Parallel Lossless Data Compression based on the Burrows-Wheeler Transform

Advanced Algorithms and Parallel Programming AY 2017/2018

prof. Ferrandi Fabrizio

ing. Lattuada Marco

POLITECNICO
DI MILANO

Stefano Valladares stefano.valladares@mail.polimi.it

Object of the presentation

The work has been carried out from the analysis of the algorithm described in the section 2.3.2 of the paper:

"Parallel Lossless Data Compression Based on the Burrows-Wheeler Transform", written by Jeff Gilchrist and Aysegul Cuhadar.

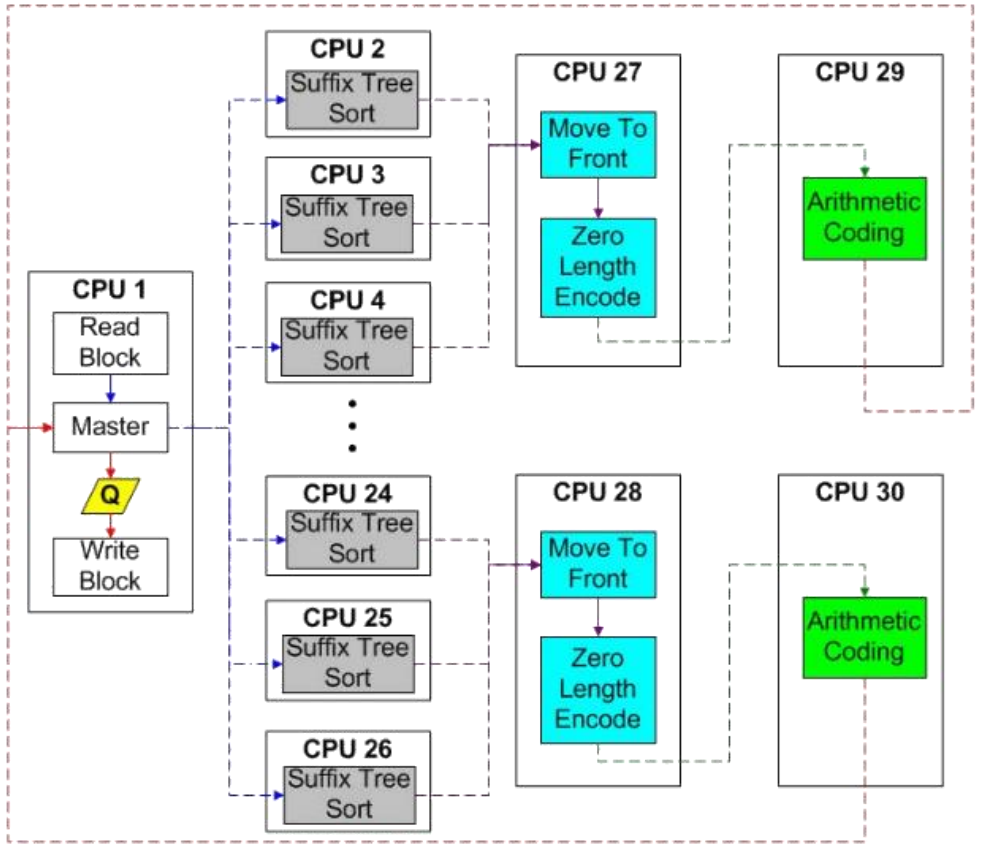
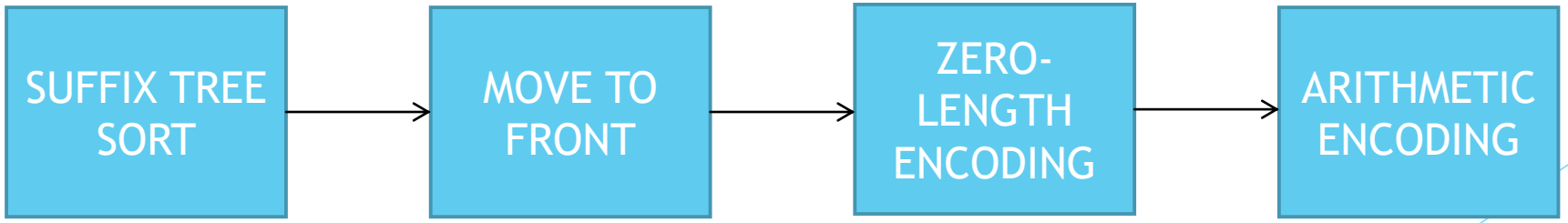


Figure 2. Message-passing bwtzip 3-stage pipeline flow diagram with 30 CPUs

The algorithm

The algorithm uses a block-sorting technique for compression. The file is initially split into blocks and then each block is processed through 4 stages.

