Project #4 – One Program, Three ways
Performance Analysis

Ryan Williams, Mark Spicer, Matt Heffel

Introduction

We were given the task of finding the largest common substring in large text file between wikipedia entries. The text file we used contained 1 million wikipedia entries, where each line contained 1 entry.

To solve this, we had an implementation similar to the dynamic programming approach used at https://www.geeksforgeeks.org/longest-common-substring/. We created a 2D array in which each value was initialized to 1. The y axis of the table corresponded to the letters of the first string, and the x axis corresponded to the second string. If two letters equalled each other, then the cell became 1 greater than its left diagonal cell, where the right diagonal sequences of numbers represented substrings, like the this:

```
1
  2
    3
```

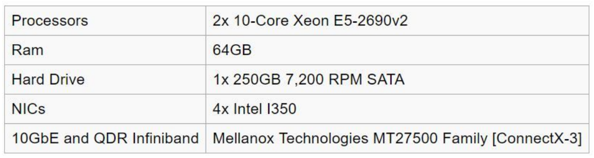
When the table was filled, we went back through the table to find the greatest right diagonal sequence of increasing numbers. This gave us a backwards version of the longest common substring, so we flipped the substring to get our final result.

It was then our job to parallelize this operation across the Beocat High Performance Computing Cluster at Kansas State University. We did this parallelization with the openMP, pthreads, and MPI libraries.

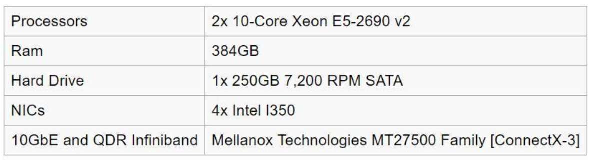
Configuration

Hardware

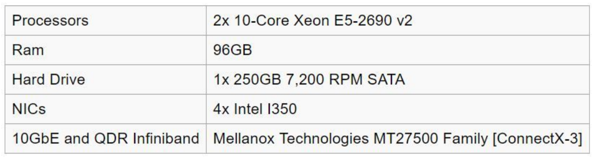
All of our jobs were submitted to the "elves" nodes, which had the following four configurations:



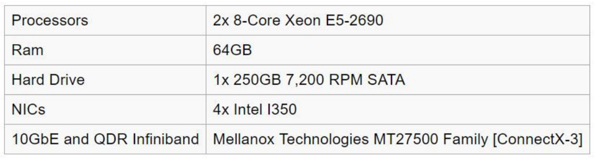
Processors	2x 10-Core Xeon E5-2690v2
Ram	64GB
Hard Drive	1x 250GB 7,200 RPM SATA
NICs	4x Intel I350
10GbE and QDR Infiniband	Mellanox Technologies MT27500 Family [ConnectX-3]



Processors	2x 10-Core Xeon E5-2690 v2
Ram	384GB
Hard Drive	1x 250GB 7,200 RPM SATA
NICs	4x Intel I350
10GbE and QDR Infiniband	Mellanox Technologies MT27500 Family [ConnectX-3]



Processors	2x 10-Core Xeon E5-2690 v2
Ram	96GB
Hard Drive	1x 250GB 7,200 RPM SATA
NICs	4x Intel I350
10GbE and QDR Infiniband	Mellanox Technologies MT27500 Family [ConnectX-3]



Processors	2x 8-Core Xeon E5-2690
Ram	64GB
Hard Drive	1x 250GB 7,200 RPM SATA
NICs	4x Intel I350
10GbE and QDR Infiniband	Mellanox Technologies MT27500 Family [ConnectX-3]

Software

We ran our tests with the following software versions:

- **BASH:** v4.2.46
- **Linux:** v3.10 x86_4
- **gcc:** v4.8.5

Implementation

For openMP and pThreads, we ran tests on the first 10k lines of the text file, 100k lines, 500k lines, and 1 million lines. We ran our tests on Beocat using 1 core, 2 cores, 4 cores, 8 cores, and 16 cores. We never exceeded 16 because 16 cores was the maximum number of cores we were able to test on the elves nodes. Each CPU was given 16 divided by the number of cores of memory. openMP and pThreads were tested 5 times and 6 times for each configuration, respectively. We calculated the average of these runs to get our results.

MPI was tested on 1 million lines, and was run 3 times on 1 core, 2 cores, 4 cores, 8 cores, and 16 cores.

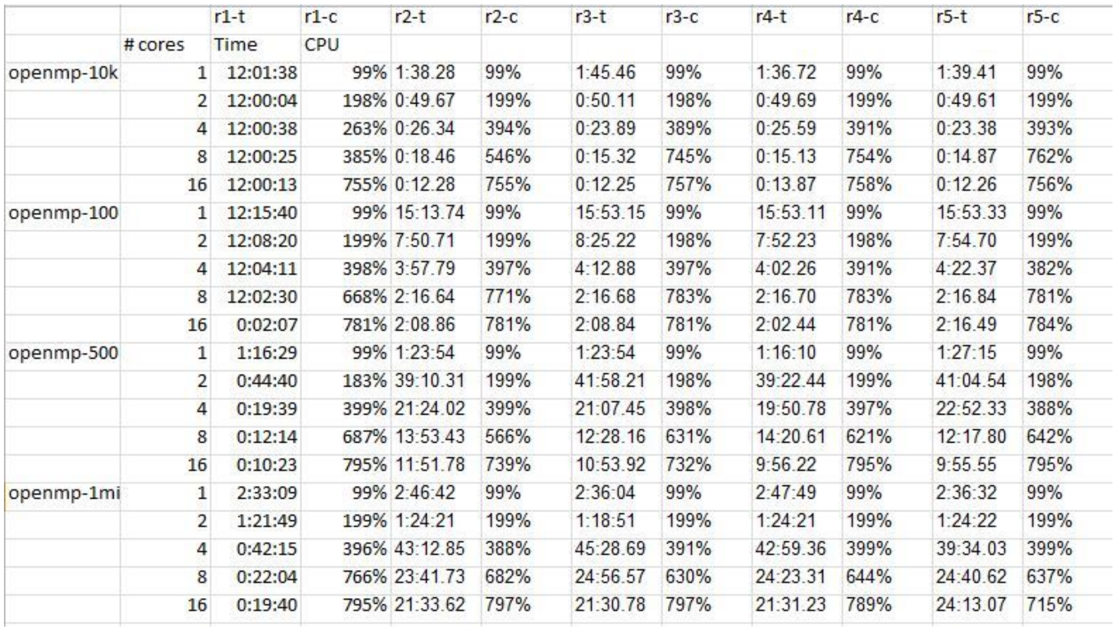
We mistakenly only specified a maximum of 8 threads in openMP. Because of this, the 16 core data in our openMP results are not accurate.

We kept track of the time it took to complete each test and the CPU utilization in each run using /usr/bin/time.

Experimental Results

OpenMP

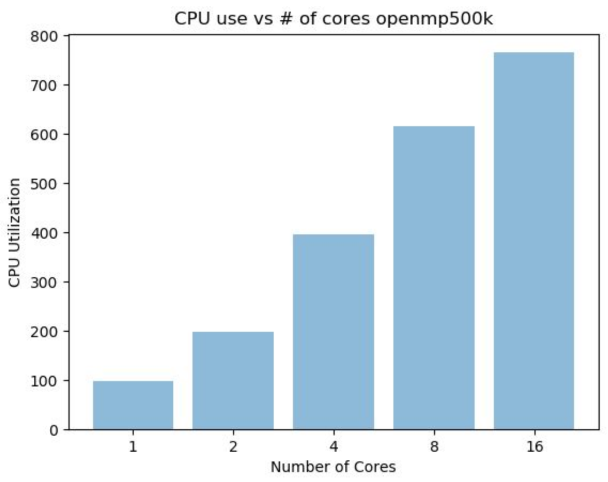
Parallelizing our code with the openMP library gave us the following CPU utilizations and execution times:

