Introduction to Lab 3

Mahdad Davari <mahdad.davari@it.uu.se>

Division of Computer Systems
Dept. of Information Technology
Uppsala University

2013-10-28

In this lab

- Parallel programming using pthreads
- A real-world example of parallelising sequential code for multicores (Gauss-Seidel numerical method)
- Putting all together parallel programming techniques, data access pattern, and cache usage when writing parallel code for multicores
- A lot of mathematics :~(

Parallel programming paradigms

- Process-level
  - MPI (Message Passing Interface) library
    - Process creation
    - Explicit message “send” and “receive”, synchronizations, barriers
- Thread-level
  - POSIX Threads (pthreads) library
    - Explicit thread creation and control by programmer
  - Open Multi-Processing (OpenMP) Compiler Extension
    - Implicit thread creation and control by compiler

Some pthread types and routines ...

```
#include <pthread.h>

pthread_t       thread;

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.

Some pthreads types and functions ...

#include <pthread.h>

Waiting for threads to terminate

```
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.

Some pthreads types and functions ...

Thread creation

An example

```
#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;

        pthread_create(&thread, NULL,
                        my_thread, NULL);
        pthread_join(thread, NULL);
        return 0;
}
```

Some pthreads types and functions ...

```
#include <pthread.h>

pthread_mutex_t             mutex;
```

```
pthread_mutex_t mutex =
        PTHREAD_MUTEX_INITIALIZER;
```

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

```
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.

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

Parameters:

mutex Pointer to mutex to destroy.

Return Value:

0 if successful, error number otherwise.

Some pthreads types and functions ...

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

```
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.

Some pthreads types and functions ...

Mutexes

Example

```
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;

}
```

Some pthreads types and functions ...

```
#include <pthread.h>

pthread_barrier_t       barrier;
```

```
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.

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

Parameters:

barrier Pointer to barrier to destroy.

Return Value:
0 if successful, error number otherwise.

Some pthreads types and functions ...

```
#include <pthread.h>

pthread_barrier_t       barrier;
```

```
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.

Some pthreads types and functions ...

```
#include <pthread.h>
pthread_barrier_t       barrier;

 void * thread() {
        // some computation
        pthread_barrier_wait(&barrier);
        // rest of computation
        return;
 }
```

Some pthreads types and functions ...

```
#include <pthread.h>

pthread_barrier_t       barrier;


 void * thread() {

        // some computation

        int ret = pthread_barrier_wait(&barrier);

        if (ret != 0 && ret != PTHREAD_BARRIER_SERIAL_THREAD) {
                // error handling routine
        } else if (ret == PTHREAD_BARRIER_SERIAL_THREAD) {
                // serial task
        }

        // rest of computation

        return;

 }
```

More information on pthread ...

- Lawrence Livermore National Laboratory [1]
- Single Unix Specification [2]
- Your local system's manual pages:
  - host$ man man
  - host$ man pthreads
- Variety of material available online

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

Gauss-Seidel Numerical Method

- Gauss-Seidel method:
  - An iterative method for solving linear systems of equations [1]

$$A\mathbf{x} = \mathbf{b}$$

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

1. http://en.wikipedia.org/

Gauss-Seidel Numerical Method

- Gauss-Seidel method: [1]
  - matrix A is decomposed into a lower triangular component , and a strictly upper triangular component U:

$$A = L_* + U \qquad \text{where} \qquad L_* = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \quad U = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}.$$

$$A\mathbf{x} = \mathbf{b}$$

$$L_*\mathbf{x} = \mathbf{b} - U\mathbf{x}$$

$$\mathbf{x}^{(k+1)} = L_*^{-1}(\mathbf{b} - U\mathbf{x}^{(k)}).$$

1. http://en.wikipedia.org/

Gauss-Seidel Numerical Method

- Gauss-Seidel method:[1] solved sequentially using forward substitution

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j<i} a_{ij} x_j^{(k+1)} - \sum_{j>i} a_{ij} x_j^{(k)} \right), \quad i, j = 1, 2, \ldots, n.$$

