# PERFORMANCE

# Speedup

- Number of cores = p
- Serial run-time = $T_{serial}$
- Parallel run-time = $T_{parallel}$

linear speedup

$$T_{parallel} = T_{serial} / p$$

# Speedup of a parallel program

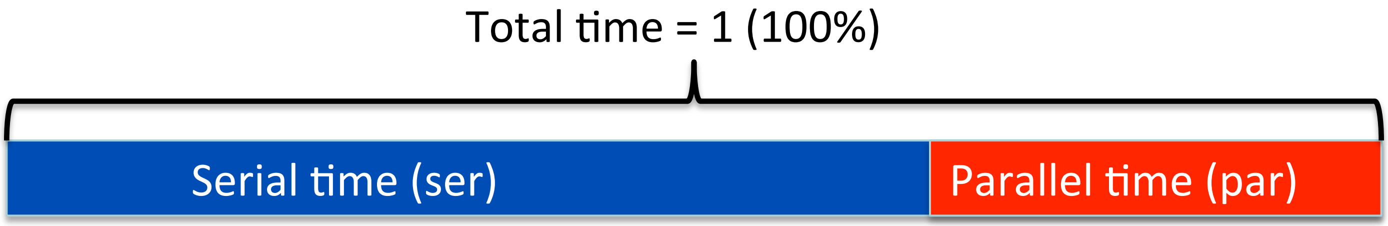
$$S = \frac{T_{serial}}{T_{parallel}}$$

# Efficiency of a parallel program

$$E = \frac{S}{p} = \frac{\left(\dfrac{T_{serial}}{T_{parallel}}\right)}{p} = \frac{T_{serial}}{p \cdot T_{parallel}}$$

# Amdahl’s Law

- Unless virtually all of a serial program is parallelized, the possible speedup is going to be very limited — regardless of the number of cores available.



$$SpeedUp = \frac{OriginalTime}{ImprovedTime} = \frac{1}{ser + \frac{par}{\# cores}} = \frac{1}{ser + \frac{(1 - ser)}{\# cores}}$$

# Scalability

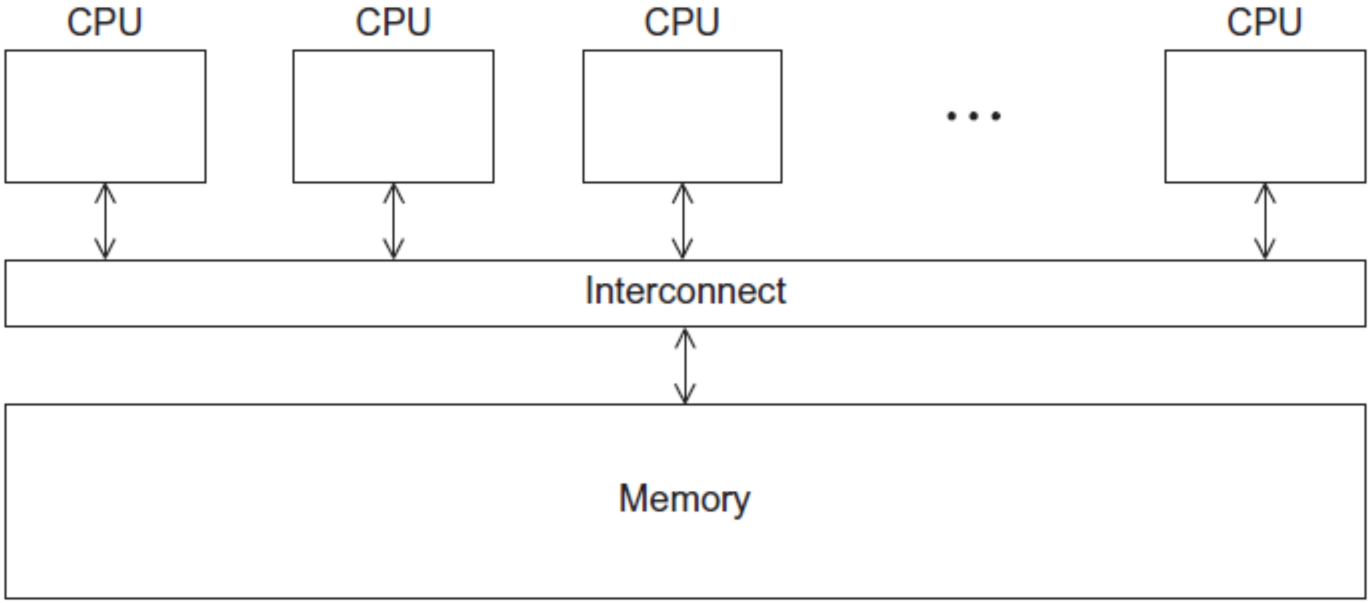
- In general, a problem is *scalable* if it can handle ever increasing problem sizes.
- If we increase the number of processes/threads and keep the efficiency fixed without increasing problem size, the problem is *strongly scalable*.
- If we keep the efficiency fixed by increasing the problem size at the same rate as we increase the number of processes/threads, the problem is *weakly scalable*.

