Parallel Programming with OpenMP and Modern C++ Alternatives

Michael F. Hava and Bernhard M. Gruber

RISC Software GmbH
Softwarepark 35, 4232 Hagenberg, Austria
Overview

- Threading Theory
- OpenMP 2.0
- C++11 Threading Facility
- Microsoft Parallel Patterns Library (PPL) and Intel Threading Building Blocks (TBB)
- C++17 Threading Facility
Threading Theory
Definitions

Process
- discrete programming task
- own address space
- each process contains at least one thread

Thread
- sequence of related instructions
- independent of other instruction sequences
- threads share address space within a process
  - shared memory model
Reasons for Threading

- better utilization of CPU resources
- faster processing of tasks
- responsive applications
- parallelization is the only way CPU performance is increasing
Speedup and Efficiency

- speedup:
  \[ S_n = \frac{T_1}{T_n} \]

- efficiency:
  \[ E_n = \frac{S_n}{n} = \frac{T_1}{n \cdot T_n} \]

- \( n \) ... number of processors
- \( T_1 \) ... sequential execution time
- \( T_n \) ... parallel execution time
Categories of Threads

- **application** (fiber)
  - managed by application
  - cooperative scheduling
  - executed on kernel threads

- **kernel** (thread)
  - managed by OS
  - preemptive scheduling
  - executed on HW-threads

- **hardware** (core)
  - past: 1 CPU == 1 thread
  - now: 1 CPU == N threads
Concurrency vs. Parallelism

Concurrency

Parallelism

T1.1

T2.1

T1.2

T2.2

T1.1

T2.1

T1.2

T2.2
Shared Memory Model (SMM)

- threads share address space
  - private stack (theoretically!)
  - shared access to
    - heap
    - global variables
    - ...

- access time for processors may be different
  - UMA uniform memory access
  - NUMA non-uniform memory access
Advantages

- simple programming model
  - exactly the same as sequential programming

- local parallelization

- data is easily & cheaply shared between threads

Disadvantages

- limited scalability
  - memory bandwidth is the key limiting factor

- not applicable to distributed systems

- explicit synchronization of threads is required
SMM – Race Condition

Thread 1

```c
++x;
x = x + 1;

LOAD [x], reg_0
INC reg_0
STORE reg_0, [x]
```

Thread 2

```c
--x;
x = x - 1;

LOAD [x], reg_0
DEC reg_0
STORE reg_0, [x]
```
# SMM – Race Condition

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th></th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD ([x], \text{reg}_0)</td>
<td>(\leftarrow)</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>LOAD ([x], \text{reg}_0)</td>
<td>(\leftarrow)</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>INC \text{reg}_0</td>
<td>(\leftarrow)</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>DEC \text{reg}_0</td>
<td>(\leftarrow)</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>STORE \text{reg}_0, ([x])</td>
<td>(\rightarrow)</td>
<td></td>
<td>-1</td>
</tr>
<tr>
<td>STORE \text{reg}_0, ([x])</td>
<td>(\rightarrow)</td>
<td></td>
<td>+1</td>
</tr>
</tbody>
</table>
SMM – Critical Section

- critical section: part of a program that may not be executed by different threads at the same time

- access to shared resources must be secured
  - otherwise race conditions may occur

- access in critical sections must be synchronized
  - atomic operations
  - locks (e.g. semaphore, mutex, ...)
# SMM – Atomic Operations

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>ATOMIC INC [x]</th>
<th>ATOMIC DEC [x]</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic_inc(x);</td>
<td>atomic_dec(x);</td>
<td>«blocked»</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>«blocked»</td>
<td>ATOMIC INC [x]</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>ATOMIC DEC [x]</td>
<td>++</td>
<td>←→</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
SMM – Semaphore and Mutex

- Edsger Dijkstra, 1968
- securing critical section using a counter N
  - remaining number of threads that may enter a critical section
- two atomic operations interact with the semaphore
  - P ... try to enter critical section (conditional decrement)
  - V ... leave critical section (increment)

