15418 Parallel Computer Architecture and Programming Project Report: Implementation and Comparison of Parallel LZ77 and LZ78 Algorithms

Fuyao Zhao
fuyaoz@cs.cmu.edu

Nagarathnam Muthusamy
nmuthusa@andrew.cmu.edu

Abstract

Our project developed an linear work parallel LZ77 compression algorithm with unlimited window size. We also implemented two version of LZW compression algorithms. We did cross comparison of all algorithms and gave suggestions on how to choose an algorithm for real application.

1 Introduction

LZ77 and LZ78 are the two most common loss-less data compression algorithms, which are published by Abraham Lempel and Jacob Ziv in 1977 [25] and 1978 [26]. These two algorithms form the basis for many variations including LZW [24], LZSS [23], and others, they are also the basis of several ubiquitous compression schemes, including GIF and the DEFLATE algorithm used in PNG.

They are both theoretically dictionary coders. LZ77 maintains a sliding window during compression. This was equivalent to the explicit dictionary constructed by LZ78 however, they are only equivalent when the entire data is intended to be decompressed. LZ78 decompression allows random access to the input as long as the entire dictionary is available,while LZ77 decompression must always start at the beginning of the input.

However, little work has been done for parallelizing the algorithms, [15] provides a summary of work done on parallelizing LZ family of algorithms. [18] constructs a binary tree dependency out of the text to be compressed by multiple processors, this technique can be used for all variants of LZ77 and LZ78 algorithm. [20] uses different dictionaries for different length strings enabling parallel lookup for LZW. [14] describe an $O(n \log n)$ work parallel algorithm for LZ77 algorithm, however they are working on an modified version of LZ77 so compression ratio is not optimal.

No linear work parallel LZ77 algorithm has been done to our knowledge, and there is no good performance comparison between those algorithm, which become our motivation of the project.

The rest of the report is organized as follow: Section 2 describe the algorithms we used for parallel LZW and LZ77. Section 3 details the our implementation we used. Experiments are done in Section 4, and result is analyzed in Section 5. We conclude our work and discuss our future work in Section 6.

2 Algorithm

2.1 LZW

The sequential LZW algorithm starts by initializing the dictionary to contain all strings of length one. Then it proceeds on finding the longest string in the dictionary that matches the current input and emits the code for it. It adds the matched string along with its following character into the dictionary. This goes on till all the characters in the input are processed.

LZW(s, n)

```
1: dict ← Initialize Dictionary
2: index ← 0
3: LW[index] ← ϵ
4: i ← 0
5: while i ≤ n do
6:     j ← min{j | existed(dict, s[i..j])}
7:     put(dict, s[i..j])
8:     LW[index] ← get(dict, s[i..j − 1])
9:     index ← index + 1
10:    i ← j
11: end while
```

Consider an input string *awedawe*. Stepping through the start of the algorithm for this string, we can see that the first pass through the loop, a check is performed to see if the string *aw* is in the table. Since it isn't, the code for *a* is output, and the string *aw* is added to the table. Since we have 256 characters already defined for codes 0-255, the first string definition can be assigned to code 256. After the third letter, *e*, has been read in, the second string code, *we* is added to the table, and the code for letter *w* is output. This continues until in the second word, the characters *a* and *w* are read in, matching string number 256. In this case, the code 256 is output, and a three character string is added to the string table. The process continues until the string is exhausted and all of the codes have been output. Table 1 shows the whole process of encoding.

Input String = *awedawe*			
Character Input	Code Output	New Code Value	New String
aw	97	256	*aw*
e	119	257	*we*
d	101	258	*ed*
a	100	259	*da*
we	256	260	*awe*
	101		

Table 1: The example of encoding string *awedawe* using LZW.

If we have N processors, the simplest way to parallelize LZW is to split the data into N chunks and assign a processor to each chunk for compression (PLZW1). This will reduce the computation time as each processor operates on a chunk independently. The limitation of this method is the increase in the number of redundant entries on the whole in the dictionaries (same entries being put into different dictionaries) in the processors which will reduce the compression ratio.

In order to overcome the limitations of **PLZW1** and improve the compression ratio, the data is split into multiple chunks and a binary tree is constructed out of those chunks. The compression starts by assigning the first processor to compress the root node. Each processor after compressing

PLZW1(s)

1: $N \leftarrow$ number of processors
2: $blocks \leftarrow$ Split s into N blocks
3: **for parallel** i **from** 1 **to** N **do**
4: **LZW**$(blocks[i])$
5: **end for**
6: output the code for each block

their assigned node starts compressing the left child of current node while assigning a new processor for compressing the right child. Figure 2.1 shows how blocks are divided among 8 processors.

