Introduction to High Performance Computing

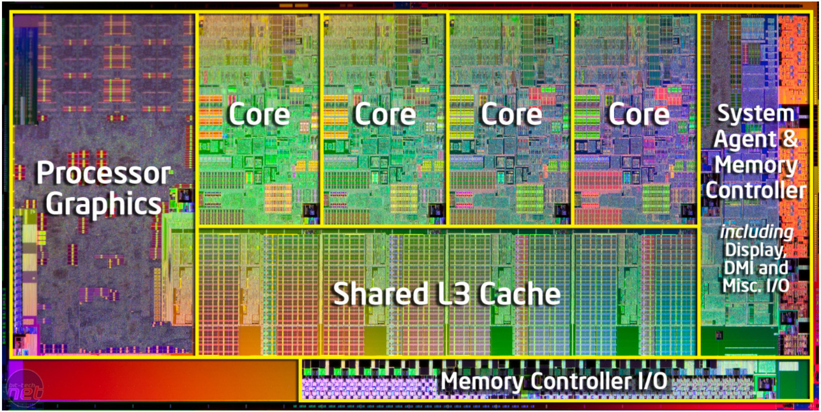
Lecture 4

Jakub Gałecki

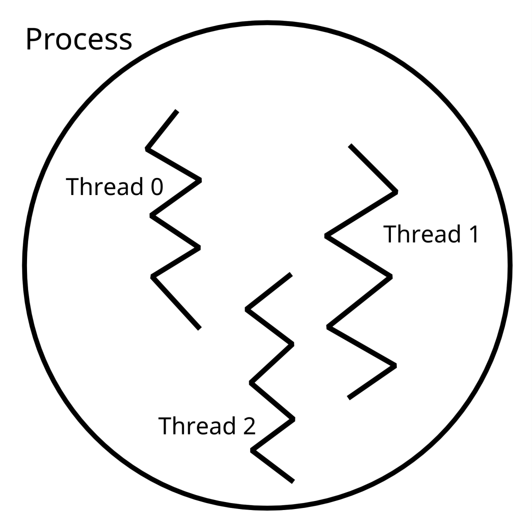
Multithreading

- Basics: core, process, thread
- Data races and how to avoid them
- Low-level concurrency facilities
- The importance of structured parallelism
- Intel TBB
- Strong and weak scaling
- Miscellanea

One CPU, multiple cores



Process vs. thread



Process vs. thread

Process

- Has its own address space
- Walled off from other processes
- consists of at least one thread
- When you launch an application, you launch a process (and its main thread)

Thread

- Smallest logical unit of execution, single stream of instructions
- Belongs to a process
- Stateful (regs, stack, …)
- **Shares the address space of the process with other threads**

The OS is aware of and manages both processes and their threads

Context switching

Example

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

void doWork(int i) { /* hard work */ }

void doWorkChunk(int begin, int end) {
  for (; begin != end; ++begin)
    doWork(begin);
}

int main() {
  std::thread t1{doWorkChunk, 0, 1000};
  std::thread t2{doWorkChunk, 1000, 2000};
  t1.join();
  t2.join();
  std::cout << "Work complete!\n";
}
```

Example

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

int main() {
  const auto id = [](int i) {
    std::cout << "Hello from thread "
              << i << '\n';
  };
  std::thread t1{id, 1};
  std::thread t2{id, 2};
  t1.join();
  t2.join();
}
```

```
Hello from thread 1
Hello from thread 2
```

```
Hello from thread 2
Hello from thread 1
```

```
Hello from thread Hello from thread 12
```

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

int main() {
  const auto id = [] {
    std::cout << std::this_thread::get_id()
              << '\n';
  };
  std::jthread t1{id};
  std::jthread t2{id};
}
```

```
139765188224576
139765179831872
```

Example

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

int main() {
  int a = 0;
  const auto inc = [&](int i) {
    for (int i = 0; i < 1'000'000; ++i)
      ++a;
  };
  {
    std::jthread t1{inc, 1};
    std::jthread t2{inc, 2};
  }
  std::cout << a << '\n';
}
```

```
1004817
```

