A LOSSLESS PARALLEL DATA COMPRESSION METHOD USING THREADS

Md. Rafiqul Islam†, Md. Shamsul Arifin†, Mohd. Noor Md. Sap‡ and Kazi Md. Shams Tibrize†
† Computer Science and Engineering Discipline, Khulna University, Khulna, Bangladesh
‡ Faculty of Computer Science and Information System, University of Technology Malaysia, Johor, Malaysia.
E – Mail: cseku@khulna.bangla.net, shumon_84@yahoo.com, mohdnoor@fsksm.utm.my, tibrize@yahoo.com,

Abstract: *The performance of lossless data compression depends mainly on compression rate and compression time. The compression time of maximum of the existing sequential compression algorithms can be reduced using the conception of parallel compression. In this paper, we have proposed a method for data compression, which significantly reduces the compression time. We used Burrows-Wheeler Inversion Coding (BWIC) algorithm and threads in the proposed method. Here, we describe stages of BWIC algorithm and show the efficiency of the runtime of the proposed method over BWIC experimentally.*

Keywords: Burrows-Wheeler Inversion Coding, Block Sorting, Multiset Permutation, Data Compression, Thread.

1. Introduction

Now-a-days computers and other computing machines have to process compute and store huge amounts of data. Some times, it is needed to use huge amount of data and manipulate so much. Data compression is a process that squeezes the data size, removing the excessive information. It may be done in lossless or in lossy way. A shorter data sequence is far better, because it decreases the costs. Without compression, we would have to launch many more satellites to transmit television program. The capability of Internet links is also limited and several methods reduce the enormous amount of transmitted data. The amount of information stored in databases grows fast, while their contents often exhibit much redundancy. A promising development in the field of lossless data compression is the Burrows-Wheeler Compression Algorithm (BWCA), introduced in 1994 by Michael Burrows and David Wheeler [1]. After then several improvements have been applied on different stages on BWCA to increase its performance [2, 3 – 5].

Parallel computing concept is the simultaneous computation among multiple processors. Now data compression concept is not concealed in only single processor. The number of the computer using single processor is extremely huge. Data compression is also an important factor for these computers. When these are used to handle huge amount of data time is an important factor to compress data as these are single processor. Therefore, we have blended the concept of parallel computing in the single processor to have a suitable time for data compression. Jeff Gilchrist has described a method that works with multiprocessor system using BZIP2 for data compression [6]. However, multiprocessor system is not still very much available to end users. Rather the percentage of using single processor is much larger than that of multiprocessor. Therefore, we have proposed here a model for parallel data compression that uses single processor system and thread instead of multiprocessor system, and the Burrows Wheeler Inversion Coding (BWIC) compression algorithm for compression. The purpose of the proposed parallel BWIC compression method is to make a practical parallel compression method that works with most of the real single processor systems. For inversion coding, we have used the method described by Arnavut Z. [7].

The paper is organized as follow: Section 2 describes some related works on BWT, BWIC and parallel compression. Section 3 provides model of BWIC. Section 4 describes different stages of BWIC, Multiset permutation and its implementation and Arithmetic Coding. In section 5, we described our proposed Parallel Data Compression Method. Section 6 and 7 describes time complexity analysis and experimental results that show the supremacy of proposed compression method over BWIC respectively.

2. Related Work

Parallel BZIP2 algorithm of Jeff Gilchrist [6] works by taking the blocks of input data and running them through BWT simultaneously on multiple processors using *pthreads*. The output of his algorithm is fully compatible with the sequential version of BZIP2 [6]. He shows a non-linear speed up achieved by using the parallel BZIP2 program on multiple processors. Implementing in multiprocessor system is complex and this kind of systems is not yet available to all end users.

After proving that BWCA was making a good compression and small time complexities than the existing ones of that time, several improvements on different stages has been done [2, 3 – 5]. Maximum improvements are related to the BWT stage and post BWT stages. Andersson and Nilsson have published several papers about Radix Sort, which can be used as a first sorting step during the BWT [1]. Fenwick describes some BWT sort improvements including sorting long words instead of single bytes [2].

