COMP3230A Principles of Operating Systems

Programming Assignment #2

Due date: Nov. 23, 2021 at 23:59 pm

Total 13 points

(version: 1.0)

Programming Exercise – multithreading and synchronization

Objectives

1. An assessment task related to ILO 4 [Practicability] – "demonstrate knowledge in applying system software and tools available in the modern operating system for software development".
2. A learning activity related to ILO 2.
3. The goals of this programming exercise are:
   - to have hands-on practice in designing and developing multithreading programs;
   - to learn how to use POSIX pthreads and semaphore libraries to create, manage, and coordinate multiple threads in a shared memory environment;
   - to design and implement synchronization schemes for multithreaded processes using semaphores, mutex locks and condition variables.

Task

Write a multi-threaded program that computes the heat distribution problem based on the Jacobi iterative method. We will provide the jacobi_seq.c file, which is the sequential version as a baseline for your task. You will use either the semaphores or (mutex locks & condition variables) to implement a multi-threading solution to the problem.

Heat Distribution Problem

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.

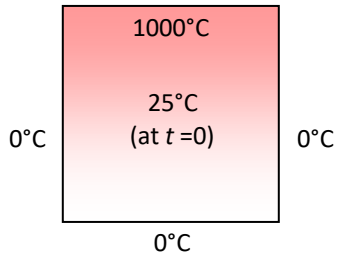
Distribution of temperature u(x, y, t) over a unit area can be modeled by the *Poisson's equation* which is a 2nd order partial differential equation (PDE) arising in physics:

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

In the problem we are modeling, the initial and boundary conditions are as follows:

Initial condition:

$u(x, y, 0) = 25,$ for $0 < x < 1, 0 < y < 1$



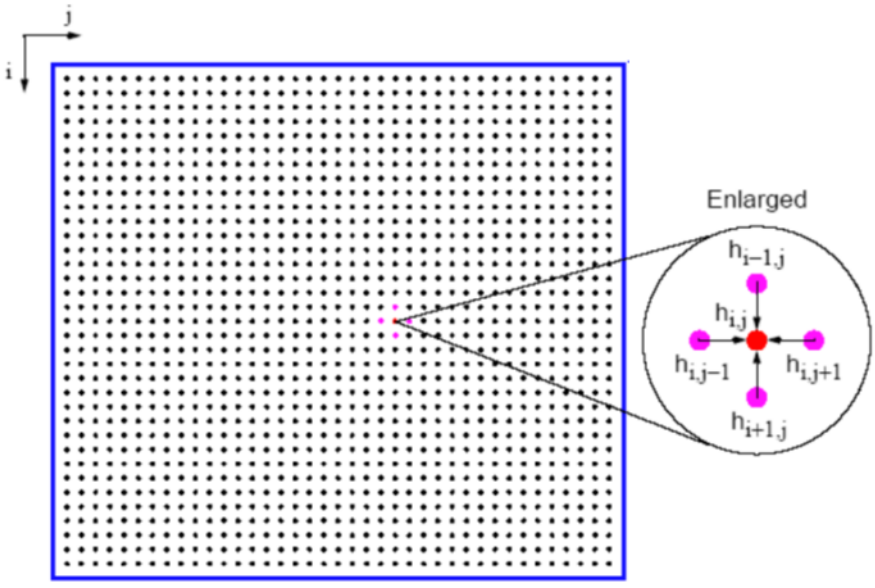
Boundary conditions:

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

There are no internal heat sources, so $\nabla^2 u = 0$

For this problem, 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) for it is snowing outdoors. 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.

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 the average temperatures of its four neighboring points, as illustrated in the diagram below.



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

$$h_{i,j} = \frac{h_{i-1,j} + h_{i+1,j} + h_{i,j-1} + h_{i,j+1}}{4}$$

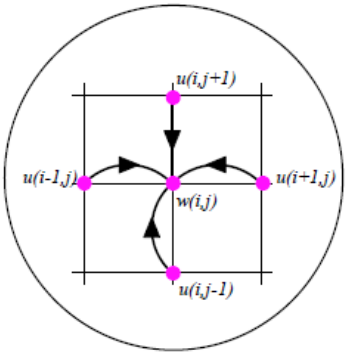
for a fixed number of iterations or until the difference between iterations of a point is less than some predefined error tolerance ε (say **0.001**).

