Universidade do Minho
Escola de Engenharia

Samuel Santos Almeida

Performance of Compressors in Scientific Data: A Comparative Study

January 31, 2014

Universidade do Minho
Escola de Engenharia
Departamento de Engenharia Informática

Samuel Santos Almeida

Performance of Compressors in Scientific Data: A Comparative Study

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de

Professor António Manuel Pina
PhD Manuel Melle-Franco

January 31, 2014

We build too many walls
and not enough bridges.

ISAAC NEWTON

Abstract

Computing resources have been increasingly growing over the last decade. This fact leads to the increasing amount of scientific data generated, reaching a I/O bottleneck and a storage problem. The solution of simply increasing the storage space is not viable, and the I/O throughput can not cope with the increasing number of execution cores on a system. The scientific community turns to the use of data compression, for both used storage space reduction, and alleviating the pressure on the I/O by making best use of the computational resources. We aim to do a comparative study of three distinct lossless compressors, using scientific data. Selecting gzip and LZ4, both general compressors, and FPC a floating-point specific compressor, we assess the performance achieved by the compressors and their respective parallel implementations. MAFISC is a adaptive filtering for scientific data compressor, and is briefly put to the test. We present a rather thorough comparison between the compressors parallel speedup and efficiency and the compression ratios. Using pigz parallel compression can yield speedup values in an average of 12 for 12 threads, achieving an efficiency close to one. gzip is the most complete compression algorithm, but LZ4 can replace it for faster compression and decompression, at the cost of compression ratio. FPC can achieve higher compression ratios and throughput for certain datafiles. MAFISC accomplishes what it proposes to, higher compression ratios, but at the cost of much increased compression time.

Resumo

Na última década tem-se vindo a assistir a um crescimento contínuo dos uso de recursos de computação. Em consequência tem também aumentado significativamente a quantidade de dados gerados em particular de dados científicos, que no final se traduz no estrangulamento da E/S de dados e num problema de armazenamento. O simples aumentar do espaço de armazenamento não é solução, nem é possível atingir taxas de transferência E/S capazes de lidar com o aumento do número de núcleos de execução, dos sistemas atuais. Assim, a comunidade científica vê-se obrigada a usar a compressão de dados, tanto para redução de espaço de armazenamento utilizado como para aliviar a pressão sobre a E/S , através do melhor aproveitamento dos recursos computacionais. Nesta dissertação fizemos um estudo comparativo de três compressores, sem perdas (lossless), aplicados a dados científicos. Avaliamos o desempenho alcançado pelos compressores e respetivas implementações paralelas, respetivamente, gzip e LZ4, ambos usados como compressores genéricos e o FPC, um compressor específico para dados em vírgula flutuante. Um outro compressor MAFISC para dados científicos, baseado em filtragem adaptativa, foi também, brevemente posto à prova. No final, apresentamos uma comparação bastante completa entre os ganhos obtido em velocidade e eficiência dos compressores paralela e as taxas de compressão. Usando compressão paralela com pigz podem obter-se ganhos médios de 12 para a velocidade, para 12 fios de execução (threads) e eficiência próxima da unidade. O estudo desenvolvido parece poder concluir que o gzip é o algoritmo de compressão mais abrangente, mas o LZ4 pode substituí-lo quando há exigências de compressão e descompressão mais rápidas, à custa de taxa de compressão. O FPC pode alcançar taxas de compressão ainda mais elevadas, para tipos de dados mais restritivos. Pelo seu lado o MAFISC parece cumprir os objetivos de obter elevadas taxas de compressão, mas à custa do aumento significativo do tempo de compressão.

Contents

List of Figures

List of Tables

Part I.

Introduction and related work

1. Introducing the problem and the motivation

Over this Chapter we introduce in Section 1.1 the current scenario and the problems that have become evident in the last few years. The motivation in Section 1.2 takes an approach over the possible solutions that can be created or applied to counter the difficulties, and we finish with Section 1.3 where the objectives for this work are written down.

1.1. Introduction

In the last decade the available computing power has been growing accordingly to Moore's law predictions. The supercomputing facilities are evolving from the Terascale to the Petascale [1]. As of November 2013 the top 31 ranked machines in the top500 are above the measured 1 Petaflops[1] mark, while that four of them are already past the 10 Petaflops mark. Based on the current expanding rate, it is expected that these systems hit the 100 Petaflops mark around 2016, and the Exascale near 2020. Accordingly, the Exascale should be expected to arrive in only six years from now. Due to the increased computing power the generated data consequently increases, as "virtual" scientific experiments are able to produce a larger quantity of numeric data. Along with it an I/O bottleneck has become evident and it turned into a difficult problem to address, because I/O throughput has not been able to comply with computing power growth. More and more data needs to be read/written to and from the storage systems, but the devices that support it struggle to cope with the demand. As the data also reaches Petascale it too becomes a problem to handle, besides the storage difficulties. Taking into consideration the current struggles faced in supercomputing, a different approach is mandatory for the coming Exascale computing.

There is a very well-known technique that has been used for the past thirty years, and is now becoming a prominent part of the solutions that aim to control the data growth problems, and its consequences. We refer naturally to data compression, and more particularly to scientific-data compression. It can have a positive impact over these problems as it allows to handle more data by making best use of the available storage

[1]Petaflop=1×10^{15} floating-point operations per second

space and I/O throughput. Nonetheless, applying compression requires extra computing resources and time, and while that the computing resources usually are readily available, the extra time can be a problem. There are metrics that can be measured to assess the usability, and the possible trade-off, of using compression.

1.2. Motivation

The motivation for this work comes from the fact that the aforementioned problems are nowhere near being handled, and that any extra research in them is a valuable addition. By becoming knowledgeable of recent compression algorithms, such as LZ4[2] and FPC[3], there is an interest of performing some tests in scientific data to assess the performance.

The bottleneck problem worsens with the increasing amount of executing units on a CPU, or other computing devices. As more units execute, more data is requested to be accessed or stored concurrently, but as the available I/O throughput reaches its limit the systems start to *starve* for data. By supplying the execution units with compressed data the pressure on I/O system is alleviated. Therefore it also implies that the data needs to be decompressed before consumed, and that it is faster than simply waiting for the same uncompressed data. Another advantage of using compressed data is the direct consequence, and original purpose, of saving storage space. Even though the storage systems are cheaper every year, the amounts of data imply a constant investment. Hypothetically, even by simply saving 10% of the storage space, the cut in monetary costs should be very significant. In a very recent work [4] the authors report that there could be savings between 36k€ and 46k€ in tapes annually, by using their compression method in the evaluated climate computer centre.

Multi-core compression can have a disadvantage of the extra I/O demand it puts on the system. Nonetheless, it also means that when consuming data there are more computing power to apply. Parallel execution is a scenario that can provide faster compression for larger input files.

Compression ratio (CR) used to be one of the most important characteristic of a compressor. Nowadays the paradigm is changing substantially, with the arrival of super-fast compression algorithms that sacrifice some CR for faster execution times. This potentially paves the way for real-time compression, an *invisible* use of compression to the user. Projects such as the Linux kernel module ZRAM (formerly known as compcache) compresses a portion of the data in the RAM memory, and are specially useful for small devices such as embedded devices (implemented in Android 4.4) and netbooks. LZ4 for being so fast decompressing allows for a real-time utilization, and is used in the most diverse domains. For example it is used in file systems, operating systems, search engines, computer games and even in Zram caching, between others. *In-situ* compression techniques are on the rise, specially useful in complex computing systems with many shared resources, such as scientific computing clusters.

Lossy compression can be a very interesting approach to deal with data that does not require full precision. Scientific data can have multiple purposes depending on the domain and the objective of the study. While some data is kept in storage for long periods of time (years) for historical comparison, such as climatic data, other data can be erased after it goes through filtering and post-processing steps until it reaches the desired results (i.e. raw data versus meaningful information). This data after being processed can have the sole purpose of visualization, which do not require the full precision as the data that is used for the numeric calculations. This is a situation that is really dependant on the field and the preferences of the scientists themselves. With lossy compression still keeping a high level of accuracy, some scientists might be able to embrace it and end up saving important resources.

In this work we assess the usability and possible advantages of using compression when having average sized volumes of data, with a scientific background, which usually translates to hard-to-compress floating-point data. It is not in the scope of this work to perform algorithmic analysis, but to evaluate how different compressors with different purposes can provide some advantages to the scientific computing community. Our motivation is to focus on the lossless algorithms, as they are still best regarded by scientists than the lossy counterparts. Nevertheless lossy compression can be one of the best solutions to deal with the increasing size of data.

1.3. Objectives

The objectives proposed in this work go in the direction of helping computation scientists that struggle to manage the data they produce as well as improve performance. To help with this cause we present a comparative study of the performance of six compressors using real samples of scientific data. The analysis consists of the three serial implementations of the compressors, and their parallel versions. At the core of our study is assessing their scalability on the growing multi-core architectures by measuring the parallel speedup and efficiency, as also verifying the compression ratios achieved. We also aim at briefly testing a different approach that has a stronger focus on data compression by using filters to reduce data entropy, therefore achieving better compression. All the tested compressors produce recoverable data to the original form, i.e. lossless compression.

As a first stage it is imperative to build a meaningful group of datasets, with a scientific source, and characterize each file individually. The datafiles should all have the floating-point type, preferably in double precision. The characterization will consist of statistical information such as number of elements, quantity of unique values, entropy and their randomness.

The second stage is to perform all the tests resultant from the interesting combinations of compressors with different settings and the datafiles. The measurements should be methodical and performed on a defined execution test bench.

In a final stage we pretend to experiment a filtering compressor, that focuses on compression rather than speed. The tests should be performed using the same datasets and execution hardware. However, this experiment is somewhat isolated from the other tests because this compressor works on top of a data library.

The last objective is to do an overall assessment of the tests we are able to execute. Some conclusions should be achieved from possible advantages or disadvantages of using the tested compressors, their scalability and compression ratios. The datasets can give us a clue of the best study disciplines to bet on compression. Future work will be based on the conclusions achieved.

2. Related work

Data compression has evolved in different directions since the time where compression ratio was the major metric. Nowadays some favour compression ratio, others compression throughput, and others even opt for a lossy approach to maximise both ends. The scientific community is very pedantic when it comes to floating-point data, therefore lossy algorithms are frowned upon and lossless algorithms are the usual choice. Commonly lossy compression can be applied to visualization data, where the user will not be able do distinguish between full precision or a integer in a CG (Computer Graphics) animation for example. Even though and as far as our knowledge goes, data compression has not been frequently used in scientific computations. The solution for the data growth has been to simply expand storage space, but it is not viable for much longer. Data must be controlled in some way, specially with so much computational resources available to help.

Throughout this chapter we cover some of the related work in the area of data compression, usually with scientific data. The covered works provide algorithms for floating-point data compression, with different purposes, or new ways of applying data compression. Focus goes to lossless implementations, but we still cover some lossy approaches for some of the compressors. The chapter is organized as follows, lossless compression implementations are covered in Section 2.1, followed by lossy Sec.2.2, and end with other more heterogeneous approaches using compression in Section 2.3.

2.1. Lossless compression

In this Section we cover some works using or implementing lossless algorithms, ordered chronologically. We focus only on the, in our opinion, most relevant algorithms for scientific data, or that are novel approaches to compression ratio and/or speed. Independently of the implementation for each work, being lossless means that the decompression is guaranteed to recover all the original bytes, no information is lost in the compression process.

The ROOT framework specification Brun and Rademakers [5], Brun et al. [6] already contemplated support for file compression. The approach was to compress each ROOT object before being written to a file. The compression is based on a gzip algorithm with nine different levels, being 1 the fastest, set by default, and 9 the highest compression setting. Since gzip is an asymmetric algorithm the decompression time can be very small compared to the compression time, because it is independent from the selected compression level. ROOT allows the use of different compression levels per each object within a ROOT file, which gives the users the ability to leverage the compression ratio and compression time. If the data is hard-to-compress (high random entropy levels) it can be chosen to not compress at all (level 0). Because a ROOT file is structured like a tree, very much like directories in a file system containing many types of data, using compression allows for good results.

The use of a gzip based compressor, that in turn implements the DEFLATE algorithm which is an evolution of original LZ77 Ziv and Lempel [7], plus a form of entropy encoding (Huffman coding), did not present anything new on compression itself. But offering the scientific community a data analysis framework that apply data compression at its core, qhile allowing for compression level control, was an important step.

Delta-compression is implemented by Engelson et al. [8] as part of a lossless algorithm that packs the higher-order differences between adjacent data elements. This algorithm focus on the fact that many scientific datasets represent ever-changing particles properties, therefore it takes into account varying domain steps (typically time or position). It is described as a simple algorithm that has high performance and delivers high compression ratio for datasets that change smoothly. In this delta-compression implementation both lossless and lossy (Sec.2.2) variants can be used. Because it uses correlation between adjacent floating point values, it is considered as an alternative to text compressors. Nonetheless it only achieves considerable compression ratio for smooth data sequences. An array is called smooth if it can be well approximated by the extrapolating polynomial based on previous values. In the simplest case, a function that has very small changes like a polynomial of low order, or a constant in the extreme case, will be greatly compressed by the algorithm, for the extreme case by a single constant value.

For the implementation it is considered how numerical data is represented in memory. A double floating-point is represented as a 64-bit integer, hence the arithmetic is made integer. The leading bit-sequences of the difference result get truncated, being them zeroes for positive or ones for negative numbers, thus reducing the size by the number of truncated bits. As an example, zeroes in 0000`0101` can be truncated and only `0101` is stored, effectively saving 4 bits. This approach takes advantage of the fact that the difference between the elements is small relatively to their own value.

The ALICE experiment at CERN has one of its main detectors Time Projection Chamber (TPC) producing big amounts of data. Worried to keep the complexity and cost of data storage as low as possible Nicolaucig et al. [9] intended to reduce the volume of data using suitable compression methods. Both lossless and lossy implementations were tested. The compression applies entropy coding to the differences of the times in two consecutive bunches (group of adjacent samples coming from the sensor pad). Thus reducing the entropy from the source by exploiting the correlation present in the TPC data. The compression achieves practically a compression ratio (CR) of 2, i.e. half the size of the original data. When compared to general compressor gzip, using maximum compression level 9, this achieves only 1.25 CR. The compression was tested for a real-time implementation in the system when fully operational. Nicolaucig et al. [9] report that the compression system can be easily implemented in real-time either in DSPs (Digital Signal Processors), FPGAs (Field-Programmable Gate Array) or ASICs (Application-Specific Integrated Circuit), this is all specific purpose hardware. An implementation, at the time, using general CPUs would probably not be effective.

FPC is a lossless compression algorithm for linear streams of 64-bit floating-point data. Its origins come from Ratanaworabhan et al. [10] work that implements a differential-finite-context-method (dfcm) prediction compressor (DFCM), subsequently used in FPC. In [11, 3] the FPC compressor is well defined and explained, which adopts another complementing predictor (fcm) besides the first dfcm. This compressor face the specific problem of scientific floating-point data and propose an implementation for a fast, lossless, compression algorithm tailored for high-performance environments where low latencies and high throughput are essential. It is single-pass (i.e. can be used as a streaming compressor), and delivers good average compression ratio on hard-to-compress 1D numeric data. The limitation of the input data reduce the chances of adoption in scientific simulations because multi-dimensional datasets are widely used. FPC implements a simple algorithm that can be implemented entirely with fast integer operations, resulting in a compression and decompression time one to two order of magnitude faster than other more generic algorithms. The algorithm is designed for 64-bit floating-point values and was stated to be fast enough to support software-based real-time compression and decompression. In [10] the algorithm is integrated in a message-passing interface (MPI) library[1], compressing the messages at the sender and decompressing at the receiver. Tests achieved between 3% and 98% faster execution times of scientific numeric programs, in the cluster used for the experiment.

The steps for the algorithm compression are described as follows: it starts by predicting each value in the sequence and performing an exclusive-or operation (`xor`) with the actual value. FPC uses two predictors, dfcm and fcm (both perform table lookups that contain values from previous predictions), which are initialized with zeroes before starting to be populated in compression or decompression. After each prediction they are updated with the real value in order to guarantee that they generate the same sequence of values (predictions) during compression and decompression. The best predicted value (i.e. closer to the actual value) is selected to be used in the `xor` operation. The closer the prediction is to the actual value the more sign, exponent and significand bits will be the same (leftmost bits, see Figure 2.1). After each prediction the predictor tables are updated with the actual double value to ensure that the sequence of predictions are the same during

[1] Message-passing library used in parallel systems to exchange data between the multiple CPUs executing a given program

both compression and decompression. A good prediction results in a substantial number of leading-zeroes in the calculated difference, which are then encoded by simply using a fixed-width count. The remaining uncompressed bits are output in the end, after the count of leading-zeroes. Both the prediction and the `xor` operation are really fast to compute. The first is a fast hash-table lookup while that the `xor` is a low level instruction implemented by the CPU. This allows for a very fast compression and decompression algorithm.

The first iteration, DFCM[10] compressor, implements a more sophisticated predictor that stores two difference values in each table entry, against only one as FPC do, and uses a more elaborate hash function. However FPC can outperform DFCM on the majority of the tested scientific datasets used in [10, 11, 3], because FPC contains the second predictor that often complements the first. Also it is possible to vary the predictors table sizes, allowing a trade off between throughput and compression ratio. The scientific datasets tested by the authors are publicly available, and therefore are used in conjunction with our own datasets in Part II.

In the decompression stage the algorithm starts by reading the predictor identifier and leading-zeros count. Then the remainder bytes are read and the sequence is extended with the zeroes to reach a full 64-bit length. Based on the predictor bit specifier this number is `xored` with the correct prediction to recreate the original value.

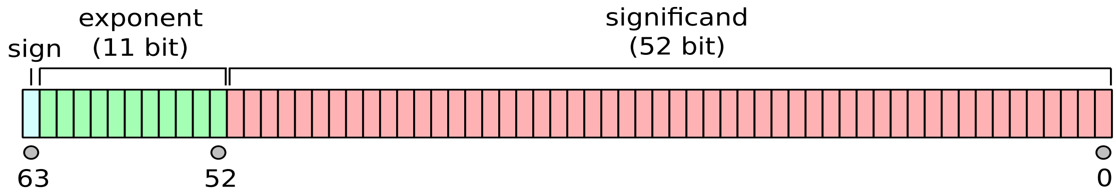

Figure 2.1.: IEEE 754 double-precision binary floating-point format

LZ4 by Collet [2] is a very fast lossless compressor based on the well-known Ziv and Lempel [7]. Unlike the former FPC, and because it is a general compressor, it is not designed to address only floating-point data. What makes LZ4 stand out from other LZ dictionary coders is the fact that it can be really fast. When compressing it can reach throughputs of more than 300MB/s per core, while that during decompression it is even faster with speeds up and beyond 1GB/s per core. Consequently it can reach RAM speed limits on multi-core systems. With this characteristics it is a very good candidate to perform real-time compression, as the compression and decompression times can be hidden by memory accesses.

