EBERHARD KARLS UNIVERSITÄT TÜBINGEN
FACULTY OF SCIENCE
Applied Bioinformatics

Programming in C/C++

- Parallelism -

Parallel Execution

Parallelization

Many problems can be solved faster by solving chunks of it in parallel.

Parallel execution strategies

- distributed memory systems (not discussed today)
  - a cluster of many compute nodes
- shared memory systems:
  - modern CPUs with >1 core
    - Intel Xeon Platinum 9282 (Skylake-SP): 56 Cores (112 Threads)
    - AMD Epyc (Zen2; 08/2019): 64 Core (128 Threads) per Socket
  - GPUs or special hardware (Xeon Phi)

Parallelization using Multi-threading

- Need to find problems that can be parallelized / decomposed into independent chunks

- **Embarrassingly parallel** problems are easily divided in subproblems.
  - add 1..N
  - fibonacci(10000000)
  - fold 100 proteins
  - brighten up 1M pixels

Each subproblem can be tackled by one thread of execution.

Threads are the machine-level foundation for concurrent and parallel programming.

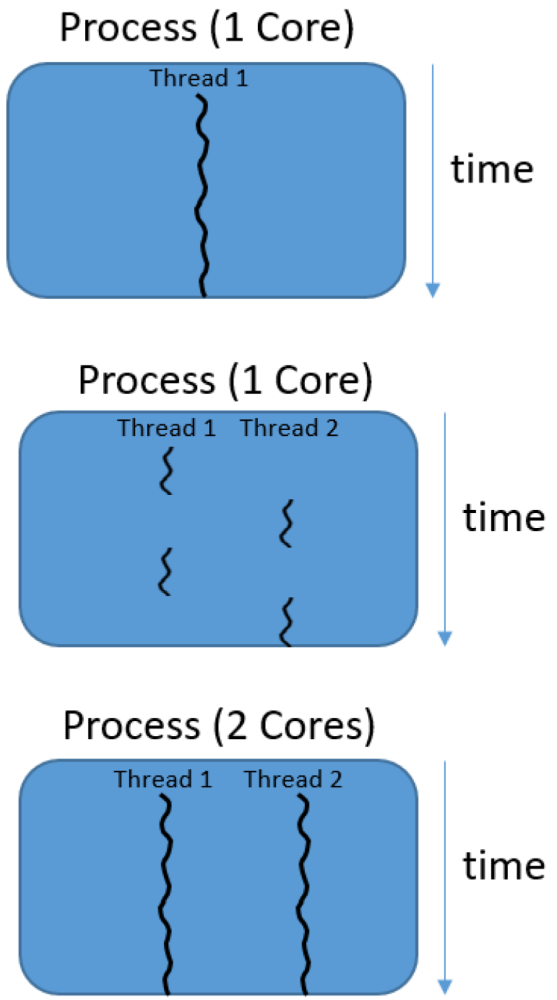
Threads

- Part of a process
- ‘Light-weight’ process with own stack
- Default: a process starts with a single **main thread**
- Threads allow running multiple sections of a program independently**, while sharing the same memory**.

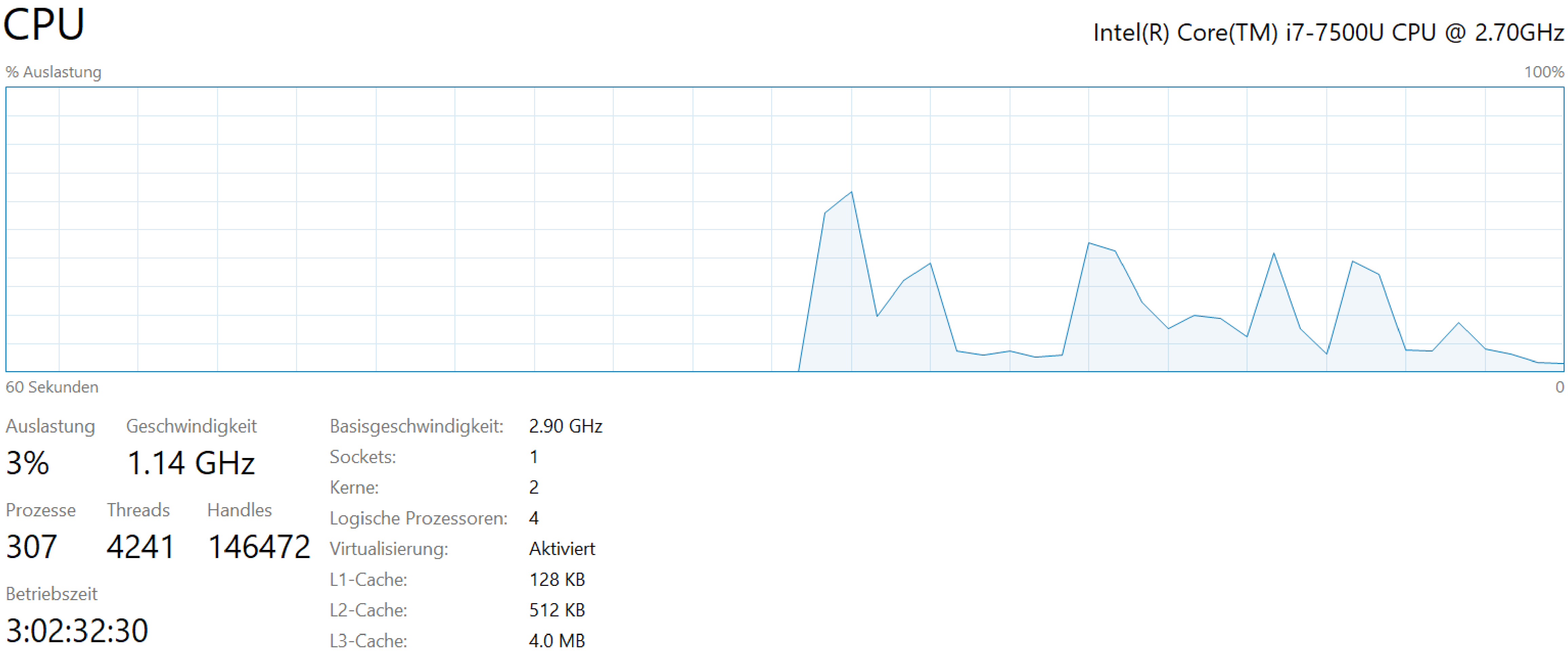
- Threads share resources of the process, e.g.
  - memory, file handles, ...
  - two threads can read/write from/to same data structures

- **Sharing data between threads is cheap**

- The OS will allocate threads to all (logic) CPU cores automatically (scheduling / interleaving)



Threads


CPU
Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz
% Auslastung
100%
60 Sekunden
0
Auslastung
3%
Geschwindigkeit
1.14 GHz
Prozesse
307
Threads
4241
Handles
146472
Betriebszeit
3:02:32:30
Basisgeschwindigkeit: 2.90 GHz
Sockets: 1
Kerne: 2
Logische Prozessoren: 4
Virtualisierung: Aktiviert
L1-Cache: 128 KB
L2-Cache: 512 KB
L3-Cache: 4.0 MB

