CIS520 Project 4

Jerry Rizzo

Connor Neil

Robert Ault

**OpenMP:**

**Software Architecture:**

The openmp.c file stores the necessary data in three different global arrays. First is a two-dimensional character array called char\_array. This will store the characters from the wiki\_dump.txt file. Next is an integer array called char\_sums, that will store the sum of the ASCII character numeric values of each character for each line. Lastly is another integer array called line\_lengths, which will store the number of characters for each line.

The main function begins by filling the arrays with data with a call to init\_arrays. This function reads in the wiki\_dump.txt file and stores its characters into char\_array. The number of characters for each line is stored in the line\_lengths array, and char\_sums is filled with zeros.

Next the main function counts the ASCII values for each character in char\_array with a call to count\_array. Parallelization is achieved by collapsing a two-layer nested for loop, which iterates through each character in each line string. Synchronization is maintained by setting the array indexes, integers i and j, as private variables for each thread. The number of threads used can also be changed at this line. The char\_sums array value at each i index is incremented by the numeric value of each character in its line. This is achieved by casting the character value to an integer value.

The main function then calls print\_results, which will interate over the number of lines that were read in and print the line number (starting at 0) and the average numeric value of each line by dividing the char\_sums value by the line\_lengths value at each index. The program avoids a divide by zero condition by checking that each line\_length value is greater than zero. If the value is zero due to a blank line, a zero value will be printed for that line.

For performance analysis purposes, the time is recorded before this process starts and after it ends. The total elapsed time is printed, as well as the number of cores used, and the virtual and physical memory utilization. Note that during analysis the print\_results function is commented out. This is to avoid the output files made by the shell scripts being full of the many lines of output information. Also, the printing is done serially anyway, so its performance time should be the same regardless of the number of threads.

**Performance Analysis:**

Memory varying the number of lines read from the input file:

Non-varying factors:

1 machine

1 core per machine

4 threads

Performed on node elf15

Varying Number of Threads and Cores:

Performed on one machine.

All nodes were elf class 1-56

Increasing the number of threads and cores improves performance time, with the optimum performance happening with 16 of each.

**Pthreads:**

Pthreads.c implements batching of line average calculations by splitting the lines to average up evenly between however many threads is set by the 1st command line argument for the program. Each thread ends up iterating over an equivalent amount of lines albeit the last thread to get assigned lines may get less if the values do not divide evenly as it will not exceed the actual\_lines value that tracks how many lines were actually read from a file. Each thread will be working with a different part of the input\_array and line\_avgs so there will be no synchronization issues with one thread overwriting another. The actual reading of the file is done by the initial thread and is not parallelized, and this also goes for the output of the results to preserve order. Initial reading could potentially be parallelized but was decided to be beyond the scope of what we looked to accomplish.

Since all threads have access to the same memory locations for global variables, little communication is needed between threads and they simply operate on those globals so changes will be reflected in main after pthread\_join, which is used to avoid race conditions so that output does not occur until all threads are complete.

**MPI:**

Mpi.c uses MPI to parse through the file in parallel. Because we do not know the length of each line, we split the file up into “chunks” of data that each thread takes a look at, which each start at different points in the file based on its size. They look at the data, and if they reach a number of characters that surpasses the number stored in “overlap” without reaching a ‘\n’ character, the data for that line is thrown out, as we know it is a line being handled elsewhere by a different thread. Then, everyone reads in their part. Race conditions are prevented by performing the actual MPI\_read\_at\_all function call at the end, because by that point we have split the file into the specific chunks so that the threads will not overlap with each other.

**Appendix:**

Source Code:

openmp.c

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <omp.h>

#include <sys/time.h>

#include <stdint.h>

#include "sys/types.h"

#include "sys/sysinfo.h"

#define ARRAY\_SIZE 1000000

#define STRING\_SIZE 2001

char char\_array[ARRAY\_SIZE][STRING\_SIZE];

int char\_sums[ARRAY\_SIZE];

int line\_lengths[ARRAY\_SIZE];

typedef struct {

uint32\_t virtualMem;

uint32\_t physicalMem;

} processMem\_t;

int parseLine(char\* line) {

// This assumes that a digit will be found and the line ends in " Kb".

int i = strlen(line);

const char\* p = line;

while (\*p < '0' || \*p > '9') p++;

line[i - 3] = '\0';

i = atoi(p);

return i;

}

void GetProcessMemory(processMem\_t\* processMem) {

FILE\* file = fopen("/proc/self/status", "r");

char line[128];

while (fgets(line, 128, file) != NULL) {

//printf("%s", line);

if (strncmp(line, "VmSize:", 7) == 0) {

processMem->virtualMem = parseLine(line);

