HICREST

UNITRENTO

Introduction to Parallel Computing

a.y. 2025 – 2026

Dr. Brahmaiah Gandham

Lecture 6

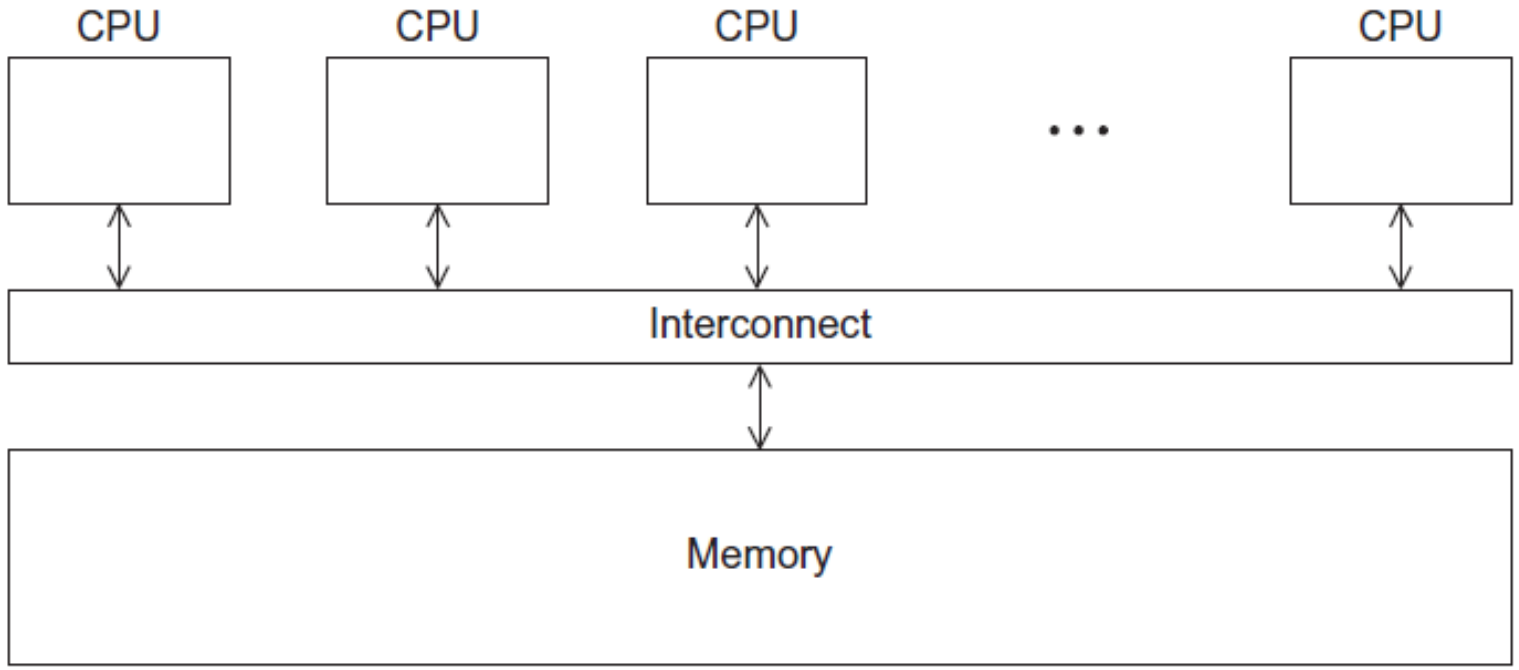
Shared Memory Programming with Pthreads

Overview

UNITRENTO

- **Problems programming shared memory systems.**
- **Controlling access to a critical section.**
- **Thread synchronization.**
- **Programming with POSIX threads.**
- **Mutexes.**