1. Burrows-Wheeler Transform
2. Move to front Transform
3. Zero-length encoding
4. Arithmetic encoding



INPUT

The input file is divided into blocks.

Each block is saved as an array of unsigned char before entering the pipeline.

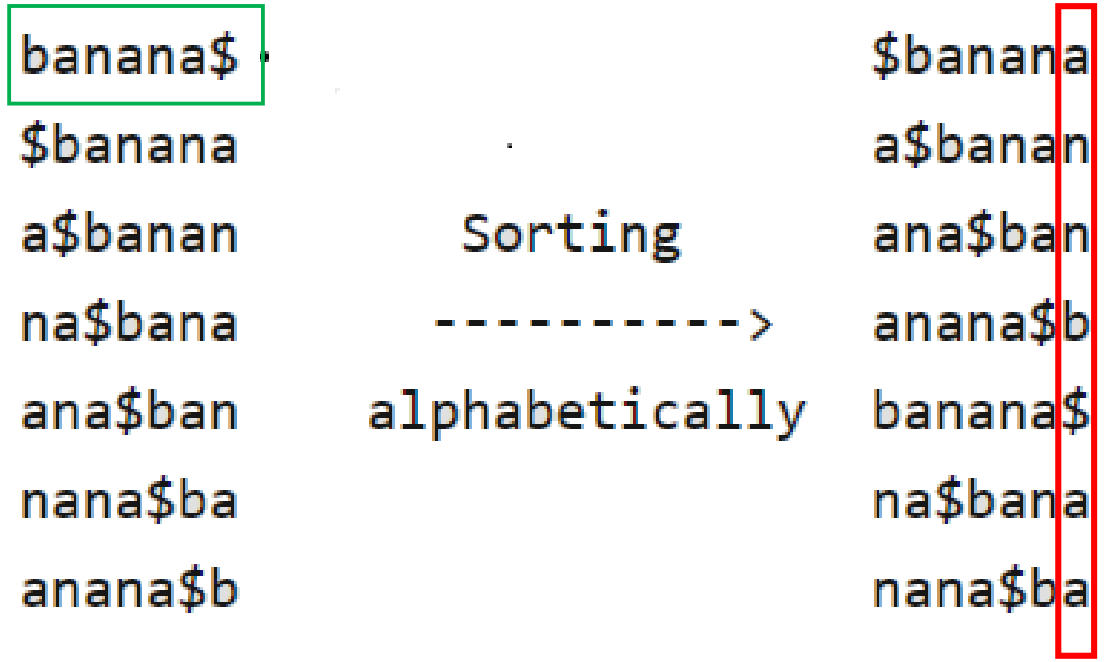
The id is used to keep track of the order of blocks for unsynchronized execution.

```
#define MAX_CHUNK_SIZE          (5 * 1024 * 1024)
#define DEFAULT_CHUNK_SIZE      (0.9  * 1024 * 1024)
#define MIN_CHUNK_SIZE          (300)

typedef struct Text
{
        unsigned char *text;
        size_t len;
        long id;

} Text;
```

1) Burrows-Wheeler Transform

The BWT is a lexicographical reversible permutation of the input string so that similar characters are grouped together.

- The transform is done by sorting all the circular shifts of the text.
- Often the output of the BWT is easier to compress.



1) Burrows-Wheeler Transform(cont.)

IMPLEMENTATION:

It can be shown that:

sort_circular_shifts(input) == sort_suffixes(input+SENTINEL)

