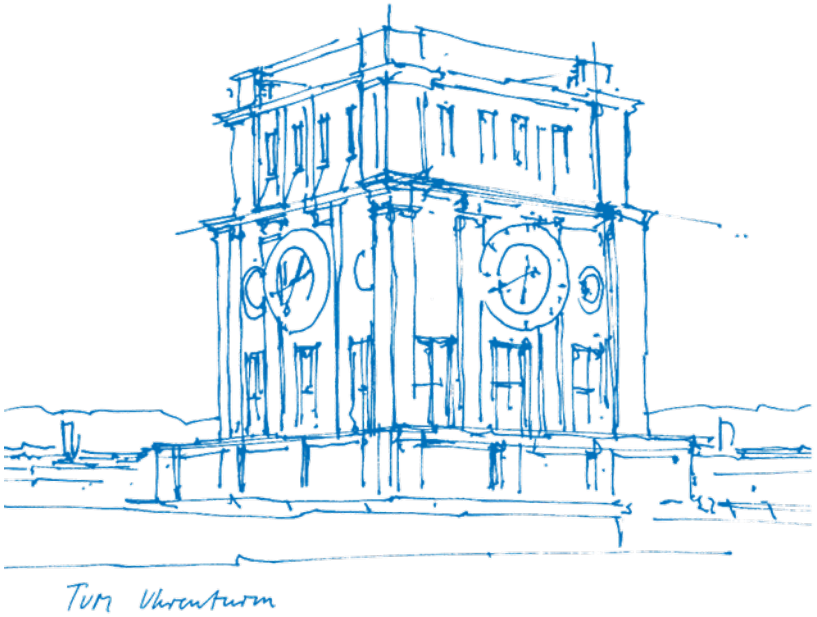
Parallel Programming Tutorial - Parallelism with Modern C++

M.Sc. Andreas Wilhelm
Technichal University Munich
May 22, 2017



Solution for Assignment 1

Solution for Assignment 1 (1/2)

The master thread...
allocates threads and arguments
computes number of blocks per thread
computes start and end block numbers
collects the local histograms
deallocates the local histograms
Each worker thread...
gets a start and end block number
gets a pointer to all blocks
allocates a local initialized histogram
returns this histogram as argument

```
typedef struct {
    int start;
    int end;
    block_t *blocks;
    int *histogram;
} compute_args;
```

```
void *compute_histogram(void *args) {
    char current_word[20] = "";
    compute_args *ca = (compute_args *) args;
    ca->histogram = (int *) calloc(NNAMES, sizeof(int));

    for (int k = 0, i = ca->start; i < ca->end; ++i) {
        for (int j = 0; j < BLOCKSIZE; ++j) {
            if (isalpha(ca->blocks[i][j])) {
                current_word[k] = ca->blocks[i][j];
                k++;
            } else {
                current_word[k] = '\0';
                for (int n = 0; n < NNAMES; ++n) {
                    if (!strcmp(current_word, names[n]))
                        ca->histogram[n]++;
                }
                k = 0;
            }
        }
    }
    return NULL;
}
```

