Paralelní a distribuované výpočty (B4B36PDV)

Jakub Mareček
jakub.marecek@fel.cvut.cz

Artificial Intelligence Center
Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University in Prague

In the previous lecture

- Parallel programming
  - Aim: speeding up computation
  - The influence of various approaches on the speedup
- Distributed programming
  - Aim: consistency across a large number of machines
  - The algorithms in state-of-the-art database engines

- CourseWare
  - https://cw.fel.cvut.cz/wiki/courses/b4b36pdv/start

In the previous lecture

You have seen the textbook

1.2 A Fable

Instead of treating coordination problems (such as mutual exclusion) as programming exercises, we prefer to think of concurrent coordination problems as if they were physics problems. We now present a sequence of fables, illustrating some of the basic problems. Like most authors of fables, we retell stories mostly invented by others (see the Chapter Notes at the end of this chapter).

Alice and Bob are neighbors, and they share a yard. Alice owns a cat and Bob owns a dog. Both pets like to run around in the yard, but (naturally) they do not get along. After some unfortunate experiences, Alice and Bob agree that they should coordinate to make sure that both pets are never in the yard at the same time. Of course, we rule out trivial solutions that do not allow any animals into an empty yard.

How should they do it? Alice and Bob need to agree on mutually compatible procedures for deciding what to do. We call such an agreement a *coordination protocol* (or just a *protocol*, for short).

The yard is large, so Alice cannot simply look out of the window to check whether Bob's dog is present. She could perhaps walk over to Bob's house and knock on the door, but that takes a long time, and what if it rains? Alice might lean out the window and shout "Hey Bob! Can I let the cat out?" The problem is that Bob might not hear her. He could be watching TV, visiting his girlfriend, or out shopping for dog food. They could try to coordinate by cell phone, but the same difficulties arise if Bob is in the shower, driving through a tunnel, or recharging his phone's batteries.

Alice has a clever idea. She sets up one or more empty beer cans on Bob's windowsill (Fig. 1.4), ties a string around each one, and runs the string back to her house. Bob does the same. When she wants to send a signal to Bob, she yanks the string to knock over one of the cans. When Bob notices a can has been knocked over, he resets the can.

Up-ending beer cans by remote control may seem like a creative solution, but it is still deeply flawed. The problem is that Alice can place only a limited number of cans on Bob's windowsill, and sooner or later, she is going to run out of cans to knock over. Granted, Bob resets a can as soon as he notices it has been knocked over, but what if he goes to Cancún for Spring Break? As long as Alice relies on Bob to reset the beer cans, sooner or later, she might run out.

So Alice and Bob try a different approach. Each one sets up a flag pole, easily visible to the other. When Alice wants to release her cat, she does the following:

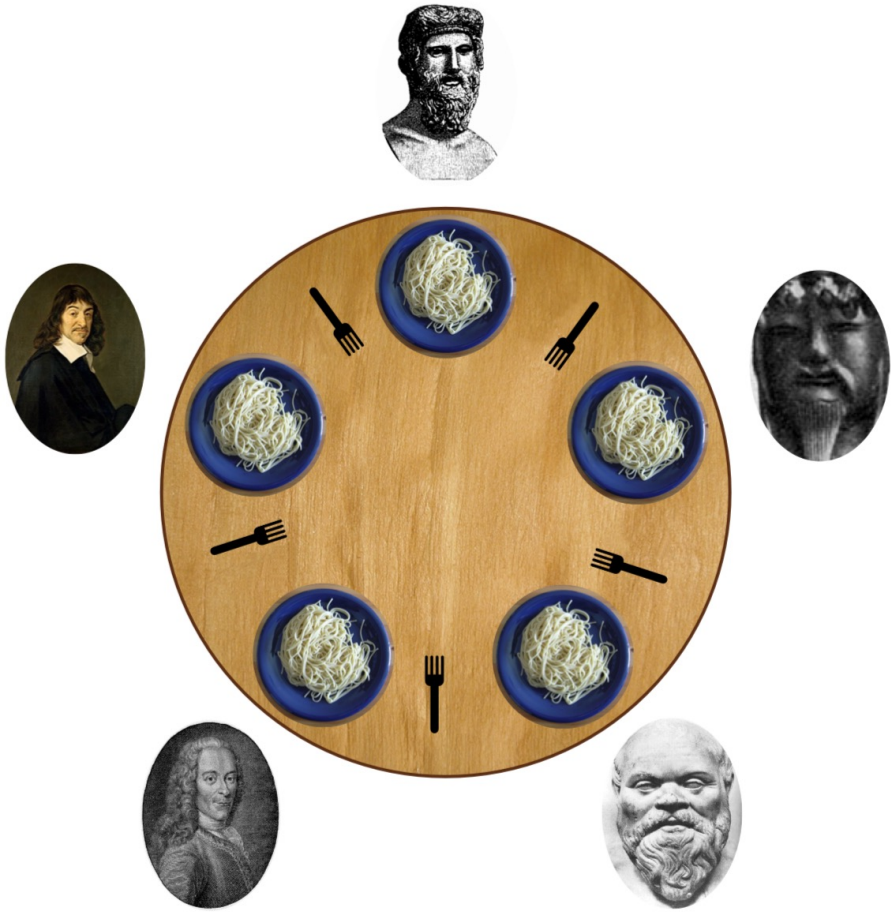
1. She raises her flag.
2. When Bob's flag is lowered, she unleashes her cat.
3. When her cat comes back, she lowers her flag.

Bob's behavior is a little more complicated.

1. He raises his flag.
2. While Alice's flag is raised
   a) Bob lowers his flag
   b) Bob waits until Alice's flag is lowered
   c) Bob raises his flag
3. As soon as his flag is raised and hers is down, he unleashes his dog.
4. When his dog comes back, he lowers his flag.

In the previous lecture

You have seen the textbook



https://upload.wikimedia.org/wikipedia/commons/7/7b/An_illustration_of_the_dining_philosophers_problem.png

The Concepts

- Parallelism means two or more tasks can be executed simultaneously. This is an option, which the compiler and operating system and processor can exercise, but does not come with any guarantees.
- Often, this means no shared variables or other resources, and need not require any synchronization primitives.

- Concurrency means that two or more tasks start, run, and complete in overlapping time periods, while sharing some resources.
- If two tasks concurrently set shared variable x to 1 and 2, it is not clear what value it would have, subsequently.
- More broadly, concurrent access to a mutable shared memory can result in issues without the use of synchronization primitives (data race, problém souběhu) and with the use of synchronization primitives (deadlock, uváznutí).

