Project 4: Parallel ASCII Analysis

Group 15: Daniel Chang, Jiwoo Jung, Moises Villegas

1. Introduction

This document analyzes our implementations for Project 4, which processes a 1.7GB wiki_dump.txt file containing 1M Wikipedia entries to find the maximum ASCII value per line. For our "One Program, Three Ways" assignment, we implemented this solution using pthreads, MPI, and OpenMP, each offering different approaches to parallelization. This document discusses the software architecture of each implementation and provides a comprehensive performance analysis comparing their effectiveness.

2. Software Architecture

2.1 Pthread Implementation

Our pthread implementation divides the workload among multiple threads, with each thread processing a designated portion of the file. The program follows these main steps:

1. Count the total number of lines in the file
2. Divide lines evenly among threads
3. Each thread processes its assigned lines independently
4. Collect and combine results from all threads
5. Output the maximum ASCII value for each line

Each thread opens the input file, seeks to its starting position, processes assigned lines, finds maximum ASCII values, stores results locally, and returns them to the main thread. We use a mutex to serialize file access while allowing computation to occur in parallel, preventing race conditions and performance degradation:

```c
pthread_mutex_t file_mutex = PTHREAD_MUTEX_INITIALIZER;
```

The key data structure for assigning work to threads is:

```c
typedef struct {
    int start;          // Starting line index
    int end;            // Ending line index (exclusive)
    char *filename;     // Input file path
} ThreadData;
```

2.2 MPI Implementation

Our MPI implementation divides the workload among multiple processes, with each process independently processing a designated portion of the file. The program follows these main steps:

1. Count the total number of lines in the file (done by rank 0)
2. Broadcast the total line count to all processes
3. Divide lines evenly among processes, with extra lines distributed among the lower-ranked processes
4. Each process processes its assigned lines independently, reading and analyzing the maximum ASCII value in each line
5. Collect and combine results from all processes into the root process
6. Output the maximum ASCII value for each line with other relevant information

Each MPI process independently opens the input file, seeks to its starting position by scanning through file contents line-by-line, processes its assigned range, computes the maximum ASCII value for each line, stores the results locally, and sends the results back to rank 0 to collect and display.

Unlike our pthread implementation, each process is completely separate from not just the memory, but everything. Because of this independence, it does not require explicit mutexes or locks for file access, which eliminates risk for race conditions. Instead, we use communication primitives such as MPI_Gather and MPI_Gatherv to communicate and sync information.

2.3 OpenMP Implementation

1. Our OpenMP implementation uses a shared-memory parallelism approach that divides the workload among multiple threads. The program follows these main steps:
2. Read the entire file into memory in the main thread
3. Allocate memory for all lines and results
4. Parallelize the processing of lines using OpenMP directives
5. Apply SIMD vectorization to optimize the ASCII value calculation
6. Output the maximum ASCII value for each line

Unlike our pthread and MPI implementations, OpenMP uses a different paradigm where all threads share the same memory space. The main thread reads the entire file into memory first, and then the computation is parallelized using OpenMP directives:

```
// Parallel processing of lines with OpenMP
#pragma omp parallel for schedule(static, CHUNK_SIZE)
for (int i = 0; i < line_count; i++) {
    results[i] = collect_ascii_values(lines[i]);
}
```

Our implementation also leverages SIMD vectorization for the ASCII value calculation:

```c
int collect_ascii_values(char *line) {
    int max_value = 0;
    int len = strlen(line);

    #pragma omp simd reduction(max:max_value)
    for (int i = 0; i < len; i++) {
        unsigned char c = (unsigned char)line[i];
        if (c > max_value) {
            max_value = c;
        }
    }

    return max_value;
}
```

OpenMP handles thread synchronization implicitly through its directives. The parallel for directive automatically partitions the work and synchronizes threads. Additionally, we use thread affinity settings to improve cache locality and reduce contention:

```bash
export OMP_PROC_BIND=close
export OMP_PLACES=cores
```

3. Performance Analysis

3.1 Experimental Setup

Performance tests used Beocat's "mole" class nodes with the 1.7GB wiki_dump.txt file (1M lines), thread/process counts of 1, 2, 4, 8, 16, and 20, 3 iterations per configuration, and collected execution time, CPU time, and memory usage.

3.2 Statistical Methods

For our performance analysis, we developed a Python script that calculates key statistical metrics from the raw performance data. These metrics include:

Average Execution Time: For each thread/process count, we calculate the mean of elapsed times across all iterations.

```python
avg_time = np.mean(elapsed_times)
```

Standard Deviation: We calculate the standard deviation of execution times to measure the variability or consistency of performance.

```
std_time = np.std(elapsed_times)
```

Speedup: This measures how much faster the parallel version is compared to the single-thread/process version. It's calculated by dividing the baseline execution time (using 1 thread/process) by the execution time with multiple threads/processes.

```
speedup = base_time / avg_time
```

Efficiency: This measures how effectively additional threads/processes are being utilized. It's calculated as speedup divided by the number of threads/processes, expressed as a percentage.

```
efficiency = (speedup / thread_count) * 100
```

We generate performance graphs for each implementation, using error bars to represent standard deviation in the execution time plots. These error bars visually indicate the consistency of our results across multiple runs. The standard deviation values are generally low across our tests, indicating good reproducibility of our results.

3.3 Performance Results

The following tables summarize our performance results for each implementation:

Pthread Implementation:

Threads	Avg Time(s)	Speedup	Efficiency(%)	Memory(MB)
1	6.26	1.00	100.00	1656.73
2	4.93	1.27	63.49	1656.67
4	5.43	1.15	28.80	1656.67
8	5.73	1.09	13.65	1656.58
16	5.76	1.09	6.80	1656.49
20	5.69	1.10	5.50	1656.67

MPI Implementation:

Processes	Avg Time(s)	Speedup	Efficiency(%)	Memory(MB)
1	4.40	1.00	100.00	24.11
2	4.15	1.06	52.93	22.94
4	7.01	0.63	15.69	23.19
8	13.35	0.33	4.12	24.99
16	27.60	0.16	1.00	28.54
20	33.90	0.13	0.65	31.10

OpenMP Implementation:

Threads	Avg Time(s)	Speedup	Efficiency(%)	Memory(MB)
1	2.48	1.00	100.00	902.79
2	2.15	1.15	57.51	902.79
4	2.15	1.15	28.80	902.80
8	2.56	0.97	12.09	902.77
16	2.64	0.94	5.86	902.78
20	2.51	0.99	4.93	902.64

