Paralelní a distribuované výpočty (B4B36PDV)

Jakub Mareček, Michal Jakob

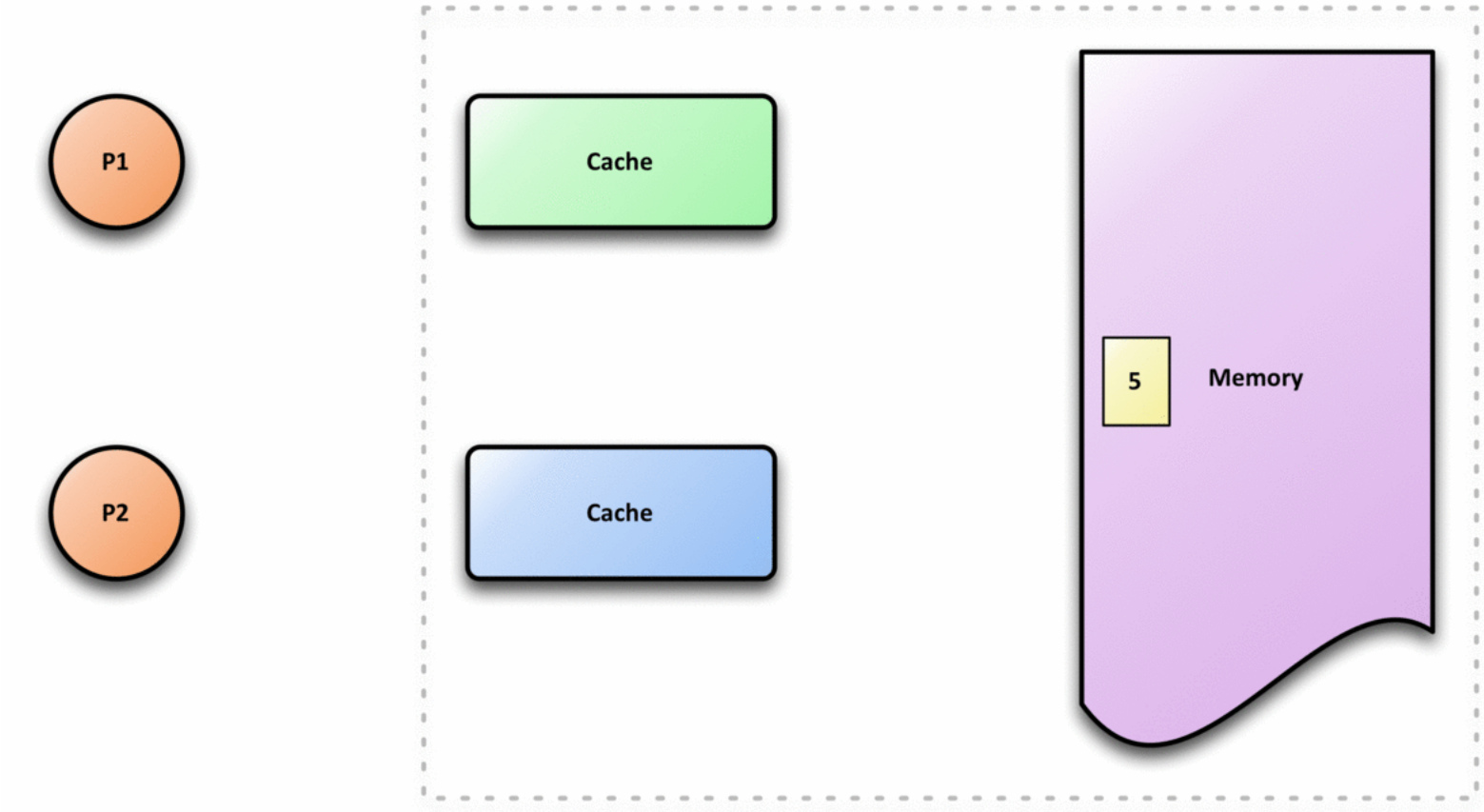
jakub.marecek@fel.cvut.cz

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

Dnes s využitím materiálů Brana Bošanského: https://cw.fel.cvut.cz/b192/courses/b4b36pdv/lectures/start

K předchozím přednáškám

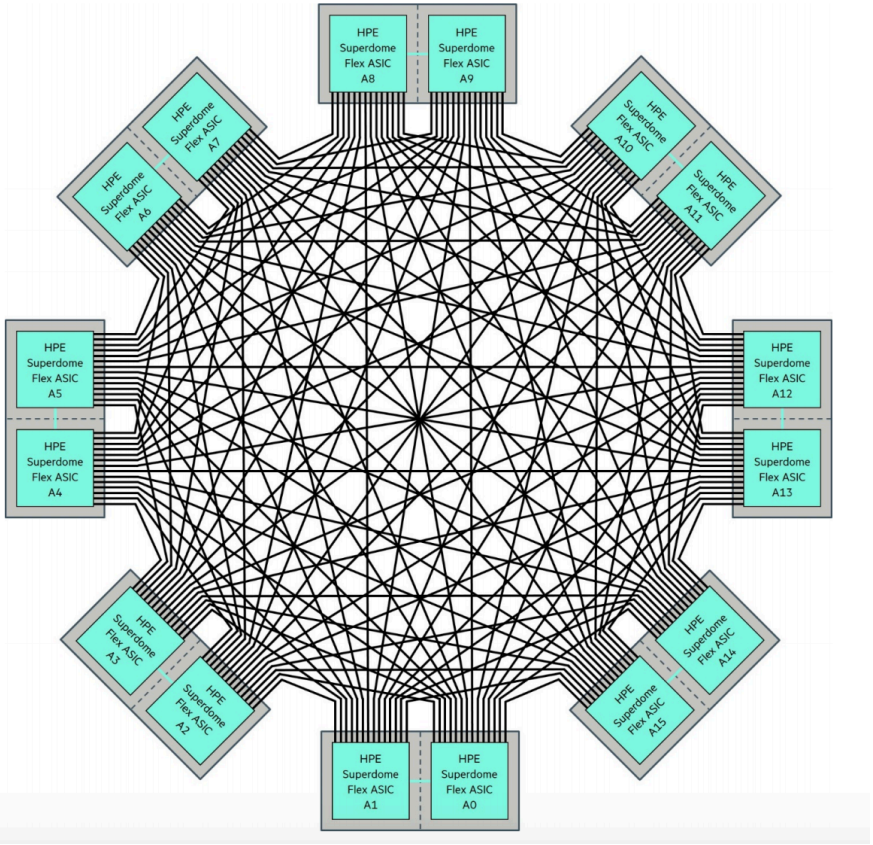
Motivace

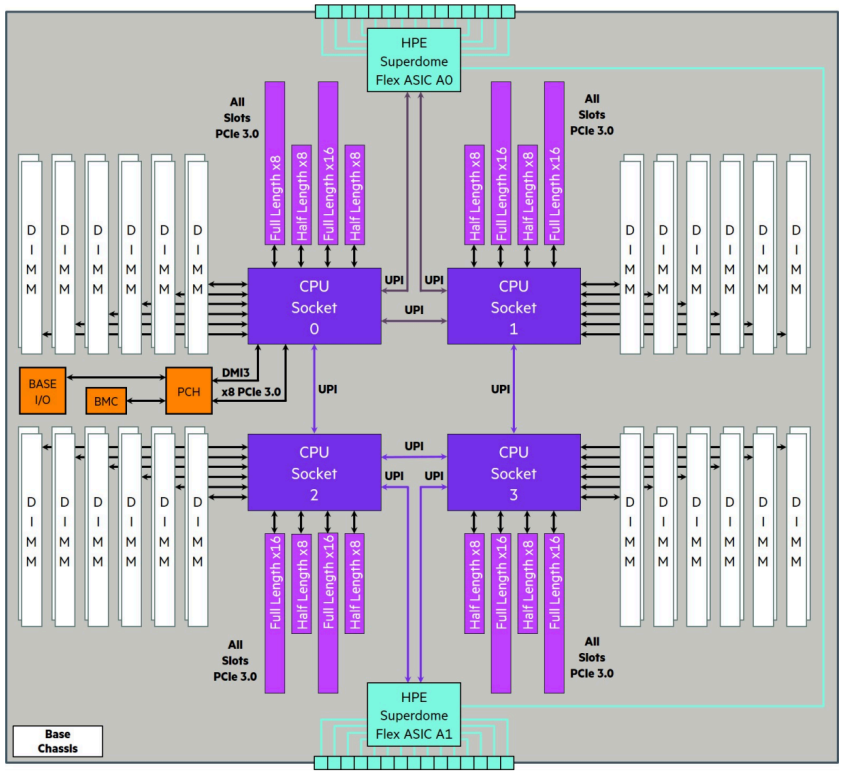


Klíčem pro porozumění moderním systémům je koherence hierarchií vyrovnávacích pamětí.
Animace od M3tainfo, https://en.wikipedia.org/wiki/Cache_coherence#/media/File:Coherent.gif

K předchozím přednáškám

Motivace



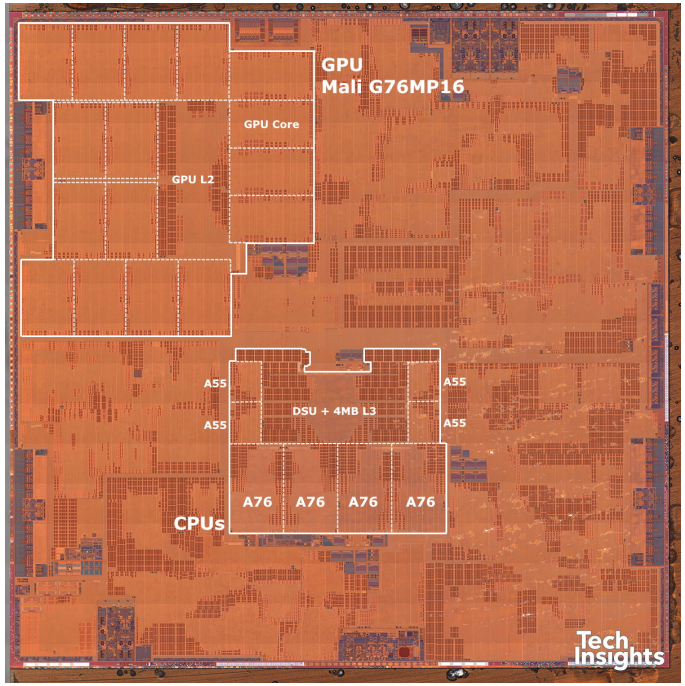


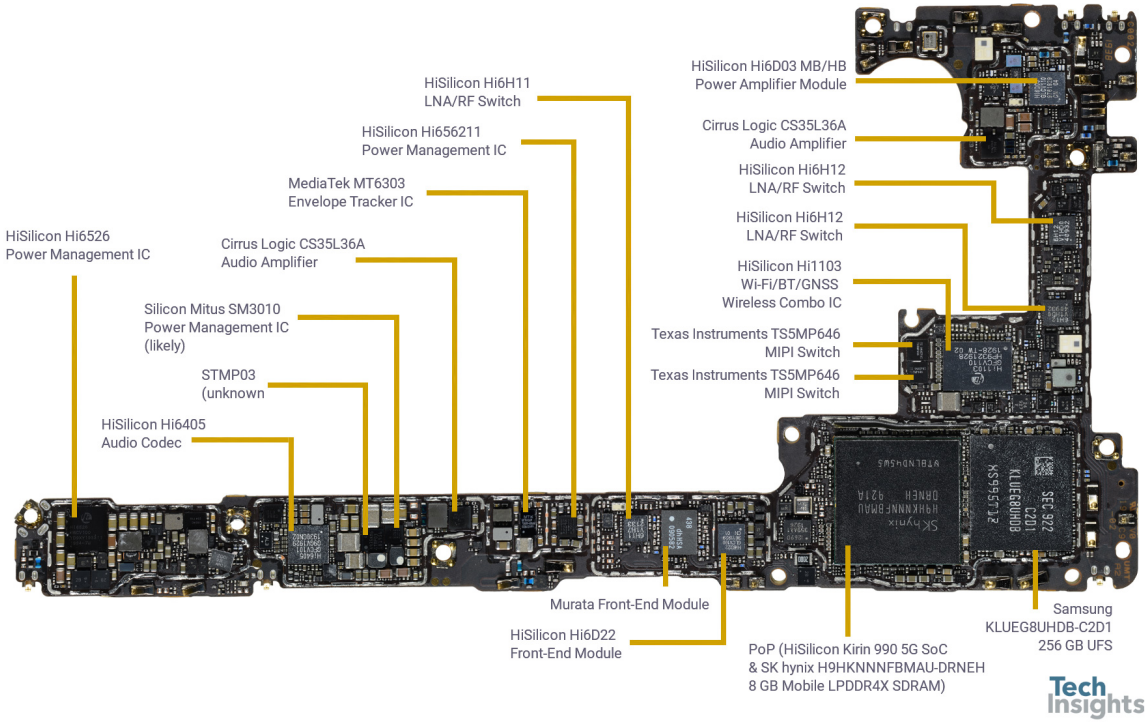
V „cache-coherent " systémech typicky máme přístup do paměti neuniformní (NUMA); různé množství kroků (hops).

