PARALLELIZATION USING THREE DIFFERENT METHODS

Kansas State University: Department of Computer Science

Bristol Miller, Shreya Kumar, Corey Vessar

bristolnmiller@ksu.edu, shreyak@ksu.edu, coreyvessar@ksu.edu
Manhattan, KS 66503

Table of Contents

Abstract

Implementing programs without parallelization can make even simple solutions run quite slowly. While reading a larger file and searching it line by line for information, it is necessary to parallelize your program in order to have the most efficient solution. Through our trial of the three different methods of parallelization: OpenMP, MPI, and PThreads. We have found that the outputs for Pthreads and OpenMP are quite similar and parallelized code outperforms single-threaded programs by an average of 2.39 hours for processing one million lines.

Introduction

A program that we have written reads in a 1.7 GB file, called wiki_dump.txt, and compares the first two lines, followed by the second and third lines, then the third and fourth lines, and so on, searching for the longest common substring between each pairing and outputting those substrings to a file. The major problem with this program is that it is inefficient.

Initially, we had not implemented any parallelization in order to not complicate the ode. Avoiding parallelization from the start led to our program being very slow. From start to end, accomplishing the tasks of reading in one million lines of the file and searching for the longest common substrings took 3.8 hours. When faced with a heavy task such as reading in a large file and searching through each line of that large file, this slow speed can be detrimental if the information is needed quickly. We have solved this problem by parallelizing the program in three different ways, in order to figure out which method provides the quickest way to determine the longest common substrings. These three methods are OpenMP, MPI, and PThreads.

By creating solutions using the OpenMP, MPI, and PThreads methods all separately, we were able to discern which method is the most efficient for this application. Each of the three methods presented a considerable decrease in runtime, and thus a dramatic increase in efficiency. Also, having the comparisons between the three different methods allowed us to make an educated and unbiased decision as to which solution is the best for this specific application. After experimentation with all three methods, we came to the conclusion that the PThreads method is the best for our program because it presents the quickest run time for larger problem sets.

Related Work

One could approach this problem in a variety of ways. One such example would be to use a single-threaded implementation. This method would decrease the complexity of such a solution, but the solution would be much slower than our implementations. Usually when you convert a program from single-threaded to multi-threaded, you get a dramatic speedup without too much effort. There is, of course, a balance to strike between the two, though. For some applications, the speed is less important, so it is considered a waste of time to turn the program into a multi-threaded one. On the other hand, there are plenty of scenarios where time is of the essence, and a faster program is valued higher than the effort required to make it so.

Another way to vary the approach of this problem would be to copy the file locally and use that instead of reading it in to memory from a separate location. Approaching with this manner isn't very spatially effective, but it may produce a speedup as far as timing is concerned. On this note, it also might be more effective to parallelize reading the file in. In our implementation, we simply parallelized the search, rather than our entire program. If we were to read multiple lines into memory at a time, we're certain that this would be quicker, although it would be more difficult to code.

There are multiple different methods one can use to parallelize their solution. The three that we decided to test were OpenMP, PThreads, and MPI, however, there are more available to programmers. Some of those other methods include OpenACC, OpenCL, and NVIDIA CUDA. For our purposes, we aren't considering the other methods, although they have their own advantages and drawbacks. CUDA tends to have very high performance, but often requires frequent updates and is expensive. OpenCL may not have as high of performance, but it isn't very expensive and doesn't require frequent updates.

Implementation

MPI is a method that implements message passing, in fact, its name is an acronym for Message Passing Interface. In message passing, there are two main functions used: send and receive. Send allows threads communicate with each other. This can be either synchronous or asynchronous. If its synchronous, then it waits for the receive end to send an acknowledgement. On the other hand, if its asynchronous, the message passed from one thread to another can get pushed into a queue, and the thread that sent it can continue about its business. When a thread reaches the receive message, it usually waits, unless there is a message in the queue. When a message is received, we send an acknowledgement.

OpenMP is a different method in which part of your program is single-threaded, while other parts are multi-threaded. The M and P in OpenMP are short for Multi-Processing. This method overall has three different components: a runtime library, environment variables that define the runtime parallel parameters, and directives for the programs. OpenMP is known for its ease of use and portability, while allowing users to standardize and “lean-out” their code.

PThreads is short for POSIX Threads. It is a method of parallel programming in which the threads are incredibly light-weight and efficient with data exchange. Within this method, there are four main types of functions included in its library. These types are mutexes, condition variables, synchronization between threads using locks and barriers on reads and writes, and thread management techniques including creating and joining threads.

Evaluation

Our methods for testing involved submitting a variety of jobs to Beocat, Kansas State University's supercomputer. We chose to run the program on the three different parallelization tools outlined above, along with our single-threaded implementation as a control. We used with a combination of 1, 2, 4, 8, 16, and 32 threads, multiple cores and a job size varying from 1000 lines to 1 million lines for comparison. After collecting each of these combinations through ten trials, we threw out the first result and averaged the remaining values to evaluate running time. Through these evaluation steps we were able to find that for smaller problem sets OpenMP performed best, while for larger problem sets, PThreads performed best. While we were not about to implement MPI on larger data sets we believe it would perform slower than Pthreads and OpenMP since it requires more memory, being a memory distributed program.