$$\begin{array}{rcrcrcl} l_{1,1}x_1 & & & & & = & b_1 \\ l_{2,1}x_1 & + & l_{2,2}x_2 & & & = & b_2 \\ \vdots & & \vdots & \ddots & & & \vdots \\ l_{m,1}x_1 & + & l_{m,2}x_2 & + \cdots + & l_{m,m}x_m & = & b_m \end{array}$$

$$\begin{aligned} x_1 &= \frac{b_1}{l_{1,1}}, \\ x_2 &= \frac{b_2 - l_{2,1}x_1}{l_{2,2}}, \\ &\vdots \\ x_m &= \frac{b_m - \sum_{i=1}^{m-1} l_{m,i}x_i}{l_{m,m}}. \end{aligned}$$

1. http://en.wikipedia.org/

Gauss-Seidel Numerical Method

- Gauss-Seidel method in lab3:

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

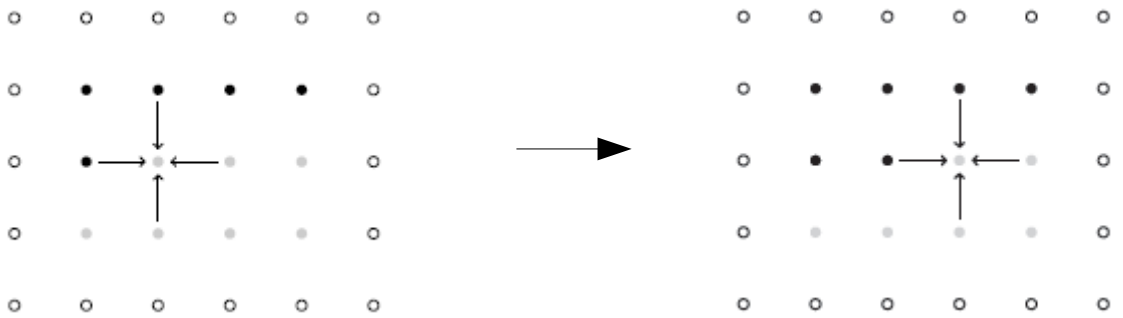
Algorithm 1 Gauss-Seidel solver for the Laplace equation on an $n \times m$ matrix, with the tolerance t.

Require: $n, m \geq 2$

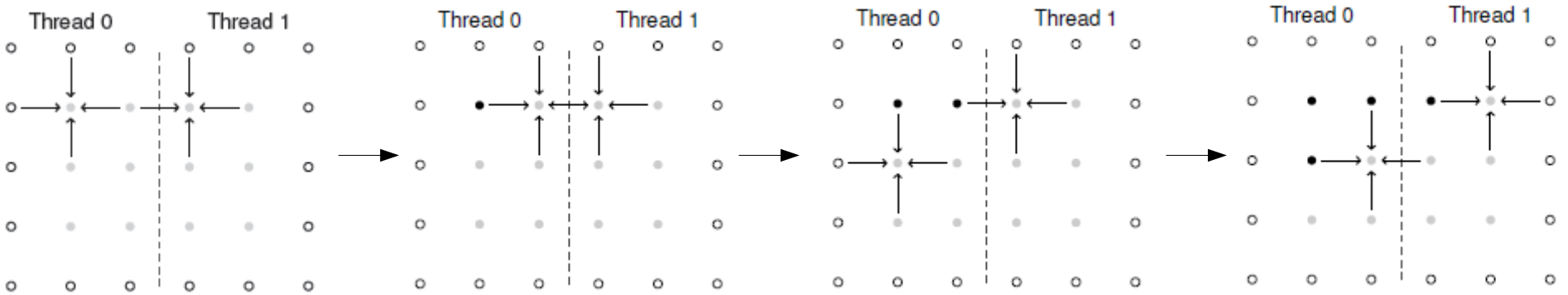
```
repeat
  e ⇐ 0
  for i = 1 to n − 1 do
    for j = 1 to m − 1 do
      v ⇐ (u_{i−1,j} + u_{i,j−1} + u_{i+1,j} + u_{i,j+1}) / 4
      e ⇐ e + |u_{i,j} − v|
      u_{i,j} ⇐ v
    end for
  end for
until e ≤ t
```

