PARALLEL IMPLEMENTATION OF THE FAPEC DATA COMPRESSOR

Sergi García-Almoril [1], Jordi Portell [1,2,3], Enrique Garcia-berro [1,4]

[1] ***Institute for Space Studies of Catalonia (IEEC)***
c/Gran Capità 2-4, 08034 Barcelona, Spain
Email: sergio.garcia.almoril@outlook.com

[2] ***DAPCOM Data Services S.L.***
ESA BIC Barcelona (Parc UPC - PMT, RDIT). C / Esteve Terrades 1, 08860 Castelldefels (Barcelona), Spain
Email: jordi.portell@dapcom.es

[3] ***Departament d'Astronomia i Meteorologia (ICC-UB), University of Barcelona***
c/Martí i Franquès 1, 08028 Barcelona, Spain

[4] ***Departament de Física Aplicada, Universitat Politècnica de Catalunya (UPC)***
C / Esteve Terrades 5, 08860 Castelldefels (Barcelona), Spain
Email: enrique.garcia-berro@upc.edu

ABSTRACT

The current space mission concept comprises a large set of different strategies, instruments and scientific objectives. Given the large diversity of data, a general-purpose coder offering the highest possible compression ratios for almost any kind of data benefits the vast majority of space missions. The Fully Adaptive Prediction Error Coder (FAPEC) was born to overcome some of the weaknesses of the CCSDS 121.0 recommendation for lossless data compression. Specifically, FAPEC solves the degradation of performance in the presence of outliers with a low computational cost and good compression ratios. This paper describes a new improvement in the current implementation of FAPEC focusing on the processing speed. The tests done so far with the current C implementation running on x86-based computers with GNU/Linux reveal that, in the worst of the cases, FAPEC requires 90% of the processing time required by the CCSDS 121.0 standard. Nevertheless, the computing load can be further reduced, taking advantage of the widely available multi-core computers, both on ground and in space (such as Leon 3). Here we present an efficient multi-process implementation of the FAPEC data compressor. It is versatile, scalable and configurable, keeping its high compression ratios but significantly reducing the processing time. We present the implementation approach and the results obtained on a variety of data files, showing its good scalability in systems of up to 16 cores.

Keywords: Data compression, lossless, FAPEC, parallel, multi-core.

I. INTRODUCTION

Throughout the history of computing, the achievement of different goals has been pursued. As technology evolves, we are trying both to decrease the processing time and to further miniaturize the hardware. The processing, transfer and storage of as much information as possible has typically been one of the main goals as well. In the scenario that concerns us, space missions keep increasing the volume of information that is generated, which requires better methods and systems to handle it. Compressing the data prior to its transfer from the satellite to the ground station allows making better use of the available communication resources, but this compression must not require excessive computing resources onboard.

In this paper we focus on the FAPEC [1] data compression algorithm (Fully Adaptive Prediction Error Coder). An efficient FPGA implementation is available [2], but here we will only consider the software implementation. FAPEC was created to meet the high requirements in space missions and aeronautical systems, but with enough versatility to be useful in massive data processing systems on ground, such as scientific research, supercomputing, data communications or multimedia. On this regard, despite of its excellent performance, FAPEC presents two key limitations. On one hand, it has a limitation on the maximum size of the file to compress. This limitation is actually imposed by the existing software framework (which loads and compresses the complete file at once), not by the core algorithm in itself. On the other hand, the compressor has several configuration options which have to be manually passed to the decompressor. It would obviously be much better to include all this configuration information in the compressed file. One of the goals of our work is to solve these two limitations.

The main motivation of this work, however, is to achieve an outstanding performance with FAPEC, allowing to compress enormous volumes of information in the minimum possible time. Nowadays, the majority of computers, either professional or domestic, on-ground or even in satellite payloads, have multi-threaded or multi-core architectures. The software implementation available for FAPEC is envisaged for a single-core processor, namely, there is only one thread of sequential execution throughout the program. The main goal of this work is to implement a concurrent (or multi-process) version of FAPEC, to allow running it on different parallel processes, thus taking full advantage of the multi-threaded or multi-core architectures leading to state-of-the-art compression performances.