Data Race

Problém souběhu

When we need to ensure mutual exclusion in access to two or more shared mutable variables, e.g., read value of one of the variables and add it to another variable, we may need to use some synchronization primitives (e.g., mutexes).Without the use of synchronization primitives, we are facing the risk of a data race.

For example, consider the a silly bank without a solid relational database management system, where there are three clients: Alice and Bob and Corporation C.

- Transaction T1: Bob has $100 in his account, but will be paying a $50 bill to Corporation C. At the same time, in
- Transaction T2, Alice will be paying $100 to Bob.

Depending on the ordering of the reading and writing operations, one may obtain several outcomes.

Data Race

Problém souběhu

For example, consider the a silly bank without a solid relational database management system, where there are three clients: Alice and Bob and Corporation C.

- Transaction T1: Bob has $100 in his account, but will be paying a $50 bill to Corporation C. At the same time, in
- Transaction T2, Alice will be paying $100 to Bob.

Depending on the ordering of the reading and writing operations, one may obtain several outcomes:

- Transaction T1 will read $100 valued of Bob's account. Transaction T2 will read $100 value. Transaction T1 will write $200. Transaction T2 will write $50 value.
- Transaction T1 will read $100 valued of Bob's account. Transaction T1 will write $50. Transaction T2 will read $50 value.Transaction T2 will write $150 value.
- Transaction T1 will read $100 valued of Bob's account. Transaction T2 will read $100 value. Transaction T1 will write $50. Transaction T2 will write $200 value.
- Transaction T2 will read $100 value. Transaction T2 will write $200 value. Transaction T1 will read $200 valued of Bob's account. Transaction T1 will write $150.

Either Bob or the bank could be up to $100 short.

Deadlock

Problém uváznutí

- When we need to ensure mutual exclusion in access to two or more shared variables, e.g., two temporary results associated with two mutexes, one may naively lock the first mutex first, and subsequently lock the other mutex.
- This, however, can lead to a deadlock.
- Instead, one needs to lock both mutexes at the same time.
- Easily, one could run:

Locking multiple mutexes at once.

```
1  void thread_operation(){
2      std::lock(mutex1,mutex2);
3       ...
4      complicated_task();
5       ...
6      mutex1.unlock();
7      mutex2.unlock();
8  }
```

Open in Compiler Explorer

Deadlock

In Theory

In theory, a deadlock (Czech: ``problém uváznutí'') can occur when:

- each lock is owned by one thread
- each thread has locked at least one lock and needs to lock at least one more lock
- it is impossible to remove the lock ownership
- there is a cyclic dependency among the lock-using threads.

Amdahl's law

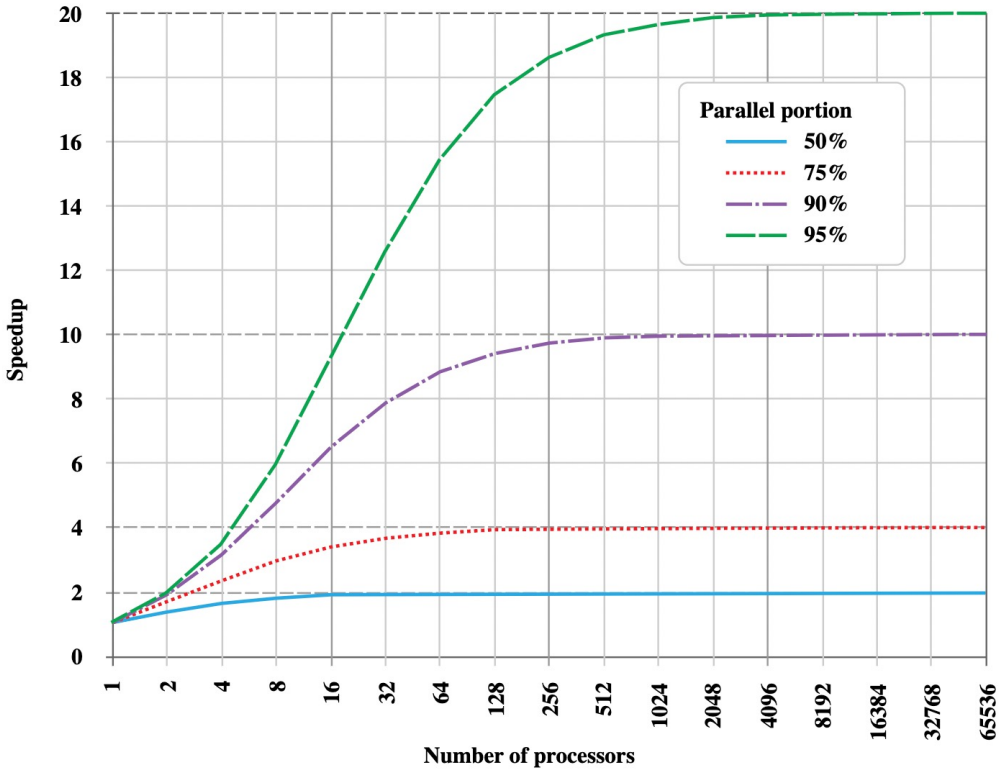
In Theory

- There is almost always some overhead in parallel programming (e.g., synchronization primitives)
- There is almost always some nonparallelizable code:
  - Let us consider the speed-up $S = \frac{T_{serial}}{T_{parallel}}$ of the parallel code
  - E.g. if 10% of the code is nonparallelizable and there are p processors (hardware threads):
  - $S = \frac{T_{serial}}{0.9 \times \frac{T_{serial}}{p} + 0.1 \times T_{serial}} \leq \frac{T_{serial}}{0.1 \times T_{serial}}$
  - In general, for a fraction n of nonparallelizable code and p processors (hardware threads):
  - $S = \frac{T_{serial}}{(1-n) \times \frac{T_{serial}}{p} + n \times T_{serial}} \leq \frac{T_{serial}}{n \times T_{serial}}$

