ISSN Electrónico: 2215-3470
DOI: http://dx.doi.org/10.15359/ru.31-2.3

UNICIENCIA Vol. 31, No. 2, pp. 37-49. Julio-diciembre, 2017.
URL: www.revistas.una.ac.cr/uniciencia
Email: revistauniciencia@una.cr

Compresión BZIP2 optimizada usando colas libres de bloqueo

Enhanced Parallel bzip2 Compression with Lock-Free Queue

José Sánchez-Salazar
jose.sanchez.salazar@una.cr
Escuela de Informática, Universidad Nacional
Heredia, Costa Rica

Edward Aymerich-Sánchez
edward.aymerich@gmail.com
Escuela de Ciencias de la Computación e Informática,
Universidad de Costa Rica
San José, Costa Rica

Recibido-Received*: **25/abr/2016*** / Corregido-Corrected*: **7/ago /2016.***
Aceptado-Accepted*: **11/ago/2016*** / Publicado-Published*: **31/jul /2017.***

Resumen
Debido a que la tendencia actual es tener más y más procesadores (cores) disponibles en cada computadora, la escalabilidad de las estructuras de datos usadas en programación paralela debe ser considerada cuidadosamente, para así garantizar que ellas saquen ventaja de los procesadores disponibles. Debido al aumento en la contención, usualmente las estructuras de datos basadas en bloqueos no mejoran su rendimiento proporcionalmente al incrementar el número de procesadores. El uso de estructuras de datos libres de bloqueos bien diseñadas, tales como las colas *first in-first out*, puede mejorar el rendimiento de un programa paralelo, cuando hay varios procesadores disponibles. En este trabajo se diseña e implementa una versión paralela de *bzip2*, un programa para compresión y descompresión de datos muy popular, usando colas libres de bloqueos en lugar de las basadas en bloqueos, y aplicando una estrategia de dos buffers de salida. Se compara el rendimiento de la implementación libre de bloqueos contra implementaciones basadas en bloqueos. Se midió el tiempo de compresión usando diferente número de procesadores y diferentes tamaños de bloques. Coincidiendo con la hipótesis de trabajo, los resultados muestran que la implementación paralela libre de bloqueos supera las otras implementaciones.

Palabras claves: Programación informática; Procesamiento de datos; Lenguaje de programación.

Abstract
Since the general trend nowadays is to have more and more processors (cores) available in each computer, the scalability of the data structures used in parallel programs must be carefully considered in order to guarantee that they take full advantage of the available processors. Because of increased containment, lock-based data structures usually do not perform proportionally as the number of processors increases. The use of well-designed lock-free data structures, like *first in-first out*, (*fifo)* queues, can boost the performance of a parallel program when many processors are available. In this work, a parallel version of *bzip2*, a popular compression and decompression program, is designed

ISSN Electrónico: 2215-3470
DOI: http://dx.doi.org/10.15359/ru.31-2.3

and implemented by using lock-free queues instead of the lock-based ones, and applying a two-buffer-output strategy. The performance of lock-free implementation is measured against lock-based implementations. Compression time was measured with different number of processors and different block sizes. Consistent with our hypothesis, the results show that our parallel lock-free implementation outperforms the other implementations.

Keywords: Computer programming; Data processing; Programming languages

As is well known, the popularization of parallel computers, like multi-core systems, has posed the challenge to parallelize programs, or data structures supporting them to take advantage of that processing power. It is also known that lock-based data structures could severally limit the amount of parallelism in a program. Lock-free programming techniques (Zhang & Dechev, 2016; Marcais & Kingsford, 2011) are known to deliver increased parallelism and scalability in scenarios of high contention and sharing of data among multiple threads. It is also agreed that this is a challenging area with a lot of active research going on.

Data compression applications are processor-intensive; therefore, they are interesting examples where the benefit from parallelization could be significant. Data compressors and decompressors are important not only by themselves, but also when used as part of bigger processes to do compression or decompression on-the-fly. The program *bzip2* (Seward, 2010) is a very popular tool used for data compression and decompression on multiple platforms like Unix, Linux and Windows.

The *bzip2* program is based on the Burrows-Wheeler Transform (Burrows & Wheeler, 1994) algorithm for data compression. The BWT algorithm works over blocks of text instead of working over an entire file. It does not process the input file sequentially in a character by character basis, but instead, processes each block of text as a single unit. BWT applies a reversible transformation to a block of text to form a new block that contains the same characters, but it is easier to process by simple compression algorithms, like Huffman (Huffman, 1952). Basically, this transformation generates all the rotations of the characters in the original sequence (block) *S*, and then sort all that generated sequences. An important observation here is that the sequence *T* (for Tails), formed by taking the last character of each sequence in the sorted collection, clearly forms a permutation of the original sequence. By keeping that *T* sequence and the position *I* of the original sequence in the sorted collection, one can eventually reproduce the original sequence. The indicated transformation is illustrated in Figure 1.