The algorithm works by finding matching sequences and then saving them in a LZ4 sequence using a token, that stores the literals (uncompressed bytes) length and the match length, followed by the literals themselves and the offset to the position of the match to be copied from (i.e. a repetition). There are optional fields for literals and match length if necessary, and the offset can refer up to 64KB. With the offset and the length of the match the decoder is able to proceed and copy the repetitive data from the already decoded bytes. This decompression is so fast due to its simplicity along with the fact that entropy coding is not used. Regarding the way that the algorithm search and finds matches there are various possibilities, and in fact it is not restricted as long as the format is kept. The author suggests that it can be a full search, using advanced structures such as MMC (Morphing Match Chain), BST (Binary Search Tree) or standard hash chains, between others. Some sort of advanced parsing, such as lazy matching, can also be achieved while respecting full format compatibility (achieved by LZ4hc, the high compression variant of LZ4). To achieve higher compression ratios more computing time can be spend on finding the best matches. This results in both a smaller datafile as well as faster decompression.

The fast version (LZ4) uses a fast scan strategy, implemented as a wide single-cell hash table. The size of the hash table can be modified and still maintain format compatibility. The ability to modify the size of the table is important because of some restricted memory systems. Consequently, the smaller the table the more collisions occur (false-positives), reducing the compression ratio. The bigger the table the better the compression possibilities, while also making it slower. The decoder, similarly to gzip, is asymmetric which means it does not need to know about the method used to find matches and, requires no additional memory. LZ4hc with higher compression ratio and along with its super fast decompression speed, can have increased interest in a write-once read-many fashion.

MAFISC (Multidimensional Adaptative Filtering Improved Scientific data Compression) lossless compressor by Hübbe and Kunkel [4] focus the effort on storage reduction for climate data while overlooking time of execution. The goal of this research, as a direct consequence of better compression ratio, was to cut down the expenses that are spent on magnetic tapes for the data storage at the DKRZ[2]. The algorithm performs compression by first applying some developed filters to the data, expectedly reducing entropy, which then goes through a dictionary and entropy coder.

Similarly with previous works [8, 3] it uses fixed point arithmetic for performing calculations between floating-point values, with the purpose of reversible operations. This conversion happens implicitly before any other filter is used. The developed algorithm itself is not performing compression as it delegates that function to LZMA, which is a general compressor, such as gzip and LZ4, from the LZ family. This compressor is known for having some of the best compression ratios, while keeping decompression speed relatively similar to the other algorithms in the family. Hence compressing more than gzip and consequently than LZ4, but being much slower, especially during compression.

The filters that MAFISC applies on the data are responsible for facilitating the work of lzma. Nonetheless, as the algorithm always falls back to lzma, that is the minimum expected compression. One of the filters

[2]Deutsches Klimarechenzentrum - German Climate Computing Centre

computes and replaces the values by the linear differences between each consecutive value in the dataset, leading to small values that imply lower entropy. Thus, it exposes very repetitive differences for datasets that are *smooth*, a scenario already explored by other compressors. There are also a bit-sorting filter that reorders the internal bits of the values, by distributing the most significant bits across the bytes that compose the value, which distributes the entropy and enhances its compressibility for entropy coders. Other two filters implemented in [4] are a prefix transformation filter and an adaptive filter. Not all the filters are applied, the decision happens accordingly to the best CR resulting out of two different combinations tested in a chunk of data (or none, if data gets inflated by the filters). The filter chain order must be stored together with the compressed data allowing for the data to be understood and decompressed.

While not working towards the reduction of execution times, the authors still take it into account. They realize it is possible to cut the storage costs greatly (between 36k€ and 46k€) with a relatively minimal investment on computing machines to solely compress the data.

2.2. Lossy compression

In this subsection two lossy implementations of algorithms covered in Sec.2.1 are quickly analysed. This kind of implementations are not in the main scope of this work, nevertheless they are important and interesting for a state-of-the-art overview. The two lossy coders dealt with scientific floating-point data.

Delta-compression based algorithm, by Engelson et al. [8] has a lossy variant with the main objective to address scientific smoothly changing data. When the purpose of the simulation results is to be visualized, in the form of 2D or 3D graphics or images/animations, the full data precision is no longer necessary. The lossy implementation is an extension of the basic algorithm and it can be parametrized to adjust the trade-off between CR and precision. The actual approach simply consists of truncating some bits at the end of the bit string representation. With less bits in the stream the compression is achieved. To compensate the propagation of error introduced, one exact value (i.e. with full precision) is used for every p lossy compressed values.

ALICE datasets originated by the TPC detector were tested with lossy compression [9]. These datasets contain many samples each with different quantities, some of them more important for the trajectories reconstruction than others. For the important Centre of Mass (CoM) positions, a quantization is applied before compressing. This reduces the range of values it can take, hence reducing the entropy. This lossy approach understandably achieves higher compression rate than the lossless implementation. The same compression process is applied, but at this stage the values already have lower entropy. With the coarser quantization level selected the compression yields a CR of approximately 4.3 (i.e. 4.3 times smaller than the original size).

2.3. Other approaches

Besides the more conventional approaches presented before, in a sense that it takes an input stream of data, compress it and then save it on a storage device, this subsection succinctly refers to some distinct methodologies using compression to improve overall performance.

Yang et al. [12], Zukowski et al. [13] implement compression directly on RAM memory and in-between RAM and Cache memory to improve the system performance. In [12] a very fast high-quality compression algorithm for working data set pages on RAM is described. The algorithm named PBPM (Pattern-Based Partial Match) explores the frequent patterns that occur within each word of memory, and takes advantage of the similarities among words by keeping a hashed small two-way set associative dictionary. The dictionary is managed with a LRU (least-recently used) replacement policy. Reducing the used memory space allows for an overall better scalability of the system. The approach in [13] is to use super-scalar compression algorithms between the RAM and CPU cache, rather than the common idea to apply the compression between RAM and storage. By super-scalar it means that the CPU can achieve an Instruction Per Cycle (IPC) higher than one, reflecting in very high throughput. Results showed that their algorithms provide a decompression speed in the range of more than 2GB/s. It is an order of magnitude faster than the conventional compression algorithms, making the decompression almost invisible. With this techniques it is possible to reduce the I/O bottleneck as it keeps CPU busy while working with data when there are I/O stalls (i.e. the CPU does not have to waste cycles waiting for data).

Lofstead et al. in Lofstead et al. [14] take an approach to improve the I/O efficiency in the accesses to underlying storage platform of a large-scale system, for different machine architectures and configurations. Therefore, the ADIOS (Adaptable I/O System) API, reported in the paper, is designed to be able to span multiple I/O realizations. This while being able to address both high-end I/O requirements and still offer a low-impact auxiliary tool integration for selecting other transport methods (i.e. with a simple XML file modification change the whole I/O parameters for the different simulations or datasets). By providing highly tuned I/O routines through their library, to different kinds of data and transport methods, it can improve the system performance even without compressing the data.

With a concern for inter-node I/O bandwidth Welton et al. [15] take an approach to compress the data between node communications in a large-scale system. They describe the IOFSL (I/O Forwarding Scalability Layer), a portable I/O forwarding implementation that by adding compression to the forwarding layer (tested with general algorithms zlib, bzlib2 and lzo), evaluates the changes in throughput to the application and to the external file system. For certain types of scientific data it was observed significant bandwidth improvements. Nonetheless it is highly dependent of the data being transferred, thus only useful on slower networks.

A very interesting approach is taken by Schendel et al. [16] where they introduce ISOBAR (In-Situ Orthogonal Byte Aggregate Reduction Compression) methodology as a pre-conditioner to lossless compression that identifies and optimize the compression efficiency and throughput of hard-to-compress datasets (using zlib and bzlib2 for the actual compression). ISABELA (In situ Sort-And-B-spline Error-bounded Lossy Abatement) from Lakshminarasimhan et al. [17], with mostly the same authors from ISOBAR, performs lossy compression by applying a sorting pre-conditioner that improves the efficacy of cubic B-spline spatial compression, and applies delta-encoding of the high order differences in the index values. Both try to identify and optimize the compression efficiency and throughput of hard-to-compress datasets. In [18] an hybrid compression I/O framework was tested, with the underlying support of ADIOS [14], allowing to separate the high-entropy components of the data from the low-entropy components thanks to the proposed pre-conditioner in [16]. Therefore, independent streams of data are formed which may be interleaved. The high-entropy components are sent across the network and to disk asynchronously while the low-entropy data can be compressed (using gzip). This allows to hide the compression costs and fully utilize all computing, network and I/O resources in the system. These works also make use of the datasets presented in [3] and that we use on Part II.

pFPC is the parallel approach by Burtscher and Ratanaworabhan [19] to their original FPC algorithm. In [20] the authors also introduce gFPC, a self-tuning implementation of FPC that provides better compression ratio and decompression speed. For the latest iteration O'Neil and Burtscher [21] describe a GPU implementation of FPC, named GFC, with the capacity to reach 75Gb/s compressing and more than 90Gb/s on decompression while providing a slightly lower compression ratio.

The majority of compression presented in this chapter execute sequentially. No related work was found to use LZ4 or pFPC, which is only independently benchmarked by the authors, i.e. it was not compared with any other compressor. The work in this field seems to be lacking a comparative study of parallel compressors.

Part II.

Test bench, methodology, results and conclusions

3. Test bench characterization and methodology

In this chapter the specifications for a stable test environment are described. First we define the physical test bench where all the executions took place, then we take a descriptive approach to the algorithms explaining their major properties and continue to the datasets characteristics. The final section describes the methodology used for the tests and measurements.

3.1. The test bench

Setting up a stable test bench is critical to achieve reliable results. Here we provide the hardware characteristics from the machines that executed our tests, and define the compilers versions and flags used across all the experiments.

3.1.1. Execution nodes characteristics

To perform all the tests in this work, and in order to get the most stable results as possible, a group of cluster execution nodes was selected and used throughout the tests. All the displayed metrics and results come from this same execution nodes, from the local SeARCH cluster hosted at University of Minho. The SeARCH cluster is a research project initially funded by FCT (Fundação para a Ciência e a Tecnologia) and is currently supported by funds from various departments. Therefore, it tries to satisfy a diverse community, and consequently contains a somewhat heterogeneous group of execution nodes, from various generations, and diverse brands and other characteristics.

A selection of six nodes is made based on the hardware specifications. They are all based on the same CPUs belonging to the nehalem microarchitecture, but two of them have four times more ram RAM memory than the other four. The Table 3.1 summarizes some of the specifics per node. Each one has two CPU chips with six cores, that can run 12 threads using Hyper-Threading technology, thus totalling 24 threads per node. The local storage devices are hard drive disks. Using solid-state disks (SSD) would be ideal for this work, as it improves on the I/O bottleneck, but this devices do not seem to be yet openly available on the cluster.

		Intel Xeon (different node brands)	
	Specifications	compute-601-1..4	compute-601-11,12
CPU	model	2×X5650	2×X5650
	Cache L2+L3	2×(1.5MB+12MB)	2×(1.5MB+12MB)
	Cores→Threads	2×(6c→12thr)	2×(6c→12thr)
	Frequency	2.66GHz	2.66GHz
	RAM	12GB	48GB
	CentOS kernel	2.6.18-128.1.14.el5	2.6.18-128.1.14.el5
	Storage	unknown hdd	unknown hdd

Table 3.1.: Hardware characteristics of the selected computing nodes.

Only when analysing some of the metrics for the performance assessment, it was discovered that the nodes compute-601-11 and compute-601-12 are slightly faster. Running a quick test, an execution time of 48 seconds was measured on the other selected nodes, while that compute-601-11 took 43 seconds to complete, hence 5 seconds faster. Apparently, and after contacting the cluster sysadmin, these two nodes are assembled by a different brand, and the difference is likely coming from an extra flag IDA[1] that control CPU frequency and that is not defined on the other four selected nodes. After looking back to the execution logs, very few instances were identified of tests that executed on these nodes. The vast majority of the tests were performed on the nodes compute-601-1 to 4. Therefore we do not expect disruptive results.

3.1.2. Compiler options

Since the purpose of this work is to assess performance, it is of great importance to use a good compiler and flags that are capable of exploiting the underlying hardware. Available on the cluster is the old 4.1.2 version of GCC (GNU C Compiler), the dominant open-source C compiler in linux, and version 11.1 of ICC (Intel C Compiler), that is commercial closed source. Recently it was known[2] that Intel is making optimizations for their compiler and new hardware to increase performance in zlib, a widely used compression library that implements the same algorithm as gzip. It is a recognized fact that Intel has optimizations in their compilers specially for their products. Both compilers were tested using LZ4 as compile test subject, to evaluate the performance. Because the available version of GCC was so old, a much more recent 4.8.1 version was compiled on the cluster to be used locally. Unexpectedly the outcome weighted in GCC's favour. Maybe because GCC 4.8.1 is newer than ICC 11.1, the execution times achieved by LZ4 when compiled using GCC were lower ranging from

[1] Intel Dynamic Acceleration technology (IDA)

[2] `http://www.phoronix.com/scan.php?page=news_item&px=MTUyNzY` Accessed January 28, 2014

some milliseconds to 9 seconds, depending on the file and compression mode. Based on this observation GCC was elected as the compiler to use for all compressors in this work.

When it comes to the compilation options in GCC only two choices were taken. First the general flag for optimization was set to its maximum level -O3. Depending on the selected level GCC will turn on specific optimizations, such as loop optimizations, vectorization, inline functions, etc. Since the nodes have relatively modern Intel chips, it was searched in the manual for an appropriate -march flag, deciding for -march=corei7. With this flag GCC tries to make use of more recent instructions that come with modern processors. In the end of the coming Section 3.2.2 it is exposed a mistake relatively to the compilers, which just proves that they have an important weight in the performance.

3.2. Compressors

This section superficially describes the six compressors, three single-threaded and their shared-memory multi-threaded implementations, that were selected to be compared with each other. We aim to assess performance and scalability, hence the focus and effort is on measurements. An in-depth understanding and tuning of the algorithms was not on the scope, although we acknowledge that it would allow possible improvements in performance. The serial compressors are gzip, LZ4 and FPC. Their multi-threaded counterparts are pigz, lz4mt and pFPC respectively. In beforehand lets clarify the diverse nomenclature used in this work. When referring to gzip, LZ4 or FPC we might use one of these (prefix suffix) forms: original, serial or single-threaded for prefix and compressors, algorithms or programs for the suffix. The same happens when referring to pigz, lz4mt or pFPC, which we might use: multi-threaded or parallel for the prefix and compressors, algorithms or programs for the suffix.

The origins of this work came from the interest to evaluate LZ4 compressor in a scientific simulation domain. This compressor is a modern fast general-purpose compressor, which can achieve RAM-bandwidth decompression speed. When studying the state of the art related to scientific compression we learnt about FPC, a specific double-precision floating-point compressor that falls perfectly in this area because scientific data is mostly produced and consumed in floating-point format. The decision for the third compressor gzip came naturally because it offers a good balance between speed and compression ratio (CR), and it is a widely used general compressor and a great point of reference. In order to get the best performance possible when compressing data, we evaluate the algorithm's parallel implementations. Aware of the possibility to deteriorate the I/O bottleneck problem, we believe that higher compression levels that require more computations can benefit from the parallelization, hence improving the overall performance.

3.2.1. Serial compressors

Gzip and LZ4 derive from the Lempel-Ziv (LZ) family compressors, implementing variants of the LZ77 algorithm [7]. They are general-purpose compression utilities that operate at byte granularity, looking for repeating sequences of bytes within a given sliding windows that goes through the input. gzip further uses entropy coding in the form of two Huffman trees, one to compress the distances in the sliding window and another to compress the lengths of the matching sequences as well as the bytes that did not belong to any sequence. LZ4 does perform a matching algorithm, eliminating repetitions, but seems to skip entropy coding, which makes it much faster but compress less. Both of this compressors take a minimum compression level setting of one, the faster mode, and maximum of nine, the highest compression mode. The different modes change the size of the window to look for matches, between other specifics, hence making it possible to find better (longer) matches. LZ4 only accepts the two extremes (one or nine), while that gzip allows for an intermediate value (one through nine).

FPC [3] was developed to only compress floating-point binary data. The internals are completely different from the two other dictionary coders. Nonetheless, it also operates at byte granularity which is more efficient than bit granularity, and compresses by predicting each value (in a reversible way), `xor`ing the real value with the predicted and leading-zero compressing the result. The better the prediction, the more zeroes come from the `xor`[3] operation, hence counting the leading-zeroes yields a higher number that is then encoded. The non-zero residual bytes are encoded without encoding. In FPC all of the floating-point doubles are interpreted as 64bit integers and it only uses integer arithmetic, for performance reasons. The compression level depends on the quality of the prediction, and for that a hash table is used to record the real values that serves the predictors. The size of the table influences the prediction, hence in FPC to specify an higher compression level the user simply specify larger hash tables. Theoretically there is no maximum, so the compression level ranges from one to hardware limit. We decided to use levels from 1 to 26, to ensure that a level higher than the authors was tested. In Section 4.4 we approach this decision and the difficulties that we failed to expect.

MAFISC by Hübbe and Kunkel [4] is a compressor implemented as a filter to HDF5, which performs filtering of the data in order to provide better compression patterns (lower entropy) to the following lzma[4] compressor, also from the LZ family. The filters are invertible, analogous to the FPC predictors, so that it is possible to reconstruct the data, losslessly, by knowing which filters were used and reapplying them. The operations are performed in integer arithmetic for performance and because it avoids floating-point problems, such as rounding errors and catastrophic cancellation [22]. To compare MAFISC against directly using lzma compression we use xz, a publicly available program that implements the lzma algorithm. Similarly to gzip

[3] `xor` operation turns identical bits into zeros
[4] LZMA - Lempel-Ziv-Markov chain algorithm

and LZ4 it accepts compression levels from level one to nine, being the latter the higher compression mode. It also accepts a flag -e for *extreme* compression, but makes it extremely slow, thus unsuitable for comparison. We performed basic testing with MAFISC and lzma in Section 4.7, after all the others tests were complete.

