Cacey Ostic and Nick Petrilli
Parallel Processing
Project 3

Project Overview

For Project 3 we worked on a case where we have an input.txt file and a pattern.txt file. The input file has a certain number of rows and columns in it, each with the same amount of characters, creating a rectangular grid. The input file is traversed, counting and logging the occurrences of the top-left most spot where the pattern is matched. To parallelize our code, we use MPI to run our code using multiple threads, and then gather all of the coordinates at the end for rank 0 to write them to the output file.

readFile Function

Our readFile function is where our input and pattern files get read. The function has four parameters: the filename, a pointer to a two-dimensional array to hold the character contents of the file, and two integer pointers to store the number of rows and columns in the file. The file is read into the two-dimensional array to be used for pattern matching. If either of the files can't be opened, the function returns false which results in MPI_Finalize running.

checkForPattern Function

Our checkForPattern function is where the contents of both files are traversed to find the pattern in the input file. The function takes eight parameters: a pointer to the two-dimensional character array representing the input matrix, a pointer to the two-dimensional character array representing the pattern matrix, the numbers of rows and column for each matrix, and i and j variables for the row and column index where the pattern matching starts. Within a nested for loop, looping through the rows and columns of both two-dimensional arrays, we check that the characters in the input file match all of the characters in the pattern. If the pattern doesn't match, or we go out of bounds, our flag variable will be set false indicating the pattern was not found at that position. During this pattern matching process, we are ignoring any wildcard characters, indicated by '*', since they can match any character and be out of bounds. If a pattern is found, the function returns true for the main function to handle.

Rotate Function

Our rotate function is used to rotate the pattern matrix 90 degrees clockwise. The function takes three parameters: the pattern array to rotate, and pointers to the array's number of rows and columns. A new rotated array is constructed by traversing the original array by columns first, then by rows going backwards. Then, the pointers to the number of rows and columns are used to switch their values. This is because rotating an X by Y matrix results in a Y by X matrix. The function returned the rotated array.

Mirror Function

Our mirror function is used to mirror the pattern matrix along the y-axis. Similar to the rotate function, this function takes three parameters: the pattern array, and its number of rows and columns. The new mirrored array is constructed by traversing the original array normally by rows first, then backwards through the columns. However, since the pattern is only mirrored, the row and column numbers do not need to be switched. The function returns the mirrored array.

Main Function

At the beginning of program execution, MPI is initialized. Then, after checking to make sure both filename arguments are present, both the input and the pattern files are read into the program. Each MPI rank reads in its own copy of the input and pattern. Though this decision increases memory consumption of the program, it guarantees that each rank can check the entire input for pattern matching if need be.

The next step is to determine which portion of the input file each rank will check. Even though the rank has the entire input contents, it will only check a specific subsection of the file for the pattern. This split is determined by the total number of characters in the input file. Each rank is given an equal number of characters regardless of how many lines the characters span. If the number of ranks doesn't evenly divide the total number of characters, then some ranks will have one extra character to check.

A 1D coordinates array is also created for each rank to hold the coordinates of patterns it finds. Since the program does not know how many patterns will be found, it creates an array equal to the maximum number of patterns it could find. This maximum is also multiplied by 2 because each coordinate needs two positions in the array: one for rows, one for columns. We use a 1D array instead of 2D array to allow the ranks to send this information later in the program.

Each rank calculates its own start and end position. Rank 0 will additionally calculate the start and end for each other rank. This is used to calculate the displacements in the master array when Rank 0 receives the data from all other ranks. For example, if each rank may find up to 3 coordinate pairs, Rank 0 will use indices 0 to 5, Rank 1 will use indices 6 to 11, etc.

Now, each rank loops through its calculated characters from the input file to search for the pattern. For each character, the pattern could be checked a total of 8 times: one for each rotation (4) and one for each mirrored rotation (4). This part of the program uses the rotate and mirror functions described above to modify the pattern. Once a pattern is found, it is added to the coordinates list and the loop moves on to the next character.

After the main loop, all ranks use the MPI_Gatherv to send their data to Rank 0. The program uses MPI_Gatherv instead of MPI_Gather because each rank may send a slightly different amount of data. This requires the displacements calculated by Rank 0.

