CTU
CZECH TECHNICAL UNIVERSITY IN PRAGUE

AI CENTER

Paralelní a distribuované výpočty (B4B36PDV)

Jakub Mareček, Michal Jakob

jakub.marecek@fel.cvut.cz

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

Paralelní a distribuované výpočty

Paralelní programování

- 1 stroj, vícero jader, hierarchie vyrovnávacích pamětí, sdílená paměť,
- 1 program, použití synchronizačních primitiv, datové struktury
- OpenMP, C++23, SYCL, Go, Rust

Rychleji nalézt řešení

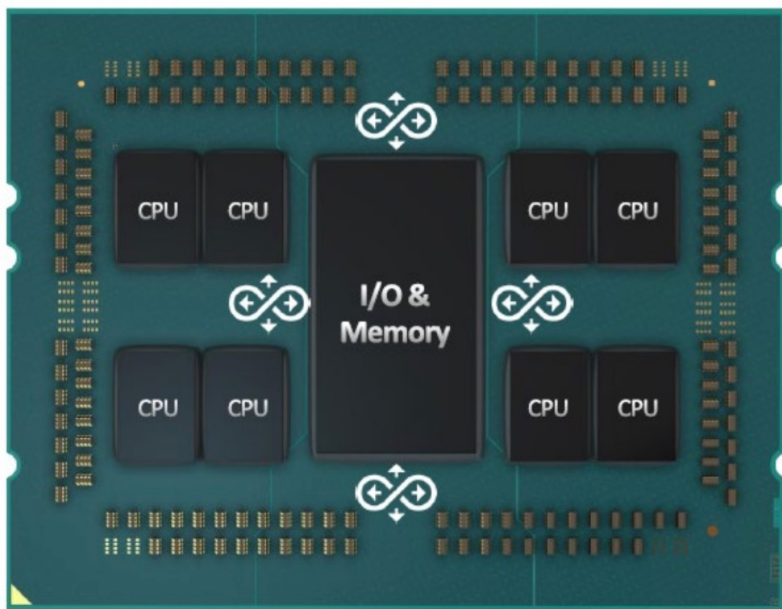
Programování v distribuovaných systémech

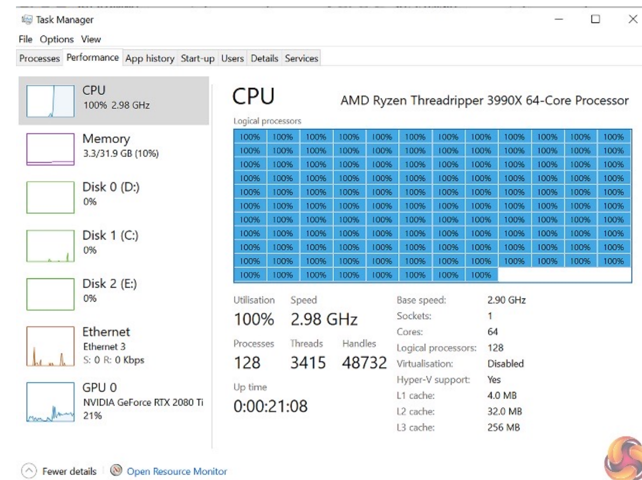
- vícero strojů komunikujících po síťových rozhraních (např. InfiniBand)
- distribuovaná data
- komunikační složitost, „HW na zakázku"
- MPI, k8s, Spark

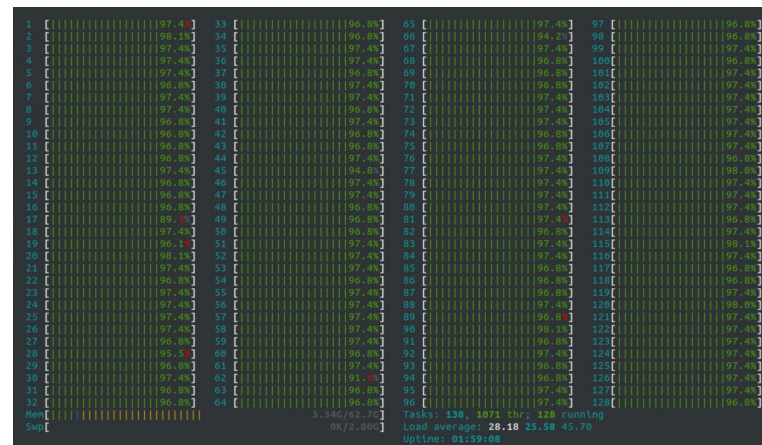
Zajistit konsistenci dat

Motivace

Co je to moderní procesor?







Obrázky převzaty z:

- https://www.tomshardware.com/news/amd-zen-3-ryzen-5000-release-date-specifications-pricing-benchmarks-all-we-know
- https://www.amd.com/system/files/documents/tr-pro-thought-leadership.pdf
- https://techgage.com/article/amd-ryzen-threadripper-3990x-64-core-linux-performance/

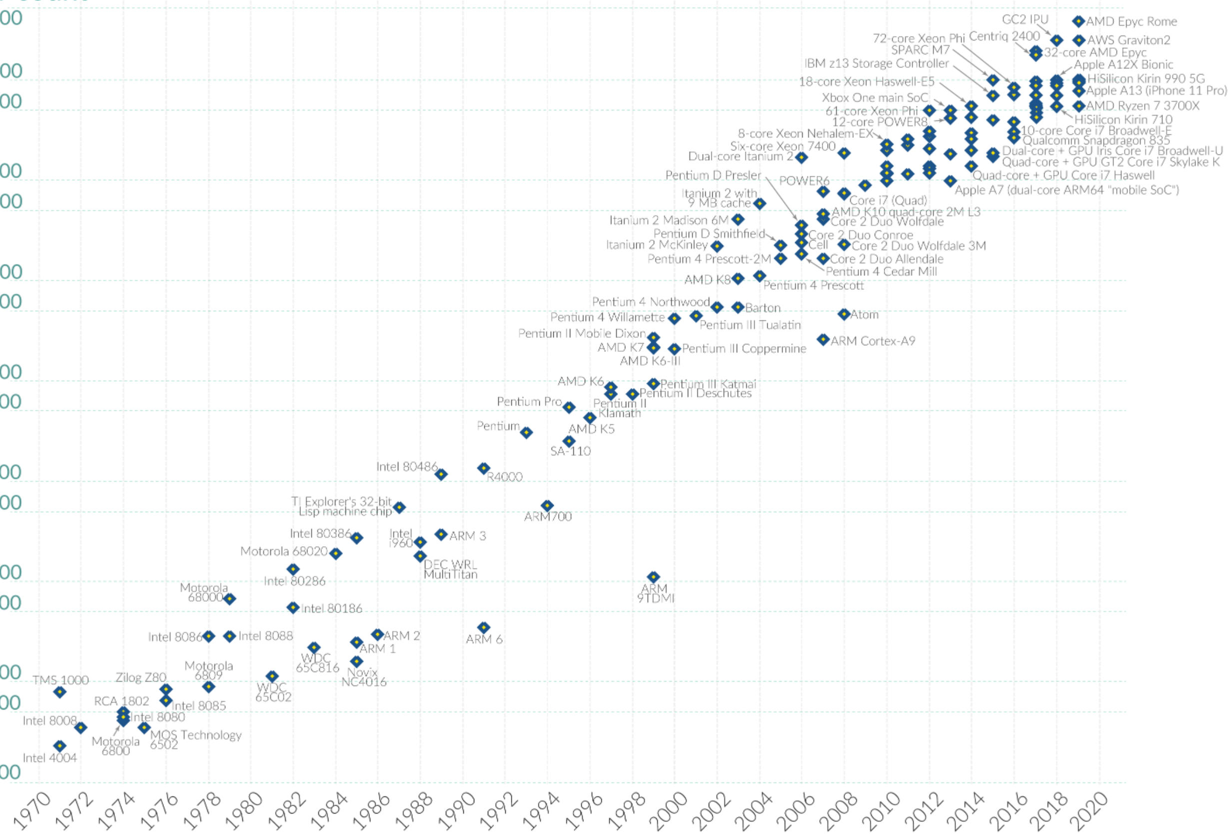
Motivace:

Nárůst počtu transistorů

Moore's Law: The number of transistors on microchips doubles every two years

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.

Our World in Data



Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)

OurWorldinData.org – Research and data to make progress against the world's largest problems.

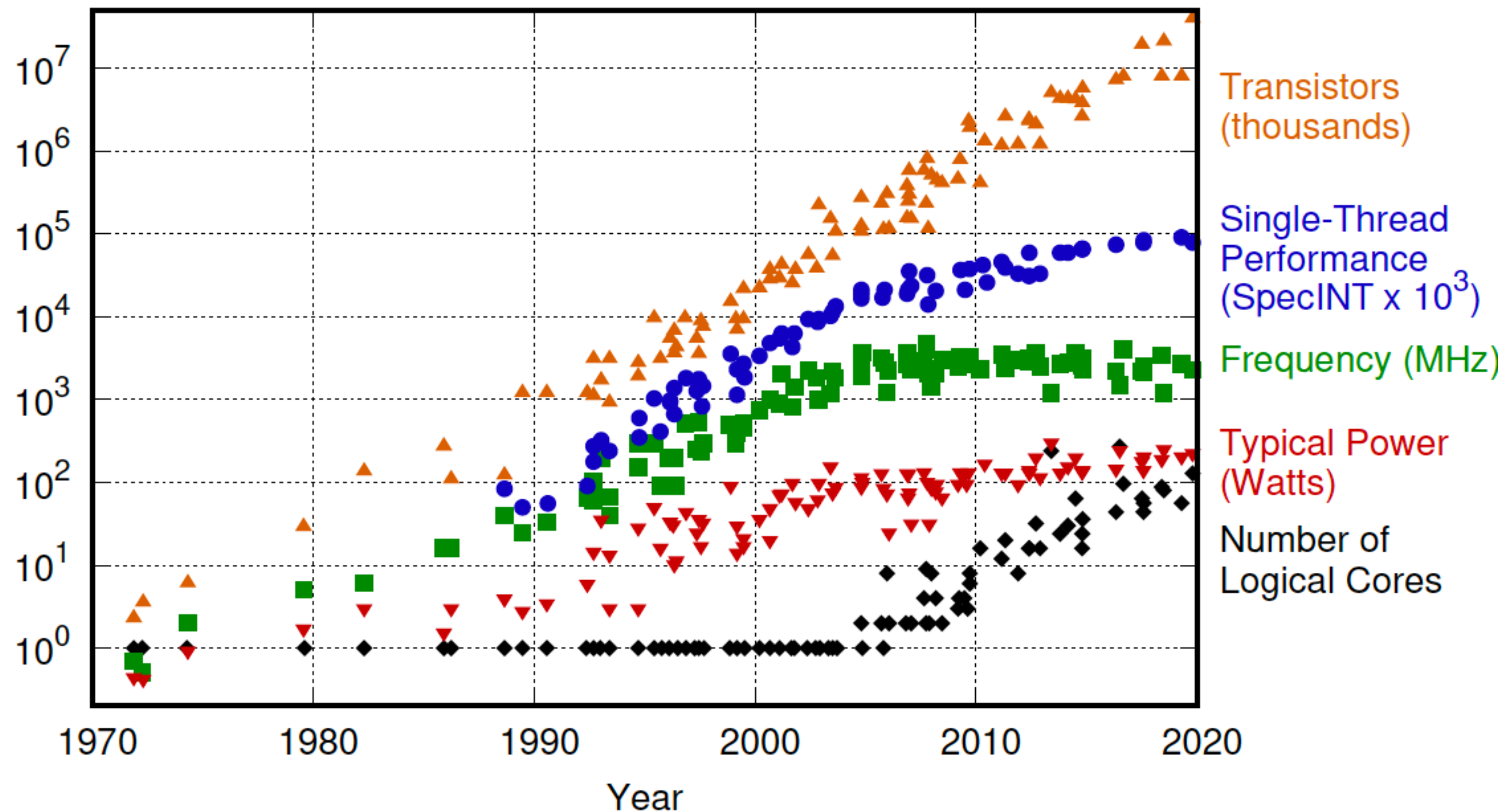
Licensed under CC-BY by the authors Hannah Ritchie and Max Roser.