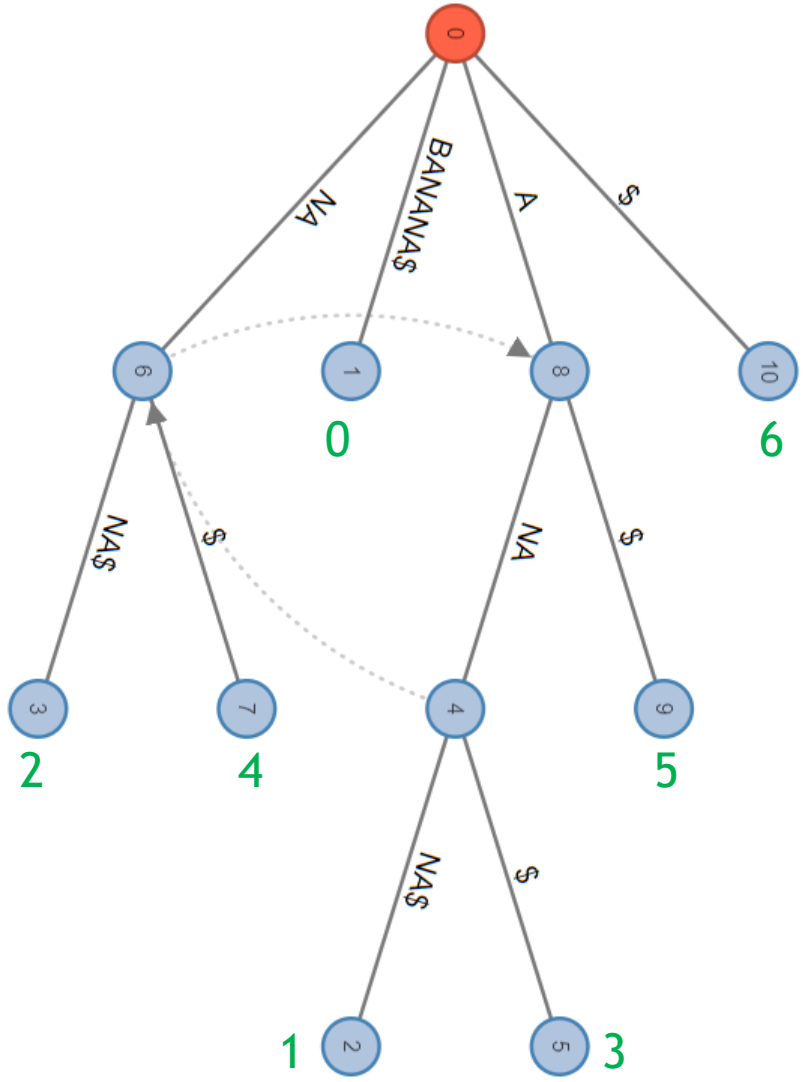
Where SENTINEL is a special character, that must differ from all the others. In the implementation is the number 256.

Sorting suffixes is done by going through the following steps :

1. Construct the suffix tree of the input - (Ukkonen's algorithm)
2. Construct the suffix array – (lexicographic DFS traversal of the suffix tree)
3. Write the output from the suffix array.

1) Burrows-Wheeler Transform(cont.)

Example: BANANA$ ($ is the sentinel index)



Visualization of Ukkonen's algorithm

BWT = T[SA[i] – 1] for each i

i	0	1	2	3	4	5	6
T[i]	B	A	N	A	N	A	$
SA[i]	6	5	3	1	0	4	2
SA[i]-1	5	4	2	0	6	3	1
BWT	A	N	N	B	$	A	A

index: 3 6

6	$
5	A$
3	ANA$
1	ANANA$
0	BANANA$
4	NA$
2	NANA$

Suffix Array

1) Burrows-Wheeler Transform(cont.)

COMPLEXITY

- Construct suffix tree ---> O(n)
- Construct suffix array ---> O(n)
- Output BWT ---> O(n)

OUTPUT

PRIMARY_INDEX + SENTINEL_INDEX + BWT(input + '$')

- The indexes are encoded as a sequence of 4 bytes each that represents the number big-endian.
- The output is 9 bytes longer.

2) Move to front transform

The MTF is a reversal transform of a sequence of input characters to an array of output numbers.

The output data has more local correlations and therefore is easier to compress.

IMPLEMENTATION

- For each byte in input, replace it with its index in the list of bytes and then move the value of the current index to the front of the list.
- The use of a linked list makes the cost of performing the move to front operation constant.

COMPLEXITY: O(n)

3) Zero-length Encoding

The ZLE is a specific run-length encoding, used by Wheeler, in which sequences of zeros are encoded in a number that holds the length of the zero-run.

- After the first two stages long sequences of zeros are frequent.
- The length of a sequence is encoded in a binary number.

