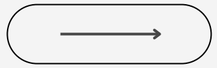
PARALLEL AND DISTRIBUTED COMPUTING

Ramel Cary Jamen

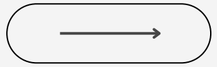
ACCELERATING HUFFMAN CODING COMPRESSION USING MPI



PARALLEL AND DISTRIBUTED COMPUTING
Ramel Cary Jamen

OVERVIEW

The main focus of this study is to speed up the sequential algorithm of Huffman Coding. This algorithm uses the frequency of variable-length strings to represent symbols in a binary tree. The study aims to find out how applying parallelization affects the running time of the algorithm and how it can be implemented.



PARALLEL AND DISTRIBUTED COMPUTING
Ramel Cary Jamen

OBJECTIVES

Formulate and Implement the following:

- Sequential Version of Huffman Coding Compressor - Serves as the basis for identifying significant difference for the formulated parallelized algorithm
- Parallelized Version of Huffman Coding Compressor - This will be formulated after thorough assessment for parallelization
- Identify Findings - To be able to assess efficiency and performance differences for both algorithm

METHODOLOGY

- Generating the Sample Input File

Figure 4: Command for Input File

```
$  tr -dc "A-Za-z 0-9" < /dev/urandom | fold -w100 | head -n
   1000000 > textfile.txt
```

The resulting file, named "textfile.txt" in this case, contains a large amount of random text. It is made up of a specific number of lines, with each line being a certain length. This file serves as a representative example for testing the Huffman Coding algorithm.

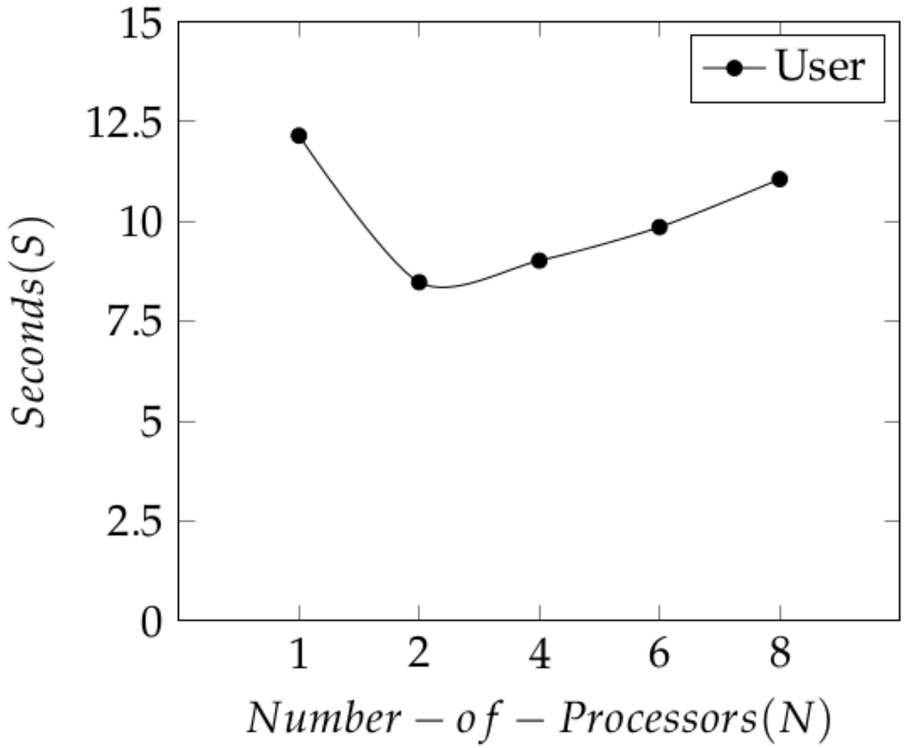
METHODOLOGY

- Assessing the Sequential Algorithm
  - Identify the time-consuming sections of the sequential algorithm
  - Explain the challenges and limitations of the sequential approach
  - Introduce the need for parallelization
- Parallelized Huffman Coding Algorithm
  - Present the concept of the Multiple Instruction, Single Data (MISD) model
  - Explain how the MISD model improves the algorithm's efficiency

FINDINGS

- In figure 5, the user time is the total number of CPU-seconds that the process used directly (in user mode), it is expressed in seconds. This is only actual CPU time used in executing the process.

