CS 420, Parallel Programming, Spring 2023
MP3

Due Date: April 7rd, 2023, at Midnight (Discussion about this MP in class on March 28^{th})

1 Overview

The purpose of this assignment is for you to gain experience with MPI and stencil computations, iterative programs that update array elements based on a stencil (a fixed pattern). Due to the complexity of this MP's skeleton and its required runs, we recommend starting it early (especially if you have no prior experience with MPI).

Note: This MP involves submitting tasks that run on different nodes. We have observed that sometimes, results are not collected across 8 nodes. If this is the case, please try re-submitting the job to the cluster a few times. If the results for 8 nodes is still unavailable, it is okay to ignore the data point for 8 nodes while plotting the required graphs.

2 A Five-Point Stencil in MPI

2.1 Skeleton Programs

Pull the skeleton for MP3 from the `release` repository: `https://gitlab.engr.illinois.edu/sp23-cs420/turnin/<NETID>/mp3.git`

2.2 Five-Point Stencil

Five-point 2D stencils are made up from a point and its four neighbors. This pattern is typically used in iterative contexts, where the output of an iteration is used as the input for the next iteration.

In a distributed context, the computation is distributed across a grid with overlapping regions between processes (also referred to as Processing Elements (PEs)). We refer to the overlapping regions as ghost cells. These ghost cells must be large enough to contain all of the neighbors needed for an iteration, and must be updated with the values from other processes between iterations. See Figure 1 for a visual representation of the grid layout used in this MP. Note, each tile of the grid is padded to accommodate the ghost cells and some of the padding might go unused (see the processes at the boundary).

2.3 Your Task

We have provided a skeleton program that handles much of the setup for this assignment, which can be pulled from the `release` repository. **Your task is to implement the communication for exchanging the ghost cells between processes inside the function** `send_recv_ghosts` using non-blocking MPI calls. The skeleton maps the processes in a grid of dimensions `kGridRows` by `kGridCols`. Each process gets assigned a tile of data with dimensions $\frac{kWidth}{kGridCols}$ by $\frac{kHeight}{kGridRows}$. $kWidth$ and $kHeight$ refer to the size of the entire dataset. The variables `kGridX` and `kGridY` represent a process's (rank's) position in the process grid; **use the** `kGridX` **and** `kGridY` **information to determine whether a process has north, south, east, and/or west neighbors that require ghost data (and their corresponding ranks to send that data)**.

We have provided helper functions to copy ghost cells between send/receive buffers and a process's tile: `copy_rowbuf_out`, `copy_colbuf_out`, `copy_rowbuf_in`, and `copy_colbuf_in`. The "in" functions should be used for receives (copy data *into* the tile), and the "out" functions for sends (copy data *out of* the tile). The provided buffers `send_buffers` and `recv_buffers`, are appropriately sized for columns and rows of the process's tile respectively. Both the send_buffers and recv_buffers are of dimensions 4 * `bufferSize` (one for each direction). You will be able to access each buffer as send_buffers[0] and so on. You will have to copy the ghost cells from/to the `tile` onto the buffers and use the same for sending and receiving data.

You should also have to wait for the non-blocking calls to complete (refer to lecture 13, slides 65 and 66) before returning from the `send_recv_ghosts` function (Before the process computes its stencil). Figure 2a explains some basic cases depending on the position of the process on the process grid. Figure 2b covers another sample process grid layout thats tested in this MP. Note: The figures do not cover all the corner cases for waiting.
You are expected to use non-blocking MPI calls, i.e. `MPI_Isend` **and** `MPI_Irecv` **coupled with** `MPI_Wait` **or** `MPI_Waitall`**, for this assignment. Please see section 4.2 for API reference.**

? Question 1: *Strong scaling* refers to how the execution time varies with the number of processes (PEs) for a fixed, overall problem size. **For a single-node run** and an 32768 by 32768 workload with 16 iterations, plot *Execution Time vs. Number of processes* for process counts: 1, 2, 4, 8, and 16. Briefly comment on your program's *strong scaling*. Section 2.7 describes how these values are obtained after submitting the batch job.

