# Parallel & Distributed Programming: A Performance Analysis

Jonah Bogusch, Ethan Khoury, Zachary Berard

May 1, 2023

**Abstract**

The paper presents an analysis of three parallel programming approaches - OpenMP, Pthreads, and MPI - applied to a problem of finding the maximum ASCII character numeric values for each line of a large text file on Beocat, a high-performance computing cluster. The authors used the same machine with consistent specs to conduct the analysis. The performance was measured in terms of the time taken to read the 1.7GB text file and print the maximum value for each line. The tests were run on 5 variations of core usage, with the number of cores doubling for each trial, up to a maximum of 16 cores. The results showed that OpenMP outperformed Pthreads and MPI for this problem, with the performance improving by 32.3 percent when 2 cores were used and plateauing at 2.31 seconds with 8 and 16 cores. The paper provides details on the general problem, solution, and testing methodology used for each programming approach. The findings suggest that parallelization can significantly improve the performance of computation-intensive tasks.

## 1 Introduction

In this project, we were to understand how parallel & distributed programming works. To accomplish this, we wrote a simple program with the three following approaches:

1. OpenMP
2. Pthreads
3. MPI

Note that this project was initially supposed to run on the "moles" machines, however, BEOCAT scheduling conflicts prevented this. Due to these conflicts, each of these approaches was run on BEOCAT with the "elves" group of machines. The "elves" have the following specifications:

| | |
|---|---|
| Nodes | Elves (1-56) |
| Processors 1 | 2x 8-Core Xeon E5-2690 |
| Ram | 64 GB |
| Hard Drive | 1x 250GB 7,200 RPM SATA |
| NICs | 4x Intel I350 |
| QDR Infiniband | Mellanox Technologies MT27500 Family [ConnectX-3] |

Using the same machine throughout our analysis shows that there is a genuine speedup or benefit to a certain approach or value. Without consistent specs on our machine, these numbers would be mostly meaningless.

### 1.1 General Problem

The problem our professor posed to us was as follows:

I am interested in the 'scorecard' that we can find in large text files. On Beocat there is a moderately large (wiki_dump.txt, 1.7GB) file containing approximately 1M Wikipedia entries, 1 entry per line. You can find the file in dan/625, along with other sample programs, on Beocat. Use this file - do not make your own copies of the data files.

Read the file into memory and find out the maximum value of ASCII character numeric values for all the characters in a line. So if the first line was ‘ab’ and the second was ‘bc’, val(a) = 97, b= 98, c=99, etc., (from https://en.wikipedia.org/wiki/ASCII#Printable_characters). then your output for lines 0 and 1 would be 98 and 99.
Print out a list of lines, in order, with the line number and minimum. E.g.

```
0: 97
1: 103
etc.
```

Your output should be identical for all versions of your code.

### 1.2 Our (General) Solution

For our solution, we went with a simple approach where we read in our file all at once, and then to compute our file we create threads to help iterate and compute.

```
main:
    lineArray <- file // This is sequential, we don't do this part in parallel


    i < numLines: // This loop uses threading
        results[i] <- getMaxValue(line)

    print(results)
```

While that is very simple, it’s just the general pattern we followed to accomplish our parallelization needs.

### 1.3 Testing

For the tests that we ran, we ran 10 trials over 5 different variations of core usage. Over these trials, we tested the time in seconds it took for our programs to read a 1.7GB text file and print the maximum value of ASCII character numeric values for all the characters in the line as mentioned above. By testing the time in seconds, we would get a good idea of how significant the performance improvements were over each trial. We also tracked the memory usage of each variation to observe the change in usage as we increased the number of cores. Our first trial utilized one core, and for each subsequent trial, the number of cores was doubled to increase the parallelism, topping out at 16 cores. We ran these tests for OpenMP, Pthreads, and MPI. There were three things these tests were designed to show: The first was how the performance was improved as more threads were added, and the second, how did each of the programming styles affect the performance, and lastly we wanted to compare the memory usage was the number of cores increased and compare each programming styles memory usage.

## 2 OpenMP

For our first implementation, we were to use OpenMP for our parallelization needs. This means that we are limited to one machine (our ”node”) and could only distribute the task across the cores on that singular machine. This software architecture was very close to the general solution and included helped methods like findMaxValue, parseLine, and GetProcessMemory. Parallelization using OpenMP was simpler due to pragma, as our code is comprised of a loop to read the files in and a loop using pragma to use multiple threads to find the maximum value.

## 3 Performance of OpenMP

OpenMP performance ranged from 7.0 seconds to 6.7 for one core, with an average of 6.78 This performance improved significantly by 32.3 percent when 2 cores were used, completing in an average of 4.59 seconds. At 4 cores, the improvements continued to steadily improve as the average of 2.94 was an improvement of 36 percent. 8 cores saw the improvement begin to slow, with an average of 2.39 which was an improvement of 19 percent. At 16 cores, the improvements plateaued, as the average of 8 and 16 remained virtually unchanged at 2.31. These results can be observed in Figure 1. Figure 1 shows all 10 trials conducted on each variation of cores using OpenMP, with the bottom row showing the average of all ten trials within that variation.

According to Figure 2, the memory increased as the number of cores increased, starting at approximately a 17million KBs at one core and then topping out at 182 million KBs with 16 cores. The trend of memory usage increased linearly, as even in the increase from 8 cores to 16 cores, it had a significant increase.