# Some Linux commands

- Information about the hardware
  - cat /proc/cpuinfo
- Display Linux tasks (processes)
  - top
- We're are going to use GCC
  - gcc --version

# INTRODUCTION TO PTHREADS

# A Shared Memory System



# 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.

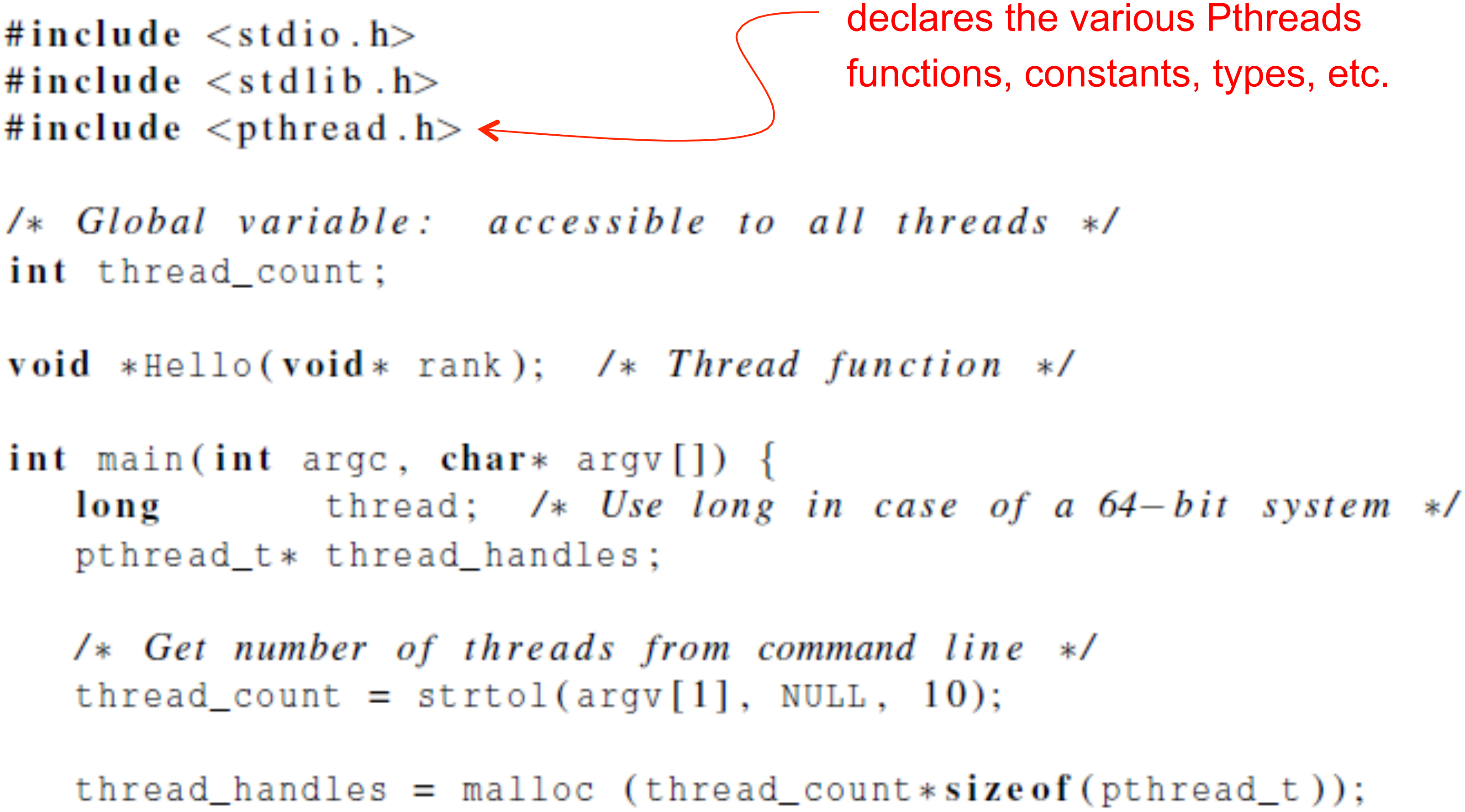
