11 Parallel and Distributed Programming

Before C++11, there was no concept of parallelism or multithreading within the language or the STL. Hence, programmers had to rely on additional (platform-specific) libraries for parallel programming. We look at one of the most commonly used libraries, POSIX threads, in Section 11.2 on page 225. C++11 finally introduced multithreading into the language standard. We cover some core features in Section 11.3 on page 232. However, within this course we can only provide a brief introduction to multithreaded programming; for further details, such as the design of thread-safe data structures and interfaces, as well as implementation and debugging details, you have to consult a more comprehensive reference, such as (Williams 2012).

Multithreading is not the only way to write parallel programs. Especially in web-scale data processing, software is now commonly run on clusters of hundreds or even thousand of computers. We look at one popular way for developing programs that can make use of such clusters, called MapReduce, in Section 11.4 on page 247.

A Warning. All code shown in this lecture is highly dependent on the platform, compiler, and libraries used. Proceed with caution.

11.1 Foundations

The first computers performed only one task at a time. After a few years of development, however, people became concerned about the waste involved in keeping a very expensive machine idle, even if the idle periods were only the milliseconds required to read a punched card or paper tape. 416

Concurrency, called *multitasking*, was first provided as a feature of operating systems, enabling the processor to switch tasks when a task became "blocked", for example, while performing input or output. The name *process* was introduced to describe a task that might or might not be running at a given time. multitasking

The theory of *cooperating sequential processes* was worked out by Edsger Dijkstra, Tony Hoare, and Per Brinch Hansen. Dijkstra introduced *semaphores* as a low-level synchronization primitive, Hoare and Brinch Hansen added *monitors* for higher-level control of synchronization. semaphores monitors

Since that time, operating systems have provided safe multitasking, usually based on some version of the basic operations worked out by Dijkstra, Hoare, and Brinch Hansen. Unfortunately, the same cannot be said for languages. The early work showed that safe concurrency could be enforced only by the compiler; it is not possible to provide safe functions in a library.

Before going further, we need some definitions. *Sequential code* executes in a fixed order determined by control structures. For example, s_1; s_2; … s_n means "execute s_1, then execute process

s_2, and so on, until s_n". A *process* is the execution of sequential code with interruptions. For example, "execute s_1, then go off and do something else for a while, execute s_2, and so on". A process uses the processor's *registers*, has its own *run-time stack*, and uses the memory-protection facilities of the processor to prevent code from accessing memory outside the area allocated to the process.

thread

A *thread* is like a process, but does not use the processor's memory-protection features. Typically, threads run within a process, and the process is memory-protected. Since changing memory bounds is typically an expensive operation, and control can be transferred between threads without changing bounds, threads are considered *light-weight* and processes are considered *heavy-weight*.

deadlock

A multi-threaded program may *deadlock*, meaning that each thread or process is waiting for
417 something to happen. Here is a simple deadlock scenario:

1. Threads T_1 and T_2 both require resources R_1 and R_2.
2. T_1 acquires R_1.
3. T_2 acquires R_2.
4. T_1 requests R_2, fails, and blocks.
5. T_2 requests R_1, fails, and blocks.

starvation

A thread, or group of threads, may monopolize the processor, leading to *starvation* of other threads.

livelock

A group of threads may get into a cyclic set of states in which the threads continue to run but make no progress: this is called *livelock*. A real-life analogy is the situation in which you walk towards someone and attempt to avoid collision by moving left ... but the other person moves
418 to its right ... so you move right ...

distributed computing

#BigData

Multi-threaded programming is notoriously difficult to get right. Hence, in many application scenarios, a different form of parallelism is used: *distributed computing*. In this paradigm, multiple processes work on a task in parallel, but do not use shared memory for communication. Instead, a form of *message passing* is used to initiate tasks and share results. This form of parallelism is extensively used in large-scale data processing, in particular for Web data, which can easily reach several (hundreds of) terabytes. Examples for distributed systems are telecommunication systems, (wireless) sensor networks, and cluster algorithms. Nowadays, application examples for large-scale data analysis are commonly grouped under the "Big Data" keyword.

11.1.1 Thread-Safe Programming

Above, we have referred to "safe" concurrency. What does this mean? What does the (incomplete)
419 program in Figure 136 on the facing page print assuming that both threads run to completion?

race conditions

If we run this program many times, we will find that it usually prints `"1111"` but that it sometimes replaces one or two `1`s by `0`s: e.g., `"1110"` or `"1011"` but never `"0100"` or `"0000"`. The program has what is known as a *race condition*, and its results depend on the exact way in which the operations of the two threads are interleaved.

```
                  struct { char c1; char c2; char c3; char c4 } s;
                  s.c1 = s.c2 = s.c3 = s.c4 = 0;


            // thread 1                                  // thread 2
            ++s.c1;                                      ++s.c3;
            ++s.c2;                                      ++s.c4;


                  cout << s.c1 << s.c2 << s.c3 << s.c4 << endl;
```

Figure 136: Fragments of a multithreaded program

The pioneers recognized that race conditions can occur only when there are variables that can be accessed by two or more threads concurrently. Let's call these *shared variables*. When a thread is manipulating a shared variable, it must be protected from interruption by another thread. A chunk of code that manipulates a shared variable is called a *critical region*. Synchronization primitives ensure that a thread can execute its critical regions without interruption.

shared variables

critical region

A concurrent system is *pre-emptive* if a process can be interrupted at any time; it is *non-pre-emptive* (or *co-operative*) if processes choose when they can be interrupted (usually by periodically calling a system function with a name like `synchronize` or `coordinate`). Programming non-pre-emptive systems is easier, but programmers must obey the discipline of giving up control at frequent intervals; failure to do so locks up the system (e.g., older versions of Windows). Programming pre-emptive systems is more difficult, because any process can be interrupted at any time, but has the advantage that no process can monopolize the system.

pre-emptive vs. co-operative multithreading

11.2 The POSIX Thread Library

Threads are not mentioned in either the C standard nor the C++98 standard. However, there have always been compiler-specific thread libraries. One of the best known is the cross-platform ***POSIX threads*** standard, often abbreviated to ***Pthreads***. The POSIX standard[48] used to be in the public domain but has now been taken over by IEEE, which means that you have to pay for the standard. POSIX threads library components are usually declared in `threads.h`, which is available on many UNIX/LINUX systems, as well as Mac OS X. On Windows, you can use *Open Source POSIX Threads for Win32*.[49] Figure 137 on the next page shows a simple threaded program. Note the declaration of thread objects of type `pthread_t` and the use of `pthread_create` to create the actual thread. The call to `pthread-exit` is needed to prevent the main program from terminating before the threads have finished their work.

