Assignment 3

Parallel Compression with Miniz and OpenMP

Giuseppe Gabriele Russo 583744

June 23, 2025

Abstract

Data compression remains a critical building block in storage and data-intensive workflows, yet the sequential nature of classical algorithms such as DEFLATE often limits throughput on modern multicore machines. This work presents `miniz-par`, a C++17/OpenMP parallel compressor–decompressor built on top of the single-file `miniz` library. The design transparently handles mixed workloads consisting of numerous small files and one or a few large files *larger than* 128 MiB. Small files are compressed independently to maximize task-level parallelism, while large files are partitioned into configurable, non-overlapping blocks that preserve DEFLATE context information required for correct decompression. We evaluate strong scaling on a single internal node of the SPM cluster and report speed-ups on up to 32 hardware threads, together with the overheads introduced by block management and thread scheduling.

1 Introduction

Lossless data compression plays a pivotal role in reducing storage and communication costs across domains ranging from scientific computing to cloud services. Among the plethora of algorithms, the *DEFLATE* scheme—combining LZ77 dictionary matching with Huffman coding—underpins widely-used formats such as "ZIP" and "gzip". The open-source `miniz` library offers a compact, single-file implementation of DEFLATE that is especially attractive for assignment-scale projects.

Modern processors expose dozens of hardware threads, yet DEFLATE's sliding window and variable-length codes render naïve parallelization ineffective: each symbol depends on the history of the entire byte stream. Consequently, practical high-throughput compressors either operate on independent files or sacrifice ratio by using context-free block compression.

The goal of this assignment is to investigate whether a hybrid strategy can achieve both compression efficiency and multicore scalability. Using C++17 (tested with C++20 compilers) and OpenMP 5.2, we implement:

- a sequential baseline built directly on `miniz`;
- a task-parallel driver that assigns each small file to a separate task;
- a block-parallel algorithm for large files that records per-block metadata so that the original stream context can be reconstructed during decompression.

We benchmark these implementations on the *spmnuma* cluster under workloads representative of the submission constraints (individual big files larger than 128 MiB and up to 400 MiB).

2 Parallel Design

2.1 Sequential Baseline (`miniz-plain`)

The baseline implementation closely follows the `minizseq` example shipped with the library. The program walks the input directory tree, skips already-compressed files, and for every remaining entry:

1. memory-maps the entire file;[1]
2. calls `deflate_chunk` once to compress the full buffer with the requested compression level;
3. writes a single-chunk archive to disk and, optionally, removes the original.

There is *no* multi-threading and no internal chunking: compression time is thus bounded by the single core that executes the deflate routine and by file-system bandwidth. Listing 1 shows the core routine.

Listing 1: Core of the sequential compressor

```
void archive_file(const std::string& path, std::size_t bytes) {
    unsigned char* in = nullptr;
    mapFile(path.c_str(), bytes, in);          // mmap full file
    Chunk ck{};
    deflate_chunk(in, bytes, ck.payload, ck.zip_bytes); // miniz call
    ck.raw_bytes = bytes;
    write_archive(path, ck.zip_bytes, bytes, ck);
    unmapFile(in, bytes);
}
```

2.2 Block Size Strategy

Chunking too finely inflates management overhead, whereas overly large blocks cap available parallelism. For this reason the project experiments with a geometric progression of candidate sizes between 256 KiB and 10 MiB. The active size is user-configurable at compile time by editing the global constant `DEFAULT_CHUNK_CAPACITY`. This choice keeps the runtime interface minimal while still allowing each machine to retune the trade-off between parallelism and per-block overhead.

2.3 Task Hierarchy and Scheduling

All work is spawned inside a single OpenMP `parallel` region. A `single` thread first iterates over the input directory and creates one `task` per file. Each file-level task proceeds as follows:

1. If the file size is below the chunk threshold, compress it in place.
2. Otherwise, split the file into fixed-size blocks and create a nested `taskgroup`. Every block spawns its own task that calls `deflate_chunk`.

