International Journal of Modeling, Simulation, and Scientific Computing

CLUSTERING AND PRESORTING FOR PARALLEL BURROWS WHEELER BASED COMPRESSION

Sergey Voronin *, Eugene Borovikov, Raqibul Hasan

We describe practical improvements for parallel BWT based lossless compressors frequently utilized in modern day big data applications. We propose a clustering based data permutation approach for improving compression ratio for data with significant alphabet variation along with a faster string sorting approach based on the application of the $O(n)$ complexity counting sort with permutation reindexing.

Keywords: Lossless data compression, Burrows-Wheeler transform, data permutation, fast string sorting.

1. Introduction

Many of the best rated universal lossless compression approaches are based on Burrows-Wheeler compression, such as bzip2 [7,8]. This approach consists of several steps: a Burrows-Wheeler local similarity transform (BWT), followed by a Global Similarity Transform (typically implemented via a variant of the move to front (MTF) transform), run length encoding (RLE) for compressing sequences of identical data and finally entropy coding (EC), such as Huffman or arithmetic coding. An illustration of the steps [4] is shown in Figure 1. For file sizes above 100 MB, serial compression implementations are typically not practical. In common parallel implementations (such as lbzip2), the input file is subdivided in even chunks and the operations are performed in parallel over the smaller chunks, with the BWT typically performed over parts of the smaller chunks [5]. While this parallel approach substantially improves runtime, the even splitting can adversly affect the resulting compression ratio compared to a serial algorithm.

BWT is the first of three most consuming steps of Burrows-Wheeler compression [2]. In particular, it consists of string sorting (hence, it's relative inefficiency, as it requires by default an $O(nlogn)$ sort) and produces an output with local similarity structure, with runs of similar characters. The MTF is an essential step performed prior to entropy encoding (EC). MTF is based on the expectation that once a symbol is read from the input buffer, it will be read in the local block several more times, becoming a common symbol. MTF moves the latest encountered symbol to the front of the list. It can significantly improve EC performance when the local frequency of symbols changes significantly from region to region in the loaded input

*Please send correspondence to svoronin@i-a-i.com.

buffer [1]. The BWT and MTF routines simply re-arrange, group, and re-code data into a same sized output to make it better compressible. RLE provides simple compression for runs of repeated symbols (typically digits, at the output of MTF) and entropy coding, typically Huffman or Arithmetic coding encodes the resulting information with fewer bits. In the case of Arithmetic coding, for example, fractional representations are used (represented by a single floating point number within the interval). Of the outlined steps, BWT plays the profounding role in achieving data compression with the latter RLE and EC steps [9].

In this article, we discuss a simple clustering and presorting strategy, performed in order to aid compression ratio and runtime, particularly for larger file sizes, for which parallel compression is beneficial. In particular, in place of a standard subdivision of the input, we outline a data permutation strategy which re-arranges small blocks of the input based on their content and groups them together into larger 'megablocks'. These larger, but more homogeneous in content blocks, can then be compressed more efficiently by the remaining steps, in parallel. The BWT can be applied over portions of these megablocks, using a counting sort implementation with permutation reindexing computation and bucketing to accelerate string sorting, a traditionally time consuming operation. The outlined approach is well suited for the suffix-array implementation of the BWT [10], where sorting on an upper triangular string matrix is performed. An optional byte swapping procedure is discussed over integer byte blocks prior to the entropy encoding application.

```
(a) BWT input    : 61 62 72 61 63 61 64 61 62 72 61 61 62 72 61 63 61 64 61 62 72 61
(b) BWT output   : 61 72 72 64 64 61 72 72 63 63 61 61 61 61 61 61 61 62 62 62 62
(c) GST output   : 61 72 00 65 00 02 02 00 65 00 02 00 00 00 00 00 00 65 00 00 00
(d) RLE0 output  : 62 73 00 66 00 03 03 00 66 00 03 00 00 00 66 00 00
(e) EC output    : 00 0D 01 8D B3 FF 81 00 72 A8 E8 2B
```

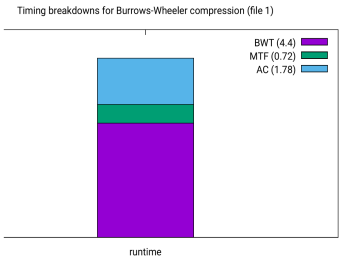

Fig. 1. Illustration of Burrows-Wheeler based compression (steps and single thread timing breakdowns for 20 MB ascii file) .

