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

Luca Lombardo
SPM Course a.a. 24/25

1 Implementation Strategies

The design of this project is based on the observation that different input scenarios require tailored parallelization strategies to ensure both efficiency and correctness. We identify three main cases: many small files, a single large file, and many large files.

For the case of many small files, we treat each file as an independent unit of work, which naturally lends itself to parallelization. We employ an OpenMP `parallel for` with dynamic scheduling to balance the workload, as file sizes may vary. Each file is memory-mapped using POSIX `mmap`, minimizing I/O overhead. Files smaller than a configurable threshold (16 MiB by default) are processed sequentially to avoid unnecessary parallelization overhead, while larger files are handled by a block-based routine.

When compressing a single large file, we split it into fixed-size blocks (default 1 MiB) and compress each block in parallel, assigning each thread a block and a thread-local Miniz compressor instance for thread safety. The block size is a key parameter: smaller blocks improve load balancing but may reduce the compression ratio and increase metadata. We empirically tune this parameter to balance throughput and effectiveness.

The most complex scenario is many large files, which exposes two levels of parallelism: across files and within each file. Naïvely parallelizing both loops with the same thread count leads to oversubscription. To address this, we implemented three strategies. The first processes files sequentially, compressing each in parallel by blocks - simple, but potentially underutilizing resources. The second parallelizes both loops fully, but risks creating $p \times p$ threads (with p hardware threads), which can be inefficient. The third, and most effective, partitions the thread budget between the two levels: for p threads, we set $t_{\text{out}} = \lceil \sqrt{p} \rceil$ and $t_{\text{in}} = \lfloor p / t_{\text{out}} \rfloor$, ensuring $t_{\text{out}} \times t_{\text{in}} \leq p$. This balances file-level and block-level parallelism without oversubscription, yielding better scalability.

Memory mapping is used consistently to reduce I/O overhead, and each thread maintains its own compressor state to avoid race conditions. All parameters, including thread count, block size, and thresholds, are configurable at runtime, allowing the compressor to adapt efficiently to a wide range of workloads.

2 Benchmarking Methodology

To evaluate the performance of our implementation, we developed a dedicated benchmark driver (`bench_main.cpp`) that allows us 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, we create thousands of files with random sizes within a user-defined range, while for the "one large file" and "many large files" cases, we 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 we 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, we perform a warmup iteration to mitigate cold-start effects such as disk caching and initial memory allocation overhead. We then execute multiple timed iterations, recording the execution time of each. Rather than relying on a single measurement, we compute the median of these runs, which provides a robust estimate of typical performance and reduces the influence of outliers. For each configuration, we report both the absolute execution time and the speedup relative to a single-threaded baseline, allowing for a clear assessment of scalability.

We 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 - we 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. We also ensure that each benchmark run starts from a clean state by deleting any previously generated compressed files before each iteration. This guarantees that we are always measuring the full cost of compression, including I/O, and not benefiting from residual files or cached results.

3 Correctness Testing

We developed a comprehensive test harness (`test_main.cpp`) to ensure program reliability. The suite covers typical use cases and edge conditions. It starts by generating random files of varying sizes (including zero-byte) and nested directories to test recursive discovery. Each test involves compression, verifying output file creation, followed by decompression, ensuring faithful restoration of original files.

Data integrity is verified using byte-wise comparisons and MD5 checksums between original and decompressed files. This dual approach confirms the compressor is lossless. Edge cases receive special attention. We test the `remove_origin` flag in both modes, handling of invalid paths, and override the small/large file threshold to exercise both processing paths. Recursive and non-recursive discovery are also explicitly tested.

Atomic flags and robust error handling ensure failures are caught immediately, even in parallel execution. The harness runs sequentially and in parallel to verify thread safety.

Implementation Details and Justifications

We centralize the ".zip" suffix in `config.hpp` to ensure consistency. The 16 MiB small-file threshold and 1 MiB block size balance parallel overhead and compression ratio; both are configurable and overridable in tests. All file I/O uses a lightweight RAII wrapper around POSIX `mmap` to minimize syscalls and data copies.

Small files start with a 64-bit size header followed by a standard zlib stream. Large files use a custom layout: a 32-bit magic number (0x4D50424C "MPBL"), a version field, and a table of compressed block sizes for random access. Each thread has its own compressor state and buffer. We employ OpenMP `parallel for` with dynamic scheduling to compress and decompress blocks in parallel, checking return codes to catch errors.

Output files are updated atomically via `mmap`. Decompression reads headers and metadata, maps input/output, and runs parallel block decompression. File discovery handles recursion, filters by mode and suffix, and reports permission errors gracefully.

The benchmark driver supports parameter sweeps over threads and block sizes, writes CSV-ready results, and cleans up between runs for consistency. The test harness exercises all paths and edge cases, uses atomic flags for thread safety, reports failures immediately, and removes generated files to guarantee isolation and reproducibility.

4 Experimental Results