Parallel Computing, 2024W

Assignment 3: Heat Equation (2D) with MPI

SOLVING HEAT EQUATION IN 2D

The heat equation is a Partial Differential Equation

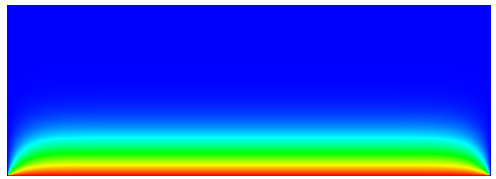
$$\frac{\partial u}{\partial t} = \alpha\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right)$$

Using Jacobi iterative method (simplified):

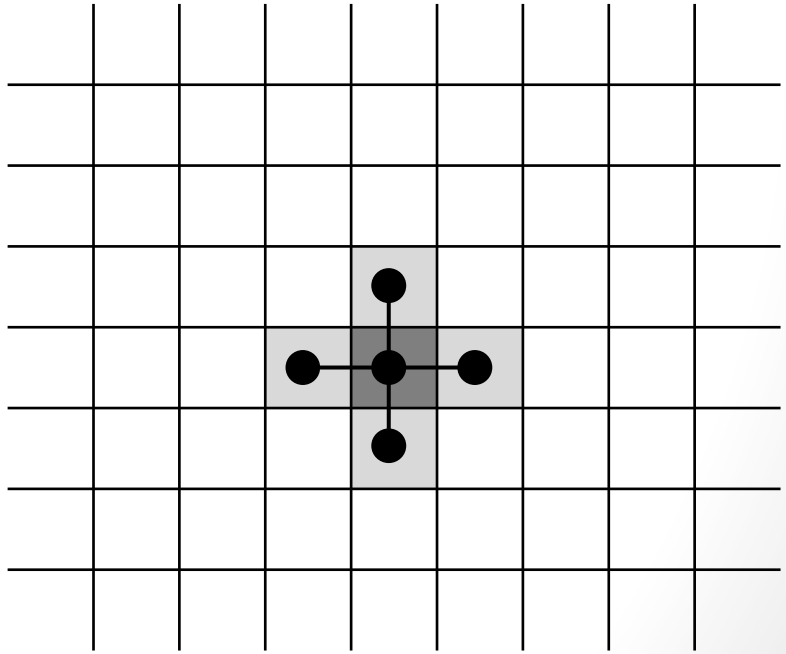
$$v_{m.l}^{n+1} = \frac{1}{4}\left(v_{m+1,l}^{n} + v_{m-1,l}^{n} + v_{m.l+1}^{n} + v_{m.l-1}^{n}\right) - \frac{h^2}{4} f_{ml}$$

Zero

- NxM Matrix (M rows and N columns)
- Basically, a five-point stencil
  - Calculation of averages



