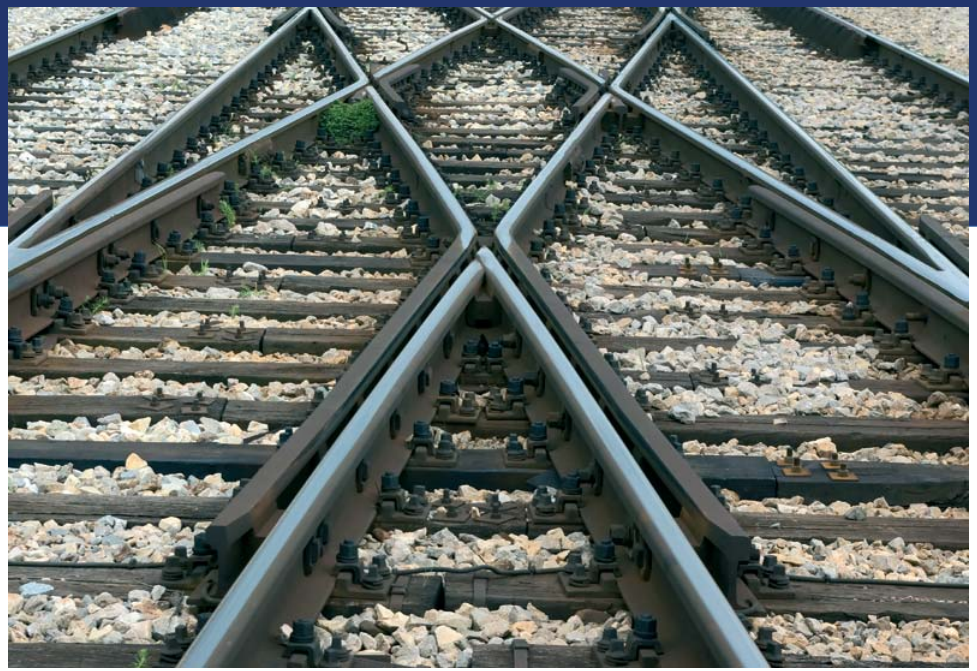
Parallel programming

C++11 threads



Libor Bukata a Jan Dvořák

FAKULTA
ELEKTROTECHNICKÁ
ČVUT V PRAZE

C++11 – programme

- Executing tasks by **async** object.
- **Future**, **promise** – synchronized access to values.
- **Atomic functions** in C++11
- Exercise – write your parallel code...

C++11 – async

- **async** executes a method asynchronously, i.e., without waiting for its completion and possibly with a delayed start
- async policy:
  - **launch::async** – creates a new thread
  - **launch::deferred** – method is started after its return value is requested (by using future object).
- Async API:
  - // Execute the method asynchronously.
  - **future**<T> ret = **async**(method, params…);
  - // The same without return value + async exec.
  - **async**(lauch::async, method, params…);

C++11 – future object

- **future** object is used to pass/obtain a value to/from a thread
- if value is not yet available:
  - blocks until the value is computed (wait)
  - waits some time (wait_for, wait_until)
- future API:
  - **future**<T> fut = **async** (method, args…);
  - T val = fut.get(); // get the returned value

C++11 – promise object

- **promise** stores a value that is subsequently obtained by using the associated future object (synchronization point) in another thread.
- promise API:
  - **promise**<T> prom; // creation
  - **future**<T> fut = prom.get_future(); // get related obj
  - prom.set_value (T());   // set promised value

Asynchronous call - example

```
#include <iostream>
#include <future>
#include <mutex>
#include <vector>
#include <thread>

using namespace std;
using namespace std::chrono;

class CountingThreads {
   public:
      CountingThreads() { }
      void run() {
         uint32_t numThreads = thread::hardware_concurrency();
         for (uint32_t t = 0; t < numThreads; ++t)      {
            promise<unsigned long> prom;
            future<unsigned long> futVal = prom.get_future();
            future<unsigned long> retVal = async(launch::async,
                                    &CountingThreads::countTask, this, ref(futVal));
            this_thread::sleep_for(seconds(t));
            prom.set_value(t);
            cout<<"thread "<<t<<" returned value "<<retVal.get()<<endl;
         }
      }
   private:
      unsigned long countTask(future<unsigned long>& futVal)  {
         uint32_t threadId = futVal.get();
         return 10u*threadId;
      }
};

int main()    {
   CountingThreads ct;
   ct.run();
   return 0;
}
```

get promised value

returned value converted
to future object

Atomicity in C++11

- Atomic operations are **indivisible**, i.e. they behave like one instruction.
- Useful for a non-blocking synchronization between threads.
- Often lock-free for **integer and pointer types**.
- Atomic operation:
  - **load** value
  - **modify** value
  - **write** value

```
int x = v;
x += 5;
v = x;
```

```
atomic<int> v(0);
...
v.fetch_add(5);
```

must be indivisible!

Atomicity in C++11

- Basic operations with atomic class:
  - load, store
  - operator++, operator--
  - fetch_add, fetch_sub
  - fetch_and, fetch_or, fetch_xor
- The **atomic_flag** is a specialization of atomic for a boolean value (flag).
- Method **test_and_set**() returns the previous boolean value and sets the current one to true.

Atomic functions - example

```
#include <atomic>
#include <iostream>
#include <future>
#include <vector>
#include <thread>

using namespace std;
using namespace std::chrono;

class CountingThreads {
   public:
      CountingThreads() : counter(0u) { }
      void run() {
         vector<future<unsigned long>> retVals;
         uint32_t numThreads = thread::hardware_concurrency();
         for (uint32_t t = 0; t < numThreads; ++t)
            retVals.push_back(async(launch::async, &CountingThreads::countTask, this));
         for (uint32_t t = 0; t < numThreads; ++t)
            cout<<"thread "<<t<<" returned value "<<retVals[t].get()<<endl;

         cout<<"Counting finished, final value is "<<counter<<"."<<endl;
      }
   private:
      unsigned long countTask()      {
         for (int i = 0; i < 1e7; ++i)
            counter.fetch_add(i);

         return counter.load();
      }

      atomic<unsigned long> counter;
};

int main()    {
   CountingThreads ct;
   ct.run();
   return 0;
}
```

Using atomic functions is much faster than using mutex!

Main exercise – C++11 barrier

- Use atomic functions to implement C++11 **barrier** – passed threads actively wait (busy waiting) until the last thread enters.
- Recommended API (**reusable class**):
  - Barrier(cont uint32_t& numThreads);
  - Barrier.wait();
  - ~Barrier();
- **Hints:**
  - Use atomic<uint32_t>::fetch_add method to increase the number of waiting threads.
  - The last thread sends a signal to other threads by using additional atomic variable (e.g., phase counter). The counter of waiting threads is not sufficient per se to satisfy thread-safe code.

Additional Assignments

- Write a parallel program that calculates histogram data from arbitrary file.
  - Repeat it for the list of English words (useful for hangman game), download it from https://github.com/dwyl/english-words page.
  - Calculate the statistics on the current kernel from https://www.kernel.org/.
- Calculate π by a parallel Monte Carlo method.
- Parallelize the matrix vector multiplication.

Good luck!