# POSIX®Threads

- 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.

# Caveat

- The Pthreads API is only available on POSIX systems — Linux, MacOS X, Solaris, HPUX, …

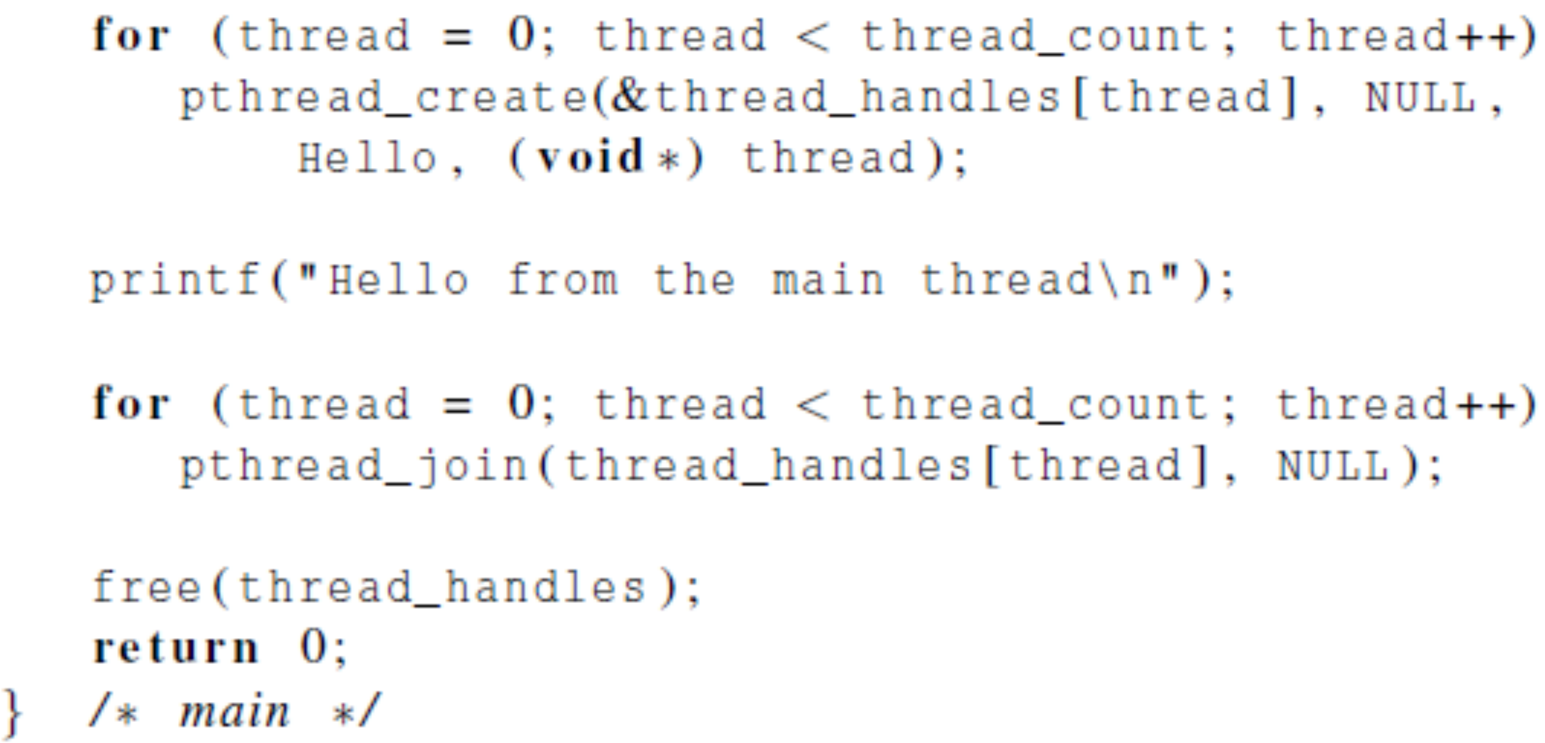
# Hello World! (1)



```
#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));
```

