Parallel and concurrent programming models

J. Daniel García Sánchez (coordinator)
David Expósito Singh
Javier García Blas

Computer Architecture
ARCOS Group
Computer Science and Engineering Department
University Carlos III of Madrid

1. Module structure

This module is structured in three lessons:

- **Parallel programming with OpenMP**. It presents the basic concepts of the OpenMP parallel programming model through examples.
- **Concurrent programming in C++11**. Introduces the C++11 concurrency model as a portable mechanism for concurrent programming.
- **Memory consistency model in C++**. Reinforces memory consistency concepts through the standard C++ memory model. Pay special attention to relaxed memory models.

2. Parallel programming with OpenMP

This lesson has the following general structure:

1. Introduction.
2. Threads in OpenMP.
3. Synchronization.
4. Parallel loops.
5. Synchronization in the master thread.
6. Data sharing.
7. Sections and scheduling.

2.1. Introduction

OpemMP is an API that allows to express parallel applications for shared memory systems, simplifying the way you write programs. It consists of a set of compiler directives, a function library, and a set of environment variables.

All OpenMP directives are indicated by a **pragma** directive that uses the **omp** prefix. For example:

```
#pragma omp parallel
{
  f();
  g();
}
```

2.2. Threads in OpenMP

The parallelism in OpenMP follows the *fork-join* model, where a sequential application has parallel sections. At the start of the program, there is a *master thread* that executes the sequential sections. When you enter a parallel region (marked with a **omp parallel** directive), a set of threads is started.

For measuring the time taken to execute a section of code, you can use the function provided by OpenMP (**omp_get_wtime()**).

2.3. Synchronization

OpenMP offers synchronization mechanisms for assisting programmer to avoid data races. The library offers high-level mechanisms (*critical*, *atomic*, *barrier* and *ordered*) and low level (*flush* and locks).

The **critical** directive guarantees that only one thread can enter into the critical section at a time. On the other hand, the **atomic** directive guarantees the atomic updates of a memory location.

2.4. Parallel loops

The **parallel for** directive performs loop division. That is, it distributes the iterations of a loop between the available threads.

One of the most frequent operations in parallel loops is reduction, which is an accumulation operation performed in a loop. The reductions can be made on different elementary operations, which must be associative.

2.5. Synchronization in the master thread

A barrier allows all threads to be synchronized at one point so that all threads are expected to reach that point.

Another way to synchronize is to use a section marked with the **omp master** directive. This section will only be executed by the master thread. If you want to ensure that the section is executed by a single thread, no matter which one, you can use the **omp single** directive. Orderly execution of a reduction with **omp ordered** can also be employed.

OpenMP provides **omp_set_lock()** and **omp_unset_lock()** lock primitives that use the type **omp_lock_t**.

2.6. Data sharing

In OpenMP, a variable can be shared or private.

A shared variable can be a global variable, a **static** variable, or an object stored in dynamic memory. A private variable is a local variable in a function invoked from a parallel section or a local variable defined in a block.

The **omp** clause can be used to control the storage attributes of a variable. The **private** directive creates a local copy of a variable in each thread. The **firstprivate** and **lastprivate** directives help control the management of the initial and final values of the variables.

2.7. Sections and scheduling

A set of parallel sections can be defined within a parallel region. In this case, each section is passed to a different thread and all sections are synchronized with a finalization barrier.

When executing parallel loops, you can select from three execution policies. The *static* policy schedules blocks of iterations of the same size for each thread. The *dynamic* policy causes each thread to take a number of iterations from one queue until all of them have been processed. The *guided* policy causes each thread to take a group of iterations from a queue. In the latter case, as time passes, the number of iterations is reduced each time.

3. Concurrent programming in C++11

This lesson has the following general structure:

- Introduction to concurrency in C++.
- Library overview.
- **Thread** class.
- *Mutex* objects and condition variables.

3.1. Introduction to concurrency in C ++.

The C++11 standard (ISO/IEC 14882: 2011) provides a concurrency model as part of the language specification. This is a significant change compared to previous approaches, since it solves the problem of concurrent code portability between different platforms. In addition, it also solves the inherent problems of concurrency solutions exclusively based on a library, since there are aspects that are most satisfactorily solved with language support. Therefore, the C++11 concurrency solution covers both aspects of the language and aspects of the standard library that accompanies it. On the other hand, this standard has had a great influence on the standard C11 (ISO/IEC 9899:2011), which follows the lines established by C++11.