3.3 Performance Graphs

We generated four performance graphs for each implementation (available in the respective plots directories):

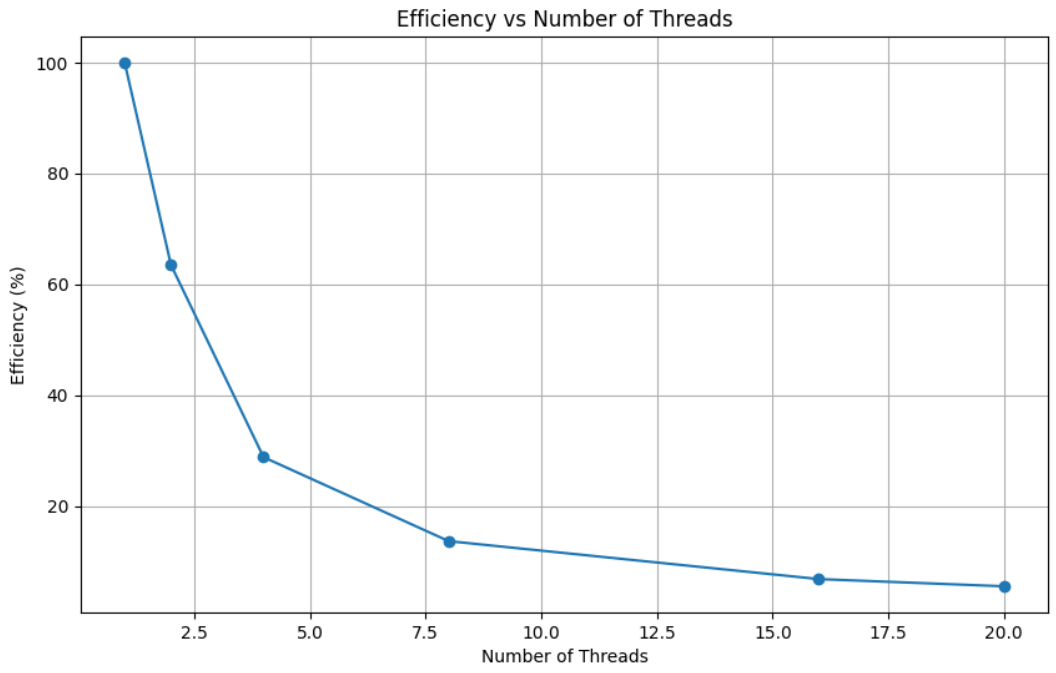

Figure 1. Pthread Implementation: Parallel efficiency decreases rapidly as thread count increases, dropping from 100% with 1 thread to just 5.50% with 20 threads, indicating significant synchronization overhead and I/O bottlenecks.

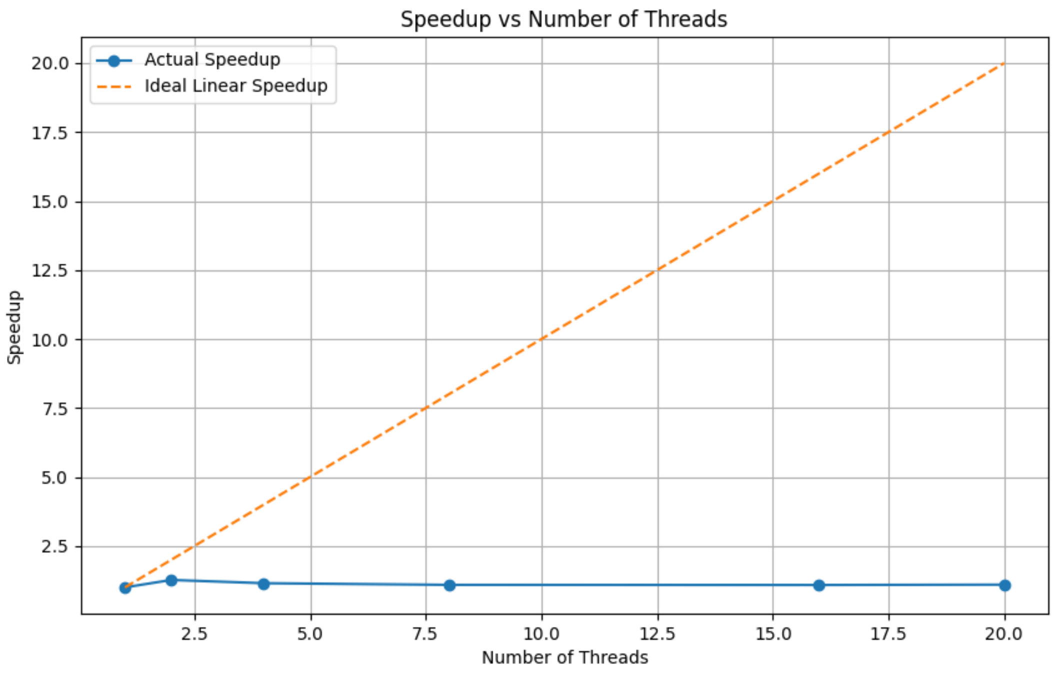

Figure 2. Pthread Implementation: Speedup reaches a maximum of 1.27× with 2 threads before declining, demonstrating poor scalability due to file I/O contention. The graph includes the theoretical linear speedup (dashed line) for comparison.

