Project: Parallelization of Relative Lempel-Ziv compression for external memory

Professor: José Fuentes Sepúlveda
Course: Multicore programming (CC7315)
University of Chile

May 23, 2017

During this semester, the students of the course *Multicore programming (CC7315)* must develop a project. In this opportunity, the theme of the project is **parallel relative Lempel-Ziv**. In Section 1, we present an introduction to relative Lempel-Ziv. In Section 2 we present a sequential implementation of the relative Lempel-Ziv algorithm. This implementation will be the baseline of the project. Finally, in Section 3 we describe the rules of the project.

1 Relative Lempel-Ziv

Relative Lempel-Ziv (RLZ) is a modification of the well-known LZ77 algorithm [2]. After 40 years, LZ77 is still one of the most effective compressors. It is the core of different compression utilities, such as zip, gzip, 7zip, and the GIF image format.

Let's see the description of LZ77 [1]: The LZ77 compression algorithm consists on two phases: *parsing* (also called factorization) and *encoding*. Given a text $T[1, N]$, a LZ77 valid parsing is a partition of T into z substrings T^i (often called phrases or factors) such that $T = T^1 T^2 \dots T^z$, and for all $i \in [1, z]$ either there is at least one occurrence of T^i with starting position strictly smaller than $|T^1 T^2 \dots T^{i-1}|$, or T^i is the first occurrence of a single character.

The encoding process represents each phrase T^i using a pair $(p_{\mathrm{i}}, l_{\mathrm{i}})$, where p_{i} is the position of the previous occurrence of T^i and $l_{\mathrm{i}} = |T^i|$, for phrases that are not single characters. When $T_{\mathrm{i}} = \alpha \in \Sigma$, then it is encoded with the pair $(\alpha, 0)$. We call the latter literal phrases and the former copying phrases. To find the position p_{i} of a copying phrase T^i, the algorithm uses a dictionary of the text prefix $|T^1 T^2 \dots T^{i-1}|$. The dictionary can be implemented by using *suffix trees* or *suffix arrays*.

Decoding LZ77 compressed text is particularly simple and fast: the pairs $(p_{\mathrm{i}}, l_{\mathrm{i}})$ are read from left to right, if $l_{\mathrm{i}} = 0$, then p_{i} is interpreted as a char and it is appended to the output, if $l_{\mathrm{i}} \neq 0$, then l_{i} characters are copied from the position p_{i} to $p_{\mathrm{i}} + l_{\mathrm{i}} - 1$ and are appended to the current output.

For example, for the following text $z = 6$, the factors are underlined and

$$\begin{array}{l} \\ text = \end{array} \begin{array}{ccccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\ \underline{\mathrm{T}} & \underline{\mathrm{C}} & \underline{\mathrm{T}} & \underline{\mathrm{G}} & \underline{\mathrm{A}} & \underline{\mathrm{C}} & \underline{\mathrm{T}} & \underline{\mathrm{G}} & \underline{\mathrm{A}} & \underline{\mathrm{C}} & \underline{\mathrm{T}} & \underline{\mathrm{G}} & \underline{\mathrm{A}} \end{array}$$

the LZ77 compression is

$$(T,0)\ (C,0)\ (1,1)\ (G,0)\ (A,0)\ (2,8)$$

The RLZ uses a dictionary that is not a prefix of the text. Instead, the dictionary is a *different text* that is provided separately. Such different ttext is called *reference* and it is smaller than the text to be compressed. Thus, the size of the dictionary will be small than the dictionary of the original LZ77 algorithm. In the RLZ algorithm all the positions p_i of the factors are positions in the reference. Thus, the choice of the reference is important. For example, for the same text `TCTGACTGACTGA` and the reference `GACT`, we have the following compression (factors are underlined):

$$\begin{array}{l} \overset{1\ \ 2\ \ 3\ 4\ \ 5\ 6\ \ 7\ \ 8\ 9\ 10\ 11\ 12\ 13}{text = \underline{\mathrm{T}}\ \underline{\mathrm{C}\ \mathrm{T}}\ \underline{\mathrm{G}\ \mathrm{A}\ \mathrm{C}\ \mathrm{T}}\ \underline{\mathrm{G}\ \mathrm{A}\ \mathrm{C}\ \mathrm{T}}\ \underline{\mathrm{G}\ \mathrm{A}}} \\ \\ \overset{1\ \ 2\ \ 3\ 4}{reference = \mathrm{G}\ \mathrm{A}\ \mathrm{C}\ \mathrm{T}} \\ \\ RLZ\ compression = (4,1)\ (3,2)\ (1,4)\ (1,4)\ (1,2) \end{array}$$

