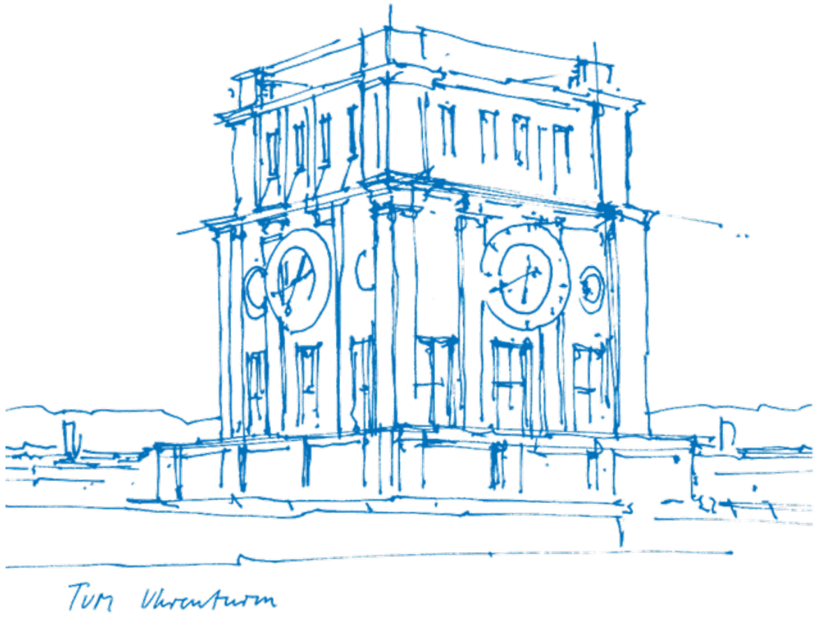
Parallel Programming Tutorial - Parallelism with Modern C++

Amir Raoofy, M.Sc.
Chair for Computer Architecture and Parallel Systems (Prof. Schulz)
Technichal University Munich
30. April 2018



Few organizational remarks

Few organizational remarks

- On Wednesday we have another tutorial on OpenMP basics
- Also Wednesday is the deadline for the second assignment.
- Again remember to check the lecture schedule:
  - Next Monday we will have a guest lecture from Intel/OpenMP ARB.
  - Next Wednesday we will have another tutorial on OpenMP advanced topics.
- Assignment 1 and 2; Did you have a problem? Were they easy? difficult?
- Did you go to the QA Sessions on Tuesdays
- Ask the tutors or send them email
  - Canberk: canberk.demirsoy(at)tum.de
  - Rakesh: rakesh.singh(at)tum.de
  - Vishnu: vishnu.anilkumar-suma(at)tum.de
- My email address is: amir.raoofy(at)tum.de

Recap

Recap

- Creating new threads with pthread_create
- Waiting for threads to finish with pthread_join
- Passing arguments to pthread kernels
- Returning results from pthread kernels

- Data hazards
- Thread Synchronization and locking in Pthreads
- Locking using Mutex

```c
#define NUM 10000000

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void * kernel_increment(void *i_void_ptr)
{
  int *i = (int *) i_void_ptr;

  for(int j=0; j < NUM; j++)
  {
    pthread_mutex_lock(&mutex);
    (*i)++;
    pthread_mutex_unlock(&mutex);
  }

    return NULL;
}
```

Solution for Assignment 1

Solution for Assignment 1

The master thread...

allocates threads and arguments

computes an iteration chunks for each thread

starts the other threads

deallocates threads and arguments

Each worker thread...

gets an iteration chunk

gets a pointer to image

executes the kernel

calculates the pixels

```c
typedef struct
{
    int start, end;
    void *image;
    int max_iter, palette_shift;
    int x_resolution, y_resolution;
    double view_x0, view_x1, view_y0, view_y1;
    double x_stepsize, y_stepsize;
} compute_args;
```

```c
void *compute_mandelbrot(void *arguments)
{
    compute_args *args = arguments;
    int start = args->start;
    int end = args->end;
    unsigned char(*image)[x_resolution1][3] =
    (unsigned char(*)[x_resolution1][3])args->image;
    int max_iter = args->max_iter;
    int palette_shift = args->palette_shift;
    int x_resolution1 = args->x_resolution;
    int y_resolution = args->y_resolution;
    double view_x0 = args->view_x0;
    // similar for the rest of the parameters ...

    for (int i = start; i < end; i++)
    {
        for (int j = 0; j < x_resolution1; j++)
        {
            // pixel calculation ...
        }
    }
}
```