Obrázky převzaty z:

- https://en.wikipedia.org/wiki/Moore%27s_law

Motivace

Nárůst výkonu jednotlivého jádra; počtu jader



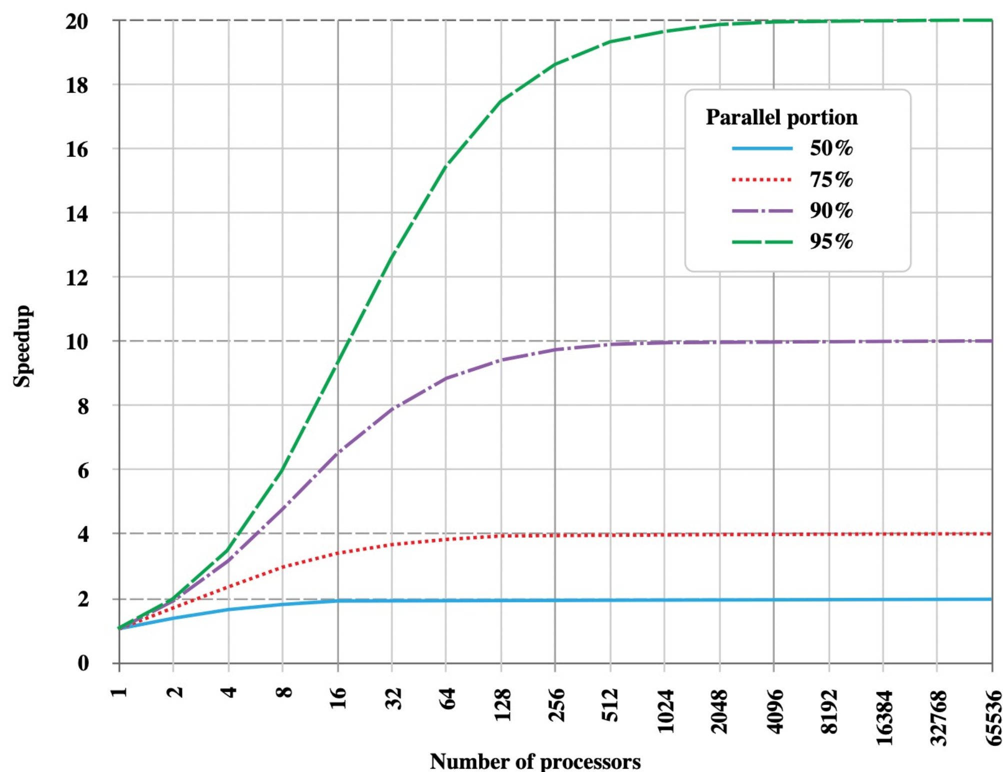
Data viz:

- https://zenodo.org/record/3947824#.YCBhyhNKhpI

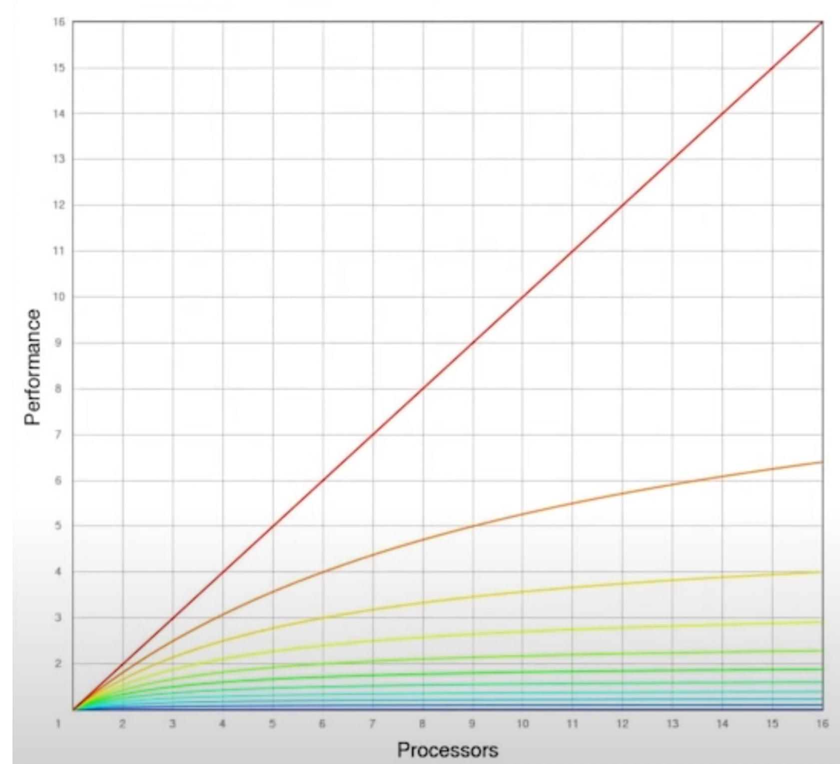
Motivace

Amdahlův zákon: Zrychlení je omezené neparalelizovatelným kódem

Log-lineární graf



Lineární graf



Grafy z:

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

Motivace

Neparalelizovatelný kód je dán jazykem a knihovnou, kterou užíváte

AMD Threadripper 3990X + NVIDIA GeForce RTX 4090 Ti

- Jednovláknová aplikace: 49 GFLOPS (Geekbench 4)
  - Tj. limit pro použití běžného C++ je pod 0.05 % celkového výkonu
- Vícevláknová aplikace: 3732 GFLOPS (Geekbench 4)
  - Tj. limit pro použití pthreads, C++23, nebo OpenMP do verze 4.5
    je pod 4% celkového výkonu
- Vícevláknová aplikace s GPGPU: přes 100 TFLOPS (AIDA, single-precision, overclocking)
  - Celkový výkon je dostupný při použití OpenMP od verze 4.5, SYCL, a dalších (CUDA, OpenGL, DirectX Compute)

Pthreads vs. C++ vs. OpenMP vs. SYCL

Co znáte z OSY (pthreads)

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

const int thread_count = 10;
void* Hello(void* rank);

int main(int argc, char* argv[]) {
    long thread;
    pthread_t *thread_handles;
    thread_handles = (pthread_t*)malloc(thread_count * sizeof(pthread_t));
    for (thread = 0; thread < thread_count; thread++)
        pthread_create(&thread_handles[thread], NULL,
                       Hello, (void *) thread);
    printf("Hello from the main thread\n");
    for (thread = 0; thread < thread_count; thread++)
        pthread_join(thread_handles[thread], NULL);

    free(thread_handles);
    return 0;
}

void* Hello(void* rank) {
    long my_rank = (long) rank;
    printf("Hello from thread %ld of %d\n", my_rank, thread_count);
    return NULL;
}
```

Pthreads vs. C++ vs. OpenMP vs. SYCL

Ochutnávka (OpenMP)

```
#include <iostream>
#include <vector>
#include "omp.h"

const int thread_count = 10;

void Hello() {
    int my_rank = omp_get_thread_num();
    int threads = omp_get_num_threads();
    std::cout << "Hello from thread " << my_rank << " of " << threads << std::endl;
}

int main(int argc, char* argv[]) {
#pragma omp parallel num_threads(thread_count)
    Hello();
    return 0;
}
```

- nutno překládat s přepínačem –fopenmp
  - (např. **g++ -fopenmp openmp-hello.cpp -o openmp-hello**)

Pthreads vs. C++ vs. OpenMP vs. SYCL

Ochutnávka (C++11)

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

const int thread_count = 10;
void Hello(long my_rank);

int main(int argc, char* argv[]) {
   std::vector<std::thread> threads;
   for (int thread=0; thread < thread_count; thread++) {
      threads.push_back(std::thread(Hello, thread));
   }

   std::cout << "Hello from the main thread\n";

   for (int thread=0; thread < thread_count; thread++) {
      threads[thread].join();
   }

   return 0;
}

void Hello(long my_rank) {
   std::cout << "Hello from thread " << my_rank << " of " << thread_count << std::endl;
}
```

Nicolai Josuttis: “it is almost impossible to use it easily and right”

