USING THREE METHODS OF PARALLELIZATION

Project 4
CIS 520

Abdulkareem Alkhiary [araasai@ksu.edu]
Jack Johnson [jjohn98@ksu.edu]
Cody Morse [cody598@ksu.edu]

Table of Contents

Abstract

Implementing Parallelization can make programs run much faster than not implementing parallelization. Parallelization is quite important when reading data in from a file line by line to gather data, as it is the most efficient and less costly solution. By using three different implementations of parallelization: pThreads, MPI, and OpenMP. The outputs pThreads and OpenMP are much alike as they follow the same graphical behavior.

Introduction

We had initially written a program that reads in a 1.7GB file, named "wiki_dump.txt", which takes each line of the file and computes the mean value of ASCII character numeric values for all the characters in a line. The program was written based on the code provided in the "/625" folder. The mean value is then outputted to a file. The issue we came to realize was the program was inefficient. This implementation was done without using any form of parallelization and posed to be very inefficient.

Without using parallelization, the program was slow but could be done in under roughly five minutes, based on how many lines were being read in and computed. The heavy job of reading in a large file and computing the mean of each line could pose issues if the data is needed to be retrieved quickly for alternative computations. We came up with a solution to increase the speed of determining the mean ASCII average for each line in a file. The three programs created are OpenMP, PThreads, and MPI, which dramatically decreased the computation time.

Utilizing our created programs using MPI, PThreads, and OpenMP implementations, the most efficient method was easy to determine based on runtimes. The three programs have all reduced overall runtimes when comparing them to the runtime of a non-parallelized program. The runtimes for each of the methods allowed us to determine the most efficient method for reading in a larger file. Based on the data we collected from each of the methods, our team concluded that PThreads is the best for our implementation as it had a considerably less runtime than its second-place rival OpenMP.

Implementation

POSIX Threads, also called pthreads, is a method used for parallel programming. There are four types of functions used when dealing with PThreads. The ones used are condition variables, mutexes, creation of and joining threads, and thread synchronization. We utilized barrier synchronization for all of our implementations, ensuring that no thread can move on before all of the others have caught up.

OpenMP is a much different method as part of the program runs like any other program, then the important parts are multi-threaded. OpenMP also stands for Open-Multi-Processing. This method uses three different methods: variables defining parallel parameters (e.g. NUM_THREADS), directives for programs, and runtime library. OpenMP is incredibly easy to use and makes parallelizing serial code trivial, as stated by Dr. Andresen many times. To avoid race conditions, the critical section of the code is only allowed to be run by one thread at a time as specified by the #pragma critical.

MPI, a method implementing message passing, is short for Message Passing Interface. There are two functions used in MPI, both send and receive. The ability to send gives threads the ability to communicate between them all. Sending can be done either asynchronously or synchronously. If synchronous, the thread waits for the receiving part to send a notification. If asynchronous, the “message” is placed into a queue, thus freeing the sending thread to do whatever work needs to be done. A thread receiving the message also sends a notification back to the sender.

Evaluation

Our group's methods for testing included submitting multiple jobs to Beocat, the on-campus supercomputer at Kansas State University. We began testing our program using the three implementations stated above. We used a variety of core counts and node counts depending on the specific job. For OpenMP and pthreads we used 1, 2, 4, 8, and 16 cores. For MPI, we used 1, 2, 4, and 8 cores spread across up to 3 different nodes. We tested 4 different job sizes: 10k, 100k, 500k, and 1 million lines. Through the runtime outputs, pthreads and OpenMP perform similarly at smaller data sets, but pthreads was faster for 10k-1million lines of computation. MPI on the other hand outputted inconsistent results as it varied from node count, core count, and job size. MPI requires more memory, and the computations took a significantly larger amount of time for the larger data sets.

In Figures 1 and 2, it can be deduced that as the job size increases (increase in the number of lines), the total time elapsed will increase as well. In addition, if we were to assign more threads to do work, the total elapsed time for the job submission decreases as well. We expected these results as modern-day computers are able to multitask efficiently. The data to verify is shown in their respective tables below: Table 1 and Table 2. In Figures 3-6 the data is hard to draw conclusions from because it varies so much from node count, core count, and job size.

