Project Objective:

Read the wiki_dump file into memory and find the maximum value of ASCII character numeric values for all the characters in a line. The file is located at ~dan/625/wiki_dump.txt. The output should be a printed list of lines, in order, with the line number and maximum ASCII. This is to be done using threads, MPI, and OpenMP (with an optional implementation in CUDA). These solutions will be run on Beocat via the job scheduler using different parameters (number of cores, etc.). The results of these tests are listed below.

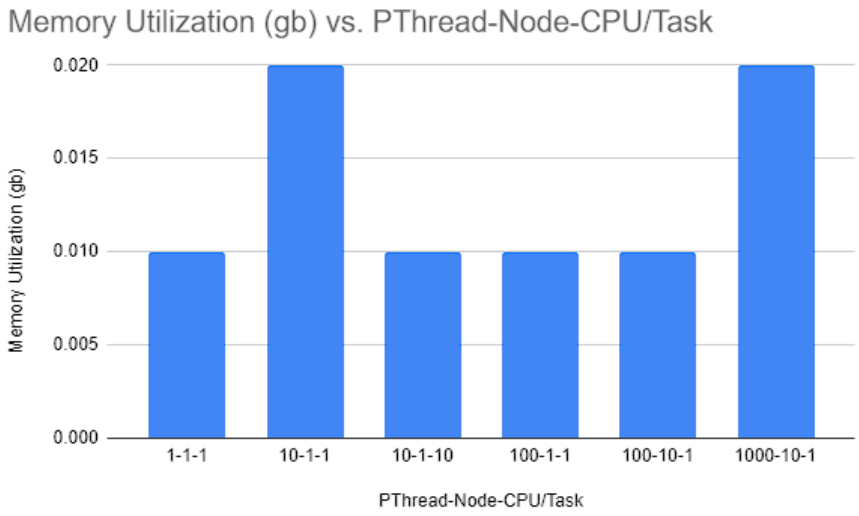
Pthreads:

Summary:

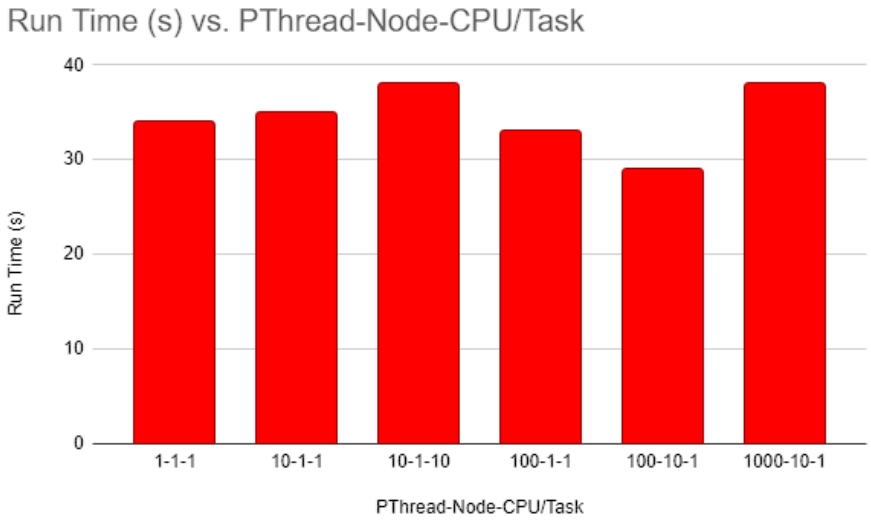
Our pthreads implementation works by making threads using pthreads, reading in the wiki dump file in chunks (lines), and then using a function to find the max ASCII value on a line. First, we attempt to open the file from "~dan/625/wiki_dump.txt" making sure to check for a NULL file. Then, memory is allocated for the lines in a chunk of data and then read into the chunks. Next, the pthreads are created with the proper arguments that will be running the "find_max_ascii" function. The results are then printed, the memory is freed for the lines, and the file is closed.

