Parallel Computing, 2023S

Assignment 2: Heat Equation (2D) with OpenMP and MPI

SOLVING HEAT EQUATION IN 2D

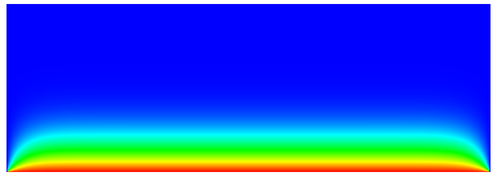
The heat equation is a Partial Differential Equation

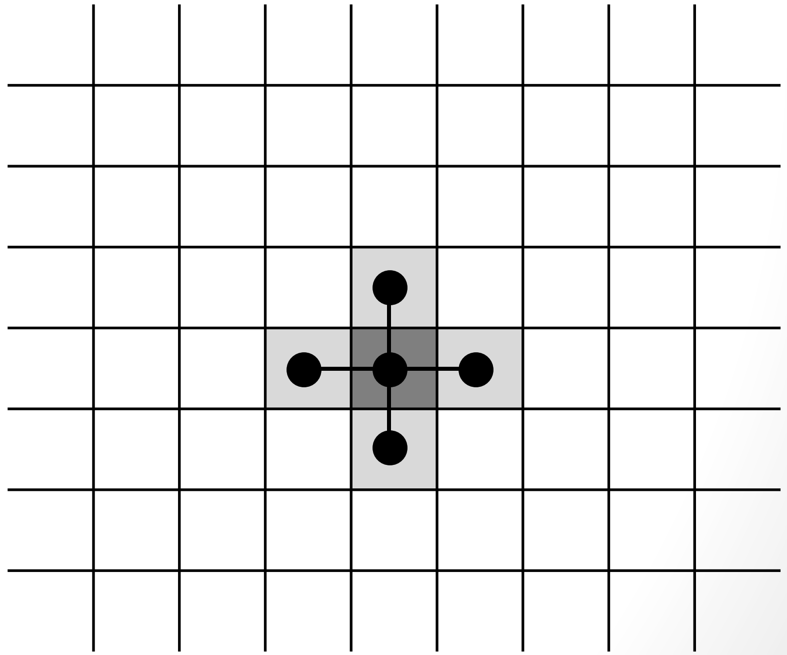
$$\frac{\partial u}{\partial t} = \alpha\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right)$$

Using Jacobi iterative method:

$$v_{m.l}^{n+1} = \frac{1}{4}\left(v_{m+1,l}^{n} + v_{m-1,l}^{n} + v_{m.l+1}^{n} + v_{m.l-1}^{n}\right) - \frac{h^2}{4}\underset{\text{Zero}}{f_{ml}}$$