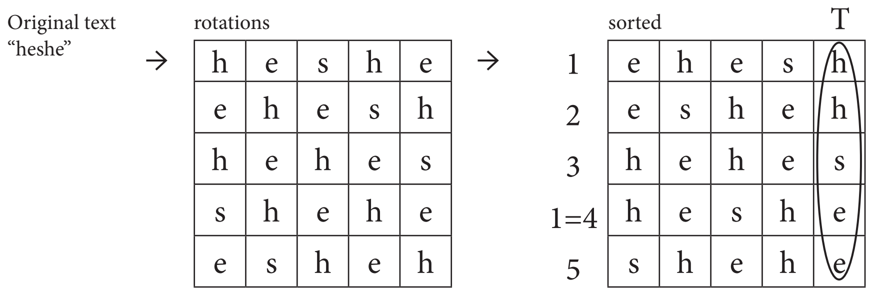


Figure 1. BWT Compressing Transformation Example. *Source:* this study.

The described sorting and tailing operations tend to put together (or close to each other) the occurrences of each character in the generated sequence *T* (like the two *h* and the two *e* in the shown example). Thus, *T* will be easy to compress with a simple locally-adaptive compression algorithm, like Huffman (Huffman, 1952) or arithmetic (Neal, Witten & Cleary, 1987) compression.

The serial implementation of *bzip2* works by splitting the original file into blocks; then, it compresses each block independently and sequentially, and writes the compressed blocks in the destination file. The program *bzip2* is based on the *libbzip2* library that provides API functions for the required tasks. In particular, there is a function that allows compressing an in-memory data block. It runs the compression code within the calling thread and leaves the compressed block in memory.

The *bzip2* serial implementation of course does not take advantage of parallel hardware like multi-core and many-core computers. It just runs a thread that continuously reads a block, compresses it, and writes the compressed block. Since the compression part is the most time consuming stage of the process, there is a clear opportunity to improve the performance by parallelizing this stage.

There already exist some parallel implementations of *bzip2* (Isakov, 2005; Jannesari, Pankratius & Tichy, 1990; Gilchrist, 2004). Basically, they fall in two categories. Some of them do lock-based fine-grained parallelism of certain operations involved into the BWT algorithm, like rotating or sorting. The data dependencies and the contention due to locks prevent them from scaling well when using hardware with many cores. The other category includes approaches that do coarse-grained parallelization, for example, parallelizing the compression of entire blocks. They use some sort of lock-based queues as buffers between the reading, compressing, and writing stages. They suffer from the contention problem inherent to lock-based designs, and consequently do not scale well when the number of processors grows.

In this work, we address the problem of building an efficient parallel implementation of the *bzip2* data compressor using a lock-free implementation of the queue data structure. The idea is to have a reader thread, many compressor threads running in parallel, and a writer thread. The reader thread reads blocks from the input file and store them into an input buffer. The compressors use the API provided in-memory-compressing function to compress the blocks and then put them in an output buffer. The writer thread takes compressed blocks and writes them to the output file, in the correct sequential order. Both, the input and output buffers, are implemented as lock-free queues, to improve the parallel processing, avoiding the contention problem of lock-based data structures. We tag the read blocks with their sequential numbers. Then, it compresses these blocks in parallel, in an out-of-order fashion, and finally reorders them before writing to the output file. This is similar to how pipelined out-of-order execution occurs in modern hardware architectures. We also use a two-buffer-output strategy for reordering the blocks before writing them to the output file so that the compression stage proceeds completely in parallel. Our approach is based on the observation that the reading and writing stages are inherently serial and the compression part is the most time-consuming stage. As the number of compressor threads increases, the throughput of the compression phase grows and so the stress is shifted to the queues (buffers) causing contention on them. The use of a lock-free queue implementation is worthwhile here as it can alleviate the contention and improve the performance stage. The complete approach is detailed in the "Parallelizing bzip2" section.

The rest of this paper is organized as follows: the next section presents the related work, the "Parallelizing bzip2" section describes our approach, the "Experimental design" section describes the setup of our experiments, the obtained results and how our approach compares to previous approaches is presented in the "Experimental results" section, finally some conclusions and future work are presented.