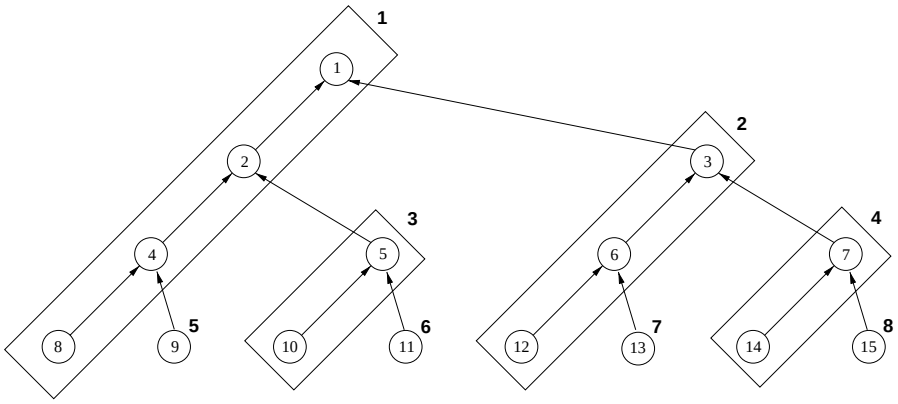

Figure 1: The nodes of the tree denotes the blocks. Since we have 8 processors, we divide the data into 15 blocks. The outer rectangle groups the blocks processed by a processor. The numbers outside the rectangle denote the id of the processor which is used to compress the given block.

Each processor maintains its own dictionary and also uses its parent's dictionary for compression, which is different from original **LZW** (say **LZW′**). For **LZW′** we redefine the look up function of dictionary. The normal **existed** accepts the dictionary and string as argument and looks up the particular string in the dictionary. We modify it in the following **existed′** such that it looks up the given string in the dictionary maintained by the processor with the specified ID and also in all its parent processors. By doing this, we are able to share parts of the dictionaries among processors to reduce the probability that same entries being put into different dictionaries. So by doing this way we would have better compression ratio than PLZW1.

Let d_i denotes the dictionary of ith processor, **existed′** will be:

existed′(s, j)

1: **while** $j \neq -1$ **do**
2: **if existed**$(d_j, s) = false$ **then**
3: $j \leftarrow$ **parent**(j)
4: **else**
5: **return** $true$
6: **end if**
7: **end while**
8: **return** $false$

The whole algorithm is shown as follows:

PLZW2(i, j)

1: **LZW′**$(blocks[i])$

2: Do in parallel $\begin{cases}\textbf{PLZW2}(2i, j)\\ \textbf{PLZW2}(2i+1, i+1)\end{cases}$

The algorithm starts with PLZW2(1,1). The first processor starts compressing the block 1. After compressing block 1 the first processor moves to block 2 and assigns processor 2 to block 3. Every processor after compressing their own block move to the left child and assign a new processor for compressing the right child. The children can use the dictionary of parent processors to compress.

2.2 LZ77

LZ77 uses the idea of sliding window and is related many applications other than compression. It will find the longest match between the string in current window and the string occurs previously, and then the window will slide based on the length of the matching.

Formally, let $S = s_1 s_2 \dots s_n$ be a string of length n, $S[i \dots j]$ be the substring of S begin from letter i to j and $|S|$ be the length of the string. $\mathsf{LCP}(A, B)$ is the longest common prefix of strings A and B. $\mathsf{LPF}[i]$ is the length of longest previous factor (LPF) of $S[i \dots n]$, defined as

$$\mathsf{LPF}[i] = \max\big(\{l \mid S[i \dots i+l-1] \text{ is a factor of } S[0 \dots i+l-1]\} \cup \{0\}\big).$$

The LZ77 algorithm will decompose the S into m factors so that $S = U_1 U_2 \dots U_m$. We denote $d = |U_0 U_1 \dots U_{i-1}| + 1$ and $Q = U_i U_{i+1} \dots U_m$, so

$$U_i = \begin{cases} S[d \dots d + \mathsf{LPF}[d] - 1] & \text{if } \mathsf{LPF}[d] > 0 \\ s_d & \text{if } \mathsf{LPF}[d] = 0. \end{cases}$$

