Department of Computer Science
Computer Science Master Degree

Third Assignment - Deflate

SPM Course 2024/2025

Github repo

Student: Luca Rizzo

ID Number: 598992

The objective of this assignment was to implement a parallel version of a file compressor and decompressor, operating on a set of input files provided via command-line arguments. Compression is performed using the DEFLATE algorithm, which combines LZ77 compression followed by Huffman coding to encode repeated sequences of symbols efficiently.

It is important to highlight that all tests were carried out using randomly generated files. Due to their high entropy, such files are inherently difficult to compress effectively, resulting in poor compression ratios regardless of the approach used. Additionally, no detailed evaluation was conducted regarding the actual compression efficiency of the sequential versus the parallel implementation. In particular, the block-based strategy adopted in the parallel version may lead to a potential loss in compression effectiveness due to the lack of global context, an aspect that DEFLATE relies on for optimal results.
All compression was performed using the default compression level, in line with the behavior of the provided sequential implementation. Further analysis could be conducted in the future to explore how different compression levels affect the performance and effectiveness of the two approaches, but this was not done here due to time constraints.

Compression specification

To enable block-based compression, the input file is divided into a number of chunks equal to the maximum number of threads specified by the `OMP_NUM_THREADS` environment variable during compression. Each thread is responsible for independently compressing its assigned block.
Since the DEFLATE algorithm requires knowledge of each block's compressed size for correct decompression, this information must be stored alongside the compressed data. However, given that all blocks (except for the last one) share the same uncompressed size, redundant metadata is avoided by storing the following values once in the file header:

- `num_blocks`: the total number of blocks,
- `block_size`: the uncompressed size of all blocks except the last,
- `last_block_size`: the uncompressed size of the final block.

Immediately following this metadata, the header includes a list of `num_blocks` values representing the compressed size of each block.

To simplify both compression and decompression logic, a `MIN_CHUNK_SIZE` threshold is defined. Files smaller than this threshold are compressed as a single block, yet still follow the same header format. This choice avoids the need for two separate processing flows and the associated overhead of storing an additional flag to distinguish between single-block and multi-block compression modes. Although this introduces a **small fixed overhead**, it significantly streamlines the overall implementation.

The space overhead introduced by the parallel compression header amounts to:

```
(3 + num_blocks) × sizeof(size_t)
```

Assuming a maximum of 40 threads, the worst-case overhead is:

```
(3 + 40) × 8 bytes = 344 bytes
```

This overhead is negligible when compared to typical input file sizes and is justified by the increased parallel efficiency and simplification of the decompression process.

Implementation description

Compression

The program initially identifies all the files to be compressed, based on the `-r` flag. If this flag is set, directories are explored recursively. Once the file list is established, a task is created for each input file using an OpenMP `taskloop`.

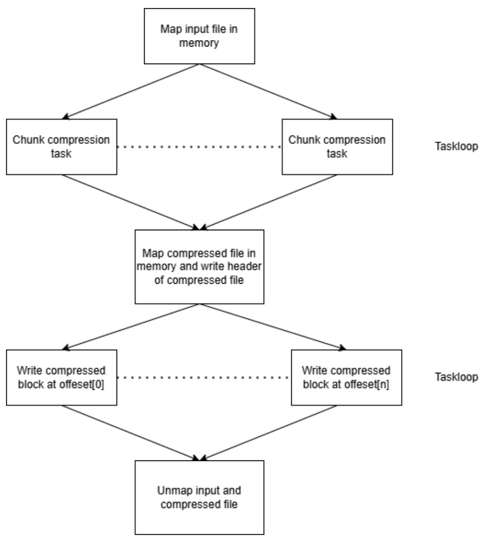
Each task, when executed, splits the file into a number of chunks equal to the number of threads available (as defined by the `OMP_NUM_THREADS` environment variable). These chunks are then compressed in parallel by creating additional tasks for each block.

Each compression task stores its output in a shared `std::vector` of `CompBlockInfo` structures. These structures hold pointers to the compressed blocks, along with their original and compressed sizes. Despite the shared nature of the vector, no synchronization mechanisms are necessary, as each task writes to a distinct index that is known in advance.
Any error encountered during compression is propagated via a parallel `reduction(|:any_error)` using a shared `any_error` variable, which allows the main thread to determine whether or not to proceed with the file writing phase.

Once all chunks have been compressed, the thread that launched the file-level task proceeds to write the result. First, the output file is memory-mapped, and the header (as previously described) is written. Then, a `taskloop` assigns threads to write the compressed chunks into their respective positions using precomputed offsets. This two-phase approach, (first compressing, then writing) is necessary because compressed blocks can vary in size. As such, threads cannot know their write positions in advance. A sequential step is therefore needed to compute the offsets after compression and before writing.