Related work

Most algorithms for data compression are designed for sequential processing. Both dictionary based approaches, like Zip family (Ziv & Lempel, 1978; Welch, 1984) and statistical approaches, like arithmetic (Neal, Witten & Cleary, 1987) and Huffman (Huffman, 1952) encodings, are essentially sequential. Even the current implementation of the popular *bzip2* algorithm runs sequentially.

Isakov (Isakov, 2005) implemented a parallel version of *bzip2* for symmetric multiprocessors called BZIP2SMP. Basically, this implementation does a fine-grained parallelization of some of the processes involved in the *bzip2* algorithm, like RLE (run-length-encoding), sorting and bit-storing. This approach is said to have good speedup, but that has not been systematically documented. Besides, it does not take advantage of any lock-free data structure as we do in our work.

An experiment was conducted and documented by Jannesari et al. (Jannesari, Pankratius & Tichy, 1990) as part of a software engineering class. Student teams were assigned the task to parallelize *bzip2* in a team competition. Most of the teams tried out some forms of fine-grained parallelization however; many parts of the sequential code were not amenable for parallelization due to data dependencies, and tricky optimizations for faster sequential execution.

Gilchrist (Gilchrist, 2004) proposed and implemented a parallel version of *bzip2* named PBZIP. Basically, this implementation parallelizes the compression stage of the process compressing many blocks in parallel. He uses a blocking fifo queue as a buffer between the reading thread and the compressing threads, and between them two and the writing thread. This implementation, and others similar, work fine but use lock-based queues suffering from the contention problem inherent to them.

There are several proposals for implementing lock-free queues (Zhang & Dechev, 2016; Feldman & Dechev, 2015; Ladan & Shavit, 2004; Michael & Scott, 1996). In particular, the proposal of Michael & Scott was adopted in the lock-free queue implementation included in the BOOST library (Boost C++ Library, 2013).

In our work, we follow an approach similar to Gilchrist's, but we use Michael's lock-free queue implementation from BOOST library, instead of mutual-exclusion queues, as explained in following sections.

Parallelizing *bzip2*

The starting point is the serial implementation of *bzip2*. Then we use the Gilchrist's implementation *pbzip2* as a reference point for our parallel implementations. Both, the serial *bzip2* and the Gilchrist's implementations were explained before. For our implementations we adopted coarse-grained parallelization where multiple compressor threads compress in parallel independent blocks. We tried out various original approaches looking for better results: lock-based one-buffer-output, lock-based two-buffer-output, and lock-free implementation.

Lock-based one-buffer-output

Although blocks of data can be processed in parallel, they must be written in the same order they were read, to reconstruct the original data correctly. To guarantee this, we add a sequential number to each block at the moment of reading and we used a modified output queue that only enqueues a block if its sequential number is one more than the last block accepted. Using this mechanism, the writer thread only receives ordered blocks when it dequeues and its work is really simplified: just pick up a block and write it to file.

The downside to this approach is that it forces the compressor threads to wait for the moment when they can enqueue a block. For example, if the compressor that handles blocks 7 finishes, it must wait until block 6 is enqueued before enqueuing block 7 because the special queue does not allow block 7 to get in.

To measure the impact of using this special queue, we compare its performance against an implementation which does not consider the output ordering of the blocks; let's call it non-ordered implementation. The non-ordered implementation will produce corrupt compressed files and therefore, it does not have practical applications, but it works as a baseline to compare the performance of the ordered implementation.

We compare the performance of these two approaches and find that the ordered implementation is 1.3X slower than the non-ordered implementation. The cost of having compressors waiting in line to enqueue their blocks is clearly significant.

Lock-based two-buffer-output

Another approach to get correctly ordered blocks in the compressed file is to allow the compressors to enqueue their blocks in the output queue in any order, and let the writer thread worry about the ordering of the blocks. We call this approach two-buffer-output, because the writer uses a secondary buffer to store some blocks.

When the writer is ready to write a block to the file, it searches for the next sequential block inside its secondary buffer. If the block is found then, it is taken out of the secondary buffer and written to the file. If the block is not found in the secondary buffer, the writer takes a block from the output queue. If this block is the next sequential block then, it is written to the file. If not, the block is stored in the secondary buffer and another block is taken from the output buffer. This process is repeated until the next sequential block is found and written to the file.

Figure 2 exemplifies the two-output-buffer approaches: (a) the writer (W) needs to write block 7 to file. (b) Since the block is not in the secondary buffer, it takes blocks from the output queue and puts them in the secondary buffer. (c) Block 7 is found in output queue, and is written to file. (d) Now the writer must write block 8, which is already in the secondary buffer.

