ETH zürich

High Performance Computing for Science and Engineering I

P. Koumoutsakos
ETH Zentrum, CLT E 13
CH-8092 Zürich

Fall semester 2018

Set 10 - MPI File I/O, Data Compression

Issued: November 30, 2018
Hand in (optional): December 10, 2018 23:59

Question 1: MPI I/O with a Custom Data Compressor

Data compression is an important topic in scientific computing and of particular interest for high performance applications. Compression of communication buffers can yield smaller communication overhead due to smaller message sizes, provided that the compression and decompression phases can be performed fast. Moreover, the footprint on the storage disk can be decreased considerably for large data sets if compression techniques are utilized. In this exercise we will focus on implementing our own distributed compression tool for (lossy) floating point data compression. In particular, we will design our own file format and identify it with the `.zfp` file ending. File operations are performed using the MPI standard. For the data compression we will use the `zfp`[1] compressor designed for *floating point* data. The compressor has already been implemented for you, your task is to implement the file protocol.

a) To get started with the exercise, you first must install the `zfp` library and download the data set we are going to work with. You can perform these tasks by executing the

```
make setup
```

command in the skeleton code directory. We are concerned with 2D data sets for this exercise. The compressor could be generalized to 3D data as well. Note that you can achieve higher compression rates if you take into account correlation of higher dimensional data.

In this task you implement the `write_data` function in the `mpi_float_compression.cpp` source file that can be found in the skeleton code directory. This function must implement the file protocol that is understood by our data compressor. In order to do this, we will store some necessary meta data in the file header which allows us to correctly read and decompress the data at some later time (our intent is to store the compressed data on the disk, similar to the `tar` or `zip` utilities). For a successful file read we need to store the compression tolerance `tol`, the global data dimensions `Nx` and `Ny` as well as the number of compressed blocks `Nb` that are stored in the file. This data is stored in a global file header at the beginning of the file. See the `FileHeader` structure in the source file. Additionally to the global file header, you will need meta data for each of the compressed blocks that are stored in the file. The `BlockHeader` structure in the source file describes this meta data and can be used for individual blocks. The `start` field is the byte offset in the file where the compressed block can be reached. The `compressed_bytes` contains the number

[1] https://computation.llnl.gov/projects/floating-point-compression

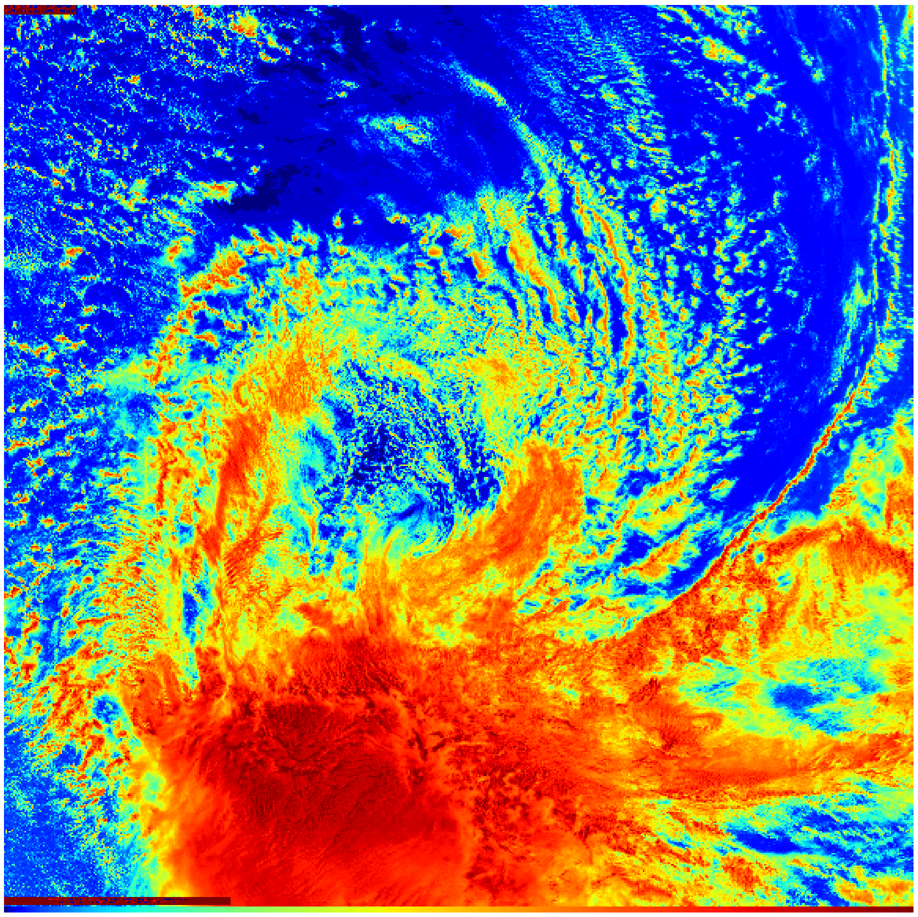

Figure 1: Cyclone test data

of bytes for the compressed block and `bufsize` stores the work buffer size required by the compressor to decompress the block. Once the required meta data has been determined, the headers and compressed blocks can be written to the file at their corresponding location. In general, you are free to design this protocol yourself, see the comments in the source file for a possible solution.