2. BWT and suffix arrays

The goal of BWT is to reversibly transform the input into a better compressible form based on string sorting. As an example, the BWT of the string T='eat ba-

nanas$' is 'tsbnne $aaaa', which has better symbol grouping than the original string and can be more readily compressed with run length encoding and move to front and entropy coding steps. The BWT output is the last column of the associated BW matrix (BWM) consisting of all sorted rotations of the input string. Conventional sorting (e.g., quicksort) takes $O(n \log n)$ average and $O(n^2)$ worst case time, which gets prohibitively expensive for large strings. The BWT can also be computed by sorting all suffixes of a string. The suffix array provides a lexicographically ordered list of all the suffixes of a string. For example, the suffixes of 'blogger' are: blogger, logger, ogger, gger, ger, er, r; and the sorting of such a set can be done more efficiently by exploiting its upper triangular matrix structure. For example, the sorted rotation and the suffix array of the string 'banana$' are shown in Figure 2. Here, '$' represents the end of string (EOF) character, which can be arbitrarily chosen. In terms of the suffix array represented numerically by S, the i-th character of the BWT is defined as $T[S[i]-1]$, if $S[i] > 0$ and set to the EOF character if $S[i] = 0$, with T the input string [9].

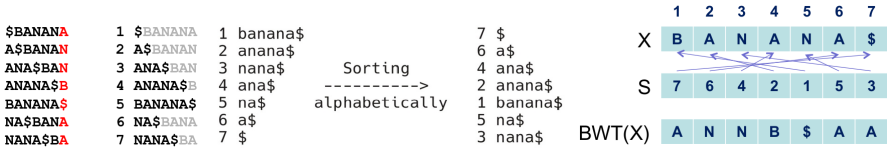

Fig. 2. String rotations and suffixes for BWT .

3. Data permutation via clustering

Typically, in parallel compressors (such as lbzip2), the input is divided uniformly in parts, without examining their contents. The use of data permutation achieved via clustering, prior to the application of BWT, can bring about significant performance improvements. We propose to subdivide input data in relatively small blocks, run a processing and clustering scheme over these data blocks, and then to merge sets of these blocks into larger *megablocks* by comparing similarity of their content. These megablocks are then processed independently by the BWT (which depending on implementation, can range from $O(n^2 \log n)$ to linear time complexity with the use of the suffix array) and can be applied over subblocks of the megablocks, with the result of the operation compressed by RLE, MTF, and the EC steps, applied over the BWT result on each megablock. An optional byte swapping procedure can be inserted in between the MTF and EC steps.

In order to cluster small blocks together, we analyze their symbol frequency contents. Large input streams often contain blocks with particular symbol distributions, typically limited to a small number of symbols that are characteristic of each block.

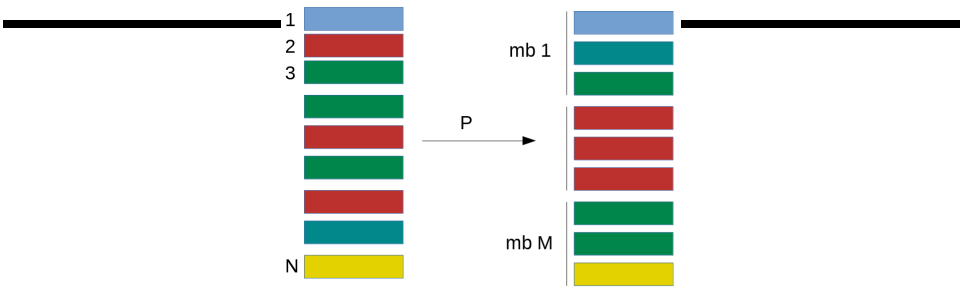

Fig. 3. Data permutation via clustering and megablock construction.