Data race

A data race occurs iff 2 threads perform an operation on the same memory location and at least one of them is a write operation

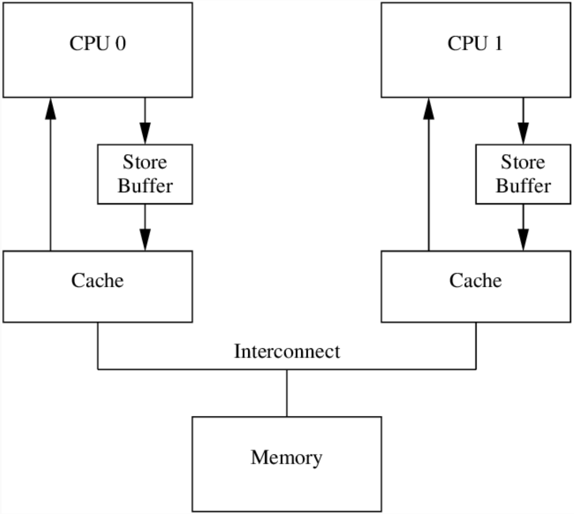
Data races fundamentally occur at the hardware level

Thread synchronization is critical

We need to structure our multithreaded programs to:

- **avoid data races**
- ensure correctness
- have good performance

Cache coherency



Atomic operations

Since data races are a hardware problem, they require a hardware solution

Multi-core architectures must support special instructions, which perform the following 3 steps as a single, indivisible, **atomic** operation:

1. Read value from a memory location
2. Modify this value
3. Write the result back to the same memory location

Example: concurrently increment a counter from multiple threads

Example

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

int main() {
  std::atomic<int> a{0};
  const auto inc = [&] {
    for (int i = 0; i < 1'000'000; ++i)
      ++a;
  };
  {
    std::jthread t1{inc};
    std::jthread t2{inc};
  }
  std::cout << a << '\n';
}
```

```
2000000
```

[Do not do this in real code]

Spinlock

Do not do this in real code

```
class BadSpinlock {
public:
  void lock() {
    while (flag.exchange(true))
      ;
  }
  void unlock() { flag.store(false); }

private:
  std::atomic<bool> flag_{};
};
```

This is an example of a **lock**: we can “turn the key” and be sure that we are the only ones “inside”

Code protected by locks is called the **critical section**

Why is the code from the previous slide problematic?

Spinlock – problems

Why is the code from the previous slide problematic?

- Waiting threads are spinning – busy-wait
- From the point of view of the OS, these threads are working hard
- `lock` and `unlock` are fighting over the same memory location (performance)

Waiting threads should ideally not take up CPU time

This fundamentally requires help from the OS

Solution: **mutex**

Summary of concurrent facilities in the STL

- `jthread` - manage a thread (start, join, request stop)
- `mutex` - mutual exlusion lock
- `semaphore` - constrains access to a resource
- `barrier/latch` - synchronize a team of threads
- `promise` + `future` - synchronize work between a producer and consumer
- `condition_variable` - wait until a condition is met
- `atomic` - fine-grained atomic operations

Hash a vector of strings. Seems easy, right?

Conclusions

- Launching a thread is **very** expensive
- Manually managing threads is a bad idea
- Statically partitioning work can be inefficient
- We need structured parallelism!

Structured concurrency

We need a programming model which:

- Manages an internal thread pool
- Exposes a high-level, expressive API
- Abstracts away the details
- Has good performance
  - Load balancing
  - Work queue
- Is portable
- Is tunable

Multithreading paradigms

OpenMP

- Ships as part of the compiler
- Compiler pragmas + C API
- Can be tuned via environment variables
- Issues around composability

Intel TBB (OneAPI)

- C++ library
- Task-based parallelism
- Fully composable
- Callable via STL algos (GCC)

Let's start with a simple **for** loop:

```
#include "oneapi/tbb.h"

