Assignment 2 report
SPM course
2024/25

Francesco SCARFATO

May 1, 2025

1 Objective

The objective of this project is to implement a parallel file compression tool using the Miniz compression library in conjunction with OpenMP for parallelization.

2 Implementation

The program supports both compression and decompression of files and directories. Small files are processed sequentially using a single compression block, while large files are split into multiple blocks and processed in parallel. Each block is handled independently, and its metadata (original size, compressed size, block index) is stored alongside the compressed data. The metadata are stored in a structure `DataBlock`. For large files, OpenMP's `taskloop` is used to distribute compression and decompression tasks among available threads, maximizing CPU utilization. The critical part is how to split and compute the block size of each block.

```
int num_threads = omp_get_max_threads();
// Compute the number of blocks to process
size_t numBlocks = (size + BIG_FILE_SIZE - 1) / BIG_FILE_SIZE;
// Limit to a reasonable number of blocks
numBlocks = std::min(numBlocks, static_cast<size_t>(num_threads * 2));
// Compute the block size of each block
size_t blockSize = (size + numBlocks - 1) / numBlocks;
```

In the C++ implementation, the ceiling division operations $\lceil \frac{a}{b} \rceil$ are performed using the integer arithmetic expression $\frac{a+b-1}{b}$, which ensures proper rounding up without floating-point operations.

a. The algorithm first determines the available processing resources by querying the OpenMP runtime for the maximum number of threads.

b. It then calculates an initial number of block based on the file size and a predefined constant block size threshold (`BIG_FILE_SIZE`). This ensures that no single block exceeds the threshold, which could impact cache efficiency.

c. The key optimization occurs in the third step, where the number of blocks is limited to twice the available thread count. This constraint serves multiple purposes: it prevents excessive threading overhead from too many small blocks, while still providing sufficient number of tasks.

d. Finally, the algorithm recalculates the actual block size by dividing the total file size by the adjusted number of blocks. This ensures uniform block sizes (except potentially the last block).

The output file begins with the number of blocks, followed by metadata for each block and then the compressed block data. During decompression, metadata is first read to determine block sizes and offsets, and blocks are then decompressed in parallel. Also the compression of files within a directory tree is done in parallel using OpenMP's `parallel for` with a dynamic scheduling because the files can have different sizes.

For compressing large files, OpenMP's `taskloop` is preferred over `parallel for` because the time to compress each block may vary significantly, even when block sizes are uniform, because the compression depends on the data content more than size (repetitve data or random data). The task-based model offers better dynamic load balancing and is more scalable for irregular workloads. In contrast, when processing a list of independent files (e.g., in a directory), `parallel for` with `schedule(dynamic)` is more appropriate due to its simplicity and lower overhead for coarse-grained, independent tasks.

To further optimize task execution, the `nowait` clause is used on the `single` directive that wraps the `taskloop`. This prevents an implicit barrier at the end of the `single` region, allowing other threads to start executing tasks immediately, rather than waiting for the single thread to finish creating all tasks. This improves parallel efficiency and reduces overhead.

Careful synchronization mechanisms, such as `#pragma omp critical`, ensure thread-safe updates to shared resources and error handling flags.

3 Performance analysis

The performance tests were conducted on the internal nodes of the cluster. Each test was repeated five times, and the average execution time was calculated to ensure reliable results. The first test measures the compression on 200 small files (500 KB), while the second on 5 large files (50 MB).

3.1 Small files

Thread Count	Sequential Time (s)	Parallel Time (s)	Speedup
1	4.657833	-	-
4	-	1.258342	3.701
8	-	0.660728	7.050
16	-	0.336616	13.841
32	-	0.306606	15.193

Table 1: Execution times and speedup for sequential and parallel executions on small files

