1 Improving the performance of the Gauss-Seidel algorithm

1.1 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:

\[ \Delta u = 0 \text{ in } \Omega \]  
\[ u = 0 \text{ on } \partial \Omega \]  

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 section 1.1.1 if you are interested, otherwise, skip to section 1.1.2.

1.1.1 Mathematical background

We discretizing the problem using a homogeneous grid with the spacing \( h \). Using central differences we can approximate the \( \Delta u \) as:

\[ \Delta u_{i,j} \approx \frac{u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i,j}}{h^2} \]  

The discretized problem is thus:

\[ u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i,j} = 0 \]  

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

\[ Ax = b \]  

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 \( x \) after iteration \( k \). A general description of a sweep in a Gauss-Seidel solver would look as follows:

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

In our case with the discretized Laplace equation [Equation 4], we get:

\[ u_{i,j}^{k+1} = \frac{u_{i-1,j}^{k+1} + u_{i+1,j}^{k+1} + u_{i,j-1}^{k+1} + u_{i,j+1}^{k+1}}{4} \]
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**

$v \leftarrow \frac{u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1}}{4}$

$e \leftarrow e + |u_{i,j} - v|$

$u_{i,j} \leftarrow v$

**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.

We continue to iterate Equation 7 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:

$$\sum_i \sum_j |u_{i,j}^k - u_{i,j}^{k+1}| \leq t \quad (8)$$

What Equation 8 really means is that the algorithm has converged when the difference in the results from two consecutive iterations is small.

### 1.1.2 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 7.

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...
Figure 2: Access pattern for the multi-threaded Gauss-Seidel implementation. The dashed line represents 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.

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.

### 1.2 What is provided?

You get the complete source code for the sequential version, but only a skeleton for the parallel version. The files reside in the folder `/it/kurs/dark2/ht10/gs`. 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`.

- **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[1]

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

---

[Autumn 2010 — Draft]

---

2010-08-17 09:59:37Z r631
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.

1.3 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.

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

1.3.1 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? (i.e. What is the memory model of the machine of 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?

---

2You may use a pthread barrier, see the documentation for pthread_barrier_init.