Zde na příkladu HPE Superdome, kde můžeme mít 32x4 Xeon procesory (tedy až 7168 HW vláken).
https://assets.ext.hpe.com/is/content/hpedam/documents/a00036000-6999/a00036491/a00036491enw.pdf

K předchozím přednáškám

Motivace

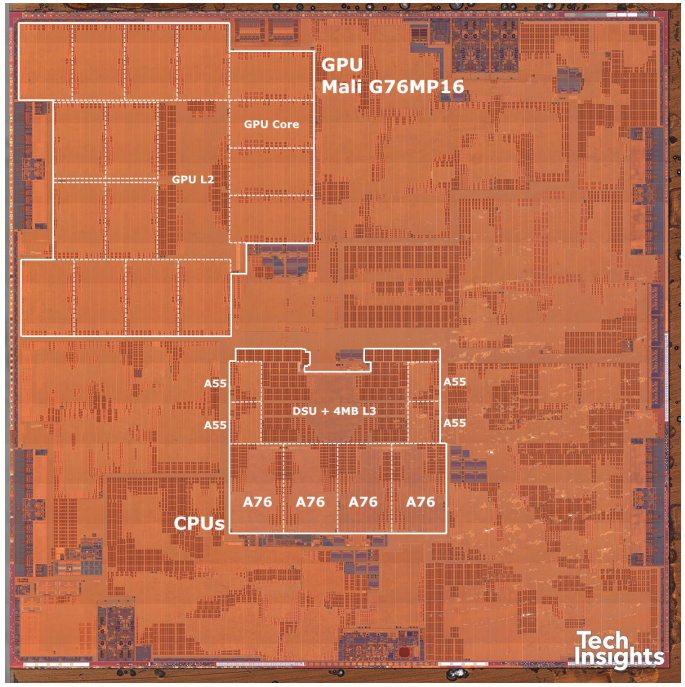


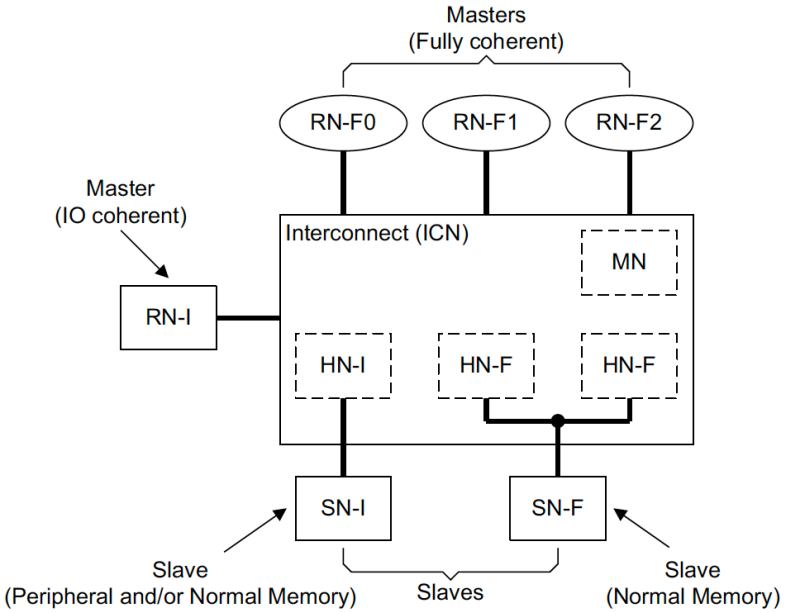


V „cache-coherent “ systémech typicky máme přístup do paměti neuniformní (NUMA); různé množství kroků (hops). Zde na příkladu Hisilicon Kirin 990 (AMBA 5, CoreLink CMN-600 Interconnect?) v Huawei Mate 30 Pro podle: https://www.techinsights.com/blog/huawei-mate-30-pro-5g-teardown

K předchozím přednáškám

Motivace

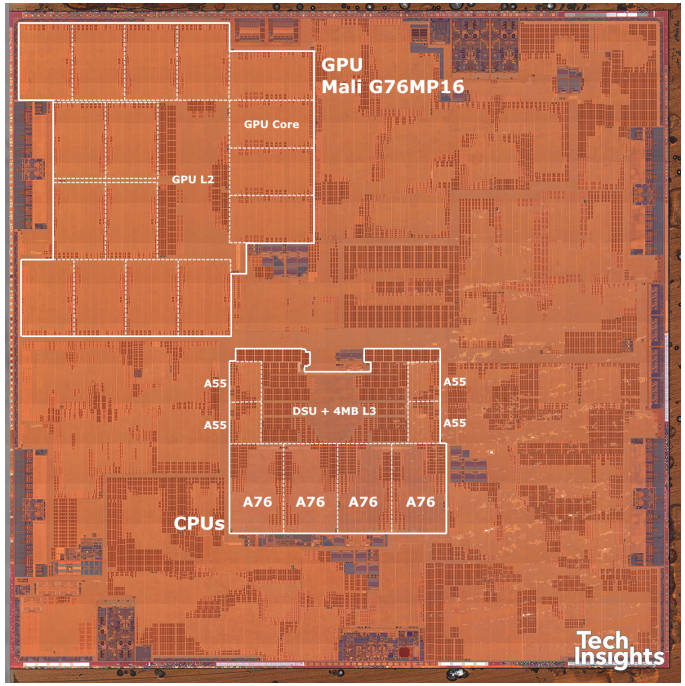


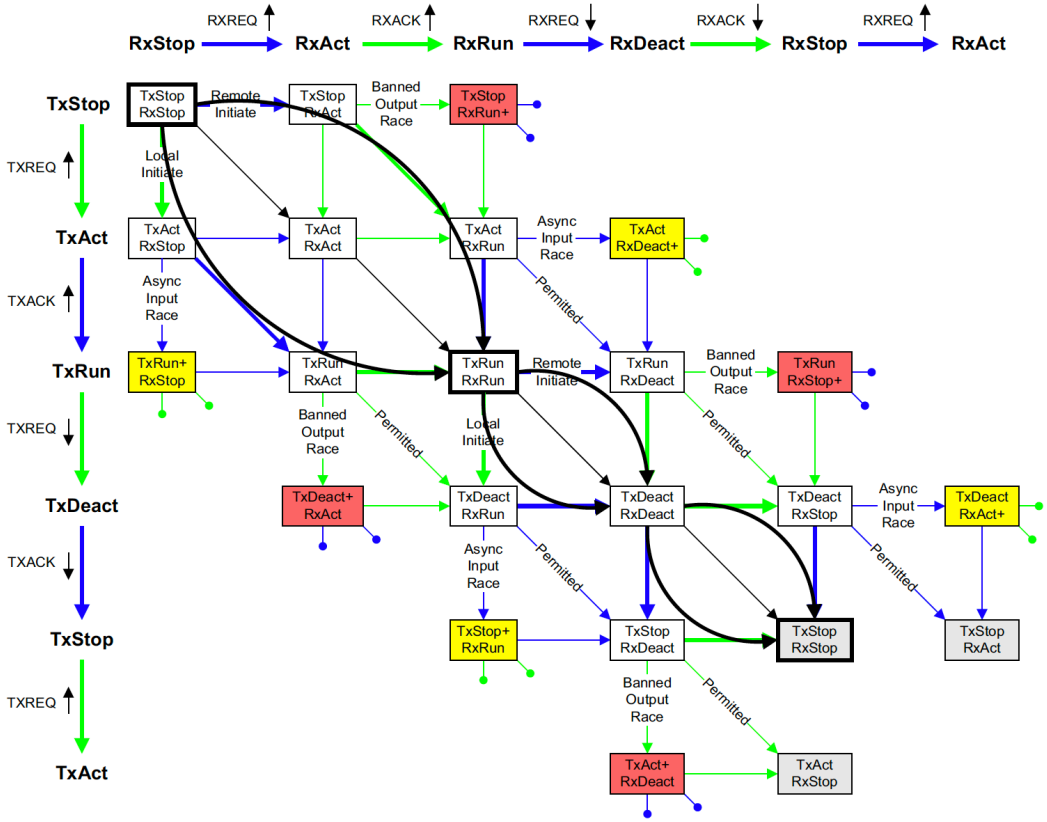


V „cache-coherent " systémech typicky máme přístup do paměti neuniformní (NUMA); různé množství kroků (hops). Zde na příkladu Hisilicon Kirin 990 (AMBA 4, CoreLink CCI-400 Interconnect?) v Huawei Mate 30 Pro podle: https://developer.arm.com/documentation/ihi0050/latest

K předchozím přednáškám

Motivace





V „cache-coherent “ systémech typicky máme přístup do paměti neuniformní (NUMA); různé množství kroků (hops). Zde na příkladu Hisilicon Kirin 990 (AMBA 4, CoreLink CCI-400 Interconnect?) v Huawei Mate 30 Pro podle: https://developer.arm.com/documentation/ihi0050/latest

K předchozím přednáškám

Motivace

Everything you always wanted to know about synchronization but were afraid to ask

Authors: Tudor David, Rachid Guerraoui, Vasileios Trigonakis Authors Info & Affiliations

Publication: SOSP '13: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles • November 2013 • Pages 33–48 • https://doi.org/10.1145/2517349.2522714

System	Opteron				Xeon			Niagara		Tilera	
State \ Hops	same die	same MCM	one hop	two hops	same die	one hop	two hops	same core	other core	one hop	max hops
loads											
Modified	81	161	172	252	109	289	400	3	24	45	65
Owned	83	163	175	254	-	-	-	-	-	-	-
Exclusive	83	163	175	253	92	273	383	3	24	45	65
Shared	83	164	176	254	44	223	334	3	24	45	65
Invalid	136	237	247	327	355	492	601	176	176	118	162
stores											
Modified	83	172	191	273	115	320	431	24	24	57	77
Owned	244	255	286	291	-	-	-	-	-	-	-
Exclusive	83	171	191	271	115	315	425	24	24	57	77
Shared	246	255	286	296	116	318	428	24	24	86	106
atomic operations: CAS (C), FAI (F), TAS (T), SWAP (S)											
Operation	all	all	all	all	all	all	all	C/F/T/S	C/F/T/S	C/F/T/S	C/F/T/S
Modified	110	197	216	296	120	324	430	71/108/64/95	66/99/55/90	77/51/70/63	98/71/89/84
Shared	272	283	312	332	113	312	423	76/99/67/93	66/99/55/90	124/82/121/95	142/102/141/115

Table 2: Latencies (cycles) of the cache coherence to load/store/CAS/FAI/TAS/SWAP a cache line depending on the MESI state and the distance. The values are the average of 10000 repetitions with $< 3\%$ standard deviation.

Cena „Compare and swap “ závisí na mnoha faktorech, ale je blízko „load “.