Jacobi Iterative Method

The above equation 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 iterative* method:

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

In the Jacobi method, we use **two** 2D arrays to store the temperature distribution. The array *w* stores the current iteration's solution while the array *u* stores the previous iteration's solution. For each iteration, we calculate the temperature value of a point by computing the average temperature value of its four neighbors obtained from the previous iteration.



We can observe that Jacobi converges slower than Gauss-Seidel and consumes more memory (two arrays are needed) but it leads to easier parallelization using multiple threads.

Programming Tasks

1. Download the sequential program – jacobi_seq.c from the Course's website at moodle.hku.hk. The program optionally accepts two input arguments:

   ./jacobi_seq [rows columns]

   with *rows* and *columns* define the number of rows and columns of space points in the 2D area. Without providing any arguments, the program uses the default setting: rows = 200, columns = 200.

   To compile the program, use ***gcc*** and link to the Math library:

   gcc jacobi_seq.c -lm -o jacobi_seq

   You can find the implementation of the Jacobi method in the **find_steady_state()** function.

   Upon termination, the program prints out the following information:

   ```
   Converged after 30628 iterations with error: 0.001000.
   Elapsed time = 2.0597 sec.
   Program completed - user: 2.0580 s, system: 0.0040 s
   no. of context switches: vol 3, invol 4
   ```

   The first output line tells us which termination condition has reached. The second line tells us the total execution time of the find_steady_state() function. The third line reports the user time and system time utilized by the process. The last line reports the number of voluntary and involuntary context switches experienced by the process.

   Examine the sequential program and understand its execution logic before moving to the pthread implementation.

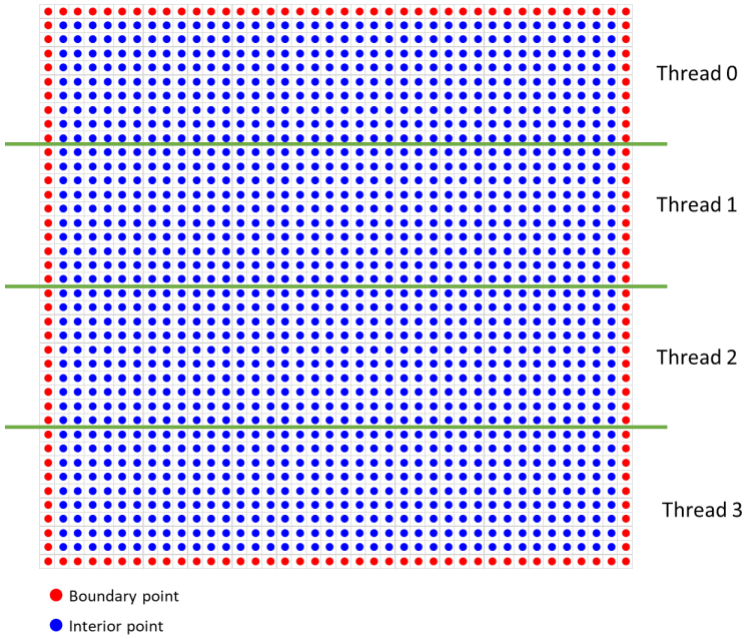
2. Convert the sequential program to accept one more optional input argument, which gives the number of worker threads to be created. Without providing any arguments, the program uses the default setting: rows = 200, columns = 200, threads = 2

3. **Parallelize the find_steady_state()** function by adding appropriate pthread functions and **use either semaphores or (mutex locks and condition variables)** as the synchronization tool.

   A template file with the filename - **jacobi_template.c** is provided to you. This file gives hints where you should place the code for this parallel implementation.

   For each iteration, the Jacobi method uses some for-loops to compute the temperature values. We can parallelize the loops by dividing the 2D matrix into sets of rows of points as depicted in the below diagram. Each worker thread is responsible for computing a set of rows of points.

```
for (i = 1; i < M-1; i++) {
   for (j = 1; j < N-1; j++) {
      w[i][j] = 0.25 * (u[i-1][j] + u[i+1][j] + u[i][j-1] + u[i][j+1]);
      if (fabs(w[i][j] - u[i][j]) > diff)
         diff = fabs(w[i][j] - u[i][j]);
   }
}
```



   After each worker thread completes its set of computations and updates w, they have to wait for the instruction from the master thread before moving to the next iteration or termination.