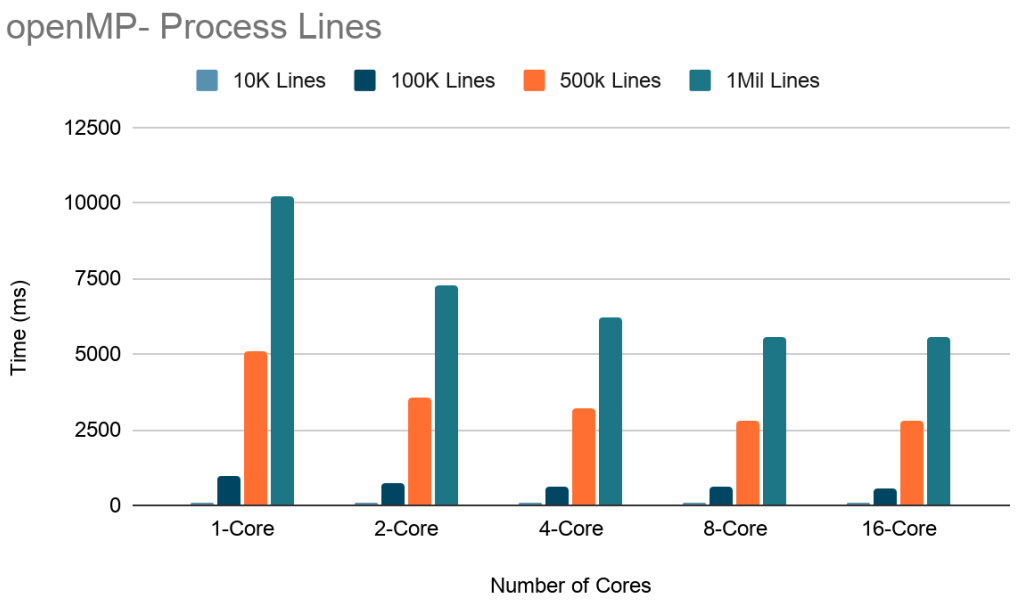

Figure 1: OpenMP time comparison after parallelization

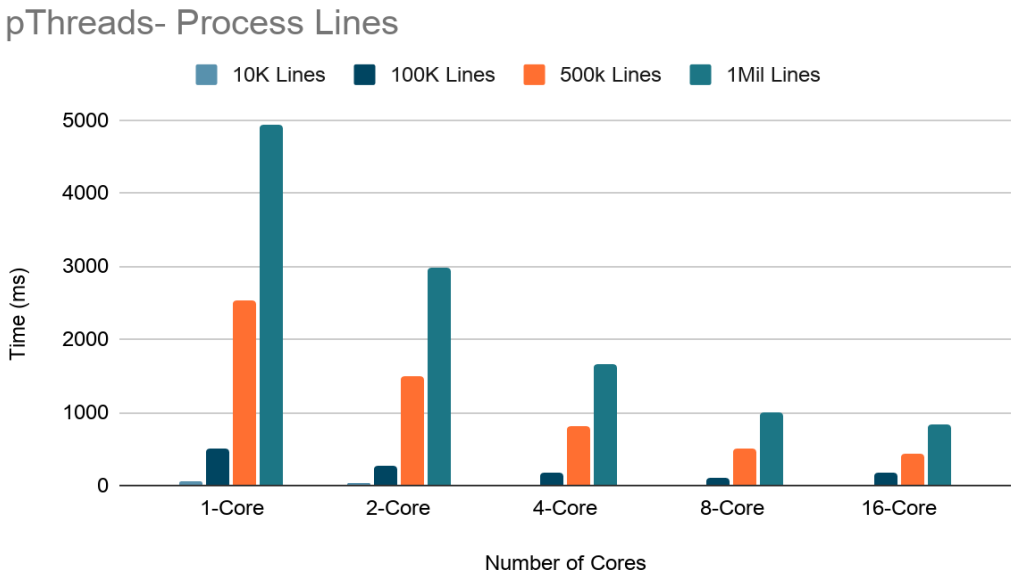

Figure 2: pThreads time comparison after parallelization

Threads	Number of Lines	Elapsed Time (millisec)
1	10000	99.020
1	100000	972.485
1	500000	5089.394
1	1000000	10230.377
2	10000	79.334
2	100000	761.713
2	500000	3757.957
2	1000000	7287.501
4	10000	66.711
4	100000	629.514
4	500000	3209.591
4	1000000	6239.766
8	10000	59.768000
8	100000	599.251
8	500000	2791.615
8	1000000	5576.671
16	10000	73.493000
16	100000	572.831
16	500000	2810.037
16	1000000	5561.671

Table 1: OpenMP Output Data

Threads	Number of Lines	Elapsed Time (millisec)
1	10000	54.530000
1	100000	502.586000
1	500000	2532.506000
1	1000000	4939.188000
2	10000	27.754000
2	100000	271.228000
2	500000	1485.230000
2	1000000	2969.615000
4	10000	16.932000
4	100000	165.345000
4	500000	813.200000
4	1000000	1660.007000
8	10000	13.454000
8	100000	102.289000
8	500000	511.150000
8	1000000	1009.289000
16	10000	14.377000
16	100000	168.956000
16	500000	445.025000
16	1000000	836.523000

