High Performance Computing for Data Science

Project report
2021/2022

Alessandra Morellini, Nicola Arpino

UNIVERSITÀ DI TRENTO
Department of Information Engineering and Computer Science

Contents

1 Introduction

The chosen challenge for the HPC course project was parallelizing Huffman coding. Huffman codes can be used to minimize number of bits required for a transmission of a certain message, typically over network. The idea of this code came to David. A. Huffman while he was a Sc.D student at MIT and was published in the 1952 in a famous paper[1].

When transmitting a message, each character of it is represent by a fixed number of bits. The idea behind this encoding method is that the characters can be differentiating based on their frequency and send using a variable length code. Using this approach a character with high frequency will be send with a shorter code with respect to a less frequent character. The saving depends on the data but may range between 20% and 90%. The cost using this coding algorithm can be evaluated using the following equation:

$$B(t) = \sum c.freq \cdot d_T(c) \tag{1}$$

Where `C` is the alphabet from which the characters are drawn, `c.freq` is the frequency of character `c` and `dT(C)` is the depth of the node in the tree. Finally `n` is the number of characters in the data.

1.1 Huffman Codes

A Huffman code is a unique binary string of variable length called `codeword`. To retrieve it, Huffman coding uses a binary tree which is created based on a table of frequency previously calculated. Considering a node as an object containing the character and its frequency is possible to describe, through some the following pseudo-code[4], how to build the tree.

Algorithm 1 Classic Huffman pseudo-code

```
1:  $n$ = length(C)
2:  $Q$ = C
3:  for $j \leftarrow 1, n$ do
4:      allocate a new node $z$
5:      $z$.left = $x$ = EXTRACT_MIN($Q$)
6:      $z$.right = $y$ = EXTRACT_MIN($Q$)
7:      $z$.freq = $x$.freq + $y$.freq
8:      INSERT($Q$,$z$)
9:  end for
10: return EXTRACT_MIN($Q$)
```

At line 2 a priority queue is created ordering the nodes by increasing frequency. In lines 4-8 the two minimum nodes `x` and `y` are extracted from the priority queue. They are joint together in a new node `z` which frequency is computed as the sum of the frequencies of `x` and `y`. The two extracted nodes are the right and left child of `z`. The node `z` in then reinserted into the queue and this process is repeated until `i` reach `n-1`. Finally at line 10 the root of the tree is extracted from the queue.
Assuming that `Q` is implemented as a binary *min-heap*, and `n` is number of characters in `C`. At line 2 we can initialize the queue in `O(n)` time. The loop at lines 3-8 execute exactly `n-1` times and since each heap operation requires time `O(lg n)`, the overall loop contributes `O(n lg n)`.

When traversing the tree we assign to the left child of each node the code 0 and 1 to the right child. Encoding is now trivial since each character is associated with a variable length codeword and is only needed to concatenate the codewords. This is possible because Huffman

Codes are prefix codes which means that no codeword is a prefix of the following codeword. For this reason the start of the encoded string is unambiguous and permits to directly translate back the encoded data.

1.2 Example

To better understand how the previous code works, consider the following example. As first step we imagine to have an input string with a total length of 100.000 characters that contains only characters a-f. Considering an encoding that uses fixed length codewords, we will need 3 bits to represent all the six characters. The total cost of the input will be of 300.000 bits. Using Huffman coding this cost decreases. First of all we have to retrieve the frequencies that are shown in the following table.(The frequency are expressed in thousand)

char	a	b	c	d	e	f
frequency	45	13	12	16	9	5

Given this data we can proceed creating the Huffman tree using the pseudo code above. The following image show all the steps to retrieve the final tree.