? Question 2: *Weak scaling* refers to how the execution time varies with the number of processes for a fixed problem size per process. In other words, you increase both the overall workload and the number of processes correspondingly, keeping the workload per process constant. **For a multi-node run**, plot *Execution Time vs. Number of processes* for a fixed tile size of 8192 by 8192 per process for process counts: 4, 16 and 64; include data labels for each point indicating the overall workload size. Briefly comment on your program's *weak scaling*. Section 2.7 describes how these values are obtained after submitting the batch job.

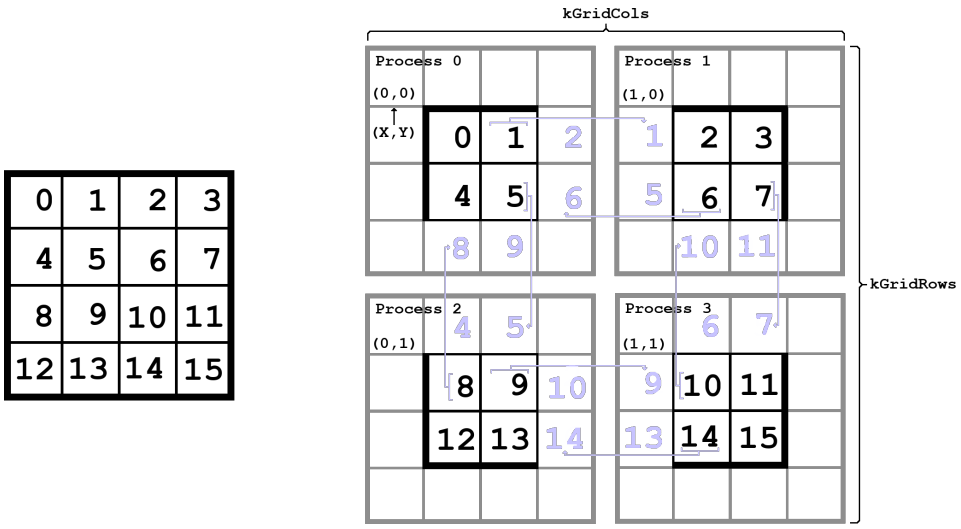
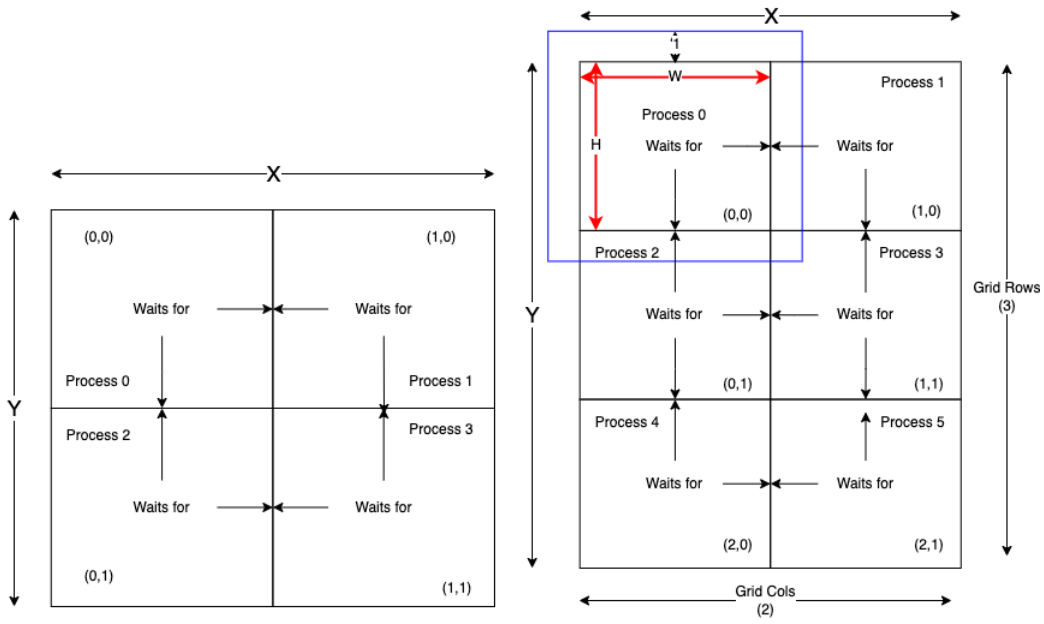


Figure 1: Process Grid layout used by this MP's skeleton.