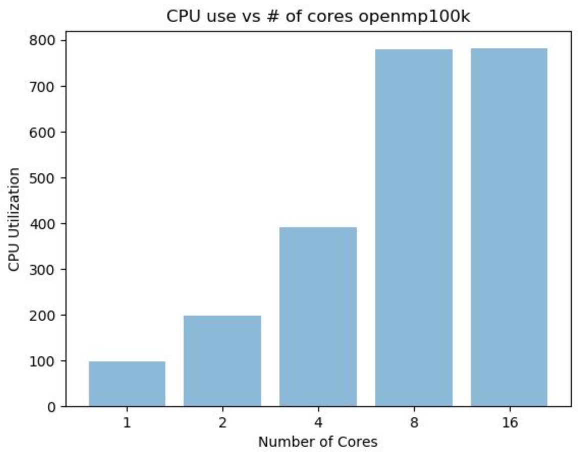


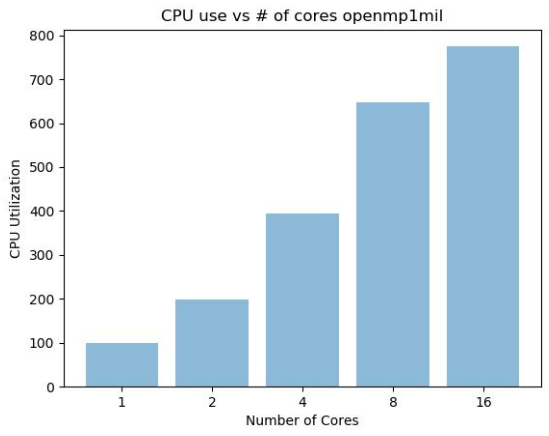
		r1-t	r1-c	r2-t	r2-c	r3-t	r3-c	r4-t	r4-c	r5-t	r5-c
	# cores	Time	CPU								
openmp-10k	1	12:01:38	99%	1:38.28	99%	1:45.46	99%	1:36.72	99%	1:39.41	99%
	2	12:00:04	198%	0:49.67	199%	0:50.11	198%	0:49.69	199%	0:49.61	199%
	4	12:00:38	263%	0:26.34	394%	0:23.89	389%	0:25.59	391%	0:23.38	393%
	8	12:00:25	385%	0:18.46	546%	0:15.32	745%	0:15.13	754%	0:14.87	762%
	16	12:00:13	755%	0:12.28	755%	0:12.25	757%	0:13.87	758%	0:12.26	756%
openmp-100	1	12:15:40	99%	15:13.74	99%	15:53.15	99%	15:53.11	99%	15:53.33	99%
	2	12:08:20	199%	7:50.71	199%	8:25.22	198%	7:52.23	198%	7:54.70	199%
	4	12:04:11	398%	3:57.79	397%	4:12.88	397%	4:02.26	391%	4:22.37	382%
	8	12:02:30	668%	2:16.64	771%	2:16.68	783%	2:16.70	783%	2:16.84	781%
	16	0:02:07	781%	2:08.86	781%	2:08.84	781%	2:02.44	781%	2:16.49	784%
openmp-500	1	1:16:29	99%	1:23:54	99%	1:23:54	99%	1:16:10	99%	1:27:15	99%
	2	0:44:40	183%	39:10.31	199%	41:58.21	198%	39:22.44	199%	41:04.54	198%
	4	0:19:39	399%	21:24.02	399%	21:07.45	398%	19:50.78	397%	22:52.33	388%
	8	0:12:14	687%	13:53.43	566%	12:28.16	631%	14:20.61	621%	12:17.80	642%
	16	0:10:23	795%	11:51.78	739%	10:53.92	732%	9:56.22	795%	9:55.55	795%
openmp-1mi	1	2:33:09	99%	2:46:42	99%	2:36:04	99%	2:47:49	99%	2:36:32	99%
	2	1:21:49	199%	1:24:21	199%	1:18:51	199%	1:24:21	199%	1:24:22	199%
	4	0:42:15	396%	43:12.85	388%	45:28.69	391%	42:59.36	399%	39:34.03	399%
	8	0:22:04	766%	23:41.73	682%	24:56.57	630%	24:23.31	644%	24:40.62	637%
	16	0:19:40	795%	21:33.62	797%	21:30.78	797%	21:31.23	789%	24:13.07	715%

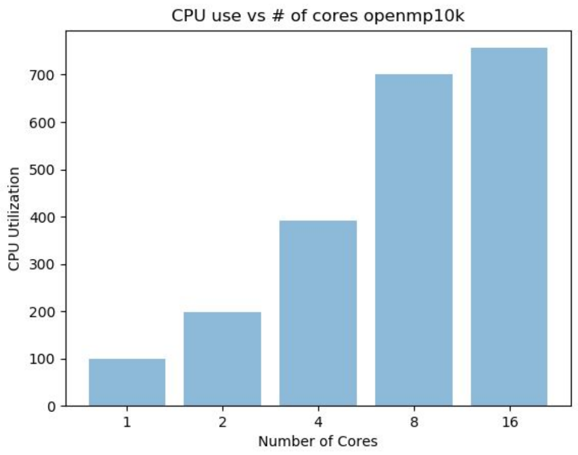
OpenMP: CPU Use vs Number of Cores

As expected, the CPU usage rose significantly each time the number of cores doubled. CPU utilization was proportional to the number of cores. That is, CPU utilization doubled each time the number of cores doubled. Utilization did not double going from 8 threads to 16 threads, but this is likely because we only specified 8 threads in our implementation.



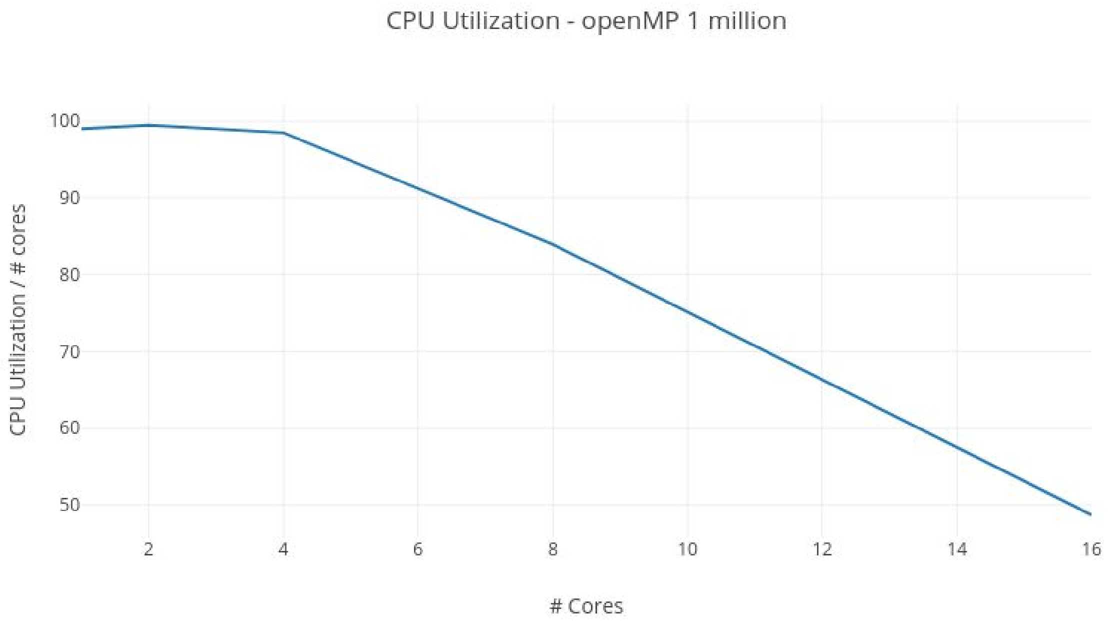






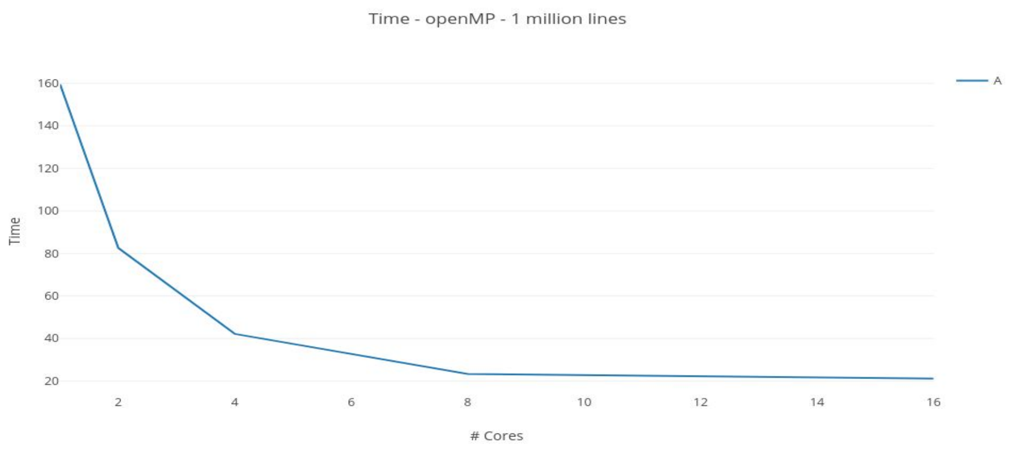
OpenMP: (CPU Utilization/Number of Cores) vs Number of Cores

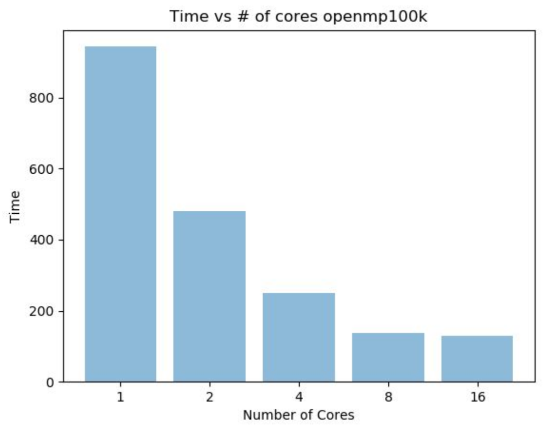
Getting the CPU utilization for each core shows us that CPU utilization goes down linearly as more than 4 cores are added .