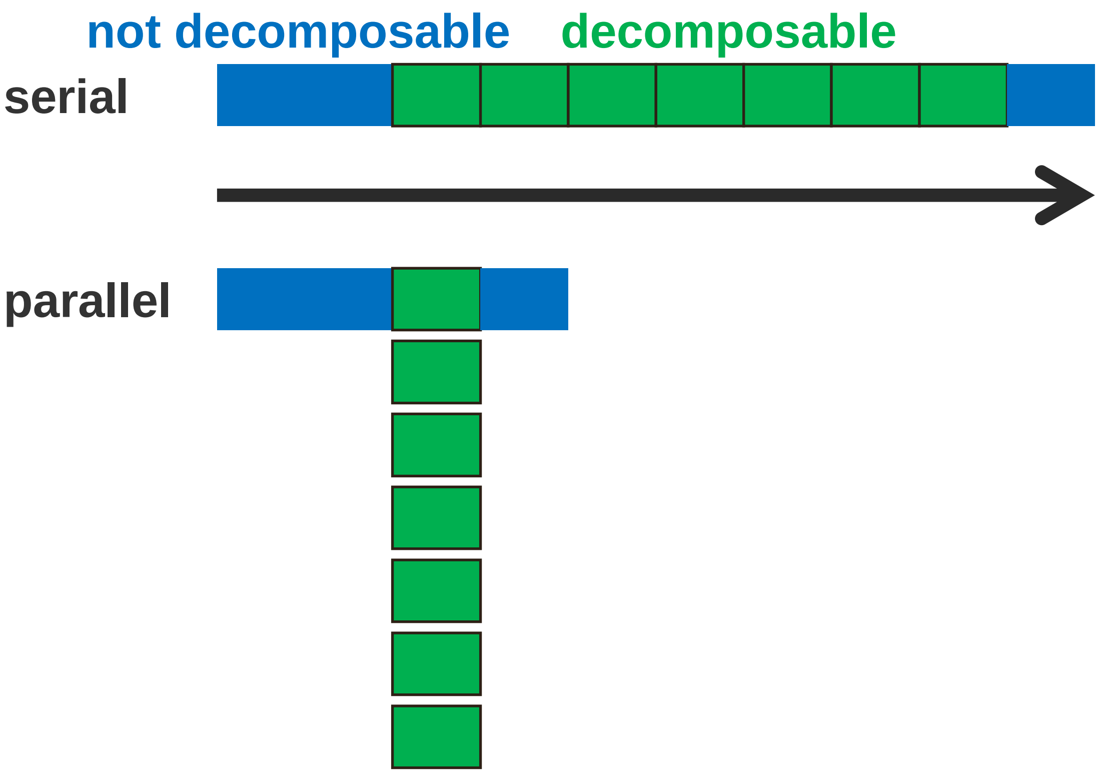
Amdahl's Law

- Split the problem A into:
  - the part P that can be parallelized and
  - the fraction (1-P) where no gain in parallelization is achieved

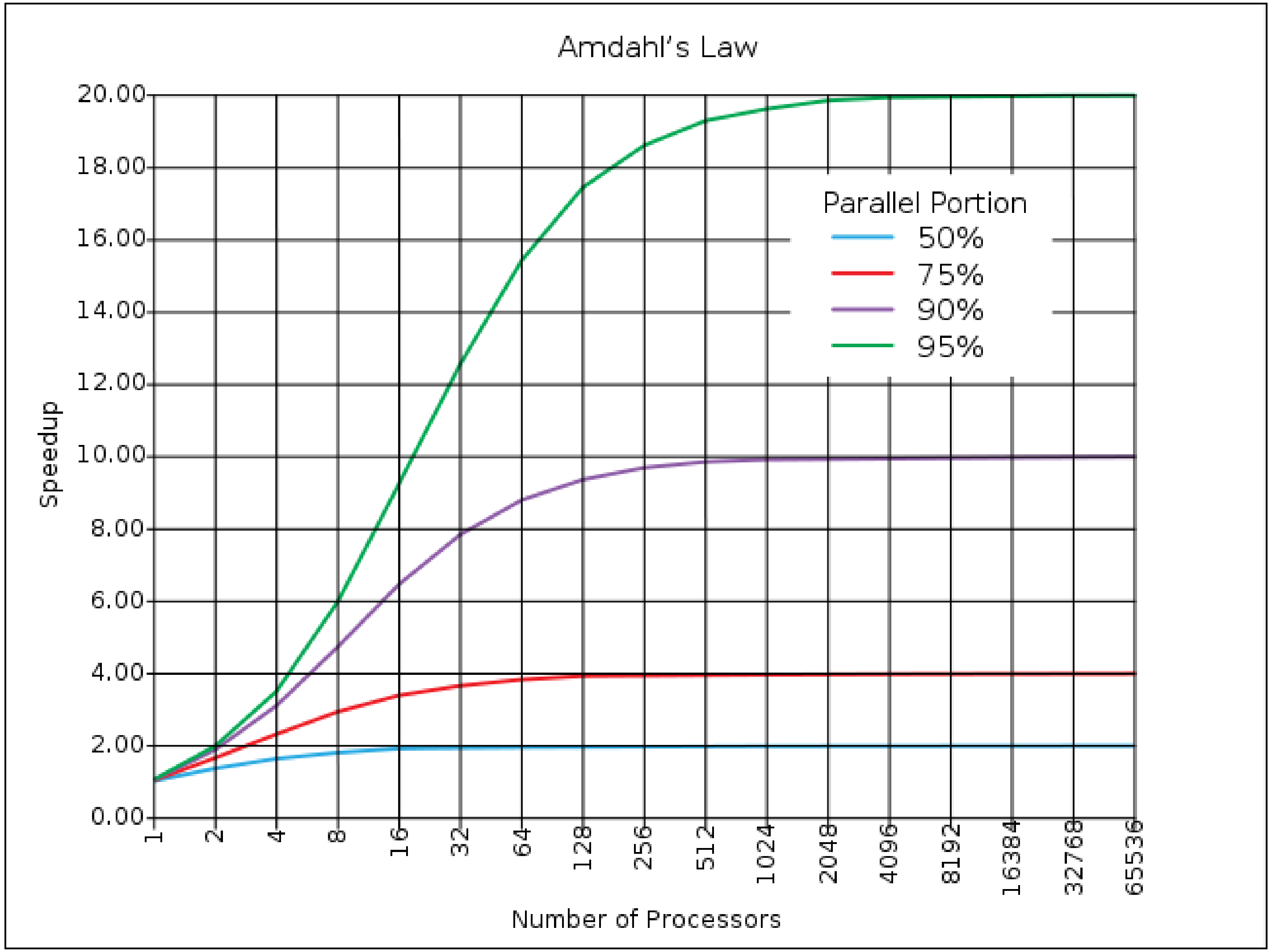
- The maximum speedup is

$$S_{\max} = \frac{t_S(n)}{t_{(1-P)}(n) + \frac{t_P(n)}{p}}$$

for p processors



Amdahl’s Law



[Wikipedia]

OpenMP

Multithreading in C++

- External libraries, for example:
  - Intel Threading Building Blocks (TBB)
  - Microsoft Parallel Patterns Library (PPL) [VS only]
  - Open Multi-Processing (OpenMP)
    - supported by most compilers natively

- Build in functionality: C++11/C++14/C++17
  - `std::thread`, `std::async`, `std::future`, `std::promise`

- **Danger Zone**: be careful when mixing different threading libraries. They often don't play well together.