(a) (b)

Figure 2: Some example process grid layouts used in this MP

Question 3: For a 65536 by 65536 workload with 16 iterations, how did running your program with multiple nodes affect its parallel performance? Plot *Execution Time vs. Number of processes* for process counts: 16, 32, 64, 128, and 256. Briefly comment on your program's *strong scaling* which occurs across multiple nodes.

Question 4: Do you think the program will work if blocking send (MPI_Send) and blocking receive (MPI_Recv) is used instead? Kindly elaborate on your reasoning as to why you believe that using blocking calls may or may not be effective. You do not need to implement the same to answer this question. Please refer to slide 59 from lecture 13 posted on Piazza.

2.4 Skeleton Notes

This MP's skeleton program is rather complex, so it may be worth taking some time to understand it before diving into the assignment. Unlike the other skeleton's we have provided – this MP's verboseness is determined at compile-time using a macro variable, `VERBOSE_LEVEL` X.Where: $X = 0$ produces no output (use this for performance analysis), $X = 1$ prints information for correctness checking, and $X = 2$ prints additional information about the program. Since no communication is necessary in single process runs, these runs can be used as the basis for correctness checking. Note, your program's output should not vary with valid process counts for a fixed workload size and number of iterations.

The expected usage of the program is `./stencil <height> <width> <numIts>`, specifying the overall width and height of the workload and the number of iterations to run for. This program assumes that the workload (dimensions) will always be evenly divisible by the number of processes.

2.5 Building on the Cluster

The following command is used to load MPI into the cluster's environment:

```
module load intel/18.0
```

Among others, this will add the program `mpiicc` to the `PATH`, used to compile MPI programs on the campus cluster. To build, you can run the provided script:

```
./scripts/compile_script.sh X
```

where the X is the desired verbosity and with the appropriate X value (0, 1, 2). The parameter is optional and can be left out for a default verbosity of 1. Alternatively, we have provided a `cmake` file for you to use to manually build your program. To build, enter your root directory and run:

```
mkdir build
cd build
cmake ..
cmake --build .
```

To specify verbosity, change

`cmake ..`

from the given commands to

`cmake -DCMAKE_C_FLAGS="-DVERBOSE_LEVEL=X" ..`

2.6 Running MPI Programs on the Cluster

You should not use the login node directly to execute such larger dimensions as its a shared node. Please use the script `test.sh` found in the `tests` folder. This script will build and test your program. We have ensured that this program tests different smaller process grid layouts. If you are in the root folder, you can run this script with

`./tests/test.sh`

Apart from running the `scripts/test.sh` to verify correctness, we can also submit interactive jobs to the cluster. This is achieved via the `srun` command. For example, the following command will run the program `stencil` (with appropriate arguments) on 4 process and one node:

```
srun --mpi=pmi2 --nodes 1 --ntasks 4 ./stencil 64 64 4
```

2.7 Benchmarking on the Cluster

If you are in the base folder, the following command will submit all of the necessary tests.

`./scripts/submit_batch.sh`

This will make two folders:

- `results`: containing all of the results from your program
- `batchFiles`: the output and error files from all of the jobs

Inside of the `results` folder, you can find a series of files:

- `strong_X`: result for the strong scaling with `X` process with fixed problem size
- `across_node_weak_X_Y`: result for the weak scaling with `X` process across Y nodes. (8 may not produce output, so those counts are optional)
- `across_node_strong_X_Y`: result for strong scaling with `X` process across Y nodes (8 may not produce output, so those counts are optional)

2.8 Adding test logs

Additional debug logs can be added to `stencil.c`. For example:

```
#if VERBOSE_LEVEL >= 2
printf("Inside send_recv_ghosts");
#endif
```

Once added please re-compile the program using

`./scripts/compile_script.sh 2`

You can then run the program on the cluster by following instructions from section 2.6 (`srun`)

3 Submission Guidelines

3.1 Expectations For This Assignment

The graded portions of this assignment will be your stencil program and answers to the short answer questions.