3.2.2. Parallel compressors

Take an input file, divide it in chunks, and compress them individually on local execution threads (preferably on separate cores). This is an approach to parallel compression, and it is what the compressors do in a shared-memory context. The compression has an underlying exploitable parallel nature, because each file is processed in blocks when compressed. The multi-threaded approach performs each block compression independently in a thread, and joins the resulting compressed blocks into the final output file. For example, pigz uses a single thread to write the data, but n other threads to compute 128KB blocks. The three parallel compressors allow to set multi-threaded mode or to execute with a single thread, analogous to their serial versions. This is useful to test the possible overhead of using the parallel implementation by comparing the serial with the parallel running on a single thread, which we did on chapter 4. The compression level parameters are the same as the serial versions, as one should expect. However pFPC requires for a chunk size specification that represent the number of floats for each thread. Based on the authors work [19], three chunk sizes seemed to be interesting (good results) and were selected to test: 1024, 8192 and 65536. The chunk size that in overall presented results with better executions times and CR was 8192, which was selected for all the tests.

Two side notes should be acknowledged about pFPC and lz4mt. First it is stated in pFPC webpage[5] that the provided code is not prepared for maximum performance due to slow sequential data accesses. Because it was not in the scope of this work to explore the approaches in parallel implementation of the compressors, the programs are tested as available. Secondly, at some point lz4mt presented some problems related to decompression, which were reported[6] and fixed by the author in a posterior commit[7].

All the tested programs, both single and multi-threaded, are publicly available under a permissive open-source license, with the exception of FPC/pFPC that are covered by two academic licenses. The implementations are all written in C/C++ language, which is known for having good performance and made compilation easy using GCC and the chosen flags. As a quick remark about the parallel implementations, pigz and pFPC use lower level pthread implementation in C, and lz4mt uses C++11 higher level threads through *future* objects. Pigz makes use of zlib, which implements DEFLATE, the same algorithm as gzip. The Table 3.2 summarizes the algorithm's versions, compilation flags and compression parameters used. MAFISC, HDF5 library and `xz`

[5] `http://users.ices.utexas.edu/~burtscher/research/pFPC/` Accessed January 28, 2014
[6] `https://github.com/t-mat/lz4mt/issues/21` Accessed January 28, 2014
[7] `https://github.com/t-mat/lz4mt/commit/2a8ed67` Accessed January 28, 2014

Compressor/library	Version	Compile flags	Compression settings
gzip	1.6	Gcc 4.8.1 -O3 -march=corei7 -fgnu89-inline	1 (faster) to 9 (higher)
LZ4	r99,r107,r109	Gcc 4.8.1 -O3 -march=corei7	1 (fast) or 9 (high)
FPC	1.1	Gcc 4.8.1 -O3 -march=corei7	1 to 26
MAFISC/HDF5	a* / 1.8.12	Gcc 4.7.2 -O3 / Gcc 4.7.2 -O2	Default, 1,6,9
xz (lzma)	5.0.5	Gcc 4.7.2 -O2	1(faster),6,9(higher)
pigz / zlib	2.3 / 1.2.8	both: Gcc 4.8.1 -O3 -march=corei7	1 (faster) to 9 (higher)
lz4mt	66990ac (28 Sep, 2013)	Gcc 4.8.1 -O3 -march=corei7	1 (fast) or 9 (high)
pFPC	1.0	Gcc 4.8.1 -O3 -march=corei7	1 to 24, chunk=8192

Table 3.2.: Compressors versions, settings and compile flags used. The double horizontal line separates the bottom parallel compressors from the others. Version a* means that no number was specified in the provided code.

were compiled with other version of GCC because a different environment was used to facilitate the testing. Nonetheless this version is quite recent, the tests that were performed ran on the same execution nodes, and the results are not meant to be directly compared to the other tests.

At some point the complete set of tests performed with pigz were discovered to be using an older version of zlib available on the cluster. When the issue was rectified by changing pigz linkage to a locally compiled zlib, using gcc 4.8.1 and -O3 -march=corei7, the performance went up considerably. Changing from the old version compilation to the new one made execution times manifest improvements in the order of 1 to 30+ seconds, depending on the file and pigz compression level.

3.3. Datasets

In order to perform the tests it was necessary to establish a solid, well defined, group of datasets to be used. Since the purpose was to assess compression using scientific data, we tried to gather the numerical data from different backgrounds and sources. In Table 3.3 the datagroups are briefly introduced, they are 33 datafiles in total.

Datagroup names	#Datafiles	Research Area	Software	Data Type
waterglobe	6	molecular modelling	TINKER	text
engraph	3	molecular modelling	TINKER	text
gauss09	4	electronic structure modelling	Gaussian 09	text
sci-files	13	message, numeric, observational	*diverse sources*	doubles
NTUPs	7	particle collision simulation	LIP code	ROOT files

Table 3.3.: Characteristics of the five datagroups originating from six different backgrounds

The six different disciplines covered by the datasets originate from various sources, mostly simulation programs. The molecular modelling datasets come from TINKER[8] and the electronic structure modelling

[8]TINKER is a complete and general package for molecular mechanics and dynamics

are produced by Gaussian 09, which provides state-of-the-art capabilities for electronic structure quantum modelling. The datagroup sci-files covers three areas and is used in various works from the authors of [10, 11, 21, 3, 20, 19, 16, 17]. For one it has five datasets covering parallel messages containing numbers sent by a node in a parallel system running NAS Parallel Benchmark(NPB) and SCI Purple applications. Second, numeric simulations results in four of the datasets, and third another four datasets this time with observational data comprising measurements from scientific instruments.

The NTUPs datagroup originate from work in simulations made at LIP[9], hence the data is stored in a ROOT file. This is the data analysis framework used by CERN and associated laboratories, and the files in ROOT are organized like directories on a file system with gzip compression applied on the objects stored (enabled by default).

3.3.1. Data makeover and transformations

Some of the datafiles arrived to us in a very raw format, specifically the files in text format. The files had to undergo some manipulation in order to contain only the desired floating-point values.

Data Clean-up The data stored in waterglobe, engraph and gauss09 (in text file format) had many types of variables and text (as opposite to floating-point numbers) written in them, so the first step was to perform a clean-up of the data by extracting, and sometimes reorganizing, just the floating-point numerical parts. We kept only floats because this tends to be the preferred format used in the majority of scientific applications, which need to have great precision, and because it is the only type of data that FPC handles. Text data is very redundant and could be ideally dictionary-compressed when using LZ coders. The transformations modify the datafiles in such degree that they will not be recoverable to the original format without further information, but nevertheless fully represent the floating-point datasets that we aim to evaluate. The purpose is to get the most streamlined datasets as possible, and assess the compression strictly on numerical values with scientific sources. Throughout this subsection the reader can follow the first column in Table 3.4, in order to have a better comprehension of the datafiles mentioned.

Besides all the stripping made to the datasets, in order to only keep the numerical values, an experimental change in the text files layout was applied. For the datasets waterglobe.arc.txt, waterglobe.vel.txt and engraph1_100.txt that have the values in a matrix style (e.g. $n$rows$\times$3columns, the three Cartesian coordinates), they were individually written into a single column text file. The single column with all the values is filled by reading the original text dataset in a row-wise fashion. By doing so we lower the entropy of a text file a little

[9]LIP - Laboratório de Instrumentação e Física Experimental de Particulas; is a Portuguese laboratory of scientific research that works in the field of high energy experimental physics. The research activities developed by LIP fit within the scope of international projects in collaboration with CERN and other scientific organizations.

more (no space character, only a value per row) and its structure becomes similar to a binary stream.

Binary representations had to be created for the text datafiles. For the conversion we wrote a small C++ program that reads the text files and outputs them to a binary file in single or double precision, as specified by a parameter. The inverse operation was also implemented for testing and correctness checking purposes, and it can write the values in text representation with the desired number of columns. For the datagroups sci-files and NTUPs this text-binary conversion was not necessary. Nevertheless NTUPs are ROOT files which require some work as described below in the Extraction paragraph.

Split/Join While that some datasets were joined into only one, others were split into more than one datafile. In the joining scenario case we created engraph1_100 that was originally separated in 100 parts (each corresponding to a time step in a molecular dynamics simulation) and NTUP1to5_floats.bin that simply contains the other five NTUPn_floats (each containing multiple events) fused together to form the biggest datafile from our entire dataset. In the case of gauss09 it suffered a split, because it had too much mixed data inside. The original unique file originated from one single simulation, but the matrices contained in it represent different properties. Two of them were quite large, hence a split to two different files seemed a logical step to take. From this split we created the datafiles gauss09_alpha and gauss09_density.

Extraction For the NTUPs datagroup the values stored inside each NTUP file (ROOT) were all extracted into two uncompressed binary files, one containing the floats and the other the doubles. This extraction process was made thanks to a small root_extractor program created with some help of ROOT scripts. In the root_extractor we loop through all the floats and doubles and output them to the according binary file. Accordingly, ten binary files were created out of the five original NTUPs, five containing the single-precision values and other five containing the double-precision values. Because the doubles were very few in quantity compared to the floats, the solution found to create a relevant datafile was to concatenate them all together into one slightly bigger file NTUP1to5_doubles. As written previously the same was made for the floats in order to create the biggest file in the dataset, with 7GB. This way, all of the resultant datafiles are in single-precision, suffixed with _floats with the exception of NTUP1to5_doubles.

Because FPC compresses double-precision floating-point data, by interpreting each double as a 64-bit integer, the use of single precision datafiles mean that it will understand a pair of floats as a 64-bit integer. While that it allows for the algorithm to run, this reduces the compression capabilities because as FPC only encodes the leading-zeroes, and the second float zeroes resultant from the `xor` operation (if any) are *lost* and encoded as is (uncompressed).

The sci-files were the only datagroup that did not need any aesthetic work because, as available, they come in separated simple binary stream files.

3.3.2. Data statistical metrics

In this subsection we go through the metrics analysed for the entire datasets (Table 3.4). In beforehand we can state that the following metrics are correctly calculated because we achieve the same values for the sci-files datagroup, as used in [3, 16]. Randomness is presented differently in the two related works, and to our understanding and opinion the formula from Schendel et al. [16] is the most correct, hence is the one we use. Equation (3.1) describes the percentage of uniques in a dataset, where V is the original vector consisting of all files, and V_{Unique} is the vector with duplicates removed.

$$Unique\ value = \frac{|V_{Unique}|}{|V|} \times 100\% \tag{3.1}$$

$$H(V) = -\sum_{i=1}^{N} (p_i \times log_2(p_i)) \tag{3.2}$$

$$Randomness = \frac{H(V)}{H\left(Random_{unique}\left(|V|\right)\right)} \times 100\% \tag{3.3}$$

Equation (3.2) represents the Shannon entropy $H(V)$, where N is the number of distinct elements (values) i, and p_i the probability of those elements, i.e. , the number of i occurrences divided by the total number of values in the file. An element of a dataset depends on the datatype that composes it. Consequently for text files an element is 1 byte (8 bits), for single-precision floats is 4 bytes (32 bits) and finally for double-precision doubles an element is 8 bytes (64 bits). The randomness is closely related with the entropy as described in (3.3). Its value reflects how close the Shannon entropy of the datafile if to that of a true 100% unique random datafile with the same number of elements. This may imply that was necessary to create synthetic datasets containing only unique elements, in the same amount of the datasets they were going to be compared. In fact the formula from [3] tells us that for a dataset with N elements all unique, the randomness is given by $H(V)/log_2(N)$. Therefore, the second form is only a simplification of the first formula we used.

The datasets have high degrees of random entropy, in average 81.43% (Table 3.4), which indicate that entropy coding will not be very effective and low compression ratios should be expected. The uniqueness varies more and reaches an average of 44.66%, whilst some files barely contain unique values others are almost entirely composed of them. What is interesting is that even the files with low uniqueness are highly random (high randomness%). A notable example of this are the datafiles waterglobe.vel, which are the velocities versus time for a molecular dynamics simulation of a small drop of liquid.

There is a percentage value specifically for zeros because this being scientific data, there is a good chance

Datafiles	Size(MB)	# elements	Unique%	Zeros%	Entropy	Randomness%
waterglobe.arc.txt	1640	172800000	35.45%	0.00%	3.762	99.81%
waterglobe.1col.arc.txt	1640	172800000	35.45%	0.00%	3.670	99.70%
waterglobe.vel.txt	1405	172800000	3.30%	0.00%	3.765	99.89%
waterglobe.1col.vel.txt	1405	172800000	3.30%	0.00%	3.657	99.36%
waterglobe.arc.bin	1318	172800000	35.45%	0.00%	25.614	93.60%
waterglobe.vel.bin	1318	172800000	3.30%	0.00%	21.466	78.44%
engraph1_100.txt	856	85190400	72.88%	0.00%	3.754	99.59%
engraph1_100.1col.txt	856	85190400	72.88%	0.00%	3.666	99.61%
engraph1_100.bin	650	85190400	72.88%	0.00%	25.664	97.42%
gauss09_alpha.txt	304	33500944	28.61%	36.35%	3.611	92.87%
gauss09_density.txt	244	16753366	39.78%	0.05%	3.824	98.34%
gauss09_alpha.bin	256	33500944	28.61%	36.35%	15.662	62.66%
gauss09_density.bin	128	16753366	39.78%	0.05%	22.535	93.90%
msg_bt	254	33298679	92.88%	5.98%	23.667	94.71%
msg_lu	185	24264871	99.18%	0.00%	24.466	99.73%
msg_sp	277	36263232	98.95%	0.00%	25.032	99.68%
msg_sppm	266	34874483	10.24%	11.56%	11.238	44.85%
msg_sweep3d	120	15716403	89.80%	1.73%	23.411	97.93%
num_brain	135	17730000	94.94%	0.00%	23.971	99.55%
num_comet	102	13418496	88.87%	7.73%	22.039	93.08%
num_control	152	19938093	98.52%	0.33%	24.140	99.55%
num_plasma	33	4386200	0.31%	0.00%	13.651	61.87%
obs_error	59	7770102	18.05%	0.00%	17.804	77.78%
obs_info	18	2366316	23.94%	0.00%	18.068	85.33%
obs_spitzer	189	24772608	5.70%	5.29%	17.359	70.67%
obs_temp	38	4991784	100.00%	0.00%	22.251	100.00%
NTUP1_floats.bin	1415	370914252	28.82%	38.70%	15.130	53.15%
NTUP2_floats.bin	1433	375644746	28.70%	38.77%	15.116	53.07%
NTUP3_floats.bin	1435	376256329	28.70%	38.74%	15.123	53.09%
NTUP4_floats.bin	1429	374624469	28.75%	38.76%	15.123	53.10%
NTUP5_floats.bin	1435	376236020	28.72%	38.75%	15.127	53.10%
NTUP1to5_doubles.bin	232	30463714	20.97%	7.51%	7.646	30.76%
NTUP1to5_floats.bin	7148	1873675816	16.10%	38.75%	15.750	51.13%
`AVG (all files)`	860	163954134	44.66%	10.47%	na	81.43%

Table 3.4.: All 33 datafiles statistical metrics and other characteristics. Highlighted in grey are the selected five to represent the datagroups on the results Chapter 4.

that zero values might be very common. This percentage is shown for curiosity only. Its meaning is somewhat irrelevant, as it is only a specific case of one element in the datasets. Taking as an example the NTUP_floats datafiles, out of the 72% of values that are not unique, more than half are zeros.

With this early but quite insightful statistical characterization we can already predict that NTUP datafiles should have the best CR with entropy coders (gzip/pigz). No prediction can be made for the FPC compressor, as that would require knowledge about the smoothness, or data continuity of the datasets, which was not analysed. The overall dataset seems well balanced, with datafiles that cover many possible combinations.

3.3.3. The five selected datafiles

It can be a daunting task to manage the 33 datafiles different results for all of the combinations assessed in this work. While that the tests were performed for every file, the analysis in the coming chapter would not be readable. Accordingly, a selection of five datafiles was made consisting of only one datafile per datagroup, based on its properties and characteristics. The only restriction was that they had to be in binary format, so that it can represent FPC and pFPC. The five datafiles selected to represent their datagroup, highlighted on Table 3.4, are: waterglobe.vel.bin; engraph1_100.bin; gauss09_alpha.bin; msp_sp.bin; NTUP2_floats.bin. The selection criteria is based on the datasets size, choosing the bigger ones will allow compressors to execute for more time, which gives room for more improvements when using the parallel implementations. In the case of waterglobe.vel.bin and NTUP2_floats.bin that are comparable in size to the fellow binary datasets in the group, the selection was made based on the lower entropy/randomness, this time to allow for higher CR. The biggest file NTUP1to5_floats.bin was not chosen because of exactly that, its size belongs to other scale and does not benefit the intended comparison with the other files, e.g. graphics readability severely affected.

3.4. Methodology

To perform the tests in the upcoming Chapter 4 a methodical approach was taken in order to obtain consistent and coherent results. With the compression programs selected and the dataset defined the only planing left before performing the tests is to decide the methodology to follow. A straight forward approach is taken and is shown in Algorithm 1.

For each file there are *nRuns* executions of compression and decompression per compression level, written to the local hard drive disk /local and to /dev/null (i.e. data is discarded). The output destinations were only decided after noticing that the initial tests were being executed through the Network File System (NFS) when writing to the user /home directory, therefore utterly slow. The solution taken, and obvious approach, is to perform the data compression in-node and only then move the files to the final destination. An advantage

Algorithm 1: Perform compression/decompression for a given compressor

```
Data: datafiles, algorithm
Result: output log file with execution times and file sizes
initialization;
for f ← file to lastFile do
    copy(f) → /local;
    for c_lvl ← 1 to maxCompression do
        foreach n in nRuns do compress(f) → /local/f.c_lvl ;
        foreach n in nRuns do compress(f) → /dev/null ;
        foreach n in nRuns do decompress(f.c_lvl) → /local/f.c_lvl-decomp ;
        foreach n in nRuns do decompress(f.c_lvl) → /dev/null ;
        List sizes ← /local;
    end
    Remove files ← /local/{f,f.c_lvl,f.c_lvl-decomp};
end
```

of this is that the traffic in the network can be alleviated because compression may improve the system throughput (less data to transfer), as evaluated in [15]. Instead of only compressing and writing data to disk, the approach of discarding the data is also adopted in order to evaluate execution times differences, by avoiding the timing component of disk I/O. As the authors state in [3] writing to null still consume the data, i.e. the whole compression takes place, just the output component is ignored.

The number of executions is controlled by *nRuns*, originally set to 20 but on the last tests changed to 10. It was realized that 20 measurements were not necessary to get consistent values, hence the reduction. This change effectively cuts the execution time to half for each instance of Algorithm 1. The other variables are *lastFile*, that symbolizes the last file to test for a given list of files (usually a datagroup), and *maxCompression* corresponding to the maximum level of compression for the compressor being tested (e.g. gzip/pigz goes from 1 to maximum 9).