Several techniques for the post BWT stages have been also published. Besides the MTF (Move To Front) improvements from Schindler [8], and from Balkenhol and Shtarkov [3], an MTF replacement, called Inversion Frequencies, was introduced by Z. Arnavut and Magliveras in [9] and in Deorowicz [4] presented another MTF replacement, named Weighted Frequency Count. Fenwick [5], Balkenhol and Shtarkov [3] and Deorowicz [4] presented various modeling techniques for the entropy coding at the end of the compression process.

3. Burrow – Wheeler Compression with Inversion Coding

We shall describe various stages of BWIC [2] in the following section. The different stages of Burrows – Wheeler Compression algorithm are shown in Fig.1.

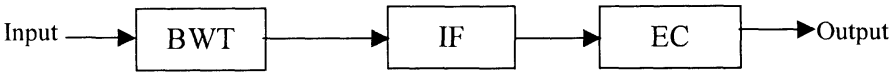

Fig. 1: different stages of compression using BWT

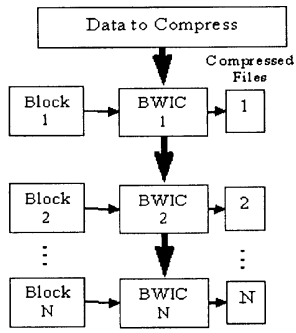

Fig. 2: BWIC Flow Diagram

The BWIC program processes data in blocks as shown in Fig. 2. At first, the input file is subdivided into blocks. It then process compression technique on each block with the BWT (Burrows-Wheeler Transformation) algorithm and each block is compressed sequentially. When one block is compressed, the compressed data is written to the output compressed file on the disk at that time. Then second block is compressed and its compressed data is appended to the same output compressed file. This process continues until all the blocks created are compressed. This continues until the entire set of data is processed. Since BWIC is a sequential program, it will only start processing the next block after the previous one has been completed.

4. Different Stages of BWIC

In this section, we review algorithms of each stage of BWIC that will be used in the proposed model. This section also continues encoding and decoding of Arithmetic coding for the same example.

4.1 The block sorting forward transformation

Burrows, M. and Wheeler D. J. [1] describe the forward transformation as the following steps:

1. Write the entire input as the first row of a matrix, one symbol per column.
2. Form all cyclic permutations of that row and write all the permutation as the other rows of the matrix.
3. Sort the matrix rows according to the lexicographical order of the elements of the rows.
4. Take as output the final column of the sorted matrix, together with the number of the row which corresponds to the original input

This transformation also is also described in another approach in the same paper [1] where the forward transform may be described as:

1. Sort the input symbols, using as a key for each symbol the symbols which immediately follow it. The symbols are therefore sorted according to their following contexts, whereas conventional data compression uses the preceding contexts.
2. Take as output the sorted symbols, together with the position in that output of the last symbol of the input data.

According to this algorithm considering the text ARIFINTIBRIZE we got the output sequence of Forward Transformation EIZIFTRRIBANI.

4.2 The block sorting reverse transformation

The reverse transformation depends on two observations:

1. The transformed input data is a permutation of the original input symbols.
2. Sorting the permuted data gives the first symbol.

The position of the symbol corresponding to the first context is needed to locate the last symbol of the output. From there we can traverse the entire transmitted data to recover the original text.

4.3 Inversion Frequencies (IF)

Several 2nd stages Like MTF have been unveiled since the birth of the BWCA [1]. Their purpose is to produce an output sequence, which is more compressible by the entropy coder than the output sequence of the original MTF stage. One of these MTF replacements is the algorithm from Arnavut and Magliveras [9], which they named Inversion Frequencies (IF). What they suggest was inversion coding for multiset permutation (data strings).

4.3.1 Inversion coding for multiset permutations

Arnavut Z. introduced a new and more memory-efficient inversion technique for multiset permutations (data strings) [7]. In this section we discuss the multiset permutation that is used in Inversion Coding. Here we have used the definitions of Arnavut Z. [7] for multiset permutation to describe our example.

Let $M = [M_1, M_2, \ldots, M_n]$ be a multiset permutation of size n of elements from an underlying ordered m-set S = (1, 2, . . . , m).