For instance, a large multi-lingual text and numerical based document may contain blocks of text in different languages, some tables with numerals, and possibly images and other binary content. No matter the symbol distribution, the max value of each individual unsigned byte will be 255. We propose to subdivide the input into small blocks using initial even subdivision and compute the approximate symbol distribution within each small block, represented by a normalized histogram. These blocks are then clustered together into larger megablocks by comparing these histograms, resulting in megablocks of similar content, as illustrated in Figure 3. That is, we first chunk the input into small portions, with each portion assigned a number from 1 to N. We then analyze the individual small blocks and create byte frequency histograms of their contents, as shown in Figure 4. The byte value range $(0-255)$ should be sufficient for ascii and other files, making the maximum number of bins per histogram of 256. Optionally, the number of bins could vary, if a different range is desired for a particular input. To construct the frequency distributions for each

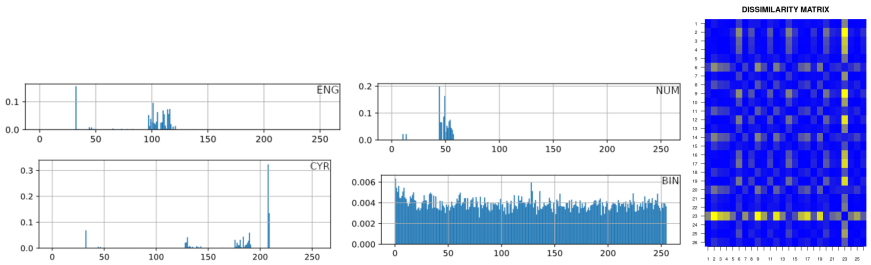

Fig. 4. Normalized 256-bin histograms of different content types that can occur in a single stream: ENG=English, NUM=Numeric, CYR=Cyrillic, BIN=Binary. Sample dissimilarity matrix between pairs of blocks.

block, the following pseudocode code applies, with the counts array normalized by the number of (one or more) byte chunks scanned:

```
while not at EOF:
  fread NUM byte chunks from file into buffer array
  num = 0;
```

```
for each (NUM bytes) in buffer array:
  val = (int) buffer[i]; counts[val]++; num++;
counts = counts/num;
```

In case of a larger size input or binary content, we can read several, e.g. $4-8$ bytes, at time from each block and record the corresponding (e.g. integer or long) value. The counter for this value is then incremented on each encounter during the block pass. Upon normalization, assigned to each block will be a probability distribution for the different values in the set range within the block. These approximate probabality distributions (each corresponding to one small block) can be clustered and used to re-arrange the smaller blocks of the input into the megablocks shown in Figure 3. These larger blocks do not need to be of the same size but will be composed of smaller blocks of similar symbol frequency content. BWT and the rest of the compression steps, can then be performed separately on these megablocks. To accomplish clustering, we propose two methods. The first method uses a dissimilarity matrix between probability distributions in pairs of small blocks. This $N \times N$ matrix records dissimilarity between different symbol (or byte) probability distributions in pairs of small blocks and is constructed by utilizing a similarity metric such as the Kullback-Leibler Divergence (KLD, as in eq. 1) or the Jeffrey divergence [12].

$$\mathrm{KLD}(p(y), p(x)) = \sum_{i=1}^{n} p(y_i) \log \left[\frac{p(y_i)}{p(x_i)} \right] \tag{1}$$

Once the pairwise dissimilarity matrix is stored, several clustering schemes to group file chunks can be employed. To form the matrix we compute the pairwise KLD distance over the counts array for each small block. That is, once the counts for the byte values on the range $[0, 255]$ are obtained over each block, we can evaluate the pairwise KLD distance between the arrays for the individual blocks:

```
for (i,j in 1:nblocks):
    diss_mat[i,j] = kld_dist(counts_mat[i,], counts_mat[j,]);
```

The clustering of the blocks can then be obtained directly with the dissimilarity matrix information. Each cluster of blocks would then make up a single larger megablock which would contain similar content. The megablock construction approach is summarized in the following listing.