Implement such a protocol in the `write_data` function using MPI file operation routines. You may use collective or non-collective operations. Note that collective file operations allow for a higher degree of optimization which results in higher performance especially for large data volumes. You can run the code to test your implementation with the following command

```
make
mpirun -n <p> ./mpi_float_compression <tol> 4096 cyclone.bin.gz
```

where `p` is the number of processors and `tol` is the compression tolerance. For simplicity, we assume that each rank works on one compressed block and the input dimension is evenly divisible by the number of ranks. The tolerance guarantees that the error in the inflated compression is not greater than `tol` in the sense of the absolute error $|y - \tilde{y}|$, where y is the exact value and $\tilde{y}$ is the reconstructed value after decompression. The tolerance depends on the magnitude of the data that is subject for compression. The data of our input image ranges from 0 to 255. The `zfp` compressor is a lossy compressor, that is, information is lost in favor of higher compression rates. However, a tolerance of 0 results in *near-lossless* compression which means that the error in the reconstruction is within machine precision. The second argument `4096` is the dimension of the data and `cyclone.bin.gz` is the data file. The input file is a 4096×4096 NASA image of a cyclone shown in Figure 1. Note that the data could be any 2D floating point data set, here we work with a square image to see the effect of lossy compression. Because we use a floating point compressor, the image data has been converted to double precision while normally such images are represented using 8bit integers. Upon completion of this task, the compressor will generate compressed `.zfp` files of our input data.

b) In order to decompress the data files at a later time, you need to implement the inverse operation of the previous task and adhere to the file protocol that you have defined.

Implement the `read_data` function in the `mpi_float_compression.cpp` source file to provide a mechanism for reading your file format. The function allocates the work buffer of size `bufsize` which is then passed to the decompression routine. Inside `read_data` you fill this buffer with the compressed data that you read from the file. Note that the number of bytes in the compressed stream is guaranteed to be smaller or equal to `bufsize`. Moreover, the meta data parameter read from the file are passed to the caller by reference.

You can execute your code in the same way as in the previous task. The decompressed data from the `.zfp` file is further written to another `.bin.gz` file which can be used together with the provided Python script `print_png.py` to compare the original image to the one that went through the compression/decompression cycle of your application. For convenience, you can use the `run.sh <tol>` script on Euler to compile and run the code as well as to post-process the data with Python. Upon completion, the two images must be identical if you chose a tolerance that yields near-lossless compression. If the compression was lossy, the decompressed image will differ because of information loss. If the compressor reports an error that is larger than zero, the compression is lossy. You may play around with different values for the compression tolerance.

c) (Optional) Generate a plot of the compression rate (printed to `stdout` by the compressor) depending on the compression tolerance. You may compile your code with `make gzipout=false` to skip the generation of `.bin.gz` from your `.zfp` files. This allows you to run faster when writing a script for batch mode. Compare the observed compression rates to the compressed `cyclone.bin.gz` (GZIP[2], lossless) input file. Can you reach higher compression ratios with `zfp`? If so, is the compression lossy or near-lossless?

[2] https://en.wikipedia.org/wiki/Gzip

Question 2: Weak Scaling

We are interested in the scaling performance of our compressor implemented in the previous question. In this question, we will evaluate the weak scaling based on a number of measurements.

The weak scaling approach differs from the strong scaling by not keeping the problem size fixed but rather maintaining the execution time. This argument stems from the fact that if you have a very large computer, you generally do not want to solve a problem with fixed size faster but you want to solve that problem on a larger domain at roughly the same execution time relative to the problem on the smaller domain. We can quantify the efficiency of the weak scaling by relating the time required to solve the problem on one process versus the time required to solve the problem using p processes. Note that the *work per process* is kept constant when turning from one to p processes. Therefore, the weak efficiency E_w is computed by

$$E_w = \frac{t_1}{t_p}, \tag{1}$$

where t_1 is the time required for one process and t_p the time for p processes. If the two timings are the same, it means that the large problem *scales* perfectly up to p processes with an efficiency of $100\,\%$. In practice, the time t_1 (or t_p) are for example the time required to perform one time step for a particular problem that evolves in time. The time t_p can become larger than t_1 because of communication overhead among the p processes that is not perfectly hidden. Recall the technique of C/T-overlap (compute-transfer overlap) which must be utilized in order to achieve an acceptable weak scaling for your parallel code. Note that in contrast to the strong scaling approach, the weak scaling *is not* concerned about the serial fraction in a code. Instead, the weak scaling determines how well the *parallel fraction* of a given code scales among a given number of processes.

a) Assume that the data compressor from the previous question has been generalized such that any number of processors p work with any image dimension N (we are working with square images of dimension $N \times N$). The following table reports the execution time for different numbers of processors p and different input image sizes N.

	runtime [s]				
p	$N = 1024$	$N = 2048$	$N = 3072$	$N = 4096$	$N = 5120$
1	2.00	8.02	18.09	32.07	50.07
2	1.09	4.00	9.00	16.04	25.07
3	0.75	2.68	6.08	10.73	16.74
4	0.52	2.13	4.59	8.00	12.50
5	0.49	1.64	3.61	6.43	10.07
9	0.28	0.96	2.25	3.64	5.58
12	0.25	0.70	1.53	2.73	4.25
16	0.13	0.56	1.15	2.27	3.14
20	0.18	0.45	0.97	1.62	2.54
25	0.10	0.38	0.82	1.30	2.30

Draw the weak scaling plot from this data, using the value for $N = 1024$ at $p = 1$ as reference. Do not forget to label the axes.