Programmazione di Sistemi ~~Embedded e~~ Multicore

Teacher: Daniele De Sensi

Recap

Recap

- MPI assumes a distributed memory model
- Cooperation happens through message exchange
- Point-to-point (send/recv) and collectives
- Custom (user-defined) datatypes
- Much more beyond that:
    - communicator creation
    - topologies
    - one-sided operations
    - non-blocking collectives (MPI_Ibcast, etc…)
    - etc…

Con MPI → sistemi a mem. distribuita, dove i processi sono collegati da qualche mezzo
Ogni nodo contiene una mem. DRAM e una CPU multicore.

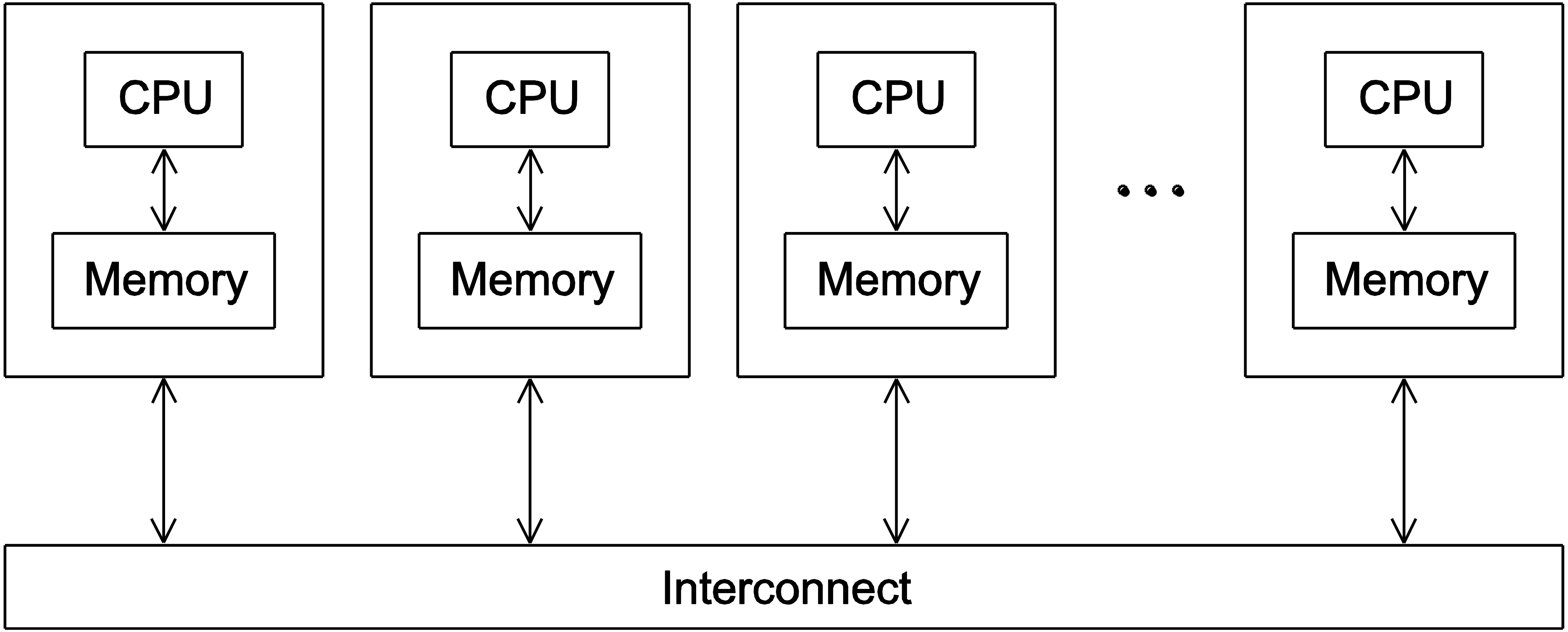
In questa CPU, per ogni core sono presenti, assieme all'ALU, anche una cache L1 priv.
La CPU ha anche assegnata una cache L2 comune.

... ↪ Di norma, si usa creare un processo MPI × ogni nodo, e poi internamente sfruttare librerie come PThreads e CUDA × gestire rispettivamente i core/threads della CPU e i core della GPU.

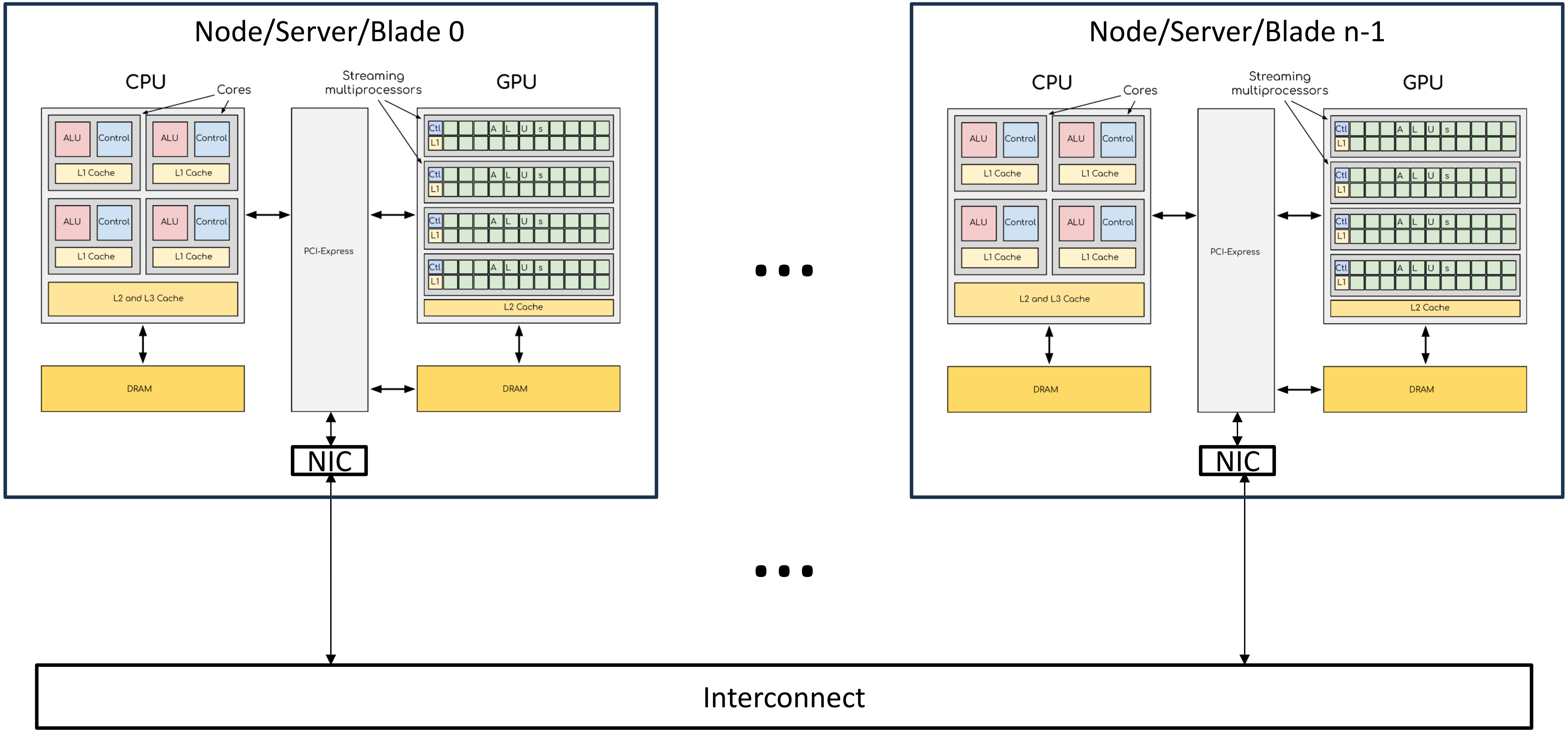
↝ Il vantaggio di usare dei threads al posto dei processi separati è che i threads condividono la mem DRAM della CPU.

Questions?

Distributed Memory Systems

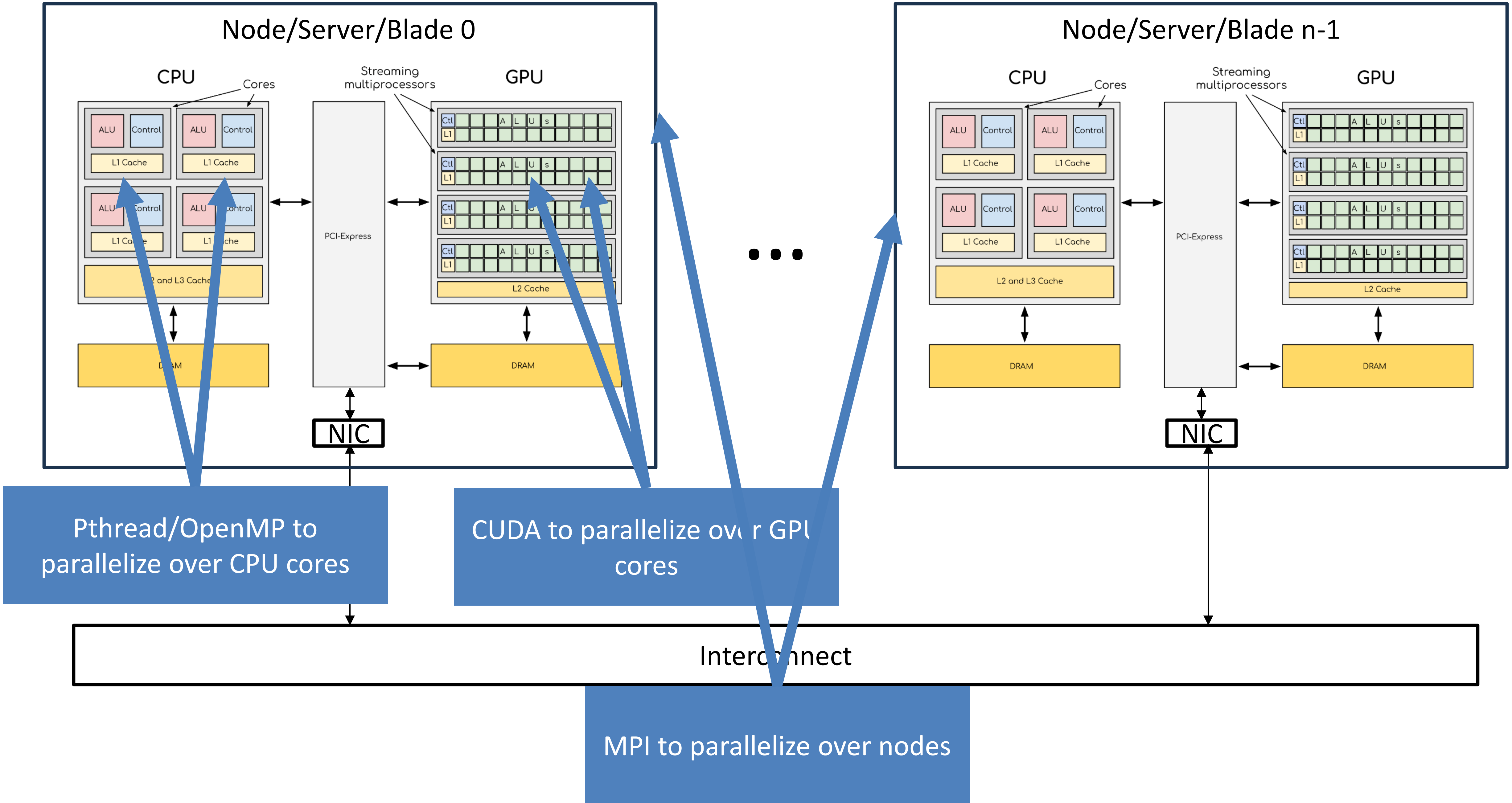


Distributed Memory Systems



Note: We can have more than one GPU per node (today, up to 4)
Note 2: We can have more than one NIC per node (today, one per GPU/one every 2 GPUs)

