

F. Wermelinger
Office: Pierce Hall 211

Homework 3

MPI synchronous and asynchronous communication, parallel MPI file I/O with compression

Issued: March 7, 2023 Due: March 28, 2023 11:59pm
--

Note this is a 3 week exercise. *Do not procrastinate the work.*

Please keep in mind that when the cluster is under high load with many queued jobs, your submitted job(s) will not execute right away. Depending on the cluster load, your job may execute only in a couple of hours (resources are limited). *Make sure you plan ahead.*

IMPORTANT: please do not include any compressed or decompressed image data from Problem 2 in your Gradescope submission zip!

Submission instructions: please see the homework submission section in the syllabus.^a

^a<https://harvard-iacs.github.io/2023-CS205/pages/syllabus.html#homework-submission>

Problem 1: Cannon's Algorithm (40 points)

Consider the matrix-matrix product

$$C = AB \tag{1}$$

with $A, B, C \in \mathbb{R}^{N \times N}$. We can compute this product with MPI in distributed memory using Cannon's algorithm. Let us partition A and B into $p \times p$ sub-matrices A_{ij} and B_{ij} . For example, consider the partition for $p = 3$:

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \\ B_{31} & B_{32} & B_{33} \end{bmatrix}, \tag{2}$$

where $A_{ij}, B_{ij} \in \mathbb{R}^{n \times n}$ with $n = N/p$ assuming N is evenly divisible by p . (Note that A_{ij} and B_{ij} are matrices **and not** scalars.) We now define the following $N \times N$ matrices:

$$\tilde{A}^{(1)} = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{22} & A_{23} & A_{21} \\ A_{33} & A_{31} & A_{32} \end{bmatrix}, \quad \tilde{B}^{(1)} = \begin{bmatrix} B_{11} & B_{22} & B_{33} \\ B_{21} & B_{32} & B_{13} \\ B_{31} & B_{12} & B_{23} \end{bmatrix}, \quad (3)$$

$$\tilde{A}^{(2)} = \begin{bmatrix} A_{13} & A_{11} & A_{12} \\ A_{21} & A_{22} & A_{23} \\ A_{32} & A_{33} & A_{31} \end{bmatrix}, \quad \tilde{B}^{(2)} = \begin{bmatrix} B_{31} & B_{12} & B_{23} \\ B_{11} & B_{22} & B_{33} \\ B_{21} & B_{32} & B_{13} \end{bmatrix}, \quad (4)$$

$$\tilde{A}^{(3)} = \begin{bmatrix} A_{12} & A_{13} & A_{11} \\ A_{23} & A_{21} & A_{22} \\ A_{31} & A_{32} & A_{33} \end{bmatrix}, \quad \tilde{B}^{(3)} = \begin{bmatrix} B_{21} & B_{32} & B_{13} \\ B_{31} & B_{12} & B_{23} \\ B_{11} & B_{22} & B_{33} \end{bmatrix}. \quad (5)$$

In general, the chosen arrangement of A_{ij} and B_{ij} in the matrices $\tilde{A}^{(k)}$ and $\tilde{B}^{(k)}$ allows to compute the partition $C_{ij} \in \mathbb{R}^{n \times n}$ by the combination

$$C_{ij} = \sum_{k=1}^p \tilde{A}_{ij}^{(k)} \tilde{B}_{ij}^{(k)}. \quad (6)$$

For example, in our 3×3 partition above

$$C_{11} = \tilde{A}_{11}^{(1)} \tilde{B}_{11}^{(1)} + \tilde{A}_{11}^{(2)} \tilde{B}_{11}^{(2)} + \tilde{A}_{11}^{(3)} \tilde{B}_{11}^{(3)} = A_{11}B_{11} + A_{13}B_{31} + A_{12}B_{21}. \quad (7)$$

Notice the pattern in $\tilde{A}^{(1)}$ compared to A : the first row of sub-matrices is the same as in A , the second row is shifted *once to the left*, the third row is shifted to the left twice and so on. That is, the i -th row of sub-matrices in $\tilde{A}^{(1)}$ is shifted $i - 1$ times *to the left*. Similarly, the j -th column in $\tilde{B}^{(1)}$ is shifted $j - 1$ times *upwards*. Furthermore, for $k > 1$, all elements in $\tilde{A}^{(k)}$ are shifted $k - 1$ times *to the right* and in $\tilde{B}^{(k)}$ all elements are shifted $k - 1$ times *downward* (relative to $\tilde{A}^{(1)}$ and $\tilde{B}^{(1)}$, respectively). Boundaries are periodic. The relation in equation 6 is called Cannon's algorithm. The p summation terms can be computed in an MPI code by performing $p - 1$ cycles, each involving the communication of sub-matrices A_{ij} and B_{ij} for some index pair ij .