For example, the string $S = ababaab$ can be factorize as $S = a/b/aba/ab$. For simplicity of discussion, we use $\mathsf{LZ}[i]$ to denote the beginning of ith factor in S. So in this example, $\mathsf{LZ} = \{0, 1, 2, 5\}$. To decompress the, we will need not only LZ, but also the array LEN indicating the matching length where $\mathsf{LEN}[i] = \mathsf{LPF}[\mathsf{LZ}[i]]$, $0 \le i < |\,\mathsf{LZ}\,|$

Multiple solutions existed in sequential algorithm [5, 11, 13], our solution is based on parallelize [12]. The idea is to build a suffix array [21], which represents the result of sorting all the suffixes of the string. Then to find the (Longest Previous Factor) for each suffix, which can be proved to the nearest smaller value in suffix array [12]. For example, gives $S = abbaabbbaaabab$, the suffix array and the LPF will be like Figure 2:

i	s_i	$\mathsf{SA}[i]$	$\mathsf{lcp}[i]$	suf_i	$\mathsf{LPF}[SA[i]]$
0	a	8	0	$aaabab$	2
1	b	9	2	$aabab$	3
2	b	3	3	$aabbbaaabab$	1
3	a	12	1	ab	2
4	a	10	2	$abab$	2
5	b	0	2	$abbaabbbaaabab$	0
6	b	4	3	$abbbaaabab$	3
7	b	13	0	b	1
8	a	7	1	**baaabab**	3
9	a	2	3	$baabbbaaabab$	1
10	a	11	2	bab	2
11	b	6	1	$bbaaabab$	4
12	a	1	4	$bbaabbbaaabab$	0
13	b	5	2	$bbbaaabab$	2

Figure 2: Suffix array of $S = abbaabbbaaabab$. To get the LPF for suffix **baaabab** starting from 7, we can find the left nearest value who starts before 7, which is the suffix startting from 4, and also the right nearest value who starts before 7, which is the suffix starting from 6, then we found the later one have longer match to current suffix (3 characters), so $\mathsf{LPF}[\mathsf{SA}[7]] = 3$.

The construction of suffix array can be parallelized [17, 19], in addition lcp could be computed as byproduct while we parallel construct suffix array [17]. The parallel algorithm need linear work and $O(\log^2 n)$ time.

Then we run ANSV algorithm in parallel, which is also very studied in [7, 16] and has linear work with $O(\log \log n)$ time. However, we will not use parallel version of `getLcp`. Since $\mathsf{LCP}(i, j) = \min(\mathsf{lcp}[k]),\ i < k \leq j$ [21], it is actually a range minimum query (RMQ) problem on lcp. Since RMQ is another classical problem whose preprocessing could be done in parallel with linear work and $O(\log n)$, and the query could be finished just in constant time [8, 7].

After computing the LPF, following algorithm would compute LZ array to get the final result.

LZ77(LPF, n)

1: $\mathsf{LZ}[0] \leftarrow 0$
2: $i \leftarrow 0$
3: **while** $\mathsf{LZ}[i] < n$ **do**
4: $\quad \mathsf{LZ}[i+1] \leftarrow \mathsf{LZ}[i] + \max(1,\ \mathsf{LPF}[\mathsf{LZ}[i]])$
5: $\quad i \leftarrow i + 1$
6: **end while**
7: **return** LZ

The problems looks similar to prefix sum, but it gets tricky to actually transform it into prefix sum problem. To simplify the problem, let $\mathsf{next}[i] = \min(n, i + \max(\mathsf{LPF}[i], 1))$ for $0 \leq i \leq n$ and $\mathsf{next}[n] = n$, so we have

$$\mathsf{LZ}[i+1] \leftarrow \mathsf{next}[\mathsf{LZ}[i]]$$

Also, the next array constructs a tree, whose root is n and every other's parent is $\mathsf{next}[i]$, then problem then will be, find out the particular path $p_0 = [0, \mathsf{next}[0], \mathsf{next}[\mathsf{next}[0]], \cdots, n]$ from leaf 0 to the root, notice p_0 just has one more element n than LZ.

Let's $\mathsf{flag}[i] = 1$ if $i \in p_0$ and $\mathsf{flag}[i] = 0$ otherwise, then by applying parallel prefix sum with linear work and $O(\log n)$ time, we could get the list in which all the element has $\mathsf{flag}[i] = 1$, in another word, the path p_0, the algorithm is show in **ParallelLZ77**.

