Report Assignment3 - Parallel Miniz Implementation

Angelo Nardone

1 Introduction

This report concerns the third assignment of the SPM course, which focuses on the **compression** and **decompression** of text files using the `miniz algorithm`. The main challenge of this assignment was to parallelize the sequential code provided by the professor, specifically by using **OpenMP**.

This report presents the implementation of a parallel version of the provided sequential code using OpenMP, outlining its structure and the rationale behind the design decisions. It then describes the datasets used for testing. The following sections analyze the experiments conducted and the results obtained, with particular attention to whether they align with the initial expectations. The report concludes by discussing the main challenges encountered during the assignment and proposing possible improvements.

2 Code Implementations

In this section is described the **implementation of the code** used for compression and decompression with the `miniz` library. Particular attention is devoted to the parallel version, as the sequential code remained unchanged from the version provided by the professor.

Before examining the implementation details, it is useful to briefly explain how the **code is executed**, in order to clarify its expected behavior and to justify certain design choices. The executable is invoked as follows:

```
./executable -r 0 -C 0 [-q 1] file1 [file2 .... filek]
./executable -r 0 -D 1 [-q 2] file1 [file2 .... filek]
```

The meaning of the flags is as follows:

- `-r` indicates whether, in the case where a directory is passed as `file`, the program should **recursively explore** its subdirectories (`-r 1`), or remain at the top level without descending into nested folders (`-r 0`).
- `-C` specifies that the input files should be **compressed**. When followed by 0, the original uncompressed files are preserved; when followed by 1, they are deleted after compression.
- `-D` specifies that the input files should be **decompressed**. This flag is mutually exclusive with `-C`. Similarly, when followed by 0, the original `.zip` files are preserved; when followed by 1, they are deleted after decompression.
- `-q` is an optional flag that controls the **verbosity level** of the program. A value of 0 suppresses all output; 1 enables basic output, including error messages; 2 enables detailed output, such as notifications about files already compressed and therefore skipped. The default is `-q 1`.
- `file` represents one or more **files or directories** to be processed. The first file or directory is mandatory, while the rest are optional.

2.1 Sequential Code: `minizseq`

Let's begin by briefly discussing how the sequential code `minizseq` works. The execution begins by reading the **command-line flags** using the `parseCommandLine()` function, which also reports any errors in the invocation of the executable. Then, for each input file, the program checks whether it is a **regular file** or a **directory** to be compressed or decompressed.

- If the item is a file suitable for processing, the `doWork()` function is called. This function maps the file into memory using `mapFile()`, compresses or decompresses the data using `compressData` or `decompressData` respectively, and finally unmaps the memory with `unmapFile()`.

- If the item is a directory, the `walkDir()` function is invoked. This function inspects the contents of the directory, either **recursively or non-recursively** depending on the `-r` flag. As soon as a suitable file is found, `doWork()` is called to process it. Additionally, the `discardIt()` function is used to skip files that are already compressed (i.e., those with the `.zip` suffix) or files that are not in a format suitable for decompression.

During execution, a **boolean variable** `success` is maintained to track whether all operations complete successfully. In the case of an error, an error message is printed and the program exits with return code `-1`. Otherwise, it prints `"Exiting with Success"` upon completion.

2.2 Parallel Code: `minizpar`

The following section presents the functionality of the **parallel code**. Before describing the implementation, it is appropriate to provide some context. In accordance with the project guidelines, greater emphasis has been placed on the compression phase, which will be the only one analyzed in detail during the experimental evaluation (see Section 4). Although the decompression functionality also yielded satisfactory results, it must be noted that less time was dedicated to its development. The code was designed primarily with **compression optimization** in mind, rather than decompression.

Another important consideration is that the parallel version of the code does not merely replicate the behavior of the sequential version with added parallelism. Instead, an additional requirement has been introduced: any file exceeding a given **size threshold** (set to 1MB in this implementation) must be **split into smaller chunks**. Each chunk is **compressed independently**, and the compressed results are then merged into a single `.zip` file.

Observation 1. This splitting process inherently affects the structure of the resulting compressed file. Specifically, the **file header** must now include additional metadata, indicating the number of chunks the original file was divided into. This information is essential, as each chunk is compressed with an independent **context window** and therefore must be **decompressed separately**. While this approach can improve performance by enabling concurrent processing of chunks, it also introduces additional complexity in both the implementation and decompression phases.