- We will look at OpenMP and the build in functionality as these are widely used in code bases.

OpenMP

- High level abstraction (no low-level control)
- Usually applicable without major code changes
- Degrades to single-threaded code if compiled without OpenMP support

```
  double A[10000];
#pragma omp parallel for
  for (int i = 0; i < 10000; ++i)
    A[i] = computation(i);
```

- Compiler directives as preprocessor macro

```
#pragma omp <construct> [<clause>]
```

- is valid for the next block / statement

- OMP functions (runtime decisions)

```
omp_... ();
// e.g.
int x = omp_get_thread_num(); // 0..N-1
```

Hello World – OpenMP-Style

```cpp
#include <iostream>
#include <omp.h> // for omp functions; #pragma only requires compile flag

int main() {
  // only 1 thread here … now spawn some threads…

#pragma omp parallel num_threads(20)
  { std::cout << "Hello, world from thread #" << omp_get_thread_num() << "\n"; }

  // only 1 thread here …
}
```

- What is the expected output?

```
Hello, world from thread #Hello, world from thread #Hello,
world from thread #Hello, world from thread #Hello, world from
thread #18
Hello, world from thread #9
15Hello, world from thread #
Hello, world from thread #Hello, world from thread #Hello,
world from thread #4Hello, world from thread #6
Hello, world from thread #13Hello, world from thread #5
Hello, world from thread #Hello, world from thread #16

10
1Hello, world from thread #
```

Controlling the Number of Threads

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- **Default**: number of logical CPU cores
- Environment variable: `SET OMP_NUM_THREADS=3` overwrites default
- within C++
  - before the parallel region:

    ```
    omp_set_num_threads(x);
    ```

  - as argument to `omp parallel`

    ```
    #pragma omp parallel num_threads(x)
    ```

- Number of threads might exceed number of cores on the machine → What is the effect?

```
// omp_set_num_threads(3000); // insane machine!
#pragma omp parallel
  if (omp_get_thread_num() == 0)
    cout << omp_get_num_threads() << " threads!\n";
```

Work Distribution

How to you distribute work in a parallel region?

- 'manually' by thread-id - **not recommended**
- `pragma section`
- `pragma for`
- `pragma task` (since OpenMP 3.0, not discussed)

Work Distribution – Manually

- In principle, we could assign threads to work on items individually

```
  std::vector<double> v(1000);
#pragma omp parallel
  {
    int tc = omp_get_num_threads(); // within PR; otherwise always = 1
    int id = omp_get_thread_num();  // within PR; otherwise always = 0
    for (int i = id; i < v.size(); i += tc)
    {
      v[i] = compute_something(i);  // must be re-entrant
    }
  }
```

- **Not recommended**

Work Distribution – Sections

- sections must be independent blocks of work and are distributed to available threads automatically.
- The order of execution is **not defined**!

```
  double a, b;
#pragma omp parallel
  {
  #pragma omp sections
    {

    #pragma omp section
      { a = computePI(); }

    #pragma omp section
      { b = computeEuler(); }

    }
  }
```

Work Distribution – `for` Loop

- OpenMP distributes loop iterations among threads
- The order of execution is only partially defined (and depends on keyword `schedule`)

```
#pragma omp for [schedule(static|dynamic|guided|auto [, chunksize)]]
```

```
  vector<int> v(10000);
#pragma omp parallel
#pragma omp for
  for (int i = 0; i < N; ++i)
    compute_something(v[i]);
```

- **Detail**: Only since OpenMP 3.0 (Clang, g++) iterators can be used in loops:

```
  vector<int> v(10000);
#pragma omp parallel
#pragma omp for
  for (auto i = v.begin(); i < v.end(); ++i) // check if already supported if you use MSVC
    compute_something(*i);
```

`for` Loop – Restrictions

Only for-loops with a certain syntax are valid:

- **no break**, **no return**, **no goto** (in OpenMP >= 4.0: cancellation points)
- integer loop variable: **signed** (OpenMP < 3.0) or unsigned
- tests only **<, <=, >, >=**
- update of loop variable via operator ++, +, +=, =
- fixed upper/lower bound of loop (once it starts)

Short form:

```
#pragma omp parallel for [schedule(static|dynamic|guided|auto [, chunksize)]]
```

`for` Loop – Scheduling

```
#pragma omp parallel for [schedule(static|dynamic|guided|auto [, chunksize)]]
```

- `schedule(static, [k=n/#t])` range is split into blocks of size **k** and handed out round robin `i = 0; i < 100; ++i;` e.g. for `#t = 3` threads
  - `k=1: 012 012 012 ...`
  - `k=5: 000001111122222 000001111122222 ...`
  - `k=?: k = #loops/ #threads`
- `schedule(dynamic, [k=1])` range is split into blocks of size k and handed out to idle threads
- `schedule(guided, [k=?])` range is split into blocks of size proportional to work remaining; k = *minimum block size*
- `schedule(runtime)` decision on runtime value (`omp_set_schedule`, `$OMP_SCHEDULE`)
- `schedule(auto)` implementation defined behavior

Synchronization – Race Condition

- If distribution of work is easy, why is concurrency so hard?

```
  int m{0};
#pragma omp parallel for
  for (int i = 0; i < 1000; ++i) {
    ++m;
  }
  cout << "iterations: " << m << endl; // number of loop iterations
```

- Why is `m < 1000` (most often)?
- **Race condition** on shared variable! (1 writer + other readers/writer)
- Only readers would be ok!

Data Sharing – Memory Model

```
#pragma omp parallel [shared(.), private(.), firstprivate(.), lastprivate(.)]
```

By default

- all variables **before** `#pragma parallel` are shared between threads
- all variables **within** `#pragma parallel` are private/local for each thread (private stack)

```
  const int N = 1000;
  vector<int> v(N);
#pragma omp parallel
  {
    int temp;
  #pragma omp for
    for (int a = 0; a < N; ++a) {
      temp = a % 3;
      v[a] = temp * temp;
    }
  }
```

In this example:

- Shared: **`v, N`**
- Private: **`temp, a`**

Data Sharing – Memory Model

```
#pragma omp parallel [shared(.), private(.), firstprivate(.), lastprivate(.)]
```

`shared()`

- Variable is read/write for all threads
- Here, **`v`** is shared by default anyway
- Same as last slide

```
  const int N = 1000;
  vector<int> v(N);
#pragma omp parallel shared (v)
  {
    int temp;
  #pragma omp for
    for (int a = 0; a < N; ++a) {
      temp = a % 3;
      v[a] = temp * temp;
    }
  }
```

In this example:

- Shared: **`v, N`**
- Private: **`temp, a`**

Data Sharing – Memory Model

```
#pragma omp parallel [shared(.), private(.), firstprivate(.), lastprivate(.)]
```

`private()`
- Every thread gets a private copy
- No initialization

```
  const int N = 1000;
  vector<int> v(N);
  int temp;
#pragma omp parallel private(temp)
  {
  #pragma omp for
    for (int a = 0; a < N; ++a) {
      temp = a % 3;
      v[a] = temp * temp;
    }
  }
```

In this example:
- Shared: `v, N`
- Private: `temp, a`

- `private()` can enable a drop-in parallelization without changing the code
- Otherwise, there is no difference to local vars

