**Introduction to 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:

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

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

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

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

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

A sweep

Generally:

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

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

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

Testing for convergence

We define convergence as:

\[ \sum_i \sum_j |u_{i,j}^k - u_{i,j}^{k+1}| \leq t \]

We say that the algorithm has converged when the absolute difference between two iterations is smaller than the tolerance.

Access pattern

Serial version

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

Parallel version

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

Introduction to Lab 3
What is 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
int pthread_create(
    pthread_t *thread,
    const pthread_attr_t *attr,
    void (*start_routine)(void *),
    void *arg);
```

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.

Thread creation
An example

```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;
}
```
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.

```c
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

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

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-lancy 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

¹http://www.unix.org/single_unix_specification/

Important dates

Groups:
- Prep. 2010-10-12, Room 1549, 10–12:00
- A 2010-10-13, Room 1549, 08:15–12:00
- B 2010-10-14, Room 1549, 08:15–12:00
- C 2010-10-14, Room 1549, 13:15–17:00
- Deadline: See course homepage

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 homepage²

²http://www.it.uu.se/edu/course/homepage/avdark/ht10
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.³

³http://www.lysator.liu.se/c/ten-commandments.html

Avdark'10 | Introduction to Lab 3