- NxM Matrix
- Five-point stencil
- Calculation of averages

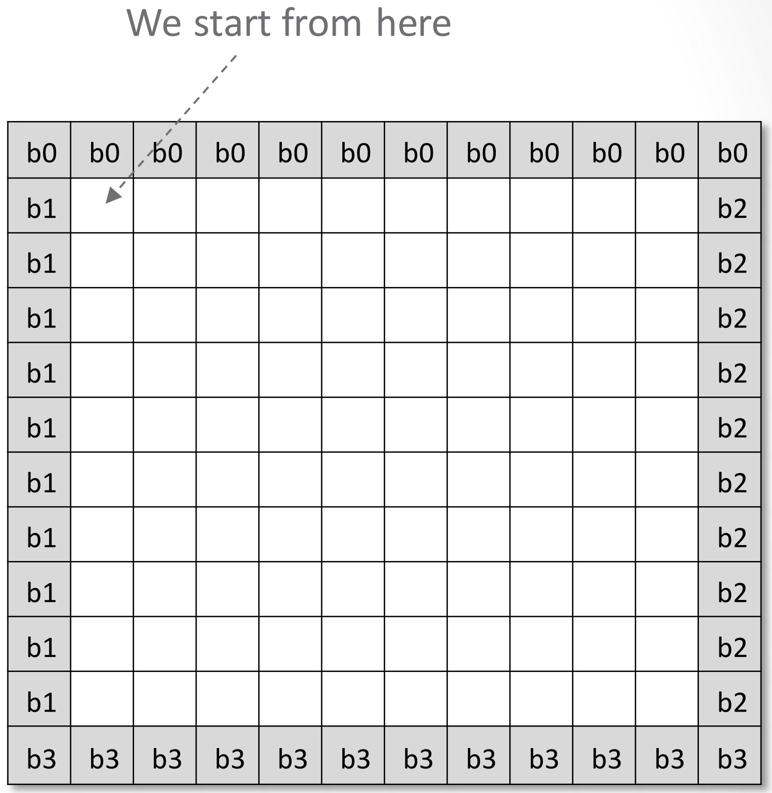

Example output


Five-point stencil

CORE OF THE SEQUENTIAL CODE

```c
//...
iteration_count = 0;
do
{
  iteration_count++;
  diffnorm = 0.0;

  /* Compute new values (but not on boundary) */
  for (i = 1; i < M - 1; ++i) {
    for (j = 1; j < N - 1; ++j) {
      W[i][j] = (U[i][j+1] + U[i][j-1] + U[i+1][j] + U[i-1][j]) * 0.25;
      diffnorm += (W[i][j] - U[i][j]) * (W[i][j] - U[i][j]);
    }
  }

  // Only transfer the interior points
  for (i = 1; i < M - 1; ++i)
    for (j = 1; j < N - 1; ++j)
      U[i][j] = W[i][j];

  diffnorm = sqrt(diffnorm);

} while (epsilon <= diffnorm && iteration_count < max_iterations);
//...
```

exit criteria



b0	b0	b0	b0	b0	b0	b0	b0	b0	b0	b0	b0
b1											b2
b1											b2
b1											b2
b1											b2
b1											b2
b1											b2
b1											b2
b1											b2
b1											b2
b1											b2
b3	b3	b3	b3	b3	b3	b3	b3	b3	b3	b3	b3

Boundary conditions

b0 = 0.02, b1 = 0.05, b2 = 0.1, b3 = 0.2

Note: Note that in most cases the code will reach the maximum number of iterations with the current setup

PART 1: HEAT 2D WITH OPENMP

Implement a parallel version of the 2D heat equation solver with OpenMP by:

1. **Identifying parallelism opportunities:**
   - Which execution hotspots of a2.cpp can (or cannot) be parallelized?

2. **Using OpenMP to incrementally parallelize the identified hotspots:**
   - Always use the **most adequate** OpenMP constructs / clauses:
     - Less OpenMP code is better code! (except if you can get better performance)
   - Do **not modify** the code provided unless strictly necessary:
     - Algorithm changes are not required to ensure correction nor performance.
   - Make sure your code stays OpenMP **stays flexible**:
     - Avoid hardcoded clauses and OpenMP routines, use environment variables instead!

3. **Ensuring your parallel version produces the same result as the sequential:**
   - Respect data-dependencies and employ synchronization constructs if required!

FURTHER REQUIREMENTS WITH OPENMP

To get full score in the implementation part provide a single OpenMP program such that:

1. The data-scope of all variables (in the extent of the parallel region) is implicitly set:
   - You might have to move/create variables.
2. The number of threads and scheduling strategy must be set via environment variables:
   - e.g., `OMP_NUM_THREADS=...` and `OMP_SCHEDULE=…`
3. Your implementation can compile without `-fopenmp` flag:
   - If you do not provide `-fopenmp` during compilation your code will execute sequentially.

Be mindful of common OpenMP mistakes:

- Typically, OpenMP work-sharing constructs cannot be nested.
- Nested parallel regions are disabled by default:
  - set `OMP_NESTED=true` to enable it.

EXPERIMENTS AND COMPILATION WITH OPENMP

1. **Compile your OpenMP code (`a2-omp.cpp`) with:**
   - `g++ a2-omp.cpp -O2 –fopenmp -lm –o a2-omp`

2. **Try** (at least) **the following code parameters:**

```
a.  <ENV-VARS> srun --nodes=1 ./a2-omp --m 2688 --n 4096 --epsilon 0.01 --max-iterations 1000
b.  <ENV-VARS> srun --nodes=1 ./a2-omp --m 2688 --n 4096 --epsilon 0.01 --max-iterations 2000
c.  <ENV-VARS> srun --nodes=1 ./a2-omp --m 1152 --n 1152 --epsilon 0.01 --max-iterations 1000
```

3. **Run** (at least) **using the following configurations:**
   - 1 node using 1, 2, 4, 8, 16 and 32 OpenMP threads