Data Sharing – Memory Model

```
#pragma omp parallel [shared(.), private(.), firstprivate(.), lastprivate(.)]
```

`private()`

- Every thread gets a private copy
- **No** initialization

```
  int done = 4;
#pragma omp parallel for private(done)
  for (int a = 0; a < 8; ++a) {
    printf("%d\n", done);
  }
  // done == 0
```

`firstprivate()`

- Every thread gets a private copy
- **With** initialization

```
  int done = 4;
#pragma omp parallel for firstprivate(done)
  for (int a = 0; a < 8; ++a) {
    printf("%d\n", done);
  }
  // done == 4
```

Data Sharing – Memory Model

```
#pragma omp parallel [shared(.), private(.), firstprivate(.), lastprivate(.)]
```

`default(none)`

- All variables must be named explicitly
- Now `shared` and `private` are used to specify each variable individually

Recommended, because it gives full control over how variables are shared.

```
  int i, x;
#pragma omp parallel for private(i), default(none)
  for (i = 0; i < 10; ++i) {
    x = 1; // error: x is private
  }
```

Synchronization – `critical`

`omp critical [(name)]`

(mutually exclusive block)

- only one thread enters the block
- all other threads wait until the next one can enter
- incurs a performance penalty for entering/leaving the block

```
  double mean = 0; // shared
#pragma omp parallel
  {
    double m = 0; // private
  #pragma omp for
    for (int i = 0; i < 100; ++i) {
      m += getItem(i);
    }
  #pragma omp critical(add_critical)
    { mean += m; }
  }
  mean /= 100;
```

- **Recommendation:** Use different names for independent shared vars to increase performance. All critical sections with the same name use the same lock!

Synchronization – `atomic`

`omp atomic` (~critical for a single statement)

- like `critical`, but **faster** and with **limitations**
- Only a single command from

  ```
  x binop=y   x,y scalar; &x!=&y
  x++;
  ++x;
  x--;
  --x;
  ```

- Where **`binop`** can not be an overloaded operator and is one of **`+, *, -, /, &, ^, |, <<, >>.`**

```
  double mean = 0; // shared
#pragma omp parallel
  {
    double m = 0; // private
  #pragma omp for
    for (int i = 0; i < 100; ++i) {
      m += getItem(i);
    }
  #pragma omp atomic
    mean += m;
  }
  mean /= 100;
```

Synchronization – `single`

`omp single [nowait]`

- the first thread to arrive, executes the block
- all others **skip** the block
- all threads continue collectively after the block (barrier, unless `nowait`)

```
#pragma omp parallel num_threads(4)
  {
  #pragma omp single nowait
    {
      cout << "I'm thread " << omp_get_thread_num() << " of "
                             << omp_get_num_threads() << "\n";
    }
  }
```

Synchronization – `master`

`omp master [nowait]`

- only the master thread executes the block
- all others **skip** the block
- all threads continue collectively after the block (barrier, unless **`nowait`**)

```
#pragma omp parallel num_threads(4)
  {
  #pragma omp master
    {
      cout << "I'm thread " << omp_get_thread_num() << " of "
                             << omp_get_num_threads() << "\n";
    }
  }
```

- Always prints: **`I'm thread 0 of 4.`**

Quiz

What is the difference between `omp single` vs. `omp critical`?

```
  int s = 0, c = 0;
#pragma omp parallel num_threads(4)
  {
  #pragma omp single
    ++s;
  #pragma omp critical
    ++c;
  }
  cout << "s: " << s << " c: " << c;
```

- Always prints `s: 1 c: 4`.

Quiz – Reduction

```
  double mean = 0;
#pragma omp parallel for
  {
    for (int i = 0; i < 100; ++i) {
  #pragma omp critical
      mean += getItem(i); // assume atomic is not sufficient!
    }
  }
  mean /= 100;
```

- Q: Is this efficient?

Reduction

```
  double mean = 0;     // shared
#pragma omp parallel
  {
    double m = 0;      // private copy
  #pragma omp for nowait
   for (int i = 0; i < 100; ++i)
   {
        m += getItem(i);
   }
  #pragma omp atomic
    mean += m;
  }
  mean /= 100;
```

Reduction

- OMP has built-in functionality for reductions

```
  double mean = 0;
#pragma omp parallel for reduction(+ : mean)
  {
    for (int i = 0; i < 100; ++i) {
      mean += getItem(i);
    }
  }
  mean /= 100;
```

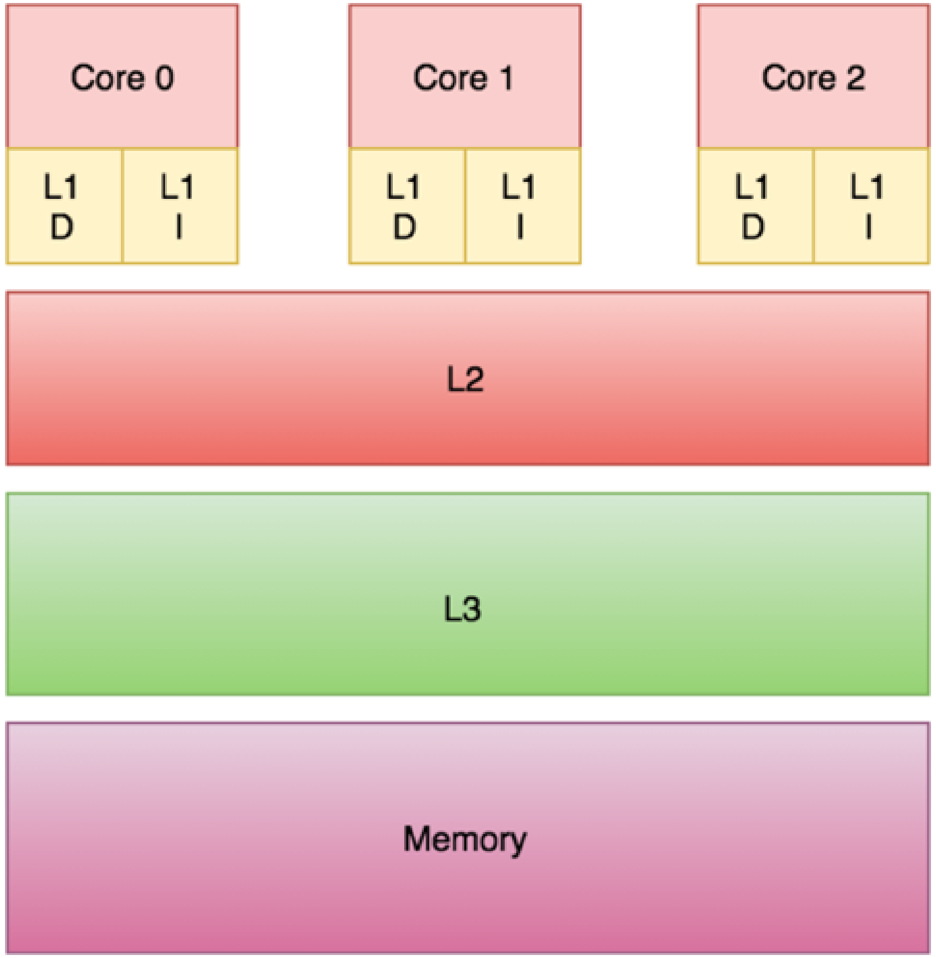
Operator	Init. Value
`+, -, |, ^, ||`	0
`*, /, &&`	1
`&`	~0
min	largest rep. no
max	Smallest rep. no

- Part of `#pragma parallel` (works for loops, sections, …)
- Initialization of local vars depends on operand
- Initialization of reduction var (`mean`) is your responsibility

False Sharing

How does false sharing happen:

- Write access to neighboring memory by one thread, **invalidates the cache** line of another which forces expensive sync.
- Cache line on modern CPUs: ~64 bytes



Consequence of false sharing:

- A significant performance penalty on systems with coherent caches (like x86).