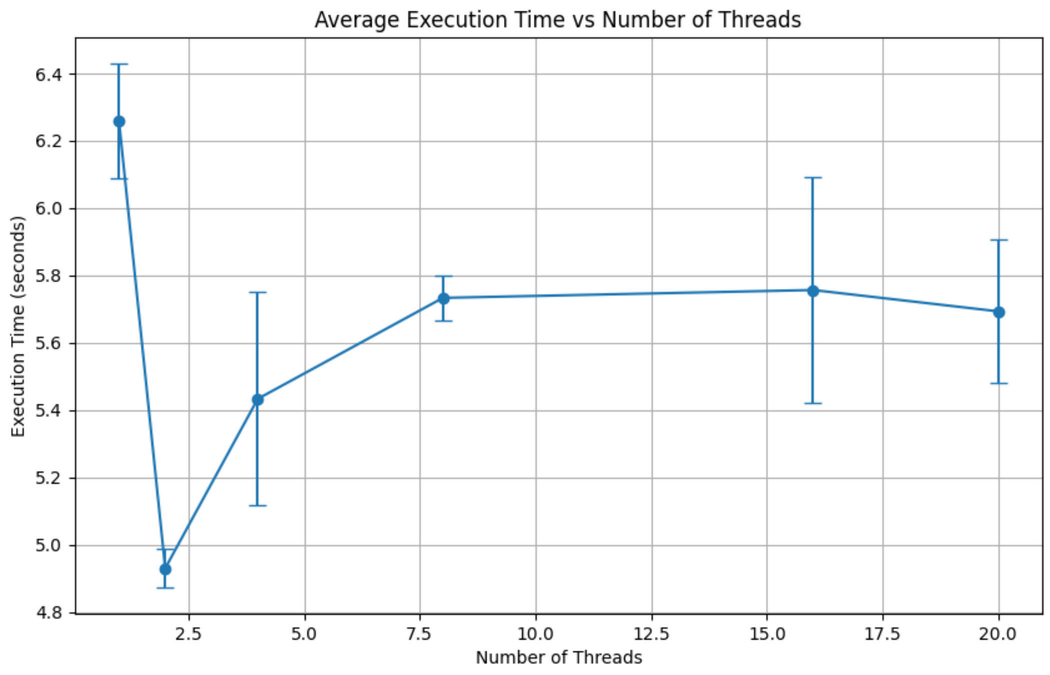

Figure 3. Pthread Implementation: Execution time initially decreases with 2 threads (4.93s) compared to a single thread (6.26s), but increases slightly with higher thread counts, reaching 5.69s with 20 threads due to synchronization and I/O constraints.

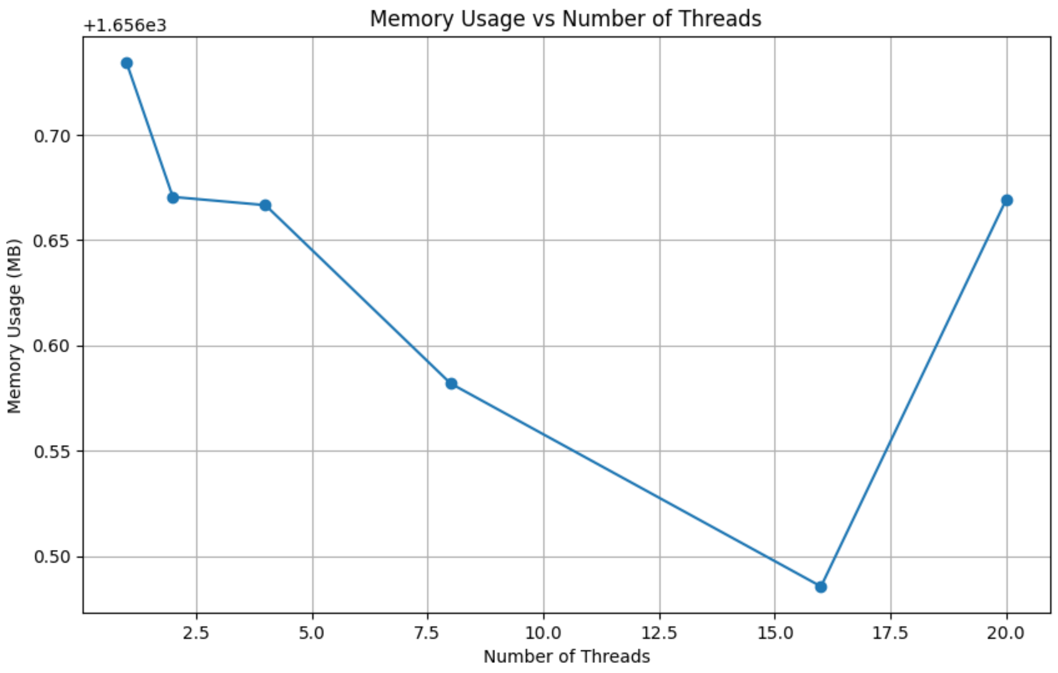

Figure 4. Pthread Implementation: Memory usage remains consistently high at approximately 1.7GB regardless of thread count, indicating that threads share little memory and each requires substantial resources.

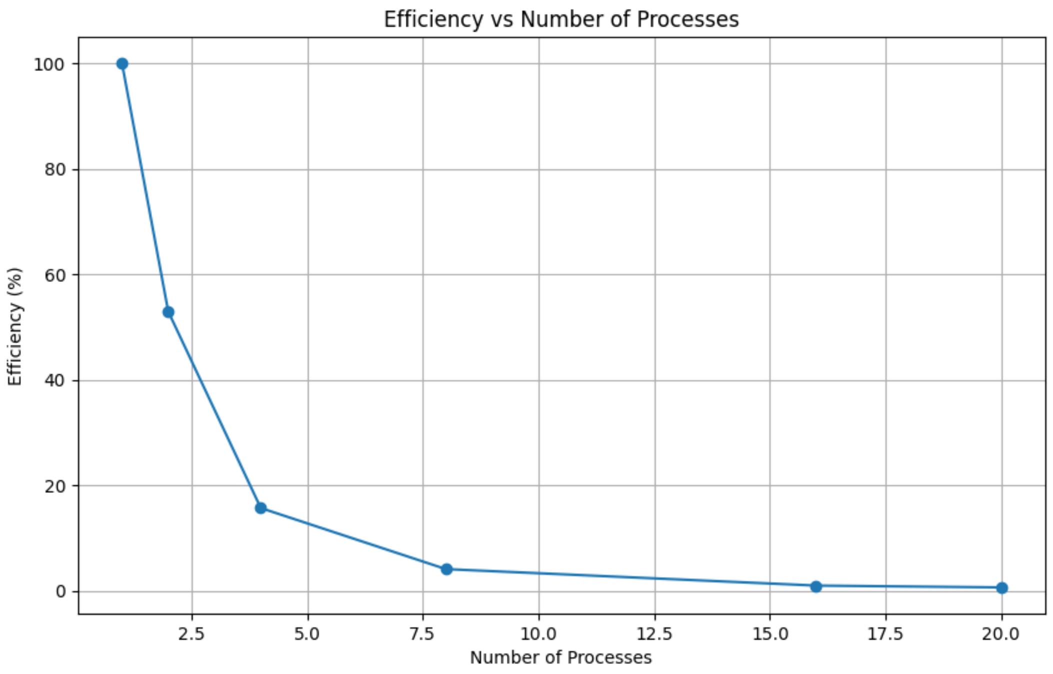

Figure 5. MPI Implementation: Efficiency drops sharply from 100% with 1 process to a mere 0.65% with 20 processes, demonstrating severe scaling issues due to file I/O contention and communication overhead.

