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

60 points total

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

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

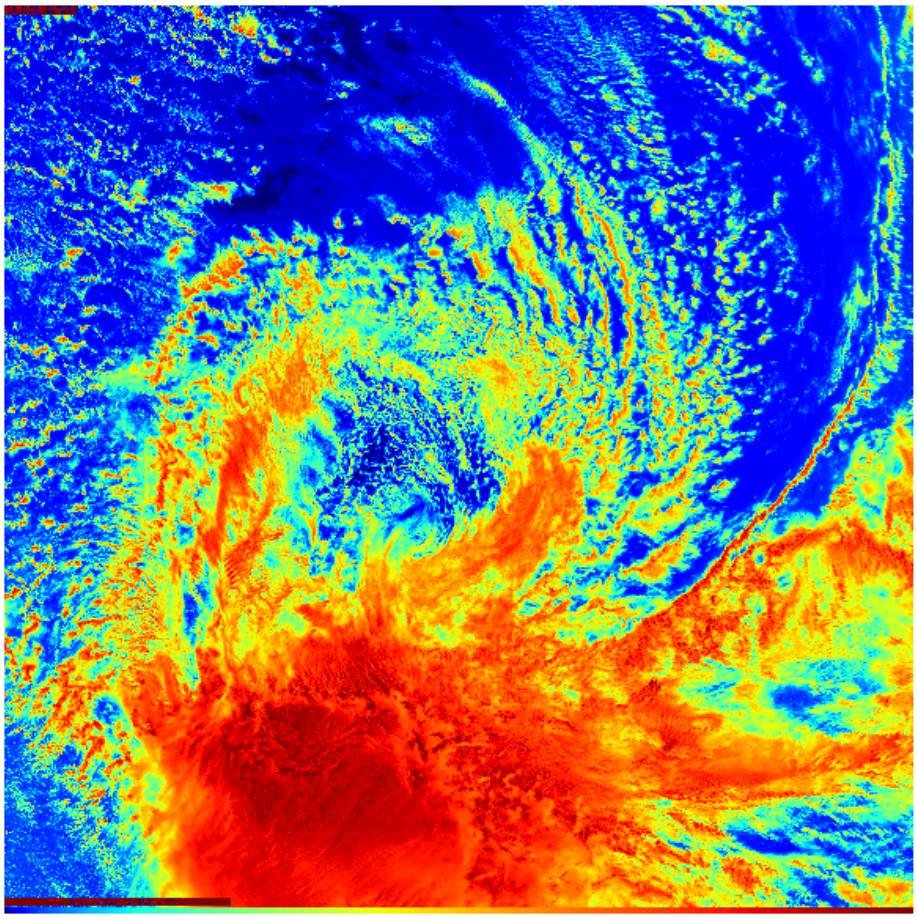


Figure 1: Cyclone test data

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 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.

The structure of the `.zfp` file has the following layout

```
[0:f]   FileHeader
[f:b0]  BlockHeader0
[b0:b1] BlockHeader1
[b1:b2] BlockHeader2
...
[bp:c0] cbuf0
[c0:c1] cbuf1
[c1:c2] cbuf2
...
EOF
```

where the numbers in brackets indicate the byte range in the file for the corresponding item. The `FileHeader` contains the global meta data which is required for the data reconstruction at a later time. For each compressed block, we need to store meta data in the file which is described by the `BlockHeader` structure. These headers describe the byte offset in the file where the compressed block data can be reached (`start`), the number of bytes for the compressed block data (`compressed_bytes`) as well as the size of the buffer that need to be allocated in order to reconstruct the compressed data (`bufsize`). Finally, the actual binary data for the compressed blocks is stored in the file at the `cbufX` location for the compressed block `X`. The first task is to determine the offset in the file where a particular MPI rank must write the data. Because the size of the compressed buffer can vary across ranks[2], the byte offset between compressed blocks may not necessarily be uniform. The correct way to do this by using the `MPI_Exscan` routine with an `MPI_SUM` operator.

```
size_t boffset = 0;
MPI_Exscan(&cbytes, &boffset, 1, MPI_UINT64_T, MPI_SUM, MPI_COMM_WORLD);
```

The result is the relative byte offset for the compressed data, returned in the `boffset` variable. Note that you could also use an `MPI_OFFSET` type here. The `boffset` variable must still be shifted by the number of bytes required for the meta data in all the headers in order to obtain an absolute byte offset.

```
boffset += hoffset;
```

[2]The size may vary because the data that is compressed (the image pixels) are not uniform across different ranks. The compressor makes use of data correlation for neighboring pixels and may choose to compress them differently to achieve optimal results.

where `hoffset` are the total number of bytes for all the header data. The `boffset` was the only missing piece required to setup the block headers, which is assigned to the `start` field of the corresponding header.

