David Apple, Luke Hetrick, Matthew Sinclair
5/1/2022

Operating Systems Project 4 Report

Introduction:

The objective of this project was to create multithreaded programs to find the minimum character on each line of a large wikipedia dump text document. To accomplish this we used OpenMP, pthreads, and MPI in their respective ways discussed below.

Implementation:

OpenMP:

To implement the threading using OpenMP, my process was similar to that demonstrated in class in pt1_openmp_*.c. I ran into issues having each thread read a line from a file, do work on it, and return, so I now read the entire file into a global array. That global array is passed to the findMinChars() function that then gives each thread a position in the array to search for. We begin comparisons on each character in the line, and after it finds the value, it places it in an array, and then moves on. Each thread's array index (or line number from the file) is given its own starting value, and increases to the next line it needs to read (lineNum += numThreads). Thus, the work to read and determine the minimum character has been parallelized.

PThread:

To implement the threading using the pthread library I took a minchar function and then created a thread function. This was where each thread would open the file, get the specific line number it was looking for and then calculate the min character of that line placing the result into a large shared array of all the lines. This split up all the work to each thread as to run the file retrieval and minimum character calculation in parallel. For loops were used to move down the file in each thread as line lengths could differ within the file. Arguments were added to the main function to allow a way to run the file using different thread and maximum line amounts. This also allowed for a clean script to execute all the necessary parts to do performance evaluation.

MPI:

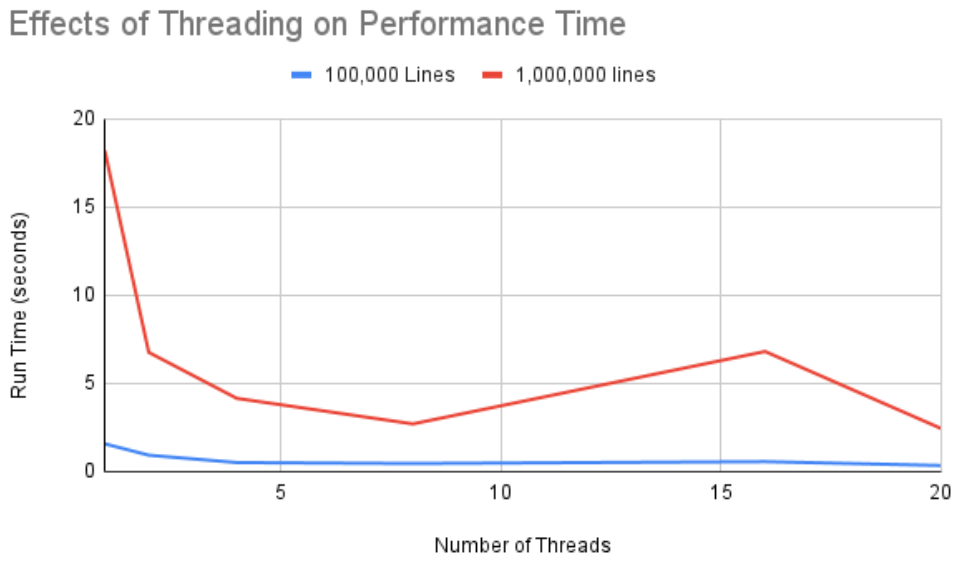
To implement threading by using MPI, I decided to chunk the amount of processing based on how many threads were passed to the command line. The master thread would keep any leftover lines in case the chunking was not divisible and send messages to the other threads. These messages would be the offset that each thread was to start at and the section of the global

array that contained the results. The master then would do its task by calling a function that would take in the thread's offset, ID, and chunk size and loop through the file until it got to its specified chunk and then do its calculations. The other threads then pass back the results for the master to display using the global results array.

