HPC-Parallel-Huffman-Coding-Decoding

1. Introduction

The main concept of this algorithm is to allow the users to achieve a lossless data compression.

This is done by deriving a dictionary that maps each symbol of the data to a variable-length code where the length is based on the probablity of the symbol itself.

This algorithm is mostly applied on texts, images and videos.

1.1 Challenges

The main challenges have been, as starter, finding some related papers as most of those focused on the concept of the algorithm and how to improve it rather than the actual application and design.

The few papers that could be useful mostly implemented different approaches, especially using the OpenCL framework for the parallelization, compared to the one required in this project which is the usage of the MPI library.

Since the OpenCL framework allows the implementation of parallel programs on GPUs and thanks to the rise in high-computation performances of the GPUs, the majority of the tests results on those papers(when actually present and satisfying) were not comparable with our results as run on entirely different systems.

Another challenge with the papers is the fact that the few that published the results, didn't also publish the code but only explained the basic theoretical concept, while the ones that showed the pseudo-code didn't show the actual results.

This are the reasons why we had to build the entire algorithm from scratch.

2. Problem analysis

The main problem to address with this algorithm is the data compression.

Due to the immense volumes of data produced by high-performance computing (HPC) applications the challenge to store those data arised and with it the problem of efficiently and accurately compress it while also being able to retrieve the original data whenever needed.

The basic solutions was to build a serial compressor, which huffman did based on prefixed codes of varying length, this was a fairly simple concept that can be seen in the following pseudo-code:

```
int main(int arg, char const *argv){
    init variables;

    if encoding then:
        //first inputFile pass
        for symbol in inputFile do:
```

```
            get symbols;
            get frequency of each symbol;

        //build encoding tree, this assigns a code to each symbol
        while sizeof(symbols != 1):
            min1 = extractMin(symbols)
            min2 = extractMin(symbols)

            //create new node with frequency equal to sum of the two mins.
            newNode = ('$',min1.frequency + min2.frequency)
            newNode.leftChild  = min1;
            newNode.rightChild = min2;

            insert newNode in symbols;

        second inputFile pass:
            read symbol;

            //recursively move on tree until symbol found.
            if treeNode.symbol is symbol then:
                write code to outputFile;
            else:
                if treeNode.leftChild != NULL
                    move left and add '0' to code;
                if treeNode.rightChild != NULL
                    move right and add '1' to code;


    if decoding do:
        //same as encoding but inputFile has as first row the symbol-frequency
        //mapping.
        //instead of moving until symbol is found, we move left or right based
        //on the value read(0 or 1)

}
```

Once the serial algorithm was to correctly work the idea was to parallelize it and that's what's been done.

3. Parallelization

There are only three possible ways to parallelize this algorithm and these are:

1. Multicore / Multiprocessor

   The basic idea is to apply the serial algorithm but split input file into different chunks, generally the size of each chunk is calculated as `chunkSize = fileSize/(number of processes)` and assign each chunk to each processor/thread to encode.

   However, since each symbols vary in code-length, the whole encoding result of each processor/thread will most likely be of different sizes from one another and this also does not assure that each result is

byte-complete as each value of the code is a bit and the total might not amount to a byte-size.

This means that for every encoded chunk we need to do a shift to correctly align the outputs of one chunk with the immediate next one in order to fill the byte.

Example of occurences of this:

First chunk encoding results in 10 bytes and 4 bits, the second chunk will have to be shifted 4 bits in order to fill the last byte of the first chunk.

2. Threading

   Very similar to the previous approach but avoids the scenario in which the first chunk is the last to be computed and every process has to wait for it to be done to know the position of the file to write onto.

   This is done by splitting the input in even smaller chunks and parceling the data to processors in a FIFO approach this allows to have a lower waiting time.

3. GPU

   While the approach is almost the same as on the CPUs, due to the native structure the GPUs are much more efficient on handling parallel workloads.

3.1 Design and implementation of parallel solution

Given the resources, the requirements and the topics explained to us during the course, the first approach has been chosen.

The idea of the implementation was to take as baseline the serial code built beforehand and add the MPI functions there, this in order to not rewrite the entire algorithm from scratch.

As there are many different possible choices that can be done in the implementation we decided to act as follows:

1. I/O

   Regarding the I/O we decided to have a file handler in each process and each one write its own output, This approach was chosen due to the fact that MPI takes care of the access conflicts allowing to have only the data each process will work on in its memory avoiding broadcasts.