Pthreads vs. C++ vs. OpenMP vs. SYCL

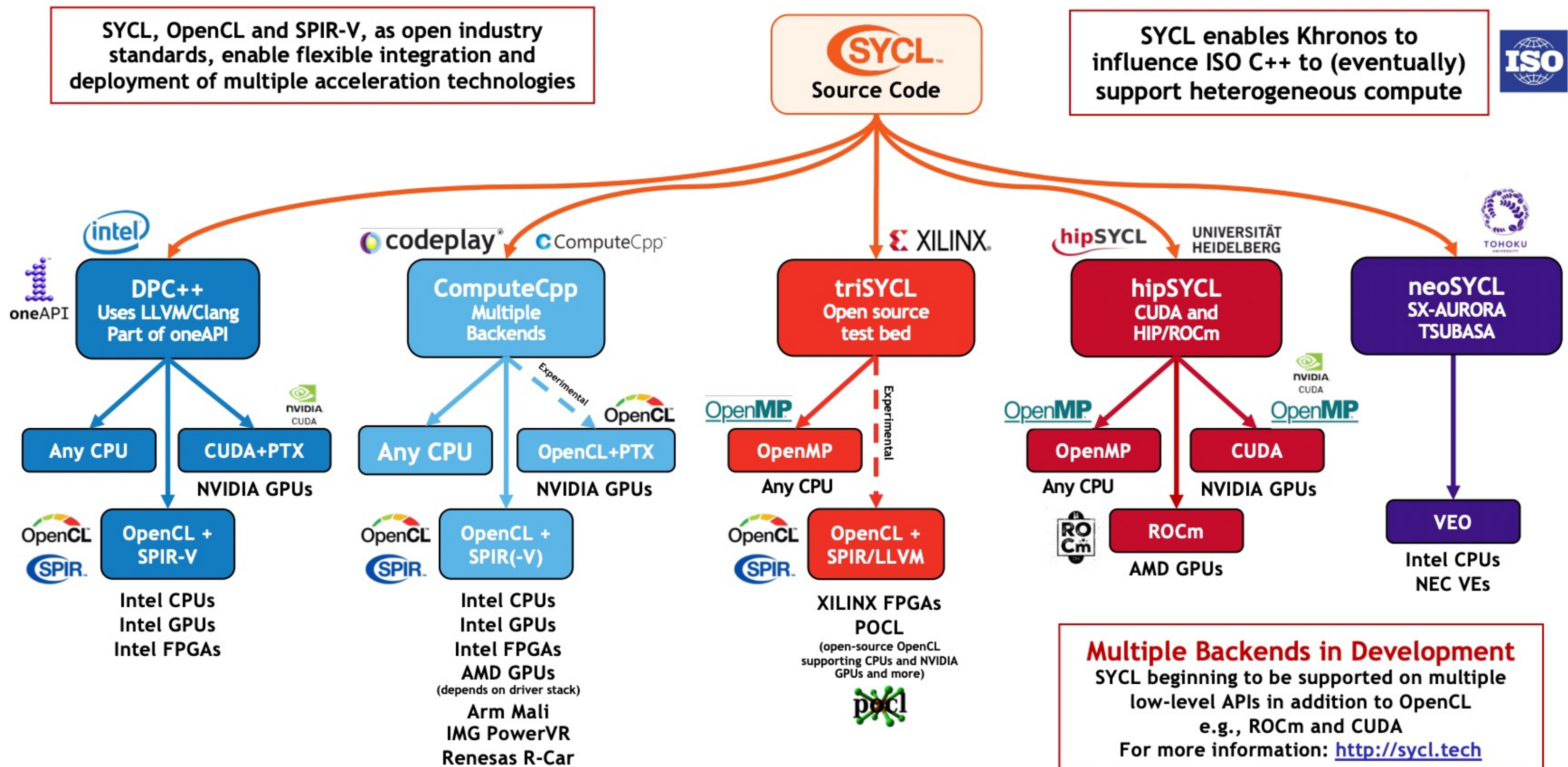
Ochutnávka (C++20)

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

const int thread_count = 10;
void Hello(long my_rank);

int main(int argc, char* argv[]) {
    std::vector<std::thread> threads;
    for (int thread=0; thread < thread_count; thread++) {
        threads.push_back(std::jthread(Hello, thread));
    }

    std::cout << "Hello from the main thread\n";

    return 0;
}

void Hello(long my_rank) {
    std::cout << "Hello from thread " << my_rank << " of " << thread_count << std::endl;
}
```

Specifikace byla schválena v listopadu 2020. GCC 11 má dobrou podporu.

Pthreads vs. C++ vs. OpenMP vs. SYCL

Ochutnávka (SYCL)



Viz https://www.khronos.org/assets/uploads/developers/presentations/SYCL-2020-Launch-Feb21.pdf

Pthreads vs. C++ vs. OpenMP vs. SYCL

Ochutnávka (SYCL)

```
#include <CL/sycl.hpp>
#include <iostream>

using namespace cl::sycl;
const int nElems = 64u;

class assign_elements;