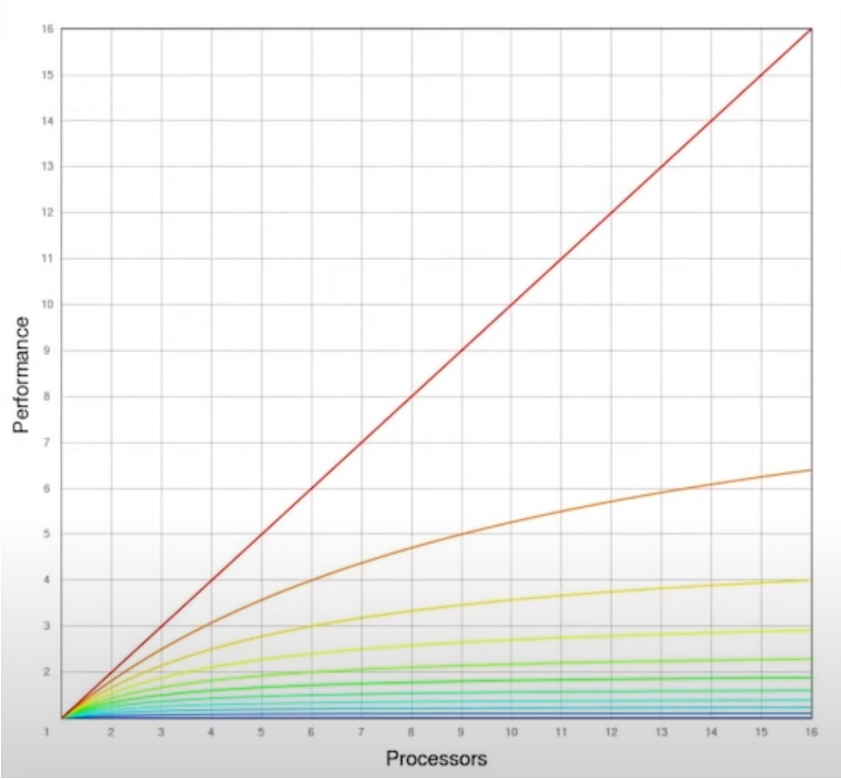
Amdahl's law

In the previous lecture

Log-linear plot for certain proportions of non-parallelizable code.



Linear plot, for multiples of 10% of non-parallizable code



Grafy z:

- https://upload.wikimedia.org/wikipedia/commons/e/ea/AmdahlsLaw.svg
- https://www.youtube.com/watch?v=QIHy8pXbneI

Concurrent programming

The Options

There are two essential models for concurrent programming: shared memory and message passing. In sharing memory, we have broadly four options:

- **Confinement**: Do not share memory between threads. This is often impossible.
- **Immutability**: Do not share any mutable data between threads.
- **Thread-safe code:** Use data types with additional guarantees for storing any mutable data shared between threads, or even better, use implementations of algorithms that are already parallelized and handle the concurrency issues for you.
  For example in C++, one can use the standard template library with a suitable execution policy.
  In particular, the header **execution** defines objects **std::execution::seq**, **std::execution::par**, **std::execution::par_unseq**, which can be passed as the first argument of any standard algorithm, e.g., **std::vector<int> v std::sort(std::execution::par, v.begin(), v.end());**
- **Synchronization**: Use synchronization primitives to prevent accessing the variable at the same time. This option is explored in this chapter in more detail.

Eventually, we will see that message passing can be implemented using the synchronization primitives and may be the least challenging to use correctly.

Concurrent programming

The Options Revisited

HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent

Feng Niu, Benjamin Recht, Christopher Re, Stephen J. Wright

Stochastic Gradient Descent (SGD) is a popular algorithm that can achieve state-of-the-art performance on a variety of machine learning tasks. Several researchers have recently proposed schemes to parallelize SGD, but all require performance-destroying memory locking and synchronization. This work aims to show using novel theoretical analysis, algorithms, and implementation that SGD can be implemented without any locking. We present an update scheme called HOGWILD! which allows processors access to shared memory with the possibility of overwriting each other's work. We show that when the associated optimization problem is sparse, meaning most gradient updates only modify small parts of the decision variable, then HOGWILD! achieves a nearly optimal rate of convergence. We demonstrate experimentally that HOGWILD! outperforms alternative schemes that use locking by an order of magnitude.

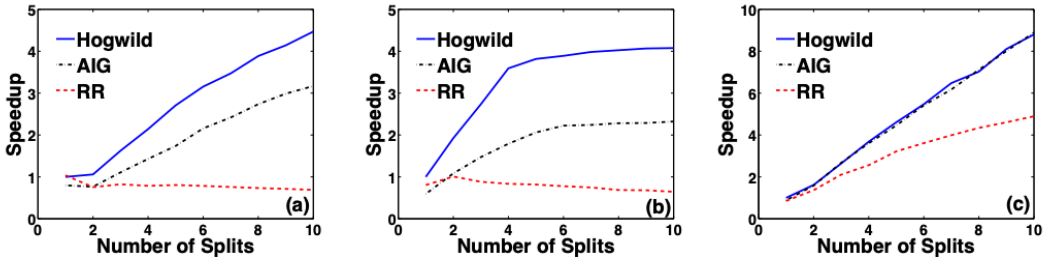


Figure 2: Total CPU time versus number of threads for (a) RCV1, (b) Abdomen, and (c) DBLife.

Lock-free approaches more broadly:
https://www.youtube.com/watch?v=Yl8Or0afcfg&ab_channel=ChurchillCompSciTalks

Structuring code

Processes, Threads, Tasks, Coroutines

- Processes, threads, tasks, and coroutines execute instructions.
- A **process** provides all of the prerequisites for executing instructions:
  loads an executable program,
  sets up a virtual address space,
  sets up the environment (e.g. environment variables and a security context),
  sets up the process control block (PCB, often stored in registers of the processor and on a per-process stack in kernel memory), opens handles to system objects (e.g., files, sockets), and often much more.
- In some sense, one can imagine ``a virtual machine ‘ ’.
- All modern operating systems (OS) are multitasking, i.e., running multiple processes with the operating system forcibly interrupting the run one one process to execute another process after a certain amount of time (``preemptive scheduling''). Switching between the processes involves swapping the process control block (PCB). In Intel architectures, this is known as the task state segment (TSS), and there is hardware support for the switch. AMD64 does not support task switches in hardware.