## 4 Pthreads

For the second programming style, our group used Pthreads. The implementation of Pthreads was more code-heavy, as the threading had to be done manually instead of using pragma. This resulted in the inclusion of another helper method in addition to the same helpers present in OpenMP. This method was called findMaxValueThread, and was used whenever a thread was assigned a task. Other changes included manually calling thread data commands when inside the for loop that calculated the maximum value. Overall, aside from the changes in how the code responsible for the threads was written, Pthreads followed the general solution.

## 5 Performance of Pthreads

Pthread's performance ranged from 6.7 seconds to 5.8 for one core, with an average of 6.48. This performance improved significantly by 31.5 percent when 2 cores were used, completing in an average of 4.44 seconds. At 4 cores, the improvement was relatively constant as the average of 3.1 was a slightly smaller improvement of 29.95. 8 cores saw another significant jump, with an average of 2.35 which was an improvement of 24.4. However, when 16 cores were used the improvements plateaued, as the average of 8 and 16 remained virtually unchanged at 2.25. These results can be observed in Figure 3. Figure 3 shows all 10 trials conducted on each variation of cores using Pthreads, with the bottom row showing the average of all ten trials within that variation.

According to Figure 4, the memory increased as the number of cores increased, starting at 1,777,640KB at one core and then topping out at 1,802,228KB with 4 cores. It was after 4 cores however that the memory usage plateaued, similarly to the performance data, although this plateau came with fewer cores used. For cores 8 and 16, the memory used remained constant at 1,802,228, which could indicate some sort of maximum value of memory usage on jobs utilizing the "elves" machines or a potential limitation in our program.

## 6 MPI

For the implementation of MPI, one thing to consider was that multiple computers can be utilized, whereas Pthreads and OpenMP can only use one at a time when doing parallelization which would limit us to only 20 cores on BEOCAT, creating a bottleneck. MPI however, is able to use 40 cores, which is able to increase the impact of parallelization. As for the software architecture, we begin by ensuring only the master process is responsible for reading the file and outputting them in order, so by checking if they rank to 0, the master process is responsible for both of these things ensuring that the order is consistent. Parallelization then occurs by default, and the result of the program follows the general solution above.

## 7 Performance of MPI

While the implementation of MPI works and gives us our expected results, we do not get any expected speedup due to parallelization. I'm not sure why this is the case, but it seems that we just did not implement this library correctly as we only tested on our local machine and not BEOCAT elves, as all the nodes were taken at the time of writing the code. If we look at Figure 5, we can see that there was nearly no benefit to adding more threads, even with 32 cores.

According to Figure 6, the memory usage was randomized. This is once again due to incorrect implementation, and no useful conclusions can be drawn from this data.

## 8 Conclusion

Our tests concluded that parallelization does indeed improve the performance of our C code, however, this improvement plateaus after 8 cores for all of the programming styles. While each programming style had slightly different averages, it seemed that the performance improved similarly across all programming styles (Other than MPI, which was not implemented correctly by us). All three programs saw a significant increase when improving from 1 to 2 cores, 2 to 4 cores, and 4 to 8 cores, but saw plateauing when going from 8 to 16. When it comes to comparing the three programming styles OpenMP was the least efficient, lagging behind Pthreads at every number of core usage. While Pthreads required more coding effort, in this specific instance it produced better performance than OpenMP. As for the memory, each program exhibited a different behavior, with the amount of memory used by Pthreads rapidly increasing as cores increased but then hitting an eventual plateau. The plateau, however, occurred sooner and was exactly the same number of Kilobytes, indicating some sort of maximum for our program, as this maximum did not appear in any other programming style. As for the OpenMP performance, it used far more memory than Pthreads, with a linearly increasing amount of memory topping out at 182 million KB. This seemed to indicate no real maximum for memory usage as was a difference between Pthreads and OpenMP. Finally, the performance of MPI was seemingly random with no discernible trend line, this is due to the fact that it was not implemented correctly. Overall, it seems that as the number of cores increases, the overall efficiency and memory usage increase.