Table 2: pThreads Output Data

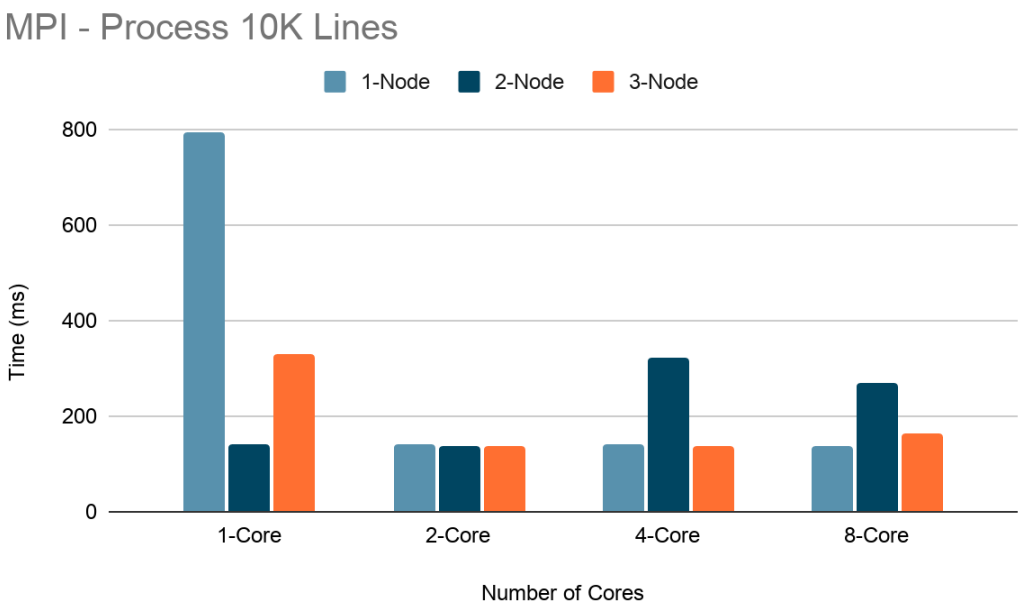

Figure 3: MPI time comparison after parallelization (10k lines)

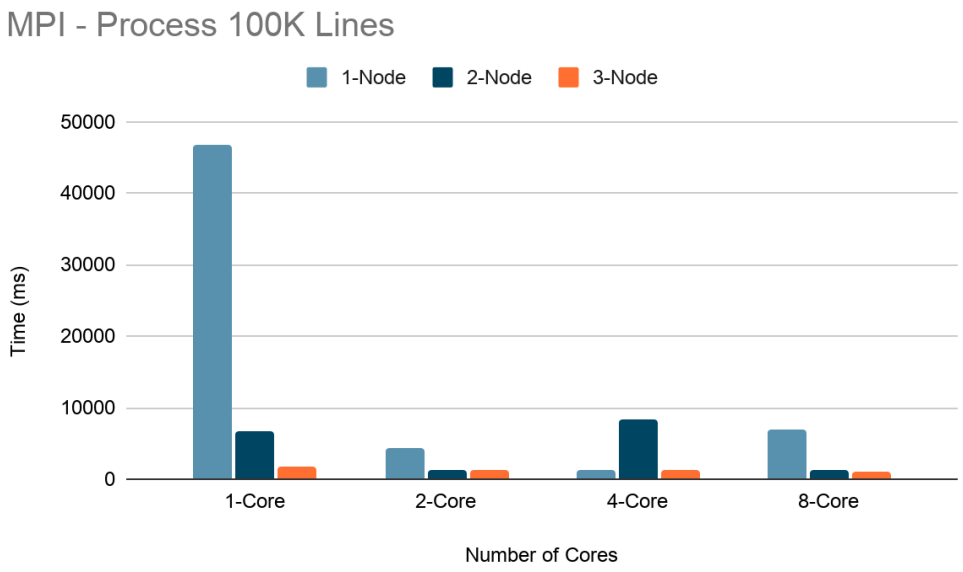

Figure 4: MPI time comparison after parallelization (100k lines)