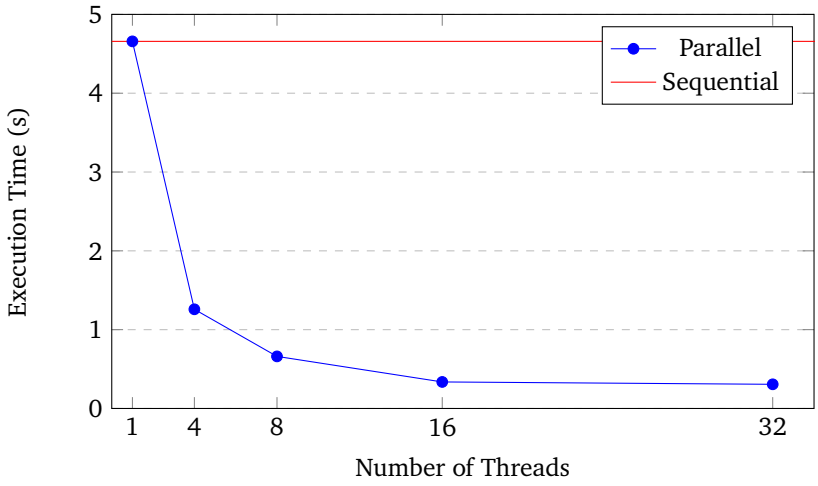

Figure 1: Comparison of sequential vs parallel execution times on small files

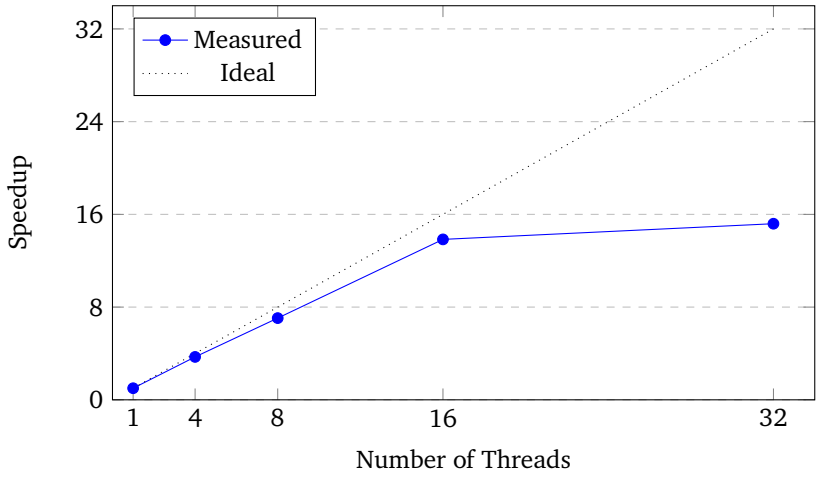

Figure 2: Speedup on small files

For inputs smaller than 2 Mb, the speedup curve in the speedup plot grows almost linearly up to 16 threads and remains strong at 32 threads. This strong scalability can be attributed to:

- The dynamic scheduling allows idle threads to pick up remaining work, helping mitigate load imbalance.
- There are no task creations or complex dependencies; each thread compresses files independently.
- Each thread operates on separate buffers, avoiding contention and reducing false sharing.

3.2 Large files

Thread Count	Sequential Time (s)	Parallel Time (s)	Speedup
1	12.179019	-	-
4	-	5.026272	2.42
8	-	2.675646	4.55
16	-	2.696441	4.51
32	-	2.710757	4.49

Table 2: Execution times and speedup for sequential and parallel executions on large files

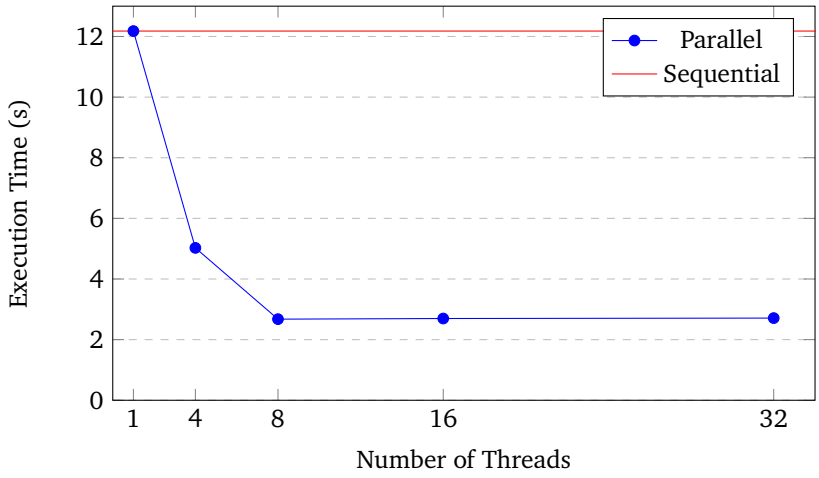

Figure 3: Comparison of sequential vs parallel execution times on large files

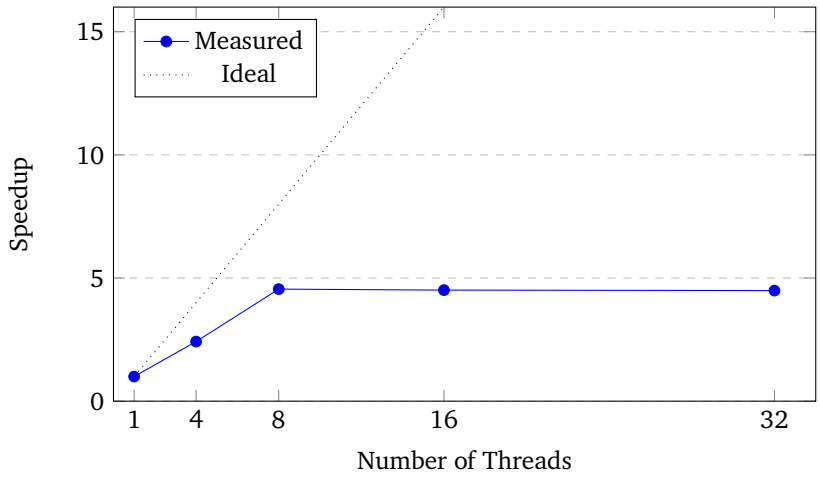

Figure 4: Speedup on large files

The baseline sequential execution with a single thread took approximately 12.18 seconds. Increasing the number of threads, we observe a significant reduction in execution time up to 8 threads, where performance nearly quadruples. However, execution time remains around 2.7

seconds for 16 and 32 threads, indicating that the workload is no longer able to scale linearly with additional threads.
Probably, this behavior is caused by overheads from task scheduling and synchronization.
Despite these limits, the approach demonstrates strong scalability up to a moderate number of threads, achieving more than 4x speedup with 8 threads.

4 Further improvements

While the current implementation demonstrates effective parallelism using OpenMP, there could be several optimizations and improvements. One key direction is to parallelize the processing of all command-line arguments (files or directories) given to the program. At present, each input is handled sequentially. Enabling parallel processing at the argument level could allow the system to compress multiple files or directory trees concurrently.
Another improvement involves testing the algorithm with larger datasets, such as very large files or collections of many large files. The current benchmarks focus on a small batch of large files, which already show significant speedup. However, evaluating the performance under heavier workloads could reveal additional scaling behaviors.
Finally, another possible enhancement lies in the parallelization of the output phase. Currently, the writing of compressed data blocks to the output file is performed sequentially after all parallel tasks complete. This introduces a serialization point that could become a bottleneck, particularly for large numbers of blocks. This could be improved using parallel techniques to improve the efficiency without compromising the structure of the output file.