The multi-threaded method simply consists of running the same Algorithm 1 but taking an input variable *nthreads* that represents the number of threads to execute. This value is a parameter for the parallel compressors, such that in each compression/decompression loop the call for the function *compress(f)* or *decompress(f)* receives *nthreads*. Therefore the execution of the algorithm is performed for each specified thread number, i.e. an instance of the algorithm runs for each *nthreads*. The number of threads tested are [1,2:2:24], i.e. one, two, four.. two in two until twenty four, therefore thirteen different tests in total. The maximum number of available threads that are able to execute at the same time on the execution nodes corresponds to the limit tested (remember, 2 cores $\times 12$ threads $= 24$).

3.4.1. Timing measurements

All timing measurements in this work, for the exception of MAFISC and `xz` (lzma) compressor, refer to the walltime reported by the routine omp_get_wtime() from the OpenMP API[10]. We use this in detriment of Unix's `time` command because it has a better granularity (smaller), specially important for the low execution times that are expected from LZ4 with small datafiles. The C/C++ routine omp_get_wtime() returns the elapsed wall clock time in seconds since "some time in the past". This reference is arbitrary and the routines are anticipated to be used on a start point and end point. Thus, the actual wall time if given by the difference of end-start. Consequently it was necessary to add to the compressor's code the OMP routines, once when the algorithm starts and a second time when the algorithm ends. By calculating the difference in both walltimes it gives us the execution time. Because OMP times are reported inside the execution of the program, they do not really encompass the I/O time that remains when the compression ends. Therefore the differences measured between writing to disk or discarding data are relatively small, as opposed to what was expected. If timing is measured using Unix's `time` capturing all the I/O time, which can be forced with the `sync` command[11], the values can increase considerably especially for the low level compressions that terminate faster.

Since we measure *nRuns* executions there are options we can take: average the execution times, use the median, or simply select the best time. We opted for a best time, i.e. lower value, but still use the other values for control. This decision is made upon the fact that every system is different, and a compression algorithm in an ideal case always takes the same time to complete, because there are no stochastic elements. Therefore the lowest time has a real meaning, it tells how capable the algorithm is in terms of speed. Nonetheless this does not imply that the values have discrepancies, in fact they usually stay within 1% to 3% of each other.

This control is made with an algorithm based on K-best scheme by Bryant and O'Hallaron [23], that defines a K number of measurements that need to agree to the fastest within a certain range. There is an e that dictates how close the measurements are required to be (i.e. the agreement range), and an M to define the maximum number of measurements before giving up. In our approach we do not define M, simply because *nRuns* tests are always executed. The error margin is set to 1% ($e = 0.01$) and $K = 3$ (based on an example from [23]), which means that at least three measured execution times, out of the total *nRuns*, should stay within 1% of each other.

The measured times are only analysed after they are all complete. This is done in the parsing stage, briefly described in next subsection, using a python script written for all the measurements parsing. When checking for the lowest execution time out of K-best in *nRuns* our scheme is put to use, and if the K lowest values get

[10] OpenMP is an Application Program Interface that supports multi-platform shared-memory parallel programming in C/C++ and Fortran on all architectures.

[11] `sync` performs a system call that writes all data buffered in memory out to disk

out of range by more than e a warning message is printed. This way if measurements start to become too inconsistent it is possible to intervene and verify the correctness of the outputs, and if necessary re-run the tests.

3.4.2. Job submission and parsing

In order to proceed with the Algorithm 1 execution it needs to be submitted to one of the execution nodes. Without going into much detail, the Algorithm is a bash script where we define the variables related to the tests to perform (files, compressor, et cetera), that is submitted into a queue of jobs managed automatically in the cluster. Each submitted job also has node related properties that we define, in the initialization phase of Algorithm 1, so that the job only runs on the specified nodes. When a job finishes we receive the output log file containing all the measurements.

Parsing the outputs is a methodical work, so a script was developed in python that does the majority of the work. As noted previously it selects the best time out of the best *nRuns* for each compression/decompression test, while looking for outliers using our K-best-like scheme. It also detects if there are missing measurements or file sizes listings. When checking the file sizes a quick verification is performed for matching the original file size and the post compressed-decompressed file. This is not a robust verification but, to a certain degree, can detect if something went wrong with file compression or decompression. All the parsed measurements are then copied manually to spreadsheets for further analysis.

4. Tests, Results and Conclusions

In this chapter we discuss the tests performed and analyse their results in the various sections. The division is made into seven sections, each focusing on one different subject. The core sections are: speedup and efficiency Sec.4.2, compression ratio Sec.4.5 and MAFISC testing Sec.4.7.

4.1. Tests metrics and notes

The goal with thread parallelism is to increase performance, and this can be explored in two ways. First by being able to execute the same amount of work in less time, i.e. performing certain tasks faster for quicker results. Secondly, for the same execution time try to perform the largest number of operations possible, i.e. get more work done in a given time window for more results. While it may seem counter intuitive, because having lower execution times inherently allows for more calculations, the ways of taking advantage of parallelism are subtly different. With file compression the same scenario applies: simply compress a file faster, or get to compress more files spending the same amount of time. Deriving from the intended purpose, there can be slightly different implementations.

In order to assess the possible advantages of using parallelism we take into consideration three metrics. The first metric compares the performance changes from the serial version to the multi-threaded one, and it is referred as speedup. The second metric is the efficiency, and it compares the attained performance gain ratio (i.e. the speedup) with the expected maximum gain. Note that performance changes do not necessarily mean positive gains, as there can also be a loss of performance. The third *metric* is not a single one but a collection of different characteristics that are analysed to give a prospect of the scalability for future challenges, such as compression ratio and memory requirements.

4.1.1. Metrics

The speedup is the main metric used to evaluate the performance gain by using parallelism. It is a ratio given by the execution time of a compression cycle with the serial compressor divided by the same compression executed with the multi-threaded compressor, and is shown in (4.1):

$$Speedup = \frac{exectime_{serial}}{exectime_{parallel}} \implies Sp_t = \frac{T_s}{T_t} \tag{4.1}$$

where T_s is the execution time of the serial version, t is the number of threads used in the multi-threaded program, and T_t is the execution time of the parallel version. As the equation shows the smaller the parallel time, the higher the values of speedup, meaning the program was Sp_t times faster with respect to the serial T_s execution time. Ideally the speedup value is the same as the number of threads used (i.e. linear $Sp_t = t$), but this implies that the measured program is parallelizable in its totality (embarrassingly parallel). Although most of programs have parts that are difficult to parallelize, hence making it almost impossible to have a linear speedup, the compression case appears to be fruitful in this scenario. Because the files can be split into chunks, the compression algorithm can work for an individual chunk of the file, and there can be as many working threads as possible, because there is no data dependency from one chunk to the other. The downside is that by compressing a smaller file chunk, the compression ratio decreases as it becomes harder to find matches in the chunks. This is true for the LZ compressors family, but as other methods exist, which do not resort in dictionary coding, the restriction may not apply.

The efficiency (4.2) is strongly dependent on the speedup because it is a relation of the attained speedup divided by the number of threads that symbolize the theoretical maximum:

$$Efficiency = \frac{Speedup_{parallel}}{number\ of\ threads} \implies Ef_t = \frac{Sp_t}{t} \tag{4.2}$$

where t is the number of threads used, and Sp_t is the measured speedup for that t. The efficiency is a value between zero and one, and it estimates how efficiently the threads were used in the execution of the program. Note that the speedup can have a high value, but it can be originated from a highly inefficient parallelization. It may be preferred to have a 2-threaded execution with efficiency close to one, than a 12-threaded execution with efficiency below 0.5, this means that more than 50% of the computing resources were wasted. With a linear speedup $Sp_t = t$ it means that efficiency will be one, the theoretical maximum $Ef_t = t/t$. Interestingly sometimes it is possible to have super-linear behaviour, due to efficient cache usage per thread. Consequently the measured speedups become greater than the theory value ($Sp_t > t$), and so do the efficiency ($Ef_t > 1$).

4.1.2. Remarks about the tests

In order to calculate the speedups we first measured the compression and decompression times, for the serial algorithms and datafiles using multiple compression parameters as explained in Section 3.2. After that we performed the same compression and decompression tests, but with the multi-threaded implementations.

We performed different runs executed with different number of threads. While that pigz and pFPC allowed to specify the number of threads to run upon execution explicitly with a command line flag, lz4mt did not. As available at the time, it only allowed to switch on or off the multi-threading property, i.e. it would execute with one thread or with the maximum threads it can detect from the hardware it is running on. To be able to perform the same tests with all the compressors it was necessary to slightly modify the lz4mt code to allow thread-number specification. The simple addition of a variable that takes the desired number of threads, and uses it instead of the maximum threads available in the hardware, was enough for the intended purpose.

To give an idea of the sheer amount of tests executed, and the corresponding outputs stored for parsing and analysis, we present below a rough estimation of N_{tests}:

$$
\begin{aligned}
&S = 3\text{serial} = 1\text{gzip} \times 9\text{comp.levels} + 1\text{LZ4} \times 2\text{comp.levels} + 1\text{FPC} \times 26\text{comp.levels} \\
&P = 3\text{parallel} = 1\text{pigz} \times 9\text{comp.levels} + 1\text{lz4mt} \times 2\text{comp.levels} + 1\text{pFPC} \times 24\text{comp.levels} \\
&E = \text{exec. params.} = 2\text{(comp.\& decomp.)} \times 2\text{(local \& null)} \times 10\text{runs} \times 33\text{files}
\end{aligned}
$$

$$N_{tests} \approx (S + P \times 13\text{threads runs (1,2:2:24)}) \times E \tag{4.3}$$

$$N_{tests} \approx (468 + 432 \times 13) \times 1320$$

$$N_{tests} \approx 8030880 \tag{4.4}$$

The number above is an approximation because FPC and pFPC do not work for all the available files, only binaries, the number of runs used is the minimum 10, but most of the tests were executed with 20 runs, and finally, it does not contemplate some more thousands of diverse and failed tests that were executed throughout this work. Nevertheless it still represents the huge number of tests that were performed.

The vast number of outputs produced were reduced one order of magnitude by parsing the multiple runs into the K-best-like values, as covered in Section 3.4.1. A great amount of values were stored but as they do not show any strong point worth of a more thorough analysis, they are mostly disregarded. The two overlooked groups of measured values are the values when output to null, with no real application, and the majority of the decompression times.

Because the decompression tests with the multi-threaded programs present values with small variations compared to the tests with the serial executions (see 4.3 for more details), they are mostly redundant, and consequently are ignored for a considerable part of the results analysis.

Therefore, all the values presented in the coming sections, and unless stated otherwise, are from compression cycle tests with output to disk (the data was written). Most of the figures in this chapter were plot from data of only five files, one for each dataset (see Section 3.3.3). This strategy was necessary to adopt in order to be able to manage the data, five instead of 33 files and all of the measurement values, but mostly to provide good readability of the plots and tables.

4.2. Speedup and efficiency

A simplification is adopted for the speedup in order to keep the complexity of the results as low as possible. With the nodes configuration of two physical processor chips, both with six cores each and Hyper-Threading (HT) active, the theoretical speedup is lower than the resulting 24 threads. Intel states[1] that HT measured a performance gain of 30%, while other sources report better performance gains. It all comes down to the specifics of the problem, and because compression is a data intensive application it makes a good scenario for HT. The 12 physical cores available on the computing nodes execute 24 threads, and while one thread running on a core is stalled for some reason the other thread can kick in and take advantage of the available resources on that core (e.g. data that is already available in the cache, previously fetched by the first thread).

The attained speedup values vary greatly depending on the datafile, the compressor and the compress level. The first step in assessing the resulting speedups of the performed tests, is to compare how the multi-threaded implementation performs with only one thread against the serial version. This comparison allows to measure the possible existence of overhead from using the multi-threaded version as it is expected that a one-thread execution of the parallel programs would perform worse than the serial version. This was the case, but there are quite a few exceptions, and they were found most consistently for pigz (refer to Table 4.2). This phenomenon was discovered only after all the tests were performed on the nodes and, initially, it was speculated that some nodes could be running faster than others. While this actually turned out to be true - two of the nodes are slightly faster than the other four and can run a given test several seconds faster - it was found that pigz seemed to be faster with one thread than gzip. This could be explained from the fact that pigz uses the same algorithms as gzip[2], but the implementation is part of zlib.

Some experiments confirmed that zlib is indeed a little faster than gzip, in the order of some seconds (Table 4.1) depending on the duration of the compression cycle (longer cycle $\rightarrow$ bigger difference). This should be happening because zlib compression/decompression routines use smaller file headers and a quicker integrity check verification, stated in the FAQ[3]. As a result of this, there are phenomena that occurs in the data appearing as *super* speedups and efficiency, i.e. values greater than the theoretical limit (e.g. $Sp_1 > 1, Ef_t > 1$). This is a result of using as a reference the serial version, gzip, that performs worse than the one-threaded

[1] http://software.intel.com/en-us/articles/how-to-determine-the-effectiveness-of-hyper-threading-technology-with-an-application Accessed January 26, 2014

[2] The name of the compression algorithm is known as DEFLATE, and the decompression as Inflate.

[3] http://www.gzip.org/zlib/zlib_faq.html#faq19 Accessed January 26, 2014

	gauss09_alpha.bin			
lvl	gzip		pigz 1thread	
1	3.894	2.124	4.023	1.438
2	3.964	2.104	4.094	1.424
3	4.816	2.231	4.311	1.419
4	5.963	2.212	5.696	1.399
5	6.372	2.173	6.088	1.379
6	7.232	2.174	6.800	1.390
7	8.421	2.170	7.990	1.376
8	25.741	2.148	23.210	1.364
9	55.044	2.111	48.405	1.345

Table 4.1.: The absolute execution times tuples (compression, decompression), in seconds, for gzip and pigz using one thread and the nine compression levels with file gauss09_alpha.bin. Highlighted in blue where the biggest difference occurs.

parallel version, pigz.

Referring to Table 4.2 one can verify that there is variability of the speedup values, and it comes from a diversity of factors. The two intuitive ones are that the speedup changes with each file, and certainly with the different compressors. Although, the most impacting factor in the differences of the speedups, is the compression level (for more pronounced effects on the compressors see Table A.1 in the Appendix). Rseults are presented for both measured output methods, on the left side is writing the data to the disk, and on the right discarding it by directing output to /dev/null. This test has no meaning for the real application of compression, but it could give an insight on the performance gain by not writing to disk storage. The improvement exists for almost all files but it is small (because the sync time is not being measured, see Section 3.4.1), with the exception of pFPC that seems to benefit more from discarding the data. This should be related with the fact that pFPC code is noted as not being optimized in the source code that is distributed (see Section 3.2.2).

	Sp_1 per compressor level, into /local								Sp_1 per compressor level, into /dev/null							
	pigz			lz4mt		pFPC			pigz			lz4mt		pFPC		
dataset	1	6	9	1	9	1	12	24	1	6	9	1	9	1	12	24
waterglobe.arc.bin	0.91	0.90	0.92	1.01	0.99	0.63	0.64	0.84	0.97	0.99	0.99	0.93	1.00	0.89	0.90	0.93
engraph1_100.bin	0.97	1.02	1.03	1.03	1.01	0.63	0.63	0.83	1.08	1.13	1.13	0.95	1.01	0.87	0.86	0.91
gauss09_alpha.bin	0.97	1.05	1.14	0.98	1.01	0.63	0.70	0.85	0.97	1.06	1.14	0.97	1.01	0.87	0.90	0.89
msg_sp	0.94	0.97	1.00	0.99	0.97	0.60	0.60	0.83	0.97	1.02	1.03	0.88	0.99	0.87	0.88	0.92
NTUP2_floats.bin	0.92	0.94	1.10	0.95	1.01	0.64	0.65	0.81	0.96	0.99	1.11	0.96	1.00	0.86	0.87	0.91
`AVG (all files)`	0.94	0.99	1.03	0.97	1.00	0.66	0.67	0.87	0.98	1.01	1.05	0.97	1.00	0.87	0.87	0.91

Table 4.2.: Speedup of all the datafiles for the multi-threaded programs using only one thread. Compression levels are the minimum, medium and maximum. Lz4mt only has two compression levels available, and pFPC do not have a defined maximum so we use 24 as addressed in Sec. 4.4. Note that the average values originate from the entire table, available in the Appendix A.1.

Overall pigz and lz4mt programs perform well with a speedup very close to or above one, meaning that the variance it suffers in performance by running the multi-threaded variants, being degradation or improvement, is very small.

Lz4mt also shows some speedups above one, even if very marginally, which means that its version running with one thread is finishing before the serial version LZ4. A quick test was performed with both algorithms on the same node, and it was determined that indeed lz4mt is faster in some cases (milliseconds). The explanation we find is based on the fact that lz4mt is using an older version of LZ4 inside (r104 versus r109 used for LZ4), which might be providing slightly faster results. Between the two different releases of LZ4, r104 and r109, were committed some changes that may be also affecting the execution times.

pFPC has the worst speedup performance in both data output settings, despite the fact that is also pFPC that appears to take the largest advantage of discarding data. It only achieves, in average for the three levels, a speedup of ≈ 0.72 when output to disk, and ≈ 0.88 when output to null, which translates to 28% and 12% performance degradation respectively.

A comparative chart for the speedups of the parallel compressors is depicted in Figure 4.1. It contains the two plots for each compressor, one on the left with a low compression level, and on the right with a high compression level. The purpose is to compare the gain of compression speedup that is possible to get from higher compression levels. The speedup is expected to grow as the number of threads used increase. This is indeed observable in almost every case for the initial nthreads, most notably on pigz with compression level 9, that yield the highest speedups of this study. On the opposite side of performance gain are lz4mt level 1 and both pFPC, which have the ideal speedup drawn in black to emphasize the low values attained.

Using LZ4 in the fast mode (level 1) is so fast that using multiple threads can actually decrease performance (e.g. 9 out of 13 datafiles in sci_files dataset take more time to compress with lz4mt than with LZ4). This happens while the datafiles are small and the execution times are really low. However, when files are bigger and/or the compression level is increased it leads to longer execution times, and in the same dataset all of the datafiles achieve better compression speedup. For example, the datafile msg_sp has the speedup 0.99 with 24 threads for compression level 1, but achieves speedup 10.71 with same 24 threads and compression level 9 (high).