Pthreads

420

421

The function `counter` defines the effect of the thread. It must have one parameter of type `void*` and must return a result of type `void*`. In the example, this parameter is used only to identify the thread; in a more practical example, it might be used to make threads perform distinct actions.

When this program is run, it prints the following (line breaks have been omitted to save space): 422

[48]POSIX is an acronym for "Portable Operating System Interface", see https://en.wikipedia.org/wiki/POSIX
[49]Pthreads Win32, http://sourceware.org/pthreads-win32/

```
void *counter(void *p)
{
    cout << "Starting thread " << *((int*)p) << endl;
    for (int i = 0; i < 10; ++i)
    {
        cout << i;
    }
    return 0;
}

void main()
{
    cout << "Creating threads ..." << endl;
    static int n1 = 1;
    pthread_t t1;
    pthread_create(&t1, 0, counter, &n1);
    static int n2 = 0;
    pthread_t t2;
    pthread_create(&t2, 0, counter, &n2);
    cout << "Waiting for threads to complete." << endl;
    pthread_exit(0);
}
```

Figure 137: Adding numbers with threads

```
Creating threads ...
Waiting for threads to complete.
Starting thread 1
0 1 2 3 4 5 6 7 8 9
Starting thread 0
0 1 2 3 4 5 6 7 8 9
```

From this output, we infer:

- The main program runs to `pthread_exit` before the threads are executed.
- Once started, a thread does not give up control until it has finished its task.

In a practical application, we would probably want threads to relinquish control from time to time so that other threads could do something. Adding one line to the function `counter` (the call to `sched_yield`) is all that we need: see Figure 138 on the facing page. 423

With this change, the program prints the following output, showing that control alternates between the threads: 424

```
void *counter(void *p)
{
    cout << "Starting thread " << *((int*)p) << endl;
    for (int i = 0; i < 10; ++i)
    {
        cout << i;
        sched_yield();
    }
    return 0;
}
```

Figure 138: Co-operative multithreading with POSIX threads

```
Creating threads ...
Starting thread 1
0
Waiting for threads to complete.
1
Starting thread 0
0 2 1 3 2 4 3 5 4 6 5 7 6 8 7 9 8 9
```

A thread library typically provides *mutex*[50] objects for synchronizing threads. If threads are pre-emptive, mutex operations cannot be coded in a high-level language because they depend on special, non-interruptible instructions provided by all modern processors. A typical instruction for mutual exclusion is test-and-set, which checks, in a single atomic (non-interruptible) operation, the value in a memory location, sets it to a new value, and returns the old value. mutex test-and-set

POSIX threads are usually implemented without pre-emption. Consequently, the library functions can be written entirely in C, without the need for special machine instructions. The following table shows the mutex types and operations provided by POSIX threads. 425

Identifier	Description
`pthread_mutex_t`	Type of mutex object
`pthread_mutex_init(pthread_mutex*, ...)`	Initialize a mutex object
`pthread_mutex_destroy(pthread_mutex*)`	Destroy a mutex object
`pthread_mutex_lock(pthread_mutex*)`	Lock using a mutex
`pthread_mutex_unlock(pthread_mutex*)`	Unlock using a mutex

11.2.1 Encapsulating Threads

In these section, we will show how to use the low-level, C-style POSIX operations in a C++ environment, using ideas from the Appendix of Alexandrescu's book *Modern C++ Design* (Alexandrescu 2001) (and also http://www.informit.com/articles/article.asp?p=25298&rl=1

[50] Mutex is short for "mutual exclusion".

29/11/05). The first step is to encapsulate mutual exclusion in a base class, as shown in
426 Figure 139. The second step is to use RAII to ensure that locking is secure, as shown in
427 Figure 140.

```
class Synchronized
{
public:
    Synchronized()
    {
        pthread_mutex_init(&mtx, 0);
    }

    void acquireMutex()
    {
        pthread_mutex_lock (&mtx);
    }

    void releaseMutex()
    {
        pthread_mutex_unlock (&mtx);
    }

private:
    pthread_mutex_t mtx;
};
```

Figure 139: A base class for synchronization

```
template<typename T>
class Lock
{
public:
    Lock(T & client) : client(client)
    {
        client.acquireMutex();
    }

    ~Lock()
    {
        client.releaseMutex();
    }

private:
    T client;
};
```

Figure 140: Using RAII to simplify locking

A class T that requires locking inherits from class Synchronized and uses class Lock<T> to obtain and release locks. Each critical operation constructs a local instance of Lock<T> to ensure that critical sections are run without interruption. At the end of the critical section, RAII ensures that the lock is automatically released. Figure 141 shows a simple account class implemented in 428
this way. 429

```
class Account1 : public Synchronized
{
public:
    Account1() : balance(0) {}

    void deposit(int amount)
    {
        Lock<Account1> guard(*this);
        balance += amount;
    }

    void withdraw(int amount)
    {
        Lock<Account1> guard(*this);
        balance -= amount;
    }

    friend ostream & operator<<(ostream & os, const Account1 & acc)
    {
        return os << acc.balance;
    }

private:
    int balance;
};
```

Figure 141: An account class with internal locking

Class Account1 uses *internal locking*. There are two other possibilities, and all three are defined below: 430

Internal Locking: Each class ensures that concurrent calls cannot corrupt the class. The usual way to do this is to place a lock on any access to the class's data, as in class Account.

External Locking: Classes guarantee to synchronize access to static or global data but do not protect their instance variables. Clients must lock all invocations of their member functions.

Caller-ensured Locking: The class has a Mutex object (like class Synchronized) but does not manipulate it. Instead, clients are expected to do the locking: 431

```
Account1 acc;
....
Lock<Account1> guard(acc);
acc.deposit(50000);
```

Unfortunately, all three methods have disadvantages. Suppose we use internal locking for an ATM which charges a fee of $5 for each withdrawal. Suppose further that the design requires that there must be no transaction between withdrawing the amount and charging the fee. The following code does not satisfy this requirement because it might be interrupted between the two withdrawals:

```
void ATMwithdraw(Account1 acc, int amount)
{
    acc.withdraw(amount);
    acc.withdraw(5);
}
```

The obvious correction is to apply a lock that holds over both calls:

```
void ATMwithdraw(Account1 acc, int amount)
{
    Lock<Account1> guard(acc);
    acc.withdraw(amount);
    acc.withdraw(5);
}
```

This will work if the thread library supports *recursive locking* – a single thread can lock/unlock the same object more than once.[51] Nevertheless, it is inefficient because two levels of locking are unnecessary. In a large application, a single object might be locked many times before actually being used. Another possibility is that the thread library does not support recursive locking; in this case, the code above will deadlock when it attempts to lock the account twice.