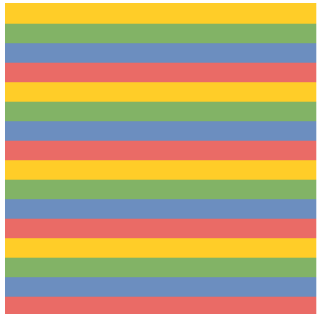
Solution for Assignment 1 (Cont.)

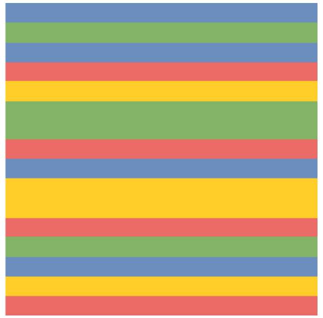
```c
void mandelbrot_draw( args ... )
{
    // pthread initialization
    pthread_t *thread = (pthread_t *)malloc(num_threads * sizeof(*thread));
    compute_args *thread_arg = (compute_args *)malloc(num_threads * sizeof(*thread_arg));
    int t, taskSize = y_resolution / num_threads;
    // set the arguments and execute the kernels
    for (t = 0; t < num_threads; t++)
   {
        thread_arg[t].start = t * taskSize;
        thread_arg[t].end = (t < (num_threads - 1)) ? (t + 1) * taskSize : (t + 1) * taskSize +
                            (y_resolution % num_threads);
        thread_arg[t].image = (void *)image;
        thread_arg[t].palette_shift = palette_shift;
        thread_arg[t].max_iter = max_iter;
        // similar for the rest of the parameters ...
        pthread_create(&thread[t], NULL, compute_mandelbrot, &thread_arg[t]);
    }
    for (t = 0; t < num_threads; t++)
        pthread_join(thread[t], NULL);

    free(thread);
    free(thread_arg);
}
```

Hints for Assignment 2

Hints for Assignment 2

- Use a profiler! (see last session)
- Try to reduce the critical region. **That is the bottleneck!**
- Think about load balancing issues.
- Think about the dynamic distribution of the loop.
  - How can you force all threads to work on all the regions of the image?
  - How can you assign small chunks of iterations to the worker threads?
  - How can you prevent different threads from working on the same chunk?