Assume you have a grid of $p \times p$ processes, each process P_{ij} initially holds sub-matrices A_{ij} and B_{ij} in local memory. The toroidal communication pattern in Cannon's algorithm is shown in figure 1. At the end of Cannon's algorithm, process P_{ij} holds the solution C_{ij} in local memory. You only need to add code in the file [Cannon2D.cpp](#) to solve this problem. It might be helpful to look at the header file [Cannon2D.h](#) for additional attributes defined in the Cannon2D class.

Hint: You are expected to use MPI procedure calls in your implementations whenever possible.

Hint: You must aim at efficient implementations. Code that wastes resources cannot get full points.

Work in directory: ./code/p1

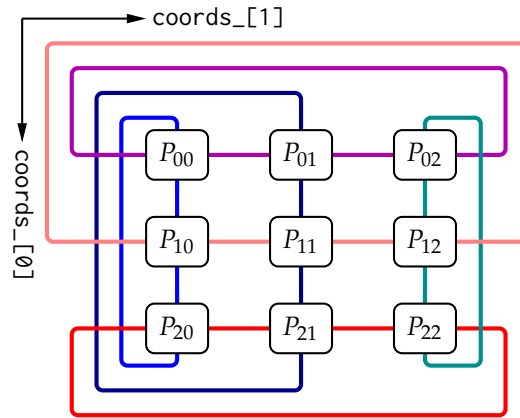


FIGURE 1: Toroidal network of MPI processes in Cannon’s algorithm. Example for a 3×3 processor topology.

a) *25 points*

Implement Cannon’s algorithm with **blocking** MPI communication (point-to-point routines). You can compile your code with the provided Makefile using

```
$ make
```

and a debug executable can be generated with

```
$ make debug
```

For debug builds, the rank local sub-matrices are of size 2×2 ($n = 2$). The given skeleton code produces the correct result when run with a single rank only. Your MPI implementation must run with any $p \times p$ grid of processes on the academic cluster. Before you start make sure you have loaded a recent compiler and the MPI library

```
$ module load gcc/12.1.0-fasrc01 openmpi/4.1.3-fasrc01
```

Once you have checked out a compute node, you can run your code by mapping one MPI rank to one core using, for example, the following commands

```
$ srun -n1 -c1 ./main
$ srun -n9 -c1 ./debug
```

where the first line would run your main executable using one MPI process only mapped to one core. The second line does the same with $3 \times 3 = 9$ ($p = 3$) MPI processes and using a debug executable if you have compiled it this way.

i) *5 points*

Complete the constructor `Cannon2D::Cannon2D` to enable MPI. You should complete the implementation of a Cartesian MPI topology, determine the four neighbors according to figure 1, obtain the coordinates i and j on process P_{ij} and write them into `coords_[0]` and `coords_[1]`, respectively. The constructor also defines a MPI datatype named `MPI_SubMatrix` for your convenience which you could use for later communication but you are not required to. See Section 5.1 in the MPI standard for more information.

ii) **5 points**

Implement the initial skewing of A and B in `Cannon2D::initial_skew()`. After this operation, process P_{ij} should have the sub-matrices $\tilde{A}_{ij}^{(1)}$ and $\tilde{B}_{ij}^{(1)}$ in its local memory. You may use the coordinates `coords_` you have determined in the previous task to determine the source and destination ranks for the communication. It is possible to solve this task with a minimum of *two* MPI operations (one for each matrix A and B). Any implementation using more than two MPI operations cannot receive full points.

Hint: The `MPI_Sendrecv_replace` routine could be used in this task.

Hint: See the comments in the code for how to print a sub-matrix to the standard output for debugging.

iii) **15 points**

Complete Cannon's algorithm in the method `Cannon2D::algorithmA()`. Use **blocking** MPI communication for the cyclic shifts to obtain the matrices $\tilde{A}_{ij}^{(k)}$ and $\tilde{B}_{ij}^{(k)}$ for $k > 1$. Rank local updates to C_{ij} can be performed by calling `gemm_()`, which computes the matrix update $C_{ij} = C_{ij} + A_{ij}B_{ij}$ with the data currently stored in `A_`, `B_` and `C_` (the array `C_` is initialized to zero). You may use any blocking MPI routines for communication

Hint: You may find `MPI_Sendrecv` or `MPI_Sendrecv_replace` most suitable.