From the definition 1 by Arnavut Z. [7] we can have the multiset permutation of [2, 4, 8, 1, 5, 8, 8, 0, 3, 7, 7, 8, 3] yielding [0, 0, 0, 3, 1, 0, 0, 7, 5, 3, 3, 0, 8] as its LBI valued array.

[0|2, 0|4, 0|8, 3|1, 1|5, 0|8, 0|8, 7|0, 5|3, 3|7, 3|7, 0|8, 8|3]

Now from the 2nd definition by the same author we find the group ordered array G_M that is associated with M and the LBI value of array M

For $M_1 = 0$, $G_1 = [7]$; $M_2 = 1$, $G_2 = [3]$; $M_3 = 2$, $G_3 = [0]$; $M_4 = 3$, $G_4 = [5, 8]$; $M_5 = 4$, $G_5 = [0]$; $M_6 = 5$, $G_6 = [1]$; $M_7 = 7$, $G_7 = [3, 3]$; $M_8 = 8$, $G_8 = [0, 0, 0, 0]$

Finally, G = [7, 3, 0, 5, 8, 0, 1, 3, 3, 0, 0, 0, 0]

For the multiset permutation [2, 4, 8, 1, 5, 8, 8, 0, 3, 7, 7, 8, 3], the LBI valued group-ordered array is [7, 3, 0, 5, 8, 0, 1, 3, 3, 0, 0, 0, 0].

From 3rd definition of the same author we find finally the ranked group ordered array RM of M from the LBI group ordered array. Thus, the LBI ranked group-ordered array of the multiset permutation [2, 4, 8, 1, 5, 8, 8, 0, 3, 7, 7, 8, 3] is [7, 3, 0, 5, 3, 0, 1, 3, 0, 0, 0, 0, 0].

Arnavut Z. [7] has described the way to recover the original multiset permutation M from R_M, we need to know the frequency $F = [f_1, f_2, \ldots, f_m]$. From F, it is easy to determine the ordered m-set $S = (1, 2, \ldots, m)$. Initially, let M = [—,—, . . . , —] where $|M| = |R_M| = \sum_{i=1}^{k} f_i$. We can recover elements of M by inserting $i \in S$, for $j = 1, \ldots, f_i$, in the $(r_{i_j} + 1)^{th}$ empty (dash) positions in M. We give an example to illustrate this process.

Suppose $F = [1, 1, 1, 2, 1, 1, 2, 4]$ and $R_M = [7, 3, 0, 5, 3, 0, 1, 3, 0, 0, 0, 0, 0]$. Then we can determine $S = (0, 1, 2, 3, 4, 5, 7, 8)$ from F and create an array M of size $\sum_{i=1}^{8} f_i = 13$. From F, we know that there are one 0 in multiset permutation M. the first one entry in R_M is the rank related to 0 in the first pass, we insert 0 in their location in M. Since the first entry in R_M is 7, it follows that the location of the first element in R_M is at position 8, hence M = [—,—,—,—,—,—,—, 0,—,—,—,—,—].

For 0, M = [—,—,—,—,—,—,—, 0,—,—,—,—,—]

For 1, M = [—,—,—, 1,—,—,—, 0,—,—,—,—,—]

For 2, M = [2,—,—, 1,—,—,—, 0,—,—,—,—,—]

For 3, from F we get frequency is 2 and from R_M we get 5, 3. It follows the location in M, 5 + 1 = 6 and after 6, 3 + 1 = 4.

For 4, M = [2, 4,—, 1,—,—,—, 0, 3,—,—,—, 3]

For 5, M = [2, 4,—, 1, 5,—,—, 0, 3,—,—,—, 3]

For 7, M = [2, 4,—, 1, 5,—,—, 0, 3, 7, 7,—, 3]

For 8, M = [2, 4, 8, 1, 5, 8, 8, 0, 3, 7, 7, 8, 3]

In this way we reconstruct M.

4.3.2 Implementation of Inversion coding for multiset permutation

For the implementation of Inversion Coding for multiset permutations in BWIC, the algorithm proposed by Arnavut Z. has been taken to use here.