# 9 Figures

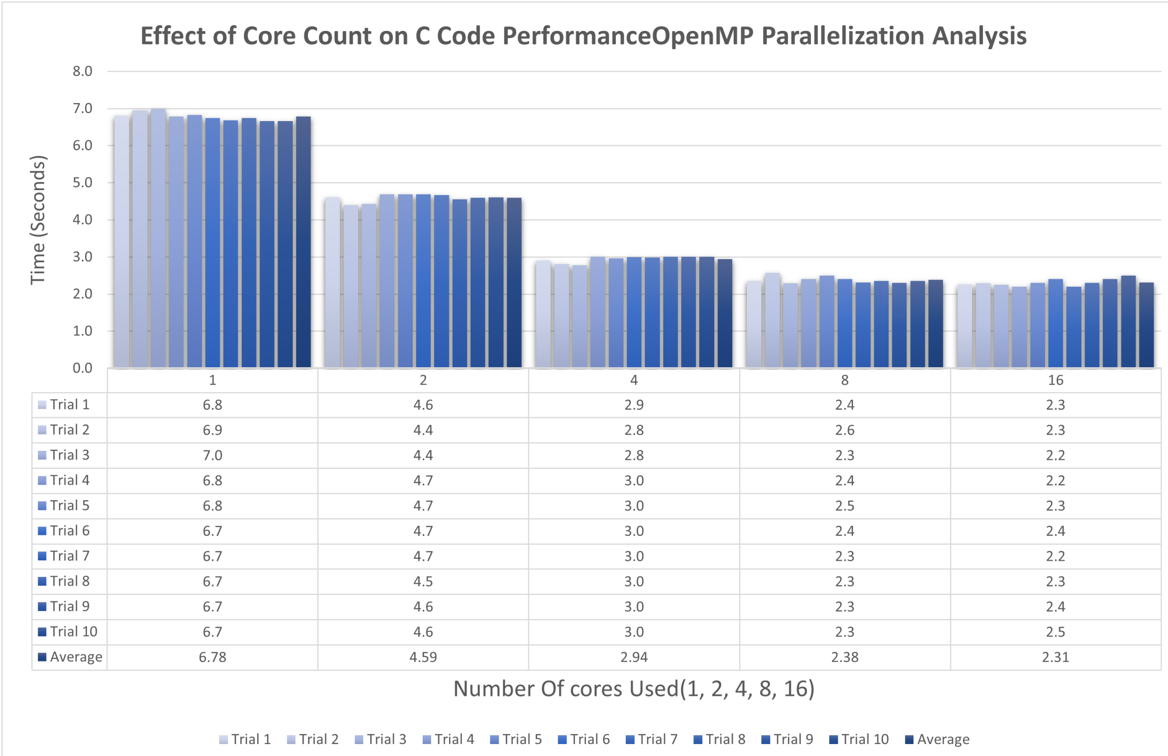



Figure 1: The bar graph displays the number of seconds it took for the program to run, with each bar representing a separate trial. The clusters of bars represent trials where a certain number of cores are being utilized, ranging from 1 to 16 cores. As seen in the graph, the program's performance improves as the number of cores increases, with the greatest improvement observed when moving from 1 to 2 cores, and not much change when using anything past 8 cores. These results demonstrate the effectiveness of OpenMP parallelization in speeding up the execution of our program.

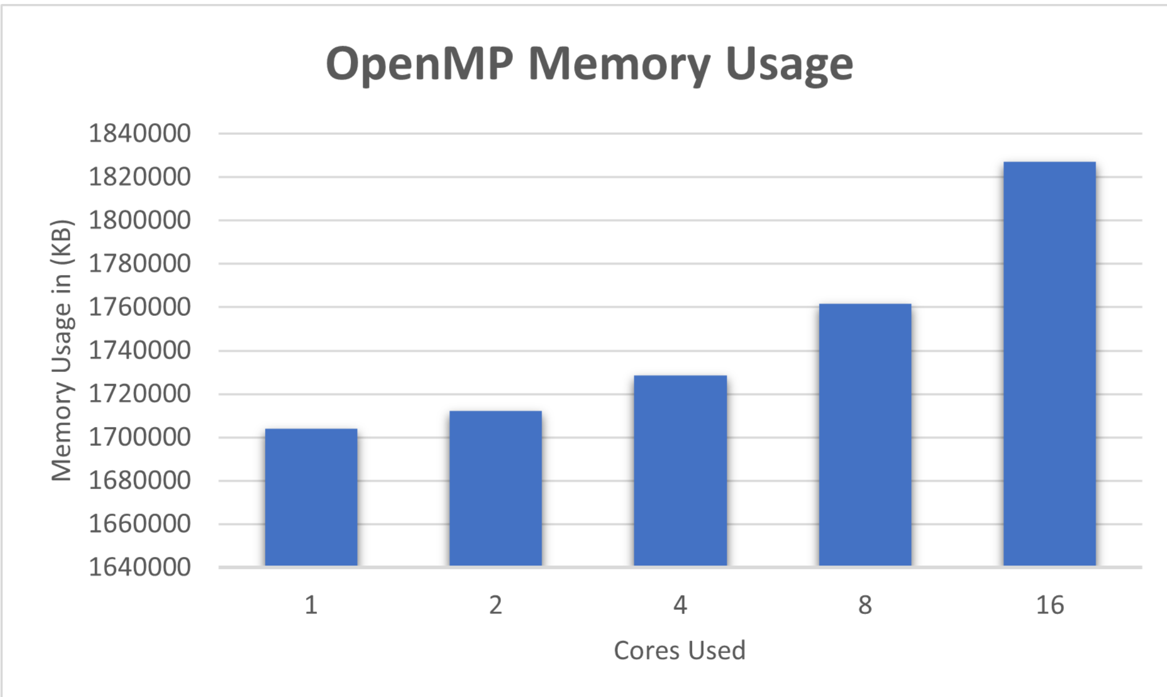
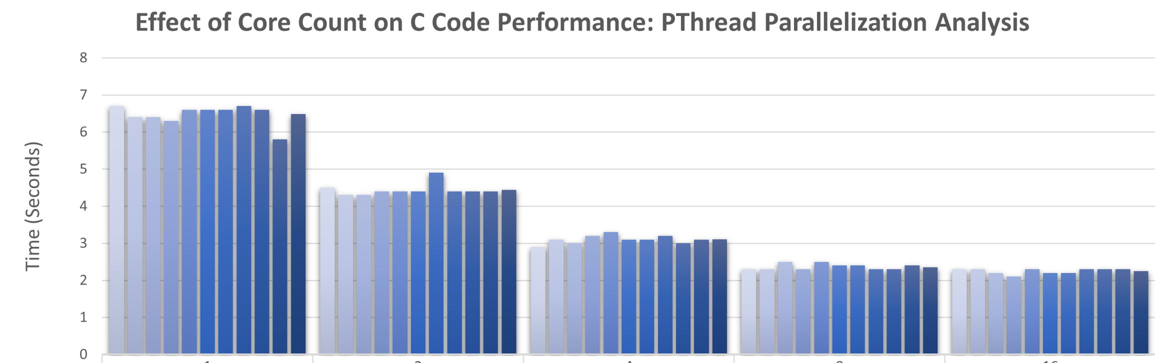