Example output



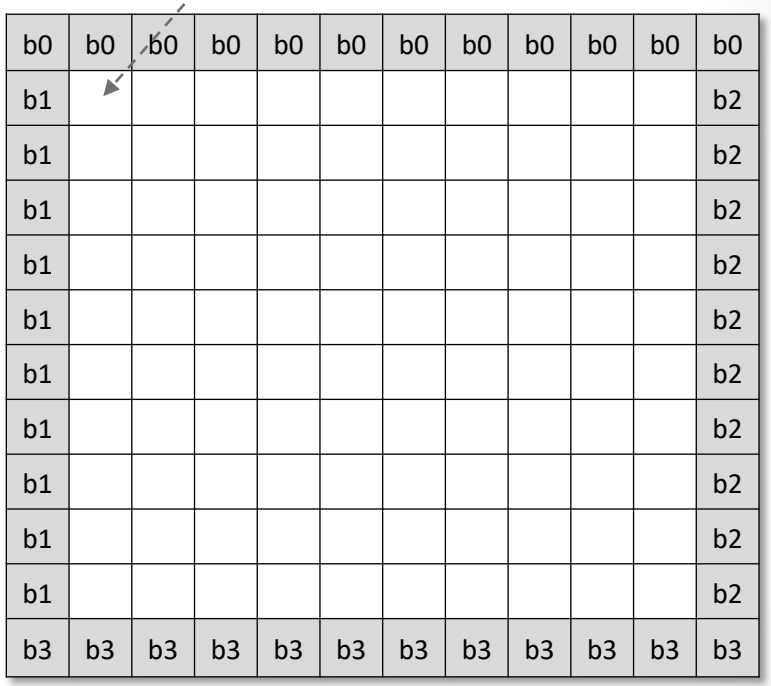
Five-point stencil

CORE OF THE SEQUENTIAL CODE

```
//...
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

We start from here



Boundary conditions

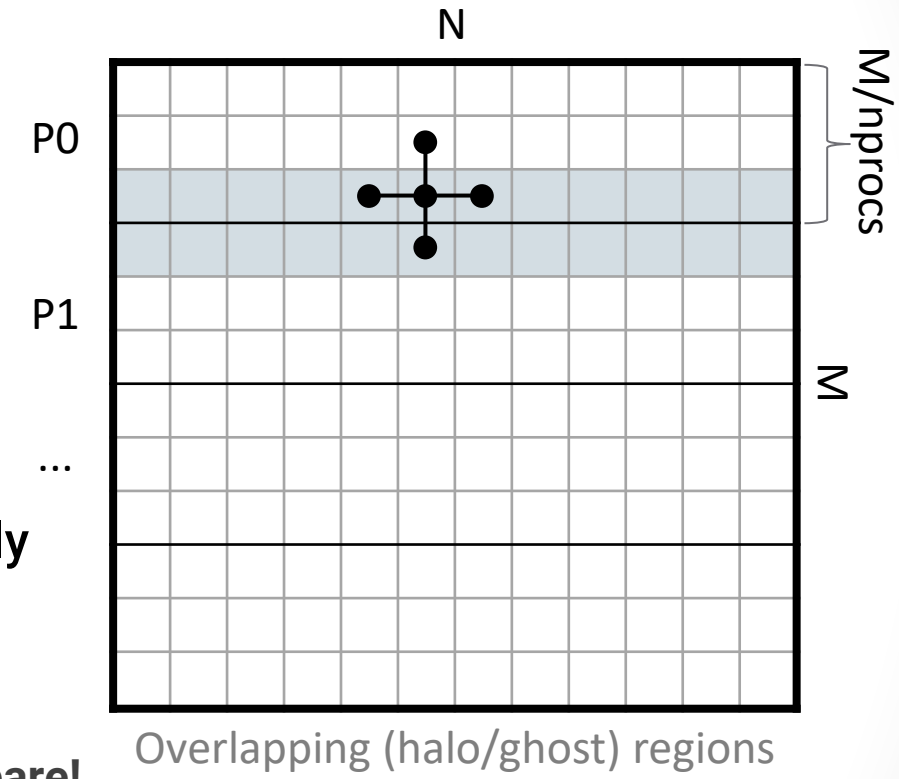
b0 = 0.02, b1 = 0.05, b2 = 0.1, b3 = 0.2

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

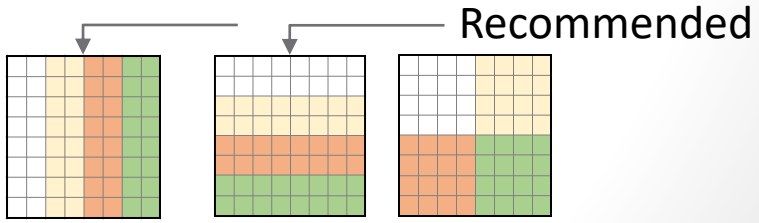
HEAT2D WITH MPI

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)

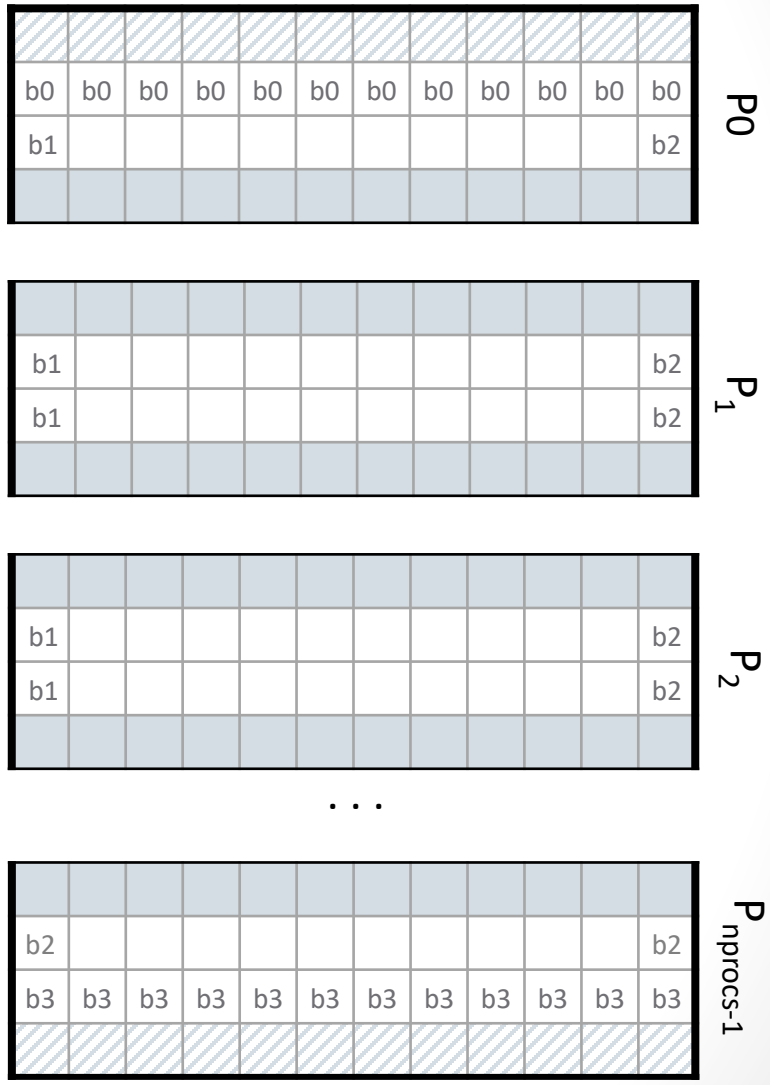
Each process gets a part of the MxN matrix

- 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
- Alternatively, you can have smaller local sizes for the top and bottom processes (tends to be more complex)



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, …)

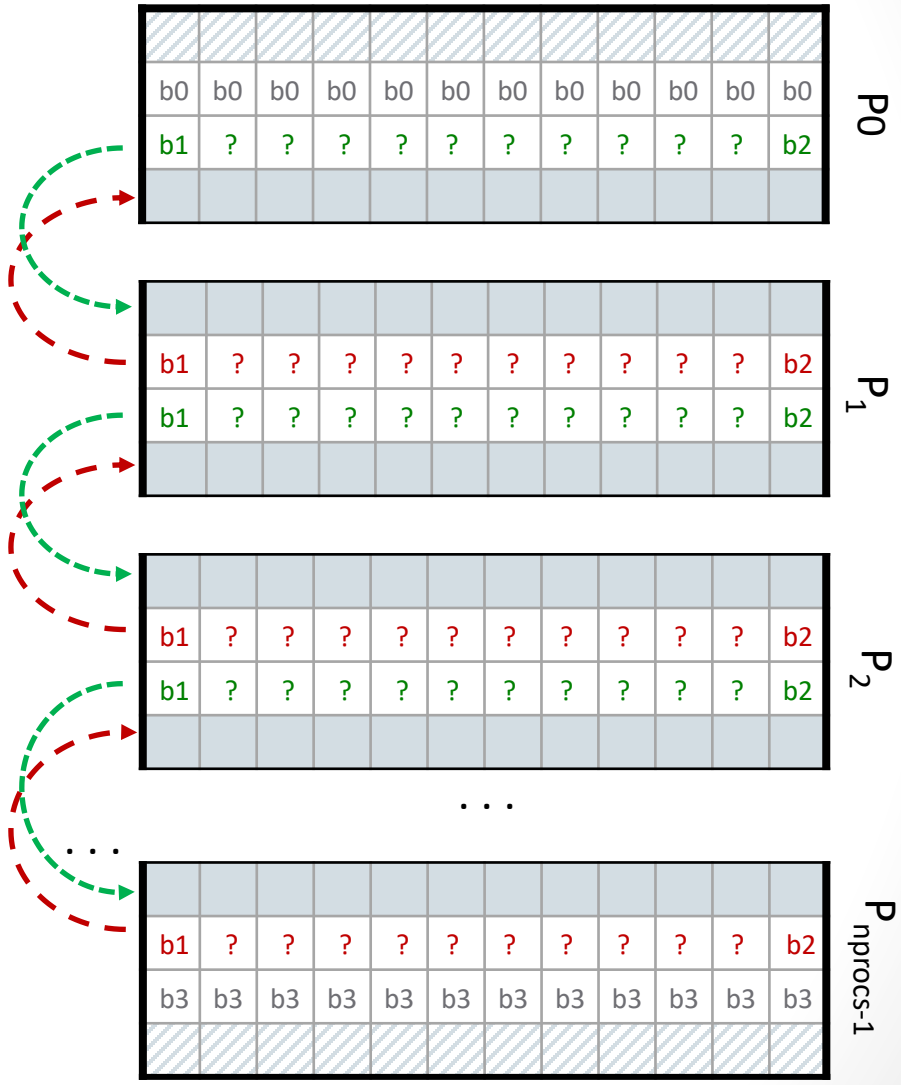
Green arrows