- mutex works similar to a semaphore N=1
  - P is called lock, V is called unlock
  - has strict ownership semantics (owned by the thread which locks it)
SMM – Mutex

Thread 1

Mutex m;
lock(m);
++x;
unlock(m);

Thread 2

Mutex m;
lock(m);
--x;
unlock(m);

never use locks for trivial critical sections!
## SMM – Mutex

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock(m)</td>
<td>lock(m)</td>
<td>0</td>
</tr>
<tr>
<td>LOAD [x], reg₀</td>
<td>«blocked»</td>
<td></td>
</tr>
<tr>
<td>INC reg₀</td>
<td>«blocked»</td>
<td></td>
</tr>
<tr>
<td>STORE reg₀, [x]</td>
<td>«blocked»</td>
<td>1</td>
</tr>
<tr>
<td>unlock(m)</td>
<td>«blocked»</td>
<td>1</td>
</tr>
<tr>
<td>LOAD [x], reg₀</td>
<td>«blocked»</td>
<td>1</td>
</tr>
<tr>
<td>DEC reg₀</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>STORE reg₀, [x]</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>unlock(m)</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
SMM – Deadlock

Thread 1

lock(m1);
lock(m2);
...
unlock(m2);
unlock(m1);

Thread 2

lock(m2);
lock(m1);
...
unlock(m1);
unlock(m2);

deadlock

- order of locks is essential!
  - responsibility of the programmer
OpenMP 2.0
OpenMP

- open industry standard for parallelization
  - cross platform
  - cross programming language (C, C++, Fortran)

- managed by the OpenMP Architecture Review Board

- ongoing development
  - first released in 1997
  - latest version (4.5) was released in November 2015
- focus on (retroactive) incremental parallelization
Fundamentals

- parallelization is based on compiler support
  - expressing parallel workloads by compiler directives
  - `#pragma omp directive-name clause`

- API to interact with the abstracted threading model
  - `#include <omp.h>`
  - `omp_get_max_threads()` max number of threads for new group
  - `omp_get_thread_num()` current thread’s id $\in [0, \text{MAX})$ in group
  - `omp_get_num_threads()` current number of threads in group
Parallel Region

#pragma omp parallel
{
    const int tid = omp_get_thread_num();
    printf("The parallel region is executed by "
           "thread %d\n", tid);
    if(!tid) printf("tThread %d does things"
                     " differently\n", tid);
}

The parallel region is executed by thread 1
The parallel region is executed by thread 7
The parallel region is executed by thread 6
The parallel region is executed by thread 0
    Thread 0 does things differently
The parallel region is executed by thread 4
The parallel region is executed by thread 5
The parallel region is executed by thread 3
The parallel region is executed by thread 2
Worksharing Directives

- distribution of work among threads
  - `#pragma omp for`
  - `#pragma omp sections`
  - `#pragma omp single`
  - outside parallel scope → serial computation

- shorthand notation for parallel regions with single directive
  - `#pragma omp parallel for`
  - `#pragma omp parallel sections`
  - improved readability
Worksharing – for

```c
#pragma omp parallel
#pragma omp for
for(int i = 0; i < 10; ++i)
    printf("Thread %d executes loop iteration\n", omp_get_thread_num(), i);
```

<table>
<thead>
<tr>
<th>Thread</th>
<th>Loop Iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>
Worksharing – sections

```c
#pragma omp parallel
#pragma omp sections
{
    #pragma omp section
    printf("section 1: executed by thread %d\n", omp_get_thread_num());
    #pragma omp section
    printf("section 2: executed by thread %d\n", omp_get_thread_num());
}
```

section 1: executed by thread 2
section 2: executed by thread 3
Worksharing – single

```c
#pragma omp parallel
{
    const int tid = omp_get_thread_num();
    #pragma omp single
    printf("Thread %d executes single\n", tid);
    printf("Hello from thread %d\n", tid);
}
```

only one thread executes the next statement

<table>
<thead>
<tr>
<th>Thread 1 executes single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hello from thread 5</td>
</tr>
<tr>
<td>Hello from thread 6</td>
</tr>
<tr>
<td>Hello from thread 7</td>
</tr>
<tr>
<td>Hello from thread 0</td>
</tr>
<tr>
<td>Hello from thread 4</td>
</tr>
<tr>
<td>Hello from thread 1</td>
</tr>
<tr>
<td>Hello from thread 3</td>
</tr>
<tr>
<td>Hello from thread 2</td>
</tr>
</tbody>
</table>
Synchronization Directives & Clauses

