SMP Assignment 3

OpenMP

DAVIDE BORGHINI

April 2025

Contents

1 Introduction and Problem Analysis

The objective of this assignment is to develop a multithreaded program that compresses and decompresses files using the DEFLATE algorithm [5], leveraging the miniz library [1] and OpenMP [4] for parallelization.

While it is technically possible to parallelize the compression of a single file without breaking compatibility with the standard ZIP format—as demonstrated by tools like `pigz` [3]—this implementation does not aim for such compatibility. As a result, files compressed with this program cannot be decompressed by standard tools such as `gzip`, and vice versa.

In the problem of compression, we can distinguish two opposite scenarios: one in which we have to compress many small files, and one in which we need to compress a single large file. These two cases should be approached differently. If we have many small files, we can simply distribute them across the available workers. However, if we have just a few large files, some resources may remain unused and the speed-up might be lower than ideal. In this case, a better option is to split the file into multiple parts and assign them to different workers.

We can also reason in terms of scheduling. When dealing with many small files—most likely of slightly different sizes—dynamic scheduling is generally preferable over static scheduling, as it better handles workload imbalance. On the other hand, in the case of a large file split among multiple workers, the work is likely to be more balanced, but this still depends on the compression algorithm, since some parts may be easier to compress than others.

With this in mind we can move on to the implementation.

2 Implementations

2.1 Sequential

The sequential implementation of the algorithm remains almost unchanged from the version provided by the professor. The main modification involves how large files are handled. Since we wanted to accurately measure speed-up, it was important for both the sequential and parallel versions to produce identical output files. To achieve this, the sequential version splits large files into chunks of a predefined size and compresses each chunk separately.

2.2 Parallel with OMP Tasks

One of the main advantages of using OpenMP, compared to manually handling parallelism, is the ability to leave most of the sequential code untouched and simply insert pragmas where parallel execution is desired. In our case, this was accomplished using OpenMP tasks. One thread reads the files passed as input and creates a task for each file.

If a file is larger than a block[1], it is divided into blocks, and each block becomes a separate task. The thread that creates these subtasks is also responsible for assembling the compressed chunks and writing the final file to disk. An alternative approach to handle this situation could involve using nested parallelism, for example by introducing a parallel for loop (which would simplify file writing using `#pragma omp ordered`).

[1] a block is defined as 1MB, this was decided after some tests. This size seemed to be the best trade-off between not creating too many tasks and creating tasks that were to big. There is not an experimental proof of this, since the optimal size depends both on the number of threads and the size of the input files, which cannot be known in advance.

However, this method was discarded, as it would complicate control over the level of parallelism and make it difficult to determine how many threads should execute within the nested parallel region.

2.3 Parallel with OMP parallel for

After some testing, the parallel implementation based on tasks did not achieve the expected speed-up, so an alternative approach was explored. The goal was to identify potential bottlenecks in the task-based version. One issue could be the overhead introduced by task creation itself, which is generally higher compared to using a simple `parallel for`. Another possibility was that the thread responsible for creating tasks was too slow compared to the consumers.

However, the latter was ruled out, as the performance drop also appeared when working with multiple large files[2]. For this reason, a simplified version of the program was implemented using a `parallel for` to investigate if better speed-up could be achieved.

In this version, one thread first scans the input and builds an array of files to process. Then, a `parallel for` distributes these files among the threads. Note that this version only handles small files in parallel; support for large files was not included, as this was just a test. Had the results shown promising improvements, more effort would have been dedicated to extending it for chunked processing of large files. It's also worth mentioning that implementing this required a substantial rewrite of the core logic.

3 Experiments and Results

As explained in the introduction, there are two extreme cases in this problem space. To reflect that, I created three different test scenarios: one consisting only of small files (a few hundred KB each), one consisting only of large files (30–200 MB), and one mixing both types. Each dataset has a total size of around 400 MB.

In the next section, I will analyse the performance of the first parallel implementation. The following section presents further experiments that were conducted to better understand performance limitations and explore possible improvements.

3.1 Experimental results

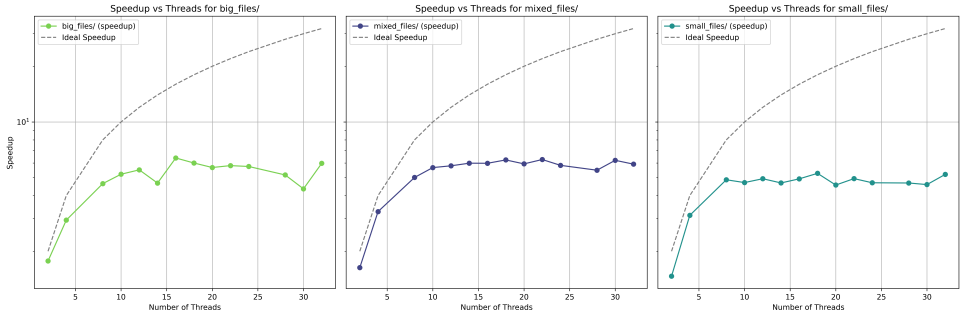

Figure 1: Graphs showing the speed-ups across three different problems, using different numbers of threads.

[2]When we have two or more big files, the number of threads inserting tasks into the queue can be greater than one.