II. THE FAPEC DATA COMPRESSOR

FAPEC is an entropy coding system based on a segmentation strategy and running on relatively small data blocks. This approach leads to a good adaptation to changing data statistics, while its processing requirements (both in terms of computation and temporary storage or memory) are very small. It is a lossless compressor following a two-stage approach, similarly as in the standards recommended by the CCSDS (such as 121.0 for lossless data compression, as well as 122.0 for images and 123.0 for multi-band images). The first stage is devoted to pre-processing, reducing the entropy of the original data by means of a reversible algorithm (which can be as simple as a differential or delta predictor). The pre-processing output is a series of prediction errors, typically (but not necessarily) following a Laplacian distribution [1]. The second stage encodes these prediction errors with variable-length codes, leading to the compression in itself. FAPEC performs an efficient statistical analysis of each small block of values to be compressed, determining the optimum coding tables and finally calling the PEC coding core [3]. This approach has proven to be resilient in front of outliers in the data (such as those caused by noise or prompt particle events in space detectors). Contrary to CCSDS 121.0, the compression ratios achieved by FAPEC do not significantly decrease in front of such contamination. Despite of requiring slightly larger data blocks than in CCSDS 121.0 (typically some hundred samples), FAPEC is still completely compatible with the requirements imposed by space communications, where small and independent data blocks are required to minimize data loss in front of errors in the space-ground link. Fig. 1 illustrates the main stages of FAPEC.

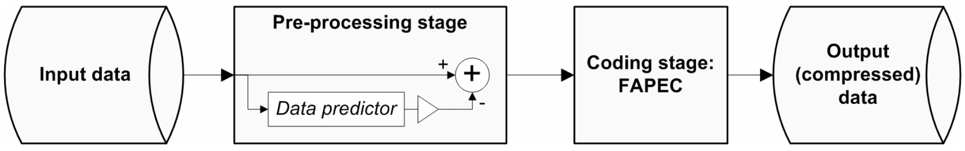

Fig. 1. Two-stage data compression approach followed by FAPEC

FAPEC was initially designed for simple and lossless compression, but it has proven to also support more elaborated and lossy pre-processing stages, such as DWT-based image compression [4] which slightly increases its processing time. It reinforces the usefulness of a parallel implementation.

The coding stage is based on PEC [3] which must be adequately configured for each block of data to compress, and here is where FAPEC plays its role. FAPEC adds an adaptive layer to PEC, so that for each block of data (typically 100-500 samples) determines, in a highly efficient manner, the best coding strategy and coding table. The operation consists of 4 main blocks, illustrated in Fig. 2. The first block corresponds to the general initialization of the system variables, buffers and an optimized histogram. The second corresponds to the input block, which comprises the acquisition of values and their pre-processing. This block is also responsible for generating the histogram of the pre-processed values and their accumulation in an internal buffer. The third block analyses the data by efficiently examining the histogram, and defines the coding strategy to be used and the coding segments (or coding table). In the last block we call the PEC coder with the configuration obtained, generating the output file.

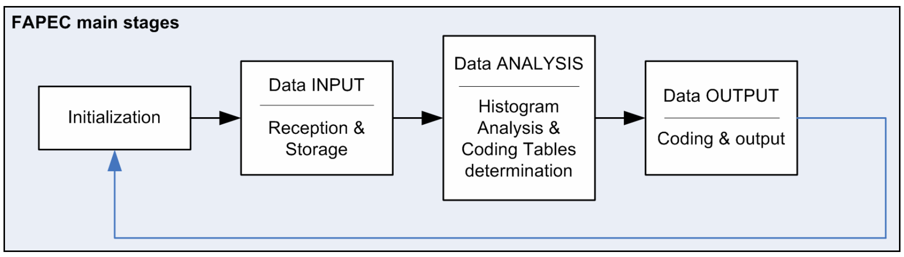

Fig. 2. Functional blocks of the FAPEC compressor

The data analysis block corresponds to the core of FAPEC, where most of the actual processing takes place – at least for the case of simple pre-processing stages. Therefore it seems to be the most suitable element for parallelisation. However, the input and output blocks also perform quite a lot of operations, first for the pre-processing (which can be quite expensive in some cases, such as DWT-based pre-processing), and then for the PEC coding in itself (which is very efficient but not negligible). In the following sections we will study the problem and decide the most adequate parallelisation approach.

