Lab 3: Scalability of the Gauss-Seidel Algorithm

Complete

Due: Fri Apr 16, 2021 12:00pm

Details

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 (https://computing.llnl.gov/tutorials/pthreads/)** if you have no prior experience with pthreads programming.

You are highly encouraged to solve this assignment in groups of two students (using the same groups as for previous Labs). Contact the teaching assistants if you have any problems with your current group. During the examination, you will be asked to demonstrate and explain your solutions.

There are two lab slots, Lab 3A and Lab 3B. You are only required to attend one of them. Please sign up for which slot you want to attend under **People -> Groups -> Lab 3 (https://uppsala.instructure.com/courses/23770/groups#tab-5828)**. Remember that it is mandatory to pass the lab in order to pass the course, with the exception of the bonus questions, which give you one bonus point.

Zoom link: **https://uu-se.zoom.us/j/69544346205 (https://uu-se.zoom.us/j/69544346205)**

Lab queue (available during the lab): **https://forms.gle/1cFCgfvNLgQwXMi69 (https://forms.gle/1cFCgfvNLgQwXMi69)**

TA email: **it-avdark-ta@lists.uu.se (mailto:it-avdark-ta@lists.uu.se)**

Improving the performance of the Gauss-Seidel algorithm

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

$$\Delta u = 0 \text{ in } \Omega \quad (1)$$
$$u = 0 \text{ on } \delta\Omega \quad (2)$$

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:

$$\frac{\delta^2 u}{\delta x^2} + \frac{\delta^2 u}{\delta y^2} = 0 \text{ in } \Omega \quad (3)$$
$$u(x, y) = 0 \text{ on } \delta\Omega \quad (4)$$

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 the *Mathematical background* if you are interested, otherwise, skip to *Implementation*.

Mathematical background

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

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

The discretized problem is thus:

$$u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i,j} = 0 \quad (6)$$

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

$$\mathbf{Ax} = \mathbf{b} \quad (7)$$

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:

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

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

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

We continue to iterate Equation 9 until the solution has *converged*, that is, 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:

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

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

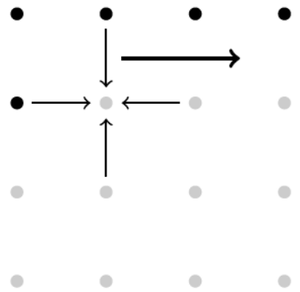
Implementation

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. The algorithm below is a pseudo code implementation of
the sweep in Equation 9.

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

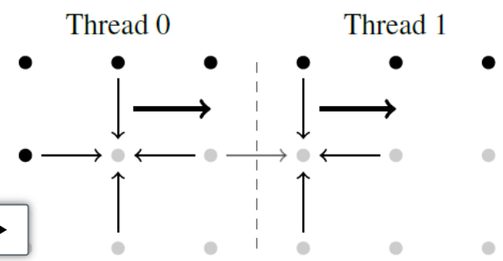
```
do {
    e := 0
    for (i := 1; i < n-1; ++i) {
        for (j := 1; j < m-1; ++j) {
            v := ( u[i-1][j] + u[i][j-1] + u[i+1][j] + u[i][j+1] ) / 4
            e := e + abs( u[i][j] - v )
            u[i][j] := v
        }
    }
} while (e > t);
```

The sequential sweep of the algorithm starts in the top left corner of the matrix, and iterates over each row, one element at a time, one row at a time, as in the figure below. We choose this order to improve spatial locality since C stores matrices in *row-major* order.



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.

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 the figure below. 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.



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.

C11 Memory Model and Atomic Operations

Modern C and C++ define a release consistency (RC) memory model where the application is guaranteed to execute as if under sequential consistency (SC) as long as no data races are present (SC-for-DRF). Note that this is different than the hardware memory model, which for our case is usually TSO. Data races are not allowed under the RC model and they are considered "undefined behavior" according to the C and C++ standards. In practice, this means that the compiler is allowed to assume that no data races exist in the code when making optimization choices. This can lead to code that behaves weird or is completely broken.

There are two main methods for eliminating data races in modern C: Locks and atomics, both of which should be provided by your C standard library. If they are not, you need to use a different/newer compiler. Both gcc and clang have full support nowadays (the Ubuntu servers listed here should have a new enough version of gcc: **https://www.it.uu.se/datordrift/maskinpark/linux (https://www.it.uu.se/datordrift/maskinpark/linux)**). We have seen how locks can be used to provide mutual exclusion in the first part of the previous lab, with `enter_critical` and `exit_critical`. The equivalent for C are the `mtx_lock` and `mtx_unlock` functions respectively (see **https://en.cppreference.com/w/c/thread (https://en.cppreference.com/w/c/thread)**). However, in this lab we will not be using locks at all and we will only focus on the atomics instead.