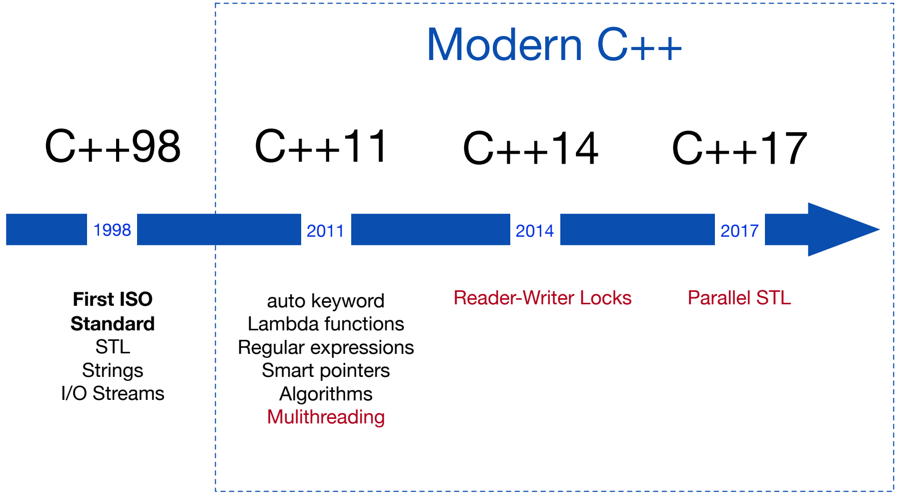
Parallelism with Modern C++

Parallelism with Modern C++

C++11 introduced multithreading concepts

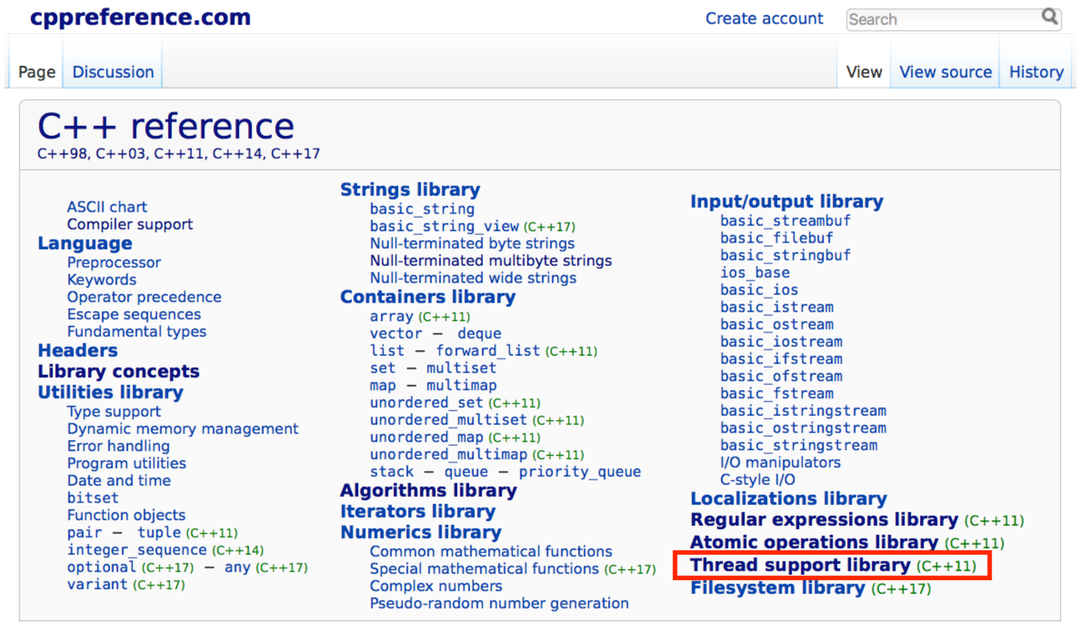
- A well defined memory model
  - Atomic operations
  - Partial ordering of operations
- A standardized threading interface
  - Type-safety!
  - Threads and tasks
  - Shared memory handling
  - Thread-local data
  - Synchronization

C++17 brings abstractions for parallelism
C++20 ...



Disclaimer

This is no in-depth tutorial, read the reference!



cppreference.com

Create account | Search

Page | Discussion | View | View source | History

C++ reference

C++98, C++03, C++11, C++14, C++17

ASCII chart
Compiler support

Language
- Preprocessor
- Keywords
- Operator precedence
- Escape sequences
- Fundamental types

Headers

Library concepts

Utilities library
- Type support
- Dynamic memory management
- Error handling
- Program utilities
- Date and time
- bitset
- Function objects
- pair — tuple (C++11)
- integer_sequence (C++14)
- optional (C++17) — any (C++17)
- variant (C++17)

Strings library
- basic_string
- basic_string_view (C++17)
- Null-terminated byte strings
- Null-terminated multibyte strings
- Null-terminated wide strings

Containers library
- array (C++11)
- vector — deque
- list — forward_list (C++11)
- set — multiset
- map — multimap
- unordered_set (C++11)
- unordered_multiset (C++11)
- unordered_map (C++11)
- unordered_multimap (C++11)
- stack — queue — priority_queue

Algorithms library

Iterators library

Numerics library
- Common mathematical functions
- Special mathematical functions (C++17)
- Complex numbers
- Pseudo-random number generation

Input/output library
- basic_streambuf
- basic_filebuf
- basic_stringbuf
- ios_base
- basic_ios
- basic_istream
- basic_ostream
- basic_iostream
- basic_ifstream
- basic_ofstream
- basic_fstream
- basic_istringstream
- basic_ostringstream
- basic_stringstream
- I/O manipulators
- C-style I/O