K předchozím přednáškám

Motivace

Everything you always wanted to know about synchronization but were afraid to ask

Authors: Tudor David, Rachid Guerraoui, Vasileios Trigonakis Authors Info & Affiliations

Publication: SOSP '13: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles • November 2013 • Pages 33–48 • https://doi.org/10.1145/2517349.2522714

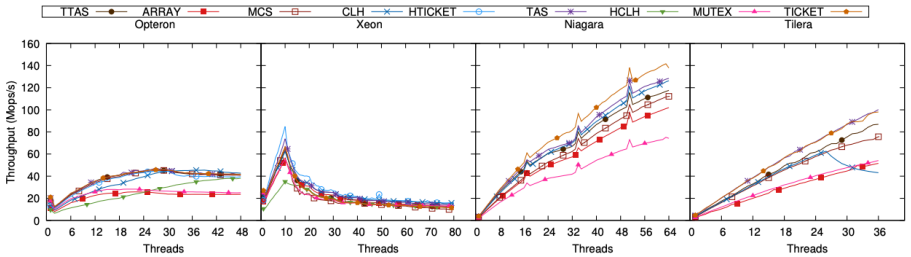


Figure 7: Throughput of different lock algorithms using 512 locks.

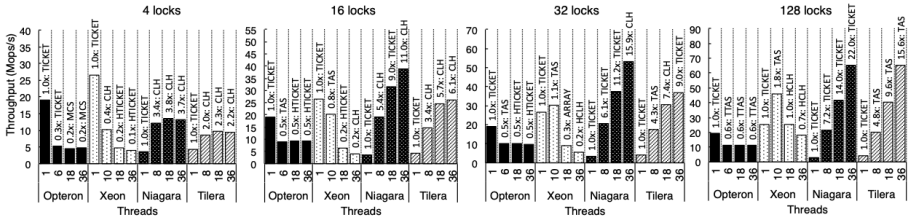


Figure 8: Throughput and scalability of locks depending on the number of locks. The "X : Y" labels on top of each bar indicate the best-performing lock (Y) and the scalability over the single-thread execution (X).

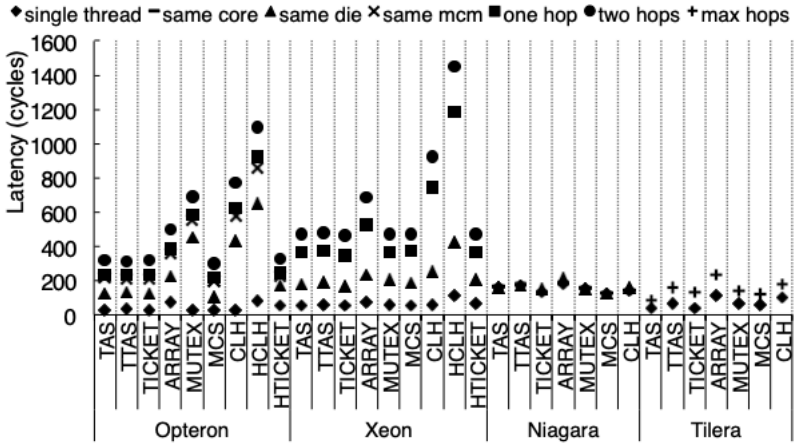


Figure 6: Uncontested lock acquisition latency based on the location of the previous owner of the lock.

Zamčít zámek je až 4 dražší, i pokud je odemčený a předchozí vlastník je na stejném jádru.

K předchozím přednáškám

Motivace

More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms

Publication: PPoPP 2015: Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming • January 2015 • Pages 1–10 • https://doi.org/10.1145/2688500.2688501

"Every locking scheme has its fifteen minutes of fame" -- ale mezi „compare and swap " a zámky může být zásadní rozdíl.

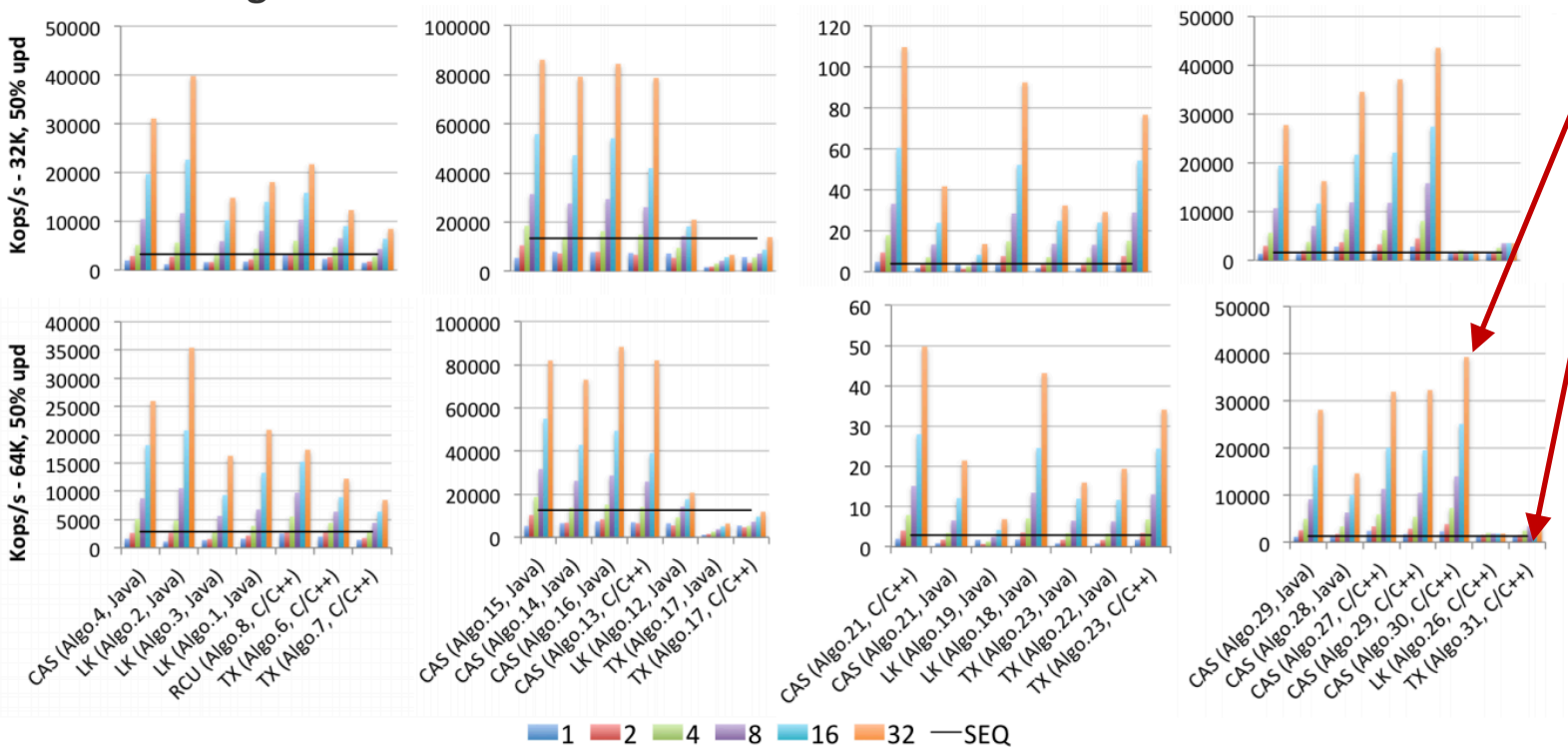


Figure 1. Synchrobench performance results for 1, 2, 4, 8, 16 and 32 threads on the 32-way Intel machine with data structures written in C/C++ and Java, and synchronized with read-modify-write (CAS), locks (LK), read-copy-update (RCU), and transactions (TX), and with

K předchozím přednáškám

HPE SuperdomeFlex w/ 28 Intel Xeon 8180 CPUs, 20 TB DRAM

The Tale of 1000 Cores: An Evaluation of Concurrency Control on Real(ly) Large Multi-Socket Hardware

Tiemo Bang
TU Darmstadt & SAP SE
tiemo.bang@cs.tu-darmstadt.de

Norman May
SAP SE
norman.may@sap.com

Ilia Petrov
Reutlingen University
ilia.petrov@reutlingen-university.de

Carsten Binnig
TU Darmstadt
carsten.binnig@cs.tu-darmstadt.de

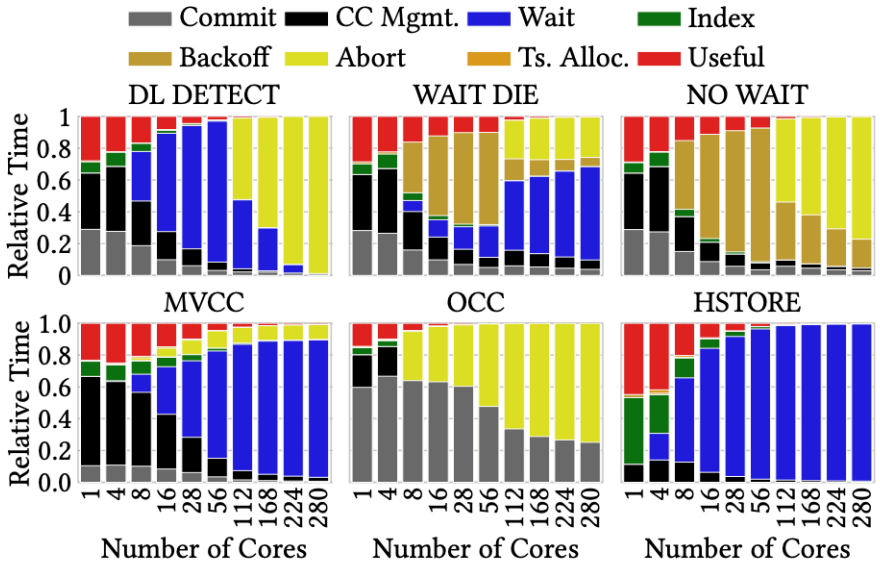


Figure 4: Breakdown of relative time spent for high conflict (4 WH) TPC-C transactions on multi-socket hardware.

ACM Reference Format:
Tiemo Bang, Norman May, Ilia Petrov, and Carsten Binnig. 2020. The Tale of 1000 Cores: An Evaluation of Concurrency Control on Real(ly) Large Multi-Socket Hardware. In *International Workshop on Data Management on New Hardware (DAMON'20), June 15, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3399666.3399910

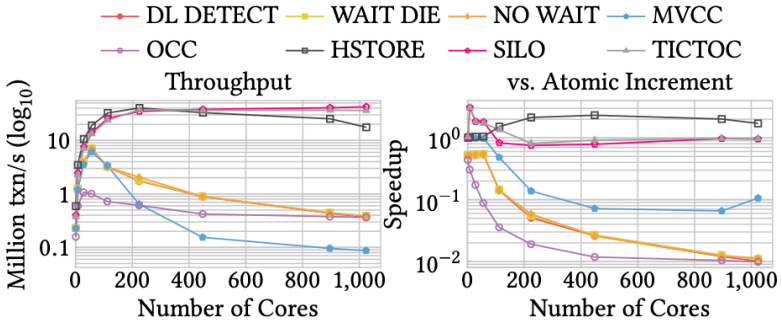


Figure 6: Throughput of TPC-C with 1024 warehouses for timestamp allocation with hardware clock.