COMPLEXITY: O(n)

Input stream	length	output code
$x0y$	1	$x0y$
$x00y$	2	$x1y$
$x000y$	3	$x00y$
$x0000y$	4	$x10y$
$x0000000y$	7	$x000y$

Examples of Wheeler's run-length coding

IMPLEMENTATION

Given a sequence of zeros of length n:

1. Convert (n+1) to binary.
2. Write the binary number in the output, starting from the LSB, leaving out the MSB.

4) Arithmetic encoding

In this stage an adaptive arithmetic encoder implements a lossless compression.

- It is a form of entropy encoder in which the entire input is encoded in an interval of real number between 0 and 1.
- It separates the model for representing data from the encoding of information with respect to that model.

IMPLEMENTATION

Start with an interval [0,1] and a model for the probability of occurrence of each possible byte in input, then for each byte:

1. Divide the current interval into subintervals with respect to the probability model.
2. Select the interval of the current byte as the new interval.
3. Update the probability model.

COMPLEXITY: O(n)

OUTPUT

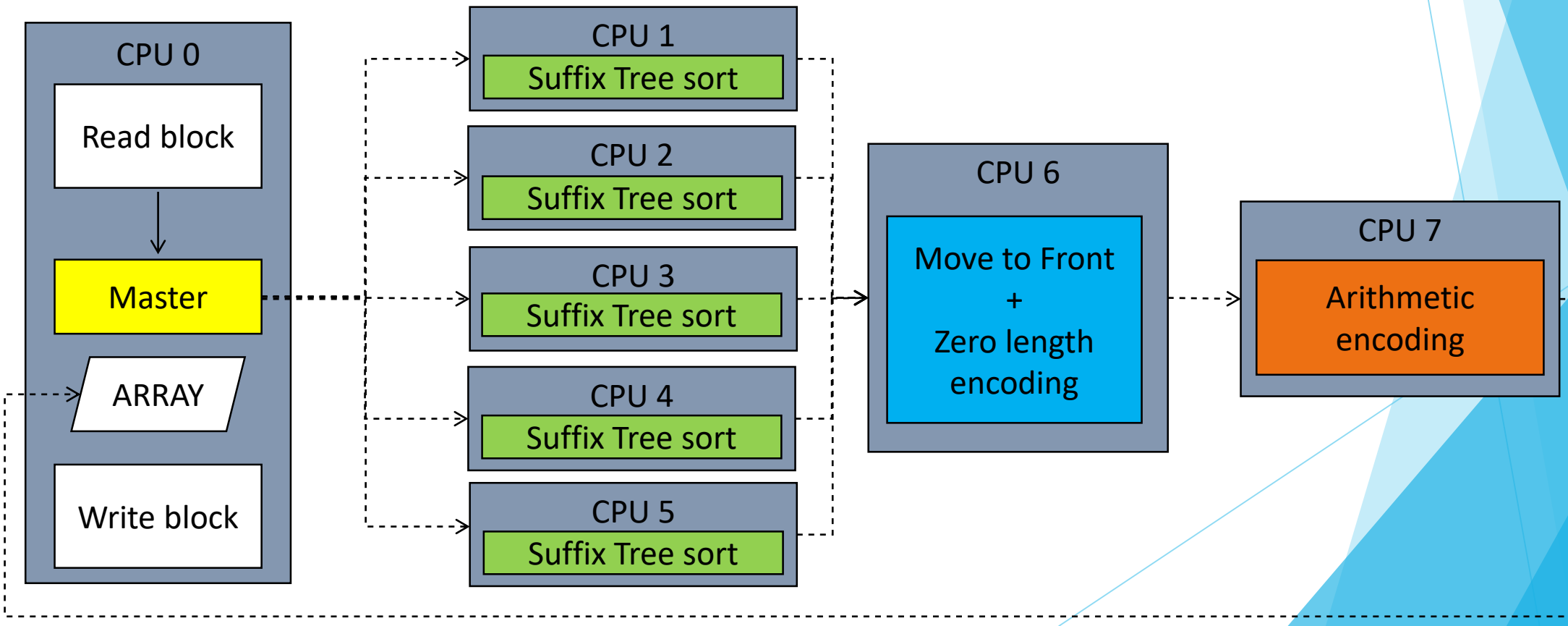
At the end of the pipeline the output of the arithmetic encoder is concatenated with its length, and the result is written into the output file.

OVERALL COMPLEXITY: O(n)

Parallel implementation

The pThread API, defined by the standard POSIX.1c thread extension, has been used to parallelize the algorithm.

The idea is to parallelize the pipeline process so as to assign different stages of the pipeline to different threads.



