Arquiteturas de Alto Desempenho 2025/2026

Practical class AAD_P02 (2025-09-22)

Tomás Oliveira e Silva

1 Objectives

Learn how to parallelize code the hard way (`pthreads`), but with absolute control of what is going on. List of functions you will learn about:

- `pthread_create` and `pthread_join`; `pthread_detach`
- `pthread_mutex_lock` and `pthread_mutex_unlock`
- `pthread_cond_signal` and `pthread_cond_wait`
- `pthread_setaffinity_np` (optional)

Please do not do a TL;DR on this stuff. The linux command `man` is your friend.

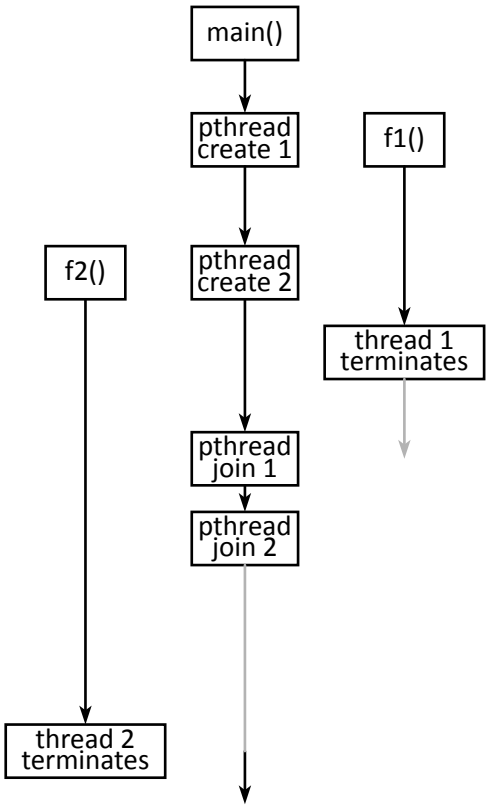
2 Introduction

Contemporary processors have more than one execution pipeline (called chip-level multiprocessing). Moreover, each execution pipeline may support more than one execution flow. This is called hyper-threading by Intel/AMD and simultaneous multithreading (SMT) by almost everybody else (ARM, IBM, …). Each execution pipeline is usually called a physical core. Each physical execution flow (a thread) runs on a so-called virtual core (vCore or vCPU). It shares resources (caches, execution units) with other vCPUs of the same physical core. Furthermore, it is also possible to have more than one processor sharing the same physical memory. Indeed, large servers may have two (2P), four (4P) or even more processors in the same motherboard.

How can we take advantage of this in our programs? In this practical class you will learn how to do this using `pthreads` (POSIX threads). If there are enough physical resources, two or more threads of the same program can be executed in parallel. To take advantage of this (using `pthreads`) it is necessary to identify the regions of your code than can be executed in parallel, and then to modify the code to take advantage of the parallel execution. Several aspects of any parallel execution have to be taken in consideration:

- How do we coordinate the work that has to be done? Does each thread blindly do part of the work, or is a more sophisticated approach necessary?
- Where do we store the results?
- How to balance the work among the threads? In recent times this has become more relevant, as some new processors have physical cores with different capabilities: the so-called E(fficient) and P(erformance) cores.

The figure on the right-hand side shows an execution flow of a program that launches two threads and then waits for their completion. The thread entry functions are `f1()` and `f2()`. Time flows from top to bottom. A line in gray means that the thread is stopped. The resources used by the thread (stack memory and so on) are only freed after a join with that thread is successful.



3 POSIX thread creation and destruction

Threads are created using the `pthread_create()` function. Waiting for a specific thread to finish its work is done using the `pthread_join()` function. There does not exist a function for the completion of an unspecified thread, so if your program has several threads running and you need to see if any of them has terminated you will have to try to join to each one of them using the `pthread_tryjoin_np()` function; it fails if the thread has not terminated (but, crucially, it does not block execution).

This is a good time to read information about each of these functions. Just do

```
$ man pthread_create
$ man pthread_join
$ man pthread_tryjoin_np
```

(On Ubuntu, these man pages are in the `manpages-dev` package.) Any thread can create other threads. Any thread can join with other threads.

The file `create_and_join.c` contains an example of the use of `pthread_create()` and `pthread_join()`. Examine it carefully. In particular, study how arguments are passed and returned from the threads that are created. Compile and run it:

```
$ make create_and_join
$ ./create_and_join 4
```

The argument to the `create_and_join` program is the number of threads you want to create. Try other values ($2 \leqslant$ **n_threads** ≤ 16) and see what happens.

It is also possible to detach a thread, so that its resources are freed immediately when it terminates. (Be aware: is you have passed the address of a local variable of a detached thread to other thread you are inviting **big** trouble.) Once a thread is detached it cannot be joined.

```
$ man pthread_detach
```

4 Critical regions: using mutexes

Planning concurrent execution of two or more threads requires careful consideration of how shared memory areas are to be modified. If the same memory area is to be modified by different threads, when one of these threads wants to modify it, it is necessary to block access to that area to all other threads. This is usually done with mutual exclusion locks (known as mutexes).