- implicit barrier after(!) every worksharing directive
  - can be removed with **nowait**-clause
  - explicit: `#pragma omp barrier`

- explicit synchronization of statements
  - `#pragma omp critical`
  - `#pragma omp atomic`
  - `#pragma omp for reduction`
Synchronization – nowait

```c
#pragma omp parallel
{
    const int tid = omp_get_thread_num();
    #pragma omp single nowait
    printf("Thread %d executes single\n", tid);
    printf("Hello from thread %d\n", tid);
}
```

removes barrier
Synchronization – critical

```c
int sum = 0;
#pragma omp parallel for
for(int i = 0; i < 10; ++i) {
    pragma omp critical (sumMutex)
    {
        lock mutex "sumMutex"
        sum += (int)pow(i, 2);
    }
    unlock mutex "sumMutex"
}
printf("in parallel computed sum: %d\n", sum);
```
Synchronization – atomic

```c
int sum = 0;
#pragma omp parallel for
for(int i = 0; i < 10; ++i) {
    #pragma omp atomic
    sum += (int)pow(i, 2);
}

printf("in parallel computed sum: %d\n", sum);
```

function call is parallel!
in parallel computed sum: 285
Synchronization – reduction

```c
int sum = 0;
#pragma omp parallel for reduction(+:sum)
for(int i = 0; i < 10; ++i) {
    sum += (int)pow(i, 2);
}
printf("in parallel computed sum: %d\n", sum);
```
Scheduling

- distribution of loop iterations across threads is controllable:
  - `omp [parallel] for schedule(kind [, chunk_size])`

- kinds:
  - static
  - dynamic
  - guided
  - runtime
    - controlled by environment variable or API functions (3.0)
Scheduling – static vs. static(4)
Scheduling – dynamic vs. dynamic(4)
Scheduling – guided vs. guided(4)
Extensions in Recent Standard Revisions

- tasks [3.0]
- SIMD support [4.0]
- thread teams [4.0]
- accelerator support [4.0]
  - GPUs, Xeon Phi, ...
Hazards using OpenMP

- OpenMP expects a serial program
  - interactions with other threading APIs are undefined!
  - fork-join may only happen on the initial/main thread!

- OpenMP 4.5 still only refers to old language standards:
  - C: 1990, 1999
  - C++: 1998

- C++ is only supported with C-semantics!
C++11 Threading
C++ Standard Threading

- C++11 defines the basis for portable multithreading
  - design has also been adopted by C11

- basic building blocks for parallelization and synchronization
  - threads `<thread>`
  - atomics `<atomic>`
  - locks `<mutex>`
  - futures `<future>`

- future extensions (C++17/20):
  - parallel algorithms, futures with continuations
Threads

- threads are implemented as standard objects
  ```cpp
  thread t{} { cout << "Hello\n"; }
  ...
t.join();
  ```

- query the number of available hardware threads
  ```cpp
  thread::hardware_concurrency()
  ```

- Interaction with own thread is available via
  ```cpp
  namespace this_thread
  ```
auto func = [](int tid) {
    cout << "thread " + to_string(tid) + '\n';
};
vector<thread> ts;
const auto n = thread::hardware_concurrency();
for(unsigned int i = 0; i < n; i++)
    ts.push_back(thread{func, i});
for(auto& t : ts)
    t.join();
Locks

- **mutex**
  - lock(), try_lock(), unlock()

- **timed_mutex**
  - try_lock_for(), try_lock_until()

- **recursive_mutex**
  - reentrant: owning thread can lock() and unlock() several times

- **shared_mutex**
  - [lock|try_lock|unlock]_shared()
  - multiple threads can lock_shared(), single thread can lock()