In the Figure 4.1 pFPC is analysed for compression level 21 instead of 24. This choice has been made because it was observed that the majority of higher speedup values are attained at this level. This differences are analysed further in the document with reference to Figure 4.3. pFPC is the compressor with worst scaling, presenting the lowest speedup values from the first nthreads. The clear best performing compressor is pigz when used with maximum compression, but with speedup increasing slower after 12 threads, also visible from its level 1 plot, and from LZ4 level 9.

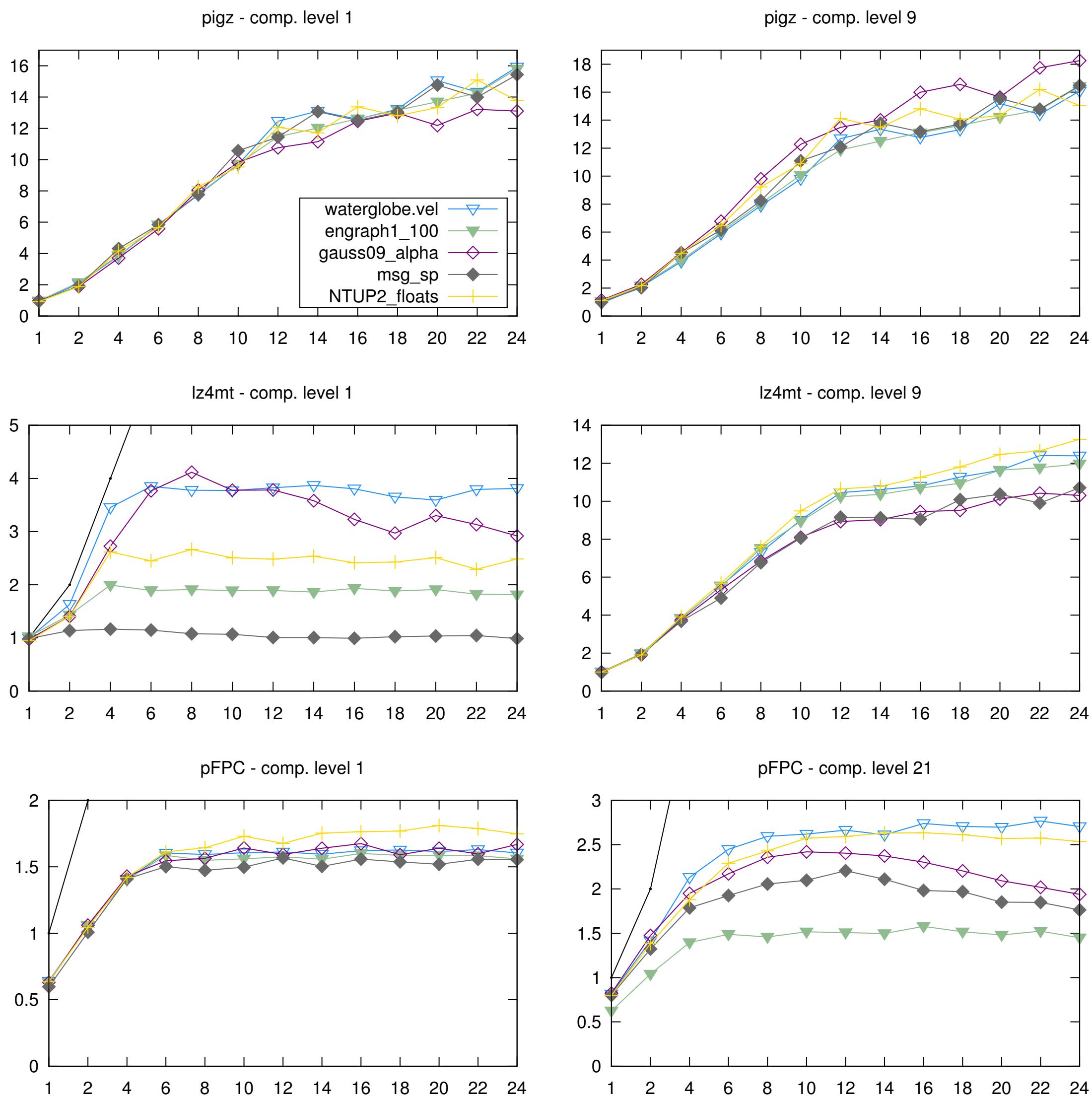

Figure 4.1.: The attained speedups (y axis) for the number of threads used (x axis). The three parallel algorithms are shown with minimal level of compression, on the left, and high level on the right. pFPC is shown with level 21 instead of the maximum 24.

Figure 4.2 represents the datasets uncompressed sizes plotted in order of their speedup for the three parallel compressors using 24 threads and maximum compression level. A fourth plot, lower right, gathers the best speedup for each file, independently from the number of threads or algorithm. What happens is that all the best speedups are coming from pigz, with the subtle difference that for seven files the speedups are achieved one with 20 threads and the remaining six with 22 threads, instead of the maximum 24. Thus, the first and fourth graphics are very similar, with basically NTUPs files having slightly higher speedup. The

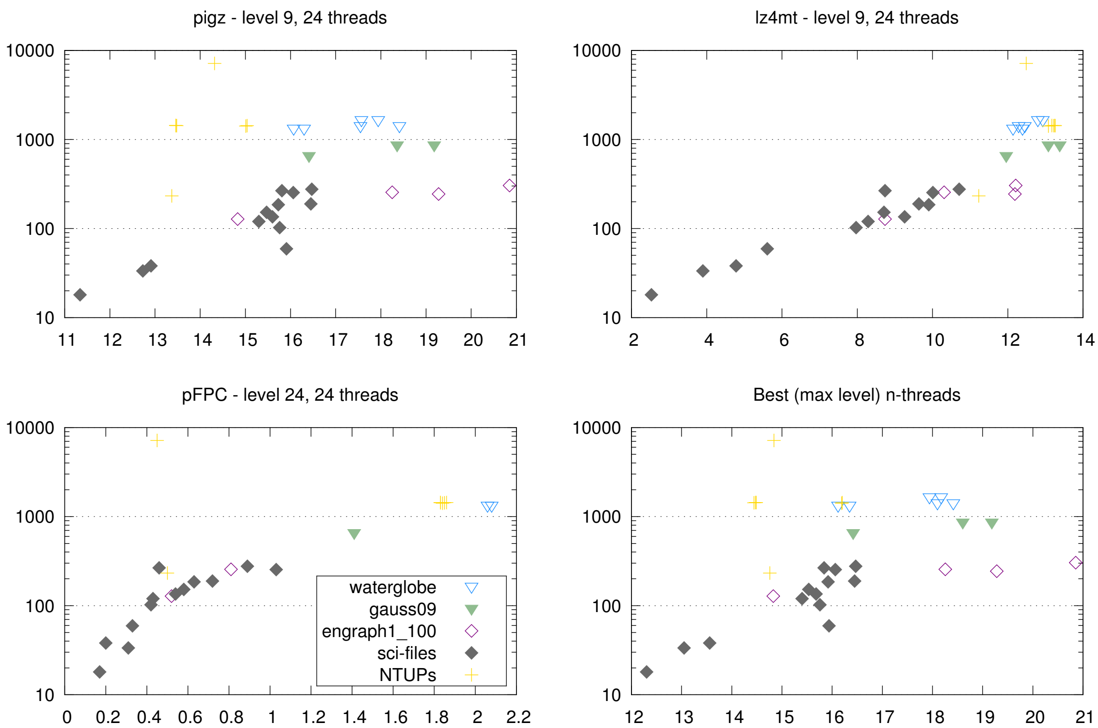

Figure 4.2.: Ratio of the maximum speedup per individual file for all the datafiles, with their uncompressed sizes. In the x axis are the attained speedups and on the y axis (logarithmic) are the file sizes in bytes.

second (top right) and third (bottom left) plots represent lz4mt and pFPC respectively. While seeming similar, they tell different stories. lz4mt has a very clean trend line, achieving higher speedups with bigger input files. On the other hand pFPC shows speedups lower than one (as low as 0.2), effectively meaning that it needed more time to finish execution than its serial version FPC. Nonetheless, the behaviour is the same but with a different connotation, as smaller files suffered a stronger negative impact on the speedup than the larger files (i.e. bigger files still perform better). One of the NTUPs files (NTUP1to5_floats.bin) is the larger in the tested datasets, with about 7.1GBytes. According to the observed pattern it would be the file to provide the higher speedup. In fact this does not happen, and it falls behind in the speedup "race" compared with other smaller files. It is one of the exceptions in the general panorama, however, these representations show a reasonably clear pattern, higher speedups come from bigger files overall.

Depicted in Figure 4.3 is the speedup relatively to the compression level, in order to assess the scalability of the algorithm when increasing the level of compression. Only 12 and 24 threads are analysed because we want to show the best performance, with 24 threads providing the maximum. The 12 threads speedups are used because it does not imply the use of the HT technology, thus it may suggest to be a better trade-off using

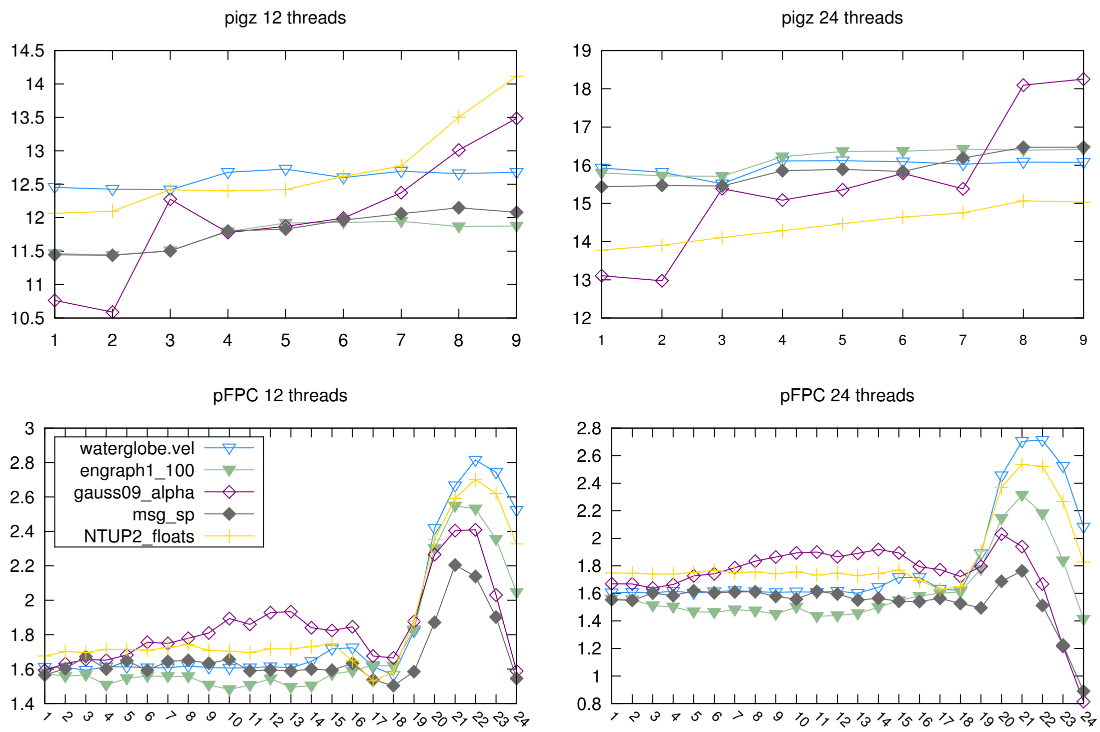

Figure 4.3.: The speedup relative to the compression level for pigz (two top plots) and pFPC (two bottom plots). In the x axis is the compression level and in the y axis is the attained speedup at that level, for the given datafile.

only 12 threads and getting a speedup with better efficiency. lz4mt is not represented because as it only has two compression levels, there is no behaviour other than having higher speedups with the high compression level. For both 12 and 24 threads lz4mt present positive slope lines from level 1 straight to level 9, i.e. straight lines going up from left to right.

Focusing on pigz plots one can establish the connection that overall higher compression levels provide the best speedups. The top left plot (12 threads) evidences that pigz was faster with one thread than gzip, because the speedups consistently surpass the theoretic limit of 12 (on y axis), achieving a fictitious super-linear status. The main difference of using 12 or 24 threads is that for one, with 24 threads, it is not possible to achieve the theoretical maximum speedup (i.e. 24); second, with 12 threads the speedups only achieve higher values with higher compression levels, while that with 24 threads the speedup values are higher from the first compression levels; third, with 24 threads the speedups are more constant with the exception of gauss09_alpha.bin that has a jump from compression level 2 to 3 and then again from 7 to 8. The same does not happen with pFPC, that shows a relatively big increase and then a drop with higher compression levels. The behaviour is the same with 12 or 24 threads, with the nuance that the speedups of two files drop below one with 24 threads, with maximum compression settings. For pFPC, and with this sample files, we can affirm that 24 threads did not pose any improvement and that compression level 21 seems the one that yields higher speedups.

When it comes to efficiency there is a repercussion originating from the simplification made to the speedup, discussed in the beginning of this section. As the speedup values are simplified for analysis, the resulting efficiency is affected and it also becomes an approximation. The values resulting from $Ef_t = Sp_t/t$ will indicate an expected efficiency value that is very difficult to achieve with 24 threads, i.e. with HT enabled. With half of the threads, as same number as physical cores available, we assume that each thread runs on a different core, as if there is no HT, thus it should be possible to obtain a $Ef = 1$ when $Sp = 12$. From Table 4.3 one can verify that indeed this speedup is plausible when using 12 threads, hence a efficiency of one (100%) is achieved.

	pigz - lvl 9			lz4mt - lvl 9			pFPC - lvl 24		
dataset	1th	12th	24th	1th	12th	24th	1th	12th	24th
waterglobe.vel.bin	0.92	12.68	16.07	1.02	10.45	12.39	0.84	2.52	2.08
engraph1_100.bin	1.03	11.88	16.41	1.01	10.24	11.96	0.83	2.05	1.41
gauss09_alpha.bin	1.14	13.49	18.25	1.01	8.93	10.31	0.85	1.59	0.81
msg_sp	1.00	12.08	16.47	0.97	9.16	10.71	0.83	1.55	0.89
NTUP2_floats.bin	1.10	14.12	15.04	1.01	10.65	13.26	0.81	2.33	1.83
`AVG (all files)`	1.03	12.34	**15.90**	1.00	9.22	**10.50**	0.87	**1.48**	0.95

Table 4.3.: Speedup of the five datafiles for the multi-threaded programs using 1, 12 and 24 threads. The maximum compression levels are used and correspond to 9 for both pigz and lz4mt, and 24 for pFPC. Note that the average values originate from the full table, available in the Appendix A.2.

The speedup efficiency Ef_t is presented in Figure 4.4, where the three parallel programs are represented with the correspondent number of threads t used. As with previous analysis, because pigz was faster with 1 thread than gzip, there are values over the theoretical top, in this case super-efficiencies above 1 (100%), marked with a dashed line. Nevertheless, an observation worth noting is that there is a somewhat clear reduction of efficiency when using more than 12 threads, presumably caused by the HT, in the pigz top left plot. The other two compressors show correct values under the maximum efficiency line. On one hand there is lz4mt (top right) that presents a linear drop in efficiency as more threads are used, while also showing a more aggressive decrease for three of the five files, after the 12 threads mark. On the other hand is pFPC with a logarithmic decay reaching efficiencies lower than 10% with more than 16 threads. pigz shows the best efficiencies all around, with values between 60% and 80% with 24 threads, while that lz4mt provides values between 30% and 60%. With 12 threads pigz accounts for the ”super-charged” efficiency of 100% to 120%, while that lz4mt reaches between 65% and 90%, depending on the file.

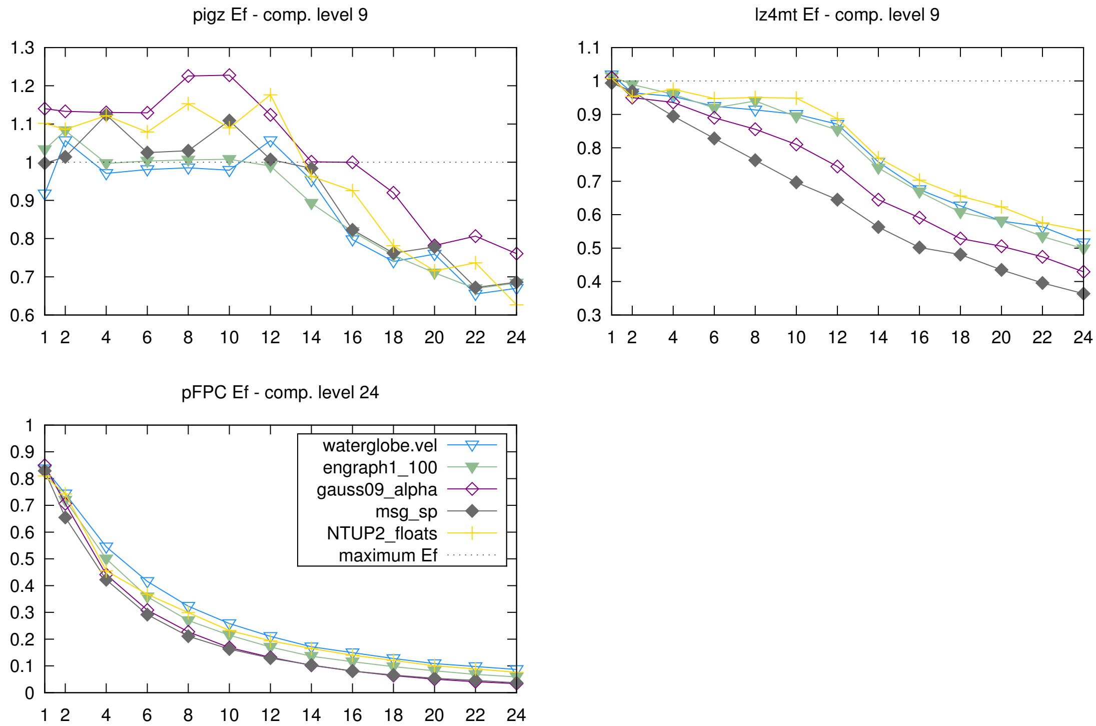

Figure 4.4.: The speedup efficiency for pigz and lz4mt with compression level 9, and pFPC level 24, using one file per dataset. In the x axis are the number of threads used, with the correspondent efficiency on y axis.

4.3. Decompression

Decompression has a different behaviour from compression, being much faster and having limited or no parallelism to explore. This happens because the process is inherently much simpler and faster for the LZ family, and is independent from the compression level used. When decompressing, the algorithm only needs to reconstruct the data following a set of steps.