Gauss-Seidel Numerical Method

- Access pattern for serial version:



- Access pattern for parallel version:

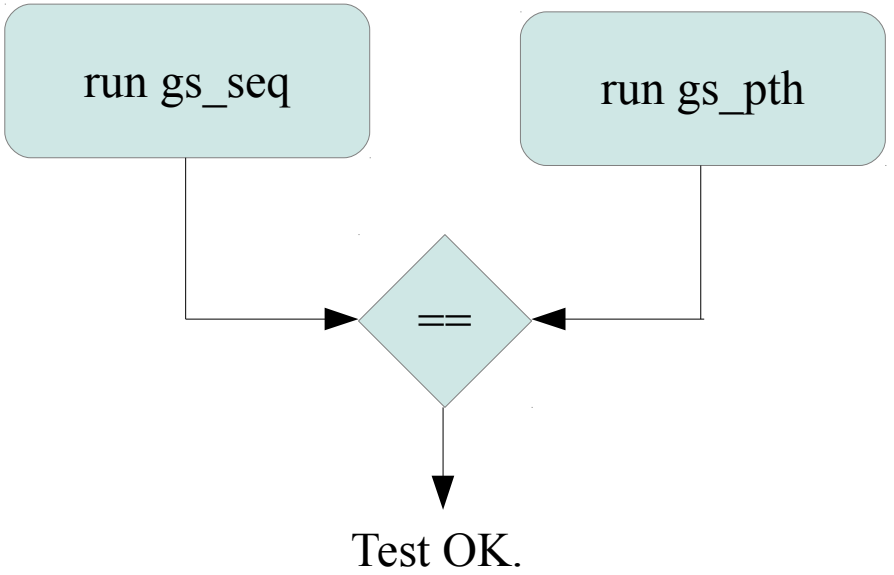


In this lab

- We will complete the synchronization between threads that implement the parallel version of Gauss-Seidel solver.
- lab3 skeleton files available on Piazza (a revised version uploaded) and extract the package:
  - Makefile: automates compilation and run
    - make gs_seq
    - make gs_pth
    - make test
  - gs_common.c: glue code for running lab3 – it is recommended to have a quick look into this file (the beginning) to see how default values are defined.
  - gs_interface.h
  - gsi_seq.c: sequential implementation of GS sweep
  - gsi_pth.c: your parallel version of GS sweep
  - solution.c: don't peek!

In this lab

- To test your solution:
  - make test



Please ignore “test did NOT converge!”

In this lab – a closer look

```
gsi_calculate () {
        .
        .
        pthread_create (&threads[t].thread, NULL, thread_compute, &threads[t]);
        .
        .
        pthread_join (…);
        .
        .
        .
}
```

In this lab – a closer look

```
typedef struct {
        int thread_id;
        pthread_t thread;

        volatile double error;

        /* TASK: Do you need any thread local state for synchronization? */
        /* !END_SOLUTION! */
} thread_info_t;
```

In this lab – a closer look

```
gsi_calculate () {
        .
        .
        pthread_create (&threads[t].thread, NULL, thread_compute, &threads[t]);
        .
        .
        pthread_join (…);
        .
        .
        .
}
```