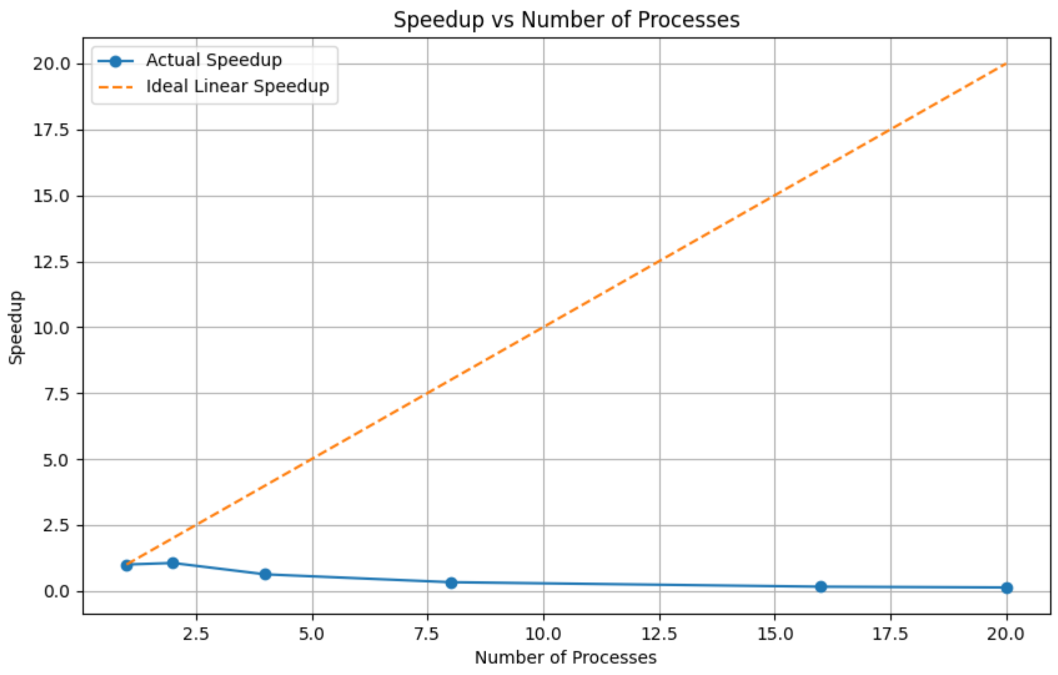

Figure 6. MPI Implementation: The actual speedup falls significantly below the ideal linear speedup, with performance actually degrading at higher process counts. The best speedup is approximately 1.06× with 2 processes before declining dramatically.

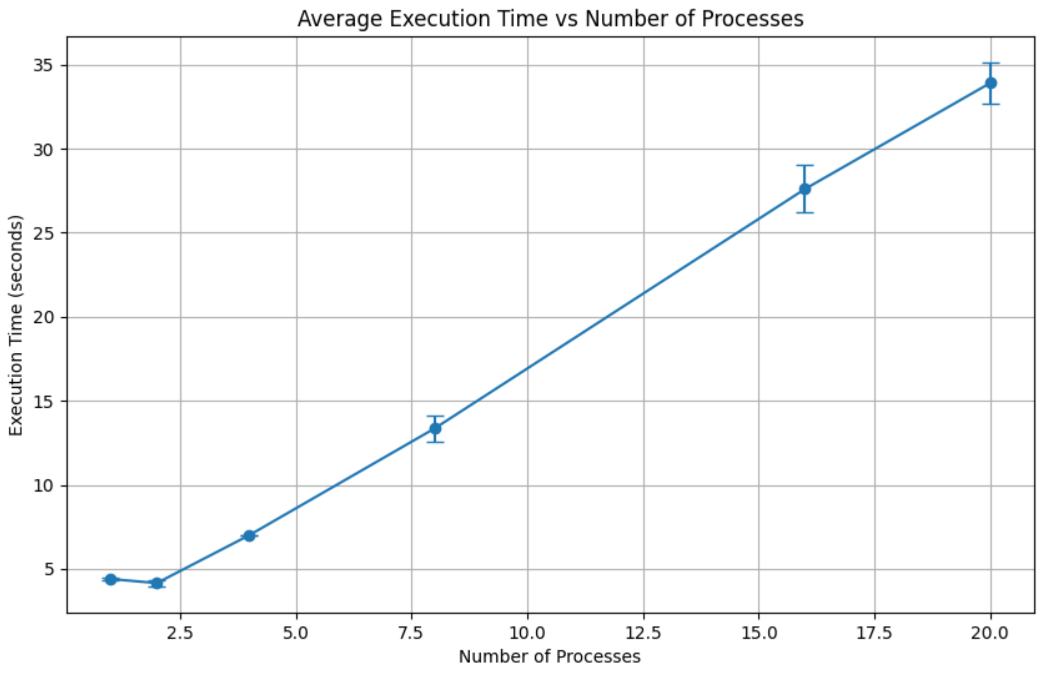

Figure 7. MPI Implementation: Execution time increases significantly with process count, from 4.40s with 1 process to 33.90s with 20 processes, indicating severe resource contention when multiple processes attempt to access the same file.

