Advanced Computer Architecture
Lab 3 — Scalability of the Gauss-Seidel Algorithm

1 Introduction

The purpose of this lab is to:

- apply what you have learned so far in the course to a real world math kernel
- get some experience in using the POSIX threads API
- demonstrate some of the issues related to scaling of numerical algorithms

In this lab assignment we will make extensive use of the *POSIX threads* (*pthreads*) API, which is the standard threading API on Unix systems. We suggest that you check out the tutorial at Lawrence Livermore National Laboratory[1] if you have no prior experience with pthreads programming. Another excellent resource is the Single Unix Specification[2] which contains the documentation for all standard Unix APIs.

You are highly encouraged to solve this assignment in groups of two students. Talk to the teaching assistant if you, for some reason, want to work in some other configuration. This lab assignment is examined in the computer lab. During the examination, you will be asked to demonstrate and explain your solutions.

2 Improving the performance of the Gauss-Seidel algorithm

2.1 Introduction

The *Gauss-Seidel algorithm* is an iterative equation solver that is used to solve linear equation systems. We'll be solving the Laplace equation:

$$\Delta u = 0 \quad \text{in } \Omega \tag{1}$$

$$u = 0 \quad \text{on } \delta\Omega \tag{2}$$

We will only solve the equation in two dimension. We get the following equations by expanding the Laplacian (Δ) and including the parameters in the boundary condition:

$$\frac{\delta^2 u}{\delta x^2} + \frac{\delta^2 u}{\delta y^2} = 0 \quad \text{in } \Omega \tag{3}$$

[1] `https://computing.llnl.gov/tutorials/pthreads/`
[2] `http://www.unix.org/single_unix_specification/`

$$u(x, y) \quad = 0 \quad \text{on } \delta\Omega \tag{4}$$

You may have noticed that Equation 1 is really a partial differential equation, it is possible to discretize such an equation and solve it as a linear equation system. See subsubsection 2.1.1 if you are interested, otherwise, skip to subsubsection 2.1.2.

2.1.1 Mathematical background

⚠ **Warning:** The following section may contain intimidating math. Those who are faint of heart might want to jump straight to subsubsection 2.1.2.

We discretizing the problem using a homogeneous grid with the spacing h. Using central differences we can approximate the Δu as:

$$\Delta u_{i,j} \approx \frac{u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i,j}}{h^2} \tag{5}$$

The discretized problem is thus:

$$u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i,j} = 0 \tag{6}$$

It is possible (consult your linear algebra textbook) to write the above equation on the form:

$$\mathbf{Ax} = \mathbf{b} \tag{7}$$

The system can then be solved as a linear equation system using an iterative method, such as Gauss-Seidel. Let x_i^k be the value of element i in the vector $\mathbf{x}$ after iteration k. A general description of a sweep in a Gauss-Seidel solver would look as follows:

$$x_i^{k+1} = \frac{b_i - \sum_{j<i} a_{ij} x_j^{k+1} - \sum_{j>i} a_{ij} x_j^k}{a_{ii}} \tag{8}$$

In our case with the discretized Laplace equation (Equation 6), we get:

$$u_{i,j}^{k+1} = \frac{u_{i-1,j}^{k+1} + u_{i,j-1}^{k+1} + u_{i+1,j}^{k} + u_{i,j+1}^{k}}{4} \tag{9}$$

We continue to iterate Equation 9 until the solution has *converged*, i.e. the difference between the approximate answer and the real answer is small. We will use the following condition, where t is the tolerance, to test for convergence:

$$\sum_i \sum_j |u_{i,j}^k - u_{i,j}^{k+1}| \leq t \tag{10}$$

What Equation 10 really means is that the algorithm has converged when the difference in the results from two consecutive iterations is small.

2.1.2 Implementation

The neat thing about Gauss-Seidel is that it allows us to update the matrix representing the solution *in-place*, unlike some other methods where the old version of the solution must be kept in temporary storage. Algorithm 1 is a pseudo code implementation of the sweep in Equation 9.

Algorithm 1 Gauss-Seidel solver for the Laplace equation on an $n \times m$ matrix, with the tolerance t.

Require: $n, m \geq 2$
  repeat
    $e \Leftarrow 0$
    for $i = 1$ to $n - 1$ **do**
      for $j = 1$ to $m - 1$ **do**
        $v \Leftarrow \frac{u_{i-1,j} + u_{i,j-1} + u_{i+1,j} + u_{i,j+1}}{4}$
        $e \Leftarrow e + |u_{i,j} - v|$
        $u_{i,j} \Leftarrow v$
      end for
    end for
  until $e \leq t$

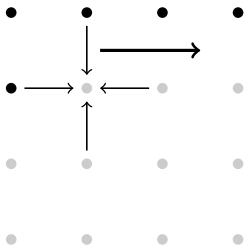

Figure 1: Access pattern for the sequential version of the Gauss-Seidel algorithm. The dark dots represent matrix elements that have been updated during the current iteration and the bright dots represent "old" values.