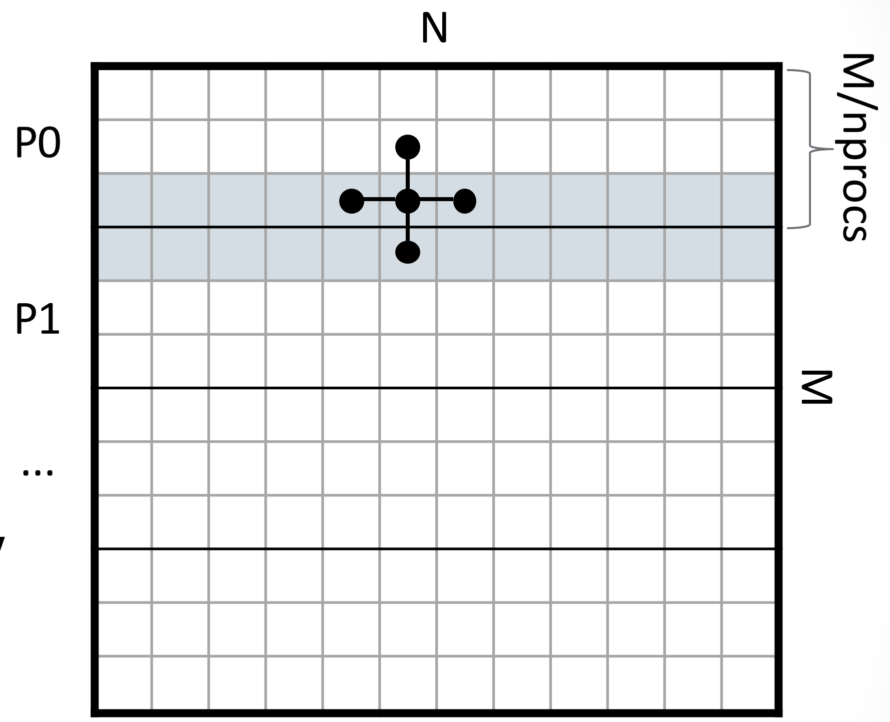
Achieve a speedup of 8+ with at least one configuration on the Alma system and observe the speedup when using other configurations

- Sequential runtimes on Alma: 2.a: ~50 seconds, 2.b: ~110 seconds, 2.c: ~5 seconds

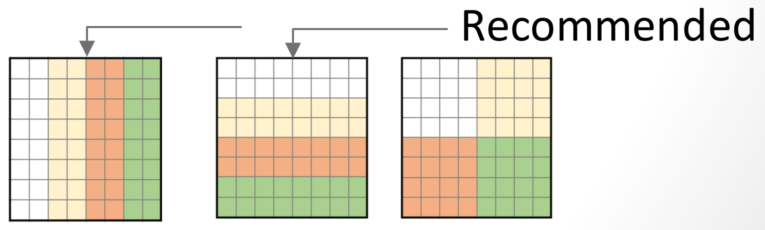
HEAT2D WITH MPI

Realize an efficient parallel implementation of the serial code for iteratively solving heat equation using MPI

1. **Distribute data over all processes**
   - Horizontal blocks (you can also choose vertical or 2D)
   - Initialize in a distributed fashion (!)
   - Each process gets only a part of the matrix
   - Support (M % nprocs != 0)
     - e.g., have larger last block
2. **Use point-to-point communication to communicate the overlapping regions**
3. **Make sure that you get the termination criteria correctly**
   - The results must be the same as if executed with the sequential code
4. **Verify/compare to the sequential version**
   - **Use collectives to transfer data to rank 0, and then compare!**
     - Also measure the time required to collect data on rank 0
   - **Results must be the same!**



Overlapping (halo/ghost) regions



Other configurations

Hints: First support M%nprocs == 0 and then improve

Useful MPI routines: MPI_Isend/Recv, MPI_Allreduce, MPI_Gather, MPI_Gatherv

DATA DISTRIBUTION (HORIZONTAL)