Dnešní přednáška

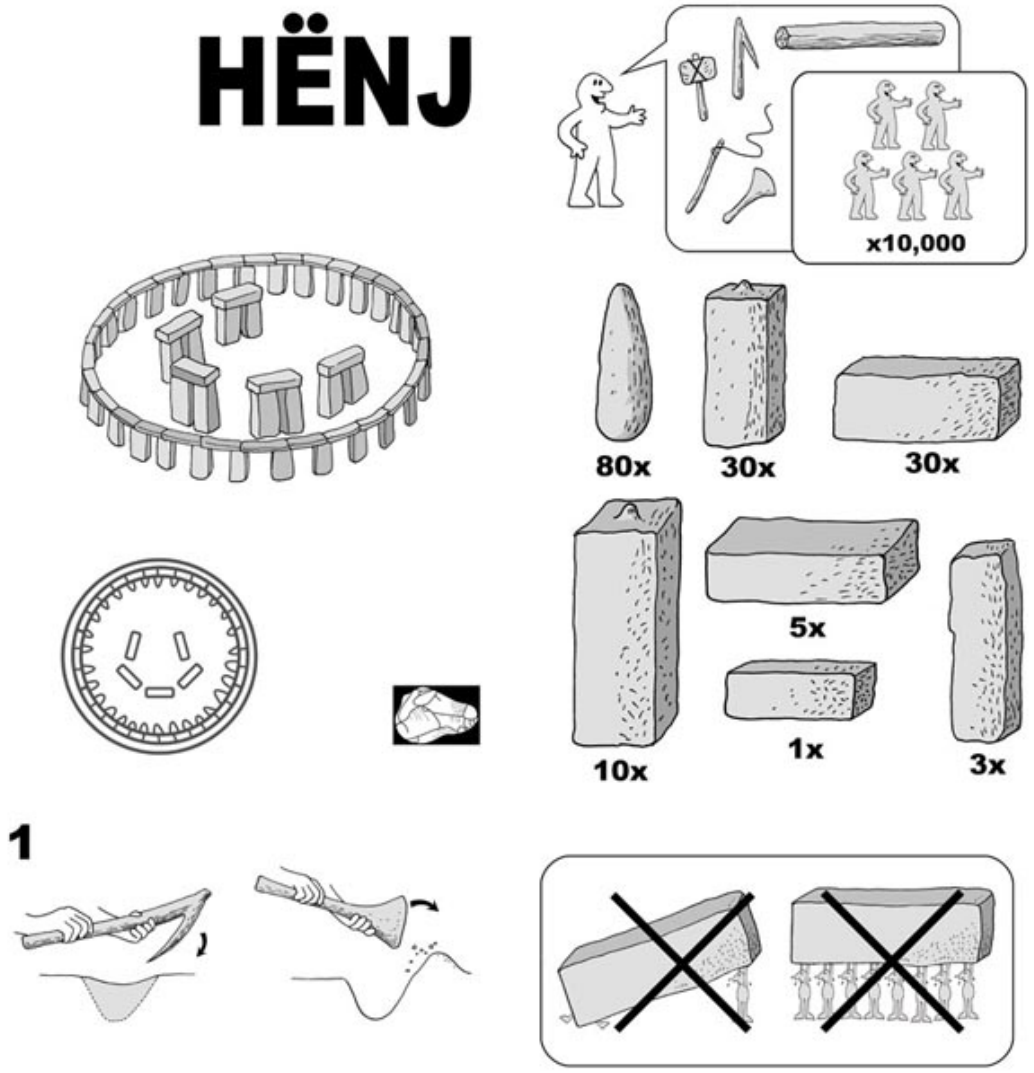
Techniky paralelizace

Chci paralelizovat algoritmus XY

Jak na to?

Dnešní přednáška

Postup – Jak na to?



Paralelní programování

Co chceme dosáhnout

- Potřebujeme se rozhodnout jak budeme úlohu **dekomponovat**, jak budeme **úkoly rozdělovat** a jakým způsobem zabezpečit celkovou orchestraci

- Klíčové cíle
  - **Vybalancování** – aby každé vlákno vykonávalo (přibližně) stejnou práci
  - **Minimalizace komunikace** – aby vlákna na sebe nemusely čekat
  - **Minimalizace duplicitní/zbytečné práce** – aby vlákna nepočítali něco, co by se nepočítalo bez paralelizace

- Neexistuje univerzální návod, musíte vždy přemýšlet jak dané cíle naplnit pro konkrétní úlohu