- Subdivide the file into N blocks with a unique index, so that each part is on the order of 1 MB or less in size.
- Construct histogram representation for each block by counting the number of occurrences of each value (of one or more bytes) in a given range (e.g. $[0, 255]$).
- Normalize histograms as probability distribution approximations on the chosen symbol range to make them comparable.
- Construct the $N \times N$ dissimilarity matrix by comparing individual pairs of probability distributions. This can be done by utilizing the KLD metric or

histogram comparison techniques [3,6]

- Once a dissimilarity matrix is constructed, clustering can be accomplished with a number of different schemes. Several approaches can be used to estimate the optimal number of clusters, e.g. based on the silhouette score. Construct $M < N$ clusters, with most similar blocks being put together into one cluster.
- Take megablocks as the M resulting clusters.
- Take re-arrangement vector to represent the permutation matrix P, to be stored for the decompression operation.

3.1. *Alternate clustering*

As an alternative to the dissimilarity matrix based approach, we propose the adaptive scheme shown below, which automatically allocates additional clusters as the formed block probability distributions are scanned. The standard K-means clustering method may require a super-polynomial number of steps in its worst case, with overall $O(n^2)$ complexity. The proposed adaptive clustering involves at most $n(n-1)/2$ distribution comparisons, has random seeding, and can be scaled for large streams using approximate descriptor matching methods.

```
IN: ByteStream # processed in chunks
OUT: Clusters # (block, histogram) buckets

def init(ByteStream, BlockSize=1024):
  while ByteStream.isValid:
    add(ByteStream.read(BlockSize))

def add(block):
  hist = histogram(block)
  cluster = nearest(hist)
  if cluster: insert(cluster, block, hist)
  else: new(block, hist)

def histogram(block, binCount=256):
  hist = array(binCount)
  for b in block: hist[b]+=1
  return hist

def nearest(histogram, MaxDistance=0.00733):
  dist = infinity
  clst = nil
  for cluster in Clusers:
    d = distance(cluster.mean, histogram)
    if d < dist: dist = d; clst = cluster
  if dist > MaxDistance: return nil
  return clst

def insert(cluster, block, hist):
  cluster.blocks.add(block, hist)
  cluster.mean.update(hist) # update cluster mean histogram

def new(block, hist):
```

```
clst = cluster(block, hist)
Clusters.add(clst)
clst.mean.update(hist)
```

3.2. *File splitting for common cases*

Some common cases, such as inputs with mixed text and numerical content, can utilize custom splitting routines, prior to regular clustering. An example is a large CSV file with a mix of text and numerical entries. For such inputs, a simple reversible scanning and re-arrangement procedure can be performed, in order to subdivide the file into two parts (one with mostly text content and one with mostly numerical content). This is beneficial, as algorithms such as prediction by partial matching (PPM) [11] are specifically designed for text inputs and can then be applied to the text portion. In order to accomplish the splitting efficiently and reversibly, an algorithm presented in Figure 5 (top) has been implemented. If necessary, the resulting output files can then be separately subdivided in chunks and combined into separate megablocks with the described clustering procedure.

```
Input: input_file
Output: two splitted data file and a configuration file

#define SIZE 64
while not end of input_file
      buffer=read SIZE bytes block from input_file
      s=average ASCII value of buffer
      if s<'A'
            block_state=numeric
            write buffer in numeric file
      else
            block_state=text
            write buffer in text file
      if block_state==chunk_state
            chunk_size=chunk_size+SIZE;
      else
            Write chunk information (start_addr, chunk_size,
            chunk_state) in a config_file
            start_addr=start_addr+chunk_size
            chunk_size=SIZE
            chunk_state=block_state
```

```
Input: mapping_file, numeric_part, text_part
Output: combined_file
#define NUMERIC 0
#define TEXT 1
for each entry (chunk_size, state) in the mapping_file
      if state==NUMERIC
            data=read chunk_size bytes from numeric_part
      else
            data=read chunk_size bytes from text_part
      write data in combined_file
```

Fig. 5. File splitting method for mixed numerical and CSV content.