III. PARALLEL IMPLEMENTATION OF FAPEC

III.1. APPROACH TAKEN

The complexity of the FAPEC parallelisation is that we already start from a very efficient algorithm, whereas parallelisation is often applied to complex algorithms that require moderate to long execution times. In our case, the goal is to achieve an outstanding performance to allow handling vast amounts of data in the shortest possible time.
In most systems, accessing to memory for reading and writing data is considered a bottleneck. Any other operation in the core of an algorithm (especially in our case) is a series of conditions, loops, and typically fast arithmetic operations. Therefore, the parallelisation approach that we have taken includes not only the algorithm core, but also the routines for data input and output. To achieve this, we take a strategy known as Grained Parallelism.
The Grained Parallelism is a fairly extended parallel programming technique. It is based on dividing a program execution in smaller subtasks in terms of lines of code and runtime, so that overall it substantially reduces the total execution time. There are two strategies within this technique, namely, fine-grained and coarse-grained. Fine-grained execution consists of parallelising a few lines of code with very fast and very simple operations, where the inter-process communication is often carried out through a few messages. In this case, such inter-process messaging (or synchronization) requires an extremely fast and especially low-latency messaging or networking system, so this approach is typically taken only in shared-memory systems (or accelerator co-processors, such as General-Purpose Graphical Processing Units). Conversely, coarse-grained communication between processes is performed once a greater amount of slightly more complex and elaborate instructions have been executed. In our case, we choose a coarse-grained parallelism approach, where each execution element will correspond to the compression of a fraction of the file. In this way each subtask is responsible for compressing a small amount of data. The inter-process communication will take place when each of these data units is compressed.
This approach brings us to the definition of the *chunk* concept. We define a data chunk as a portion of data from the original file. This portion will be analysed and processed with the FAPEC algorithm normally. It will be handled as a self-consistent data element, that is, any pre-processing status or reference will be reset at the start of a chunk – not at the start of the file. In this way, chunks will be independent each other, except for any global definition of the compression parameters, if necessary. This approach also enforces the suitability of this FAPEC implementation for space communications (and for a better error-resilience of the compressed file, in general). The creation and encapsulation of this chunk, the synchronization of processes for the correct file division, and the subsequent distribution of these chunks in the processing will be of special importance in our parallelisation work. Based on this definition of a chunk, we define a multitasking system by specifying the number of processes and the chunk size, which will compress the data by exploiting multi-core or multi-processor architectures. Thus, we assume shared-memory systems, and avoid the need of inter-node communications systems such as MPI (Message Passing Interface).
The functional description of the system is as follows. First of all, a *parent process* initializes the structures and variables needed, retrieves the specified configuration, and encapsulates all the options into a message structure that secondary processes (or child processes) inherit. The parent process creates as much processes as required (or as requested by the user). Then, each of these starts to run asynchronously.
Children processes iteratively perform the same function: they access the memory, extracting data to conform their own chunk, then compressing it with FAPEC, and finally writing their “compression done” message in the shared queue. After that, they are blocked, waiting for being awakened by the parent process, after which they enter a stage with mutual exclusion between children where the compressed chunks are written in the output file in the adequate order.
From the point of view of the parent process, once it creates the children processes it remains waiting for messages from them (indicating that the compression of their chunks is done), which will be received through a shared queue structure. When the parent receives a message, it parses the child identifier (ID) and awakes the process with the next ID. This parent-children synchronism with concurrent and uncontrolled access to memory gives a good compromise between processing speed and robustness of the system. The asynchronous part of accessing memory is very efficient, and the parent-children chunk-to-chunk communication allows some error control if necessary.
Once all the processes have exhausted their iterations and all the chunks have been compressed, they “die”, releasing the structures used, and the parent process ends the program execution. Fig. 3 shows the overall processing flow.