}

if (strncmp(line, "VmRSS:", 6) == 0) {

processMem->physicalMem = parseLine(line);

}

}

fclose(file);

}

void init\_arrays()

{

int i, err;

FILE\* fd;

// Read in the lines from the data file

fd = fopen("/homes/dan/625/wiki\_dump.txt", "r");

for (i = 0; i < ARRAY\_SIZE; i++) {

err = fscanf(fd, "%[^\n]\n", char\_array[i]);

if (err == EOF) break;

line\_lengths[i] = strlen(char\_array[i]);

char\_sums[i] = 0;

}

fclose(fd);

}

void count\_array()

{

int i, j;

#pragma omp parallel private(i,j) num\_threads(4)

{

#pragma omp for collapse(2)

for (i = 0; i < ARRAY\_SIZE; i++) {

for (j = 0; j < STRING\_SIZE; j++) {

char\_sums[i] += ((int)char\_array[i][j]);

}

}

} //omp parallel

}

void print\_results()

{

int i;

float total = 0;

for (i = 0; i < ARRAY\_SIZE; i++) {

if (line\_lengths[i] > 0)

total = (char\_sums[i]) / ((float)line\_lengths[i]);

else

total = 0;

printf("%d: %.1f\n", i, total);

}

}

int main()

{

int i, j = 0;

struct timeval t1, t2;

double elapsedTime;

int numSlots, myVersion = 1;

processMem\_t myMem;

gettimeofday(&t1, NULL);

printf("DEBUG: starting loop on %s\n", getenv("HOSTNAME"));

// Fill arrays with lines from file, line lengths, or zeros

init\_arrays();

// Count values of the characters

count\_array();

// Print out the results

print\_results();

gettimeofday(&t2, NULL);

GetProcessMemory(&myMem);

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, %d, %s, %f, %u, %u\n", myVersion, getenv("SLURM\_NTASKS"), elapsedTime, myMem.virtualMem, myMem.physicalMem);

}

openmp\_sbatch.sh

#!/bin/bash -l

##$ -l h\_rt=0:00:01 # ask for 1 minute runtime

/homes/jrizzo/cis520Proj4/3way-openmp/openmp #change to match the path to your code

