High Level Parallel Programming

Advanced Multi-Threding in C++

Alessandro Savino
Dipartimento di Automatica e Informatica
Politecnico di Torino

Introduction

- Sometimes we need to run asynchronous tasks producing output data that will become useful later
  - Thread objects are OK for that but what about getting a return value from the executed function?
- We saw the basics of thread-based parallel programming but in C++11 we can talk also about task-based parallel programming
  - It relies on a different constructs (no **std::thread** objects)
  - It enables the possibility of handling return values

Futures

- C++11 introduced class future to access values set values from specific providers
  - Definition
    - #include <future>
  - Providers
    - calls to async(), objects promise<> and packaged_task<>
  - Providers set the shared state to ready when the value is set

Future

- A future is an object that can retrieve a value from some provider object or function, properly synchronizing this access if in different threads
  - Calling **future::get** on a valid future blocks the thread until the provider makes the shared state ready (either by setting a value or an exception to it)
- future::get can only be called once on a future
  - future has move semantics

```
future<T> <future_name>;
```

T is the type of the future

Promise

- A promise is an object that can store a value to be retrieved by a future object (possibly in another thread)
- On construction, promise objects are associated to a new shared state on which they can store a value of type **T**
- This shared state can be associated to a future object by calling member **get_future**

Promise

- After the call, both objects share the same shared state
  - The promise object is the asynchronous provider and is expected to set a value for the shared state at some point
  - The future object is an asynchronous return object that can retrieve the value of the shared state, waiting for it to be ready, if necessary
- A promise can store an exception too (which is not possible using threads only)
  - set_exception() instead of set_value()
  - Exception will be fired when the get() on the future is called

Futures & Promises

```
#include <future>
using namespace std;

void factorial(const int &N, promise<int>& pr) {
   int res = 1;
   for ( int i=N; i> 1; i-- )
   res *=i;
   pr.set_value(res);
}

int main () {
   promise<int> p;
   future<int> f = p.get_future();
   thread t = thread(factorial, 4, ref(p));
   // here we have the data
   int x = f.get();
   t.join();
}
```

ref generates an object of type promise<int> to hold a reference to p.

Futures Providers

- C++11 introduced function async (namespace std)
- Definition
  - #include <future>
- Higher level alternative to std::thread to execute functions in parallel

```
future<T> async(launch_policy, function, args...);
```

T is the type of the future

Three different launch policies for spawning the task

Futures Providers: Policies

Policy	Description
launch::async	Asynchronous: launch of a new thread to call function
launch::deferred	The call to function is deferred until the shared state of the future is accessed (call to **wait** or **get**)
launch::async | launch::deferred	System and library implementation dependent. Choose the policy according to the current availability of concurrency in the system.

Future Providers

```
#include <future>
#include <iostream>

// function to check if a number is prime
bool is_prime (int x) { ... }

int main () {
  std::future<bool> fut = std::async(
    std::launch::async, is_prime, 117);
  // ... do other work ...
  bool ret = fut.get();
  // waits for is_prime to return
  return 0;
}
```

Future Providers: A better Factorial

```
#include <future>
using namespace std;

int factorial( std::future<int>& f ) {
   int res = 1;
   int N = f.get();
   for ( int i=N; i> 1; i-- )
   res *=i;
   return res;
}

int main () {
   std::promise<int> p;
   std::future<int> f = p.get_future();
   std::future<int> fu = async(std::launch::async,
                              factorial, std::ref(f));
   // here we have the data
   p.set_value(4);
   int x = fu.get();
}
```

future must be passed by reference, since it doesn't support copy semantics

Future Providers: Thread Communication

```
usign namespace std;

void func1( promise<int> p ) {
   int res = 18;
   p.set_value(res);
}

int func2 ( future<int> f) {
   int res=f.get();
   return res;
}

int main () {
   promise<int> p;
   future<int> f = p.get_future();
   future<void> fu1 = async(func1, move(p) );
   future<int> fu2 = async(func2, move(f) );
   int x = fu2.get();
   return 0;
}
```

the move semantics is achieved by std::move ...

Futures Providers

- C++11 introduced a further facility, the class std::packaged_task<>
- This class wraps a callable element (e.g., a function pointer) and allows to retrieve asynchronously its return value

```
std::packaged_task<function_type> tsk(args);
```

Future Providers

```
#include <future>
...
using namespace std;

int compute_double(int value) { return value*2; }

int main() {
   packaged_task<int(int)> tsk(compute_double);
   future<int> fut = tsk.get_future();
   tsk(1979);
   int r_value = fut.get();
   cout << "Output: " << r_value << endl;
   return 0;
}
```

Future Providers

```
#include <future>
#include <thread>
...
using namespace std;

int compute_double(int value) { return value*2; }

int main() {
   packaged_task<int(int)> tsk(compute_double);
   future<int> fut = tsk.get_future();
   thread th(std::move(tsk), 1979);
   int r_value = fut.get();
   cout << "Output: " << r_value << endl; th.join();
   return 0;
}
```

Shared Future

- A shared_future object behaves like a future object, except that it can be copied
- More than one shared_future can share ownership over their end of a shared state
- The value in the shared state can be retrieved multiple times once ready

Shared Future

```
usign namespace std;

int factorial( shared_future<int> f ) {
  int res = 1;
  int N = f.get();
  for ( int i=N; i> 1; i--)
    res *=i;
  return res;
}

int main () {
  promise<int> p;
  future<int> f = p.get_future();
  shared_future<int> sf = f.share();

  future<int> fu = async(std::launch::async, factorial, sf);
  future<int> fu2 = async(std::launch::async,factorial, sf);
  future<int> fu3 = async(std::launch::async, factorial, sf);
  p.set_value(4);
  int x = fu.get();
  return 0;
}
```

Task-based vs thread-based approaches

- Task-based approaches
  - Functions return value accessible
  - Smart task/thread spawning with default policy
    - CPU load balancing
      - The C++ library can run the function without spawning a thread
    - Avoid the raising of **std::system_error** in case of thread number reached the system limit
  - Future objects allows us to catch exceptions thrown by the function
    - While with **std::thread()** the program terminates

Task-based vs thread-based approaches

- Thread-based approaches
  - Used to execute tasks that do not terminate till the end of the application
    - A thread entry point function is like a second, concurrent **main**
  - More general concurrency model, can be used for thread-based design patterns
  - Allows us to access to the pthread native handle
    - Useful for advanced management (priority, affinity, scheduling policies, etc.)