Next we can prepare the file by a call to `MPI_File_open`. We first write the headers, where the `root` rank additionally writes the global file header. We choose collective file operations because they will perform better for large data buffers. Because `root` is the only rank that writes the global file header, we can not use a collective routine for this task. When writing binary files, we are often not concerned with the particular type of the data. All we need to know is how many bytes a certain string of bytes is composed of and where we need to write it. For example, the compressed data buffer can not be assigned to type `double` of `int`. It simply is a string of bytes that only the `zfp` compressor knows how to interpret. Therefore, all file operations work with the `MPI_CHAR` type which is one byte long. This is also convenient to write the headers as it allows us to add more fields to the headers at a later time, if necessary, without touching the file write routines. Because we work with a single byte type is also the reason why the pointer to the compressed data has type `unsigned char`. Note that `char` would also work but is less intuitive as we only care about the length of the type and not whether a sign exists. The file write operations are performed on lines 120–126 in the solution code. Finally, you must close the file to free the file handle (the OS can only deal with a limited number of open files at a time) and conclude the file write.

- **5 points for offset computation with `MPI_Exscan`. 3 points for correct offset computation without using `MPI_Exscan`**
- **5 points for using `MPI_File_open` with `MPI_MODE_CREATE` and `MPI_MODE_WRONLY`. 4 points if any of these flags is missing.**
- **5 points if only one rank writes `FileHeader`. 3 points if all ranks write `FileHeader`.**
- **5 points if each rank writes its `BlockHeader` at the correct offset. 3 points if each rank writes `BlockHeader` at the same offset.**
- **5 points if each rank writes the correct number of bytes for the compressed block at the correct offset. 2 points if offset is wrong (e.g. offset shift for file headers not added).**
- **3 points for closing the file.**
- **2 points for a flexible implementation of the header writes using a byte representation with `MPI_CHAR` and the `sizeof` operator for the byte length of the `struct`'s.**

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.

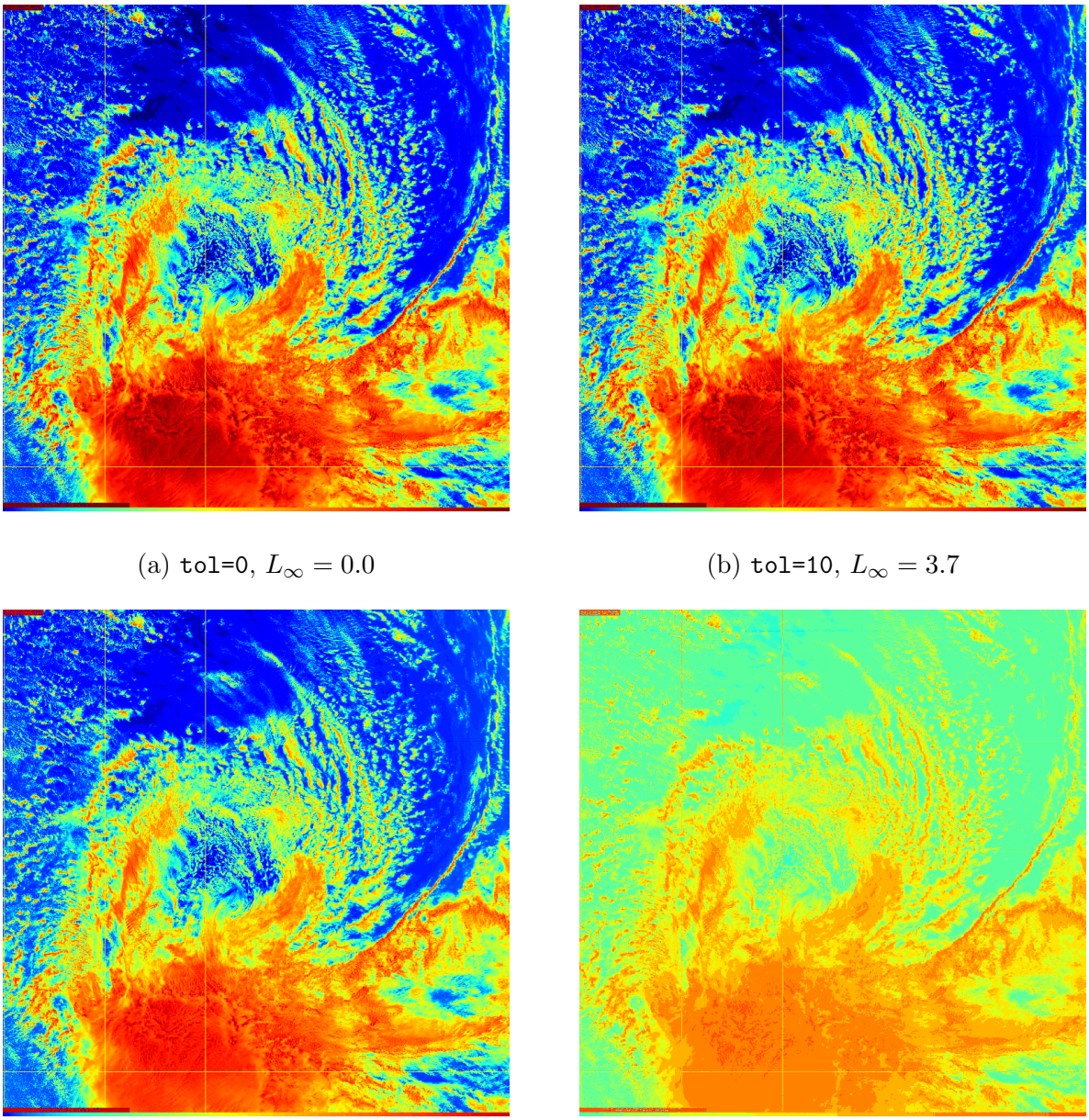
You can execute your code in the same way as in the previous task. The decompressed data from the `.zfp` file is further written to another `.bin.gz` file which can be used together with