Distributed Memory Systems



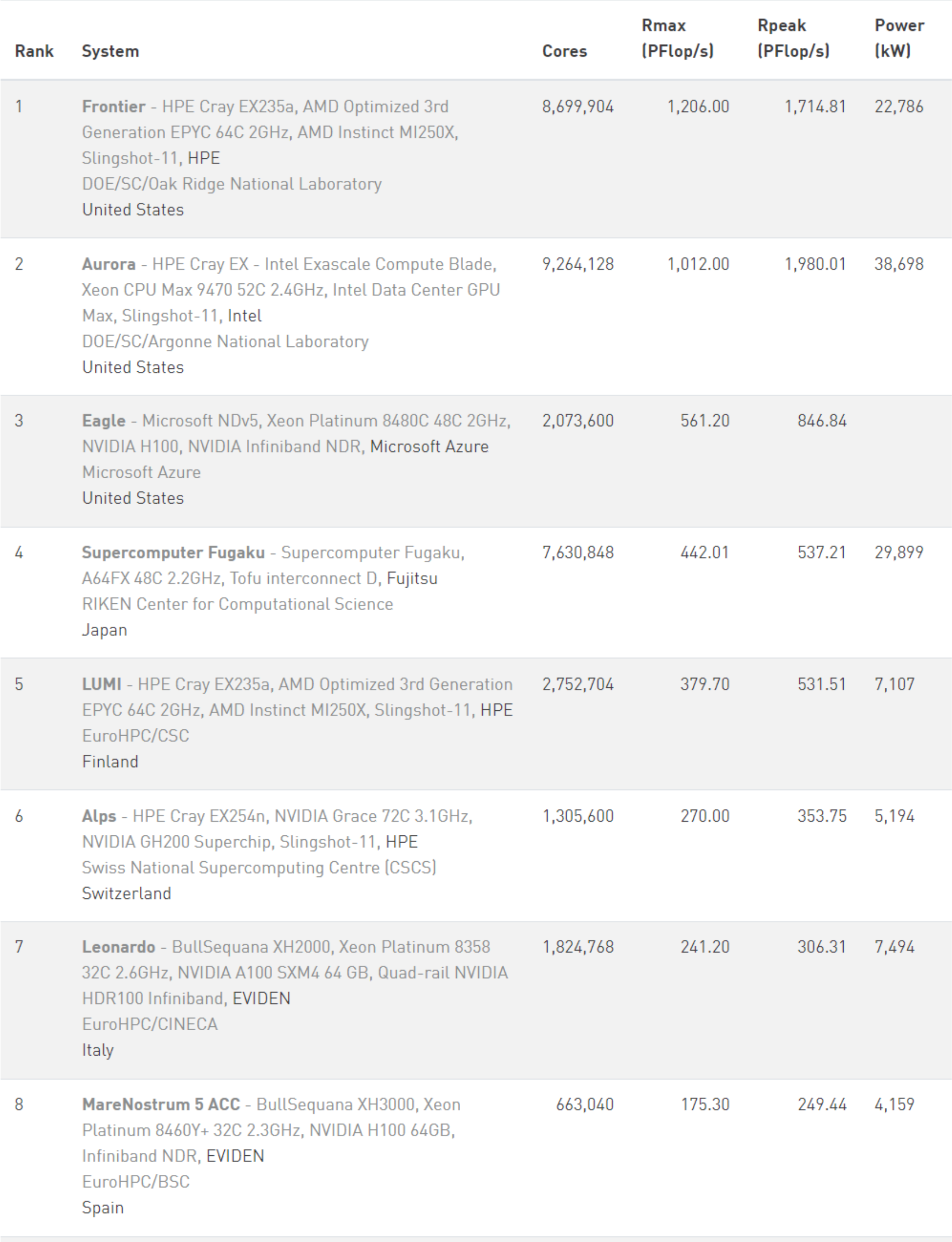
Pthread/OpenMP to parallelize over CPU cores

CUDA to parallelize over GPU cores

MPI to parallelize over nodes

BSC
Barcelona
Supercomputing
Center

Top500 List (top500.org)



Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,699,904	1,206.00	1,714.81	22,786
2	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	9,264,128	1,012.00	1,980.01	38,698
3	**Eagle** - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, Microsoft Azure Microsoft Azure United States	2,073,600	561.20	846.84	
4	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
5	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,752,704	379.70	531.51	7,107
6	**Alps** - HPE Cray EX254n, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11, HPE Swiss National Supercomputing Centre (CSCS) Switzerland	1,305,600	270.00	353.75	5,194
7	**Leonardo** - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband, EVIDEN EuroHPC/CINECA Italy	1,824,768	241.20	306.31	7,494
8	**MareNostrum 5 ACC** - BullSequana XH3000, Xeon Platinum 8460Y+ 32C 2.3GHz, NVIDIA H100 64GB, Infiniband NDR, EVIDEN EuroHPC/BSC Spain	663,040	175.30	249.44	4,159

Questions?

Chapter 4

Shared Memory Programming with Pthreads

Roadmap

- Problems programming shared memory systems.
- Controlling access to a critical section.
- Thread synchronization.
- Programming with POSIX threads.
- Mutexes.
- Producer-consumer synchronization and semaphores.
- Barriers and condition variables.
- Read-write locks.
- Thread safety.

Processes and Threads

- 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.

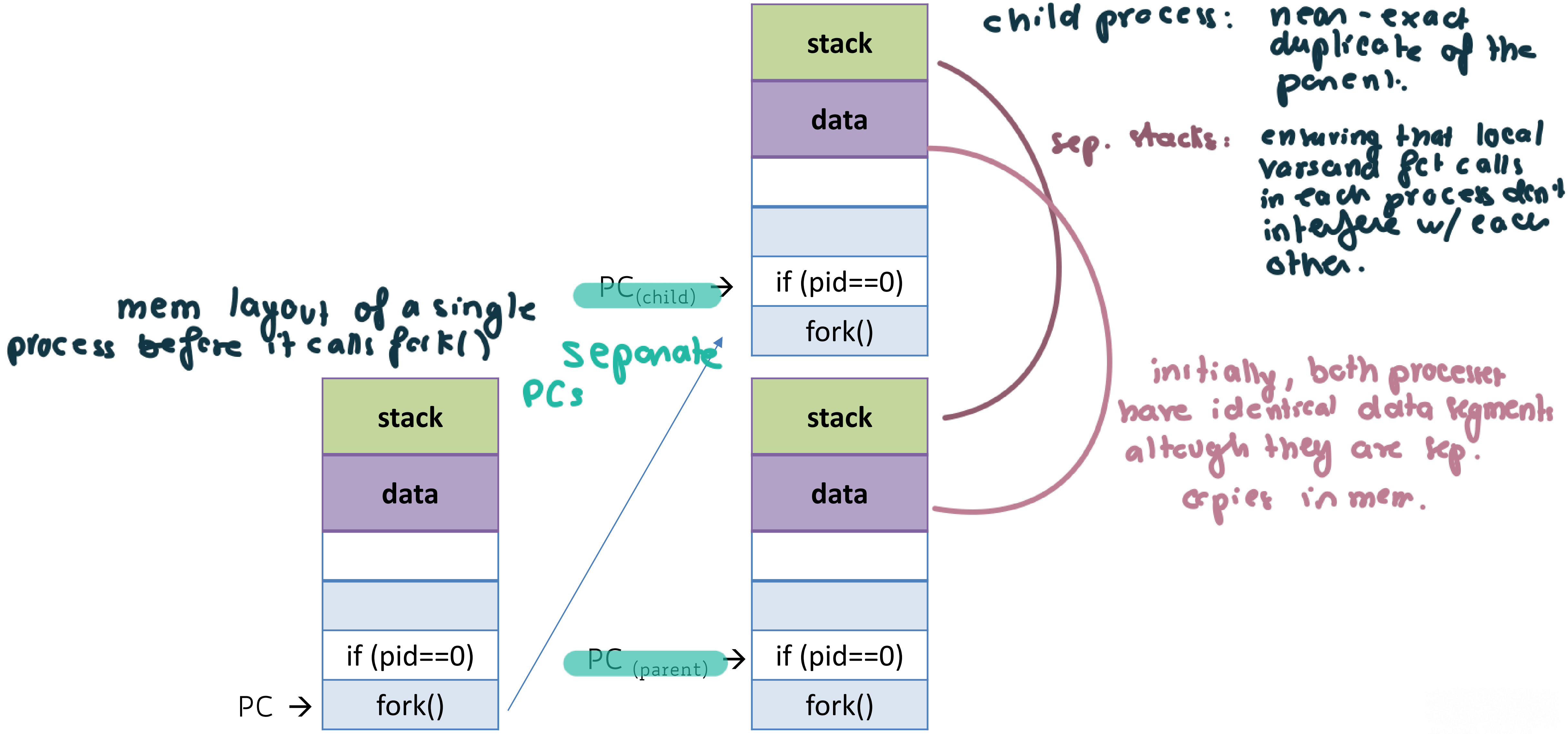
Processo: istanza di computazione di un programma che puo' essere in esecuzione, in stato di attesa o sospeso.

Thread: istanza di computazione + piccola e indipendente che puo' essere eseguita su un computer.

↳ defines a single seq. exec. stream w/in a process → consists of: pc - stack - set of regs and thread ID

thread : bound to a spec. process

Memory layout: process



mem layout of a single process before it calls fork()

separate PCs

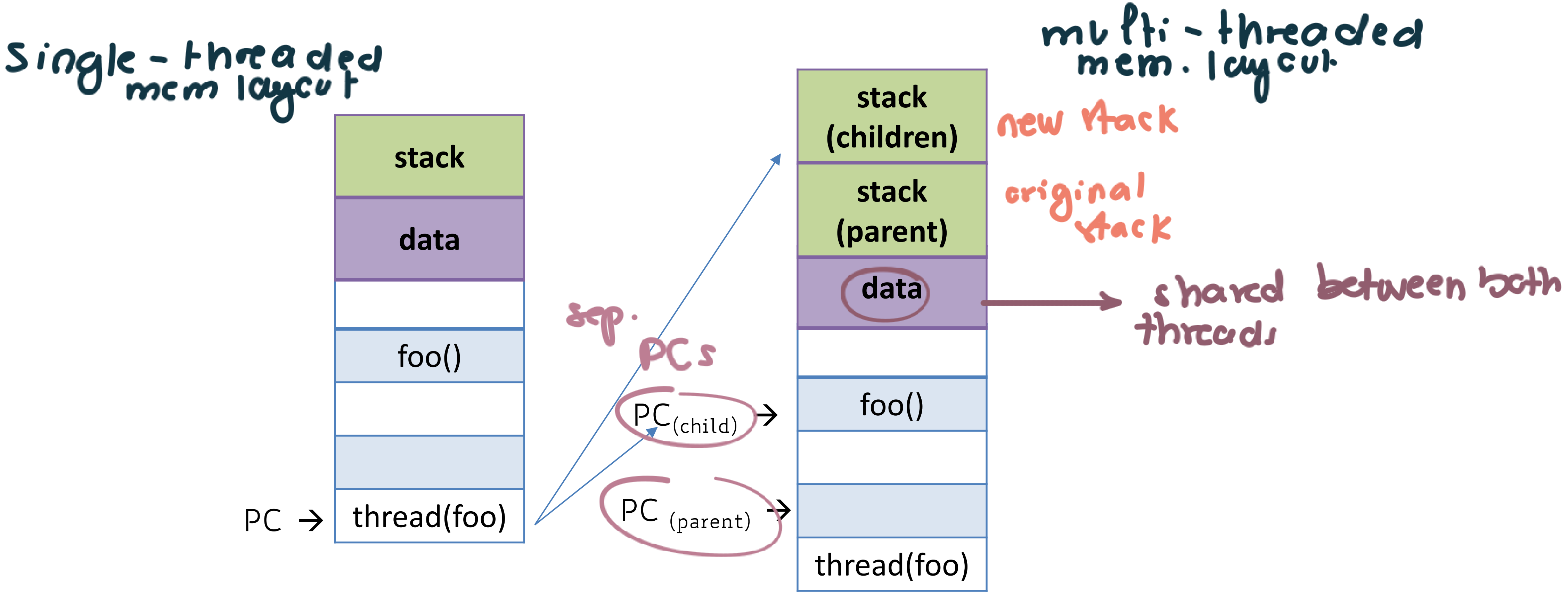
child process: near-exact duplicate of the parent.

sep. stacks: ensuring that local vars and fct calls in each process don't interfere w/ each other.

initially, both processes have identical data segments altough they are sep. copies in mem.

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

Memory layout: thread



↝ Data seg: shared, allowing for inter-thread comm via global vars

↝ stacks: sep., to track their individual fct calls and local vars (no interfering w/ each other's exec)

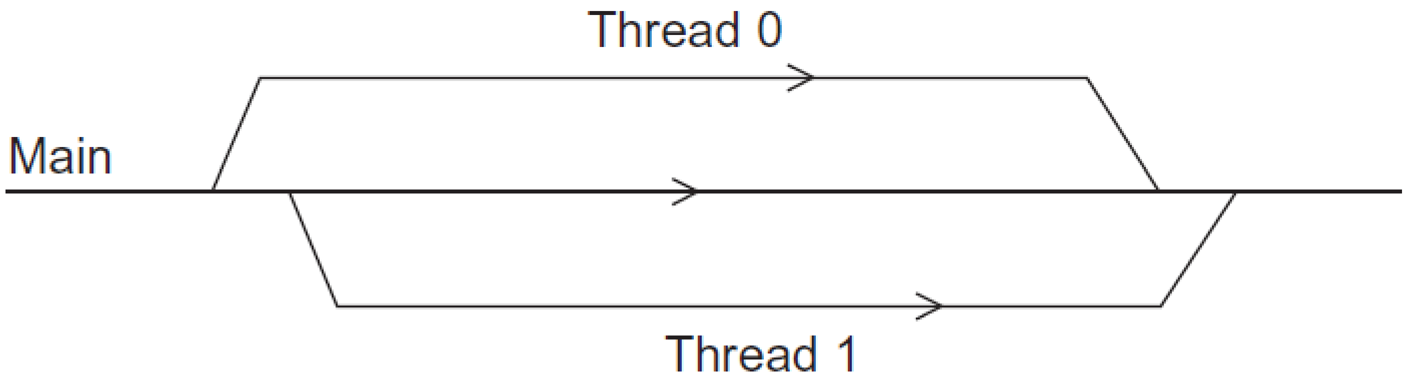
POSIX® Threads

linking allows a C program to access fcts and resources from the library. which are compiled and ready to use.