The two-output-buffer approach has several advantages. First, the secondary buffer is only used by the writer thread so it does not need to be protected from other threads (no locking or multi-threading worries). Second, this mechanism keeps the output queue as empty as possible so compressor threads do not get stalled because of a full queue. Finally, the compressor threads can enqueue in any order, so they do not have to wait. These advantages are reflected in the performance: the two-output-buffer implementation performs just as good as the non-ordered implementation

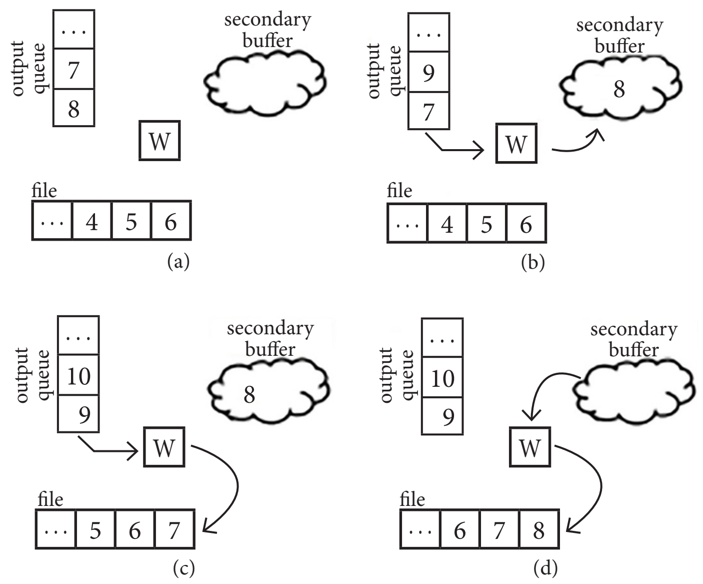


Figure 2. Two Buffers Approach. *Source:* this study.

Lock-free implementation

For our parallel *bzip2* implementation we replace the lock-based queue with a lock-free queue and use it for both, the reading buffer (between reader and compressors) and the writing buffer (between the compressors and the writer). We keep our two-output-buffer strategy, but now combined with a lock-free queue.

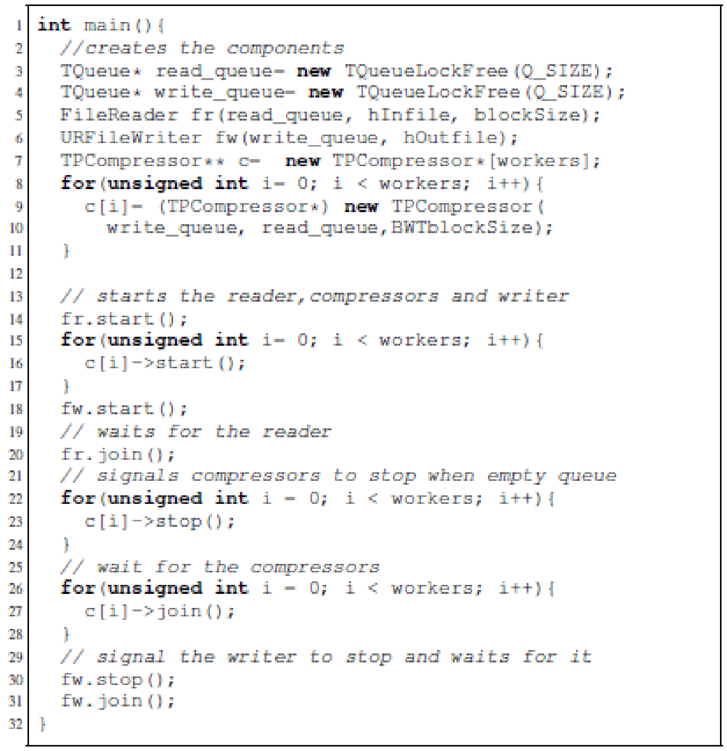
There are several designs of lock-free queues. In this work we used the lock-free queue implementation included in the BOOST library (Boost C++ Library, 2013) which is based on the design of Michael et al. (Michael & Scott, 1996). This queue is implemented as a singly-linked list with Head and Tail pointers where Head always points to a dummy node at the beginning of the list, and Tail points to either the last or second to last node in the list. The algorithm uses compare-and-swap with modification counters to avoid the ABA problem. We selected this queue because it is based in a well-documented design and has a well-engineered implementation (BOOST library).

As Figure 3 shows, the main program uses the following components: two lock-free queues (read_queue and write_queue), one reader, and one writer threads (fr, fw), and n compressor threads (c).

The algorithm proceeds as follows: at the starting phase, it creates the components and starts the threads (reader, compressors and writer). During the working phase all the threads work in parallel, mediated by the queues. The finishing phase begins when the reader finishes. At this point, all the compressors are notified to stop as soon as the reading queue is empty. When all they finish, the writer thread is similarly notified to stop when the writing queue is empty. The algorithm finishes when the writer thread stops.

