

Programmazione di Sistemi ~~Embedded e~~ Multicore

Teacher: Daniele De Sensi

Recap

Recap

- High-level view of a supercomputer
- How to create/join pthreads
- Hello world example

Questions?

a_{00}	a_{01}	\cdots	$a_{0,n-1}$
a_{10}	a_{11}	\cdots	$a_{1,n-1}$
\vdots	\vdots		\vdots
a_{i0}	a_{i1}	\cdots	$a_{i,n-1}$
\vdots	\vdots		\vdots
$a_{m-1,0}$	$a_{m-1,1}$	\cdots	$a_{m-1,n-1}$

x_0
 x_1
 \vdots
 x_{n-1}

 $=$

y_0
y_1
\vdots
$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1}$
\vdots
y_{m-1}

MATRIX-VECTOR MULTIPLICATION IN PTHREADS

Serial pseudo-code

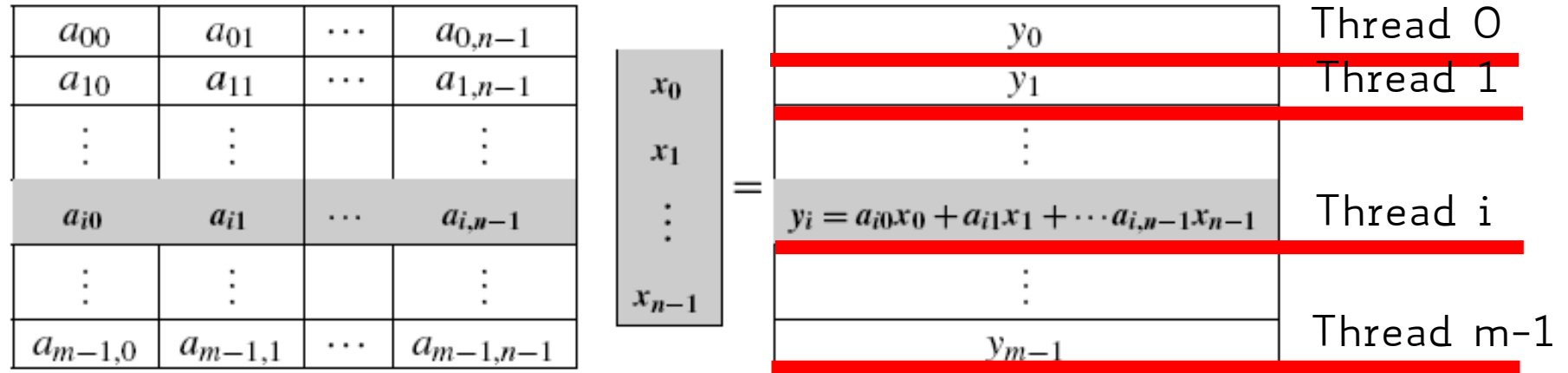
```
/* For each row of A */  
for (i = 0; i < m; i++) {  
    y[i] = 0.0;  
    /* For each element of the row and each element of x */  
    for (j = 0; j < n; j++)  
        y[i] += A[i][j]* x[j];  
}
```

$$y_i = \sum_{j=0}^{n-1} a_{ij}x_j$$

How to do this in parallel?

Partition the matrix (by row) among threads, replicate the vector

Intuition



- In principle, you should try to avoid having more threads than cores
- There are situations where it might make sense (we'll discuss those later)
- Anyway, in general you will partition the m rows across t threads, with $t < m$
- Each processes m/t rows
- Thread q processes rows starting from $q \times \frac{m}{t}$ to $(q + 1) \times \frac{m}{t} - 1$
- Note: We do not need to do scatter/broadcast, every thread accesses the same memory/matrix/vector

Pthreads matrix-vector multiplication

```
void *Pth_mat_vect(void* rank) {  
    long my_rank = (long) rank;  
    int i, j;  
    int local_m = m/thread_count;  
    int my_first_row = my_rank*local_m;  
    int my_last_row = (my_rank+1)*local_m - 1;  
  
    for (i = my_first_row; i <= my_last_row; i++) {  
        y[i] = 0.0;  
        for (j = 0; j < n; j++)  
            y[i] += A[i][j]*x[j];  
    }  
  
    return NULL;  
} /* Pth_mat_vect */
```


How many threads should we run?