mass\_sbatch.sh

#!/bin/bash

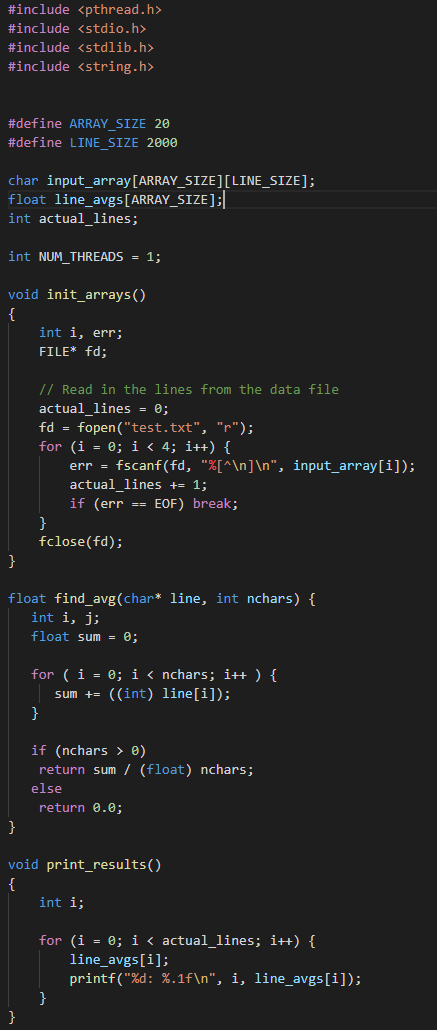
for i in 1 2 4 8 16

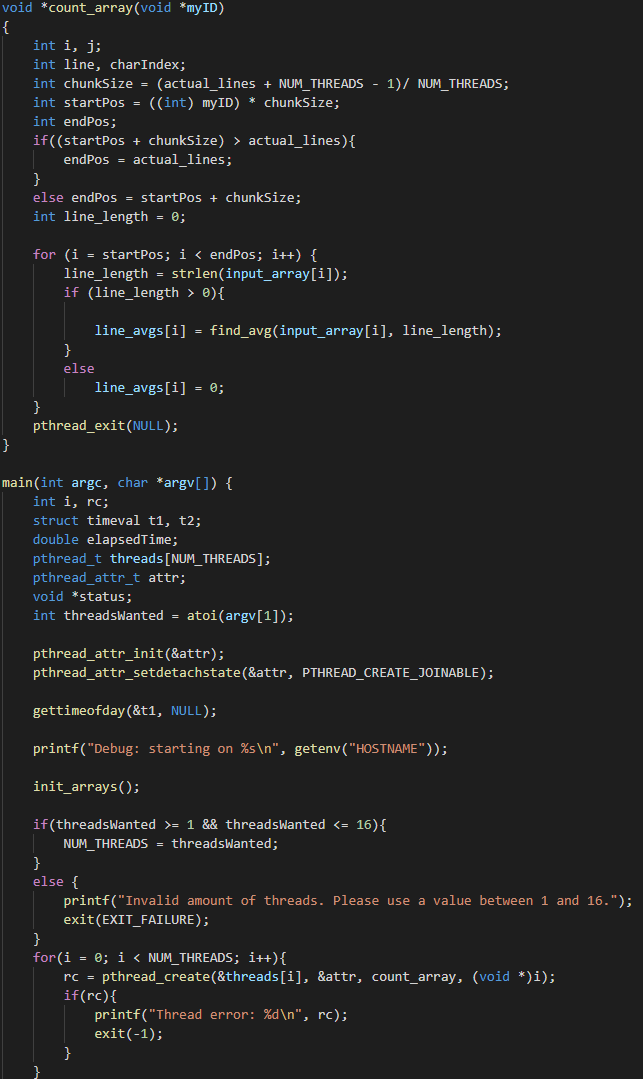
do

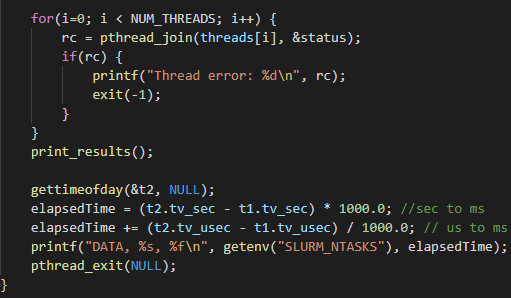
sbatch --nodes=1 --constraint=elves --mem-per-cpu=1500M --ntasks-per-node=$i --nodes=1 openmp\_sbatch.sh

done

pthreads.c







mpi.c

|  |
| --- |
| #include <mpi.h>  #include <stdlib.h>  #include <stdio.h>  #include <string.h>  #include <sys/time.h>  #include <stdint.h>  #include <ctype.h>  #include <string.h>  #include "sys/types.h"  #include "sys/sysinfo.h"  void parprocess(MPI\_File \*in, const int rank, const int size, const int overlap)  {  MPI\_Offset globalstart;  int mysize;  char \*chunk;    MPI\_Offset globalend;  MPI\_Offset filesize;    MPI\_File\_get\_size(\*in, &filesize);  filesize--;  mysize = filesize / size;  globalstart = rank \* mysize;  globalend = globalstart + mysize - 1;    if (rank == size - 1)  {  globalend = filesize - 1;  }    if (rank != size - 1)  {  globalend += overlap;  }    mysize = globalend - globalstart + 1;    chunk = malloc((mysize + 1) \* sizeof(char));    MPI\_File\_read\_at\_all(\*in, globalstart, chunk, mysize, MPI\_CHAR, MPI\_STATUS\_IGNORE);  chunk[mysize] = '\0';    int locstart = 0;  int locend = mysize - 1;    if (rank != 0)  {  locend -= overlap;  while (chunk[locend] != '\n')  {  locend++;  }  }  mysize = locend - locstart + 1;    int total = 0;  int count = 0;  int average;    for (int i = locstart; i <= locend; i++ )  {  total += chunk[i];  count++;  }    average = total / count;    printf("%d", average);    return;  }  int main(int argc, char \*argv[])  {  MPI\_File in;  int numtasks, taskid, len;  const int overlap = 100;  char hostname[MPI\_MAX\_PROCESSOR\_NAME];  MPI\_Init(&argc, &argv);  MPI\_Comm\_size(MPI\_COMM\_WORLD, &numtasks);  MPI\_Comm\_rank(MPI\_COMM\_WORLD,&taskid);  MPI\_Get\_processor\_name(hostname, &len);    MPI\_File\_open(MPI\_COMM\_WORLD, "/homes/dan/625/wiki\_dump.txt", MPI\_MODE\_RDONLY, MPI\_INFO\_NULL, &in);    parprocess(&in, taskid, numtasks, overlap);    MPI\_File\_close(&in);    MPI\_Finalize();  return 0;  } |