False Sharing – Example

```
// x and y are neighbors
struct foo { int x = 1; int y = 1; } f;

int sum_a() {
  int s = 0;
  for (int i = 0; i < 1e7; ++i) s += f.x;
  return s;
}
void inc_b() {
  for (int i = 0; i < 1e7; ++i)
    ++f.y;
}

int main() {
  omp_set_num_threads(2);
#pragma omp parallel sections
  {
#pragma omp section
    sum_a();
#pragma omp section
    inc_b();
  }
}
```

Problem:

`++f.y` in `inc_b()` invalidates cache line of `f.x` in `sum_a()`

False Sharing – Example


```
// x and y are neighbors
struct foo { int x = 1; int y = 1; } f;

int sum_a() {
  int s = 0;
  for (int i = 0; i < 1e7; ++i) s += f.x;
  return s;
}
void inc_b() {
  for (int i = 0; i < 1e7; ++i)
    ++f.y;
}

int main() {
  omp_set_num_threads(2);
#pragma omp parallel sections
  {
#pragma omp section
    sum_a();
#pragma omp section
    inc_b();
  }
}
```

Problem:

`++f.y` in `inc_b()` invalidates cache line of `f.x` in `sum_a()`

Fix 1: no read in loop

```
int temp = f.x;
for (…) s += temp;
```

Fix 2: no write in loop

```
int temp = f.y;
for (…) ++temp;
f.y = temp;
```

with false sharing:	150 msec
no false sharing (Fix 1):	80 msec
no false sharing (Fix 2):	74 msec

Performance

Ideally, perfect scaling with number of threads

Reality

- not all code paths are parallelizable
- CPU-Throttling (thermals)
- imperfect load balancing (wait for last thread)
- cache contention
- memory bandwidth contention
- multi-threading overhead
  - creating threads (`omp parallel`)
  - sync overhead (avoid `atomic`/`critical` if possible; name critical sections)
  - avoid false sharing (cache invalidation)

Spot the Bug

Common error:

```
int sumUp(int num_steps) {
  int sum(0);
  #pragma omp parallel reduction(+ : sum)
  #pragma omp master
    std::cout << "running with " << omp_get_num_threads() << "\n";
  #pragma omp for
  for (int i = 0; i < num_steps; ++i) {
    sum += i;
  }
  return sum;
}
```

- {} block notation missing for `omp parallel`

Spot the Bug 2

```
#define N2 1024
#pragma omp parallel
  {
    double a[N2][N2];
    int tid = omp_get_thread_num();
    /* Each thread works on its own private copy of the array */
    for (int i = 0; i < N2; i++)
      for (int j = 0; j < N2; j++)
        a[i][j] = tid + i + j;

    cout << tid << " done. Last element= " << a[N2 - 1][N2 - 1] << endl;
  }
```

- `StackOverflow`! Every thread allocates 1024*1024*8 byte = 8 MB on its own stack.
- Depending on OS, every thread only gets 0.0X/1/2/4 MB.
- Environment variable: **`$OMP_STACKSIZE="20M"`** or
- Use heap:

```
std::vector< std::vector<double> >
            a(N2, std::vector<double>(N2, 0));
```

Danger Zone

- STL containers are not thread-safe
  - **read-only access** by all threads in parallel is fine
  - do not write (and read/write) concurrently:
  - no `.push_back(), .insert(), .append(), .resize(), .reserve()` within `parallel` unless synchronized using e.g., **`named`** `critical` sections)

- Exceptions within `parallel` must be caught by the same thread!
  Otherwise: runtime exception!

Nice collection of typical errors

- [32 OpenMP traps for C-developers]()

Threads in C++

Outline

- Threads in C++11
- Memory model
- Synchronization
- Futures and Promises
- Async
- (sleep and wakeup: condition variables)

Threading in C++

C++03: no definition of threads or locks in the language (!)

- **Alternatives:** platform specific extensions (pthreads,...) or external libraries (OpenMP, Boost-Threading, Intel TBB, etc)

C++11 ships with:

- Thread memory model
- Thread management (Async, Future, Promise)
- Protection of shared data, i.e. sync (lock/mutex, atomics)

Design goal: no abstraction costs (performance comparable to native threads)

Hello World with Threads

```
#include <iostream>
#include <thread>

 void call_from_thread() { std::cout << "Hello, World" << std::endl; }

 int main() {
  std::thread t1(call_from_thread); // run call_from_thread() in a new thread
  t1.join();                        // main program waits until t1 returns

  return 0;
}
```

- Every thread executes a function
- How many threads run in parallel?

std::thread

- A thread constructed without arguments is just an empty hull
  - `std::thread t1; //` does nothing and is not linked to a system thread
- A thread constructed with a functions starts to work immediately

  `std::thread t1(some_function);`
- Thread objects cannot be copied, only moved from (transfer of ownership)!

  `t1(std::move(t2)); or t1 = std::move(t2); // t2` is not a running thread anymore

- **Exceptions** must be handled within the thread (or it kills the program)

- The return value of the called function is discarded.

  (but could be fetched using `std::promise/future` – later more)

- After that, the thread object is useless! (in C++11)
  - Create a new thread (ineffective?)
  - Write the function such that it does not return immediately, thus keeping the thread alive (`condition_variable`, `wait()`, `notify_*()`)

`join` vs. `detach`

- `join()` or `detach()` **must be called before the thread object goes out of scope**

```
{ // new scope
  std::thread t(func);
  // program will violently terminate here
}

{ // new scope
  std::thread t(func);
  t.join(); // wait for t to finish func()
}


{ // new scope
  std::thread t(func);
  t.detach(); // do not wait for func()
  assert(t.joinable() == false);
}
```

`join`

- blocks the parent (bad for GUI-Main thread)
- ensures the thread function ended
- if in doubt use join()

`detach`

- never blocks the parent
- thread ends after main() is done or if thread function (func) is done (whatever happens first)
- Thread is disconnected from its thread object (i.e. doing operations, e.g. `operator=(Thread&&)` is illegal)

More than one Thread?

```cpp
// it is only a HINT! (might return 0)!
static const uint num_threads = std::thread::hardware_concurrency();

void call_from_thread(int n) {
  std::cout << "Hello, World from Thread #" << n << " with id "
            << std::this_thread::get_id() << std::endl;
}

int main() {
  std::vector<std::thread> t;
  for (int i = 0; i < num_threads; ++i) {
    t.push_back(std::thread(call_from_thread, i + 1)); // move into t
  }

  call_from_thread(0);

  for (int i = 0; i < num_threads; ++i) {
    t[i].join();
  }
}
```

- Function parameters are passed as additional arguments
- Order of execution is undefined → Mutex/Lock for shared objects (e.g. `std::cout`)
- Why not merge the two for-loops?