- In principle, you should try to avoid having more threads than cores
- There are situations where it might make sense (we'll discuss those later)
- How to check how many cores do you have?

```
$ lscpu | grep -E '^Thread|^Core|^Socket|^CPU\('
```

```
CPU(s):                32
```

```
Thread(s) per core:    2
```

```
Core(s) per socket:    8
```

```
Socket(s):             2
```

Questions?

Critical Sections

Estimating π

$$\pi = 4 \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + (-1)^n \frac{1}{2n+1} + \cdots \right)$$

```
double factor = 1.0;  
double sum = 0.0;  
for (i = 0; i < n; i++, factor = -factor) {  
    sum += factor/(2*i+1);  
}  
pi = 4.0*sum;
```

Parallel algorithm: each thread computes a subset of that series

A thread function for computing π

```
void* Thread_sum(void* rank) {  
    long my_rank = (long) rank;  
    double factor;  
    long long i;  
    long long my_n = n/thread_count;  
    long long my_first_i = my_n*my_rank;  
    long long my_last_i = my_first_i + my_n;  
  
    return NULL;  
} /* Thread_sum */
```

Using a dual core processor

	<i>n</i>			
	10^5	10^6	10^7	10^8
π	3.14159	3.141593	3.1415927	3.14159265
1 Thread	3.14158	3.141592	3.1415926	3.14159264
2 Threads	3.14158	3.141480	3.1413692	3.14164686

Note that as we increase n , the estimate with two threads diverge from the real value

A thread function for computing π

```
void* Thread_sum(void* rank) {  
    long my_rank = (long) rank;  
    double factor;  
    long long i;  
    long long my_n = n/thread_count;  
    long long my_first_i = my_n*my_rank;  
    long long my_last_i = my_first_i + my_n;  
  
    if (my_first_i % 2 == 0) /* my_first_i is even */  
        factor = 1.0;  
    else /* my_first_i is odd */  
        factor = -1.0;  
  
    for (i = my_first_i; i < my_last_i; i++, factor = -factor) {  
        sum += factor/(2*i+1);  
    }  
  
    return NULL;  
} /* Thread_sum */
```

1. Load 'factor' into register
2. Load 'i' into register
3. Load 'sum' into register
4. Compute $\text{sum} + \text{factor}/(2i + 1)$
5. Store the result into 'sum'

Possible race condition

```
y = Compute(my_rank);  
x = x + y;
```


Possible race condition

```
y = Compute(my_rank);  
x = x + y;
```

Time	Thread 0	Thread 1
1	Started by main thread	

Possible race condition

```
y = Compute(my_rank);  
x = x + y;
```

Time	Thread 0	Thread 1
1	Started by main thread	
2	Call Compute ()	Started by main thread

Possible race condition

```
y = Compute(my_rank);  
x = x + y;
```

Time	Thread 0	Thread 1
1	Started by main thread	
2	Call Compute ()	Started by main thread
3	Assign y = 1	Call Compute ()

Possible race condition

```
y = Compute(my_rank);  
x = x + y;
```

Time	Thread 0	Thread 1
1	Started by main thread	
2	Call Compute ()	Started by main thread
3	Assign y = 1	Call Compute ()
4	Put x=0 and y=1 into registers	Assign y = 2

Possible race condition

```
y = Compute(my_rank);  
x = x + y;
```

Time	Thread 0	Thread 1
1	Started by main thread	
2	Call Compute ()	Started by main thread
3	Assign y = 1	Call Compute ()
4	Put x=0 and y=1 into registers	Assign y = 2
5	Add 0 and 1	Put x=0 and y=2 into registers

Possible race condition

```
y = Compute(my_rank);  
x = x + y;
```

Time	Thread 0	Thread 1
1	Started by main thread	
2	Call Compute ()	Started by main thread
3	Assign y = 1	Call Compute ()
4	Put x=0 and y=1 into registers	Assign y = 2
5	Add 0 and 1	Put x=0 and y=2 into registers
6	Store 1 in memory location x	Add 0 and 2

Possible race condition

```
y = Compute(my_rank);  
x = x + y;
```

Time	Thread 0	Thread 1
1	Started by main thread	
2	Call Compute ()	Started by main thread
3	Assign y = 1	Call Compute ()
4	Put x=0 and y=1 into registers	Assign y = 2
5	Add 0 and 1	Put x=0 and y=2 into registers
6	Store 1 in memory location x	Add 0 and 2
7		Store 2 in memory location x

Questions?

