Parallelization of Huffman Coding

`https://github.com/fpezzuti/Parallel-Huffman-Coding`

Parallel & Distributed Systems course's final project

University of Pisa, Master Degree in Artificial Intelligence

Francesca Pezzuti

Academic Year 2022/2023

Contents

1 Introduction **2**
1.1 Algorithm analysis 2

2 Sequential implementation **2**
2.1 Performance test & need for parallelization 2
2.1.1 Preliminary performance test 2
2.1.2 Relationship between file size and completion time 3

3 Towards parallelization: theoretical performances & limits **4**
3.0.1 Problem characteristics 4
3.1 Parallelization limits: serial fraction 4
3.2 Theoretical performances of many parallel versions 5

4 Pure C++ thread-based implementation **6**

5 FastFlow implementation **7**

6 Experimental results **7**
6.1 Benchmark of $100MB$ 7
6.1.1 Considering I/O operations 7
6.1.2 Excluding I/O operations 8
6.2 Speedup, scalability, and efficiency 8
6.2.1 Excluding I/O operations 9
6.3 Comment 10

7 Build, execute and test **11**

1 Introduction

The project is focused on the parallelization of the well-known *Huffman Coding* algorithm, the classical data compression algorithm for texts.

In this work will be proposed a sequential of the algorithm, a pure C++ thread-based version, and a *FastFlow* version.

1.1 Algorithm analysis

To correctly design the parallelization of the *Huffman Coding*, an initial analysis of the algorithm has been performed.

The main operations that compose the *Huffman Coding* algorithm are:

During the symbol's frequency computation, the frequency (i.e., the number of occurrences in the file) of each symbol appearing file is computed, then based on symbols' frequencies the Huffman tree for the input file is built, once the Huffman tree is ready, the input file is compressed substituting each symbol with its compressed version, lastly the result is written to file.

2 Sequential implementation

The sequential version of the Huffman Coding has been implemented in files `sequential.cpp` and `sequential.hpp`.

2.1 Performance test & need for parallelization

To understand which operation needs to be parallelized and which not, a measurement of each operation of the Huffman Code computation has been performed running the sequential version of the *Huffman Coding* and exploiting the class *utimer* for time measurements.

2.1.1 Preliminary performance test

The total completion time of the sequential version has been estimated performing 10 runs of the algorithm over an input file of $100MB$ containing randomly generated ASCII symbols.

The test has been carried on the *SPM remote machine*[1], and for each run it has been computed the mean value and the standard deviation of each computation phase (the ones described in Section 1.1) and all the results have been documented in Table 1.
A graphical representation of the mean duration of each of the phases of the sequential computation of the Huffman Coding is reported in Figure 1.

From the results of this test emerged that the most expensive phase is the compression of the file, immediately followed by the count of the frequencies of the symbols appearing in the file and by the write of the compressed file, while the construction of the Huffman tree showed to be a

[1] a *Dell PowerEdge R7425* equipped with 32 cores

T_c (μs)	Computation	$E[T_c]$ (μs)	Std. Dev.	Percentage
3.908.834	Read input file	116.178	3.948	2,97%
	Count symbol freqs.	993.049	6.369	25.41%
	Tree construction	77	3	0.00%
	File compression	1.850.710	100.453	47.35%
	Write output file	904.056	3.875	23.13%

Table 1: Mean duration of sequential computations (10 runs on a $100MB$ input file)

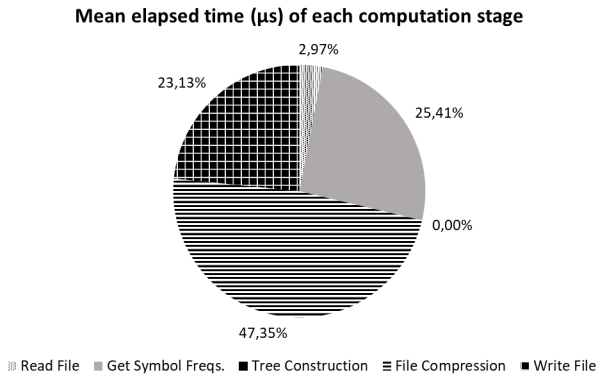

Figure 1: Duration of sequential phases run on *SPM remote machine* on a $100MB$ benchmark.

lightweight operation; this result is explained by the fact that the construction of the tree and the compression table are operations with complexity $O(|A|)$ where A is the number of symbols of the alphabet of the file, while all the other phases have a complexity which is $O(M)$ where M is the length of the input file; thus, as the dimension of the input file increases, the duration of all the phases will increase almost in the same proportion except for the creation of the tree which is expected not to increase too much for the reasons just discussed.