From Figure 1, we can see that the speed-up quickly reaches a maximum of around 5 to 6 across the different problems. The speed-up is quite good when using a low number of threads (up to 4). As already mentioned, this could be due to the overhead introduced by the tasks; however, this hypothesis will be tested later by studying the version that exploits the `parallel for` construct.

Another possible cause, not yet mentioned, is that the problem might simply be memory-bounded: in this case, the machine would not be able to fully exploit its maximum parallelism. This option is plausible, considering that a significant amount of data is being moved through a NFS (Network File System).

3.2 Further experiments

3.2.1 Parallel for

In Table 1, we can see that the `parallel for` implementation did not bring any improvement as it performs on par with the task-based version. Various tests were also conducted using different scheduling policies, but in this case, the policy doesn't seem to significantly influence performance.

Since this version handles part of the workload sequentially (specifically, the preparation and listing of files to compress), one might suspect that this sequential section is a bottleneck. However, some quick measurements showed that file reading took less than 0.01 seconds, making it unlikely that this step is responsible for the lack of performance gains compared to the task-based version.

Table 1: Selected performance results on the `small_files` dataset. Rows with `task` scheduling represent the reference times using 8 and 16 threads.

Scheduling	Block Size	Threads	Mean Time (s)	Variance
task	-	8	5.01	0.07
task	-	16	4.95	0.34
dynamic	1	8	5.12	0.13
static	8	8	5.23	0.31
static	3	8	5.28	0.11
static	1	16	4.73	0.06
dynamic	8	16	4.87	0.06
dynamic	1	16	4.92	0.27

3.2.2 OMP_PROC_BIND

Given the modest results obtained even with the `parallel for` implementation, it seems increasingly difficult to achieve a better speed-up than what was already reached. At this point, it makes sense to limit execution to 8 threads, since using more does not yield meaningful improvements.

If we fix the number of threads to 8, it's worth investigating how their placement can impact performance. Because the program uses a considerable amount of memory, placing threads across two sockets could increase the usable cache, potentially improving performance. However, this might also introduce overhead due to slower inter-thread communication.

To explore this trade-off, we ran a test using 8 threads to compress the files in the `small_files` and `big_files` datasets while varying the value of `OMP_PROC_BIND`.

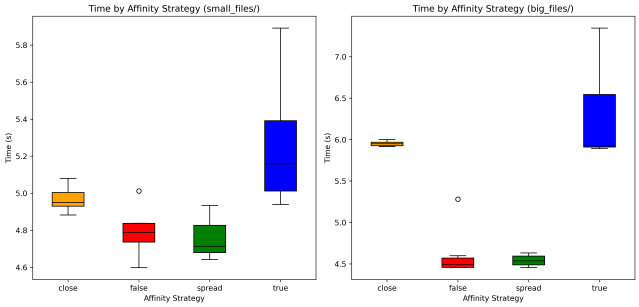

Figure 2: Performances in seconds with different value of `OMP_PROC_BIND`

As shown in Figure 2, thread placement has a significant effect on performance[3]. The results are especially evident on the `big_files` dataset. Placing threads close together (e.g., on the same socket) tends to hurt performance, whereas letting the OS decide (option `false`) or explicitly spreading threads (`spread`) yields better results.

This behaviour supports the hypothesis that the program is memory-bound: distributing threads across sockets allows for better use of memory bandwidth and cache, improving overall throughput. A final test will further explore this hypothesis.

3.2.3 Non-compression time

This final test is simpler and not explored as deeply as the previous ones. It involves a program that, in sequence: reads the files, maps them to memory, duplicates their content (inside a parallelized `for` loop in the multi-threaded version), and finally unmaps the memory. The test was run both sequentially and using 8 threads. Results are reported in Table 2. We observe that the `mmap` and `munmap` operations do not introduce significant overhead. Additionally, switching from 1 to 8 threads results in a speed-up of nearly 2×. Since this task is essentially embarrassingly parallel, the limited speed-up confirms that memory bandwidth is a bottleneck for this application.

Table 2: Time measured (on few handmade run) duplicating files. The data used are the ones contained in `small_files/`

Version	**Map**	**Duplicate**	**Unmap**
Sequential	0.13	12.31	0.10
Parallel	0.10	6.84	0.11

4 Conclusions and Possible Further Optimizations

In this work we explored the performance of a parallel compression program based on the deflate algorithm. We considered different input scenarios, from many small files to large individual files, and implemented multiple parallel strategies using OpenMP. The task-based version offered a decent speed-up, especially with a small number of threads, but scaling further proved difficult. Attempts to improve the results using `parallel for` didn't lead to noticeable gains and confirmed that the overhead from tasks wasn't the main limiting factor.

[3]The `master` option was excluded as it produced out-of-scale results

Further tests pointed towards memory as the real bottleneck, with the application quickly saturating bandwidth when using more threads. Affinity settings showed that spreading threads across sockets can help in some cases, supporting the idea that memory usage is the dominant factor. Lastly, synthetic tests without compression confirmed this, showing limited speed-up in a situation that should be highly parallel.

Overall, we can say that beyond a certain point, adding more threads doesn't help much, and performance depends more on memory layout and data movement than on raw CPU power.

References

[1] Github repo of miniz: GitHub. (2025)

[2] Github repo of the assignement: GitHub. (2025)

[3] pigz official website: pigz. (2025)

[4] OpenMP website: OpenMP. (2025)

[5] An overview of the deflate algorithm: zlib. (2025)