Localizations library

Regular expressions library (C++11)

Atomic operations library (C++11)

Thread support library (C++11)

Filesystem library (C++17)

Threads in Modern C++

- A `std::thread` is defined in <thread> and gets a callable object to work on immediately.
- A callable object can be...
  - a function: `std::thread t( func );`
  - a function object: `std::thread t( fobj() );`
  - a lambda function: `std::thread t( [](int a, int b){ return a + b; } );`
- The creating thread has to...
  - wait for the child thread t with `t.join()`
  - defer t with `t.detach()` $\rightarrow$ daemon thread
- A thread t is joinable, if there was no `t.join()` or `t.detach()`
- A joinable thread calls an exception (`std::terminate()`) on its destructor, thus `t.detach()`
- A `std::thread` gets data as reference (with `std::(c)ref()`) or copy.

Methods for `std::thread`

- `t.joinable()`
  Checks if a thread t is joinable
- `t.get_id()` / `std::this_thread::get_id()`
  Returns the thread identifier of a thread t / the current thread
- `std::thread::hardware_concurrency()`
  Returns the number of threads which can be started
- `std::this_thread::sleep_until(abs_time)`
  Lets the thread sleep until a time stamp
- `std::this_thread::sleep_for(rel_time)`
  Lets the thread sleep for a duration
- `std::this_thread::yield()`
  Allows the OS to execute another thread
- `t.swap(t2)`, `std::swap(t1, t2)`
  Exchanges the underlying handles of two thread objects
- `t.native_handle()`
  Returns the implementation defined underlying thread handle

Example 1: Passing parameters to a `std::thread`

```cpp
#include <iostream>
#include <thread>
#include <string>

void thread_function(std::string& s)
{
    s += " and white";
    std::cout << "message is = " << s << std::endl;
}

int main()
{
    std::string s = "The cat is black";
    std::thread t(&thread_function, std::ref(s));
    t.join();
    std::cout << "main message = " << s << std::endl;
}
```

- Creates and joins a thread
- Passes a string as reference to the child thread

```
./example
message is = The cat is black and white
main message = The cat is black and white
```

Example 2: Without std::join()

```cpp
#include <iostream>
#include <thread>
#include <string>

void thread_function(std::string& s)
{
    s += " and white";
    std::cout << "message is = " << s << std::endl;
}

int main()
{
    std::string s = "The cat is black";
    std::thread t(&thread_function, std::ref(s));
    //t.join();
    std::cout << "main message = " << s << std::endl;
}
```

- The joinable thread calls `std::terminate()` in destructor

```
./example2
main message = The cat is black
terminate called without an active exception
message is = The cat is black and white
Aborted (core dumped)
```

Example 3: std::detach()

```cpp
#include <iostream>
#include <thread>
#include <string>

void thread_function(std::string& s)
{
    s += " and white";
    std::cout << "message is = " << s << std::endl;
}

int main()
{
    std::string s = "The cat is black";
    std::thread t(&thread_function, std::ref(s));
    t.detach();
    std::cout << "main message = " << s << std::endl;
}
```

- The second thread does not have the time to write to std::cout

```
./example3
main message = The cat is black
```

Example 4: Join again?

```cpp
#include <iostream>
#include <thread>
#include <string>

void thread_function(std::string& s)
{
    s += " and white";
    std::cout << "message is = " << s << std::endl;
}

int main()
{
    std::string s = "The cat is black";
    std::thread t(&thread_function, std::ref(s));
    t.detach();
    std::cout << "main message = " << s << std::endl;
    t.join();

}
```

- The thread is not joinable after `std::detach()`

```
./example4
main message = The cat is black
message is = The cat is black and white
terminate called after throwing an instance of 'std
  what():  Invalid argument
Aborted (core dumped)
```

Example 5: What happens if main thread waits?

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

void thread_function(std::string& s)
{
    s += " and white";
    std::cout << "message is = " << s << std::endl;
}

int main()
{
    std::string s = "The cat is black";
    std::thread t(&thread_function, std::ref(s));
    t.detach();
    std::cout << "main message = " << s << std::endl;
    //t.join();
    std::this_thread::sleep_for(std::chrono::seconds(1));

}
```

- Is this a good idea?
- Is it deterministic?