A Shared Memory System

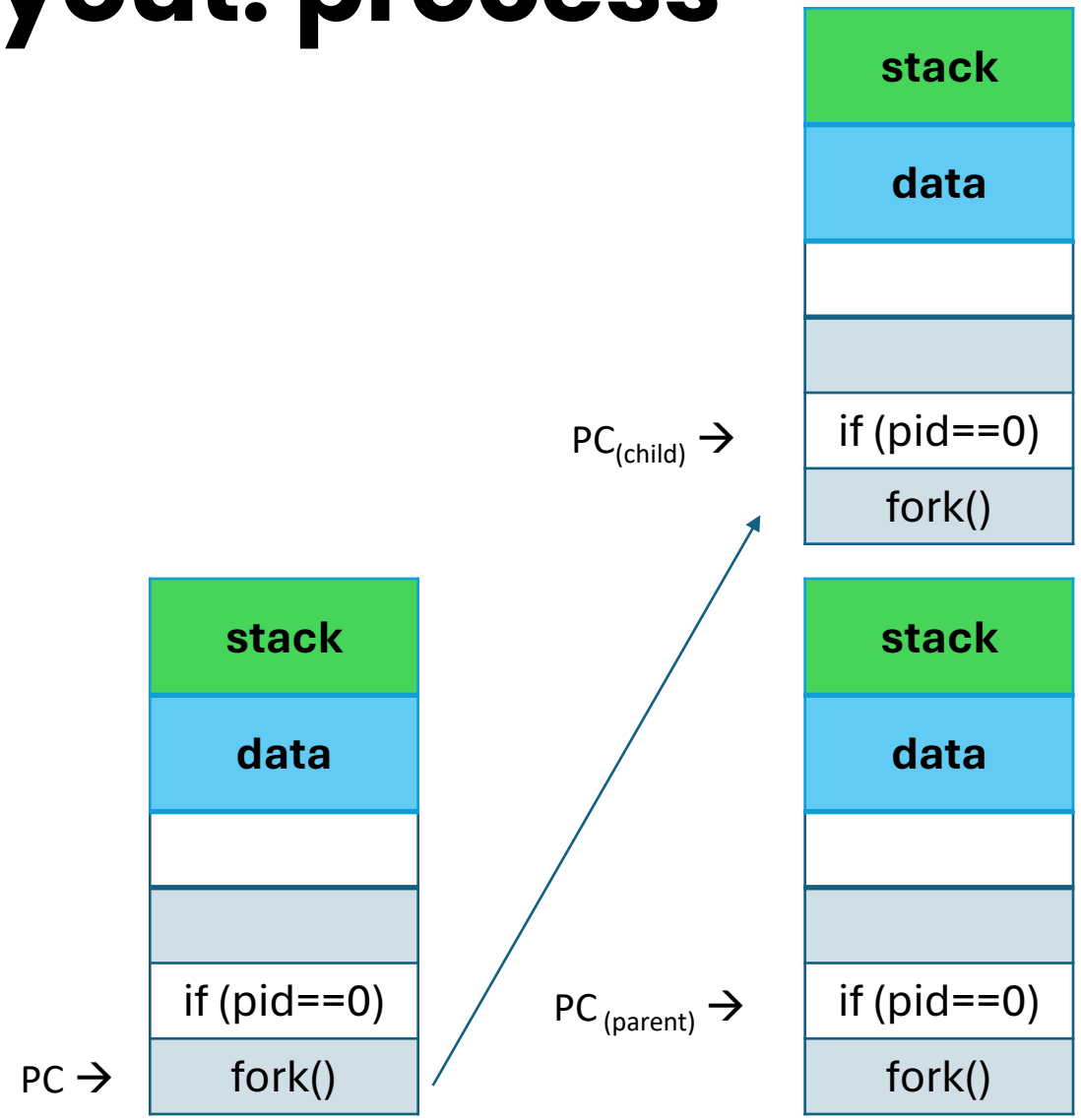


Processes and Threads

UNITRENTO

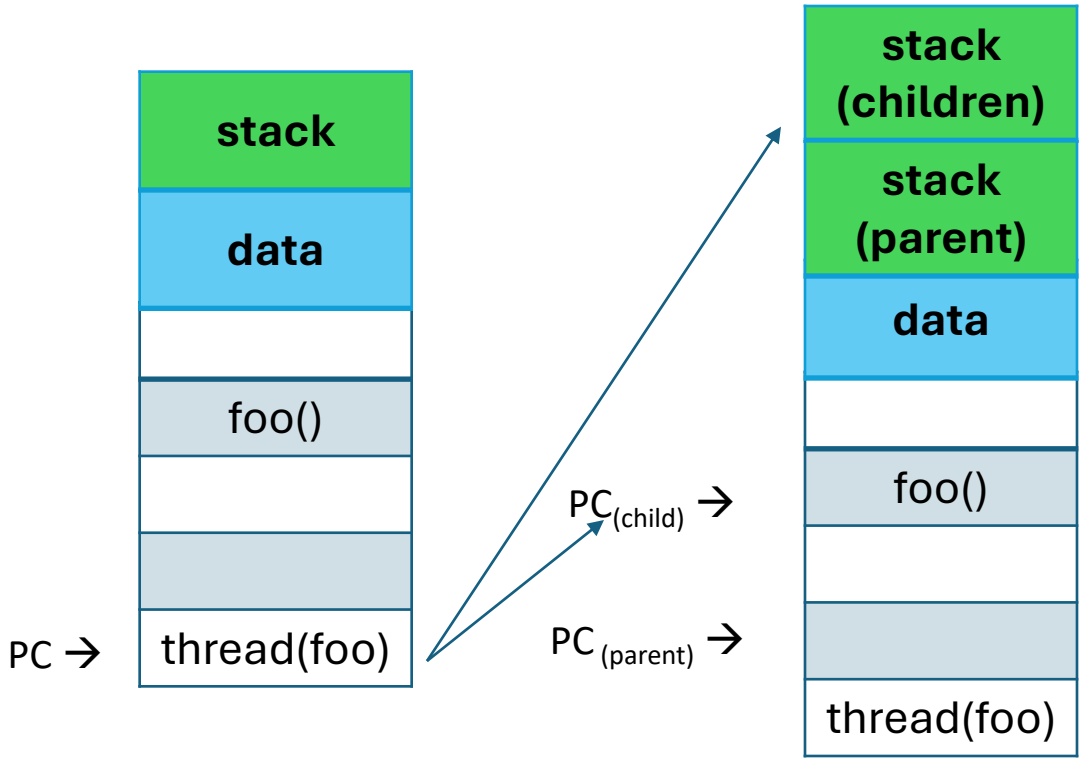
- A process is an instance of a running (or suspended) program.
- Threads are analogous to a “light-weight” process.
- In a shared memory program a single process may have multiple threads of control.

Memory layout: process



N.B.: modern OSes usually use COW (copy-on-write) policy to optimize memory allocation