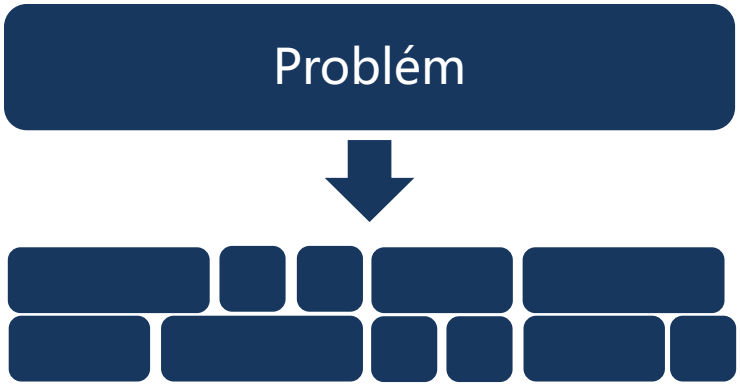
Paralelní programování

Náhled

Problém

Paralelní programování

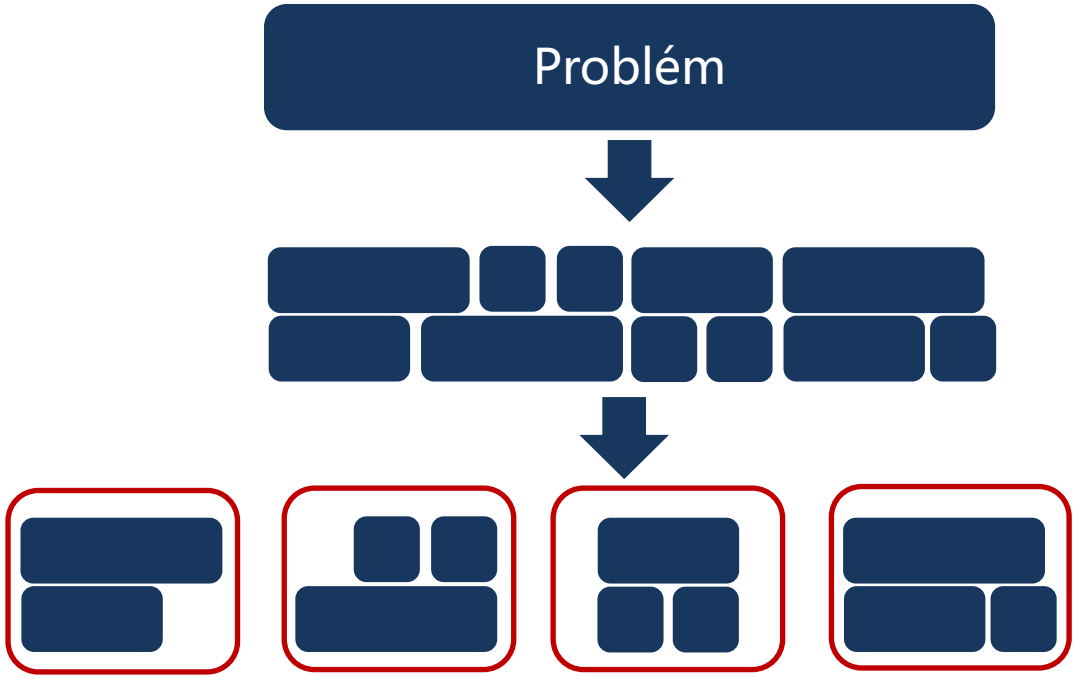
Náhled



Dekompozice

Paralelní programování

Náhled

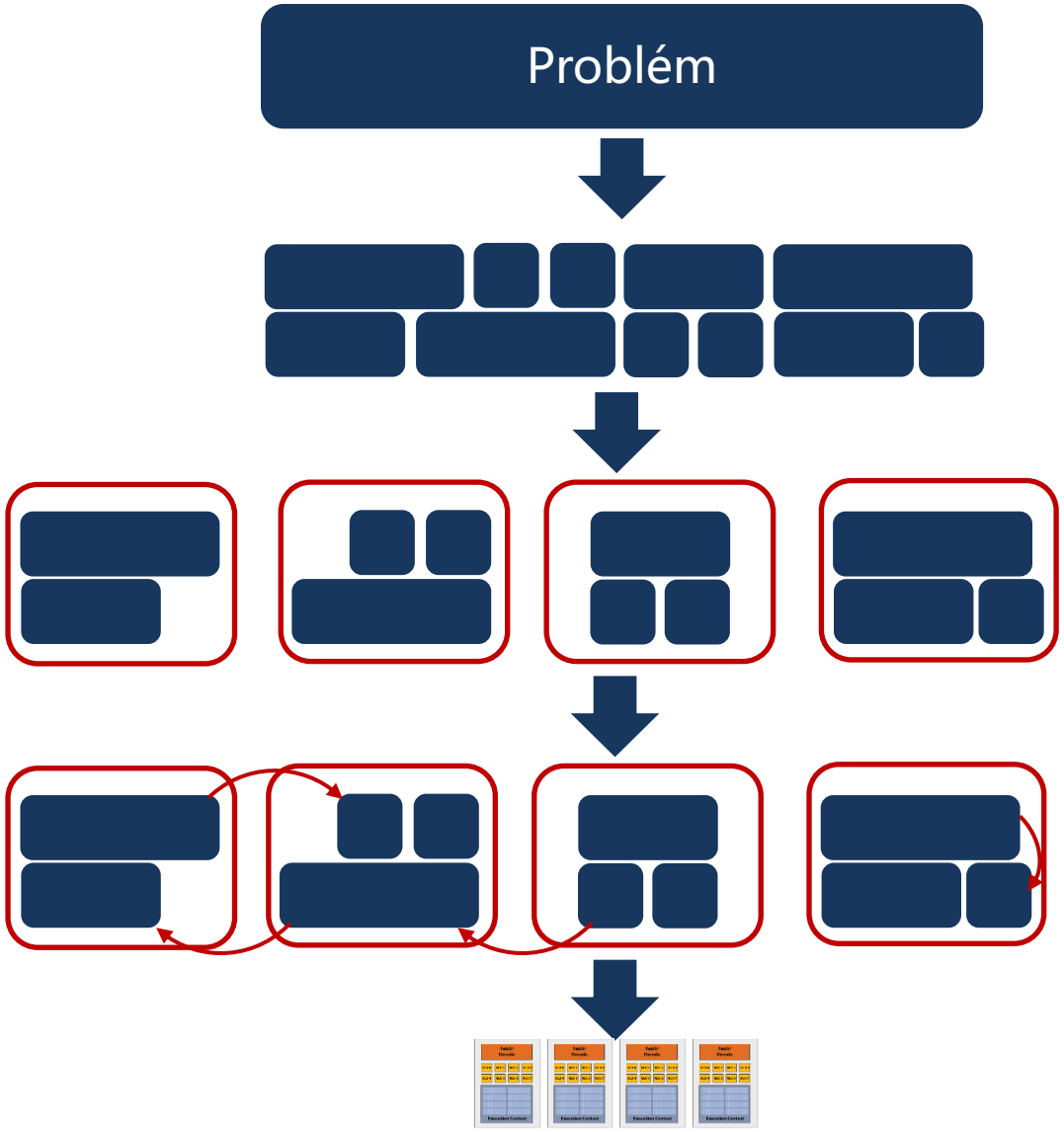


Dekompozice

Rozdělení vláknům

Paralelní programování

Náhled



Dekompozice

Rozdělení vláknům

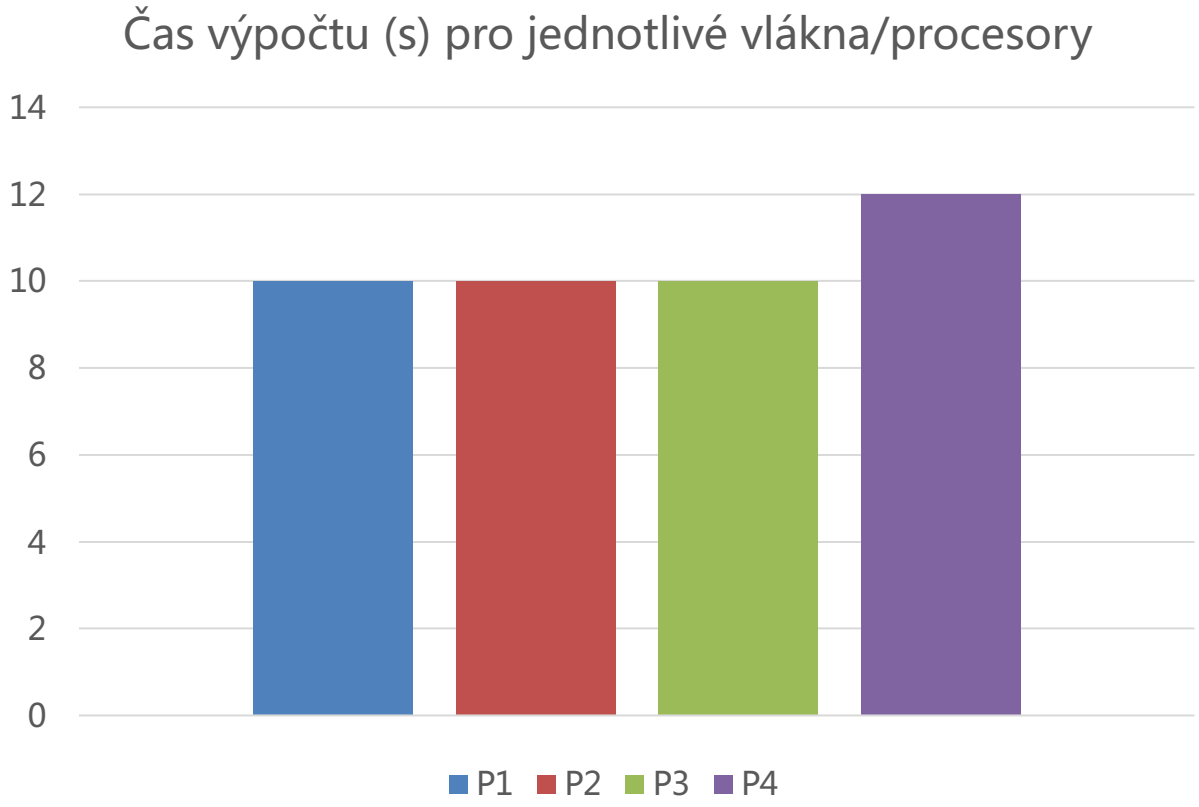
Orchestrace

Exekuce

Paralelní programování

Balancování

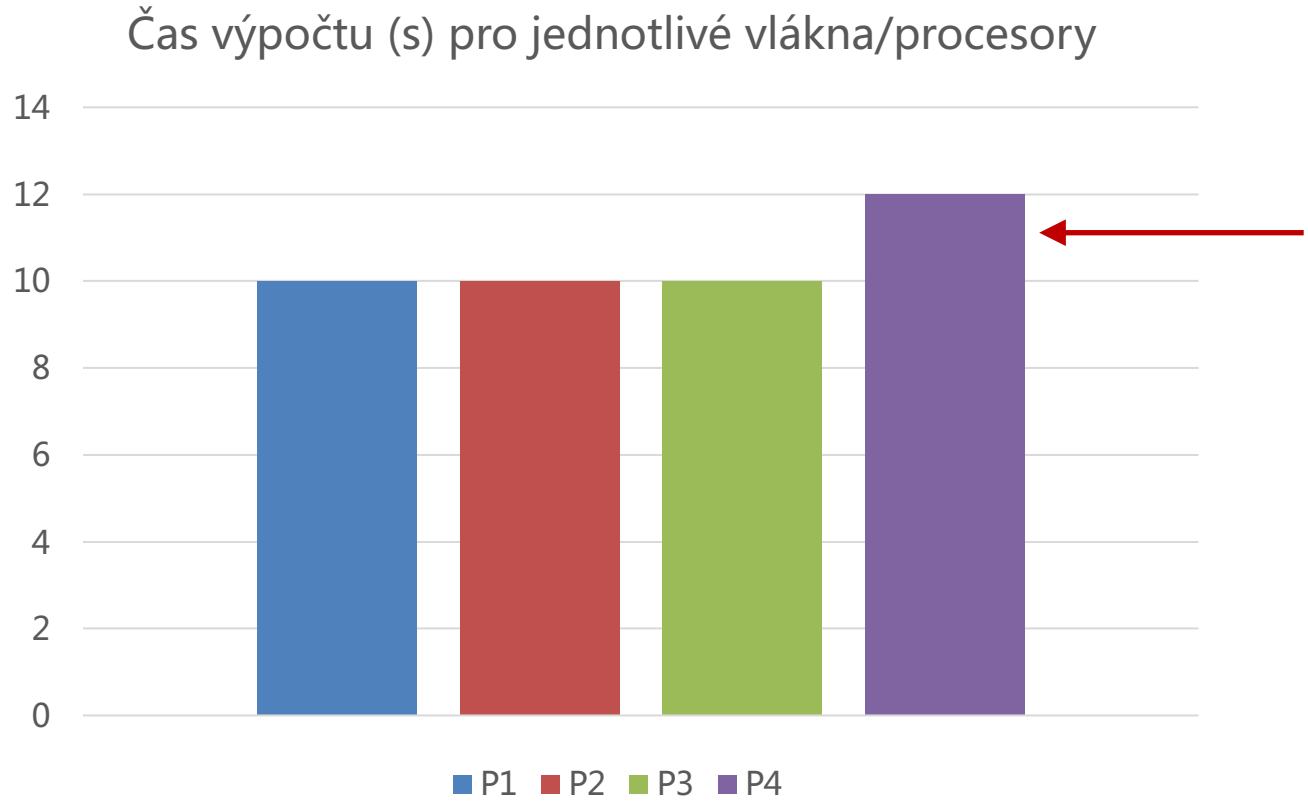
- Ideálně chceme, aby všechna vlákna/jádra pracovaly a skončily současně



Paralelní programování

Balancování

- Ideálně chceme, aby všechna vlákna/jádra pracovaly a skončily současně

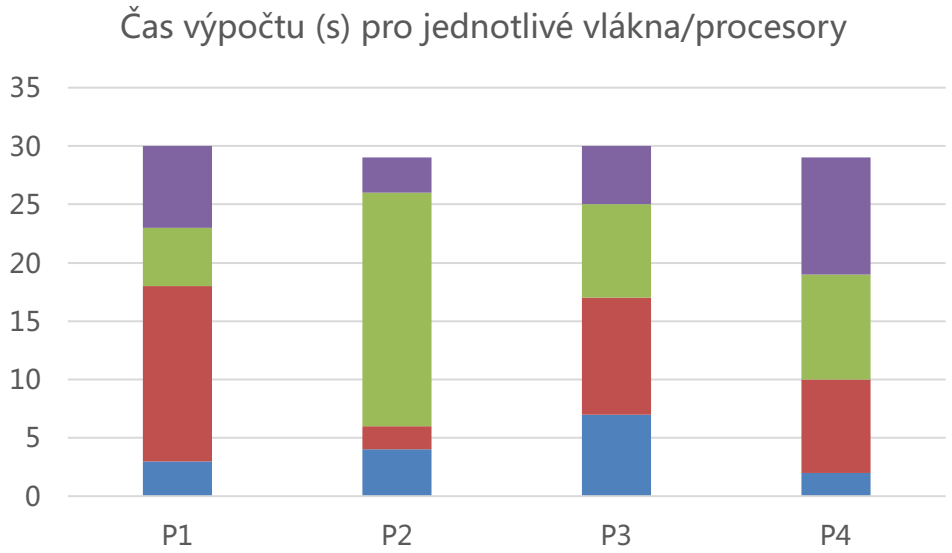


- Pokud 1 procesor pracuje o 20% déle, celý program pracuje o 20% déle
- Vzpomeňte si na Amdahlův zákon

Rozdělení práce

Statické rozdělení

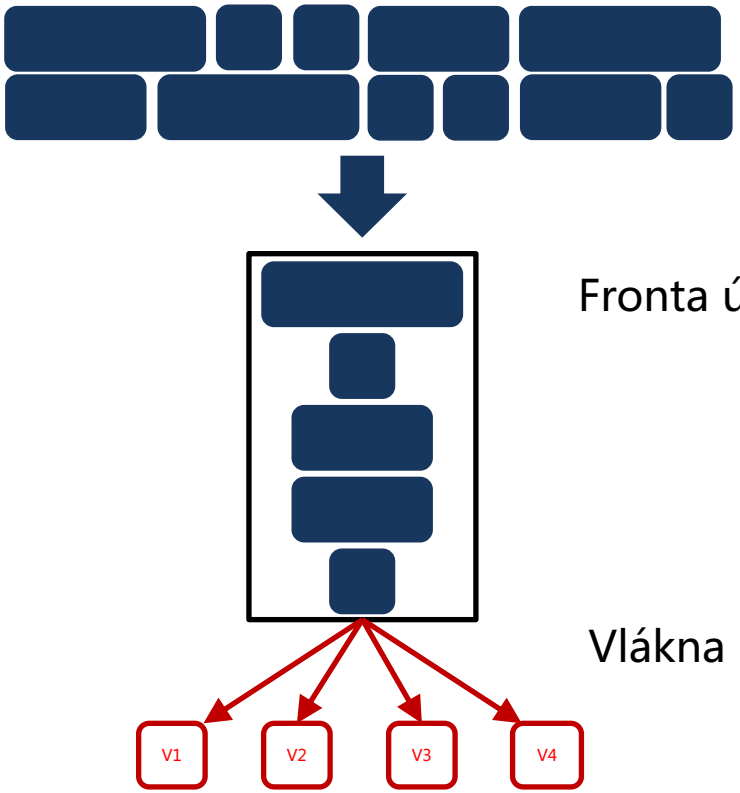
- Fixní a statické rozdělení úkolů pro jednotlivá vlákna
  - Ne nutně v době kompilace
  - Jednou přidělíme vláknům úkoly a toto přidělení je neměnné
- Kdy nám statické rozdělení pomůže?
  - Všechny úkoly trvají (přibližně) stejně dlouho
  - Každý úkol může trvat různě dlouho, ale víme předem očekávanou dobu trvání



Rozdělení práce

Dynamické rozdělení

- Program přiděluje úkoly dynamicky na základě aktuálního vytížení jednotlivých vláken
  - Threadpool a fronta úkolů



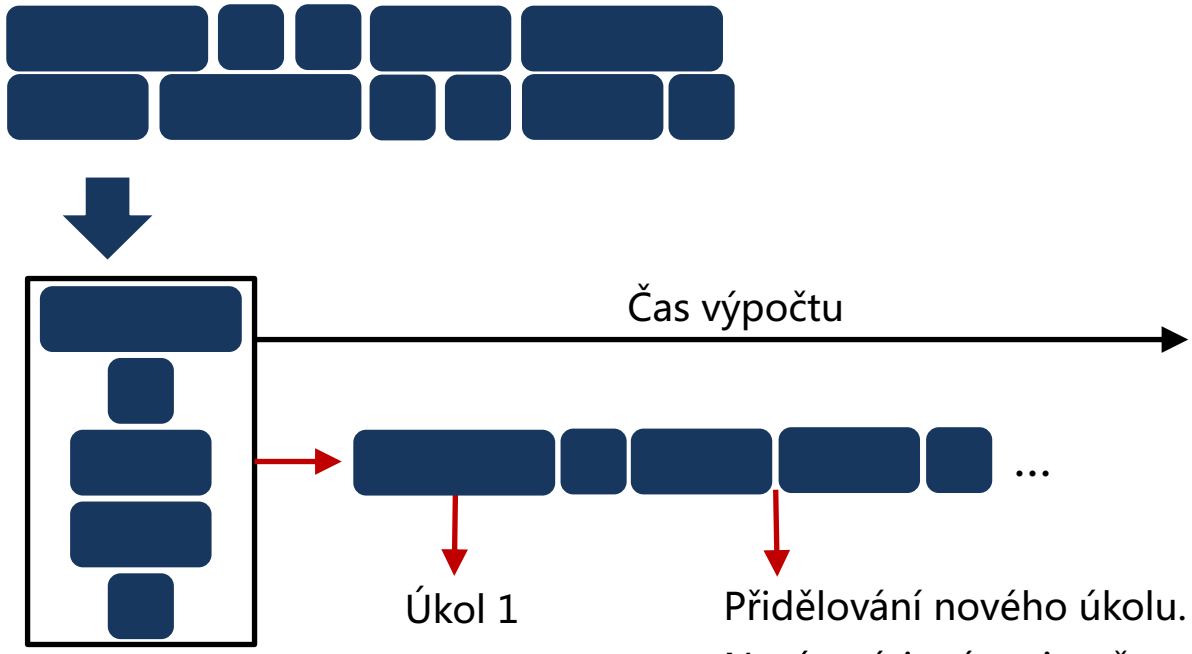
Fronta úkolů

Vlákna berou úkoly z fronty.

Rozdělení práce

Dynamické rozdělení

- Jak to bude vypadat z pohledu jednoho vlákna?
  - Threadpool a fronta úkolů



Přidělování nového úkolu.

Není v sériové variantě.