Note that read/write operations may have an high weight on the execution time and they are not parallelizable, thus, when parallelizing it should be shown the result obtained both considering the I/O operations, and both without considering them since the speedup may vary a lot.

2.1.2 Relationship between file size and completion time

Another test has been carried out running on the *SPM remote machine* the sequential algorithm varying the size of the input file: three different benchmarks have been tested on 10 runs each ($1MB$, $10MB$, $100MB$) and the results are shown in Table 2

Input Size (Mega Bytes)	Mean T_c (μs)	Std. Dev.
1	32.047	1.125
10	367.606	6.566
100	3.908.834	101.736

Table 2: Relationship between input size and mean completion time (10 runs)

From the statistics shown in Table 2 it is evident that the algorithm needs to be parallelized

since increasing the size of the input, the total duration of the program increases almost in a proportional way, thus the compression of huge files using this algorithm would take too long.

Looking at the graph reported in Figure 2 which shows how the duration of each of the phases that compose the sequential version vary when increasing the size of the input file, it appears clear that the most costly operation in terms of execution time is always the compression of the file and the other very heavy operation is always the count of the frequencies, while the construction of the tree is constant in duration (exactly as it was imagined); therefore it makes sense to parallelize both the count of the frequencies and the compression of the input file.

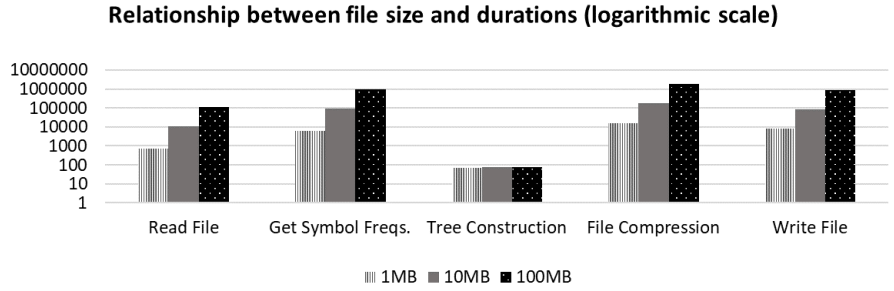

Figure 2: Relationship between input file dimension and duration of sequential computations (10 runs on the *SPM remote machine*). Note that the y-axis is in logarithmic scale.

3 Towards parallelization: theoretical performances & limits

To increase the performances when executing the program on large text files, the computation of the frequencies of the symbols appearing in the document and the compression of the input file needs to be parallelized since they are heavyweight operations.

3.0.1 Problem characteristics

From the point of view of the parallelization, the problem is a **data parallel problem** in which we are given an input file, and we can parallelize the compression of the input file splitting it in chunks and assigning each chunk to one computational resource that will be in charge of performing some computation, and then the central computational resource will merge the local results into the final result.

3.1 Parallelization limits: serial fraction

The stages of the computation that compose the **serial fraction** are the load of the uncompressed file into main memory, the store of the compressed file into disk, and the construction of the Huffman tree, and together they compose the $27,25\%$ of the total execution time (for the $100MB$ benchmark); the speed-up is limited by the serial fraction which is of $27,25\%$, thus for the *Amhdal Law* the maximum speed-up achievable is $3,67x$. If instead the I/O operations are excluded from the computations, the serial fraction is composed just by the computations of the construction of the Huffman tree and table, therefore the serial fraction is of 1.55% and it is possible to achieve a far higher speedup.

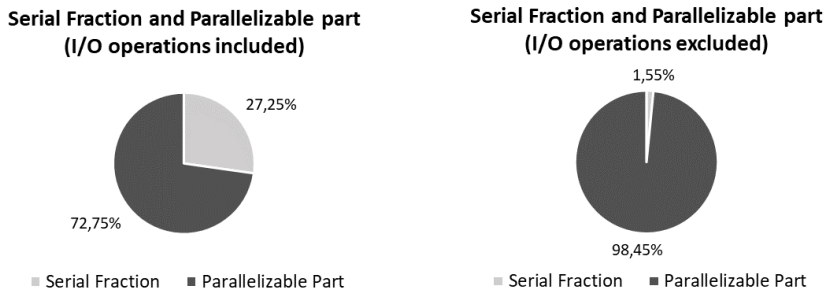

Figure 3: Serial fraction and parallelizable part measured for $100MB$ benchmark.