```
./example5
main message = The cat is black
message is = The cat is black and white
```

Synchronization with Modern C++

Shared Memory Handling in Modern C++

- C++11 ensures that in- and output-streams do not have to be protected
  → Writing to `std::cout` is thread-safe
- **Mutexes** (defined in `<mutex>`)
  - Variations: `std::mutex`, `std::recursive_mutex`, `std::timed_mutex`, `std::recursive_timed_mutex`
  - `m.lock()` locks a mutex m
  - `m.unlock()` unlocks a mutex m
  - `m.try_lock()` tries to lock a mutex m. Returns true on success, otherwise false
  - `m.try_lock_for(rel_time)` tries to lock a mutex m for a duration
  - `m.try_lock_until(abs_time)` tries to lock a mutex until a time stamp
- **Lock guards** and **unique locks** (defined in `<mutex>`)
  - `std::lock_guard(m)` locks m automatically and releases it on destruction
  - `std::unique_lock` is more sophisticated and can
    ...be created without a mutex.
    ...explicitly set or release a lock.
    ...be used with timing semantics.

Example: Mutexes and Locks

```cpp
std::list<int> myList; // a global variable
std::mutex myMutex;    // a global mutex

void addToList(int start) {

    std::lock_guard<std::mutex> guard(myMutex);
    for (int i = start; i < start + 10; i++) {
        myList.push_back(i);
        std::this_thread::\
        sleep_for(std::chrono::milliseconds(2));
    }
}

int main() {
    std::thread t1(addToList, 0);
    std::thread t2(addToList, 10);
    t1.join();
    t2.join();

    for (auto& item : myList)
        std::cout << item << ",";
}
```

- Two threads are pushing elements to a list
- To trigger a race condition, each thread sleeps for 2ms after a push operation
- A lock guard is used to lock a mutex in the scope before pushing elements
- **Result:** Thread t1 pushes all elements before t2
- **Or:** Thread t2 pushes all elements before t1

```
./mutexes
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,

./mutexes
10,11,12,13,14,15,16,17,18,19,0,1,2,3,4,5,6,7,8,9,
```

Tasks in Modern C++

Tasks

”Often, we would like to provide a lot of small tasks and let the system worry about how to map their execution onto hardware resources and how to keep them out of problems with data races, etc.“ B. Stroustrup

- A task is an abstraction of an executable piece of work that may map to a thread
- You should prefer task-based concurrency to thread-based concurrency because...
  - you do not want to rely on the hardware architecture below (#cores, #hw-threads, caches..)
  - you can rely on the runtime scheduler to gain better scalability
  - you can deal with shared state by return values and communication channels
  - you can deal with exceptions
- There may be situations where you want to use threads
  - provide access to lower-level handles (pthread, windows threads...)
  - gain top performance if you know the execution profile
  - you need to implement your own threading technology

Threads vs. Tasks in Modern C++

Thread

```cpp
int res;
std::thread t( [&]{ res = 3 + 4 }; );
t.join();
std::cout << res;
```

Task

```cpp
auto fut = std::async( []{ return 3 + 4; } );
std::cout << f.get();
```

Criteria	**Thread**	**Task**
Header	`<thread>`	`<future>`
Actors	creating-thread and child-thread	promise and future
Communication	shared variables	channel
Thread	obligatory	optional
Synchronization	`t.join()`	`f.get()`
Exceptions	threads terminate	return values

Asynchrounous Tasks

`std::async` is like an asynchronous function call.