3.1.1 Points Breakdown

- $4 * 11pts.$ — Each Short Answer Question
- $56pts.$ — Correctness of your Stencil Program

3.2 Pushing Your Submission

Follow the git commands to commit and push your changes to the remote repo. Ensure that your files are visible on the GitHub web interface, your work will not be graded otherwise. Only the most recent commit before the deadline will be graded.

3.2.1 Files to change

- `stencil.c`

4 Appendix

4.1 Terminology

- kGridRows: The number of process rows there are in the process grid
- kGridCols: The number of process columns there are in the process grid
- kWidth: Total width of the entire matrix across all processes
- kHeight: Total height of the entire matrix across all processes
- kNumIts: Number of iterations to run the stencil
- w: width of one process grid item
- h: height of one process grid item
- kRank: The same as process number
- kNumPes: Total number of processes
- kGridX: x-coordinate of the process in the process grid.
- kGridY: y-coordinate of the process in the process grid.

You do not need to write to any of mentioned variables, but you do need to read from them. There is no special variable or function to access any of these, i.e. inside send_ghosts, you can do

```
int i = kGridX + 5;
int j = kGridRows + kGridCols;
```

4.2 Prototypes to be used for this MP

1.
```
MPI_ISend(void* buf,
int count,
MPI_Datatype datatype,
int destination_rank,
int tag,
MPI_Comm communicator,
MPI_Request *request)
```

   - ***buf*** A pointer to the buffer that contains the data to be sent.
   - ***count*** The number of elements in the buffer. If the data part of the message is empty, set the count parameter to 0.
   - ***datatype*** The data type of the elements in the buffer. For example: MPI_FLOAT
   - ***destination_rank*** The rank of the destination process within the communicator that is specified by the communicator parameter.
   - ***tag*** The message tag that can be used to distinguish different messages. (For the purpose of this MP, the tag we use is the current iteration value. This is done to distinguish between the different iterations)
   - ***communicator*** The handle to the communicator.
   - ***request [out]*** On return, contains a handle to the requested communication operation.

2.
```
MPI_IRecv(void* buf,
int count,
MPI_Datatype datatype,
int destination_rank,
int tag,
MPI_Comm communicator,
MPI_Request* request)
```

   - ***buf*** A pointer to the buffer that is used to store the data that is being received
   - ***count*** The number of elements in the buffer. If the data part of the message is empty, set the count parameter to 0.
   - ***datatype*** The data type of the elements in the buffer. For example: MPI_FLOAT
   - ***destination_rank*** The rank of the sending process within the specified communicator.
   - ***tag*** The message tag that can be used to distinguish different messages. (For the purpose of this MP, the tag we use is the current iteration value. This is done to distinguish between the different iterations)
   - ***communicator*** The handle to the communicator.
   - ***request [out]*** On return, contains a handle to the requested communication operation.

3. `MPI_Waitall(int count, MPI_Request requests[], MPI_Status)`

- ***count*** The number of entries in the array_of_requests parameter.
- ***array_of_requests*** An array of MPI_Request handles of outstanding operations.
- ***array_of_statuses*** MPI_STATUSES_IGNORE can be used for this MP.

4. `copy_rowbuf_out(int w, int h, float tile[w + 2][h + 2], int row_no,float send_buffer[]);`

   This function copies the contents of row `row_no` from the `src` tile onto send_buffer

5. `copy_colbuf_out(int w, int h, float tile[w + 2][h + 2], int col_no, float send_buffer[]);`

   This function copies the contents of column `col_no` from the `src` tile onto send_buffer

6. `copy_rowbuf_in(int w, int h, float tile[w + 2][h + 2], int row_no, float recv_buffer[]);`

   This function copies the contents of recv_buffer onto row `row_no` of the tile

7. `copy_colbuf_in(int w, int h, float tile[w + 2][h + 2], int col_no, float recv_buffer[]);`

   This function copies the contents of recv_buffer onto column `col_no` of the tile

   Apart from these definitions, we have also given hints in the skeleton code, please check `stencil.c`.