Memory layout: thread

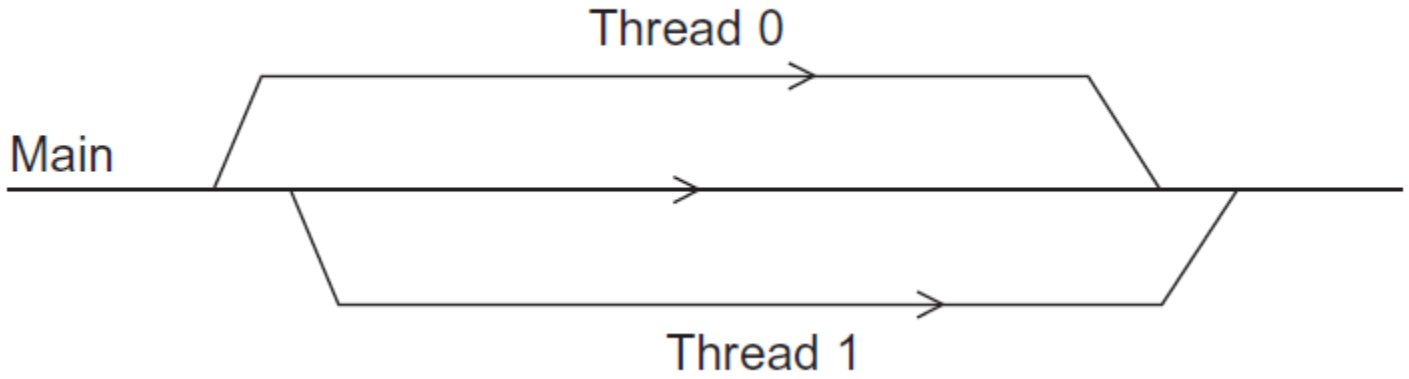


POSIX® Threads

UNITRENTO

- Also known as Pthreads.
- A standard for Unix-like operating systems.
- A library that can be linked with C programs.
- Specifies an application programming interface (API) for multi-threaded programming.
- The Pthreads API is only available on POSIX systems — Linux, MacOS X, Solaris, HPUX, …

Running the Threads



Main thread forks and joins two threads.

Starting the Threads

- Processes in MPI are started by mpirun/mpiexec
- In Pthreads the threads are started directly by the program executable.

```
#include <pthread.h>
int pthread_create (pthread_t*  thread_p /* out */ ,
                    const pthread_attr_t*  attr_p /* in */ ,
                    void*  (*start_routine ) ( void* ) /* in */ ,
                    void*  arg_p /* in */ ) ;
```

Starting the Threads

UNITRENTO

```
#include <pthread.h>
int pthread_create (pthread_t*  thread_p /* out */ ,
                    const pthread_attr_t*  attr_p /* in */ ,
                    void*  (*start_routine ) ( void* ) /* in */ ,
                    void*  arg_p /* in */ ) ;
```

- It is an handle (one per thread). I.e., an “object” representing a thread
- Must be allocated before the call
- Opaque
- The actual data that they store is system-specific.
- Their data members aren’t directly accessible to user code.
- However, the Pthreads standard guarantees that a pthread_t object does store enough information to uniquely identify the thread with which it’s associated.

Starting the Threads

```
#include <pthread.h>
int pthread_create (pthread_t*  thread_p /* out */ ,
                    const pthread_attr_t*  attr_p /* in */ ,
                    void*  (*start_routine ) ( void* ) /* in */ ,
                    void*  arg_p /* in */ ) ;
```

- We won't use it, just set it to NULL

Starting the Threads

```
#include <pthread.h>
int pthread_create (pthread_t*  thread_p /* out */ ,
                    const pthread_attr_t*  attr_p /* in */ ,
                    void*  (*start_routine ) ( void* ) /* in */ ,
                    void*  arg_p /* in */ ) ;
```

- The function the thread is going to execute
- It is a pointer to a function (the address of a piece of memory containing code rather than data)
- In this case, we need a function returning a void* and taking as argument a void*

Function Pointer in C

- In C, like normal data pointers (int *, char *, etc), we can have pointers to functions.

- A function's name can be used to get functions' address.

```
void func(int a)
{
    printf("a=%d\n", a);
}

void main()
{
   void(*func_ptr)(int) = func;
   *func_ptr(10);
   printf("addr of func is: %p\n",func_ptr);
}
```

Function started by pthread_create

UNITRENTO

- Prototype:
  void* thread_function (void* args_p) ;

- Void* can be cast to any pointer type in C.

- So args_p can point to a list containing one or more values needed by thread_function.

- Similarly, the return value of thread_function can point to a list of one or more values.