For the implementation, a binary tree is constructed with m (m is the size of alphabet of order set S) nodes. There are 4 - tuple $[i, a_i, pa_i, pos_i]$ in each node. The variable i, $i \in S$ is the search key. The variable $a_i \geq$ i (i = Mt) evaluates the

number of elements of M (multiset permutation) at time t to the left of i by processing M from left to right. The variable pa_i denotes of the number of elements of M greater than or equal to i ($i = M_{t-1}$) at time ($t - 1$) to the left of i. Initially both variables a_i and pa_i are set to zero. Initially, variable pos_i is set to the starting position of each LBI ranked group R_i of R_M (Ranked group array of M). Whenever M is known, pos_i can be initialized as follows:

$pos_i \leftarrow 1, pos_i = f_1 + i, \ldots, pos_m = \sum_{i=1}^{m-1} f_i + 1$. Here f is the frequency of ordered m-set $S = (1, 2, \ldots m)$ of multiset permutation M.

Let *low* and *high* be two integer variables where initially, for the root node, *low* is set to 1 and *high* is set to m. The initial balanced binary tree is then constructed recursively. Information fields of each node are computed according to Arnavut Z. [7] as follows:

Step 1. Initialize $i \leftarrow$ (low + high) / 2, $a_i \leftarrow 0$, $pa_i \leftarrow 0$ and $pos_i \leftarrow \sum_{j=1}^{i-1} f_j + 1$.

Step 2. If $low \neq high$, then compute descendants of a node as follows:

a. The *high* value for left descendants of a node is $high \leftarrow VP - 1$, where VP is the value I of the parent node.

b. The *low* value for right descendants of a node is $low \leftarrow VP + 1$, where VP is the value i of the parent node.

Step 3. If $low = high$, set $i = low$ and return.

Whenever a balanced binary tree is constructed, in order to get the LBI rank group-ordered array R_M of M, the tree should be traversed starting from the root until M_k is located in the key value field (i) of a node, for every k, $1 \leq k \leq n$. Then following steps are maintained:

(i) The values a_j of the nodes whose key values $\geq M_k$ has been built up i.e. obtain $\sum_{j \geq M_k} a_j$, and move to the left subtree;

(ii) If a node's key value field is less than M_k, then $a_j \leftarrow a_j + 1$, and a_j is moved to the right subtree.

When the node that contains the value M_k is reached,

(i) The rank $r_{i_t}\left(\leftarrow \sum_{j \geq i}^{t} a_j - pa_i, 1 \leq t \leq f_i \right)$ is calculated and inserted into position pos_i in R_M.

(ii) The values of a_i and pa_i are updated where $a_i \left(\leftarrow a_i + 1\right)$ and $pa_i \leftarrow \left(\sum_{j \geq i}^{t} a_j + 1, 1 \leq t \leq f_i\right)$.

(iii) pos_i is incremented by 1 so that next time it points to the correct position.

For M = [3, 2, 1, 4, 1, 3, 2, 1, 4, 2, 4] and the ordered set S = (1, 2, 3, 4) with F = [3, 3, 2, 3] a tree can be constructed using the algorithm described above:

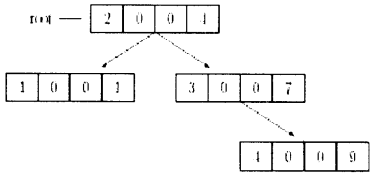

Fig. 3: Initial state of balanced binary tree.

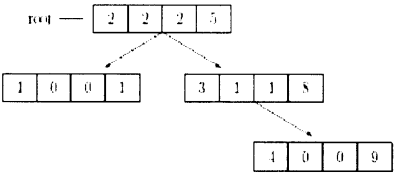

Fig. 4: After processing two elements of M

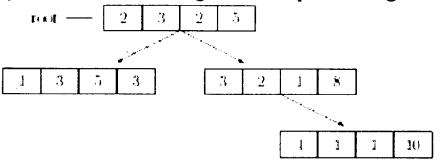

Fig. 5: Balanced binary tree after processing five elements of M

4.4 Arithmetic Coding

For having the output of arithmetic coding process, we need to find first the probability of the symbol of massage or string where the probability will range from 0 to 1. Then we have to encode the message according to the algorithm of arithmetic encoding. From the arithmetic encoding, we shall get an encoded number for the message. That number will be used to decode the message by the algorithm of arithmetic decoding with the help of the probability table of the symbol.