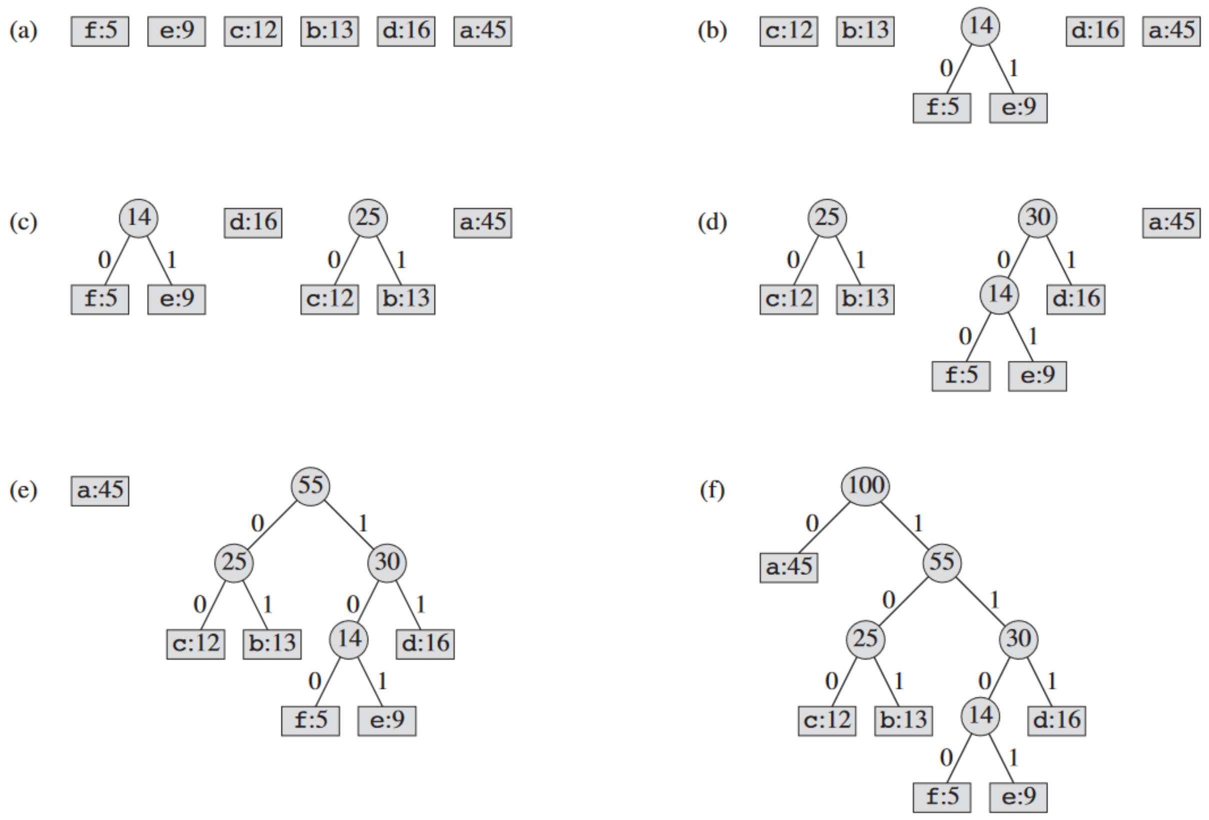


Figure 1: Steps to generate the Huffman tree, starting from the priority queue of frequencies

In figure 1 the sub-image (a) show the priority queue before the loop cycle. From this queue the two nodes with minimum frequency are extracted and became the right and the left child of a new node `z` which is then reinserted into the queue. This can be seen in sub-figure (b). This process is repeated until remain only one node in the queue that is now the root of the final tree as shown in sub-image (f).

Using the equation 1 is possible to calculate the cost of this encoding technique.

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1000 = 224.000 bits \quad (2)$$

2 Serial code

2.1 Approach

The problem was approached identifying a series of steps:

1. Frequencies calculation
2. Creation of the binary tree
3. Tree traversing to retrieve the codewords
4. Encoding
5. Decoding

2.1.1 Note on implementation

Before moving on describing what above steps consist of, we would like to precisely state our original contribution to the code you are going to see. In essence, the second step is related to the creation of the Huffman binary tree and, in order to do so, some data-structures are needed. The code required for this step has been taken from a public implementation [1] and modified for our purposes meanwhile the code for all the others steps is from our side.

2.2 Implementation

2.2.1 Frequencies

The function `calculate_frequencies` defined in the header file `frequencies_utils.h` handles the computation of characters frequency. It takes as parameters the alphabet which is an array of allowed characters, the input string to be encoded and an output buffer where to store the frequencies.

```
void calculate_frequencies(char *alphabet, char *input_string,
int *out_buffer){
    int i, idx = 0;
    char *ret;

    /* Finding occurencies */
    for (i = 0; i < strlen(input_string); i++)
    {
        ret = strchr(alphabeth, input_string[i]);
        idx = strlen(alphabeth) - strlen(ret);
        out_buffer[idx] += 1;
    }
}
```

The function cycles through the input string and for each character retrieves a pointer to its position into the input string. To do so uses the `C` function `strchr()`. Then retrieves the character's index and increases the corresponding frequency at the `idx` position in the array `out_buffer` which has the same size of `alphabet`. This allows to have a relation between the two arrays. In a certain index of `alphabet` there is the characters whereas at the same index of `out_buffer` there is its corresponding frequency.

[1] https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/

2.2.2 Huffman tree

As stated before, in order to generate the Huffman tree we took advantage of a public implementation. Such a generation can be started via `HuffmanCodes()` function. It takes as parameters the data array that contains the alphabet, the array of frequencies previously calculated and their size. It returns the pointer to the root of the tree.

```
struct MinHeapNode *HuffmanCodes(char data[], int freq[], int size)
{
    // Construct Huffman Tree
    struct MinHeapNode *root = buildHuffmanTree(data, freq, size);
    return root;
}
```

The function builds the Huffman tree using a series of utilities functions that aren't shown to the caller. The complete functionalities and implementation details of those methods can be found in the file `tree_utils.c`.

2.2.3 Tree traversing

To allow the encoding phase to perform the character encoding with a time `O(n)` the tree is traversed one time and the code-words are saved in a special data structure.

```
struct nlist
{
    char name; /* defined char */
    char code[CODES_LEN]; /* code */
};
struct nlist codes_list[HASHSIZE];
```

The structure `nlist` contains two fields, the first one called `name` contains the character and `code` is a string containing the corresponding codeword. An object of this struct type represents a node of the tree and are stored into an array.

To traverse the tree and gather the codewords (create the actual mapping table) a recursive function is used which takes as input parameters the pointer to the root of the tree, the char array where store the partial codewords and the current level of the tree.

```
void FillCodesList(struct MinHeapNode *root, char arr[], int top)
{
    // Assign 0 to left edge and recur
    if (root->left){
        arr[top] = '0';
        FillCodesList(root->left, arr, top + 1);
    }

    // Assign 1 to right edge and recur
    if (root->right){
        arr[top] = '1';
        FillCodesList(root->right, arr, top + 1);
    }

    if (isLeaf(root)){
        arr[top] = '\0';
        install(root->data, arr);
    }
}
```

2.2.4 Encoding

Once the tree is traversed and the code_list array is complete e.g the codeword table is ready, is possible to proceed with the actual encoding.

```
char *calculate_huff_code(char *in_str)
{
    int i = 0, code_len = 0, buff_len = 10;
    char *code;
    char *out_string = (char *)calloc(buff_len, sizeof(char));
    int len = strlen(in_str);
    for (i = 0; i < len; i++)
    {
        code = codes_list[hash(in_str[i])].code;
        code_len = strlen(code);
        if ((buff_len - strlen(out_string)) <= code_len)
        {
            buff_len += (code_len * SYMBOL_MAX_BITS);
            out_string = (char *)realloc(out_string, buff_len);
        }
        strcat(out_string, code);
    }
    return out_string;
}
```

The function calculate_huff_code() cycles over an input string and, for each literal, extracts the corresponding codeword from the code_list array. The extraction of the codeword requires a time O(1) due to the hash function which returns an index based on the character position w.r.t the alphabet, avoiding collisions. The total computation time is O(n) with n as length of the input string.