The modified version of the original code provided by the professor introduces changes primarily in two areas: the addition of two functions in `utility.hpp`, namely `compressParallelFile` and `decompressParallelFile`, and the creation of the `minizpar` file. The latter can be logically divided into four main parts:

1. In the first part, similarly to `minizseq`, the `parseCommandLine()` function is used to read the **command-line flags**.
2. The second part is still executed sequentially. It analyzes the arguments passed to the program and constructs a `fileList` containing all **valid files to be processed** (compressed or decompressed), excluding those that do not meet the criteria (e.g., already compressed or uncompressed, depending on the selected mode). If an argument is a directory, the code scans its contents: in recursive mode (`RECUR`), subdirectories are also explored; otherwise, only the main directory is considered. If the argument is a single file, it is verified directly. Files that are excluded from processing are reported when a sufficient verbosity level is enabled (`QUITE_MODE` $\geq$ `2`). If no valid files are found, an error message is printed and the program exits.
3. The third part constitutes the **core of the parallel implementation**. This section employs **OpenMP** to parallelize the compression or decompression of the files in `fileList`. A fixed number of threads is set, nested parallelism is enabled, and two levels of parallel activation are allowed to support both external and internal parallelism (i.e., parallelism within `compressParallelFile` or `decompressParallelFile`). A parallel region is then created, within which a `single` section generates an **OpenMP task** for each file in `fileList`. Each task uses `firstprivate(f)` to ensure that each thread works on an independent copy of the file name, thereby avoiding **race conditions**, while `shared(success)` allows for coordinated updates of the global success variable. Each task invokes either `compressParallelFile` or `decompressParallelFile`, depending on the selected mode (`COMP`). The global `success` variable is **updated atomically** to reflect the overall status of the operation. A `taskwait` directive is used to ensure that the program waits for the completion of all tasks before proceeding.
4. The final part mirrors the behavior of `minizseq`: the `success` variable is checked. If it is `true`, a success message is printed; otherwise, an error message is displayed.

Observation 2. The key difference between `minizpar` and `minizseq` lies in the separation of phases 2 and 3 described above. In `minizseq`, these phases are executed simultaneously. At first glance, splitting them may appear inefficient. However, the rationale behind this design choice is as follows. The **cost of scanning** directories and files is minimal and consistently represents less than 0.1% of the total runtime (often much less in practice). Performing this task sequentially allows for the construction of a list of files to be processed in parallel, a simple task. In fact, attempting to initiate threads from the start, especially in the presence of deeply nested directory structures, may result in some threads remaining idle or underutilized due to long scanning times. The chosen design therefore ensures a safe and practical approach that not only avoids performance degradation but may also yield performance gains. Notably, an earlier version of the code attempted to parallelize the two phases as done in `minizseq`, but it was found to be slightly less efficient than the current design.

At this point, the implementation of the `compressParallelFile` function can be explained. This function is a **boolean function** responsible for the parallel compression of a single file by dividing it into 1MB chunks, each compressed independently using OpenMP. Its operation closely resembles that of `doWork()`.

1. Initially, the file is **memory-mapped** using the `mapFile()` function, enabling direct access to the bytes without the need for intermediate copies.
2. Once the total file size is obtained, the **number of chunks** required is calculated using the following formula:

$$\texttt{n\_chunks} = \frac{\texttt{size}}{\texttt{CHUNK\_SIZE}}$$

   where `CHUNK_SIZE=1MB`. In this manner, for files larger than 1MB, more than one chunk will be used. For each chunk, space is allocated in two vectors: one for pointers to the **compressed buffers** (`out_ptr`), and one for the **sizes of the compressed data** (`out_sz`).
3. At this point **parallelism comes into play** within an OpenMP `taskgroup`, where a `task` is launched for each chunk. For each task, the variables `firstprivate(i, size, fname, ptr)` are defined, where `i` is the **chunk index**, `size` is the **file size**, `fname` is the **file name**, and `ptr` is the **pointer to the memory-mapped** file. Additionally, two **shared variables**, `shared(out_ptr, out_sz)`, are used. Each task then determines the boundaries of its respective chunk, allocates a temporary buffer, and uses the `compress` (**miniz**) function to compress the corresponding data portion. The results (pointer and size of the compressed chunk) are stored in the respective shared vectors.
4. After the compression of all the chunks is completed, an output file is opened with the same name as the original file, appended with a suffix (`SUFFIX=.zip`). The **file header** is written, containing the number of chunks and the original file size. Then, the **compressed chunks are written sequentially**: for each chunk, its size is written first, followed by the actual compressed data. It is important to note that the file is written sequentially at this stage to avoid potential **race conditions**, and because writing is generally not resource-intensive, working sequentially allows for better flow control.
5. After the writing is completed, the file is closed, memory is freed, and the original file is optionally removed (if `REMOVE_ORIGIN` is active). Finally, the function returns `true` to indicate that the compression was successful.