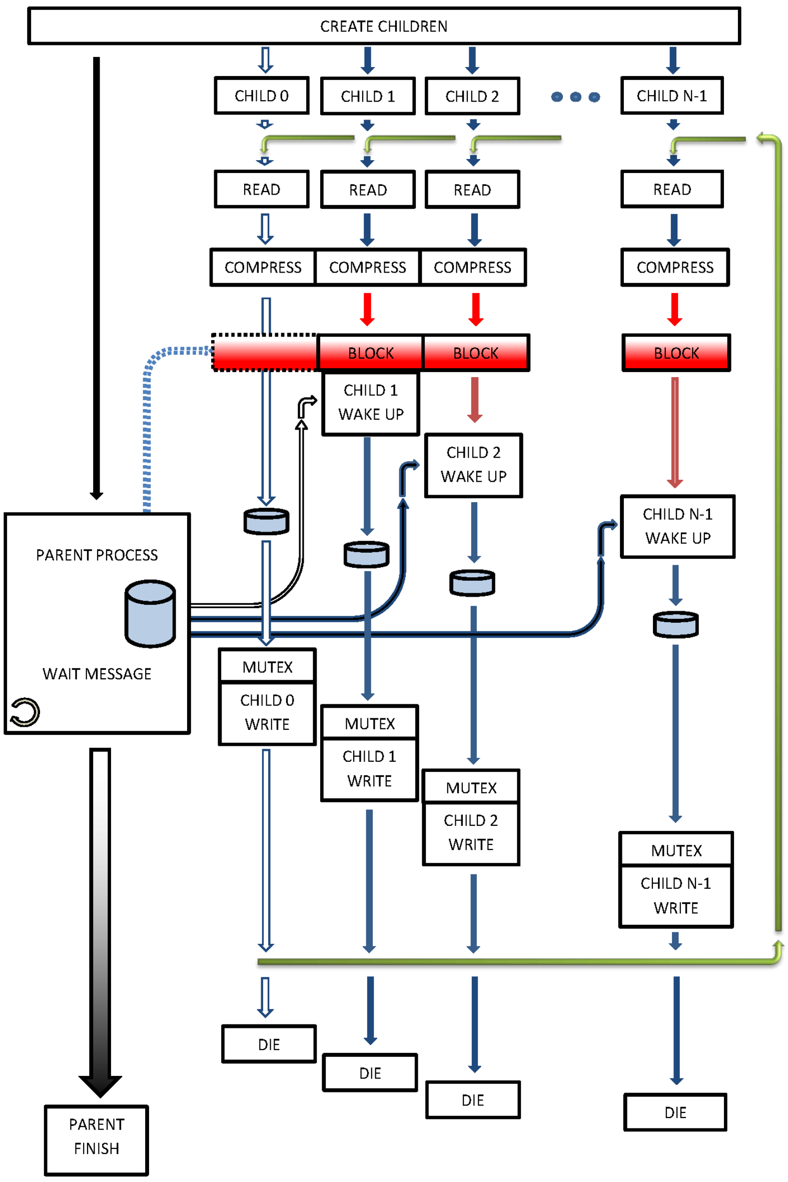

Fig. 3. Flowchart of the Parallel-FAPEC compression approach

III.2. IMPLEMENTATION

The parallelization code is implemented following the POSIX standard for UNIX systems. The main tools used are shared semaphores and shared queues. The excellent results come from the optimization of the processes of input and output data. The FAPEC software implementation available, memory was sequential accessed by a single processor. In the Parallel-FAPEC implementation, each process accesses the file concurrently, retrieving the appropriate chunk and compressing it using the FAPEC core. The concurrent and asynchronous access provides a great performance. Each process identified with an ID is encapsulated in a message structure containing some additional information concerning the number of iterations to be performed, which is the ID of the process that will read the last chunk, among others. Owing to this information, a child process can calculate the file position to be read, after which it can proceed to read the data chunk and compress it. Fig. 4 illustrates this file reading and compressing approach.

The process of writing is performed differently. A blocked process will wake up when indicated by the parent, following the order of creation (that is, its ID). In this way, any process will not write to the output file until the process compressing the previous chunks (that is, with lower IDs) finish their writing. Therefore, we automatically keep the adequate order in the output file without having to know the size of the previous compressed chunks of each child. Newer chunks are simply appended to the output file.

