C++11 ~~and Intel TBB~~

Klaas van Gend

October 26, 2012

Let me introduce myself

Name: Klaas van Gend
Affiliation: Vector Fabrics
Job title: Sr. FAE
History / Claim to fame:

- Started programming C at 14
- First experience with Linux in 1993
- Professional Linux dev since 1999
- Author of "umtsmon"
- Author of game "The Butterfly Effect"
- Author of approx 60 magazine articles

Hobbies:

- Glider piloting
- Card and Board games

Vector Fabrics

Founded: 2007
Motto: Do more with multicore!
Employees: 20
Address:

- Vector Fabrics B.V.
  Vonderweg 22
  5616 RM Eindhoven
  The Netherlands
- +31 40 8200 960
- `http://www.vectorfabrics.com/`
- also on Twitter, LinkedIn and Facebook

Vector Fabrics

C++ History

- 1979 Bjarne Stroustrup starts working on **C with Classes**
  - no true language, uses "C++ to C compiler"
- 1983 renamed to **C++**, version 1.0
  - adds virtual functions, operator overloading
- 1989 C++ 2.0 released
  - adds multiple inheritance, abstract classes,
  - adds static, const, protected members
- 1998 ISO Standard **C++98**
  - first official standard
  - adds templates, exceptions, namespaces, bool, STL
- 2003 ISO Standard C++03
  - only typos and inconsistencies fixed
- 2007 ISO Standard TR1
  - adds regular expressions, smart pointers, hash tables, random number generators
- 2011 ISO Standard C++0x turns **C++11**
  - adds *multithreading*, move semantics, lambdas

C++11 and gcc

With the latest compilers+libs, support is looking great!
Don't forget to specify `-std=gnu++11` or `-std=c++11`

- g++ 4.8 http://gcc.gnu.org/projects/cxx0x.html
  - nearing completion, missing are mostly low-level concurrency items
  - g++ 4.7 lacks TLS and alignment support
  - don't assume any compatibility with 4.5 and below
- Clang / LLVM 3.1 http://clang.llvm.org/cxx_status.html
  - nearing completion, missing are mostly low-level concurrency items
  - Clang 3.1 support is roughly comparable to GCC 4.7
- libstdc++ http://gcc.gnu.org/onlinedocs/libstdc++/manual/status.html
  - `thread::id` comparisons ill-defined - do not use
  - timed waits have minor issues
  - minor issues at thread exit

It's all in the libraries!

- Most of complexity and platform compatibility nicely hidden in SL and STL:
  - `std::thread` run lambda or function (object) as a thread
  - `std::mutex`, `std::lock_guard`, `std::lock` for locking
  - `std::unique_lock`, `std::defer_lock`, `std::adopt_lock` to manage locks
  - `std::condition_variable`
  - `std::atomic` templates for atomic access
  - barriers and fences
  - `std::futures`, `std::promises` nice higher level interface
  - `std::chrono` for accurate time-keeping
  - exception handling in threaded environments
- Do not forget to link against pthreads: `-lpthread`

std::thread example

- std::thread simplest use: function pointer and arguments
- return codes are **inaccessible**!

Listing 1: Hello Concurrent World

```
1  #include <iostream>
2  #include <thread>
3  using namespace std;
4
5  void hello(int n)  {
6    cout << "Hello World " << n << endl;
7  }
8
9  int main(void)  {
10   thread t(hello, 684);
11   t.join();
12   return 0;
13 }
```

std::thread more constructors

Listing 2: thread using lambda expression or function object

```
 1 #include <iostream>
 2 #include <thread>
 3
 4 class funobject
 5 {
 6 public:
 7   void operator()(int n) const
 8   {
 9     std::cout << "Hi funobject " << n << std::endl;
10   }
11 };
12
13 int main(void)
14 {
15   funobject fo;
16   std::thread f(fo, 42);
17
18   std::thread l( [] (int n=684)
19     { std::cout << "Hi lambda " << n << std::endl; }
20   );
21
22   f.join();
23   l.join();
24 }
```

scope and lifespan

- `thread.join()` waits for a thread to complete
- `thread.detach()` detaches a thread from current scope
  - object no longer "`join`able"!
  - thread object can go out of scope - doesn't matter
  - real (inaccessible) thread object destroyed when thread ends
  - access to variables doesn't change

- If a thread object goes out of scope and it wasn't joined or detached, **the application ends**.