Structuring code

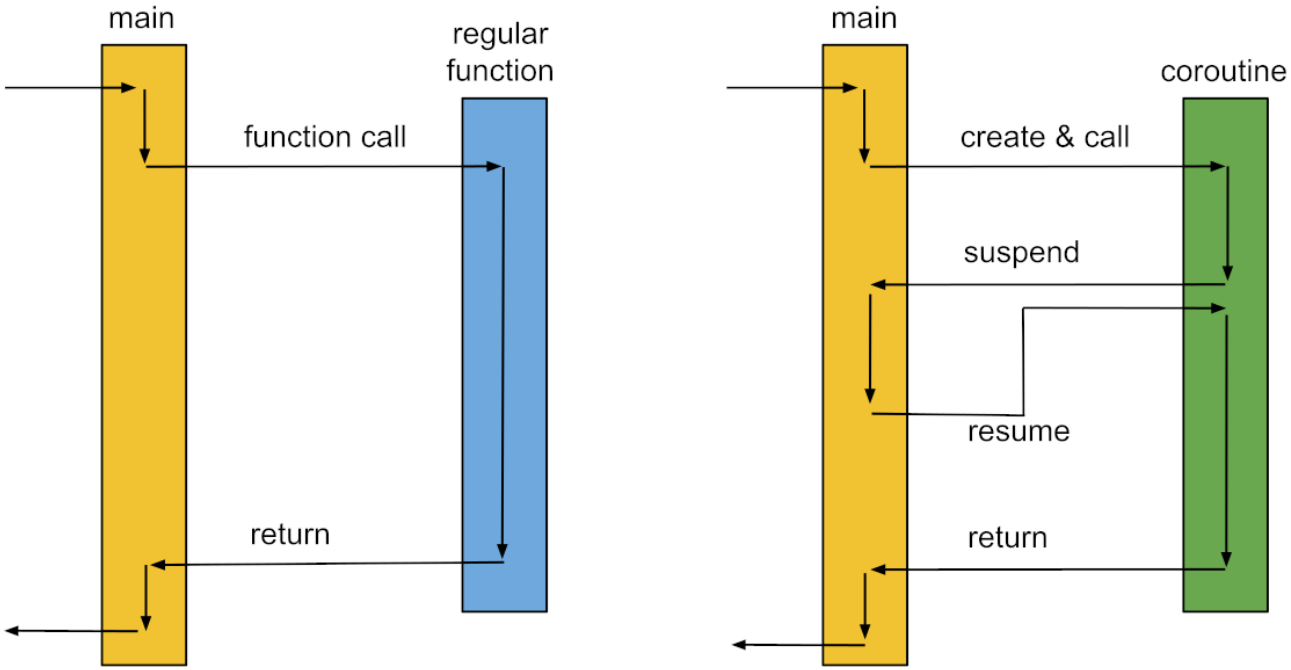
Processes, Threads, Tasks, Coroutines

- Within a particular process, there is at least one **thread**. All threads of a particular process share the same virtual address space and handles to system objects. Each thread, independently, operates its own context (registers, stack, exception handlers).
- Unless declared otherwise, threads of a particular process share memory and are allocated ``time slices'' by the operating system.
- This can be seen as a ``virtual processor'' within a ``a virtual machine'' of a process, often with no guarantees on the time slicing.
- Most modern processors are multi-core and support multithreading in some form. This means that each process can execute multiple ``**hardware threads**'' and there is some support for switching between those. In Intel architectures, hyper-threading means each hardware core can execute multiple threads, e.g., two, to take advantage of idle time (e.g., loading data, network communications).

Structuring code

Processes, Threads, Tasks, Coroutines

- Within a particular thread, one may utilize multiple **coroutines**, which can be seen as subroutines that can run in multiple steps, but sometimes can serve as a light-weight alternative to hardware threads.



https://blog.eiler.eu/posts/20210512/images/coroutines.png

Structuring code

Processes, Threads, Tasks, Coroutines

- Within a particular thread, one may utilize multiple **coroutines**, which can be seen as subroutines that can run in multiple steps, but sometimes can serve as a light-weight alternative to hardware threads. Coroutines can be called, can return when completed, but also can suspend themselves, yielding control and partial results, and be resumed by another co-routine. Typical uses involve generators andfactories and various other concepts within ``lazy evaluation'', as well as event-driven architectures within cooperative multi-tasking.
- That is: two coroutines within one thread never run in parallel, but one can have the runs of two or more coroutines interleaved. We can suspend a co-routine in one thread and resume it within another thread.
- As it turns out, the ``context switch'' with user-level threads has a similar cost to a function call or suspending a coroutine (**co_yield**). Indeed, coroutines are typically implemented with user-level threads, which leads to cheaper context-switch compared with hardware threads. Within the user-level threads, one can distinguish stackful and stackless versions, where coroutine state is saved on the heap (as in C++).

Structuring code

Processes, Threads, Tasks, Coroutines

- Coroutines can be called, can return when completed, but also can suspend themselves, yielding control and partial results, and be resumed by another co-routine.
- Typical uses involve generators and factories and various other concepts within ``lazy evaluation'', as well as event-driven architectures within cooperative multi-tasking.
- That is: two coroutines within one thread never run in parallel, but one can have the runs of two or more coroutines interleaved. We can suspend a co-routine in one thread and resume it within another thread.
- As it turns out, the ``context switch'' with user-level threads has a similar cost to a function call or suspending a coroutine (**co_yield**). Indeed, coroutines are typically implemented with user-level threads, which leads to cheaper context-switch compared with hardware threads. Within the user-level threads, one can distinguish stackful and stackless versions, where coroutine state is saved on the heap (as in C++).

Structuring code

Processes, Threads, Tasks, Coroutines

```
#include <coroutine>
#include <iostream>

// The caller-level type
struct Generator {
    // The coroutine level type
    struct promise_type {
        using Handle = std::coroutine_handle<promise_type>;

        Generator get_return_object() {
            return Generator{Handle::from_promise(*this)};
        }
        std::suspend_always initial_suspend() { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        std::suspend_always yield_value(int value) {
            current_value = value;
            return {};
        }
        void unhandled_exception() { }
        int current_value;
    };

    explicit Generator(promise_type::Handle coro) : coro_(coro) {}
    // Make move-only
    Generator(const Generator&) = delete;
    Generator& operator=(const Generator&) = delete;
    Generator(Generator&& t) noexcept : coro_(t.coro_) { t.coro_ = {}; }
    Generator& operator=(Generator&& t) noexcept {
        if (this == &t) return *this;
        if (coro_) coro_.destroy();
        coro_ = t.coro_;
        t.coro_ = {};
        return *this;
    }

    int get_next() {
        coro_.resume();
        return coro_.promise().current_value;
    }

private:
    promise_type::Handle coro_;
};
```

```
Generator myCoroutine() {
    int x = 0;
    while (true) {
        co_yield x++;
    }
}

int main() {
    auto c = myCoroutine();
    int x = 0;
    while ((x = c.get_next()) < 10) {
        std::cout << x << "\n";
    }
}
```

