Assignment 5 : files compressor

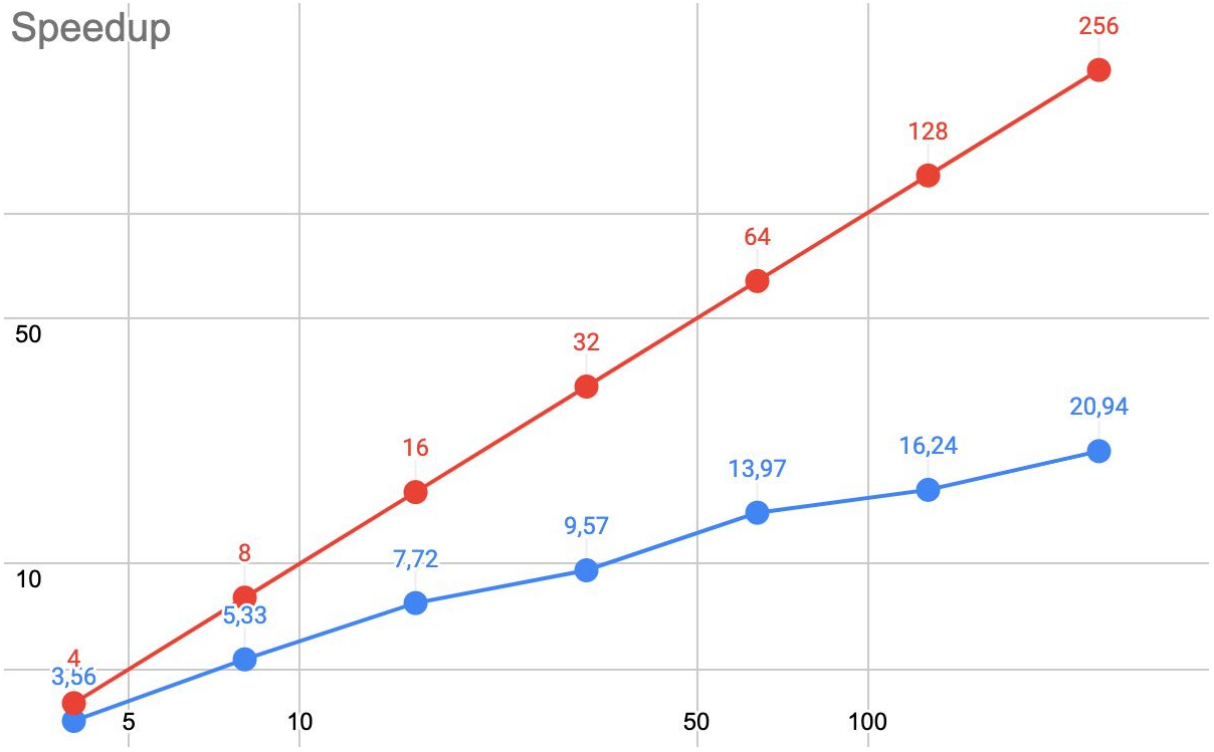
Exercise 1

Those are the main differences with respect to the pipeline version given.

```
ff_Pipe<Task> pipe(compress, writer);

std::vector<std::unique_ptr<ff_node>> W;
for (size_t i = 0; i < nw; ++i)
{
    std::unique_ptr<ff_node_t<Task>> com(new Compressor());
    std::unique_ptr<ff_node_t<Task>> wrt(new Write());

    W.push_back(std::unique_ptr<ff_Pipe<Task>>(new ff_Pipe<Task>(com, wrt)));
}

ff_Farm<Task> farm(std::move(W), reader);

farm.remove_collector();
farm.wrap_around();

std::cout << nw;
utimer u("");

if (farm.run_and_wait_end() < 0)
{
    error("running farm");
    return -1;
}
```

I decided to let the Master walk through the folders and open the files. Then it passes each file found to the NW workers in a Round-robin way. Those workers will compress the file received and then compressed it using the library *miniz*.