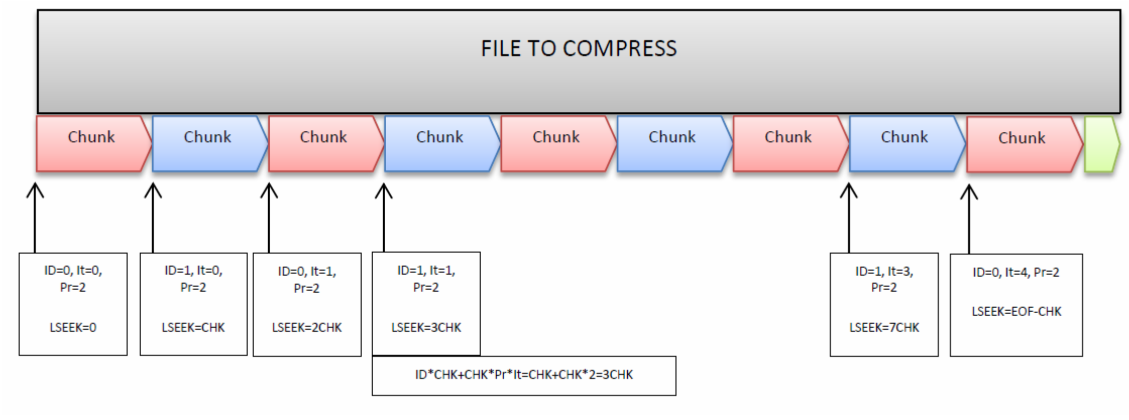

Fig. 4. Reading and compressing a file in parallel

III.3. DECODER

The decoding process is completely equivalent to the compression process. The FAPEC decompression core is not changed either: we only optimize the process of reading and writing, as well as the handling of compressed chunks, that is, the memory access routines. In this case, the specified chunk size does not have a fixed value. It depends obviously on how many bytes we generate in the resulting compressed chunk. In compression, before writing the chunk in memory, we allocate a few bytes at the beginning of each chunk to indicate its resulting compressed size. In this way, this chunk header eases the reading process when decompressing. In addition, there is an initial file header storing certain information related to the original file, chunk size and FAPEC compression settings. In this way we solve the initial limitation that consisted in the necessity to know the compression method (and configuration) when calling the decompressor.

When launching the decompression, before starting to read compressed chunks, the mentioned file header is read and the compression settings are extracted. Then, each process created will read, according to its ID and the ancillary information of the header (such as the number of chunks in the file), all the chunk headers of its predecessors. In this way, it determines the file location where it must start reading its chunk. Subsequently, we run the core decompression algorithm, storing the chunk to memory in an orderly manner once the parent wakes this child up. Fig. 5 illustrates this file decompression approach.