- Transfer ownership using *move semantics* (new in C++11):

```
std::thread t1(some_function);
std::thread t2 = std::move(t1);
t1.join(); // error !
```

The four mutexes

`std::mutex`

- `std::mutex.lock()` locks the mutex, blocks if already locked
- `std::mutex.try_lock()` returns true on success or false if already locked
- `std::mutex.unlock()` unlocks

`std::timed_mutex` adds:

- `std::timed_mutex.try_lock_for(d)` waits max duration `d`
- `std::timed_mutex.try_lock_until(t)` waits until time `t`

Also available:

- `std::recursive_mutex` allows multiple locks in same thread
- `std::recursive_timed_mutex`

A better locking mechanism

- Hard to do exception safe locking using `.lock()` / `.unlock()`
- Better: use `std::lock_guard`
  - its constructor locks, its destructor unlocks
  - as long as the object is in scope, your lock is held

Listing 3: using std::lock_guard

```
 1 int theI = 0;
 2 std::mutex theMutex;
 3
 4 void safe_inc(int n)
 5 {
 6     std::lock_guard<std::mutex> lock(theMutex);
 7     theI += n;
 8     std::cout << "now: " << theI << std::endl;
 9 }
10
11 int main()
12 {
13     std::thread t1(safe_inc, 42);
14     std::thread t2(safe_inc, 684);
15
16     t1.join();
17     t2.join();
18 }
```

std::unique_lock

- `std::lock_guard` locks the mutex upon creation
- If you need more flexibility or need to it move around:
  `unique_lock<mutex> myObject(aMutex, adopt_lock)`
  - Operates just like `std::lock_guard`
  - But consumes slightly more memory and cycles

  `unique_lock<mutex> myObject(aMutex, defer_lock)`
  - Creates `myObject`, but **doesn't** lock `aMutex` yet
  - Convienient if you need to lock two locks at the same time using e.g. `std::lock()`
- You use `std::unique_lock` like a local mutex
  - But still have exception safeness...
  - And you can `std::move()` the `std::unique_lock` !!!

Atomics in C++

`std::atomic<type>`

- Where `type` can be any type.
- Is a (templatized) class - and thus is a type in itself
- Class members include:
  - `.load()`
  - `.store()`
  - `.exchange()`
  - `.compare_exchange_weak()` and `.compare_exchange_strong()`
- If `type` is an integral type like `int` or `double`, there are additional members:
  - `.fetch_add()`, `.fetch_sub()`, `.fetch_or()`, `.fetch_and()`, `.fetch_xor()`
  - each of the above also exists as += and similar
  - there are also overloaded ++ and – operators

C++ atomics in practice

The easy part:

`int myValue = 684;`	not atomic
`myValue++;`	not atomic
`atomic<int> myAtom = myValue;`	assignment works
`myAtom += 42;`	atomic addition
`myValue = myAtom.fetch_sub(3);`	atomic subtraction

Both variables will now contain $684 + 42 - 3$

Note that myValue did not become atomic!

C++ also allows different memory synchronization mechanisms.

- a topic that gives you a headache, guaranteed!
- suggested reading: *C++ Concurrency in Action* by Anthony Williams, chapter 5.

Futures and Promises

- The work flow so far:
  - use `std::thread` to spawn a new thread
  - make sure all data is available
  - take care of protecting read/write access to shared data
  - `.join()` when you need the output of the thread
  - and make sure the output is still available at that stage
- Every step complicated due to possible exceptions

- By now, everyone is planning their own thread class abstraction?
- Hopefully not!