2.2.5 Decoding

Decoding: During the decoding phase each bit of the encoded binary string is evaluated and, based on its value, the pointer of the output decoded string moves to the right or to the left child of the current node. Once a leaf is reached the character in that node is extracted and concatenated to the decoded string.

```
for(i=0; i<len; i++)
    {
        if(final_string[i] == '0' && node->left != NULL){
            node = node->left;
        }else if(node->right != NULL){
            node = node->right;
        }
        if(isLeaf(node)){
            strncat(decoded_string, &node->data, 1);
            node = root;
        }
    }
```

3 Parallelization

3.1 Design of the parallel solution

The proposed solution adopts an *hybrid* parallelization that employs both *openMPI* and *openMP*. The development followed an incremental pattern: once the application has been parallelized with OpenMPI, OpenMP directives were added gradually to the most intensive part of the code. In general, the **Master-only syle** has been enforced: only master thread performs MPI calls. Of course this solution is not as portable as pure MPI code nor easier to maintain[2], but allowed us parallelize the whole application. In particular:

- **Frequencies calculation** ⇒ MPI
- **Binary tree calculation** ⇒ Serial
- **Tree traversing: codewords calculation** ⇒ OpenMP
- **Encoding** ⇒ MPI
- **Decoding** ⇒ OpenMP

The only part of the serial application that cannot be parallelized is the tree generation, which is also the reason why we adopted a public implementation rather than developing our own. The reason behind our statement is based on the fact that *is not* possible to generate and combine sub-trees that lead to a correct tree generation.
Recall that typical strategy to manage trees in parallel is to exploit **divide-et-impera**, a good example can be found here[5]. The main issue in this case is that even if the tree could be divided in two sub-trees, the data cannot. In our opinion a correct tree is the one that associate to the character having higher frequency the shortest possible codeword. When dividing the frequency array in two or more partitions to create sub-trees and them merge them we may encounter the following situation: the higher frequency character still has a shorter codeword with respect to other characters but *is not* the best possible option. This increases the total cost of transmitting the message which is still lower than using fixed-length codewords but in our opinion do not generate a valid tree. For this reason we decided not to implement a parallel solution. In the following paragraph there is a demonstration of the cost increase and the corresponding tree. figure 2 shows the two sub-trees whereas figure 3 shows the final tree using the *divide-et-impera* approach.

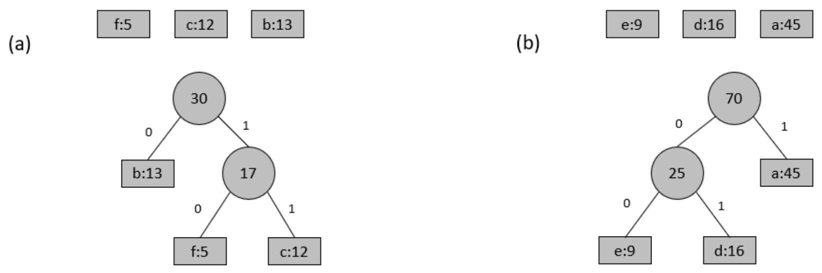


Figure 2: Two sub-trees generated from the divided frequency array

[2]http://www.intertwine-project.eu/news/2017/best-practice-guide-program-hybrid-mpi-openmp-17-03-30

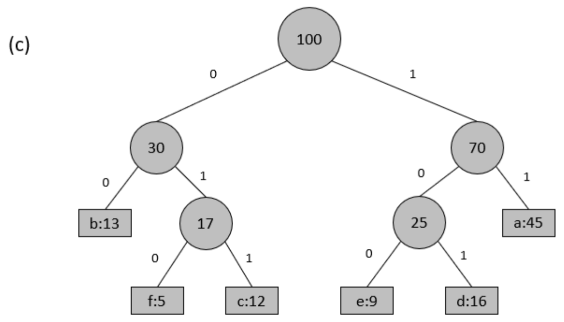


Figure 3: Huffman tree using divide-et-impera approach

The cost of the tree can be evaluated using equation 1.

$$(45 \cdot 2 + 13 \cdot 2 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 3 + 5 \cdot 3) \cdot 1000 = 242.000 bits \tag{3}$$

As written previously the cost increases slightly using this approach, but is important also to consider that the data used for his example is only about 100.000 characters whereas during our test we used data containing up to 800.000 characters. The cost in that case is still lower than using fixed-length codewords but the increase cannot be disregard.

Before analysing in depth the implementation choices for the parallel code an high level structure is presented.

```
int main(){
    MPI datatype initialization
    variable and struct declaration
    if(rank == 0){
        // 1) only process 0 read data from the input text file.
        // 2) generation of parameters for MPI_Scatterv
    }
    // 3) execution of the collective MPI_Scatterv to spread the input string
    // 4) calculation of frequencies - MPI
    // 5) execution of MPI_Reduce to collect partial results into process 0
    if(rank == 0){
        // 6) generation of the Huffman tree - serial
        // 7) generation of code-table: tree traversing - openMP
    }
    // 8) encoding: computation of Huffman code on the piece of the string
       received - MPI
    // 9) execution of MPI_Gatherv to send back to process 0 the encoded
       strings
    if(rank == 0){
        // 10) decoding: execute decoding of the gathered string - openMP
    }
}
```