Evaluation:

OpenMP:

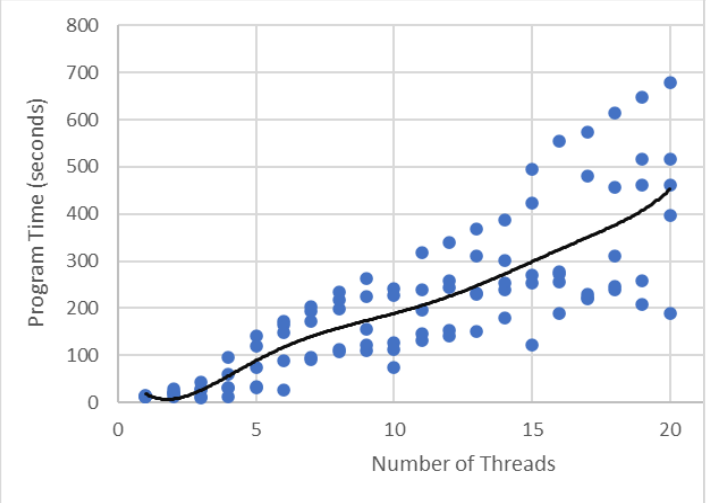
The implementation of OpenMP can be evaluated from the following plots. To construct these plots, I took an average of 3 data points on several thread number intervals (1, 2, 4, 8, 16, and 20 threads) and line numbers 100,000 to 1,000,000. It is interesting to note that the time values increase with the number of threads, and I believe this to be due to the overhead of initializing and setting up the threads themselves. It may also work better if the threading is implemented to read the file too, but since the threading only searches for the smallest character in a line, we lose out on efficiency.

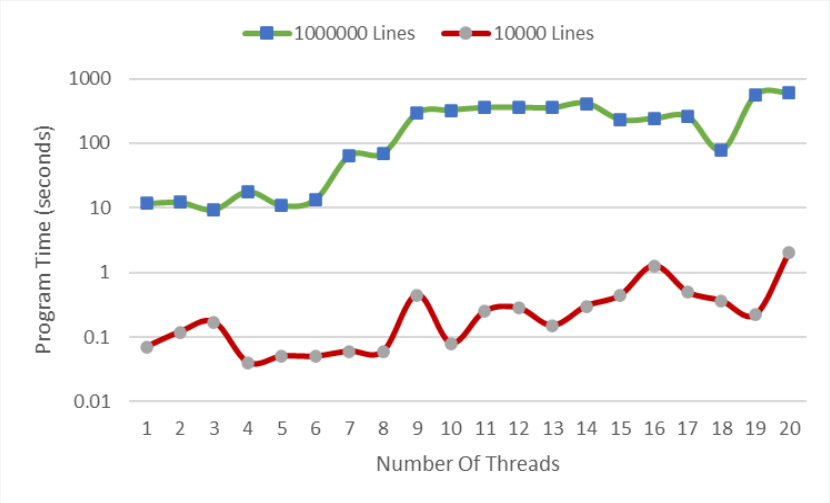


As we can see, the 1 million lines sees a significant drop in time as we increase threads, but we do not see that same behavior on the 100,000 lines attempt. I believe this to be because of the overhead required, and that there is not much room for optimization on a "small" amount of lines.

PThread:

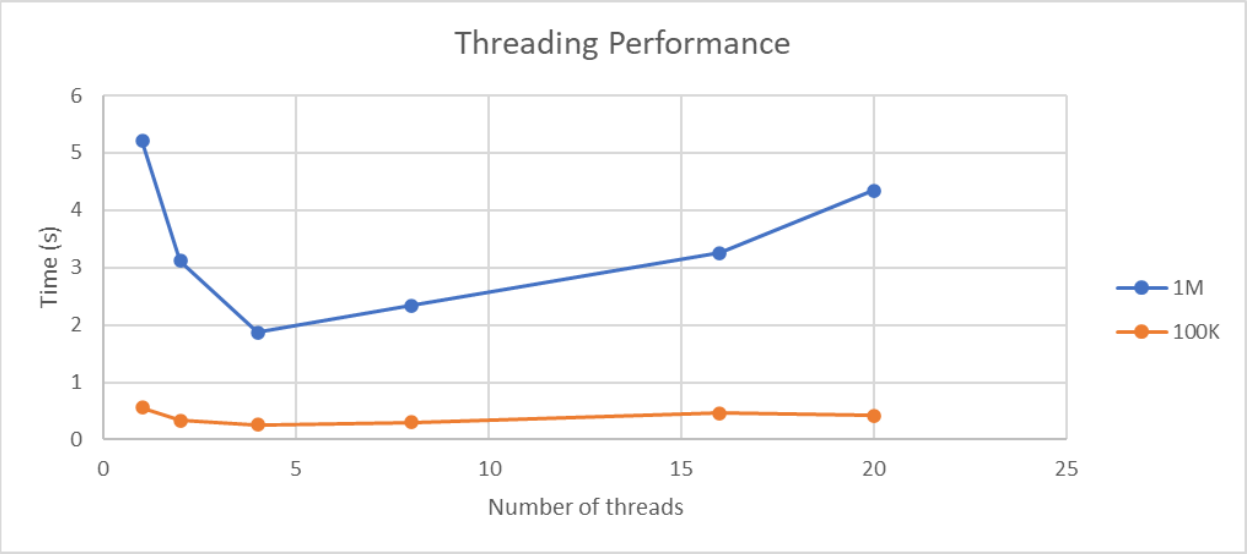
The overall evaluation of the program can be seen in the following two plots which show the time it took to run the program vs the number of threads/cores that were requested to be used when running with the whole file which is on the left. Then there is a plot on the right that shows one run each using the full file or the first 10k lines for reading in the file. These two graphs show a sort of increasing trend as the thread count is increased for the program. I believe this to be due to the overhead for threading hurting the performance of the program execution. The data was gathered using the clock function and finding the difference between the start and after the program ran divided by the clock cycles per second constant. This was then printed out in a JSON style format to allow me to use a python script to parse through the data to create a csv file which was used to create the graphs below using excel.





MPI:

Much like for open MP, to evaluate the performance of the threading, I used the data points of the powers of 2 up until 16 and added 20 with line numbers of 100,000 and 1,000,000. And as with the other threading methods, the run time gets substantially better when going from 1 thread to 2 and 4 but then as the thread count increases, the overhead required to create that many threads and have them communicate results in a loss of performance indicating that the optimal performance for this program would require between 2 and 8 threads with 4 seeming like a good choice.



Conclusion:

We were successfully able to create multiple threads to parse through and find the minimum character of each line inside a text file dump of wikipedia entries. Each threading API we used has its own strengths and weaknesses, and we are all now more experienced and have better hands-on exposure to parallel programming.

Appendix A: Output From First 100 Lines of Program

1: 33
2: 39
3: 39
4: 39
5: 34
6: 39
7: 34
8: 33
9: 33
10: 39
11: 33
12: 38
13: 39
14: 38
15: 39
16: 39
17: 39
18: 39
19: 39
20: 39
21: 34
22: 39
23: 34
24: 34
25: 38
26: 34
27: 35
28: 34
29: 33
30: 39
31: 39
32: 34
33: 39
34: 39
35: 33
36: 39
37: 33
38: 39
39: 39

40: 39
41: 33
42: 39
43: 39
44: 39
45: 33
46: 38
47: 38
48: 34
49: 33
50: 39
51: 33
52: 33
53: 34
54: 38
55: 33
56: 34
57: 37
58: 33
59: 34
60: 35
61: 34
62: 33
63: 34
64: 39
65: 39
66: 39
67: 38
68: 33
69: 39
70: 33
71: 34
72: 33
73: 34
74: 33
75: 36
76: 38
77: 38
78: 38
79: 34

80: 34
81: 39
82: 38
83: 39
84: 33
85: 34
86: 38
87: 34
88: 34
89: 38
90: 34
91: 39
92: 39
93: 39
94: 34
95: 33
96: 34
97: 39
98: 39
99: 33
100: 33

Appendix B: Code

OpenMP:

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <omp.h>

#define MAX_LINE_LENGTH 2000
#define MAX_LINES 1000000

char gFileContents[MAX_LINES][MAX_LINE_LENGTH];

static void findMinChars(char* minCharAtLine, int numThreads, int maxLines)
{
```