From the language point of view, C++11 offers a new memory model (presented in the next lesson), as well as a new type of variables with local storage (**thread_local**). The standard library offers a family of atomic types, useful in the context of lock-free programming, as well as, a set of portable abstractions for concurrency (**thread**, **mutex**, **Lock**, **packaged_task**, **future**).

3.2. Library overview

3.2.1. Threads

La abstracción de hilo de ejecución se ofrece a través de la clase **std::thread** y representa a un hilo ofrecido por la plataforma. Como en cualquier solución de concurrencia, dos hilos pueden acceder a un objeto compartido, lo que podría dar lugar a una carrera de datos.

Execution thread abstraction is provided through the **std::thread** class and represents a thread offered by the platform. As in any concurrency solution, two threads can access a shared object, which could lead to a data race.

In C++11, the threads provide a simplified argument passing mechanism without having to perform any type of conversions (*casts*).

In general, a thread can be constructed from any invocable object. This includes functions and function objects as well as lambda expressions.

3.2.2. Access to shared data

A **std::mutex** is a type that allows controlling access with mutual exclusion to a shared resource. It provides two basic acquisition operations: (**lock**) and (**unlock**). To avoid possible problems, such as releasing miss or the problems derived from exceptions, a wrapper (**unique_lock**) is available. This operation frees the lock on the destruction.

Another classic problem solved by the library is the acquisition of multiple locks. In this case, the function **lock()** takes an arbitrary number of locks, allowing the acquisition of all in a single operation.

3.2.3. Waits

The standard library provides mechanisms for accessing various clocks. Of these, the **high_resolution_clock** type is the highest resolution available, deserves special attention. The differences between two temporal points (time differences) can be expressed in different units that can be explicitly expressed .

The **sleep_for()** function allows you to specify a wait for a certain amount of time. Note that this function is in the nested namespace **std::this_thread**.

A condition variable is a mechanism that allows you to synchronize threads that access shared resources. A condition variable allows you to specify that a thread waits for notification. The wait is associated with a **mutex**. The condition variable also provides two notification operations (**notify_one()** and **notify_all()**.

3.2.4. Asynchronous tasks

An asynchronous task allows the simple deployment of the execution of a task, either in another thread or as a deferred task. When an asynchronous task is invoked, a *future* is obtained, which is an object that allows to return a value transporting it from one thread to another.

3.3. std::thread class

The **std::thread** class, introduced above, represents the thread abstraction offered by the platform (either the hardware or the operating system). All threads that are created within a program share the same address space which simplifies memory sharing.

It is important to note that each thread that is created has its own memory stack. This poses potential dangers to programs. On the other hand, if you pass a pointer or a non-constant reference to another thread, access is given to the stack of the original thread. Also, passing a reference through a capture of a lambda expression is giving access to the stack of the original thread. Why is this a problem? If the original thread terminates, its stack is freed and this memory could be assigned to another object, but other threads would still have references to that memory.

Objects in the **std::thread** class can not be copied. However they do support the movement semantics, which allows a thread to be transferred from one context to another.

There are multiple ways to build threads. In all cases, one of the characteristics is that when constructing the thread you can pass the arguments of the function to execute without having to do type conversions or make unnecessary use of pointers.

One technique that is often used is the two-phase construction. For this, a class defined by the constructor and the invocation operator is defined by each thread (**operator()**). In the first phase, the object is constructed. In the second phase, a thread is created by passing the created object. This technique is especially useful is complex structures of dependencies between threads.

Each thread has a unique identifier (obtained by **mythread.get_id()**. Although the type of the identifiers is a type defined by each implementation, the set of requirements that must satisfy allows the identifiers can be stored in a memory data structure.

When it is desired to wait for the termination of a thread, the operation **join()** must be invoked. If a thread object is destroyed without being invoked for it, the **join()** operation fails, and the library invokes the **terminate()** function. There are different alternatives to this solution. One possibility is to define a type of specialized thread whose destructor invokes **join()** whenever necessary. Another solution is the use of unrelated threads. However this solution is usually indicated only in cases of threads that act as demons.

3.4. *Mutex* objects and condition variables

The C++11 standard library offers multiple types of **mutex**. The simplest type is **std::mutex**. If the object needs to be acquired more than once by the same thread (e.g. recursive functions) you can use **std::recursive_mutex** instead. If it is necessary to use time-limit operations, **std::timed_mutex** can be used. The properties of the latter two types are combined in **std::timed_recursive_mutex**.