3.2 Implementation

3.2.1 Frequency

The parallel version of the frequency calculation requires that the input string is divided among the available processes. To do so the `MPI_Scatterv` function was used since in some occasions the numbers of characters in the input string is not perfectly divisible between the number process. When this situation occurs the last process receives the remainder of this split. Once each process has received its own piece of the string, they proceed to execute the calculation of the frequency. The function is the same coded in the serial version 2.2.1 but is called from the various processes. Finally each process calls the `MPI_Reduce` function with the parameter `MPI_SUM` and destination 0.

```
MPI_Scatterv(input_string, sendcount, displs, MPI_CHAR, recv_buff,
RECV_SIZE, MPI_CHAR, 0, MPI_COMM_WORLD);

calculate_frequencies(alphabeth, recv_buff, frequencies);

MPI_Reduce(frequencies, reduce_buff,
sizeof(frequencies) / sizeof(int), MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

The process zero has the complete frequency array and proceed to generate the tree.

3.2.2 Tree traversing

The tree traversing is done only by process 0 which is the one that has executed the tree generation. To increase the performance this process is parallelized using openMP. The parallel solution can be achieved in two ways. The first one that was identified handles the recursive function using openMP tasks.

```
void FillCodesList(struct MinHeapNode *root, char arr[], int top)
{
    if (root->left)
    {
        #pragma omp task firstprivate(root, arr, top) depend(out : top)
        {
            #pragma omp critical
                arr[top] = '0';

            FillCodesList(root->left, arr, top + 1);
        }
    }
    if (root->right)
    {
        #pragma omp task firstprivate(root, arr, top) depend(in : top)
        {
            #pragma omp critical
                arr[top] = '1';

            FillCodesList(root->right, arr, top + 1);
        }
    }
    #pragma omp taskwait
    if (isLeaf(root))
    {
        arr[top] = '\0';
        install(root->data, arr);
    }
}
```

The function is called from process 0 in this way:

```
#pragma omp parallel
{
    #pragma omp single
            FillCodesList(root, arr, top);
}
```

This approach takes advantage of a `work-oriented` paradigm introduced from version 4 of openMP. It is based on a `producer-consumer-paradigm`. In general tasks are *produced* and *consumed*, in any case they share a single task-queue. In particular we adopted the `single-producer multiple-consumer`: within the parallel region, a single thread produces many tasks, then once a task is generated any of the threads in the pool can pop one from the queue and execute it. This task-based model is suitable to manage *recursive* algorithms. In addition tasks offers better synchronization mechanism respect to `sections` as well documented here[2]. Each time a recursion is needed the process generates a new task. This approach had some limitation, e.g. the *depend* clause. (More information of depend clause can be found at OpenMp specification [3]). In our implementation means that the `depend(in:top)` clause will generate a task dependent on its sibling task with `depend(out:top)` clause and will be executed only after the sibling task is completed. This behaviour is required in order to traverse the tree in consistent way e.g producing a valid binary assignment to edges. Moreover, the `#pragma omp taskwait` clause causes some performance decrease since enforces that a thread needs to wait the end of computation of its children. The overall benefit in terms of speedup using this method was not very relevant due to this clauses, which led us to try another approach. In this other case the binary tree was divided only in two partition. Two tasks are needed and each of them takes one of the two sub-trees. The code executed is the same seen in paragraph 2.2.3 with the only difference that is done twice over two sub-trees, instead of once over the overall tree.

```
 #pragma omp parallel firstprivate(arr, top) num_threads(2)
{
    if (omp_get_thread_num() == 0)
    {
        arr[top] = '1';
        FillCodesList(root->right, arr, top + 1);
    }
    else if (omp_get_thread_num() == 1)
    {
        arr[top] = '0';
        FillCodesList(root->left, arr, top + 1);
    }
}
```

This methods has better performance when the tree is balanced, which usually happens when the characters in the input string are several and some of the frequency are similar. In the case when the tree is not balanced the performance are similar to the serial version.

3.2.3 Encoding

To execute the encoding each process accesses the array resulting form the tree traversing. All the process execute the MPI_Bcast() function to propagate this data structure.

```
MPI_Bcast(codes_list, 1, mpi_codelist, 0, MPI_COMM_WORLD);
```

Is interesting to notice that the datatype send with the broadcast is a derived data type since code_list is an array of struct. Is possible to recalling the declaration of code_list in paragraph 2.2.3. The datatype used in the broadcast is called mpi_codelist and consist of a contiguous allocation of mpi_codeblock which map the structure of struct nlist.

At this point each process owns the divided input string that was already sent using the MPI_Scatterv function and the mpi_codelist array from which extract the codewords. The processes proceed calling the function calculate_huff_code which has the same implementation as the serial code in paragraph 2.2.4. Similar to the implementation of parallel frequency calculation, the function per se is not parallelized but is called by the various processes over a smaller string.

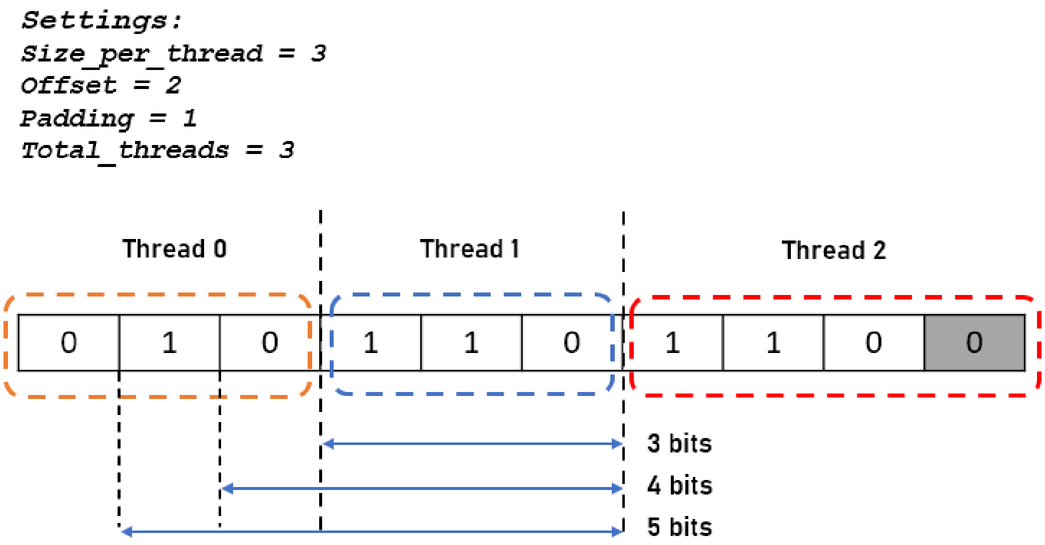
```
//recv_buff contains the string received using MPI_Scattev
out = calculate_huff_code(recv_buff); //call to function done by each process
```

An interesting challenge was faced when gathering all the encoded string. They need to be send to process zero with the exact order they were send. To do so is possible to use the function MPI_Gather, but each of them has a different length that depends on the input string and the codeword, which cannot be known before the encoding itself. To solve this problem each process needs to communicate two times, first of all using MPI_Gather to send to process zero the various length of the encoding string. Then is possible to generate two arrays that represent the number of elements received from each process and displacements needed to collect and combine the complete encoded string. After that each process can call the function MPI_Gatherv and communicate their own encoded string.