Figure 2: Memory Usage by Core Count: A Bar Graph Showing Memory in KB Used by Different Core Counts




| | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| Trial 1 | 6.7 | 4.5 | 2.9 | 2.3 | 2.3 |
| Trial 2 | 6.4 | 4.3 | 3.1 | 2.3 | 2.3 |
| Trial 3 | 6.4 | 4.3 | 3 | 2.5 | 2.2 |
| Trial 4 | 6.3 | 4.4 | 3.2 | 2.3 | 2.1 |
| Trial 5 | 6.6 | 4.4 | 3.3 | 2.5 | 2.3 |
| Trial 6 | 6.6 | 4.4 | 3.1 | 2.4 | 2.2 |
| Trial 7 | 6.6 | 4.9 | 3.1 | 2.4 | 2.2 |
| Trial 8 | 6.7 | 4.4 | 3.2 | 2.3 | 2.3 |
| Trial 9 | 6.6 | 4.4 | 3 | 2.3 | 2.3 |
| Trial 10 | 5.8 | 4.4 | 3.1 | 2.4 | 2.3 |
| Average | 6.48 | 4.44 | 3.11 | 2.35 | 2.25 |

Number Of cores Used(1, 2, 4, 8, 16)

Trial 1 Trial 2 Trial 3 Trial 4 Trial 5 Trial 6 Trial 7 Trial 8 Trial 9 Trial 10 Average

Figure 3: The bar graph displays the number of seconds it took for the program to run, with each bar representing a separate trial. The clusters of bars represent trials where a certain number of cores are being utilized, ranging from 1 to 16 cores. As seen in the graph, the program's performance improves as the number of cores increases, with the greatest improvement observed when moving from 1 to 2 cores, and not much change when using anything past 8 cores. These results demonstrate the effectiveness of Pthread parallelization in speeding up the execution of our program.

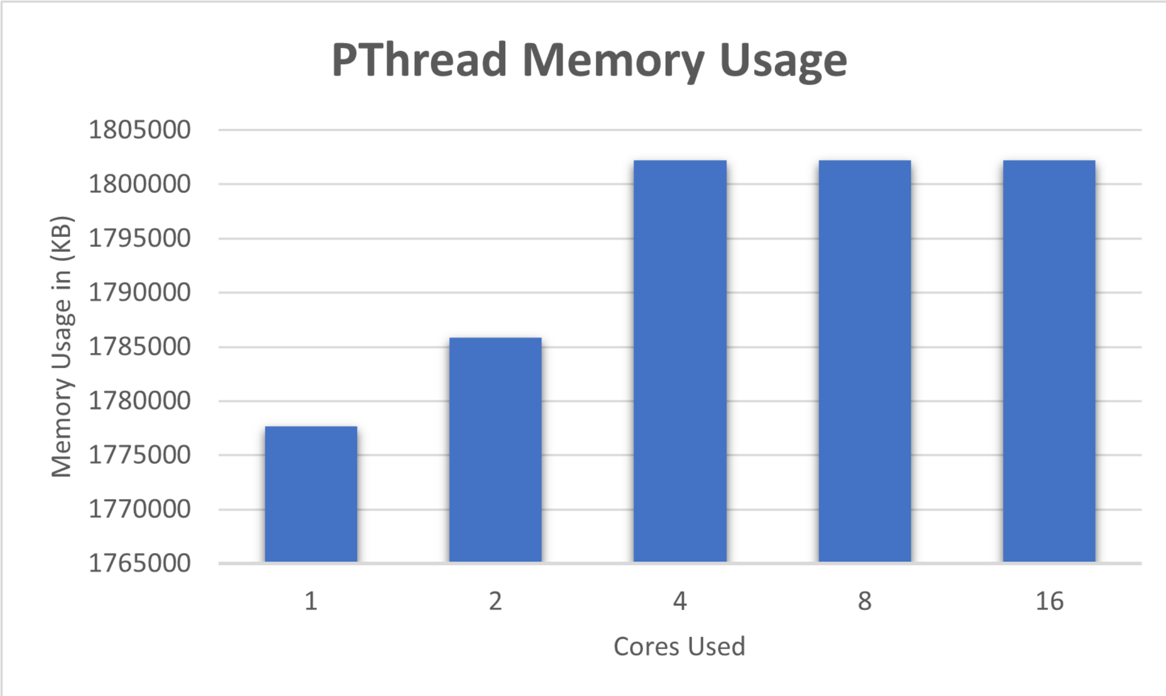
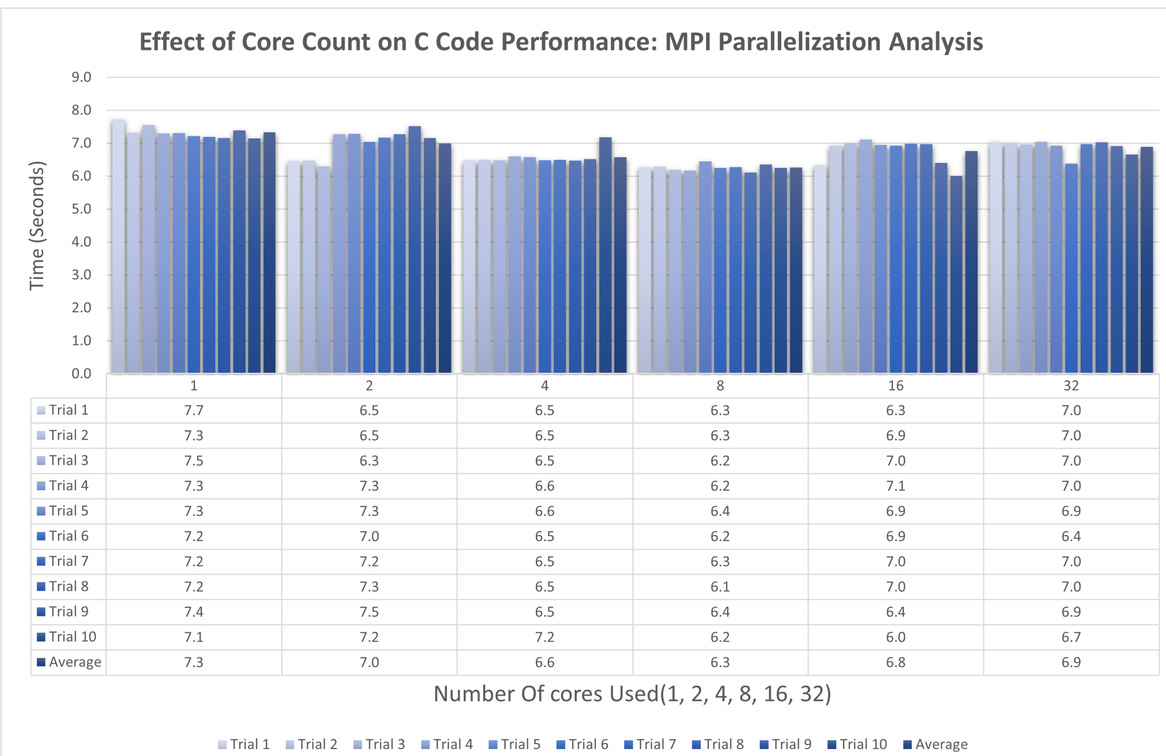