The **lock()** and **unlock()** operations allow blocking and unlocking a **mutex**. In cases where the blocking acquisition is not guaranteed, you can use **try_lock()**, which attempts to acquire the object and returns a success/failure flag.

In the case of **std::timed_mutex** operations, we can add additional operations to acquire a **mutex**, indicating a time period in the form of duration (**try_lock_for(dur)**) or in the form of a temporary point (**try_lock_until(t)**).

Complementing these types, the library offers condition variables. The type **std::condition_variable** is optimized for use with **std::mutex**. In another case, the type **std::condition_variable_any** can be used. It should be remembered that before destroying a variable condition, it should wake up all the threads that are waiting in it or it runs the risk of generating a deadlock.

There are two notification operations on a condition variable. The **notify()** operation wakes up one of the threads waiting in it. The **notify_all()** operation wakes up all the threads that are waiting.

Wait operations always take as an argument a mutex. In this way, a **wait()** operation locks until the last lock is acquired, passed as argument. You can specify a timeout as the second argument (**wait_for()** or **wait_until()**.

4. Memory consistency model in C++

This lesson has the following general structure:

1. Memory model.
2. Atomic types.
3. Ordering relations.
4. Consistency models.

5. Barriers.

4.1. Memory model

As previously mentioned, C++11 defines a concurrency model as part of the language. The ultimate goal is to avoid the need to write code in lower-level languages for improved performance. In addition, the library also incorporates a set of low-level synchronization operations.

A basic definition of the memory model is related to *object*. From this point of view, an *object* is a storage region. That is, it is a sequence of one or more bytes. A *memory location* is an object of a scalar type or a sequence of adjacent bit fields. With this in mind, it can be said that an object of any type is always stored in one or more memory locations.

The definition of *memory location* is important because the conditions that may eventually give rise to a data race are defined in terms of the concept of memory location itself.

Thus, if two threads try to simultaneously access the same memory location and some of the accesses is writing, there is a potential race condition. To prevent this race condition, it is necessary to force an order between the memory access operations.

To force an order in the operations accessing to memory, a solution is to use high-level synchronization mechanisms. An alternative that may be in some cases more efficient, although more complex, is the use of atomic operations.

4.2. Atomic types

Atomic operations are indivisible operations. Therefore, if two threads perform read or write operations over the same variable, such operations are performed in some order and do not produce data races. However, if any of the operations are not atomic, the behaviour is not defined.

The C++ library provides access to atomic operations through a set of types (**atomic<T>**). Atomic operations are offered for integral types, pointers, and booleans. However, they are not offered for floating-point types.

Operations on atomic types support the specification of the memory consistency model. By default, the sequential consistency model is used, but more relaxed consistency models can be specified.

4.3. Ordering relations

There are two relations that are used to reason about the behaviour of concurrent programs: *synchronizes-with* and *happens-before*.

The relation *synchronize-with* is a relation that occurs between operations performed on atomic types from different threads. On the other hand, the relation *happens-before* is a relation that occurs between operations occurring in the same thread.

4.4. Consistency models

The consistency model on which it is simpler is the sequential consistency, since in this case, the program is consistent with a sequential view. That is, all operations are performed in some particular order in a single thread. Although it is a model that simplifies the reasoning, it has a greater cost in performance.

In sequentially non-consistent models, there is no longer a global order of events and each thread can have a different view of the order in which they occur. Even so, all threads must agree on the order of modification of each variable. Among the non-consistent models are the relaxed consistency and the release-acquire consistency models.

Operations that use a relaxed consistency do not participate in *synchronize-with* relations, although if they participate in *happens-before* relations within the same thread. As a consequence, operations can not be reordered within the same thread, but nothing can be assured about the ordering of operations on different threads.

The operations that use the release-acquire consistency mechanism need an intermediate level between the sequential consistency and the relaxed consistency since they do not exactly establish the order if they are establishing restrictions.

An interesting property is that, in many cases, the effect of sequential consistency can be obtained by combining release-acquire consistency and relaxed consistency, whereby better performance can be obtained.

4.5. Barriers

Barriers are synchronization operations that establish ordering between memory accesses without modifying data. They are typically used to set restrictions on other accesses to atomic variables.