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.

The file read follows the same protocol that has been implemented in the previous task. The operations are simply reversed. Here each rank must read the `FileHeader` data because each must initialize the decompressor, which in turn must know the parameter of the global data (not only the per rank data). Because this data must be returned to the caller, the `read_data` function is called with non-`const` argument references, which expect these values to be returned (otherwise these arguments would have to be passed by value). Therefore, after reading the `FileHeader` the data must be assigned to these function arguments. The calling rank does not know yet where to read the compressed data. Thus, each rank must read the `BlockHeader` for its assigned block. (A simplification for this exercise is that each rank is assigned only one block. This could be generalized to more than one block.) In order to read the compressed data from the file into the RAM, we must allocate a memory buffer which is then returned to the caller. The length of this buffer is obtained from the `BlockHeader` data (`bufsize`). Once the buffer is allocated (data type `unsigned char`) the file content can be read into the buffer. Finally, the file must be closed again. Figure 2 compares the image reconstruction using the lossy compressor for four different compression tolerances. The L_∞ error indicates the magnitude of detail that is lost during the lossy compression/decompression cycle. The tolerance specifies the upper bound of the $L_\infty = \max_i |y_i - \tilde{y}_i|$, where y_i is the exact pixel value and $\tilde{y}_i$ is the reconstructed pixel value after decompression. The actual L_∞ error may be less, which is indeed the case for all images shown in Figure 2. For tolerances around 10 no visible difference can be observed, even though there is information loss. For tolerances around 100 detail loss can be observed visually, while for larger tolerances information loss is severe which may not be tolerable for certain applications.

- **5 points for using `MPI_File_open` with `MPI_MODE_RDONLY`. 4 points if the `MPI_MODE_RDONLY` flag is missing.**
- **5 points if all ranks read `FileHeader`. 3 points if only `root` reads `FileHeader`.**
- **5 points if each rank reads its `BlockHeader` at the correct offset.**
- **5 points if the required header data is returned to the caller. 3 points if the assignment of the meta data to the function arguments is missing.**
- **5 points for allocating `bufsize` bytes of memory for `cbuf`. 2 points for wrong/missing buffer allocation.**
- **3 points if each rank reads the compressed data at the correct offset.**
- **2 points for closing the file.**