and using the reference `TCGA`, we obtain the following compression:

$$\begin{array}{l} \overset{1\ \ 2\ \ 3\ 4\ \ 5\ 6\ \ 7\ \ 8\ 9\ 10\ 11\ 12\ 13}{text = \underline{\mathrm{T}\ \mathrm{C}}\ \underline{\mathrm{T}}\ \underline{\mathrm{G}\ \mathrm{A}}\ \underline{\mathrm{C}}\ \underline{\mathrm{T}}\ \underline{\mathrm{G}\ \mathrm{A}}\ \underline{\mathrm{C}}\ \underline{\mathrm{T}}\ \underline{\mathrm{G}\ \mathrm{A}}} \\ \\ \overset{1\ \ 2\ \ 3\ 4}{reference = \mathrm{T}\ \mathrm{C}\ \mathrm{G}\ \mathrm{A}} \\ \\ RLZ\ compression = (1,2)\ (1,1)\ (3,2)\ (2,1)\ (1,1)\ (3,2)\ (2,1)\ (1,1)\ (3,2) \end{array}$$

In this project, we will use a modified version of the RLZ algorithm, introduced on [1]. The modified version has two features: First, it represents the reference using LZ77, and second, is that instead of using an arbitrary reference, the algorithm uses a prefix of the input text. Finally, the compressed reference and compressed text are concatenated in the output. The implementation of this version is described in Section 2.

2 Baseline implementation

The baseline code for this project corresponds to the RLZ implementation of *CHICO*, a compressed hybrid index for repetitive collections [1]. The implementation corresponds to an *external memory algorithm*, which allows us to efficiently process texts that do not fit in main memory. The original code is available at `https://www.cs.helsinki.fi/u/dvalenzu/software/CHICO-0.1.tar.gz`[1]. For the project purpose, we provide a clean version of the code at `https://github.com/jfuentess/multicore-programming/tree/master/Project`.

The different steps of the implementation are representated in the workflow chart of Figure 1. In the first steps, the references is read and parsed to create the dictionary. The created dictionary will be used to parse the input text. Then, in the following steps, the input text is read block by block. The size of a block is setted in running time. Each block is divided into partitions and each partition is parsed independently. The number of partitions is also setted in running time. After parsing an entire block, the list of factors is written into the output. The algorithm finishes after parsing all the blocks.

[1]Many thanks to Daniel Valenzuela to allow us to use his implementation.