For example, if we were going to encode the random message {2, 4, 8, 1, 5, 8, 8, 0, 3, 7, 7, 8, 3}. We would have a probability distribution that looks like this:

Table-4.1: Illustration of Arithmetic Coding

Character	Probability	Range (Low – High)
0	1 / 13	0.000 – 0.075
1	1 / 13	0.075 – 0.150
2	1 / 13	0.150 – 0.225
3	2 / 13	0.225 - 0.375
4	1 / 13	0.375 – 0.450
5	1 / 13	0.450 – 0.525
7	2 / 13	0.525 – 0.675
8	4 / 13	0.675 – 1.000

We can encode any message of any length and then decode it to original message using the encoding and decoding algorithms of Arithmetic Coding proposed by Ian H. [10]. According to this algorithm the final low value, 0.1822008352 will uniquely encode the random message {2, 4, 8, 1, 5, 8, 8, 0, 3, 7, 7, 8, 3}. The decoding algorithm of Arithmetic Coding can decode the low value to the original message.

5. Parallel Data Compression Method

Here we shall propose a method that will make the parallel data compression using threads in single processor systems. The method uses the following steps to accomplish the task:

Step 1. Split the Original file into *N* small files that will be the input for the next step.

Step 2. Create a thread for each splitted file so total number of threads will be *N*.

Step 3. All splitted files will be sequenced using a list.

Step 4. Compress each splitted file in thread using BWIC algorithm.

Step 5. Intermediate compressed splitted file will be produced in each thread.

Step 6. Merge all intermediate compressed splitted files produced in step 5.

Step 7. Merged file will be the output compressed file.

According to this method, firstly we have made several splitted files from the input file and these splitted files were sequenced in a list then each of these files was compressed independently by BWIC in parallel using threads. By using threads and dividing main file into splitted files, we shall show that the compression time is significantly less than compressing the file without using threads experimentally. We have described BWIC algorithm in section 3 that constructs a balanced binary tree and we have used thread in the proposed model.

The proposed model reduces the run-time with BWIC using single processor and thread. We have made several splitted files of the input file and each splitted file is the input of each BWIC algorithm. Each splitted file is compressed as the threads run.

The benefit of multithreading is that the main loop/polling mechanism is eliminated. One thread can pause without stopping other parts of the program. Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause. Multithreading enables to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum. This is especially important for the interactive systems because idle time is common. Even local file system resources are read and written at a much slower pace than the single-threaded environment can process them; program has to wait for each of these tasks to finish before it can proceed to the next one – even though the CPU is sitting idle most of the time. Multithreading lets gain access to this idle time and put it to good use.

By using BWIC and thread, the proposed model is depicted below:

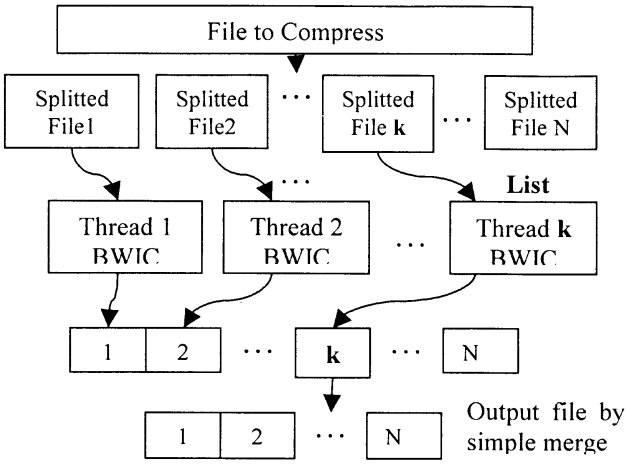

Fig. 6: The Proposed Model.