- 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, ...

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

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.

↳ i thread vengono creati automaticamente una volta lanciato il programma.

-create a thread:

e' un handle che rappresenta lo stato del thread. e' OPACO quindi non va modificato, e il pointer va sempre alloc. prima della chiamata di fct. Pthread garantisce che tale handle contenga abbastanza info x id. il thread

```
#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 that var

no null

pass NULL, the thread will be created w/ default attrs

puntatore a fct, che def da dove il thread inizia ad exg. il codice.

of type void pointer

↳ puntatore ai par. che la funzione start_routine richiede

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 */ ) ;
```

- 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 */ ) ;
```

assigning each thread a unique int rank

- 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***

The fct that's started by pthread_create should look like:

```
void* thread_function(void* args_p);
```

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

- 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

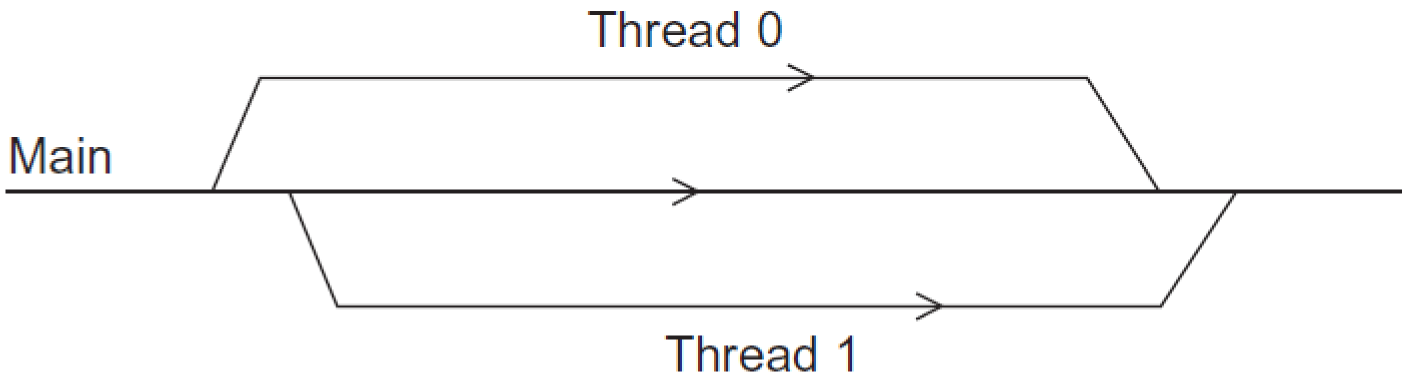
- Can introduce subtle and confusing bugs!
- ~~Limit use of global variables to situations in which they’re really needed.~~
  - Shared variables.

```
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);
    …
}
```

Questions?

Running the Threads

If we think of the main thread as a single line in our diagram, then when we call pthread_create, we can create a branch or fork off the main thread.



Main thread forks and joins two threads.

Waiting for the Threads to finish

- 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

↳ pthread_join also frees up the resources associated w/ the thread

→ ! zombie threads
pthread_detach

Correct way to wait for thread completion

```
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

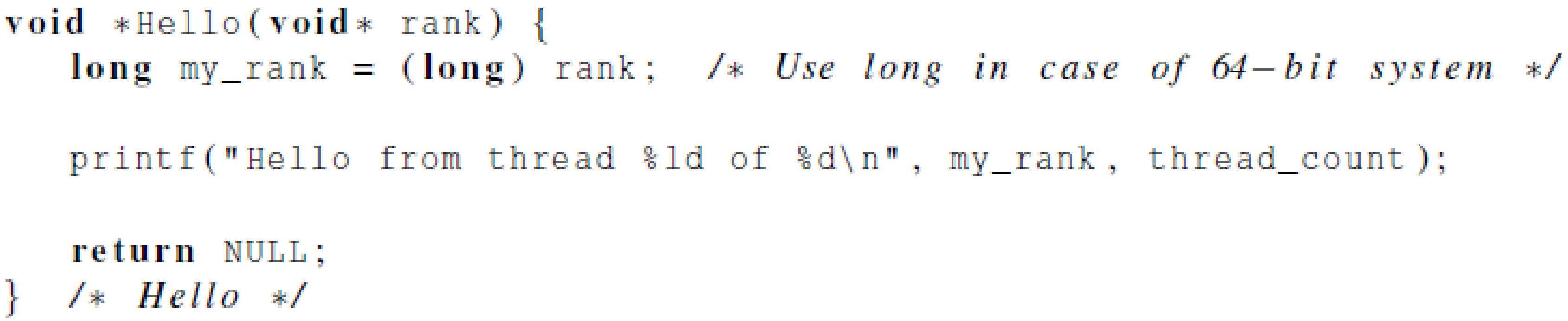
```
pthread_t pthread_self(void);
```

pthread_equal compares thread IDs

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

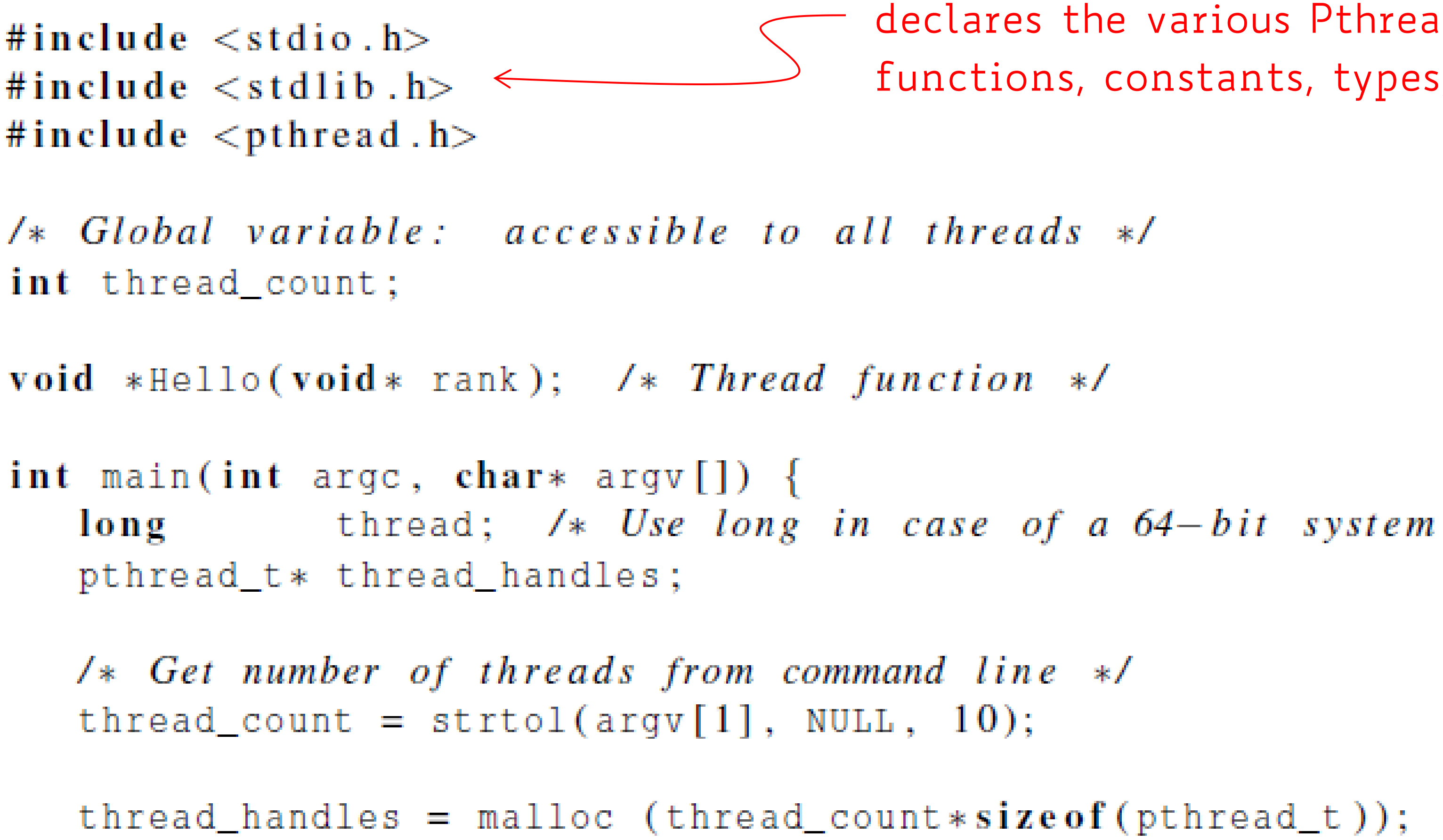
Example – Hello World

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 */
```

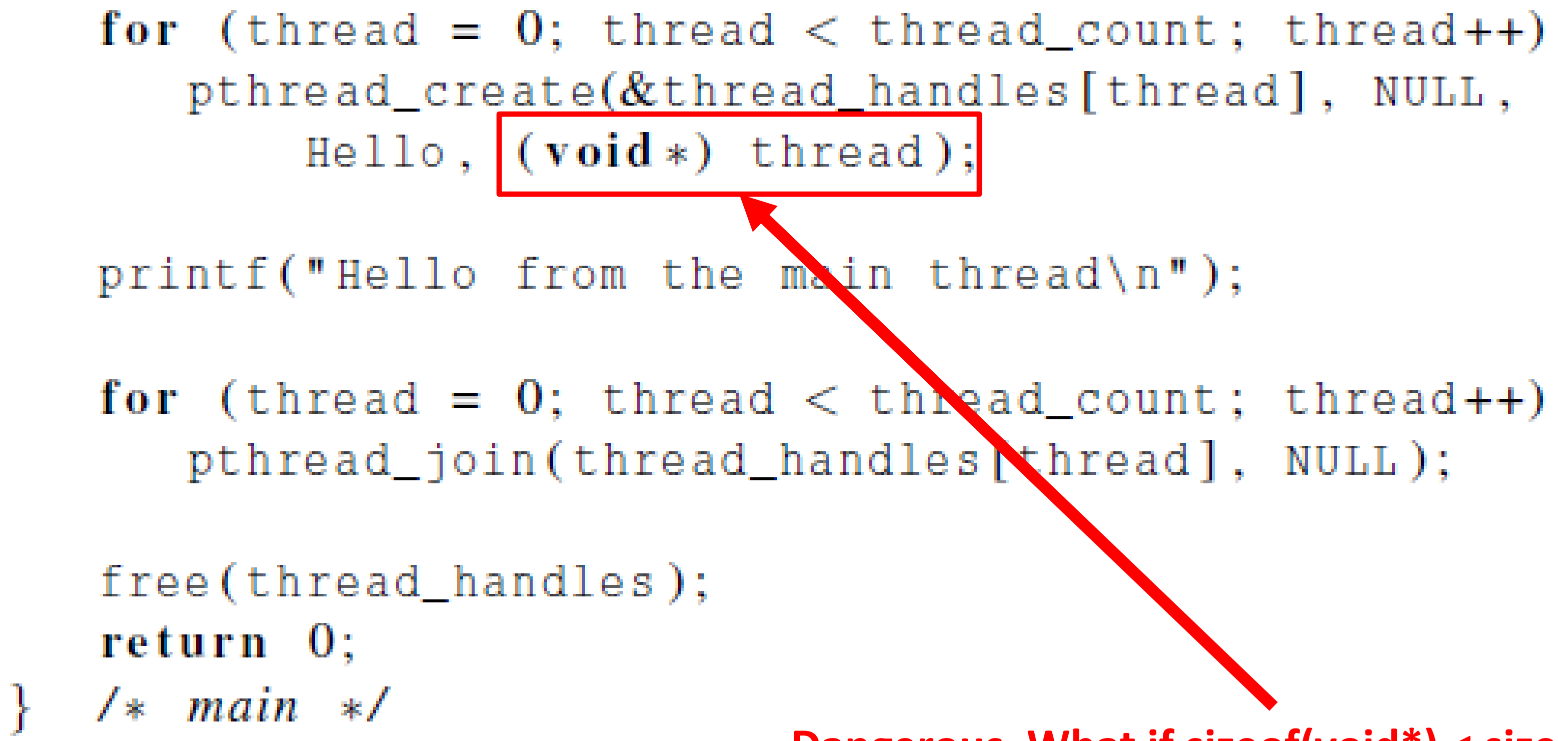
Hello World! (2)



```
#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
```

link in the Pthreads library

Running a Pthreads program

```
./  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

```c
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;
}
```

```c
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);
```

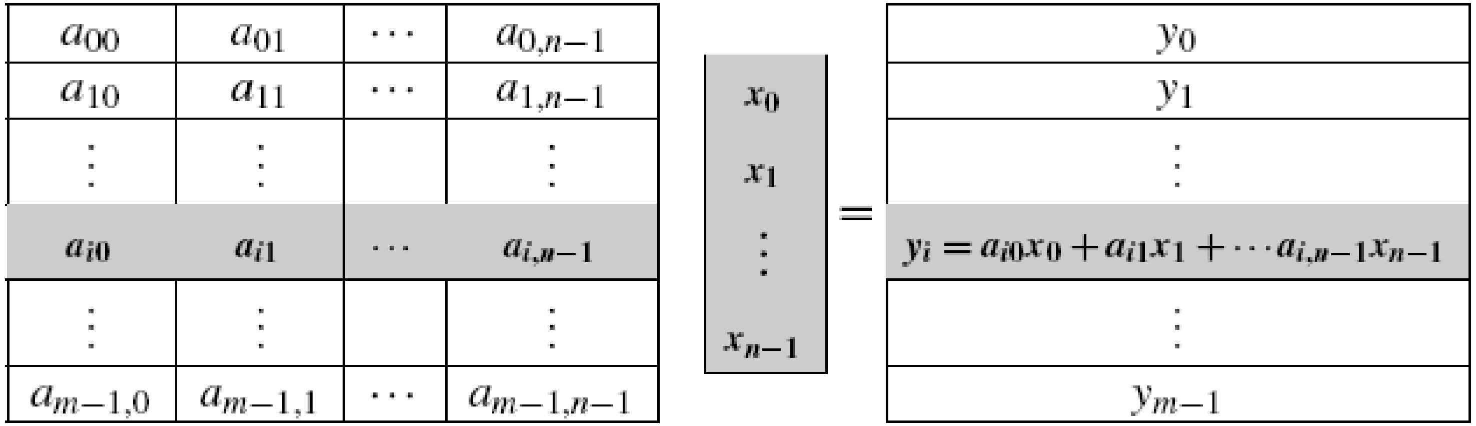
main

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?