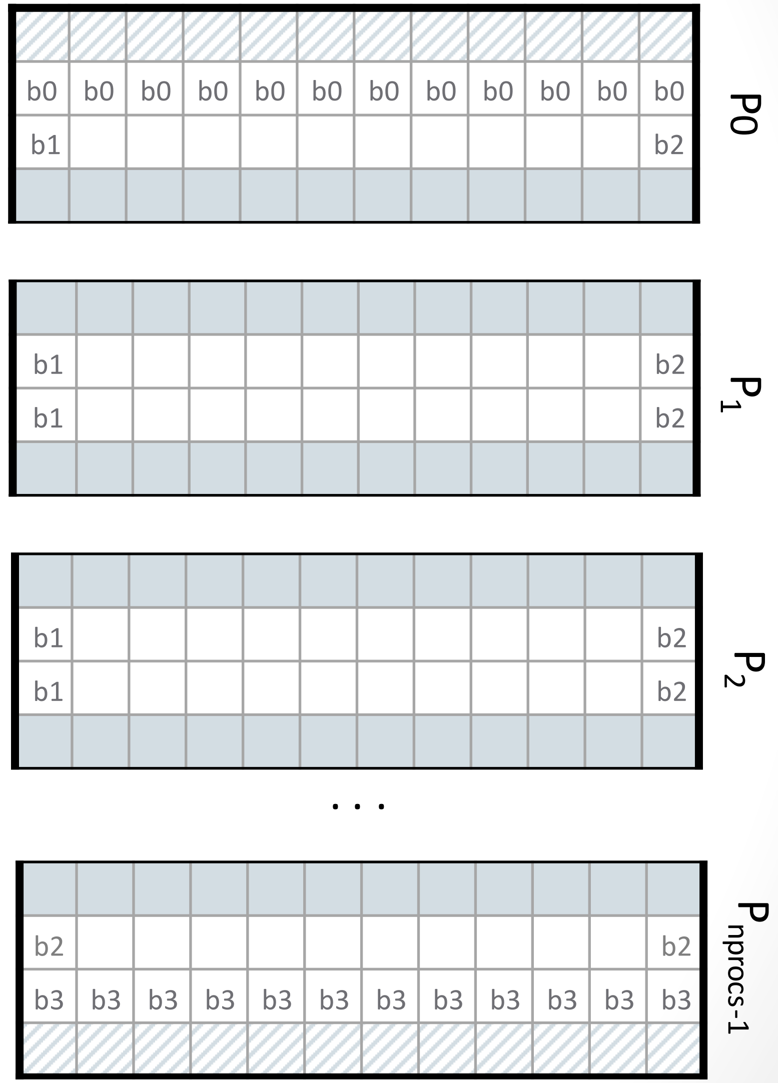
Each process gets a part of the MxN matrix

- e.g., each process gets ~M/nprocs (the # of processes) and additional rows for transfering the overlapping rows

Neighbouring blocks need to exchange data so that the 5-point stancil can be computed

Example M=8, nprocs=4

- Local matrix size → 8/4 = 2 (+ 2 for the halo region)
- Blue rows = extra rows to handle the “overlapping/halo” region
- You can have same local size for all process
  - Pattern fill ignored rows for this case
  - Tends to be easier
- Alternativelly, you can have smaller local sizes for the top and bottom processes (tends to be more complex)


b0 b0 b0 b0 b0 b0 b0 b0 b0 b0 b0 b0
b1
b2
P0
b1
b2
b1
b2
P1
b1
b2
b1
b2
P2
. . .
b2
b2
b3 b3 b3 b3 b3 b3 b3 b3 b3 b3 b3 b3
Pnprocs-1

COMMUNICATION

Red arrows

- Sends from process below to process above
- e.g,: MPI_Send/ISend from the process below, sending its second row (the first after the padding) to process above → e.g., MPI_Send(&U[1][0], N, …)