   The only exception to this was to retrieve the size of the file where we made one process read it and the broadcast it as opposed to have each process get it because it's faster to broadcast a value than to go through the whole file `n` times where `n` is the number of processes.

2. Building of the huffman tree

   Due to the way the tree is built it's not possible to parallelize this process.

   The only choice to be done about the tree was between:

   1. having a process build it and broadcast it
   2. having each process build it locally

The second approach has been chosen as the tree was built in the heap and only a pointer to the root was kept, meaning we would've had to broadcast a whole part of the heap.

While it would be theoretically better to have only process 0 build the tree and broadcast it, it would take way too long and, based on the size of the tree, we could saturate the bandwidth. Calculating it locally takes a bit of time but not as much as broadcasting it hence why this was the chosen approach.

3. Building the dictionary

Similar concept to the tree, in this case parallelization is possible but only once at least a process is done building the tree, meaning that in the worst case each process could be done building the tree and have to wait for the only process delegated to the building of the dictionary.

This approach wouldn't also make sense as then it would be just better to have a single process build the tree and the dictionary instead of everyone and broadcast the dictionary, but similar to before due to the size it would probably be slower to do this as `n-1` processes would've to wait until one process is done with computation and communication.

Hence why this, like the tree has been build locally for each process, this also allows to have each process read its own input data and build a smaller tree, this would require storing a new mapping of symbol-frequency on the output for each chunk.