```cpp
int a = 20, b = 10;
auto fut = std::async( [=]{ return a + b; } );
std::cout << "a + b = " << fut.get();
```

- `std::future`
  - `fut.get()` returns the shared state if it is available, othrewise it waits for it
  - `fut.wait()` wait until the shared state is ready
  - `fut.valid()` checks if the shared state is ready
  - `fut.wait_for(rel_time)`, `fut.wait_until(abs_time)`
- `std::future<..> std::async( (std::launch_policy), Function&& f, Args&&.. args )`
  - std::async executes f(args) according to a launch policy
  - f can be any callable unit (function, function object, lambda function...)
  - args can contain any number of real arguments
  - returns a `std::future` to retrieve the result
  - the launch policy can be `std::launch::async` or `std::launch::deferred`
    async ensures that the task is executed on another thread
    deferred executes the task when it is required by `get()` by the same thread
    attention: the default policy is (async | deferred), i.e., we can not assume that the thread was executed at all

Communication Channels of Tasks

Using `std::async` is only a shortcut! If we need more control about communication between tasks we have to use promise/future pairs.

Value

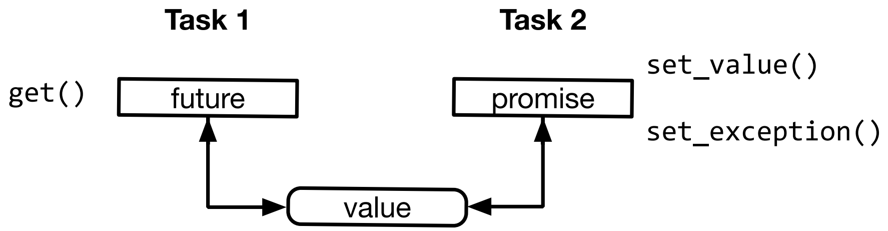
- Is the shared state, the information that is safeley exchanged
- Can have a specific type, a exception, or void
- Is set by the promise and consumed by the future

Promise

- is where a task can deposit its result for futures
- can send data, exceptions, or notifications (`void`)

Future

- Is the handle to the shared state.
- Is where a task can retrieve a result deposited by a promise
- Calling `get()` blocks until the result is ready.
- `get()` can only be called once. If multiple tasks need the result of a promise, use `shared_future`



Methods for `std::promise` and `std::future`

std::promise

- `p.get_future()`
  Returns the `std::future` object
- `p.set_value(val)`
  Sets the value of the shared state
- `set_exception(exc)`
  Sets the exception of the shared state
- `p.set_value_at_thread_exit(val)`
  Stores the value and puts it into the shared state on exit
- `p.set_exception_at_thread_exit(exc)`
  Stores the exception and puts it into the shared state on exit

std::future

- `fut.share()`
  Returns a `std::shared_future` object.
- `fut.get()`
  Returns the shared state if available, otherwise it waits on it
- `fut.valid()`
  Checks if the shared state is ready
- `fut.wait()`
  Wait until the shared state is ready
- `fut.wait_for(rel_time)`, `fut.wait_until(abs_time)`
  Waits for a duration or until a time stamp

Example: Promises and Futures (get value from thread)

```cpp
void product(std::promise<int>&& p, int a, int b) {
  p.set_value(a * b);  // set value of shared state
}

int main()
{
  int a = 20, b = 10;

  // create promise and get future object
  std::promise<int> myPromise;
  std::future<int> myFuture = myPromise.get_future();

  // start task on a new thread giving the promise
  std::thread myThread(product,
                       std::move(myPromise),
                       a, b);

  // getting the shared state (may wait)
  std::cout << "20 * 10 = " << myFuture.get() << "\n";
  myThread.join();
}
```

- `product` is executed in a new task by a spearate thread
- We defined a promise `myPromise`
- We obtained the corresponding future `myFuture`
- We started the new thread and passed `myPromise` as argument
- `product` sets the value of the shared state
- The main thread waits on the result (shared state) by `get()`
- Can you come up with an idea to send values to threads using promises?