int main() {
    int data[nElems] = {0};
    try {
        default_selector selector;
        queue myQueue(selector, [](exception_list l) {
            for (auto ep : l) {
                try {
                    std::rethrow_exception(ep);
                } catch (const exception& e) {
                    std::cout << "Asynchronous exception caught:\n" << e.what();
                }
            }
        });
        buffer<int, 1> buf(data, range<1>(nElems));
        myQueue.submit([&](handler& cgh) {
            auto ptr = buf.get_access<access::mode::read_write>(cgh);
            auto myRange = nd_range<1>(range<1>(nElems), range<1>(nElems / 4));
           auto myKernel = ([=](nd_item<1> item) {
                ptr[item.get_global_id()] = item.get_global_id()[0];
            });
            cgh.parallel_for<assign_elements>(myRange, myKernel);
        });
    } catch (const exception& e) {
        std::cout << "Synchronous exception caught:\n" << e.what();
        return 2;
    }
    return 0;
}
```

Motivace

Máme cloud, ne?

aws

Products Solutions Pricing Documentation Learn Partner Network AWS Marketplace

AWS HPC Overview Solution Components Getting Started Resources

AWS ParallelCluster

Quickly build HPC compute environments on AWS

Google Cloud Why Google Solutions Products Pricing Getting Started Docs Support Langu

Solutions Contact Us

Solutions
Overview
Media
Gaming
Financial services
Internet of Things
Backup & archive
Digital marketing
Retail & commerc
Web hosting & we
Big data analytics
Development & te
High performanc
Overview
Articles

MPI libraries and user applications

Microsoft | Docs Documentation Learn Q&A Code Samples Search

Azure Product documentation Architecture Learn Azure Develop Resources

Azure / Virtual Machines / Workloads / High Performance Computing Bookmark Feed

Filter by title
Overview
Configuration
Configure and optimize VMs
Enable InfiniBand
Set up MPI
Scaling applications
Virtual machines

Set up Message Passing Interface for HPC

08/06/2020 • 5 minutes to read • +1

The Message Passing Interface (MPI) is an open library and de-facto standard for distributed memory parallelization. It is commonly used across many HPC workloads. HPC workloads on the RDMA capable H-series and N-series VMs can use MPI to communicate over the low latency and high bandwidth InfiniBand network.

The SR-IOV enabled VM sizes on Azure (HBv2, HB, HC, NCv3, NDv2) allow almost any flavor of MPI to be used with Mellanox OFED. On non-SR-IOV enabled VMs, supported MPI implementations use the Microsoft Network Direct (ND) interface to communicate between VMs. Hence, only Microsoft MPI (MS-MPI) 2012 R2 or later and Intel MPI 5.x versions are supported. Later versions (2017, 2018) of the Intel MPI runtime library may or may not be compatible with the Azure RDMA drivers.

For SR-IOV enabled RDMA capable VMs, CentOS-HPC version 7.6 or a later version VM images in the Marketplace are optimized and pre-loaded with the OFED drivers for RDMA and various commonly used MPI libraries and scientific computing packages and are the easiest way to get started.

best

kubernetes Documentation Kubernetes Blog Training Partners

Search
Home
Getting started
Concepts
Tasks
Install Tools
Administer a Cluster
Configure Pods and Containers
Manage Kubernetes Objects
Managing Secrets
Inject Data Into Applications

How does the Horizontal Pod Autoscaler work?

Pod 1
Pod 2
. . .
Pod N
RC / Deployment
Scale
Horizontal Pod Autoscaler

APACHE Spark Lightning-fast unified analytics engine

Download Libraries Documentation Examples Community Developers

Apache Spark™ is a unified analytics engine for large-scale data processing.

Speed

Run workloads 100x faster.

Apache Spark achieves high performance for both batch and streaming data, using a state-of-the-art DAG scheduler, a query optimizer, and a physical execution engine.

Running time (s)
120
90
60
30
0
110
0.9
Hadoop
Spark

Logistic regression in Hadoop and Spark

Více viz:

- https://docs.aws.amazon.com/parallelcluster/latest/ug/tutorials_03_batch_mpi.html
- https://cloud.google.com/solutions/best-practices-for-using-mpi-on-compute-engine
- https://docs.microsoft.com/en-us/azure/virtual-machines/workloads/hpc/setup-mpi
- https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
- https://spark.apache.org/

Motivace

Máme cloud, ne?

Oracle Real Application Clusters features

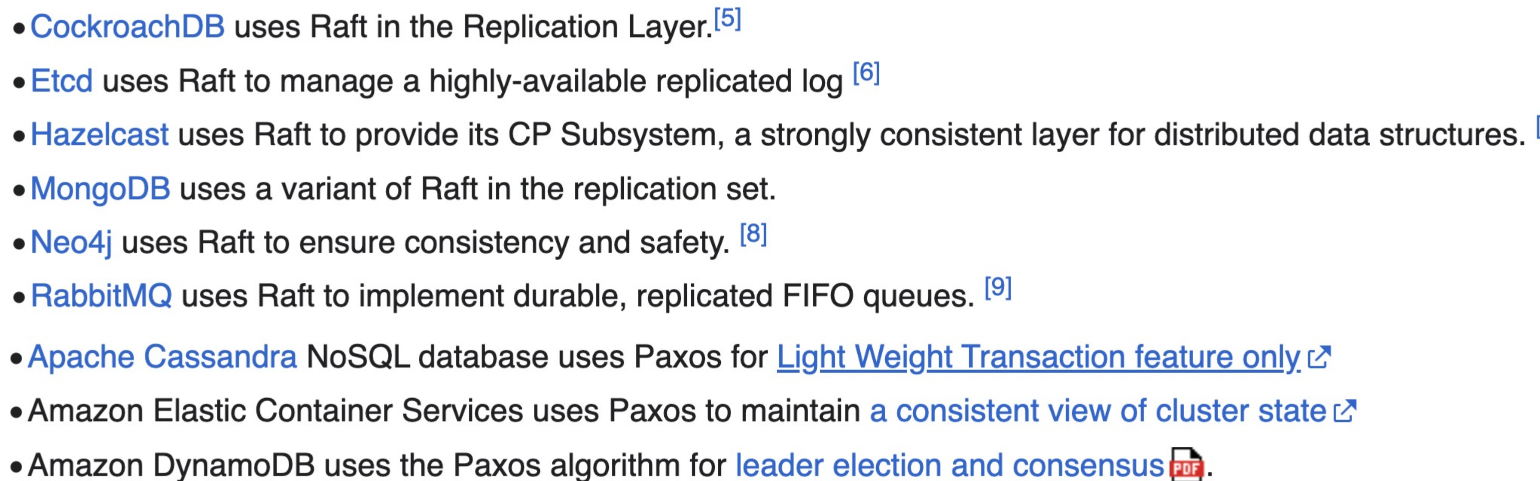
- Key capabilities
- Flexibility
- **Scalability**
- Availability

Scalability

Scale features and functions
Rapidly scale all Oracle Database features and functions across multiple instances to address sudden changes in customer workloads, without the need for manual intervention.

No application changes
Scale applications up to one hundred nodes in a cluster without application changes, improving DBA and developer productivity.

More scalable analytics
Oracle RAC allows customers to easily scale analytics workloads by eliminating the need for replica databases and database level conflict resolution, significantly improving database administrator productivity.



Production use of Raft [edit]

- CockroachDB uses Raft in the Replication Layer.[5]
- Etcd uses Raft to manage a highly-available replicated log [6]
- Hazelcast uses Raft to provide its CP Subsystem, a strongly consistent layer for distributed data structures. [
- MongoDB uses a variant of Raft in the replication set.
- Neo4j uses Raft to ensure consistency and safety. [8]
- RabbitMQ uses Raft to implement durable, replicated FIFO queues. [9]

- Apache Cassandra NoSQL database uses Paxos for Light Weight Transaction feature only
- Amazon Elastic Container Services uses Paxos to maintain a consistent view of cluster state
- Amazon DynamoDB uses the Paxos algorithm for leader election and consensus.

Více viz:
- https://www.oracle.com/database/real-application-clusters/#rc30p3
- https://en.wikipedia.org/wiki/Raft_(algorithm)
  https://en.wikipedia.org/wiki/Paxos_(computer_science)

Organizační

Kdo jsme

Přednášející

Jakub Mareček

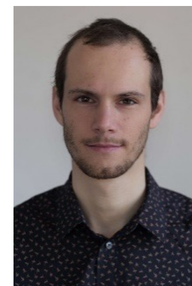
Michal Jakob

Cvičící

Peter Macejko

David Fiedler

Jan Mrkos



David Milec

Matěj Kafka

Michal Slouka

Organizační

Kde a kdy jsme

B4B36PDV - Paralelní a distribuované výpočty (2P+2C) - 1.-8. týden

hodina	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
čas	07:30	08:15	09:15	10:00	11:00	11:45	12:45	13:30	14:30	15:15	16:15	17:00	18:00	18:45	19:45	20:30
pondělí																
úterý																
středa							**KN:E-107** Mareček J., Jakob M. 1 (119 stud.)				**KN:E-311** Kafka M. 101 (20 stud.)		**KN:E-311** Kafka M. 102 (20 stud.)			
čtvrtek			**KN:E-311** Macejko P. 105 (20 stud.)		**KN:E-311** Macejko P. 106 (20 stud.)						**KN:E-310** Fiedler D. 104 (20 stud.)		**KN:E-310** Slouka M. 107 (19 stud.)			
pátek																

Přednášky	**Cvičení**	**Laboratoře**	**Ostatní**

pondělí																
úterý																
středa							**KN:E-107** Mareček J., Jakob M. 1 (119 stud.)				**KN:E-311** Milec D. 101 (20 stud.)		**KN:E-311** Mrkos J. 102 (20 stud.)			
čtvrtek			**KN:E-311** Macejko P. 105 (20 stud.)		**KN:E-311** Macejko P. 106 (20 stud.)						**KN:E-310** Mrkos J. 104 (20 stud.)		**KN:E-310** Milec D. 107 (19 stud.)			
pátek																

Přednášky	**Cvičení**	**Laboratoře**	**Ostatní**

Organizační

Co chceme

- Paralelní část
  - Motivace: programování her, AI
  - Získat základní informace a prostor pro praktické zkušenosti v oblasti programování efektivních paralelních programů
  - Paralelní programování jednoduchých algoritmů a vliv různých způsobů paralelizace na rychlost výpočtu
- Distribuovaná část
  - Motivace: Oracle Real Application Cluster
  - Konzistence a shoda v distribuovaných systémech

Přehled paralelní části

- Základní úvod
  - Souběh (race condition), uváznutí (deadlock), iluze sdílení (false sharing).
  - Hardwarová vlákna, softwarová vlákna, synchronizační primitiva
- C++: C++20 jthready, C++23 coroutines
- OpenMP
  - Specifikace nadstavby nad kompilátorem (např. C, C++, Fortran). Kdysi hardwarová vlákna (např. pthreads), od verze 5.0 softwarová vlákna (tasks) a podpora GPGPU.
- SYCL
  - Specifikace nádstavby nad kompilátorem (např. C++) a knihovny pro zjednodušení implementace paralelních programů na CPU i GPGPU.

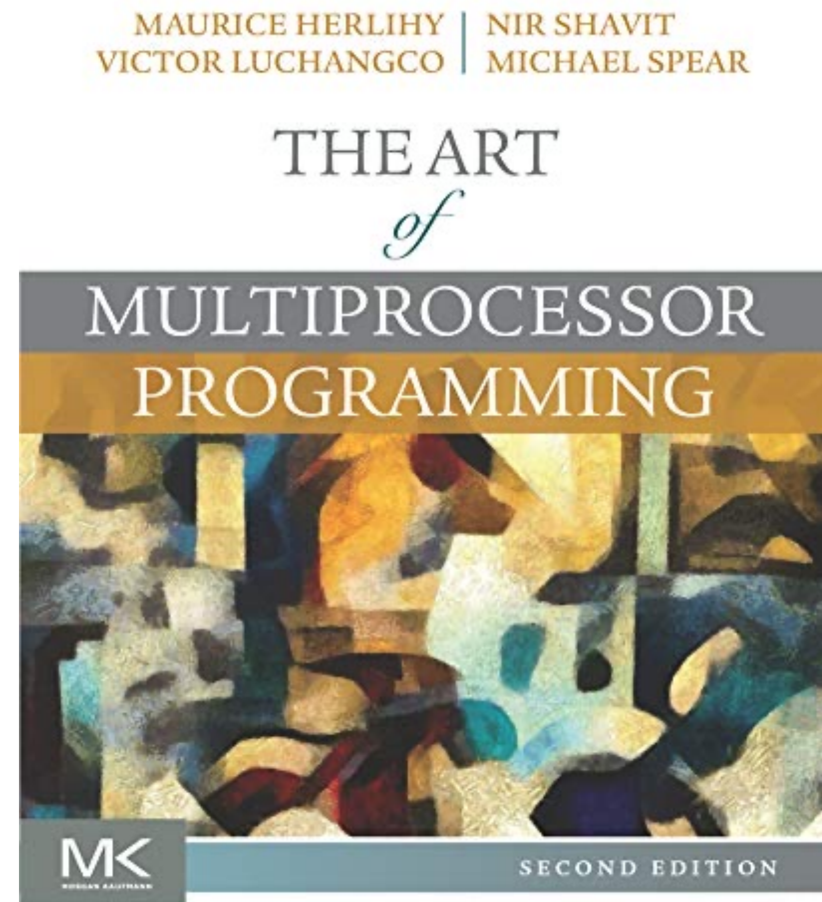
Přehled paralelní části

- Techniky dekompozice
- Datové struktury umožňující přístup vícero vláken
- Základní paralelní řadící algoritmy a vektorové instrukce
- Základní paralelní maticové algoritmy

Materiály k paralelní části

Principy

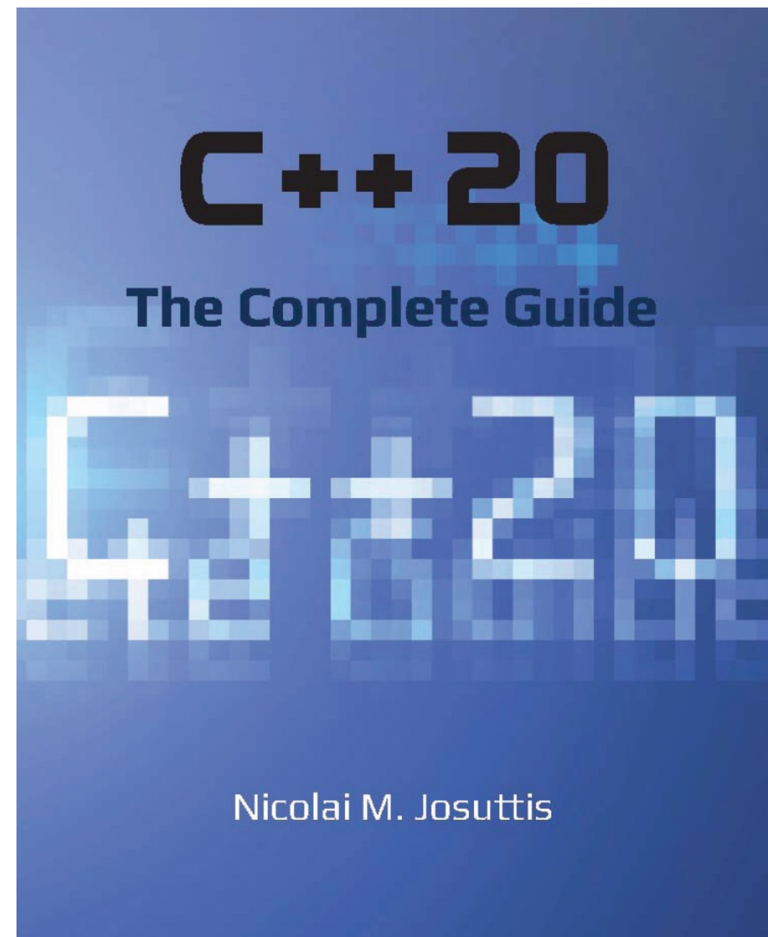
- Standardní učebnice: The Art of Multiprocessor Programming (by Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear).
- Druhé vydání (září 2020).
  Kapitola 1: Úvod
  Kapitola 2: Vzájemné vyloučení
  Kapitola 9: Seznamy
  Kapitola 12: Řazení
  Kapitola 16: Rozvrhování
- Viz také https://www.youtube.com/watch?v=nrUszqrlvi8.



Materiály k paralelní části

C++23

- Standardní učebnice:
  C++20 - The Complete Guide
  (by Nicolai M. Josuttis).
- První vydání (listopad 2022).
  http://cppstd20.com/
- Kapitoly:
  - 12: jthread
  - 13: Concurrency
  - 14: Coroutines
- Viz také
  https://en.cppreference.com/w/cpp/thread
  (povoleno na praktické zkoušce)



Materiály k paralelní části

OpenMP

- Standardní dokumentace OpenMP:
  https://www.openmp.org/resources/refguides/
  je povolena na praktické zkoušce. Nikoli příklady
  https://github.com/OpenMP/Examples/
- Velmi praktické rady: Using OpenMP (Portable Shared Memory Parallel Programming, by Barbara Chapman, Gabriele Jost and Ruud van der Pas). Vydání z roku 2007 je dostupné přes NTK.
- Neformální úvod: Programming on Parallel Machines (by Norm Matloff), 2012, k dispozici zdarma on-line

Materiály k paralelní části

Protoskripta

- Protoskripta:
  Parallel Programming in C++.
- Postupně na webu předmětu:
- https://cw.fel.cvut.cz/wiki/courses/b4b36pdv/lectures/start
- Řada příkladů.

An example of the use of coroutines.