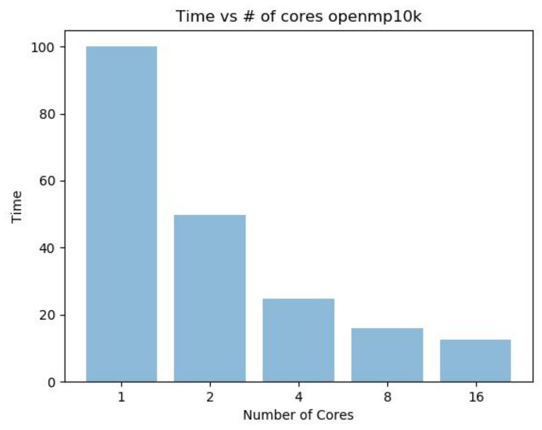


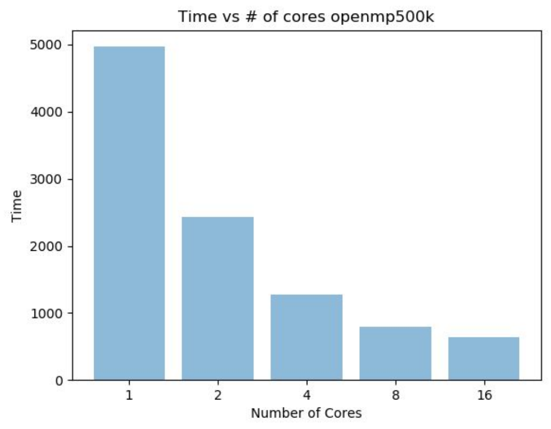
OpenMP: Time to read all lines vs number of cores

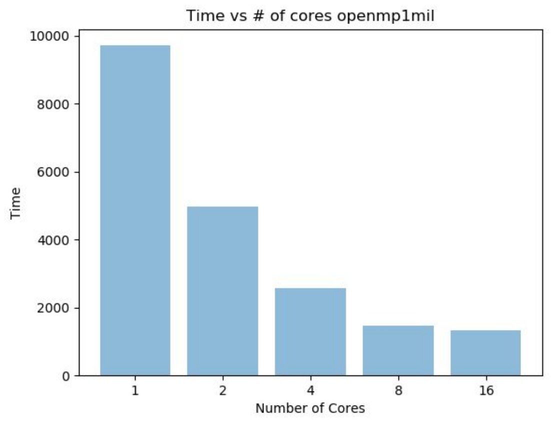
The time to find the longest substring of each line decreases by about 50% when the number of cores double. This is likely because the work is distributed evenly among the cores, so when twice as many cores are added, the work on each CPU is split in half. The decrease is not exactly 50% each time because, for each core added, an extra amount of time has to be spent managing and communicating between the cores. The time drop between 8 and 16 cores is not close to 50%, but this is likely because our implementation only utilized 8 cores.





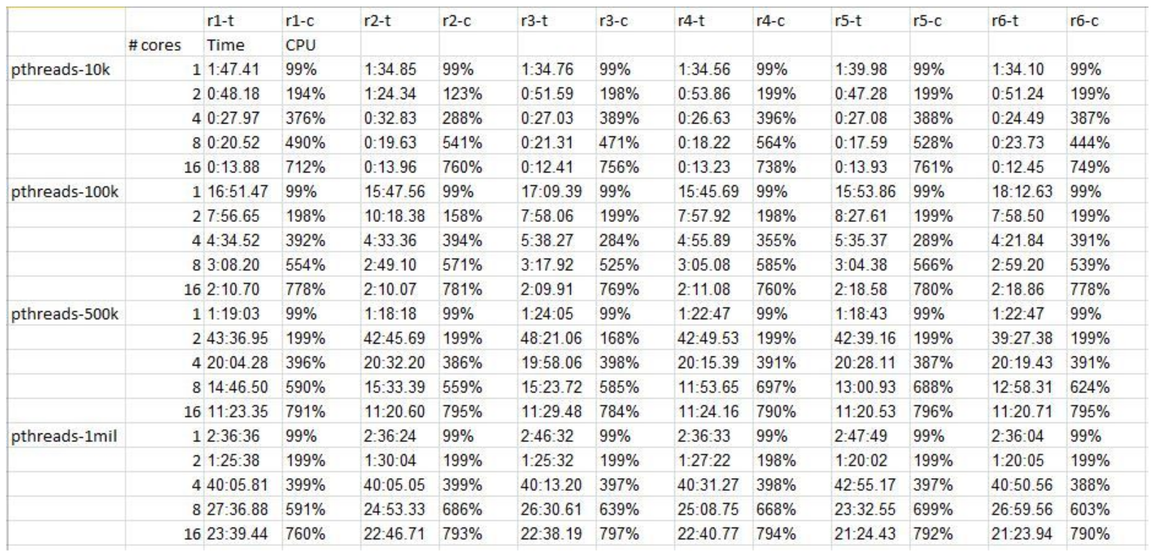






pThreads

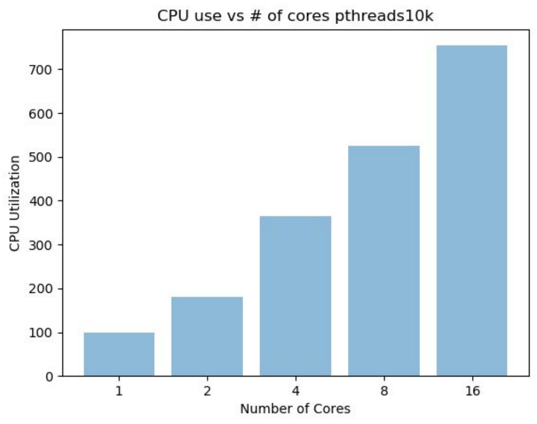
Parallelizing our code using the pThreads library gave us the following:

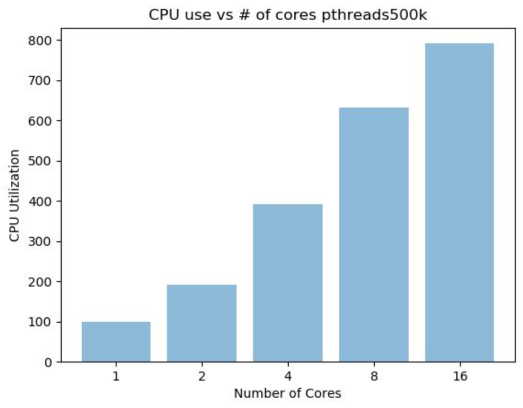


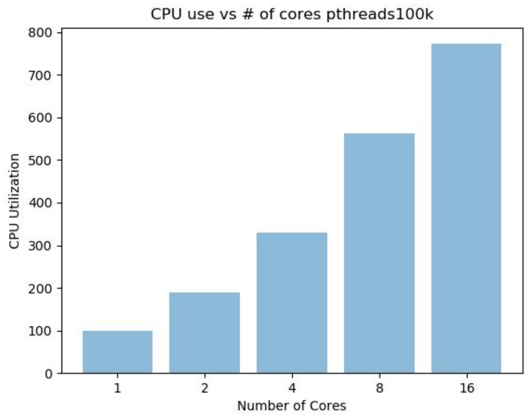
		r1-t	r1-c	r2-t	r2-c	r3-t	r3-c	r4-t	r4-c	r5-t	r5-c	r6-t	r6-c
	# cores	Time	CPU										
pthreads-10k	1	1:47.41	99%	1:34.85	99%	1:34.76	99%	1:34.56	99%	1:39.98	99%	1:34.10	99%
	2	0:48.18	194%	1:24.34	123%	0:51.59	198%	0:53.86	199%	0:47.28	199%	0:51.24	199%
	4	0:27.97	376%	0:32.83	288%	0:27.03	389%	0:26.63	396%	0:27.08	388%	0:24.49	387%
	8	0:20.52	490%	0:19.63	541%	0:21.31	471%	0:18.22	564%	0:17.59	528%	0:23.73	444%
	16	0:13.88	712%	0:13.96	760%	0:12.41	756%	0:13.23	738%	0:13.93	761%	0:12.45	749%
pthreads-100k	1	16:51.47	99%	15:47.56	99%	17:09.39	99%	15:45.69	99%	15:53.86	99%	18:12.63	99%
	2	7:56.65	198%	10:18.38	158%	7:58.06	199%	7:57.92	198%	8:27.61	199%	7:58.50	199%
	4	4:34.52	392%	4:33.36	394%	5:38.27	284%	4:55.89	355%	5:35.37	289%	4:21.84	391%
	8	3:08.20	554%	2:49.10	571%	3:17.92	525%	3:05.08	585%	3:04.38	566%	2:59.20	539%
	16	2:10.70	778%	2:10.07	781%	2:09.91	769%	2:11.08	760%	2:18.58	780%	2:18.86	778%
pthreads-500k	1	1:19:03	99%	1:18:18	99%	1:24:05	99%	1:22:47	99%	1:18:43	99%	1:22:47	99%
	2	43:36.95	199%	42:45.69	199%	48:21.06	168%	42:49.53	199%	42:39.16	199%	39:27.38	199%
	4	20:04.28	396%	20:32.20	386%	19:58.06	398%	20:15.39	391%	20:28.11	387%	20:19.43	391%
	8	14:46.50	590%	15:33.39	559%	15:23.72	585%	11:53.65	697%	13:00.93	688%	12:58.31	624%
	16	11:23.35	791%	11:20.60	795%	11:29.48	784%	11:24.16	790%	11:20.53	796%	11:20.71	795%
pthreads-1mil	1	2:36:36	99%	2:36:24	99%	2:46:32	99%	2:36:33	99%	2:47:49	99%	2:36:04	99%
	2	1:25:38	199%	1:30:04	199%	1:25:32	199%	1:27:22	198%	1:20:02	199%	1:20:05	199%
	4	40:05.81	399%	40:05.05	399%	40:13.20	397%	40:31.27	398%	42:55.17	397%	40:50.56	388%
	8	27:36.88	591%	24:53.33	686%	26:30.61	639%	25:08.75	668%	23:32.55	699%	26:59.56	603%
	16	23:39.44	760%	22:46.71	793%	22:38.19	797%	22:40.77	794%	21:24.43	792%	21:23.94	790%

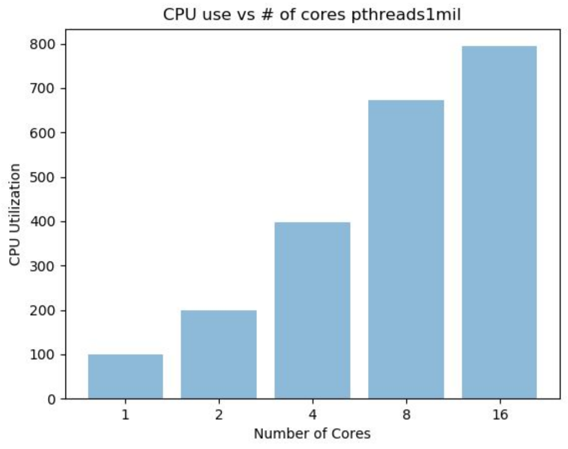
pThreads: CPU Use vs Number of Cores

The CPU utilization of pthreads was similar to OpenMP's. The CPU usage rose significantly each time a core was added because there were more cores to utilize. CPU utilization nearly doubled each time the number of cores were doubled.