Caller-ensured locking avoids this particular problem, but still requires discipline from clients. Here is the design problem: the class `Account` consists of a finite amount of code and is therefore *bounded*; the rest of the application uses `Account` and may be of any size – it is *unbounded*. With caller-ensured and external locking, we have to examine unbounded code to be sure that synchronization is correct. We have lost encapsulation. 432

433 Figure 142 on the next page shows how we can combine internal and external locking. Each function in the class comes in two flavours. The first version has a `Lock` parameter and is intended to be used with external locking: the client acquires a lock and passes it by reference to the function. The second version has no `Lock` parameter and implements internal locking by acquiring the lock itself. For a simple transaction, we rely on the internal lock: 434

```
Account2 acc;
acc.deposit(500);
```

[51] Although POSIX threads support recursive locking, some people, including POSIX implementors, argue against its use.

```
class Account2 : public Synchronized
{
public:
    Account2() : balance(0) {}

    void deposit(int amount, Lock<Account2>&)
    {
        balance += amount;
    }

    void deposit(int amount)
    {
        Lock<Account2> guard(*this);
        balance += amount;
    }

    void withdraw(int amount, Lock<Account2>&)
    {
        balance -= amount;
    }

    void withdraw(int amount)
    {
        Lock<Account2> guard(*this);
        balance -= amount;
    }

    friend ostream & operator<<(ostream & os, const Account2 & acc)
    {
        return os << acc.balance;
    }

private:
    int balance;
};
```

Figure 142: The best of both worlds

For a more complex transaction, we use external locking:

```
void ATMwithdraw(Account2 acc, int amount)
{
    Lock<Account2> guard(acc);
    acc.withdraw(amount, guard);
    acc.withdraw(5, guard);
}
```

11.2.2 A Better `Lock`

435 436 Alexandrescu also shows how to make better locks by exploiting C++ features. Figure 143 shows his class `Lock`. The new kind of `Lock` must be a stack-allocated variable and must be associated with a particular object. `Lock`s cannot be passed by value to functions or returned from functions. It is possible to pass `Lock`s by reference because no copy is made. Disabling the address operator makes it hard (but not impossible) to create aliases of a `Lock`. The result is that if you own a `Lock<T>` then you have in fact locked a `T` object and that object will eventually be unlocked.

11.3 Multithreading in C++11

C++11 added multithreading as a major new feature to the language. Rather than relying on compiler- and platform-specific libraries, it is now possible to write portable code involving multithreading. Within this course, we can only cover some parts of the new C++11 standard. The reference book (Stroustrup 2013, Chapter 41) covers all new language features, but for a more programming-oriented discussion, the book (Williams 2012) is recommended. An overview is also available in (Josuttis 2012, Chapter 18).

11.3.1 Hello, C++11 Thread!

437 Figure 144 on page 234 shows a minimal example for a multithreaded program using the C++11 standard thread library. Important new features are:

- We import the standard thread library header, with `#include <thread>`
- Our “hello” code has been moved to a function. This is necessary, because each thread is started with a function. For single-threaded programs, this has always been `main()`, but for any new threads we create, we have to provide the starting function explicitly.
- In the main program, we create a new `std::thread t` and tell it to start the function `hello()`. At this point in the code, there are now two threads running: one for the `main()` program, one executing `hello()`.
- Finally, we synchronize our two threads, by telling the main program thread to wait for thread `t`, using `t.join()`. Without this statement, the main program could finish execution (thus ending the program), before thread `t` had a chance to write any output.

Running this program requires (i) a compiler with support for C++11 threads and (ii) the availability of a multi-threading library for your specific platform. GNU `gcc` started adding multithreading support in version 4.3,[52] but you might have to explicitly enable C++11 features with the option `-std=c++11` in case your compiler defaults to C++98. You probably also have to tell it which threading library to use on your platform; on Linux, this is usually `pthreads`. Thus, to compile the program you would run:

```
$ g++ -std=c++11 hellothread.cpp -l pthread
```

If your compiler coughs up something like `error: thread is not a member of std`, then your implementation does not support C++11 threads.

[52] For details on supported C++ language versions in gcc, see https://gcc.gnu.org/projects/cxx-status.html.

```
template <class T>
class Lock
{
public:
    Lock(T & client) : client(client)
    {
        client.acquireMutex();
    }

    ~Lock()
    {
        client.releaseMutex();
    }

private:

    // The user of the lock
    T client;

    // no default constructor
    Lock();

    // no copy constructor
    Lock(const Lock &);

    // no assignment
    Lock & operator=(const Lock &);

    // no heap allocation of individual objects
    void *operator new(std::size_t);

    // no heap allocation of arrays
    void *operator new[](std::size_t);

    // no destruction of heap objects
    void operator delete(void *);

    // no destruction of heap arrays
    void operator delete[](void *);

    // no address taking
    Lock * operator&();
};
```

Figure 143: A lock with restricted uses

```
#include <iostream>
#include <thread>

void hello()
{
  std::cout << "Hello, thread!"  << std::endl;
}

int main()
{
    std::thread t(hello);
    t.join();
}
```

Figure 144: Hello, C++11 Thread!

11.3.2 Working with threads

Let's look at how to work with threads in more detail. Most operations are managed through an object of the `std::thread` class. In the example in Figure 144 above, we've already seen how to start a new thread, by defining a `std::thread` object `t` and passing it a function to execute: 438

```
std::thread t(hello);
```

You might be tempted to write

```
std::thread t(hello());
```

since `hello()` is a function, but the compiler will read this completely differently: this declares (but does not define) a new function `t` that takes a single parameter (of type 'pointer to function') and returns a `std::thread` object.

Of course, instead of passing a function, you can also pass a function object. In this case, the object will be copied to the thread's local stack space, so you must ensure the copy constructor of your class behaves correctly.

