Assignment 2: Miniz-based parallel compressor/decompressor in OpenMP

Luca Lombardo
SPM Course a.a. 24/25

1 Implementation Strategies

My approach to the design of the compressor is rooted in the observation that different input scenarios require different parallelization strategies to achieve both efficiency and correctness. I distinguish three main cases: many small files, a single large file, and many large files.

When compressing many small files, I treat each file as an independent unit of work. This is a natural fit for parallelization, since there are no dependencies between files and the workload can be evenly distributed among threads. I use an OpenMP `parallel for` with dynamic scheduling to ensure that threads are kept busy even if file sizes vary significantly. Each file is memory-mapped using POSIX `mmap`, which allows the operating system to handle paging efficiently and avoids explicit read calls and unnecessary buffer copies. For files smaller than a configurable threshold (16 MiB by default), I process them in a single thread to avoid the overhead of spawning parallel regions, which would outweigh any potential speedup for such small data sizes. Larger files are instead handed off to a block-based routine, described below.

For a single large file, I divide the file into fixed-size blocks (with a default block size of 1 MiB) and compress each block independently in parallel. Each thread is assigned a block and uses its own thread-local Miniz compressor instance, which avoids repeated initialization and ensures thread safety. The choice of block size is critical: smaller blocks can improve load balancing and parallel efficiency, but increase the amount of metadata and may reduce the compression ratio due to loss of context between blocks. I experiment with different block sizes to empirically determine the best trade-off for both throughput and compression effectiveness.

The most challenging scenario is that of many large files, because it exposes two distinct layers of parallelism: parallelism across files and parallelism within each file (i.e., across blocks). A naïve approach would be to parallelize both the outer file loop and the inner block loop using the same number of threads, but this can easily lead to oversubscription, where the system creates more threads than there are available cores, resulting in excessive context switching and degraded performance. To address this, I implemented three distinct strategies, each with its own rationale.

The first strategy is to process files sequentially, but to compress each file in parallel by distributing its blocks among threads. This approach is simple and avoids oversubscription, but may underutilize available cores if the number of files is small or if some files are much larger than others.

The second strategy is to parallelize both the file loop and the block loop, assigning the same number of threads to each. While this maximizes the potential for concurrency, it can easily result in the creation of $p \times p$ threads (where p is the number of hardware threads), which is almost always counterproductive on real systems.

The third and most effective strategy is to control the nesting of parallel regions by partitioning the total thread budget between the outer and inner loops. Specifically, for a given total number of threads p, I choose the number of threads for the outer file loop (t_{out}) and the inner block loop (t_{in}) such that $t_{\text{out}} \times t_{\text{in}} \leq p$. I use the heuristic $t_{\text{out}} = \lceil \sqrt{p} \rceil$ and $t_{\text{in}} = \lfloor p / t_{\text{out}} \rfloor$, which balances the available resources between file-level and block-level parallelism. This approach ensures that the system never creates more threads than there are cores, while still exploiting both levels of parallelism to the fullest extent possible. In practice, this controlled nesting leads to much better scalability and avoids the pitfalls of oversubscription.

In all cases, I pay careful attention to resource management and error handling. Memory mapping is used throughout to minimize I/O overhead and simplify buffer management. Each thread maintains its own compressor state to avoid race conditions. All parameters, including the number of threads, block size, and file size thresholds, are configurable at runtime, allowing for flexible experimentation and adaptation to different workloads. This modular and configurable design ensures that the compressor can be tuned for a wide range of real-world scenarios, from processing millions of tiny files to handling a handful of massive datasets.

2 Benchmarking Methodology

To evaluate the performance of my implementation, I developed a dedicated benchmark driver (`bench_main.cpp`) that allows me to systematically explore the effects of parallelism and configuration parameters. The benchmarking process begins by generating synthetic test data tailored to each scenario: for the "many small files" case, I create thousands of files with random sizes within a user-defined range, while for the "one large file" and "many large files" cases, I generate one or more files of substantial size, again with the option to randomize their exact dimensions. This approach ensures that the benchmarks are not biased by peculiarities of real-world data and that I can control for file size distribution and total data volume.

Each benchmark run is structured to minimize measurement noise and isolate the effects of parallelism. Before timing, I perform a warmup iteration to mitigate cold-start effects such as disk caching and initial memory allocation overhead. I then execute multiple timed iterations, recording the execution time of each. Rather than relying on a single measurement, I compute the median of these runs, which provides a robust estimate of typical performance and reduces the influence of outliers. For each configuration, I report both the absolute execution time and the speedup relative to a single-threaded baseline, allowing for a clear assessment of scalability.