Example using std::async and std::future

Listing 4: workload: reverse string

```
 1 #include <string.h>
 2
 3 char* revString(char* arg)
 4 {
 5    // WARNING mem leak here!!!
 6    char* result = strdup(arg);
 7    size_t end = strlen(arg);
 8    for (int i=0; i< end; i++)
 9      result[i] = arg[end-i-1];
10    return result;
11 }
```

Listing 5: serial version

```
13 int smain(int argc, char* argv[])
14 {
15    for(int i=1; i<argc; i++)
16      cout <<revString(argv[i]) <<" ";
17    cout << endl;
18 }
```

Listing 6: parallel version

```
 1 #include <future>
 2
 3 int fmain(int argc, char* argv[])
 4 {
 5    std::future<char*> myV[argc];
 6
 7    for(int i=1; i<argc; i++)
 8      myV[i-1] =
 9          std::async(revString, argv[i]);
10
11    for(int i=0; i<argc-1; i++)
12      cout << myV[i].get() << " ";
13    cout << endl;
14 }
```

std::async and std::future - 1

- `std::async()` has similar notation to `std::thread()` :
  - `std::async(myFunction)`
  - `std::async(myFunction, variable)`
  - `std::async(&Class::member, &instance, var)`
- Conceptually, they are different:
  - a `async` starts an asynchronous function call
  - a `thread` is something that runs in parallel to the current thread.
  - so an `async` *could* be implemented as a separate thread

`std::async` and `std::future` - 2

- a `future` holds a value that should become available sometime as the result of an asynchronous function call.
- `std::future<T>` holds the *future* return value of type `T`
- Calling `std::async`, running the asynchronous code can happen:
  - immediately after start
  - only once you call the `.get()`
  - you can specify by adding another parameter to `std::async` :
    - `async(std::launch::async|std::launch::deferred, myFunction)`
    - (the default behavior specifies both)
- Exceptions thrown inside `myFunction()` will be caught, kept and rethrown in the `.get()`

std::packaged_task

- `std::async` has a few limitations
  - no control over the wrapped "task"
  - future is returned immediately - and is only movable.
- Higher level abstraction: the `std::packaged_task` template.
- Example notation:
  - `std::packaged_task<int()> task( [](){return 7;} );`
  - `<int()>` implies a function object that returns an `int`
  - `[](){return 7;}` is a lambda that just returns 7.
- The task is now wrapped, can be executed anywhere anytime.
- A future return value can be obtained via `.get_future()`
- The wrapped task (lambda in our case) will be executed:
  - *synchronously* if we call the `operator()` on the packaged_task
  - *asynchronous* if we `std::move` the packaged_task into a thread
  - *"likely" asynchronous* if we just call `.get()` on the future

`std::promise`

- A `std::promise` is a vehicle to move parameters between threads
- It holds all necessary synchronization primitives
- Used internally by `std::packaged_task`
  - It passes the arguments residing in the packaged_task to the wrapped function
  - It hides the `std::future` from the wrapped function, but puts the return value in the future
  - It helps in catching all exceptions and storing them in the future
- "Fun":
  - `std::move` a `std::promise`
  - request its future through `.get_future()`
  - an exception of type `std::future_errc:broken_promise`.
- You probably don’t need to toy with promises yourself.

Shared futures

- `std::future` is only `std::moveable`
- But what if you need the answer to a certain operation for two other operations?
  - Use a `std::shared_future` !
- This enables all kinds of 'fun' coding, like functional programming.
- No need for explicit locks to have e.g. spreadsheet cells depend on each other
- The *language* will sort out the ordering in which futures can be calculated!
- But let's not dive into that here...

Waiting...

- `future.get()` waits indefinitely
- But what if you don't want to wait that long?
  - In addition to `.lock()` `.get()` and C++11 defines things like `.wait_until()`, `.wait_for()` and `.try_lock_until()`
- Both require timing notation: a *point in time* and *duration*
  - neither being defined well in C
  - nor POSIX
  - let alone across platforms
- Cue the `chrono` library:
  - `std::time_point`, a point in time
  - `std::duration`, a time interval
  - `std::system_clock`, wall clock time from the system-wide realtime clock
  - `std::steady_clock`, monotonic clock that will never be adjusted
  - `std::high_resolution_clock`, the clock with the shortest tick period available

Summary

C++11

- Adds several layers of multithreading concepts:
  - "Basic":
    - `std::thread`
    - `std::mutex` and `std::lock_guard`
    - `std::atomic`
  - "Abstracted":
    - `std::packaged_task` and `std::async`
    - `std::future` and `std::promise`
- It also provides abstract but accurate timing and waiting

- Now only we have to wait for 100% compliant compilers and libraries...

