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.

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.)

#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**

#include <pthread.h>*//p4 addition*

#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); fflush(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;

}