3.2 Theoretical performances of many parallel versions

Count of symbols' frequencies
The simplest way to parallelize the count of symbols' frequencies consists in splitting the input in chunks and assigning each chunk to a worker which will be in charge of performing the computation for its portion of input file, then merge the partial counts together in a unique alphabet of pairs $(symbol,\ freq.)$; so, this stage can be parallelized using a **map followed by a reduce** where the reduction of the local results does not need to follow a specific order. The theoretical performances obtainable using this solution are a decrease of factor nw (number of workers) in the time needed to count the frequencies. Note that in this solution there is an additional resource that behaves as emitter and collector (reducer).

Note that the workload of each of the worker is almost the same, thus, dynamic scheduling is not needed and the static scheduling is a good approach.

Another possible approach applicable to the parallelization of the count of symbols' frequencies is to consider the stages of read from file, construction of the alphabet by means of Map and Reduce as a sub-problem of stream parallel kind where one computational resource reads the input file chunk by chunk, each time emitting a chunk so that the worker assigned to that chunk can immediately start the computation of the alphabet for that chunk, and after all the file has been read, the computational resource in charge of reading the file can start reducing the partial results as soon as they arrive. This way the count of symbol frequencies overlaps in time the read of the input file, and similarly the reduction overlaps the Map; given that the read operation is far faster than the count of the symbol frequencies, interrupting the read to spawn a task/worker at each chunk read would cause some additional overhead, therefore I decided not to follow this approach.

Compression
Since the compression of different characters appearing in the text file is independent from the compression of the other characters, it can be parallelized using a **map** in such a way that the input file is divided in chunks and each computational resource will take care of compressing a chunk of characters. From a theoretical point of view, the performances of are improved by a factor nw (where nw is the number of workers).

Also in this case static scheduling is good enough since the workload of each worker will be almost the same.

Exactly as for the count of symbols' frequencies, a possible improvement consists in overlapping the compression of the file and the write of the compressed file, but this time since the write

operation is a heavy weight operation, it may worth it to try to anticipate it writing to file each compressed chunk immediately, avoiding to wait for the compression of all the chunks; thus, I decided to anticipate the reduction in this case.

Conversion to writable bitstring
The conversion of the compressed ASCII bitstring where each $'0'$ and $'1'$ is a character (thus, its size is of 8 bits) into a bitstring where each $'0'$ and $'1'$ occupies just a single bit, has been considered to be part of the serial fraction because codes are of variable length and to append one code to another code it may happen that the code to be appended must be realigned, therefore it only makes sense to first compute the whole compressed ASCII bitstring, and then to convert it into a writable bitstring.

4 Pure C++ thread-based implementation

The C++ thread-based version has been implemented in `par_thread.cpp` and `par_thread.hpp` using the **fork-join model** to manage the threads. The main methods used are the `MapAndReduceFreqs` and `reducePartialresults` for the frequency count, and the `PipeMapAndReduceCompression` for the compressiona and write of the file.

Count of symbols' frequencies
In this implementation, once the main thread has read the whole file, it spawns nw threads assigning to each of them a chunk of the file and then it awaits for partial results to be reduced (the main thread works as consumer), once awaken, it reduces the received local result and awaits for the next local result up until all the local results have been merged in the final result. Each worker thread performs the count of the symbols' frequencies for its portion of file, inserts the resulting alphabet in the vector of local results and notifies the main thread (the workers behave as producers).

One **conditional variable and its mutex** are used in order to protect the shared vector of local results: the workers use it to write in mutual exclusion their local results and to notify the main thread that a new local result is ready, and the main thread use them to await for new local results to be reduced.

The reduction phase simply consists into joining the local alphabets (made by pairs $(symb., freq.)$ into a unique alphabet.

Compression
Regarding the parallelization of the compression, the approach used is the same as the one for counting symbols' frequencies but the worker function and the reduction phase are different: this time workers compress their chunk of the file string (thus, their local result is a string), and once the main thread receives a local result from one worker, it converts it converts it into the writable format eventually appending it to a byte remained from the previously reduced local result and writes it to file; the main thread has to keep track of the wasted bits in the last byte converted to the writable format so that once it receives a new local result, it aligns it to the next bit available. So, every time the main thread writes to file a new partial result, it avoids writing the last byte if there are some wasted bits and keeps track of the next bit available for write (except for the last local result, in this case all the bytes are written to file).