Questions?



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

MATRIX-VECTOR MULTIPLICATION IN PTHREADS

Serial pseudo-code

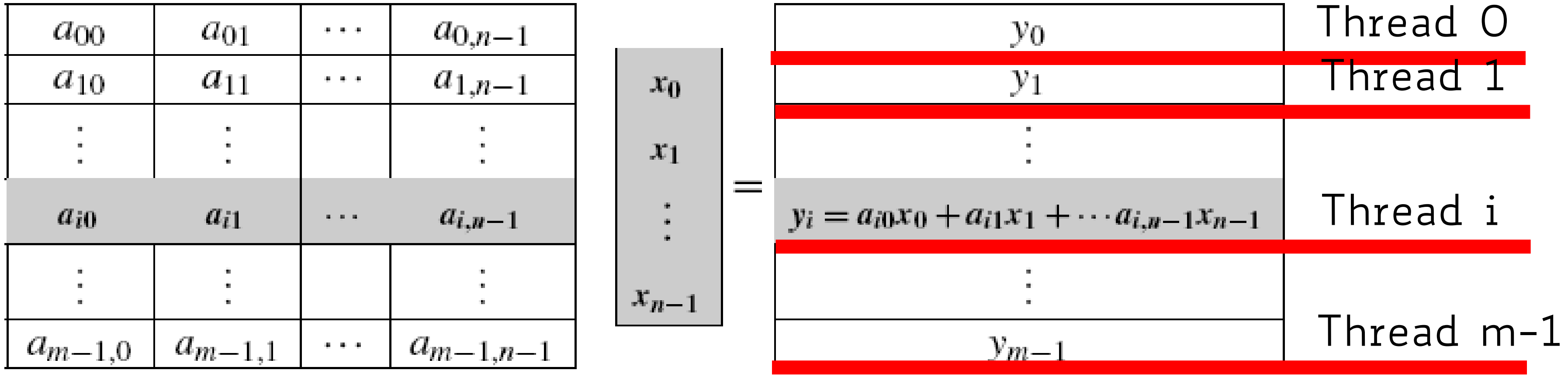
```
/* For each row of A */
for (i = 0; i < m; i++) {
   y[i] = 0.0;
   /* For each element of the row and each element of x */
   for (j = 0; j < n; j++)
      y[i] += A[i][j]* x[j];
}
```

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

How to do this in parallel?

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

Intuition

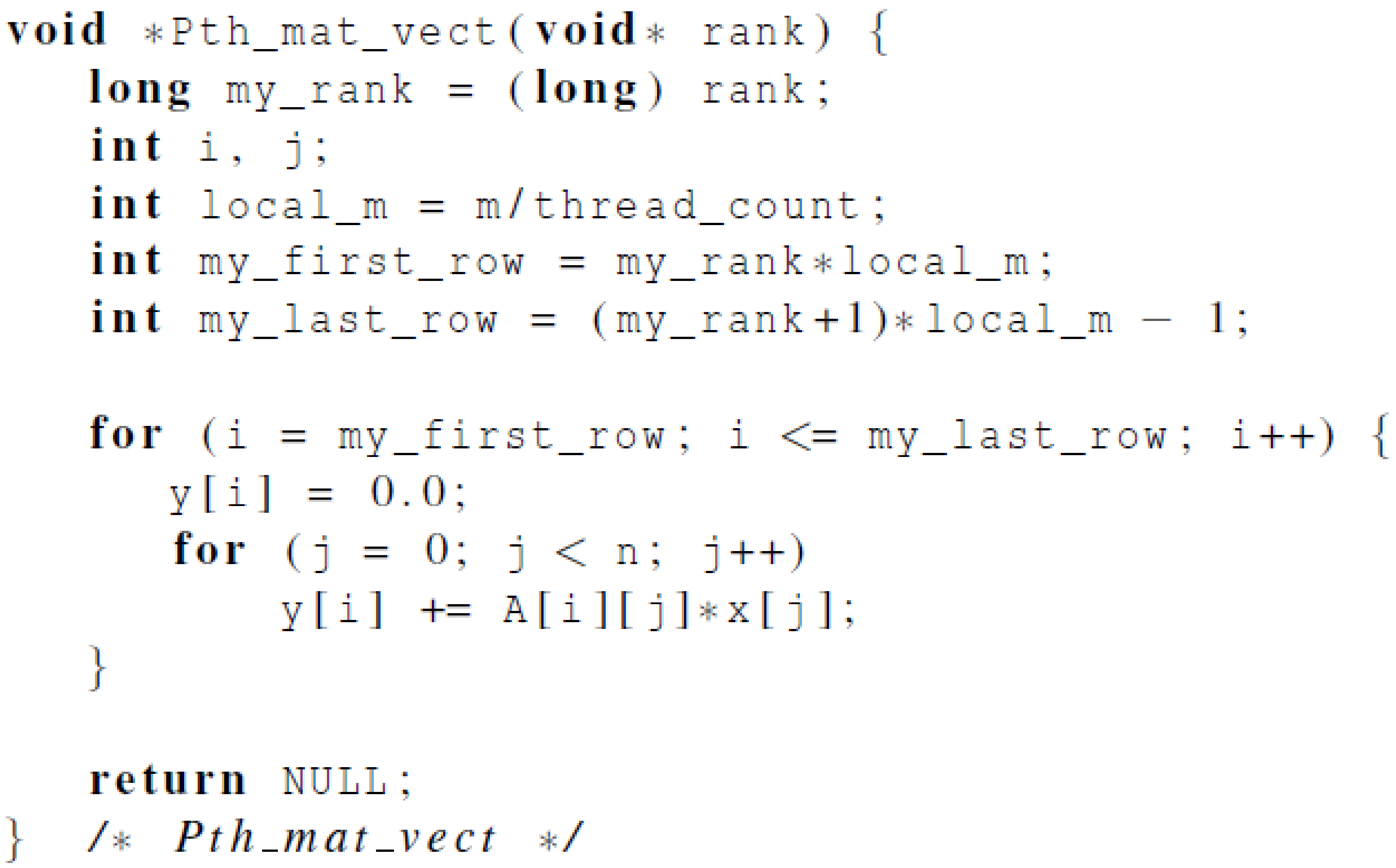

a_{00} a_{01} $\cdots$ $a_{0,n-1}$
a_{10} a_{11} $\cdots$ $a_{1,n-1}$
a_{i0} a_{i1} $\cdots$ $a_{i,n-1}$
$a_{m-1,0}$ $a_{m-1,1}$ $\cdots$ $a_{m-1,n-1}$
x_0
x_1
x_{n-1}
$=$
y_0
y_1
$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1}$
y_{m-1}
Thread 0
Thread 1
Thread i
Thread m-1

- 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

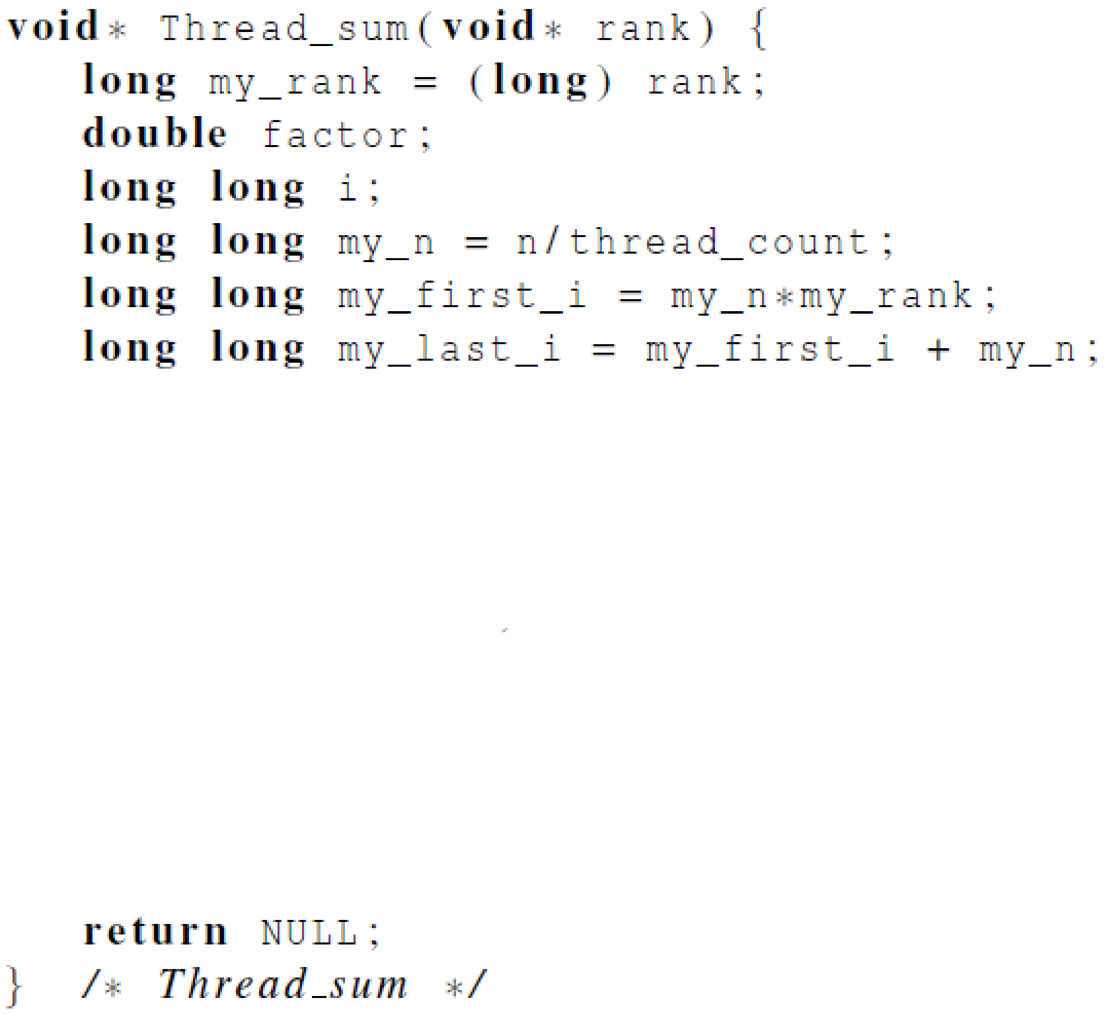
Estimating π

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

```
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;
```

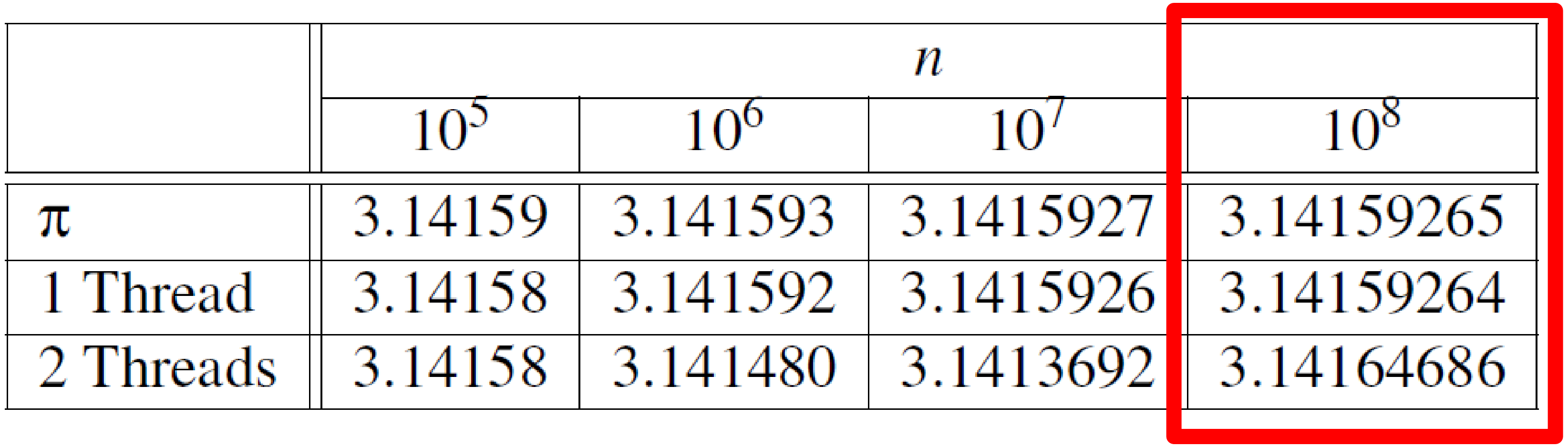
Parallel algorithm: each thread computes a subset of that series