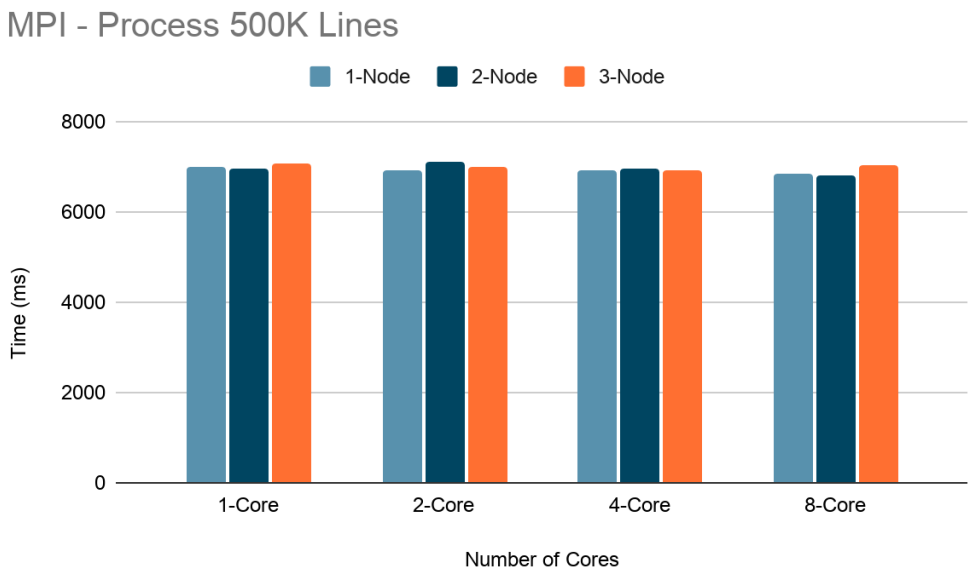

Figure 5: MPI time comparison after parallelization (500k lines)

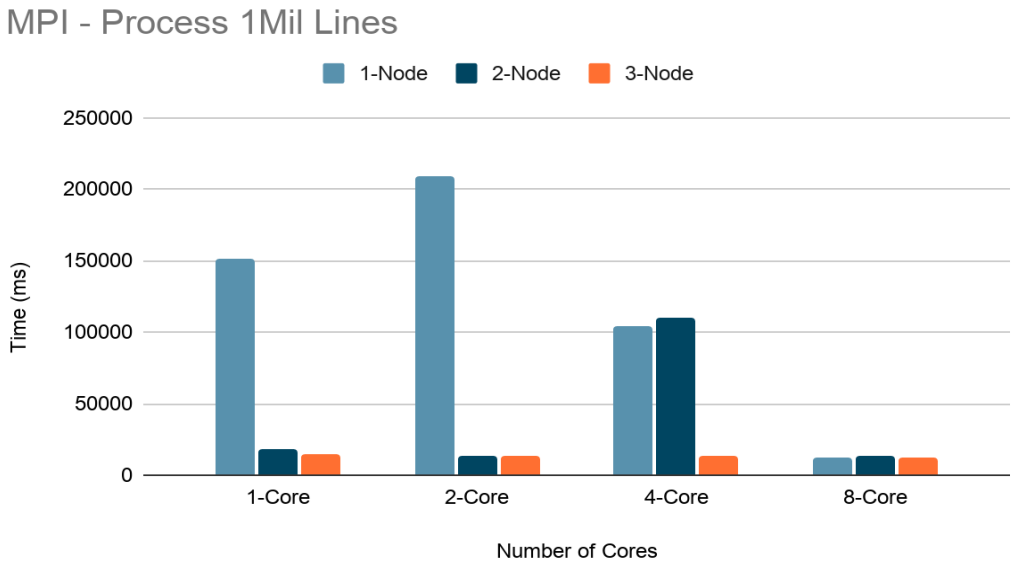

Figure 6: MPI time comparison after parallelization (1M lines)

Conclusion

After finishing the project using the three different methods of parallelization, we have found that OpenMP was by far the easiest to implement. Following this, we found that pthreads gave the fastest resulting time and MPI was able to utilize more cores across multiple nodes. While MPI does parallelize the best, it does run slower than both of the other two methods the best. The reason MPI runs slower is probably due to its shared memory properties.

We found that the impact of multiple cores on OpenMP gave relatively good performance for the amount of effort required to implement it, while pThreads always had the fastest output times. MPI, while being able to spread its workload out the most, did not have the performance we were expecting given this ability.

We greatly appreciate the help of Dr. Andreson in solving multiple issues that arose when completing the pthreads part of the project. We likely would not have been as successful in our findings without him.

Appendix

For each of the shell scripts, there are multiple variations within the github repository. Each variation will run the program with a different number of lines to process.

OpenMP:

https://github.com/cody598/Project4/tree/master/3way-openMP

```
/* Finds the average values of read character lines from file.
   openMP - Parallel
   Project 4 - Team 20
*/

#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <sys/time.h>
#include "sys/types.h"
#include "sys/sysinfo.h"

#define ARRAY_SIZE 1000000
#define STRING_SIZE 2001

int NUM_THREADS = 4;
float line_avg[ARRAY_SIZE];
char lines[ARRAY_SIZE][STRING_SIZE];
FILE *fd;

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

//Finds the average of the characters
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;
}


//Removed file pointer argument in this version as it seems unnecessary/unused.
void *count_array(int myID)
{
    int i, startPos, endPos;
    float local_line_avg[ARRAY_SIZE];
    //Directive that instructs each thread to keep its own copy of function's private variables.
    #pragma omp private(myID, i, startPos, endPos, local_line_avg)
    {
        startPos = ((int) myID) * (ARRAY_SIZE / NUM_THREADS);
        endPos = startPos + (ARRAY_SIZE / NUM_THREADS);
        printf("myID = %d startPos = %d endPos = %d \n", (int) myID, startPos, endPos);

        //initialize local line average
        for ( i = 0; i < ARRAY_SIZE; i++ )
        {
              local_line_avg[i] = 0.0;
         }

        for ( i = startPos; i < endPos; i++) {
            local_line_avg[i] = find_avg(lines[i], strlen(lines[i]));
        }

 //Critical section is indicated here, should wrap the addition of all local character counts to the
global.
        #pragma omp critical
        {
```

```
            for ( i = startPos; i < endPos; i++) {
               line_avg[i] += local_line_avg[i];
            }
         }
      }
   }

void print_results(float lineavg[])
{
   int i;
   for(i = 0; i<ARRAY_SIZE; i++)
   {
         printf("%d: %.1f\n", i, lineavg[i]);
   }
}

main()
{
   // Sets the number of threads.
   NUM_THREADS = atoi(getenv("SLURM_NTASKS"));
   omp_set_num_threads(NUM_THREADS);
   struct timeval t1, t2, t3, t4;
   double timeElapsedInit, timeElapsedProcess, timeElapsedPrint, timeElapsedTotal;
   processMem_t memory;

   /* Timing analysis begins */
   gettimeofday(&t1, NULL);

   readFile();

   gettimeofday(&t2,NULL); //t2 - t1

   #pragma omp parallel
   {
      count_array(omp_get_thread_num());
   }
  GetProcessMemory(&memory);
   gettimeofday(&t3, NULL); //t3 - t2
```

```
    print_results(line_avg);

    gettimeofday(&t4, NULL); //t4 - t1

    //total program time
    timeElapsedTotal = (t4.tv_sec - t1.tv_sec) * 1000.0; //Convert to to milliseconds
    timeElapsedTotal += (t4.tv_usec - t1.tv_usec) / 1000.0;

    printf("Tasks: %s\nTotal Time: %fms\n", getenv("SLURM_NTASKS"), timeElapsedTotal);
    printf("DATA, %s,%f\n", getenv("SLURM_NTASKS"), timeElapsedTotal);
    printf("size = %d, Node: %s, vMem %u KB, pMem %u KB\n", NUM_THREADS,
    getenv("HOSTNAME"), memory.virtualMem, memory.physicalMem);

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

}
```

OpenMP Shell Scripts:

https://github.com/cody598/Project4/blob/master/3way-openMP/TestingOutput/output-1mil/openmp_sbatch.sh

```
#!/bin/bash -l
#SBATCH --time=0:10:00
#SBATCH --mem=100G
#SBATCH --constraint=elves

module load OpenMPI
module load foss/2020a --quiet

echo openMP

time /homes/cody598/cis520/Project4/3way-openMP/openMP-1mil
grep DATA *.out > 1-milTimes.csv
```

https://github.com/cody598/Project4/blob/master/3way-openMP/TestingOutput/output-1mil/mass_sbatch.sh

```
#!/bin/bash
#SBATCH --job-name=openMP
#SBATCH -o 3way-openMP-MASSBATCH-STATS.out
for i in 1 2 4 8 16
do
        echo "Tasks: $i"
        sbatch --constraint=elves --ntasks-per-node=$i --nodes=1 --job-name=openMP -o
$i-core-1mil.out openmp_sbatch.sh
Done
```

OpenMP First 100 Lines of Output:

```
openMP
myID = 0 startPos = 0 endPos = 10000
0: 94.1
1: 91.9
2: 92.3
3: 93.6
4: 92.2
5: 91.1
6: 92.2
7: 93.8
8: 90.6
9: 92.8
10: 90.5
11: 89.0
12: 94.1
13: 89.1
14: 90.1
15: 94.4
16: 93.2
17: 90.9
18: 92.4
19: 93.5
20: 93.6
21: 90.1
22: 90.3
23: 92.0
24: 92.5
25: 94.1
26: 94.9
27: 89.6
28: 87.8
29: 92.2
30: 92.4
31: 90.5
32: 93.6
33: 93.1
34: 82.6
```

35: 90.4
36: 89.6
37: 94.2
38: 95.0
39: 94.5
40: 84.8
41: 92.5
42: 91.8
43: 92.4
44: 90.1
45: 91.6
46: 90.3
47: 84.3
48: 95.8
49: 92.4
50: 94.3
51: 91.8
52: 91.3
53: 86.9
54: 90.9
55: 93.3
56: 94.3
57: 89.5
58: 90.5
59: 91.8
60: 94.1
61: 81.9
62: 92.6
63: 94.4
64: 92.9
65: 93.5
66: 93.4
67: 93.2
68: 86.6
69: 85.2
70: 91.9
71: 89.0
72: 93.0
73: 87.8