```
//counts contains the length of each encoded string
//gather_disps is the array of displacement
int counts[world_size], gather_disps[world_size], i;
int nelem = strlen(out);

MPI_Gather(&nelem, 1, MPI_INT, counts, 1, MPI_INT, 0, MPI_COMM_WORLD);

for (i = 0; i < world_size; i++)
    gather_disps[i] = (i > 0) ? (gather_disps[i - 1] + counts[i - 1]) : 0;

MPI_Gatherv(out, nelem, MPI_CHAR, final_string, counts, gather_disps, MPI_CHAR
    , 0, MPI_COMM_WORLD);
```

3.2.4 Decoding

The decoding has been parallelized using openMP. The main problem we had to address was about the correct splitting of the encoded string. A naive solution relies again on *divde-et-impera*: distribute a piece of encoded string to each thread and then, on a collector thread/process reconstruct the original input string. The problem is that such a division of the encoded data might produce bad decoded literals. There is no guarantee that the initial binary value of each piece of encoded string encodes to a valid code-word. Each thread might have to ”skip” dirty bits up to a synchronization point. Assuming that *thread 0* starts from the initial (left-most) position of the encoded string, it starts reading and decoding a valid string. When should it end? How is it possible to synchronize decoded chunks from all threads?
To answer these questions, let’s have a look to the following figure.



```
Settings:
Size_per_thread = 3
Offset = 2
Padding = 1
Total_threads = 3
```

Figure 4: Description of decoding synchronization mechanism

In the above figure, the scenario is described with help of *Settings*. As you can see we assume to have an encoded string of ten bits. Since we employs three threads, the sub-string that each thread takes care of is 3 bits long. The last thread is in charge of reading one bit more (padding, gray box) which is the reminder of the division. The most important parameter is the **offset**. Each thread is able to calculate its own initial and final index w.r.t the input string. Moreover all threads excluded the first one, calculates exactly `offset + 1` candidate solutions. The first thread is in charge of decoding three bits; it might happen that only two out of three bits encode a valid codeword. This said, each thread, will save the number of *dirty* bits e.g the number of bits that do not decode to a codeword. For example, *thread 1* decodes three candidate strings, each one is one bit longer. Thanks to this procedure, each thread has an overlapping window that allows it to decode bits that might have been excluded by the previous thread; this is the base of the synchronization mechanism. As you may have already guessed, threads cannot know in advance how many bits the previous thread excluded, that is why candidate solutions are needed. The value of **offset**, in this case two, should be large enough to allow a thread to decode at least one valid code-word. In fact dirty bits can be present when shifting the original window size that each thread calculated. Such a number can be set equal to: $\log_2 length(alphabet)$; this is an upper bound w.r.t how many bits are needed to encode a symbol of a given alphabet.
Above procedure is implemented in the function `decode_string`.

```
struct decoded_node *decode_string(struct MinHeapNode *root, char *in_string,
    int size_per_thread, int padding, int total_threads, int offset)
{
    /* We omit here some configuration variables, below there is the
     * indices calculation performed by each thread based on their rank
     */
    int initial_offset, end_offset, i, k, local_initial_offset;
    initial_offset = thread_rank * size_per_thread;
    end_offset = initial_offset + size_per_thread;

    /* Last thread will manage also padding */
    if (total_threads == thread_rank && padding > 0)
        end_offset += padding;

    /* Each thread will decode many candidate decode-strings. How many? up to
         'offset' */
    for (k = 0; k < offset; k++)
    {
        /* Those indices makes the synchronization strategy for each thread
            */
        local_initial_offset = initial_offset - k;
        local_string = calloc(len, sizeof(char));
        int last_literal_index = 0;
        node = root;
        for (i = local_initial_offset; i < end_offset; i++)
        {
            /* This part is similar to the serial code
             * Based on the node value '0' or '1', the tree is
             * traversed on the left or on the right
             */
        }
        /* Saving 'dirty' bits value and decoded string */
        d_node[k].string = (char *)calloc(sizeof(char), len);
        strncpy(d_node[k].string, local_string, len);
        d_node[k].padding_bits = (i - last_literal_index - 1);

        /* Thread 0, since started from beginning produces only one string */
        if (thread_rank == 0)
            break;
    }
    return d_node;
}
```

Below there is a fragment of code that the MPI *process 0* executes, enforcing the *master-only-style*. Such process uses a parallel region in which calls the decoding function a number of times equals to the number of available threads.