Structuring code

Processes, Threads, Tasks, Coroutines

An example of the use of coroutines, which currently does not compile in GCC 12.2.

```cpp
#include <coroutine>
#include <generator>
#include <iostream>
#include <syncstream>

std::generator<int> work() {
    for (int i = 0; i < 10; i++) {
        co_yield i;
    }
}

int main() {
    for (int i : work()) {
        std::osyncstream(std::cout) << ch << '\n';
    }
}
```

Open in Compiler Explorer

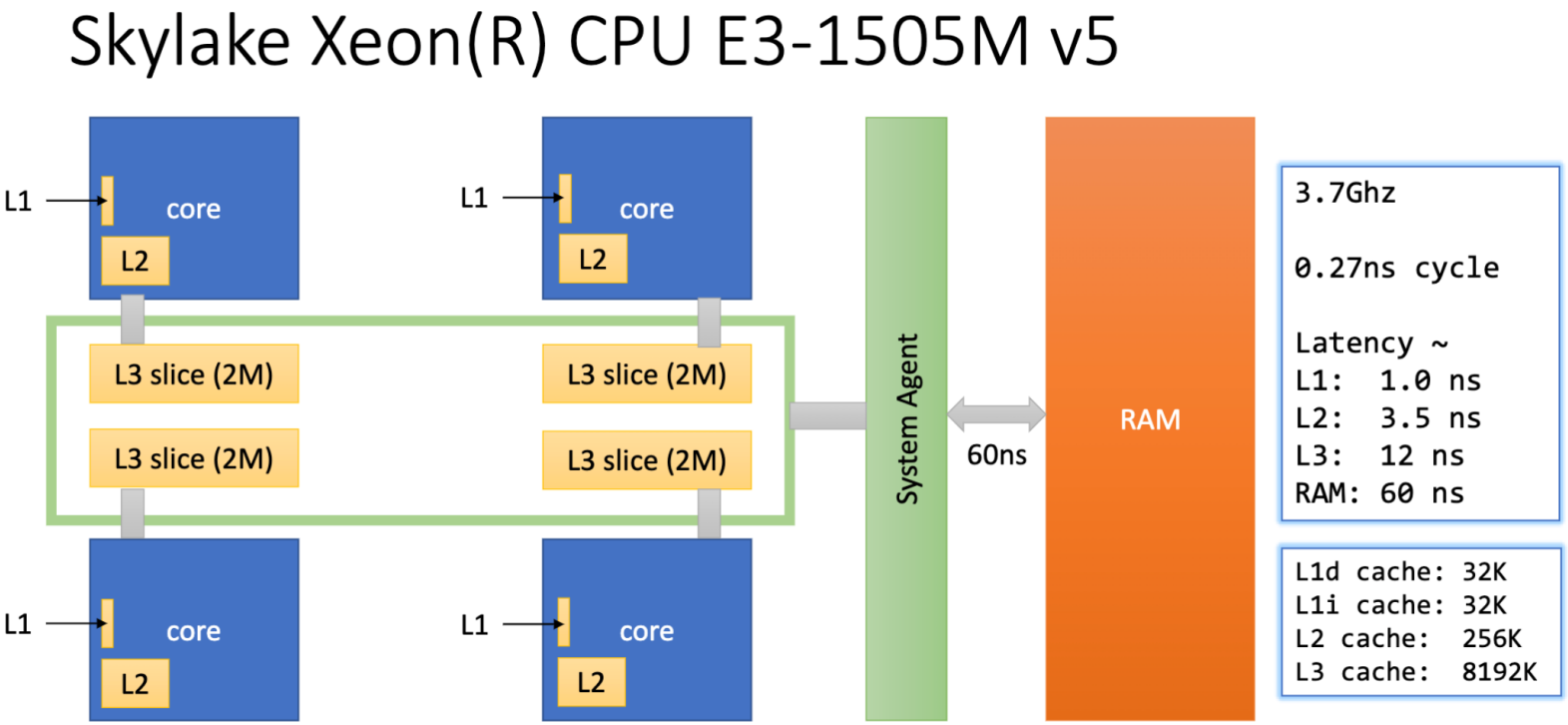
Structuring code

Processes, Threads, Tasks, Coroutines

- A **task** is a rather abstract unit of work, e.g., a function, which can be executed by any thread, but often allocated to one of a many threads within a pool.

Memory order

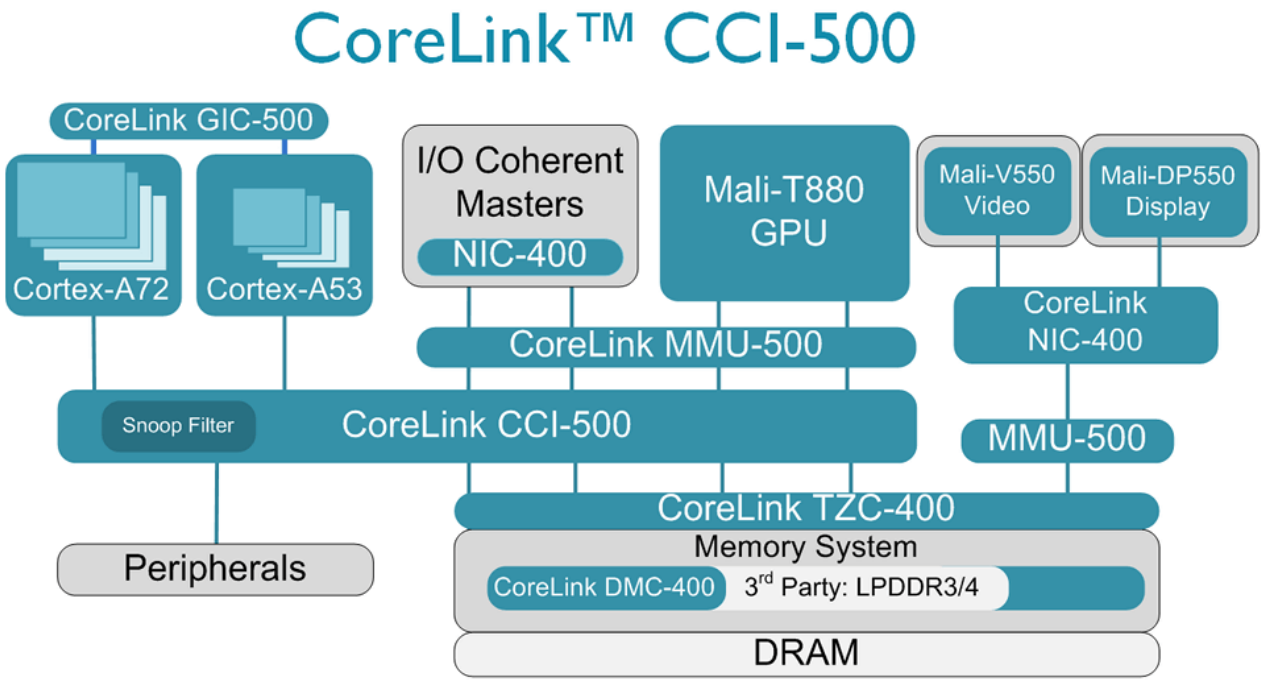
- First, one should like to understand several options for implementing synchronization primitives, known as memory orders. All guarantee atomicity and modification-order consistency.