```
// inspired by
↪ https://www.incredibuild.com/blog/cpp-coroutines-lets-play-with-them

#include <coroutine>
#include <generator>
#include <iostream>
#include <syncstream>
#include <memory>
#include <string>

std::generator<char> split-by-value(std::string s) {
    for (char ch : ps) {
        co_yield ch;
    }
}

std::generator<char>
↪ split-by-uniqueptr(std::unique_ptr<std::string> ps) {
    for (char ch : *ps) {
        co_yield ch;
    }
}

int main() {
    for (char ch : split-by-value("test")) {
        std::osyncstream(std::cout) << ch << '\n';
    }
    for (char ch :
    ↪ split-by-uniqueptr(std::make_unique<std::string>("west")))
    ↪ {
        std::osyncstream(std::cout) << ch << '\n';
    }
}
```

Open in Compiler Explorer

Bonusová úroveň

Práce s větším hardware

- RCI ČVUT cluster
  - n01-20 CPU nodes: 24 cores/48 threads 3.2GHz (2 x Intel Xeon Scalable Gold 6146), 384GB RAM,
  - n21-n32 GPU nodes: 36 cores/72 threads 2.7GHz (2 x Intel Xeon Scalable Gold 6150), 384GB RAM, 4 x Tesla V100 with NVLink,
  - n33 multi-CPU node: 192 cores/ 384 threads 2.1GHz (8 x Intel Xeon Scalable Platinum 8160), 1536GB RAM

How to start to work on RCI cluster

There are information mainly for new users and also links to follow-up parts of the documentation.

Account

RCI Account is required for access to RCI cluster. This account is automaticaly created for all researches working on RCI project. Non RCI researchers can obtain access to RCI cluster for one year by filling registration form.

Bonusová úroveň

- Metacentrum
  - spojení výpočetních prostředků akademické sítě
  - volně dostupné pro akademické pracovníky, studenty
  - mnoho dostupných strojů (CPU, GPU, Xeon Phi)
  - https://metavo.metacentrum.cz/pbsmon2/hardware

- IT4Innovations (www.it4i.cz)
  - 30 000 jader v Ostravě, příkon přes 1MW
  - komerční výpočty, lze zažádat a získat výpočetní čas pro výzkum

Hodnocení

- Domácí úkoly (40%)
  - Malé domácí úkoly (7x)
  - Velké domácí úkoly (2x)
- Praktický test z paralelního programování (20%).
  Za COVIDu online, letos nejspíše v učebně.
- Teoretický test (40%).
  Podlední dva roky v Brute jako výběr z vice možností.

Pro úspěšné ukončení musíte získat alespoň 50% z každé části

Hodnocení

- Malá tolerance zpoždění u odevzdávání úloh
- Problémy je dobré řešit včas na fóru CourseWare
  - https://cw.fel.cvut.cz/wiki/courses/b4b36pdv/start
- Prostředí BRUTE, CLion (https://download.cvut.cz/jetbrains/), příp. Windows Subsystem for Linux

Malé úlohy

Zpoždění	Penalizace
0h $< x \leq$ 1h	-0.5b
1h $< x \leq$ 24h	-1b
24h $< x$	-2b

Semestrální úlohy

Zpoždění	Penalizace
0h $< x \leq$ 1h	-1b
1h $< x \leq$ 12h	-2b
12h $< x \leq$ 3d	-6b
3d $< x \leq$ 6d	-10b
6d $< x$	-100%

Co udělat pro úspěšné zvládnutí PDV?

- Programovat
  - zkoušejte si kódy z přednášek, upravujte jej, analyzujte co se stane
  - nechte si čas na vypracování domácích úkolů

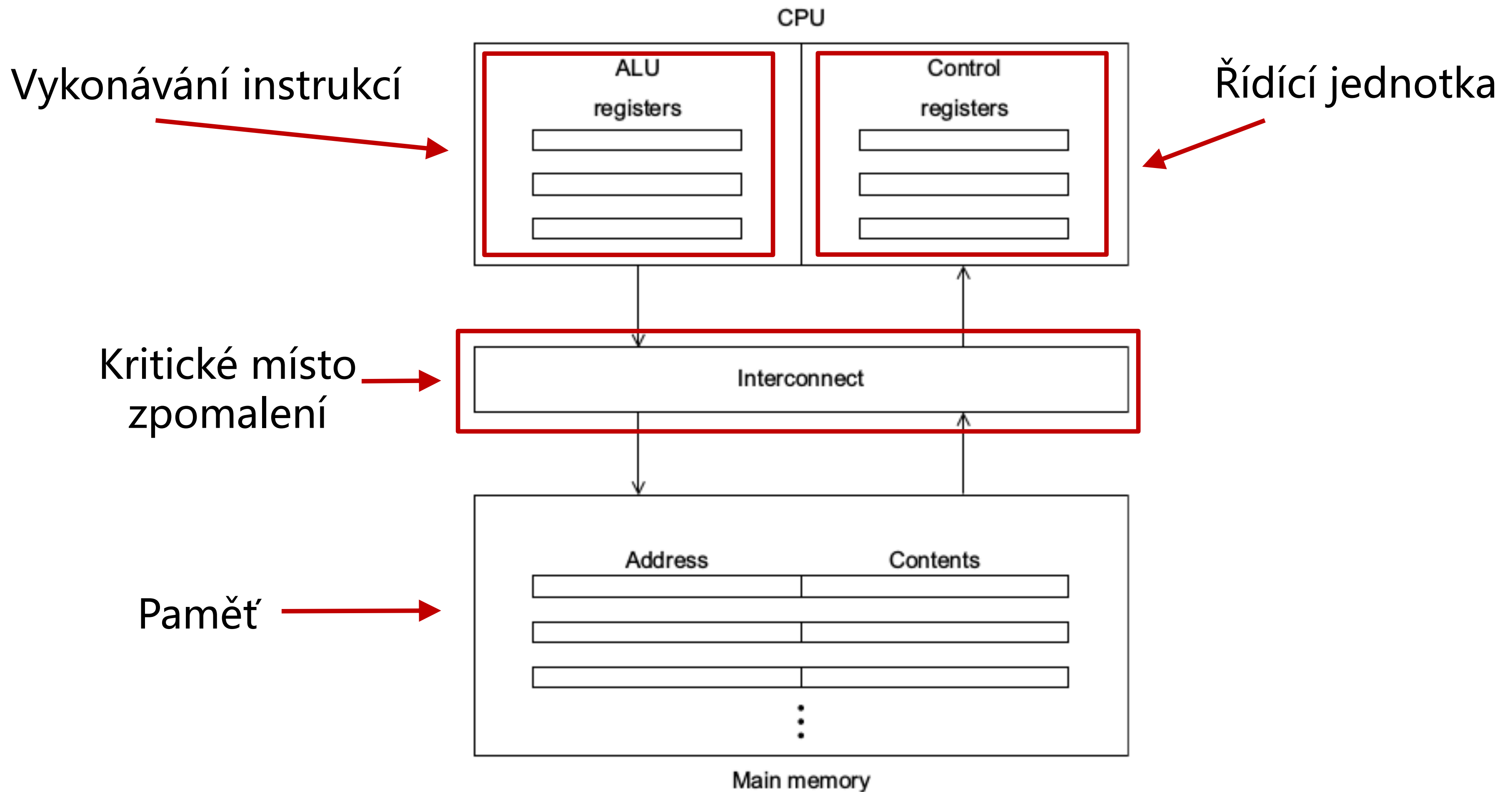
- Přemýšlet
  - paralelní / distribuované programy se špatně ladí
  - chyby ve vícevláknové aplikaci se těžko odhalují
  - pokud program nepracuje jak očekáváte (např. není dostatečně rychlý, výsledek není správný), **zastavte se a zamyslete se proč tomu tak je**