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 */
```

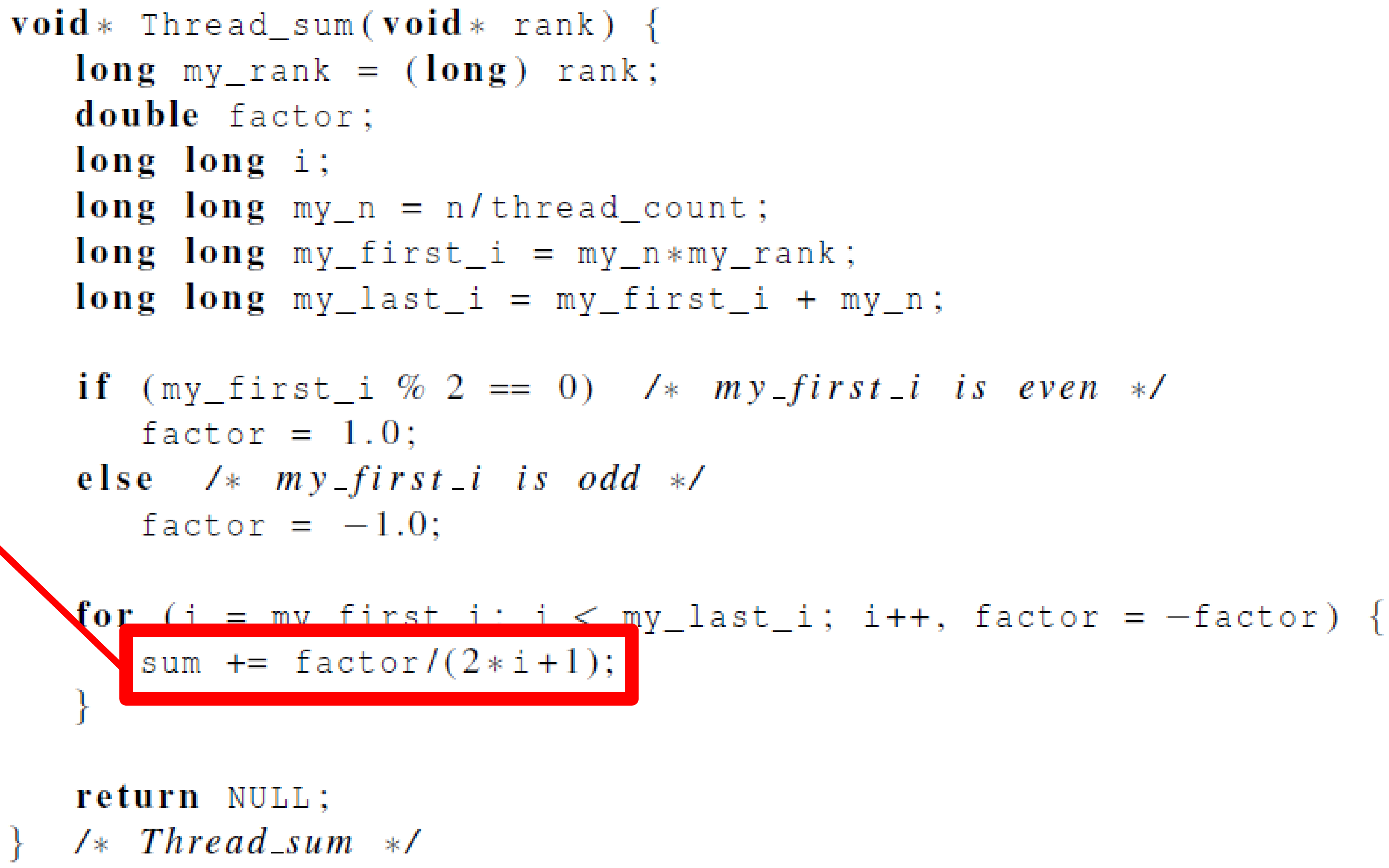
Using a dual core processor



	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

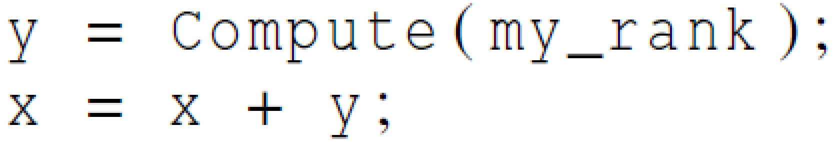
A thread function for computing π



```
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 */
```

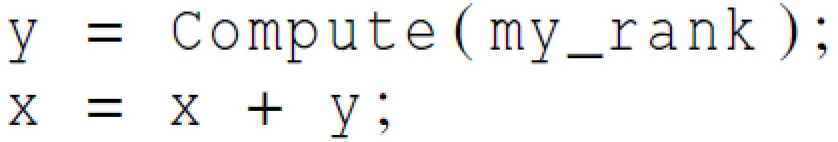
1. Load ‘factor’ into register
2. Load ‘i’ into register
3. Load ‘sum’ into register
4. Compute sum + factor/(2i + 1)
5. Store the result into ‘sum’

Possible race condition

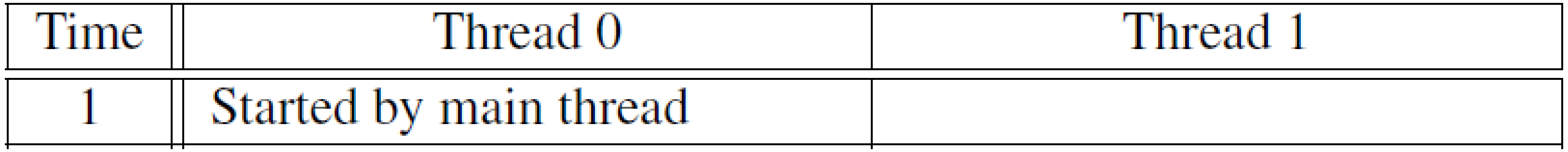


```
y = Compute(my_rank);
x = x + y;
```

Possible race condition

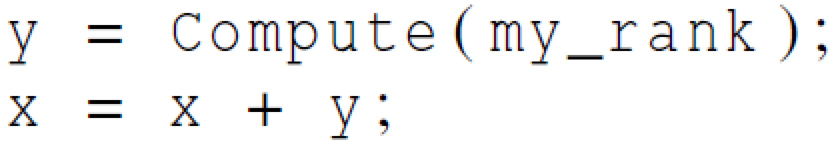


```
y = Compute(my_rank);
x = x + y;
```

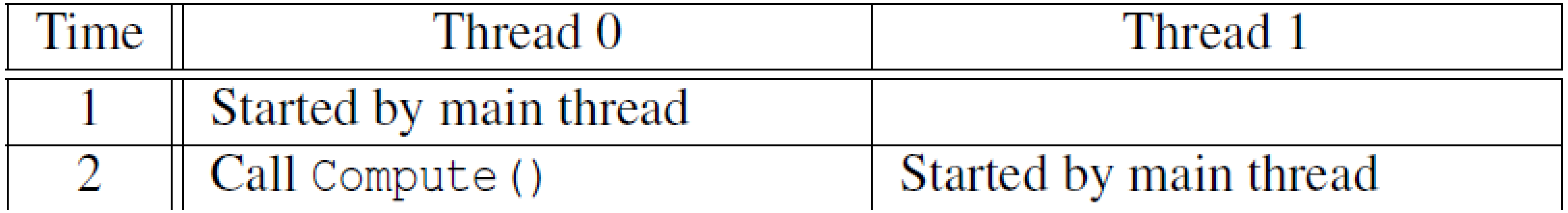


Time	Thread 0	Thread 1
1	Started by main thread	

Possible race condition

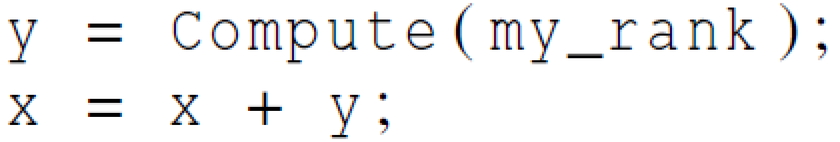


```
y = Compute(my_rank);
x = x + y;
```

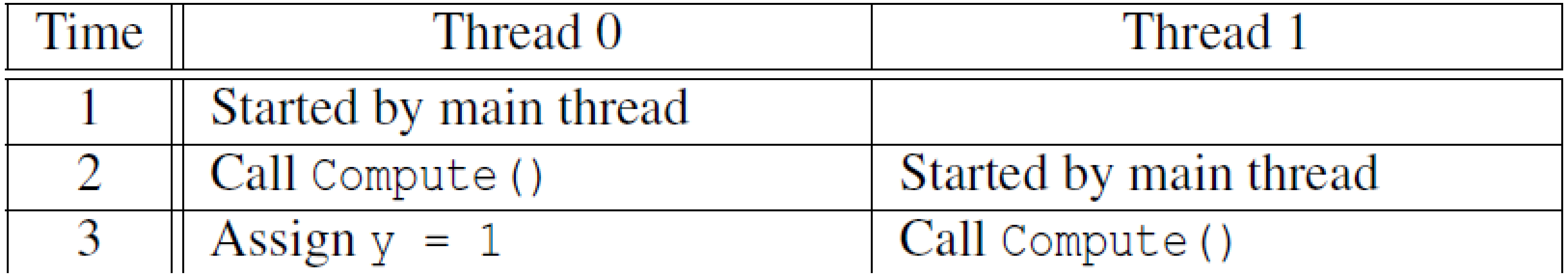


Time	Thread 0	Thread 1
1	Started by main thread	
2	Call `Compute()`	Started by main thread

Possible race condition

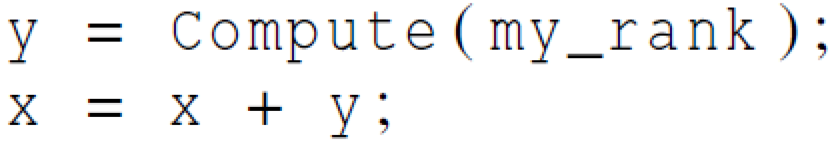


```
y = Compute(my_rank);
x = x + y;
```

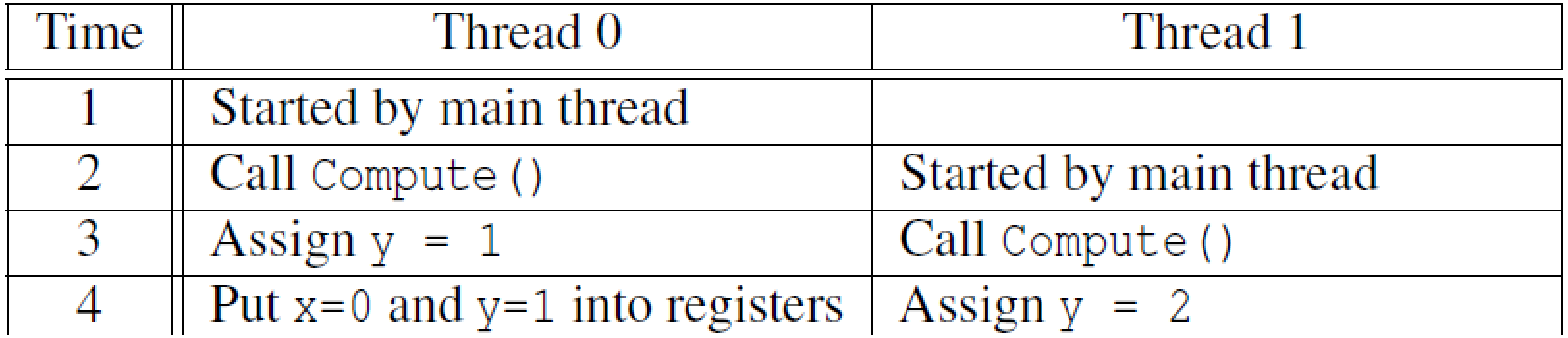


Time	Thread 0	Thread 1
1	Started by main thread	
2	Call `Compute()`	Started by main thread
3	Assign `y = 1`	Call `Compute()`

Possible race condition

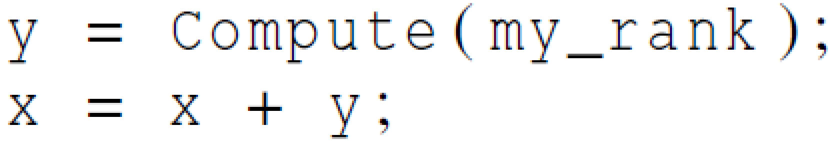


