CSIS7303 High Performance Computing

Assignment 1 – Parallel Solver for the Steady-State Heat Distribution Problem

Objective

The objective of this assignment is to give students hands-on experience in parallel program implementation in C with the MPI and OpenMP API.

Problem Description

The heat distribution problem is an important study in science/engineering and plays a key role in climate modeling and weather forecasting. The two-dimensional (2D) heat distribution problem is to find the steady-state temperature distribution within an area which has known fixed temperatures along each of its edges.

Mathematical Background

Distribution of temperature $u(x, y, t)$ over a unit area can be modeled by the *Poisson's equation* which is a 2nd order partial differential equation (PDE) arising in physics:

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \frac{\partial u}{\partial t}, \qquad 0 < x < 1, \quad 0 < y < 1, \quad t > 0 \qquad \ldots (1)$$

In the problem we are interested in, there are no internal heat sources, so $\partial u / \partial t = 0$ and it reduces to the *Laplace's equation*:

$$\nabla^2 u = 0 \qquad \ldots (2)$$

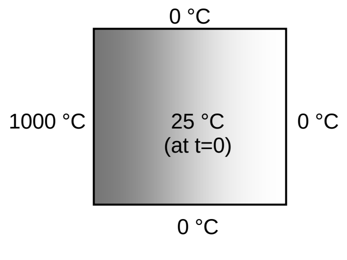
We would set the initial and boundary conditions for the problem as follows:

Initial condition:

$$u(x, y, 0) = 25, \qquad \text{for } 0 < x < 1, \quad 0 < y < 1$$

Boundary conditions:

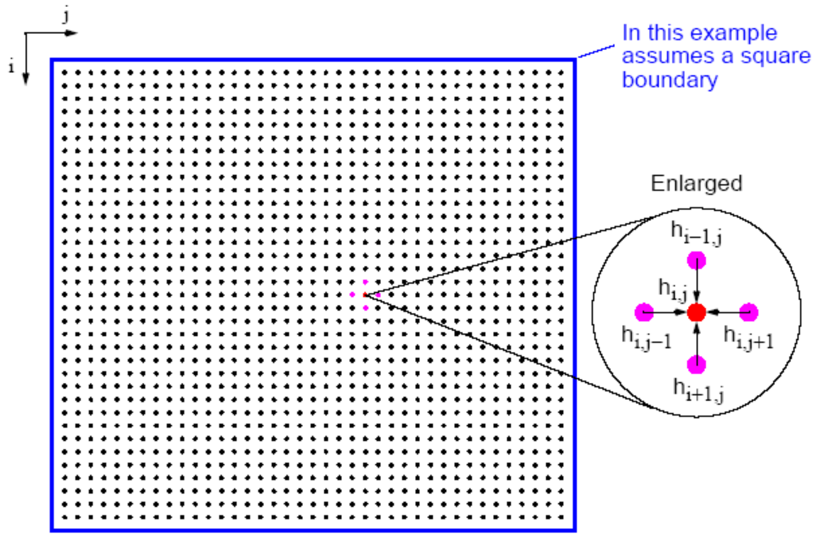
$$u(0, y, t) = 0 \qquad u(x, 0, t) = 1000$$
$$u(1, y, t) = 0 \qquad u(x, 1, t) = 0$$



You may imagine this is a top view of a square room which has a wall heater (at 1,000 °C) at one side and the other three sides are freezing glass windows (at 0 °C) as it is snowing outdoors. For convenience, the initial temperature over the entire area is set to room temperature 25 °C except the boundaries. You can expect the temperature at each point of the whole interior area will keep changing according to heat diffusion and finally stabilizes at some equilibrium since the boundary temperatures are fixed.