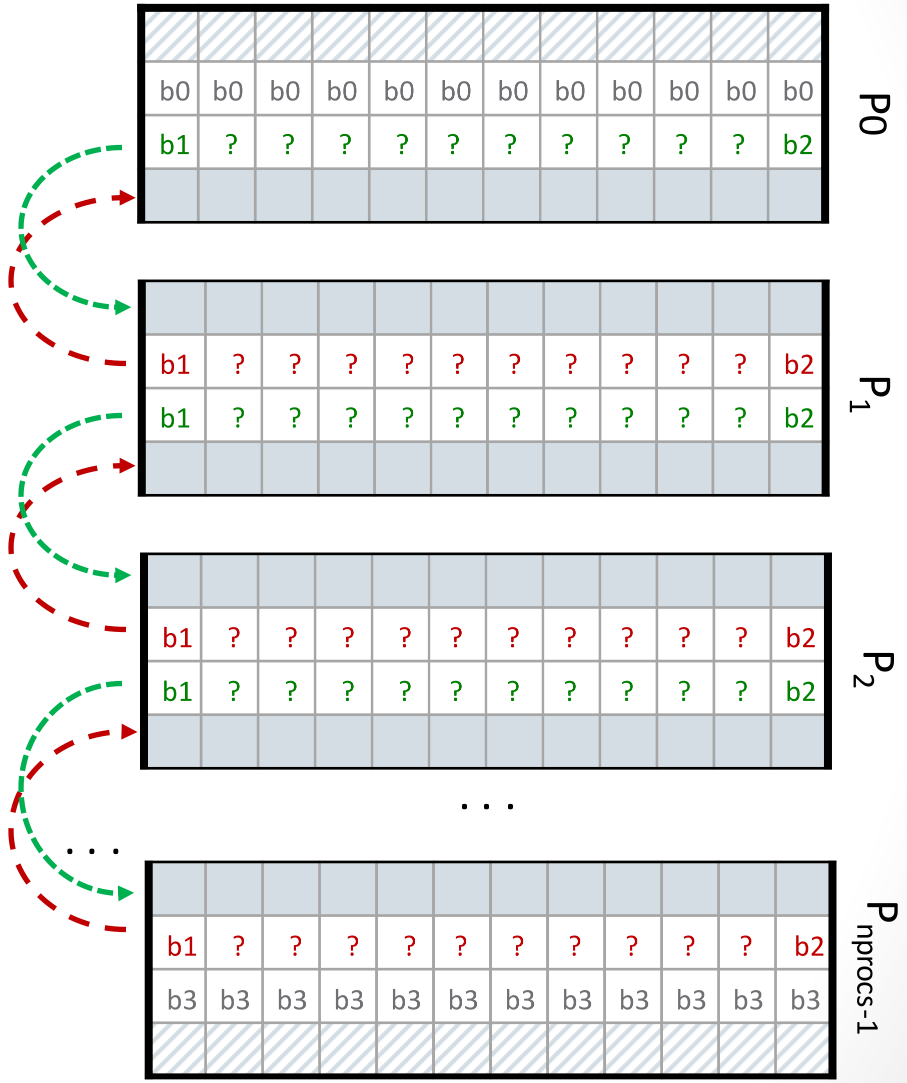
Green arrows

- A message from process above to process below
- E.g., using MPI_Send to send its last row (befor the padding) to the process below