Figure 4: Memory Usage by Core Count: A Bar Graph Showing Memory in KB Used by Different Core Counts




| | 1 | 2 | 4 | 8 | 16 | 32 |
|---|---|---|---|---|---|---|
| Trial 1 | 7.7 | 6.5 | 6.5 | 6.3 | 6.3 | 7.0 |
| Trial 2 | 7.3 | 6.5 | 6.5 | 6.3 | 6.9 | 7.0 |
| Trial 3 | 7.5 | 6.3 | 6.5 | 6.2 | 7.0 | 7.0 |
| Trial 4 | 7.3 | 7.3 | 6.6 | 6.2 | 7.1 | 7.0 |
| Trial 5 | 7.3 | 7.3 | 6.6 | 6.4 | 6.9 | 6.9 |
| Trial 6 | 7.2 | 7.0 | 6.5 | 6.2 | 6.9 | 6.4 |
| Trial 7 | 7.2 | 7.2 | 6.5 | 6.3 | 7.0 | 7.0 |
| Trial 8 | 7.2 | 7.3 | 6.5 | 6.1 | 7.0 | 7.0 |
| Trial 9 | 7.4 | 7.5 | 6.5 | 6.4 | 6.4 | 6.9 |
| Trial 10 | 7.1 | 7.2 | 7.2 | 6.2 | 6.0 | 6.7 |
| Average | 7.3 | 7.0 | 6.6 | 6.3 | 6.8 | 6.9 |



Figure 5: The bar graph displays the number of seconds it took for the program to run, with each bar representing a separate trial. The clusters of bars represent trials where a certain number of cores are being utilized, ranging from 1 to 32 cores. Unlike the others, we observed that the number of cores did not have much of a direct effect on the run time of the code with our MPI implementation.