Parallel implementation(cont.)

SYNCHRONIZATION:

- Threads run asynchronously.
- Queues between stages to exchange values.
- Exclusive access to queues is guaranteed by mutexes, one associated with each queue.
- Signal mechanism to wait when the queue is empty.
- The output is saved into an array before it is written into the output file.

```
typedef struct Buffer
{
        Queue           *queue;
        pthread_mutex_t  mutex;
        pthread_cond_t   cond;

} Buffer;

typedef struct Queue
{
        int counter;
        Elem *front;
        Elem *rear;

} Queue;
```

Threads: Main

It has to deal with two tasks:

1. Read the input file block by block and add it to the read queue.
2. Write the compressed blocks stored in the output array into the output file.

```
for(int j=0; j<nBlocks-6 && !flag; j++) {

        usleep(10000);

        inZip = readFile(input, chunkSize);

        inZip.id = i++;

        pthread_mutex_lock(&readin.mutex);
        enqueue(inZip, readin.queue);
        pthread_cond_signal(&readin.cond);
        pthread_mutex_unlock(&readin.mutex);
}
```

Read loop.

```
while(index < nBlocks) {
        writeOutput(output, &index, littleChunk);
        usleep(1000);
}
```

Write loop.

Threads: BWT

```
void *bwtStage(void *arg)
{
        Text bwtInput;
        while(1) {
                pthread_mutex_lock(&readin.mutex);
                while(empty(readin.queue) && readin.queue->counter < nBlocks)
                        pthread_cond_timedwait(&readin.cond, &readin.mutex, &timeout);

                if(!empty(readin.queue))
                        bwtInput = dequeue(readin.queue);
                else if(readin.queue->counter == nBlocks) {
                        pthread_mutex_unlock(&readin.mutex);
                        break;
                } else {
                        pthread_mutex_unlock(&readin.mutex);
                        continue;
                }
                pthread_mutex_unlock(&readin.mutex);

                Text bwtOutput = bwtTransformation(bwtInput);

                pthread_mutex_lock(&bwt.mutex);
                enqueue(bwtOutput, bwt.queue);
                pthread_cond_signal(&bwt.cond);
                pthread_mutex_unlock(&bwt.mutex);
        }
        return 0;
}
```

READ THE INPUT

STAGE TRANSFORMATION

WRITE THE OUTPUT

Threads: MTF and ZLE

```
void *mtfZleStage(void *arg)
{
        Text mtfInput;
        while(1) {
                pthread_mutex_lock(&bwt.mutex);
                while(empty(bwt.queue) && bwt.queue->counter < nBlocks)
                        pthread_cond_timedwait(&bwt.cond, &bwt.mutex, &timeout);

                if(!empty(bwt.queue))
                        mtfInput = dequeue(bwt.queue);
                else if(bwt.queue->counter == nBlocks) {
                        pthread_mutex_unlock(&bwt.mutex);
                        break;
                } else {
                        pthread_mutex_unlock(&bwt.mutex);
                        continue;
                }
                pthread_mutex_unlock(&bwt.mutex);

                Text mtfOutput = mtf(mtfInput);
                Text zleOutput = zleEncoding(mtfOutput);

                pthread_mutex_lock(&arith.mutex);
                enqueue(zleOutput, arith.queue);
                pthread_cond_signal(&arith.cond);
                pthread_mutex_unlock(&arith.mutex);
        }
        return 0;
}
```

READ THE INPUT

STAGE TRANSFORMATION

WRITE THE OUTPUT

Threads: ARITH

```
void *arithStage(void *arg)
{
        Text arithInput;
        while(1) {
                pthread_mutex_lock(&arith.mutex);
                while(empty(arith.queue) && arith.queue->counter < nBlocks)
                        pthread_cond_timedwait(&arith.cond, &arith.mutex, &timeout);
                if(!empty(arith.queue))
                        arithInput = dequeue(arith.queue);
                else if(arith.queue->counter == nBlocks) {
                        pthread_mutex_unlock(&arith.mutex);
                        break;
                } else {
                        pthread_mutex_unlock(&arith.mutex);
                        continue;
                }
                pthread_mutex_unlock(&arith.mutex);

                Text compressed = encodingRoutine(arithInput);

                result.text[compressed.id] = compressed;
        }
        return 0;
}
```

READ THE INPUT

STAGE TRANSFORMATION

WRITE THE OUTPUT

Out of order execution and in order completion

The main thread continuously calls this function to check if the next block to be output is available.