Once all threads have submitted their data, Rank 0 prints the contents of the array to a file. Since the array is likely larger than the amount of patterns found, there will be many empty spaces in the array. These are skipped over when printing.

After printing to the file, there are some additional calculations and reductions to determine the elapsed time for each major step of the program.

Test Results

Our test results contain the file size of the input file, the total number of occurrences found, the memory consumption of the program, and the runtime of the pattern counting within our program. We are only using this metric since this is the area of code that is parallelized. After gathering all occurrences, rank 0 performing file I/O operations of writing to the file is single threaded, and therefore won't get faster with parallelization. There are pictures included of each pattern, and either the whole input file (for input1.txt), or a truncated version of what we duplicated many times. Patterns 3 and 4 include wildcards to confirm that values outside of the input file are the starting point of where the pattern was found. Also, the input files of these test cases include rotated versions of the pattern to be counted.

CPU Model

12th Gen Intel(R) Core(TM) i7-1255U
Base Speed: 1.70 GHz
Cores: 10
Logical Processors: 12

input1.txt and pattern1.txt

Pattern:

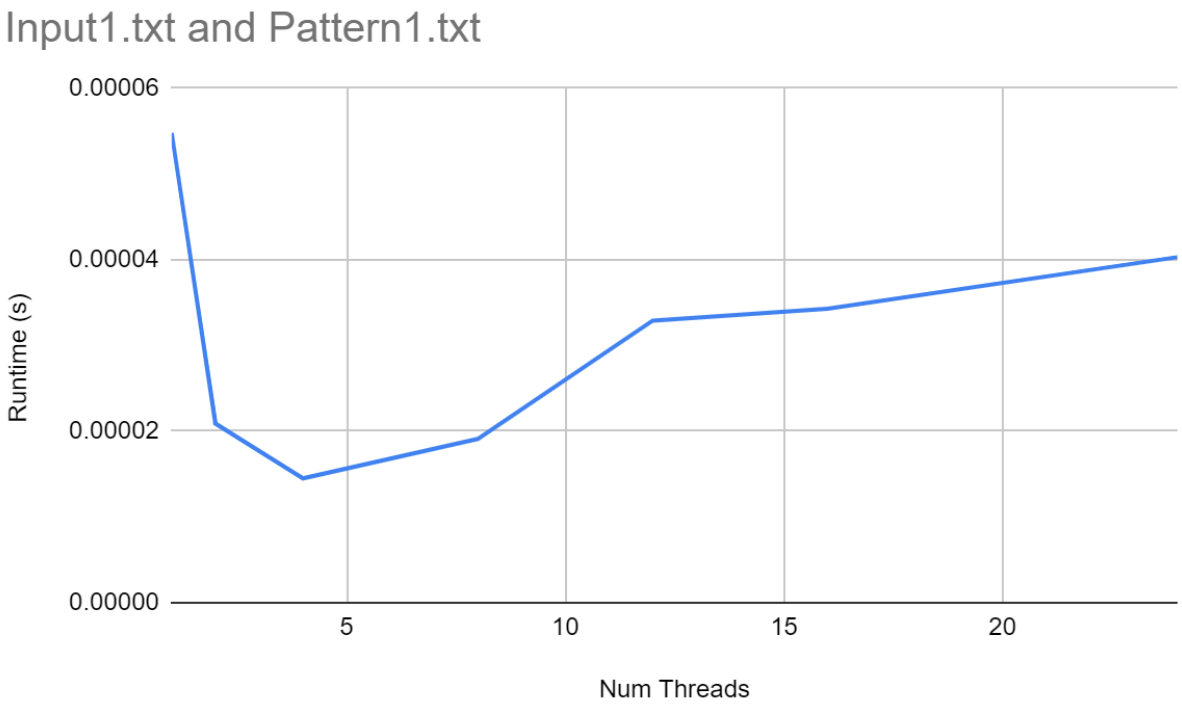
```
aa
```

Input:

```
aaaa
bbbb
abab
aabb
bbaa
```

Num Threads	Input File	Pattern File	File Size	Occurrences	Memory	Runtime (s)
1	input1.txt	pattern1.txt	1 KB	6	5.12 MB	0.0000548
2					5.14 MB	0.0000209
4					5.25 MB	0.0000145
8					5.26 MB	0.0000191
12					5.44 MB	0.0000329