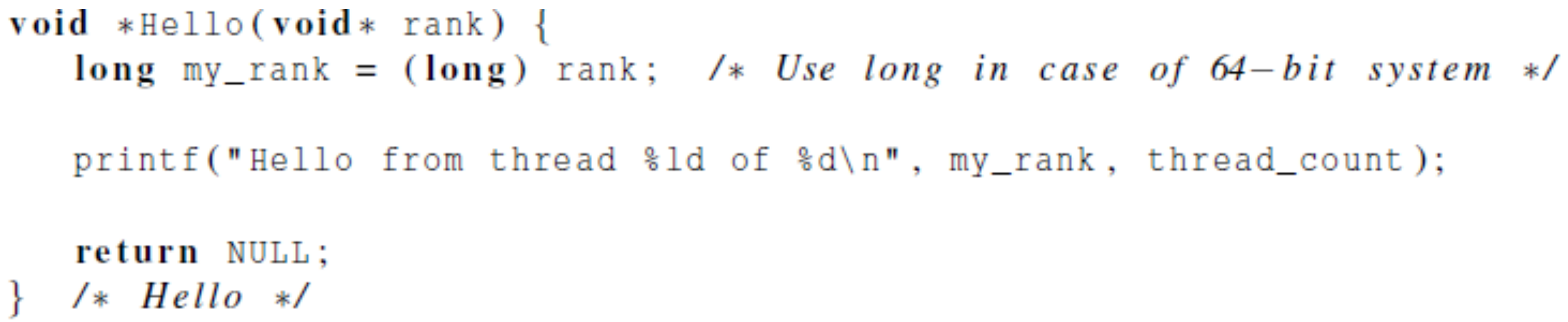
declares the various Pthreads functions, constants, types, etc.