Analysis:

When running our pthread implementation the memory utilization is almost identical regardless of number of pthreads, nodes, or CPUs per task. Since the results only go to the hundredth of a decimal point, it's possible that the results are even more similar than they appear (varying between .01 gb and .02 gb) and just be rounded to make them look farther apart (.0149 would round to .01 whereas .0151 would round to .02). This indicates that the CPUs are not being used efficiently.



The run time does vary more, with the fastest time being with 100 pthreads, 10 nodes, and 1 CPU per task, and the slowest time being a tie between 10 pthreads 1 node and 10 CPUs per task, and 1000 pthreads 10 nodes and 1 cpu per task. Since the test with the highest number of CPUs per task also has the longest run time, this also indicates that the CPUs are not being used efficiently



There was a race condition for the pthread mutex object, however it was handled using a lock/unlock. The pthreads are not communicating, and there is no synchronization between processes, instead at the end the result is printed in the proper order in a print_results method.

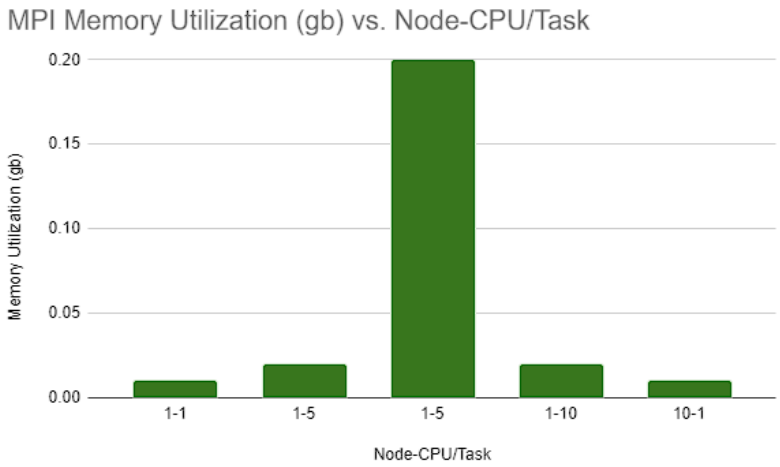
MPI:

Summary:

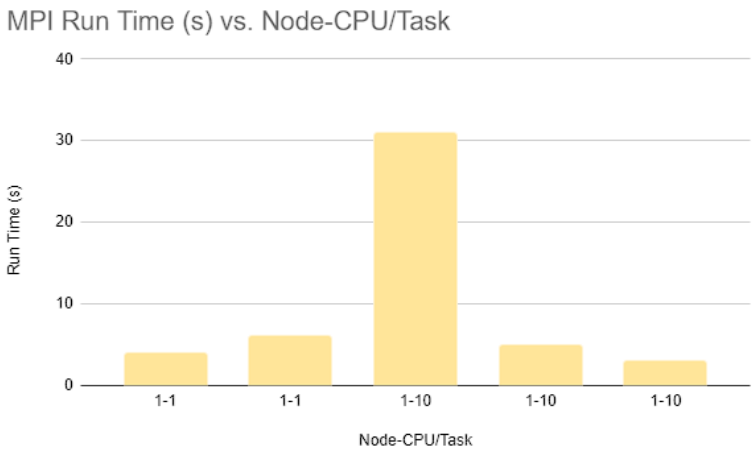
The MPI implementation is quite similar to the pthreads implementation. There is still the "find_max_ascii" function which calls the "max_ascii_value" function. The primary difference is that we start our main function by initializing the MPI environment. Next, the main function attempts to open the file and allocate memory for all the lines of text to process. Then, the calculations of max ASCII values happen and are printed until all lines are processed. Finally, the file is closed, and the MPI environment is finalized (terminated).

Analysis:

The memory utilization for the MPI implementation was similar to the p-threads implementation with most of the tests using .01-.02gb, however there was one test ran with the MPI implementation that used .2 gb. Since that test also ran in significantly more time compared to the rest of them, it likely just got caught up on something. Another test ran with the same settings had a result in the range we expected which indicates something was possibly off with the set up or BeoCat when that test was run.



