Introduction to Lab 3

Andreas Sandberg <andreas.sandberg@it.uu.se>
Division of Computer Systems
Dept. of Information Technology
Uppsala University

2011-10-28

What is lab 3?
…who cares about scalability anyway?

The purpose of this assignment is to give insights into:
1. how to program multi-processors
2. introduce the pthreads threading API
3. how different sharing patterns can affect performance
4. show how algorithm design affects scalability

What is Gauss-Seidel?
…and why do I care?

Gauss-Seidel is:
- an iterative linear equation solver.
- ancient and low-performing on its own.
- used as a component in modern multi-grid solvers.

How we will use Gauss-Seidel

We will use Gauss-Seidel to solve the Laplace equation:

\[ \Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \text{ in } \Omega \\
\]

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

Note: The equation above is not a linear equation system!

…but we can approximate it as one using finite differences!

\[ \Delta u_{ij} = \frac{u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{ij}}{h^2} \]
The Gauss-Seidel algorithm

A sweep

Generally:

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

Applied to the Laplace equation (with \( h = 1 \)):

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

Each element is the average of its neighbors. The “new” value is used for the north and west neighbor.

Access pattern

Serial version

Each element is the average of its neighbors. The “new” value is used for the north and west neighbor.

Access pattern

Parallel version

We will parallelize column wise. This requires synchronization between the threads along the “border”. You will implement that synchronization.
We will parallelize column wise. This requires synchronization between the threads along the “border”. You will implement that synchronization.

The Gauss-Seidel algorithm

We define convergence as:

$$\sum_i \sum_j |u_{ij}^k - u_{ij}^{k+1}| \leq t$$

We say that the algorithm has converged when the absolute difference between two iterations is smaller than the tolerance.
What are Posix Threads?

Pthreads is:
- a standardized way to create and synchronize threads
- the default threading API on most Unix systems. This includes:
  - GNU/Linux
  - (Net|Free)...BSD
  - Sun Solaris
  - Apple MacOS X
  - ...

Creating threads

```c
#include <pthread.h>
#include <stdio.h>

static void *my_thread(void *arg) {
    printf("Hello Threads!\n");
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t thread;
    /\ TODO: No error handling :( */
    pthread_create(&thread, NULL, my_thread, NULL);
    pthread_join(thread, NULL);
    return 0;
}
```

Parameters:
- thread Where to store the thread ID.
- attr Attributes for the thread, NULL defaults.
- start_routine Procedure to call in the new thread.
- arg Argument passed to start_routine

Return Value:
0 if successful, error number otherwise.

Waiting for threads to terminate

```c
int pthread_join(
    pthread_t thread,
    void **value_ptr);
```

Parameters:
- thread Thread to wait for.
- value_ptr Pointer to variable to store return value in, NULL to discard return value.

Return Value:
0 if successful, error number otherwise.
Mutexes
Initialization

```c
int pthread_mutex_init(
    pthread_mutex_t *mutex,
    const pthread_mutexattr_t *attr);
```

**Parameters:**
- `mutex` Pointer to mutex to initialize.
- `attr` Pointer to mutex attributes, NULL for default attributes.

**Return Value:**
0 if successful, error number otherwise.

Mutex initialization the easy way, uses default attributes. No need for explicit cleanup.

Mutexes
Cleanup

```c
int pthread_mutex_destroy(
    pthread_mutex_t *mutex);
```

**Parameters:**
- `mutex` Pointer to mutex to destroy.

**Return Value:**
0 if successful, error number otherwise.

Mutexes
Locking

```c
int pthread_mutex_lock(
    pthread_mutex_t *mutex);
int pthread_mutex_unlock(
    pthread_mutex_t *mutex);
```

**Parameters:**
- `mutex` Pointer to mutex to lock or unlock.

**Return Value:**
0 if successful, error number otherwise.
**Mutexes**

Example

```c
static int balance = 512;
static pthread_mutex_t balance_mutex = PTHREAD_MUTEX_INITIALIZER;

static int withdraw(int amount) {
    int ret = 0;
    pthread_mutex_lock(&balance_mutex);
    if (balance > amount) {
        balance -= amount;
        ret = amount;
    }
    pthread_mutex_unlock(&balance_mutex);
    return ret;
}
```

**Barriers**

Initialization

```c
int pthread_barrier_init(  
    pthread_barrier_t *barrier,  
    const pthread_barrierattr_t *attr,  
    unsigned count);
```

*Note:* Barriers are optional in the Posix specification.

*Parameters:*
- `barrier` Pointer to barrier to initialize.
- `attr` Pointer to barrier attributes, NULL for defaults.
- `count` Number of threads to wait for.

*Return Value:*
- 0 if successful, error number otherwise.

Cleanup

```c
int pthread_barrier_destroy(  
    pthread_barrier_t *barrier);
```

*Parameters:*
- `barrier` Pointer to barrier to destroy.

*Return Value:*
- 0 if successful, error number otherwise.

Waiting

```c
int pthread_barrier_wait(  
    pthread_barrier_t *barrier);
```

*Parameters:*
- `barrier` Pointer to barrier to wait for.

*Return Value:*
- PTHREAD_BARRIER_SERIAL_THREAD or 0 on success, error number otherwise.
Barriers
Example

```c
static pthread_barrier_t barrier;

static void init_barrier() {
    pthread_barrier_init(&barrier, NULL, 2);
}

static void destroy_barrier() {
    pthread_barrier_destroy(&barrier);
}

static void do_stuff() {
    /* TODO: Super-fancy algorithm here */
    pthread_barrier_wait(&barrier);
}
```

Documentation
... or the answer to Life, the Universe and Everything

There are two sources of “truth” if you are hacking Unix:

- The Single Unix Specification¹
- Your local system’s man-pages, for example:
  ```
  host$ man man
  host$ man pthreads
  ```


Files in the lab package

- Makefile: Controls compilation. Contains a `test` target.
- gs_common.c: Boring stuff you don’t need to touch.
- gs_interface.h: Contains declarations and documentation for the interface between `gs_common.c` and your GS implementation.
- gsi_seq.c: Sequential reference implementation.
- gsi_pth.c: Write your code here.
Important dates

Groups:

- Prep. Room 1515, now--17:00
  - A 2011-11-01, Room 1412, 08:15–12:00
  - B 2011-11-01, Room 1412, 13:15–17:00
  - C 2011-11-03, Room 1412, 08:15–12:00

- Deadline: 2011-11-07 15:14

Summary

You will:

- Parallelize a Gauss Seidel implementation using Pthreads and flag synchronization
- Study the performance of your parallel implementation
- Perform architecture specific optimizations on the parallel application
- Complete lab manual on the course homepage

And remember...

And remember...

Thou shalt study thy libraries and strive not to reinvent them without cause, that thy code may be short and readable and thy days pleasant and productive.

\[^{2}\text{http://www.it.uu.se/edu/course/homepage/avdark/ht11}\]

\[^{3}\text{http://www.lysator.liu.se/c/ten-commandments.html}\]