Concepts and Models of Parallel and Data-centric Programming

Shared Memory II

Lecture, Summer 2020

Dr. Christian Terboven <terboven@itc.rwth-aachen.de>

i12 High Performance Computing

RWTH AACHEN UNIVERSITY

Outline

0. Organization
1. Foundations
2. **Shared Memory**
3. GPU Programming
4. Bulk-Synchronous Parallelism
5. Message Passing
6. Distributed Shared Memory
7. Parallel Algorithms
8. Parallel I/O
9. MapReduce
10. Apache Spark

a. Processes and Threads
b. Threading in C++
c. RAII idiom, Move Semantics
d. Mutual Exclusion
e. Condition Variable
f. Example: Queue

Threading in C++: Basics

Concurrency vs. Parallelism

- To many, these terms mean the same
  - Difference in focus and intent
- Both terms: running multiple tasks simultaneously
- Parallelism: performance-oriented
  - Taking advantage of hardware to increase performance
- Concurrency: separation of concern
  - Responsiveness

Threads – a programmer's view

- Execution stream within a program
  - Multiple threads working together may deliver a speedup
    - Programmer is responsible to distribute work
    - Programmer is responsible to manage the threads
- You can tell a thread:
  - What to do
  - When to start
- You can:
  - Wait for it to finish
- Other stuff (not relevant here):
  - Interrupt it, give it priority, etc.

Threads in C++ / 1

- Every C++ program has at least one thread running `main()`
  - Started by the C++ runtime
  - Additional threads run concurrently with each other, and the initial one

- Class `std::thread`
  - Representation of a “system thread”
  - Each thread has a function which is executed when it starts
  - Thread vanishes when the function returns
  - Defined in header `<thread>`
  - Reference: https://en.cppreference.com/w/cpp/thread/thread

Threads in C++ / 2

- After a thread has been started, the main thread could
  - wait for it to finish
    - `std::thread::join()`
    - blocks until the thread has finished execution
  - detach it
    - `std::thread::detach()`
    - permits the thread to execute independently from the thread handle
- Has to be handled before `std::thread` instance is destroyed
  - otherwise the `std::thread` destructor calls `std::terminate()`
- Data access by a thread has to be valid until it finishes

Threads in C++ / 3

- Thread identifiers are of type `std::thread::id`
- Obtained via
  - `get_id()` member function of a `std::thread` object
  - Current thread: `std::this_thread::get_id()`

- The id can be used to direct the control flow / divide work

```
std::thread::id master_thread;
/* ... code ... */

if (std::this_thread::get_id() == master_thread)
   do_master_work();
do_common_work();
```

Examples

Starting a thread / 1

- Starting a simple function as a thread

```
void do_some_work();
std::thread my_thread(do_some_work);
```

- Starting a lambda expression as a thread

```
std::thread my_thread([]{
   do_something();
   do_something_else();
});
```

Review: Lambdas in C++

- A lambda expression constructs a closure
  - unnamed function object capable of capturing variables in scope
  - Reference: https://en.cppreference.com/w/cpp/language/lambda
- Most commonly used syntax:
  - [*captures*] (*params*) { *body* }
  - [*captures*] { *body* }
  - return type can be derived from the return statement, or void if none
- Capture defaults:
  - &: capture by reference
  - =: capture by copy

Starting a thread / 2

- Using the ()-operator of a struct or class

```
class background_task
{
public:
    void operator() () const
    {
        do_something();
        do_something_else();
    }
};

background_task f;
std::thread my_thread(f);
```

Starting a thread / 2

- Using the `()`-operator of a struct or class

```
class background_task
{
public:
    void operator() () const
    {
        do_something();
        do_something_else();
    }
};

background_task f;
std::thread my_thread(f);
```

Function call operator

Possible error: undefined behavior

- Access to local (stack) data of the initial thread

```
struct func
{
   int& i;
   func(int& i_) : i(i_) {}
   void operator() ()
   {
      for (unsigned j = 0; j < 1000000; ++j)
         do_something(i);
   }
};

void oops()
{
   int some_local_var = 0;
   func my_func(some_local_var);
   std::thread my_thread(my_func);
   my_thread.detach();
}
```

Possible error: undefined behavior

- Access to local (stack) data of the initial thread

```cpp
struct func
{
    int& i;
    func(int& i_) : i(i_) {}
    void operator() ()
    {
        for (unsigned j = 0; j < 1000000; ++j)
            do_something(i);
    }
};

void oops()
{
    int some_local_var = 0;
    func my_func(some_local_var);
    std::thread my_thread(my_func);
    my_thread.detach();
}
```

Thread might still be running

Possible error: undefined behavior

- Access to local (stack) data of the initial thread

```
struct func
{
    int& i;
    func(int& i_) : i(i_) {}
    void operator() ()
    {
        for (unsigned j = 0; j < 1000000; ++j)
            do_something(i);
    }
};

void oops()
{
    int some_local_var = 0;
    func my_func(some_local_var);
    std::thread my_thread(my_func);
    my_thread.detach();
}
```

Potential access to dangling ref.