ParallelLZ77(LPF, n)

```
1: LZ[0] ← 0
2: i ← 0
3: for parallel i from 0 to n do
4:    next[i] ← min(n, i + max(LPF[i], 1))
5: end for
6: flag = PointJump(next, n)
7: flag = exclusiveScan(next, n)
8: for parallel i from 0 to n − 1 do
9:    LZ[flag[i]] ← i
10: end for
11: return LZ
```

To parallel compute flag, we could use pointer jumper algorithm [9] which could be done in linear work and $O(\log^2 n)$ time [6] by using random sample technique. To clearly show how our algorithm works, we will just show the simplest $O(n \log n)$ work and $O(\log n)$ time algorithm in **PointJump**. However, same technique could be easily applied to random sample version [6].

PointJump(next, n)

```
1: flag[0] ← 1
2: flag[i] ← 0 for i > 0
3: for j from 1 to |log n| do
4:    for parallel i from 0 to n − 1 do
5:       if flag[i] = 1 then
6:          flag[next[i]] ← 1
7:       end if
8:       next[i] ← next[next[i]]
9:    end for
10: end for
11: return flag
```

Overall, the parallel algorithm for LZ77 has linear work and $O(\log^2 n)$ time.

3 Implementation

All our code is written in C++ with using of standard C++ library, we use OpenMP [3] to achieve parallelism, mostly by using directive `#omp parallel for`.

3.1 LZW

We construct a dictionary consists of 5000 entries for each processor. The dictionary is made up of a hash table of size 50000 with linear probing for collision resolution. We use Arash Partow's hash algorithm [1] in our hash table. The algorithm outputs code for strings that are already known and adds the unknown strings from the stream to the dictionary.

An array `flags` is created with every block assigned a flag. The flag of block 1 is set to 1 while the other flags are set to zero. The processor proceeds to compress the block only if the flag is set to 1. Initially the ith processor waits on the flag of block $2 \times i - 1$. In this case only processor 1 can start the compression. When a processor finishes the compression of a block it moves to the left

child of the block being compressed and sets the flag for the right child enabling other processor to start compressing in parallel.

During the process of compression the processor will first search its dictionary for the string. If the string is new to its dictionary, it will search for the string in the dictionaries of parent processors. If it is found no-where it a new entry is added to the dictionary of compressing processor else the index of the dictionary in parent processor is used along with the processor ID to encode the string.

In the current algorithm, the search string is appended with characters sequentially till we get a string which is not in the dictionary. [20] provided the idea of parallel dictionary look up for strings of different length. We implemented it by making each thread search the dictionary independently for strings of different length.

By this method we could skip the sequential search for longest string not present in the dictionary. However, since our dictionary contained only 5000 entries the maximum length of a single string in the dictionary was short that is not worth parallel searching. The cost of creation and synchronizing the those threads in OpenMP overshadowed the performance gain that would be obtained so we decided to drop this approach.

3.2 LZ77

We adopt the implementation of parallel suffix array from [4], in which the code is optimized for Intel Cilk [2]. However, since the original program uses a lot of nested parallelism that doesn't get well support from OpenMP, we rewrote the parallel merge and parallel sorting, we tried 3 parallel sorting algorithms: original radix sort, parallel merge sort and parallel sort by regular sampling [22]. Among them PSRS is most scalable, and we will give more result in following section. Suffix array construction is also the most expensive part of the algorithm.

We implemented the all nearest value algorithm in $O(n \log n)$ version, while the theoretical linear version is too complex to implemented in short period of time and should have a very big factor. In experiment we will see that this part didn't take huge portion of computation.

We implemented the parallel range minimum query problem in $O(n \log n)$ as well (linear one would need linear all nearest value algorithm). Particularly, we tried to explore the locality by making each processors do sequential algorithm on a chunk.

We implemented both the $O(n \log n)$ and $O(n)$ version of pointer jumper problem, the later one is much faster, and it seems the actual performance might be sub-linear on our particular data.

We also implemented the $O(n \log n)$ and $O(n)$ version of prefix sum, and we choose later one, and we optimize it by making each processors do sequential prefix sum a chunk.

4 Experiements

The experiments are done on Blacklight super computer of Pittsburgh Super Computing Centre. It is a 4096 core NUMA shared memory system of which we were allocated 16 cores during our experiments. OpenMP [3] was used to implement all the algorithms.

We we both random text and real text, for random text. The random alphabets data set was prepared using the 'rand()' function of C, with alphabet size of 4 to 128. For English text we use linux code, which is prepared by merging all the files of 'linux-3.4-rc4' source code. The size for all data size are all 64MB.

We measure the implementations by wall clock time, speedup and compression ration. The wall clock time includes computation time and the synchronization time, which are obtained by using timers during the starting and ending of the core algorithm. The I/O time and other delays like time for memory allocations are ignored.