Solution for Assignment 1 (2/2)

```
void get_histogram(int nBlocks, block_t *blocks, histogram_t histogram, int num_threads) {
    pthread_t* thread = (pthread_t *) malloc(num_threads * sizeof(*thread));
    compute_args* thread_arg = (compute_args *) malloc(num_threads * sizeof(*thread_arg));

    int taskSize = nBlocks / num_threads;
    for (int 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+(nBlocks % num_threads);
        thread_arg[t].blocks = blocks;
        pthread_create(&thread[t], NULL, &compute_histogram, &thread_arg[t]);
    }

    for (int t = 0; t < num_threads; t++) {
        pthread_join(thread[t], NULL);
        for (int i = 0; i < NNAMES; i++)
            histogram[i] += thread_arg[t].histogram[i];
        free(thread_arg[t].histogram);
    }
    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!**
- Grab multiple blocks (think about cache sizes)

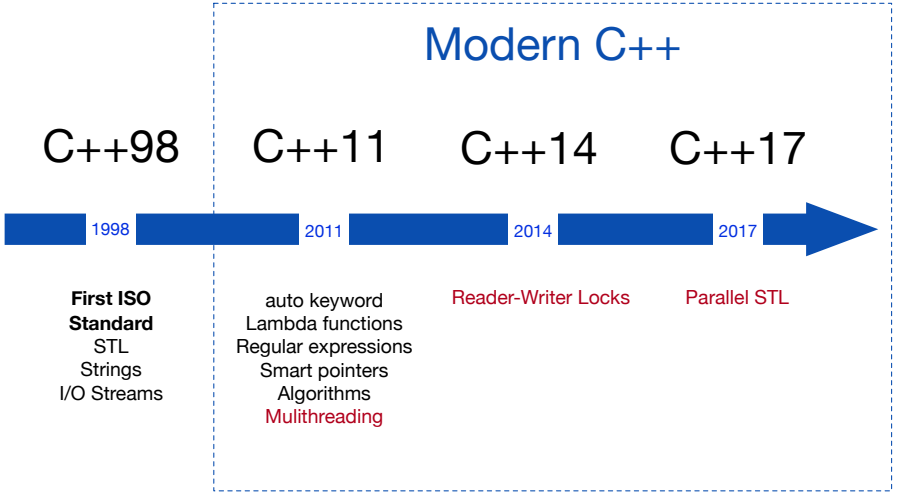
Parallelism with Modern C++

Parallelism with Modern C++

C++11 introduced multithreading concepts

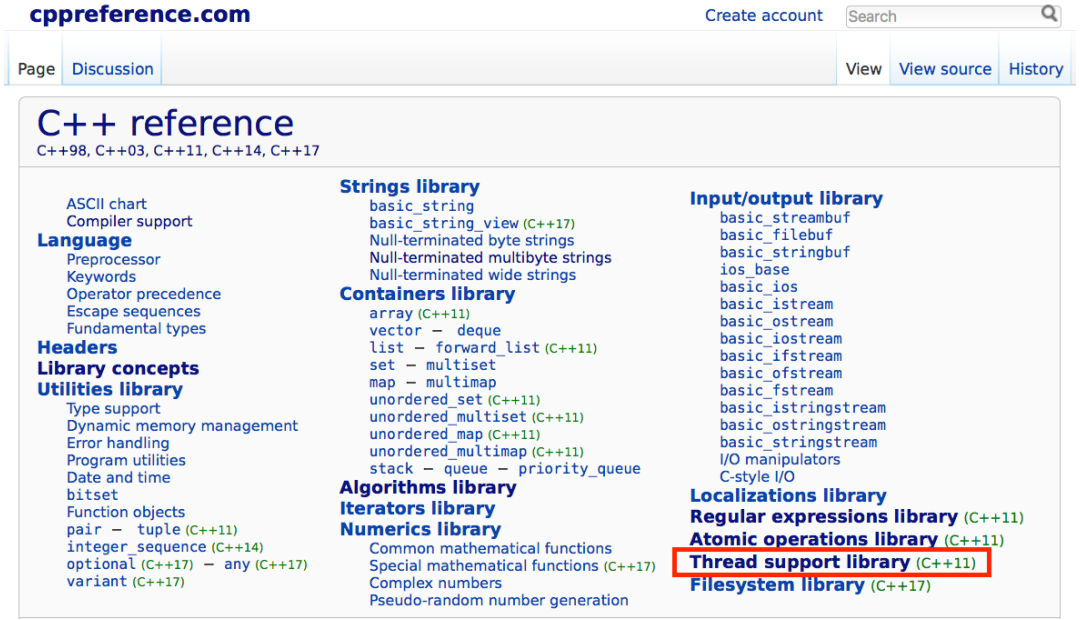
- A well defined memory model
  - Atomic operations
  - Partial ordering of operations
- A standardized threading interface
  - Type-safety!
  - Threads and tasks
  - Shared memory handling
  - Thread-local data
  - Synchronization

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



Disclaimer

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



One Word about the Memory Model