Thread might still be running

RAII idiom, Move Semantics

RAII idiom

- RAII: Resource Acquisition Is Initialization
  - binds the life cycle of a resource to the lifetime of an object
  - resource availability is a class invariant
    - guarantees the release of the resource in correct order

- Implementation of RAII:
  - encapsulate each resource into a class
    - constructor acquires the resource and establishes all class invariants or throws an exception if that cannot be done,
    - destructor releases the resource and never throws exceptions;
  - always use the resource via an instance of a RAII-class
    - with automatic storage duration or temporary lifetime

Using RAII to wait for completion

```cpp
class thread_guard
{
   std::thread& t;
public:
   explicit thread_guard(std::thread& t_): t(t_) {}
   ~thread_guard()
   {
      if (t.joinable())
         t.join();
   }
   thread_guard(thread_guard const&) = delete;
   thread_guard& operator= (thread_guard const&) = delete;
};

void f()
{
   int some_local_var = 0;
   func my_func(some_local_var);
   std::thread t(my_func);
   thread_guard g(t);
}
```

Using RAII to wait for completion

```
class thread_guard
{
   std::thread& t;
public:
   explicit thread_guard(std::thread& t_): t(t_) {}
   ~thread_guard()
   {
      if (t.joinable())
         t.join();
   }
   thread_guard(thread_guard const&) = delete;
   thread_guard& operator= (thread_guard const&) = delete;
};

void f()
{
   int some_local_var = 0;
   func my_func(some_local_var);
   std::thread t(my_func);
   thread_guard g(t);
}
```

See above

Using RAII to wait for completion

```
class thread_guard
{
   std::thread& t;
public:
   explicit thread_guard(std::thread& t_): t(t_) {}
   ~thread_guard()
   {
      if (t.joinable())
         t.join();
   }
   thread_guard(thread_guard const&) = delete;
   thread_guard& operator= (thread_guard const&) = delete;
};

void f()
{
   int some_local_var = 0;
   func my_func(some_local_var);
   std::thread t(my_func);
   thread_guard g(t);
}
```

See above

Destructor will join if possible and necessary

Using RAII to wait for completion

```cpp
class thread_guard
{
    std::thread& t;
public:
    explicit thread_guard(std::thread& t_): t(t_) {}
    ~thread_guard()
    {
        if (t.joinable())
            t.join();
    }
    thread_guard(thread_guard const&) = delete;
    thread_guard& operator= (thread_guard const&) = delete;
};

void f()
{
    int some_local_var = 0;
    func my_func(some_local_var);
    std::thread t(my_func);
    thread_guard g(t);
}
```

See above

Destructor will join if possible and necessary

g gets out of scope, destructor is called

C++ move semantics

- rvalue, lvalue, and &&
  - An *lvalue* is an expression whose address can be taken. Anything you can make assignments to is an *lvalue*
  - An *rvalue* is an unnamed value that exists only during the evaluation of an expression.
  - The && operator is like the reference operator (&), but whereas the & operator can only be used on *lvalues*, the && operator can only be used on *rvalues*.

- It is possible to do a move (rather than a copy) if:
  - the object is an *rvalue*
  - the object's class defines the *special member move functions*
    - move constructor and move assignment operator

Ownership of threads

- `std::thread` employs the move semantic
  - moveable, but not copyable
  - because it is resource-owning
- Example:

```cpp
void func1();
void func2();
std::thread t1(f1);
std::thread t2 = std::move(t1);
t1 = std::thread(f2);
std::thread t3;
t3 = std::move(t2);
t1 = std::move(t3);
```

Ownership of threads

- `std::thread` employs the move semantic
  - moveable, but not copyable
  - because it is resource-owning
- Example:

```
void func1();
void func2();
std::thread t1(f1);
std::thread t2 = std::move(t1);
t1 = std::thread(f2);
std::thread t3;
t3 = std::move(t2);
t1 = std::move(t3);
```

Program will be terminated, as t1 owns a thread