The sequential sweep of Algorithm 1 starts in the top left corner of the matrix, and iterate over each row, one element at a time, one row at a time, see Figure 1. We choose this order to improve spatial locality since C stores matrices in *row-major* order.

To improve performance, we can set up several threads working in parallel on different (vertical) chunks of the matrix. When a thread arrives at the right end (assuming that we sweep from left to right) of its chunk, it moves to the first element on the next line and waits until the thread to the left has computed its last value for that row before it continues, see Figure 2. In order to achieve this we have to include some kind of synchronization between the threads. There are a couple of different strategies to solve this, either you use a flag array with one flag per row and thread, or you use a progress counter for each thread. To simplify things, you may (should) have a barrier at the end of each iteration.

2.2 What is provided?

All the files related to the assignment can be downloaded from the course homepage. The source code package contains the complete source code for the sequential version of the algorithm, but only a skeleton for the parallel version. The source code for the parallel version contains comments (pay particular interest to the *TASK:* comments) to guide you towards what functionality should be implemented. Note that the comments really only applies to one particular way of solving the problem, you may of course solve the parallelization in a different way.

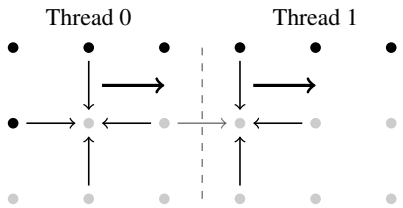

Figure 2: Access pattern for the multi-threaded Gauss-Seidel implementation. The dashed line represent the division between two threads. In this example thread 1 is waiting for thread 0 to update the last matrix element in its chunk on row 2, once that element has been updated thread 1 can start working on the row.

We have split the project into several source files to make the project structure cleaner and prepared a `Makefile`. In the source directory, you will find the following files:

Makefile Controls the compilation using the *make* tool. You can simply type **`make gs_pth`** to compile the pthreads version, or **`make gs_seq`** for the sequential version. There is also a *test* target that you should use to verify your solution, you may run it with **`make test`**.

gs_common.c Contains the common functions, like command line argument handling, initialization etc. Mostly boring stuff you don't need to bother yourselves about, most of the interesting stuff resides in separate implementation files.

gs_interface.h Contains declarations and documentation for the interface between `gs_common.c` and the GS implementations.

gsi_seq.c Contains the implementation of the `gsi_calculate` function for the *sequential* GS sweeps.

gsi_pth.c Will contain *your* version of the *parallel* `gsi_calculate` function.

solution.c Don't peek![3]

The code compiles as-is, but the parallel version doesn't do any computations nor does it contain any synchronization. You can set the debug mode by defining the macro `DEBUG` to 1 at the top of the file.

The default matrix size is 2048x2048, so that the matrix (filled with double elements, i.e. 32MB of data) doesn't fit in the cache. We start 4 threads by default. Inputs must be a power of 2.

2.3 Tasks

Edit the `gsi_pth.c` file and implement the `gsi_calculate` function using the pthreads library. The comments should give you some hints. Implement the synchronization using a progress counter (or flags) and then the iteration barrier.

Check that your results are correct by using the **`make test`** command. Running the *test* target of the `Makefile` will execute both the sequential and the parallel version of the program and compare the output.

[3]Not built by the `Makefile`, but can be built with **`gcc -o solution ./solution.c`**.

1. Implement the synchronization between threads working on the same *row* in the matrix.

2. Implement the barrier[4] at the end of the iteration.

   (a) Extra: Think of a solution without the barrier (a little more efficient), and propose it to me (you don't need to implement it).

3. Demonstrate your working solution implementation of the parallel Gauss-Seidel algorithm (i.e. same outputs for **`gs_seq`** and **`gs_pth`**, but faster!).

4. The current parallel implementation is really slow, this is due to how the local reduction variable for the error is stored. There is a simple thing that you can do to improve this, you should have heard about this in the lectures. What kind of miss is involved? Modify the `thread_info_t` data structure to improve the performance.

2.3.1 Bonus

1. Is the performance gain linear with the number of processors/threads? Why/Why not? (Elaborate your answer)

2. What kind of memory model do we rely on, here? (That is, what is the memory model of the machine in the lab and what does it guarantees does it provide?)

3. Suppose we decide to implement the solution differently. We want to use a thread pool, where each thread sits waiting for a task to be assigned. We implement the solution as follows: As soon as a thread finishes with its current task (i.e. computations for a row), it is assigned another line (maybe from another chunk). How would this affect performance? What is the name of the miss that is introduced?

[4] You may use a pthread barrier, see the documentation for `pthread_barrier_init`.