```
// Thread 1
char x = 0; // global variable x

void foo()
{
   x = 1;
}
```

```
// Thread 2
char y = 0; // global variable y

void bar()
{
   y = 1;
}
```

- What are the values of `x` and `y` after executing `foo` and `bar`?

One Word about the Memory Model

```
// Thread 1
char x = 0; // global variable x

void foo()
{
   x = 1;
}
```

```
// Thread 2
char y = 0; // global variable y

void bar()
{
   y = 1;
}
```

- What are the values of `x` and `y` after executing `foo` and `bar` (in pre-Modern-C++ times)?
- `Answer:`

  `x == 1` `y == 1` or

  `x == 1` `y == 0` or

  `x == 0` `y == 1`
- The linker may allocate `x` and `y` in the same word in memory.
- Without proper memory model the threads might read the old value of the other thread and overwrite the new one.
- Therefore, modern C++ has a contract to avoid problems like this... The memory model.

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()` → 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: 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

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

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

void addToList(int start) {
   using namespace std::chrono_literals;

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

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

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

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

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

Task

```
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.

```
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

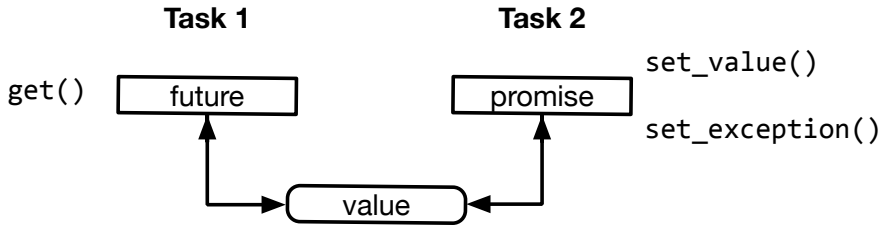
- Is the shared state, the information that is safeley exchanged
- Can have a specific type, a exception, or void
- Is set by the promise and consumed by the future

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

```
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 creating thread waits on the result (shared state) by `get()`

```
./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 - histogram (Modern c++)

Assignment 2: histogram (Modern C++)

- You have one week time for this assignment, hand in with assignment 2
- Use C++ tasks to parallelize `get_histogram()`
- You can either use `std::async` or `std::thread` directly
- We only check if you use std::futures, no valgrind, helgrind, threads...
- The speedup with 32 cores must be at least 10
- This time, we have a different signature:

```
void get_histogram (
     const std::vector<word_t>& words,  // a vector of std::<array, 8> words
     histogram_t& histogram             // the histogram of type std::array<int, 10>
     int num_threads                    // the number of threads you should use
)
```

- Consider:
  - Handling one word per task may be unpractical
  - Previous strategies may apply here

Assignment 3: histogram (Modern C++) - get_histogram()

```
#include "histogram.h"
#include <algorithm>
#include "names.h"

void get_histogram(const std::vector<word_t>& words, histogram_t& histogram, int num_threads)
{
    for_each(begin(words), end(words), [&histogram](const word_t& word)
    {
        int res = getNameIndex(word.data());
        if (res != -1) histogram[res]++;
    });
}
```

Assignment: histogram (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 `get_histogram()`
- histogram.h
  - Header file for histogram.c and histogram_*.c
- histogram.c
  - Defines helper functions
- histogram_seq.c
  - Sequential version of `get_histogram()`.
- student/histogram_par.c
  - Implement the parallel version in this file

Assignment: histogram (Modern c++) - Provided Files

- names.c / .h
  - Contains `getNameIndex()` to return the name index or -1.
- war_and_peace.txt
  - Input data: The book war and peace.
- unit_test.c
  - The unit tests that execute both the serial and parallel version to compare results.