- **Combinations**
  - [recursive_|shared_|timed_]mutex
Scope Guards

- automate locking and unlocking
- exception-aware
- based on RAII-pattern
  - constructor locks mutex
  - destructor unlocks mutex

- `lock_guard<T>`
- `unique_lock<T>`
- `shared_lock<T>`
Locks

```cpp
int x{0};
mutex m;
using guard = lock_guard<mutex>;

Thread 1
{
  guard lock{m};
  ++x;
}

Thread 2
{
  guard lock{m};
  --x;
  throw exception{};
}

x == 0
```
std::atomic<T>

- wrapper to provide atomic access to a value
  - load(), store(), exchange(), compare_exchange()
  - if T is integral: fetch_add(), fetch_or(), ++, +=, |=, ...
- T must be trivially copyable
  - structs with no custom constructor or assignment operator
  - int, double[100], struct { bool a; float b; }
- if possible, atomic CPU instructions are used
  - otherwise locks are used (VS2015 uses spin lock)
- primitive for building concurrent data structures
  - use those instead of atomics directly!
# std::atomic<T>

```cpp
atomic<int> x{0};
```

**Thread 1**

```cpp
++x;
```

**Thread 2**

```cpp
--x;
```

**ATOMIC INC [x]**

**ATOMIC DEC [x]**

```cpp
x.fetch_add(1);
```

```cpp
x.fetch_sub(1);
```

```cpp
x == 0
```
Future and Promise

- future represents a result which will become available
  - e.g. set by another thread
  - exceptions are seen as results and propagate
- future is fulfilled by a promise
- communication via a thread-safe, shared state

```
future
get()
```

```
shared state
(set_value/empty/value/exception)
```

```
promise
set_value()
set_exception()
```

thread 1

thread 2
Future and Promise

Thread 1

promise<int> p;
future<int> f = p.get_future();
thread t{thread2, move(p)};
int result1 = func1();
int result2 = f.get();
t.join()

Thread 2

void thread2(promise<int> p){
    try {
        int result = func2();
        p.set_value(result);
    } catch(...) {
        p.set_exception(current_exception());
    }
}
std::async

- simple asynchronous execution of independent sub-tasks

```cpp
future<int> f = async([] { return fib(100); });
...
int result = f.get();
```

- launch options:
  - `launch::async` ⇒ execute in a different thread
  - `launch::deferred` ⇒ execute on `get()` in calling thread

```cpp
auto f = async(launch::async, [] { ... });
```
std::async

Thread 1

future<int> f =
    async(launch::async,
        func2);
int result1 = func1();
int result2 = f.get();

Thread 2

int func2() { ... }
Thread Local Storage (TLS)

- global storage per thread for variables
- `thread_local` type `ident`;
- thread local variable is created for each thread "before first use"

- examples
  - random number generator with per thread state/seed
  - per thread "global" error flag, cf. `errno` in C
  - returning pointers to static buffers, cf. `localtime`, `strtok` in C
```cpp
struct TLS {
    TLS(char name) : name(name) {
        print(this_thread::get_id(), " ctor ", name);
    }
    ~TLS() {
        print(this_thread::get_id(), " dtor ", name);
    }
    void func() {
        print(this_thread::get_id(), " func ", name);
    }
    private: char name;
};
```
thread_local TLS g_tls('G');

void test_tls() {
    auto func = []{
        thread_local TLS tls('L');
        tls.func();
        g_tls.func();
    };
    func();
    thread t{func};
    t.join();
    func();
}
Microsoft’s Parallel Patterns Library (PPL)
Intel’s Threading Building Blocks (TBB)
PPL/TBB Fundamentals

- C++ template library
  - designed to look like the STL
  - namespace concurrency
  - parallel algorithms
    - `<[tbb/compat/]ppl.h>`
    - based on transient thread pool
  - lock-free containers
    - `<[tbb/]concurrent_\w+>`
    - concurrent memory allocation
- PPL and TBB have a common, portable(!) subset
Parallel Algorithms

C++ standard library

- for-loop
- for_each
- transform
- accumulate
- sort

PPL/TBB (parallel_...)

