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

Luca Lombardo
SPM Course a.a. 24/25

1 Implementation strategies

1.1 Many Small Files

Compressing a large set of small files naturally lends itself to parallel execution since each file can be treated as an independent unit of work. In our implementation we employ an OpenMP parallel for loop with dynamic scheduling to assign each file to a separate thread, accommodating disparities in individual file sizes and ensuring a balanced distribution of tasks across available cores. Before invoking the Miniz compression routine, every file is memory-mapped, which significantly reduces system call overhead and guarantees contiguous in-memory access. This optimization lowers latency and increases throughput, although for extremely small files the per-file initialization costs and shared I/O contention may diminish the returns of fine-grained parallelism.

Memory mapping is performed via the POSIX `mmap` system call, which creates a direct mapping between the file's contents and the process's virtual address space. This strategy obviates explicit read calls and extra buffer copies, allowing the compressor to treat file data as a simple pointer array. While the first access to each memory page may trigger a page fault, the amortized cost is lower than repeated I/O operations, particularly for large contiguous regions.

To distinguish between small and large inputs, we selected a threshold of 16 MiB. Files below this limit are processed by a single-threaded routine to avoid the overhead of thread management, whereas larger files are divided into parallel blocks to exploit multicore throughput. Empirically, 16 MiB represents a compromise point at which block-level parallelism begins to outweigh its setup costs without incurring excessive memory pressure.

In the custom format for large files, the first four bytes encode the magic number `0x4D50424C`, corresponding to the ASCII string "MPBL". This marker enables instant recognition of our block format and guards against accidental or malicious input of unsupported data.

1.2 One Large File

When handling a single, monolithic input, we divide the file into configurable, fixed-size blocks to expose parallelism at the segment level. Each thread reads its assigned block directly from a memory-mapped region, applies Miniz compression using a thread-local compressor instance to avoid repeated state initialization, and records both compressed output and block metadata. Smaller block sizes permit finer load balancing and higher concurrency but introduce extra metadata overhead and may fragment compression context; larger blocks preserve context and can improve compression ratios, yet they reduce the degree of parallelism. We tested various block sizes to find the optimal trade-off between compression efficiency and parallel throughput.

1.3 Many Large Files

For multiple large files, we exploit two orthogonal forms of parallelism: across files and within each file. We evaluated three dispatch strategies. The first processes files in a sequential outer loop while compressing blocks in parallel, which prevents thread oversubscription but fails to utilize all cores when the file count is lower than the core count. The second naively parallelizes both the outer file loop and the inner block loops over the full thread budget, often resulting in $p \times p$ threads, excessive context switching, and degraded performance. The third, and preferred, approach partitions the total thread budget p into t_{out} threads for file-level dispatch and t_{in} threads for block-level work under the constraint $t_{\text{out}} \times t_{\text{in}} \leq p$. In practice we select $t_{\text{out}} = \lceil \sqrt{p} \rceil$ and $t_{\text{in}} = \lfloor p / t_{\text{out}} \rfloor$, a simple heuristic that balances inter-file throughput and intra-file concurrency without oversubscribing CPU resources.

2 Benchmarking Methodology

The driver in `bench_main.cpp` parses command-line options, generates test data (many small or large files), and measures median execution time using warmup runs. For each scenario (many small, one large, many large sequential, oversubscribed nesting, controlled nesting), we adjust OpenMP settings (nested parallelism, thread counts) and record results in CSV.

3 Correctness Testing

The test harness in `test_main.cpp` sets up a controlled environment with random files, then executes compression and decompression in both sequential and parallel modes. It verifies:

- Round-trip integrity via byte-wise comparison and MD5 checksums.
- Edge cases: zero-length files, remove_origin flag, invalid paths, threshold override, recursion on subdirectories.

4 Experimental Results