```
	int lineNum = 0;
	char minChar;

	#pragma omp parallel private(lineNum, minChar) num_threads(numThreads)
	{
	lineNum = omp_get_thread_num();

	while(lineNum < maxLines)
	{
	/* set to max value each new line */
	minChar = 127;
	for(int i = 0; i < MAX_LINE_LENGTH; i++)
	{
		/* only look in valid range */
		if ((gFileContents[lineNum][i] > 32) && (gFileContents[lineNum][i] < 127)) //
Everything Only between Space and Delete
		{
		if (gFileContents[lineNum][i] < minChar)
		{
		minChar = gFileContents[lineNum][i];
		}
		}
	}

	/* update so other threads can go ahead */
	lineNum += numThreads;

	/* wait for each thread to write to file */
	#pragma omp critical
	{
		minCharAtLine[lineNum - numThreads] = minChar;
	}
	}
	}
}


int main(int argc, char** argv)
{
	/* adjust numThreads as needed. Originally, this was supposed to be
```

```
        determined via command line args, but this is easier for consistency */
        int numThreads = 20, maxLines = MAX_LINES;


        char* minCharAtLine = malloc(sizeof(char) * maxLines);
        char* lineRead = malloc(sizeof(char) * MAX_LINE_LENGTH);

        int numRead;

        /* need this for getline */
        size_t maxLineLength = MAX_LINE_LENGTH;

        FILE* fp = fopen("/homes/dan/625/wiki_dump.txt", "r");
        if(!fp)
        return -1;

        for(int i = 0; i < maxLines; i++)
        {
        numRead = getline(&lineRead, &maxLineLength, fp);
        if(numRead == -1)
        break;
        /* i know this is gross but its all I could do */
        strncpy(gFileContents[i], lineRead, MAX_LINE_LENGTH);
        }


        /* do the threading and calculation stuff */
        findMinChars(minCharAtLine, numThreads, maxLines);