Possible solution: Busy-Waiting

- A thread repeatedly tests a condition, but, effectively, does no useful work until the condition has the appropriate value.

```
y = Compute(my_rank);  
while (flag != my_rank);  
x = x + y;  
flag++;
```

flag initialized to 0 by main thread

Possible danger: Optimizing compilers

This code:

```
y = Compute(my_rank);  
while (flag != my_rank);  
x = x + y;  
flag++;
```

Could be rearranged by the compiler as:

```
y = Compute(my_rank);  
x = x + y;  
while (flag != my_rank);  
flag++;
```

(the compiler does not know if the code is going to use threads or not. It might rearrange the code this way because it might believe that it is going to make a better use of registers)

Pthreads global sum with busy-waiting

```
void* Thread_sum(void* rank) {
    long my_rank = (long) rank;
    double factor;
    long long i;
    long long my_n = n/thread_count;
    long long my_first_i = my_n*my_rank;
    long long my_last_i = my_first_i + my_n;

    if (my_first_i % 2 == 0)
        factor = 1.0;
    else
        factor = -1.0;

    for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
        while (flag != my_rank);
        sum += factor/(2*i+1);
        flag = (flag+1) % thread_count;
    }

    return NULL;
} /* Thread_sum */
```

If I run this with n threads, this is slower than sequential code, why?

Global sum function with critical section after loop

```
void* Thread_sum(void* rank) {  
    long my_rank = (long) rank;  
    double factor, my_sum = 0.0;  
    long long i;  
    long long my_n = n/thread_count;  
    long long my_first_i = my_n*my_rank;  
    long long my_last_i = my_first_i + my_n;  
  
    if (my_first_i % 2 == 0)  
        factor = 1.0;  
    else  
        factor = -1.0;  
  
    for (i = my_first_i; i < my_last_i; i++, factor = -factor)  
        my_sum += factor/(2*i+1);  
  
    while (flag != my_rank);  
    sum += my_sum;  
    flag = (flag+1) % thread_count;  
  
    return NULL;  
} /* Thread_sum */
```

Questions?

Mutexes

Mutexes

- A thread that is busy-waiting may continually use the CPU accomplishing nothing.
- Mutex (mutual exclusion) is a special type of variable that can be used to restrict access to a critical section to a single thread at a time.

Mutexes

- Used to guarantee that one thread “excludes” all other threads while it executes the critical section.
- The Pthreads standard includes a special type for mutexes: `pthread_mutex_t`.

```
int pthread_mutex_init(  
    pthread_mutex_t*      mutex_p    /* out */  
    const pthread_mutexattr_t* attr_p  /* in  */);
```


Mutexes

- When a Pthreads program finishes using a mutex, it should call

```
int pthread_mutex_destroy(pthread_mutex_t* mutex_p /* in/out */);
```

- In order to gain access to a critical section a thread calls

```
int pthread_mutex_lock(pthread_mutex_t* mutex_p /* in/out */);
```

- When a thread is finished executing the code in a critical section, it should call

```
int pthread_mutex_unlock(pthread_mutex_t* mutex_p /* in/out */);
```

- The non blocking version of lock is:

```
int pthread_mutex_trylock(pthread_mutex_t* mutex_p /* in/out */);
```

Starvation

- Starvation happens when the execution of a thread or a process is suspended or disallowed for an indefinite amount of time, although it is capable of continuing execution.
- Starvation is typically associated with enforcing of priorities or the lack of fairness in scheduling or access to resources.
- If a mutex is locked, the thread is blocked and placed in a queue Q of waiting threads. If the queue Q employed by a semaphore is a FIFO queue, no starvation will occur.