In Figures 1 and 2, we can visually see that as the problem size increases, the time elapsed increases as well. Furthermore, we can see that as we add more threads to do the work, the amount of time it takes to accomplish the tasks decreases. These results can be expected, as computers have been designed to multitask in order to finish jobs in the most efficient manner possible. This can be verified by viewing the numbers listed in their respective tables: Table 1 and Table 2.

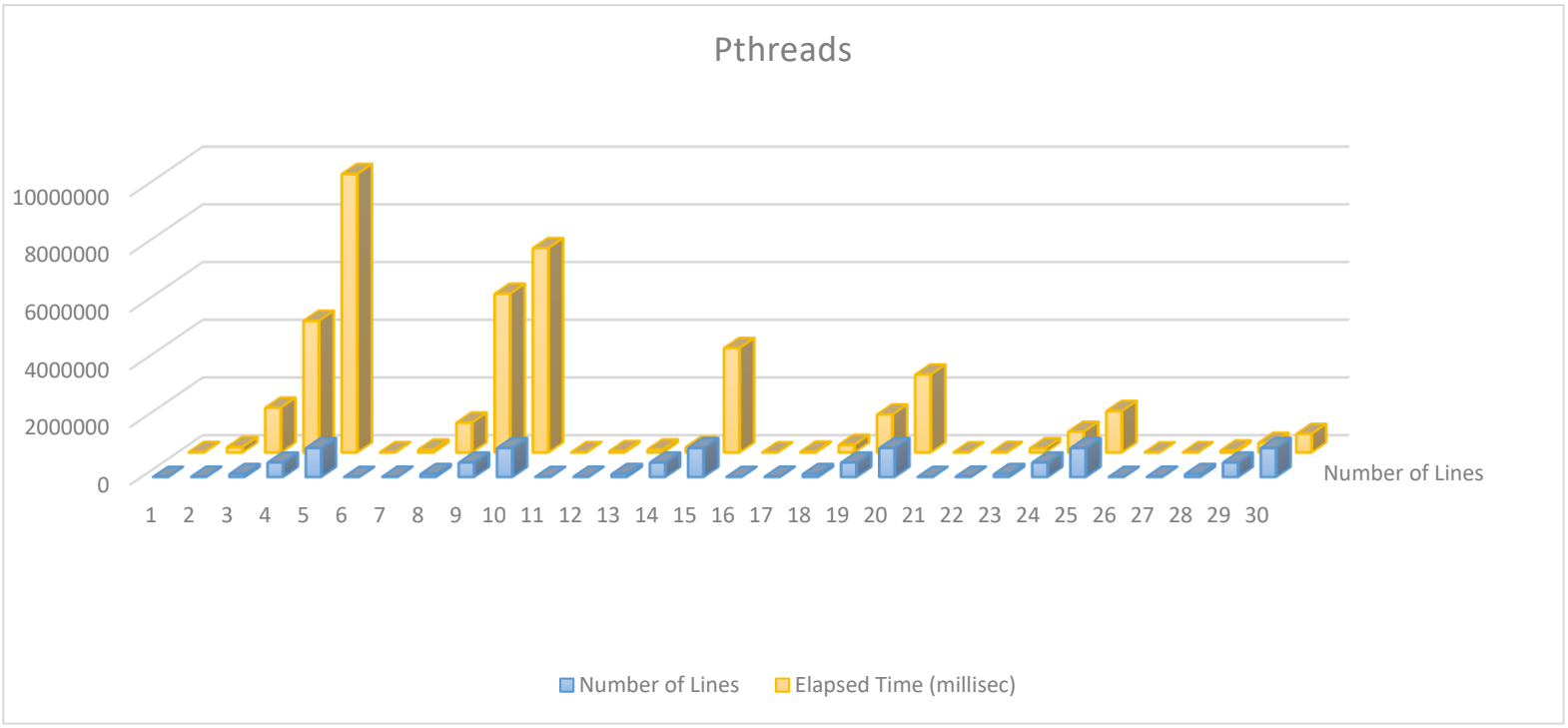

Figure 1: PThreads' averaged results when run on 1000, 10000, 100000, 500000, and 1 million lines and 1, 2, 4, 8, 16, and 32 cores.