Once all blocks have been written (ensured by the implicit synchronization of `taskloop`), the memory-mapped file is unmapped by the same thread that initiated the process.
An alternative writing strategy was also implemented for testing purposes, activated using an additional `-s` flag. In this version, the main thread writes all blocks sequentially using an `std::ofstream`, without memory mapping. Performance measurements revealed that, due to the use of a Network File System (NFS), the memory-mapped version is significantly more efficient, likely because the final `munmap` flushes all memory pages in a single operation, reducing the number of I/O transactions compared to the sequential writes.
A third variant was tested using parallel `pwrite` calls. However, this approach resulted in poor performance, likely due to contention on the remote file system caused by concurrent disk writes from multiple threads.

Additionally, for files that fit in a single block, compression and writing are performed sequentially by the same thread using `if (num_blocks > 1) clause` in taskloop directive, avoiding unnecessary task creation and reducing overhead. The overall flow of computation is illustrated in a diagram that follows the Work-Span model, highlighting the sequential and parallel phases involved in the compression processes.


Map input file in memory
Chunk compression task
Chunk compression task
Taskloop
Map compressed file in memory and write header of compressed file
Write compressed block at offeset[0]
Write compressed block at offeset[n]
Taskloop
Unmap input and compressed file

Decompression

The decompression phase mirrors the compression logic, relying on the metadata stored in the compressed file's header. Similarly to compression, a task is created for each file to be decompressed. The thread that picks up a task acts as the file's main thread and first reads the header to retrieve the number of blocks and compute the compressed data offsets. This operation is performed sequentially.

A `taskloop` is then launched, assigning each block to a thread. Each thread decompresses its assigned chunk and stores the result in a shared vector for subsequent use.
The output file is memory-mapped using a size inferred from the header, and the decompressed data is written directly to the mapped region by each thread during the decompression step. This optimization is possible because, unlike compression, the decompression phase operates on blocks with known output sizes that are independent of the results of other threads. As each block is written to a fixed, pre-calculated position in the output file, threads can safely perform parallel writes. This eliminates the need for an additional sequential write phase and leads to significantly improved performance, especially when handling large files or a high number of blocks.

Challenges and Solutions

One of the main challenges encountered during development was the performance degradation caused by the use of a Network File System (NFS), particularly during the write phase of the compression process. Since the compute nodes of the target system do not have local storage, all file I/O operations rely on remote disk access over the network, making write efficiency a critical concern.
To address this, several strategies were explored:

- **Memory-mapped output (`mmap + munmap`):**
  The primary solution adopted involved memory-mapping the output file and allowing each compression task to write its compressed block directly into the mapped memory region. This approach minimizes explicit I/O calls and takes advantage of the operating system's page cache. At the end of the compression phase, a single `munmap` flushes all memory pages to disk. Performance analysis (see later) showed that this method yields the best performance, likely due to reduced network traffic.

- **Sequential write via stream (`-s` flag):**
  As a baseline comparison, a version was implemented where the main thread performs all writes sequentially using `std::ofstream` and standard file output. While simpler, this approach resulted in significant performance penalties when compressing large files or many blocks, as each write incurs an immediate I/O operation, saturating the network link with small, sequential requests.

- **Parallel `pwrite` alternative:**
  A further variant was also tested using `pwrite` calls to allow each thread to write its compressed block directly to disk in parallel. However, in practice this approach did not lead to the expected performance improvements. It is suspected that concurrent low-level writes from multiple threads may have caused increased contention on the NFS, potentially resulting in higher latency and reduced throughput.

Correctness analysis

To validate the correctness of the implementations and support regression testing across code changes, two simple test scripts are provided in the `tests` folder. Both scripts verify the integrity of the compression and decompression steps by checking whether the file obtained after the full compress-decompress pipeline is identical to the original input.
The first script, `simple_test.sh`, performs this check using a small text file (`utility.hpp`) as input. The second, `complex_test.sh`, creates two random files of 50 MB each and verifies that the round-trip transformation preserves the original content exactly.

Performance analysis

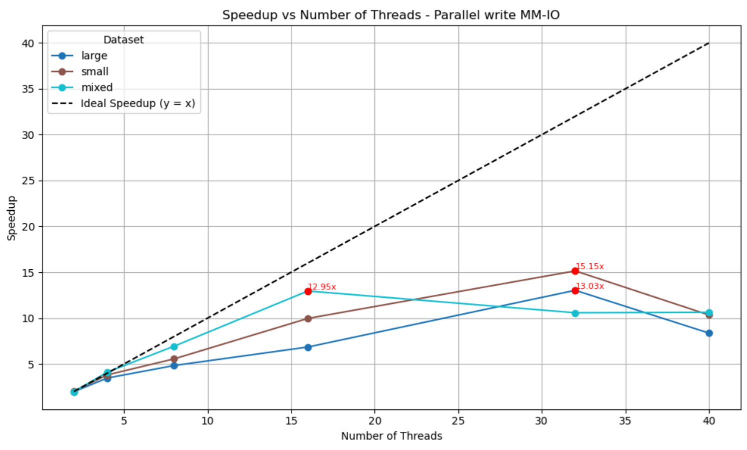
To evaluate strong speedup, three datasets were created:

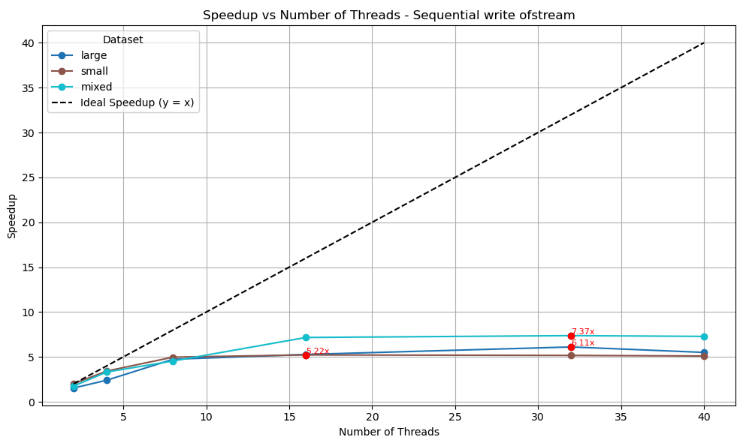
- **Large**, consisting of three individual files of 100 MB each;
- **Small**, composed of many small files ranging from 512 KB to 2 MB, organized recursively to reach a total of 100 MB;
- **Mixed**, which includes files of varying sizes from 256 KB up to 80 MB, also totaling around 100 MB.

For each dataset, the parallel version of the compressor and successive decompression was executed three times at different thread counts (2, 4, 8, 16, 32, and 40), using both memory-mapped and sequential write strategies. Finally, the sequential implementation was also run as a baseline. All execution times were averaged over three runs to ensure stable measurements. The benchmark was submitted using `sbatch ./submit_batch`, which in turn runs the `run_benchmark.sh` script on node 06 of the cluster.

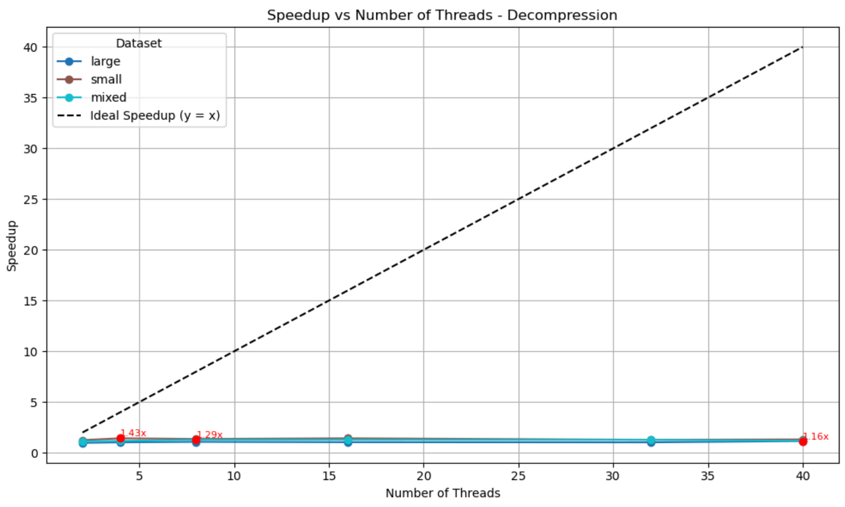
The compression process in this application is primarily I/O-bound, meaning that its performance is heavily influenced by the efficiency of disk and network operations, rather than purely by computational throughput. This characteristic becomes particularly critical when running on a system where file I/O relies on a Network File System (NFS), as is the case with the SPM cluster.

When using sequential write via `std::ofstream`, all output is handled by the main thread, which writes each compressed block one after the other. This approach generates a large number of small, synchronous write operations over the network. As a result, the network becomes the primary bottleneck, severely limiting the benefits of parallel compression. This is evident in the first plot, where the speedup plateaus early and fails to scale beyond modest gains, even with 32 or 40 threads—reaching a maximum of around 6,4× for the large dataset, about 5,22× for the small dataset, and just under 7,37× in the mixed case.
By contrast, the memory-mapped I/O approach offers a more scalable alternative. In this mode, each thread writes a previously compressed data block (most likely produced by a different thread during the compression phase) directly into a mapped memory region, corresponding to its offset in the output file. These writes are kept in memory and flushed collectively during the `munmap` call, which **probably** results in a single large I/O operation. The second plot demonstrates the clear performance advantage of this strategy: speedup increases more consistently with the number of threads, reaching approximately 13× for the large dataset, around 15× for the small dataset, and up to 13× in the mixed case.
Overall, this comparison highlights how crucial the choice of output strategy is in I/O-bound, parallel workloads on NFS systems. **Reducing the frequency and fragmentation of write operations** is key to unlocking better strong scalability in such environments.





Decompression speedups appear to be limited, likely because the process is heavily influenced by memory and file system bandwidth rather than computation time. As a result, the main cost lies in transferring data from memory rather than in actual decompression work, which reduces the benefits of parallel execution.


Speedup vs Number of Threads - Decompression
Dataset
large
small
mixed
Ideal Speedup (y = x)
Speedup
Number of Threads
1.43x
1.29x
1.16x