A je to exekuce v kritické sekci, která zpomaluje výpočet

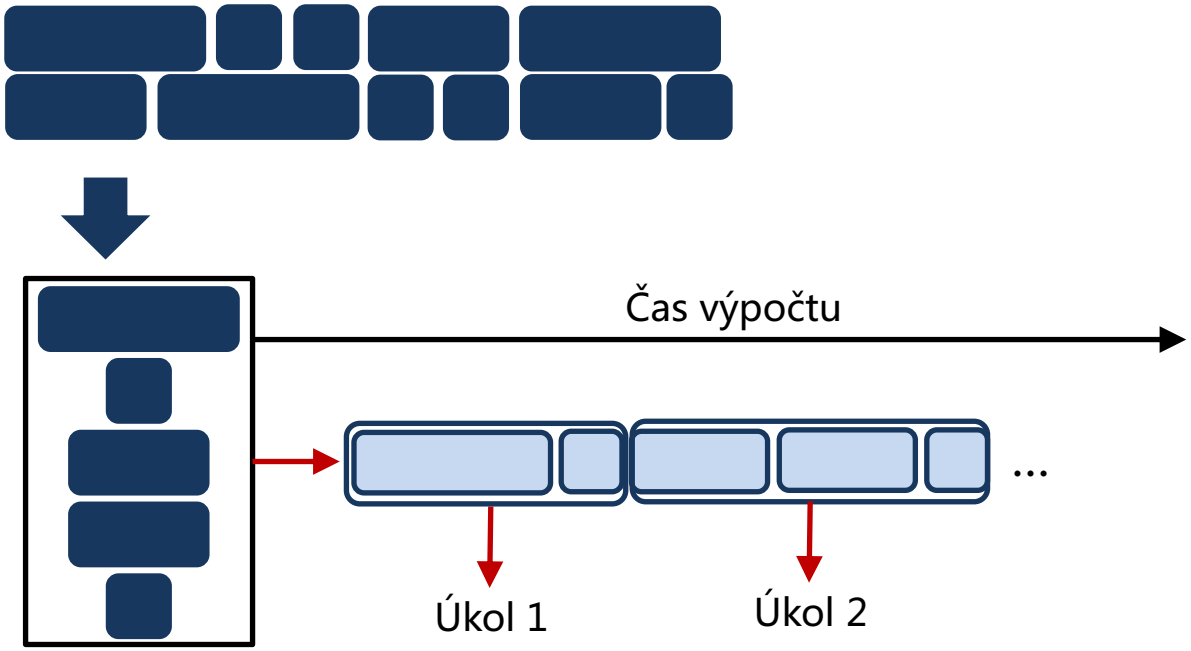
Více malých úkolů znamená dobré vybalancování mezi vlákna.

Více malých úkolů znamená více synchronizace a zpomalení.

Rozdělení práce

Dynamické rozdělení

- Můžeme měnit granularitu dekompozice



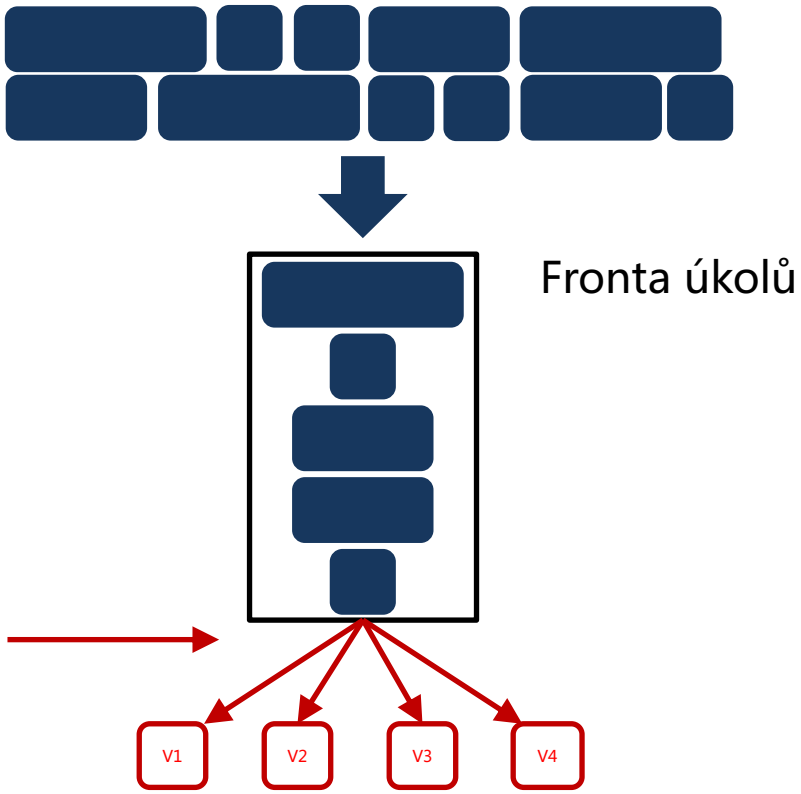
Zmenšení počtu úkolů sníží zpomalení kvůli synchronizaci

Ale můžeme mít problém s vybalancováním.

Rozdělení práce

Dynamické rozdělení – problémy

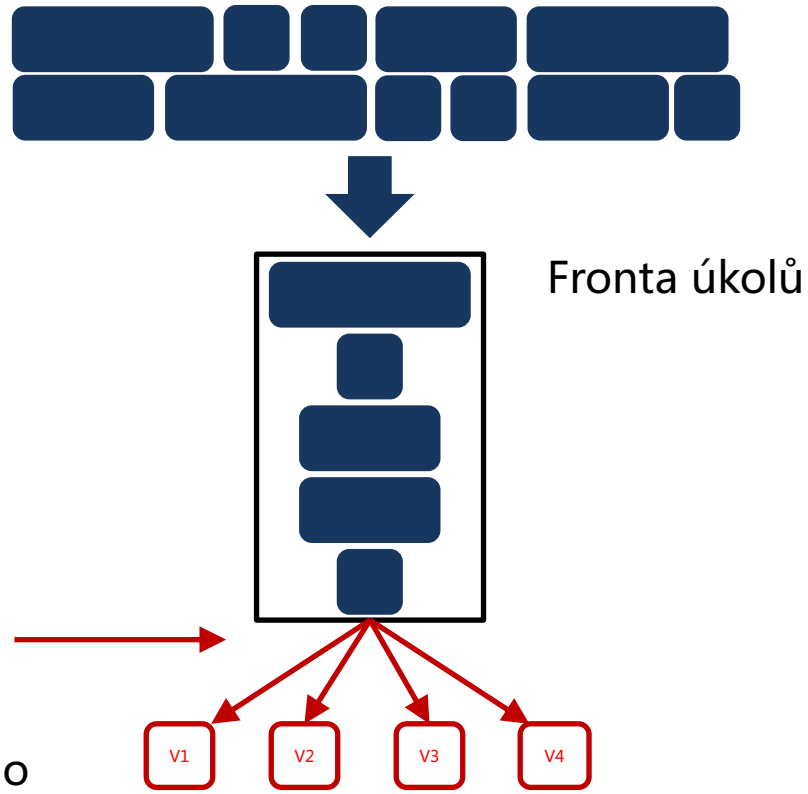
- Kde je kritická sekce?



Rozdělení práce

Dynamické rozdělení – problémy

- Kde je kritická sekce?

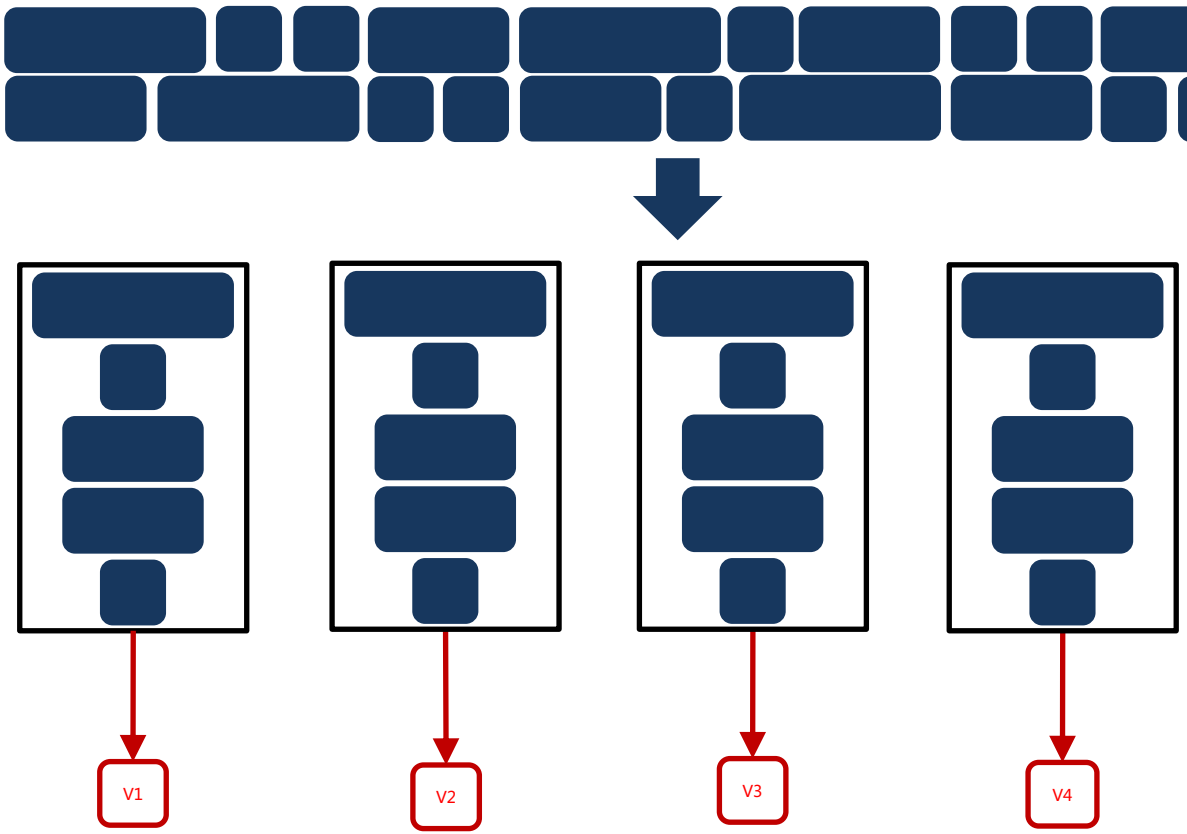


Všechna vlákna přistupují k jedné společné frontě.

Co kdyby mělo každé vlákno vlastní frontu?

Rozdělení práce

Dynamické rozdělení – vlastní fronty



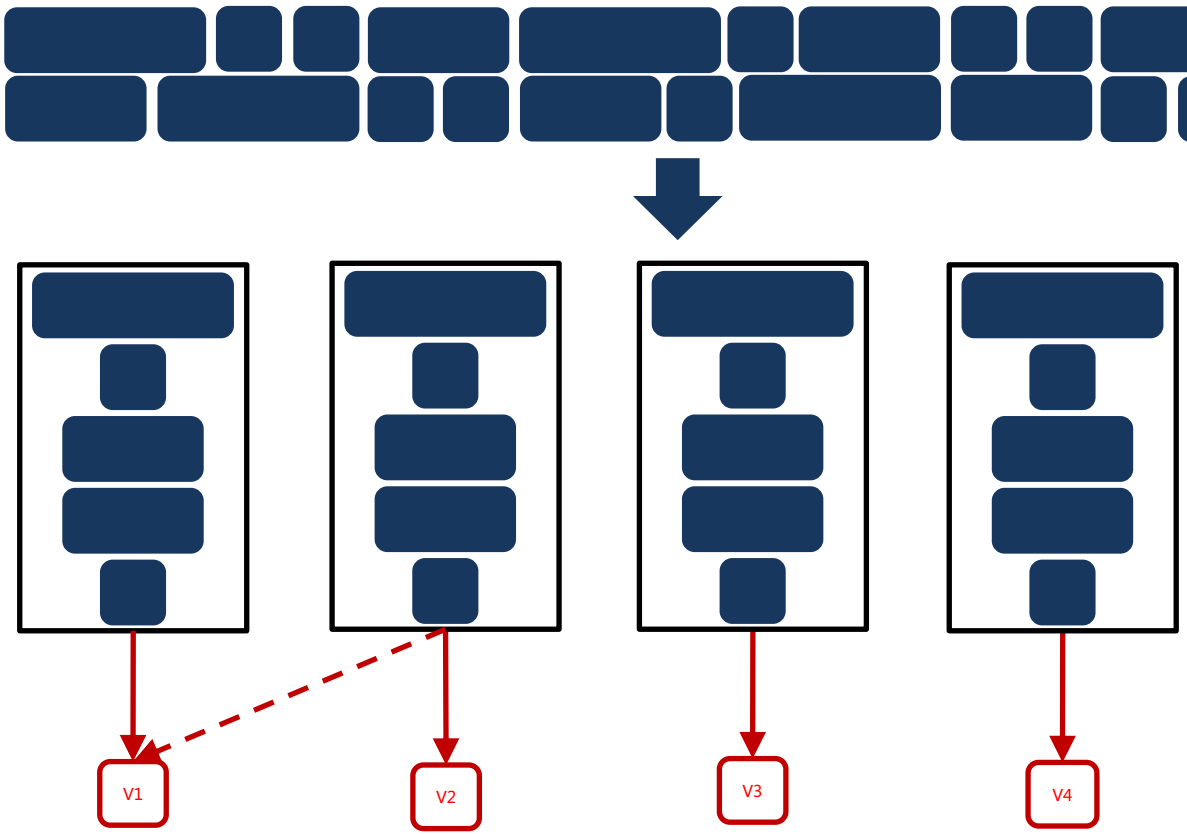
Můžeme vytvořit vlastní frontu pro každé vlákno

Vlákno vkládá a vybírá úkoly z vlastní fronty

Jak zabezpečíme vybalancování?