Because the parallel decompression is not as easy to exploit as parallel compression, the compressors present limited parallel decompression performance. pFPC takes the same approach as during compression, it chunks the data and assigns them to threads. When decompressing it automatically uses the same amount of threads and chunks that were used during compression, which fits with the symmetric properties of the algorithm. pigz seem to behave differently, because it spawn some extra threads but not as much as specified nthreads for execution. The pigz documentation states that *"decompression can not be parallelized, at least not without specially prepared deflate streams for that purpose. As a result, pigz uses a single thread (the main thread) for decompression, but will create three other threads for reading, writing, and check calculation, which can speed up decompression under some circumstances."*. In fact we were able to verify the existence of four threads during decompression, even if more are specified. The same does not happen with lz4mt, because it spawns the specified nthreads, even during decompression. It is not clear at this point if all the threads do useful work, or if internally lz4mt takes a similar approach to pigz. Nonetheless, the performance differences are very small for both. Due to the fact of parallel decompression not providing any real advantage, we do not perform speedup and efficiency analysis for decompression.

It is still worth noting some realities about decompression. The nice property that LZ algorithms afford by having the decompression independent from the compression parameters is represented by the top plots in Figure 4.5, showing very steady decompression execution times. The same does not apply to FPC due to its internal mechanics. Both gzip and LZ4 plots look similar, with the difference that LZ4 only has two compression levels and does not allow to see the small variation with initial levels. These variations represent a small decrease in decompression times, accompanied by the expected increase in compression times when higher compression levels are used. It arises from the fact that when decompressing files that had been compressed with higher levels, the files are naturally smaller and therefore less data is read. Other important point is that at higher levels, or at least after a certain level "threshold" (dependent on each file contents), the compressing algorithm produce less but longer matches, leading to faster reassembly of the data, i.e. the decompression stage. For gzip this seems to happen after level 4 or 5, especially for waterglobe datagroup (the down slope on the waterglobe lines).

The fact that FPC operates in a symmetric fashion makes it perform slower on decompression, as shown in bottom plot in Figure 4.5. As the FPC algorithm needs to refill prediction tables and calculate the same `xor` operations as in the compression cycle, it performs slower when decompressing data that was compressed with bigger prediction tables (i.e. higher compression settings). It happens because when decompressing FPC has to write more data to the disk spending more time in output, but taking the same time with the computations, hence making the overall decompression process take more time to complete. FPC presents an approximately linear behaviour, thus with larger compression levels come higher compression and decompression times, deriving from the symmetric nature of the algorithm.

LZ4 decompression is specially fast as it can yield throughputs in the order of GB/s, possibly achieving RAM-speed limits on some platforms. There is a significant difference in decompression times when LZ4/lz4mt decompress to null (not shown). In this case the decompression times can perform twice as fast or more, derived from the fact that it already is extremely fast and the data is discarded. This is an unexpected scenario because gzip/pigz and FPC/pFPC do not show to benefit from output to null as LZ4/lz4mt do with some datafiles. Considering that the real output time is not measured (only with `sync` we could force that), we find no explanation of why LZ4 and lz4mt have this advantage when decompressing to null (code analysis could help in this situation). Consequently, with these measured low decompression times LZ4/lz4mt can easily achieve a throughput of more than 2GB/s when output to null.

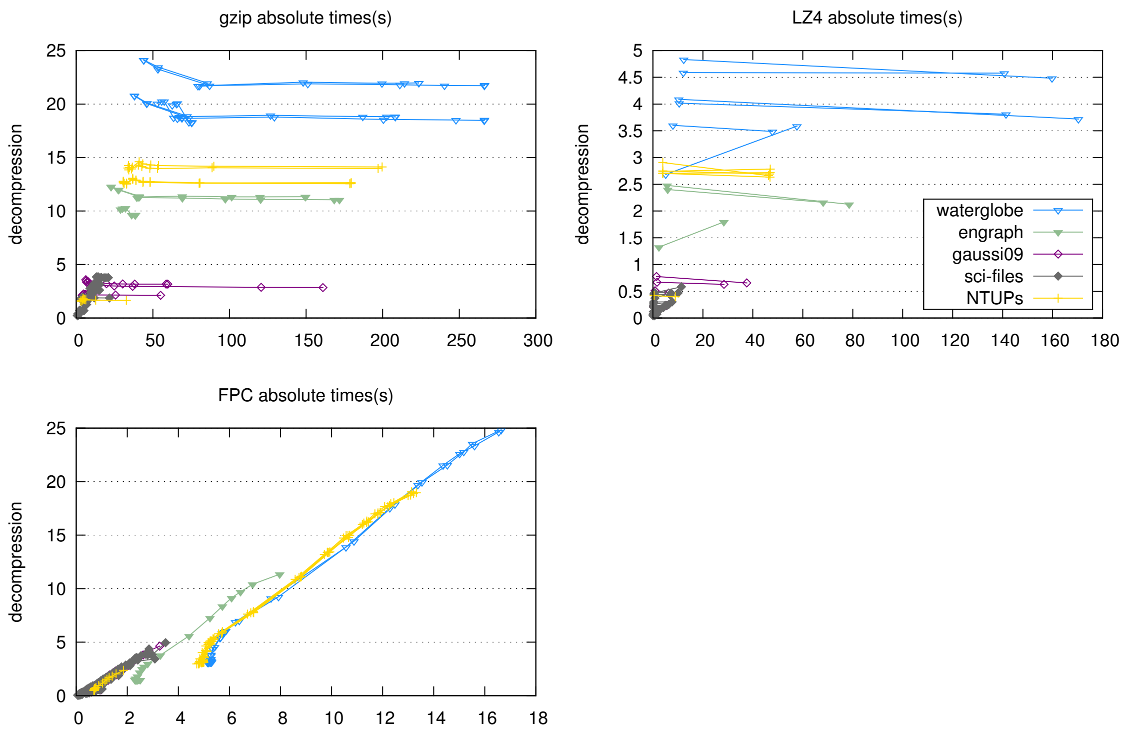

Figure 4.5.: Absolute execution times of decompression (y axis) in order of absolute execution times of compression (x axis), for all the datafiles and the three serial compressors. For each file the points represents the consecutive compression levels, hence 9 points for gzip, 2 for LZ4 and 26 for FPC.

Our biggest datafile (NTUP1to5_floats.bin) was hidden from the plots because as it takes much longer time to compress and decompress across all compressors, it would extend both axis ranges and severely affect the readability.

4.4. Memory requirements

Nowadays RAM memory is a resource usually available in large quantities within computing clusters. Nevertheless, it actually became a problem when using pFPC in our tests nodes (with a somewhat "limited" 12G of RAM). The problem originates from the way pFPC works, as it allocates a table with 2^{n+4} bytes for each thread, with n being passed as a parameter to the program.

For the tests n was selected within an entire integer range of [1:26], because in [3] the authors of FPC tested it with 25, so it only seemed interesting to take it one step further. This range worked without problems when used the serial version (FPC), but its memory requirements increased exponentially in pFPC (Figure 4.6). This means that while 2^{26+4} bytes are used for one thread, which represents 1GB of memory, 24 threads require twenty four times that (24GB), but the nodes selected to run the tests have only 12GB of memory installed. The schematic depicted in Figure 4.6 represents this limit with a vertical line, meaning that beyond that point the performance is severely affected as swap memory kicks in, until there is no more virtual memory and the program terminates. In orange and red are all the number of threads that cross the limit with compression level 26.

To stay within the limit n is decremented, and with $n = 25$ only the 24 threads tests (the red one) surpass the memory limit, so it is necessary to decrement one more to $n = 24$. Now we are within the limit, which allow for the use of all available threads $24 \times 2^{24+4} = 6$GBytes. It should be pointed out that tests were still performed with $n = 25, 26$ for all the threads possible, i.e. within the node memory limit, and that it offers better CR for 12 out of the 25 binary files tested in FPC and pFPC.

The other parallel compressors, pigz and lz4mt, do use more memory than their serial versions, estimating $Parallel_{mem} = \#thr \times Serial_{mem}$ where $\#thr$ is the number of threads used. Since they derive from the LZ family, the requirements are much lower than pFPC. This values were measured in the form of reserved memory only using the `top`[4] program. The observed values for pigz are around 10MB with 12 threads and 18.5MB with 24 threads. This values agree with the estimation, knowing that gzip uses less than 1MB (measured 800KB). lz4mt reserves about 100MB and 196MB with 12 and 24 threads respectively, and measuring 8MB in LZ4 it does stays close to the estimation. A summary of this values is presented in Table 4.4.

	Algorithms – Comp. Level		
#thr	pigz – 9	lz4mt – 9	pFPC - 24
1	1.3	10	279
12	10	100	3017
24	18.5	196	6017
Serial	0.8	8	263
Est. 12thr	9.6	96	3156
Est. 24thr	19.2	192	6312

Table 4.4.: The memory usage measured for compression of the three parallel algorithms using high compression level. Bottom rows show the memory measured for the single-threaded programs, and the estimation values to expect from 12 and 24 threads. All memory quantities are in MB.

For decompression the memory requirements are 100-400KB lower with the gzip/pigz and LZ4/lz4mt compressors. FPC and pFPC still require roughly the same memory because it is needed to refill prediction tables. This implies that decompressing a file that is compressed with different levels will require different quantities of memory for decompression. Because pFPC decompresses with the same number of threads as it was used in compression, it will require the same amount of memory, e.g. decompressing a file that

[4]`top` is an Unix program that provides a dynamic real-time view of a running system.

was compressed with level 24 and 12 threads will yield a RAM usage of 3GB ($12 \times 2^{24+4}$) just like during compression.

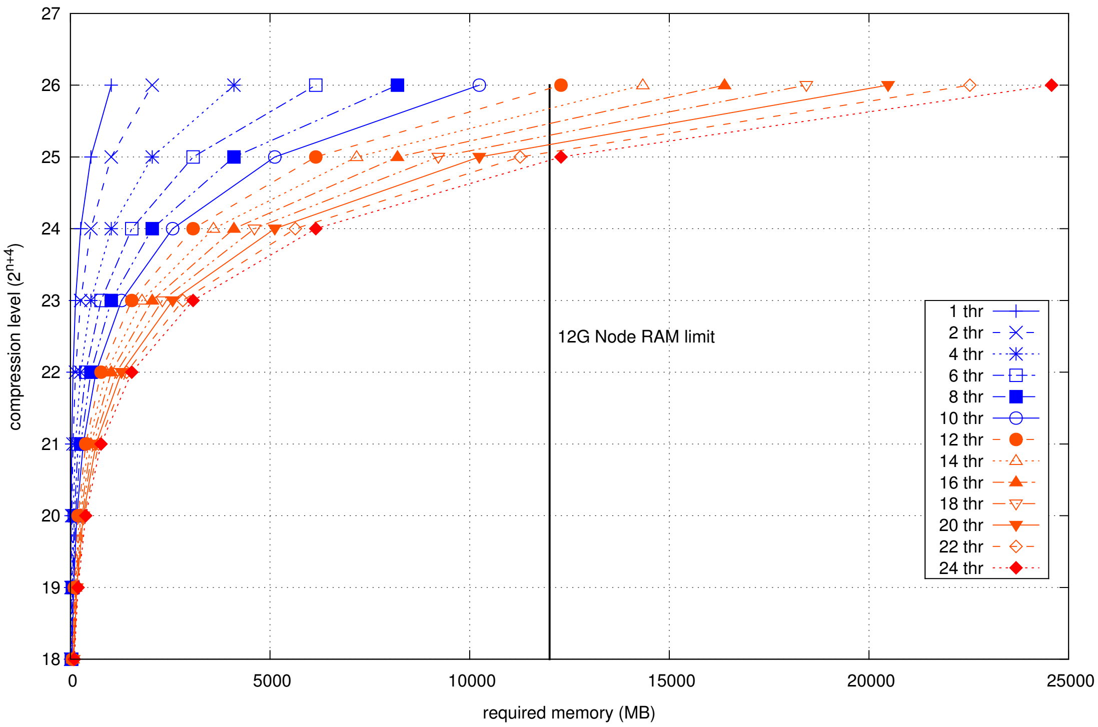

Figure 4.6.: RAM memory required for pFPC corresponding to the amount of threads. The vertical line represents the node memory limit.

4.5. Compression ratio

The three compressors we assess have different objectives and properties. While gzip is one of the most common compressors used (virtually every open-source software package is distributed with a *gzipped* option), it also has a a very good balance between compression and execution time. LZ4 offers a mode that is super-fast, potentially RAM-speed bound while decompressing, but expectedly looses compression capabilities. FPC, designed to compress binary floating-point scientific data, is not meant to do dictionary and entropy coding as the LZ based compressors, which leads to a possible good balance between speed and compression ratio (CR), but falls short on the usability as it is not general. This, however, should not be a big problem for the scientific community as floats are the preferred data type used.

Interestingly when compressing with pFPC, and it is the case that CR is lower than FPC for a great number of files. A strong example is the file num_plasma.bin that deliver a CR of 15 when compressed with FPC level 24, but only 6.6 with pFPC level 24 with 24 threads executed. This file is very small with around 33MB uncompressed, and when compressed it shrinks to 2.2MB with FPC and 5MB with pFPC, not a big difference in absolute terms but relatively it is more than twice its compressed size on the most effective form. pFPC assigns each thread with chunks to compress (8192 was the elected chunk size, see Section 3.2.2), and as more threads are used they will only compress certain parts of the data for the input datafile. Depending on the dimensionality (e.g. number of variables) of the data, the threads can end up getting the values from the same dimension (variable) as they process the file. Therefore, it will affect the predictions and CR for the best if the same dimension ends up with same thread, or worse if the threads get chunks from different dimensions.

When it comes to the variability of compression ratio there are some unexpected events with pFPC. The other two pairs of algorithms, gzip/pigz and LZ4/lz4mt, present the expected behaviour. The CR is kept exactly the same between LZ4 and lz4mt, while that between gzip and pigz is varies very little. With some files gzip has higher CR, while with others it is pigz who has the higher CR, although the differences are very small. Comparing to gzip, in 19 out of 33 files pigz shows a reduction of CR, while that in the remaining 14 files it shows an increase. These variations of the CR represent absolute values of less than 3MB (e.g. with NTUP2_floats.bin pigz compresses around 1.2MB more than gzip).

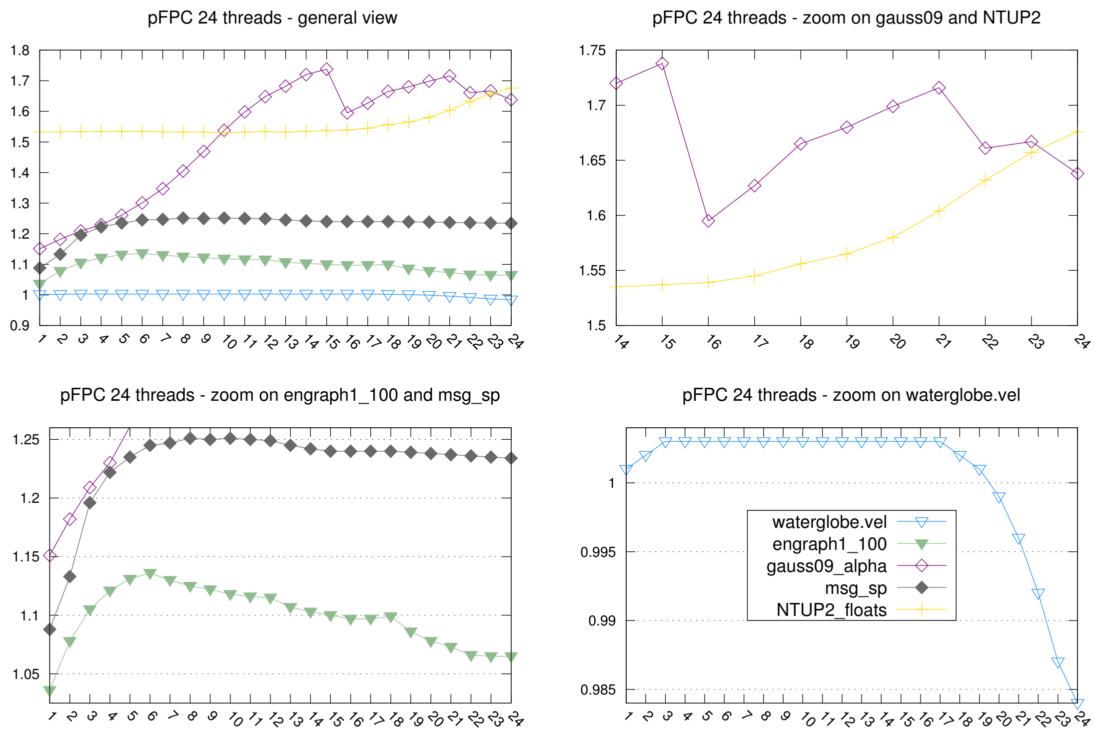

Figure 4.7.: CR (y axis) relatively to the compression level used (x axis) in pFPC. The five selected files are used as a sample, nevertheless this behaviour naturally happens on other datafiles.

Presented in Figure 4.7 are the CR of the files, using pFPC, relatively to the compression level used. One can immediately spot an unusual drop, and recovery, of CR with the file gauss09_alpha on the first plot. Besides this eye-catching event are other uncommon behaviours taken into a closer look, shown on the rest of the plots in the figure. The pattern that appears with gauss09_alpha (alpha) repeats itself, much more subtly, with engraph1_100 (engraph) on the bottom left plot. Both of them show a decrease in compression ratio, from 15 to 16 on alpha and 6 to 7 on engraph, which then start to recover with higher levels. With the alpha file it happens abruptly, as it drops down it goes back up with the next level, while that with engraph it takes more compression levels to expose the variation. The inversion points happen at 6 to 7, then 12 to 13 and then 18 to 19, exactly six levels between each other (can be related with data dimensionality). The case of waterglobe.vel and NTUP2_floats file is different, because there is no recurring changes, it takes one direction and apparently sticks to it. With waterglobe.vel.bin the CR line starts to decrease after compression level 17, while that with NTUP2 it starts to increase after level 14. All of these events depend on the file itself and the compression level of FPC/pFPC, due to the fact that these algorithms resort to predictors. The predicted values vary with each compress level, thus giving a chance to detect this behaviours. Summing this events it is clear that a higher compression level in FPC/pFPC does not seem to yield higher compression ratios, as opposed to what general compressors usually do.

