PARALLEL PROGRAMMING PROJECT

Task Parallel Lossless Data Compression Based on the Burrows-Wheeler Transform

Andrea Longobardi
Umberto Picariello
??/02/2019

Summary

- Introduction
- BWTZip
- Parallelizing BWTZip
- Experiments and results
- Time issues
- Conclusions

Introduction

- The Burrows-Wheeler transform is a block-sorting lossless data compression algorithm, that provides compression ratios within a few percent of the best lossless statistical algorithms while using much less computational resources.

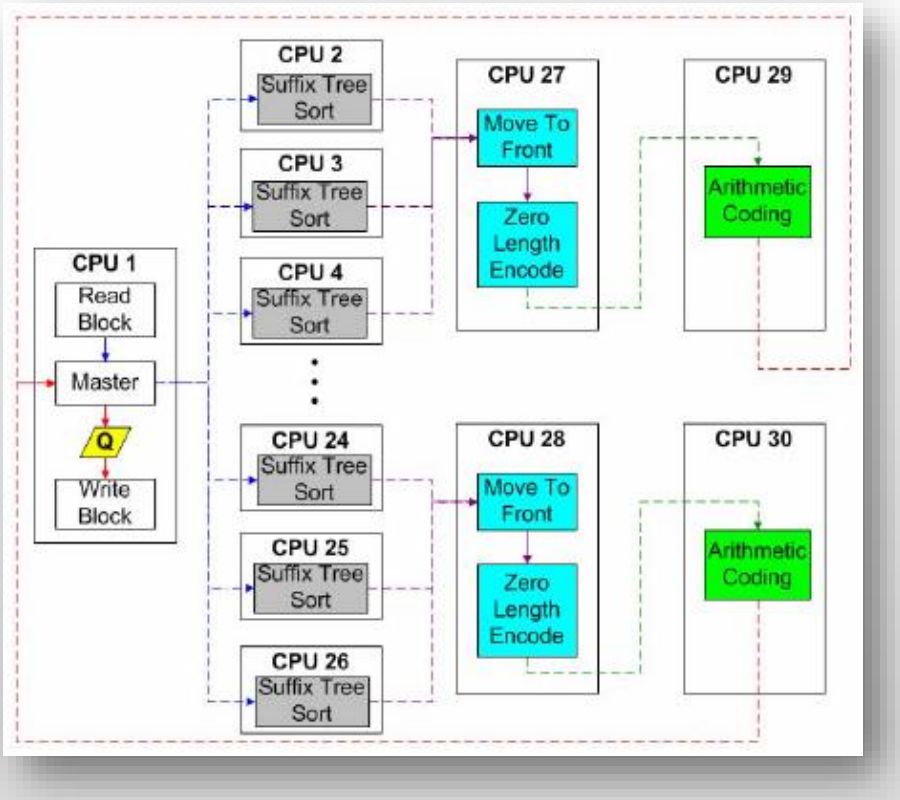
- The BWT algorithm processes blocks of data in a sequence of three stages:
  1. *The first stage sorts the input data lexicographically so that similar contexts are grouped together.*
  2. *Second, the move-to-front stage converts local symbol groups into a global structure.*
  3. *Finally, the compression stage uses an order-0 arithmetic coder to take advantage of the skewed transformed data producing effective smaller output sizes.*

BWTZip

- BWTZip is a lossless data compressor that uses the BWT compression algorithm.
- It is a highly modular opensource implementation of the BWT algorithm designed to be used for research purposes.
- The sequential version of BWTZip splits the input data into multiple blocks of user defined size. Only one block of data is processed by BWTZip at a time.
- The BWT portion of the BWTZip software can be split into three distinct tasks.
    1. *The first is the sorting stage of the BWT.*
    2. *The second task implements the move to front (MTF) stage of the BWT and (in sequence) a zero length encoder (ZLE) that compresses the repeated sequence of bytes containing 0 which commonly occur after the sorting and MTF stages of BWT.*
    3. *The fourth, is an adaptive arithmetic encoder task implements the compression stage of the BWT.*
- The modular design of BWTZip software allowed us access to the various stages of the BWT which required for designing the software pipeline.

Parallelizing BWTZip (1)

- We were inspired by the task parallel message-passing version of BWTZip called “mpibwtzip”: a three-stage pipeline designed to work on cluster machines (figure on the right).
- In our parallel implementation of “mpibwtzip”, we replace the cluster machines’s CPUs with threads.
- The threads were grouped into a three stage pipeline for the three stages of the BWTZip algorithm.
- To enable parallelism in our project, we’ve used the OpenMP API. In this way we could reuse the C++ implementation of the BWTZip compressor stages.