Table 2∼8 shows all our experiments results, Figure 4 visualize the results we got for compressing linux kernel code and random text with alphabet size 16.

Table 2: Experimental results on running the algorithm over Linux Code

	PLZ77			PLZW1			PLZW2		
Cores	Time	Speedup	Compressed	Time	Speedup	Compressed	Time	Speedup	Compressed
1	118.00	1.00	3638369	3.80	1.00	30659467	3.88	1.00	30659422
2	62.50	1.89	3638369	1.95	1.95	30653559	2.58	1.51	29790069
4	34.00	3.47	3638369	1.01	3.75	31603467	1.77	2.19	27662585
8	20.90	5.65	3638369	0.55	6.91	30950116	1.13	3.42	26617332
16	14.00	8.43	3638369	0.31	12.35	31794407	0.80	4.81	25419950

Table 3: Experimental results on running the algorithm over Alphabet size 4

	PLZ77			PLZW1			PLZW2		
Cores	Time	Speedup	Compressed	Time	Speedup	Compressed	Time	Speedup	Compressed
1	86.20	1.00	5525283	2.51	1.00	11899385	2.63	1.00	11884439
2	43.50	1.98	5525283	1.25	2.00	11895516	1.71	1.54	11615788
4	22.70	3.92	5525283	0.63	3.98	11901535	1.38	1.91	11346527
8	14.30	6.03	5525283	0.32	7.78	11906504	0.92	2.88	11143676
16	11.00	7.84	5525283	0.18	13.95	11915801	0.61	4.34	10935448

Table 4: Experimental results on running the algorithm over Alphabet size 8

	PLZ77			PLZW1			PLZW2		
Cores	Time	Speedup	Compressed	Time	Speedup	Compressed	Time	Speedup	Compressed
1	83.20	1.00	8460035	2.51	1.00	18111076	2.72	1.00	18036313
2	43.10	1.93	8460035	1.26	1.99	18111001	1.71	1.59	17609128
4	23.00	3.87	8460035	0.64	3.89	18106110	1.31	2.07	17181581
8	13.70	6.07	8460035	0.34	7.47	18109767	0.93	2.94	16847221
16	9.80	8.49	8460035	0.20	12.84	18114143	0.57	4.76	16514651

Table 5: Experimental results on running the algorithm over Alphabet size 16

	PLZ77			PLZW1			PLZW2		
Cores	Time	Speedup	Compressed	Time	Speedup	Compressed	Time	Speedup	Compressed
1	96.10	1.00	11502979	2.89	1.00	24762932	3.21	1.00	24749573
2	49.30	1.95	11502979	1.47	1.96	24749739	1.97	1.63	23970565
4	26.20	3.40	11502979	0.76	3.79	24731518	1.34	2.39	23360617
8	15.40	6.24	11502979	0.40	7.30	24732399	0.97	3.32	22889490
16	9.71	9.90	11502979	0.24	12.15	24738264	0.64	5.02	22449464

Table 6: Experimental results on running the algorithm over Alphabet size 32

	PLZ77			PLZW1			PLZW2		
Cores	Time	Speedup	Compressed	Time	Speedup	Compressed	Time	Speedup	Compressed
1	94.70	1.00	14718845	2.90	1.00	31846724	3.54	1.00	31872203
2	48.60	1.95	14718845	1.46	1.99	31856228	1.97	1.79	31290480
4	26.00	3.42	14718845	0.75	3.86	31854387	1.39	2.55	30674316
8	15.20	6.23	14718845	0.39	7.45	31882190	0.90	3.93	30202899
16	9.50	9.97	14718845	0.23	12.45	31893344	0.60	5.86	29552034

Table 7: Experimental results on running the algorithm over Alphabet size 64

	PLZ77			PLZW1			PLZW2		
Cores	Time	Speedup	Compressed	Time	Speedup	Compressed	Time	Speedup	Compressed
1	77.30	1.00	17863712	3.48	1.00	39172693	3.56	1.00	39307922
2	39.90	1.94	17863712	1.65	2.11	39329339	2.22	1.61	37761754
4	21.10	4.22	17863712	0.85	4.11	39375928	1.44	2.48	36503778
8	12.30	6.28	17863712	0.46	7.62	39335587	1.05	3.39	35666135
16	7.72	10.01	17863712	0.29	11.87	39326359	0.65	5.47	34846443