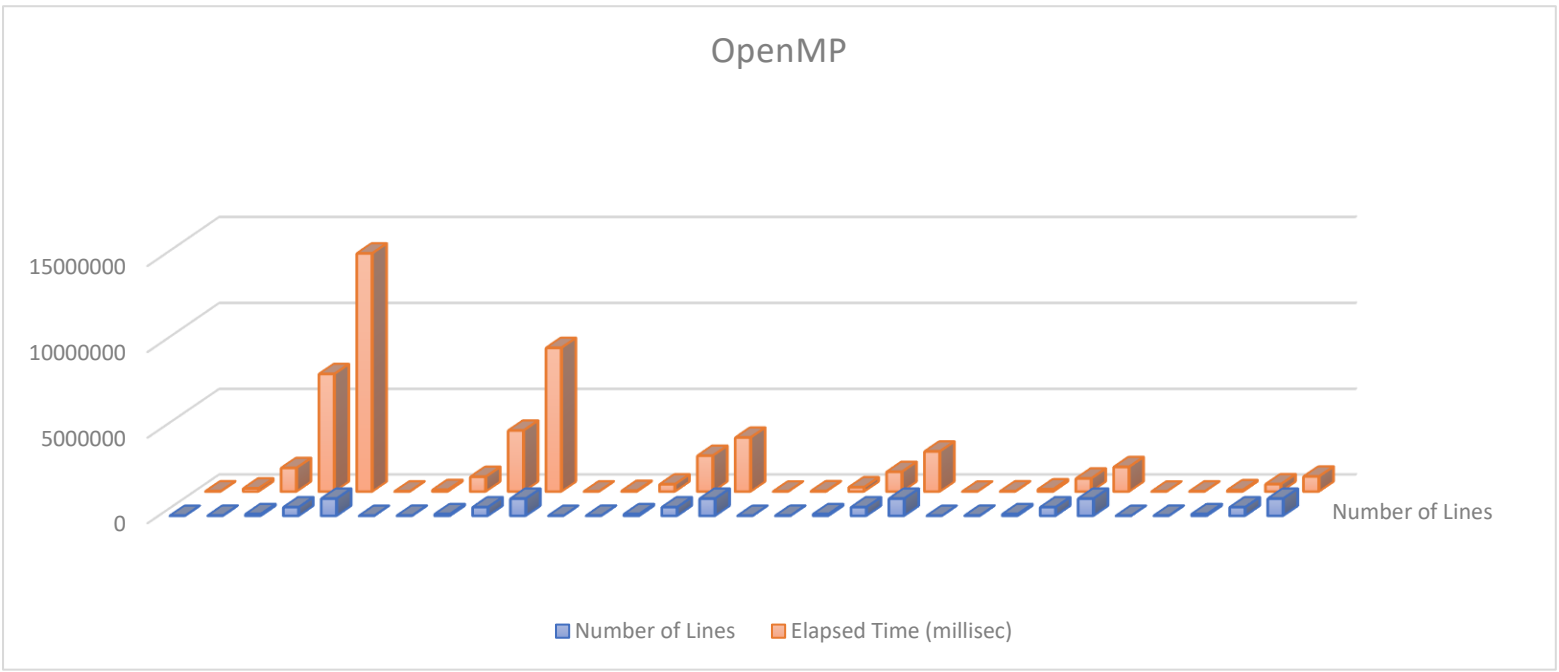

Figure 2 OpenMP's averaged results when run on 1000, 10000, 100000, 500000, and 1 million lines and 1, 2, 4, 8, 16, and 32 cores.

Threads	Number of Lines	Elapsed Time (millisec)
1	1000	15375.153
1	10000	188362.171
1	100000	1551852.414
1	500000	4563331.758
1	1000000	9651787.175
2	1000	8493.575
2	10000	97362.171
2	100000	1034568.271
2	500000	5502505.555
2	1000000	7081787.175
4	1000	6092.226
4	10000	47171.256
4	100000	104540.148
4	500000	189983.335
4	1000000	3619966.03
8	1000	3123.155
8	10000	28113.218
8	100000	262954.525
8	500000	1314773.748
8	1000000	2698547.329
16	1000	1660.103
16	10000	14730.765
16	100000	143998.781
16	500000	717360.846
16	1000000	1434721.693
32	1000	1651.429
32	10000	7842.544
32	100000	67143.883
32	500000	335719.383
32	1000000	641439.766

Table 1: PThreads' averaged results

Threads	Number of Lines	Elapsed Time (millisec)
1	1000	15371.298
1	10000	179652.522
1	100000	1373300.683
1	500000	6866503.698
1	1000000	13873356.65
2	1000	9439.489
2	10000	94613.462
2	100000	857779.949
2	500000	3563331.758
2	1000000	8386087.198
4	1000	5472.77
4	10000	44907.409
4	100000	418127.889
4	500000	2090639.856
4	1000000	3151208.136
8	1000	3861.276
8	10000	24083.699
8	100000	249077.285
8	500000	1148957.054
8	1000000	2340611.178
16	1000	1554.738
16	10000	12906.174
16	100000	130598.257
16	500000	758941.357
16	1000000	1432768.258
32	1000	1383.535
32	10000	8367.317
32	100000	81924.023
32	500000	426822.374
32	1000000	875214.207

Table 2 OpenMP's averaged results

Conclusions

We have found that out of the following three methods of parallelization: OpenMP, MPI, and PThreads, Pthreads is the fastest for parallelization of our application because it performed best with larger data sets. OpenMP had similar performance though it was moderately slower on files larger than five hundred thousand lines. In our experience, Pthreads was also the easiest to implement and we were able to use a lot of the algorithm and code from the Pthreads implementation for OpenMP as well.

Furthermore, we believe MPI runs comparatively slower than OpenMP and noticeably slower than Pthreads based on our initial trial run of MPI before implementing the algorithm. While we were unable to run MPI on larger files, the believe that the distributed memory property of MPI, where every parallel process is working in its own memory space in isolation from the others, would cause it to run at a rate slower than the other two parallelization methods. Thus, when it is necessary to read in a large file and search that file for the longest common substrings, we have found Pthreads to be one of the more efficient ways to solve the problem.

Additionally, in our results we were able to observe that addition of cores impacted the running time of OpenMP on smaller data sets to a greater degree as shown in Table 2 while it impacted the performance of Pthreads to a greater degree on larger data sets as shown in Table 1.