```
        struct decoded_node **decoded_list = (struct decoded_node **)malloc(
            sizeof(decoded_list) * thread_count);
        char *final_decoded_string = (char *)calloc(len, sizeof(char));
        /* Parallel decoding */
        tstart = omp_get_wtime();
        /* Parallel region */
        #pragma omp parallel
        {
                printf("Thread␣num␣%d\n", omp_get_thread_num());
                decoded_list[omp_get_thread_num()] = decode_string(root,
                    final_string, size_per_process, padding, thread_count - 1,
                    offset);
        }
        tstop = omp_get_wtime();
```

Merging single threads contributions is, at this point, straightforward. As you may remember form Figure 4, *thread 0* is different from others because decodes just one string. It is also ”privileged” because it starts decoding from the beginning of the encoded string therefore does not need to synchronize. Merging is done at *process 0* and it is a serial operation.

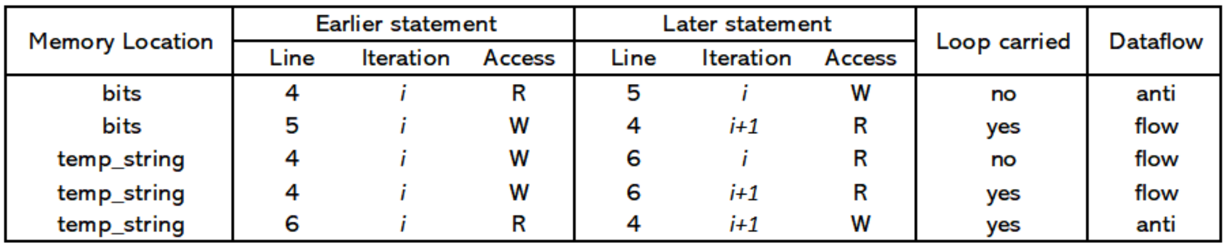
```
        /* Thread 0 token is special, getting bits */
        temp_string = decoded_list[0][0].string;
        bits = decoded_list[0][0].padding_bits;
        strncat(final_decoded_string, temp_string, strlen(temp_string));

        /* Other threads tokens */
        for (i = 1; i < thread_count; i++)
        {
            temp_string = decoded_list[i][bits].string;
            bits = decoded_list[i][bits].padding_bits;
            strncat(final_decoded_string, temp_string, strlen(temp_string));
        }
```

After the bits offset is read from the thread 0 contribution, the merging phase requires to loop over the number of threads contributions and, for each of them, select among the candidate solutions, the one that considers the right ’dirty’ value of bits w.r.t the previous thread.

3.3 Dependencies analysis

In the last fragment of code presented the chapter before, there is a *for-loop* which manages threads contributions which has not been parallelized. In case we would like to apply some openMP parallelization, the following analysis should be taken into account.