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



(a) `tol=0`, $L_\infty = 0.0$

(b) `tol=10`, $L_\infty = 3.7$

(c) `tol=100`, $L_\infty = 24.5$

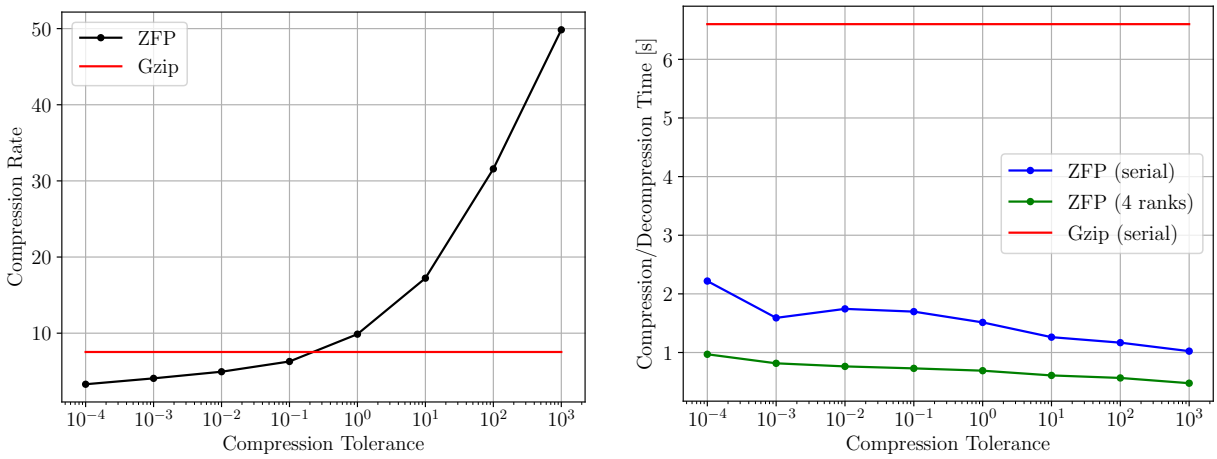
(d) `tol=1000`, $L_\infty = 374.0$

Figure 2: Image reconstructions using four different compression tolerances and the associated L_∞ error for the lossy compression.

rates to the compressed `cyclone.bin.gz` (GZIP[3], lossless) input file. Can you reach higher compression ratios with `zfp`? If so, is the compression lossy or near-lossless?

Figure 3 shows the compression rate and the time required for a compression/decompression cycle depending on the compression tolerance. The GZIP compressor is lossless and achieves a fixed compression rate of 7.53. ZFP manages a near-lossless compression for a tolerance of 0.1, which is slightly less compared to the GZIP compressor. However, for slightly lossy compression (tolerance of 10) ZFP achieves already twice the compression rate compared to

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



(a) Compression Rate

(b) Compression/Decompression cycle time

Figure 3: Compression rate (left) and time for compression/decompression cycle (right) depending on the compression tolerance.

GZIP. See Figure 2 for a visual comparison. Another important metric is the time required for a compression/decompression cycle. Here ZFP outperforms GZIP by a factor of $3\times$. Measurements for such compression/decompression cycles are shown in the right plot in Figure 3.

