In this project, we are going to write a locally synchronous parallel program to solve the problem of heat distribution over a two-dimensional square plate.

We assume the plate has a dimension of l by l. On the boundary of the plate, the temperature is fixed as follows: the left and the right sides have a constant fixed temperature of 0ºC; while the top and the bottom sides have a varying but fixed temperature described by the following equation:

$$u = 255 \sin \frac{\pi x}{l}$$

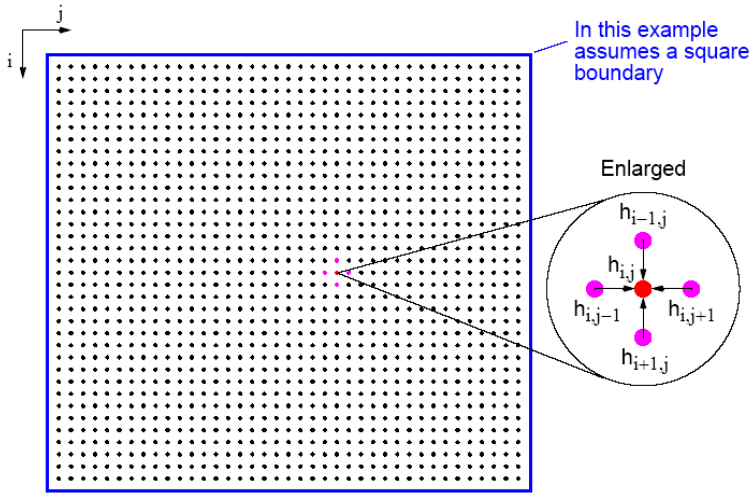
Here x is the distance measured from the left side of the plate.

The two-dimensional heat distribution problem can be described using the following partial differential equation:

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

where u is the equilibrium temperature and α is a constant reflecting the property of the plate. Like many partial differential equations, this equation has an analytic solution only for trivial boundary conditions. Most of the time, we can only depend on finding a numerical solution.

To find a numerical solution, we need to first divide the plate into a set of grid points. See Chapter 6 of Wilkinson and Allen for details.



If we divide the plate into a set of n by n small squares, then we will have $n + 1$ by $n + 1$ grid points for the entire plate. It turns out that the equations we use to update the temperatures are surprisingly simple:

$$u_{i,j,t+1} = \frac{1}{4}\left(u_{i-1,j,t} + u_{i+1,j,t} + u_{i,j-1,t} + u_{i,j+1,t}\right)$$

Your job is to write an MPI program to calculate the equilibrium temperatures of the plate.

A set of utility functions has been provided in the several files.

To make it easier to program, we use a numpy matrix to represent the grid. This allows us to take advantage of the extremely optimized matrix computation provided by numpy.

The files `utils.py` defines a set of functions that may be useful in implementing both the sequential and parallel solutions:

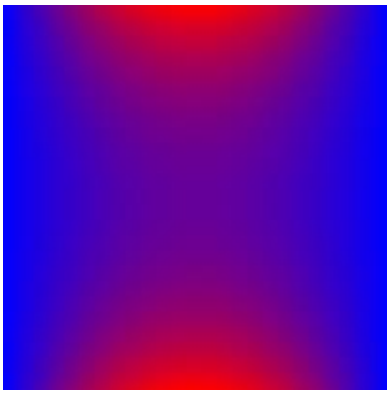
```
…
def initialize_matrix(data, process_id, number_of_processes):
    …

def do_single_round(data):
    …

def write_matrix_as_image(filename, matrix):
    …
```

The function `initialize_matrix` is used to initialize the boundary conditions for the matrix based on the equation given above. Note that the function can also be used (and is used) in the sequential version of the solution – if we set the values of `process_id` to 0 and `number_of_processes` to 1. Also note that if the boundary conditions change, this function also needs to be changed. The function `do_single_round` handles the update of the matrix for one single iteration.

The function `write_matrix_as_image` writes out the matrix as a PPM image for easy visualization of the results. It also writes out the raw data for reference. Without visualization, it is difficult to comprehend the large amount of output data. The figure below shows the equilibrium temperatures of the plate, where blue represents the cold region, while red represents the hot region:



You can use the `xview` program to see the image.

The files `mpiUtils.py` defines functions that may be useful in implementing the parallel solution:

```
…
def get_num_rows(matrix_dimension, process_id, number_of_processes):
    …

def gather_data(matrix_dimension, process_id, number_of_processes, local_matrix):
    …
```

The function `get_num_rows` computes the number of rows of grid points each process is responsible to update. The value returned does not include the two ghost rows at the top and the bottom of the region. It takes care the case that the number of rows is not evenly divisible by the number of processes (which is always a headache in parallel computation). The function `gather_data` is designed to free you from worrying about handling data of uneven sizes when moving data back to the root process.

Finally, the file `sequential_heat_distribution.py` provides a reference sequential implementation for solving the heat distribution problem. This program sets the terminating condition to be when the maximum absolute difference of all grid points during any iteration is less than 10^{-6}. That is: $\max_{0 \le i,j \le n} \left(\left| u_{i,j,t+1} - u_{i,j,t} \right| \right) \le 10^{-6}$.

Your parallel solution should use the same terminating condition and produce exactly the same result. Please note that this maximum absolute difference needs to be reduced to the root process and then broadcasted back to every process in order for all processes to stop at the same time.

A script named `testrun` is also provided for you to test your program with 1 to 16 processes and with varying data size from 128 to 1024. Please note that it may take days to execute the script depending on the speed of your cluster. Here is an example output:

```
Testing for N = 16 ...
Sequential ... (0.021) Done
P = 1 ... (0.019) Done.
P = 2 ... (0.018) Done.
P = 3 ... (0.015) Done.
P = 4 ... (0.039) Done.
P = 5 ... (0.060) Done.
P = 6 ... (0.129) Done.
P = 7 ... (0.074) Done.
P = 8 ... (0.218) Done.
…
```

The values in parentheses are the number of seconds needed to complete the computation (please note the time data depends on the speed of the cluster and/or the cluster configuration).

Reference: Wilkinson and Allen, “Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers,” 2nd edition, Pearson Prentice Hall, 2005.