In the future, one could compare other methods, like OpenACC and OpenCL to our findings in this paper. Another way to improve upon our work would be to parallelize the reads, although this would provide only slightly better timing results. From here, these findings could be used for a plethora of different things, including to check for copywrite infringement in students' papers by comparing the lines in a student's paper to those in scholarly articles already written, and without taking too long. Another use of this could be to modify the algorithm to search for keywords within two texts in order to compare the words following them. This could be useful for scanning through two articles that highlight opposing sides of similar issues, like when comparing political parties' ideas.

Appendix

In this section, we have included copies of our code, along with links to the files on Github.

OpenMP

https://github.com/shreyakumar1010/cis520-Project4/blob/master/OpenMP/OpenMP.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#include <stdbool.h>
#include <omp.h>
#define WIKI_ARRAY_SIZE 50000
#define WIKI_LINE_SIZE 2001
#define num_threads 32
int LCS (char * s1, char * s2, char ** longest_common_substring);
//load the lines into an array
char  **wiki_array;
char **longestSub;
omp_lock_t my_lock;
void readToMemory();
void printResults();
void printToFile();
int main()
{
	omp_init_lock(&my_lock);
	struct timeval time1;
	struct timeval time2;
	struct timeval time3;
	struct timeval time4;
	double e1, e2, e3;
	int numSlots, Version = 3; //base = 1, pthread = 2, openmp = 3, mpi = 4
	gettimeofday(&time1, NULL);
	readToMemory();
	gettimeofday(&time2, NULL);
	//time to read to memory
	e1 = (time2.tv_sec - time1.tv_sec) * 1000.0; //sec to ms
	e1 += (time2.tv_usec - time1.tv_usec) / 1000.0; // us to ms
	printf("Time to read full file to Memory: %f\n", e1);
	gettimeofday(&time3, NULL);
	int i,j, startPos, endPos, myID;
	char** longestSubPointer;
	omp_set_num_threads(num_threads);

	#pragma omp parallel private(myID, startPos, endPos, j, longestSubPointer)
	{
		myID = omp_get_thread_num();
		startPos = (myID) * (WIKI_ARRAY_SIZE / num_threads);
		endPos = startPos + (WIKI_ARRAY_SIZE / num_threads);
		longestSubPointer = longestSub + startPos;
	if(myID == num_threads-1)
	{
	  endPos = WIKI_ARRAY_SIZE - 1 ;
	}
			for (j = startPos; j< endPos; j++)
			{
				LCS((void*)wiki_array[j], (void*)wiki_array[j+1], longestSubPointer);
				longestSubPointer++;
			}
	}
```

```
	printToFile();
	gettimeofday(&time4, NULL);
	//time to find all longest substrings
	e2 = (time4.tv_sec - time3.tv_sec) * 1000.0; //sec to ms
	e2 += (time4.tv_usec - time3.tv_usec) / 1000.0; // us to ms
	printf("Time find all Substrings: %f\n", e2);
	//total elapsed time between reading and finding all longest substrings
	e3 = (time4.tv_sec - time1.tv_sec) * 1000.0; //sec to ms
	e3 += (time4.tv_usec - time1.tv_usec) / 1000.0; // us to ms
	printf("DATA, %d, %s, %f, %d\n", myVersion, getenv("NSLOTS"), e3, num_threads);
}
void readToMemory()
{
	int nlines, maxlines = 10;
	int k, n, err, *count;
	int i;
	double nchars = 0;
	FILE *fd;
	 //Adding malloc for space
	wiki_array = (char **) malloc( WIKI_ARRAY_SIZE * sizeof(char *));
	for (i = 0; i < WIKI_ARRAY_SIZE; i++)
	{
		wiki_array[i] = malloc(2001);
	}
	//saved results
	longestSub = (char **) malloc( WIKI_ARRAY_SIZE * sizeof(char *));
	for (i = 0; i < WIKI_ARRAY_SIZE -1; i++)
	{
		longestSub[i] = malloc(2001);
	}
	fd = fopen("/homes/dan/625/wiki_dump.txt", "r");
	nlines = -1;
	do
	{
		err = fscanf(fd, "%[^\n]\n", wiki_array[++nlines]);
		if(wiki_array[nlines] != NULL)
			nchars += (double) strlen(wiki_array[nlines]);
	}
	while (err != EOF && nlines < WIKI_ARRAY_SIZE);

	fclose(fd);
	printf("Read in %d lines averaging %.01f chars/line\n", nlines, nchars / nlines);
}
void printToFile()
{
	FILE *f = fopen("LargestCommonSubstringsONETHREAD.txt", "w");
	if (f == NULL)
	{
		printf("Error opening LargestCommonSubstrings.txt!\n");
		exit(1);
	}
	int i;
	for(i = 0; i < WIKI_ARRAY_SIZE - 2; i++)
	{
		fprintf(f, "%d-%d: %s", i, i + 1,longestSub[i]);
		fprintf(f, "\n");
	}
	fclose(f);
}
void printResults()
{
	int i;
	for(i = 0; i <= WIKI_ARRAY_SIZE - 2; i++)
```