Joining and detaching threads. You now have a new thread object (in this example `t`) that is associated with a running thread. It is created on the stack, and at one point will be deleted – either when the variable goes out of scope or an exception is thrown and the runtime system starts unwinding the stack. In either case, the destructor of the thread object `t` will be called. It is important that you instruct your thread what should happen in this case: otherwise your program will be terminated! You can either wait for the thread to finish using `join()` or ***detach*** the thread and let it run independently of your thread object `t`.

`join()` In cases where you need the results of a thread, you would typically use `join()`, since you cannot do anything useful until it completed its job. Once you `join`ed the thread, all memory associated with the thread will be cleaned up and the thread object will no longer be associated with a running thread. Consequently, you cannot call `join()` on the same thread more than once, so you might want to check your thread object to see if it holds a currently `joinable` thread: `joinable()` 439

```
    if (t.joinable())
    {
        t.join();
    }
```

The other option is to let your new thread run in the background, completely independent of your other threads. This is achieved by calling `detach()` on your thread object. Once detached, the thread is managed independently by the runtime environment and you can no longer access it through your thread object. The conditions for detaching a thread are the same as for joining a thread, so you can check whether a thread can be detached with `joinable()`. `detach()` 440

```
    t.detach();
    assert (!t.joinable());
```

Detached threads are useful when you have a specific task for each thread, such as opening multiple files in a GUI, where each file is managed by its own window, running in its own thread. You can also use them for background tasks that perform some periodic maintenance, like cleaning up unused files (in UNIX/LINUX, such background processes are known as ***daemon*** programs). daemon

To initiate background processes, you probably want to be able to pass arguments to the function executed by the thread. The thread library provides a simple way of doing this, as shown in Figure 145 on the next page. This program allows you to specify a sleep time in seconds and then creates a new thread that will wait the specified amount of seconds before waking up, issuing an alarm, and exit (of course, you could have a thread that does something more useful than sleeping). As you can see in the example, arguments are passed to a thread by providing them as additional arguments to the thread constructor (you can pass more than one argument this way). Note that these arguments are always copied; if you want to pass reference parameters you have to explicitly specify this in the thread constructor with `std::ref(arg)`. 441 442 `std::ref`

Threads and Exceptions. Once you detached a thread, you no longer have to worry about the thread object that was used to create the thread. But in case of `join`, you have to consider all possible ways the object `t` might get deleted. In case of an exception, the run-time system will start unwinding the stack, thus calling the destructor of `t`, which in turn will call `terminate()` and abort your program. You could try to avoid this by placing additional `t.join()` operations in each `catch` statement. But as shown before, a much better way is to use RAII – see Figure 146. Here, threads are encapsulated in a `thread_join` object. The destructor takes care of joining the thread, so it will work correctly even in case of an exception. The code also shows another C++11 feature: The `=delete` notation for a member function tells the compiler to not create the default version, so it will not be possible to copy or assign a `thread_join` object. This replaces the C++98 idiom of declaring a member function, like the assignment operator, as private, without implementing it (and there's also a corresponding `=default` to make it explicit that you do want the default version). thread destructors can terminate your program! 443 444 `=delete` `=default`

```
#include <thread>
#include <chrono>
#include <iostream>

using namespace std;

void set_alarm(int secs)
{
  cout << "\n    Setting alarm for " << secs << " seconds." << endl;
  this_thread::sleep_for(chrono::seconds(secs));
  cout << "    " << secs << " seconds have passed. Time to wake up!" << endl;
}

int main()
{
  int s;

  cout << "Welcome to multi-alarm.\n";
  do {
    cout << "How many seconds do you want to sleep? 0 to exit: ";
    if (cin >> s && s > 0)
    {
      thread t (set_alarm, s);
      t.detach();
    }
  } while (s);
}
```

Figure 145: Starting multiple detached threads with parameters

11.3.3 Sharing Data between Threads

std::mutex

As discussed above, mutexes can be used to protect data shared between threads from becoming inconsistent due to concurrent writes (if all threads do is read from a common data structure, there is no problem). The workflow is to first lock the mutex associated with the data structure, then do the changes, then release the lock. All other threads trying to lock the same mutex will be blocked until the first thread releases the lock. Figure 147 on page 238 shows a basic example. 445

lock_guard

Note that we do not unlock the mutex: the `lock_guard` will automatically unlock the mutex in its destructor, so it will be released at the end of the function scope or if an exception is thrown (another example of RAII at work).

In this example, we used global variables for the data structure and its mutex, but in a real implementation, you would encapsulate both in a class that makes the connection between the mutex and its data structure explicit.

preventing deadlocks

Once you start locking, you have to think about possible deadlocks in your system. Deadlocks

```
#include <iostream>
#include <thread>

using namespace std;

class thread_join
{
public:
    explicit thread_join(void func())
    {
        cout << "starting thread...";
        t = thread(func);
    }

    ~thread_join()
    {
        cout << "joining thread.";
        t.join();
    }
    thread_join(thread_join const &)=delete;
    thread_join& operator=(thread_join const &)=delete;

private:
    thread t;
};

void hello()
{
    cout << "Hello, thread! ";
}

void multi_hello() {
    thread_join tj1(hello);
    thread_join tj2(hello);
    throw true;
}

int main()
{
    try
    {
        multi_hello();
    }
    catch (...)
    {
        cout << "exceptional code!" << endl;
    }
}
```

Figure 146: RAII for joining threads

```
#include <mutex>

list<Widget> widget_list;
mutex widget_mutex;

void add_widget(Widget w)
{
    lock_guard<mutex> guard(widget_mutex);
    widget_list.push_back(w);
}
```

Figure 147: Locking a shared data structure through a mutex

can happen when a thread needs to lock more than one mutex to do its work and the same resources are needed by a second thread, but locked in a different order: both threads will now wait until the other resources become available, which will never happen. Deadlock avoidance is a complex topic, but one strategy that can be employed is to always lock multiple resources *in the same order*. This can be implemented in multiple ways; one option is to use `defer_lock`, as shown in Figure 148 on the next page. 446

447 defer_lock The program attempts to swap data from two members of `Gizmo`, so it will need to lock both objects to be able to perform the swap. The `defer_lock` option leaves the mutex unlocked when the lock object is constructed. Only when we pass the `unique_lock` objects to `lock`, the runtime system will attempt to acquire *all* locks. This operation is atomic: if it is not possible to obtain all locks, `lock` will fail with an exception and also unlock all partial locks it might have acquired.