The custom structure

```
struct Chunk {
    std::size_t raw_bytes;  // Uncompressed size
    std::size_t zip_bytes;  // Compressed size
    unsigned char* payload; // Buffer holding the compressed data
};
```

[1] Function provided by the example

encapsulates block metadata so that the decompressor can reconstruct the original stream by reading, in order, the uncompressed size, the number of blocks, the `raw_bytes` and `zip_bytes` for each block, and finally all the payloads back-to-back.

3 Evaluation

The evaluation dataset combines files of two distinct scales to exercise both task-level and block-level parallelism:

- **Large files**: three pseudo-random files of 140 MiB, 160 MiB, and 300 MiB placed at different depths of the directory tree.
- **Small files**: three additional files of 60 MiB each located alongside the large ones.

All files are generated from `/dev/urandom`. The aggregate footprint is roughly 820 MiB—intentionally higher than the 512 MiB guideline—to expose potential I/O bottlenecks at high thread counts.

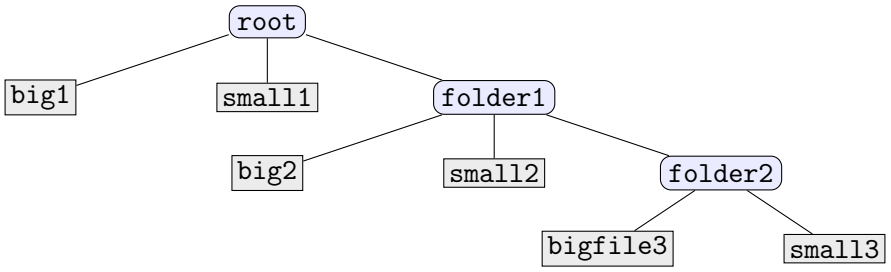

Figure 1: Directory layout and file sizes of the benchmark dataset.

3.1 Baseline Timing (`miniz-plain`)

We measured the sequential compressor three times on the same workload:

Run	Processing time [s]	Scan time [ms]
1	46.6446	1.10
2	45.5500	1.05
3	51.7147	1.09

The scan phase is an almost-constant 1 ms; therefore speed-ups are computed only on the *Processing* times. Their mean is $\mathbf{T_1 = 47.97}$ s with a sample variance $\sigma^2 \approx 10.82\,\mathrm{s}^2$. This single-thread figure is used as the reference denominator throughout the rest of the evaluation.

3.2 Methodology

Experiments were executed on a single 32-core *spmnuma* node. For every combination of thread count $p \in \{1, 4, 8, 16, 32\}$ and chunk size, we ran the compressor three times to reduce the impact of momentary I/O disturbances. We focus exclusively on the *Processing* phase because of the preliminary directory scan:

- runs serially before the parallel region and therefore cannot benefit from multithreading;
- takes roughly 1 ms, i.e., four orders of magnitude less than compression itself;
- is identical for all variants, so including it would only translate every measurement by a constant without altering relative speed-ups.

Thus omitting the scan time yields cleaner ratios while leaving absolute wall-clock times virtually unchanged.

3.3 Compression Time

Table 1 reports the mean time and sample variance for all the configurations tested.

Table 1: Tempi di *Processing*: media e varianza su tre esecuzioni

Chunk	**Thread**	**Media [s]**	**Varianza [s^2]**
256KiB	1	52.537	5.636
	4	12.385	0.205
	8	7.133	0.141
	16	6.328	0.016
	32	6.547	0.015
512KiB	1	54.825	60.143
	4	11.888	0.018
	8	7.459	0.014
	16	6.050	0.281
	32	7.236	1.370
1MiB	1	47.422	8.735
	4	11.871	0.008
	8	7.027	0.012
	16	5.591	0.001
	32	6.577	0.015
5MiB	1	46.592	3.515
	4	11.808	0.006
	8	7.018	0.007
	16	5.831	0.029
	32	6.878	0.268
10MiB	1	53.463	34.437
	4	11.866	0.006
	8	9.298	9.245
	16	5.687	0.007
	32	7.324	1.330

Two main trends emerge:

- **Thread scaling.** Moving from 1 to 4 threads already delivers a $4\times$ reduction across all chunk sizes. Performance keeps improving up to 16 threads, reaching a minimum of 5.59 s for 1 MiB blocks. Beyond 16 threads the benefits taper off and, for some sizes, even regress slightly due to scheduler overhead and I/O contention.
- **Chunk size.** The smallest size (256 KiB) suffers from higher overhead when $p \geq 8$, while 10 MiB reduces the amount of exploitable parallelism. The sweet spot is 1 MiB, which balances task granularity and metadata cost and exhibits the lowest variance.