Hint: The number of processes per dimension p is given by the `procs_per_dim_` class attribute in the code.

b) **15 points**

Implement a variation of Cannon's algorithm that *enables compute/transfer overlap* with **asynchronous** point-to-point communication. You may use `Arecv_` and `Brecv_` as temporary receive buffers in case you need it. Complete the `Cannon2D::algorithmB()` method in the file `Cannon2D.cpp` for this task.

Hint: The goal in this task is to hide communication overhead with useful computational work.

Problem 2: MPI File I/O with Data Compression (60 points)

Data compression is an important topic in computational science and of particular interest for high performance applications. Example applications include the compression of communication buffers to achieve smaller communication overhead due to smaller message sizes, provided that the compression and decompression can be performed fast and is lossless. The more obvious application is the reduction of file footprints on the storage disk which can be important for large data sets.

In this exercise we will focus on implementing our own distributed compression tool for (lossy) *floating point data compression*. In particular, you will design a compressed file format and identify it with the `.zfp` file ending. The parallel file operations are performed using MPI. The data compression tool we will use is called `zfp`¹ designed for floating point and integer data. The compression tools have already been implemented for you and the required library is installed on the academic cluster. Your task is to implement the parallel file protocol.

Hint: You are expected to use MPI procedure calls in your implementations whenever possible.

Hint: You must aim at efficient implementations. Codes that waste resources cannot get full points (e.g., multiple MPI calls where one would suffice or a different order of routines would result in fewer MPI calls).

Hint: See the comments in the code for additional hints.

a) 25 points

We are concerned with 2D data sets for this exercise. The compressor could be generalized to 3D data as well.

In this task you implement the `write_data` function in the file `mpi_compressor.cpp`. You do not need to add code in other files for this problem. The `write_data` function must implement the file protocol that is understood by our application. In order to do this, we need to store some necessary meta data at the beginning of the file (file header), which allows us to correctly read and decompress the data later. You can think of this application as another zip or tar utility but parallel and optimized for floating point data.

For a successful file read we need to store the following meta data in the file:

- The compression tolerance `tol` (defines how much information is lost during encoding and decoding cycles)
- The global data dimensions `Nx` and `Ny`
- The total number of compressed blocks `Nb` 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.

In addition 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. Each of these headers contain the following meta data:

¹<https://computing.llnl.gov/projects/zfp>

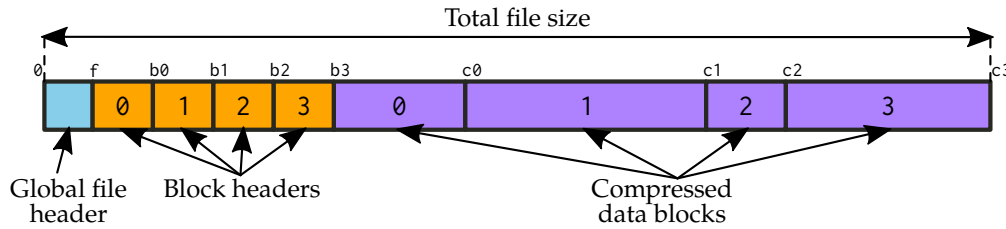


FIGURE 2: Possible layout for a file with 4 compressed data blocks.

- The start field is the byte offset in the file where the compressed block can be reached
- The compressed_bytes field contains the number of bytes of the compressed block (its size)
- The bufsize field stores the work buffer size required by the compression library to decompress the block later. *When you read data from the file, the buffer you store this data in must be allocated using bufsize.*

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 strategy. Figure 2 illustrates this proposed file layout for an example with 4 MPI ranks.

Implement such a protocol in the `write_data` function in the file `mpi_compressor.cpp` using MPI file I/O routines. You may use collective or non-collective operations. You can run the code to test your implementation with 4 MPI ranks each on one core (for example) using the following command

```
make clean main && srun -n4 -c1 ./main <tol> \
4096 ${SHARED_DATA}/hw3/cyclone.bin.gz test.zfp
```

where $\text{tol} \geq 0$ is the compression tolerance (see the `run.sh` job script as well). To be fair towards other with resource usage, you should get an interactive node with 4 cores which is sufficient for this exercise (note that you need a bit more memory for this task however):

```
salloc -N1 -c4 --mem=2048 -t 1:00:00
```

Alternatively, you can use the script `run.sh` in the handout files to submit jobs with

```
sbatch run.sh <tol>
```

For simplicity, we assume that each rank works on one compressed block and the input dimension of the 2D data is evenly divisible by the number of ranks. The tolerance guarantees that the error in the decompressed data 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 the 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.