Parallelizing BWTZip (2)

- Our main loop code(on the right), represent the pipeline implementation using OpenMP pragma directives.
- Each of the three tasks can have one or more threads executing that stage, each one processing one or more blocks.
- Thanks to the software pipeline approach, the stages can be overlapped.
- Once a stage in the pipeline completes a block, it passes the same on to the succeeding stage and can start processing the next block immediately.
- To avoid synchronization problems between threads of adjacent stages, we've used the double-buffering technique on a vector.
- The whole structure, should allow the algorithm to achieve speedup with multiple core systems.

```
113  #pragma omp parallel num threads(NUM STAGES PIPELINE)
114      {
115          while (i < cycles + NUM STAGES PIPELINE - 1)

116          {
117  #pragma omp sections
118              {
119  #pragma omp section
120                  {
121                      if (i < cycles) {
122                          firstStage(&i);
123                      }
124                  }
125  #pragma omp section
126                  {
127                      int m = i - 1;
128                      if ( m >= 0 && m < cycles) {
129                          secondStage(&m);
130                      }
131                  }
132  #pragma omp section
133                  {
134                      int n = i - 2;
135                      if ( n >= 0 && n < cycles) {
136                          thirdStage(&n);
137                      }
138                  }
139              }
140  #pragma omp single
141              {
142                  i++;
143                  printf("ROUND %d, thread %d\n", i, omp get thread num());
144              }
145          }
146      }
```

Parallelizing BWTZip (3)

- Our parallel implementation is easily tunable thanks to some key constants contained in Utils.h .
- We can set (in order):
  1. *Number of bytes per block*
  2. *Number of blocks that each stage can work on (lower than the number of blocks of the input file)*
  3. *Number of threads assigned to the first, second and third task*

```
5   #define BLOCK_SIZE 900000
6
7
8   #define NUM_BLOCKS_PER_STAGE 4
9   #define NUM_THREADS_FIRST 4
10  #define NUM_THREADS_SECOND 1
11  #define NUM_THREADS_THIRD 1
```

Time issues (1)

- Trying the executable of the original BWTZip algorithm (the sequential one and providing the same block size), our first implementation of the parallel pipeline program was 2 order of magnitude more time consuming than the executable sequential version provided by the library.
- Some attempts was conducted to minimize the read write operation on vectors
  - *Using directly references to our blocks avoiding useless copies of blocks*
  - *Match the data types of the libriaries (unsigned char) to avoid data type conversions.*
- Four versions of the program, with all the possible combinations of the two previous said variations, where implemented:
  1. *vector<string>*
  2. *vector<unsigned char>*
  3. *vector<unsigned char>**
  4. *vector<string>**

Time issues (2)

- The best performance are reached by using all references and directly specifying the datatype provided by the library
- The following times are an average of the times when the given version of the program is executed (input file is the same as before and configuration for the parallel pipeline is the one with the highest speedup so 4.1.1 with 1KB blocks):

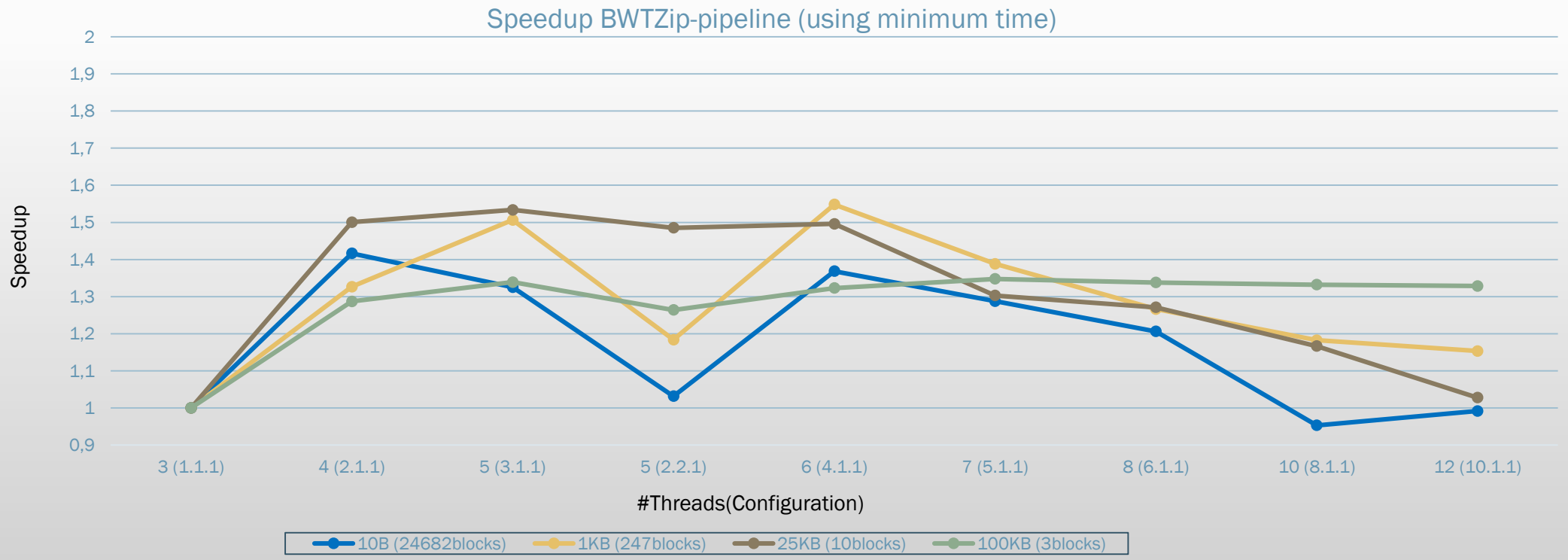
1.	*vector<string>*	*11,144060 s*
2.	*vector<unsigned char>*	*12,034369 s*
3.	*vector<unsigned char>**	*11,039750 s*
4.	*Vector<string>**	*11,067657 s*
5.	*Executable(MinGW compiled)*	*0,1465000 s*
6.	*ExeCode compiled with MSVC*	*18,611800 s*