Starting the Threads

```
#include <pthread.h>
int pthread_create (pthread_t*  thread_p /* out */ ,
                    const pthread_attr_t*  attr_p /* in */ ,
                    void*  (*start_routine ) ( void* ) /* in */ ,
                    void*  arg_p /* in */ ) ;
```

- Pointer to the data that will be passed to start_routine

```
void* func(void* a)
{
    int* px = (int*) a;
    int x = *px;
    printf("x=%d\n", x);
}
void main()
{
    …
    int x = 2;
    pthread_create(…,…,func, (void*) &x);
    …
}
```

Recap

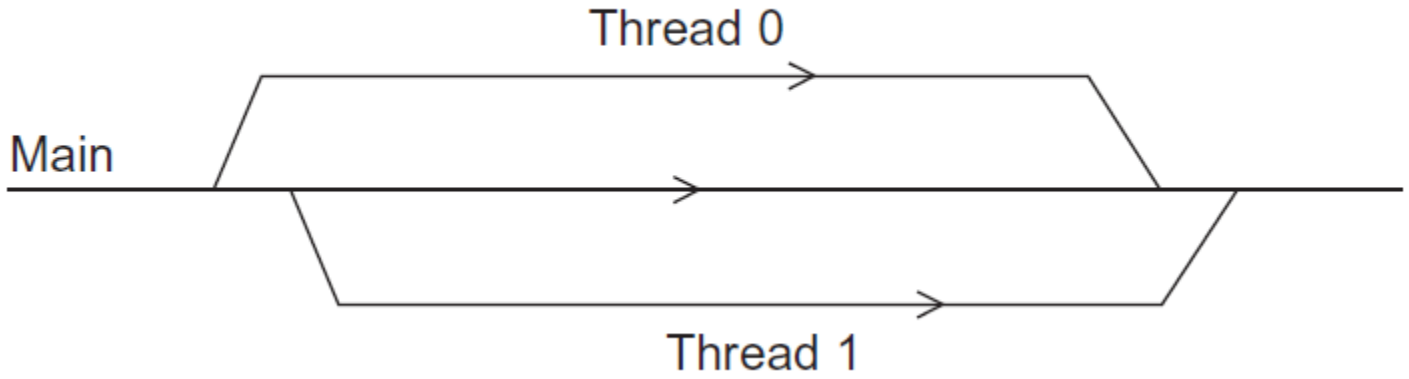
```
#include <pthread.h>
int pthread_create (pthread_t*  thread_p /* out */ ,
                    const pthread_attr_t*  attr_p /* in */ ,
                    void*  (*start_routine ) ( void* ) /* in */ ,
                    void*  arg_p /* in */ ) ;
```

Global variables

- Can introduce subtle and confusing bugs!
- Limit use of global variables to situations in which they're really needed.
  - Shared variables.

```
int g; // Visible both from func and main

void* func(void* a)
{
    int* px = (int*) a;
    int x = *px;
    printf("x=%d\n", x);
}

void main()
{
    …
    int x = 2;
    pthread_create(…,…,func, (void*) &x);
    …
}
```

Running the Threads



Main thread forks and joins two threads.

Waiting for the Threads to finish

UNITRENTO

- We call the function pthread_join once for each thread.
- A single call to pthread_join will wait for the thread associated with the pthread_t object to complete.

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

- value_ptr (if not NULL) has return value of the thread function

Correct way to wait for thread completion

UNITRENTO

```
for(int i = 0; i < num_threads; i++){
    pthread_create(…);
}

for(int i = 0; i < num_threads; i++){
    pthread_join(…);
}
```

Correct

vs.

```
for(int i = 0; i < num_threads; i++){
    pthread_create(…);
    pthread_join(…);
}
```

Wrong
(everything would be executed sequentially)

Thread identification

- pthread_self provides the thread ID of the calling thread

```
pthread_t pthread_self(void);
```

- pthread_equal compares thread IDs

```
int pthread_equal(pthread_t t1, pthread_t t2);
```