```
        {
                printf("%d-%d: %s", i , i + 1 ,longestSub[i]);
                printf("\n");
        }
}
int LCS(char *s1, char *s2, char **longest_common_substring)
{
        int s1_length = strlen(s1);
        int s2_length = strlen(s2);

        int **_matrix;
        int _matrix_row_size = 0;
        int _matrix_collumn_size = 0;
        if (s1_length+1 > _matrix_row_size || s2_length+1 > _matrix_collumn_size)
        {
                int i;
                /* malloc matrix */
                _matrix = (int **)malloc((s1_length+1) * sizeof(int*));
                for (i = 0; i < s1_length+1; i++)
                        _matrix[i] = (int *)malloc((s2_length+1) * sizeof(int));
                _matrix_row_size = s1_length+1;
                _matrix_collumn_size = s2_length+1;
        }
        int i;
        for (i = 0; i <= s1_length; i++)
                _matrix[i][s2_length] = 0;
        int j;
        for (j = 0; j <= s2_length; j++)
                _matrix[s1_length][j] = 0;
        int max_len = 0, max_index_i = -1;
    for (i = s1_length-1; i >= 0; i--)
    {
                for (j = s2_length-1; j >= 0; j--)
                {
                        if (s1[i] != s2[j])
                        {
                                _matrix[i][j] = 0;
                                continue;
                        }
                        _matrix[i][j] = _matrix[i+1][j+1] + 1;
                        if (_matrix[i][j] > max_len)
                        {
                                max_len = _matrix[i][j];
                                max_index_i = i;
                        }
                }
         }
        if (longest_common_substring != NULL)
        {
                omp_set_lock(&my_lock);
                *longest_common_substring = malloc(sizeof(char) * (max_len+1));
                strncpy(*longest_common_substring, s1+max_index_i, max_len);
                (*longest_common_substring)[max_len] = '\0';
                omp_unset_lock(&my_lock);
        }               /* free matrix */
        for (i = 0; i < _matrix_row_size; i++)
                free(_matrix[i]);
        free(_matrix);
        return max_len;
}
```

OpenMP Shell Scripts

https://github.com/shreyakumar1010/cis520-Project4/blob/master/OpenMP/openmp.sh

```
#!/bin/bash -l
for i in 1 2 3 4 5 6 7 8 9 10
do
  homes/coreyvessar/cis520/cis520-Project4/OpenMP/OpenMP $1 $2
done
```

https://github.com/shreyakumar1010/cis520-Project4/blob/master/OpenMP/openMPDoubleLoop.sh