3.3 Benchmarks on the cluster

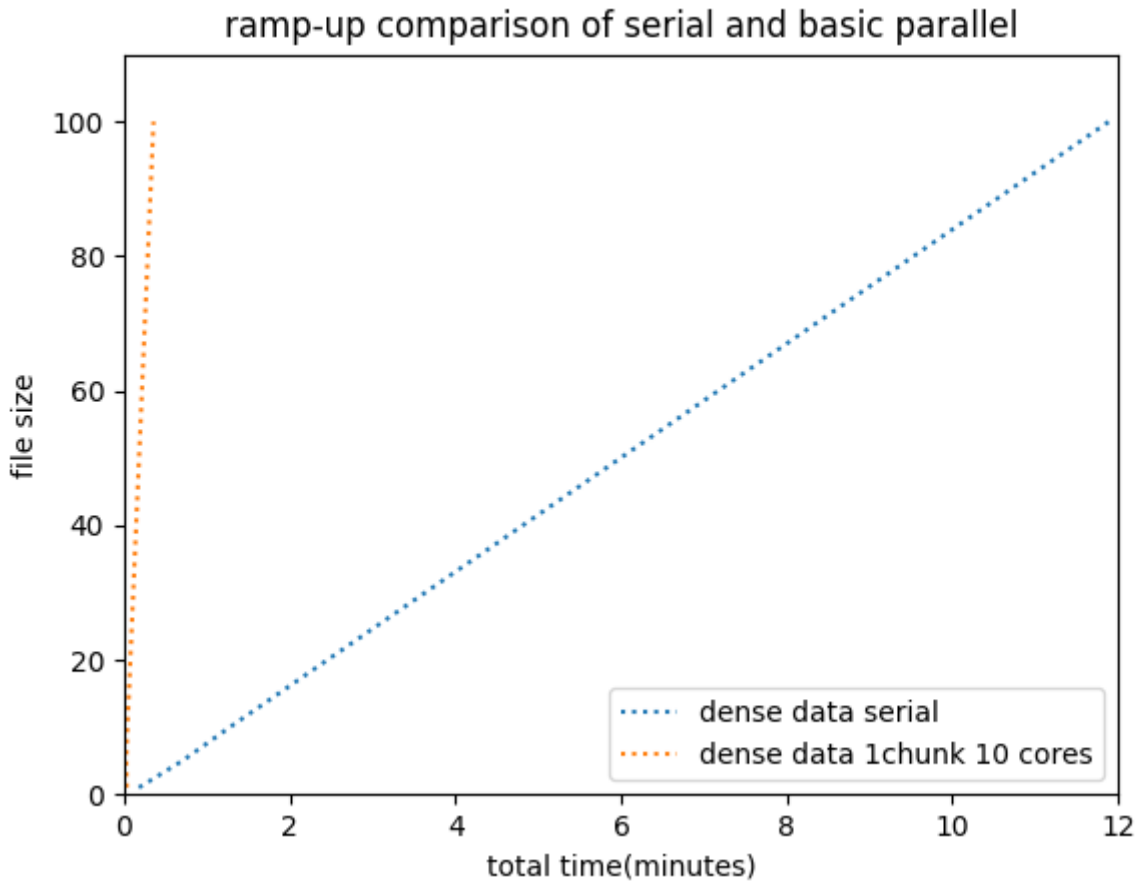

fig. 1 - Comparison between serial and parallel ramp-up

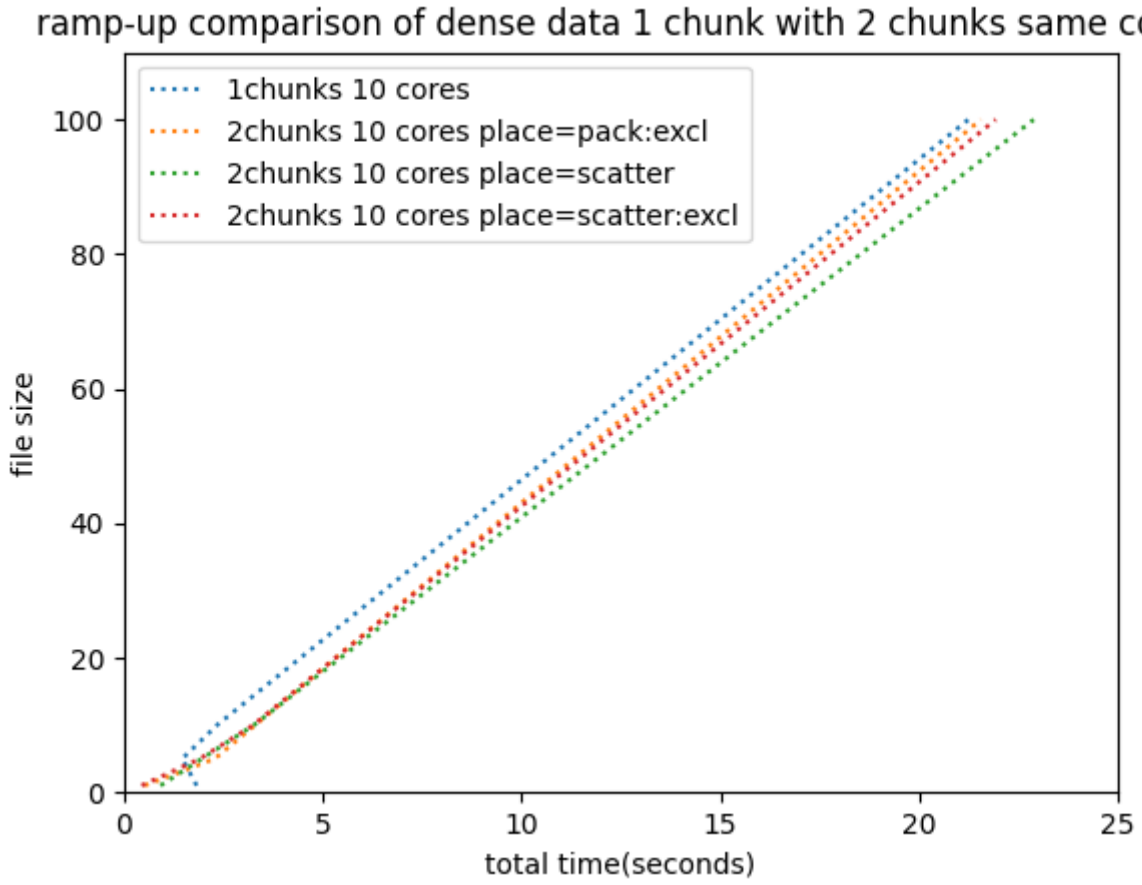

fig. 2 - Comparison between different placing ramp-up

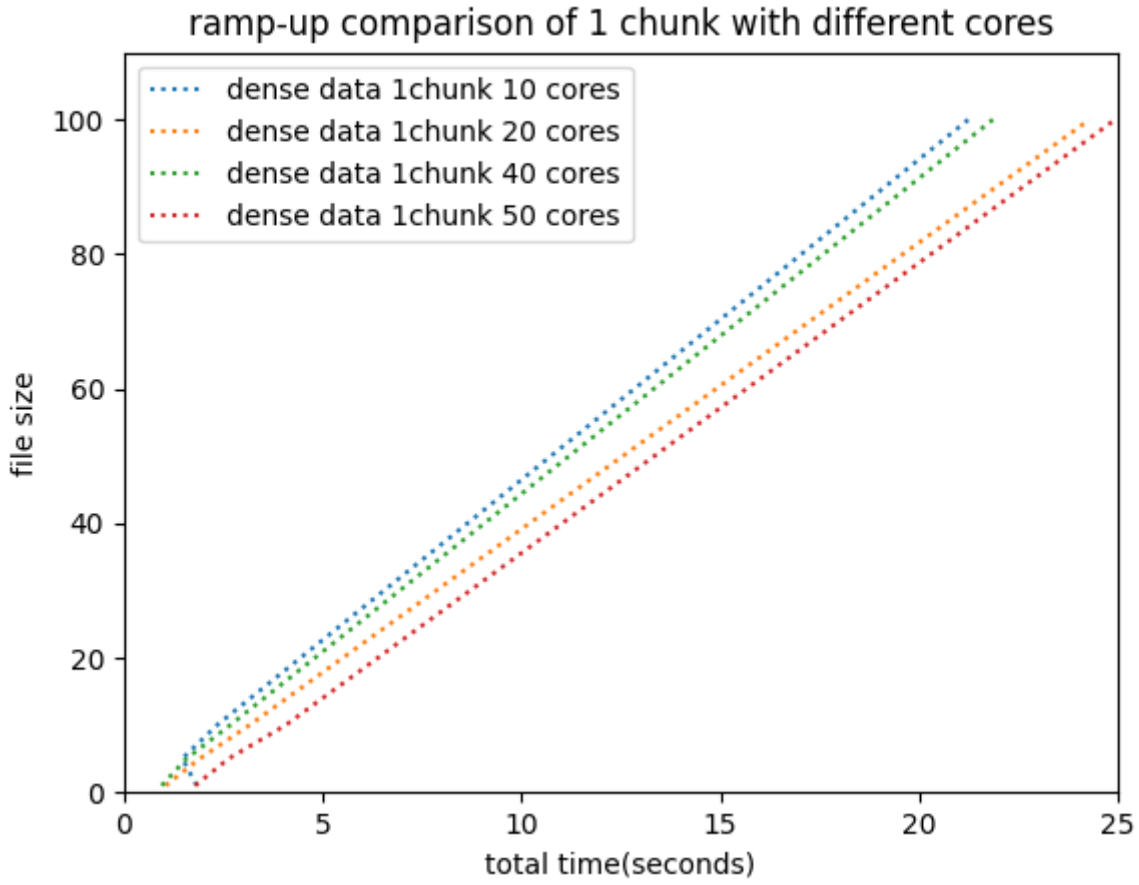

fig. 3 - Comparison between different cores ramp-up

4. Conclusions

As expected, the parallel application is much faster compared to the serial counterpart by a large margin, as we were able to see in fig. 1 the serial algorithm is on a scale of minutes while the parallel algorithm takes less than a minute.

An interesting thing to notice is that using different dataset we discovered that in lower sized documents having sparse data takes less time, but the higher the size gets the better having dense data becomes, while in this case it's not relevant as it's on the scale of 0.5 seconds for larger datased with bigger documents it might be important to note.

As we can see in fig. 2 we have the comparison between different placements of the processes using the same amount of cores to have equal computational power and as expected we notice that having one chunk or chunks, but on the same node(place=pack) is better than having it scattered, this is due to the communications.

In fig. 3 we have an unexpected twist as we compare how using different cores affects the ramp-up, we would expect to see better results as the number of cores grows but as we can see it's not the case as the ramp-up was obtained using only 10 cores and the worts obtained using 50 cores, this is most likely caused by the accesses to the file, having each process write its own results affects a lot this results, if we were to have only one process store all the results we would most likely see that the more cores the faster.

To conclude, the results are strictly connected to the implementation and in the future trying to implement it using the other options described in the design section of the parralel solution could lead to interesting results to compare.

One thing that we wanted to implement is the threading inside the parallelization, the main idea is to use parallel implementation as baseline,divide further the chunk of data, and use OpenMP to have each process create a defined amount of thread to encode the little chunk of data, this would probably lead to small improvements but could be interesting to implement and optimize further.