Rozdělení práce

Dynamické rozdělení – vlastní fronty



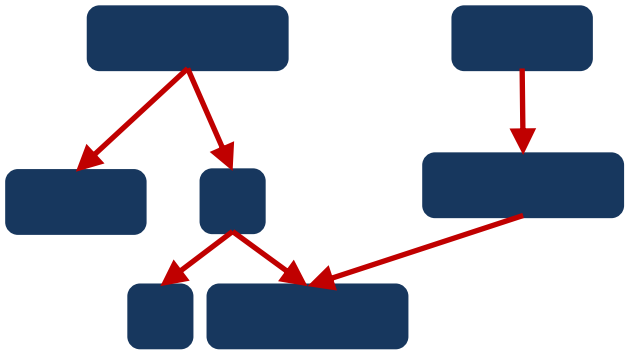
Jak zabezpečíme vybalancování?

Když má vlákno prázdnou frontu, může „ukradnout“ úkoly z jiné fronty.

Rozdělení práce

Dynamické rozdělení – závislosti

- Ne vždy je možné pustit libovolný úkol (např. pro spuštění úkolu X musíme znát aktuální hodnotu proměnné Y)
- Úkol bude zpracovaný vláknem/procesorem pouze v případě, že všechny závislosti jsou splněny



- V OpenMP např. pomocí
  - #pragma omp tasks depend([in/out/inout]:variables)

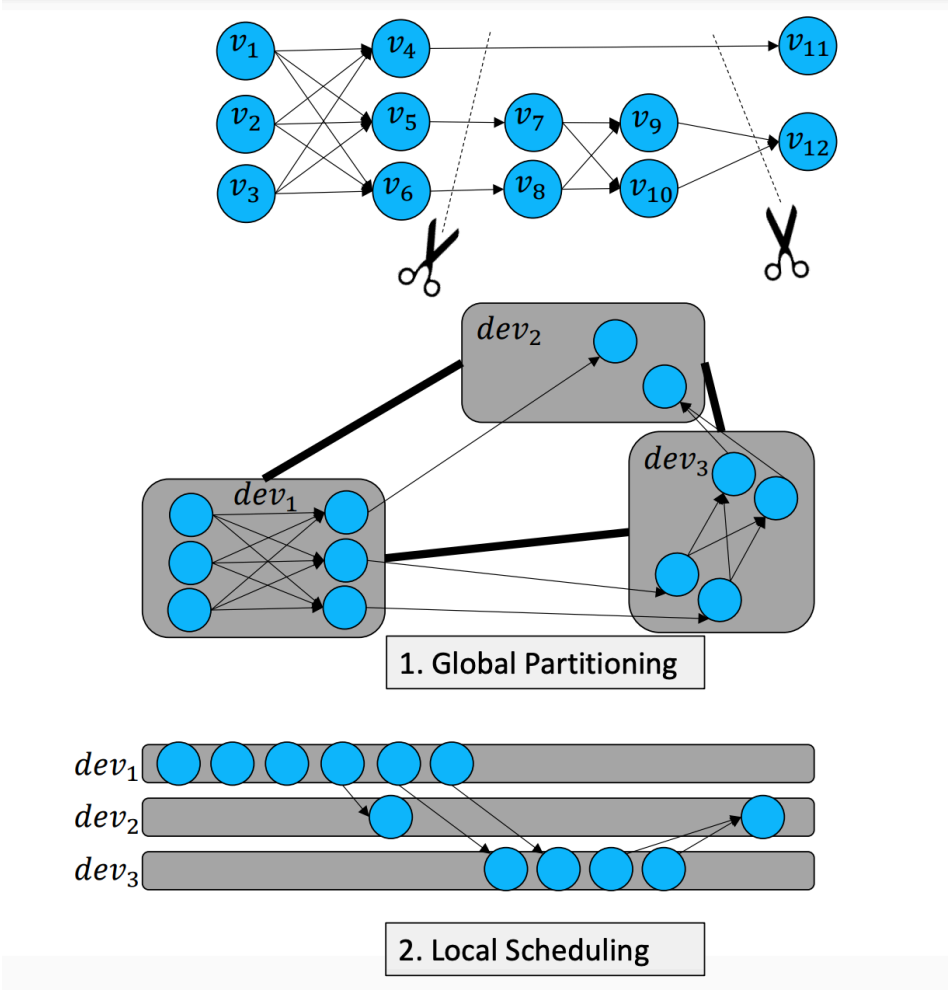
Rozdělení práce

Dynamické rozdělení

- Jak zvolit správnou velikost úkolu?
- Neexistuje univerzální odpověď – závisí na problému/HW (#CPU) atd.
- Pro mnoho kombinací (např. uniformní jádra a arbitrární graf závislostí, nad kterým se provádí výpočet) je nejlepší možné řešení NP-Těžké. Existuje ale řada dobrých heuristik.
- Pokud lze (máme odhad), můžeme přiřazovat dlouhé úkoly nejdřív a pak krátké úkoly.
- V nějakém smyslu asymptoticky optimální je pracovat s tzv. kritickou cestou.