5 FastFlow implementation

The FastFlow version of the Huffman Coding has been implemented in files `par_fastflow.cpp` and `par_fastflow.hpp` and the main methods are `MapFreqsCount` for alphabet construction, and `PipeMapCompression` for the compression and write.

Count of symbols' frequencies
The FastFlow implementation of the count of the frequencies of each symbol consists into a **parallel for reduce** with **static scheduling** and where the variable subjected to reduction is an alphabet. The *iteration function* (for phase) simply consists in increasing in the local alphabet the frequency of i-th char appearing in the file string, thus each worker of the parallel for will have its own alphabet and will take care of performing a bunch of iterations of the parallel for. The *reduction function* (reduce phase) consists in merging a local result, thus an alphabet, to the global one.

Compression
Concerning the implementation of the compression, an **ordered farm** with customized emitter and collector has been used. The **emitter** splits the input file in chunks and sends out a task containing the start and the size of the chunk, the **worker** in its `svc` method compresses the chunk it has been assigned and stores its local result inside the task, the **collector** in its `svc` method converts the compressed bitstring into a writable bitstring (with eventual append and realignment to a writable byte remaining from the previous local result received) and writes it. Clearly, using a Farm has two additional resources used: one for the emitter and one for the collector. Note that the ordered farm is needed since we must guarantee that the local results are reduced in order.

6 Experimental results

A first test of the performances has been executed on the file of $100MB$ varying the number of workers, running 10 iterations, and comparing the versions, then peedup, scalability, and efficiency have been calculated for each version, varying the size of the input file; all the data is stored in file `/evaluation/HuffmanCoding.xls`.

6.1 Benchmark of $100MB$

6.1.1 Considering I/O operations

Considering I/O operations, the results obtained for the $100MB$ benchmark are the ones in Table 3.

Nw	Sequential	C++ Threads	FastFlow
1	3.908.834	4.802.961	3.945.936
2	-	3.192.331	2.559.841
4	-	2.278.096	1.842.807
8	-	1.826.990	1.610.333
16	-	1.595.047	1.393.779
32	-	1.383.344	1.411.509
64	-	1.298.997	1.621.211

Table 3: Comparison between the performances on benchmark of $100MB$ with 10 runs of each version (I/O operations included). The execution time is expressed in μs.

6.1.2 Excluding I/O operations

Excluding the I/O operations from consideration, the results obtained are the ones in Table 4.

Nw	Sequential	C++ Threads	FastFlow
1	2.888.600	3.782.727	2.925.702
2	-	2.172.097	1.539.607
4	-	1.257.862	822.573
8	-	806.756	590.099
16	-	574.812	373.545
32	-	363.109	391.275
64	-	278.763	600.976

Table 4: Comparison between the performances on benchmark of $100MB$ with 10 runs of each version (I/O operations excluded). The execution time is expressed in μs.

6.2 Speedup, scalability, and efficiency

The statistics have been plotted for three different file sizes and the results are shown in Figures below.

The graph of the speedup shown in Figure 4 reports the speed-up obtained considering the I/O operations in the measurements. The speed-up obtained by both the two parallel versions is not far from the maximum achievable, thus the results in terms of speedup obtained by the two parallel implementations are satisfactory, but it must be also observed that the threads version produces an higher speedup probably due to the fact that FastFlow is more structured. Obviously, the I/O operations are part of the serial fraction, therefore a more accurate analysis of the results must be performed taking out the I/O operations from the measurements.

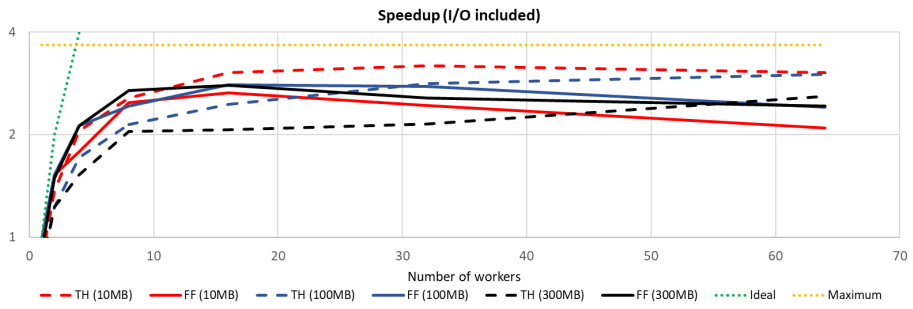

Figure 4: Speedup considering I/O operations.