#include <vector>

struct S {};

void foo(S &s) {}

void fooVector(std::vector<S> &vec) {
  for (S &s : vec)
    foo(s);
}

void fooVectorPar(std::vector<S> &vec) {
  using range_t = oneapi::tbb::blocked_range<size_t>;
  const auto foo_range = [&vec](const range_t &r) {
    const auto end = r.end();
    for (auto i = r.begin(); i != end; ++i)
      foo(vec[i]);
  };
  oneapi::tbb::parallel_for(range_t{0, vec.size()}, foo_range);
}
```

`blocked_range` represents an iteration range

This iteration range can be split – this is controlled by the TBB scheduler

Composability – we can call a `parallel_for` within a `parallel_for` (or use a 2D range where appropriate)

`parallel_reduce` works in a similar way

This is still not as expressive as we'd ideally like…

This looks better:

```cpp
#include "oneapi/tbb.h"

#include <algorithm>
#include <ranges>
#include <vector>

struct S {};

void foo(S &s) {}

void fooVector(std::vector<S> &vec) { std::ranges::for_each(vec, &foo); }

void fooVectorPar(std::vector<S> &vec) {
  oneapi::tbb::parallel_for_each(vec, &foo);
}
```

Calling TBB via the STL

Since C++17, many STL algorithms accept an additional leading argument representing an **execution policy**

Execution policies represent parallelization strategies we'd like to use for a given algorithm (`execution` header):

- `std::execution::seq` – execute sequentially, possibly out of order
- `std::execution::unseq` (C++20) – no sequence guarantee (SIMD)
- `std::execution::par` – parallelize
- `std::execution::par_unseq` – parallelize & vectorize

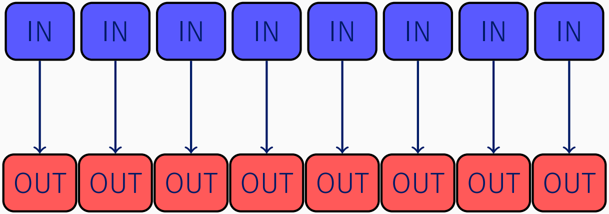
Execution policies are not binding, but in practice we usually get what we expect

In gcc, parallel algorithms call into TBB (remember to link against it)

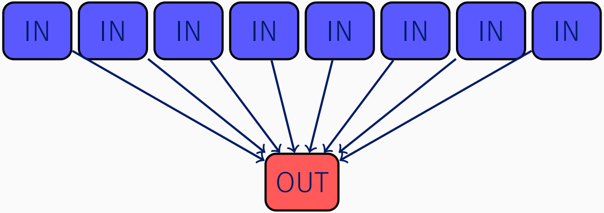
We get the expressiveness of the STL with the parallel performance of TBB essentially for free!!!

Execution policies are extensible

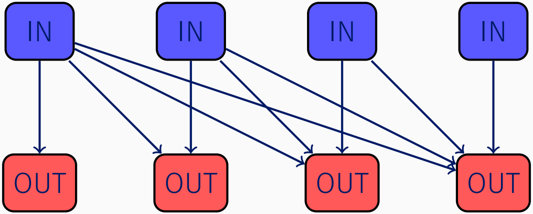
Algorithm: map (`for_each`)



Algorithm: reduction



Algorithm: scan



TBB – topics not covered

- Multidimensional ranges
- Flow graphs
- Work separation – task groups and arenas
- Concurrent containers
- Controlling the partitioner


Multithreaded programming