- ...for
- ...for_each
- ...invoke
- ...transform
- ...reduce
- ...sort
- ...buffered_sort
- ...radixsort
Parallel for

```cpp
parallel_for(0, 20, [](int i) {
    print(this_thread::get_id(), i);
});
parallel_for(0, 20, 2, [](int i) {
    print(this_thread::get_id(), i);
});
```
Parallel for each

```cpp
vector<int> arr = create_ints();

atomic<int> count{0};
parallel_for_each(begin(arr), end(arr),
    [&](int& i) {
        if(i < 0) ++count;
        i += 100;
    });

cout << "negatives: " << count << endl;
```
Parallel for each

```cpp
vector<Point3> vec = create_points();

parallel_for_each(begin(vec), end(vec),
    [](Point3& v) {
        v.normalize();
    });
```
Parallel reduction (PPL)

```cpp
vector<int> arr = create_ints();

const auto sum =
    parallel_reduce(begin(arr), end(arr), 0);

cout << "sum: " << sum << endl;
```
Parallel reduction (PPL)

```cpp
vector<Point3> vec = create_points();

const Point3 sum = parallel_reduce(  
    begin(vec), end(vec), Point3{}  
);

cout << "avg: " << sum / vec.size() << endl;
```

uses `operator+(Point3, Point3)`
Parallel reduction (TBB)

```cpp
vector<int> vec = create_ints();
using Range = blocked_range<vector<int>::iterator>;
const auto sum = parallel_reduce(
    Range{begin(vec), end(vec)},
    0,
    [](const Range& range, int value) {
        return accumulate(begin(range), end(range), value);
    },
    [](int a, int b) { return a + b; }
);
cout << "sum: " << sum << endl;
```
Parallel reduction (TBB)

```cpp
vector<Point3> vec = create_points();
using Range = blocked_range<vector<Point3>::iterator>;
const auto sum = parallel_reduce(
    Range{begin(vec), end(vec)},
    Point3{},
    [](const Range& range, Point3 value) {
        return accumulate(begin(range), end(range), value);
    },
    [](Point3 a, Point3 b) { return a + b; }
);
cout << "avg: " << sum / vec.size() << endl;
```
Parallel transform (PPL only)

```cpp
vector<int> vec = create_ints();

vector<int> vec2(vec.size());
parallel_transform(begin(vec), end(vec),
                 begin(vec2),
                 [](int i) { return i * 2; })
```

Parallel transform (PPL only)

```cpp
vector<Point3> vec = create_points();

vector<Point3> vec2(vec.size());
parallel_transform(begin(vec), end(vec),
    begin(vec2), [](const Point3& v) {
        return v.normalized();
    });
```
Race Conditions

- Altering containers may not be thread safe
  
  ```cpp
  vector<Point3> vec = create_vectors();
  vector<Point3> vec2;
  parallel_transform(begin(vec), end(vec),
      back_inserter(vec2),
      [](auto p) { return p.normalize(); } )
  ```

- relies on `vec2.push_back()` to be thread-safe, use e.g. `concurrent_vector<Point3>`
Partitioners

- determine how work is distributed in parallel algorithms
  - `parallel_for(begin(vec), end(vec), func, part)`
- `static_partitioner`
  - splits the range into equally sized chunks, one for each thread
- `simple_partitioner`
  - custom chunk size, works different in TBB and PPL
- `auto_partitioner` (default)
  - uses work stealing to balance load among threads
- `affinity_partitioner`
  - remembers iteration-to-thread assignment for subsequent parallel algorithms
Partitioners

auto f = [](int i) { ... };

parallel_for(0, 20, f, static_partitioner{});
parallel_for(0, 20, f, simple_partitioner{5});
parallel_for(0, 20, f, auto_partitioner{});

affinity_partitioner p;
parallel_for(0, 20, f, p);
parallel_for(0, 20, f, p);
Concurrent Data Structures

C++ standard
- vector
- queue
- priority_queue
- unordered_map
- unordered_multimap
- unordered_set
- unordered_multiset