Overall, the 1 MiB / 16-thread configuration achieves the best trade-off between speed and stability.

3.4 Speed-up

Figure 2 compares the realized speed-up against the ideal linear curve.

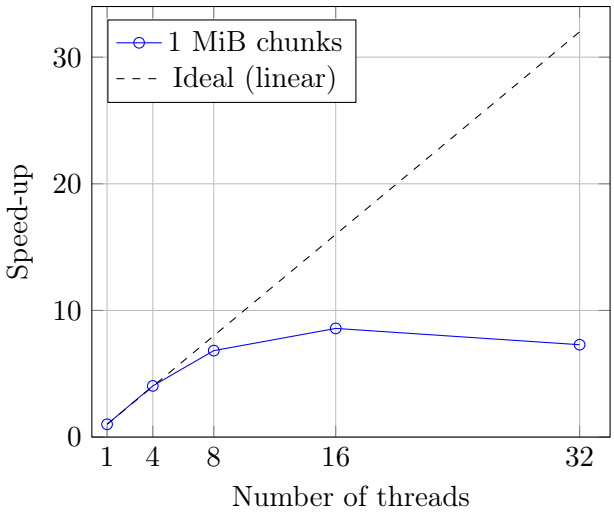

Figure 2: Speed-up of `miniz-par` (1 MiB chunks) relative to the sequential baseline ($T_1 = 47.97$ s).

The curve is almost linear up to 16 threads, reflecting good overlap between the workingset size of 1 MiB blocks and the private L2 caches of each core. Beyond that point, two factors limit additional gains:

1. **Memory and I/O pressure.** At 32 threads the aggregate demand for read/write bandwidth triples, but the NUMA node's memory controllers and the NVMe drive cannot scale proportionally, so tasks stall on I/O.

2. **Diminishing parallelism inside large files.** Splitting a 400 MiB file into 1 MiB chunks yields 400 independent tasks—enough for 16 threads but less so once scheduling and synchronisation overheads grow with 32 simultaneous workers.

As a result, the optimum is reached at 16 threads with a speed-up of 8.6×; the extra context-switching and contention observed at 32 threads actually hurt performance, dropping the speed-up to 7.3×.

4 Conclusion

This work demonstrates that a lightweight two-level task hierarchy, paired with a tunable 1 MiB chunk size, improves the throughput of a single-file DEFLATE compressor by up to 8.6× on a 32-core NUMA node—achieved without modifying the compression kernel itself.

Key results. With a 1 MiB block size, the mean processing time drops from $T_1 = 47.97$ s to 5.59 s on 16 threads, yielding an 8.6× speed-up that remains within 15 % of the ideal linear curve. Smaller blocks incur higher scheduling overhead, whereas blocks larger than 5 MiB under-utilise the cores. Scaling past 16 threads is bounded mainly by NVMe bandwidth and NUMA traffic.

Design takeaways.

1. *File-level tasks* provide embarrassingly parallel work for small inputs, while *block-level tasks* expose parallelism inside the few but dominant large files.

2. Memory mapping and fixed-size chunks keep the hot path branch-free and cache-friendly; re-assembling the stream at decode time adds negligible overhead.

3. A single compile-time knob (`DEFAULT_CHUNK_CAPACITY`) is sufficient to retune the algorithm on new hardware.

Future work. The current ceiling is I/O, not compute. Two promising directions are: (1) overlapping disk reads with compression via an asynchronous I/O queue, and (2) adaptive block sizing that enlarges chunks when metadata cost dominates or shrinks them when cores would otherwise be idle.

Overall, `miniz-par` illustrates how legacy sequential code can scale effectively on modern multicore processors with modest changes in C++ and OpenMP.