Obrázek z https://github.com/GorNishanov/await/blob/master/2018_CppCon/NanoCoroutines%20-%20Gor%20Nishanov%20-%20CppCon%202018.pdf

Memory order

- First, one should like to understand several options for implementing synchronization primitives, known as memory orders. All guarantee atomicity and modification-order consistency.



https://developer.arm.com/Processors/CoreLink%20CCI-500

Memory order

- Let us focus on ARM in particular:

  The ARMv8 architecture employs a **weakly-ordered** model of memory. In general terms, this means that
  - the order of memory accesses is not required to be the same as the program order for load and store operations.
  - The processor is able to re-order memory read operations with respect to each other.
  - Writes may also be re-ordered (for example, write combining).

  As a result, hardware optimizations, such as the use of cache and write buffer, function in a way that improves the performance of the processor, which means that the required bandwidth between the processor and external memory can be reduced and the long latencies associated with such external memory accesses are hidden.

  https://developer.arm.com/documentation/den0024/a/Memory-Ordering?lang=en

Memory order

In C++11

- In **memory_order_relaxed**, no further guarantees are provided and specifically no order is imposed on concurrent memory accesses. This is also how weakly-ordered architectures (e.g. ARM) operate, by default: if two threads access shared memory the load in one thread does not have to read a value written by another thread very recently.

- With **memory_order_release** and memory_order_acquire specifiers, we force weakly-ordered achitectures to behave closer to strongly-ordered architectures (e.g., Intel). If one thread writes into shared memory atomically with **memory_order_release** and another thread reads the memory atomically with memory_order_acquire, the load in the second thread is guaranteed to read the value written by another thread.

- With **memory_order_seq_cst**, we additionally require a single total ordering of all modifications (with this specifier). A load with this specifier gets its value either from the last store with this specifier or from some store without this specifier that did not precede the most recent **memory_order_seq_cst store**. This is the default option.

Compare and swap

In General

Synchronization primitives are typically implemented using some hardware instructions, typically **compare-and-swap**. In locking, these make it possible to test whether the lock is free, and if so, acquire the lock within a single operation that the hardware guarantees to execute atomically.

The atomic compare and swap (CAS) instruction compares the value of an atomic variable against a given value. If there is a match, CAS stores a given new value in the atomic variable. That is:

- we declare an atomic variable (and a pointer to it)
- (*) we save the value of an atomic variable to a local, private variable (by dereferencing the pointer)
- based on the saved value in a local, private variable, we compute the new value, which we would like to store in the atomic variable
- the CAS instruction is used. If the current value matches the value saved in the local, private variable, we will overwrite the value with the newly computed value. If the current value no longer matches the value saved in the local, private variable, we wait (some random and growing from a small starting value) and repeat from (*).

Compare and swap

In C++

In C++, the **atomic** header defines two variants of ``compare and swap'' and a specialization thereof for pointers:

- bool compare_exchange_weak(_Tp& __e, _Tp __i, memory_order __s, memory_order __f) noexcept
- bool compare_exchange_strong(_Tp& __e, _Tp __i, memory_order __s, memory_order __f) noexcept

Both are called with the desired value **e**, the new value **i**, and the memory orders to consider if there is a match and if there is no match.

Typically, if there is a match and we want to replace the value, we may use **std::memory_order_release**. If there is no match, we are just reading the value and **std::memory_order_acquire** would suffice. In the latter variant, we pass two pointers.

Compare and Swap

Weak and strong variants

- The difference between the **weak** and **strong** variant is in that the weak variant may return false even if there is a match, in certain cases, but can be much faster in certain architectures. This notably entails ARM architectures (RISC-V and MIPS), where the weak variant will be implemented using the so called load-link/store-conditional pair of instructions (load exclusive register / ldxr and store exclusive register / stxr in ARM version 8). These are much faster than the comparable instructions issuing a barrier (ldaxr/stlxr in ARM version 8).
- All four ARM instructions utilize only two registers, compared to three registers for CAS proper in Intel architectures (Compare and exchange / cmpxchg since 80486 and cmpxchg8b and cmpxchg16b since Intel Core 2). On recent Intel and AMD processors, cmpxchg is only marginally slower than a non-cached load.

Memory order

In C++11

If you want to understand memory orders in more detail:

- See https://arxiv.org/abs/1803.04432

[Submitted on 12 Mar 2018]

Memory Models for C/C++ Programmers

Manuel Pöter, Jesper Larsson Träff

The memory model is the crux of the concurrency semantics of shared-memory systems. It defines the possible values that a read operation is allowed to return for any given set of write operations performed by a concurrent program, thereby defining the basic semantics of shared variables. It is therefore impossible to meaningfully reason about a program or any part of the programming language implementation without an unambiguous memory model.
This note provides a brief introduction into the topic of memory models, explaining why it is essential for concurrent programs and covering well known memory models from sequential consistency to those of the x86 and ARM/POWER CPUs. Section 4 is fully dedicated to the C++11 memory model, explaining how it can be used to write concurrent code that is not only correct and portable, but also efficient by utilizing the relaxed memory models of modern architectures.