```

for (i = 1; i < thread_count; i++)
        {
            temp_string = decoded_list[i][bits].string;
            bits = decoded_list[i][bits].padding_bits;
            strncat(final_decoded_string, temp_string, strlen(temp_string));
        }
```



Memory Location	Earlier statement			Later statement			Loop carried	Dataflow
	Line	Iteration	Access	Line	Iteration	Access		
bits	4	*i*	R	5	*i*	W	no	anti
bits	5	*i*	W	4	*i+1*	R	yes	flow
temp_string	4	*i*	W	6	*i*	R	no	flow
temp_string	4	*i*	W	6	*i+1*	R	yes	flow
temp_string	6	*i*	R	4	*i+1*	W	yes	anti

Figure 5: Dependencies table

The variable `temp_string` is associated to both flow and anti-dependence. In case openMP parallel for directive is applied to such loop, a simple way to eliminate those dependencies would be via changing the scope of the variable making it to `private`. The flow dependence related to the variable `bits` is instead hard to remove. That variable value depends on the previous iteration but such value is *non-deterministic*. Is not possible to pre-compute those values because they not depend on a simple function nor on the loop index; so it is not possible to fill any supporting vector. For that reason the loop above cannot be parallelized with openMP.

4 Benchmarks

4.1 Encoding

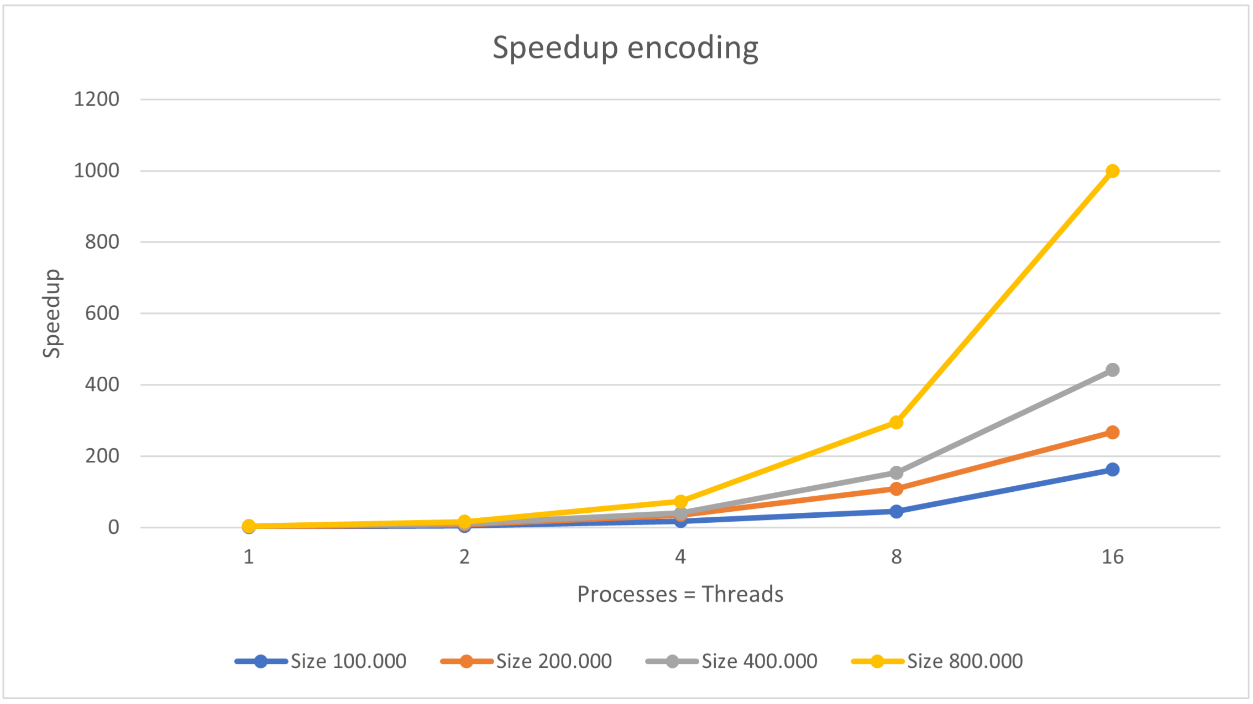


Figure 6: Speedup of encoding with different problem size.

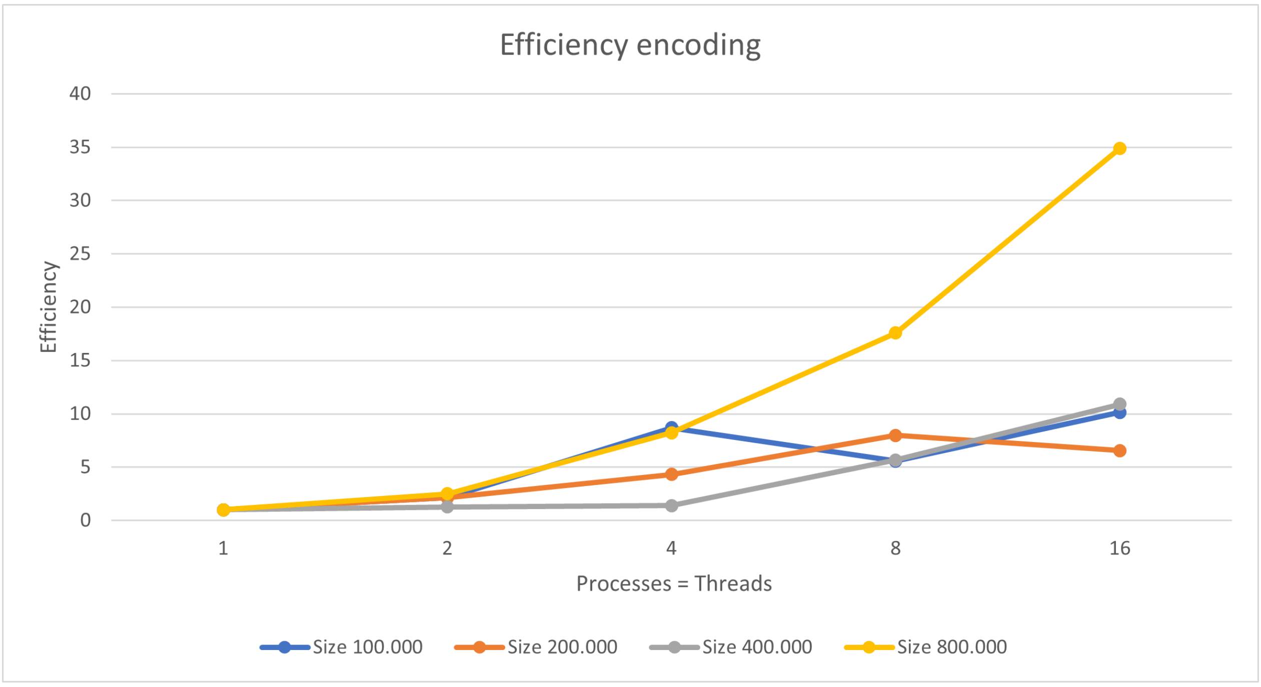