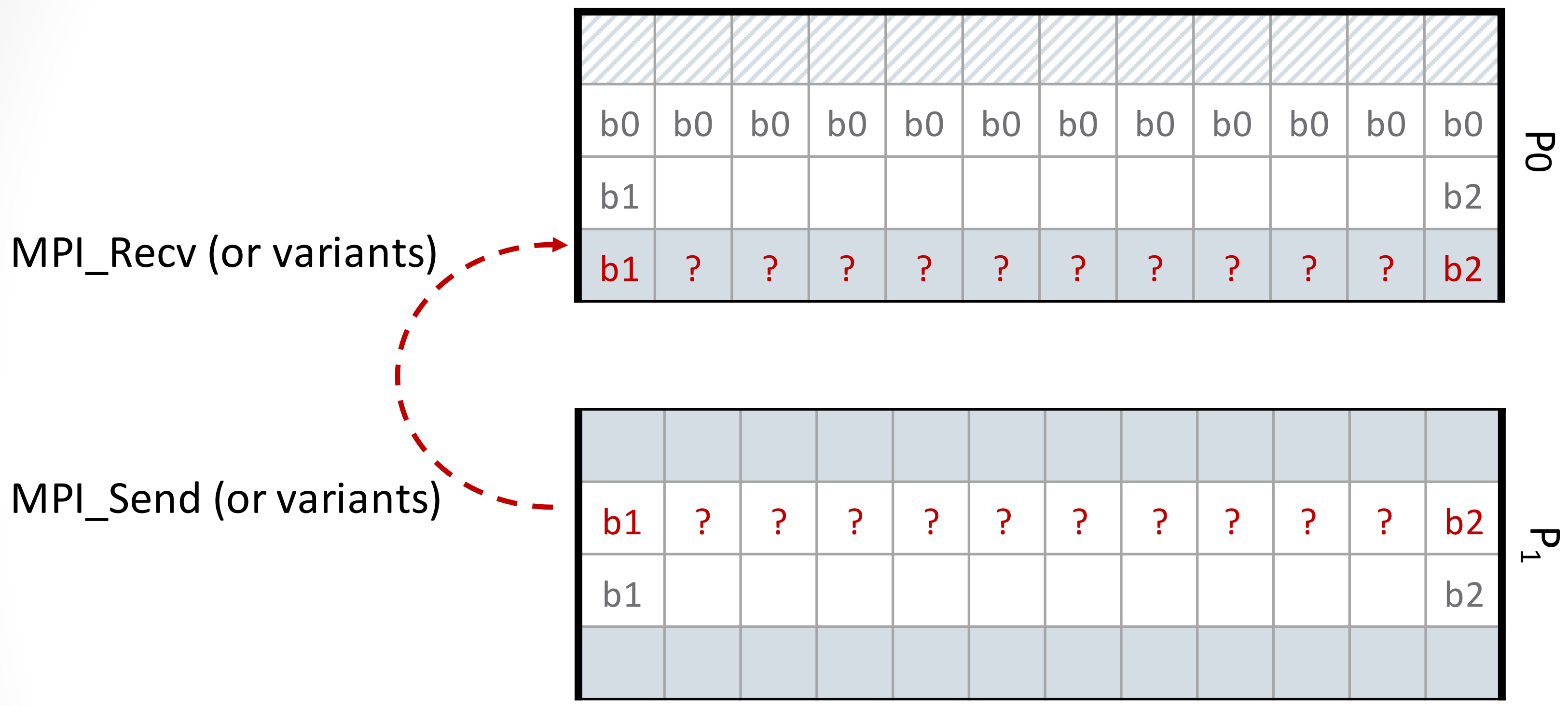
You need MPI_Send and MPI_Receive on each end

- Four calls for each neighbour exchange
- Basically 3 different cases
  - **Top** communicating only with the process below
  - Each **middle** process with both above and below
  - **Bottom** communicating with the process above

Note: can also be simplified, and implemented with MPI_Send/Recv, MPI_Isend/Irecv variants and MPI_Sendrecv, …


b0 b0 b0 b0 b0 b0 b0 b0 b0 b0 b0 b0
b1 ? ? ? ? ? ? ? ? ? ? b2
P0
P_1
P_2
. . .
$P_{nprocs-1}$
b3 b3 b3 b3 b3 b3 b3 b3 b3 b3 b3 b3

COMMUNICATION :: SINGLE MESSAGE


MPI_Recv (or variants)
MPI_Send (or variants)
P0
P1
b0 b0 b0 b0 b0 b0 b0 b0 b0 b0 b0 b0
b1 b2
b1 ? ? ? ? ? ? ? ? ? ? b2
b1 ? ? ? ? ? ? ? ? ? ? b2
b1 b2

Example:

- **P1** sends its first row to **P0**, **P0** receives the row and puts it int its last row (the additional row)
- Point-to-point communication
- E.g., with MPI_Send, MPI_Isend, MPI_Recv, MPI_Irecv, MPI_Sendrecv (avoid deadlocks!)
- *the values in cells are not the actual values

Experiments (MPI)

1. **Your code should work with different sizes of matrix**
   - M rows, N columns (you can assume that matrix is larger than some minimum)

2. **Try** (at least) **the following code parameters:**
   a. `mpirun ./a2 --m 2688 --n 4096 --epsilon 0.01 --max-iterations 1000`
   b. `mpirun ./a2 --m 2688 --n 4096 --epsilon 0.01 --max-iterations 2000`
   c. `mpirun ./a2 --m 1152 --n 1152 --epsilon 0.01 --max-iterations 1000`

3. **Run** (at least) **using the following cluster configurations:**
   a. 1 node using 2, 4, 8, 16 and 32 MPI processes (for comparison with OpenMP)
   b. 2 nodes, 32 processes
   c. 4 nodes, 64 processes
   d. 6 nodes, 96 processes

Achieve a speedup of ~12 using all available nodes on the Alma system (using 2.a. and 3.d) and observe the speedup when using other configurations