16					5.52 MB	0.0000343
24					5.59 MB	0.0000403

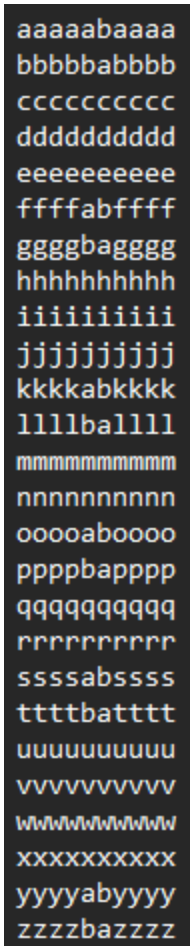


input2.txt and pattern2.txt

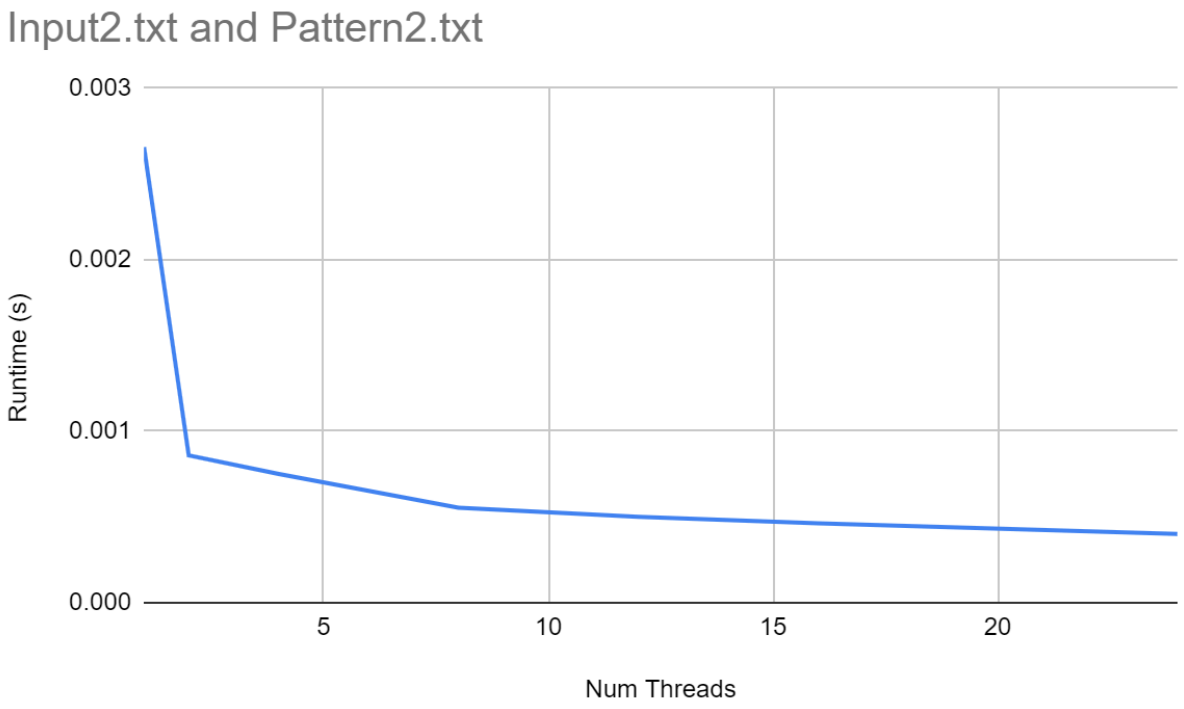
Pattern:

```
ab
ba
```

Input:



Num Threads	Input File	Pattern File	File Size	Occurrences	Memory	Runtime (s)
1	input2.txt	pattern2.txt	2 KB	47	5.13 MB	0.0026565
2					5.16 MB	0.0008588
4					5.26 MB	0.000751
8					5.33 MB	0.0005541
12					5.36 MB	0.0005003
16					5.39 MB	0.0004625
24					5.45 MB	0.0004009



input3.txt and pattern3.txt

Pattern:

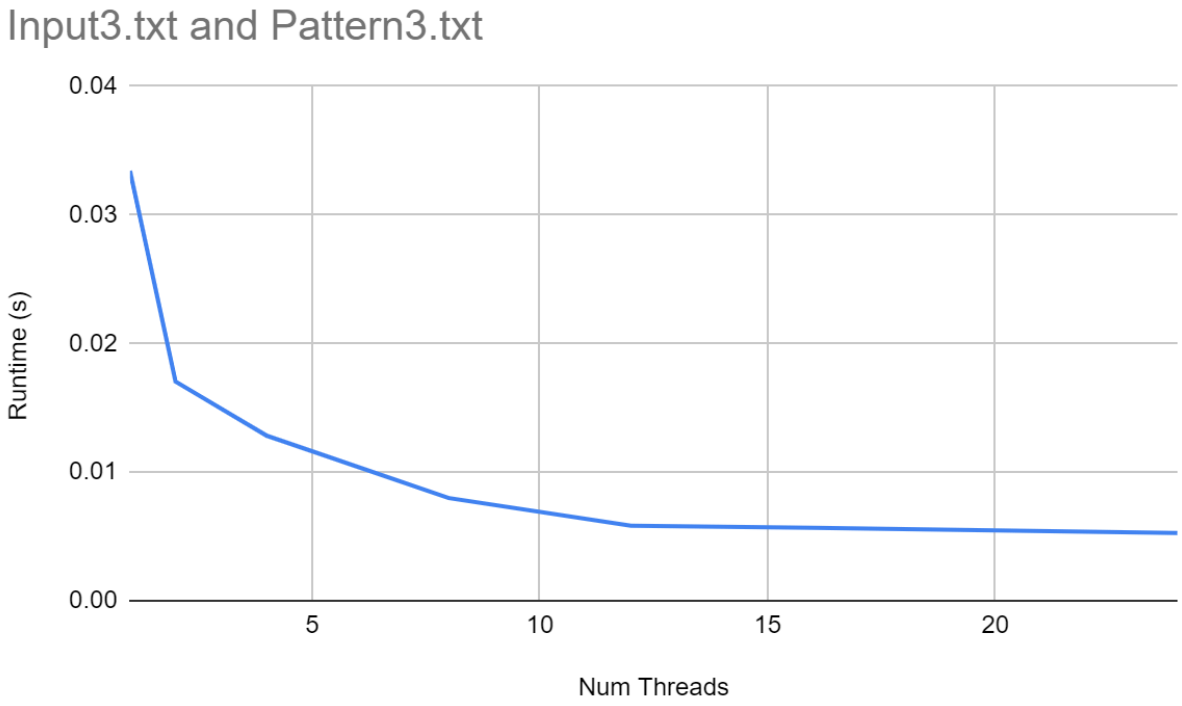
```
**ac
**bd
```

Input:

```
acooobaooo
bdooodcodb
ooooooooca
ooooocdooo
oooooabooo
acooobaooo
bdooodcodb
ooooooooca
ooooocdooo
oooooabooo
acooobaooo
bdooodcodb
ooooooooca
ooooocdooo
oooooabooo
```

Num Threads	Input File	Pattern File	File Size	Occurrences	Memory	Runtime (s)

1	input3.txt	pattern3.txt	25 KB	1243	5.13 MB	0.0334325
2					5.14 MB	0.0170464
4					5.20 MB	0.0128328
8					5.24 MB	0.0079777
12					5.31 MB	0.0058391
16					5.41 MB	0.0056707
24					5.45 MB	0.0052654