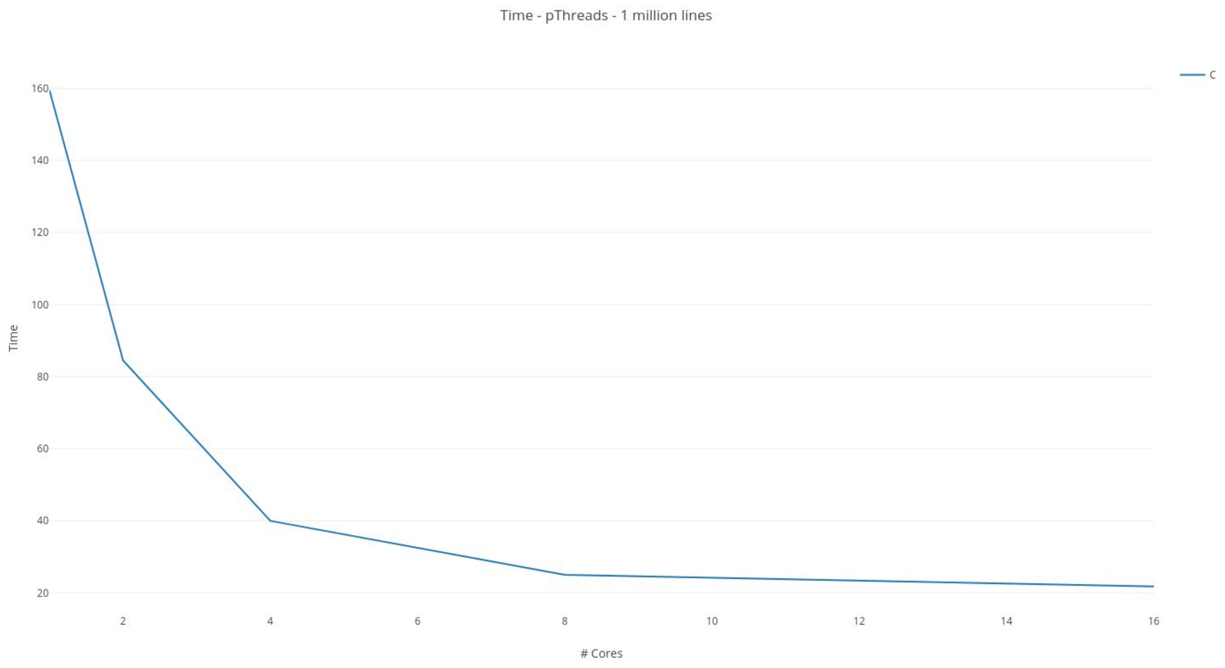


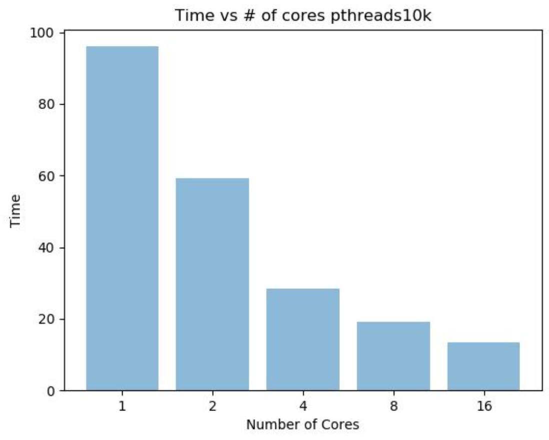


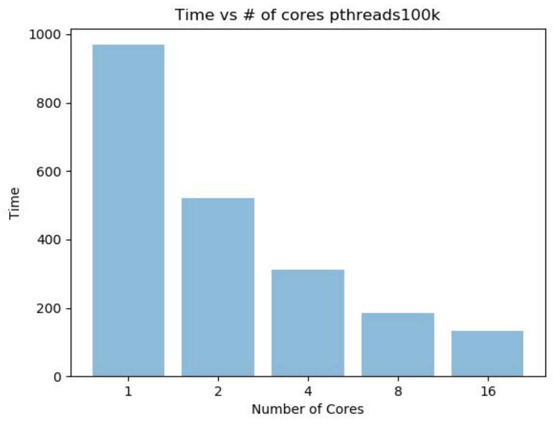


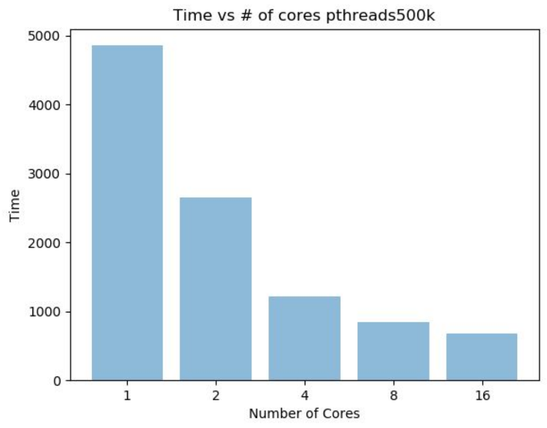
pThreads: Time to read all lines vs number of cores

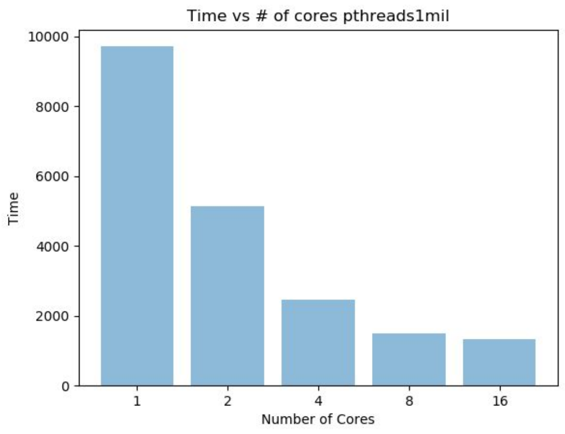
Also like OpenMP, the time to perform computations on every line decreases as the number of cores increase. The speed up is about twice as fast each time the number of cores is doubled. Unlike OpenMP, the trend continues and does not flatten out for the jump between 8 cores and 16 cores, because our pThreads implementation utilized all 16 cores.


Time - pThreads - 1 million lines
C
Time
# Cores
160
140
120
100
80
60
40
20
2
4
6
8
10
12
14
16