- Times of our programs (1 to 4) are still two order of magnitude (10^2) far from the executable sequential program.
- Fortunately, the C++ code of the executable version was available. An «__asm__()» statement was used for console printing inside the original C++ code of the executable. This statement is compatible only with GCC compilers (for example MinGW that we found out to be the compiler of our executable).
- Summarizing:
  - *(3) Our fastest version, compiled in Microsoft Visual C++ (MSVC compiler).* *11,04 s*
  - *(5) The original executable code of BWTZip, built with MinGW.* *0,147 s*
  - *(6) The same source code of the executable, built in Visual Studio C++ (MSVC compiler).* *18,61 s*
- Is pretty clear now that MSVC compiles without taking any optimization and this reflects to a really bad performing final program.

Experiments and results (1)

	Time (sec)			
Threads(configuration)	10B (24682blocks)	1KB (247blocks)	25KB (10blocks)	100KB (3blocks)
3 (1.1.1)	54,688327	17,063274	11,38697	10,811172
4 (2.1.1)	38,608456	12,861963	7,589392	8,400402
5 (3.1.1)	41,264060	11,327858	7,425646	8,075056
5 (2.2.1)	53,009235	14,413693	7,666125	8,552614
6 (4.1.1)	39,968464	11,021188	7,611811	8,172206
7 (5.1.1)	42,473006	12,291413	8,740597	8,022794
8 (6.1.1)	45,337805	13,476942	8,959567	8,080080
10 (8.1.1)	57,39278	14,429584	9,760057	8,1155045
12 (10.1.1)	55,156884	14,794081	11,081365	8,137064