```
#!/bin/bash -l
for i in 1000 10000 100000 500000 1000000
do
 for j in 1 2 4 8 16 32
 do
   corestring="_$j"
   job_name="omp$corestring"
   sbatch --mem-per-cpu=4G  --time=03:00:00  --partition=killable.q --nodes=1 --ntasks-per-node=$j --
constraint=dwarves --job-name=$job_name openmp.sh $j $i
   done
done
```

MPI

https://github.com/shreyakumar1010/cis520-Project4/blob/master/MPI/MPI.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#include <stdbool.h>
#include <mpi.h>
#define WIKI_ARRAY_SIZE 500
#define WIKI_LINE_SIZE 2001
#define MASTER 0
//int lengthOfSubstring [WIKI_ARRAY_SIZE];
int LCS (char * s1, char * s2, char ** longest_common_substring);
int num_threads;
char * numCores = "1";
char  ** wiki_array;
char ** longestSub;
char ** localLongestSub;
void readToMemory();
void printResults();
void printToFile();
void * findem(void * rank);
int main(int argc, char* argv[])
{
        struct timeval time1;
        struct timeval time2;
        struct timeval time3;
        struct timeval time4;
        double e1, e2, e3;
        int numSlots, Version = 4; //base = 1, pthread = 2, openmp = 3, mpi = 4
        //longestSub = (char **) malloc( WIKI_ARRAY_SIZE * sizeof(char *)); //saved results
        int i;
        //for (i = 0; i < WIKI_ARRAY_SIZE -1; i++)//initializing saved results
          //        longestSub[i] = malloc(2001);

        //==================INIT MPI======================
        int rc, NumTasks, rank;
        MPI_Status status;
        rc = MPI_Init(&argc, &argv);
        if(rc != MPI_SUCCESS)
```

```
	{
		printf("MPI isn't working");
		MPI_Abort(MPI_COMM_WORLD, rc);
	}
	MPI_Comm_size(MPI_COMM_WORLD, &NumTasks);
	MPI_Comm_rank(MPI_COMM_WORLD, &rank);
	num_threads = NumTasks;

	//==================READ TO MEMORY==================
	gettimeofday(&time1, NULL);
	<<<<<<< HEAD
	readToMemory(); //reading
	=======
	gettimeofday(&time2, NULL);//time to read to memory
	e1 = (time2.tv_sec - time1.tv_sec) * 1000.0; //sec to ms
	e1 += (time2.tv_usec - time1.tv_usec) / 1000.0; // us to ms
	printf("Time to read full file to Memory: %f\n", e1);
	//====FINDING LONGEST CMN SUBSTR. & PARALLELIZING====
>>>>>>> 75b7374bf3009e45a76c1b5ab6a6927162442560
	gettimeofday(&time3, NULL);
	MPI_Bcast(wiki_array, WIKI_ARRAY_SIZE * WIKI_LINE_SIZE, MPI_CHAR, 0, MPI_COMM_WORLD);
	findem(&rank);
	MPI_Reduce(localLongestSub, longestSub, WIKI_ARRAY_SIZE * WIKI_LINE_SIZE, MPI_CHAR,
MPI_SUM, 0, MPI_COMM_WORLD);
	if(rank == MASTER)
	{
	//=================FINAL TIMING==================
		gettimeofday(&time4, NULL);

		printToFile();
		//time to find all longest substrings
		e2 = (time4.tv_sec - time3.tv_sec) * 1000.0; //sec to ms
		e2 += (time4.tv_usec - time3.tv_usec) / 1000.0; // us to ms
		printf("Time find all Substrings: %f\n", e2);
		//total elapsed time between reading and finding all longest substrings
		e3 = (time4.tv_sec - time1.tv_sec) * 1000.0; //sec to ms
		e3 += (time4.tv_usec - time1.tv_usec) / 1000.0; // us to ms
		printf("DATA, %d, %s, %f, %d, %d, %s\n", Version, getenv("NSLOTS"), e3, num_threads,
WIKI_ARRAY_SIZE, numCores);
	}
	<<<<<<< HEAD
	else {
	     MPI_Finalize();
	     return -1;
	    }

	MPI_Finalize();
	return 0;
=======
	MPI_Finalize();
>>>>>>> 75b7374bf3009e45a76c1b5ab6a6927162442560
}
void * findem(void * rank)
{
	int myID = *((int *) rank);
	int startPos = ((long) myID) * (WIKI_ARRAY_SIZE / num_threads);
	int endPos = startPos + (WIKI_ARRAY_SIZE / num_threads);
	if(myID == num_threads - 1)
		endPos = WIKI_ARRAY_SIZE;
	int i, j;
		for(j = startPos; j < endPos; j++)
		{
			LCS((void*)wiki_array[i], (void*)wiki_array[i+1], localLongestSub);
```

```
                                localLongestSub++;
                        }
    }
}
void readToMemory()
{
        int nlines, maxlines = 10;
        int k, n, err, *count, nthreads = 24;
        int i;
        double nchars = 0;
        FILE *fd;
        wiki_array = (char **) malloc( WIKI_ARRAY_SIZE * sizeof(char *));
        for (i = 0; i < WIKI_ARRAY_SIZE; i++)//initializing the array
        {
                wiki_array[i] = malloc(2001);
        }
        longestSub = (char **) malloc( WIKI_ARRAY_SIZE * sizeof(char *));
        localLongestSub = (char **) malloc( WIKI_ARRAY_SIZE * sizeof(char *));
        //==================READING FILE=================
        fd = fopen("/homes/dan/625/wiki_dump.txt", "r");
        nlines = -1;
        do
        {
                err = fscanf(fd, "%[^\n]\n", wiki_array[++nlines]);
                if(wiki_array[nlines] != NULL)
                        nchars += (double) strlen(wiki_array[nlines]);
        }
        while (err != EOF && nlines < WIKI_ARRAY_SIZE);
        fclose(fd);
        printf("Read in %d lines averaging %.01f chars/line\n", nlines, nchars / nlines);
}
void printToFile()//self-explanitory. Prints the substrings found to a file for processing
{
        FILE *f = fopen("LargestCommonSubstrings.txt", "w");
        if (f == NULL)//if there's a problem with the file error handling
        {
                printf("Error opening LargestCommonSubstrings.txt!\n");
                exit(1);
        }
        //longestSub = longestSub - (WIKI_ARRAY_SIZE - 1);
        int i;
        for(i = 0; i < WIKI_ARRAY_SIZE - 2; i++)
        {
                fprintf(f, "%d-%d: %s", i, i + 1,longestSub[i]);
                fprintf(f, "\n");
        }
        fclose(f);
}
void printResults()//printing to console for debugging
{
        int i;
        //longestSub = longestSub - (WIKI_ARRAY_SIZE - 1);
        for(i = 0; i <= WIKI_ARRAY_SIZE - 2; i++)
        {
                printf("%d-%d: %s", i , i + 1 ,longestSub[i]);
                printf("\n");
        }
}
int LCS(char *s1, char *s2, char **longest_common_substring)
{
        int s1_length = strlen(s1);
        int s2_length = strlen(s2);
        int ** _matrix;
```

```
	int _matrix_row_size = 0;
	int _matrix_collumn_size = 0;
	if (s1_length+1 > _matrix_row_size || s2_length+1 > _matrix_collumn_size)
	{
		int i;
		/* malloc matrix */
		_matrix = (int **)malloc((s1_length+1) * sizeof(int*));
		for (i = 0; i < s1_length+1; i++)
			_matrix[i] = (int *)malloc((s2_length+1) * sizeof(int));
		_matrix_row_size = s1_length+1;
		_matrix_collumn_size = s2_length+1;
	}
	int i;
	for (i = 0; i <= s1_length; i++)
		_matrix[i][s2_length] = 0;
	int j;
	for (j = 0; j <= s2_length; j++)
		_matrix[s1_length][j] = 0;
	int max_len = 0, max_index_i = -1;
    for (i = s1_length-1; i >= 0; i--)
    {
		for (j = s2_length-1; j >= 0; j--)
		{
			if (s1[i] != s2[j])
			{
				_matrix[i][j] = 0;
				continue;
			}
			_matrix[i][j] = _matrix[i+1][j+1] + 1;
			if (_matrix[i][j] > max_len)
			{
				max_len = _matrix[i][j];
				max_index_i = i;
			}
		}
	 }
	if (longest_common_substring != NULL)
	{
		*longest_common_substring = malloc(sizeof(char) * (max_len+1));
		strncpy(*longest_common_substring, s1+max_index_i, max_len);
		(*longest_common_substring)[max_len] = '\0';
	}		/* free matrix */
	for (i = 0; i < _matrix_row_size; i++)
		free(_matrix[i]);
	free(_matrix);
	return max_len;
}
```