The scalability graph reported in Figure 6 shows that the scalability of the thread version is a better than the one of the FastFlow version and that the best scalability is of around 4.

The graph of the **efficiency** in Figure 6 shows that in both the versions the efficiency decreases as the number of workers increases because increasing the number of workers the overhead increases and the benefit coming from using more workers decreases. This was expected since handling more threads adds some complexity, and especially for small files the chunks assigned to each worker becomes too small.

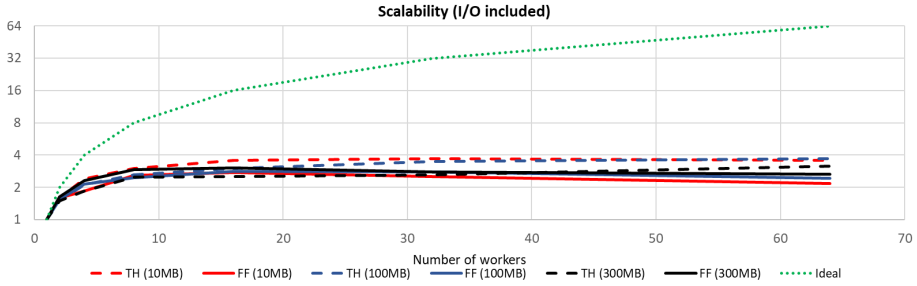

Figure 5: Scalability considering I/O operations.

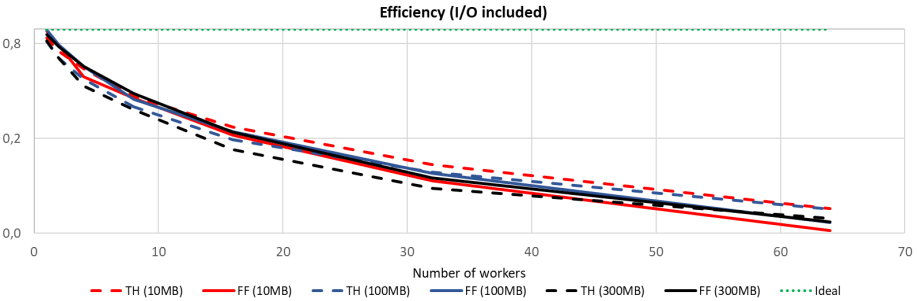

Figure 6: Efficiency without considering I/O operations.

6.2.1 Excluding I/O operations

Graph in Figure 7 shows the speedup obtained without considering the weight of the I/O operations in the time measurements. From this graph we can see that the thread version works better with higher number of workers, while the FastFlow version works better for low number of workers used. As we can see, the speedup is not optimal, but it is still a good result since the maximum speedup achieved for the $10MB$ benchmark is of 12.13 (threads version with 32 workers), the highest speedup for the $100MB$ benchmark is of 10.36 (threads version with 64 workers), and the maximum speedup achieved for the $300MB$ benchmark is of 7.57 and has been obtained by FastFlow with 16 workers.

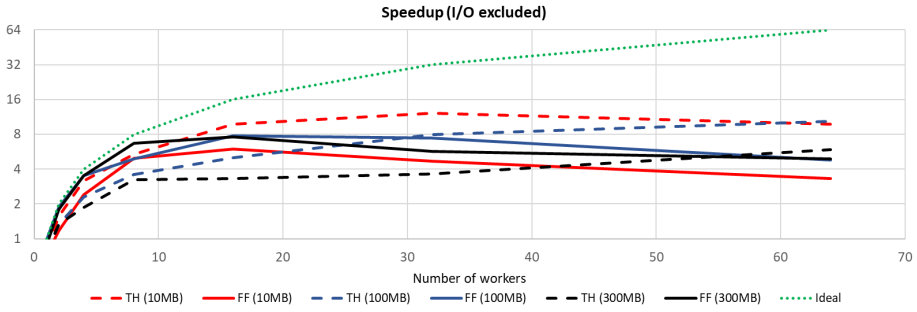

Figure 7: Speedup without considering I/O operations.