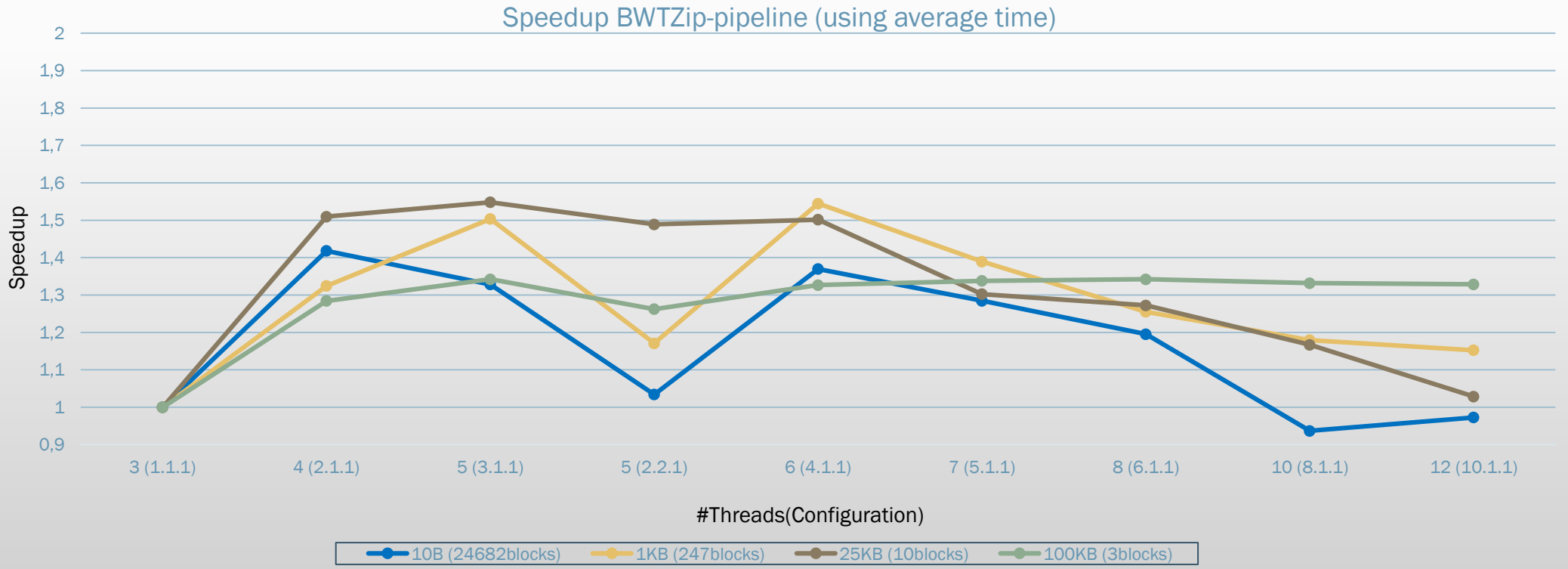
	Speedup			
Threads(configuration)	10B (24682blocks)	1KB (247blocks)	25KB (10blocks)	100KB (3blocks)
3 (1.1.1)	1	1	1	1
4 (2.1.1)	1,41648573	1,326646174	1,500379741	1,286982694
5 (3.1.1)	1,325325889	1,506310725	1,533465237	1,338835545
5 (2.2.1)	1,031675462	1,183823882	1,485361901	1,264078094
6 (4.1.1)	1,368286932	1,548224565	1,495960685	1,322919662
7 (5.1.1)	1,287601989	1,388227212	1,302767992	1,347556973
8 (6.1.1)	1,206241171	1,266108736	1,270928606	1,338003089
10 (8.1.1)	0,952878167	1,182520161	1,166690932	1,332162652
12 (10.1.1)	0,991505013	1,153385195	1,027578281	1,32863303



Experiments and results (2)

	Time (sec)			
Threads(configuration)	10B (24682blocks)	1KB (247blocks)	25KB (10blocks)	100KB (3blocks)
3 (1.1.1)	55,0499446	17,1673518	11,537316	10,882055
4 (2.1.1)	38,8124156	12,9623846	7,6423234	8,4725514
5 (3.1.1)	41,4410796	11,416591	7,4527438	8,107771
5 (2.2.1)	53,2268678	14,6624918	7,7484568	8,6198138
6 (4.1.1)	40,1885938	11,1140904	7,6837216	8,2022244
7 (5.1.1)	42,8457566	12,3540384	8,8575814	8,1333636
8 (6.1.1)	46,0482868	13,6773304	9,0679082	8,1081004
10 (8.1.1)	58,761388	14,5543578	9,8855818	8,1700685
12 (10.1.1)	56,6006836	14,8968296	11,215108	8,1895496

	Speedup			
Threads(configuration)	10B (24682blocks)	1KB (247blocks)	25KB (10blocks)	100KB (3blocks)
3 (1.1.1)	1	1	1	1
4 (2.1.1)	1,418359145	1,324397658	1,509660792	1,284389328
5 (3.1.1)	1,328390697	1,503719613	1,548062876	1,342175871
5 (2.2.1)	1,034251063	1,170834537	1,488982426	1,262446597
6 (4.1.1)	1,369790266	1,544647486	1,501527073	1,326719932
7 (5.1.1)	1,284840063	1,389614573	1,302535701	1,337952554
8 (6.1.1)	1,195483012	1,255168319	1,272323864	1,342121343
10 (8.1.1)	0,936838738	1,179533445	1,167085179	1,33194166
12 (10.1.1)	0,972602115	1,152416471	1,028729817	1,32877327



Conclusions

- High speedup (in our case above 1.5) is obtained exploiting the hyperthreading that, in our case (i.e. with an Intel i7 CPU) is efficient with 6 threads, prioritizing always the allocation of threads on the first stage in a similar way they did in the paper (84% of threads to the first stage, 8% to the second and the remaining 8% to the second).
- The curve is inline with our available hardware: all of our parallelism is lost after 6 threads. Obviously, using a CPU with more cores would lead us to even better performances.
- As far as it concerns the block size, in generale there is no precise rule to decide it but it's obvious that too small blocks will slow down the execution since most of the time is passed to «pass» blocks between the buffers. With our particular input file, instead, too large blocks will shrink dramatically the number of blocks to be processed and this makes the parallelization less efficient.
- Compiling in MSVC does not ensure to get a very good performant executable so take care of compiler optimization is important when time is an important requirement.