ISSN Electrónico: 2215-3470
DOI: http://dx.doi.org/10.15359/ru.31-2.3

UNICIENCIA Vol. 31, No. 2, pp. 37-49. Julio-diciembre, 2017.
URL: www.revistas.una.ac.cr/uniciencia
Email: revistauniciencia@una.cr



```
int main(){
  //creates the components
  TQueue* read_queue= new TQueueLockFree(Q_SIZE);
  TQueue* write_queue= new TQueueLockFree(Q_SIZE);
  FileReader fr(read_queue, hInfile, blockSize);
  URFileWriter fw(write_queue, hOutfile);
  TPCompressor** c=  new TPCompressor*[workers];
  for(unsigned int i= 0; i < workers; i++){
    c[i]= (TPCompressor*) new TPCompressor(
      write_queue, read_queue,BWTblockSize);
  }

  // starts the reader,compressors and writer
  fr.start();
  for(unsigned int i= 0; i < workers; i++){
    c[i]->start();
  }
  fw.start();
  // waits for the reader
  fr.join();
  // signals compressors to stop when empty queue
  for(unsigned int i = 0; i < workers; i++){
    c[i]->stop();
  }
  // wait for the compressors
  for(unsigned int i = 0; i < workers; i++){
    c[i]->join();
  }
  // signal the writer to stop and waits for it
  fw.stop();
  fw.join();
}
```

Figure 3. Compressing algorithm.   this study

Experimental design

We used a 64-core shared memory computer (Opteron 6272 x4) to compare the performance of the various versions of the parallel *bzip* algorithm. We designed experiments to compare the performance of the following implementations: the Gilchrist's (Gilchrist, 2004) lock-based implementation, our lock-based two-buffer-output and our lock-free implementation. Our lock-based one-buffer-output was not considered for the final experiment as it was significantly outperformed by our two-buffer-output implementation.

All the programs were compiled with gcc 4.6.3 compiler. For the experiments we compressed a 3.1GB data file using a block size of 900KB. Reducing the block size (from 900KB to 100KB in this case) marginally improves the performance of all the parallel approaches, but it negatively affects the compression ratio. Since the block size reduction has a similar effect on the different approaches and negatively affects the compression ratio, it was discarded as a parameter for comparison.

Each one of the compared implementations was run with the following number of processors: 1,2,4,8,16,24,32,40,48,56, and 64. All the experiments were measured as wall clock times in milliseconds. Then the time data for each implementation were used to obtain the speedup for each number of processors (compared to the corresponding one-processor (serial) configuration). These speedup data is presented in the graphs.

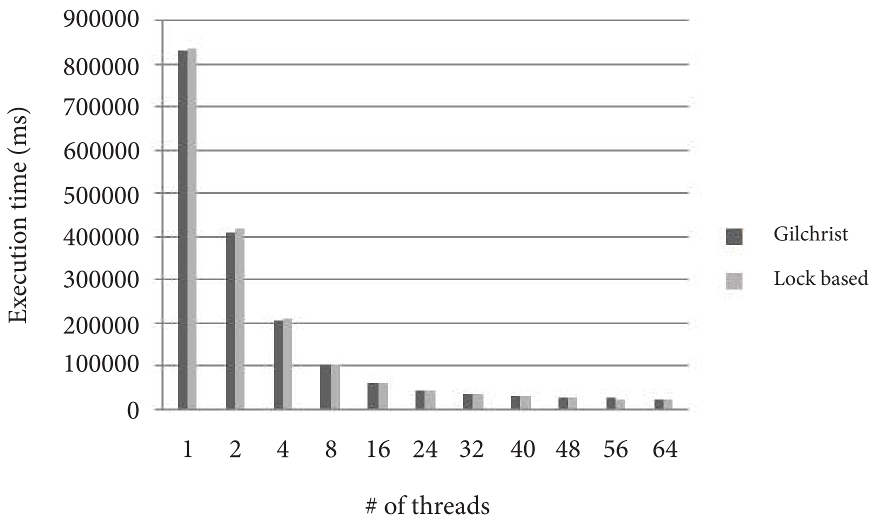
Experimental results

The following subsections summarize the results of the experiments in both, settings using the lock-based and the lock-free implementations.

Lock-based implementations

Our two-buffer-output lock-based implementation gets similar result to Gilchrist's. This was foreseeable as both implementations are very similar using lock-based queues. The clock-time decreases as the number of processor grows, as shown in Graph 1.