Time vs # of cores pthreads10k
Time
Number of Cores
1
2
4
8
16

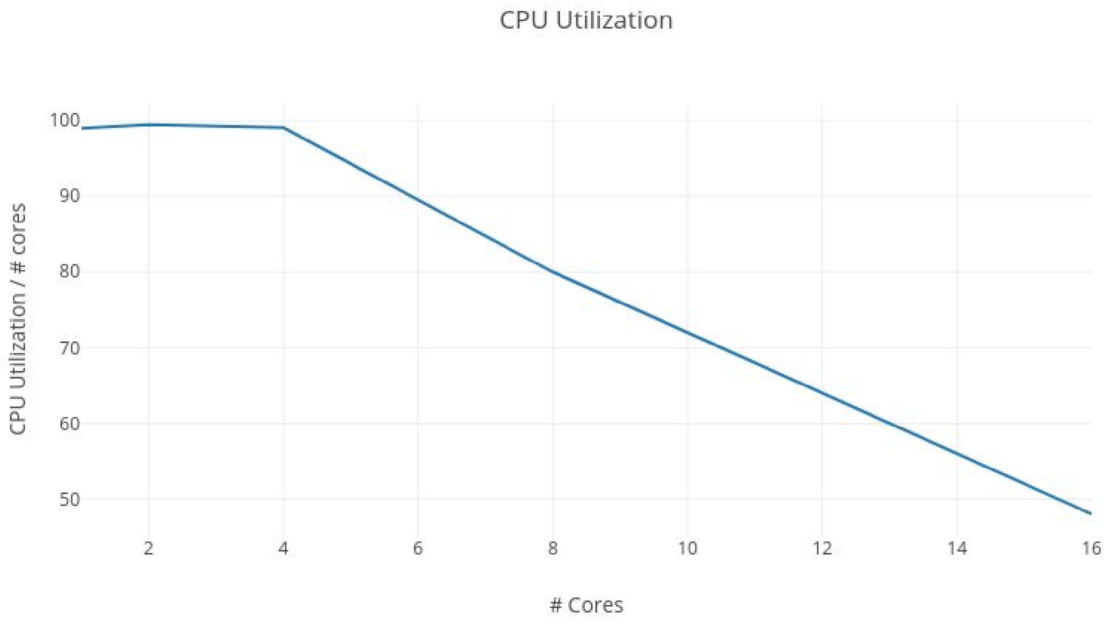

Time vs # of cores pthreads100k
Time
Number of Cores
1
2
4
8
16


Time vs # of cores pthreads500k
Time
Number of Cores
1
2
4
8
16


Time vs # of cores pthreads1mil
Time
Number of Cores
1
2
4
8
16

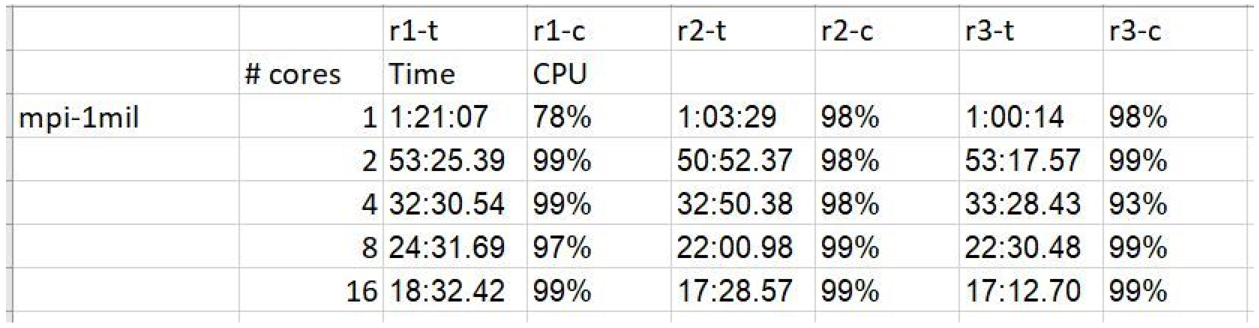
pThreads: (CPU Utilization/Number of Cores) vs Number of Cores

When we divide the CPU utilization by the number of cores from our pThreads implementation that read all 1 million lines, we can see that CPU utilization for each CPU significantly decreases when more than 4 cores are used.



MPI

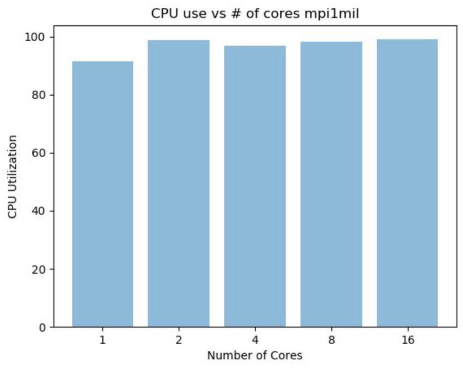
Parallelizing our code using the MPI library gave us the following:



		r1-t	r1-c	r2-t	r2-c	r3-t	r3-c
	# cores	Time	CPU				
mpi-1mil	1	1:21:07	78%	1:03:29	98%	1:00:14	98%
	2	53:25.39	99%	50:52.37	98%	53:17.57	99%
	4	32:30.54	99%	32:50.38	98%	33:28.43	93%
	8	24:31.69	97%	22:00.98	99%	22:30.48	99%
	16	18:32.42	99%	17:28.57	99%	17:12.70	99%

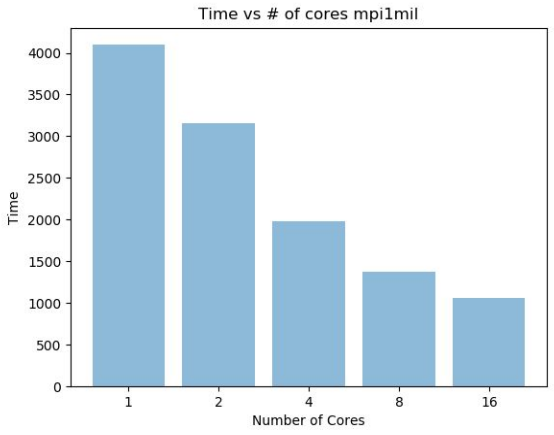
MPI: CPU Use vs Number of Cores

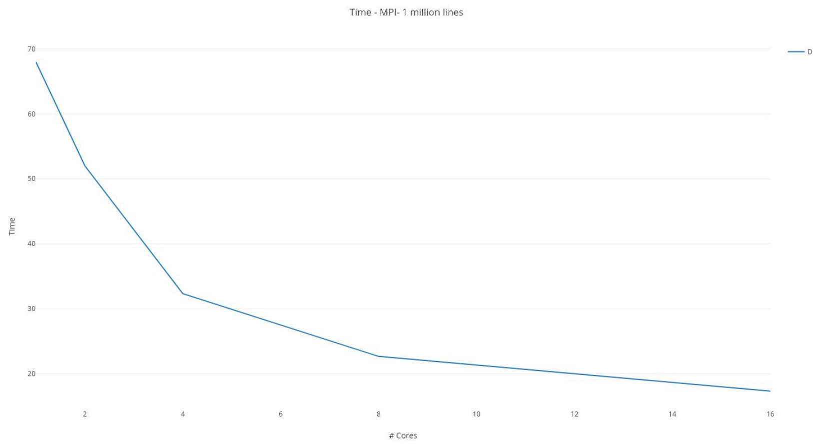
MPI performed differently from the other two parallelization methods. Instead of the CPU utilization increasing when more cores are added, MPI ran at nearly 100% CPU utilization for every test, despite the number of cores.



MPI: Time vs Number of Cores

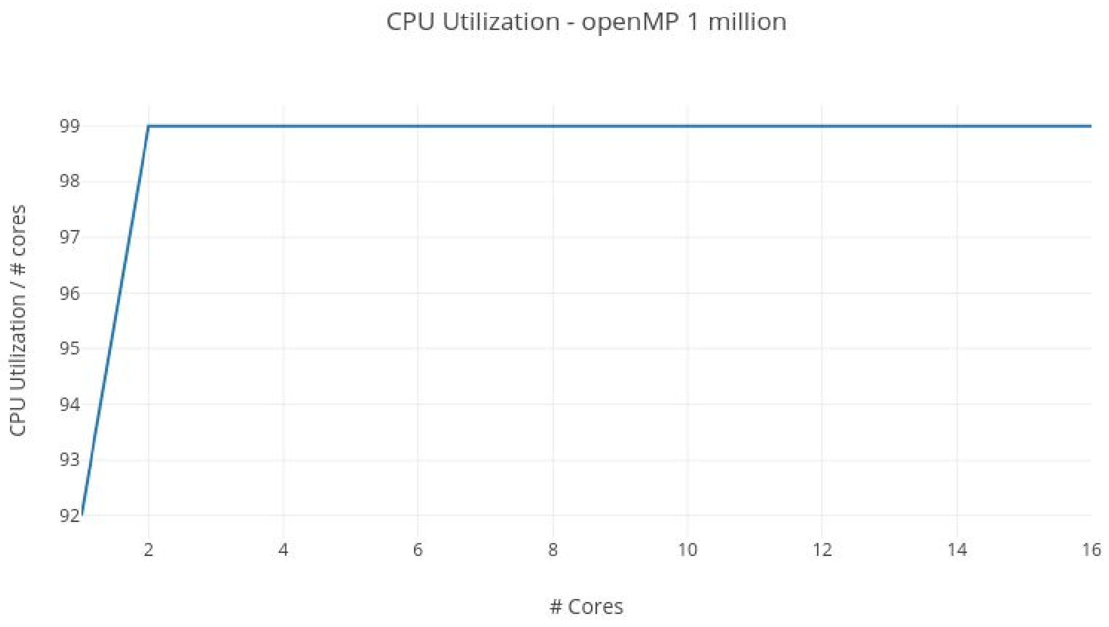
Like OpenMP and pthreads, the time to run each test decreased every time the number of cores increased. The decrease was about 25% each time the number of cores was doubled, as opposed to the close to 50% in pThreads and openMP.





MPI: (CPU Utilization/Number of Cores) vs Number of Cores

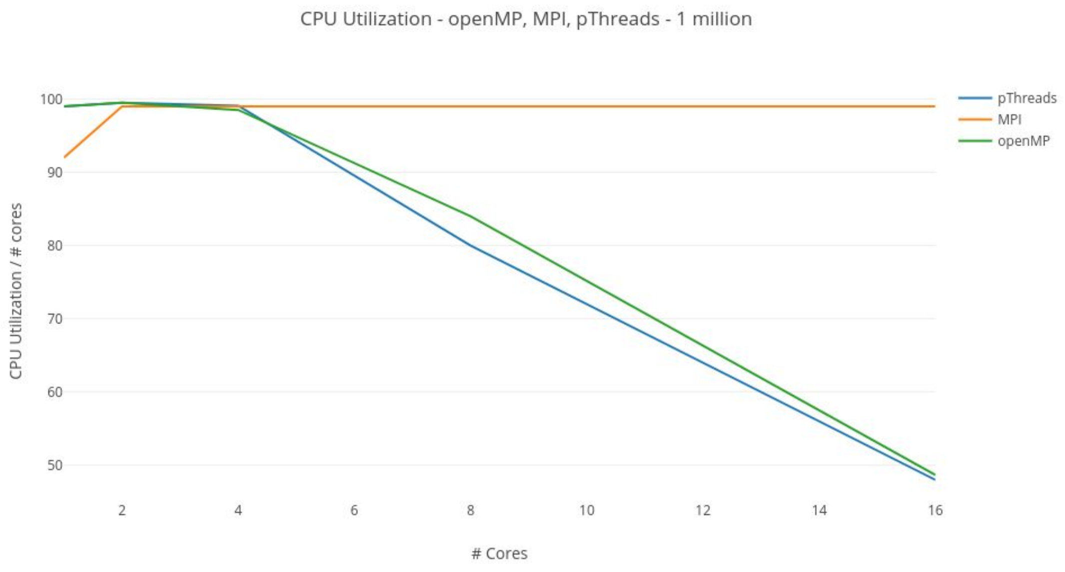
Each test, with the exception of 1 core, gave us a CPU utilization of 99% per CPU.