6. Time Complexity Analysis

In BWIC, to generate R_M (Ranked grouped array for multiset permutation in section 3.3), from a given multiset premeditation M with n elements requires $O(n \log m)$ time since the depth of the height-balanced tree cannot be bigger than $\log(m) + 1$, where m is the size of alphabet (ordered set S). In case of parallel BWIC (proposed algorithm), the size of alphabet will be $\frac{m}{N}$, where N is the number of threads used in the proposed algorithm.

To obtain the data string (multiset permutation) from its corresponding inversion ranks, the algorithm described in the previous section 3.3.1 utilized a binary tree data structure with T nodes in sequential but in case of parallel BWIC it is T_1 ($T_1 \leq T$). It is well known that height balancing cannot be guaranteed with binary tree data structures. Therefore, AVL tree is implemented. The worst case running time of the proposed algorithm with a height-balanced tree data structure is as follows: each time we select T elements from $n = |R_M|$. Selecting T elements out of n requires a constant time overhead, say c, hence the total time required to select all the elements is $O(c \frac{n}{T})$. Building an RB tree with T nodes requires $O(T \log T)$. To traverse the tree with T nodes in-order requires $O(T \log T)$. Therefore, the total running time is: $c\frac{n}{T}(T \log T + T \log T)$. In terms of the worst case behavior the running time is then $O(n \log T)$. But as we have used thread and the input file has been splitted, so the height of the balanced tree will be $\log(m_l) + 1$ for each splitted file where $m_l \leq m$ and m_l is the size of alphabet (of order set S) for splitted file. In the section 4, the proposed algorithm utilizes a binary tree data structure of T_l nodes as well sequential BWIC data structure where $T_l \leq T$. As we splitted the input files into N splitted files, so the total elements of each splitted file will be $\frac{n}{N}$ where

N is the number of Threads. So, selecting T_1 elements out of $\frac{n}{N}$ requires a constant time overhead, say c, hence the total time required to select all the elements of the splitted file is $O(c\frac{n}{N T_1})$. So, here we again compute run time for the algorithm.

Building an AVL or Red – black tree with T_1 nodes requires $O(T_1 \log T_1)$, here $T_1 \leq T$.

To traverse the T_1 nodes in-order requires $O(T_1 \log T_1)$.

Therefore, total running time of the proposed algorithm is $c\frac{n}{N T_1}$ $(T_1 \log T_1 + T_1 \log T_1)$.

In terms of asymptotic Φ notation the time complexity is $O(\frac{n}{N} \log T_1)$ which is smaller than that of sequential BWIC $O(n \log T)$.

7. Experimental Results

To illustrate the proposed model, let us consider the file *hi* from the *Protein Corpus*. As we have used N ($N = 2, 3, 4$) threads, we are describing here the procedure for using 2 threads. According to step 1, we split the input file *hi* into two smaller splitted files named *hi_1* and *hi_2*. As in step 2, we create two threads for compressing these smaller files. In step 3, the sequence of splitted files (here *hi_1* and *hi_2* respectively) is listed. This list will be used later while merging temporary compressed files to produce the output-compressed file in proper sequence. In these two threads the BWIC compression algorithm was used taking as input the two splitted files *hi_1* and *hi_2* to produce temporary intermediate compressed files according to step 4. Two threads will create two intermediate compressed files. Finally, these intermediate compressed files are merged according to listed sequence to obtain the output-compressed file.

For experimental results, we implemented the proposed method in JAVA. Our used configuration consisted of Athlon Sempron 2400+ (1.6 GHz) processor, 320 MB of RAM in Windows XP. For experimental results we have used *Protein Corpus.* Execution time of file compression is too low and measured in milliseconds (ms). So, recorded time was the average of 50 run of BWIC for compression that is used in our program.

The performance of the proposed model over sequential BWIC is its run time. Table-7.1 shows the comparison of the time required to compress the file between the sequential BWIC and Parallel BWIC using Threads. In Fig.7, data of Table-7.1 has been depicted graphically. Here, compressing time decreases with the increase of threads used for large files.