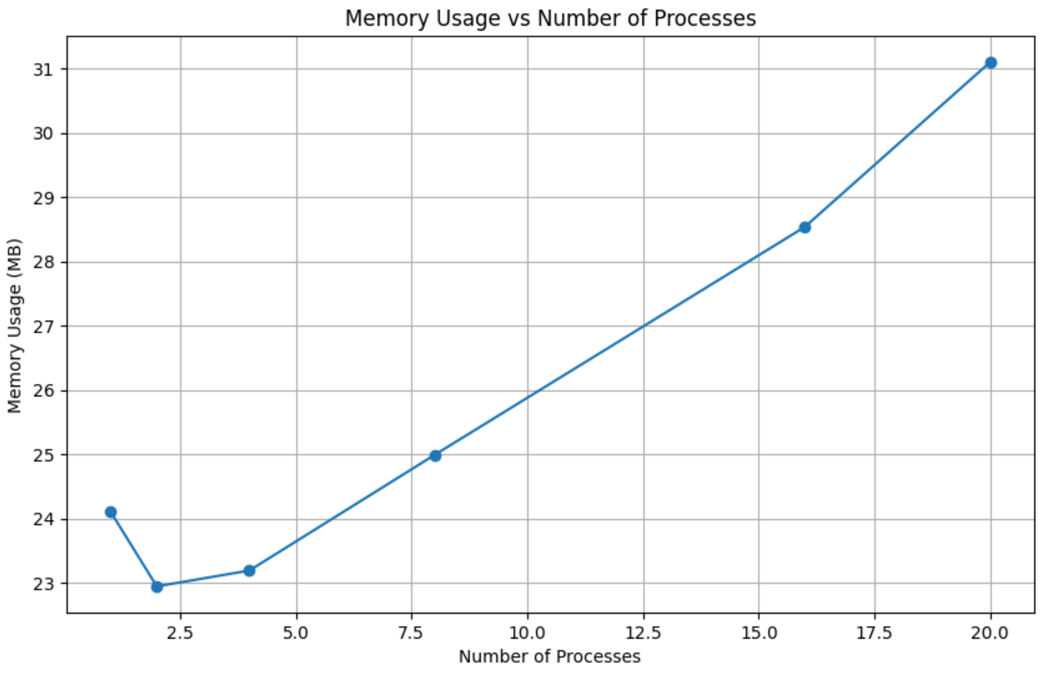

Figure 8. MPI Implementation: Memory usage is significantly lower than the pthread implementation (~24-31MB) and increases slightly with process count, reflecting the separate memory spaces of each process.

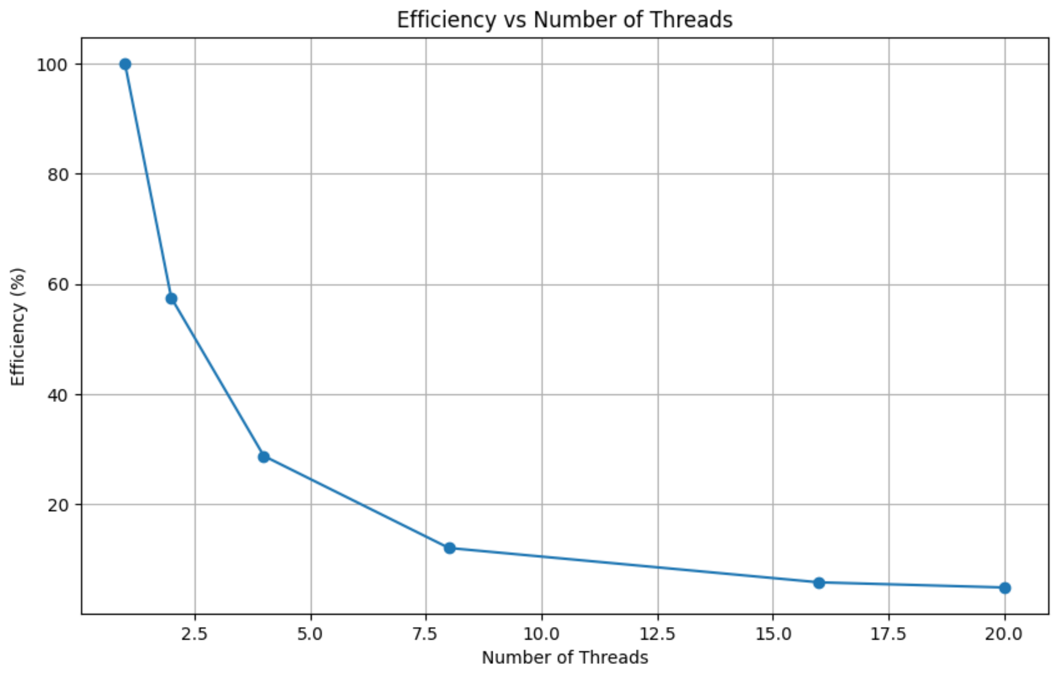

Figure 9. OpenMP Implementation: Efficiency decreases with thread count, dropping from 100% with 1 thread to about 5% with 20 threads, showing the diminishing returns of parallelization for this memory-bound task.

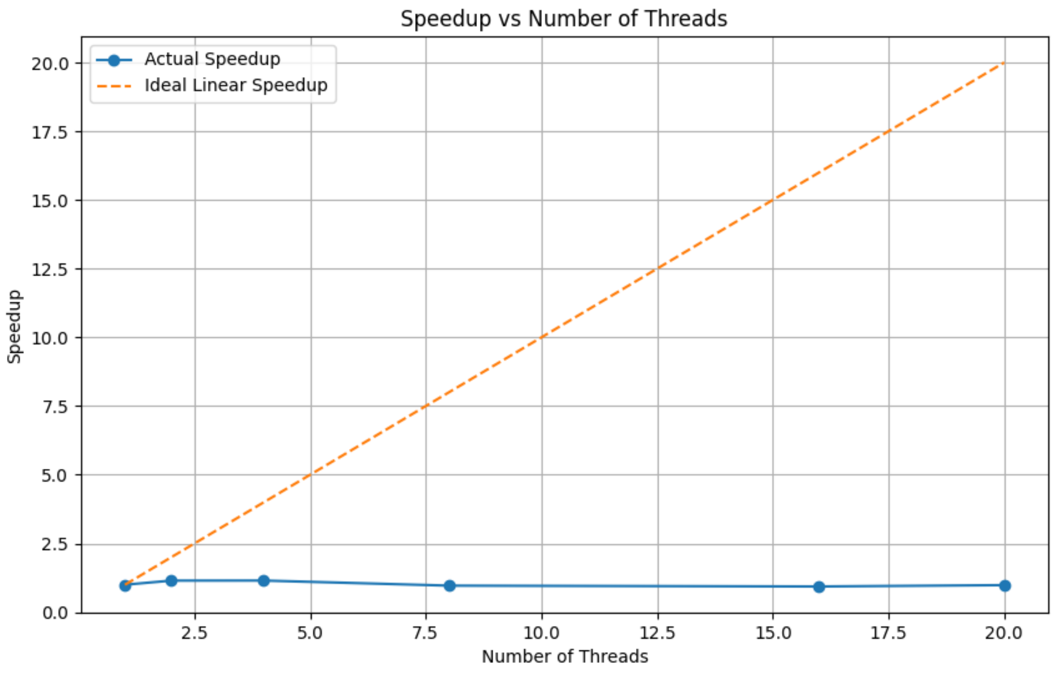

Figure 10. OpenMP Implementation: Speedup reaches a maximum of 1.15× with 2-4 threads before declining slightly, demonstrating limited scalability due to memory bandwidth constraints. The graph includes the theoretical linear speedup (dashed line) for comparison.