Zbytek dnešní přednášky: Opakování

- Potřebný HW základ a krátká historie
- Jednoduché příklady (jak to dělat a nedělat);
  Vliv architektury
- Nové státnicové otázky

Potřebný HW základ

Von Neumannova architektura



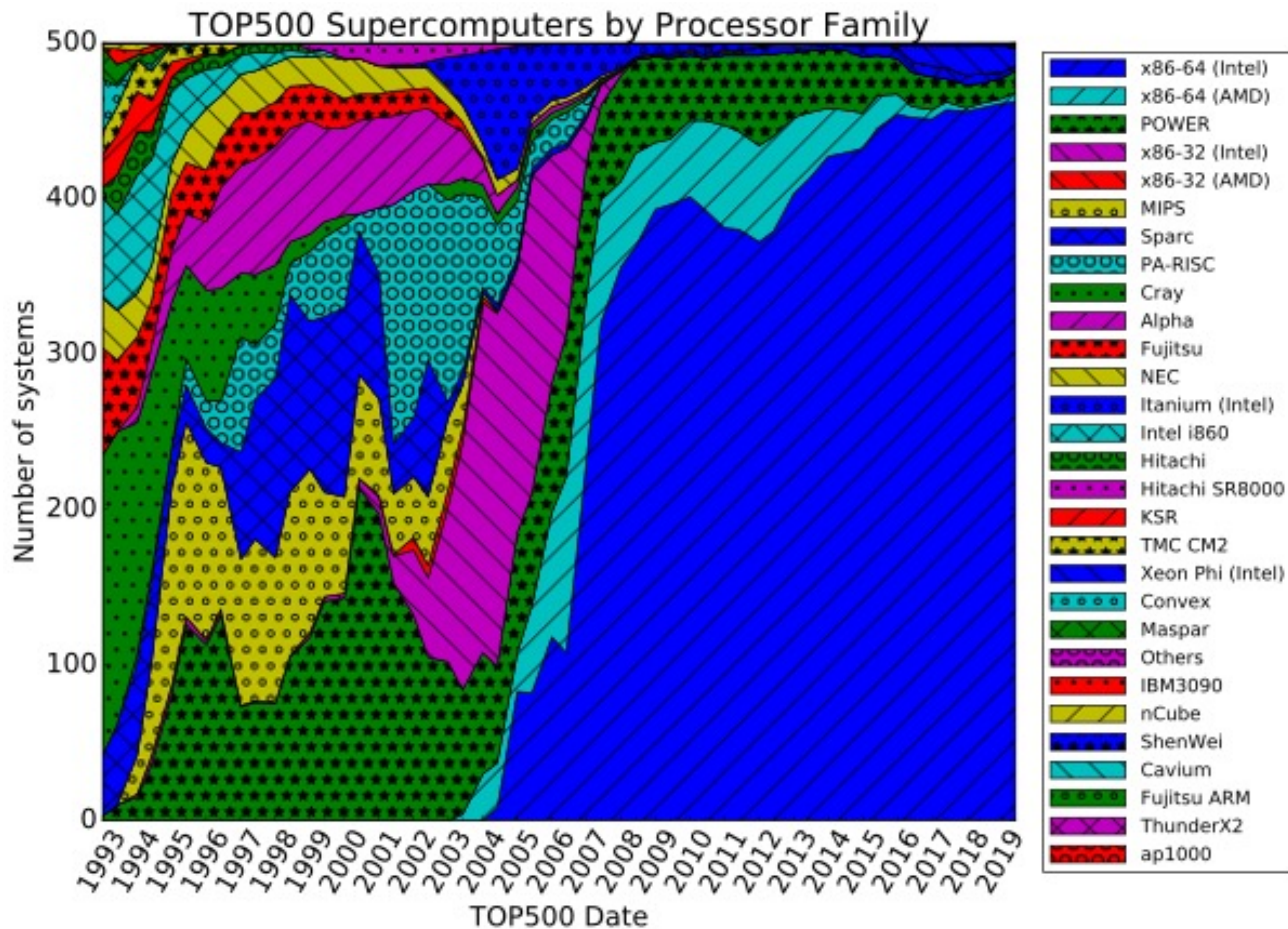
Obrázek převzat z knihy Parallel Programming (by Peter Pacheco)

Krátká historie paralelních výpočtů

Dnešní paralelní stroje

Krátká historie paralelních výpočtů

TOP 500 superpočítačů

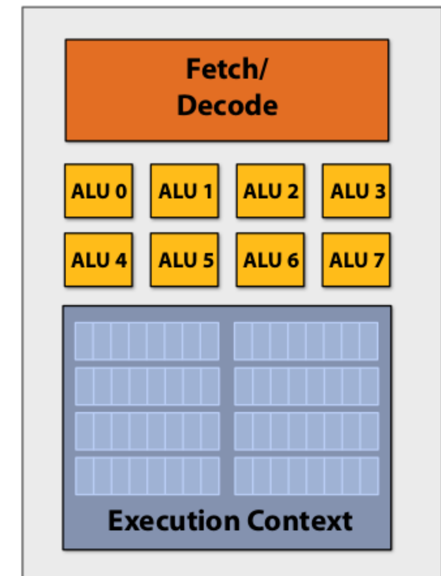


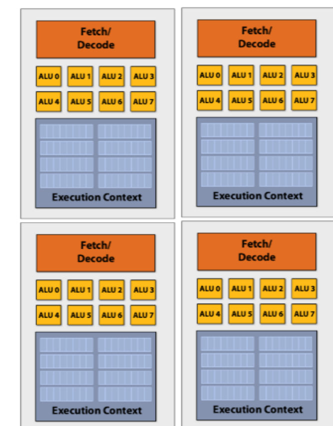
Zdroj: https://en.wikipedia.org/wiki/TOP500

Potřebný HW základ

Abstrakce paralelního hardware dle Flynnovy taxonomie

- SIMD (Single Instruction Multiple Data)
  - Jedna řídící jednotka, vícero ALU jednotek
  - Datový paralelismus
  - Vektorové procesory, GPU
  - Běžné jádra CPU podporují SIMD paralelizmus
    - instrukce SSE, AVX
- MIMD (Multiple Instruction Multiple Data)
  - Více-jádrové procesory
    - např. i v mobilních telefonech
  - Různé jádra vykonávají různé instrukce
  - Víceprocesorové počítače



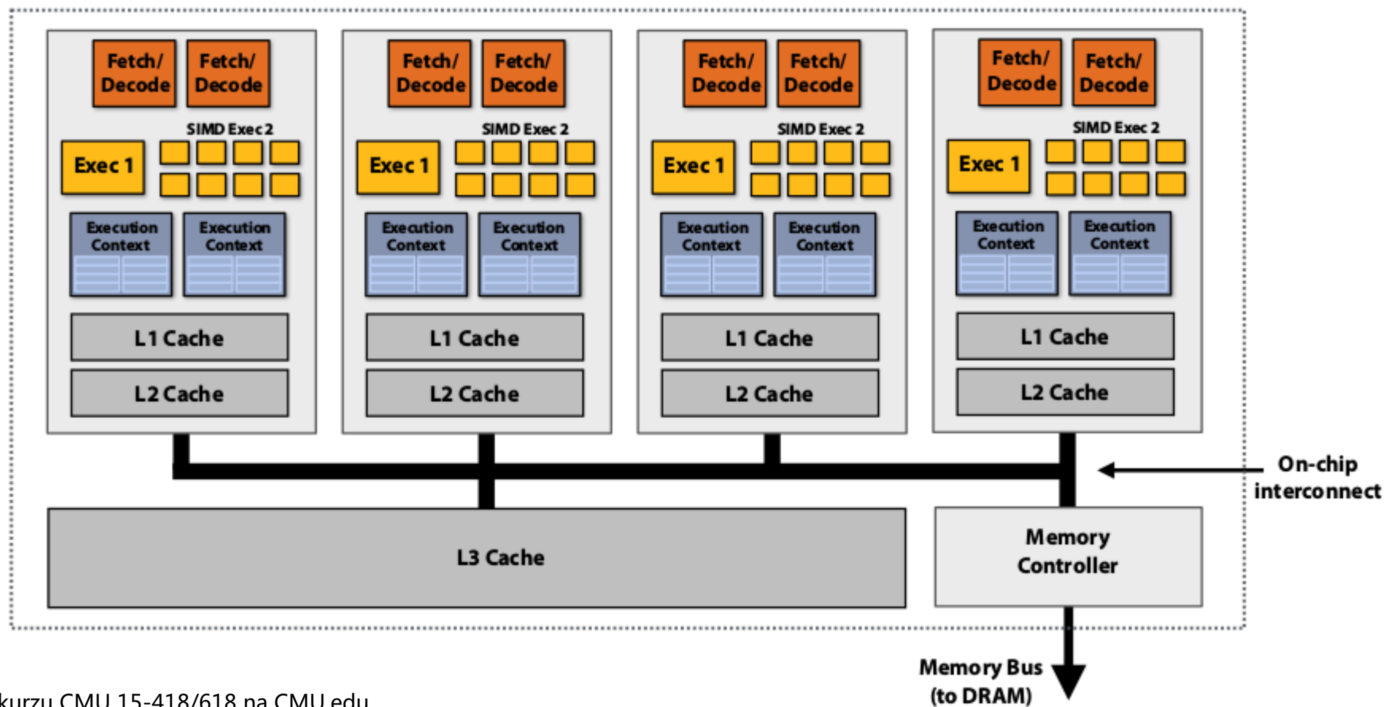


Obrázky převzaty z kurzu CMU 15-418/618 na CMU.edu

Potřebný HW základ

Abstrakce levného moderního procesoru

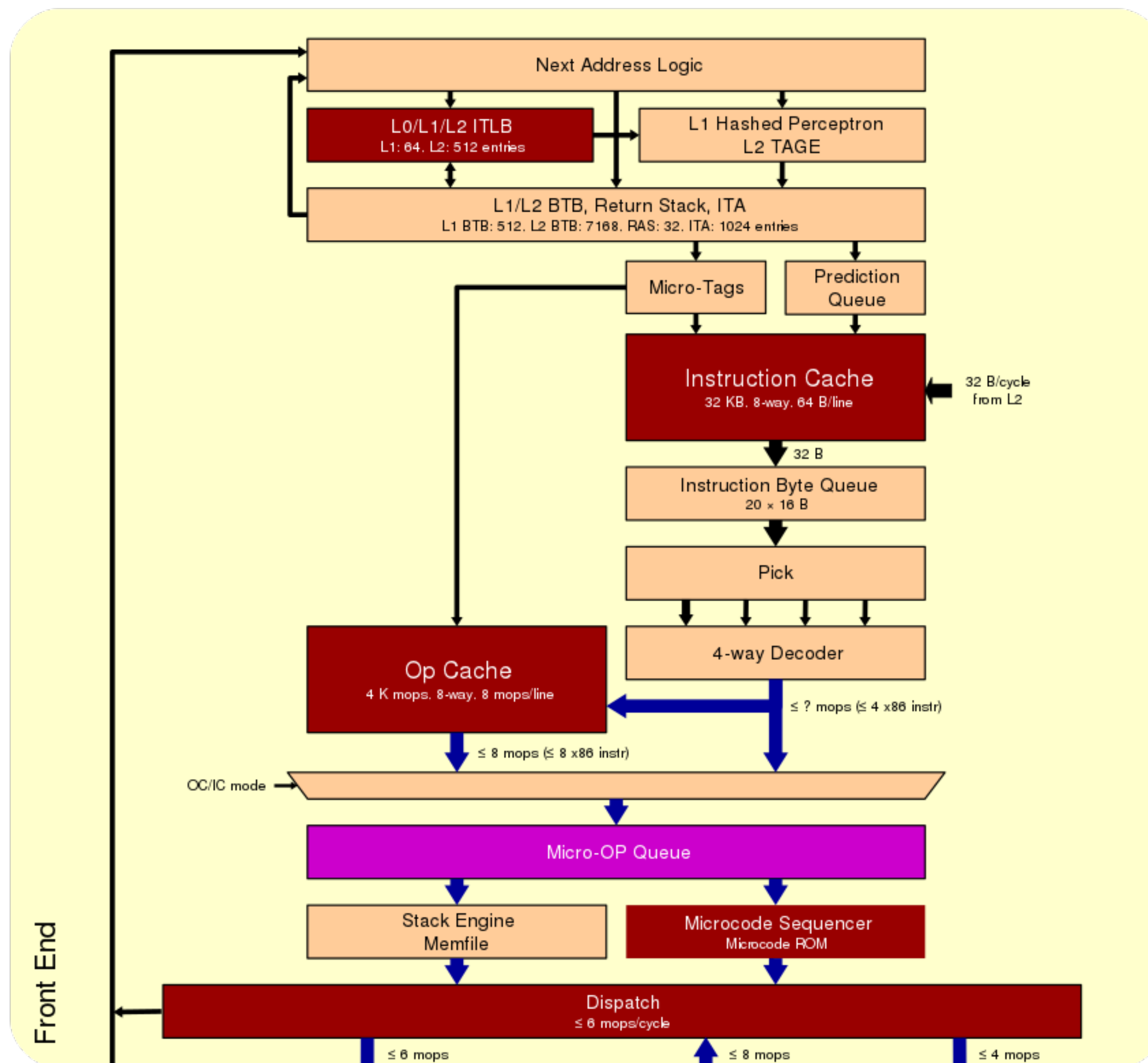
- Vyrovnávací paměť (CPU cache)
  - Programy často přistupují k paměti lokálně (lokalita v prostoru a čase)
  - Cache se upravuje po řádcích (lines)
- Každé jádro má vlastní cache + existuje společná cache



Obrázek převzaty z kurzu CMU 15-418/618 na CMU.edu

Potřebný HW základ

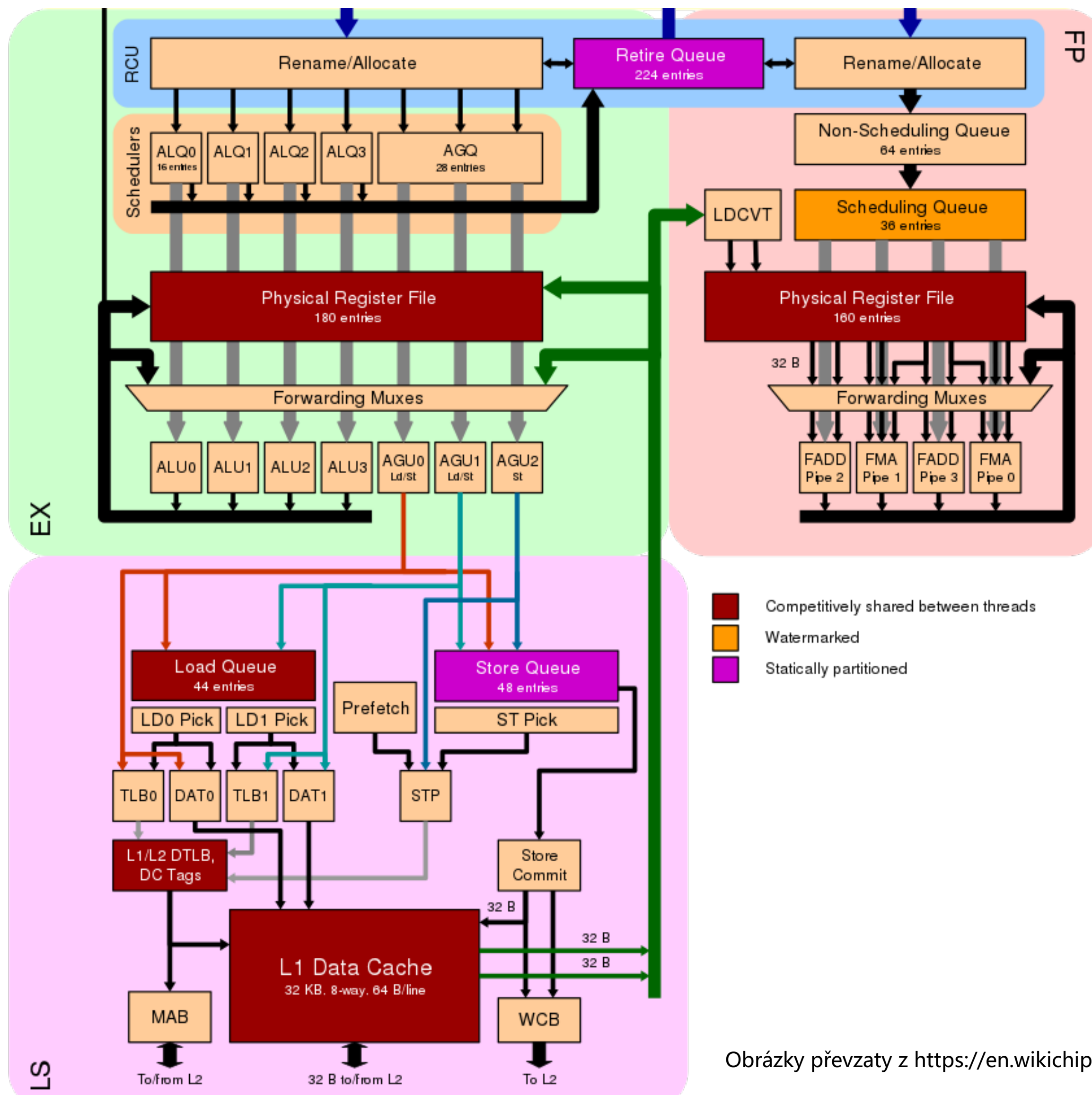
Realita levného moderního procesoru je podstatně složitější



Obrázky převzaty z https://en.wikichip.org/wiki/amd/microarchitectures/zen_3

Potřebný HW základ

Realita levného moderního procesoru je podstatně složitější



Competitively shared between threads

Watermarked

Statically partitioned

Obrázky převzaty z https://en.wikichip.org/wiki/amd/microarchitectures/zen_3

Potřebný HW základ

Pipelines

- Zopakujeme 5 konceptů z APO.
- Paralelizace na úrovni instrukcí (ILP). Příklad:
  - Chceme sečíst 2 vektory reálných čísel (float [1000])
  - 1 součet – 7 operací
    - Načtení (fetch)
    - Porovnání exponentů
    - Posun
    - Součet
    - Normalizace
    - Zaokrouhlení
    - Uložení výsledku
  - Bez ILP – 7x1000x (čas 1 operace; 1ns)

Potřebný HW základ

Pipelines

- Paralelizace na úrovni instrukcí (ILP)
- Příklad:
  - Chceme sečíst 2 vektory reálných čísel (float [1000])
  - 1 součet – 7 operací
  - Bez ILP – 7x1000x (čas 1 operace; 1ns)
  - ILP (a 7 jednotek) – 1005 ns

Table 2.3 Pipelined Addition. Numbers in the Table Are Subscripts of Operands/Results

Time	Fetch	Compare	Shift	Add	Normalize	Round	Store
0	0						
1	1	0					
2	2	1	0				
3	3	2	1	0			
4	4	3	2	1	0		
5	5	4	3	2	1	0	
6	6	5	4	3	2	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
999	999	998	997	996	995	994	993
1000		999	998	997	996	995	994
1001			999	998	997	996	995
1002				999	998	997	996
1003					999	998	997
1004						999	998
1005							999

Obrázek převzat z knihy Parallel Programming (by Peter Pacheco)

Potřebný HW základ

Superskalární procesory

- Současné vyhodnocení vícero instrukcí
  - uvažme cyklus