Synchronization

- To avoid **Race Conditions** (X readers/writers; 1 writer to shared memory)
- Ensures **sequential consistency** (same behavior as if it were a sequential program)
- general rule: every **shared** variable must be protected if there is **at least 1 writer**

- **Detail:** „**`volatile`**“ keyword in C++ has no meaning in threading! (rather hardware and restrictions on optimization)!

http://channel9.msdn.com/Shows/Going+Deep/Cpp-and-Beyond-2012-Herb-Sutter-atomic-Weapons-1-of-2

Synchronization `std::atomic` <atomic>

- `std::atomic<T>` with scalar T ∈ (`int`, `char`, `intptr_t`)

- Simple arithmetic operations are atomic
  - operators: `++`, `--`, `+=`, `&=`, `|=`
  - `fetch_add(T)`, `fetch_sub(T)`, `fetch_and(T)`, ...

```
std::atomic<int> x{0};
x += 10;
x.fetch_add(10);

x = x + 1;          // not atomic
```

- Every mention of an atomic variable is an atomic operation; if you mention it twice… its not atomic

- Support for operations on `float`/`double` only added in C++20 (x64 has no asm instructions for it)

```
std::atomic<double> x{0};
x += 10.0;
x.fetch_add(10.0);
x = 100.0;
```

Overhead of `std::atomic<>` for `int, char, ...`

- **Reading**: none (same as reading any non-atomic), it's a mov in asm
  (only up to a certain register size, usually %mmx = 16 bytes on x86; larger sizes will indirectly force a lock in the background, i.e. not lock-free)

- **Write**: "a bit more" than writing to non-atomics

- **But**: every load/store of an atomic variable implies a memory-order-sequential barrier (by default at least), i.e. all loads/stores (also of other variables) will be visible globally at this point, and the compiler/CPU cannot reorder instructions across this barrier (down or up).

‘Atomic’ of Arbitrary Types

- `compare_exchange_strong()`
- a.k.a. compare-and-swap (CAS) idiom
- On any type T of arbitrary size, which is **trivially copyable**

```
bool success = x.compare_exchange_strong(y, z); // T x,y,z
// if x == y, set x = z and return true;
// if x != y, set y = x and return false;
```

- A **trivially copyable** class is a class that:
  - uses the implicitly defined copy and move constructors, copy and move assignments, and destructor.
  - has **no virtual members**.
  - its base class and non-static data members (if any) are themselves also trivially copyable types.

Detail: 'Atomic' of Arbitrary Types

- Lock-free datastructures, other ..._exchange_... variants exist

```
struct node {
   int data;
   node* next;
   node(const int& data) : data(data), next(nullptr) {}
};

class stack {
   std::atomic<node*> head;
   public:
      void push(const int& data) {
         node* new_node = new node(data);      // create a new node...
         new_node->next = head.load();          // ... and let it point to head

        // Now we want to make the new_node the new head
        // if new_node->next still points to the (old) head
        //   CES sets head to new_node. -> Old head is first node after head - we are done.
        // But if/while new_node->next suddenly points to a different node than head
        // (some other thread must have inserted a node just now)
        // we need to update new_node->next with the (externally changed) head and try again
        while(!head.compare_exchange_strong(new_node->next, new_node));
      }
};

int main() {
   stack s;
   s.push(1);
   s.push(2);
   s.push(3);
}
```

Mutex `<mutex>`

Mutual **ex**clusion Locks

- has more overhead than the ‘lock-free’ `atomic<T>`
- possibility of deadlocks (even with a single lock)

```
std::mutex m;

m.lock();
// … protected …
m.unlock();
```

`(constructor)`	Construct mutex
`.lock`	Lock mutex
`.try_lock`	Lock mutex if not locked
`.unlock`	Unlock mutex
`.native_handle`	Get native handle

Mutex

- A mutex ‘m’ protects a certain shared memory/resource/variable ‘v’
- ‘m’ must be used by all threads accessing ‘v’
- Coupling of ‘m’ and ‘v’ must be established by YOU
- **Recommendation**: Protect independent variables by their own mutex (performance)
- What does this remind you of? - critical region

Mutex – Example

```cpp
#include <mutex>

std::mutex mtx; // global mutex

void print_block(int n, char c) {
  mtx.lock();
  for (int i = 0; i < n; ++i) {
    std::cout << c;
  }
  std::cout << '\n';
  mtx.unlock();
}

int main() {
  std::thread th1(print_block, 100, '*');
  std::thread th2(print_block, 100, '$');

  th1.join();
  th2.join();

  return 0;
}
```

- Coupling of `mtx` to `cout` is implicit.
- Use the same mutex for all potentially concurrent usage of `cout`.
- Why is explicit `lock()`/`unlock()` dangerous? Encapsulate?

Mutex and Exceptions – Example

```cpp
#include <mutex>
std::mutex mtx; // global mutex

void print_block(int n, char c) {
  try
  {
    mtx.lock();
    for (int i = 0; i<n; ++i) {
        std::cout << c; }
    if (rand() == 10) throw "unexpected";
    std::cout << '\n';
    mtx.unlock();
  }
  catch (...)  {  }
}

int main() {
  std::thread th1(print_block, 100, '*');
  std::thread th2(print_block, 100, '$');

  th1.join();
  th2.join();
}
```

- **Remember**: **Exceptions** must be dealt with within the thread!
- Potential deadlock:
  - One thread always obtains the lock first
  - **Exception** prevents running `unlock()`
  - The other thread waits indefinitely at `lock()`

Mutex and Exceptions – Better Example

```
#include <mutex>
std::mutex mtx; // global mutex

void print_block(int n, char c) {
  try
  {
    mtx.lock();
    for (int i = 0; i<n; ++i) {
        std::cout << c; }
    if (rand() == 10) throw "unexpected";
    std::cout << '\n';
    mtx.unlock();
  }
  catch (...)  {  }
}

int main() {
  std::thread th1(print_block, 100, '*');
  std::thread th2(print_block, 100, '$');

  th1.join();
  th2.join();
}
```

```
void print_block(int n, char c) {
  try
  {
    std::lock_guard<std::mutex> l(mtx);
    for (int i = 0; i<n; ++i) {
        std::cout << c; #
    }
    if (rand() == 10) throw "unexpected";
    std::cout << '\n';
  }
  catch (...) {}
}
```

- `lock_guard`:
  - Constructor: `calls mtx.lock()`
  - Destructor: `calls mtx.unlock()`

- Why does this work?
  - **RAII in action**

Example – Mutex und RAII (Scoped Locking)

- **`template <class Mutex> class lock_guard;`**
  - **Lock at construction** (will block until successful!)
    ```
    std::lock_guard<std::mutex> lock(mtx);
    ```
  - Unlock during destruction (end of context)
  - Easy!

- **`template <class Mutex> class unique_lock;`**
  - **Initialization as desired (locked/unlocked/lock-attempt for a certain period)**
    ```
    std::unique_lock<std::mutex>(mtx, std::defer_lock);
    ```
  - Unlock during destruction (end of context)
  - More fine-grained control:

`.lock`	Lock mutex
`.try_lock`	Lock mutex if not locked
`.try_lock_for`	Try to lock mutex during time span
`.try_lock_until`	Try to lock mutex until time point

Mutex – Where?