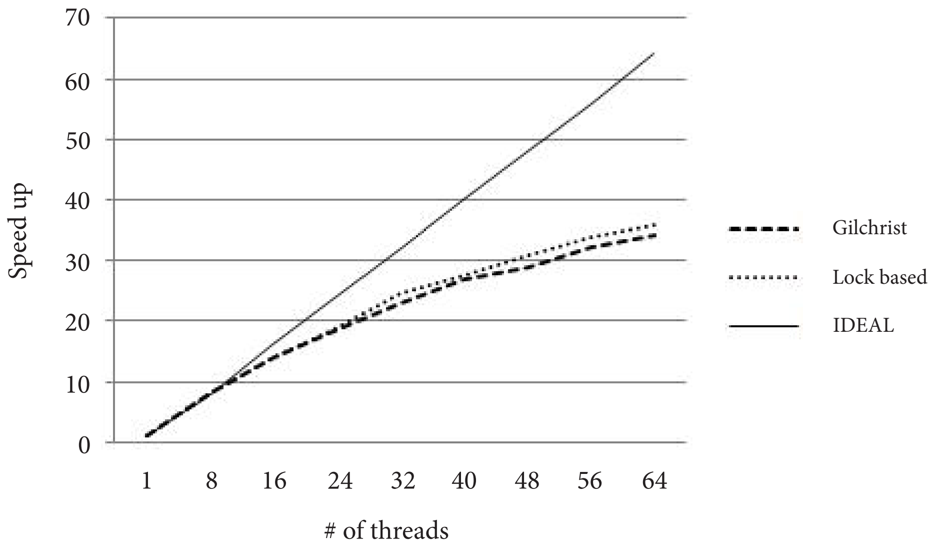
Graph 1
Lock Based Clock Time



Source: this study.

The speedup that ideally should be equal to the number of processor is shown in Graph 2.

Graph 2
Lock Based *SpeedUp*



Source: this study.

ISSN Electrónico: 2215-3470
DOI: http://dx.doi.org/10.15359/ru.31-2.3

UNICIENCIA Vol. 31, No. 2, pp. 37-49. Julio-diciembre, 2017.
URL: www.revistas.una.ac.cr/uniciencia
Email: revistauniciencia@una.cr

Both, our two-buffer-output implementation and Gilchrist's implementation, show a good speedup which is 99% of the ideal speedup using 8 processors and around 88% at 16 cores. After that their performance starts to decay.

Lock-free implementation

The lock-free implementation uses lock-free queues. Beyond that, it is similar to the lock-based implementations. All phases (reading, compressing, and writing) of the program run in parallel so both, I/O and compressing times are considered. Different block sizes were tried out (like 100k and 900k). This strategy scales very well up to around 16 processors. It performed at 99% of optimal speedup for 8 processors and at 90.44% of optimal speedup for 16 processors, as shown in Table 1.

After 16 processors the speedup starts to decay, achieving only 56.46% of optimal speedup when using 64 processors.

Table 1
Percent of ideal speedup achieved by different queue implementations

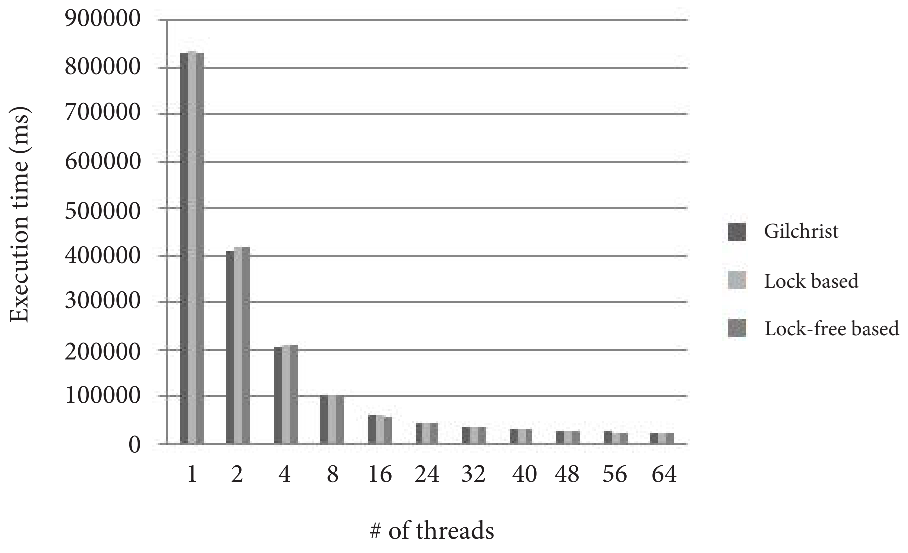
# of threads	Gilchrist	Lock based	Lock free
1	100.00%	100.00%	100.00%
2	101.52%	99.62%	99.81%
4	101.11%	99.59%	99.69%
8	100.76%	99.03%	99.08%
16	88.20%	86.98%	90.44%
24	77.61%	79.16%	79.17%
32	72.20%	76.87%	77.21%
40	67.24%	68.67%	69.83%
48	60.17%	64.63%	64.43%
56	57.23%	60.33%	60.47%
64	53.44%	56.26%	56.46%

Source: this study.