```
for (i=0; i<1000; i++)
    z[i]=x[i]+y[i];
```

  - jedna jednotka může počítat z[0], druhá z[1], ...
- Spekulativní vyhodnocení

```
z = x + y;
if (z > 0)
    w = x;
else
    w = y;
```

https://cw.fel.cvut.cz/wiki/_media/courses/b35apo/en/lectures/06/b35apo_lecture06-speculative.pdf

Vliv architektury

Cache

- Proč je důležité vědět o architektuře?
  - Uvažme příklad násobení matice vektorem

```
int x[MAXIMUM], int y[MAXIMUM], int A[MAXIMUM*MAXIMUM]
```

Varianta A

```
for ( int i = 0; i < MAXIMUM ; i ++)
    for ( int j = 0; j < MAXIMUM ; j ++)
        y[i] += A->at(i * MAXIMUM + j)*x[j];
```

Varianta B

```
for ( int j = 0; j < MAXIMUM ; j ++)
    for ( int i = 0; i < MAXIMUM ; i ++)
        y [i] += A->at(i * MAXIMUM + j)*x[j];
```

Který kód bude rychlejší?

Vliv architektury

Cache

```
for ( int i = 0; i < MAXIMUM ; i ++)
    for ( int j = 0; j < MAXIMUM ; j ++)
        y[i] += A->at(i * MAXIMUM + j)*x[j];
```