Looking at the graph of the **scalability** shown in Figure 8, if we don't consider the I/O operations, the scalability is higher than when considering them in the measurements; in particular we obtain the maximum scalability of 14.82 with 32 workers in the threads version in the $10MB$ case, while for the FastFlow version the highest scalability has been obtained for the $300MB$ case using 16 workers and is of 8.52.

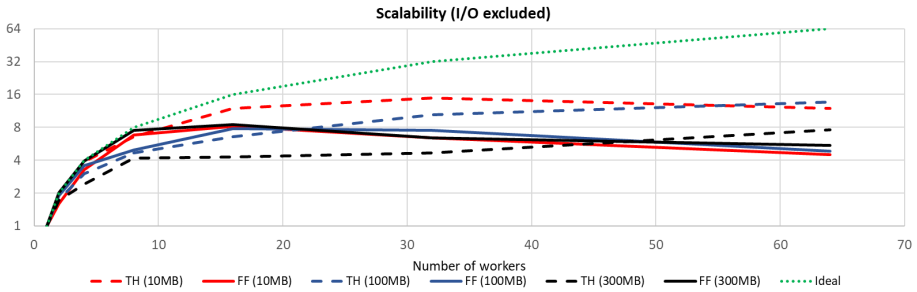

Figure 8: Scalability without considering I/O operations.

The graph of the **efficiency** is shown in Figure 9 and as we can see, differently from the case that considers the I/O operations, now the efficiency decreases slower.

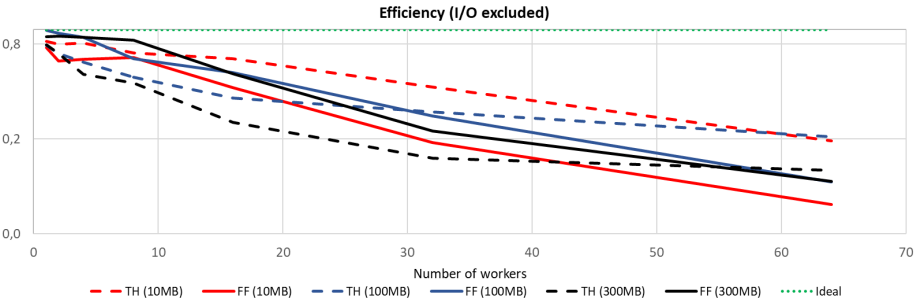

Figure 9: Efficiency without considering I/O operations.

6.3 Comment

In conclusion, the results obtained are quite good in terms of speed-up, scalability and efficiency; FastFlow's results are slightly worse than the ones of the threads versions, probably due to the fact that it is less flexible and more structured. The point of stregth of FastFlow is the fact that it optimizes the communications, therefore when the number of workers is low, it works better than the threads version. Both the versions suffer a big overhead when using a large number of workers with not-so-big files, therefore it would be interesting to perform some tests with files of higher dimensions in order to see which is the speedup when increasing more the workload, especially since when testing the performances with small files (e.g., the $10MB$ benchmark) the variance is high. A possible improvement can be to assign to workers a not-too-small chunk and avoiding spawning workers if the file is small in such a way to reduce the overhead.

7 Build, execute and test

Build
In order to **build** the project, from directory `/code/` launch:

```
. ./buildProject.sh
```

This script file creates the build directory, builds the project, and moves to the build directory (`/build-dir/`).

Execute
To **execute** the project, from directory `/build-dir/` launch:

```
./HuffmanCoding STANDARD nw [inputFileName]
```

where `STANDARD` means standard execution mode, `nw` is an integer number representing the parallelism degree required, and `inputFileName` is an optional argument representing the input file to be compressed (if not specified, a default input file will be used for testing purposes).

Important note: before executing the project, be sure that the input file is a `.txt` file already stored inside directory `/data/inputs` and containing only ASCII characters.

Test
To test the performances of one version of the program, from directory `/build-dir/` launch:

```
./HuffmanCoding TIMING nw inputFileName {SEQ, TH, FF} nIter
```

where `TIMING` means performance tests mode, `nw` is an integer number representing the parallelism degree required, `inputFileName` is the input file used as test benchmark, then an execution mode argument (`SEQ`, `TH`, `FF`) must be provided in order to specify which is the tested version, and lastly the number of iterations `nIter` must be specified.

To **collect the statistics** of all the versions, from directory `/test/` launch the default script:

```
./timingTests.sh
```