Similar to the atomics we saw in Lab 2, modern C also provides a specific atomic type qualifier, `_Atomic`, to be used with atomic variables. This means that the variable itself needs to be marked as atomic, and all operations on that variable have to be atomic. For convenience, your compiler might be able to automatically replace common C operations (e.g., `var += 1`) with the atomic equivalent (`atomic_fetch_and_add(&var, 1)`). You can read more about atomics and the functions available here: **https://en.cppreference.com/w/c/atomic (https://en.cppreference.com/w/c/atomic)** .

Atomics and Memory Ordering

All atomic operations in modern C come in two variants, the implicit and the explicit version. The difference between these two is that in the explicit version it is possible to specify the memory ordering that the atomic enforces.

By default, all atomic operations should be sequentially consistent. If we oversimplify, this means that all operations before the atomic have to be ordered (completed, made visible in the global memory order) before the atomic, and all operations after the atomic have to wait for the atomic. Essentially, it's the equivalent of having memory fences both before and after the atomic. In the explicit version, it is possible to specify other memory orders. A list, with attached explanations of each, can be found here: **https://en.cppreference.com/w/c/atomic/memory_order (https://en.cppreference.com/w/c/atomic/memory_order)** . The default one, used in the implicit versions, is `memory_order_seq_cst`, regardless if the operation is a load, a store, or a RMW. You can think of this memory order as if a fence is added both before and after the atomic operation (again, oversimplified). Other useful orders to consider are:

- `memory_order_relaxed` This memory order is essentially the equivalent of no fences and enforces no memory order between atomics. Atomic operations with this memory order can be freely reordered in the memory order.
- `memory_order_release` This memory order is for store (or RMW) operations only. You can think of it as if a memory fence is added **after** the store.
- `memory_order_acquire` This memory order is for load (or RMW) operations only. You can think of it as if a memory fence is added **before** the load.

There are two more orders, `acq_rel` and `consume`, which we will not discuss. **For this lab, especially for the mandatory part of the lab, you should only use the default memory order, that is, you should only use the implicit functions.** However, while solving the lab, be mindful of possible optimization opportunities, as we will ask you about these in the bonus part.

A word of warning: *When writing multithreaded applications, it might sometimes appear while testing that your application is actually working correctly, even if there are synchronization errors. There are tools and techniques for testing such applications, but they are beyond the scope of this lab. However, if you are curious (or maybe stuck) take a look at tools like* `drd` *from Valgrind (**https://valgrind.org/ (https://valgrind.org/)**) or Thread Sanitizer from Clang and GCC (**https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual (https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual)**). While writing your code, be mindful of the various variables you are using, which data is shared, and which of them need to be made atomic. Remember that in modern C atomic variables are not just for RMW operations, such as increment, but for all shared variables that cause data races.*

What is provided?

All the files related to the assignment can be downloaded from **Assignments 3 -> Lab 3 Files (https://uppsala.instructure.com/courses/23770/files/1676027?wrap=1)** **(https://uppsala.instructure.com/courses/23770/files/1676027/download?download_frd=1)** . 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.

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` (however, this test is not suitable to measure execution times as it operates on a smaller matrix).
- `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. It is a good idea to look here to figure out what global variables are available.
- `gsi_seq.c` Contains the implementation of the `gsi_calculate` function for the *sequential* GS sweeps. This is a reference implementation that you shouldn't modify.
- `gsi_pth.c` Will contain *your* version of the *parallel* `gsi_calculate` function.

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 to enable the `dprintf` macro.

Parameters for the compiled solvers can be supplied on the command line. Use the `-h` option to see available settings.

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. The parallel version starts 4 threads by default. (These inputs must be powers of 2.)

With the default iteration limit and error tolerance (20 and 1.0) the solution will not converge. This is expected and you don't need to worry about it. The purpose of the lab is to explore the scaling of the kernel through parallelization, not the convergence speed of the algorithm.

Tasks

Edit the `gsi_pth.c` file and implement the `gsi_calculate` function using the pthreads library. You probably do not need to change the `gsi_calculate` function itself, but you will need to change other functions in the file to make it work. 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.

1. Implement the synchronization between threads working on the same *row* in the matrix.
2. Implement the barrier at the end of the iteration. (You may use a pthread barrier, see the documentation for `pthread_barrier_init`.)
   - Extra: Think of a solution without the barrier (a little more efficient), and propose it during examination (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.

Bonus

When you are done with the mandatory part of the lab, continue with the **Lab 3 Bonus (https://uppsala.instructure.com/courses/23770/assignments/57901)** assignment and complete it for a bonus point.