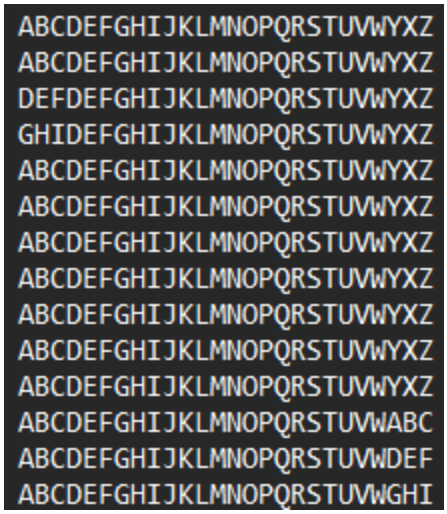


input4.txt and pattern4.txt

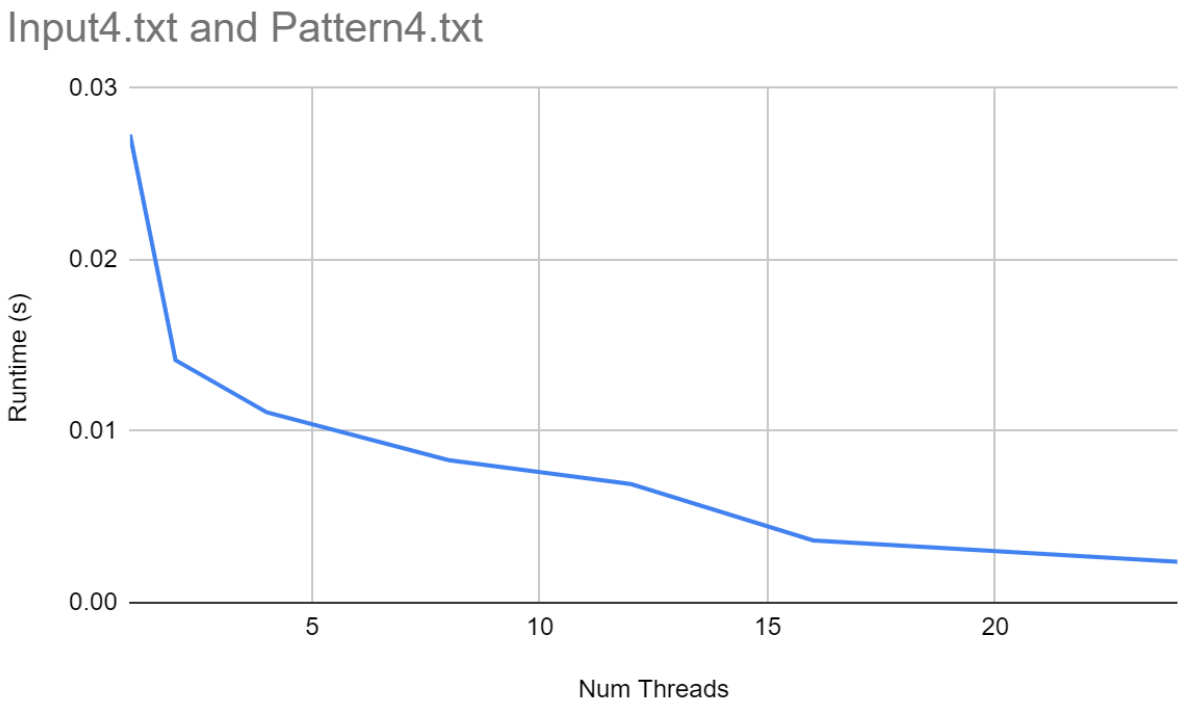
Pattern:

```
**ABC
**DEF
**GHI
```

Input:



Num Threads	Input File	Pattern File	File Size	Occurrences	Memory	Runtime (s)
1	input4.txt	pattern4.txt	825 KB	120,656	10.28 MB	0.0273066
2					10.40 MB	0.01415
4					10.48 MB	0.0110985
8					10.60 MB	0.0083144
12					10.76 MB	0.0069199
16					10.82 MB	0.0036303
24					10.94 MB	0.0023853



Conclusions

Our first test case, input1.txt and pattern1.txt was by far our smallest text file, with the input file only containing 5 lines, and finding 6 occurrences of the pattern. With this test case, the runtime decreased from 1-2-4 threads, and then after that point the runtime increased. This was probably due to the fact that the file was so small, and the overhead of the additional threads caused the runtime to increase, without enough work for all of the threads to work concurrently.

All of the other test cases, our runtime steadily decreased with the increase of threads. These files were bigger than our first input file, especially our last file: input4.txt. There were more than enough characters in these files for all threads to have adequate work and for us to actually see the benefits of parallelization.