Table 8: Experimental results on running the algorithm over Alphabet size 128

	PLZ77			PLZW1			PLZW2		
Cores	Time	Speedup	Compressed	Time	Speedup	Compressed	Time	Speedup	Compressed
1	77.20	1.00	21832551	3.51	1.00	53517150	3.68	1.00	53359862
2	39.80	1.94	21832551	1.75	2.00	53439151	2.44	1.51	51074114
4	21.00	4.24	21832551	0.90	3.89	53382627	1.51	2.43	48800231
8	12.30	6.28	21832551	0.49	7.21	53387358	1.05	3.51	46907442
16	7.93	9.74	21832551	0.30	11.66	53395380	0.68	5.43	45034548

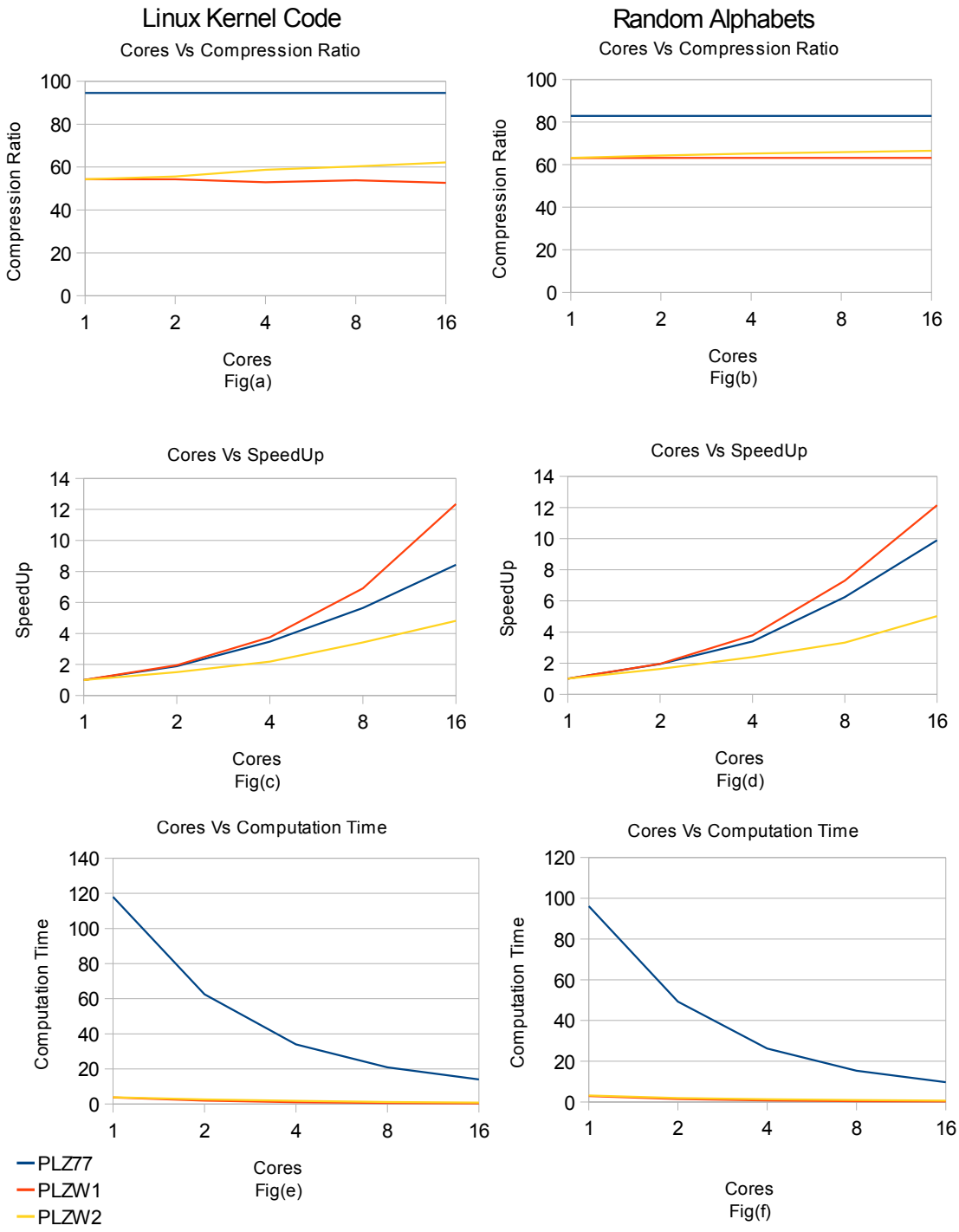

Figure 3: These are all the results of our experiements. Fig (a),(c) and (e) denotes the comparison of PLZZ, PLZW1, PLZW2 over Linux Kernel code and Fig (b), (d) and (f) denotes the comparison of PLZZ, PLZW1, PLZW2 over random generated alphabets of size 16. Computation time is in seconds and Compression Ratio is in percentage.