Deadlocks

- **Deadlock:** is any situation in which no member of some group of entities can proceed because each waits for another member, including itself, to take action, such as sending a message or, more commonly, releasing a lock.
- E.g., locking mutexes in reverse order

```
/* Thread A */  
pthread_mutex_lock(&mutex1);  
pthread_mutex_lock(&mutex2);
```

```
/* Thread B */  
pthread_mutex_lock(&mutex2);  
pthread_mutex_lock(&mutex1);
```

Global sum function using mutex

```
void* Thread_sum(void* rank) {
    long my_rank = (long) rank;
    double factor;
    long long i;
    long long my_n = n/thread_count;
    long long my_first_i = my_n*my_rank;
    long long my_last_i = my_first_i + my_n;
    double my_sum = 0.0;

    if (my_first_i % 2 == 0)
        factor = 1.0;
    else
        factor = -1.0;

    for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
        my_sum += factor/(2*i+1);
    }
    pthread_mutex_lock(&mutex);
    sum += my_sum;
    pthread_mutex_unlock(&mutex);

    return NULL;
} /* Thread_sum */
```

Threads	Busy-Wait	Mutex
1	2.90	2.90
2	1.45	1.45
4	0.73	0.73
8	0.38	0.38

Run-times (in seconds) of π programs using $n = 10^8$ terms on a system with two four-core processors.

In both cases, the critical section is outside the loop

Questions?

Producer-consumer Synchronization and Semaphores

Issues

- Busy-waiting enforces the order threads access a critical section.
- Using mutexes, the order is left to chance and the system.
- There are applications where we need to control the order threads access the critical section.

Example: message exchange in a ring (receive from left, send to right)

```
/* messages has type char**. It's allocated in main. */  
/* Each entry is set to NULL in main. */  
void *Send_msg(void* rank) {  
    long my_rank = (long) rank;  
    long dest = (my_rank + 1) % thread_count;  
    long source = (my_rank + thread_count - 1) % thread_count;  
    char* my_msg = malloc(MSG_MAX*sizeof(char));  
  
    sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);  
    messages[dest] = my_msg;  
  
    if (messages[my_rank] != NULL)  
        printf("Thread %ld > %s\n", my_rank, messages[my_rank]);  
    else  
        printf("Thread %ld > No message from %ld\n", my_rank, source);  
  
    return NULL;  
} /* Send_msg */
```

Issue: Some threads might read messages[dest] before the other thread put something in there

How to fix it?

- We could fix it with busy waiting, but would have the same problems discussed before

```
while (messages[my rank] == NULL);  
    printf("Thread %ld > %s\n", my  
        rank, messages[my rank]);
```

- In principle it is possible to fix it with mutex (but in a complex way)
- POSIX provides a better way: **semaphores** (it is not part of pthread, could be not available on macOS)

Syntax of the various semaphore functions

```
#include <semaphore.h>
```

← Semaphores are not part of
Pthreads;
you need to add this.

```
int sem_init(  
    sem_t*    semaphore_p    /* out */,  
    int        shared         /* in  */,  
    unsigned   initial_val    /* in  */);
```

↑ Semaphores can be shared also
among processes (shared !=0)

```
int sem_destroy(sem_t*    semaphore_p    /* in/out */);  
int sem_post(sem_t*      semaphore_p    /* in/out */);  
int sem_wait(sem_t*      semaphore_p    /* in/out */);
```

Semaphores

`sem_wait(sem_t *sem)` blocks if the semaphore is 0. If the semaphore is > 0, it will *decrement* the semaphore and proceed.

`sem_post(sem_t *sem)` if there is a thread waiting in `sem_wait()`, that thread can proceed with the execution. Otherwise, the semaphore is *incremented*.

`sem_getvalue(sem_t *sem, int *sval)` places the current value of the semaphore pointed to `sem` into the integer pointed to by `sval`.

Notes:

- Mutexes are binary. Semaphores are **unsigned int**.
- Mutexes start unlocked. Semaphores start with initial value.
- Mutexes are usually locked/unlocked by the same thread. Semaphores are usually increased/decreased by different threads.