LAB

Parallelize an encheferizer

Encheferizer

- Also known as *Jive filter*
- Converts text into "Svedeesh", Bork Bork Bork.

- Example code written in 'normal' C++
- Your jobs:
  - Parallelize the code using C++11 constructs
  - Prove it is paralellized
  - Prove it still has the same output

Svedeesh Cheff

Frum Veekipedia, zee free-a incyclupedeea

Zee **Svedeesh Cheff** is a Mooppet vhu eppeered in zee lung-roonneeng *Zee Mooppet Shoo* und ves oopereted by Jeem Hensun und Frunk Ooz seemooltuneuoosly. Bork Bork Bork!

Cuntents [hide]

1 Cherecter

Main

Listing 7: serial encheferizer main

```
 1 #include <cstdio>
 2 #include <fstream>
 3 #include <sstream>
 4 #include <iostream>
 5
 6 extern std::string chefLine(const std::string& aLine);
 7
 8 int smain(void)
 9 {
10     std::ifstream file("infile.txt");
11     char line[512];
12     while(!file.eof())
13     {
14       file.getline(line, 511);
15       std::cout << "  " << chefLine(line) << std::endl;
16     }
17 }
```

LAB

Parallelize an encheferizer

Solution

Encheferizer

Important things to remember:

- Do not forget to link against pthread: `-lpthread` !!!
- `std::async` needs to always have access to its arguments
- Split the `while` loop:
  - reading the file and starting the `async`s
  - `.get()`ing the `futures`
- Do not forget to force `async` to actually work asynchronously

Solution

Listing 8: parallel encheferizer main

```
 1 #include <fstream>
 2 #include <iostream>
 3 #include <future>
 4 #include <vector>
 5
 6 extern std::string chefLine(const std::string& aLine);
 7
 8 int pmain(void)
 9 {
10     std::vector<std::string> myLines;
11     std::vector<std::future<std::string>> myConvertedLines;
12
13     std::ifstream file("infile.txt");
14     char line[512];
15     while(!file.eof())
16     {
17       file.getline(line, 511);
18       myLines.push_back(line);
19       myConvertedLines.push_back(
20         std::async(std::launch::async, chefLine, myLines.back()) );
21     }
22
23     // note the &: myLineFuture is not the iterator but the future
24     for(auto& myLineFuture : myConvertedLines)
25       std::cout << "  " << myLineFuture.get() << std::endl;
26 }
```

Is it really multithreaded?

Listing 9: parallel encheferizer main

```
> gdb ./main.out
GNU gdb (GDB) SUSE (7.3-41.1.2)
...
Reading symbols from main.out...done.
(gdb) set args p
(gdb) run
Starting program: main.out p
[New Thread 0x7ffff70b7700 (LWP 7255)]
[New Thread 0x7ffff68b6700 (LWP 7256)]
[New Thread 0x7ffff60b5700 (LWP 7257)]
[Thread 0x7ffff70b7700 (LWP 7255) exited]
[Thread 0x7ffff60b5700 (LWP 7257) exited]
[New Thread 0x7fffeffff700 (LWP 7258)]
[New Thread 0x7fffef7fe700 (LWP 7259)]
[Thread 0x7ffff68b6700 (LWP 7256) exited]
[New Thread 0x7fffeeffd700 (LWP 7260)]
[Thread 0x7fffeffff700 (LWP 7258) exited]
[New Thread 0x7fffee7fc700 (LWP 7261)]
[Thread 0x7fffef7fe700 (LWP 7259) exited]
[Thread 0x7fffeeffd700 (LWP 7260) exited]
[New Thread 0x7fffedffb700 (LWP 7262)]
[New Thread 0x7fffed7fa700 (LWP 7263)]
[Thread 0x7fffee7fc700 (LWP 7261) exited]
[Thread 0x7fffedffb700 (LWP 7262) exited]
[Thread 0x7fffed7fa700 (LWP 7263) exited]
  Threedeeng fecileeteees
...
```

Questions?

Questions, anyone?

klaas AT vectorfabrics.com
http://www.vectorfabrics.com/