5 Analysis

From the experiment result, we can see:

- Efficiency: the performance of PLZW1 and PLZW2 is just outrun the PLZ77 algorithm. They are 10∼25 times faster. This is not especially surprise, for similar reason, people seldom use sequential LZ77 in practice. In PLZW1 and PLZW2, the most cost operation is hashing, however in PLZ77, we have multiple stages, and each stage evolve non-trivial and much complex algorithms, especially for the suffix array. Table 9 shows how much time it takes to finish different part of the algorithm.

Cores	Suffix Array	ANSV & RMQ	Point Jump	Prefix Sum
1	103	16.2	0.693	0.187
2	54.5	9.09	0.38	0.0973
4	30.4	4.78	0.197	0.0547
8	18.2	2.5	0.105	0.0399
16	11.5	1.59	0.0632	0.0219

Table 9: Timing for different parts of PLZ77 algorithm run on 64MB linux code.

- Scalability:
  - As we see in the result, PLZW1 has the best scalability, because all the chunks are proceed data independently, yet we cannot get perfect speedup, because the workload is not perfectly balanced, although the trunk size is the size, the compression rate for each trunk is not exactly the same, and the time we query or update for the hash table is not the same.
  - PLZ77 also has reasonable scalability, since most of the time PLZ77 is just computing the suffix array, and most of work of computing suffix array is sorting, our scalability is limited to the parallel sorting algorithm we implemented, Table 10 shows the different performance we got for 3 different algorithms. We finally use Parallel Sorting by regular sampling because we yield similar speedup with paper [19]. However we think there is still a lot of space to improve the performance by using more advance sorting algorithms.
  - PLZW2 don't has worst scalability, although it has better compression ratio than PLZW1. As described before, the workload is not well balanced, e.g. 1st processor will take $\log p$ chunk but last processor will only take 1.

Cores	Radix sort	Merge sort	PSRS
1	**88.7**	120	118.0
2	**48.2**	64.2	62.5
4	**27.3**	35.5	34.0
8	21.0	20.9	**20.9**
16	16.0	14.3	**14.0**

Table 10: The comparison of speed of PLZ77 using three different parallel sort algorithms. Time is in seconds. Although theoritically radix sort has linear work, the peformance is not ideal due to the bad support of nested parallelism in OpenMP, which is extensively used in the Radix sort code from [4].

  - For memory Consumption, PLZ77's memory usage is constantly $O(n)$ while PLZW1 and PLZW2 is $O(n + ph)$, where p is the number of processors and h is the hash table

size, this may not be the issue when processor number is low, but it might become a problem when we have hundreds of processors and the total memory space is limited.

- Compression ratio: as described, PLZ77 algorithm will always have the optimal compression result while PLZW1 and PLZW2 will not. In the linux kernel code test, the size of compression result of PLZ77 is about 10 times smaller than PLZW1 and PLZW2. In random text, PLZ77 is about 2 times smaller than them.

  The compression ratio of PLZW1 decreases for more cores, and the compression ratio of PLZW2 will increases. This is what we expected, the reason as we explained in Section 2, is PLZW2 could share the dictionaries so we have less redundant entries overall.

- Machine: The results got from checkpoint showed a drop in the performance of our program on 16 cores. In the process of optimization we removed many memory intensive operations which enabled us to scale to 16 cores. As reduction in memory usage increased our performance we think our working set became small so that the overall caching effort for the system become lower. When we tried to scale our program to 32 cores by reducing further memory usages but we were not able to scale it. We think this is due to the architecture of Blacklight. Since Blacklight has only 16 cores per blade, and it is CC-NUMA. We suspect that the cache performance is bottle-necked by the speed of PCI bus between the blades.

6 Conclusion and Future work

In this project, we developed and implemented the Parallel LZ77 algorithm with unlimited window size and linear work, which we believe is the first implementation. We also implemented two version of Parallel LZW algorithm, and we do cross comparison of the three algorithms. The implementation are all non-trivial, and we would likely to make our source available for public.

Based on the experiments, we suggest that on random text or binary data, and if compression time is priority, PLZW1 should be considered first. If we are compressing English text or the data has huge amount of repetitive occurrences, and compression size is priority, we can choose PLZ77. If the case is between the above two, it might depend on the programmers os preference and actual tests.