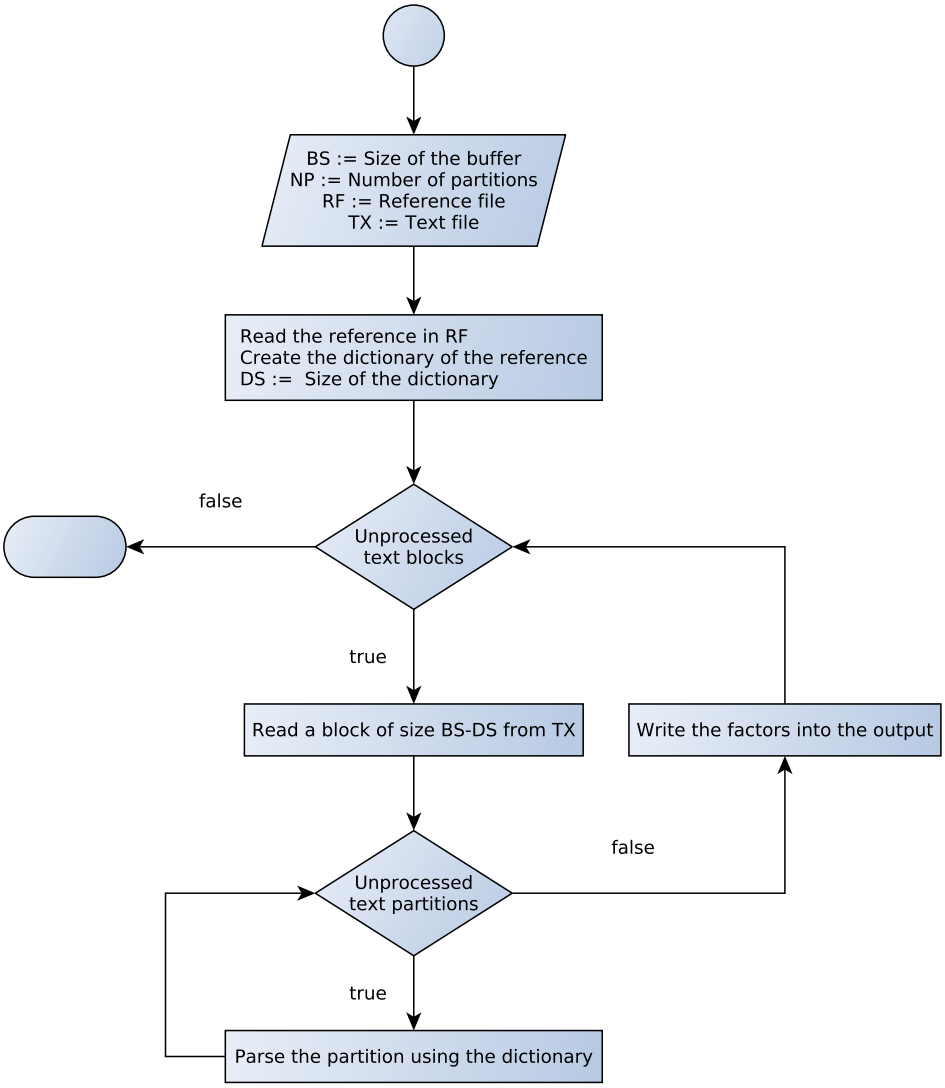

Figure 1: Workflow of the implementation of the RLZ algorithm for external memory

2.1 Setup and execution

To run the baseline implementation, the following steps must be accomplished:

1. Download the code from `https://github.com/jfuentess/multicore-programming/tree/master/Project`.

2. In a terminal, run the bash script `setup.sh`. It will download and install a needed library called SDSL[2].

3. Then, enter to the folder `LZ/RLZ/src` and run the `make` command. It will compile the RLZ compressor.

[2]SDSL - Succinct Data Structure Library. `https://github.com/simongog/sdsl-lite`. Last accessed: 22 May 2017

4. To run the compressor, execute the following command: `bash rlz_parser.sh INPUT_FILE OUTPUT_FILE REF_SIZE_MB N_PARTITIONS MAX_MEM_MB`, where:

   - `INPUT_FILE`: It is the path to the input text file
   - `OUTPUT_FILE`: It is the path to save the compressed file
   - `REF_SIZE_MB`: It is the size of the reference
   - `N_PARTITIONS`: It is the number of partitions used to parse each block
   - `MAX_MEM_MB`: It is the maximum amount memory to be used. It is also used to compute the number of blocks

   Before executing the compressor, the `rlz_parser.sh` script creates the reference by reading the first `REF_SIZE_MB` MB from the input text file. The reference is stored temporarily in the file `tmp.reference`.

5. Finally, to decompressed the output of the RLZ implementation, go to the folder `LZ/LZ-Decoder` and run the `make` command. Then, execute `./decode COMPRESSED_FILE DECOMPRESSED_FILE BUFFER_SIZE`, where:

   - `COMPRESSED_FILE`: It is the path to the compressed file. It corresponds to the output of the RLZ compressor
   - `DECOMPRESSED_FILE`: It is the path to save the uncompressed file
   - `BUFFER_SIZE`: The decompressor works block by block. `BUFFER_SIZE` is the size of each block

   Thus, to verify if the compressor is working properly, we can use the Linux command `cmp DECOMPRESSED_FILE INPUT_FILE`.

Several datasets are available in the folder `/home/jose/Datasets/Sequences`, in the multicore Machine *Keira*. More details of the datasets in Section 3.2.

3 The project

The aim of this project is to *improve the performance* of the implementation of the RLZ compressor (see Section 2). To improve the performance, the students must use all the parallel techniques learned during the course or (for one solution) any other technique, including, caching, compressing, etc.

3.1 Rules

Deadline: June 24, 2017

Individual work: Each student must do the project individually.

Parallel solutions: Each group must include at least **three** solutions to improve the performance of the RLZ compressor. At least two solutions must be related with parallelism (Cilk Plus). The other solution may be related with memory behavior or any other performance improvenment.

Machine: All the experiments must by carry out in *Keira* (or, if it possible, in a machine with more than 12 cores).

Code: The code must be uploaded to github, with installation instructions.

Cilk Plus : All the parallel solutions must be implemented using Cilk Plus.

