Shared memory parallelism

Lu Li
Linköping University
Parts of slides developed by Usman Dastgeer

TDDC 78 Labs: Memory-based Taxonomy

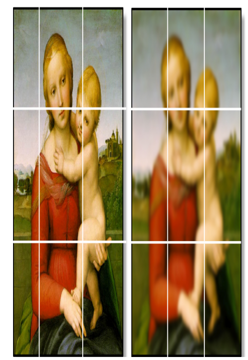
Memory	Labs	Use
Distributed	1	MPI
Shared	2 & 3	POSIX threads & OpenMP
Distributed	5	MPI

LAB 4 (tools). Saves your time for LAB 5.

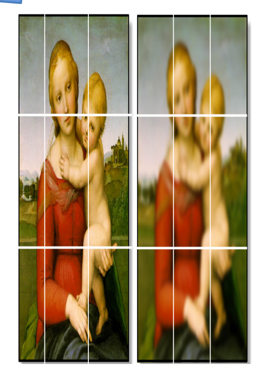
Lab-2: Image Filters with PThreads

- **Blur** & **Threshold**
  - See compendium for details

Threshold

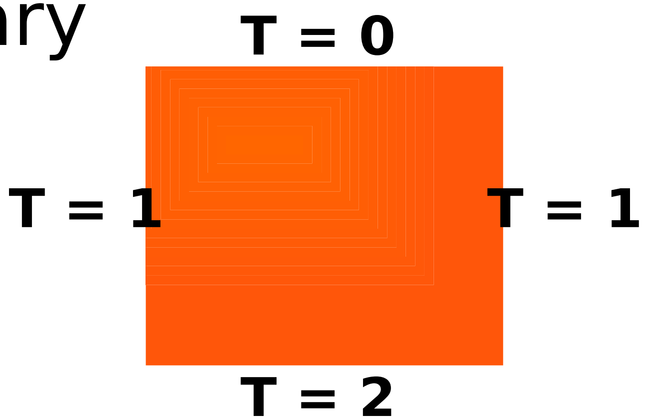


Blur



Lab 3 – Stationary Heat Conduction

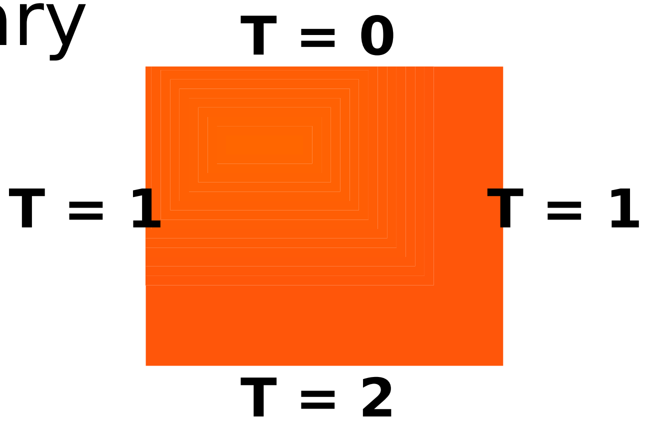
- **Problem**
  - Find stationary temperature distribution in a square given some boundary temperature distribution
  - SHMEM, OpenMP
  - Serial code in Fortran



- **Solution**
  - Requires solving differential equation
  - Iterative Jacobi method
  - Detailed algorithm in Compendium
- **Primary concern**
  - Synchronize access

Lab 3 – Stationary Heat Conduction

- **Problem**
  - Find stationary temperature distribution in a square given some boundary temperature distribution
  - SHMEM, OpenMP
  - Serial code in Fortran
- **Solution**
  - Requires solving differential equation
  - Iterative Jacobi method
  - Detailed algorithm in Compendium
- **Primary concern**
  - Synchronize access, *O(N)* extra memory



Main Concept: Synchronization (1)

- Different from MPI’s Send-Receive
- Thread safety = protect shared data
- Deterministic behavior

Main Concept: Synchronization (2)

- Synchronization objects:
  - **Mutex Locks** ()
    - Serialize access to shared resources
    - **Mutual Exclusion**!
  - **Semaphores**
    - Block a thread until count is positive
    - Set of resources (>1).
  - **Condition Variables**
    - Block a thread until a (global) condition is true.

Mutex lock example

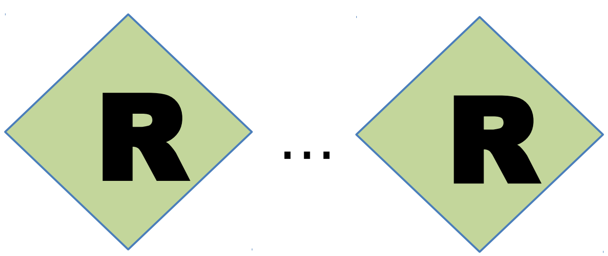
```c
#include<pthread.h>

pthread_mutex_t count_mutex = ... ;
long count;

void increment_count() {
    pthread_mutex_lock(&count_mutex);
    count = count + 1;
    pthread_mutex_unlock(&count_mutex);
}

long get_count() {
    long c;
    pthread_mutex_lock(&count_mutex);
    c = count;
    pthread_mutex_unlock(&count_mutex);
    return (c);
}
```

Main Concept: Synchronization (2)

- Synchronization objects:
  - **Mutex Locks** ()
    - Serialize access to shared resources
    - **Mutual Exclusion**!
  - **Semaphores**
    - Block a thread until count is positive
    - Set of resources (>1).
  - **Condition Variables**
    - Block a thread until a (global) condition is true.