Finite Difference Methods

We are going to solve this problem numerically using the so-called *finite difference method*, by which we *discretize* the space and time such that there are an integer number of space points and time steps at which we can calculate the solution. In our case, this means the area is evenly decomposed into a fine mesh of points and the temperature $h_{i,j}$ at an interior point $[i, j]$ is taken to be average of temperatures of its four neighboring points, as illustrated in the diagram below:



For an $n \times n$ grid, there are $(n\text{-}2) \times (n\text{-}2)$ interior points and the temperature of each of them can be computed by iterating the equation:

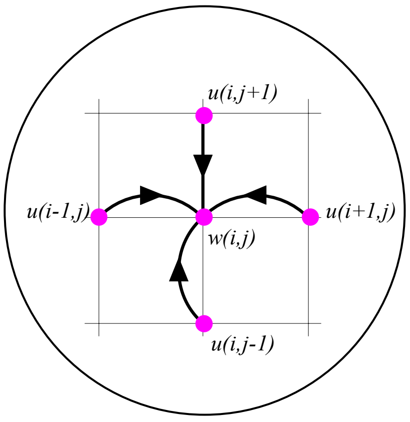
$$h_{i,j} = \frac{h_{i-1,j} + h_{i+1,j} + h_{i,j-1} + h_{i,j+1}}{4} \qquad \ldots (3)$$

for a fixed number of iterations or until the difference between iterations of a point is less than some predefined error tolerance ε (say 0.001). (For your interest, you may refer to the appendix for why four points are taken for the average and how this computation relates to the Laplace's equation.)

Jacobi Iterative Method

The equation (3) above is essentially the *Gauss-Seidel* iterative method which relies on a sequential order of computation and is thus more difficult to parallelize. In this assignment, we adopt the *Jacobi* method:

$$w[i, j] = \frac{u[i-1, j] + u[i+1, j] + u[i, j-1] + u[i, j+1]}{4} \quad \ldots (4)$$



where w stores the current iteration's solution computed from the four solutions carried by *u's* in the previous iteration.

We can observe that (4) converges slower than (3) and consumes more memory (two arrays w and u needed) but it leads to easier parallelization.

Programming Tasks

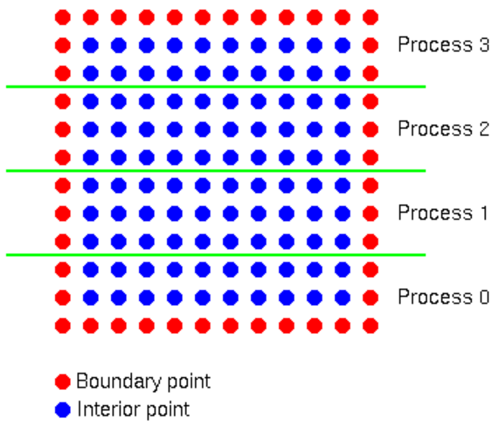
You are going to implement two C programs – one using MPI and one using OpenMP – based on the Jacobi iterative method to compute the heat distribution problem in parallel on multiple processors. We will provide the Jacobi sequential version for you as the baseline and your task is to modify it into MPI and OpenMP versions.

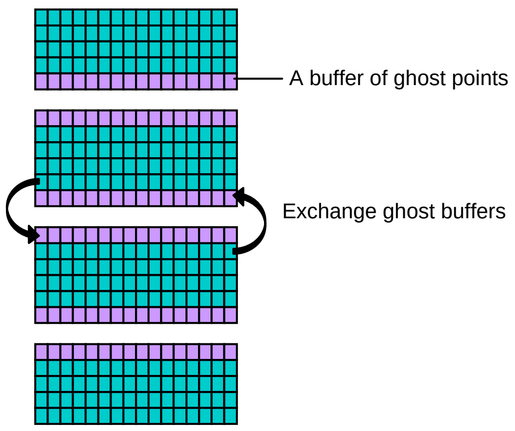
Part 1: OpenMP Programming (30%)

The Jacobi program uses some for-loops to compute temperatures. Add OpenMP primitives to apply loop parallelization to speed up the program. You may try different scheduling schemes (static, dynamic, guided), different chunk sizes as well as other OpenMP clauses to arrive at your best-performing program.

Part 2: MPI Programming (70%)

Develop your MPI code based on row-striped decomposition, depicted as follows. Each process is responsible for computing a set of rows of points. Computing points at the edges of stripes would require the data held by other processes. So you need to arrange the processes to exchange their respective wanted rows. For programming convenience, each process should have an additional row at each edge, called *ghost points*, to store the row received from another process so that the temperature computations can see a seamless array. At the end of execution, your program should output the steady-state temperature array; this needs another communication phase for collecting all processors' computed results.