- Try to explain the differences (if any) in the performance as M, N, and max-iterations change
- If you want to test for M % nprocs != 0, then test with different value of M

REPORT AND DELIVERABLES

Write a report and include:

1. Speedup results (graphs + tables)
   - Measure performance and compare speedup

2. Discuss
   - Implementation, performance differences (between different versions and different inputs), problems, ...
   - And **all other points** in the report template

Submission (Moodle)

- Submit a zip-file with:
  - `a2-omp.cpp`
  - `a2-mpi.cpp`
  - `report.pdf`
- and other source code files, e.g., jobscripts for MPI
- Upload to Moodle before the deadline

Note: both source codes and the report are required, submissions without the report will not be graded positive!

COMPILATION, NAMING CONVENTION

Your starting point is the sequential version of the code (in Moodle):

- `a2.cpp` (also a template for your solution)
- `a2-helpers.hpp`

Sequential version:

- Compile: `g++ -O2 -lm -o a2 a2.cpp`
- Run: `./a2` <arguments>

MPI version:

- Compile: `mpic++ -O2 -lm -o a2-mpi a2-mpi.cpp`
- Run: **mpirun -np** <num_processes> **a2-mpi** <arguments> (at home)
- Use **Slurm** jobscript to run on Alma (see examples in the slides)

OpenMP version:

- Compile: **g++ -O2 -lm -o a2-omp a2-omp.cpp**
- Run: <ENV_VARS_...> **srun --nodes=1 ./a2-omp** <arguments>
- ENV_VARS example: OMP_NUM_THREADS=16, ...

JOB SCRIPT EXAMPLE

```
#!/bin/bash
#SBATCH -N 4
#SBATCH --ntasks 16
#SBATCH -t 200
mpirun ./a2 --m 2688 --n 4096 --epsilon 0.001 --max-iterations 1000
```

4 nodes

MPI tasks

16 processes over 4 nodes

Timeout (in the case of a deadlock)

Submit for execution

```
> sbatch jobscript.sh
Submitted batch job 6235

> squeue
JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
 6236       all jobscrip johndoe  R       0:11      4 alma[01-04]
```

Check if it is running

The result will be saved in slurm-6235.out when complete

MATRIX DATA STRUCTURE AND HELPER FUNCTIONS

Contains a custom data structure “Mat”

- Contiguous 2D data structure
- Note the ”(“ and “)” can also be used: `mat(i,j)`

```
Mat mat(height, width);
for (int i=0; i<mat.height; i++) {
   for (int j = 0; j < mat.width; j++) {
      mat[i][j] = f(x);
   }
}
// with MPI: MPI_Send(&mat[0][0], ... );
```

And number of member functions that you do not need to modify, but they may be helpful:

- `void save_to_disk(std::string filename);`
  Purpose: Save data to disk so it can be converted to an image
  Example: `mat.save_to_disk("heat2d.txt");`

- `void print();`
  Purpose: Printing data to standard output
  Example: `mat.print();`

- `bool compare(Mat& m, double eps=std::pow(10, -8));`
  Purpose: Compare data element by elements, used for the verification of the results
  Example: `mat.compare(mat2);`

OTHER HELPER FUNCTIONS, AND USAGE

Other functions:

-
  ```
  void process_input(int argc, char **argv, int& N, int& M, int& max_iterations,
                     double& epsilon, bool& verify=true, bool& print_config=false);
  ```
  Purpose: Processing of input arguments

  Notes: By default results are verified, and configuration is not printed. Both `verify` and `print_config` are optional.

-
  ```
  void heat2d_sequential(Mat& mat, int max_iterations,
                         double epsilon, int& iteration_count);
  ```
- Purpose: Compute the data sequentially and use it for verification

  Note1: Matrix `mat` and `iteration_count` are `inout` arguments, they return the resulting sequentially computed data and the number of iterations for verification purposes.

Usage

```
./a2 --m <int> --n <int> --max-iterations <int> --epsilon <double> [ --no-verify --print_config ]
```

`--m`	The number of rows (required)
`--n`	The number of columns (required)
`--max-iterations`	Integer value (default == 1000)
`--epsilon`	Double value (default == 1.0e-3)
`--no-verify`	Disable verification. Could be useful while measuring performance (default == false)
`--print_config`	Print the configuration in use (default == false)