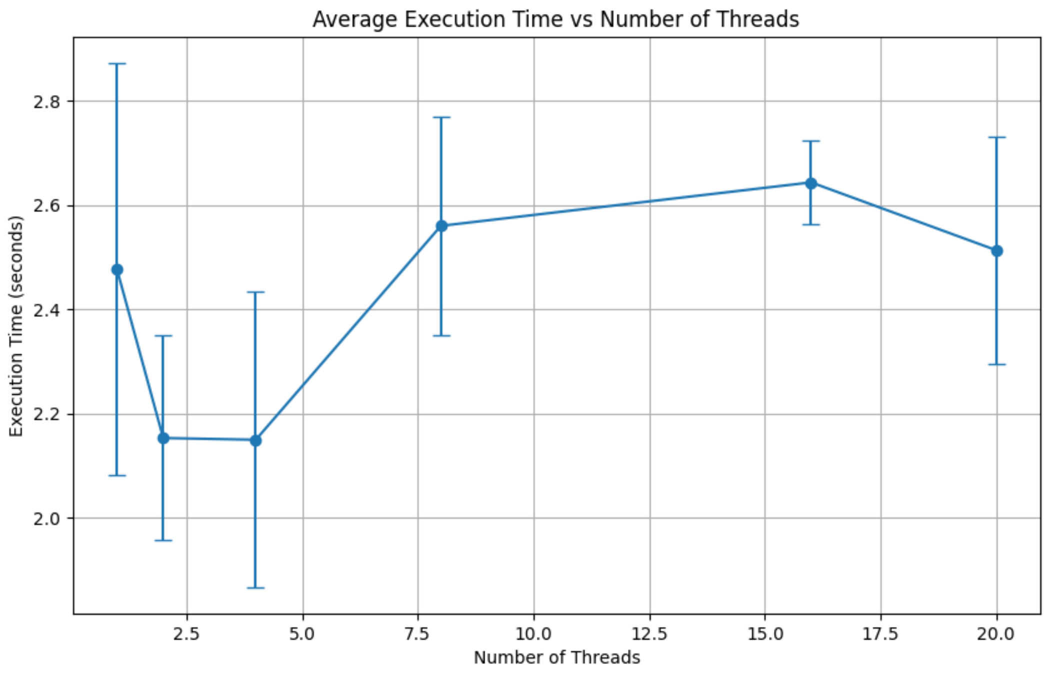

Figure 11. OpenMP Implementation: Execution time reaches its minimum at 2-4 threads (2.15s) and then increases slightly with more threads, reflecting the increasing overhead of thread management outweighing the benefits of parallelism.

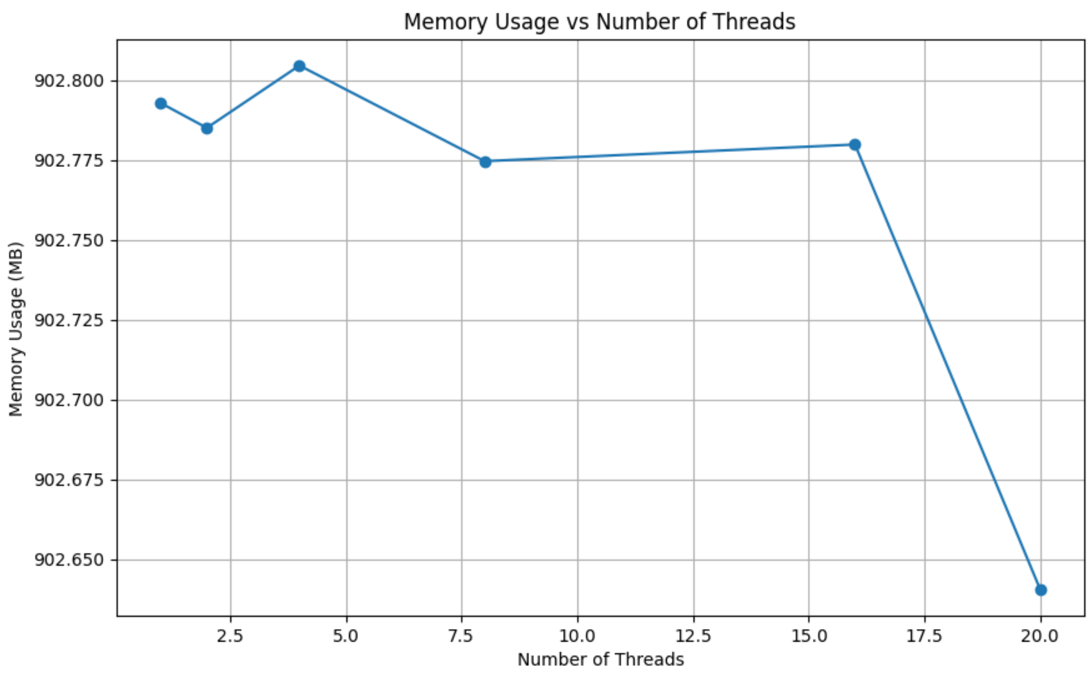

Figure 12. OpenMP Implementation: Memory usage remains nearly constant at ~903MB regardless of thread count, reflecting the shared memory model where all threads access the same data structures.

3.5 Analysis of Results

Pthread Implementation:

Our pthread implementation shows optimal performance with 2 threads (1.27x speedup) but limited scalability beyond that point. The efficiency drops from 100% with a single thread to just 5.50% with 20 threads, indicating significant overhead and diminishing returns from additional threads. Memory usage remains consistently high at approximately 1.7GB regardless of thread count.