MPI Shell Scripts

https://github.com/shreyakumar1010/cis520-Project4/blob/master/MPI/mpi.sh

PThreads

https://github.com/shreyakumar1010/cis520-Project4/blob/master/Pthreads/Pthreads.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#include <stdbool.h>
#include <pthread.h>
//#define WIKI_ARRAY_SIZE 50000
#define WIKI_LINE_SIZE 2001
```

```
//int lengthOfSubstring [WIKI_ARRAY_SIZE];
int LCS (char * s1, char * s2, char ** longest_common_substring);
void loopingFunc(void *myID);
char * numCores = "0";
int WIKI_ARRAY_SIZE;
int num_threads;
//load the lines into an array
char  **wiki_array;
char **longestSub;
void readToMemory();
void printResults();
void printToFile();
int main(int argc, char** argv)
{
            WIKI_ARRAY_SIZE = atoi(argv[1]);
            num_threads = atoi(argv[2]);
            struct timeval time1;
            struct timeval time2;
            struct timeval time3;
            struct timeval time4;
            double e1, e2, e3;
            int numSlots, Version = 2; //base = 1, pthread = 2, openmp = 3, mpi = 4

            int myID, k, rc, i;
            pthread_t threads[num_threads];
            pthread_attr_t attr;
            void *status;
            /* Initialize and set thread detached attribute */
            pthread_attr_init(&attr);
            pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
            gettimeofday(&time1, NULL);
            readToMemory();
            gettimeofday(&time2, NULL);
            //time to read to memory
            e1 = (time2.tv_sec - time1.tv_sec) * 1000.0; //sec to ms
            e1 += (time2.tv_usec - time1.tv_usec) / 1000.0; // us to ms
            printf("Time to read full file to Memory: %f\n", e1);
            gettimeofday(&time3, NULL);
            for ( i = 0; i < num_threads; i++)
            {
                rc = pthread_create(&threads[i], &attr, loopingFunc, (void *)i);
          if(rc)
                {
                        printf("ERROR; return code from pthread_create() is %d\n", rc);
                    exit(-1);
                 }
            }
            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);
           }
      }
            //printResults();
            printToFile();
            gettimeofday(&time4, NULL);
            //time to find all longest substrings
            e2 = (time4.tv_sec - time3.tv_sec) * 1000.0; //sec to ms
```

```
	e2 += (time4.tv_usec - time3.tv_usec) / 1000.0; // us to ms
	printf("Time find all Substrings: %f\n", e2);

	//total elapsed time between reading and finding all longest substrings
	e3 = (time4.tv_sec - time1.tv_sec) * 1000.0; //sec to ms
	e3 += (time4.tv_usec - time1.tv_usec) / 1000.0; // us to ms
	printf("DATA, %d, %s, %f, %d, %d\n", Version, getenv("NSLOTS"), e3, num_threads, WIKI_ARRAY_SIZE);
}
void loopingFunc(void *myID)
{
	//start position of the array
	int startPos = ((int) myID) * (WIKI_ARRAY_SIZE / num_threads);
	char ** tempPos = longestSub + startPos;
	//end position of the array
	int endPos = startPos + (WIKI_ARRAY_SIZE / num_threads);
	if((int)myID == num_threads -1)
	{
	endPos = WIKI_ARRAY_SIZE - 1 ;
	}
	int i, j;
	  for(j = startPos; j < endPos; j++)
	  {
			printf("%d-%d: %s", j , j + 1 ,"lines submitted to LCS");
			printf("\n");
		LCS((void*)wiki_array[j], (void*)wiki_array[j+1], tempPos);
		tempPos++;
	  }
	pthread_exit(NULL);
}
void readToMemory()
{
	int nlines, maxlines = 10;
	int k, n, err;
	int i;
	double nchars = 0;
	FILE *fd;
	 //Adding malloc for space
	wiki_array = (char **) malloc( WIKI_ARRAY_SIZE * sizeof(char *));
	for (i = 0; i < WIKI_ARRAY_SIZE; i++)
	{
		wiki_array[i] = malloc(2001 * sizeof(char));
	}
	//saved results
	longestSub = (char **) malloc( WIKI_ARRAY_SIZE * sizeof(char *));
	fd = fopen("/homes/dan/625/wiki_dump.txt", "r");
	//fd = fopen("/homes/coreyvessar/cis520/cis520-Project4/wiki_dump.txt", "r");
	nlines = -1;
	do
	{
		err = fscanf(fd, "%[^\n]\n", wiki_array[++nlines]);
		if(wiki_array[nlines] != NULL)
			nchars += (double) strlen(wiki_array[nlines]);
	}
	while (err != EOF && nlines < WIKI_ARRAY_SIZE);
	fclose(fd);
	printf("Read in %d lines averaging %.01f chars/line\n", nlines, nchars / nlines);
}
void printToFile()
{
	FILE *f = fopen("pthreadsCommonSubstrings.txt", "w");
	if (f == NULL)
	{
		printf("Error opening LargestCommonSubstrings.txt!\n");
```