I pay particular attention to the configuration of OpenMP parallelism. For scenarios involving nested parallelism—such as compressing many large files, where both the file loop and the block loop can be parallelized—I explicitly control the number of threads at each level and enable or disable nesting as appropriate. This is crucial to avoid oversubscription, which can degrade performance due to excessive context switching. All relevant parameters, including the number of threads, block size, and file size thresholds, are exposed as command-line options, making it easy to sweep across a wide range of settings and to reproduce results.

Benchmark results are written both to CSV files, for later analysis and plotting, and to standard output for immediate inspection. I also ensure that each benchmark run starts from a clean state by deleting any previously generated compressed files before each iteration. This guarantees that I am always measuring the full cost of compression, including I/O, and not benefiting from residual files or cached results. The driver is designed to fail fast and report errors clearly if any step of the process does not complete successfully, ensuring that only valid results are recorded.

3 Correctness Testing

To guarantee the reliability and robustness of my compressor, I implemented a comprehensive test harness in `test_main.cpp`. My testing strategy is designed to cover not only the typical use cases but also a wide range of edge conditions that could reveal subtle bugs. The test suite begins by generating a set of random files of various sizes, including very small files, large files, and even zero-byte files. I also create nested directory structures to verify that recursive discovery works as intended. For each test, I first perform a compression phase, ensuring that all expected compressed files are created, and then a decompression phase, checking that the original files are faithfully restored.

To verify data integrity, I use both byte-wise comparisons and MD5 checksums. After decompression, I compare each output file to its original counterpart, ensuring that every byte matches. I also generate and compare checksum files before and after the round-trip to catch any discrepancies that might not be detected by simple file existence checks. This dual approach provides strong assurance that the compressor is truly lossless.

I pay special attention to edge cases that are often overlooked. For example, I test the `remove_origin` flag in both compression and decompression modes, verifying that files are deleted only when requested and that no data is lost in the process. I also test the handling of invalid input paths, ensuring that the program fails gracefully and reports errors clearly rather than crashing or producing incorrect output. The threshold for distinguishing between small and large files is overridden in some tests to force all files down the large-file path, ensuring that both code paths are exercised regardless of the actual file sizes. Recursive and non-recursive discovery are both tested to confirm that directory traversal logic is correct.

Throughout the test suite, I use atomic flags and robust error handling to ensure that failures are detected and reported immediately, even in parallel regions. The test harness is designed to be run in both sequential and parallel modes, allowing me to verify that the implementation is correct and thread-safe under all configurations. After each test run, I clean up all generated files and directories to avoid interference between tests and to ensure that each run starts from a known state.

Implementation Details and Justifications

Every design decision in my implementation is motivated by the need for correctness, efficiency, and maintainability. I define the file suffix ".zip" as a constant in `config.hpp` so that all modules—whether handling file discovery, compression, or decompression—use a consistent convention. This avoids subtle bugs where files might be skipped or processed multiple times due to mismatched suffix logic.

The choice of a 16 MiB threshold for distinguishing between small and large files, and a 1 MiB default block size for large-file processing, reflects a balance between the overhead of parallelization and the benefits of increased throughput. These values are not hardcoded: I expose them as configurable parameters so that users and testers can adjust them as needed, and I explicitly override them in certain tests to ensure that all code paths are exercised.

To minimize I/O overhead and simplify resource management, I use POSIX `mmap` for all file access, wrapping it in a simple RAII class that ensures memory is unmapped and file descriptors are closed automatically. This approach reduces the number of system calls and avoids unnecessary data copying, which is especially important for large files. For small files, I prepend a 64-bit header containing the original size, followed by a standard zlib stream. This format is simple, compact, and easy to parse during decompression.

For large files, I designed a custom format that begins with a 32-bit magic number (`0x4D50424C`, corresponding to "MPBL") and a version field, followed by a table of compressed block sizes. This metadata enables random-access decompression and robust format validation. During compression, I allocate per-thread buffers and compressor state to avoid repeated initialization and to ensure that each thread can operate independently. I use OpenMP parallel for loops with dynamic scheduling to distribute blocks among threads, which helps balance the workload even when block sizes vary.

All output files are written via `mmap` to minimize syscall overhead and to ensure that the entire file is updated atomically. During decompression, I read the header and metadata, map both the input and output files, and decompress each block in parallel, checking return codes and output sizes to catch any errors. File discovery is handled by a dedicated module that supports optional recursion, filters files based on mode and suffix, and handles permission errors gracefully.

The benchmark driver is designed to be flexible and robust, supporting sweeps over thread counts and block sizes, and exporting results in a format suitable for further analysis. The test harness exercises every feature and edge case, using atomic flags and immediate error reporting to ensure that failures are never missed. By cleaning up all generated files and directories after each run, I guarantee that tests are isolated and reproducible.

4 Experimental Results