Program 4.8: Using semaphores so that threads can send messages

```
1  /* messages is allocated and initialized to NULL in main */
2  /* semaphores is allocated and initialized to 0 (locked) in
   main */
3  void* Send_msg(void* rank) {
4      long my_rank = (long) rank;
5      long dest = (my_rank + 1) % thread_count;
6      char* my_msg = malloc(MSG_MAX*sizeof(char));
7
8      sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
9      messages[dest] = my_msg;
10     sem_post(&semaphores[dest])
        /* 'Unlock' the semaphore of dest */
11
12     /* Wait for our semaphore to be unlocked */
13     sem_wait(&semaphores[my_rank]);
14     printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
15
16     return NULL;
17 } /* Send_msg */
```

Note: This problem does not have a critical section. It has a type of synchronization known as **producer-consumer**

Questions?

Barriers and Condition Variables

Barriers

- Synchronizing the threads to make sure that they all are at the same point in a program is called a barrier.
- No thread can cross the barrier until all the threads have reached it.

Using barriers to time the slowest thread

```
/* Shared */
double elapsed_time;
. . .
/* Private */
double my_start, my_finish, my_elapsed;
. . .
Synchronize threads;
Store current time in my_start;
/* Execute timed code */
. . .
Store current time in my_finish;
my_elapsed = my_finish - my_start;

elapsed = Maximum of my_elapsed values;
```

Using barriers for debugging

```
point in program we want to reach;  
barrier;  
if (my_rank == 0) {  
    printf("All threads reached this point\n");  
    fflush(stdout);  
}
```

Busy-waiting and a Mutex

- Implementing a barrier using busy-waiting and a mutex is straightforward.
- We use a shared counter protected by the mutex.
- When the counter indicates that every thread has entered the critical section, threads can leave the critical section.

Busy-waiting and a Mutex

```
/* Shared and initialized by the main thread */
int counter; /* Initialize to 0 */
int thread_count;
pthread_mutex_t barrier_mutex;
. . .

void* Thread_work(. . .) {
    . . .
    /* Barrier */
    pthread_mutex_lock(&barrier_mutex);
    counter++;
    pthread_mutex_unlock(&barrier_mutex);
    while (counter < thread_count);
    . . .
}
```

What if we want to use the barrier again?

Someone must reset counter to 0. Who and when?


If a thread reset it to 0, this might be reset
before all the threads see it was equal to
thread_count

Implementing a barrier with semaphores

```
/* Shared variables */
int counter;          /* Initialize to 0 */
sem_t count_sem;      /* Initialize to 1 */
sem_t barrier_sem;    /* Initialize to 0 */
. . .
void* Thread_work(...) {
    . . .
    /* Barrier */
    sem_wait(&count_sem);
    if (counter == thread_count-1) {
        counter = 0;
        sem_post(&count_sem);
        for (j = 0; j < thread_count-1; j++)
            sem_post(&barrier_sem);
    } else {
        counter++;
        sem_post(&count_sem);
        sem_wait(&barrier_sem);
    }
    . . .
}
```

Implementing a barrier with semaphores

```
/* Shared variables */
int counter;          /* Initialize to 0 */
sem_t count_sem;      /* Initialize to 1 */
sem_t barrier_sem;    /* Initialize to 0 */
. . .
void* Thread_work(...) {
    . . .
    /* Barrier */
    sem_wait(&count_sem);
    if (counter == thread_count-1) {
        counter = 0;
        sem_post(&count_sem);
        for (j = 0; j < thread_count-1; j++)
            sem_post(&barrier_sem);
    } else {
        counter++;
        sem_post(&count_sem);
        sem_wait(&barrier_sem);
    }
    . . .
}
```



Do we have the same problem here? Yes, there could be a race condition (check the book)

Condition Variables

- A condition variable is a data object that allows a thread to suspend execution until a certain event or condition occurs.
- When the event or condition occurs another thread can signal the thread to “wake up.”
- A condition variable is always associated with a mutex.
- A condition variable indicates an event; cannot store or retrieve a value from a condition variable

Condition Variables

```
lock mutex;  
if condition has occurred  
    signal thread(s);  
else {  
    unlock the mutex and block;  
    /* when thread is unblocked, mutex is relocked */  
}  
unlock mutex;
```