- 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

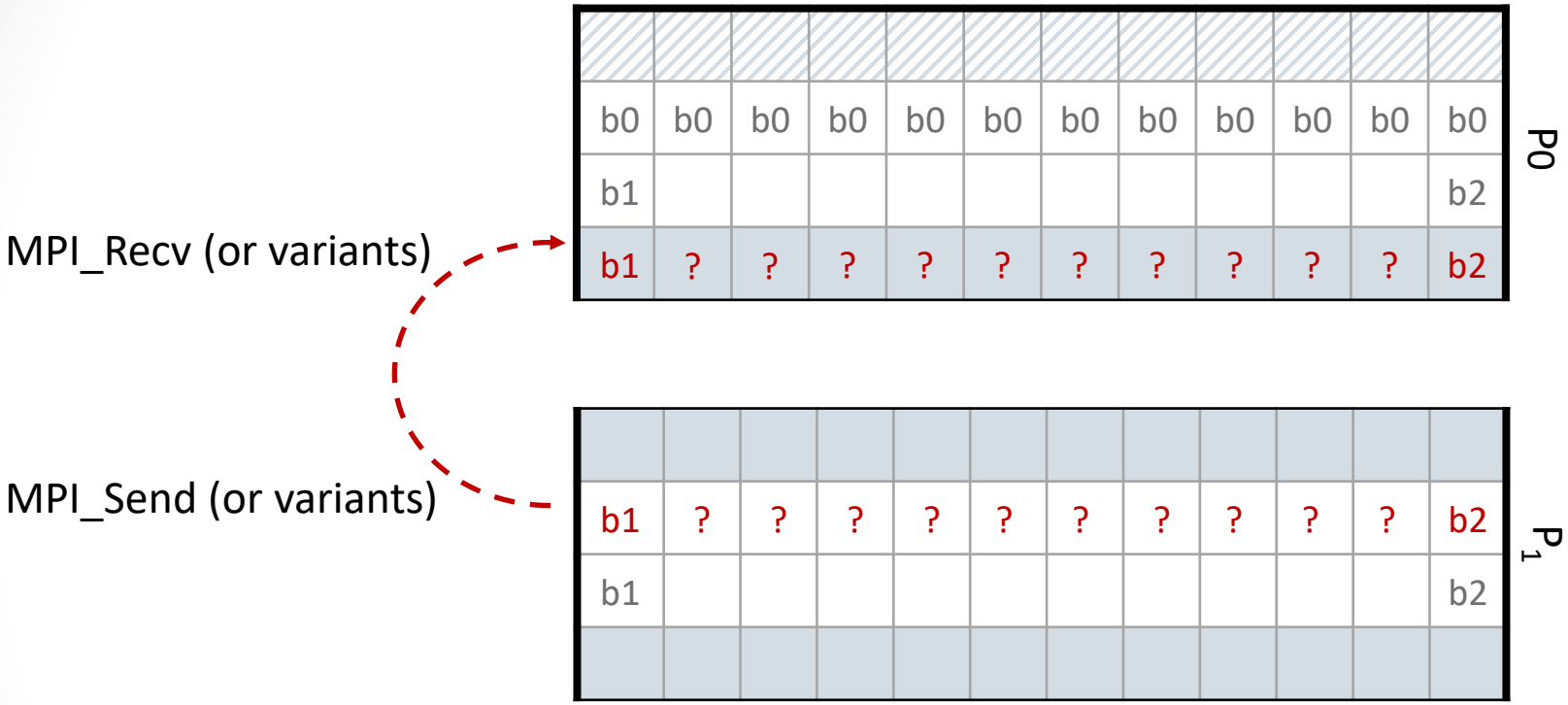
You need MPI_Send and MPI_Recv on each end

- 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, …



COMMUNICATION :: SINGLE MESSAGE



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. **On Alma, run the program with the following configuration using Slurm:**
   a. `mpirun ./heat2d --m 2688 --n 4096 --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
   b. 2 nodes, 32 processes
   c. 4 nodes, 64 processes

Achieve a speedup* of ~11 using 4 nodes on the Alma system (using 2.a. and 3.c) and measure the speedup* when using other configurations

- Note: to test for M % nprocs != 0, then test with different (non-default) value of M

*with respect to the sequential version that uses the same configuration