```
for ( int j = 0; j < MAXIMUM ; j ++)
    for ( int i = 0; i < MAXIMUM ; i ++)
        y [i] += A->at(i * MAXIMUM + j)*x[j];
```

- Pole jsou v paměti uložena sekvenčně (po řádcích)
- CPU při přístupu k A[0][0] načte do cache vícero hodnot (cache line)

Cache Line	Elements of A			
0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
1	A[1][0]	A[1][1]	A[1][2]	A[1][3]
2	A[2][0]	A[2][1]	A[2][2]	A[2][3]
3	A[3][0]	A[3][1]	A[3][2]	A[3][3]

- Při přístupu k A[1][0] se změní celý řádek

V rámci paralelních programů může k podobným problémům docházet častěji

Paralelizace

Jednoduchý příklad

- Suma vektoru čísel

0	**1**	**2**	**3**	**4**	**5**	**6**	**…**	**…**	**$5x10^9$**
17	2	9	4	22	0	1			8

Jak paralelizovat?

- Mějme 4 jádra – každé jádro může sečíst čtvrtinu vektoru, pak sečteme částečné součty

Vláken	**1**	**2**	**3**	**4**
Čas	0.389s	0.262s	0.258s	0.244s

Měřeno na 8 jádrovém (16 vláknovém) Intel Xeon v4 E5-2620

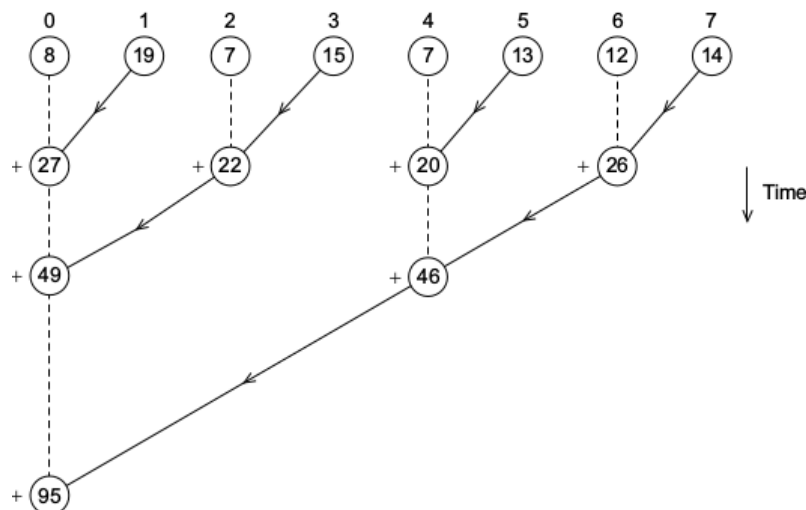
Paralelizace

Jednoduchý příklad

- Suma vektoru čísel

Co když máme tisíce jader?

- Pokud částečné součty sčítá pouze jedno jádro, kód není velmi efektivní



Obrázek převzat z knihy Parallel Programming (by Peter Pacheco)

Pokročilejší příklad

- Zkusme sčítat celou část druhých odmocnin

sčítané pole

id vlákna

pole pro dílčí součty

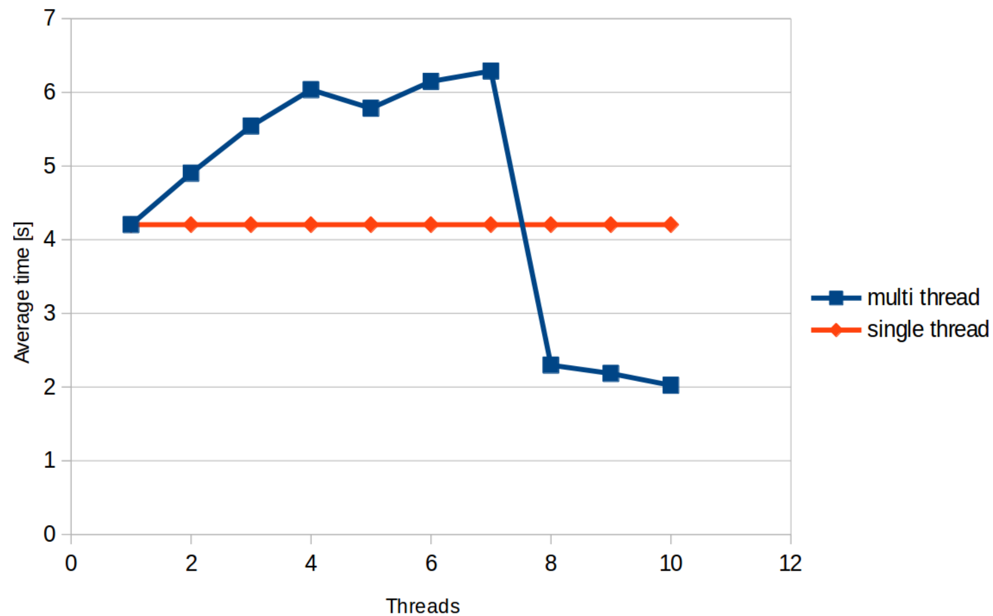
```
long sum(std::vector<int>& vector_to_sum, int thread, std::vector<long>& sums) {
    for (int i=thread; i<SIZE; i += thread_count)
        sums[thread] += sqrt(vector_to_sum[i]);

    for (int j=1; j<log2(thread_count)+1; j++) {
        if ((thread % (int)pow(2,j)) != 0) break;
        int k = (int)pow(2,j-1);
        if ((thread + k) >= thread_count) break;
        if (threads[thread + k].joinable()) threads[thread + k].join();
        sums[thread] += sums[thread + k];
    }
}
```

logaritmický součet dílčích výsledků

každé vlákno zapisuje na vlastní index pole

Pokročilejší příklad

Jak nám to bude fungovat?



Nic moc :(

Měřeno na 8 jádrovém (16 vláknovém) Intel Xeon v4 E5-2620

Pokročilejší příklad

Kde je chyba?

```
long sum(std::vector<int>& vector_to_sum, int thread, std::vector<long>& sums) {
    for (int i=thread; i<SIZE; i += thread_count)
        sums[thread] += sqrt(vector_to_sum[i]);

    for (int j=1; j<log2(thread_count)+1; j++) {
        if ((thread % (int)pow(2,j)) != 0) break;
        int k = (int)pow(2,j-1);
        if ((thread + k) >= thread_count) break;
        if (threads[thread + k].joinable()) threads[thread + k].join();
        sums[thread] += sums[thread + k];
    }
}
```

každé vlákno zapisuje na vlastní index pole

0	1	2	3	4	5	6	7	8	9
17	2	9	4	22	0	1	0	0	8

Pokročilejší příklad

Kde je chyba?

```
long sum(std::vector<int>& vector_to_sum, int thread, std::vector<long>& sums) {
    for (int i=thread; i<SIZE; i += thread_count)
        sums[thread] += sqrt(vector_to_sum[i]);

    for (int j=1; j<log2(thread_count)+1; j++) {
        if ((thread % (int)pow(2,j)) != 0) break;
        int k = (int)pow(2,j-1);
        if ((thread + k) >= thread_count) break;
        if (threads[thread + k].joinable()) threads[thread + k].join();
        sums[thread] += sums[thread + k];
    }
}
```

- vlákno 0 upraví hodnotu
- jenže vlákno 0 má celý vektor **sums** v cache jádra
- a podobně i jiné vlákna
- při změně 1 hodnoty se musí zabezpečit konzistence

0	1	2	3	4	5	6	7	8	9
17	2	9	4	22	0	1	0	0	8

False Sharing

False Sharing

možné řešení

```
long sum_local(std::vector<int>& vector_to_sum, int thread, std::vector<long>& sums) {
    long local = 0;
    for (int i=thread; i<SIZE; i += thread_count) {
        local += sqrt(vector_to_sum[i]);
    }
    sums[thread] = local;

    for (int j=1; j<log2(thread_count)+1; j++) {
        if ((thread % (int)pow(2,j)) != 0) break;
        int k = (int)pow(2,j-1);
        if ((thread + k) >= thread_count) break;
        if (threads[thread + k].joinable()) threads[thread + k].join();
        local += sums[thread + k];
    }
    sums[thread] = local;
}
```

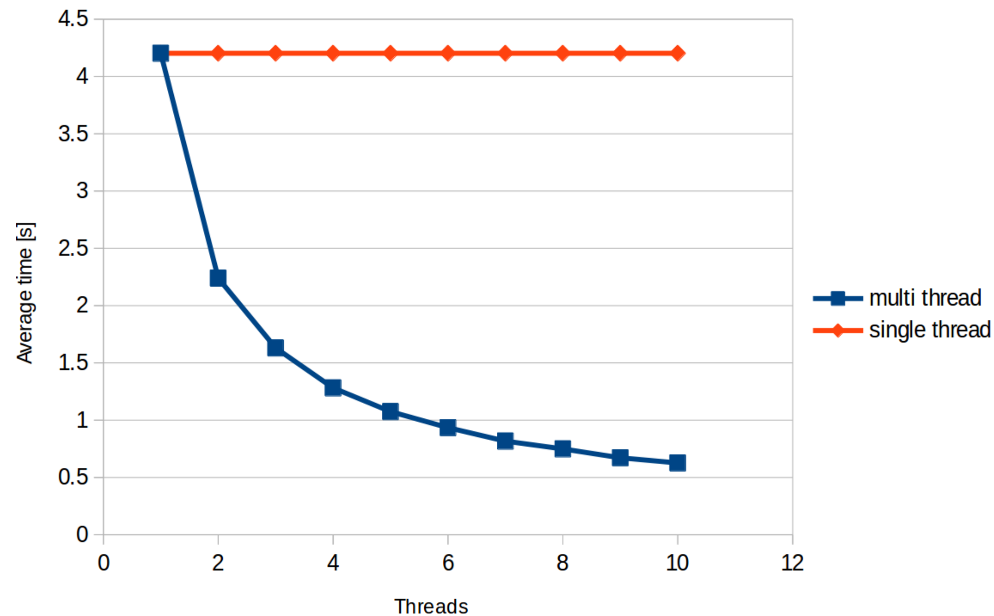
každé vlákno zapisuje do lokální proměnné

pouze finální výsledek se zapíše do vektoru

Potřebný HW základ

False Sharing

lokální proměnná – opravdu to pomůže?



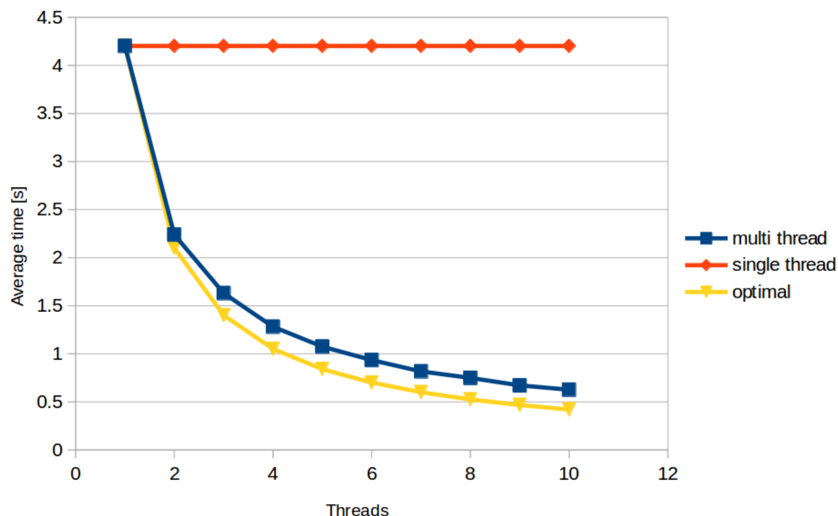
Paralelní programování

Měření zrychlení

Je dané zrychlení dostatečné? Můžeme být rychlejší?

- V optimálním případě se paralelní verze zrychluje proporčně s počtem jader

Vláken	1	2	3	4
Čas	x	x/2	x/3	x/4



Často vyjádřeno jako zrychlení:

$$S = \frac{T_{serial}}{T_{parallel}}$$

Paralelní programování

Měření zrychlení

Můžeme se vždy dostat k lineárnímu zrychlení?

- Paralelní verze algoritmů mají (téměř) vždy další režii
  - spouštění vláken
  - zámky
  - synchronizace
  - …
- Program/algoritmus často vyžaduje určitou sériovou část
  - Nechť jsme schopni přepsat 90% kódu s lineárním zrychlením
  - $S = \frac{T_{serial}}{0.9 \times \frac{T_{serial}}{p} + 0.1 \times T_{serial}} \leq \frac{T_{serial}}{0.1 \times T_{serial}}$
  - To znamená, že pokud sériový program trvá 20 sekund, nikdy nedosáhneme zrychlení větší než 10

Amdahlův zákon

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.