Observation 3. As previously mentioned, `compressParallelFile` works in a manner very similar to `doWork()`. The difference lies in the division of the file to be compressed into chunks, processing these chunks in parallel, and adjusting the file writing process to account for multiple chunks, specifically by modifying the header section.

In conclusion, the **decompression process** follows the inverse of compression, in a manner quite similar to the sequential approach, with the distinction that in the case of multiple chunks, these are decompressed in parallel.

3 Datasets

Having explained how the code operates, the first step in the performance analysis is to identify the type of **data** on which the evaluation will be conducted. For this reason, **artificial data** was generated using the command suggested in the assignment:

```
dd if=/dev/urandom of=file.dat bs=size count=repetition
```

where, naturally, `size` and `repetition` were set to obtain the desired file size. A preliminary definition was then introduced:

Definition 1 (Large files and small files)**.** Files larger than 1MB are considered **large file**, while files with size less than or equal to 1MB are considered **small file**.

This definition is motivated by the behavior of the parallel algorithm: large files are those that will be split into **multiple subfiles during compression**, whereas small files will not be divided in the parallel compression process. The goal is to evaluate how the algorithm performs under these two distinct scenarios.

At this point, to evaluate the code and its properties, three datasets were prepared. Accordingly, three folders were created: `big_files`, `small_files`, and `nested_files`. Each of these folders contains a total of **400MB of data**, as shown in Code 1, but they were constructed with a fundamental difference:

- `big_files`: consists of only **7 large files**, with sizes ranging from 5MB to 200MB.
- `small_files`: consists of **770 small files**, each with a size ranging from 100KB to 1MB. These files were generated randomly using a script provided in the submission folder.
- `nested_files`: contains a combination of files and **nested directories**, as illustrated in Code 1. This folder was created to test the behavior of the algorithm when the `-r 1` flag is enabled, i.e., when recursive traversal of directories is required. The contained files include a mix of **both large and small files**.

```
[a.nardone5@spmln data]$ du -h big_files/ small_files/ nested_files/
400M    big_files/
400M    small_files/
24M     nested_files/nested1/nested3
192M    nested_files/nested1
14M     nested_files/nested2/nested4
141M    nested_files/nested2
400M    nested_files/
```

Code 1: Disk usage of the test datasets used for evaluation.

4 Experiments and Results

At this point, the experimental behavior of the code can be analyzed. To compare the sequential and parallel implementations, both versions were **executed on the three datasets** described earlier. The commands used to run the code are shown in Code 2; they apply to both implementations, so the generic placeholder `miniz` is used to refer to either executable.

```
./miniz -r 0 -C 0 -q 1 small_files
./miniz -r 0 -D 1 -q 1 small_files
./miniz -r 0 -C 0 -q 1 big_files
./miniz -r 0 -D 1 -q 1 big_files
./miniz -r 1 -C 0 -q 1 nested_files
./miniz -r 1 -D 1 -q 1 nested_files
```

Code 2: Commands used to run the compression and decompression experiments

So, during decompression, the corresponding `.zip` files were **removed**, while the original files were **preserved** after compression. Recursive traversal was enabled only for the `nested_files` directory, as it was the only one requiring it.

The sequential version was executed 10 times on each dataset, and the **average execution** time over these runs was recorded as the representative value. A similar procedure was followed for the parallel version, with an additional variation in the **number of threads**. Specifically, for each value in the set `num_thread` = {1, 2, 4, 8, 16, 32}, the program was executed 10 times per dataset. The mean execution time across the 10 runs was then computed for each dataset and thread count.

Observation 4. All experiments were conducted on **internal nodes** of the `spmcluster`, each equipped with 32 physical cores. Therefore, the optimal performance is expected when using 32 threads.

In Figure 1, the speedup results derived from the analyses described above are presented. These results pertain exclusively to the **compression phase**. Recall now that the **speedup** is defined as

$$S(p) = \frac{T_{\text{seq}}}{T_p}.$$

Although the parallel implementation consistently outperforms the sequential version, the observed speedup remains **far from the ideal curve** (indicated by the red dashed line). Indeed, for all three datasets and at the maximum thread count of 32, the measured speedup barely exceeds 6. This indicates that only approximately 20% of the workload is effectively balanced across threads, since $32 \times 0.2 = 6.4$.