Future work might be improving the parallel sorting algorithm we used so that it can scale better. Also, it would be very interesting to compare with even more parallel compression algorithms such as BWT [10] which is also a very commonly used one. We would want to see how does our algorithms scale on even more than 16 cores by using more proper machine.

7 Distribution of Work

Fuyao implemented the Parallel LZ77 algorithm and Nagarathnam implemented the Parallel LZW algorithm. We did experiments and writeup together. We thank Julian Shun for providing PBBS [4] code and early discussion.

References

[1] Hash functions. `http://www.partow.net/programming/hashfunctions/#APHashFunction`.

[2] Intel Cilk Plus. `http://software.intel.com/en-us/articles/intel-cilk-plus/`.

[3] OpenMP. `http://openmp.org/wp/`.

[4] Problem based benchmark suite. `http://www.cs.cmu.edu/afs/cs.cmu.edu/Web/People/aladdin/pbbs/new/benchmarks.html`.

[5] Abouelhoda, M. I., Kurtz, S., and Ohlebusch, E. Replacing suffix trees with enhanced suffix arrays. *J. of Discrete Algorithms 2* (March 2004), 53–86.

[6] Anderson, R. J., and Miller, G. L. A simple randomized parallel algorithm for list-ranking. *Inf. Process. Lett. 33*, 5 (Jan. 1990), 269–273.

[7] Berkman, O., Schieber, B., and Vishkin, U. Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. *J. Algorithms 14* (May 1993), 344–370.

[8] Berkman, O., and Vishkin, U. Recursive star-tree parallel data structure. *SIAM J. Comput. 22* (April 1993), 221–242.

[9] Blelloch, G. E., and Maggs, B. M. Parallel algorithms.

[10] Burrows, M., Wheeler, D. J., Burrows, M., and Wheeler, D. J. A block-sorting lossless data compression algorithm. Tech. rep., 1994.

[11] Chen, G., Puglisi, S. J., and Smyth, W. F. Fast and practical algorithms for computing all the runs in a string. In *Proceedings of the 18th annual symposium on Combinatorial Pattern Matching* (Berlin, Heidelberg, 2007), Springer-Verlag, pp. 307–315.

[12] Crochemore, M., and Ilie, L. Computing longest previous factor in linear time and applications. *Inf. Process. Lett. 106* (April 2008), 75–80.

[13] Crochemore, M., Ilie, L., and Smyth, W. F. A simple algorithm for computing the lempel ziv factorization. In *Proceedings of the Data Compression Conference* (Washington, DC, USA, 2008), IEEE Computer Society, pp. 482–488.

[14] Crochemore, M., and Rytter, W. Efficient parallel algorithms to test square-freeness and factorize strings. *Information Processing Letters 38*, 2 (1991), 57 – 60.

[15] De Agostino, S. Lempel-ziv data compression on parallel and distributed systems. In *Data Compression, Communications and Processing (CCP), 2011 First International Conference on* (june 2011), pp. 193 –202.

[16] He, X., and Huang, C.-H. Communication efficient bsp algorithm for all nearest smaller values problem. *J. Parallel Distrib. Comput. 61* (October 2001), 1425–1438.

[17] Kärkkäinen, J., Sanders, P., and Burkhardt, S. Linear work suffix array construction. *J. ACM 53* (November 2006), 918–936.

[18] Klein, S. T., and Wiseman, Y. Parallel lempel ziv coding. *Discrete Appl. Math. 146*, 2 (Mar. 2005), 180–191.

[19] Kulla, F., and Sanders, P. Scalable parallel suffix array construction. *Parallel Comput. 33* (September 2007), 605–612.

[20] Lin, M.-B. A hardware architecture for the lzw compression and decompression algorithms based on parallel dictionaries. *J. VLSI Signal Process. Syst. 26*, 3 (Nov. 2000), 369–381.

[21] Manber, U., and Myers, G. Suffix arrays: a new method for on-line string searches. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms* (Philadelphia, PA, USA, 1990), SODA '90, Society for Industrial and Applied Mathematics, pp. 319–327.

[22] Shi, H., and Schaeffer, J. Parallel sorting by regular sampling. *J. Parallel Distrib. Comput. 14*, 4 (Apr. 1992), 361–372.

[23] Storer, J. A., and Szymanski, T. G. Data compression via textual substitution. *J. ACM 29*, 4 (Oct. 1982), 928–951.

[24] Welch, T. A technique for high-performance data compression. *Computer 17*, 6 (june 1984), 8 –19.

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

[26] Ziv, J., and Lempel, A. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory 24*, 5 (1978), 530–536.