Figure 7: Efficiency of encoding with different problem size.

4.2 Decoding

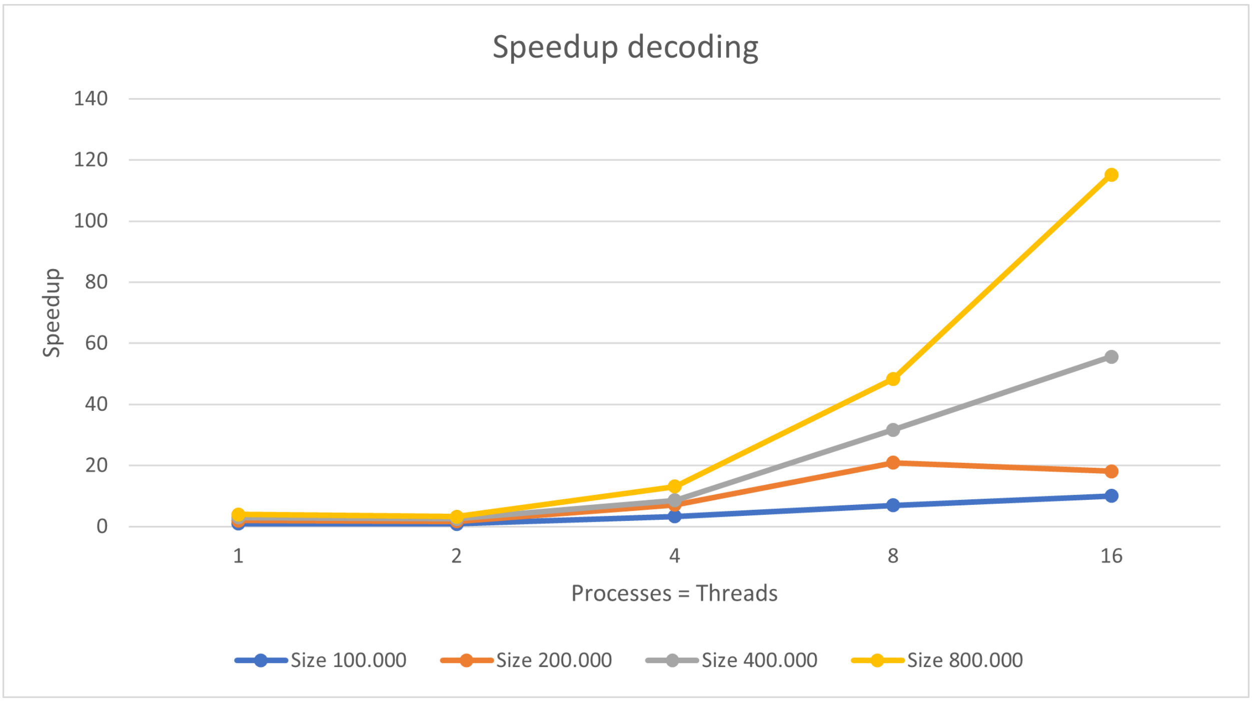


Figure 8: Speedup of decoding with different problem size.

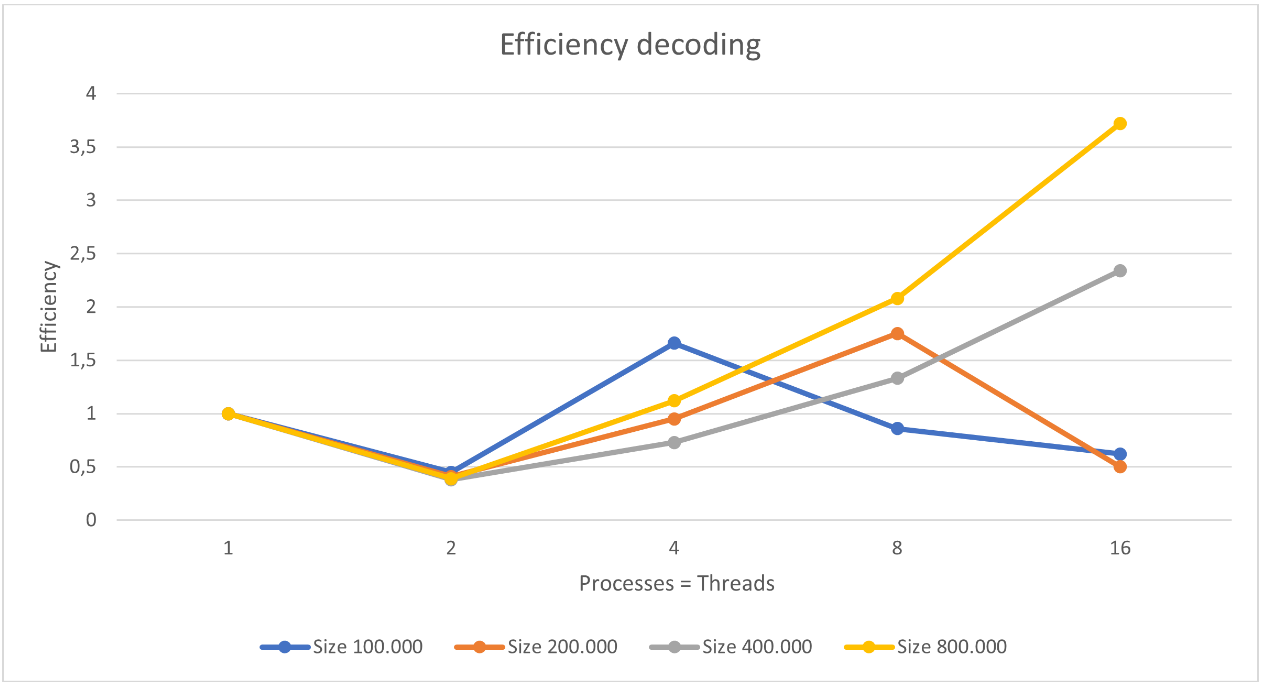


Figure 9: Efficiency of decoding with different problem size.

4.3 Final thoughts

To sum up our work, we were able to parallelize most of the application serial code. The overall performances in terms of speedup and efficiency are interesting and surprisingly we were able to reach a superlinear improvement. There is still margin to build a better algorithm if space requirements would be stricter. In this case we aimed to achieve the best time reduction and, as usually happens, a trade off between space and time must be found.

References

[1] David A. Huffman. “A Method for the Construction of Minimum-Redundancy Codes”. In: (1952).

[2] Andrea Marongiu - UniMoRe Paolo Burgio. *Tasking in OpenMP*. 2020. URL: `http://algo.ing.unimo.it/people/andrea/Didattica/HPC/SlidesPDF/10.%20OMP%20tasks.pdf` (visited on 01/11/2022).

[3] OpenMP API Specification. *depend Clause*. URL: `https://www.openmp.org/spec-html/5.0/openmpsu99.html`.

[4] Thomas H. Cormen. Charles E. Leiserson. Ronald L. Rivest. Clifford Stein. *Introduction to Algorithms. Third Edition.* The MIT Press, 2009.

[5] Mike Bailey - Oregon university. *OpenMP Tasks*. 2021. URL: `https://web.engr.oregonstate.edu/~mjb/cs575/Handouts/tasks.1pp.pdf` (visited on 01/11/2022).