Performance: openMP vs pThreads vs MPI

CPU Utilization

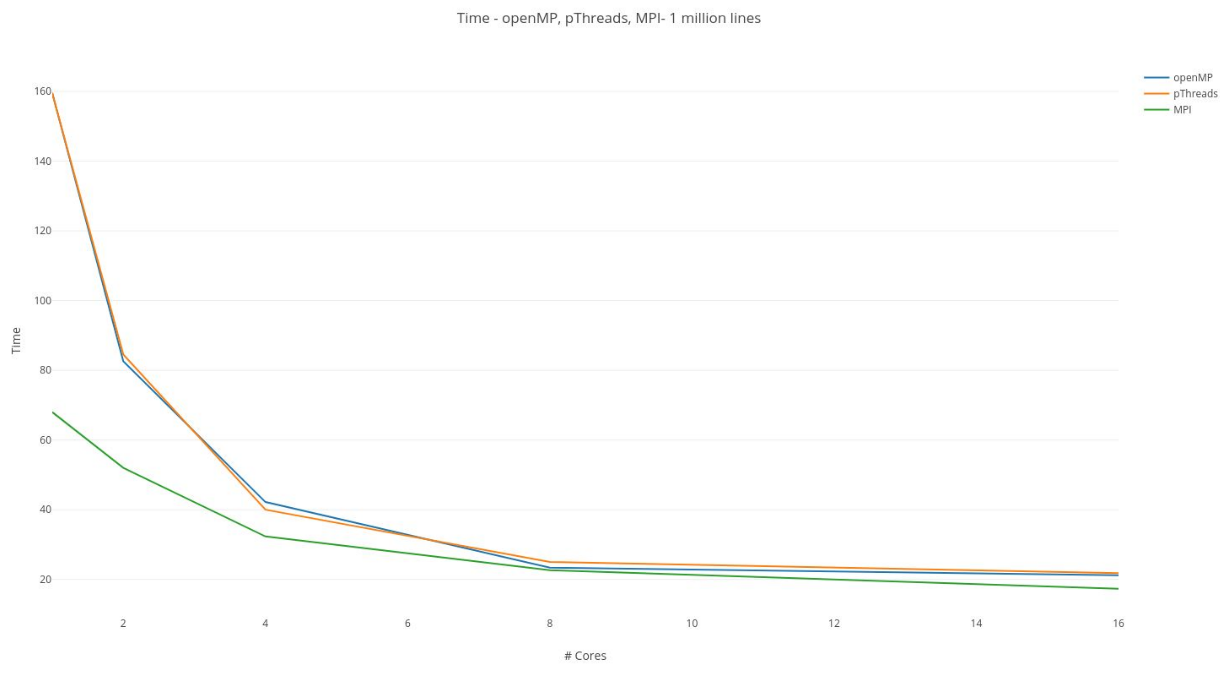
MPI had drastically different CPU utilization results when compared to openMP and pThreads. pThreads had the lowest CPU utilization overall. However, openMP was a very close 2nd.



Time

Of the 3 solutions, MPI completed the jobs the fastest by a significant margin. The greatest differences were in the configurations with a lower number of cores.

openMP and pThreads varied in their performances. pThreads ran faster by a small margin on 4 cores while pThreads ran faster on 8 cores by a small margin. Overall, openMP and pThreads had minimal differences in performance.



Conclusion

We wrote a program that found the longest common substring between wikipedia entries in a text file with 1 million lines. We parallelized this operation with the OpenMP, pThreads, and MPI libraries.

OpenMP and pThreads gave us similar CPU utilization results. However, pThreads had the least CPU utilization % per CPU, though not by much.

MPI had a CPU utilization that was drastically different from OpenMP and pThreads. It remained at close to 99% utilization regardless of the number of cores.

OpenMP and pThreads also had similar results in regards to the time it took to complete a test. Initially, pThreads was fastest, then OpenMP caught up and surpassed pThreads. We can conclude the performance of both in regards to time are nearly identical.

MPI, on the other hand, surpassed the time performance of both OpenMP and pThreads by a significant amount. This is due to the fact that OpenMP's and pThread's time dropped by about 50% each time the number of cores doubled, while MPI only dropped by 25%.

It is worth noting that our report never put the complexity of implementation into account. OpenMP was the simplest to implement, while MPI, which performed the best, was the most complex.

Appendix A
OpenMP source code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <omp.h>

/* Constants */
#define NUM_ENTRIES 1000000 // Should be 1000000
#define NUM_THREADS 8
#define LINE_LENGTH 2003 //should be 2003, increasing to this size causes a segmentation fualt due to size
of table.

/* Global Variables */
char entries[NUM_ENTRIES][LINE_LENGTH];

/* Function prototypes */
void max_substring(int myID);
char *strrev(char *str);
void read_file();

void main() {
        struct timeval t1, t2, t3, t4;
        double elapsedTime;
        int numSlots, myVersion = 2; // 1 = base, 2 = openmp, 3 = pthreads, 4 = mpi

        gettimeofday(&t1, NULL);
        /* Read the file into the the list of entries */
        read_file();
        gettimeofday(&t2, NULL);

        omp_set_num_threads(NUM_THREADS);

        gettimeofday(&t3, NULL);
        /* Get the max substring of each line */
        #pragma omp parallel
        {
                max_substring(omp_get_thread_num());
        }
        gettimeofday(&t4, NULL);

        elapsedTime = (t2.tv_sec - t1.tv_sec) * 1000.0; //sec to ms
        elapsedTime += (t2.tv_usec - t1.tv_usec) / 1000.0; // us to ms
        printf("Time to read file: %f\n", elapsedTime);

        elapsedTime = (t4.tv_sec - t3.tv_sec) * 1000.0; //sec to ms
        elapsedTime += (t4.tv_usec - t3.tv_usec) / 1000.0; // us to ms
        printf("Time to get max substrings: %f\n", elapsedTime);

        printf("DATA, %d, %s, %f\n", myVersion, getenv("SLURM_CPUS_ON_NODE"),  elapsedTime);
}

/* Read the file from wiki_dump.txt into the list of entries */
void read_file() {
        FILE *fp;
        char str1[LINE_LENGTH];
        fp = fopen("/homes/dan/625/wiki_dump.txt", "r");
        // fp = fopen("test.txt", "r");

        /* If the file could not be found, return */
        if(fp == NULL) {
                perror("Failed: ");
                return;
        }

        /* Add each line of the file into entries */
        int i = 0;
        while(fgets(str1, LINE_LENGTH, fp) != NULL && i < NUM_ENTRIES){
                strcpy(entries[i], str1);
                i++;
        }

        fclose(fp);
}

/* Find and prints the biggest AND most common substring between 2 str1 and str2.
   Code inspired by https://www.geeksforgeeks.org/longest-common-substring/ */
void max_substring(int myID) {
        int startPos = myID * (NUM_ENTRIES / NUM_THREADS);
        int endPos = startPos + (NUM_ENTRIES / NUM_THREADS);

        char str1[LINE_LENGTH];
        char str2[LINE_LENGTH];
        int m, n, i;
        char biggest[LINE_LENGTH]; // The biggest/most common substring
        char temp[LINE_LENGTH];

        int row, col;
        int biggest_row, biggest_col; // Index of the LAST letter of the biggest substring we've found
        int max_len; // The length of the biggest substring we've found

        #pragma omp private(str1,str2,startPos,endPos,m,n,i,biggest,temp,row,col,biggest_row,biggest_col)
                for(i = startPos; i < endPos; i++)
                {
                        strcpy(str1, entries[i]);
                        strcpy(str2, entries[i+1]);
                        m = strlen(str1);
                        n = strlen(str2);

                        /* Allocate memory for a table */
                        int (*table)[n] = malloc(sizeof(int[m + 1][n + 1]));

                        max_len = 0;
                        row =  0;
                        col = 0;
                        biggest_row = 0;
                        biggest_col = 0;

                        /* Populate the table */
                        for(row = 0; row < m; row++) {
                                for(col = 0; col < n; col++) {
                                        /* Initialize table[1..m][0] and table[0][1..n] to 1 */
                                        if(row == 0 || col == 0) {
                                                table[row][col] = 1;
                                        }
                                        /* If the letters of the two strings are the same, add 1 to the
left diagonal

                                           value and set it to the current position */
                                        else if(str1[row] == str2[col] && str1[row] != '\n'){
                                                table[row][col] = table[row-1][col-1] + 1;


                                                /* If the current position is bigger than the biggest
substring

                                                   we've found so far, update our variables so that we
can find

                                                   it later on. */
                                                if(table[row][col] > max_len) {
                                                        max_len = table[row][col];
```

```c
                                                        biggest_row = row;
                                                        biggest_col = col;
                                                }
                                        }
                                        /* If the letters are not the same, the substring length is 0 */
                                        else {
                                                table[row][col] = 0;
                                        }
                                }
                        }

