[i] = 0.0;
  }
              // count up our section of the global array
  for ( i = startPos; i < endPos; i++) {
     local_line_avg[i]=find_avg(line_array[i], strlen(line_array[i]));
  }
              // sum up the partial counts into the global arrays
  pthread_mutex_lock (&mutexsum);
   for ( i = 0; i < ARRAY_SIZE; i++ ) {
      line_avg[i] += local_line_avg[i];
   }
  pthread_mutex_unlock (&mutexsum);

  pthread_exit(NULL);
}

void print_results(float the_line_avg[])
{
  int i = 0;

              // then print out the totals
  for ( i = 0; i < ARRAY_SIZE; i++ ) {
```

```
   printf("%d: %.1f\n", i, the_line_avg[i]);
  }
}
//p4 stop

main(int argc, char* argv[])
{
   //printf("####\n");
   //printf("%s\n", (char *)argv[1]);     //2
   //printf("%d\n", strtol(argv[1],NULL, 10));  //2

   // ./Pthreads cpu_num NUM_THREADS
   cpu_num = strtol(argv[1], NULL, 10);
   printf("cpu_num: %d\n", cpu_num);
   NUM_THREADS = strtol(argv[2], NULL, 10);//was a static 4
   printf("num_threads: %d\n", NUM_THREADS);
   //ARRAY_SIZE = strtol(argv[2], NULL, 10);//1000000
   //printf("array_size: %d\n", ARRAY_SIZE);

   //line_array = (char *)malloc(ARRAY_SIZE * STRING_SIZE * sizeof(char));
   //line_avg = malloc(sizeof(float)*ARRAY_SIZE);

   //printf("####\n");

   clock_t begin = clock();//p4

   int a;
   for (a = 0; a < sizeof(argv); a++) {
      printf("%s", argv[a]);
   }
   printf("%d", argc);

   //int nlines = 0, maxlines = 1000000;
   int i, j, err, rc;
   //p4 start
   pthread_t threads[NUM_THREADS];
   pthread_attr_t attr;

   struct timeval t1, t2;
   double elapsedTime;
   gettimeofday(&t1, NULL);

   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
```

```
   init_arrays();


   //p4 test start
   for (i = 0; i < NUM_THREADS; i++ ) {
      rc = pthread_create(&threads[i], &attr, count_array, (void *)i);
      if (rc) {
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }

   for (j = 0; j < NUM_THREADS; j++)
       pthread_join(threads[j], NULL);

   print_results(line_avg);

   clock_t end = clock();

   double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;

   gettimeofday(&t2, NULL);
   elapsedTime = (t2.tv_sec - t1.tv_sec) * 1000.0; //sec to ms
   elapsedTime += (t2.tv_usec - t1.tv_usec) / 1000.0; // us to ms
   printf("DATA, %s, %f\n", getenv("SLURM_NTASKS"),  elapsedTime);

   //cpu efficiency=cpu_time / (run_time x number_of_cpus)
   printf("CPU efficiency: %f\n", elapsedTime / (elapsedTime * cpu_num));
```

OpenMP.c

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <sys/time.h>

//#define NUM_THREADS 1//changed 5.04.2021 and 4.30pm
int NUM_THREADS = 0;

#define ARRAY_SIZE 1000000 //Numeber of lines
#define STRING_SIZE 2001 //size of lines
//#define CPU_NUM 8//number of cpus
int cpu_num = 0;
```

```
char line_array[ARRAY_SIZE][STRING_SIZE];
float line_avg[ARRAY_SIZE];         // count of individual characters


void init_arrays()
{
  int i, j, err;
  FILE *fd;

   fd = fopen( "/homes/dan/625/wiki_dump.txt", "r" );
   for ( i = 0; i < ARRAY_SIZE; i++ )  {
      err = fscanf( fd, "%[^\n]\n", line_array[i]);
      if( err == EOF ) break;
   }

  for ( i = 0; i < ARRAY_SIZE; i++ ) {
    line_avg[i] = 0.0;
  }
}

float find_avg(char* line, int nchars) {
   int i, j;
   float sum = 0;

   for ( i = 0; i < nchars; i++ ) {
      sum += ((int) line[i]);
   }

   if (nchars > 0)
        return sum / (float) nchars;
   else
        return 0.0;
}

void *count_array(int myID)
{
    char theChar;
    int i, j, charLoc;
    float local_line_avg[ARRAY_SIZE];
    int startPos, endPos;

    #pragma omp private(myID,theChar,charLoc,local_char_count,startPos,endPos,i,j
)
    {
```

```
        startPos = myID * (ARRAY_SIZE / NUM_THREADS);
        endPos = startPos + (ARRAY_SIZE / NUM_THREADS);

        printf("myID = %d startPos = %d endPos = %d \n", myID, startPos, endPos);

                        // init local count array
        for ( i = 0; i < ARRAY_SIZE; i++ ) {
            local_line_avg[i] = 0.0;
        }
                        // count up our section of the global array
        for ( i = startPos; i < endPos; i++) {
            local_line_avg[i]=find_avg(line_array[i], strlen(line_array[i]));
        }

                        // sum up the partial counts into the global arrays
        #pragma omp critical
        {
            for ( i = 0; i < ARRAY_SIZE; i++ ) {
                line_avg[i] += local_line_avg[i];
            }
        }
    }
}

void print_results(float the_line_avg[])
{
  int i,j, total = 0;

  // then print out the totals
  for ( i = 0; i < ARRAY_SIZE; i++ ) {
    printf("%d: %.1f\n", i, the_line_avg[i]);
  }
}

main(int argc, char* argv[]) {

    //clock_t begin = clock();
    cpu_num = strtol(argv[1], NULL, 10);
    printf("cpu_num: %d\n", cpu_num);
    NUM_THREADS = strtol(argv[2], NULL, 10);
    printf("num_threads: %d\n", NUM_THREADS);


    struct timeval t1, t2;
    double elapsedTime;
```

```
    gettimeofday(&t1, NULL);

    omp_set_num_threads(NUM_THREADS);

    init_arrays();

    #pragma omp parallel
    {
        count_array(omp_get_thread_num());
    }

    print_results(line_avg);

    //clock_t end = clock();
    //double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;

    gettimeofday(&t2, NULL);
    elapsedTime = (t2.tv_sec - t1.tv_sec) * 1000.0; //sec to ms
    elapsedTime += (t2.tv_usec - t1.tv_usec) / 1000.0; // us to ms
    printf("DATA, %s, %f\n", getenv("SLURM_NTASKS"),  elapsedTime);

    //cpu efficiency=cpu_time / (run_time x number_of_cpus)
    printf("CPU efficiency: %f\n", elapsedTime / (elapsedTime * cpu_num));

    //printf("Main: program completed. Exiting.\n");
}
```