The index keeps trace of the id of the next block to output.

The id of the result are initialized to -1 so the main thread knows when a compressed block has been stored into the array.

```
void writeOutput(FILE *output, int *const index, const int littleChunk)
{
	unsigned char length[4];
	unsigned char id[1];
	int i = *index;

	for(; i<nBlocks; i++) {

		if(result.text[i].id != *index)
			return;

		encodeUnsigned(result.text[i].len, length, 0);

		if(i == nBlocks - 1 && littleChunk)
			id[0] = 0;
		else
			id[0] = 1;

		writeFile(output, length, 4);
		writeFile(output, id, 1);
		writeFile(output, result.text[i].text, result.text[i].len);

		free(result.text[i].text);
		(*index)++;
	}
}
```

Results

TESTED FILES:

- World192.txt - the CIA world fact book (2.5 MB)
- Bible.txt - the King James version of the bible (4 MB)
- E.coli - complete genome of E.coli bacterium (4.6 MB)
- Dickens - collected works of Charles Dickens (10.2 MB)
- Samba - tarred source code of Samba 2-2.3
- Mozilla - tarred executable of Mozilla 1.0 (51.2 MB)

The first three files come from "The large corpus", the others come from the "Silesia corpus".

TEST MACHINE

- CPU Intel® Core™ i7-6700HQ, Quad-core, 8 Threads, 2.60GHz, 6 MB cache, 16 GB RAM.

Compression performance

Bigger blocks lead to better compression, but at the cost of running the algorithm slower.

File	Size (MB)	Compressed (%)				
		0,1 MB	0,9 MB	1,8 MB	3,6 MB	5 MB
World192.txt	2,5 MB	72,77	79,94	81,19	82,26	82,26
Bible.txt	4 MB	74,46	78,52	79,16	19,64	79,88
E.coli	4,6 MB	74,33	74,60	74,63	74,68	74,71
Dickens	10,2 MB	66,58	71,78	72,73	73,64	74,02
Samba	21,6 MB	75,79	78,13	78,26	78,33	78,35
Mozilla	51,2 MB	62,37	63,73	64,13	64,56	64,66

Sequential vs Parallel

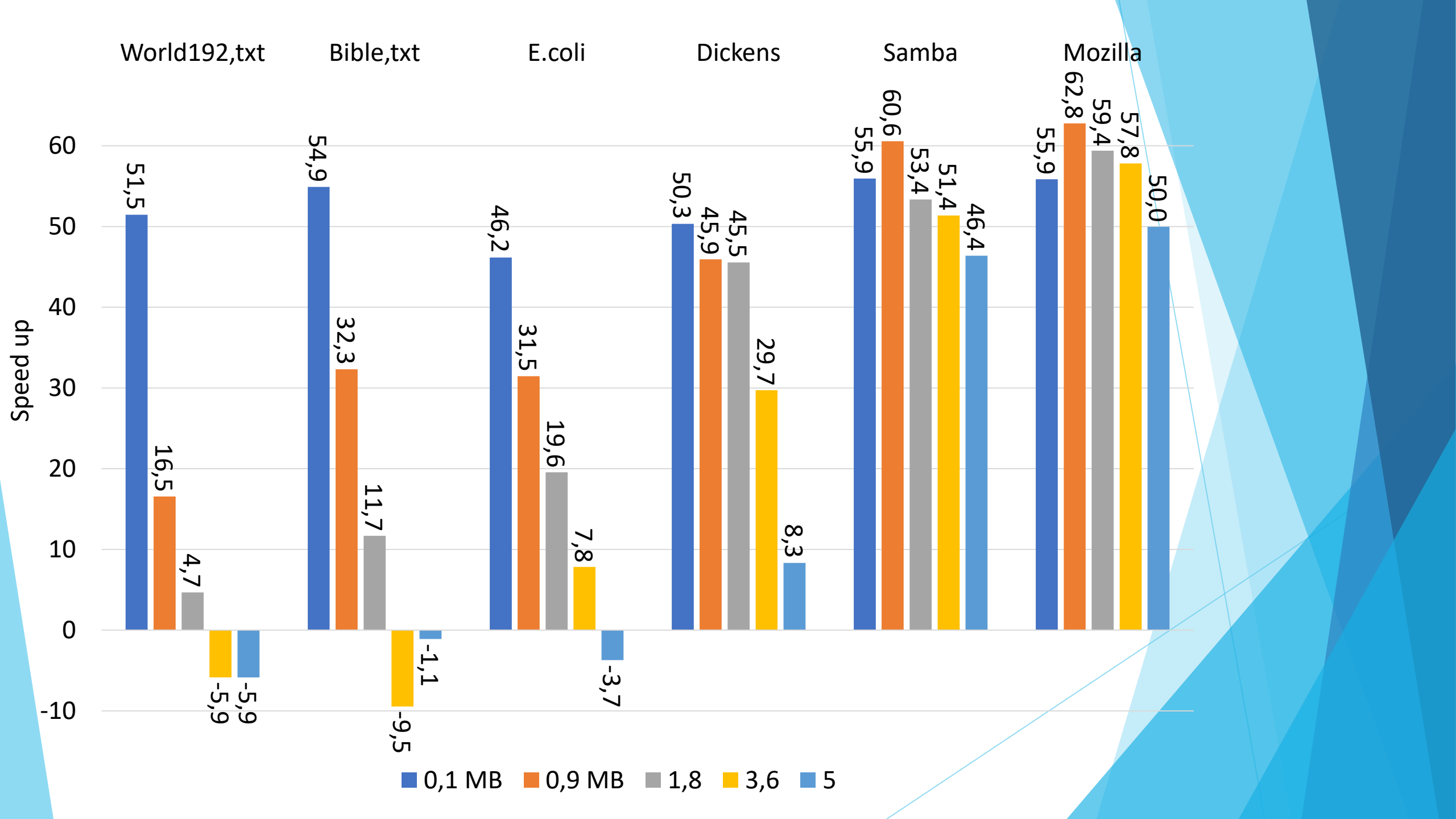
The sequential algorithm is a little faster then the parallel implementation only for small files (less then 5 MB) with a few blocks (1 or 2).