                        if(max_len == 0) {
                                printf("%d-%d:, %s\n", i, i+1, "No common substring.");
                        }
                        else{
                                memset(biggest, '\0', LINE_LENGTH);
                                /* Starting at the bottom right of the biggest substring in the table,
build biggest substring */
                                while(table[biggest_row][biggest_col] != 1) {
                                        /* Convert the letter to a string, since strcat requires a string
*/

                                        memset(temp, '\0', LINE_LENGTH);

                                        temp[0] = str1[biggest_row];

                                        /* Concatonate the string */
                                        strcat(biggest, temp);

                                        /* Move to the next letter (top left) in the backwards substring
*/

                                        biggest_row--;
                                        biggest_col--;
                                }
                                strcat(biggest, "\0");

                                /* Print and reverse the biggest substring */
                                printf("%d-%d: %s\n", i, i+1, strrev(biggest));
                        }

                        /* Free the table */
                        free(table);
                }
}

/* Reverse a string
   From https://stackoverflow.com/questions/8534274/is-the-strrev-function-not-available-in-linux */
char *strrev(char *str)
{
      char *p1, *p2;

      if (! str || ! *str)
            return str;
      for (p1 = str, p2 = str + strlen(str) - 1; p2 > p1; ++p1, --p2)
      {
            *p1 ^= *p2;
            *p2 ^= *p1;
            *p1 ^= *p2;
      }
      return str;
}
```

Appendix B

OpenMP sbatch file

```
#!/bin/bash -l

        //usr/bin/time -o /homes/rpwilliams96/CIS520_Proj4/src/3way-openmp/
        /homes/rpwilliams96/CIS520_Proj4/src/3way-openmp/
```

Appendix C

OpenMP mass-sbatch file

```
#!/bin/bash



for i in 1 2 4 8 16

do

        for j in 1 2 3 4 5

        do

                sbatch --constraint=elves --time=04:00:00 --ntasks-per-node=$i --nodes=1
--mem-per-cpu=$((16/$i))G --partition=killable.q openmp-sbatch.sh
        done

done
```

Appendix D
Pthreads source code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

/* Constants */
#define NUM_ENTRIES 1000000 // Should be 1000000
#define NUM_THREADS 8
#define LINE_LENGTH 2003 //should be 2003, increasing to this size causes a segmentation fualt due to size
of table.

/* Global Variables */
char entries[NUM_ENTRIES][LINE_LENGTH];
pthread_mutex_t mutexsum;                        // mutex for char_counts

/* Function prototypes */
void max_substring(int myID);
char *strrev(char *str);
void read_file();

void main() {
        struct timeval t1, t2, t3, t4;
        double elapsedTime;
        int numSlots, myVersion = 3; // 1 = base, 2 = openmp, 3 = pthreads, 4 = mpi

        int i, rc;
        /* Create variables needed for pthreads */
        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(&t1, NULL);
        /* Read the file into the the list of entries */
        read_file();
        gettimeofday(&t2, NULL);

        gettimeofday(&t3, NULL);
        for (i = 0; i < NUM_THREADS; i++ ) {
          rc = pthread_create(&threads[i], &attr, max_substring, (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(&t4, NULL);

        elapsedTime = (t2.tv_sec - t1.tv_sec) * 1000.0; //sec to ms
        elapsedTime += (t2.tv_usec - t1.tv_usec) / 1000.0; // us to ms
        printf("Time to read file: %f\n", elapsedTime);

        elapsedTime = (t4.tv_sec - t3.tv_sec) * 1000.0; //sec to ms
        elapsedTime += (t4.tv_usec - t3.tv_usec) / 1000.0; // us to ms
        printf("Time to get max substrings: %f\n", elapsedTime);

        printf("DATA, %d, %s, %f\n", myVersion, getenv("SLURM_CPUS_ON_NODE"),  elapsedTime);

        pthread_mutex_destroy(&mutexsum);
        pthread_exit(NULL);
}

/* Read the file from wiki_dump.txt into the list of entries */
void read_file() {
        pthread_mutex_init(&mutexsum, NULL);
        FILE *fp;
        char str1[LINE_LENGTH];
        fp = fopen("/homes/dan/625/wiki_dump.txt", "r");
        // fp = fopen("test.txt", "r");

        /* If the file could not be found, return */
        if(fp == NULL) {
                perror("Failed: ");
                return;
        }

        /* Add each line of the file into entries */
        int i = 0;
        while(fgets(str1, LINE_LENGTH, fp) != NULL && i < NUM_ENTRIES){
                strcpy(entries[i], str1);
                i++;
        }

        fclose(fp);
}

/* Find and prints the biggest AND most common substring between 2 str1 and str2.
   Code inspired by https://www.geeksforgeeks.org/longest-common-substring/ */
void max_substring(int myID) {
        int startPos = myID * (NUM_ENTRIES / NUM_THREADS);
        int endPos = startPos + (NUM_ENTRIES / NUM_THREADS);

        char str1[LINE_LENGTH];
        char str2[LINE_LENGTH];
        int m, n, i;
        char biggest[LINE_LENGTH]; // The biggest/most common substring
        char temp[LINE_LENGTH];
        int row, col;
        int biggest_row, biggest_col; // Index of the LAST letter of the biggest substring we've found
        int max_len; // The length of the biggest substring we've found


                for(i = startPos; i < endPos; i++)
                {
                        strcpy(str1, entries[i]);
                        strcpy(str2, entries[i+1]);
                        m = strlen(str1);
                        n = strlen(str2);

                        /* Allocate memory for a table */
                        int (*table)[n] = malloc(sizeof(int[m + 1][n + 1]));

                        max_len = 0;
                        row =  0;
                        col = 0;
                        biggest_row = 0;
                        biggest_col = 0;

                        /* Populate the table */
                        for(row = 0; row < m; row++) {
                                for(col = 0; col < n; col++) {
                                        /* Initialize table[1..m][0] and table[0][1..n] to 1 */
                                        if(row == 0 || col == 0) {
                                                table[row][col] = 1;
                                        }
                                        /* If the letters of the two strings are the same, add 1 to the
left diagonal
                                           value and set it to the current position */
                                        else if(str1[row] == str2[col] && str1[row] != '\n'){
                                                table[row][col] = table[row-1][col-1] + 1;

                                                /* If the current position is bigger than the biggest
substring
                                                   we've found so far, update our variables so that we
can find
                                                   it later on. */
                                                if(table[row][col] > max_len) {
                                                        max_len = table[row][col];
                                                        biggest_row = row;
                                                        biggest_col = col;
                                                }
                                        }
                                        /* If the letters are not the same, the substring length is 0 */
                                        else {
                                                table[row][col] = 0;
                                        }
                                }
                        }

                        if(max_len == 0) {
                                printf("%d-%d:, %s\n", i, i+1, "No common substring.");
                        }
                        else{
                                memset(biggest, '\0', LINE_LENGTH);
                                /* Starting at the bottom right of the biggest substring in the table,
build biggest substring */
                                while(table[biggest_row][biggest_col] != 1) {
                                        /* Convert the letter to a string, since strcat requires a string
*/
                                        memset(temp, '\0', LINE_LENGTH);

                                        temp[0] = str1[biggest_row];

                                        /* Concatonate the string */
                                        strcat(biggest, temp);

                                        /* Move to the next letter (top left) in the backwards substring
*/
                                        biggest_row--;
                                        biggest_col--;
                                }
                                strcat(biggest, "\0");

                                /* Print and reverse the biggest substring */
                                printf("%d-%d: %s\n", i, i+1, strrev(biggest));
                        }

                        /* Free the table */
                        free(table);
                }
}

/* Reverse a string
   From https://stackoverflow.com/questions/8534274/is-the-strrev-function-not-available-in-linux */
char *strrev(char *str)
{
      char *p1, *p2;

      if (! str || ! *str)
            return str;
      for (p1 = str, p2 = str + strlen(str) - 1; p2 > p1; ++p1, --p2)
      {
            *p1 ^= *p2;
            *p2 ^= *p1;
            *p1 ^= *p2;
      }
      return str;
}
```

Appendix E
Pthreads sbatch file

```
#!/bin/bash -l
	//usr/bin/time -o
	/homes/rpwilliams96/CIS520_Proj4/src/3way-pthread/time/time-10k/time-$RANDO
	M.txt /homes/rpwilliams96/CIS520_Proj4/src/3way-pthread/output/pthreads-prod
```

Appendix F
Pthreads mass sbatch

```
#!/bin/bash

for i in 1 2 4 8 16
do
       for j in 1 2 3 4 5 6
       do
       sbatch --constraint=elves --time=04:00:00 --ntasks-per-node=$i --nodes=1
--mem-per-cpu=$((16/$i))G --partition=killable.q pthread-sbatch.sh
       done
done
```

Appendix G
MPI source code

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

/* Constants */
#define NUM_ENTRIES 1000000 // Should be 1000000
#define LINE_LENGTH 2003 //should be 2003, increasing to this size causes a segmentation fualt due to size
of table.
//#define NUM_THREADS 8
int NUM_THREADS;

/* Global Variables */
char entries[NUM_ENTRIES][LINE_LENGTH];

/* Results of the common subtring*/
char* results_array[NUM_ENTRIES];
char* local_results_array[NUM_ENTRIES];

/* Function prototypes */
void max_substring(void *rank);
char *strrev(char *str);
void read_file();
//void init_arrays();

void main(int argc, char* argv[]) {
        struct timeval t1, t2, t3, t4;
        double elapsedTime;
        int numSlots, myVersion = 4; // 1 = base, 2 = openmp, 3 = pthreads, 4 = mpi

        int i, rc;
        int numtasks, rank;
        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);