Table-7.1: Experimental Results

File Name	File Size (byte)	Time for BWIC (millisecond)	Time for Parallel BWIC (millisecond) No. of threads = 2	Time for Parallel BWIC (millisecond) No. of threads = 3	Time for Parallel BWIC (millisecond) No. of threads = 4
Hi	509,519	262.14	104.41	82.18	53.76
Hs	3,295,751	1048.44	524.68	488.76	441.24
Sc	2,900,352	1190.32	959.4	463.44	389.68
Mj	448,779	157.8	112.8	75	55.94

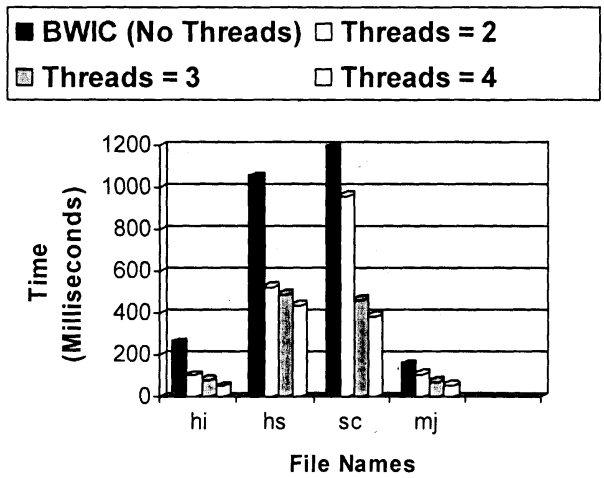

Fig. 7: Runtime comparison of BWIC and Parallel BWIC for different Threads

8. Conclusion

Compression time is an important factor as well as compression ratio. Whenever we compress a file, we have to measure compression time as we measure compression rate. Though compression rate may be better but if compression time is not that good that does not well in different areas. So, we proposed a parallel BWIC algorithm using Thread, which gives very good compression rate and reduces the compression time. Here compression rate does not fall when we use this method but the reduction of time is outstanding. That means a simple modification of BWIC algorithm in its implementation gives significant performance in execution time.

We have been able to show the reduction of time for the data compression in the single processor using threads. We are trying for the improvement in case of data compression as well as its executing time. Since we are using BWIC as the compression algorithm we want to make improvement in it as it can give better compression as well as it takes small time for execution. So the blending of the better data compression and less time for execution of compression algorithm by using thread will give a better result in future.

References

[1] Burrows, M. and Wheeler, D. J. (1994) A *block sorting Lossless Data Compression Algorithm,* SRC Research Report 124, Digital System Research center, Palo Alto, CA, gatekeeper.doc.com,/pub/DEC/SRC/research-reports/SRC-124.ps.Z.

[2] Fenwick P., *The Burrows-Wheeler Transform for block sorting text compression: principles and improvements*, The Computer Journal 1996, 39 (9), 731-740.

[3] Balkenhol, B., Kurtz, S. and Shtarkov, Y. M., *Modifications of the Burrows and Wheeler Data Compression Algorithm*, In Proceedings of the IEEE Data Compression Conference 1999, Snowbird, Utah, STORER, J. A. AND COHN, M. Eds. 188-197.

[4] Deorowicz, S., *Improvements to Burrows-Wheeler Compression Algorithm*, Software Practice and Experience 2000, 30 (13), 1465-1483.

[5] Fenwick, P. 1995, *Improvements to the Block Sorting Text Compression Algorithm*, Technical Report 120, University of Auckland, New Zealand, Department of Computer Science.

[6] Jeff Gilchrist, Elytra Enterprises Inc., *Parallel Data Compression With BZIP2.*

[7] Arnavut Z., *Inversion Coding*, The Computer Journal, Vol. 47, No. 1, 2004, 16 September 2002.

[8] Schindler, M., *A Fast Block-sorting Algorithm for lossless Data Compression*, In Proceedings of the IEEE Data Compression Conference 1997, Snowbird, Utah, STORER, J. A. AND COHN, M. Eds. 469.

[9] Arnavut Z. & Magliveras, S. S. 1997, *Block Sorting and Compression*, In Proceedings of the IEEE Data Compression Conference 1997, Snowbird, Utah, Storer, J. A. and Cohn, M. Eds. 181-190.

[10] Ian H. Witten, Radford M. Neal, John G. Cleary, *Arithmetic coding for data compression*, communications of the ACM, June 1987 volume 30 number 6.