```
y = Compute(my_rank);
x = x + y;
```

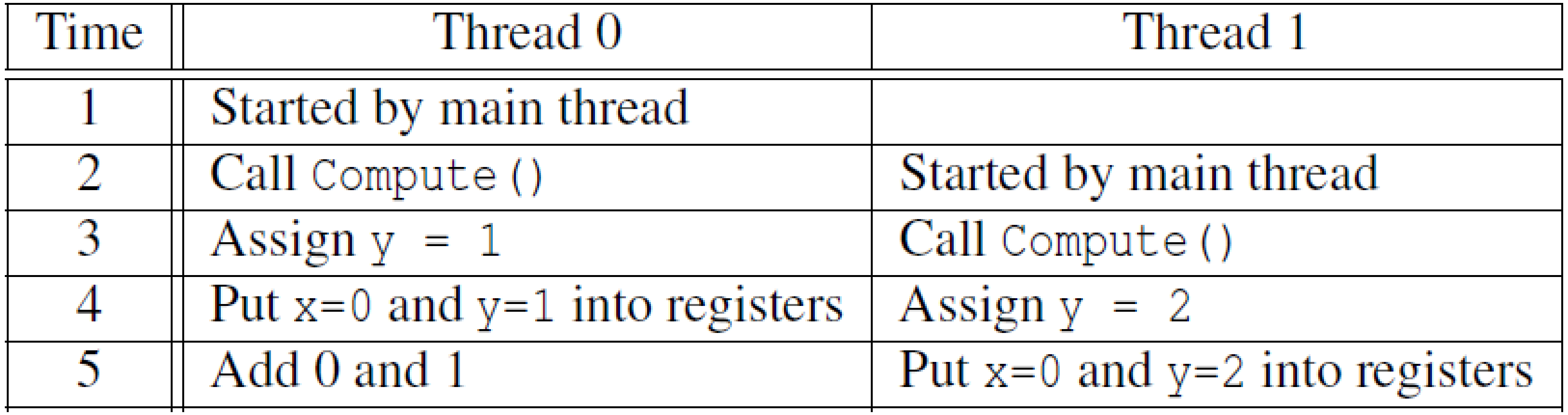


Time	Thread 0	Thread 1
1	Started by main thread	
2	Call `Compute()`	Started by main thread
3	Assign `y = 1`	Call `Compute()`
4	Put `x=0` and `y=1` into registers	Assign `y = 2`

Possible race condition

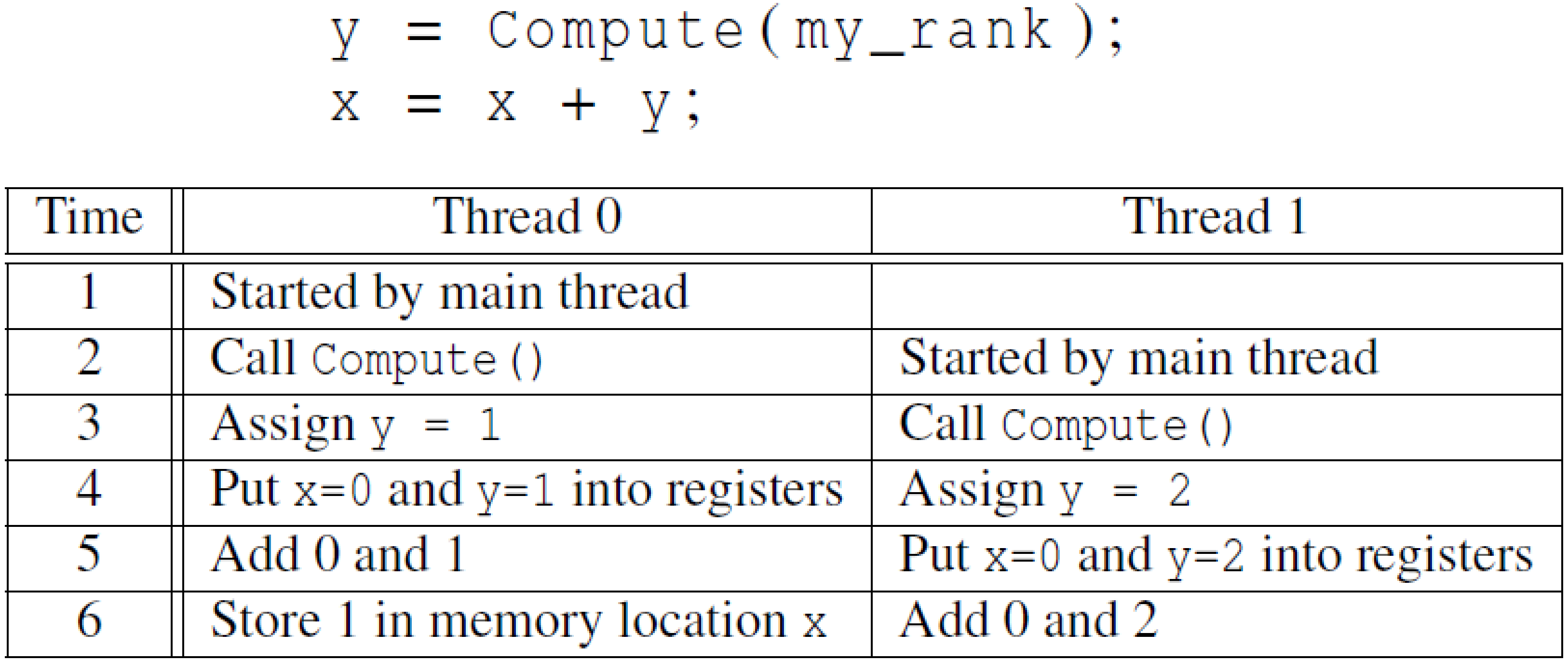


```
y = Compute(my_rank);
x = x + y;
```



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

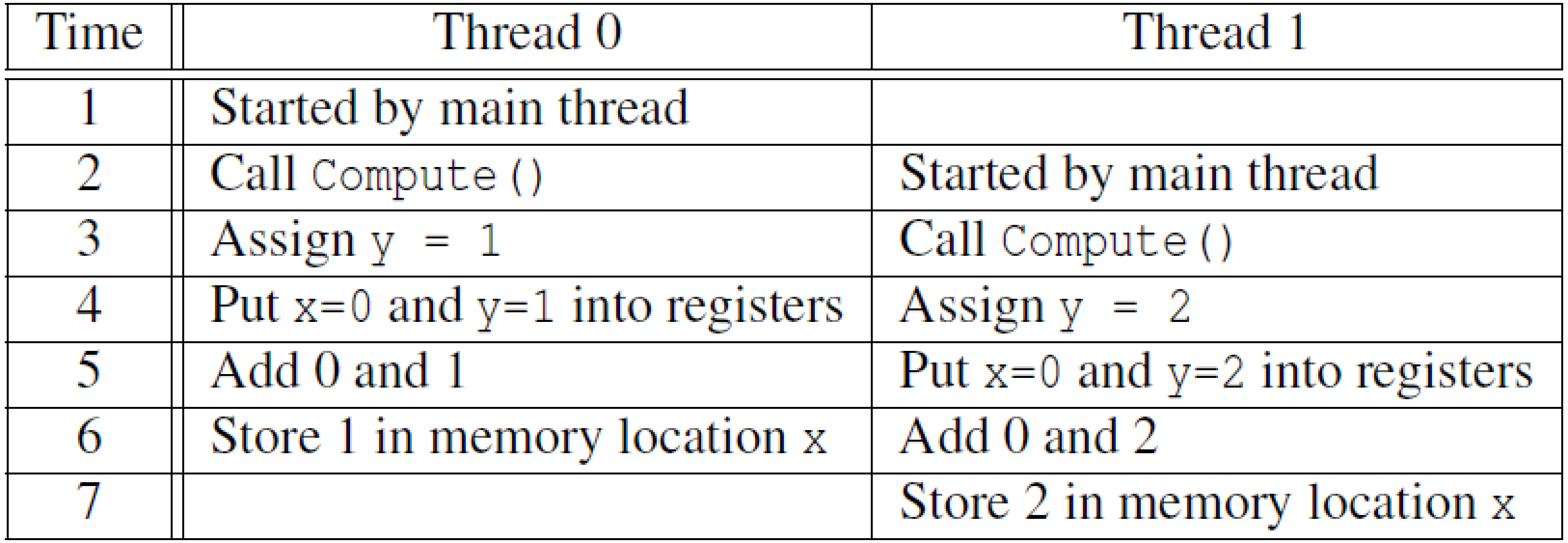


```
y = Compute(my_rank);
x = x + y;
```

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;
```

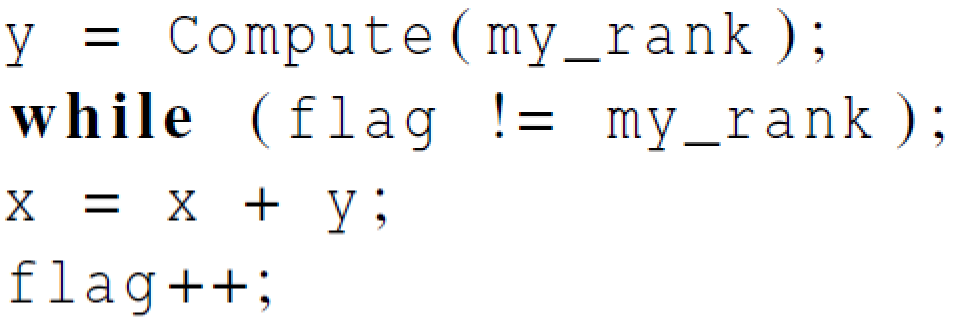


Time	Thread 0	Thread 1
1	Started by main thread	
2	Call `Compute()`	Started by main thread
3	Assign `y = 1`	Call `Compute()`
4	Put `x=0` and `y=1` into registers	Assign `y = 2`
5	Add 0 and 1	Put `x=0` and `y=2` into registers
6	Store 1 in memory location `x`	Add 0 and 2
7		Store 2 in memory location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

Questions?

Possible solution: Busy-Waiting

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

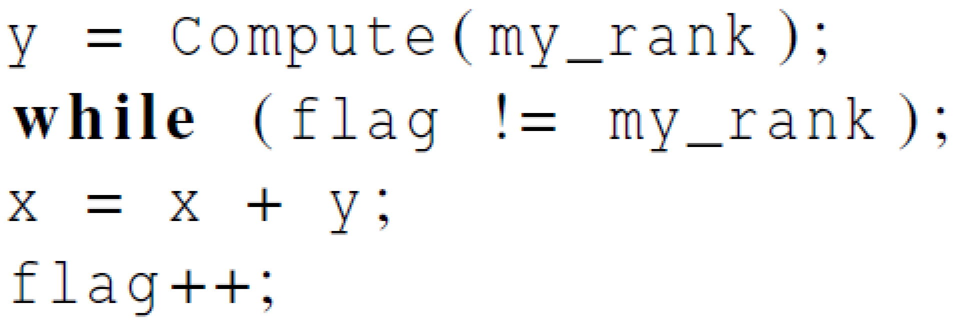


```
y = Compute(my_rank);
while (flag != my_rank);
x = x + y;
flag++;
```

flag initialized to 0 by main thread

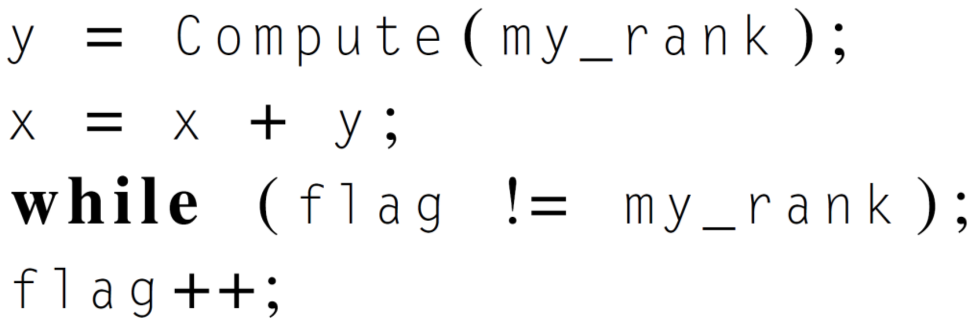
Possible danger: Optimizing compilers

This code:



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

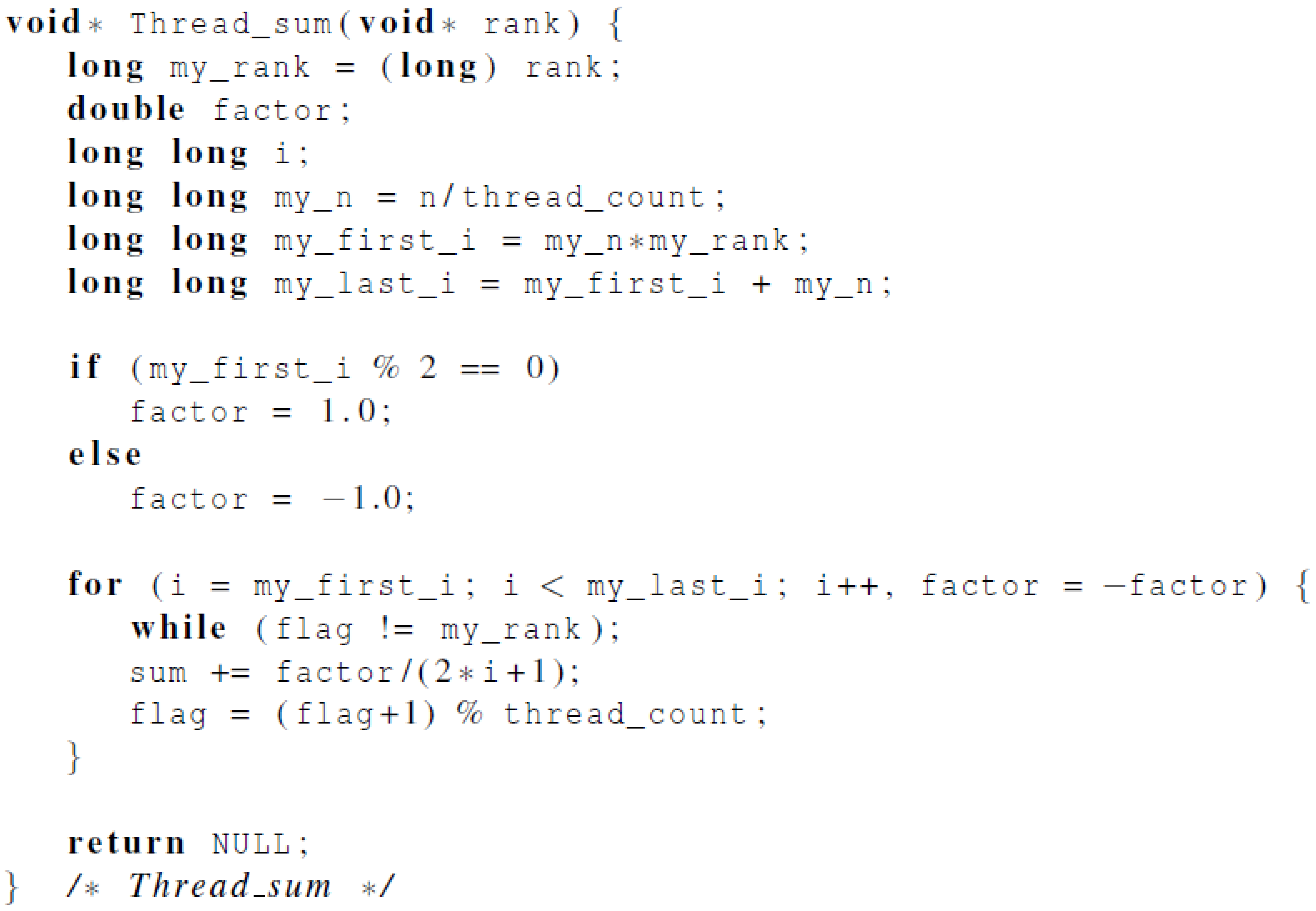
Could be rearranged by the compiler as:



```
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)

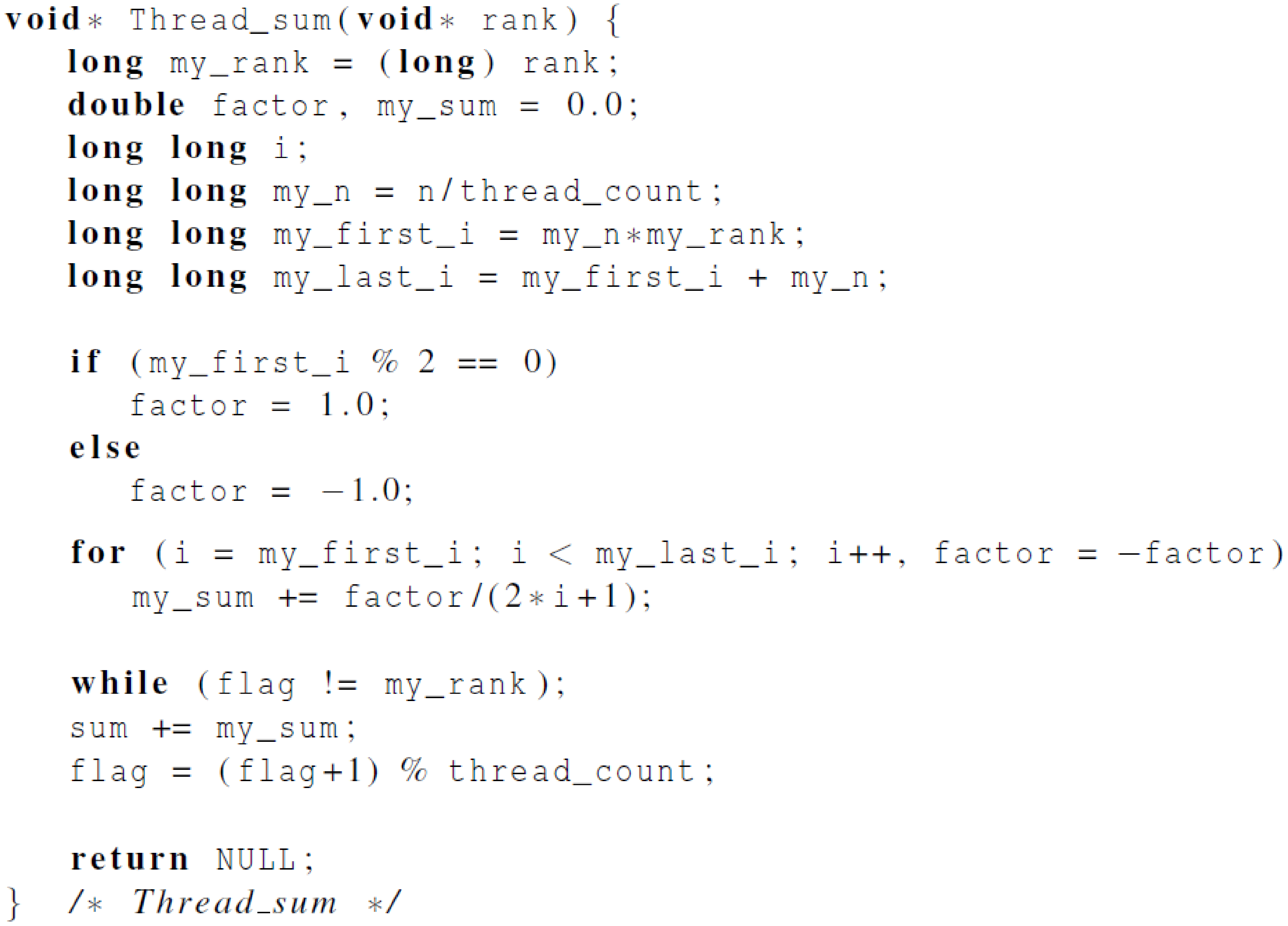
Pthreads global sum with busy-waiting



```
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

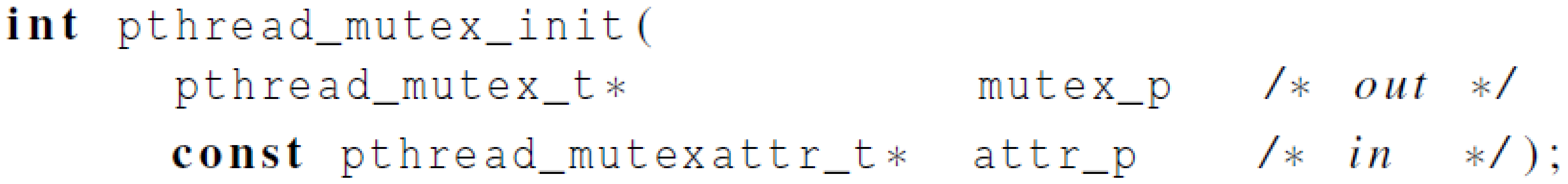
- A thread that is busy-waiting may continually use the CPU accomplishing nothing.

- 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.

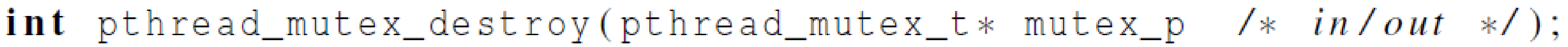
- The Pthreads standard includes a special type for mutexes: pthread_mutex_t.



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

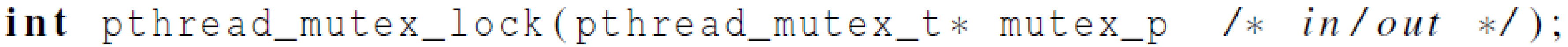
Mutexes

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



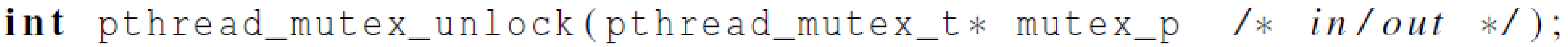
```
int pthread_mutex_destroy(pthread_mutex_t* mutex_p  /* in/out */);
```

- 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.

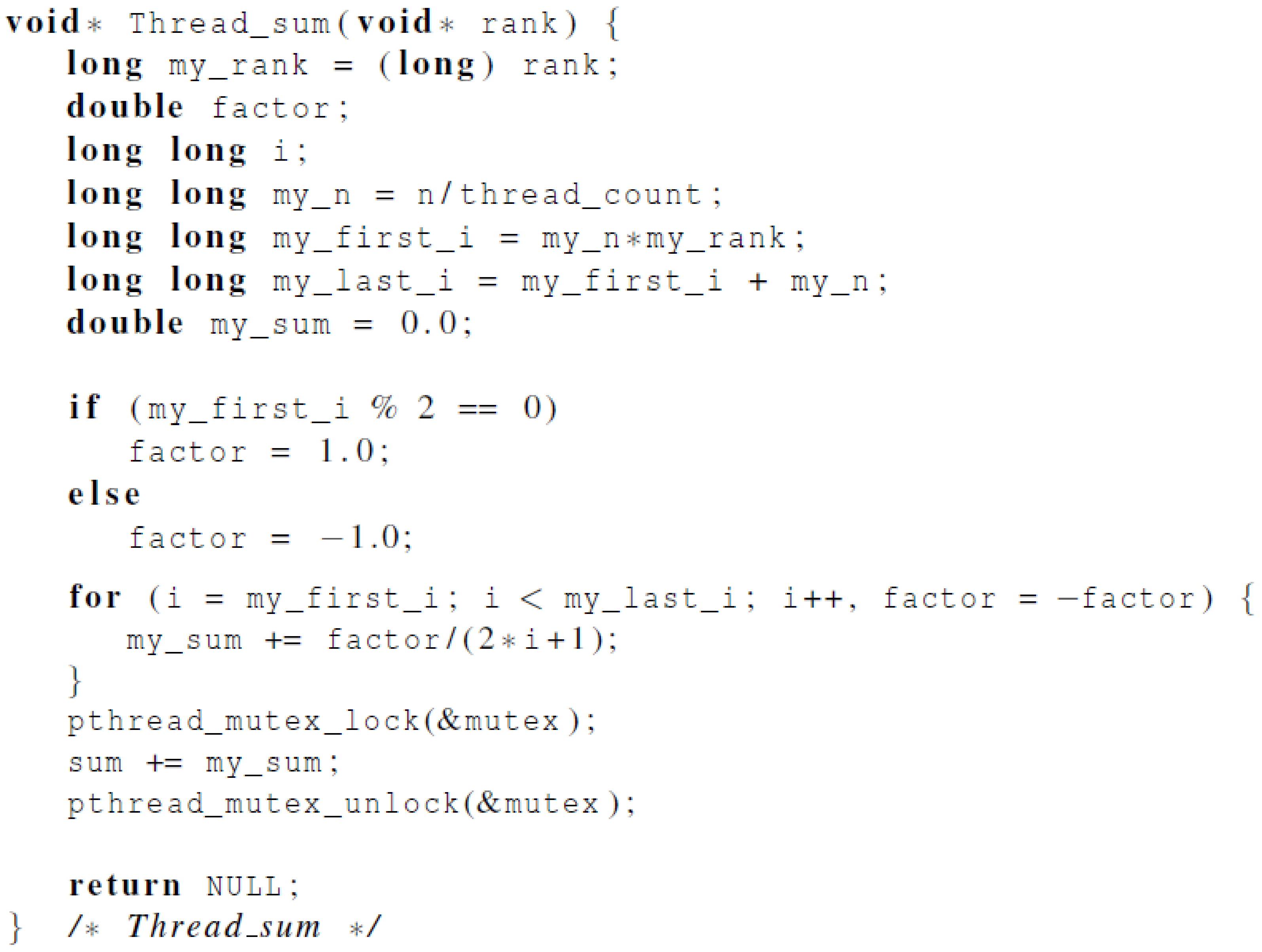
- If a mutex is locked, the thread is blocked and placed in a queue Q of waiting threads. If the queue Q employed by a semaphore is a FIFO queue, no starvation will occur.