11.3.4 Communication between Threads

So far, we have not discussed how two threads can work collaboratively on a problem. We can wait for a thread to finish using `join()`, but then this thread must end execution. What if one thread continuously creates new resources (e.g., objects) that are then consumed by a second thread?

We can use a shared data structure, protected with a mutex, to exchange data between threads. For example, one thread could add new elements to a queue, and a second thread takes them out for processing. But how does the second thread know there is a new element available in the queue? It could continuously check the queue for any (new) elements, but that is very wasteful, since it consumes resources – this is called ***spinning*** (or *busy waiting*) and is a well-known anti-pattern in programming. We could also let the second thread `sleep_for` a period of time, before checking for new input: this is better, but has the risk of waking up too often (and wasting resources again) or oversleeping (perhaps now the first thread will create new data faster than we consume it, due to taking a nap). A much better solution is to use the standard library facilities for triggering events between threads.

avoid spinning (busy waiting)

condition_variable One implementation option is to use ***condition variables***. You can define a variable with 448

```
std::condition_variable my_condition;
```

```
#include <iostream>
#include <utility>
#include <mutex>

using namespace std;

class Gizmo
{

public:
    Gizmo() = default;
    Gizmo(int i):content(i){}

private:
    int content;
    mutable mutex m;

friend void locked_swap(Gizmo &, Gizmo &);
friend ostream & operator<<( ostream &, const Gizmo & );
};

ostream & operator<<( ostream & os, const Gizmo & g)
{
    os << g.content;
    return os;
}

void locked_swap(Gizmo & l, Gizmo & r)
{
  if(&l==&r)
    return;
  unique_lock<mutex> lock_a(l.m, defer_lock);
  unique_lock<mutex> lock_b(r.m, defer_lock);
  lock(lock_a, lock_b);
  swap(l.content, r.content);
}

int main()
{
    Gizmo g1(1);
    Gizmo g2(2);
    cout << "Gizmo_1 = " << g1 << "; Gizmo_2 = " << g2 << endl;
    locked_swap(g1, g2);
    cout << "Gizmo_1 = " << g1 << "; Gizmo_2 = " << g2 << endl;
}
```

Figure 148: Deferred locking of multiple mutexes to avoid deadlocks

(note that you will need to `#include <condition_variable>`) and then call a notification function on this condition object with

notify_one

```
my_condition.notify_one();
```

wait()

The consuming thread that wants to be notified can `wait()` on this condition object by supplying a function (e.g., is the queue not empty?), like here:

```
my_condition.wait(data_lock, check_function);
```

where `check_function` would be implemented as

```
bool check_function() { return !data.empty(); }
```

Using C++11's support for unnamed functions, we can actually write this definition directly into the `wait()` call, using a so-called ***lambda***:

```
my_condition.wait(data_lock, []{return !data.empty();})
```

(we will discuss lambdas in more detail in the last lecture, see Section 14.4.1 on page 319).

```
#include <queue>

template<typename T>
class threadsafe_queue
{
public:
    threadsafe_queue();
    threadsafe_queue(const threadsafe_queue &);
    threadsafe_queue& operator=(const threadsafe_queue &) = delete;

    void push(T new_value);
    bool try_pop(T& value);
    void wait_and_pop(T& value);
    bool empty() const;

private:
    mutable std::mutex mut;
    std::queue<T> data_queue;
    std::condition_variable data_cond;
};
```

Figure 149: Interface for a thread-safe queue with condition variables

A thread-safe queue. The `threadsafe_queue` interface shown in Figure 149 on the facing page, adapted from (Williams 2012), is an example for a thread-safe queue that provides synchronization between threads through a condition variable. The interface is based on the STL queue, but combines the two functions `front()` and `pop()` into a single function `try_pop()` that returns the front element and removes it from the queue at the same time if one is available (returning false otherwise). In the STL, they are split for the same reason as discussed in Section 10.3 on page 211 for the STL stack: to provide exception safety. But this interface is not thread-safe: two threads could check if the queue has at least one element and then pop it – but both would be able to get the same element and process it, leading to an inconsistent system state: 449 450

```
          Thread 1                        Thread 2

     if( !q.empty() )
     {                               if( !q.empty() )
         v = q.front();              {
                                         v = q.front();
         q.pop();
         process(v);                     q.pop();
                                         process(v);
     }                               }
```

Note that mutexes do not help us here: this problem occurs even when all queue functions are protected by a mutex. The general issue is the design of a ***thread-safe interface***. There are various solutions to this problem; one option is to combine both `front()` and `pop()` into a single call. To also make it exception-safe, we pass the argument to pop by reference. To provide for synchronization, we have two versions of pop: The first, `try_pop()`, always returns immediately but will not succeed in returning an element if the queue is empty. The second, `wait_and_pop()`, will block in case of an empty queue until another thread added a new element.

designing thread-safe interfaces

The class in Figure 150 on the following page shows a complete implementation for the thread-safe queue. Note that each operation is protected by a mutex. The assignment operator is not implemented to make the example simpler (we do provide a copy constructor, so a thread-safe queue can still be copied). The call to `wait` makes use of the lambda syntax mentioned earlier. Also note that we make the mutex `mutable` to allow locking a `const` queue. 451 452 453

We can now run two threads, one producing and one consuming data, which synchronize using this queue: 454

```
threadsafe_queue<Widget> widget_queue;

void data_preparation_thread()
{
    while(more_data_to_prepare())
    {
        Widget const w=prepare_widget();
        widget_queue.push(data);
    }
}
```

```
#include <mutex>
#include <condition_variable>
#include <queue>

template<typename T>
class threadsafe_queue
{
public:
    threadsafe_queue() {}

    threadsafe_queue(threadsafe_queue const& other)
    {
        std::lock_guard<std::mutex> lk(other.mut);
        data_queue=other.data_queue;
    }
    void push(T new_value)
    {
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(new_value);
        data_cond.notify_one();
    }
    void wait_and_pop(T& value)
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk, [this]{return !data_queue.empty();});
        value=data_queue.front();
        data_queue.pop();
    }
    bool try_pop(T& value)
    {
        std::lock_guard<std::mutex> lk(mut);
        if(data_queue.empty())
            return false;
        value=data_queue.front();
        data_queue.pop();
        return true;
    }
    bool empty() const
    {
        std::lock_guard<std::mutex> lk(mut);
        return data_queue.empty();
    }

private:
    mutable std::mutex mut;
    std::queue<T> data_queue;
    std::condition_variable data_cond;
};
```