Example: Hello World! (1)

```
void *Hello(void* rank) {
   long my_rank = (long) rank;  /* Use long in case of 64-bit system */

   printf("Hello from thread %ld of %d\n", my_rank, thread_count);

   return NULL;
}  /* Hello */
```

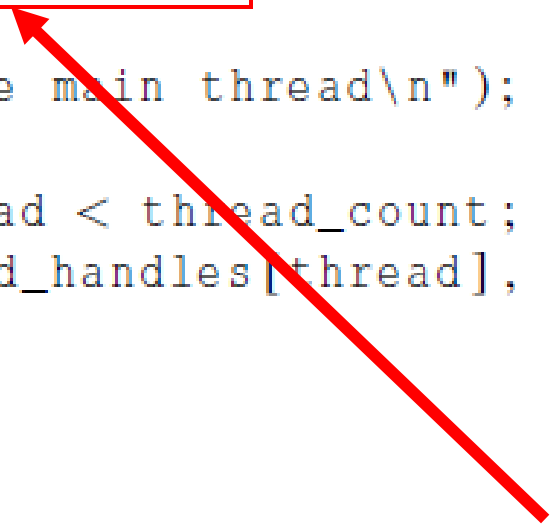
Hello World! (2)

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

/* Global variable:  accessible to all threads */
int thread_count;

void *Hello(void* rank);  /* Thread function */

int main(int argc, char* argv[]) {
   long       thread;  /* Use long in case of a 64-bit system */
   pthread_t* thread_handles;

   /* Get number of threads from command line */
   thread_count = strtol(argv[1], NULL, 10);

   thread_handles = malloc (thread_count*sizeof(pthread_t));
```

declares the various Pthreads functions, constants, types, etc.

Hello World! (3)

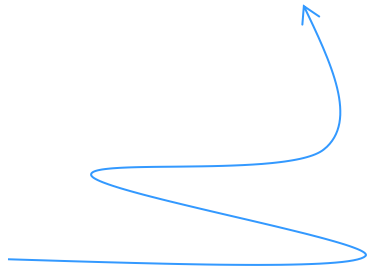
```
   for (thread = 0; thread < thread_count; thread++)
      pthread_create(&thread_handles[thread], NULL,
          Hello, (void*) thread);

   printf("Hello from the main thread\n");

   for (thread = 0; thread < thread_count; thread++)
      pthread_join(thread_handles[thread], NULL);

   free(thread_handles);
   return 0;
}  /* main */
```



Dangerous. What if sizeof(void*) < sizeof(long) ?

Compiling a Pthread program

```
gcc -g -Wall -o pth_hello pth_hello . c -lpthread
```



Running a Pthreads program

UNITRENTO

```
. / pth_hello   <number of threads>
```

```
. / pth_hello  1
```

Hello from the main thread
Hello from thread 0 of 1

```
. / pth_hello 4
```

Hello from the main thread
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4

Threads execute in parallel, the order of the prints is not guaranteed

More complex thread args

```
struct thread_args {
    long my_rank;
    char *task_name;
};

void *Hello(void *args) {
   struct thread_args* t_args
      = (struct thread_args *) args;
   printf("Thread %ld is working on task '%s'\n",
      t_args->my_rank, t_args->task_name);
   return NULL;
}
```

```
struct thread_args *t_args
   = malloc(sizeof(struct thread_args));

t_args->my_rank = thread;
t_args->task_name = "Hello task";