Deadlocks

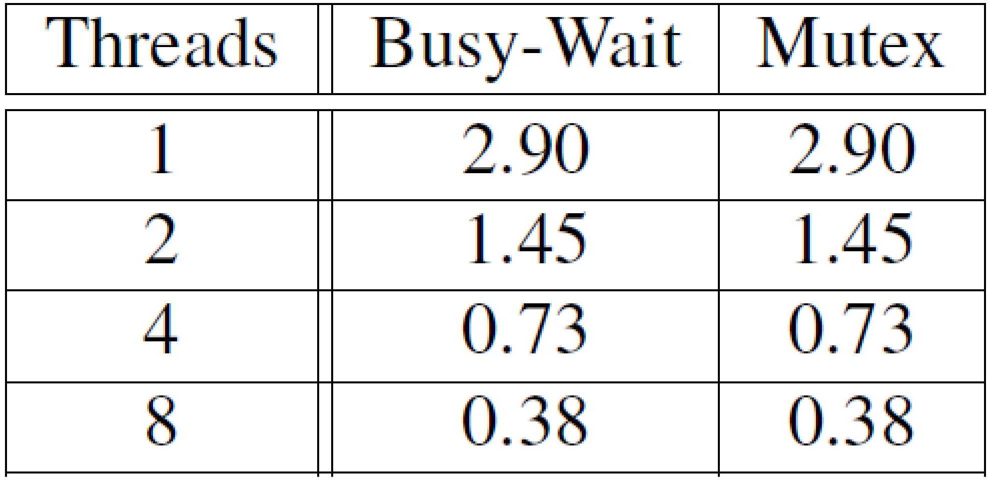
- **Deadlock:** is any situation in which no member of some group of entities can proceed because each waits for another member, including itself, to take action, such as sending a message or, more commonly, releasing a lock.
- E.g., locking mutexes in reverse order

```
/* 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

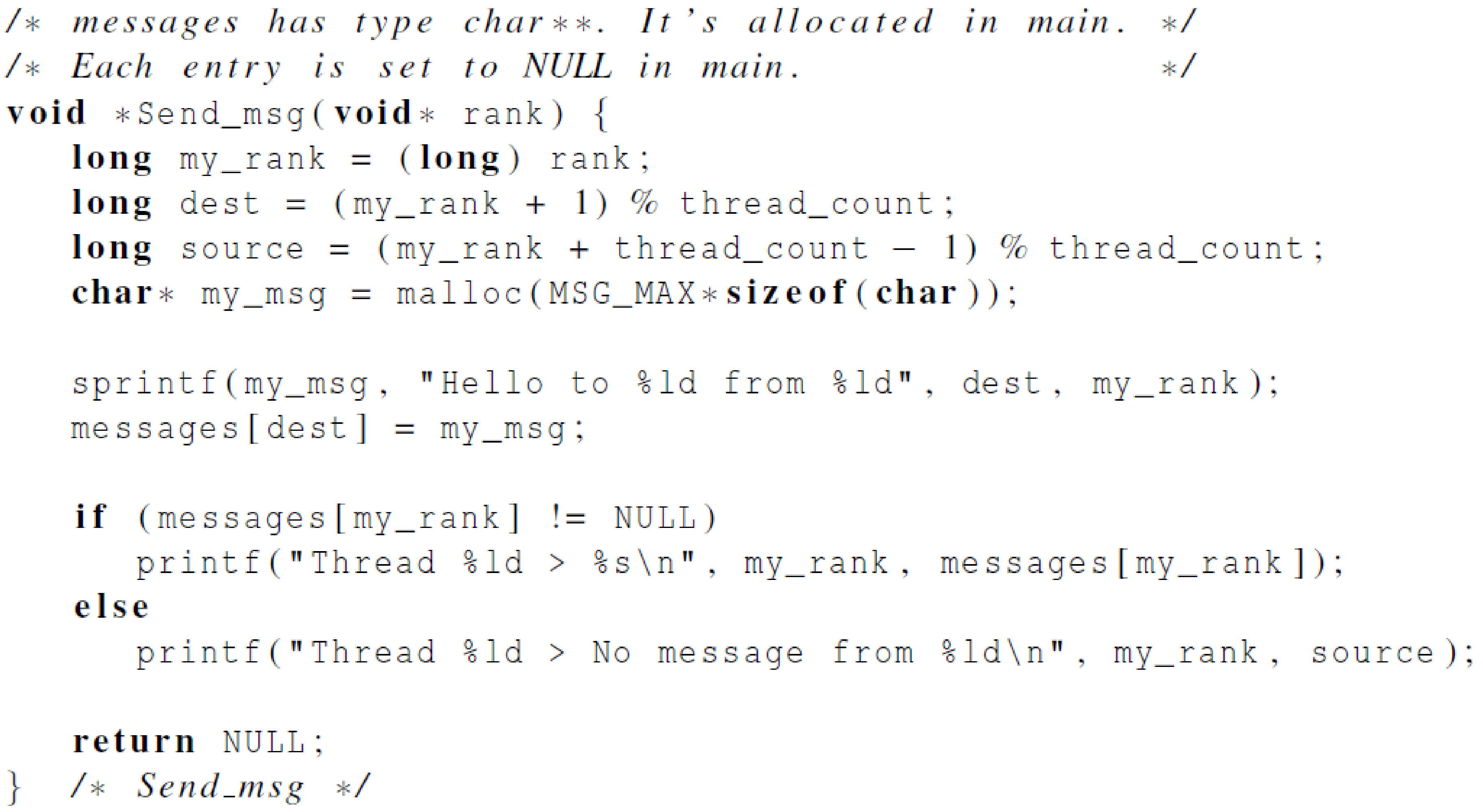
Questions?

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

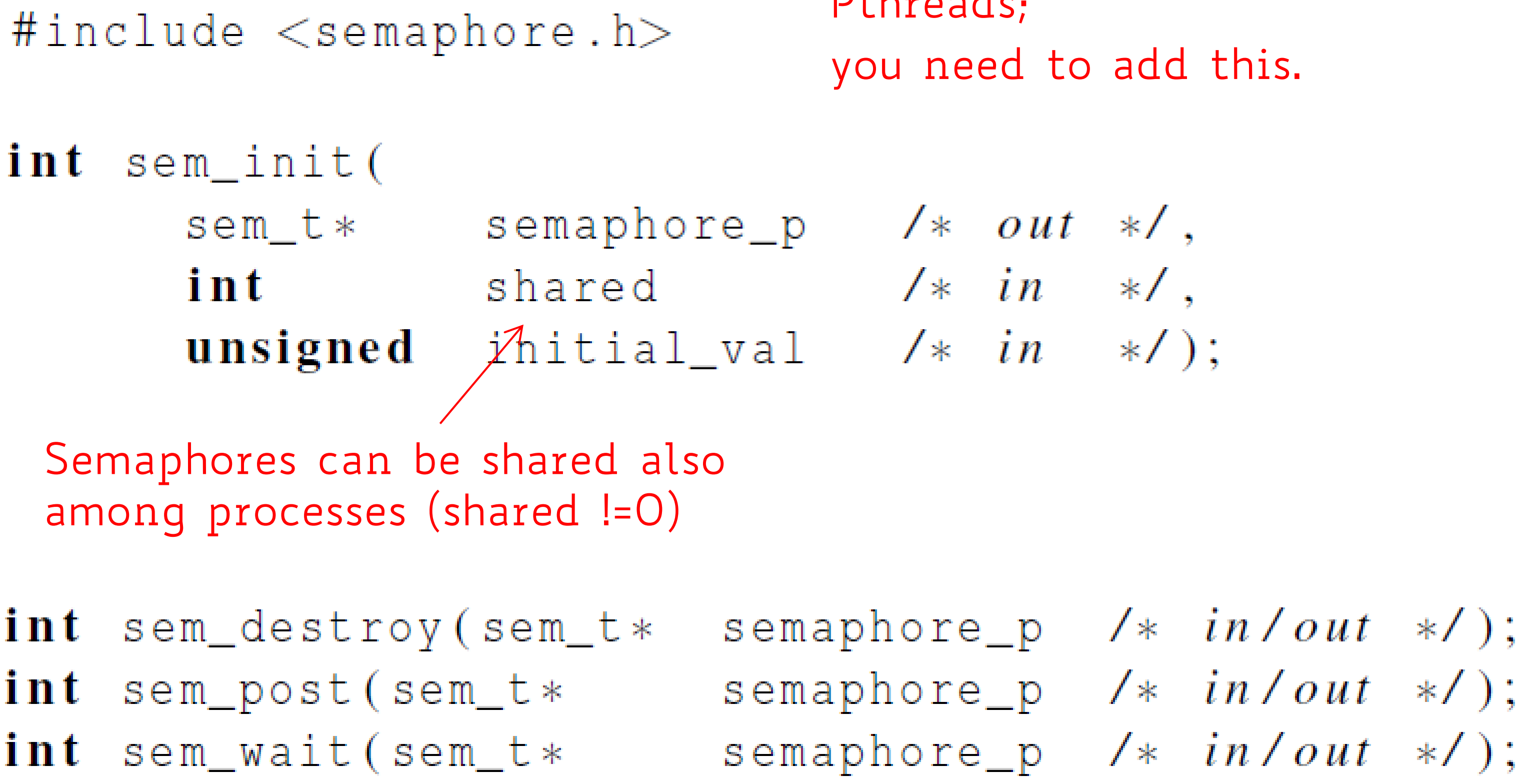
How to fix it?

- 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>

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

Semaphores are not part of Pthreads;

you need to add this.

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.

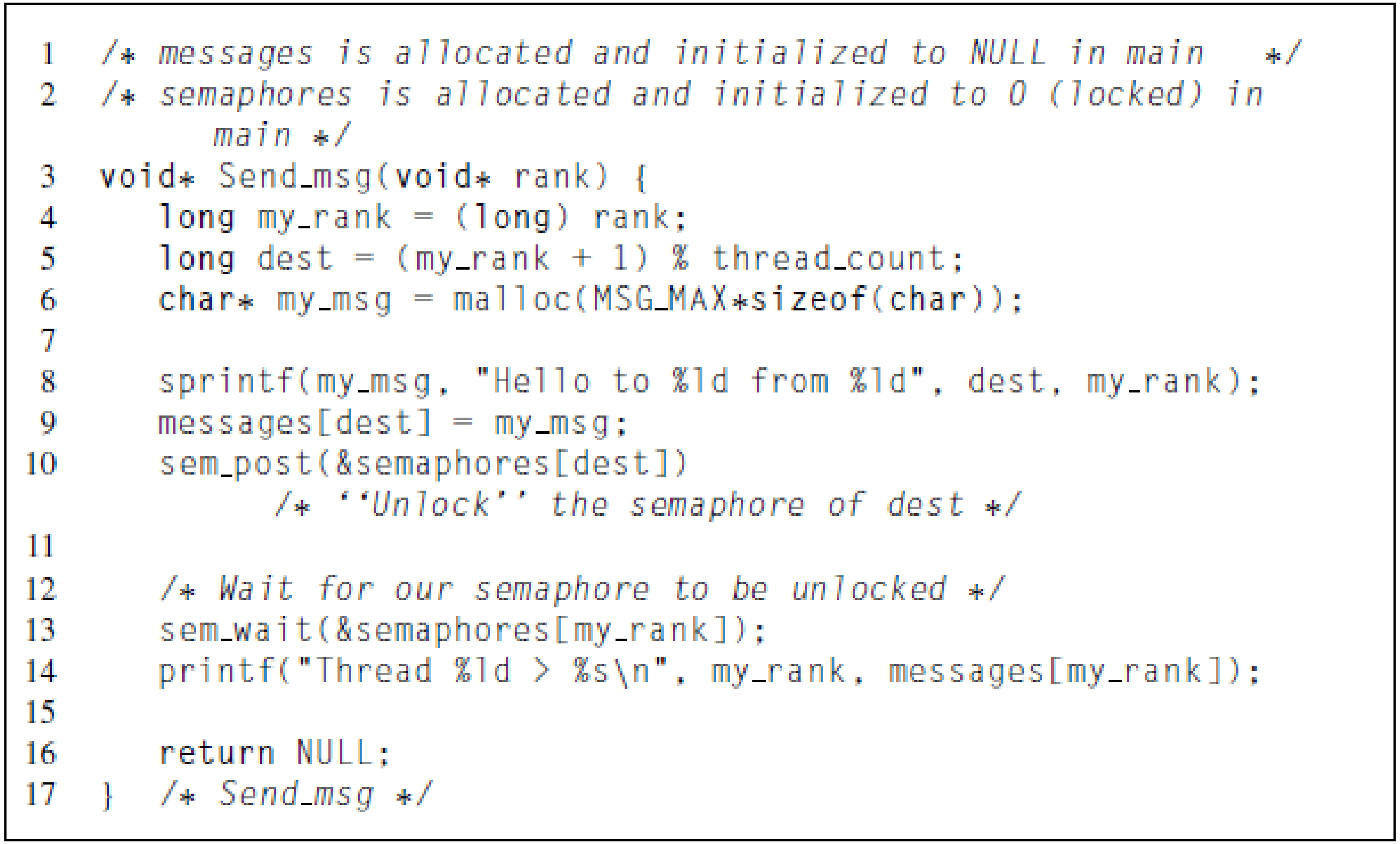
*sem_post(sem_t *sem)* if there is a a thread waiting in *sem_wait()*, that thread can proceed with the execution. Otherwise, the semaphore is *incremented*.

*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



```
/* messages is allocated and initialized to NULL in main   */
/* semaphores is allocated and initialized to 0 (locked) in
      main */
void* Send_msg(void* rank) {
   long my_rank = (long) rank;
   long dest = (my_rank + 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;
   sem_post(&semaphores[dest])
         /* ``Unlock'' the semaphore of dest */

   /* Wait for our semaphore to be unlocked */
   sem_wait(&semaphores[my_rank]);
   printf("Thread %ld > %s\n", my_rank, messages[my_rank]);

   return NULL;
}  /* Send_msg */
```

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