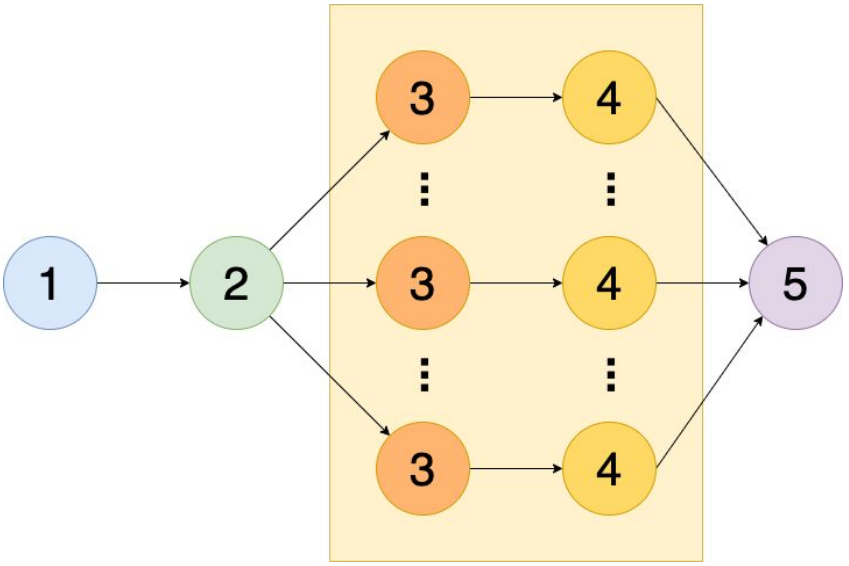
This approach scales quite well, but strictly depends both on the number of files and the size of them.



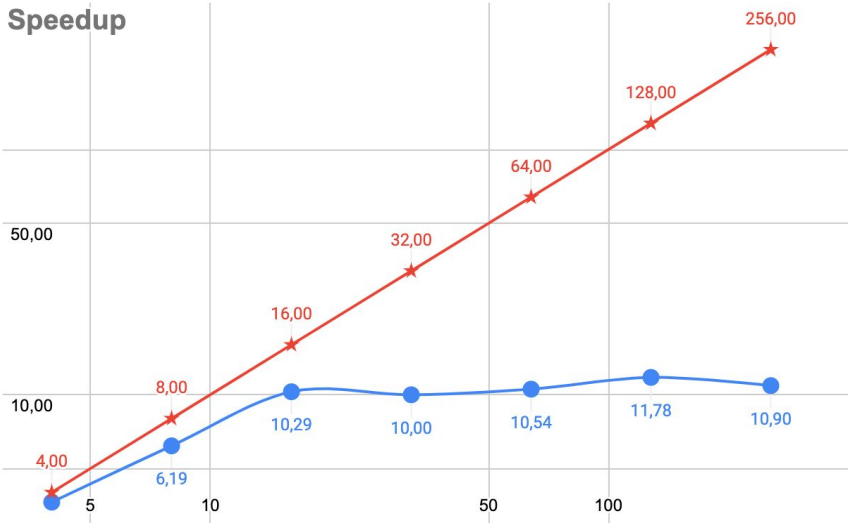
Exercise 2

To implement the second exercise I decided to organize the architecture as follows:

1. Explore the folder (or file) and if there is a file larger than a fixed threshold, then split it into chunks of dimension equal to the threshold and save those to the disk. Finally, send to phase 2 the filename of the chunks.
2. Open each file coming from the previous stage and then pass the raw bytes to the compressor stage.
3. Compress in memory each chunk.
4. Store each chunk.
5. Collect all the compressed chunks and merge them into a single zip file. Finally, delete all useless temporary files.



The speedup curve grows until 16 workers then it flattens.



Part of the source code implemented

```
int nw = atoi(argv[1]);
size_t threshold = atoll(argv[3]);

//Stage 1
Splitter split(argv[2], threshold);
//Stage 2
Read reader;
//Stage 3
Compressor compress;
//Stage 4
Write writer;
//Stage 5
FileMerge fmerge;

ff_Pipe<Task> pipe(compress, writer);

std::vector<std::unique_ptr<ff_node>> W;
for (size_t i = 0; i < nw; ++i)
{
    std::unique_ptr<ff_node_t<Task>> com(new Compressor());
    std::unique_ptr<ff_node_t<Task>> wrt(new Write());
    W.push_back(std::unique_ptr<ff_Pipe<Task>>(new ff_Pipe<Task>(com, wrt)));
}

ff_Farm<Task> farm(std::move(W), reader);
farm.set_scheduling_ondemand();
ff_Pipe<Task> complete(split, farm, fmerge);

if (complete.run_and_wait_end() < 0){ ..... }

//Remove temporary files
system("find . -name \"*.tmp.part*\" -delete");
```

Using a small threshold leads this approach to suffer some problems. In fact, using a small threshold size, the number of chunks will grow exponentially and the performance will be bad. Another peculiar aspect of this architecture is that sometimes the directory walker will find the completed file, this happens because the last stage finishes before the first has completed his work. A possible solution could be a queue of tasks between the first and the second stage.