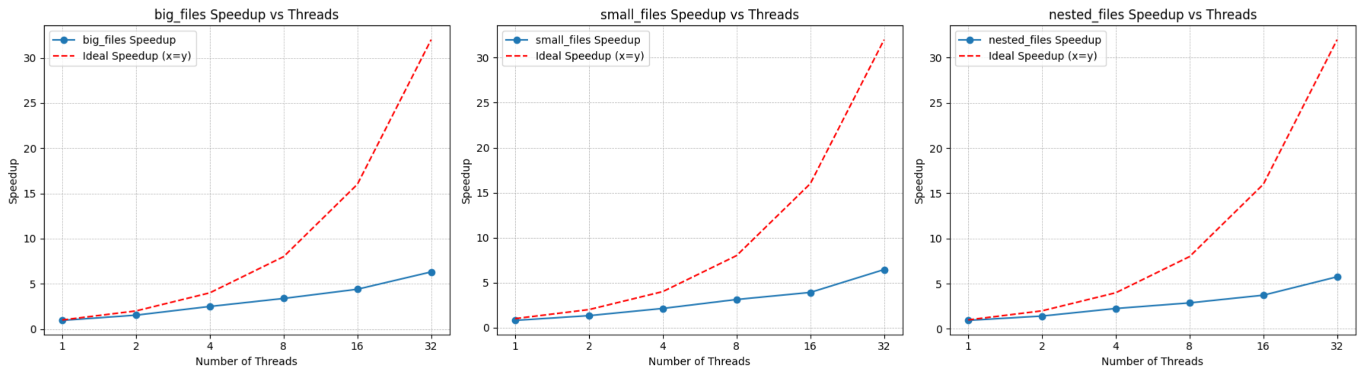

Figure 1: Speedup comparison for different datasets. The red dashed line shows ideal speedup.

This performance is **not optimal**, and several factors contribute to the **limited scalability**. First, the 400 MiB of input data must be transferred from the **frontend node** to the internal nodes over the network, introducing a fully serial latency that impacts the total execution time. Second, OpenMP incurs non-negligible overhead when **creating and synchronizing** hundreds of 1 MB tasks: each enqueue and dequeue operation, together with the implicit barriers of the `taskgroup` and `taskwait` constructs, subtracts from the time available for actual compression.

Moreover, although chunk compression proceeds in parallel, the **write-back phase remains serial** within each output file. If the storage device or network file system cannot sustain the aggregate bandwidth, threads stall awaiting I/O completion. Additionally, file mapping, stream opening/closing, and optional removal of the original file are inherently serial or poorly parallelized. According to **Amdahl's Law**, even a modest serial fraction suffices to cap the maximum attainable speedup at around 6 when using 32 threads.

Finally, **memory and allocator** contention further degrade performance: dozens of threads simultaneously access disparate regions of the mapped RAM, saturating both memory bandwidth and the TLB, while repeated `new/delete` calls for chunk buffers incur lock contention within the C++ heap. Collectively, these factors explain why the blue speedup curve remains well below the ideal red dashed line.

5 Conclusion

In this report the **third assignment** of the SPM course has been addressed. The use of **OpenMP** to parallelize a program for compressing and decompressing text files has been demonstrated. The obtained results can be considered satisfactory, particularly in light of the various challenges and complexities discussed in the previous section.

This assignment proved to be the most intricate to date. Several difficulties were encountered during its development. Initially, **understanding** the professor's provided code and the internal dynamics of file compression and decompression required substantial effort. Moreover, selecting the most effective **parallelization strategy** presented its own challenges: three different implementations were developed in an early phase (prior to chunk-based partitioning) to evaluate OpenMP's behavior. Only two of these versions were retained in subsequent development, and remnants of discarded approaches remain in `utility.hpp`. Another significant challenge involved **managing nested threads** with a fixed thread count, ensuring that the inner parallel level did not exceed the intended number of threads.

Ultimately, the final implementation can be regarded as successful, although there remains room for improvement. One promising extension would be parallelizing the **output-writing phase**: while maintaining proper file order is critical in compression, the introduction of pointer-based buffers and synchronization mechanisms could enable concurrent writes. Additionally, merging the **file-scanning and compression phases** into a single pipeline may yield performance gains. This approach was tested but did not outperform the current design; nonetheless, alternative redesigns might improve overall throughput.