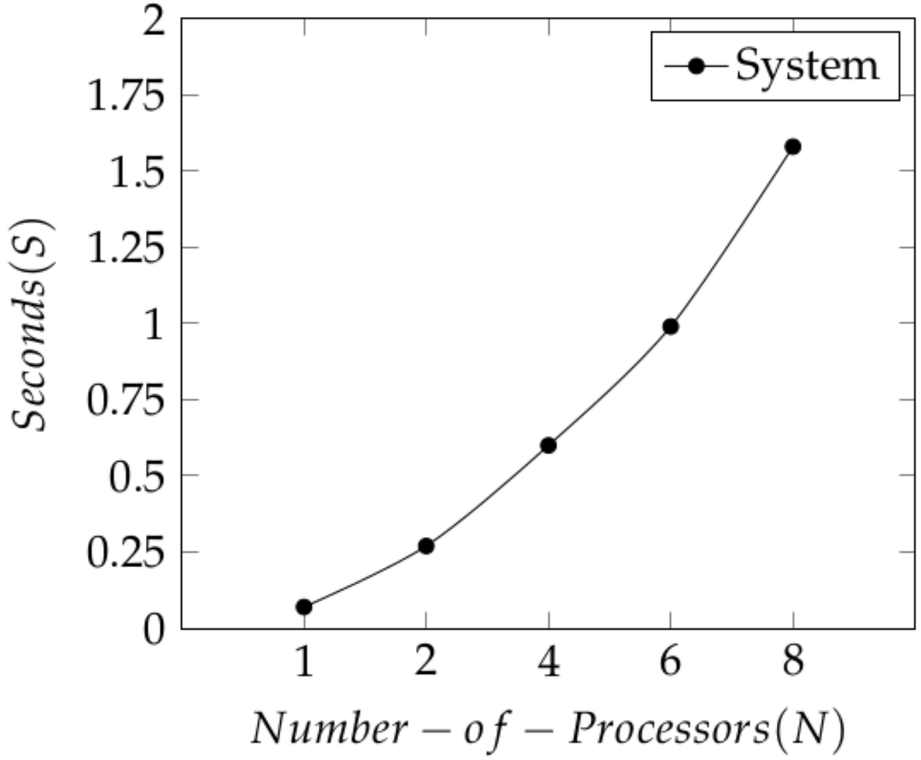
Figure 5: User Run Time



FINDINGS

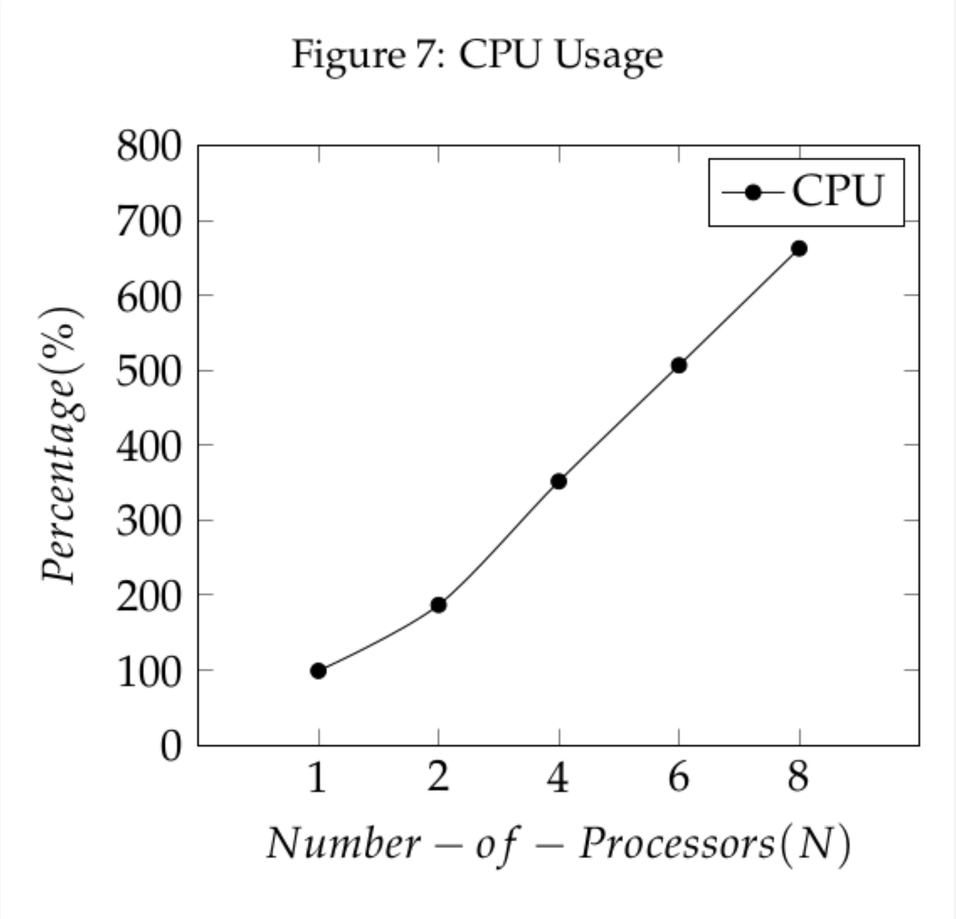
- In figure 6, it is total number of CPU-seconds used by the system on behalf of the process (in kernel mode), it is expressed in seconds. This means executing CPU time spent in system calls within the kernel, as opposed to library code, which is still running in user-space. Like 'user', this is only CPU time used by the process. Graph shows an increase of system run time as the number of processor increases.