# Hello World! (2)



```
   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 */
```

# Hello World! (3)



```
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 */
```

# 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
```

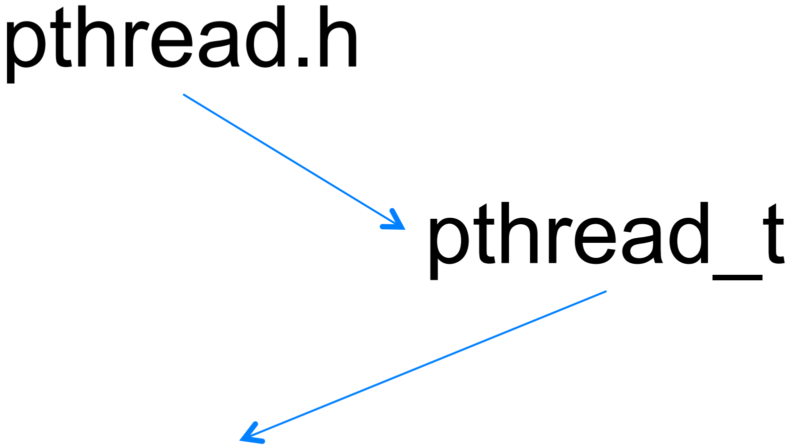
```
. / pth_hello 4
```

```
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
```

# Global variables

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

# Starting the Threads


pthread.h

pthread_t

One object for each thread.

```
int pthread_create (
        pthread_t*  thread_p /* out */ ,
        const pthread_attr_t*  attr_p /* in */ ,
        void*  (*start_routine ) ( void ) /* in */ ,
        void*  arg_p /* in */ ) ;
```

# pthread_t objects

- 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.

# A closer look (1)

```
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 be using, so we just pass NULL.

Allocate before calling.

# A closer look (2)

```
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 argument that should be passed to the function *start_routine*.

The function that the thread is to run.

Copyright © 2010, Elsevier Inc. All rights Reserved

# Function started by pthread_create

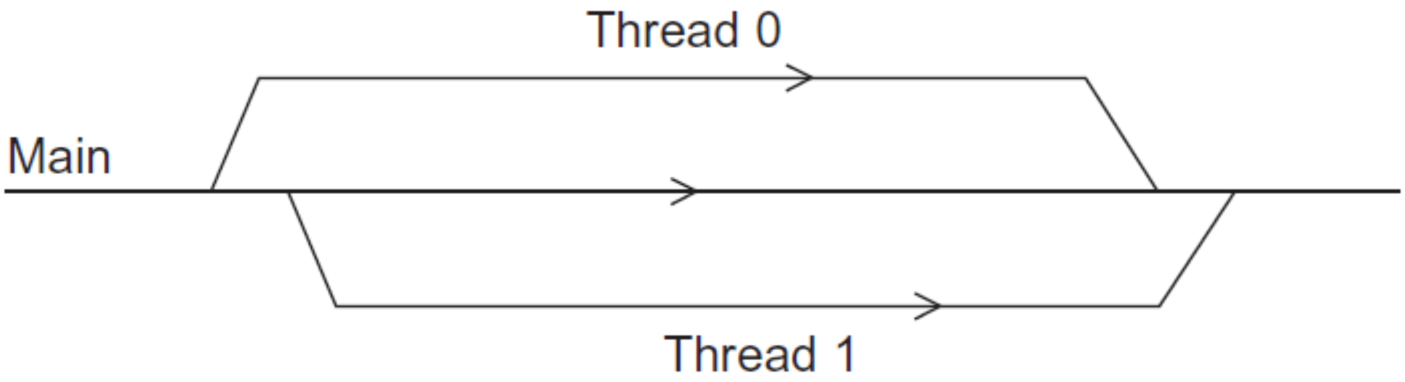
- Prototype:

  ```
  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.

# Running the Threads



Main thread forks and joins two threads.

# Stopping the Threads

- 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.

# Input and Output

- In shared memory programs, only the master thread or thread 0 will access *stdin*.

- In shared memory programs, all the processes/threads can access *stdout* and *stderr*.

# Input and Output

- However, because of the indeterminacy of the order of output to *stdout*, in most cases only a single process/thread will be used for all output to *stdout* other than debugging output.
- Debug output should always include the rank or id of the process/thread that’ s generating the output.

# Input and Output

- Only a single process/thread will attempt to access any single file other than *stdin*, *stdout*, or *stderr*. So, for example, each process/thread can open its own, private file for reading or writing, but no two processes/threads will open the same file.