4.5.1. The best CR and speedup values

The best CR and best speedup measured values are presented on Table 4.5, and are independent from each other as we only looked for the highest yield. We summarize all the best CR on the left three columns and all the best speedups on the rightmost four columns. For both metrics there is column with its value, other with the properties or the compressors properties that originated it, as well as a third column with the throughput(MB/s) for that value of compression or speedup. The properties consist of the name of the compressor, the number of threads used (1 is shown for serial compressors) and the compression level used. The speedup values have an extra fourth column that presents the associated parallel efficiency of the best attained speedup. As one can verify, the serial algorithms have the best CR, with the exception of NTUPs that are best compressed with pigz, which is also the best achieving speedups compressor. This is to be expected because gzip is the the compressor with the longest execution times, hence giving pigz a better chance to improve. The best compression ratios come mostly from highest compression levels, which is also expected. Nonetheless, for 10 datafiles the best CR is achieved before highest compression level is used.

If the Table 4.5 was assembled with the purpose to show other metrics best values, specifically throughput, it would be populated with mostly LZ4/lz4mt and FPC/pFPC.

datafiles	Val(ratio); properties; Throughput(MB/s) Best - CR			Best – Sp_t			Ef_t
waterglobe.arc.txt	2.14	[gzip 1 8]	7.3	17.94	[pigz 24 9]	131.6	0.75
waterglobe.1col.arc.txt	2.20	[gzip 1 8]	6.2	18.17	[pigz 24 8]	111.9	0.76
waterglobe.vel.txt	2.19	[gzip 1 8]	6.7	18.10	[pigz 24 8]	122.1	0.75
waterglobe.1col.vel.txt	2.27	[gzip 1 8]	5.3	18.41	[pigz 24 9]	97.1	0.77
waterglobe.arc.bin	1.20	[gzip 1 3]	20.0	16.35	[pigz 24 8]	286.0	0.68
waterglobe.vel.bin	1.48	[gzip 1 5]	19.1	16.12	[pigz 24 5]	307.6	0.67
engraph1_100.txt	2.33	[gzip 1 9]	5.7	18.60	[pigz 24 8]	106.5	0.78
engraph1_100.1col.txt	2.44	[gzip 1 9]	5.0	19.18	[pigz 24 9]	95.5	0.80
engraph1_100.bin	1.22	[gzip 1 3]	20.3	16.42	[pigz 24 7]	276.4	0.68
gauss09_alpha.txt	4.37	[gzip 1 9]	1.9	20.85	[pigz 24 9]	39.4	0.87
gauss09_density.txt	2.36	[gzip 1 9]	4.1	19.28	[pigz 24 9]	78.6	0.80
gauss09_alpha.bin	3.87	[gzip 1 9]	4.6	18.25	[pigz 24 9]	84.7	0.76
gauss09_density.bin	1.09	[FPC 1 24]	82.8	14.83	[pigz 24 9]	294.5	0.62
msg_bt	1.29	[FPC 1 24]	82.4	16.06	[pigz 24 9]	263.5	0.67
msg_lu	1.17	[FPC 1 20]	173.7	15.92	[pigz 24 7]	279.4	0.66
msg_sp	1.26	[FPC 1 24]	116.7	16.47	[pigz 24 9]	217.9	0.69
msg_sppm	5.30	[FPC 1 19]	314.8	15.84	[pigz 24 8]	360.0	0.66
msg_sweep3d	3.09	[FPC 1 24]	166.3	15.40	[pigz 24 7]	272.4	0.64
num_brain	1.16	[FPC 1 24]	96.0	15.68	[pigz 24 5]	263.2	0.65
num_comet	1.16	[FPC 1 24]	88.7	15.76	[pigz 24 9]	236.3	0.66
num_control	1.16	[gzip 1 9]	18.0	15.53	[pigz 24 7]	280.0	0.65
num_plasma	15.00	[FPC 1 24]	127.3	13.05	[pigz 24 5]	322.3	0.54
obs_error	3.54	[FPC 1 24]	91.4	15.94	[pigz 24 8]	193.2	0.66
obs_info	2.27	[FPC 1 24]	65.7	12.30	[pigz 20 7]	232.8	0.62
obs_spitzer	1.23	[gzip 1 3]	18.0	16.45	[pigz 24 9]	203.7	0.69
obs_temp	1.04	[gzip 1 4]	18.3	13.56	[pigz 24 6]	247.7	0.56
NTUP1_floats.bin	2.19	[pigz 1to24 9]	107.8	16.19	[pigz 22 9]	116.2	0.74
NTUP2_floats.bin	2.19	[pigz 1to24 9]	107.9	16.20	[pigz 22 9]	116.3	0.74
NTUP3_floats.bin	2.19	[pigz 1to24 9]	107.6	14.47	[pigz 22 8]	257.2	0.66
NTUP4_floats.bin	2.19	[pigz 1to24 9]	107.8	14.44	[pigz 22 8]	257.1	0.66
NTUP5_floats.bin	2.19	[pigz 1to24 9]	107.7	14.49	[pigz 22 8]	257.6	0.66
NTUP1to5_doubles.bin	4.27	[pigz 1to24 9]	94.9	14.76	[pigz 22 3]	884.7	0.67
NTUP1to5_floats.bin	2.19	[pigz 1to24 9]	114.4	14.84	[pigz 24 8]	263.8	0.62

Table 4.5.: Summary of the all-best values for each datafile. The third and sixth columns contain the properties, enclosed in square braces, for the best values. These properties are composed of algorithm, number of threads and compression level (in this order).

4.6. Blocks (split datafiles) versus entire-file

At some point during the realization of all the tests a question arose to us, should the datafiles be a single file within manageable limits, or should be split into smaller parts. Looking for an answer a few experiments were performed by splitting two files into smaller parts, and after initial assessment, proceed with more splitting with only one of the files (the most promising, if any). Tests were performed only for compression with the serial algorithms. The two selected datafiles are binary, so it is possible to test all algorithms, and consists of waterglobe.arc.bin with 1.3GB and NTUP1_floats.bin with 1.4GB. We selected the first NTUP file arbitrarily, as all of them are similar on size and properties. The largest file from our datasets is NTUP1to5_float.bin and consists of the five NTUP files concatenated together, hence would not make sense to use it for this purpose. The waterglobe binary files are the second largest in the datasets, and it was also simply selected once more arbitrarily (waterglobe.arc.bin comes before than waterglobe.vel.bin alphabetically).

What was observed in the initial assessment is that, with a split into four parts of waterglobe.arc.bin, the sum of those parts usually take some milliseconds less to complete but compress fewer 5K with gzip, 13KB with lz4mt and 6MB with FPC (i.e. the sum of the split files have more bytes after compression). For NTUP1_floats.bin split tests, gzip and LZ4 showed a very small increase in compression ratio for maximum level on NTUP1 (around 7-10KB less on compressed file), but FPC managed to compress 23MB more, with around 0.2sec of increased compression/decompression execution time. It only happens with NTUP1, as with waterglobe.arc.bin the compression gets consistently worse, when increasing the compression level, with the splitting. The second stage of this analysis now focus only FPC and NTUP1 datafile, which was split into 2, 4, 8, 16, 32 and 64 parts. This means that in the smallest split the algorithm compress a $1/64$ of the file (≈22MB) 64 times instead of 1.4GB once.

	FPC – compression level 24		
#parts	C-time diff (s)	D-time diff (s)	Size diff (MB)
2	-0.102	-0.230	-9.02
4	0.111	-0.062	-21.97
8	0.715	0.635	-13.08
16	1.678	1.538	-11.60
32	3.674	3.554	-19.04
64	6.332	6.090	-58.32

Table 4.6.: Differences summary table of using file splitting, in the concrete case of NTUP1_floats.bin with FPC. C-time is the compression time and D-time the decompression, with blue cells representing the gain.

The outcome is summarized in Table 4.6, in the form of absolute differences between the sum of splits (sum of the times and sizes of the split files) and the one file originals. Apparently, in this test case, FPC takes advantage of using smaller inputs, saving 58MB when the file was split into 64 pieces. pFPC, which chunks the data for the threads, also compresses more, saving 46MB with 24 threads and same compression level. Using split into 64 parts takes relatively more time to complete, around six seconds for both compression and decompression. This is expected because of the overhead coming from executing the program 64 times independently. In this case pFPC has the advantage, as it is faster than the serial version (but with a miserable speedup of approximately 2 using the 24 threads)

With these observations it suggests that the little compression gain (58MB in a 1.4GB file) of split is not worth the increase in execution times and the the increased complexity in managing more files. Splitting into two, or especially four pieces, looks much more reasonable, at least for this particular example.

4.7. MAFISC Pre-filtering for compression

When researching the state of the art in compression of scientific data we read about MAFISC, a compressor that applies filters to the data before actually compressing it. This trade-off yields better compression ratio for the cost of spending extra computing cycles to apply the filters. It consist of data reorganization (lower entropy) that boosts the compressibility.

Interested by this approach we decided to test it, and for doing that we used HDF5[5] (hdf5). MAFISC is implemented as a filter that plugs-in to HDF5 own high-level programs, and can be selected as a compression method for the hdf5 format files. Two big binary datafiles were used, the NTUP1_floats.bin file (converted to the hdf5 format and keeping its 1.4GB size), for the same reason as in previous section, and a fusion of all sci-files into a bigger file (2G). This fusion was made with the features of HDF5 allowing for all the files in the sci-files datagroup to be structured inside a unique file. When importing a file to hdf5 format it is possible to specify a chunk size, which defaults to 32MB. With some testing using chunk sizes equivalent to the CPU's L2 (1.5MB) and L3 (12MB) caches it was selected for comparison a size of L2 for showing better performance.

There is a bug in hdf5[6], unresolved at the time of writing, that inhibits decompression testing because h5repack fails to load the filter plug-in, in this case MAFISC, and fails to decompress. h5repack is a high-level program that comes with the library and allows to repack files in the hdf5 format using a filter, being the filter already integrated or user defined (the case of MAFISC). To compress a file with one of integrated compressors we simply use h5repack and use the filter code for it, for MAFISC it is exactly the same but the filter identification code is different.

The tests performed with MAFISC are all executed serially as hdf5 still does not offer multi-threading ability. We measure compression time and compressed file size with the standard MAFISC settings, and repeat with a modification on the filter. This modification simply consists of modifying a line of MAFISC source code, so that it uses the specified lzma compression level instead of the default. Regarding the default configuration, it uses a specific fast option of LZMA with parameters defined manually, which probably have been tuned by the authors.

Comparison with lzma takes place, using the program `xz`, and also with a LZ4 filter for hdf5 that is available on public domain. Table 4.7 summarizes the gains, in percentage, of the compression time and compressed file size relatively to a compression directly using gzip with level 9 (i.e. not using hdf5 gzip filter). Therefore we are looking for negative percentages, meaning that the execution time took less t% time to complete, or the file size had less s% of the size, compared to using gzip.

[5] HDF5 (Hierarchical Data Format) is a data model, library, and file format for storing and managing data. It supports an unlimited variety of datatypes, and is designed for flexible and efficient I/O and for high volume and complex data.

[6] http://hdf-forum.184993.n3.nabble.com/HDF-Newsletter-137-td4026685.html Accessed January 26, 2014

		NTUP1		fusion-sci_files	
		chunk-L2	default-chunk	chunk-L2	default-chunk
comp. time	MAFISC	**114.96%**	183.63%	**419.15%**	627.46%
	MAFISC mod. -9	489.52%	587.74%	1080.62%	1313.30%
	xz(lzma) -9	304.30%	306.00%	878.85%	889.52%
	hdf5-LZ4 -9	-72.81%	-74.58%	-32.38%	-37.32%
comp. size	MAFISC size	**-8.18%**	-2.71%	**-21.06%**	-19.09%
	MAFISC mod. -9 size	-10.86%	-6.58%	-23.01%	-20.36%
	xz(lzma) -9 size	-12.78%	-12.74%	-29.08%	-29.02%
	hdf5-LZ4 -9 size	4.62%	4.57%	6.62%	6.65%

Table 4.7.: Summary table for MAFISC tests compared to gzip. Top values are the percentage gains for compression time and bottom values for file size after compression. Bigger values are worse, thus the negatives mean that it was better than gzip (highlighted in blue). Best MAFISC values shown in bold.

As expected the MAFISC and lzma tests perform much slower than gzip (positive percentages), because lzma algorithm is known to offer some of the best compression ratios at the cost of execution time. Therefore, both MAFISC and solo lzma output smaller files, i.e. compress more. The best defaults for MAFISC are displayed in bold and, as previously explained, the best results come from the files that were chunked using sizes that fit the CPUs L2 cache of the test nodes. When compressing with lzma level 9, and comparing to the modified version of MAFISC, the execution is faster and output sizes are smaller. The latter was unexpected because both are using lzma -9 and MAFISC should be able to compress more after applying its filters. A possible explanation is that while MAFISC operates within hdf5, it compresses individual chunks of 1.5MB or 32MB, while that using lzma separately do not have limits, therefore it handles the entire file and can find better matches. A somewhat unfair comparison can be made for LZ4 filter within hdf5. As LZ4 is already less capable of compressing than gzip it does perform the worse in terms of compressed file size, however it is the only one who surpass gzip in terms of speed, an expected outcome.

4.8. Full measure of I/O

In this section we present the times for compression including full measurement of I/O time which were not contemplated on the main tests. We do it by performing some measurements using Unix `time` to capture both the full execution of the compression cycle (internally measured by OpenMP routines) and the `sync` call at the end. This way all the data on the buffers is written to disk on completion and the I/O is properly measured. As a consequence of this the reported times can increase greatly (Table 4.8).

This is evaluated with the datafile engraph1_100.bin.The datafile engraph1_100.bin is a binary dataset and it has a size of 650MB, which is the closest to the average size of our datasets (860MB). Both gzip and pigz are not shown because they add nothing to the comparison, as both compressors have very similar times

	engraph1_100.bin					
	compression level 1			compression level 9 or 24		
test case	OMP wtime	`time`	both→null	OMP wtime	`time`	both→null
LZ4 sync	2.283	13.905	1.564	28.575	31.648	28.362
LZ4	2.262	2.265	1.466	29.296	29.299	31.23
FPC sync	2.455	15.971	1.747	6.665	17.879	6.116
FPC	2.405	2.528	1.669	6.688	6.797	6.138
lz4mt sync	2.290	16.395	1.627	28.497	31.438	27.861
lz4mt	2.323	2.332	1.613	28.309	28.316	27.513
pFPC sync	4.059	17.355	2.146	8.404	18.152	6.658
pFPC	4.046	4.158	2.138	8.163	8.284	6.515
`cp` sync	na	14.984				
`cp`	na	1.016				

Table 4.8.: Execution time measurements comparing the various compressors when including all I/O time (with sync), and the execution time measured by the OpenMP walltime routines. Only compression times are analysed for a single datafile.

when using sync or not. This happens because their base (lowest) execution time is higher than the I/O time enforced by sync. For the present parallel compressors, the shown execution times come from only one thread.

First observation that should be made is that OMP routines do not capture the real I/O time, because it happens after the compressor algorithm has already finished. However, when output is directed to null FPC/pFPC and LZ4/lz4mt present an even smaller compression time.

The time it takes to copy the file is around 15 seconds, as visible on `cp` sync time cell. Therefore and with sync enforced, no other test case is expected to be faster than this (i.e. have lower execution time than a simple file copy). Incredibly LZ4 level 1 compression shows to be faster (LZ4 takes 14 seconds, 1 second less than using `cp`) than a simple disk copy (both forcing sync). This is unexpected and is likely due to the reduced amount of data that needs to be written and to the sheer speed of LZ4 level 1.

When higher compression levels are used (9 for LZ4/lz4mt and 24 for FPC/pFPC) the I/O time can be mostly hidden by the compression time. That is the case for LZ4/lz4mt which present similar time values with and without sync. Although, FPC/pFPC show smaller compression time (even at this compression level) when there is no sync, achieving less than half of the compression time for when sync is enforced. Therefore, in this scenario FPC/pFPC shows to be faster for the computations alone (i.e. without accounting for complete I/O time).

Regarding this specific test case, we can say that the cost of I/O varies between 11 and 14 seconds for LZ4/lz4mt and FPC/pFPC, when compressing engraph1_100.bin with one thread and both low and high compression levels.

5. Conclusions and future work

In this final chapter only two sections are presented. We start with our conclusions based on the experiments performed, and then follow to the section were some observations are made about the work that we prospect can be done next.

5.1. Summary and conclusions

This dissertation reflects an effort to compare the performance of some selected data compressors on scientific data. An analysis of the state of the art has been presented in the Chapter 2. The compression trend seems to be growing within the scientific community, as computational resources available grow towards the Exascale and scientific simulations produce an increasing quantity of data. While lossless compression is still the preferred choice, there is a growing interest towards lossy compression, useful for data visualization (as it is straightforward to estimate in that context the effect of approximations). Lossy compression can also be used in scientific applications, if high percentages of correlation with original data are guaranteed. It seems that parallel compression has not yet attracted much attention, as the related work on this topic is as yet scarce.

Over this work a combination of relevant scientific datasets were collected and tested with three compression algorithms: gzip and LZ4 are both based on the general LZ dictionary coders, and FPC is a specific floating-point data compressor. We also briefly tested MAFISC which applies filters to the floating-point data in order to facilitate data compression.

On Chapter 3 we defined the entire test bench used throughout the tests performed. The selected compressors are presented, the datasets characterized and the methodology for the tests execution and time measurements are explained. The content in the tested scientific datasets is highly random, with an average random entropy (randomness) of 81.43%. Very dependent on the datafile is its uniqueness (% of uniques), with some being composed of mostly unique values (typically zeroes and ones) , and others with a low percentage of uniques. The datasets with with lower uniqueness still present high randomness.

It is from the results, presented in Chapter 4, that we manage to take the most conclusions. The best performing compressors when it comes to CR are gzip/pigz, best in 22 of 33 datafiles, and FPC with the remaining 11 datafiles. For parallel speedup pigz yields the best values, but is the slowest compressor when it comes to absolute serial execution times. LZ4 is the fastest compressor (for the lowest compression level), especially when decompressing. The most efficient parallel compressor is pigz, but closely followed by lz4mt, presenting efficiency around one when 12 threads are used on a 12 core machine. pigz showed a better performance than gzip on our testing system, as it is faster even with only one thread. The minor compression ratio lost when present is caused the datafiles. Using pigz instead of gzip is a trade-off whose benefit increases with the number of threads and the attained speedups close to linear.

pFPC was tested as available, even though it is not optimized to be used as such. pFPC performance was poor, as a consequence of this, but with slight changes we believe it could become the best compressor for specific binary files, because of low execution times and the best CR it achieves for those files. The memory requirements for the pFPC compressor can become critical, when high compression levels are used together with many threads, by requesting several gigabytes of RAM memory. However gzip/pigz and LZ4/lz4mt use memory sparingly in comparison, with lz4mt being the one that needs more memory (196MB with 24 threads).