Theory
Actual
STARECAT.COM

- Avoid synchronization as much as possible
- Parallelize outer loops first
- Scalar optimization concerns still apply within a thread
- Threads should sequentially operate on contiguous chunks of data

Speedup:

$$S(n) = \frac{t_1}{t_n}$$

Strong scaling: Amount of work stays constant, number of parallel agents (workers) increases

Applications can only strongly scale up to a point (Amdahl)

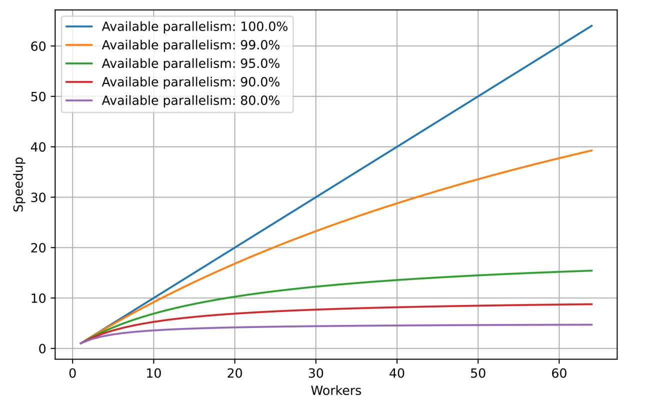
Weak scaling: Amount of work increases proportionally to the number of parallel agents

Weak scaling is not inherently limited (Gustafson)

Amdahl's law

Given constant work which has available parallelism p and n workers, the highest speedup we can achieve is

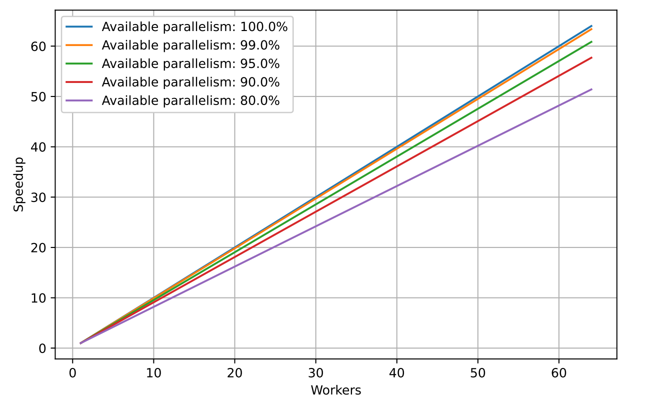
$$S(p,n) = \frac{1}{(1-p) + \frac{p}{n}}$$



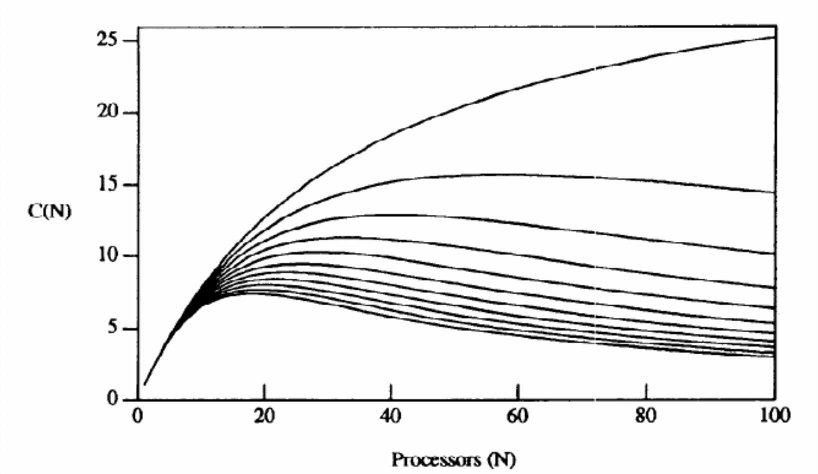
Gustafson's law

Given work which has available parallelism p and n workers, assuming that the amount of work increases proportionally to the number of workers, the highest speedup we can achieve is

$$S(p, n) = 1 + (n - 1)p$$



Universal scalability law



N. J. Gunther, "A Simple Capacity Model of Massively Parallel Transaction Systems," CMG National Conference, 1993

SMT

SMT – simultaneous multithreading (Intel: hyperthreading)

The CPU has multiple (usually 2) logical cores (hardware threads) per physical core

OS sees logical cores and can schedule different threads independently [1]

Opportunities:

- better exploitation of ILP
- less context-switching

Potential pitfalls:

- resources (cache!) are not duplicated
- power consumption

[1] The OS scheduler is aware of SMT and can optimize accordingly

Cores can have private memory caches, or share caches with other cores (the latter is straightforward to leverage)

The CPU must maintain a coherent picture of the memory regardless of caching effects (e.g. MESI protocol)

If we write to a memory location, other cores' caches need to be updated... **true sharing**

But caches operate on cachelines, not bytes! **false sharing**

Demo...

- Multithreading is hard
- Manually managing threads is usually a bad idea
- Use structured parallelism, only reach for low-level primitives when necessary

Q&A