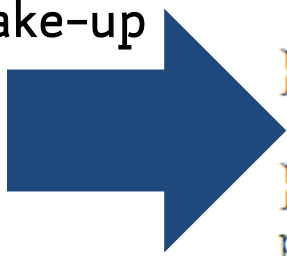

Implementing a barrier with condition variables

```
/* Shared */
int counter = 0;
pthread_mutex_t mutex;
pthread_cond_t cond_var;
. . .
void* Thread_work(. . .) {
    . . .
    /* Barrier */
    pthread_mutex_lock(&mutex);
    counter++;
    if (counter == thread_count) {
        counter = 0;
        pthread_cond_broadcast(&cond_var);
    } else {
        while (pthread_cond_wait(&cond_var, &mutex) != 0);
    }
    pthread_mutex_unlock(&mutex);
    . . .
}
```

Implementing a barrier with condition variables

```
/* Shared */
int counter = 0;
pthread_mutex_t mutex;
pthread_cond_t cond_var;
. . .
void* Thread_work(. . .) {
    . . .
    /* Barrier */
    pthread_mutex_lock(&mutex);
    counter++;
    if (counter == thread_count) {
        counter = 0;
        pthread_cond_broadcast(&cond_var);
    } else {
        while (pthread_cond_wait(&cond_var, &mutex) != 0);
    }
    pthread_mutex_unlock(&mutex);
    . . .
}
```

Spurious wake-up



Condition variables

Condition variables in Pthreads have type *pthread_cond_t*. The function

```
int pthread_cond_signal(pthread_cond_t* cond_var_p /* in/out */;
```

will unblock one of the blocked threads, and

```
int pthread_cond_broadcast(pthread_cond_t* cond_var_p /* in/out */);
```

will unblock all of the blocked threads. The functions

```
int pthread_cond_init(  
    pthread_cond_t*      cond_p      /* out */,  
    const pthread_condattr_t* cond_attr_p /* in */);
```

```
int pthread_cond_destroy(pthread_cond_t* cond_p /* in/out */);
```

Create and destroy the condition variable.

Condition variables

The function

```
int pthread_cond_wait(  
    pthread_cond_t*    cond_var_p    /* in/out */,  
    pthread_mutex_t*    mutex_p      /* in/out */);
```

- 1) unlock the mutex;
- 2) block the thread until it is unblocked by another thread's call to *pthread_cond_signal* or *pthread_cond_broadcast*
- 3) when the thread is unblocked, lock the mutex

It is like:

```
pthread_mutex_unlock(&mutex_p);  
wait_on_signal(&cond_var_p);  
pthread_mutex_lock(&mutex_p);
```

Condition variables

pthread_cond_wait(), *pthread_cond_signal()* differ from *sem_wait()* e *sem_post()* because:

- *pthread_cond_wait()* **ALWAYS** blocks process execution
- *pthread_cond_signal()* can be ignored. If there are no threads in the cond_wait, the signal is lost
- *sem_t* can get values ≥ 0 .

Exercises

Programming assignment

4.1. Write a Pthreads program that implements the histogram program in Chapter 2.

- 4.2. Suppose we toss darts randomly at a square dartboard, whose bullseye is at the origin, and whose sides are 2 feet in length. Suppose also that there's a circle inscribed in the square dartboard. The radius of the circle is 1 foot, and its area is square feet. If the points that are hit by the darts are uniformly distributed (and we always hit the square), then the number of darts that hit inside the circle should approximately satisfy the equation:

$$\frac{\text{number in circle}}{\text{total number of tosses}} = \frac{\pi}{4},$$

since the ratio of the area of the circle to the area of the square is $\pi/4$.

Programming assignment

We can use this formula to estimate the value of π with a random number generator:

```
number_in_circle = 0;
for (toss = 0; toss < number_of_tosses; toss++) {
    x = random double between -1 and 1;
    y = random double between -1 and 1;
    distance_squared = x*x + y*y;
    if (distance_squared <= 1) number_in_circle++;
}
pi_estimate = 4*number_in_circle/((double) number_of_tosses);
```

Write a Pthreads program that uses this method to estimate π .

The main thread should read in the total number of tosses and print the estimate value

Programming assignment

4.3 Write a Pthreads program that implements the trapezoidal rule. Use a shared variable for the sum of all the threads' computations. Use busy-waiting and mutexes to enforce mutual exclusion in the critical section. What advantages and disadvantages do you see with each approach?

4.6. Modify the mutex version of the π calculation program so that it uses a semaphore instead of a mutex.