4. Counting sort for accelerated suffix sorting

We now outline an approach for string sorting based on the $O(n)$ counting sort and bucketing, which complements the megablock based construction. In order to form the suffix array, from which the BWT of the input can be deduced, it is necessary to sort the string suffixes [10]. An example is given in Figure 6, where we see that presorting strings by the first character bring us close to the correctly sorted result. The default approach is to sort the suffixes with quicksort using a suitable comparison function such as *strcmp()* in C. This procedure is predictibly slow for large inputs, as it relies on the $O(n \log n)$ quick sort with additional $O(n)$ complexity for string comparisons, which would approach $O(n^2 \log n)$ total time complexity. Faster algorithms are based on the idea that the suffixes to be sorted

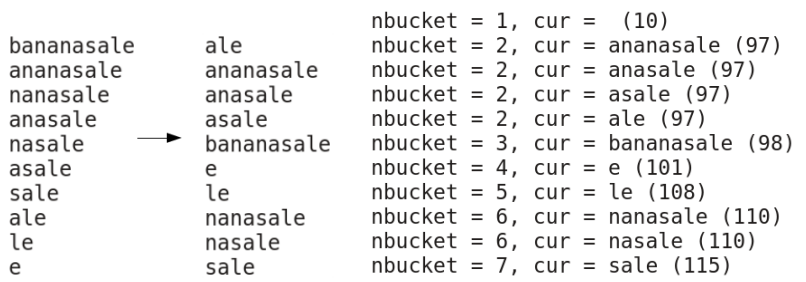

Fig. 6. Suffix array sorting for input string 'bananasale' and result of one level of counting sort and associated bucketing.

are all suffixes of a single string, sorting first according to 2^i characters, and then using the result to sort to 2^{i+1} characters in $0(n \log n)$ time, taking advantage of $O(1)$ comparison time for two values. The resulting complexity is $O\left(n(\log(n))^2\right)$ and can be improved to $O(n \log n)$ using the radix sort [13].

We discuss the use of an $O(n)$ complexity counting method based pre-sort, followed by bucketing, which can be used to accelerate suffix array string sorting within each megablock similar to the radix sort method. In this approach, we utilize a counting sort procedure which sorts a non-negative integer array and returns the resulting permutation re-arrangement vector, which is necessary for the array formation. The approach is applied over one character of the stings at a time, after which bucketing is performed to separate strings into baskets. The approach can be repeated again within each bucket and quick sort on each (small) bucket is eventually employed to sort the strings. In Figure 5, we can see the suffixes of the string 'bananasale', as well as the sorted result. One iteration of the $O(n)$ sort on the first character of each string followed by bucketing, would yield buckets of strings starting with 'a', 'b', 'e', 'l', 'n', and 's'. In the next iteration, these buckets can themselves be sorted with respect to their first character (the second character of the original set of strings). This is still not enough to properly position 'anansale' and 'anasale', which require 4 levels of sorting (up to the fourth character). However, it is clear from the example that one or more levels of sorting and subsequent bucketing with respect to one character can reduce the number of full comparisons needed in the subsequent $O(n \log n)$ sort over application on the original full string set. With one level of the scheme, instead of n $O(n \log n)$ sorts for large n, we would have an $O(n)$ (counting) sort, followed by linear cost bucketing, and several $O(m \log m)$ sorts of the buckets with $m \ll n$. At any stage, if the resulting buckets are of many different sizes, the larger buckets can be split up by sorting them by their first character.

To sort the suffix array and obtain the BWT, one must not only sort the strings, but record the re-ordered permutation of the string sequence in relation to the unsorted order. In order to accomplish this, a simple data structure was used together with the counting sort, as shown in Figure 7. This structure is necessary as duplicate

integers in the array may occur. Thus, an index of integers is used for every unique value. The algorithm appears below, where the array *inds* is overwritten with the

```
struct val_inds   before sort:
{                 5  4  2  1  1  3  4  12  10
    int val;      after sort:
    int num_inds; 1  1  2  3  4  4  5  10  12
    int *inds;    inds after sort:
};                3  4  2  5  1  6  0  8  7
```

Fig. 7. Values struct and example sort.