Figure 150: Implementation for a thread-safe queue with condition variables

```
void data_processing_thread()
{
    while(true)
    {
        Widget w;
        widget_queue.wait_and_pop(w);
        process(w);
    }
}
```

11.3.5 The Keyword `volatile`

In multi-threaded code, one thread may wait for an event that is caused by another thread. In this example, an instance of `Gadget` has a function `wait` that checks the status of its variable `sleeping` once every second. Eventually, another thread calls `wakeup`, changes the value of `sleeping`, and terminates the `while` loop. 455

```
class Gadget
{
public:
    void wait()
    {
        while (sleeping)
        {
            pause(1000); // pauses for 1000 milliseconds
        }
    }

    void wakeup()
    {
        sleeping = false;
    }

private:
    bool sleeping;
};
```

Unfortunately, this class has a serious problem: The compiler may notice that `pause`, being a system or library function, cannot possibly change the value of the instance variable `sleeping`. There is an obvious optimization: copy the value of `sleeping` into a processor register and inspect the register each time round the loop, thereby avoiding an expensive memory access. But, of course, the register never changes, the `wakeup` call goes by unnoticed, and the thread is stuck in the `wait` loop.

Compilers always try to use registers as much as possible; these optimizations have a significant effect on performance. Preventing optimizations of this kind globally would not be a good idea.

Both C and C++ provide an alternative in the form of the modifier `volatile`. Changing the declaration of `sleeping` to 456

```
private:
    volatile bool sleeping;
```

informs the compiler that `sleeping` may change its value at any time, and that this value must therefore be refreshed from memory whenever needed, not saved in a register.

The original use of `volatile` was for peripheral devices that used "direct memory access". A keystroke, for example, would change the value stored at a particular location (and probably also raise an interrupt). The program would bind a variable to that location (C and C++ have a mechanism for that, too) and use the variable to read the keyboard. It was necessary to declare the variable to be `volatile` for the same reasons as above – optimization would be a disaster. Nowadays, `volatile` is still used, but usually to make code thread-safe.

11.3.6 Constant and Mutable

457 The following program compiles successfully and prints "1":

458

```
class Counter
{
public:
   Counter() : timesUsed(0) {}

   void add()
   {
      ++timesUsed;
   }

   void show() const
   {
      cout << timesUsed << endl;
   }

private:
   int timesUsed;
};

int main()
{
   Counter c;
   c.add();
   c.show();
   return 0;
}
```

If we change the declaration of c to 459

```
const Counter c;
```

the compiler complains because Counter::add is not declared const and therefore cannot be used with a const object.

If we attempt to correct this by changing the declaration of Counter::add to

```
void add() const
```

the compiler complains that the value of a const object is being changed. This is because all data members of a const object are considered const. (The problem here, of course, is that we are changing timesUsed.)

Suppose that timesUsed is not logically part of the object. It might, for example, just be a counter that we are using to see how often a function is called. The keyword mutable, used to qualify a variable, informs the compiler that the variable should not be considered as part of the object. "Mutable" can be interpreted as "can never be const". The following version of the program compiles and runs correctly. 460

```
class Counter
{
public:
   ....
private:
   mutable int timesUsed;
};

int main()
   ....
```

Here is another application of mutable. We have a class Widget with a member function that returns a string representation of the object. The object's value changes only occasionally, and it is rather expensive to compute the string representation. Consequently, we would like to cache the stringified version and do the conversion only when necessary. The class would look something like Widget in Figure 151. The problem with this class is that getRep *ought* to be declared const because, logically, it does not change the state of the object. But it cannot be declared const, because it changes the value of cache and staleCache, which are not *logically* part of the object. 461 462

The solution, of course, is the qualifier mutable: 463

```
class Widget
{
public:
   string getRep() const
   ....
```

```
class Widget
{
public:
   string getRep()
   {
      if (staleCache)
      {
         cache = ....;
         staleCache = false;
      }
      return cache;
   }

   void change()
   {
      ....
      staleCache = true;
   }

private:
   bool staleCache;
   string cache;
};
```

Figure 151: Is a cache part of an object?

```
private:
   mutable bool staleCache;
   mutable string cache;
};
```

11.3.7 Promising the Future

So far, we have been working on a rather low abstraction level, manually creating threads and manipulating them. A higher-level abstraction is the concept of a ***Future***, which encapsulates a unit of (asynchronous) work. The counterpart of a future is a ***Promise***, which a function can provide as a result.[53] Working with futures and promises is especially useful for developing code that can dynamically scale on multi-core systems. It is also an important foundation for modern systems based on ***Reactive Programming***.[54]

multi-core programming

Here is one way to make use of futures and promises: Rather than waiting for the result of a function call to return (and blocking a program in the meanwhile), we let it run asynchronously,

[53] Futures and Promises in C++ roughly correspond to `Java` 8's `CompletableFuture` and `CompletionStage`.

[54] See the ***Reactive Manifesto*** for more information on reactive programming: https://www.reactivemanifesto.org/

in a different thread, until we need the result. Let's say a function `deep_thought` needs some time to calculate an `int` value. Rather than calling it directly with

```
int result = deep_thought();
```

we can process it asynchronously through a future: 464

```
future<int> result = async(deep_thought);
do_other_stuff();
cout << "The answer is: " << result.get() << endl;
```

Here, `result.get()` blocks if the result is not available yet. In this example, we created the `future` object directly. A more general approach is to pass the result through a promise object, as shown in Figure 152. 465

11.4 Distributed Computing

So far, we were concerned with single-computer concurrency, where multiple processors with multiple cores execute the threads of a program. But what if a single computer is not fast enough to solve a problem? A typical example is web-scale data processing, where you have large amount of data to analyze ("Big Data"). "Big Data"