MPI.c

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>

//#define NUM_THREADS 4
int NUM_THREADS;

#define ARRAY_SIZE 1000000
#define STRING_SIZE 2001
//#define CPU_NUM 8
int cpu_num = 0;
int set = 0;//helps to initialize the cpu_num
```

```
char line_array[ARRAY_SIZE][STRING_SIZE];
float line_avg[ARRAY_SIZE];     // count of individual characters
float local_line_avg[ARRAY_SIZE];

void init_arrays()
{
  int i, j, err;
  FILE *fd;

   fd = fopen( "/homes/dan/625/wiki_dump.txt", "r" );
   for ( i = 0; i < ARRAY_SIZE; i++ )  {
      err = fscanf( fd, "%[^\n]\n", line_array[i]);
      if( err == EOF ) break;
   }

  for ( i = 0; i < ARRAY_SIZE; i++ ) {
    line_avg[i] = 0.0;
  }
}

float find_avg(char* line, int nchars) {
   int i, j;
   float sum = 0;

   for ( i = 0; i < nchars; i++ ) {
      sum += ((int) line[i]);
   }

   if (nchars > 0)
        return sum / (float) nchars;
   else
        return 0.0;
}

void *count_array(void *rank)
{
  char theChar;
  int i, j, charLoc;
  int myID =  *((int*) rank);

  int startPos = ((long) myID) * (ARRAY_SIZE / NUM_THREADS);
  int endPos = startPos + (ARRAY_SIZE / NUM_THREADS);

  printf("myID = %d startPos = %d endPos = %d \n", myID, startPos, endPos); fflus
h(stdout);
```

```
    // init local count array
  for ( i = 0; i < ARRAY_SIZE; i++ ) {
    local_line_avg[i] = 0.0;
  }

    // count up our section of the global array
  for ( i = startPos; i < endPos; i++) {
     local_line_avg[i]=find_avg(line_array[i], strlen(line_array[i]));
  }
}

void print_results(float the_line_avg[])
{
  int i,j, total = 0;

  // then print out the totals
  for ( i = 0; i < ARRAY_SIZE; i++ ) {
  printf("%d: %.1f\n", i, the_line_avg[i]);
  }
}

main(int argc, char* argv[])
{

  int i, rc;
  int numtasks, rank;
    //mpirun -np 2 MPI 5
    //printf("###%d###\n", strtol(argv[0],NULL, 10));
    //printf("###%d###\n", strtol(argv[1],NULL, 10));//5
    //printf("###%d###\n", strtol(argv[3],NULL, 10));
    //printf("###%d###\n", strtol(argv[4],NULL, 10));
    cpu_num = strtol(argv[1], NULL, 10);

  MPI_Status Status;

    struct timeval t1, t2;
    double elapsedTime;
    gettimeofday(&t1, NULL);

  rc = MPI_Init(&argc,&argv);
  if (rc != MPI_SUCCESS)
    {
      printf ("Error starting MPI program. Terminating.\n");
        MPI_Abort(MPI_COMM_WORLD, rc);
```

```
    }

    MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  NUM_THREADS = numtasks;
  printf("size = %d rank = %d\n", numtasks, rank);
  fflush(stdout);

  if ( rank == 0 ) {
    init_arrays();
  }
  MPI_Bcast(line_array, ARRAY_SIZE * STRING_SIZE, MPI_CHAR, 0, MPI_COMM_WORLD);

  count_array(&rank);

  //MPI_Reduce(local_char_count, char_counts, ALPHABET_SIZE, MPI_INT, MPI_SUM, 0,
 MPI_COMM_WORLD);
  MPI_Reduce(local_line_avg, line_avg, ARRAY_SIZE, MPI_INT, MPI_SUM, 0, MPI_COMM_
WORLD);



  if ( rank == 0 ) {
    print_results(line_avg);

    gettimeofday(&t2, NULL);
    elapsedTime = (t2.tv_sec - t1.tv_sec) * 1000.0; //sec to ms
    elapsedTime += (t2.tv_usec - t1.tv_usec) / 1000.0; // us to ms
    printf("DATA, %s, %f\n", getenv("SLURM_NTASKS"),  elapsedTime);

    //cpu efficiency=cpu_time / (run_time x number_of_cpus)
    printf("CPU efficiency: %f\n", elapsedTime / (elapsedTime * cpu_num));

    //printf("Main: program completed. Exiting.\n");

    /*
    int help;
    char command[20] = "./Pthreads";
    help = system(command);
    */
  }
```

```
  MPI_Finalize();
  return 0;
}
```