positions of the sorted elements in the original array. Here the *max* variable specifies the max value in the array and the *maxrpts* variable specifies the assumed max number of repeats of each value, which can vary per each unique value. We can assume all integers are non-negative.

```
Begin
   max = get maximum element from array.
   ind = 0;

   // initialize struct array
   for i:=0 to max do
    val_inds[i].inds = (int*) calloc(maxrpts,sizeof(int));
    val_inds[i].num_inds = 0;
   done

   // scan input contents
   for i:=0 to len do
      val_inds[a[i]].inds[val_inds[a[i]].num_inds] = i;
      val_inds[a[i]].num_inds++;
      counts[a[i]]++;
   done

   // find cumulative frequency
   tot = 0;
   for i := 1 to max do
      cnt = counts[i];
      counts[i] = tot;
      tot += cnt;
   done

   // record sorted array in b
   for i:=0 to len do
        b[counts[a[i]]] = a[i];
   done

   // record re-indexing
   for i := 1 to size do
     for j := 1 to val_inds[a[i]].num_inds do
         inds[ind++] = val_inds[a[i]].inds[j];
     done
   done
End
```

The bucketing is accomplished by comparing strings within the suffix array by their

designated character (e.g. first character for first level buckets) and then subsequent characters within buckets. This can be done efficiently, via a linear pass through the output of the sorted set with 0(1) comparisons between characters (integers). The procedure can be repeated inside first level buckets to form sub-buckets, with sorting done with respect to the second character of the suffix strings. Once the resulting string buckets are formed, they can be sorted with full string comparisons:

```
for(i=0; i<nbucket; i++){
    qsort(str_buckets[i].str_arr, str_buckets[i].num_strings,
          sizeof(*(str_buckets[i].str_arr)), saCompare);
}
```

5. Numerical experiments

In this section, we show the results of some numerical experiments. The orginal parallel BWT based compression approach we consider is to subdivide the input in even chunks and then apply BWT with $O(n \log n)$ sort over parts of the even blocks. In the approach we put forward, we first subdivide the input in small chunks, compute the pairwise dissimilarity matrix with respect to the frequency contents of the chunks, use this to cluster chunks into larger megablocks, then apply BWT on small chunks of megablocks with at least one level integer pre-sorting before bucketing and subsequent $O(m \log m)$ sorting with $m < n$. This is followed by MTF, RLE, and entropy encoding (arithmetic coding). For common cases, such as mixed text and numerical content, we first split the input into two bundles (per the approach in Figure 5) and then perform megablock construction. An extra step which in some cases further improves the compression ratio is to utilize byte-swapping either for a portion of the original input, for e.g. the binary content megablocks or for the compression system output prior to the application of entropy encoding, where the data is read in chunks of several bytes and the order of bytes is swapped from greatest to least significant order. This can be accomplished for each multi byte chunk, with a standard swap routine:

```
Begin
  x = (unsigned char*) &val;
  i = 0; j = sizeof(x) - 1;
  while (i<j) do
    temp = x[i];
    x[i] = x[j];
    x[j] = temp;
    i++; j--;
  done
End
```

If utilized, this routine then needs to be appropriately applied to blocks of bytes in the decoder sequence during the decompression step. In an experiment, a mixed ascii text and numerical content file of size 14 MB compressed with the BWT, MTF, RLE, AC sequence to 3 MB and with integrated byte swap between RLE

and AC steps to just over 2 MB. Here the file was read in 4 bytes at a time and the swap performed over each four byte set:

```
fseek(fp, 0, SEEK_END);
long nbytes = ftell(fp); int i = 0;
for (i < (nbytes/4)) do
   fread(buf_in, 4*one, one, fp); val = buffer[i];
   ByteSwap(val);
   memcpy(buf_out+i*4, &val, 4); i++;
end
fwrite(buffer2, one, nbytes, fp2);
```

Compression ratio improvements with the byte swap scheme were observed for different test files.