PPL/TBB (concurrent_...)
- vector
- queue
- priority_queue (TBB)
- bounded_queue (TBB)
- unordered_map
- unordered_multimap
- unordered_set
- unordered_multiset
**concurrent_vector<T>**

- Resembles `std::vector<T>` , but
  - allows only appending new elements, `push_back()` , no `insert()`
  - no element removal, `erase()`, `pop_back()`
  - no resize, only `grow_by()`, `grow_to_at_least()`
  - elements are not stored continuously, no `data()` and `&v[0] + n` is invalid
  - no relocation on `push_back()` or grow
concurrent_vector<T>

vector<float> vec = create_floats();

concurrent_vector<Point3> res;
parallel_for_each(begin(vec), end(vec),
  [&](float f) {
    res.push_back(Point3(f, f, f));
  });

bad example, use parallel_transform() instead
concurrent_vector<T>

vector<Tri> vec = create_triangles();

concurrent_vector<Tri> res;
parallel_for_each(begin(vec), end(vec) [&](const Tri& t) {
    vector<Tri> s = subdivide(t);
    copy(begin(s), end(s), back_inserter(res));
});
Memory Allocation

- TBB provides custom allocators
  - `cache_aligned_allocator<T>`
    - allocates each element on a separate cache line
    - avoids false-sharing (cf. advanced parallelism talk)
  - `scalable_allocator<T>`
    - faster concurrent allocation
    - uses per-thread private heap
    - built on `scalable_malloc()`, `scalable_free()`
  - drop-in replacements for `std::allocator<T>`
    - e.g. `vector<int, scalable_allocator<int>>`
- PPL offers only `Alloc()` and `Free()`
scalable_allocator<T> (TBB)

```cpp
vector<int, scalable_allocator<int>> vec;
// use vec ...
```

```cpp
parallel_for(0, 10, [](
    vector<int, scalable_allocator<int>> vec;
    // use vec ...
})
```

Serial context, NOT useful

Parallel context, useful
C++17 Parallel STL and Concurrency
Parallelism TS

- C++ extensions for parallelism
- a.k.a. Parallel STL
  - provides parallel versions of many STL algorithms
- will probably be part of C++17
  - merged into C++ standard as of march 2016
  - approval from national bodies and final ballot pending
- some implementations already available
  - Microsoft: parallelstl.codeplex.com
  - High Performance ParalleX (HPX) library
Execution Policies

- specifies how STL algorithms are executed
  - sequential
  - parallel
  - parallel and vectorized
  - ...

- implemented as types and global instances
  - `std::execution::sequenced_policy` seq
  - `std::execution::parallel_policy` par
  - `std::execution::parallel_unsequenced_policy` par_unseq

- passed as first argument to STL algorithms
Execution Policies

- `for_each(execution::par, begin(vec), end(vec), func);`
- `sort(execution::par, begin(vec), end(vec));`
- `vec1.resize(vec.size());
  transform(execution::par, begin(vec), end(vec), begin(vec1),
  [](Point3 p) { return p.normalized(); });`
- `vec2.resize(vec.size());
  transform(execution::par_unseq, begin(vec), end(vec), begin(vec2),
  [](float f) { return sqrt(f); });`
### Parallel Algorithms

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

- C++ extensions for concurrency
  - continuations for `future<T>`
  - barriers and latches
  - `atomic_shared_ptr<T>` and `atomic_weak_ptr<T>`
- not (yet) part of the C++ standard
  - maybe included in C++20
- most features available in Boost.Thread
Future Continuations

```c
int func1() { /* ... */ }
int func2(int i) { /* ... */ }
float func3() { /* ... */ }

future<int> f1 = async(launch::async, func1);
future<int> f2 = f1.then(func2);
future<float> f3 = async(launch::async, func3);

future<tuple<int, float>> f4 = when_all(f1, f3);

future<float> f5 = f4.then([](tuple<int, float> t) {
    return get<0>(t) + get<1>(t); });

float result = f5.get();
```
Castor, 4228m
Pollux, 4092m
zwischen
Monte-Rosa-Massiv
und Matterhorn
Wallis, Schweiz