74: 86.3
75: 93.9
76: 91.9
77: 92.3
78: 88.2
79: 91.8
80: 80.5
81: 90.7
82: 99.6
83: 92.4
84: 92.8
85: 81.3
86: 89.0
87: 84.1
88: 95.5
89: 93.7
90: 93.1
91: 94.6
92: 92.3
93: 91.7
94: 80.5
95: 92.2
96: 88.0
97: 90.3
98: 89.4
99: 89.7
100: 89.4

MPI:

https://github.com/cody598/Project4/blob/master/3way-mpi

```
/* Finds the average values of read character lines from file.
   MPI - Parallel
   Project 4 - Team 20
*/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/time.h>
#include <mpi.h>
#include <stdint.h>
#include "sys/types.h"
#include "sys/sysinfo.h"
#include <math.h>

#define MAXIMUM_TASKS 32
#define STRING_SIZE 2001
#define ARRAY_SIZE 1000000

/* Global variables. */
float NUM_THREADS;
unsigned int thread_locations[MAXIMUM_TASKS];
float line_averages[ARRAY_SIZE];
float local_average[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) {
                if (strncmp(line, "VmSize:", 7) == 0) {
                        processMem->virtualMem = parseLine(line);
                }

                if (strncmp(line, "VmRSS:", 6) == 0) {
                        processMem->physicalMem = parseLine(line);
                }
        }
        fclose(file);
}

/* Find average char value of a line. */
float find_line_average(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;
}

/* Computes the local average array. Work-load distributed equally.*/
void find_avg(int rank, FILE * fp)
```

```
{
	char tempBuffer[STRING_SIZE];
	int currentLine = rank * (ARRAY_SIZE/NUM_THREADS);

	fseek(fp, thread_locations[rank], SEEK_SET);

	/* While not at EOF and not beyond assigned section ... */
	while(currentLine < (rank+1) * (ARRAY_SIZE/NUM_THREADS)
		&& fscanf(fp, "%[^\n]\n", tempBuffer) != EOF)
	{
			/* Find and save average of line of char locally */
			int lineLength = strlen(tempBuffer);
			local_average[currentLine] = find_line_average(tempBuffer, lineLength);
			currentLine++;
	}
}


/* Prints the char averages. */
void printResults()
{
	int i;
	for(i = 0; i<ARRAY_SIZE; i++)
	{
		printf("%d: %.1f\n", i, line_averages[i]);
	}
}

main(int argc, char *argv[])
{
	/* Time variables. */
	struct timeval t1, t2;
	double timeElapsedTotal;

	int i, rc;
	int rank, numtasks;
	FILE * fp;
	MPI_Status Status;
	processMem_t myMem;
```

```
	/* MPI Setup. */
	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);

/* Start performance */
if(rank == 0)
{
		gettimeofday(&t1, NULL);
		printf("DEBUG: starting time on %s\n", getenv("HOSTNAME"));
}

	NUM_THREADS = numtasks;

	fp = fopen("/homes/dan/625/wiki_dump.txt","r");
	if(fp == NULL)
	{
		printf("file not found\n");
		exit(-1);
	}

	/* Distributes workload */
	if(rank == 0)
	{
		char tempBuffer[STRING_SIZE];
		i = 1;
		int currentLine = 0;

		/* Set starting position. */
		fseek(fp, 0, SEEK_SET);
		thread_locations[0] = ftell(fp);
		int prevPos = ftell(fp);
```

```
                /* Count lines and save their positions in file in the array. */
                while(currentLine < ARRAY_SIZE && fscanf(fp, "%[^\n]\n", tempBuffer) !=
EOF)
                {
                        if(currentLine == i * ARRAY_SIZE/NUM_THREADS)
                        {
                                thread_locations[i] = prevPos;
                                i++;
                        }
                        prevPos = ftell(fp);
                        currentLine++;
                }

                fseek(fp, 0, SEEK_SET);
        }
        MPI_Bcast(thread_locations, NUM_THREADS+1, MPI_UNSIGNED, 0,
MPI_COMM_WORLD);

        find_avg(rank, fp);

        fclose(fp);

        /* Merge local to global arrays */
        MPI_Reduce(local_average, line_averages, ARRAY_SIZE, MPI_FLOAT, MPI_SUM, 0,
MPI_COMM_WORLD);

        /* Print results and record important data */
        if(rank == 0)
        {
                printResults();
                gettimeofday(&t2, NULL);
                timeElapsedTotal = (t2.tv_sec - t1.tv_sec) * 1000.0; // Convert to ms
                timeElapsedTotal += (t2.tv_usec - t1.tv_usec) / 1000.0; // Convert to ms
                        /* Performance metrics. */
                GetProcessMemory(&myMem);
                printf("size = %d rank = %d, Node: %s, vMem %u KB, pMem %u KB\n",
numtasks, rank, getenv("HOSTNAME"), myMem.virtualMem, myMem.physicalMem);
                printf("Tasks: %s, Elapsed Time: %fms\n",  getenv("SLURM_NTASKS"),
timeElapsedTotal);
```

```
            printf("DATA, %s,%fms\n", getenv("SLURM_NTASKS"), timeElapsedTotal);
        }

        MPI_Finalize();
        return 0;
}
```