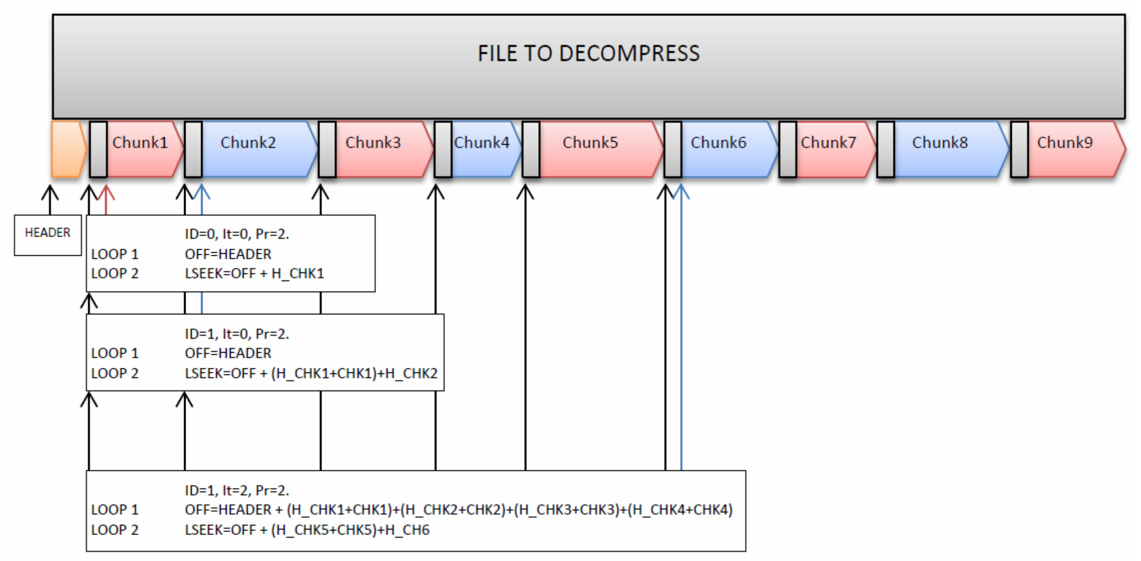

Fig. 5. Shaping chunks in decompression

IV. TESTS AND RESULTS

In this section we test the performance of the system on a variety of computing platforms. As a first example, Table 1 shows the compression speeds achieved by the Parallel FAPEC implementation on different image files, including a comparison with the currently available version of FAPEC. We have used 2 and 4 threads in this case. The test was done on an Intel® Core™ i5-3230M CPU @ 2.60 GHz running Ubuntu 64-bit under Virtual Box with 2 GB of RAM. The throughput calculated shows how the performance improves for longer file sizes, as we could otherwise expect. With small image files the overhead of launching the executable combined with the tiny wall clock times makes it difficult (actually inaccurate) to determine the actual performance, but when we increase the file size, better performances and scalability are observed. In all cases the current FAPEC compression times are better than those obtained with the 121.0 implementation available. Additionally, when activating parallelism (using 2 or 4 threads), even in this domestic (and virtualized) computer, we can easily see the improvement achieved.

Table 1. Performance obtained with CCSDS 121.0 and FAPEC with different threads

Image	Size	121.0 ratio	FAPEC ratio	CCSDS 121.0	FAPEC	FAPEC 2 threads	FAPEC 4 threads
banyoles-1024x600.raw	0.6 MB	1.3	1.3	14.1 MB/s	20.0 MB/s	24.2 MB/s	27.8 MB/s
pyrenees.raw	5.3 MB	1.5	1.4	18.3 MB/s	33.1 MB/s	34.8 MB/s	43.4 MB/s
pleiades.raw	15.4 MB	0.3	0.9	17.0 MB/s	34.1 MB/s	34.4 MB/s	45.9 MB/s
GOES	32.4 MB	1.4	1.4	19.4 MB/s	39.3 MB/s	40.7 MB/s	51.2 MB/s

Now that we see that the approach seems to work fine, one of the parameters to test is the chunk size. Fig. 6 shows the normalized processing time when using different chunk sizes in the compression of one raw data file from the ALMA radio telescopes and one from MAGIC data, tested on an Intel® Core™ 2 Quad CPU Q6600 @ 2.40 GHz running Gentoo Linux 64-bit. Therefore, the maximum speedup (with respect to the single-process case) is four, owing to the four cores of this processor. The chunk size varies from 1 to 64 MB, always using 8 concurrent processes. The ALMA file is 256 MB in size. The best configuration (16-bit little Endian samples and an interleaving of 8 samples) still does not allow any compression (as this file requires a specific pre-processing), but it allows to test the system under such extreme case. The MAGIC file is 1.9 GB, compressed as 16-bit little Endian samples, achieving a ratio of 2.43. As can be seen in the figure, the case of ALMA (256 MB) shows slightly better results for chunks between 1 and 8 MB and it decreases for larger chunks, which is otherwise expected considering the 8 processes and the file size. On the other hand, for the case of MAGIC (nearly 2 GB) the optimum is found between 2 and 16 MB chunks, which we will consider as the range of optimum chunk sizes.