- For simple programs: global in `main.cpp`

- Member of the class or function
  - `static`: if the mutex protects a global resource (`std::cout, File::read()`);
  - Non-`static`: if only data of the object must be protected (better performance)
  - How to lock `mutex` in `const member function`? Make it `mutable`!

```
public:
  void print() const
  {
    std::lock_guard<std::mutex> l(mtx); // calls mtx.lock()
    std::cout << "print me!";
  }
private:
  mutable std::mutex mtx; // allows change in
                          //   const member function
```

Pushing Data into Threads

Initialization
```
template <class Fn, class... Args>
explicit thread (Fn&& fn, Args&&... args); // arguments are passed down
```

move
```
thread (thread&& x) noexcept; // move runtime of „x“;  „x“ empty afterwards
```

a) global functions + arguments

```
std::atomic<int> global_counter(0);
void add_global(int n) {
  global_counter += n;
}

int main() {
  std::thread t = std::thread( add_global, 1000);
  t.join();
  std::cout << global_counter << std::endl;
  return 0;
}
```

Note:

- By default, all arguments to the thread will be copied and then passed to the function.

Pushing Data into Threads

b) data as reference

```
void add_reference(std::atomic<int>& v, int n) {
  v += n;
}

int main()
{
  std::atomic<int> my_ref = 0;

  std::thread t = std::thread(
    add_reference, std::ref(my_ref), 1000);

  t.join();
  std::cout << my_ref << std::endl;
  return 0;
}
```

Detail:

`std::ref()` needed because without it **`&`** references a copy of my_ref on the new stack.

- Reference must persist **`until t.join()`**
- (or forever when using **`t.detach()`**)

Pushing Data into Threads

c) Member function

```cpp
struct myClass {
  myClass() : v(0) {}
  void add_member(int n) { v += n; }
  std::atomic<int> v;
};

int main() {
  myClass bar;

  std::thread t = std::thread(
    &myClass::add_member, std::ref(bar), 1000);

  t.join();
  std::cout << bar.v << std::endl;
  return 0;
}
```

Note:

- Reference must persist `until t.join()`
- (or forever when using `t.detach()`)

Fetching Output

- Threads have **no return** value – they just run and join…

- Suboptimal solutions so far for return value problem:
  - Global var (bad!)
  - Arguments as reference
  - Member function (access to member vars)

- **Danger Zone**:
  - Scope and lifetime issues!
  - **Exceptions** may happen!

- Better solutions:
  - Instead of using rather low-level abstractions `thread`, `atomic`, `mutex/unique_lock/lock_guard`
  - Use a slightly higher level of abstraction: `future`, `promise`
  - Handling of exceptions

Future & Promise

- A `std::promise` allows
  - to store a **value (**or an **exception).**
  - this can also be provided after the current thread has finished

- A `std::future` allows
  - to ask the Promise if the value is already available
  - to wait until value is available: possible without and with absolute or relative time
  - to retrieve the **value (**or **exception)** of the Promise

> "When someone makes a **`std::promise`**
> you need to wait and see if they honor it in the **`std::future`**."
> some guy on StackOverflow

Example


```cpp
#include <future>
#include <iostream>
#include <thread>
#include <utility>

void product(std::promise <int> &&prom, int a, int b) { prom.set_value(a * b); }
void division(std::promise <int> &&prom, int a, int b) { prom.set_value(a / b); }

int main() {
  int a = 20, b = 10;
  std::promise<int> prodPromise; // define the promises
  std::promise<int> divPromise;

  std::future<int> prodResult = prodPromise.get_future();  // get the futures
  std::future<int> divResult = divPromise.get_future();

  // calculate the result in a separate thread
  std::thread prodThread(product, std::move(prodPromise), a, b);
  std::thread divThread(division, std::move(divPromise), a, b);

  std::cout << "20*10= " << prodResult.get() << std::endl; // get the result
  std::cout << "20/10= " << divResult.get() << std::endl;

  prodThread.join(); divThread.join();
}
```

[see https://www.grimm-jaud.de/index.php/blog/promise-und-future]

`std::async`

- Calls a function (usually asynchronous, or serial if desired)
- Return the result or exception via `future::get()`
- Runtime decides when to run; e.g. might be a queue (many async calls)
- Destructor of the thread will block until thread is done

Example 1st attempt

```
int main() {

  std::cout << std::endl;

  std::async(std::launch::async, [] {
    std::this_thread::sleep_for(std::chrono::seconds(2));
    std::cout << "first thread" << std::endl;
  }); // temporary thread ends here

  std::async(std::launch::async, [] {
    std::this_thread::sleep_for(std::chrono::seconds(1));
    std::cout << "second thread" << std::endl;
  }); // temporary thread ends here

  std::cout << "main thread" << std::endl;
}
```

- Serialized output as temporal thread objects will be effectively block the async

 [see https://www.grimm-jaud.de/index.php/blog/std-async-warten-im-destruktor]

Example 1st attempt

```
int main() {

  std::cout << std::endl;

  std::async(std::launch::async, [] {
    std::this_thread::sleep_for(std::chrono::seconds(2));
    std::cout << "first thread" << std::endl;
  }); // temporary thread ends here

  std::async(std::launch::async, [] {
    std::this_thread::sleep_for(std::chrono::seconds(1));
    std::cout << "second thread" << std::endl;
  }); // temporary thread ends here

  std::cout << "main thread" << std::endl;
}
```

```
slow thread
fast thread
main thread
```

?!?

- Serialized output as temporal thread objects will wait at end of lifetime

[see https://www.grimm-jaud.de/index.php/blog/std-async-warten-im-destruktor]

Example 2nd attempt

```cpp
int main() {

  std::cout << std::endl;

  auto first = std::async(std::launch::async, [] {
    std::this_thread::sleep_for(std::chrono::seconds(2));
    std::cout << "first thread" << std::endl;
  });

  auto second = std::async(std::launch::async, [] {
    std::this_thread::sleep_for(std::chrono::seconds(1));
    std::cout << "second thread" << std::endl;
  });

  std::cout << "main thread" << std::endl;
}
```

- This should extend lifetime of temporary thread object...

 [see https://www.grimm-jaud.de/index.php/blog/std-async-warten-im-destruktor]

Example 2nd attempt

```
int main() {

  std::cout << std::endl;

  auto first = std::async(std::launch::async, [] {
    std::this_thread::sleep_for(std::chrono::seconds(2));
    std::cout << "slow thread" << std::endl;
  });

  auto second = std::async(std::launch::async, [] {
    std::this_thread::sleep_for(std::chrono::seconds(1));
    std::cout << "fast thread" << std::endl;
  });

  std::cout << "main thread" << std::endl;
}
```

```
main thread
fast thread
slow thread
```

- now its really asynchronous

[see https://www.grimm-jaud.de/index.php/blog/std-async-warten-im-destruktor]

Example: async + future

```
#include <chrono>
#include <future>
#include <iostream>
#include <thread>

// sleeps for 500ms
bool takeANap() {
  std::this_thread::sleep_for(std::chrono::milliseconds(500));
  return true;
}

int main() {
  std::cout << "start" << std::endl;

  // takeANap and get the std::future
  auto napFuture = std::async(std::launch::async, takeANap);

  // do other stuff until future is ready…
  do
  {
    std::cout << "idle" << std::endl;
  } while (napFuture.wait_for(std::chrono::milliseconds(25)) != std::future_status::ready);

  if (copyFuture.get())
  {
    std::cout << "got future" << std::endl;
  }
}
```

```
start
idle
idle
idle
idle
idle
idle
idle
idle
idle
idle
idle
idle
idle
idle
idle
idle
idle
idle
idle
idle
got future
```

- async is often used with future (+ promise). Any idea why?