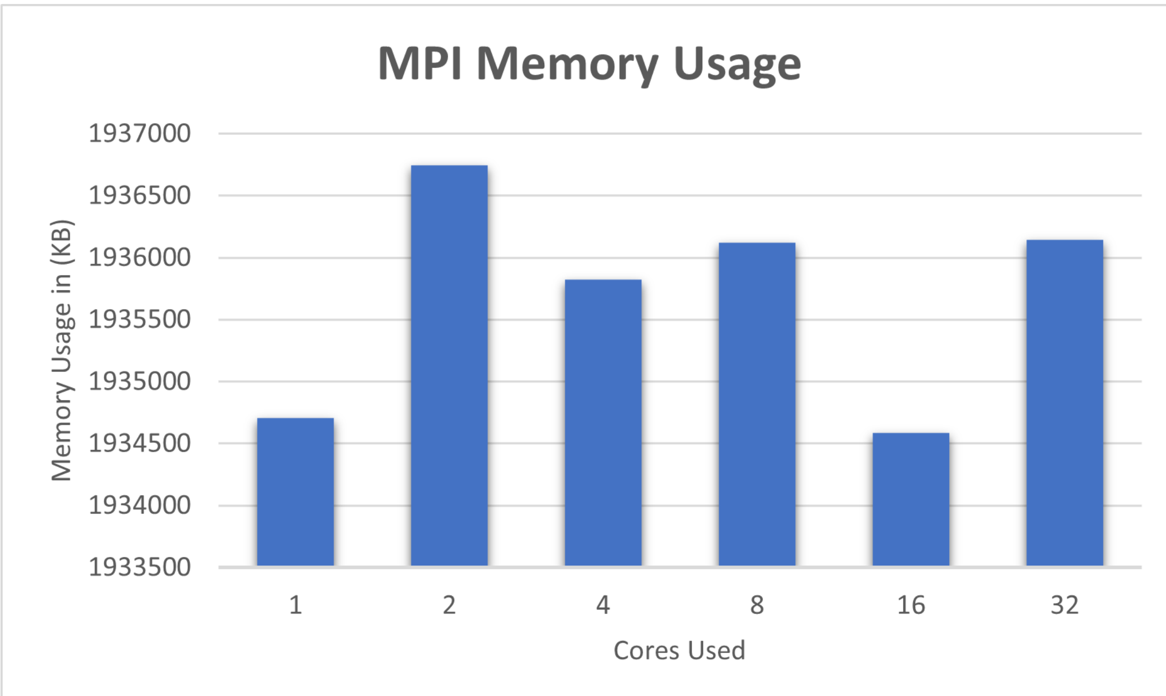



Figure 6: Memory Usage by Core Count: A Bar Graph Showing Memory in KB Used by Different Core Counts

## 10 Code

```c
#include <omp.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/time.h>
#include <stdint.h>

#include "sys/types.h"
#include "sys/sysinfo.h"

typedef struct {
    uint32_t virtualMem;
    uint32_t physicalMem;
} processMem_t;

int parseLine(char *line) {
    int i = strlen(line);
    const char *p = line;
    while (*p < '0' || *p > '9') p++;
    line[i - 3] = '\0';
    i = atoi(p);
    return i;
}

void GetProcessMemory(processMem_t* processMem) {
    FILE *file = fopen("/proc/self/status", "r");
    char line[128];

    while (fgets(line, 128, file) != NULL) {
        if (strncmp(line, "VmSize:", 7) == 0) {
            processMem->virtualMem = parseLine(line);
        }

        if (strncmp(line, "VmRSS:", 6) == 0) {
            processMem->physicalMem = parseLine(line);
        }
    }
    fclose(file);
}

int findMaxValue(char* line, int nchars) {
    int i;
    int maxVal = 0;

    for (i = 0; i < nchars; i++) {
        int asciiVal = (int)line[i];
        if (asciiVal > maxVal) {
            maxVal = asciiVal;
        }
    }

    return maxVal;
}

int main() {
    //Analysis variables

    // Time
    struct timeval t1, t2;
```

```c
    double elapsedTime;
    int myVersion = 1;

    // Memory
    //processMem_t afterRead; //Not sure if I want to keep, peak is after comp
    processMem_t afterComp;



    //Program variables
    const int maxlines = 1000000;
    //const int chunk_size = 1000;
    int nlines = 0;
    int i, nchars;
    FILE *fd;
    int *results = (int*)malloc(maxlines * sizeof(int));

    //Analysis setup
    gettimeofday(&t1, NULL);

    //Program start
    fd = fopen("/homes/dan/625/wiki_dump.txt", "r");

    // Read the entire file into memory
    char **lines = (char**)malloc(maxlines * sizeof(char*));
    char *line = NULL;
    size_t len = 0;
    ssize_t read;

    for (i = 0; i < maxlines; i++) {
        read = getline(&line, &len, fd);
        if (read == -1) {
            break;
        }
        lines[i] = (char *)malloc((read + 1) * sizeof(char));
        strncpy(lines[i], line, read + 1);
        nlines++;
    }
    free(line);
    fclose(fd);

     //GetProcessMemory(&afterRead);

    #pragma omp parallel for private(i, nchars) schedule(static)
    for (i = 0; i < nlines; i++) {
        char line[2001];
        strncpy(line, lines[i], 2001);
        nchars = strlen(line);
        results[i] = findMaxValue(line, nchars);
    }

    GetProcessMemory(&afterComp);

    for (i = 0; i < nlines; i++) {
        printf("%d: %d\n", i, results[i]);
    }

    // Free memory
    for (i = 0; i < maxlines; i++) {
        free(lines[i]);
    }
    free(lines);
```

```
    free(results);
    //End program, start analysis again

    gettimeofday(&t2, 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("DATATIME(ms), %d, %s, %f\n", myVersion, getenv("SLURM_NTASKS"),
        elapsedTime);
    //printf("DATAREADMEM(vMemKB)(pMemKB), %u, %u\n", afterRead.virtualMem,
        afterRead.physicalMem);
    printf("DATACOMPMEM(vMemKB)(pMemKB), %s, %u, %u\n", getenv("SLURM_NTASKS")
        , afterComp.virtualMem, afterComp.physicalMem);

    return 0;
}
```

Listing 1: OpenMP

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/time.h>
#include <stdint.h>
#include <pthread.h>