pthread_create(&thread_handles[thread],
   NULL,
   Hello,
   (void *) t_args);
```

main

Caveats

To keep the code simpler, I did not explicitly check for erros

- What if the program is called without command line arguments?
- What if one of the pthread functions fail?

Matrix-Vector Multiplication in pthreads

UNITRENTO

$$
\begin{bmatrix}
a_{00} & a_{01} & \cdots & a_{0,n-1} \\
a_{10} & a_{11} & \cdots & a_{1,n-1} \\
\vdots & \vdots & & \vdots \\
a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\
\vdots & \vdots & & \vdots \\
a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1}
\end{bmatrix}
\begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix}
=
\begin{bmatrix}
y_0 \\ y_1 \\ \vdots \\ y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} \\ \vdots \\ y_{m-1}
\end{bmatrix}
$$

Serial pseudo-code

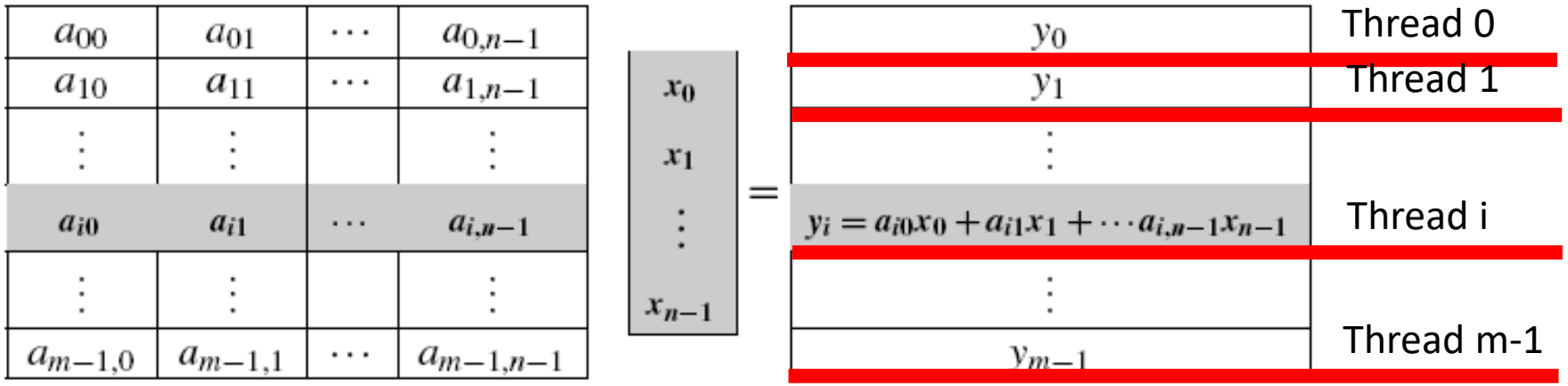
```
/* For each row of A */
for (i = 0; i < m; i++) {
    y[i] = 0.0;
    /* For each element of the row and each element of x */
    for (j = 0; j < n; j++)
        y[i] += A[i][j]* x[j];
}
```

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

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 $q \times \frac{m}{t}$ $(q+1) \times \frac{m}{t} - 1$

- 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?

UNITRENTO

- 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
```

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

	n			
	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 sum + 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`

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

UNITRENTO

- 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 */
```

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

UNITRENTO

- 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

UNITRENTO

- 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.

Key Takeaways

UNITRENTO

- A thread in shared-memory programming is often lighter-weight than a full-fledged process.
- In Pthreads programs, all the threads have access to global variables, while local variables usually are private to the thread running the function.
- When indeterminacy results from multiple threads attempting to access a shared resource such as a shared variable or a shared file, at least one of the accesses is an update, and the accesses can result in an error, we have a race condition.
- A critical section is a block of code that updates a shared resource that can only be updated by one thread at a time.
- So the execution of code in a critical section should, effectively, be executed as serial code.

Key Takeaways

UNITRENTO

- Busy-waiting can be used to avoid conflicting access to critical sections with a flag variable and a while-loop with an empty body.
- It can be very wasteful of CPU cycles.
- It can also be unreliable if compiler optimization is turned on.
- A mutex can be used to avoid conflicting access to critical sections as well.
- Think of it as a lock on a critical section, since mutexes arrange for mutually exclusive access to a critical section.