Semaphores

- Coordinate access to resources
  - **Initialize** to the number of free resources
  - Atomically **increment** the count when resources are added
  - Atomically **decrement** the count when resources are removed.
  - Threads **block and wait** until the count becomes greater than zero.



Main Concept: Synchronization (2)

- Synchronization objects:
  - **Mutex Locks** ()
    - Serialize access to shared resources
    - **Mutual Exclusion**!
  - **Semaphores**
    - Block a thread until count is positive
    - Set of resources (>1).
  - **Condition Variables**
    - Block a thread until a (global) condition is true.

Conditional variables example

```
pthread_mutex_t count_lock;
pthread_cond_t count_positive;
long count;

decrement_count() {
    pthread_mutex_lock(&count_lock);
    while (count <= 0)
        pthread_cond_wait(&count_positive, &count_lock);
    count = count - 1;
    pthread_mutex_unlock(&count_lock);
}

increment_count() {
    pthread_mutex_lock(&count_lock);
    count = count + 1;
    if (count > 0)
        pthread_cond_signal(&count_positive);
    pthread_mutex_unlock(&count_lock);
}
```

Passing a single parameter

```
...
void *PrintHello(void *threadId) {
long tId;
tId = *((long *)threadId);
printf("Hello World! It's thread #%ld!\n", tId);
return NULL;
}

...
long param[NUM_THREADS];
...
for(t=0; t<NUM_THREADS; t++) {
  param[t] = t;
  printf("Creating thread %ld\n", t);

  ret = pthread_create(&threads[t],NULL, PrintHello, (void *)&param[t]);
     ...
}
```

Passing multiple parameters

```
struct thread_data{
    int threadId;
    char *msg;
};

struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *tParam) {
    struct thread_data *myData;
    …
    myData = (struct thread_data *) tParam;
    taskId = myData->threadId;
    helloMsg = myData->msg;
    …
}

int main (int argc, char *argv[]) {
...
thread_data_array[t].threadId = t;
thread_data_array[t].Msg = msgPool[t];
rc = pthread_create(&threads[t], NULL, PrintHello,
(void *) &thread_data_array[t]);
```

Compiling and linking

- Don't forget to include
  - pthread.h, semaphore.h
- Link with
  - -lpthread, -lposix4

Typical problems (1)

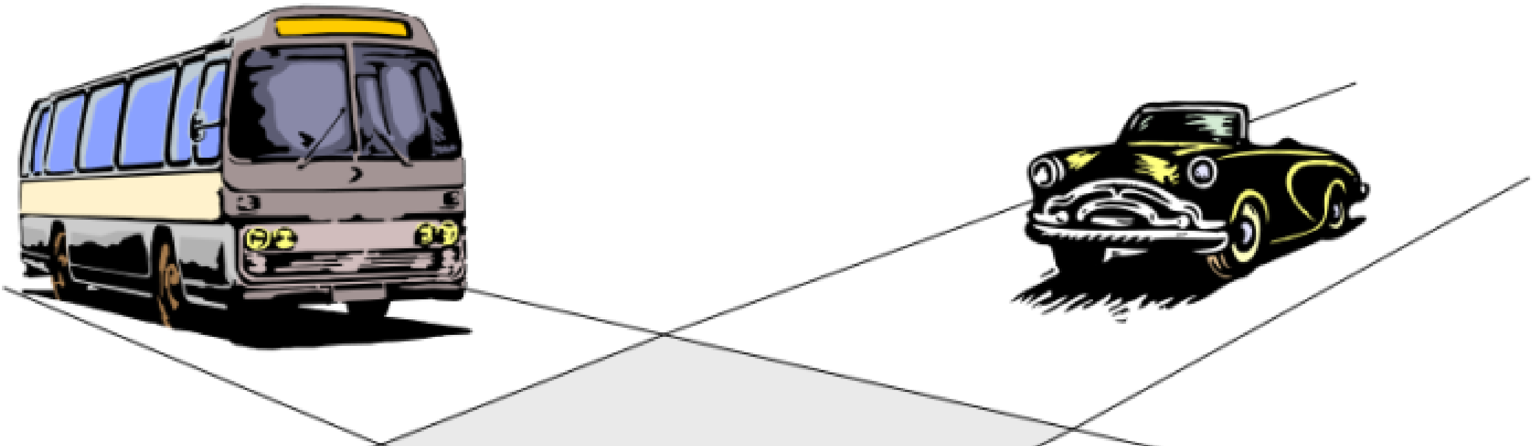
- **Uninitialized variables**
  - Uninitialized synchronization objects lead to strange behavior
  - **Tip:** check the return codes!
- **Poor performance**
  - Too many synchronizations
  - Cache effects kill gains of using multiprocessors

Typical problems (2)

- **Deadlocks** (>= 2 waiting for each other)

Typical problems (3)

- **Race conditions (one misbehaves)**



```
long param = ...;
...
for(t=0; t<NUM_THREADS; t++) {
  printf("Creating thread %ld\n", t);
  ret = pthread_create(&threads[t], NULL, PrintHello, (void *)&param);
    ...
}
```

Possible race condition!

Use synchronization

Summary and goals for your lab

- Understand
  - Threads and their use
  - Synchronization vs. Send / Receive
  - Resource ordering
  - Low level parallelism - PThreads vs.
  - Higher-level specification - OpenMP
- Implement
  - Lab 2, 3