ISSN Electrónico: 2215-3470
DOI: http://dx.doi.org/10.15359/ru.31-2.3

The clock-time as a function of the number of processors is shown in Graph 3.

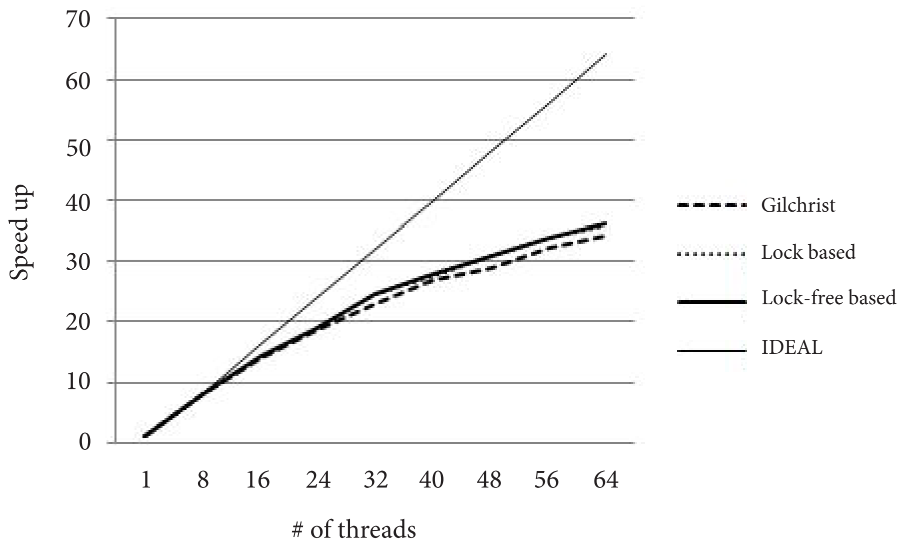
Graph 3
Lock Free Clock Time



Source: this study.

The speedup obtained by this lock-free implementation is shown in Graph 4.

Graph 4
Lock Free SpeedUp



Source: this study.

The decaying behavior, as the number of processors grows bigger, lead us to think about possible causes for this situation:

- The I/O could limit the speedup when we have enough compressing power (more than 16 processors)
- The queue implementation could suffer contention as the number of processor grows significantly, preventing the program to achieve better speedup.
- Cache and memory bus bandwidth could become a bottle-neck when the number of processors grows.

We prepared and ran other experiments trying to isolate and measure the effect of the above indicated factors to look for possible specific solutions.

I/O 'free' compression

The program was modified for doing all the reading phase before starting the compression phase and the writing phase was eliminated (no data is written to disk). Of course, this is a fake implementation, but it let us ponder the effect of I/O in the performance of the program.

We ran various tests with this implementation and basically obtained the same results. That showed us that the I/O was not the limiting factor.

Simulated queue implementation

To assess the effect of the possible contention on the queue implementation we created a simulated queue. We implemented it as an array big enough to store all the blocks of the entire input file. We read all the input blocks into it before compressing and use an atomic counter to keep track of the queue head.

This is a very simple queue implementation that avoids the more intricate lock-free logic of a thread-safe implementation. We ran various tests with this implementation and observed the same decaying behavior as the number of processors grows therefore; we discarded the queue contention for poor performance.

Memory system bottleneck

The BWT transformation that is at the core of the bzip algorithm needs to generate all the possible rotations of the input block. Then, it sorts them and takes the last character of the sorted rotations to form the transformed block before compressing it. This requires a support structure in memory, at least as big as the original block. Thus, the compression of a block requires the processor to read from memory the original block then, read and write to the support structure, and finally store the compressed block in memory. As the number of processors increases, the traffic on the memory system (caches, bus and memory) increases significantly, this could cause contention on it, preventing the program from getting better performance.

Conclusions and future work

Taking full advantage of parallel processing power of modern computers requires to pay careful attention to the locking mechanism used by programming data structures. Lock-free data structures are known to deliver increased parallelism and scalability in scenarios of high contention and sharing of data among multiple threads. We have implemented a parallel version of the *bzip2* data compression algorithm, using lock-free queues and applying a two-buffer-output strategy.