The runtime of the MPI was generally much faster than the runtime of p-threads with most tests taking around 5 seconds to complete (with the one outlier being mentioned above).



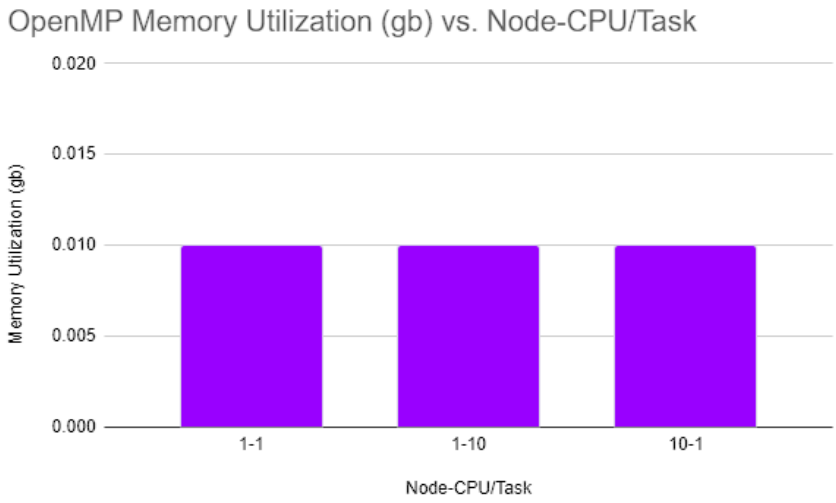
OpenMP:

Summary:

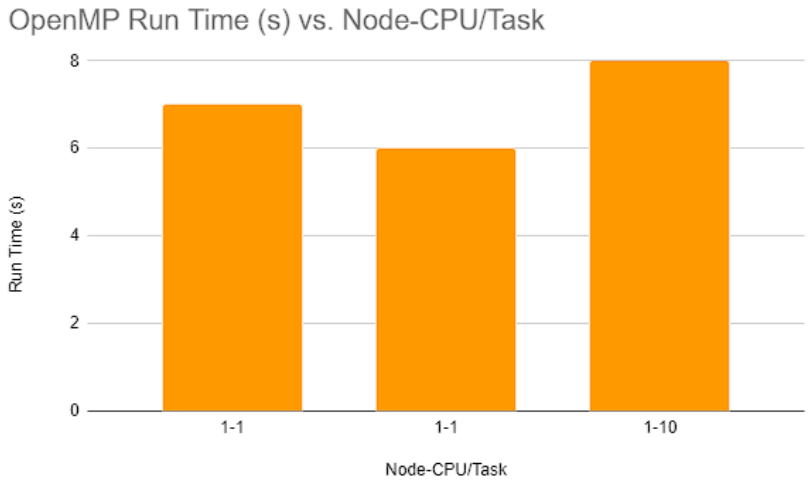
Once again, this implementation was very similar to the others. We still have the "find_max_ascii" function which calls the "max_ascii_value" function. The difference now is that we start our main function by initializing the number of threads desired. Next, the main function blocks off the functionality that each thread will run via openMP. In there, memory is allocated, and calculations are made to find the max ASCII value. Finally, to end this process, results are printed out, and the file is closed.

Analysis:

The memory utilization for openMP was also very similar to p-threads with all the tests using .01 gb. Since the results file rounds to the nearest tenth, this means that all the tests could've used less than .01 gb, and is therefore very efficient in regard to memory utilization.



The performance for openMP was significantly better than the performance of p-threads with a median run time of 7 seconds for openMP compared to a median run time of 34.5 seconds for p-threads. It performed slightly worse than MPI with MPI having a median run time of 5 seconds which is faster than the low run time of MPI (6 seconds).