In this lab – a closer look

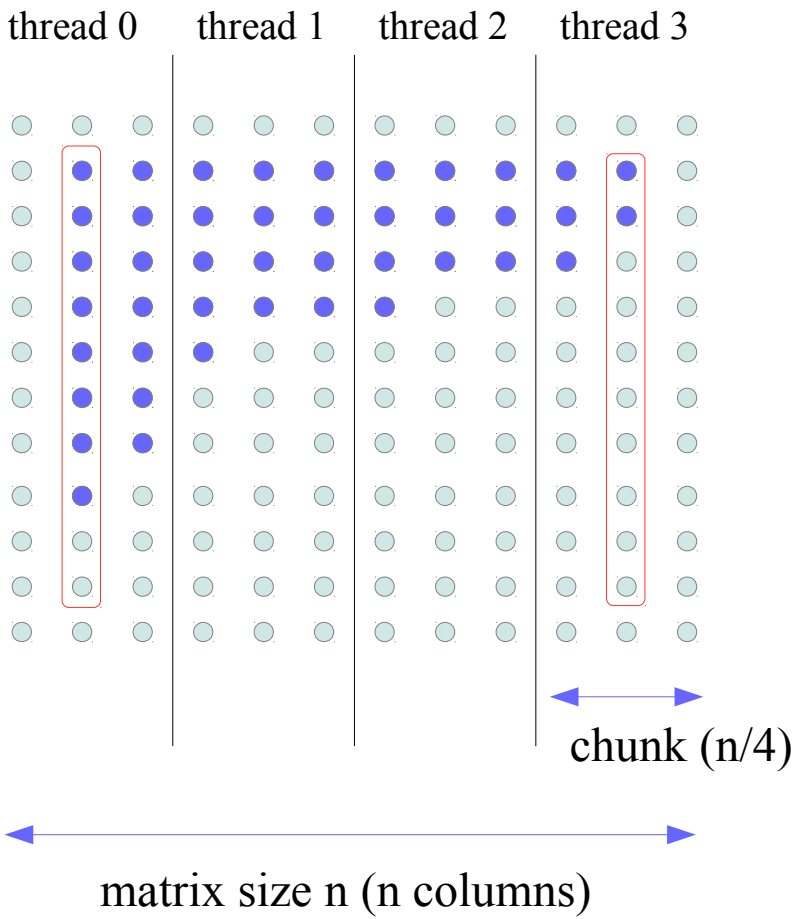
```
thread_compute (…) {
        // TASK compute bounds
        for (iteration = 0; < gs_iterations && global_error > gs_tolerance; ++) {
                thread_sweep(tid, iter, lbound, rbound);
                // TASK update global error → which thread should update this ?
                // TASK barrier: all threads should finish the current iteration before next one
        }
}
```

In this lab – a closer look

```
thread_sweep (…) {

        threads[tid].error = 0.0;

        for (int row = 1; row < gs_size - 1; row++) {

                // TASK when should we start sweeping a row ?

              for (int col = lbound; col < rbound; col++) {
                        // sweep the row
                }

                // TASK tell the thread to the right to continue
        }

}
```

In this lab – a closer look

- We use four threads to parallelize GS sweep




```
for (int row = 1; row < gs_size - 1; row++) {
        for (int col = lbound; col < rbound; col++) {
                gs_matrix[GS_INDEX(row + 1, col)] +
                gs_matrix[GS_INDEX(row - 1, col)] +
                gs_matrix[GS_INDEX(row, col + 1)] +
                gs_matrix[GS_INDEX(row, col – 1)]);
        }
}
```

In this lab – a closer look

- Pay attention to gsi_init() and gsi_finish()
- Define your barrier before initialization and use
- Notice that first thread does not need to synchronize with its left thread (use tid)
- To debug, set DEBUG to 1 in gsi_pth.c (#define DEBUG 1)

- Lab tasks review
  - Implement synchronization between threads working on the same row in matrix
  - Barrier implementation at the end of each iteration
    - extra: a solution without using barrier
  - Speed-up your design by modifying how error variable is stored (modify thread_info_t)

Lab sign-up and groups

Sign up using doodle: piazza → Course Page → Resources → General Resources

Lab Preparation: 2013-10-28, 13:00 ~ 17:00, room 1412

Group A: 2013-10-29, 08:00 ~ 12:00, room 1412

Group B: 2013-10-31, 08:00 ~ 12:00, room 1412

Group C: 2013-11-01, 13:00 ~ 17:00, room 1412

Good luck!