Question 2: Weak Scaling

15 points total

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

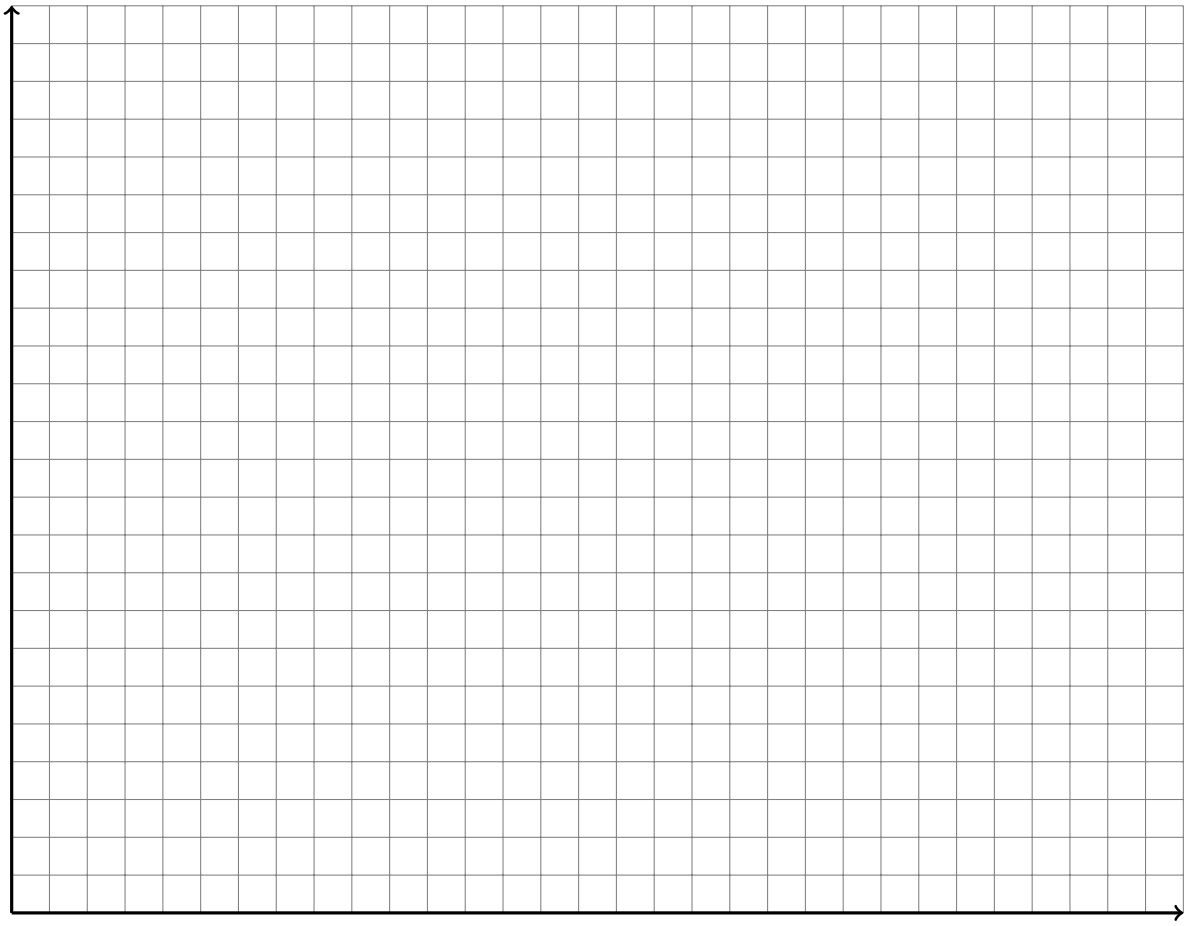
$$E_w = \frac{t_1}{t_p}, \tag{1}$$

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

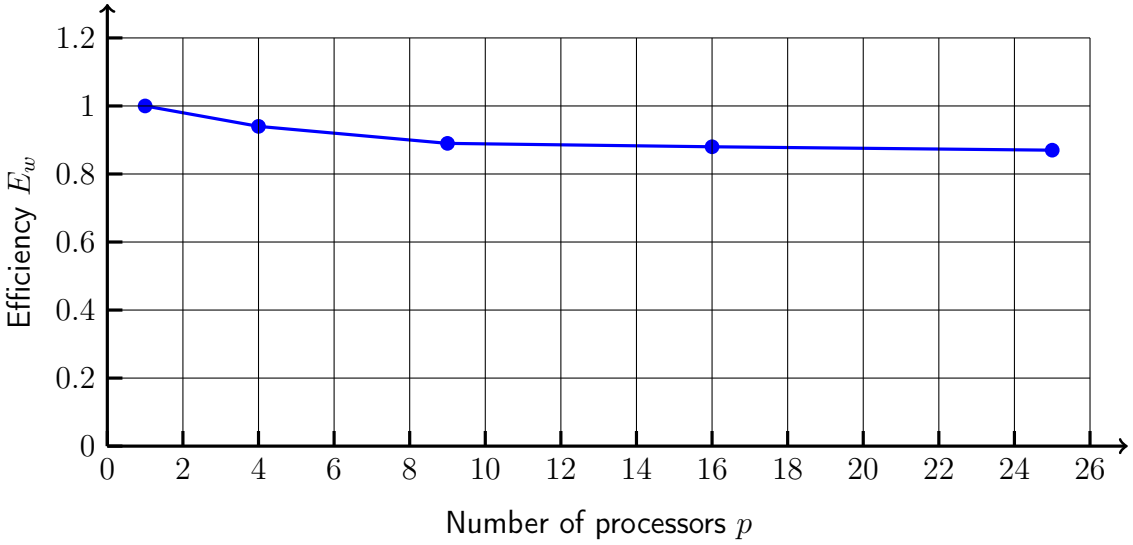
Draw the weak scaling plot from this data, using the value for $N = 1024$ at $p = 1$ as reference. Do not forget to label the axes.



The weak scaling efficiency is computed from Equation (1) based on the following values:

p	1	4	9	16	25
N	1024	2048	3072	4096	5120
t_p	2.0	2.13	2.25	2.27	2.30
E_w	1.0	0.94	0.89	0.88	0.87

Note that since we have an $N \times N$ image, increasing N by a factor of two means that the total work increases by a factor of four. The corresponding weak scaling plot is shown below.



- 2 points for each correct point p in the table (10 points total).
- 3 points for the drawing the correct graph.
- 2 points for labeling the axes.