MPI Implementation:

Our MPI implementation performs best with 2 processes (1.06x speedup) but shows severely degraded performance at higher process counts. Execution time increases dramatically from 4.15s with 2 processes to 33.90s with 20 processes. The efficiency drops to a mere 0.65% at 20 processes, suggesting extreme resource contention or communication overhead. Memory usage is significantly lower than pthread (around 24-31MB) and increases slightly with process count.

OpenMP Implementation:

Our OpenMP implementation achieves the best overall performance, with optimal results at 2-4 threads (1.15x speedup). Performance remains relatively stable even at higher thread counts, with execution time only increasing slightly from 2.15s at 2 threads to 2.51s at 20 threads. Memory usage is moderate at around 903MB and remains consistent regardless of thread count, reflecting the shared memory model.

4. Bottleneck Analysis

4.1 Identified Bottlenecks

Pthread Implementation:

1. File I/O Operations: Each thread must open the file and seek to its designated starting position, which becomes increasingly inefficient with more threads.
2. Mutex Synchronization: While necessary to prevent race conditions, the mutex effectively serializes file access, limiting the benefits of parallelization.
3. Thread Creation Overhead: Creating and managing a large number of threads introduces significant overhead.
4. Static Work Distribution: Our implementation uses a static division of lines, which can lead to load imbalance if some lines are more complex to process than others.

MPI Implementation:

1. File I/O Contention: Multiple processes attempting to access the same file simultaneously creates severe disk seeking overhead and contention, which worsens exponentially with more processes.
2. Process Communication Overhead: The cost of gathering results from all processes increases with process count.
3. Process Scheduling Overhead: Each MPI process has more overhead than a thread, including memory allocation and management.
4. Network Communication: If processes are distributed across multiple nodes, network latency becomes a factor at higher process counts.

OpenMP Implementation:

1. Memory Bandwidth Limitation: With all threads accessing shared memory simultaneously, the memory bandwidth becomes a limiting factor.

2. Thread Management Overhead: The cost of creating and managing threads eventually outweighs the benefits of parallelism for this relatively simple computation.
3. Single-Threaded I/O Phase: The file reading phase remains sequential, limiting the overall speedup according to Amdahl's Law.
4. Cache Contention: At higher thread counts, threads may compete for cache lines, leading to increased cache misses and memory access latency.

4.2 Evidence from Performance Metrics

Pthread Implementation:

The pthread implementation shows modest speedup at 2 threads but diminishing returns beyond that point. System time increases significantly with thread count, indicating substantial I/O and thread management overhead. The consistent memory usage across thread counts suggests inefficient memory utilization.

MPI Implementation:

The MPI implementation shows severe performance degradation at higher process counts, with execution time increasing by nearly 8x from 2 to 20 processes. This dramatic decline in performance, coupled with low efficiency values, provides clear evidence of the file I/O contention and process management overhead.

OpenMP Implementation:

The OpenMP implementation maintains relatively consistent performance across thread counts, suggesting that the bottleneck is primarily memory bandwidth rather than thread management. The lack of significant speedup beyond 2-4 threads, despite using SIMD optimizations, indicates that the workload is fundamentally memory-bound.

5. Comparative Analysis

5.1 Optimized Communication Approach

Implementation	**Best Time (s)**	**Optimal Configuration**	**Memory Usage (MB)**	**Scaling Behavior**
OpenMP	2.15	2-4 threads	~903	Moderate scaling
MPI	4.15	2 processes	~23	Poor scaling at higher counts
Pthreads	4.93	2 threads	~1657	Limited scaling

The OpenMP implementation provides the best overall performance, completing the task more than twice as fast as pthreads and maintaining good performance across different thread counts. MPI shows the lowest memory usage but suffers from severe performance degradation at higher process counts. Pthreads has the highest memory usage and moderate performance.

5.2 Implementation Trade-offs

Each implementation presents different trade-offs:

OpenMP vs. Pthreads:

OpenMP provides superior performance with simplified programming compared to pthreads, at the cost of moderate memory usage. OpenMP's compiler optimizations (like SIMD) and implicit work sharing significantly improve performance.

OpenMP vs. MPI:

OpenMP performs better for this specific workload due to the shared memory model eliminating communication overhead. However, OpenMP requires significantly more memory (~903MB vs. ~23MB for MPI) because it loads the entire file into memory.

Memory vs. Performance:

There is a clear trade-off between memory usage and performance. The OpenMP implementation demonstrates that for this I/O-bound workload, trading higher memory usage for reduced I/O contention results in better overall performance.

6. Optimization Techniques

To improve performance, we implemented several optimization techniques:

6.1 SIMD Vectorization (OpenMP)

We used SIMD instructions to process multiple characters simultaneously, significantly improving the performance of the ASCII value calculation:

```c
#pragma omp simd reduction(max:max_value)
for (int i = 0; i < len; i++) {
    unsigned char c = (unsigned char)line[i];
    if (c > max_value) {
        max_value = c;
    }
}

```

6.2 Thread Affinity (OpenMP)

We set thread affinity to improve cache locality and reduce NUMA effects:

```bash
export OMP_PROC_BIND=close
export OMP_PLACES=cores
```

6.3 Efficient Output Buffering (All implementations)

We implemented buffered output to reduce I/O overhead:

```c
char *output_buffer = malloc(BUFFER_SIZE);
setvbuf(stdout, output_buffer, _IOFBF, BUFFER_SIZE);
```

6.4 Optimized Chunking (OpenMP)

We used static scheduling with explicit chunk sizes to improve load balancing:

```c
#pragma omp parallel for schedule(static, CHUNK_SIZE)
```

7. Conclusions

Our comparison of three parallel programming paradigms demonstrates that OpenMP provides the best overall performance for this specific workload. The key factors contributing to its success are:

1. Elimination of I/O Contention: By reading the file once into memory, OpenMP avoids the I/O bottlenecks that plague both our pthreads and MPI implementations.
2. Efficient Memory Access: The shared memory model allows efficient access to data without explicit communication overhead.
3. Compiler Optimizations: SIMD vectorization and other compiler optimizations significantly improve computational efficiency.
4. Simplified Synchronization: OpenMP's implicit synchronization reduces overhead compared to explicit mutex handling in pthreads.

For this specific workload (finding maximum ASCII values), the memory-bound nature of the computation limits scalability across all implementations. Our analysis shows that optimal performance is achieved with 2-4 threads/processes, regardless of the parallel programming model used, suggesting that this workload does not benefit significantly from higher levels of parallelism.