Example – Returning exceptions

```
#include <chrono>
#include <future>
#include <iostream>
#include <thread>

// sleeps for 500ms
void takeANap() {
  std::this_thread::sleep_for(std::chrono::milliseconds(500));
  throw std::runtime_error("not tired"); // gets stored in (internal) promise and will be available in the future
}

int main() {
  std::cout << "start" << std::endl;

  auto napFuture = std::async(std::launch::async, takeANap);

  do {
    std::cout << "idle" << std::endl;
  } while (napFuture.wait_for(std::chrono::milliseconds(25)) != std::future_status::ready);

  try {
    if (copyFuture.get() { // throws
      std::cout << "got future" << std::endl;
    }
  }
  catch (const std::exception& e) {
      std::cout << "exception: " << e.what() << std::endl;
  }
}
```

Condition Variables <condition_variable>

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- Used for synchronization / communication between threads

- Works in tandem with a mutex (at ‘global’ scope)

- Usage:
  - Blocks the thread (put to sleep/spinning) until a **wakeup signal** from another thread
  - Wake up one/all other threads

- Prevents spins with 100% core usage and sleeping with fixed duration (more on this in exercises …)

function	operation
`notify_one`	Notifies one of the waiting threads
`notify_all`	Notifies all waiting threads
`wait`	Blocks the current thread until the cv is woken up
`wait_for`	… until woken up or after some duration
`wait_until`	… until woken up or time point

Condition Variable – Example

```
std::mutex mtx;
std::condition_variable cv;
bool ready{false};

void print_id(int id) {
  std::unique_lock<std::mutex> lck(mtx);
  while (!ready)
    cv.wait(lck);
  // ...
  std::cout << "thread " << id << '\n';
}

void go() {
  // prepare something big here ...
  std::unique_lock<std::mutex> lck(mtx);
  ready = true;
  cv.notify_all();
}
```

```
int main() {
  // spawn 10 threads:
  std::thread threads[10];
  for (int i = 0; i < 10; ++i) {
    threads[i] = std::thread(print_id, i);
  }
  std::cout << "10 threads ready to race...\n";
  int i; std::cin >> i; // wait for input
  go(); // go!
  for (auto &th : threads) { th.join() };
}
```

[www.cplusplus.com]

- Wait for **`mtx.lock()`** → syncs
- **`cv.wait()`**
  - **`unlock()`**
  - **`sleep()`**
  - **`lock()`**

  } before signal (unlock(), sleep()) } after signal (lock())
- Wait for **`mtx.lock`**; when done: all threads in **`print_id()`** are sleeping!
- Happens within **`mtx.lock()`**; i.e. its synced!
- All other threads wake up (and call **`mtx.lock()`**)

Parallel STL

- Easy parallelization for simple cases
- (since C++17)

Parallel STL

- Since C++-17 most STL algorithms support parallel execution natively

- E.g. standard operation

```
template< class InputIt, class UnaryFunction >
UnaryFunction for_each( InputIt first, InputIt last, UnaryFunction f );
```

augmented by an execution policy

```
template< class ExecutionPolicy, class InputIt, class UnaryFunction2 >
void for_each( ExecutionPolicy&& policy, InputIt first, InputIt last, UnaryFunction2 f );
```

Execution Policies

`std::execution::seq`

- Sequential execution, default
- Benefit: correctness (no race conditions), no parallelization overhead, simplicity

`std::execution::par`

- Parallel execution (might however be ignored)
- You have to ensure that all invocations do not violate any data dependencies

`std::execution::par_unseq`

- Requires even stronger guarantees than par
- Ordre of operations even within the same thread might be scambled
- Benefit: Might use vectorization / SIMD or migration to other threads

```
const double factor{2.0};
for_each( execution::par_unseq, begin(v), end(v),
    [factor](T& val) { val *= factor; }
);
```

Example – Parallel STL

- helper function

```
#include <algorithm>
#include <chrono>
#include <cmath>
#include <execution>
#include <iostream>
#include <random>
#include <string>
#include <vector>

constexpr long long size = 1e7;
const double pi = std::acos(-1);

template <typename Func>
void getExecutionTime(const std::string &title, Func func) { // (4)
  const auto sta = std::chrono::steady_clock::now();
  func(); // (5)
  const std::chrono::duration<double> dur =
      std::chrono::steady_clock::now() - sta;
  std::cout << title << ": " << dur.count() << " sec. " << std::endl;
}
```

 [https://www.modernescpp.com/index.php/performance-of-the-parallel-stl-algorithm]

Example – Parallel STL

```
int main() {
  std::vector<double> randValues;  randValues.reserve(size);

  std::mt19937 engine;
  std::uniform_real_distribution<> uniformDist(0, pi / 2);
  for (long long i = 0; i < size; ++i)
    randValues.push_back(uniformDist(engine));

  std::vector<double> workVec(randValues);

  getExecutionTime("std::execution::seq", [workVec]() mutable {          // (6)
    std::transform(std::execution::seq, workVec.begin(), workVec.end(), // (1)
                   workVec.begin(), [](double arg) { return std::tan(arg); });
  });

  getExecutionTime("std::execution::par", [workVec]() mutable {          // (7)
    std::transform(std::execution::par, workVec.begin(), workVec.end(), // (2)
                   workVec.begin(), [](double arg) { return std::tan(arg); });
  });

  getExecutionTime("std::execution::par_unseq", [workVec]() mutable { // (8)
    std::transform(std::execution::par_unseq, workVec.begin(),
                   workVec.end(), // (3)
                   workVec.begin(), [](double arg) { return std::tan(arg); });
  });
}
```

```
std::execution::seq: 0.64093 sec.
std::execution::par: 0.140672 sec.
std::execution::par_unseq: 0.111494 sec.
```

[https://www.modernescpp.com/index.php/performance-of-the-parallel-stl-algorithm]

Summary

- OpenMP
  - Simple operations to parallelize your loops or different sections

- C++ Thread Model – some manual work
  - **`join, detach, async`**
  - **`atomic`**
  - **`mutex`**
  - Future, Promise, Async, dealing with exceptions in parallel code

- Process synchronization
  - Locks and condition variables need careful thinking!

- Parallel STL
  - Few changes for your code
  - Be aware of data dependence between elements

Resources

- Introduction to OpenMP: 01 Introduction
  - Tim Mattson (Intel, OpenMP committee)
  - https://www.youtube.com/watch?v=nE-xN4Bf8XI&index=1&list=PLLXQ6B8xqZ8n8bwjGdzBJ25X2utwnoEG

- Wikipedia
  - Summary of OpenMP commands
  - https://en.wikipedia.org/wiki/OpenMP