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

using namespace std;

void deep_calc(int i, promise<int> * promiseResult)
{
    // calculation takes 7.5 million years
    this_thread::sleep_for(chrono::seconds(1));
    promiseResult->set_value(i+1);
}

int main()
{
  promise<int> promiseObj;
  future<int> futureObj = promiseObj.get_future();
  thread t(deep_calc, 41, &promiseObj);
  cout << "I can do other stuff now..." << endl;
  cout << "The answer is: " << futureObj.get() << endl;
  t.join();
}
```

Figure 152: Futures and Promises

cluster computing

One solution is to use *clusters*, collections of individual machines running the same computations on parts of the data. It then becomes desirable to automatically distribute a large job to a large number of machines (e.g., tens of thousands of PCs), without having to make changes to the code doing the processing. A solution that has been widely adopted by industry and science is *MapReduce*. As we will see below, this paradigm can also be applied for parallel programming on multi-core systems.

11.4.1 MapReduce

MapReduce[55] is a parallel programming approach introduced by Google engineers (Dean and Ghemawat 2004) for easing the development and deployment of large-scale data processing applications. It is the framework currently used by Google for many of its applications performing massively parallel processing of hundreds of terabytes of data in its distributed data centers. The MapReduce framework has meanwhile also been adopted by companies such as Yahoo!, Facebook, or LinkedIn and proved to be a useful (albeit not actually new) paradigm for cloud or grid computing. A well-known commercial cloud computing service offering MapReduce is Amazon's *Elastic MapReduce*[56] running on top of its *Elastic Compute Cloud (EC2)* service.

Together with cloud computing, MapReduce can save significant costs and time compared with buying traditional server resources when they are only needed for specific, large-scale jobs ("burstiness"). For example, the *New York Times* used MapReduce running on 100 Amazon EC2 instances and a MapReduce application (implemented with Hadoop) to process 4TB of raw image TIFF data (stored in Amazon's Simple Storage Service, S3) into 11 million finished PDFs in the space of 24 hours at a computation cost of about $240 (not including bandwidth).[57]

11.4.2 MapReduce Concept

The basic idea of MapReduce stems from *functional programming*.[58] A particular feature of functional programming is that functions have no side-effects, hence they are trivially parellelizable. This is part of the reason for the currently growing interest in functional and object-functional programming languages (like Scala), as moderns PCs are increasing the number of cores, but not clock speed – hence requiring better support for parallel programming to make use of these multi-core systems.[59]

MapReduce takes one idea of functional programming and brings it to object-oriented languages (originally only C++, but now many languages offer MapReduce implementations or bindings). To compute a certain result on a large set of data, first split it out into n partitions (e.g., 4 for a quad-core system, or 1000 for a large cluster of PCs). Then, execute the same function on each of the partitions. This is expressed by a `map` function that takes the input job in form of a `list` and computes an intermediate result for each list element (which can be, e.g., a single

[55] MapReduce in Wikipedia, http://en.wikipedia.org/wiki/MapReduce

[56] Elastic MapReduce, http://aws.amazon.com/elasticmapreduce/

[57] New York Times Blog, http://open.blogs.nytimes.com/2007/11/01/self-service-prorated-super-computing-fun/

[58] See http://en.wikipedia.org/wiki/Functional_programming for an introduction if you are not familiar with the functional programming paradigm.

[59] See Herb Sutter's article, *"The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software"*, Dr. Dobb's Journal Vol. 30, No. 3 (March 2005), http://www.gotw.ca/publications/concurrency-ddj.htm.

document, a single sensor reading or database record). Then, in a second step, the intermediate results have to be combined to the end result. This is the `reduce` step: 466