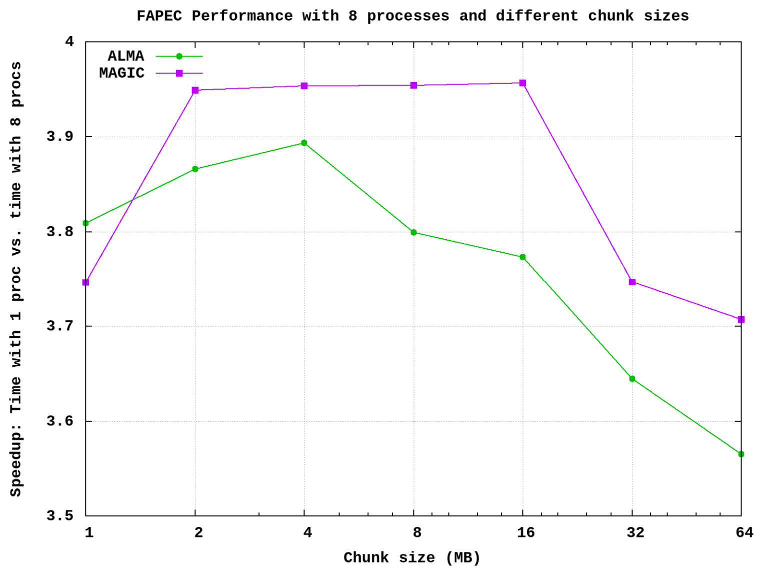

Fig. 6. Compression performances for different chunk sizes on a 4-core processor

To see the performance that we can achieve with this compressor, the last set of tests were run on a high-performance computing environment, namely, in one node of the BSC MareNostrum III supercomputer. It is an Intel® Xeon® CPU E5-2670 @ 2.6 GHz with 20 MB L3 Cache and 32 GB RAM, running SUSE Linux 64-bit. One of the tests to be considered is the performance penalty when compressing small files in such an environment – which is obviously beyond what we can find in a satellite payload, but gives a hint on the start-up overhead of the compressor. Fig. 7 shows the normalized execution time (using the single-process time as reference) for different file sizes (obtained from a portion of a MAGIC file). The compression options are the same as in Fig. 6, namely, 8 concurrent processes. The chunk size selected was 4 MB. We can see a slight decrease in the speedup for file sizes smaller than 64 MB, and therefore we conclude that this compressor appears to perform optimally for files larger than this size (at least in on-ground environments). Beyond that, we get quite close to the ideal 8× speedup.

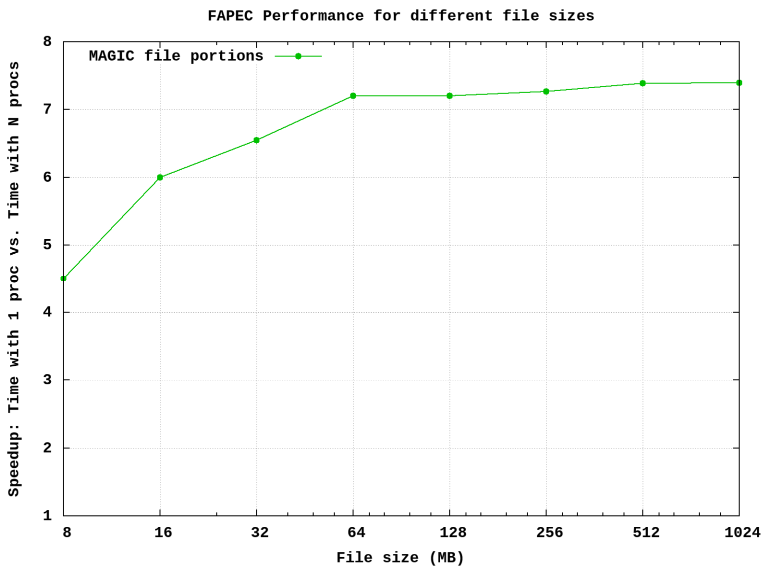

Fig. 7. Performance with 8 processes for different file sizes

Finally, Fig. 8 shows the results from scalability tests on 3 files from the Gaia, ALMA and MAGIC projects. The simulated Gaia image is 77 MB with 16-bit big Endian samples (achieving a ratio of 2.689. The files from ALMA and MAGIC are the same as shown in Fig. 6 with the same compression ratios. The chunk sizes selected are 1 MB for Gaia, 4 MB for ALMA and 8 MB for MAGIC. The test has been executed on a BSC MareNostrum III node again. The scalability test has used from 1 to 16 processes, showing normalized execution times (that is, the time with 1 processes divided by the time with N processes). Executing this test on a supercomputing node provides valuable information, as it indicates what we can expect from the compressor. For the three different types, ratios and volumes of data used we always see an excellent scalability. As we increase the number of processes to be used, the performance obviously improves. The speedup ratios are quite close to the ideal values, reaching 3.9× for 4 processes, 6.9× for 8 processes and 13.9× for 16 processes in the best case. In the worst case the speedup for 16 processes is 11.5, which is still an excellent result. It is worth mentioning that the 1.9GB file of MAGIC raw data was processed in just 3.1 seconds (having the file in the Linux memory buffers after repeated tests), which means that the core parallel compressor and the I/O routines are able to compress up to 600 MB/s (300 Msamples/s) using 16 threads. In other words, the only bottleneck here appears to be the disk access.

V. FUTURE IMPROVEMENTS

Despite of the excellent results presented here, we are working on several improvements to the current implementation. On one hand, the software approach (or API) chosen for the parallelization could be changed to *boost*, a well known and maintained library. On the other hand, the decompression approach could be more efficient if the parent-child communication is avoided. We also intend to study the possibility of using concurrency for the automatic configuration of the FAPEC parameters (such as the symbol size and pre-processing options). That is, a parallel compression test bench, getting the best rate or execution time for different configurations on a portion of the data, and using that for the rest of the file – thus avoiding the need of configuring FAPEC for each case.