The next experiment demonstrates the effectiveness of the simple splitting strategy for mixed content data and the linear runtime of the counting sort procedure. We have utilized a 50 MB file with a mix of ascii text and numerical content and applied the splitting strategy from Figure 5. Figure 8 shows the content distribution of the original and two generated data files (the mostly text and mostly numerical portion). In the same Figure, we plot the runtimes of the counting sort, using arrays generated with random integers, as below.

```
arr = (int*)calloc(n,sizeof(int)); inds = (int*)calloc(n,sizeof(int));
for (i = 0; i < n; i++) {
    num = (rand() % (high - low + 1)) + low;
    arr[i] = num; inds[i] = i;
}
counting_sort_with_permind(arr, inds, n, high+1);
```

We also illustrate sorting runtimes on a 13 MB file formed by taking a subset of the rows of the file used for the splitting experiment. In particular, we see that quick sort on the full set took about 58 sec, counting sort with index permutation on first char about 0.7 sec and quicksort on 3 subdivided buckets (which can be efficiently formed based on the index permutation information) a total of 48 sec, suggesting that even crude pre-sorting and bucketing can lead to speed advantages for the BWT step.

Next, to motivate the benefit of megablock clustering in parallel compression, we used an English ebook (The Adventures of Sherlock Holmes), a Cyrillic ebook (War and Peace), and a numerical csv file. These files were then appended together and shuffled, with the following command sequence:

```
cat ASH.txt WaP.txt num1.csv >> big2.txt
shuf big2.txt -o big2s.txt
```

The size of the two resulting files was 38 MB. However, the shuffled content (with greater entropy) compresses worse than the original unshuffled file. The unshuffled file compressed with a combination of BWT, RLE, MTF, and AC implementations with an output size of 3.3 MB, while the resulting shuffled file compressed to 4.0 MB. The compression took about $3.7s$ on a 2.9 Ghz desktop CPU (a large file would necessitate parallel compression for suitable runtime). The file was then split up into 26 equal blocks and probability distributions were generated for the range $[0-255]$

in each portion, as described. The small blocks were then parallel compressed individually. The small blocks compressed significantly faster, but the total output size was 4.4 MB. We implemented the proposed clustering strategy using a dissimilarity matrix construction. The resulting dendrogam is shown in Figure 9, which shows grouping of the small blocks into larger megablocks. Compressing the 5 individual megablocks yielded an output size of 4.1 MB, close to that of the serial compression result. Then we proceeded to split the input into text and numerical portions using our approach. This resulted in two files of 23 and 15 MB each. In turn, cluster compressing these portions in 4 blocks each, yielded a total output size of 3.9 MB, slightly below that of the (serial) single block shuffled content compression, with noticeable improvement on the basic parallel compression strategy.

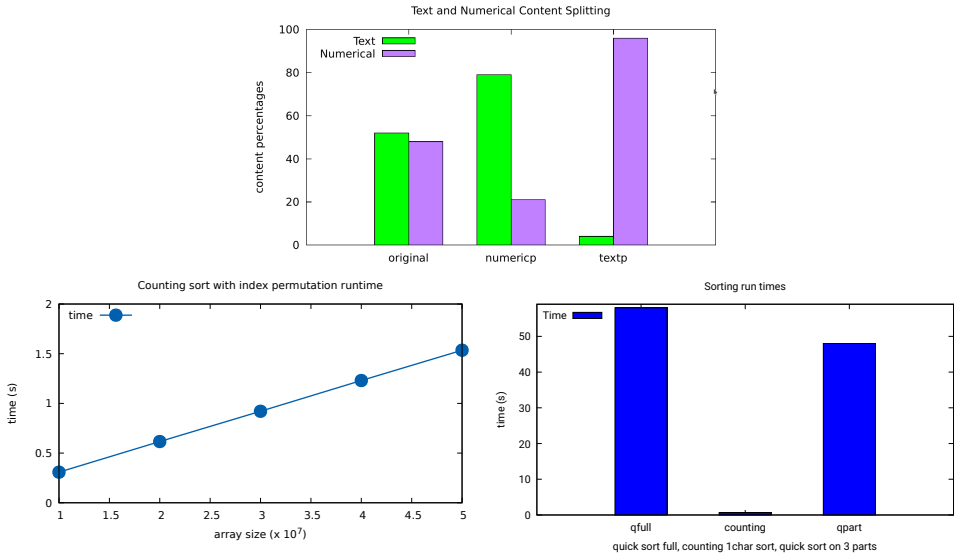

Fig. 8. Top: sample performance of text/numerical splitting. Bottom: linear runtime of counting sort with index permutation vs array size and sample sorting times (full set with quicksort, counting sort on first character, and quicksort on 3 buckets).