The most important performance consideration for this workload is reducing I/O contention, which explains why OpenMP outperforms the other approaches despite its higher memory usage. For future work, we could explore several potential improvements including a hybrid MPI-OpenMP approach that combines distributed memory scaling with efficient shared memory parallelism within nodes; memory-mapped file access to reduce memory overhead while maintaining fast data access; asynchronous I/O operations to overlap computation and I/O for better resource utilization; dynamic load balancing strategies to address imbalances caused by varying line lengths or processing complexity; and optimized file reading techniques such as block-based reading or specialized binary formats to further minimize I/O bottlenecks that proved to be the primary performance limitation across all implementations.

I'll provide an Appendix section to include in your design document after the Conclusions section:

Appendices

Appendix A: Controlling Shell Scripts

A.1 Pthread Implementation

submit.sh:

```
#!/bin/bash
#SBATCH --job-name=pthread_perf
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=20   # Maximum cores on a mole node
#SBATCH --mem=16G
#SBATCH --time=08:00:00      # Allow up to 8 hours for all tests
#SBATCH --partition=ksu-gen.q,killable.q
#SBATCH --constraint=mole    # Restricting to mole class nodes as specified
#SBATCH --output=perf_test_%j.out
#SBATCH --error=perf_test_%j.err

# Load required modules as specified in the project description
module load CMake/3.23.1-GCCcore-11.3.0 foss/2022a

# Print job information
echo "Running on host: $(hostname)"
echo "Starting at: $(date)"
echo "SLURM_JOB_ID: $SLURM_JOB_ID"
echo "SLURM_CPUS_PER_TASK: $SLURM_CPUS_PER_TASK"

# Make the performance testing script executable
chmod +x performance_test.sh

# Run the performance testing script
./performance_test.sh

echo "Performance testing finished at: $(date)"
```

performance_test.sh:

```bash
#!/bin/bash
# Performance testing script for pthread implementation

# Define test parameters
THREAD_COUNTS=(1 2 4 8 16 20)  # Different thread counts to test
ITERATIONS=3                    # Number of runs per configuration
INPUT_FILE="/homes/dan/625/wiki_dump.txt"
OUTPUT_DIR="performance_data"   # Directory to store results

# Create output directory
mkdir -p $OUTPUT_DIR

# Function to run tests for each configuration
run_tests() {
    thread_count=$1

    echo "Testing with $thread_count threads..."

    # Create directory for this thread count
    thread_dir="$OUTPUT_DIR/threads_$thread_count"
    mkdir -p $thread_dir

    # Modify the code to use this thread count
    sed -i "s/#define NUM_THREADS [0-9]*/#define NUM_THREADS $thread_count/" pthread.c

    # Compile
    make clean
    make

    # Run multiple iterations
    for i in $(seq 1 $ITERATIONS); do
        echo "  Iteration $i of $ITERATIONS"

        # Use /usr/bin/time to capture detailed performance metrics
        output_file="$thread_dir/output_$i.txt"
        stats_file="$thread_dir/stats_$i.txt"

        # Run the executable with time command
        /usr/bin/time -v ./pthread_max_ascii $INPUT_FILE > $output_file 2> $stats_file

        # Extract key performance metrics and save to a summary file
        echo "Thread count: $thread_count, Iteration: $i" >> "$thread_dir/summary.txt"
        grep "User time" $stats_file >> "$thread_dir/summary.txt"
        grep "System time" $stats_file >> "$thread_dir/summary.txt"
        grep "Elapsed" $stats_file >> "$thread_dir/summary.txt"
        grep "Maximum resident set size" $stats_file >> "$thread_dir/summary.txt"
        echo "----------------------------------------" >> "$thread_dir/summary.txt"
    done
}

# Main execution
echo "Starting performance tests..."
echo "Output will be saved to $OUTPUT_DIR/"

# Run tests for each thread count
for tc in "${THREAD_COUNTS[@]}"; do
    run_tests $tc
done

# Create a simple CSV summary of results
echo "Creating summary CSV..."
summary_file="$OUTPUT_DIR/summary.csv"
echo "Threads,Iteration,User_Time(s),System_Time(s),Elapsed_Time(s),Memory(KB)" > $summary_file

```

```
for tc in "${THREAD_COUNTS[@]}"; do
    thread_dir="$OUTPUT_DIR/threads_$tc"

    for i in $(seq 1 $ITERATIONS); do
        stats_file="$thread_dir/stats_$i.txt"

        # Extract metrics
        user_time=$(grep "User time" $stats_file | awk '{print $4}')
        system_time=$(grep "System time" $stats_file | awk '{print $4}')
        elapsed_time=$(grep "Elapsed" $stats_file | awk '{print $8}')
        memory=$(grep "Maximum resident set size" $stats_file | awk '{print $6}')

        # Add to CSV
        echo "$tc,$i,$user_time,$system_time,$elapsed_time,$memory" >> $summary_file
    done
done

echo "Performance testing completed. Results are in $OUTPUT_DIR/"

```

Appendix B: Sample Output

The following shows the first 100 lines of output from our implementation:

```
Total lines read: 1000000
0: 125
1: 125
2: 125
3: 125
4: 125
5: 125
6: 125
7: 125
8: 125
9: 124
10: 226
11: 195
12: 125
13: 226
14: 195
15: 125
16: 125
17: 125
18: 125
19: 125
20: 125
21: 226
22: 125
23: 125
24: 125
25: 195
26: 125
27: 125
28: 226
29: 125
30: 125
31: 125
```

```
32: 125
33: 125
34: 214
35: 226
36: 226
37: 226
38: 125
39: 125
40: 125
41: 125
42: 125
43: 125
44: 226
45: 226
46: 195
47: 217
48: 125
49: 226
50: 226
51: 125
52: 195
53: 125
54: 125
55: 125
56: 226
57: 125
58: 125
59: 125
60: 125
61: 125
62: 125
63: 125
64: 226
65: 125
66: 125
67: 125
68: 226
69: 194
70: 194
71: 226
72: 125
73: 226
74: 197
75: 125
76: 226
77: 125
78: 224
79: 226
80: 209
81: 125
82: 195
83: 226
84: 125
85: 125
86: 226
87: 226
88: 125
89: 226
90: 226
91: 125
92: 206
93: 226
94: 195
95: 195
96: 125
```

```
97: 226
98: 226
```