All the experiments measure the wall clock time. Each experiment has been carried out 10 times and the average of the times of each run has been taken.

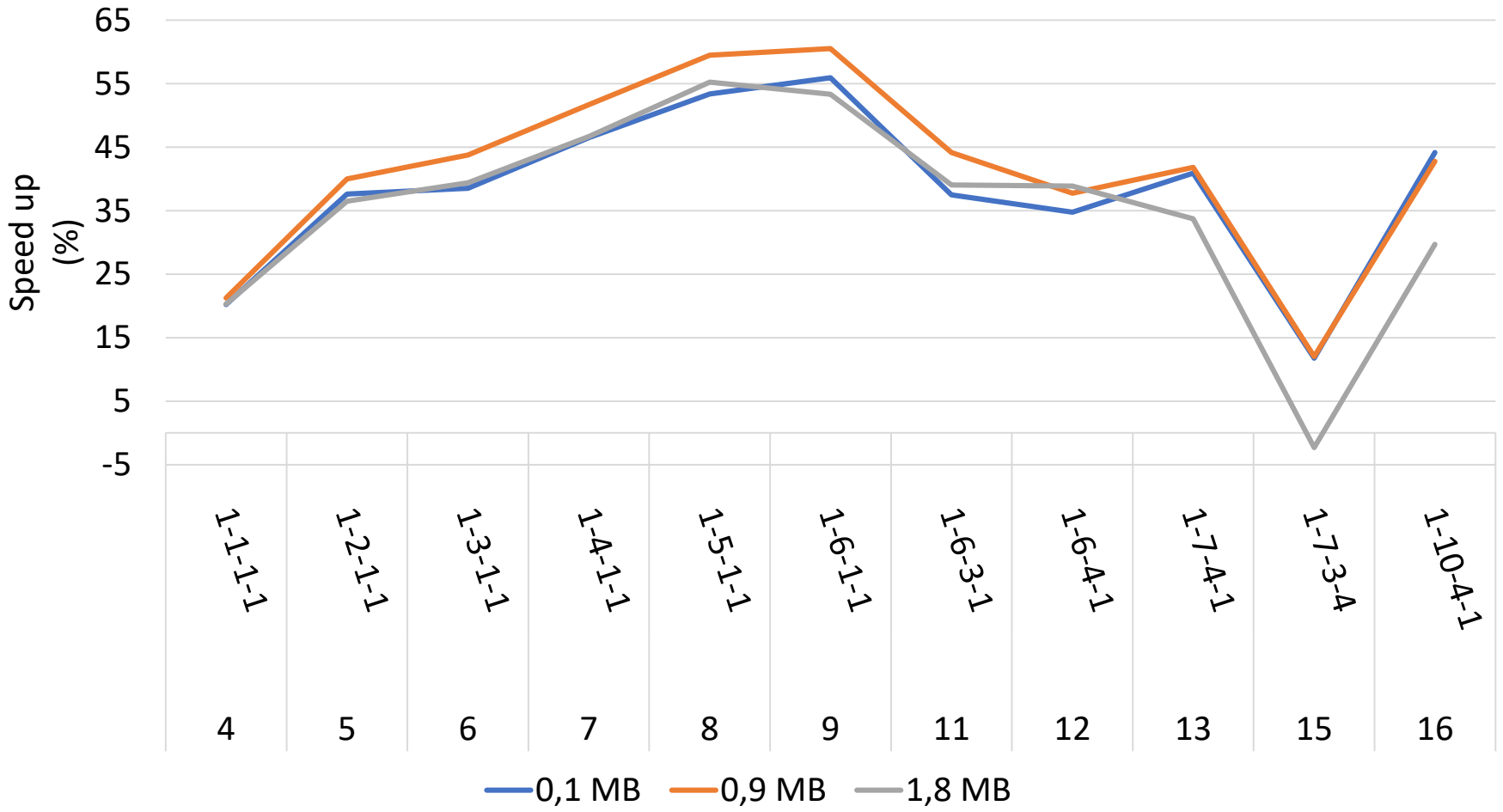
The results on the next table show that the bigger the file, the better the parallel algorithm perform with respect to the sequential.