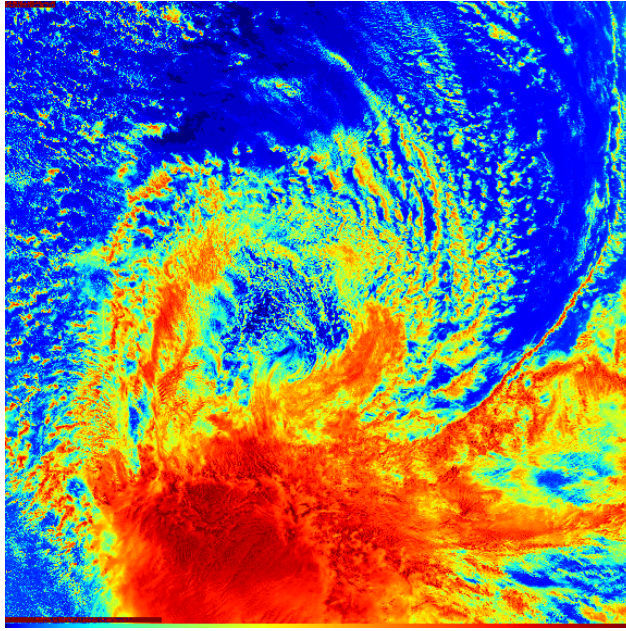


FIGURE 3: Cyclone test data

The zfp compressor is a lossy compressor which means 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 input data given in the third argument. The input data is a 4096×4096 NASA image of a cyclone shown in figure 3². Upon completion of this task, the compressor will generate the compressed output in the file specified as the last argument `test.zfp` above. A larger image of the earth can be found in `${SHARED_DATA}/hw3/` as well with dimension 9500×9500 if you want to try another image.

b) *25 points*

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 file `mpi_compressor.cpp` to provide a mechanism for reading your file format. The function allocates a work buffer in dynamic memory of size `bufsize` which is used to store the compressed file contents. The buffer is then returned from `read_data` function where the `main` function passes the data to the decompressor which reconstructs the original image with possible information loss.

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 (see the function signature).

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 to-

²Because we use a floating point compressor, the image data has been converted to double precision while normally such images are represented using 8 bit integers.

gether 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

```
sbatch run.sh <tol>
```

job script on the academic cluster to compile and run the code as well as to post-process the data with Python on a compute node.

Upon completion, the two images should be visually 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) **10 points**

Generate a plot with the compression ratio on the ordinate (printed to stdout by your application) as a function of the compression tolerance on the abscissa. Collect data for compression tolerances starting with 10^{-4} up to 10^3 (computing a data point for every decade is sufficient) and plot the abscissa in log-scale, do not forget to label the axes. You may compile your code with

```
make extra=-D_NO_GZIP_OUT_
```

to skip the generation of `.bin.gz` files from your `.zfp` files. This allows you to run faster when collecting the measurements. Collect your data using 4 MPI ranks.

Compare your compression ratios to the compression ratio of the `cyclone.bin.gz` input file (gzip³, lossless). The file can be decompressed using the `gunzip` utility. Plot the compression ratio for the gzip'ed input as a horizontal line in your plot, since it is a lossless compression it does not depend on a compression tolerance. Include the plot in your report and discuss the following items based on your findings:

1. Does your application allow for higher compression ratios than the lossless gzip utility?
2. If so, is the compressed data of your application lossy or near-lossless at the point where both tools operate at about the same compression ratio? Try to answer this question by arguing about the quality of your compression using the peak signal to noise ratio called PSNR⁴. This number is printed to standard output by the compressor as well. A value of infinity means no information loss which would correspond to a lossless (up to machine precision) reconstruction of the compressed data.

Hint: The compression ratio is the ratio between the uncompressed data and the compressed data. The higher the ratio, the smaller the memory footprint of the compressed data. High compression ratios are usually only achievable with lossy compression algorithms.

³<https://en.wikipedia.org/wiki/Gzip>

⁴https://en.wikipedia.org/wiki/Peak_signal-to-noise_ratio