```
map (in_key, in_value) -> (out_key, intermediate_value) list

reduce (out_key, intermediate_value list) -> out_value list
```

In more detail, these functions work in the following way. First, a `list` of input data sources has to be built. For example, a list of documents, database rows, or sensor readings. Then, instead of developing a single function that computes the required result from the input (e.g., a word count over all input documents), we need to split the computations into the `map` and `reduce` functions.

Map. The `map` function takes as input a `(in_key, in_value)`-pair and computes a corresponding `(out_key, out_value)`-pair. For example, if we want to count words across a large number of documents, we can map the word-count function onto each individual document. The (intermediate) result is then a word count for each individual document (independent from the other documents).

Reduce. When the map phase is over, the reduce phase starts to combine the individual results into the final end result. In the word count example, we now know how often a word appears in each document, so we still have to add up the results for each word across all documents to obtain the final result. This is done by the `reduce` function. To distribute the reduce job onto multiple machines, we need to aggregate the results by key (in practice, the key values are often stored in a hash table and then distributed onto parallel reduce jobs modulo the number of available nodes).

Figure 153[60] shows the parallel workflow in a MapReduce application. Note that the reduce phase cannot start until the map phase has been completed. 467

11.4.3 MapReduce Implementations

The original (and current) implementation by Google is developed in C++. Not surprisingly, this implementation is not available outside Google, but many other implementations in various languages have been developed as well, including Java (Hadoop), Ruby (Skynet), Haskell (Holumbus), and Erlang (Disco). Many of them are free/open source.

For writing mappers and reducers in C++, two free options are *QtConcurrent* (used in this lecture) and the *Pipes* interface for Hadoop.

[60] Image Copyright 2007 University of Washington, Distributed under Creative Commons Attribution 2.5 License, Google lectures on MapReduce, http://code.google.com/edu/submissions/mapreduce-minilecture/listing.html

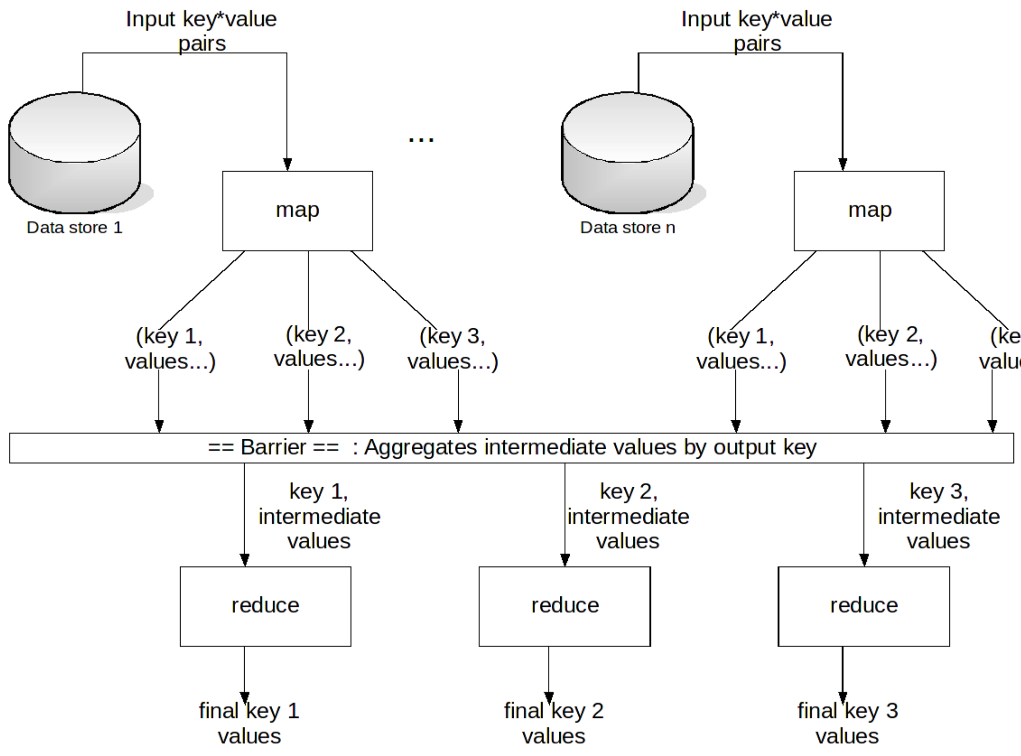

Figure 153: MapReduce Workflow

Hadoop. Apache Hadoop[61] is an open source, Java-based framework for cluster computing using the MapReduce concept. It includes a distributed file system (HDFS) and provides language bindings for many programming languages, including *Pipes* for C++.[62] With these bindings, it is also possible to write mappers and reducers in different languages – e.g., a `map` function in C++ and a `reduce` function in Python.

One of the main code contributors to Hadoop is Yahoo!, who also runs one of the largest Hadoop clusters.[63] Hadoop can be run on a single PC, but is primarily targeting deployment on large clusters (e.g., it is rack-aware) and features support for cloud computing (Cloudstore, Amazon S3).

QtConcurrent. Digia maintains the open-source, cross-platform (Linux, Mac, Windows, as well as some mobile platforms) C++ *Qt* template library,[64] For more on Qt, see https://en.wikipedia.org/wiki/Qt_(framework). which includes the parallel namespace *QtConcurrent*,[65] that comes with a MapReduce implementation. Unlike Hadoop, this implementation is targeting single PCs and is optimized for multi-core parallelism. Since it is significantly easier to setup this library

[61]Hadoop, http://hadoop.apache.org/

[62]See https://wiki.apache.org/hadoop/C++WordCount for a Hadoop version of the "WordCount" program.

[63]Yahoo! repeatedly won the yearly sorting contests using its Hadoop cluster, e.g., in April 2009 the minute sort by sorting 500 GB in 59 seconds (on 1400 nodes) and the 100 terabyte sort in 173 minutes (on 3400 nodes).

[64]Originally developed by the Norwegian company Trolltech, which was later acquired by Nokia where it became part of QtLabs, which was then sold to Digia around the time Nokia was bought by Microsoft and finally spun off into The Qt Company in 2014.

[65]QtConcurrent, http://doc.qt.io/qt-5/qtconcurrent-index.html

than a Hadoop cluster, we use QtConcurrent's MapReduce implementation in the following examples. However, note that the programming methodology is the same as for a large cluster.

11.4.4 Hello, MapReduce!

The "Hello, world" for MapReduce is a parallel word count example. Input is a set of documents. Expected output is a count for each word appearing in the documents.

To implement this with MapReduce, we have to:

1. build the list of input documents to count
2. write a mapper function (here called `countWords`)
3. write a reducer function (here called `reduce`)

The main data structure passed between the `map` and `reduce` phases is a `Map` (using Qt's cross-platform implementation, `QMap`, rather than an STL `map`):

```
typedef QMap<QString, int> WordCount;
```

Then we can call the `mappedReduced` function that takes as input the list to be processed, the map, as well as the reduce function and calls them in the appropriate order on the input documents (Figure 154). 468

```
int main() {
    QStringList files = findFiles("../qt/src/corelib",
         QStringList() << "*.cpp" << "*.h");
    WordCount total = mappedReduced(files, countWords, reduce);
}
```

Figure 154: Counting words, MapReduce-style

The map function `countWords` (Figure 155 on the next page) simply has to count the words in each input list element, i.e., each document.[66] 469

Finally, the `reduce` function (Figure 156 on the following page) has to add up the individual word count results (per-document) for all input documents. 470

The `wordcount` example program shipped with QtConcurrent additionally demonstrates the speedup achieved through MapReduce by comparing the runtime with a single-threaded version: 471

```
finding files...
1115 files
warmup
warmup done
single thread 1881
MapReduce 482
MapReduce speedup x 3.90249
```

[66] The `foreach` statement is a macro supplied by the Qt library, see http://doc.qt.io/qt-4.8/containers.html.

```
WordCount countWords(const QString &file) {
    QFile f(file);
    f.open(QIODevice::ReadOnly);
    QTextStream textStream(&f);
    WordCount wordCount;

    while (textStream.atEnd() == false)
        foreach (QString word, textStream.readLine().split(" "))
            wordCount[word] += 1;

    return wordCount;
}
```

Figure 155: Counting words with MapReduce: Map function

```
void reduce(WordCount &result, const WordCount &w) {
    QMapIterator<QString, int> i(w);
    while (i.hasNext()) {
        i.next();
        result[i.key()] += i.value();
    }
}
```

Figure 156: Counting words with MapReduce: Reduce function

This program was executed on a quad-core processor. However, it is important to realize that the MapReduce framework will automatically scale to a larger number of cores, without requiring any changes to the code: Unlike classical multi-threading, where the number of threads has to be specified by the programmer using features of the multithreading library, here the MapReduce framework takes care of starting and stopping threads. Using a large cluster (and a suitable MapReduce implementation like Hadoop), you can test your program on, e.g., 4 nodes and then deploy the same code on 10,000 nodes.

References

Alexandrescu, A. (2001). *Modern C++ Design: Generic Programming and Generic Patterns Applied*. Addison-Wesley.

Dean, J. and S. Ghemawat (2004, December). MapReduce: Simplified Data Processing on Large Clusters. In *Sixth Symposium on Operating System Design and Implementation (OSDI'04)*, San Francisco, CA, USA. http://labs.google.com/papers/mapreduce.html.

Josuttis, N. M. (2012). *The C++ Standard Library: A Tutorial and Reference* (2nd ed.). Addison-Wesley. http://www.cppstdlib.com/.

Stroustrup, B. (2013). *The C++ Programming Language* (4th ed.). Addison-Wesley.

Williams, A. (2012). *C++ Concurrency in Action*. Manning. http://www.manning.com/williams/.