Figure 6: System Run Time



FINDINGS

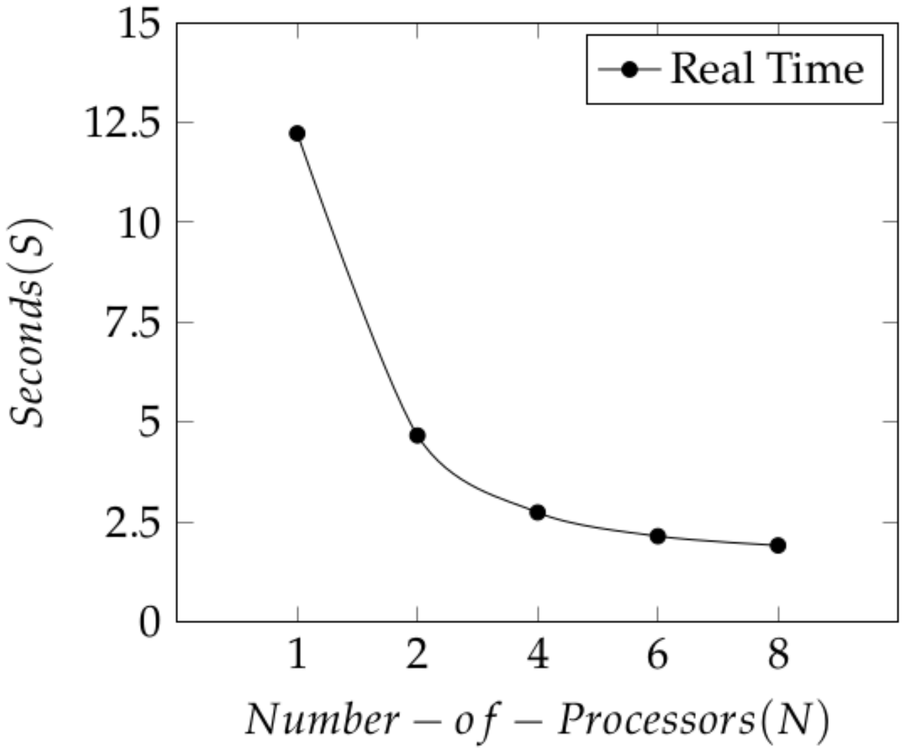
- In figure 7, percentage of the CPU that this job got. This is just user + system times divided by the total running time. Graph shows an increase of CPU usage as the number of processor increases.



FINDINGS

- In figure 8, Elapsed real (wall clock) time used by the process, in seconds. This is all elapsed time including time slices used by other processes and time the process spends blocked. Graph shows a drastic decrease of time-spent from single processor to double processor used. However, from 2 to 8 processors, it shows a minimal effectivity.

Figure 8: Real Run Time



CONCLUSION

- In summary, parallelization brings many advantages in making computations faster and improving system performance by doing multiple tasks at the same time on different processing units. It works best for tasks that can be divided into smaller independent parts.
- However, parallelization has some limitations when tasks depend on each other or require a lot of coordination and communication. The availability of resources and scalability also affect its effectiveness.
- To get the most out of parallelization, it's important to consider the nature of the tasks and use suitable techniques. While parallelization has great potential for boosting performance, it's essential to be aware of its limitations as well.