- See also: https://www.youtube.com/watch?v=A_vAG6LIHwQ&ab_channel=ACCUConference

Synchronization primitives

- Synchronization primitives make it possible to synchronize or restrict access of multiple threads to some resources (e.g., global variables, file handles, sockets). You can use them as an interface, without knowing their implementation.
- **Raw synchronization primitives:** Lock, Mutex, Semaphore, Atomic, Memory Fence, Condition Variable are synchronization primitives, which make it possible to synchronize or restrict access of multiple threads to some resources.
- Lock is a very general term for a synchronization primitive. Mutexes are usually used by one thread only, while semaphores are shared between multiple threads.
- The binary semaphore is the most simple type of a lock, which provides exclusive access for both reading and writing.
- The counting semaphore limits the use of a single resource by at most a given number of threads.
- A spinlock, the thread simply waits ("spins") until the lock becomes available. This is efficient if threads are blocked for a short time, because it avoids the overhead of operating system process re-scheduling. It is inefficient if the lock is held for a long time, or if the progress of the thread that is holding the lock depends on preemption of the locked thread.

Synchronization primitives

In C++

- In C++, the **only** synchronization primitive that is guaranteed to be hardware implemented is a particular atomic boolean type, which is known as **std::atomic_flag**.
- Unlike all specializations of **std::atomic**, it is guaranteed to be lock-free.
- Prior to C++20, it has been very restricted, because there was no way to check the value of **std::atomic_flag** without setting it. C++20 adds method **test()**.

Synchronization primitives

And how to implement them

A silly implementation of a spin lock.

```
 1  // based on https://en.cppreference.com/w/cpp/atomic/atomic_flag
 2
 3  class SpinLock {
 4      std::atomic_flag locked = ATOMIC_FLAG_INIT ;
 5  public:
 6      void lock() {
 7          while (locked.test_and_set(std::memory_order_acquire)) {
 8          #if defined(__cpp_lib_atomic_flag_test)
 9              while (locked.test(std::memory_order_relaxed))
10          #endif
11              ;
12   }
13      }
14      void unlock() {
15          locked.clear(std::memory_order_release);
16      }
17  };
```

Open in Compiler Explorer

Further features

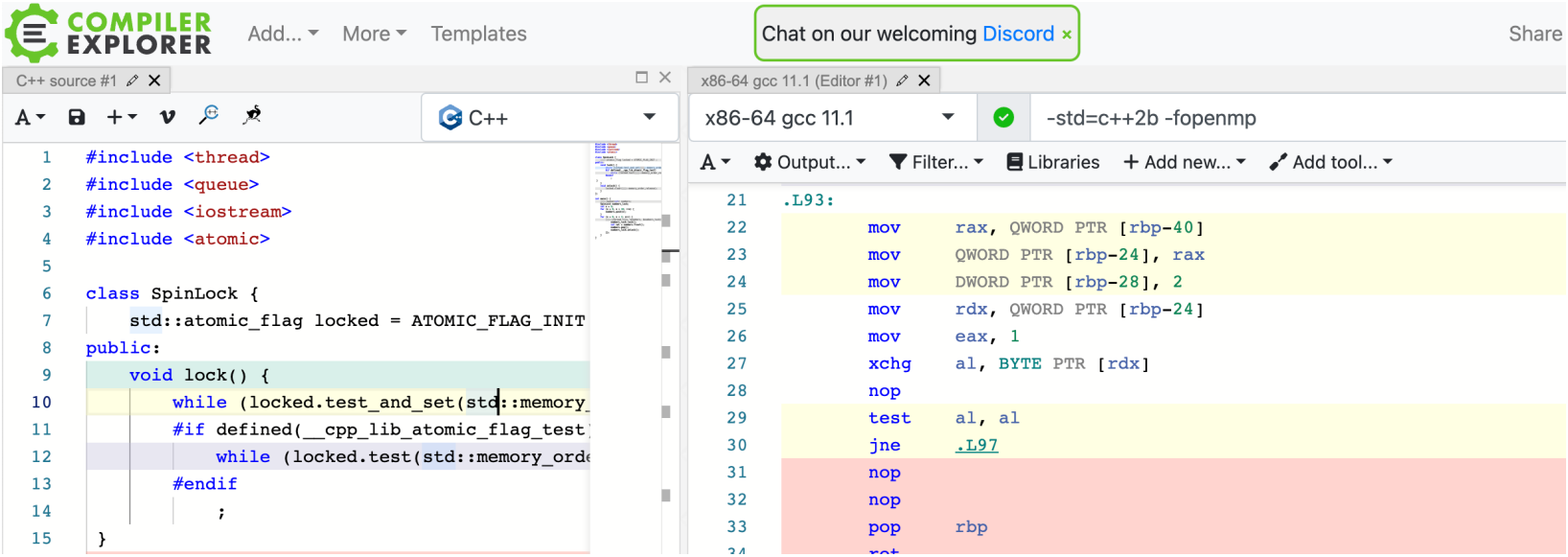
In C++23

Further synchronization features

- Fences help order non-atomic and atomic memory accesses, without any associated operations. On Intel architectures (including x86-64), **atomic_thread_fence** do not issue any instructions, except **std::atomic_thread_fence(std::memory_order::seq_cst).**
- Barrier provides a thread-coordination mechanism that blocks a group of threads until all threads in that group have reached the barrier. Such a barrier can be used repeatedly to wait until a number of threads have finished their operations.
- Latch and is a downward counter, whose initial value is initialized and then threads may block on the latch until the counter is zero. One thread may decrement a latch multiple times, but no thread can increment the latch. Thus, it serves as a single-use barrier.
- We will also see synchronized output streams. The synchronized buffer is flushed only when the destructor of the synchronized buffer is called, but provides for guarantees of atomicity for the access. (That is, **std::endl** and **std::flush** no longer flush!)