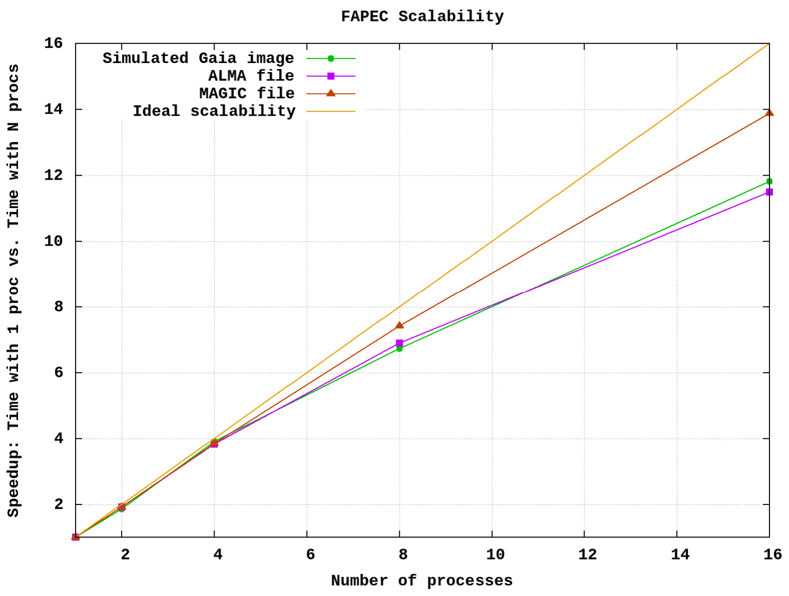

Fig. 8. Scalability of the Parallel FAPEC implementation for 1 to 16 processes

VI. CONCLUSIONS

The tests shown in this paper demonstrate that our Parallel FAPEC compressor is efficient and versatile, offering excellent compression ratios with an outstanding performance. The parallelization stage added to the original FAPEC implementation, based on the POSIX standard and also optimizing the memory reading and writing routines, has revealed an almost ideal scalability for up to 16 processes, reaching 14× for large files. Different chunk sizes can also be specified – which solves an existing limitation in the framework of the original compressor. A chunk size between 2 and 16 MB seems adequate for most cases. Tests on a 16-core supercomputing node have shown compression rates beyond 600 MB/s. Most important, we have demonstrated the feasibility and efficiency of the approach taken, namely, to compress a large file (or data stream, in general) in small and independent chunks, each on an asynchronous process just requiring minimum coordination to ensure a correct output sequence. It means that this approach could also be applied in a hardware implementation [2] in order to speedup further its performance. In summary, the Parallel FAPEC compressor is lightweight and has demonstrated an excellent versatility and performance in a variety of computing platforms, being suitable for multi-core processors onboard satellites.

VII. ACKNOWLEDGEMENTS

The images shown in this paper were obtained from GOES-NOAA, EUMETSAT and CCSDS. Also simulated Gaia images from the DPAC CU2-GIBIS simulator have been used. Finally, we are grateful to the MAGIC Collaboration for providing real raw data from their telescopes. This work was supported by the MINECO (Spanish Ministry of Economy) - FEDER through grant AYA2009-14648-C02-01, AYA2010-12176-E, AYA2012-39551-C02-01 and CONSOLIDER CSD2007-00050.

REFERENCES

[1] Portell, J., Villafranca, A.G., García-Berro, E.: *Quick outlier-resilient entropy coder for space missions*. Journal of Applied Remote Sensing (2010) 4, 041784, SPIE.

[2] Villafranca, A.G., Mignot, S., Portell, J., García-Berro, E.: *FAPEC in an FPGA: a simple low-power solution for data compression in space*, Satellite Data Compression VII (San Diego 2011), SPIE.

[3] Villafranca, A.G., Portell, J., García-Berro, E.: *Prediction Error Coder: a fast lossless compression method for satellite noisy data*. Journal of Applied Remote Sensing (2013) 7 (1), 074593, SPIE.

[4] Artigues, B., Portell, J., Villafranca, A.G., Ahmadloo, H., García-Berro, E.: *DWTFAPEC: image data compression based on CCSDS 122.0 and fully adaptive prediction error coder*. Journal of Applied Remote Sensing (2013) 7 (1), 074592, SPIE.