3.2 Experiments

To test the implementations, we provide several datasets. The datasets are available in the folder `/home/jose/Datasets/Sequences`, in *Keira*. Here there is a description of the datasets:

- `dna`: A sequence of newline-separated gene DNA sequences. Each of the 4 bases is coded as an uppercase letter A,G,C,T, and there are a few occurrences of other special characters. Alphabet size: 16, text size: 385 MB.
- `english`: Concatenation of English text files. Alphabet size: 239, text size: 2107 MB.
- `proteins`: A sequence of newline-separated protein sequences. Each of the 20 amino acids is coded as one uppercase letter. Alphabet size: 27, text size: 1129 MB.
- `sources`: C/Java source code obtained by concatenating all the .c, .h, .C and .java files of the linux-2.6.11.6 and gcc-4.0.0 distributions. Alphabet size: 230, text size: 201 MB
- `rna.fa`: FASTA file of the RNA data of Homo sapiens. Text size: 588 MB.
- `random.1GB.16`: Random file of 1GB of symbols from the alphabet is [1,16]. Alphabet size: 16, text size: 1024 MB
- `random.4GB.16`: Random file of 4GB of symbols from the alphabet is [1,16]. Alphabet size: 16, text size: 4096 MB
- `random.1GB.256`: Random file of 1GB of symbols from the alphabet is [1,256]. Alphabet size: 256, text size: 1024 MB
- `random.4GB.256`: Random file of 4GB of symbols from the alphabet is [1,256]. Alphabet size: 256, text size: 4096 MB
- `random.8GB.16`: Two concatenated copies of the file random.4GB.16. Alphabet size: 16, text size: 8192 MB
- `random.8GB.256`: Two concatenated copies of the file random.4GB.256. Alphabet size: 256, text size: 8192 MB

The datasets `dna`, `english`, `proteins` and `sources` are part of the *Pizza&Chili Corpus*[3]. The dataset `rna.fa` belongs to the *National Center for Biotechnology Information*[4]. Finally, the datasets `random.1GB.16`, `random.4GB.16`, `random.8GB.16`, `random.1GB.256`, `random.4GB.256` and `random.8GB.256` were generated with the random file generator available at *Pizza&Chili Corpus*[5].

[3]Text collection, Pizza&Chili Corpus. URL: `http://pizzachili.dcc.uchile.cl/texts.html`

[4]RNA information of Homo sapiens, National Center for Biotechnology Information. URL: `ftp://ftp.ncbi.nih.gov/genomes/Homo_sapiens/RNA/rna.fa.gz`

[5]Random file generator, Pizza&Chili Corpus. URL:`http://pizzachili.dcc.uchile.cl/utils/gentext.c`

Performance measurement: During the experimental step, the students must measure running time, speedup, cache misses and compression ratio.

- *Running time*: To measure the running time, the students must use standard C/C++ functions, like `clock_gettime`.
- *Speedup*: To measure the speedup, the students must use the speedup formula.
- *Cache misses*: To measure the number of cache misses, the students must use `perf`.
- *Compression ratio*: To measure the compression ratio, the students must use the data compression ratio formula $uncompressed\_size/compressed\_size$.

The experiments must include, at least, change the dataset, change the number of blocks, change the number of partitions. Each experiment must be repeated at least five times and the median must be reported.

3.3 Project report

At the end of this project, each group must include a report, besides the implementation. The report must include the following sections:

- *Introduction*: An explanation of the project and its main goal. The report must be self-content, so, this section must be long enough.
- *Proposed solutions*: A description of each solution. For each solution, the theoretical and practical fundamentals must be explained. The complexity analysis must be included in this section.
- *Implementation details*: In this section, the technical details of each proposed solution must be explained. Only pseudocode could be included in this section (code is not allowed).
- *Experiments*: Tables and graphs reporting the experimental results must be included in this section. Each table and graph must be explained.
- *Discussion*: In this section the experimental results must be discussed. For example, discuss if the compression ratio is better or worse after including the usage of threads or discuss if the running time is related with the number of blocks/partitions, among others.
- *Conclusions*: Conclusions of the project.

3.4 Evaluation

The evaluation of the project will consider the following ponderation:

- Theoretical analysis: 30% (Proposed solutions)
- Implementation: 35% (Code)
- Experimental study: 35% (Experiments and discussions)

References

[1] Daniel Valenzuela. Chico: A compressed hybrid index for repetitive collections. In *Proceedings of the 15th International Symposium on Experimental Algorithms - Volume 9685*, SEA 2016, pages 326–338, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[2] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.