Mutexes can be used to protect a given region of code, called a critical section, that modifies a shared memory area. To lock access to the shared area call `pthread_mutex_lock()`. The call will succeed if the lock variable was “open”, and the lock variable becomes “closed”. Otherwise, the thread will wait until the lock becomes open again. Waiting is usually performed by yielding the processor. Releasing the lock is done by calling `pthread_mutex_unlock()`. Doing this will open a closed lock. If another thread was waiting for that lock to open, it resumes execution (if more than one thread was waiting only one of them resumes execution).

The file `lock_and_unlock.c` contains a simple example of the use of the `pthread_mutex_lock()` and `pthread_mutex_unlock()` functions. Examine it carefully. In particular, run the program when locking and unlocking is enforced and when it is not. Observe also the difference in execution times. Just compile and run it:

```
$ make lock_and_unlock
$ time ./lock_and_unlock 0
$ time ./lock_and_unlock 1
```

Experiment with other number of threads to see what happens. Be aware that writting concurrent programs using locks requires care (deadlocks…) so spend some time reading the documentation. Notice that there exist several kinds of locks:

```
$ man pthread_mutex_lock
```

In general, each critical section should be small and its execution should be fast, so that other threads do not have to wait much time trying to acquire the lock. Outside of the critical section, the execution time can be as long as needed. Doing things in this way will ensure that a very large fraction of the execution time is actually spend doing useful work and not spend locking and unlocking.

5 Pool of thread workers: using conditions variables

Using a critical region to coordinate work among threads forces the critical region code to do all that coordination. That is not bad by itself, but when there is not enough work to distribute to all threads, the idle threads will have to enter and leave the critical region repeatedly until there is work to do or it is time to terminate. This can be done in a busy-wait loop, perhaps sleeping for a small amount of time — there exists a `usleep()` function for that — before entering the critical section again. That either consumes lots of computational resources or has a suboptimal reaction time.

Semaphores (not studied here) or condition variables can help us here. A condition variable with accompanying mutex allow us to put a thread to sleep and to wake it up in an optimal way. After creating and initializing a condition variable, threads can be made to wait on that condition variable by using the `pthread_cond_wait()` function. If someone does a `pthread_cond_signal()` call on that variable, one, and only one, of the threads waiting for it is woken; if none is waiting, nothing happens. This is quite useful when there is not enough work, as the idle threads can be woken up one by one in an optimal way when new work becomes available. It is also possible to broadcast the wake up signal to all waiting threads. That is done using the `pthread_cond_broadcast()` function. As usual, to know more read the documentation:

```
$ man pthread_cond_signal
```

Another problem where condition variables are useful is the producer-consumer problem. To distribute work, each producer thread creates work units and places them is a queue. It keeps producing them until the queue is full. Each consumer thread keeps taking work units from the queue until is is empty. Condition variables can be used to wake up either producer or consumer threads, as appropriate.

The `producer_consumer.c` file contains a simple implementation of a producer/consumer program that uses two condition variables to control when a producer or consumer has to be woken. Study it carefully. Compile and execute the program. Were all producer items output by a consumer? How about the order the items were printed?

6 Pinning threads to cores (optional stuff, information only)

It is possible to restrict execution of a thread to a given list of core numbers. This is done using the `pthread_setaffinity_np()` function. To restrict to execution to core number `core_number` do something like this:

```
{
  cpu_set_t affinity;
  CPU_ZERO(&affinity);
  CPU_SET(core_number,&affinity);
  if(pthread_setaffinity_np(pthread_self(),sizeof(affinity),&affinity) != 0)
  {
    fprintf(stderr,"pthread_setaffinity_np() failed\n");
```

```
    exit(1);
  }
```

For more information, just read the documentation. On Linux, a simple way to find out the number of vCPUs is the following:

```
$ grep 'model name' /proc/cpuinfo
```

Just count the number of lines…

7 Putting it all together: parallelization of `MandelbrotCount.c`

It is now time to put all that you learned about `pthreads` in action. Modify the `MandelbrotCount.c` source code so that it runs in parallel. (For this class, the size of the square that has to be processed was increased to 8193×8193.) Try it using:

1. A static subdivision of work: divide the 8193×8193 square in vertical stripes of about the same width. Each thread processes one strip.
2. Likewise, but use horizontal stipes.
3. Use a critical region to coordinate work among the threads: divide the 8193×8193 square in smaller squares, say 482×482, and let the threads process the next small square not yet processed. Why 482?
4. Do it as in the previous item, but use the producer/consumer paradigm with a single producer. We do not need more that one because the consumers take much more time to process a work item than the producer takes to generate one!

In each case, print the wall time just before exiting each worker thread and measure the execution time with the `time` command.

Which approach is the best in terms of

- execution time (as a function of the number of threads),
- efficiency (keep all threads always busy), and
- simplicity of the implementation?