#include "sys/types.h"
#include "sys/sysinfo.h"

#define NUM_THREADS 4

typedef struct {
    uint32_t virtualMem;
    uint32_t physicalMem;
} processMem_t;

typedef struct {
    int start;
    int end;
    int *results;
    char **lines;
} thread_data_t;

int parseLine(char *line) {
    int i = strlen(line);
    const char *p = line;
    while (*p < '0' || *p > '9') p++;
    line[i - 3] = '\0';
    i = atoi(p);
    return i;
}

void GetProcessMemory(processMem_t* processMem) {
    FILE *file = fopen("/proc/self/status", "r");
    char line[128];

    while (fgets(line, 128, file) != NULL) {
        if (strncmp(line, "VmSize:", 7) == 0) {
            processMem->virtualMem = parseLine(line);
        }

        if (strncmp(line, "VmRSS:", 6) == 0) {
```

```
            processMem->physicalMem = parseLine(line);
        }
    }
    fclose(file);
}

int findMaxValue(char* line, int nchars) {
    int i;
    int maxVal = 0;

    for (i = 0; i < nchars; i++) {
        int asciiVal = (int)line[i];
        if (asciiVal > maxVal) {
            maxVal = asciiVal;
        }
    }

    return maxVal;
}

void *findMaxValueThread(void *thread_data) {
    thread_data_t *data = (thread_data_t *)thread_data;
    int start = data->start;
    int end = data->end;
    int i, nchars;

    for (i = start; i < end; i++) {
        char line[2001];
        strncpy(line, data->lines[i], 2001);
        nchars = strlen(line);
        data->results[i] = findMaxValue(line, nchars);
    }

    pthread_exit(NULL);
}

int main() {
    //Analysis variables

    // Time
    struct timeval t1, t2;
    double elapsedTime;
    int myVersion = 1;

    // Memory
    //processMem_t afterRead; //Not sure if I want to keep, peak is after comp
    processMem_t afterComp;



    //Program variables
    const int maxlines = 1000000;
    //const int chunk_size = 1000;
    int nlines = 0;
    int i, nchars;
    FILE *fd;
    int *results = (int*)malloc(maxlines * sizeof(int));

    //Analysis setup
    gettimeofday(&t1, NULL);

    //Program start
```

```c
    fd = fopen("/homes/dan/625/wiki_dump.txt", "r");

    // Read the entire file into memory
    char **lines = (char**)malloc(maxlines * sizeof(char*));
    char *line = NULL;
    size_t len = 0;
    ssize_t read;

    for (i = 0; i < maxlines; i++) {
        read = getline(&line, &len, fd);
        if (read == -1) {
            break;
        }
        lines[i] = (char *)malloc((read + 1) * sizeof(char));
        strncpy(lines[i], line, read + 1);
        nlines++;
    }
    free(line);
    fclose(fd);

     //GetProcessMemory(&afterRead);
    char *slurm_ntasks_env = getenv("SLURM_NTASKS");
    int num_threads = NUM_THREADS;

    if (slurm_ntasks_env) {
        num_threads = atoi(slurm_ntasks_env);
    }

    int lines_per_thread = nlines / num_threads;
    pthread_t threads[num_threads];
    thread_data_t thread_data[num_threads];
    int rc;
    long t;

    for (t = 0; t < num_threads; t++) {
        thread_data[t].start = t * lines_per_thread;
        thread_data[t].end = (t == num_threads - 1) ? nlines : (t + 1) *
            lines_per_thread;
        thread_data[t].results = results;
        thread_data[t].lines = lines;
        rc = pthread_create(&threads[t], NULL, findMaxValueThread, (void *)&
            thread_data[t]);
        if (rc) {
            printf("ERROR: return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }

    for (t = 0; t < num_threads; t++) {
        pthread_join(threads[t], NULL);
    }

    GetProcessMemory(&afterComp);

    for (i = 0; i < nlines; i++) {
        printf("%d: %d\n", i, results[i]);
    }

    // Free memory
    for (i = 0; i < maxlines; i++) {
        free(lines[i]);
    }
```

```
    free(lines);
    free(results);
    //End program, start analysis again

    gettimeofday(&t2, 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("DATATIME(ms), %d, %s, %f\n", myVersion, getenv("SLURM_NTASKS"),
        elapsedTime);
    //printf("DATAREADMEM(vMemKB)(pMemKB), %u, %u\n", afterRead.virtualMem,
        afterRead.physicalMem);
    printf("DATACOMPMEM(vMemKB)(pMemKB), %s, %u, %u\n", getenv("SLURM_NTASKS")
        , afterComp.virtualMem, afterComp.physicalMem);