ISSN Electrónico: 2215-3470
DOI: http://dx.doi.org/10.15359/ru.31-2.3

The parallel version of the *bzip2* algorithm uses two queues of data blocks, we used lock-free queues to improve its performance. The algorithm consists of three tasks that run in parallel and use queues as buffers for intermediate data. The *reading* task reads blocks from an input file into a queue, multiple *compressing* tasks compress the blocks in parallel and put them into another queue, and finally, the *writing* task takes blocks from the queue and writes them into the output, compressed, file. We compared the performance of our lock-free queue based implementation against other lock-based implementations. The results show that our lock-free parallel implementation outperforms the other implementations, mainly in heavily-threaded scenarios.

Using a small-to-medium number of threads (in the range of 1 to 16), the speedup is very good (>90% of ideal speedup). With a higher number of processors, the performance starts to decay. After discarding other possible causes, our intuition is that the inherent intense memory usage of the *bzip* algorithm could cause contention on the memory system (caches, bus and memory), thus preventing better performance.

There are various extensions of this study there are suggested as **future work**. A more in-depth study of the memory behavior of the program would be required to assess the possible negative effect of the memory system on the performance of the program, when used with a large number of processors. Once the memory bottleneck situation has been addressed, another interesting suggested future work is to extend the experiments by compressing different types of data files, and to apply some non-parametric statistical hypothesis test (Wilcoxon, 1945) to validate the results. Another vein of future work would be to try to improve the fine-grained parallelization of the intra-block compressing algorithm. This, added to the use of lock-free queues could improve even more the overall performance of the compressing system.

References

Boost C++ Library (n.d.). (2013). Retrieved from https://svn.boost.org/trac/boost/

Burrows, M., & Wheeler, D. J. (1994). A block-sorting lossless data compression algorithm. (SRC Research Report 124). California: Digital systems research center. http://www.hpl.hp.com/techreports.

Feldman, S. & Dechev, D. (2015). A wait-free multi-producer multi-consumer ring buffer. *ACM SIGAPP Applied Computing Review*, 15(3), 59-71. http://dx.doi.org/10.1145/2835260.2835264

Gilchrist, J. (2004). Parallel data compression with bzip2. *Proceedings of the 16th IASTED International Conference on Parallel and Distributed Computing and Systems(PDCS)*. http://gilchrist.ca/jeff/comp5704/Final_Paper.pdf

Huffman, D.A. (1952). A method for the construction of minimum-redundancy codes. *Proceedings of the I.R.E.* http://dx.doi.org/10.1109/jrproc.1952.273898

Isakov, K. (2005). Bzip2smp. Retrieved from http://bzip2smp.sourceforge.net

Jannesari, A., Pankratius, V. & Tichy, W. (1990). Parallelizing bzip2-a case study in multicore software engineering. *IEEE Software*, 26(6). Recovered from https://www.semanticscholar.org

Ladan-mozes, E. and Shavit, N. (2004). An optimistic approach to lock-free fifo queues. *Proceedings of the 18th International Symposium on Distributed Computing,* Springer, Berlin, Germany. http://dx.doi.org/10.1007/978-3-540-30186-8_9

Marcais, G. & Kingsford, C. (2011). A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27, 764–770. http://dx.doi.org/10.1093/bioinformatics/btr011

Michael, M., & Scott, M. (1996). Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. http://dx.doi.org/10.1145/248052.248106

Neal, R., Witten, I. & Cleary, J. (1987). Arithmetic coding for data compression. *Communications of the ACM*, 30(6), 520-540. http://dx.doi.org/10.1145/214762.214771

Seward, J. (2010). *bzip2*. Retrieved from http://www.bzip.org/

Valois, D. (1994). Implementing lock-free queues. *Proceeding of the Seventh International Conference On Parallel and Distributed Computing Systems*, 1994.

Welch, T. (1984). A technique for high-performance data compression. *IEEE Computer*, 17(6). http://dx.doi.org/10.1109/MC.1984.1659158

Wilcoxon, F. (1945). *Individual comparisons by ranking methods. Biometrics Bulletin.* 1 (6). http://dx.doi.org/10.2307/3001968

Zhang, D. & Dechev, D. (2016). A Lock-Free Priority Queue Design Based on Multi-Dimensional Linked Lists. *IEEE Trans. Parallel Distrib. Syst*. 27(3) 613-626. http://dx.doi.org/10.1109/tpds.2015.2419651

Ziv, J., & Lempel, A. (1978). Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5). 530-536. http://dx.doi.org/10.1109/tit.1978.1055934