        for(int i = 0; i < maxLines; i++)
        {
        /* if you want to see output, uncomment this out */
        //printf("Line %d: %c\n", i, minCharAtLine[i]);
        }
        return 0;
}
```

Pthread:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <ctype.h>
#include <pthread.h>
int findMinChars(char *line, int len);
void *thread_func(void *id);
int *Num_Threads;
int *Max_Lines;
int *min_char;
/* comparison function to use to find the lowest character in each row */
int findMinChars(char *line, int len)
{
   char compChar = '~';
   int i;
   for (i = 0; i < len; i++)
   {
      if ((line[i] > 32) && (line[i] < 127)) // Everything Only between Space and Delete
      {
         if (line[i] < compChar)
         {
            compChar = line[i];
            if (compChar == 33)
            {
               break;
            }
         }
      }
   }
   return (int)compChar;
}
void *thread_func(void *id)
```

```
{
   int line_num = 0;
   int c;
   int t_id = *(int *)id;
   int i, j;
   int ret;
   FILE *fp;
   fp = fopen("/homes/dan/625/wiki_dump.txt", "r");
   if (!fp)
      exit(-1);
   char *line = malloc(sizeof(char) * 3000);
   if (line == NULL)
   {
      printf("allocation Failure\n");
      return (void *)-1;
   }
   size_t line_len = sizeof(line);
   // Get to the starting line for the thread
   for (i = 0; i <= t_id; i++)
   {
      line_num++;
      ret = getline(&line, &line_len, fp);
      if (ret == -1)
         break;
   }
   // run function to get first minchar
   c = findMinChars(line, ret);
   min_char[(line_num - 1)] = c;
   // run until we are out of lines
   for (i = 0; i < *Max_Lines; i++)
   {
      for (j = 0; j < *Num_Threads; j++)
      {
         line_num++;
```

```
            ret = getline(&line, &line_len, fp);
            if (ret == -1 || line_num > *Max_Lines)
                break; // this will only get us out of the inner loop
        }
        if (ret == -1 || line_num > *Max_Lines)
            break;
        c = findMinChars(line, ret);
        min_char[(line_num - 1)] = c;
    }
    free(line);
    free(id);
    fclose(fp);
    pthread_exit(0);
}
int main(int argc, char **argv)
{
    if (argc < 3)
    {
        printf("%s ThreadCount MaxLineNumber\n", argv[0]);
        return EXIT_FAILURE;
    }
    Num_Threads = malloc(sizeof(int));
    if (Num_Threads == NULL)
    {
        printf("allocation Failure\n");
        return -1;
    }
    *Num_Threads = atoi(argv[1]);
    Max_Lines = malloc(sizeof(int));
    if (Max_Lines == NULL)
    {
        printf("allocation Failure\n");
        return -1;
    }
```

```
*Max_Lines = atoi(argv[2]);
min_char = malloc(sizeof(int) * (*Max_Lines));
if (min_char == NULL)
{
    printf("allocation Failure\n");
    return -1;
}
pthread_t *threads = malloc(sizeof(pthread_t) * (*Num_Threads));
if (threads == NULL)
{
    printf("allocation Failure\n");
    return -1;
}
clock_t begin = clock();
int i;
int ret;
void *status;
for (i = 0; i < *Num_Threads; i++)
{
    int *arg = malloc(sizeof(*arg));
    if (arg == NULL)
    {
        printf("allocation Failure\n");
        return -1;
    }
    *arg = i;
    ret = pthread_create(&threads[i], NULL, thread_func, (void *)arg);
    if (ret != 0)
    {
        printf("pthread create error:\n");
        return -1;
    }
}
for (i = 0; i < *Num_Threads; i++)
```

```
    {
        ret = pthread_join(threads[i], &status);
        if (ret != 0)
        {
            printf("pthread join error: %d\n", ret);
            return -1;
        }
    }
    for (i = 0; i < *Max_Lines; i++)
    {
        int val = min_char[i];
        printf("%d: %d\n", i + 1, val);
    }
    clock_t end = clock();
    double run_time = (double)(end - begin);
    printf("\n\n\n{ \"NumThreads\":%d, \"NumLines\":%d, \"RunTime\":%lf }\n",
*Num_Threads, *Max_Lines, run_time / CLOCKS_PER_SEC);
    free(min_char);
    free(Max_Lines);
    free(Num_Threads);
    free(threads);
    return 0;
}
```

MPI:

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

#define MAX_LINES 1000000
int NUM_THREADS;
size_t lineLen = 2500;

char minChars[MAX_LINES];

/* comparison function to use to find the lowest character in each row */
char findMinChars(char* line, int len)
```

```
{
    char compChar = '~';
    int i;
    for (i = 0; i < len; i++)
    {
        if ((line[i] > 32) && (line[i] < 127)) // Everything Only between
Space and Delete
        {
            if (line[i] < compChar)
            {
                compChar = line[i];
                if (compChar == 33)
                {
                    break;
                }
            }
        }
    }
    return compChar;
}

// a wrapper function that handles the calculations for the threads
int handleMinChar(int offset, int chunk, int myID)
{

    /* don't allow more than 2500 chars in one string */
    char* line = (char*)malloc(sizeof(char) *lineLen);
    if (line == NULL)
        return 0;

    int length;
    FILE* fp;
    fp = fopen("/homes/dan/625/wiki_dump.txt", "r");
    if(!fp)
        return 0;

    // move through the file to get to the starting postition
    for(int i = 0; i < offset; i++)
    {
        // stop if it accidently hits the end of the file
        if(getline(&line, &lineLen, fp) == -1)
            break;
    }

    // do the calculation for the lines that are this thread's responsibility
    for(int i = offset; i < offset + chunk; i++)
    {
        length = getline(&line, &lineLen, fp);
        if(length == -1)
```

```
            break;
        minChars[i] = findMinChars(line, length);
    }

    free(line);
    fclose(fp);
    return 1;
}


// general structure of this code was inspired by:
https://hpc-tutorials.llnl.gov/mpi/examples/mpi_array.c
int main(int argc, char* argv[])
{
    clock_t begin = clock();
    int rc, rank, numtasks;
    int MASTER = 0;   // the MASTER rank
    int chunksize, offset, leftover, tag1, tag2, destThread, srcThread;
    MPI_Status status;