File	Size (MB)	Time (sec)									
		0,1 MB		0,9 MB		1,8 MB		3,6 MB		5 MB	
		S	P	S	P	S	P	S	P	S	P
World192,txt	2,5 MB	8,695	5,741	8,411	7,217	8,552	8,170	8,623	9,160	8,623	9,160
Bible.txt	4 MB	14,616	9,436	14,970	11,314	15,378	13,770	16,014	17,687	16,204	16,383
E.coli	4,6 MB	21,010	14,375	22,798	17,343	23,615	19,751	24,530	22,750	25,117	26,087
Dickens	10,2 MB	39,931	26,562	41,747	28,609	43,621	29,970	45,169	34,817	45,550	42,049
Samba	21,6 MB	66,21	42,46	69,97	43,58	71,31	46,50	-	-	-	-
Mozilla	51,2 MB	169,646	108,847	180,590	110,947	183,865	115,355	187,500	118,820	189,850	126,593

In this case the parallel implementations used to compare with the sequential version uses 9 threads: the main thread in core 0, cores from 0 to 5 for BWT threads, core 6 for MTF and ZLE thread and core 7 for Arithmetic encoding thread.


World192,txt
Bible,txt
E.coli
Dickens
Samba
Mozilla
Speed up
60
50
40
30
20
10
0
-10
51,5
16,5
4,7
-5,9
-5,9
54,9
32,3
11,7
-9,5
-1,1
46,2
31,5
19,6
7,8
-3,7
50,3
45,9
45,5
29,7
8,3
55,9
60,6
53,4
51,4
46,4
55,9
62,8
59,4
57,8
50,0
0,1 MB
0,9 MB
1,8
3,6
5

CPUs Scalability



File: Samba, 21,6 MB.

Conclusions

- The compression rates are similar to the ones of the bwtzip.
- The bottleneck of the application is the BWT stage, probably it should be split into more pipeline stages to balance more the pipeline and to increase the parallelism of the other stages.
- The CPU scalability should be analyzed with a more powerful computer and bigger files.

References

- Parallel Lossless Data Compression Based on the Burrows-Wheeler Transform, Jeff Gilchrist and Aysegul Cuhadar, Carleton University.
- Silesia corpus, http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia.
- Large corpus, http://corpus.canterbury.ac.nz/descriptions/.
- Visualization suffix tree, http://brenden.github.io/ukkonen-animation/.
- Construction of the suffix tree (Ukkonen's algorithm), https://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-1/.
- Arithmetic encoding, https://github.com/wonder-mice/arcd.
- POSIX Thread documentation, https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html.
- BWT compression, Burrows Wheeler Compression: Principles and Reflections, Peter Fenwick, The University of Auckland.
- Bwt zip, https://nuwen.net/bwtzip.html.