    return 0;
}
```

Listing 2: PThreads

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

#include "sys/types.h"
#include "sys/sysinfo.h"

typedef struct {
    uint32_t virtualMem;
    uint32_t physicalMem;
} processMem_t;

int parseLine(char *line) {
    int i = strlen(line);
    const char *p = line;
    while (*p < '0' || *p > '9') p++;
    line[i - 3] = '\0';
    i = atoi(p);
    return i;
}

void GetProcessMemory(processMem_t* processMem) {
    FILE *file = fopen("/proc/self/status", "r");
    char line[128];

    while (fgets(line, 128, file) != NULL) {
        if (strncmp(line, "VmSize:", 7) == 0) {
            processMem->virtualMem = parseLine(line);
        }

        if (strncmp(line, "VmRSS:", 6) == 0) {
            processMem->physicalMem = parseLine(line);
        }
    }
    fclose(file);
}

int findMaxValue(char* line, int nchars) {
    int i;
```

```
    int maxVal = 0;

    for (i = 0; i < nchars; i++) {
        int asciiVal = (int)line[i];
        if (asciiVal > maxVal) {
            maxVal = asciiVal;
        }
    }

    return maxVal;
}

int main(int argc, char* argv[]) {
    // MPI initialization
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    //Analysis variables

    // Time
    struct timeval t1, t2;
    double elapsedTime;
    int myVersion = 1;

    // Memory
    //processMem_t afterRead; //Not sure if I want to keep, peak is after comp
    processMem_t afterComp;



    //Program variables
    const int maxlines = 1000000;
    //const int chunk_size = 1000;
    int nlines = 0;
    int i, nchars;
    FILE *fd;
    int *results = (int*)malloc(maxlines * sizeof(int));

    //Analysis setup
    gettimeofday(&t1, NULL);

    char **lines = NULL;
    // Program start
    if (rank == 0) {
        fd = fopen("/homes/dan/625/wiki_dump.txt", "r");

        // Read the entire file into memory
        lines = (char**)malloc(maxlines * sizeof(char*));
        char *line = NULL;
        size_t len = 0;
        ssize_t read;

        for (i = 0; i < maxlines; i++) {
            read = getline(&line, &len, fd);
            if (read == -1) {
                break;
            }
            lines[i] = (char *)malloc((read + 1) * sizeof(char));
            strncpy(lines[i], line, read + 1);
            nlines++;
```

```c
        }
        free(line);
        fclose(fd);
    }

     //GetProcessMemory(&afterRead);

    // Broadcast the number of lines to all processes
    MPI_Bcast(&nlines, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // Broadcast the contents of lines to all processes
    if (rank == 0) {
        for (int i = 0; i < nlines; i++) {
            int line_length = strlen(lines[i]) + 1;
            MPI_Bcast(&line_length, 1, MPI_INT, 0, MPI_COMM_WORLD);
            MPI_Bcast(lines[i], line_length, MPI_CHAR, 0, MPI_COMM_WORLD);
        }
    } else {
        for (int i = 0; i < nlines; i++) {
            int line_length;
            MPI_Bcast(&line_length, 1, MPI_INT, 0, MPI_COMM_WORLD);
            lines[i] = (char *)malloc(line_length * sizeof(char));
            MPI_Bcast(lines[i], line_length, MPI_CHAR, 0, MPI_COMM_WORLD);
        }
    }
    // Calculate the workload distribution
    int local_nlines = nlines / size;
    int start_line = rank * local_nlines;
    int end_line = (rank == size - 1) ? nlines : (rank + 1) * local_nlines;

    // Process the lines
    for (i = start_line; i < end_line; i++) {
        char line[2001];
        strncpy(line, lines[i], 2001);
        nchars = strlen(line);
        results[i] = findMaxValue(line, nchars);
    }

    // Print the results only on the master process
    if (rank == 0) {
        GetProcessMemory(&afterComp);
        for (i = 0; i < nlines; i++) {
            printf("%d: %d\n", i, results[i]);
        }
    }

    // Free memory
    for (i = 0; i < maxlines; i++) {
        free(lines[i]);
    }
    free(lines);
    free(results);
    //End program, start analysis again

    gettimeofday(&t2, 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("DATATIME(ms), %d, %s, %s, %f\n", myVersion, getenv("SLURM_NTASKS")
        , getenv("SLURM_NNODES"), elapsedTime);
    //printf("DATAREADMEM(vMemKB)(pMemKB), %u, %u\n", afterRead.virtualMem,
        afterRead.physicalMem);
```

```
    printf("DATACOMPMEM(vMemKB)(pMemKB), %s, %s, %u, %u\n", getenv("
        SLURM_NTASKS"), getenv("SLURM_NNODES"), afterComp.virtualMem,
        afterComp.physicalMem);

    MPI_Finalize();

    return 0;
}
```

Listing 3: MPI

## 11 Scripts

This script runs our executable on BEOCAT as a job. We used this for testing and quickly running our jobs with different settings.

```
#!/bin/bash -l
#SBATCH --mem=4G
#SBATCH --time=03:00:00
#SBATCH --constraint=moles
#SBATCH --job-name=openMP
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=4

/homes/jonahmbog/cis520/Project4/3way-openmp/openmp
```

Listing 4: run.sh

This script is for the mass_sbatch.sh script, which will queue up multiple jobs.

```
#!/bin/bash -l
##$ -l h_rt=0:00:30              # ask for 1 minute runtime

/homes/jonahmbog/cis520/Project4/3way-openmp/openmp  #change to match the path
     to your code
```

Listing 5: openmp_sbatch.sh

This script queues up multiple jobs with varying core sizes, repeating 10 times to ensure we have lots of data to analyze. This script was primarily used when we were ready to get our data to analyze performance of different threads.

```
#!/bin/bash

for j in {1..10}
do
    for i in 1 2 4 8 16
    do
             sbatch --mem=4G --constraint=elves --ntasks-per-node=$i --nodes=1
                 openmp_sbatch.sh
    done
done
```

Listing 6: mass_sbatch.sh

## 12 Output

```
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
```

Listing 7: output