Regarding total execution times, and if the full I/O time is taken into account, we found that LZ4 with fast compression level can be faster, if marginally, than a simple memory copy using Unix `cp`. Note that LZ4 applies the compression and performs the data output to disk when `cp` only performs a memory copy, yet this advantage will probably get lost if we were to add the decompression time too. When decompressing and discarding the output to null more than 2GB/s of throughput were achieved by lz4mt for certain datafiles. lz4mt can deliver exceptional compression and decompression speed when CR is not the main goal.

MAFISC, the compressor that applies filters prior to compressing the data with lzma, presents higher CR than gzip on two datasets tested, as was expected. While that the compression speed is faster than simply using lzma, which is used inside MAFISC, the decompression speed could not be analysed because it was impossible to test due to a bug on the underlying data library HDF5.

Ending fun fact: the size of this pdf document is around 450KB and the folder with all the source files is 2236KB. Therefore, after compilation and internal pdf compression, this dissertation achieves a CR of 4.97.

5.2. Final considerations and future work

It is mostly clear that compression is not an ancient paradigm for the lack, and high cost, of storage and communication bandwidth. The modern days and future do and will take advantages of data compression.

Because it was not in the scope of this work to explore the approaches in parallel implementation of the compressors, this could well be the next step to take. We found in pFPC a high potential but the current version is not yet fully mature. pFPC will require a proper implementation to allow its full potential to develop. For lz4mt a tuning can also improve performed as this implementation is quite recent. Overall pigz seems the most mature of the parallel compressors studied, so the work should focus lz4mt and pFPC for performance tuning and possible CR improvements.

The filtering and data reordering approaches provide a better outcome than pure compression. This is more versatile because for each kind of problem it is possible to adjust the filtering and reordering that might give better compression, without the need to sacrifice precious computing time with a higher compression algorithm. MAFISC should be tested with more datasets and compared with more compressors, as well as investigate memory requirements. An analysing of the decompression speed, which was not possible to assess, should be made. More filters can be developed, as long as they are reversible, and learn the best order they can be applied to the data. In order to make MAFISC faster the general compressor lzma can be changed for other faster compressor, and performance trade-off should be contemplated. Parallelization of MAFISC can be studied on how to be applied: using parallel HDF5, or directly apply MAFISC to the datafiles.

Bibliography

[1] Jack Dongarra Horst Simon Hans Meuer, Erich Strohmaier. top500 Super computer sites. http://www.top500.org/. Accessed January 31, 2014.

[2] Yann Collet. Development blog on compression algorithms. http://fastcompression.blogspot.in/2011/05/lz4-explained.html. Accessed Januray 31, 2014.

[3] M. Burtscher and P. Ratanaworabhan. FPc: A high-speed compressor for double-precision floating-point data. *Computers, IEEE Transactions on*, 58(1):18–31, January 2009. ISSN 0018-9340. doi: 10.1109/TC.2008.131.

[4] Nathanael Hübbe and Julian Kunkel. Reducing the HPC-datastorage footprint with mafisc—multidimensional adaptive filtering improved scientific data compression. *Computer Science-Research and Development*, pages 1–9, 2013. doi: 10.1007/s00450-012-0222-4.

[5] Rene Brun and Fons Rademakers. Root - an object oriented data analysis framework. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 389(1-2):81–86, 1997. ISSN 0168-9002. doi: 10.1016/S0168-9002(97)00048-X. URL http://www.sciencedirect.com/science/article/pii/S016890029700048X. <ce:title>New Computing Techniques in Physics Research V</ce:title>.

[6] R Brun, F Rademakers, and S Panacek. Root, an object oriented data analysis framework. 2000.

[7] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *Information Theory, IEEE Transactions on*, 23(3):337–343, May 1977. ISSN 0018-9448. doi: 10.1109/TIT.1977.1055714.

[8] Vadim Engelson, Dag Fritzson, and Peter Fritzson. Lossless compression of high-volume numerical data from simulations. In *Proceedings of the Conference on Data Compression*, DCC '00, pages 574–, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0592-9. URL http://dl.acm.org/citation.cfm?id=789087.789763.

[9] A Nicolaucig, M Mattavelli, and S Carrato. Compression of tpc data in the alice experiment. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 487(3):542–556, 2002. ISSN 0168-9002. doi: 10.1016/S0168-9002(01)02195-7. URL `http://www.sciencedirect.com/science/article/pii/S0168900201021957`.

[10] Paruj Ratanaworabhan, Jian Ke, and Martin Burtscher. Fast lossless compression of scientific floating-point data. In *Proceedings of the Data Compression Conference*, DCC '06, pages 133–142, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2545-8. doi: 10.1109/DCC.2006.35.

[11] M. Burtscher and P. Ratanaworabhan. High throughput compression of double-precision floating-point data. In *Data Compression Conference, 2007. DCC '07*, pages 293–302, March 2007. doi: 10.1109/DCC.2007.44.

[12] L. Yang, H. Lekatsas, and R.P. Dick. High-performance operating system controlled memory compression. In *DESIGN AUTOMATION CONFERENCE*, volume 43, page 701. ACM/IEEE, 2006.

[13] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-scalar RAM-CPU cache compression. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, pages 59–, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2570-9. doi: 10.1109/ICDE.2006.150.

[14] Jay F. Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. Flexible IO and integration for scientific codes through the adaptable IO system (ADIos). In *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, CLADE '08, pages 15–24, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-156-9. doi: 10.1145/1383529.1383533.

[15] B. Welton, D. Kimpe, J. Cope, C.M. Patrick, K. Iskra, and R. Ross. Improving I/O forwarding throughput with data compression. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 438–445, September 2011. doi: 10.1109/CLUSTER.2011.80.

[16] E.R. Schendel, Ye Jin, N. Shah, J. Chen, C.S. Chang, Seung-Hoe Ku, S. Ethier, S. Klasky, R. Latham, R. Ross, and N.F. Samatova. ISObar preconditioner for effective and high-throughput lossless data compression. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 138–149, April 2012. doi: 10.1109/ICDE.2012.114.

[17] Sriram Lakshminarasimhan, Neil Shah, Stephane Ethier, Seung-Hoe Ku, C. S. Chang, Scott Klasky, Rob Latham, Rob Ross, and Nagiza F. Samatova. Isabela for effective in situ compression of scientific data.

Concurrency and Computation: Practice and Experience, 25(4):524–540, 2013. ISSN 1532-0634. doi: 10.1002/cpe.2887.

[18] Eric R. Schendel, Saurabh V. Pendse, John Jenkins, David A. Boyuka, II, Zhenhuan Gong, Sriram Lakshminarasimhan, Qing Liu, Hemanth Kolla, Jackie Chen, Scott Klasky, Robert Ross, and Nagiza F. Samatova. ISObar hybrid compression-I/O interleaving for large-scale parallel I/O optimization. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 61–72, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0805-2. doi: 10.1145/2287076.2287086.

[19] M. Burtscher and P. Ratanaworabhan. pfpc: A parallel compressor for floating-point data. In *Data Compression Conference, 2009. DCC '09.*, pages 43–52, March 2009. doi: 10.1109/DCC.2009.43.

[20] M. Burtscher and P. Ratanaworabhan. gfpc: A self-tuning compression algorithm. In *Data Compression Conference (DCC), 2010*, pages 396–405, March 2010. doi: 10.1109/DCC.2010.42.

[21] Molly A. O'Neil and Martin Burtscher. Floating-point data compression at 75 Gb/s on a GPU. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 7:1–7:7, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0569-3. doi: 10.1145/1964179.1964189.

[22] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, March 1991. ISSN 0360-0300. doi: 10.1145/103162.103163.

[23] Randal E. Bryant and David R. O'Hallaron. Computer Systems A Programmer's Perspective 1 (Beta Draft). 2001.

Part III.

Apendices

A. Tests, Results and Conclusions

	Sp_1 per compressor level, into /local or /tmp								Sp_1 per compressor level, into /dev/null							
	pigz			lz4mt		pFPC			pigz			lz4mt		pFPC		
dataset	1	6	9	1	9	1	12	24	1	6	9	1	9	1	12	24
waterglobe.arc.txt	0.94	1.05	1.06	1.01	1.03	nd	nd	nd	0.99	1.06	1.06	1.00	1.04	nd	nd	nd
waterglobe.1col.arc.txt	0.93	1.05	1.06	1.01	1.02	nd	nd	nd	0.99	1.06	1.07	0.99	1.03	nd	nd	nd
waterglobe.vel.txt	0.99	1.18	1.18	1.02	1.02	nd	nd	nd	1.11	1.18	1.19	0.99	1.05	nd	nd	nd
waterglobe.1col.vel.txt	0.99	1.18	1.20	1.02	1.03	nd	nd	nd	1.11	1.18	1.20	0.99	1.01	nd	nd	nd
waterglobe.arc.bin	0.91	0.90	0.92	1.01	0.99	0.63	0.64	0.84	0.97	0.99	0.99	0.93	1.00	0.89	0.90	0.93
waterglobe.vel.bin	0.90	0.93	0.92	1.00	1.02	0.64	0.64	0.84	0.97	0.99	0.99	0.95	1.01	0.88	0.90	0.86
engraph1_100.txt	1.03	1.17	1.18	0.99	1.01	nd	nd	nd	1.12	1.18	1.18	0.99	1.03	nd	nd	nd
engraph1_100.1col.txt	1.00	1.17	1.18	0.99	1.02	nd	nd	nd	1.13	1.17	1.19	1.00	1.03	nd	nd	nd
engraph1_100.bin	0.97	1.02	1.03	1.03	1.01	0.63	0.63	0.83	1.08	1.13	1.13	0.95	1.01	0.87	0.86	0.91
gauss09_alpha.txt	0.98	1.03	1.06	0.99	1.01	nd	nd	nd	0.99	1.04	1.06	1.00	1.01	nd	nd	nd
gauss09_density.txt	0.98	1.04	1.07	0.99	1.00	nd	nd	nd	1.00	1.04	1.07	0.99	1.02	nd	nd	nd
gauss09_alpha.bin	0.97	1.05	1.14	0.98	1.01	0.63	0.70	0.85	0.97	1.06	1.14	0.97	1.01	0.87	0.90	0.89
gauss09_density.bin	0.96	0.98	0.98	1.02	1.00	0.63	0.65	0.86	0.97	0.98	0.98	0.94	1.00	0.83	0.87	0.91
msg_bt	0.95	0.93	0.95	0.98	1.00	0.62	0.62	1.11	0.97	1.01	1.00	0.94	0.98	0.87	0.88	0.88
msg_lu	0.97	0.91	0.91	0.96	0.98	0.62	0.63	0.82	0.97	1.00	0.99	0.96	0.98	0.87	0.86	0.89
msg_sp	0.94	0.97	1.00	0.99	0.97	0.60	0.60	0.83	0.97	1.02	1.03	0.88	0.99	0.87	0.88	0.92
msg_sppm	0.96	0.93	1.07	0.97	0.99	0.71	0.79	0.87	0.95	0.95	1.07	0.96	1.00	0.87	0.87	0.87
msg_sweep3d	0.97	0.98	0.98	0.94	0.99	0.65	0.68	0.90	0.98	1.00	0.99	0.94	1.00	0.88	0.87	0.96
num_brain	0.96	1.00	0.99	0.93	0.99	0.64	0.63	0.84	0.97	1.01	1.01	0.93	1.00	0.87	0.88	0.93
num_comet	0.98	0.98	1.01	0.99	0.99	0.66	0.64	0.83	0.97	1.00	1.01	0.95	1.00	0.89	0.88	0.88
num_control	0.96	1.01	0.99	1.03	1.00	0.63	0.64	0.84	0.98	1.00	1.00	0.97	0.99	0.88	0.89	0.90
num_plasma	0.98	0.98	0.99	0.98	0.98	0.65	0.66	1.08	0.97	0.97	0.97	1.02	1.00	0.85	0.84	1.07
obs_error	0.98	1.00	1.03	0.93	0.99	0.62	0.65	0.90	0.98	1.00	1.03	0.95	1.01	0.85	0.86	0.95
obs_info	0.97	0.99	0.99	0.91	0.98	0.74	0.66	0.97	0.96	0.98	0.98	1.14	1.01	0.86	0.87	0.99
obs_spitzer	0.95	1.00	1.03	0.95	0.99	0.64	0.65	0.85	0.97	1.01	1.03	0.95	1.00	0.88	0.87	0.94
obs_temp	0.98	0.99	0.99	0.89	0.99	0.68	0.65	0.85	0.96	0.99	0.98	1.12	1.01	0.86	0.86	0.90
NTUP1_floats.bin	0.90	0.95	1.10	0.95	1.00	0.63	0.64	0.82	0.95	0.98	1.11	0.96	1.00	0.86	0.86	0.90
NTUP2_floats.bin	0.92	0.94	1.10	0.95	1.01	0.64	0.65	0.81	0.96	0.99	1.11	0.96	1.00	0.86	0.87	0.91
NTUP3_floats.bin	0.81	0.82	0.99	0.97	1.00	0.64	0.64	0.83	0.85	0.88	0.99	0.96	1.01	0.86	0.86	0.91
NTUP4_floats.bin	0.81	0.83	0.99	0.97	1.00	0.64	0.65	0.82	0.86	0.88	0.99	0.95	1.00	0.86	0.87	0.91
NTUP5_floats.bin	0.84	0.82	0.99	0.95	1.01	0.64	0.64	0.81	0.85	0.88	0.99	0.96	1.00	0.86	0.87	0.91
NTUP1to5_doubles.bin	0.84	0.96	1.04	0.96	1.00	0.70	0.72	0.83	0.84	0.96	1.04	0.97	1.00	0.86	0.87	0.90
NTUP1to5_floats.bin	0.80	0.81	0.99	0.95	0.90	1.10	1.17	0.82	0.85	0.88	0.99	0.88	0.90	0.87	0.87	0.85
`AVG per level`	0.94	0.99	1.03	0.97	1.00	0.66	0.67	0.87	0.98	1.01	1.05	0.97	1.00	0.87	0.87	0.91

Table A.1.: Speedup of all the datafiles for the multi-threaded programs using only one thread. Compression levels are the minimum, medium and maximum. Lz4mt only has two compression levels available, and pFPC do not have a defined maximum so we use 24 as addressed in 4.4

	pigz			lz4mt			pFPC		
dataset	1th	12th	24th	1th	12th	24th	1th	12th	24th
waterglobe.arc.txt	1.06	13.43	17.94	1.03	11.44	12.93	nd	nd	nd
waterglobe.1col.arc.txt	1.06	13.63	17.57	1.02	11.29	12.80	nd	nd	nd
waterglobe.vel.txt	1.18	13.61	17.55	1.02	11.03	12.29	nd	nd	nd
waterglobe.1col.vel.txt	1.20	13.58	18.41	1.03	11.46	12.45	nd	nd	nd
waterglobe.arc.bin	0.92	12.82	16.30	0.99	10.44	12.14	0.84	2.51	2.06
waterglobe.vel.bin	0.92	12.68	16.07	1.02	10.45	12.39	0.84	2.52	2.08
engraph1_100.txt	1.18	12.51	18.36	1.01	11.12	13.08	nd	nd	nd
engraph1_100.1col.txt	1.18	12.68	19.18	1.02	10.96	13.38	nd	nd	nd
engraph1_100.bin	1.03	11.88	16.41	1.01	10.24	11.96	0.83	2.05	1.41
gauss09_alpha.txt	1.06	12.69	20.85	1.01	10.45	12.21	nd	nd	nd
gauss09_density.txt	1.07	12.67	19.28	1.00	9.81	12.19	nd	nd	nd
gauss09_alpha.bin	1.14	13.49	18.25	1.01	8.93	10.31	0.85	1.59	0.81
gauss09_density.bin	0.98	11.41	14.83	1.00	8.71	8.74	0.86	1.17	0.52
msg_bt	0.95	11.89	16.06	1.00	8.89	10.01	1.11	1.90	1.03
msg_lu	0.91	11.71	15.73	0.98	8.44	9.90	0.82	1.21	0.63
msg_sp	1.00	12.08	16.47	0.97	9.16	10.71	0.83	1.55	0.89
msg_sppm	1.07	11.73	15.81	0.99	7.74	8.74	0.87	0.81	0.46
msg_sweep3d	0.98	11.49	15.30	0.99	8.27	8.29	0.90	0.91	0.43
num_brain	0.99	11.64	15.60	0.99	8.49	9.26	0.84	1.13	0.54
num_comet	1.01	11.68	15.76	0.99	7.36	7.97	0.83	0.95	0.42
num_control	0.99	11.63	15.47	1.00	7.77	8.71	0.84	1.25	0.58
num_plasma	0.99	10.27	12.73	0.98	4.62	3.90	1.08	0.77	0.31
obs_error	1.03	11.87	15.91	0.99	5.78	5.61	0.90	0.78	0.33
obs_info	0.99	9.48	11.34	0.98	3.01	2.53	0.97	0.43	0.17
obs_spitzer	1.03	12.17	16.45	0.99	8.96	9.64	0.85	1.38	0.72
obs_temp	0.99	10.83	12.91	0.99	5.83	4.78	0.85	0.56	0.20
NTUP1_floats.bin	1.10	14.03	15.01	1.00	10.70	13.08	0.82	2.30	1.84
NTUP2_floats.bin	1.10	14.12	15.04	1.01	10.65	13.26	0.81	2.33	1.83
NTUP3_floats.bin	0.99	12.85	13.46	1.00	10.73	13.25	0.83	2.37	1.86
NTUP4_floats.bin	0.99	12.58	13.48	1.00	10.91	13.22	0.82	2.37	1.85
NTUP5_floats.bin	0.99	12.59	13.46	1.01	10.54	13.17	0.81	2.35	1.83
NTUP1to5_doubles.bin	1.04	13.31	13.37	1.00	10.14	11.23	0.83	1.08	0.50
NTUP1to5_floats.bin	0.99	12.33	14.32	0.90	10.00	12.49	0.82	0.65	0.45
`AVG (all files)`	1.03	12.34	15.90	1.00	9.22	10.50	0.87	1.48	0.95

Table A.2.: Speedup of all the datafiles for the multi-threaded programs using 1, 12 and 24 threads. The maximum compression levels are used and correspond to 9 for both pigz and lz4mt, and 24 for pFPC. Note that the average values originate from the entire table, available in the appendix.