    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);

    //initialize variables now that MPI is initialzed
    NUM_THREADS = numtasks;

    chunksize = MAX_LINES / NUM_THREADS;
    leftover = MAX_LINES % NUM_THREADS;
    tag1 = 2;
    tag2 = 1;


    /************************** MASTER thread tasks **********************/
    if(rank == MASTER)
    {
        //MASTER thread handles its own chunk and the leftovers
        offset = chunksize + leftover;
        for (destThread = 1; destThread < NUM_THREADS; destThread++)
        {
            //send the part of the array and the offset values to the other
threads
            MPI_Send(&offset, 1, MPI_INT, destThread, tag1, MPI_COMM_WORLD);
```

```
            MPI_Send(&minChars[offset], chunksize, MPI_BYTE, destThread, tag2,
MPI_COMM_WORLD);
            offset = offset + chunksize;
        }

        //master does it's part of the work
        offset = 0;
        if(!handleMinChar(offset, chunksize + leftover, rank))
        {
            printf("rank %d failed\n", rank);
        }

        //wait to recieve results from the other threads
        for(int i = 1; i < NUM_THREADS; i++)
        {
            srcThread = i;
            MPI_Recv(&offset, 1, MPI_INT, srcThread, tag1, MPI_COMM_WORLD,
&status);
            MPI_Recv(&minChars[offset], chunksize, MPI_BYTE, srcThread, tag2,
MPI_COMM_WORLD, &status);
        }

        //print results at the end
        for(int i = 0; i < MAX_LINES; i ++)
        {

            // printf("Line %d: min char: %c \tcharNum: %d\n", i, minChars[i],
minChars[i]); debugging print
            printf("%d: %d\n", i+1, minChars[i]);
        }

        clock_t end = clock();
        double run_time = (double)(end - begin);
        run_time++;
        printf("Num threads: %d, NumLines: %d, time: %lf\n", NUM_THREADS,
MAX_LINES, run_time / CLOCKS_PER_SEC);

    }

    /*************************** Non-MASTER thread tasks ********************/
    if(rank > MASTER)
    {
        //recieve this thread's portion of the work
        srcThread = MASTER;
        MPI_Recv(&offset, 1, MPI_INT, srcThread, tag1, MPI_COMM_WORLD,
&status);
        MPI_Recv(&minChars[offset], chunksize, MPI_BYTE, srcThread, tag2,
MPI_COMM_WORLD, &status);
```

```
        //do the work
        if(!handleMinChar(offset, chunksize + leftover, rank))
        {
            printf("rank %d failed\n", rank);
        }

        //send the results back to MASTER
        destThread = MASTER;
        MPI_Send(&offset, 1, MPI_INT, destThread, tag1, MPI_COMM_WORLD);
        MPI_Send(&minChars[offset], chunksize, MPI_BYTE, destThread, tag2,
MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;

}
```

Mass_run.sh

```
#!/bin/bash -l
for i in {1..20}
do
   sbatch --ntasks-per-node=$i sbatch_script.sh $i
Done
```

Sbatch_script.sh

```
#!/bin/bash -l
## Taken From A Sample slurm script created by Kyle Hutson
## Specify the amount of RAM needed _per_core_. Default is 1G
#SBATCH --mem-per-cpu=2G
## Specify the maximum runtime. Default is 1 hour (1:00:00)
#SBATCH --time=0:10:00   # Use the form DD-HH:MM:SS
#SBATCH --partition=killable.q   # Job may run as killable on owned nodes
#SBATCH --constraint=warlocks
#SBATCH --nodes=1
## Name my job, to make it easier to find in the queue
```

```
#SBATCH --job-name=pthread
## Send email when a job begins, fails, or ends
#SBATCH --mail-type=ALL   # same as =BEGIN,FAIL,END
## Email address to send the email to based on the above line.
#SBATCH --mail-user=email@ksu.edu

time ./pthread $1 1000000
```