MPI Shell Scripts:

https://github.com/cody598/Project4/blob/master/3way-mpi/TestingOutput/output-1mil/mpi_sbatch.sh

```
#!/bin/bash -l
#SBATCH --time=0:10:00
#SBATCH --mem=3G
#SBATCH --constraint=elves

module load OpenMPI
module load foss/2020a --quiet

echo MPI

time /homes/cody598/cis520/Project4/3way-mpi/mpi-1mil
grep DATA *.out > 1-milTimes.csv
```

https://github.com/cody598/Project4/blob/master/3way-mpi/TestingOutput/output-1mil/mass_sbatch.sh

```
#!/bin/bash
#SBATCH --job-name=MPI
#SBATCH -o 3way-MPI-MASSBATCH-STATS.out
for j in 1 2 3
do
        for i in 1 2 4 8
        do
        if(($i == 8))
        then
                sleep 20
        fi
                echo "Nodes: $j, Tasks: $i"
```

```
                sbatch --constraint=elves --ntasks-per-node=$i --nodes=$j --job-name=MPI -o
$j-node-$i-core-1mil.out mpi_sbatch.sh
        done
Done
```

MPI Output First 100 Lines:

```
MPI
DEBUG: starting time on elf61
0: 94.1
1: 91.9
2: 92.3
3: 93.6
4: 92.2
5: 91.1
6: 92.2
7: 93.8
8: 90.6
9: 92.8
10: 90.5
11: 89.0
12: 94.1
13: 89.1
14: 90.1
15: 94.4
16: 93.2
17: 90.9
18: 92.4
19: 93.5
20: 93.6
21: 90.1
22: 90.3
23: 92.0
24: 92.5
25: 94.1
26: 94.9
27: 89.6
28: 87.8
29: 92.2
```

30: 92.4
31: 90.5
32: 93.6
33: 93.1
34: 82.6
35: 90.4
36: 89.6
37: 94.2
38: 95.0
39: 94.5
40: 84.8
41: 92.5
42: 91.8
43: 92.4
44: 90.1
45: 91.6
46: 90.3
47: 84.3
48: 95.8
49: 92.4
50: 94.3
51: 91.8
52: 91.3
53: 86.9
54: 90.9
55: 93.3
56: 94.3
57: 89.5
58: 90.5
59: 91.8
60: 94.1
61: 81.9
62: 92.6
63: 94.4
64: 92.9
65: 93.5
66: 93.4
67: 93.2
68: 86.6

69: 85.2
70: 91.9
71: 89.0
72: 93.0
73: 87.8
74: 86.3
75: 93.9
76: 91.9
77: 92.3
78: 88.2
79: 91.8
80: 80.5
81: 90.7
82: 99.6
83: 92.4
84: 92.8
85: 81.3
86: 89.0
87: 84.1
88: 95.5
89: 93.7
90: 93.1
91: 94.6
92: 92.3
93: 91.7
94: 80.5
95: 92.2
96: 88.0
97: 90.3
98: 89.4
99: 89.7
100: 89.4

pThreads:

https://github.com/cody598/Project4/blob/master/3way-pthread/

```
/* Finds the average values of read character lines from file.
   pThreads - Parallel
   Project 4 - Team 20
*/

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <sys/time.h>
#include "sys/types.h"
#include "sys/sysinfo.h"


#define ARRAY_SIZE 1000000
#define STRING_SIZE 2001
#define ALPHABET_SIZE 26

pthread_mutex_t mutexsum;                       // mutex for char_counts
int NUM_THREADS = 2;
float line_avg[ARRAY_SIZE];                             // count of individual characters
char lines[ARRAY_SIZE][STRING_SIZE];
FILE *fd;

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

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);
}

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, startPos, endPos, err;
        int currentLine = 0;
   char line[STRING_SIZE];
        int nchars;
        int nlines = 0;

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

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

        for(i = startPos; i < endPos; i++)
        {
                nchars = strlen( lines[i] );
                line_avg[i] = find_avg(lines[i], nchars);
        }

        pthread_exit(NULL);
}

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

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