        NUM_THREADS = numtasks;
        printf("size = %d rank = %d\n", numtasks, rank);
        fflush(stdout);

        gettimeofday(&t1, NULL);
        if ( rank == 0 ) {
                //init_arrays();
                /* Read the file into the the list of entries */
                read_file();
        }
        gettimeofday(&t2, NULL);

        MPI_Bcast(entries, NUM_ENTRIES * LINE_LENGTH, MPI_CHAR, 0, MPI_COMM_WORLD);
        gettimeofday(&t3, NULL);
        /* Get the max substring of each line */
        max_substring(&rank);
        gettimeofday(&t4, NULL);
        MPI_Reduce(local_results_array, results_array, NUM_ENTRIES * LINE_LENGTH, MPI_CHAR, MPI_SUM, 0,
MPI_COMM_WORLD);

        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("Time to read file: %f\n", elapsedTime);

        elapsedTime = (t4.tv_sec - t3.tv_sec) * 1000.0; //sec to ms
        elapsedTime += (t4.tv_usec - t3.tv_usec) / 1000.0; // us to ms
        printf("Time to get max substrings: %f\n", elapsedTime);

        printf("DATA, %d, %s, %f\n", myVersion, getenv("SLURM_CPUS_ON_NODE"),  elapsedTime);

        MPI_Finalize();
        return 0;
}

/* Read the file from wiki_dump.txt into the list of entries */
void read_file() {
        FILE *fp;
        char str1[LINE_LENGTH];
        fp = fopen("/homes/dan/625/wiki_dump.txt", "r");
        // fp = fopen("test.txt", "r");

        /* If the file could not be found, return */
        if(fp == NULL) {
                perror("Failed: ");
                return;
        }

        /* Add each line of the file into entries */
        int i = 0;
        while(fgets(str1, LINE_LENGTH, fp) != NULL && i < NUM_ENTRIES){
                strcpy(entries[i], str1);
                i++;
        }

        fclose(fp);
}

/* Find and prints the biggest AND most common substring between 2 str1 and str2.
   Code inspired by https://www.geeksforgeeks.org/longest-common-substring/ */
void max_substring(void *rank) {
        int myID =  *((int*) rank);

        int startPos = myID * (NUM_ENTRIES / NUM_THREADS);
        int endPos = startPos + (NUM_ENTRIES / NUM_THREADS);

        char str1[LINE_LENGTH];
```

```
	char str2[LINE_LENGTH];
	int m, n, i;
	char biggest[LINE_LENGTH]; // The biggest/most common substring
	char temp[LINE_LENGTH];

	int row, col;
	int biggest_row, biggest_col; // Index of the LAST letter of the biggest substring we've found
	int max_len; // The length of the biggest substring we've found


		for(i = startPos; i < endPos; i++)
		{
			strcpy(str1, entries[i]);
			strcpy(str2, entries[i+1]);
			m = strlen(str1);
			n = strlen(str2);

			/* Allocate memory for a table */
			int (*table)[n] = malloc(sizeof(int[m + 1][n + 1]));

			max_len = 0;
			row =  0;
			col = 0;
			biggest_row = 0;
			biggest_col = 0;

			/* Populate the table */
			for(row = 0; row < m; row++) {
				for(col = 0; col < n; col++) {
					/* Initialize table[1..m][0] and table[0][1..n] to 1 */
					if(row == 0 || col == 0) {
						table[row][col] = 1;
					}
					/* If the letters of the two strings are the same, add 1 to the
left diagonal

					   value and set it to the current position */
					else if(str1[row] == str2[col] && str1[row] != '\n'){
						table[row][col] = table[row-1][col-1] + 1;

						/* If the current position is bigger than the biggest
substring

						   we've found so far, update our variables so that we
can find

						   it later on. */
						if(table[row][col] > max_len) {
							max_len = table[row][col];
							biggest_row = row;
							biggest_col = col;
						}
					}
					/* If the letters are not the same, the substring length is 0 */
					else {
						table[row][col] = 0;
					}
				}
			}
```

```
                        if(max_len == 0) {
                                printf("%d-%d:, %s\n", i, i+1, "No common substring.");
                        }
                        else{
                                memset(biggest, '\0', LINE_LENGTH);
                                /* Starting at the bottom right of the biggest substring in the table,
build biggest substring */
                                while(table[biggest_row][biggest_col] != 1) {
                                        /* Convert the letter to a string, since strcat requires a string
*/
                                        memset(temp, '\0', LINE_LENGTH);

                                        temp[0] = str1[biggest_row];

                                        /* Concatonate the string */
                                        strcat(biggest, temp);

                                        /* Move to the next letter (top left) in the backwards substring
*/
                                        biggest_row--;
                                        biggest_col--;
                                }
                                strcat(biggest, "\0");

                                /* Print and reverse the biggest substring */
                                printf("%d-%d: %s\n", i, i+1, strrev(biggest));
                                local_results_array[i] = strrev(biggest);
                        }

                        /* Free the table */
                        free(table);
                }
}

/* Reverse a string
   From https://stackoverflow.com/questions/8534274/is-the-strrev-function-not-available-in-linux */
char *strrev(char *str)
{
      char *p1, *p2;

      if (! str || ! *str)
            return str;
      for (p1 = str, p2 = str + strlen(str) - 1; p2 > p1; ++p1, --p2)
      {
            *p1 ^= *p2;
            *p2 ^= *p1;
            *p1 ^= *p2;
      }
      return str;
}
```

Appendix H

MPI sbatch file

```
#!/bin/bash -l

module load OpenMPI

mpirun //usr/bin/time -o
/homes/mspicer60/CIS520_Proj4/src/3way-mpi/time/time-10k/time-$RANDOM.txt
/homes/mspicer60/CIS520_Proj4/src/3way-mpi/output/output-10k/mpi-10k
```

Appendix I

MPI mass batch file

```
#!/bin/bash

for i in 1 2 4 8 16
do
	for j in 1 2 3
	do
	 sbatch --time=04:00:00 --ntasks-per-node=$i --nodes=1 --mem-per-cpu=$((16/$i))G
--partition=killable.q mpi-sbatch.sh
	done
done
```

Appendix J

Sample output first 100 lines

```
0-1: /title><text>\'\'\'aa
1-2: page> <title>aa
2-3: }</text> </page>
3-4: page> <title>a
4-5: n|foundation = 19
5-6: page> <title>abc_
6-7: page> <title>abc_
7-8: the [[australian broadcasting corporation]]
8-9: the [[australian broadcasting corporation]]
9-10: [[australian broadcasting corporation]]
10-11: /title><text>{{infobox
11-12: page> <title>ab
12-13: /title><text>\'\'\'ab
13-14: /title><text>\'\'\'a
14-15: page> <title>ac
15-16: page> <title>acc
16-17: n\n{{disambig}}</text> </page>
17-18: }</text> </page>
18-19: n\n{{disambiguation}}</text> </page>
19-20: /title><text>\'\'\'ac
20-21: page> <title>ac
21-22: page> <title>ac_
22-23: page> <title>ac_
23-24: /text> </page>
24-25: '\'\' may refer to:\n
25-26: '\'\' may refer to:\n\n
26-27: page> <title>ad
27-28: page> <title>ad
28-29: page> <title>a
29-30: n\n{{disambig}}</text> </page>
30-31: page> <title>afc
31-32: page> <title>af
32-33: page> <title>af
33-34: page> <title>a
34-35: /title><text>{{infobox
35-36: /title><text>{{
36-37: page> <title>a
37-38: page> <title>aid
38-39: page> <title>ai
39-40: [australian institute of
40-41: page> <title>a
41-42: n\n{{disambig}}</text> </page>
42-43: ]\n\n{{disambig}}</text> </page>
43-44: page> <title>aj
44-45: -stub}}</text> </page>
45-46: page> <title>ak
46-47: /text> </page>
47-48: page> <title>alar
48-49: page> <title>al
49-50: page> <title>alp
50-51: page> <title>a
51-52: page> <title>am
52-53: page> <title>am
```

```
53-54: page> <title>am
54-55: page> <title>am
55-56: }\n\n{{defaultsort:am
56-57: /title><text>{{unreferenced
57-58: lass=\"wikitable\"
58-59: page> <title>an
59-60: page> <title>a
60-61: page> <title>a
61-62: page> <title>ap
62-63: page> <title>ap
63-64: page> <title>ap
64-65: page> <title>ap
65-66: n\n==external links==\n*
66-67: page> <title>ap
67-68: /title><text>{{
68-69: {cite web|url=http://www.
69-70: page> <title>ar
70-71: page> <title>a
71-72: =\n{{reflist}}\n\n==
72-73: page> <title>asa_
73-74: page> <title>as
74-75: page> <title>as
75-76: page> <title>as
76-77: page> <title>as
77-78: page> <title>as
78-79: page> <title>as
79-80: merican society for
80-81: [[association fo
81-82: '\'\' is a [[france|french]] [[association football]]
82-83: page> <title>a
83-84: page> <title>at
84-85: page> <title>at
85-86: page> <title>at
86-87: page> <title>at
87-88: page> <title>a
88-89: {cite web|url=http://www.
89-90: page> <title>a
90-91: page> <title>a
91-92: ]\n\n{{disambig}}</text> </page>
92-93: }</text> </page>
93-94: /title><text>{{
94-95: {unreferenced|date=
95-96: page> <title>a_b
96-97: page> <title>a_be
97-98: page> <title>a_be
98-99: 2012}}\n{{infobox album
99-100: name        = a big
```