6. Conclusion

We discuss improvements to parallel BWT-based compression implementations, especially for application to large non-homogeneous data inputs. The presented strategy is based on data permutation, accomplished by clustering of blocks of the data input into larger megablocks before the BWT and subsequent operations are applied. This is especially effective for mixed alphabet inputs which do not compress optimally with even subdivision. We also outline an optional byte swapping procedure in between the move to front and entropy encoding steps and the use of

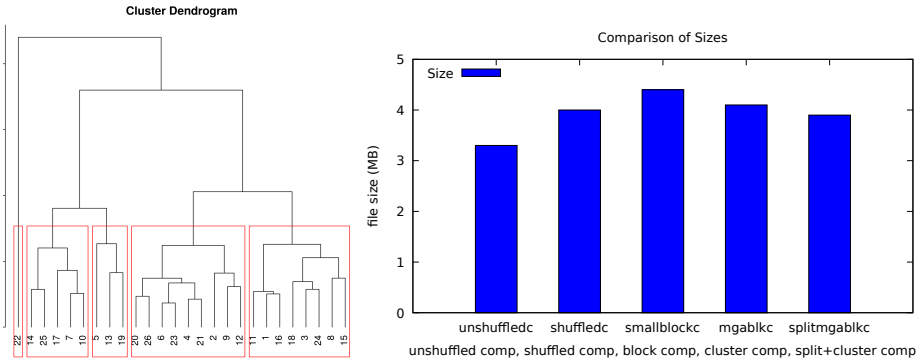

Fig. 9. Sample performance of text/numerical splitting and compression improvements using combined approach with clustering.

an $O(n)$ complexity counting sort with permutation reindexing, followed by bucketing and sorting of multiple smaller $m < n$ size sets, in place of many $O(n \log n)$ sorts. Compression ratio improvements due to content clustering are demonstrated in examples.

Acknowledgement: We are sincerely thankful for DOE funding (contract DE-SC0021467).

References

1. Balkenhol, Bernhard, and Stefan Kurtz. “Universal data compression based on the Burrows-Wheeler transformation: Theory and practice.” IEEE Transactions on Computers 49, no. 10 (2000) : 1043-1053.
2. Burrows, Michael and David Wheeler. “A block-sorting lossless data compression algorithm.” Technical report 124, Digital Equipment Corporation Systems Research Center, 1994.
3. Ester, Martin, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. “A density-based algorithm for discovering clusters in large spatial databases with noise.” In Kdd, vol. 96, no. 3 4, pp. 226-231. 1996.
4. Fenwick, Peter. “Burrows Wheeler Compression.” In Lossless Compression Handbook, pp. 169-194. San Diego, CA: Academic Press, 2003.
5. Gilchrist, Jeff. “Parallel data compression with bzip2.” In Proceedings of the 16th IASTED international conference on parallel and distributed computing and systems, vol. 16, pp. 5 59-564. 2004.
6. Ling, Haibin, and Kazunori Okada. “Diffusion distance for histogram comparison.” In Computer vision and pattern recognition, 2006 IEEE computer society conference on, vol. 1, pp. 246 -253. IEEE, 2006.
7. Mahoney, Matt. “Large text compression benchmark.” URL: http://www. mattmahoney. net/text/text. html (2011).
8. Mahoney, Matt. “10 GB compression benchmark.’ URL: http://mattmahoney.net/dc/10gb.html. (2016).

9. Sayood, Khalid. "Lossless compression handbook." Elsevier, 2002.
10. Sayood, Khalid. "Introduction to data compression." Morgan Kaufmann, 2017.
11. Shkarin, Dmitry. "PPM: One step to practicality." In Proceedings DCC 2002. Data Compression Conference, pp. 202-211. IEEE, 2002.
12. Stokely, Murray, and Tim Hesterberg. "Package 'HistogramTools'." (2013).
13. Vladu, Adrian, and Cosmin Negruseri. "Suffix arrays – a programming contest approach.", GInfo 15, no. 7, 2005.