```
}

main() {

        int i, rc;
        NUM_THREADS = atoi(getenv("SLURM_NTASKS"));
        pthread_t threads[NUM_THREADS];
        pthread_attr_t attr;
        void *status;
        struct timeval t1, t2, t3;
   double timeElapsedTotal;
   processMem_t memory;

        pthread_mutex_init(&mutexsum, NULL);

        readFile();

        /* Timing analysis begins */
   gettimeofday(&t1, NULL);
        printf("DEBUG: starting time on %s\n", getenv("HOSTNAME"));

        /* Initialize and set thread detached attribute */
        pthread_attr_init(&attr);
        pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

        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);
             }
        }

        /* Free attribute and wait for the other threads */
        pthread_attr_destroy(&attr);
        for(i=0; i<NUM_THREADS; i++) {
            rc = pthread_join(threads[i], &status);
            if (rc) {
                    printf("ERROR; return code from pthread_join() is %d\n", rc);
```

```
                exit(-1);
            }
        }

        gettimeofday(&t2, NULL);

        print_results(line_avg);

        gettimeofday(&t3, NULL);


   //total program time
   timeElapsedTotal = (t3.tv_sec - t1.tv_sec) * 1000.0; //Converted to milliseconds
   timeElapsedTotal += (t3.tv_usec - t1.tv_usec) / 1000.0;

        pthread_mutex_destroy(&mutexsum);

        printf("Tasks: %s\n Total Elapsed Time: %fms\n", getenv("SLURM_NTASKS"),
timeElapsedTotal);
        printf("DATA, %s,%f\n", getenv("SLURM_NTASKS"), timeElapsedTotal);
   GetProcessMemory(&memory);

        printf("size = %d, Node: %s, vMem %u KB, pMem %u KB\n", NUM_THREADS,
getenv("HOSTNAME"), memory.virtualMem, memory.physicalMem);
        printf("Main: program completed. Exiting.\n");

        pthread_exit(NULL);
}
```

pThreads Shell Scripts:

https://github.com/cody598/Project4/blob/master/3way-pthread/TestingOutput/output-1mil/pThreads_sbatch.sh

```
#!/bin/bash -l
#SBATCH --time=0:10:00
#SBATCH --mem=100G
#SBATCH --constraint=elves

module load OpenMPI
module load foss/2020a --quiet
```

```
echo pthread

time /homes/cody598/cis520/Project4/3way-pthread/pThreads-1mil
grep DATA *.out > 1-milTimes.csv
```

https://github.com/cody598/Project4/blob/master/3way-pthread/TestingOutput/output-1mil/mass_sbatch.sh

```
#!/bin/bash
#SBATCH --job-name=pThreads
#SBATCH -o 3way-pThreads-MASSBATCH-STATS.out
for i in 1 2 4 8 16
do
        echo "Tasks: $i"
        sbatch --constraint=elves --ntasks-per-node=$i --nodes=1 --job-name=pThreads -o
$i-core-1mil.out pThreads_sbatch.sh
done
```

pThreads Output First 100 Lines:

```
pThread
DEBUG: starting time on elf79
myID = 0 startPos = 0 endPos = 10000
0: 94.1
1: 91.9
2: 92.3
3: 93.6
4: 92.2
5: 91.1
6: 92.2
7: 93.8
8: 90.6
9: 92.8
10: 90.5
11: 89.0
12: 94.1
13: 89.1
14: 90.1
15: 94.4
```

16: 93.2
17: 90.9
18: 92.4
19: 93.5
20: 93.6
21: 90.1
22: 90.3
23: 92.0
24: 92.5
25: 94.1
26: 94.9
27: 89.6
28: 87.8
29: 92.2
30: 92.4
31: 90.5
32: 93.6
33: 93.1
34: 82.6
35: 90.4
36: 89.6
37: 94.2
38: 95.0
39: 94.5
40: 84.8
41: 92.5
42: 91.8
43: 92.4
44: 90.1
45: 91.6
46: 90.3
47: 84.3
48: 95.8
49: 92.4
50: 94.3
51: 91.8
52: 91.3
53: 86.9
54: 90.9

55: 93.3
56: 94.3
57: 89.5
58: 90.5
59: 91.8
60: 94.1
61: 81.9
62: 92.6
63: 94.4
64: 92.9
65: 93.5
66: 93.4
67: 93.2
68: 86.6
69: 85.2
70: 91.9
71: 89.0
72: 93.0
73: 87.8
74: 86.3
75: 93.9
76: 91.9
77: 92.3
78: 88.2
79: 91.8
80: 80.5
81: 90.7
82: 99.6
83: 92.4
84: 92.8
85: 81.3
86: 89.0
87: 84.1
88: 95.5
89: 93.7
90: 93.1
91: 94.6
92: 92.3
93: 91.7

94: 80.5
95: 92.2
96: 88.0
97: 90.3
98: 89.4
99: 89.7
100: 89.4