There is no race condition in the openMP implementation, but the threads do communicate with a shared line number variable. There are no attempts to optimize this as a global line number integer variable is already very efficient.

Appendix

First 100 output lines for the 1 Node, 1 CPU/Task Test:

0: 125

1: 125

2: 125

3: 125

4: 125

5: 125

6: 125

7: 125

8: 125

9: 124

10: 125

11: 125

12: 125

13: 126

14: 125

15: 125

16: 125

17: 125

18: 125

19: 125

20: 125

21: 125

22: 125

23: 125

24: 125

25: 124

26: 125

27: 125

28: 125

29: 125

30: 125

31: 125

32: 125

33: 125

34: 125

35: 125

36: 125

37: 125

38: 125

39: 125

40: 125

41: 125

42: 125

43: 125

44: 125

45: 125

46: 125

47: 125

48: 125

49: 125

50: 125

51: 125

52: 122

53: 125

54: 125

55: 125

56: 125

57: 125

58: 125

59: 125

60: 125

61: 125

62: 125

63: 125

64: 125

65: 125

66: 125

67: 125

68: 125

69: 125

70: 124

71: 125

72: 125

73: 125

74: 125

75: 125

76: 125

77: 125

78: 125

79: 125

80: 125

81: 125

82: 125

83: 125

84: 125

85: 125

86: 125

87: 125

88: 125

89: 125

90: 125

91: 125

92: 125

93: 125

94: 125

95: 125

96: 125

97: 122

98: 125

99: 125

100: 12

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

#define NUM_THREADS 10 // Define the number of pthreads

typedef struct {
    char *line;
    int length;
    int line_number;
} LineData;

int max_ascii_value(char *line, int nchars) {
    int max = 0;
    for (int i = 0; i < nchars; i++) {
        if ((int)line[i] > max)
            max = (int)line[i];
    }
    return max;
}

void *find_max_ascii(void *args) {
    LineData *line_data = (LineData *)args;
    int max = max_ascii_value(line_data->line, line_data->length);
    printf("%d: %d\n", line_data->line_number, max);
    free(line_data->line);
    free(line_data);
    pthread_exit(NULL);
    return NULL;
}

int main() {
    FILE *file = fopen("/homes/dan/625/wiki_dump.txt", "r");
    if (file == NULL) {
        perror("Error opening file");
        return 1;
    }

    pthread_t threads[NUM_THREADS];
    char *line_buffer = NULL;
    size_t buffer_size = 0;
    int line_number = 0;

    while (getline(&line_buffer, &buffer_size, file) != -1) {
        LineData *line_data = (LineData *)malloc(sizeof(LineData));
        if (line_data == NULL) {
            perror("Memory allocation failed");
            fclose(file);
            return 1;
        }

        line_data->line = strdup(line_buffer);
        if (line_data->line == NULL) {
            perror("Memory allocation failed");
            fclose(file);
            free(line_data);
            return 1;
        }

        line_data->length = strlen(line_data->line);
        line_data->line_number = line_number;

        int thread_index = line_number % NUM_THREADS;
        if (pthread_create(&threads[thread_index], NULL, find_max_ascii, (void *)line_data) != 0)
```

```
            {
                perror("Failed to create thread");
                fclose(file);
                free(line_data->line);
                free(line_data);
                return 1;
            }

            pthread_join(threads[thread_index], NULL);
            line_number++;
        }

        fclose(file);
        if (line_buffer) free(line_buffer);
        return 0;
    }

```