Debugging

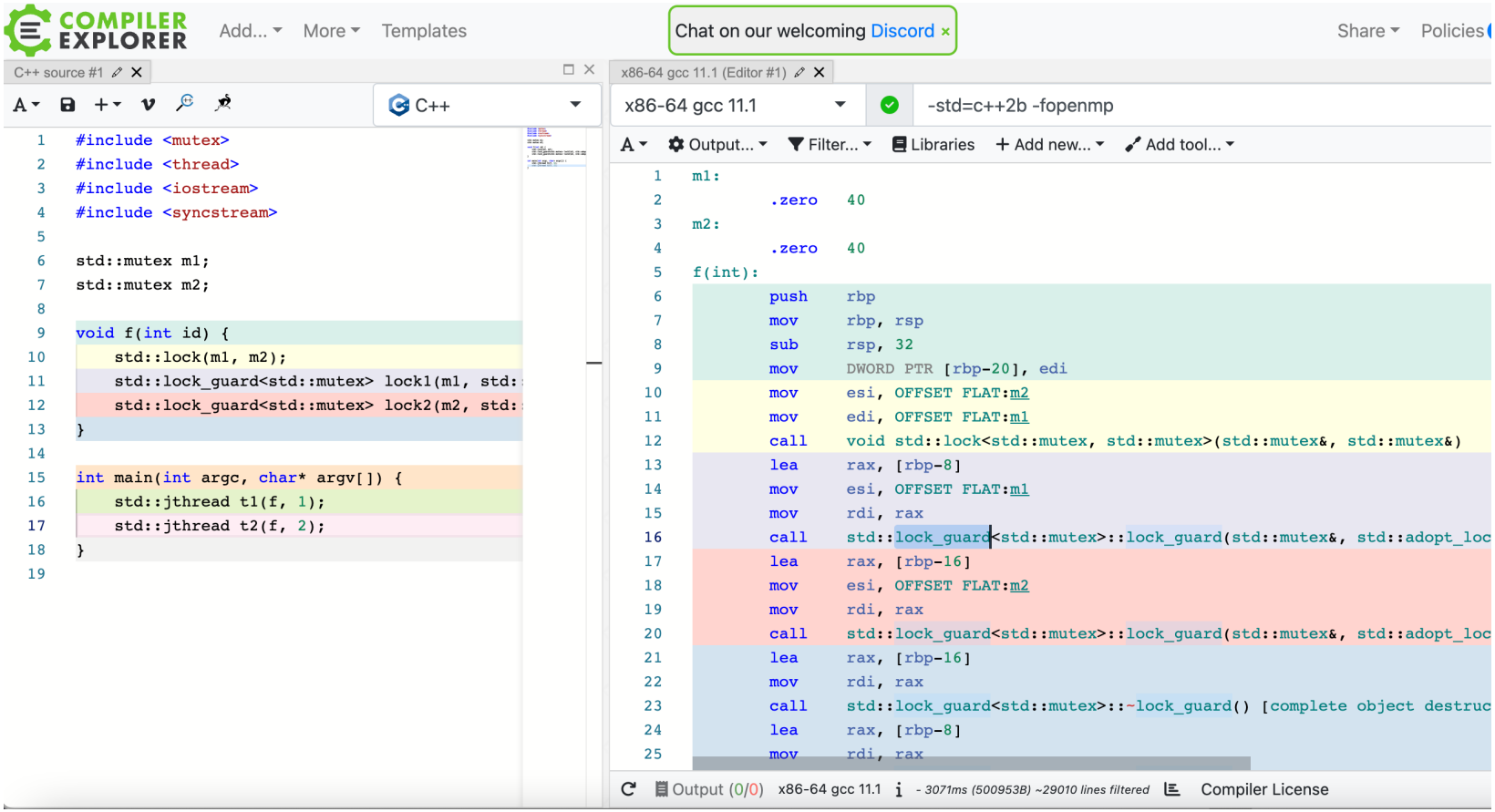
https://godbolt.org/



https://godbolt.org/z/cEdE7r5fq

Debugging

https://godbolt.org/



Debugging

https://godbolt.org/

Debugging

https://clang.llvm.org/docs/ThreadSanitizer.html

- https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual

```
$ clang++ simple_race.cc -fsanitize=thread -fPIE -pie -g
$ ./a.out
==================
WARNING: ThreadSanitizer: data race (pid=26327)
  Write of size 4 at 0x7f89554701d0 by thread T1:
    #0 Thread1(void*) simple_race.cc:8 (exe+0x000000006e66)

  Previous write of size 4 at 0x7f89554701d0 by thread T2:
    #0 Thread2(void*) simple_race.cc:13 (exe+0x000000006ed6)

  Thread T1 (tid=26328, running) created at:
    #0 pthread_create tsan_interceptors.cc:683 (exe+0x00000001108b)
    #1 main simple_race.cc:19 (exe+0x000000006f39)

  Thread T2 (tid=26329, running) created at:
    #0 pthread_create tsan_interceptors.cc:683 (exe+0x00000001108b)
    #1 main simple_race.cc:20 (exe+0x000000006f63)
==================
ThreadSanitizer: reported 1 warnings
```

What comes next?

Odpovídající státnicové otázky

Paralelní část

Hardwarová podpora pro paralelní výpočty: (super)skalární architektury, pipelining, spekulativní vyhodnocování, vektorové instrukce, vlákna, procesy, GPGPU. Hierarchie cache pamětí.

Komplikace v paralelním programování: souběh (race condition), uváznutí (deadlock), iluze sdílení (false sharing).

Podpora paralelního programování v C a C++: pthreads, thread, jthread, atomic, mutex, lock_guard.

Podpora paralelního programování v OpenMP: sériově-paralelní model uspořádání vláken (fork-join), paralelizovatelná úloha (task region), různé implementace specifikace. Direktivy parallel, for, section, task, barrier, critical, atomic.

Techniky dekompozice programu: statické a paralelní rozdělení práce. Threadpool a fronta úkolů. Balancování a závislosti (dependencies).

Techniky dekompozice programu na příkladech z řazení: quick sort, merge sort.

Techniky dekompozice programu na příkladech z numerické lineární algebry a strojového učení: násobení matice vektorem, násobení dvou matic, řešení systému lineárních rovnic.

Odpovídající státnicové otázky

Distribuovaná část

Úvod do distribuovaných systémů (DS). Charakteristiky DS. Čas a typy selhání v DS.

Detekce selhání v DS. Detektory selhání a jejich vlastnosti.

Čas a kauzalita v DS. Uspořádání událostí v DS. Fyzické hodiny a jejich synchronizace. Logické hodiny a jejich synchronizace.

Globální stav v DS a jeho výpočet. Řez distribuovaného výpočtu. Algoritmus pro distribuovaný globální snapshot. Stabilní vlastnosti DS.

Vzájemné vyloučení procesů v DS. Algoritmy pro vyloučení procesů a jejich vlastnosti.

Volba lídra v DS. Algoritmy pro volbu lídra a jejich vlastnosti.

Konsensus v DS. FLP teorém. Algoritmy pro distribuovaný konsensus.