DELIVERABLES, REMARKS

Submit your source code to the online platform (moped.par.univie.ac.at)

- Only one source file: heat2d.cpp

Measure speedup on Alma, and enter speedup results on the online platform

- Create a job script for Slurm on Alma and use it to run your program for performance measurements

Remarks

- Remember not to measure on the frontend node
- Some errors may appear only after you test your program on multiple nodes
- To prepare for the test, try to explain the differences (if any) in the performance as M, N, and max-iterations change

Additional Notes

COMPILATION, NAMING CONVENTION

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

- **`heat2d.cpp`** (also a template for your solution with comments)
- **`helpers.hpp`**

Sequential version:

- Compile*: **`g++ -O2 -lm -o heat2d heat2d.cpp`**
- Run: **`./heat2d`** <arguments>
- *you may need to also provide -std=c++17 or -std=c++20 switch on your machine

MPI version:

- Compile: **`mpic++ -O2 -lm -o heat2d heat2d.cpp`**
- Run: **mpirun -np** <num_processes> **./heat2d** <arguments> (at home)
- Use **Slurm** jobscript to run on Alma (see examples in this slide set)

SLURM ON ALMA :: JOB SCRIPT EXAMPLE

```
#!/bin/bash
#SBATCH -N 4
#SBATCH --ntasks 16
#SBATCH -t 3
mpirun ./heat2d --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 4242

> squeue
JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
 4242       all jobscrip johndoe  R       0:11      4 alma[01-04]
```

Check if it is running

The result will be saved in slurm-4242.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

```
./heat2d --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)