```
#!/bin/bash
g++ -pthread -o3 proj4_pthread.c -o proj4_pthread
```

```
#!/bin/bash
./proj4_pthread $1 $2 $3 $4
```

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

typedef struct {
    char *line;
    int length;
    int line_number;
} LineData;

int max_ascii_value(char *line, int nchars) {
    int max = 0;
    for (int i = 0; i < nchars; i++) {
        if ((int)line[i] > max)
            max = (int)line[i];
    }
    return max;
}

void *find_max_ascii(void *args) {
    LineData *line_data = (LineData *)args;
    int max = max_ascii_value(line_data->line, line_data->length);
    printf("%d: %d\n", line_data->line_number, max);
    free(line_data->line);
    free(line_data);
    pthread_exit(NULL);
    return NULL;
}

int main() {
    int world_rank, world_size; //int for rank and number of processes

    MPI_Init(NULL, NULL); //initialize the MPI environment
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank); //get rank of processes
    MPI_Comm_size(MPI_COMM_WORLD, &world_size); //get size of processes

    FILE *file;
    file = fopen("/homes/dan/625/wiki_dump.txt", "r");
    if (file == NULL) {
        perror("Error opening file");
        MPI_Finalize();
            return 1;
    }

    char *line_buffer = NULL;
    size_t buffer_size = 0;
    int line_number = 0;

```

```c
    while (getline(&line_buffer, &buffer_size, file) != -1) {
        if (line_number % world_size == world_rank){
            LineData *line_data = (LineData *)malloc(sizeof(LineData));
            if (line_data == NULL) {
                perror("Memory allocation failed");
                fclose(file);
                MPI_Finalize();
                return 1;
            }

            line_data->line = strdup(line_buffer);
            if (line_data->line == NULL) {
                perror("Memory allocation failed");
                fclose(file);
                free(line_data);
                MPI_Finalize();
                return 1;
            }

            line_data->length = strlen(line_data->line);
            line_data->line_number = line_number;

            int max = max_ascii_value(line_data->line, line_data->length);
            printf("%d: %d\n", line_data->line_number, max);

            free(line_data->line);
            free(line_data);
        }
        line_number++;
    }

    if (line_buffer) free(line_buffer);
    if (world_rank == 0) fclose(file);

    MPI_Finalize();
    return 0;
}

```

```
#!/bin/bash
# Load the desired OpenMPI module
module load OpenMPI/4.1.4-GCC-11.3.0

# Compile the MPI program using mpicc
mpicc -O3 mpi.c -o proj4_mpi

```

```
#!/bin/bash
./proj4_mpi $1 $2 $3 $4

```

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

typedef struct {
    char *line;
    int length;
    int line_number;
} LineData;

int max_ascii_value(char *line, int nchars) {
    int max = 0;
    for (int i = 0; i < nchars; i++) {
        if ((int)line[i] > max)
            max = (int)line[i];
    }
    return max;
}

void find_max_ascii(LineData *line_data) {
    int max = max_ascii_value(line_data->line, line_data->length);
    printf("%d: %d\n", line_data->line_number, max);
    free(line_data->line);
    free(line_data);
}

int main() {
    FILE *file;
    file = fopen("/homes/dan/625/wiki_dump.txt", "r");
    if (file == NULL) {
        perror("Error opening file");
        return 1;
    }

    char *line_buffer = NULL;
    size_t buffer_size = 0;
    int line_number = 0;

    int num_threads = 4; // Change this to adjust the number of threads

    #pragma omp parallel num_threads(num_threads)
    {
        #pragma omp single
        {
            while (getline(&line_buffer, &buffer_size, file) != -1) {
                LineData *line_data = (LineData *)malloc(sizeof(LineData));
                if (line_data == NULL) {
                    perror("Memory allocation failed");
                    fclose(file);
                    return 1;
                }

                line_data->line = strdup(line_buffer);
                if (line_data->line == NULL) {
                    perror("Memory allocation failed");
                    fclose(file);
                    free(line_data);
                    return 1;
                }

                line_data->length = strlen(line_data->line);
                line_data->line_number = line_number;

                #pragma omp task
```

```
                {
                    find_max_ascii(line_data);
                }
                line_number++;
            }
        }
    }

    if (line_buffer) free(line_buffer);
    fclose(file);

    return 0;
}

```

```
#!/bin/bash
g++ -openmp -o3 proj4_openmp.c -o proj4_openmp

```

```
#!/bin/bash
./proj4_openmp $1 $2 $3 $4

```