   After detecting all worker threads have completed their computation for an iteration, the master thread swaps the two matrixes and determines whether the termination condition has reached (i.e., the change of temperature at any point is less than the threshold or the maximum number of iterations has reached).

If the termination condition has reached, the master thread informs all worker threads to terminate; otherwise, the master thread informs all worker threads to continue to the next iteration.

When a worker thread notices that it is going to terminate, it uses the getrusage() function to retrieve its execution statistics and then returns/passes the data to the master thread.

For the master thread, after notifying all worker threads to terminate, it uses the pthread_join() function to wait for each thread to terminate and printout their execution statistics, e.g.,

Thread 0 has completed - user: 1.2722 s, system: 0.0905 s
Thread 1 has completed - user: 1.3669 s, system: 0.0900 s

The master thread also uses the getrusage() function to retrieve its execution statistics and print out the following line before returning from the find_steady_state() function.

find_steady_state - user: 0.1584 s, system: 0.2622 s

In the end, the main() function still reports the same output as the sequential program.

Converged after 30628 iterations with error: 0.001000.
Elapsed time = 1.6189 sec.
Program completed - user: 2.6482 s, system: 0.3568 s
no. of context switches: vol 94535, invol 8

Please name this version of your pthread program with the name – jacobi_cond.c if you are using mutex lock and condition variables for synchronization or jacobi_sema.c if you are using semaphores for synchronization.

To check whether your pthread program correctly implements the Jacobi method, we can compare the .dat file of the sequential version (jacobi_seq.dat) with the parallel version (jacobi_cond.dat or jacobi_sema.dat) using the *diff* utility command. In principle, they should be the same.

5. Benchmark the sequential program and your parallel program with different settings on the workbench2 server:

	No. of worker threads	Data size
jacobi_seq	NA	200x200, 400x400, 600x600
jacobi_cond or jacobi_sema	2 to 16	200x200, 400x400, 600x600, 800x800

Examine the benchmark results and write a short paragraph (~500 words) to explain the relationship between the number of worker threads, data size, synchronization time, and elapsed (execution) time.

Submission

Submit your program to the Programming # Two submission page at the course's moodle website. Name the program to jacobi_cond_StudentNumber.c or jacobi_sema_StudentNumber.c (replace StudentNumber with your HKU student number) and name the report to report_StudentNumber.txt (or docx or pdf). As the Moodle site may not accept source code submission, you can compress the three files to the zip or tgz format before uploading.

Documentation

1. At the head of the submitted source code, state the
   - File name
   - Student's name
   - Student Number
   - Development platform
   - Remark – describe how much you have completed
2. Inline comments (try to be detailed so that your code could be understood by others easily)

Computer platform to use

For this assignment, you can develop and test your program on any Linux platform, but you must benchmark the program on the workbench2 Linux server (which has enough CPU and memory resources). Your program must be written in C and successfully compiled with gcc.

Grading Criteria

1. Your submission will be primarily tested on the workbench2 server. Make sure that your program can be compiled *without any error*. Otherwise, we have no way to test your submission and you will get a zero mark.
2. As the tutor will check your source code, please write your program with good readability (i.e., with good code convention and sufficient comments) so that you will not lose marks due to possible confusion.

Documentation (1 point)	• Include necessary documentation to clearly indicate the logic of the program • Include required student's info at the beginning of the program
Report (1 point)	• Clearly indicate how the performance is affected by the number of threads and data size
jacobi_cond.c or jacobi_sema.c (11 points)	• The program should be compiled and executed successfully, and the total no. of worker threads created in this program should be equaled to the input parameter.
	• Worker threads must be executed in parallel.
	• Obtain the "correct" results as compared to the sequential program.
	• Can work with different numbers of worker threads and data sizes.
	• Worker threads can detect the termination condition and terminate successfully.
	• Worker threads must pass the execution statistics to the master thread.
	• The master thread should wait for all worker threads to terminate before displaying the results

Plagiarism

Plagiarism is a very serious offence. Students should understand what constitutes plagiarism, the consequences of committing an offence of plagiarism, and how to avoid it. **Please note that we may request you to explain to us how your program is functioning as well as we may also make use of software tools to detect software plagiarism.**