```
                exit(1);
        }

        //longestSub = longestSub - (WIKI_ARRAY_SIZE - 1);
        int i;
        for(i = 0; i < WIKI_ARRAY_SIZE - 2; i++)
        {
                fprintf(f, "%d-%d: %s", i, i + 1,longestSub[i]);
                fprintf(f, "\n");
        }

        fclose(f);
}
void printResults()
{
        int i;
        //longestSub = longestSub - (WIKI_ARRAY_SIZE - 1);
        for(i = 0; i <= WIKI_ARRAY_SIZE - 2; i++)
        {
                printf("%d-%d: %s", i , i + 1 ,longestSub[i]);
                printf("\n");
        }
}
int LCS(char *s1, char *s2, char **longest_common_substring)
{
        int s1_length = strlen(s1);
        int s2_length = strlen(s2);

        int ** _matrix;
        int _matrix_row_size = 0;
        int _matrix_collumn_size = 0;

        if (s1_length+1 > _matrix_row_size || s2_length+1 > _matrix_collumn_size)
        {
                int i;
                /* malloc matrix */
                _matrix = (int **)malloc((s1_length+1) * sizeof(int*));
                for (i = 0; i < s1_length+1; i++)
                        _matrix[i] = (int *)malloc((s2_length+1) * sizeof(int));
                _matrix_row_size = s1_length+1;
                _matrix_collumn_size = s2_length+1;
        }
        int i;
        for (i = 0; i <= s1_length; i++)
                _matrix[i][s2_length] = 0;
        int j;
        for (j = 0; j <= s2_length; j++)
                _matrix[s1_length][j] = 0;
        int max_len = 0, max_index_i = -1;
    for (i = s1_length-1; i >= 0; i--)
    {
                for (j = s2_length-1; j >= 0; j--)
                {
                        if (s1[i] != s2[j])
                        {
                                _matrix[i][j] = 0;
                                continue;
                        }
                        _matrix[i][j] = _matrix[i+1][j+1] + 1;

                        if (_matrix[i][j] > max_len)
                        {
                                max_len = _matrix[i][j];
```

```
                                        max_index_i = i;
                                }
                        }
                }
        if (longest_common_substring != NULL)
        {
                //omp_set_lock(&theLock);
                *longest_common_substring = malloc(sizeof(char) * (max_len+1));
                strncpy(*longest_common_substring, s1+max_index_i, max_len);
                (*longest_common_substring)[max_len] = '\0';
                //omp_unset_lock(&theLock);
                //printf("%s\n", *longest_common_substring);
        }                       /* free matrix */
        for (i = 0; i < _matrix_row_size; i++)
                free(_matrix[i]);
        free(_matrix);
        return max_len;
}
```

PThreads Shell Scripts

https://github.com/shreyakumar1010/cis520-Project4/blob/master/Pthreads/pthread.sh

```
#!/bin/bash -l
for i in 1 2 3 4 5 6 7 8 9 10
do
 homes/coreyvessar/cis520/cis520-Project4/Pthreads/Pthreads $1 $2
done
```

https://github.com/shreyakumar1010/cis520-Project4/blob/master/Pthreads/pthreadsDoubleLoop.sh

```
#!/bin/bash -l
for i in 1000 10000 100000 500000 1000000
do
 for j in 1 2 4 8 16 32
 do
  corestring="_$j"
  job_name="pthreads$corestring"
  sbatch --mem-per-cpu=4G  --time=03:00:00  --partition=killable.q --nodes=1 --ntasks-per-node=$j --
constraint=dwarves --job-name=$job_name pthreads.sh $j $i
  done
done
```

References

[1] Hughes, Christian J. “ChristianJHughes/Pintos-project4.” GitHub, 9 May 2017, github.com/ChristianJHughes/pintos-project4.

[2] Barney, Blaise. “Open MP.” OpenMP, U.S. Department of Energy, 17 July 2017, computing.llnl.gov/tutorials/openMP/.

[3] Barney, Blaise. “POSIX Threads Programming.” POSIX Threads Programming, U.S. Department of Energy, 7 Mar. 2017, computing.llnl.gov/tutorials/pthreads/.

[4] Barney, Blaise. “Message Passing Interface (MPI).” Message Passing Interface (MPI), U.S. Department of Energy, 19 June 2017, computing.llnl.gov/tutorials/mpi/.

[5] “Longest Common Substring Problem.” Wikipedia, Wikimedia Foundation, 9 Nov. 2017, en.wikipedia.org/wiki/Longest_common_substring_problem.

[6] Hancy. “Hancyxhx/Longest-Common-Substring.” GitHub, 20 Mar. 2014, github.com/hancyxhx/Longest-Common-Substring.

[7] “Parallel Programming Techniques.” Parallel Programming Techniques, 27 Mar. 2017, jcsites.juniata.edu/faculty/rhodes/smui/parex.htm.

[8] “What Is OpenMP.” Into To OpenMP, Dartmouth College, 23 Mar. 2009, www.dartmouth.edu/~rc/classes/intro_openmp/.

[9] “POSIX Threads.” Wikipedia, Wikimedia Foundation, 24 Apr. 2018, en.wikipedia.org/wiki/POSIX_Threads.

[10] Schroeder, Simon. “Which One Do You Prefer: UDA or OpenCL.” Research Gate, Research Gate, 2014, www.researchgate.net/post/Which_one_do_you_prefer_CUDA_or_OpenCL.