Note:

In both programs, termination is determined by a convergence test on the maximum temperature difference between successive iterations. If the difference is smaller than the predefined tolerance $\varepsilon = 0.001$, then all threads or processes should stop further computing and report the result. (Still, you should retain the check of maximum iterations allowed as a safeguard to erroneous infinite loop as our provided sequential version did.) This termination condition should be kept in your parallel code. For any other implementation details, it is up to your own choice like whether synchronous or non-blocking communication is used.

Performance Benchmarking

After development and sufficient correctness verification, you are going to benchmark and possibly fine-tune your MPI program on the Gideon cluster and your OpenMP program on the Belief multi-core SMP server respectively. We will reserve dedicated environments for your benchmarking done in turns through some job queuing system. Details will be announced later.

Measure the execution times of the provided sequential program and your parallel programs (note: compiled with -O3) with the combinations below. Compute absolute speedups and efficiencies for each combination. Performance results should be plotted as charts for visualization. A sample spreadsheet will be provided for your reference.

• Problem size (*n*):	128, 256 and 384	
• Number of threads (*p*):	1, 2, 4, 8	(for OpenMP)
• Number of processes (*p*):	1, 2, 4, 8, 16	(for MPI)

Grading Criteria

Hint: Achieve (1) correct result at the (2) best achievable efficiency/scalability by (3) least required parallel coding effort. Relative importance: (1) > (2) > (3). To hit point (2), your program should exploit highest parallelism with least communications. We will apply the rough weighting below to each part of your work.

Correctness and Style (~ 70%)

- Both parallel programs must give the consistent output as the sequential code did. A fast but incorrect parallel program is meaningless and thus we emphasize the program's correctness. Failing to meet this requirement would result in about 60% score deduction. In particular, ensure no deadlock in all processor combinations and in the various problem sizes.
- You should avoid hard-coded programming style, e.g. skip communication if the iteration $\leq$ 30,000 which is hard-coded and makes sense only for some cases. But programming based on dynamics or heuristics is acceptable as long as they are reasonable, e.g. skip communication if maximum difference $\leq 10 \times \varepsilon$.

Performance (~ 30%)

- Achieving a fair speedup in most cases is already good enough in this assignment. However, higher score would be given to those achieved better overall efficiency obtained for all the specified processor combinations.

Submission (Deadline: March 14, 2008 at 12:00 noon)

Submit the following via MSc(CS) Intranet (https://msccs.cs.hku.hk):

- Source code of OpenMP C program (must be able to compile on Belief)
- Source code of MPI C program (must be able to compile on Gideon)
- A report of performance charts, either in PDF (preferred), Microsoft Word or Excel format like our sample. (You may also write briefly on the report to explain your charts but this is pretty optional.)

Appendix (For your interest only):

Relationship between PDEs and Finite Difference Methods

Recap the Laplace's equation:

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

Using the following formula for 2nd derivative,

$$f''(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}$$

we get

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{1}{\Delta^2}\left[u(x+\Delta, y) - 2u(x, y) + u(x-\Delta, y)\right]$$

$$\frac{\partial^2 u}{\partial y^2} \approx \frac{1}{\Delta^2}\left[u(x, y+\Delta) - 2u(x, y) + u(x, y-\Delta)\right]$$

Substituting into the Laplace's equation, we get

$$\frac{1}{\Delta^2}\left[u(x+\Delta, y) + u(x-\Delta, y) + u(x, y+\Delta) + u(x, y-\Delta) - 4u(x, y)\right] = 0$$

Rearranging terms gives

$$u(x, y) = \frac{u(x-\Delta, y) + u(x, y-\Delta) + u(x+\Delta, y) + u(x, y+\Delta)}{4}$$

Rewritten as an iterative formula:

$$u^{(k)}(x, y) = \frac{u^{(k-1)}(x-\Delta, y) + u^{(k-1)}(x, y-\Delta) + u^{(k-1)}(x+\Delta, y) + u^{(k-1)}(x, y+\Delta)}{4}$$

where $u^{(k)}(x, y)$ denotes the solution in the *k*-th iteration. The equation means the solution in the *k*-th iteration is computed from four solutions in the (*k*-1)-th iteration.