```
./promises\_futures
20 * 10 = 200
```

Parallel STL in C++17

adjacent_difference	adjacent_find	all_of	any_of
copy	copy_if	copy_n	count
count_if	equal	exclusive_scan	fill
fill_n	find	find_end	find_first_of
find_if	find_if_not	for_each	for_each_n
generate	generate_n	includes	inclusive_scan
inner_product	inplace_merge	is_heap	is_heap_until
is_partitioned	is_sorted	is_sorted_until	lexicographical_compare
max_element	merge	min_element	minmax_element
mismatch	move	none_of	nth_element
partial_sort	partial_sort_copy	partition	partition_copy
reduce	remove	remove_copy	remove_copy_if
remove_if	replace	replace_copy	replace_copy_if
replace_if	reverse	reverse_copy	rotate
rotate_copy	search	search_n	set_difference
set_intersection	set_symmetric_difference	set_union	sort
stable_partition	stable_sort	swap_ranges	transform
transform_exclusive_scan	transform_inclusive_scan	transform_reduce	uninitialized_copy
uninitialized_copy_n	uninitialized_fill	uninitialized_fill_n	unique
unique_copy			

What we didn't cover...

- The memory model of modern C++
- Condition variables
- Packaged tasks
- ...

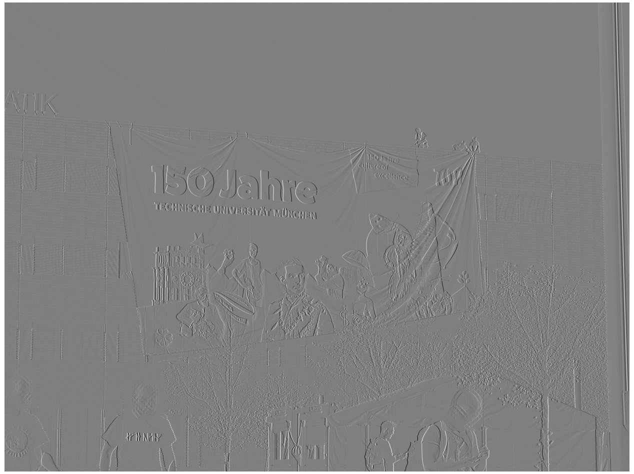
`http://en.cppreference.com/w/cpp/thread`

`http://en.cppreference.com/w/cpp/algorithm`

`https://goo.gl/vMcujN` `https://goo.gl/cQDa7L`

Assignment 3 - Edge detection (Modern c++)

Assignment 3 - Edge detection (Modern c++)



Assignment 3 - Edge detection (Modern c++) CHECK FOR UPDATES

- You have two weeks time for this assignment
- You can either use `std::async` or `std::thread` directly
- no valgrind, helgrind, #threads...
- The speedup with 32 cores must be at least 10
- Consider:
  - Previous strategies may apply here

Assignment 3 - Edge detection (Modern c++) - x_gradient()

```cpp
template <typename SrcView, typename DstView>
void x_gradient(const SrcView &src, const DstView &dst, int num_threads)
{
    typedef typename channel_type<DstView>::type dst_channel_t;

    for (int y = 0; y < src.height(); ++y)
    {
        typename SrcView::x_iterator src_it = src.row_begin(y);
        typename DstView::x_iterator dst_it = dst.row_begin(y);

        for (int x = 1; x < src.width() - 1; ++x)
        {
            static_transform(src_it[x - 1], src_it[x + 1], dst_it[x],
                             halfdiff_cast_channels<dst_channel_t>());
        }
    }
}
```

Assignment 3 - Edge detection (Modern c++) - Provided Files

- Makefile
  - contains rules to build executables
  - available targets: parallel, sequential, all (default), clean
  - 'mode=debug make [target]' to build debug version, use 'make clean' before
- main.c
  - main function - argument handling + file handling + call `x_luminosity_gradient()`
  - `x_luminosity_gradient()` calls `x_gradient()`
  - you implement the parallel version of `x_gradient()`
- x_gradient.h
  - Header file for `x_luminosity_gradient()`
- x_gradient_seq.h
  - Sequential version of `x_gradient()`
- student/x_gradient_par.h
  - Implement the parallel version in this file
- unit_test.c
  - The unit tests that execute both the serial and parallel version to compare results.

Assignment 3 - Edge detection (Modern c++) - Compilation and execution

- Compilation
  - You need to install libjpeg, boost and boost/gil
  - `make [all] [sequential] [parallel] [unit_test]`
  - You implement your solution in a header file
  - You have to `make clean` every time and `make` again
- Execution
  - `./student/x_gradient_seq`
  - `./student/x_gradient_par -t 4 -f tum.jpg`
  - `./student/unit_test`