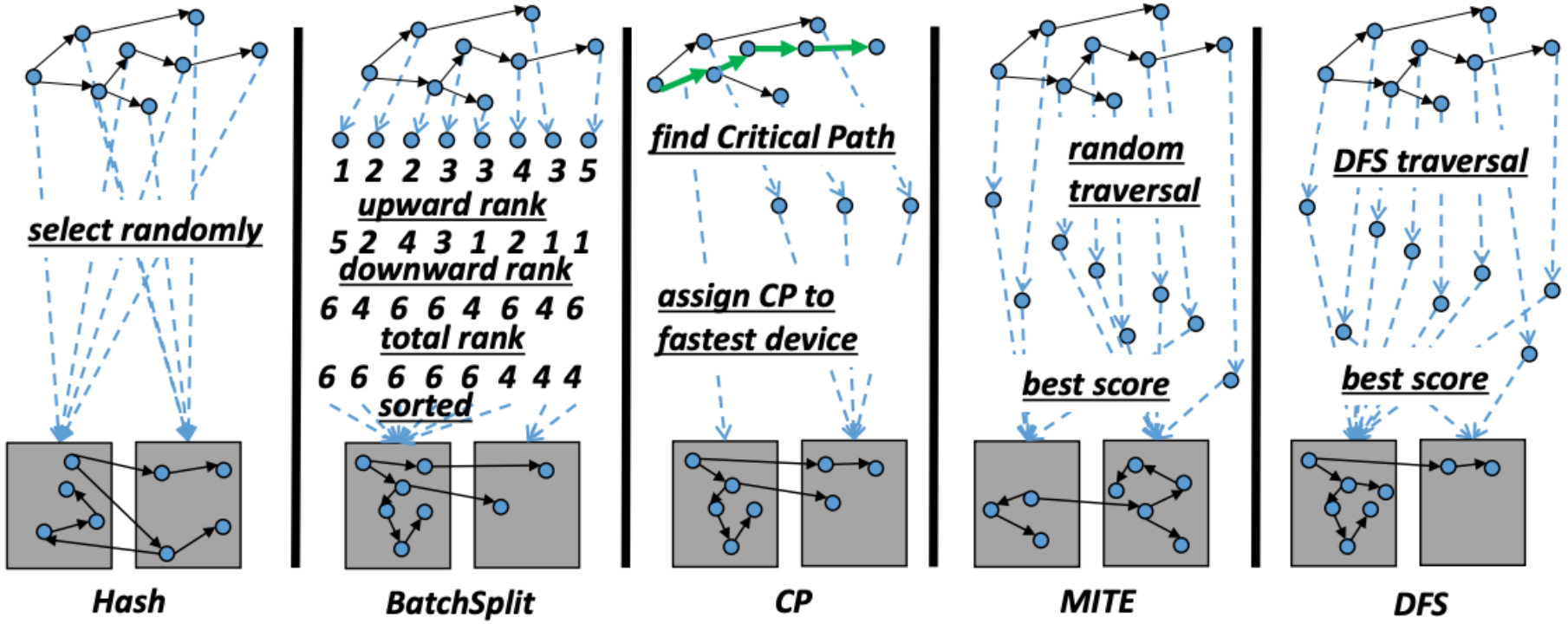
Rozdělení práce

Dynamické rozdělení



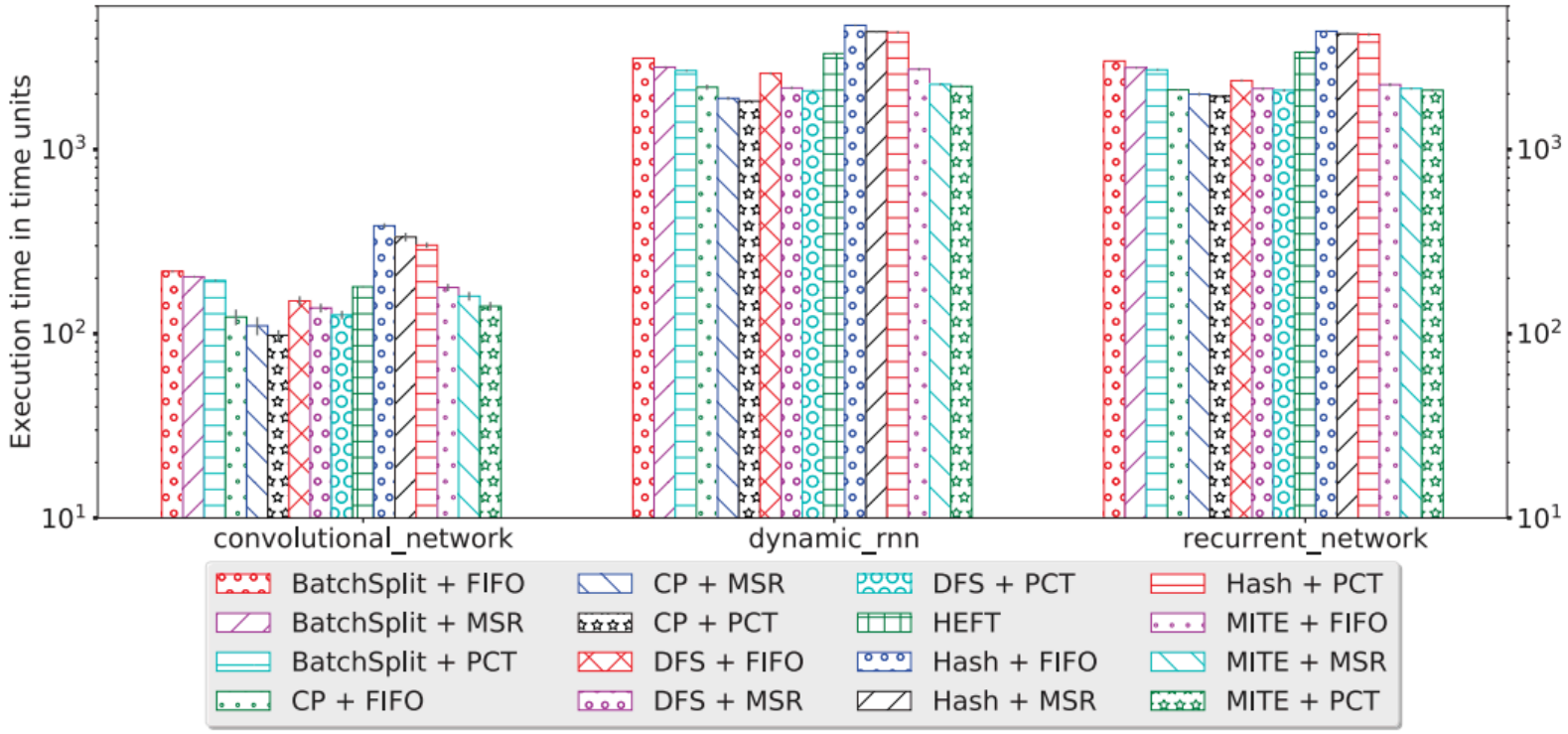
Rozdělení práce

Dynamické rozdělení



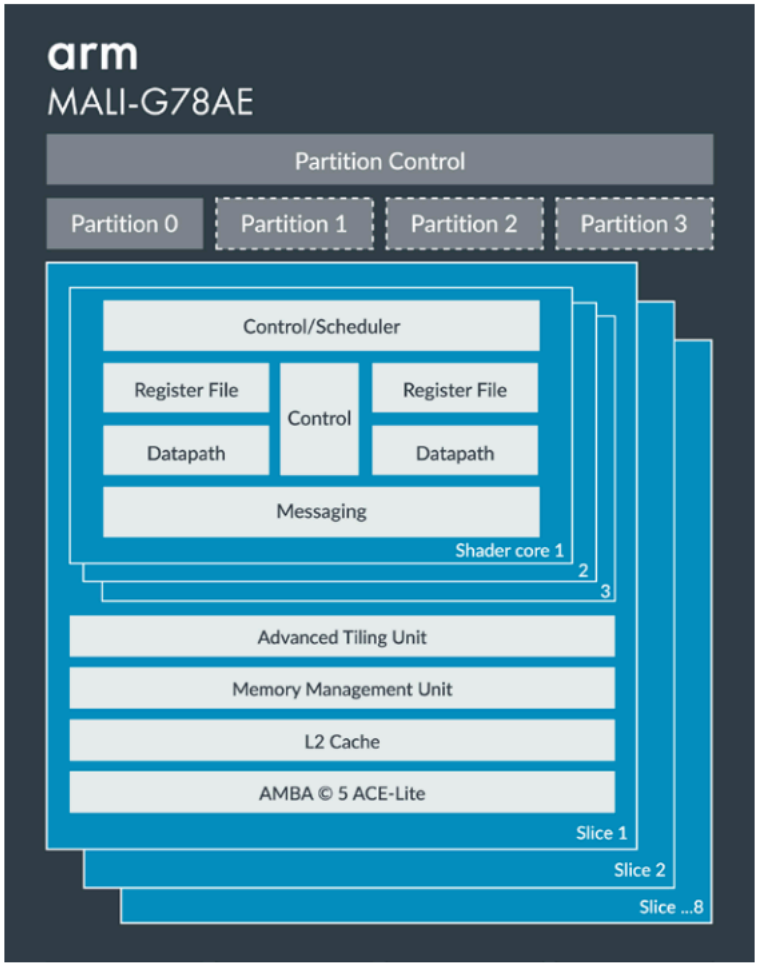
Rozdělení práce

Dynamické rozdělení



Vzpomínky starého zbrojnoše

Jak pracovat s takovou frontou, když jsou úloh miliony?



Dynamic Data Structures for Taskgraph Scheduling Policies with Applications in OpenCL Accelerators

Jakub Mareček · Andrew J. Parkes · Edmund K. Burke · Robert Elliot · Hedley Francis · Anton Lokhmotov

Abstract OpenCL is an emerging open framework for parallel programming in heterogenous systems. Devices compliant with OpenCL need to schedule the execution of submitted jobs with no (or only very imprecise) estimates of execution times, but respecting dependencies among them, which are given in the form of directed acyclic graph. This problem is known as stochastic taskgraph scheduling, stochastic scheduling with precedencies, or stochastic scheduling with data dependencies.

We study the complexity of implementing static out-of-order policies for taksgraph scheduling, which approach optimality in the long run, under certain assumptions. We present a simple data structure allowing for the "what next" query of such scheduling policies to be answered in time $O(1)$, while vertices can be added in time $O(1)$.

Rozdělení práce

Dynamické rozdělení – závislosti v OpenMP

```
int main(int argc, char* argv[]) {

   int x = 0;

#pragma omp parallel num_threads(thread_count) shared(x)
   {
#pragma omp single
      {
#pragma omp task depend(out:x)
         {
            std::this_thread::sleep_for(std::chrono::milliseconds(10));
            x++;
            std::cout << "1: x " << x << "\n";
         }
#pragma omp task depend(in:x)
         {
            x *= 3;
            std::cout << "2: x " << x << "\n";
         }

      }
   }
   std::cout << "final: x " << x << "\n";

   return 0;
}
```

Když definujeme „in “ závislost, vytvoří se závislosti, vytvoří se závislost úkolu na již generovaných úkolech, které mají pro danou proměnnou nastavenou dependenci „out “ případně „inout “.

Rozdělení práce

Hybridní přístupy

- Rozdělení nemusí být pouze statické nebo dynamické
- V podstatě je možné zvolit libovolný mezistupeň mezi dvěma extrémy
  - Rozdělím úkoly
  - Sbírám statistiky o délce zpracování
  - Přerozdělím úkoly a opakuji

Vzorce paralelizace

- Datový paralelismus
  - SIMD přístup
  - Rozdělím data a rovnou spustím zpracování pro jednotlivá vlákna
- Fork-join
  - Jedno vlákno zpracovává část úkolu
  - Identifikuje možné podúkoly a spustí nové vlákna/úkoly

Rozděluj a panuj

Quick Sort

- Základní třídící algoritmus
- Jak budeme paralelizovat?

```
void qs(std::vector<int>& vector_to_sort, int from, int to) {
   if (to - from <= base_size) {
      std::sort(vector_to_sort.begin() + from, vector_to_sort.begin() + to);
      return;
   }

   //rozdeleni dle pivota (vector_to_sort[from])
   int part2_start = partition(vector_to_sort,from,to,vector_to_sort[from]);

   if (part2_start - from > 1) {
         qs(vector_to_sort, from, part2_start);
   }
   if (to - part2_start > 1) {
         qs(vector_to_sort, part2_start, to);
   }
}
```

Rozděluj a panuj

Quick Sort

- Můžeme asynchronně volat rekurzivní úkoly

```
void qs(std::vector<int>& vector_to_sort, int from, int to) {
    if (to - from <= base_size) {
        std::sort(vector_to_sort.begin() + from, vector_to_sort.begin() + to);
        return;
    }

    //rozdeleni dle pivota (vector_to_sort[from])
    int part2_start = partition(vector_to_sort,from,to,vector_to_sort[from]);

    if (part2_start - from > 1) {
#pragma omp task shared(vector_to_sort) firstprivate(from,part2_start)
        {
            qs(vector_to_sort, from, part2_start);
        }
    }
    if (to - part2_start > 1) {
            qs(vector_to_sort, part2_start, to);
    }
}
```

Rozděluj a panuj

Quick Sort

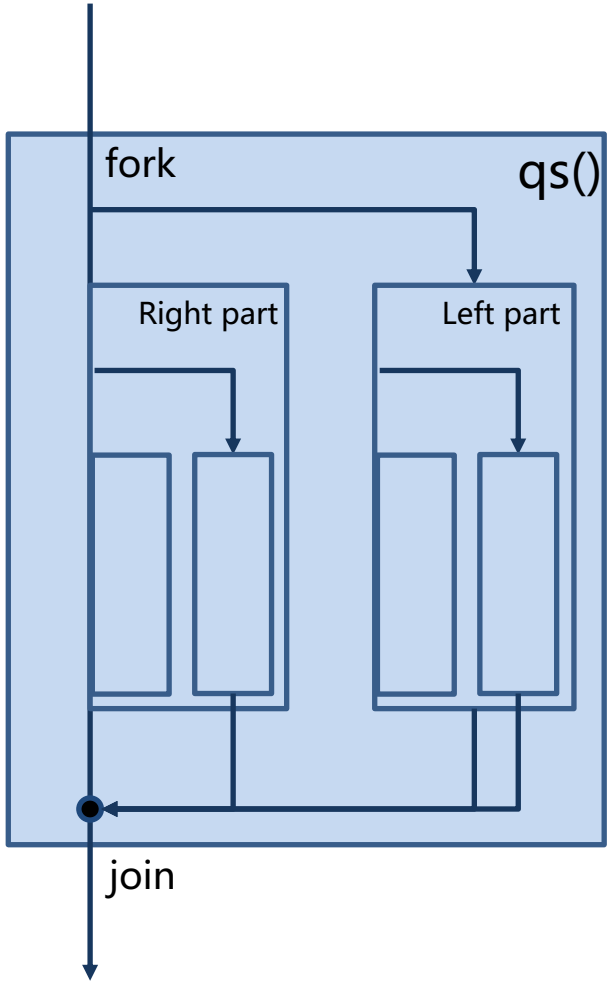
- Omezíme minimální velikost, aby nedocházelo k false-sharingu
- Můžeme asynchronně volat rekurzivní úkoly

```
void qs(std::vector<int>& vector_to_sort, int from, int to) {
    if (to - from <= base_size) {
        std::sort(vector_to_sort.begin() + from, vector_to_sort.begin() + to);
        return;
    }

    //rozdeleni dle pivota (vector_to_sort[from])
    int part2_start = partition(vector_to_sort,from,to,vector_to_sort[from]);

    if (part2_start - from > 1) {
#pragma omp task shared(vector_to_sort) firstprivate(from,part2_start)
        {
            qs(vector_to_sort, from, part2_start);
        }
    }
    if (to - part2_start > 1) {
            qs(vector_to_sort, part2_start, to);
    }
}
```

Rozděluj a panuj

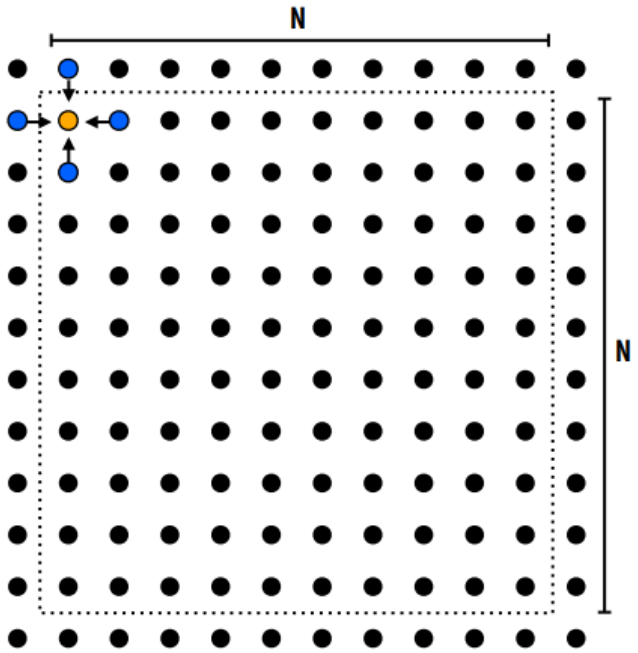
Quick Sort

```
void qs(std::vector<int>& vector_to_sort, int from, int to) {
    if (to - from <= base_size) {
        std::sort(vector_to_sort.begin() + from, vector_to_sort.begin() + to);
        return;
    }

    //rozdeleni dle pivota (vector_to_sort[from])
    int part2_start = partition(vector_to_sort,from,to,vector_to_sort[from]);

    if (part2_start - from > 1) {
#pragma omp task shared(vector_to_sort) firstprivate(from,part2_start)
        {
            qs(vector_to_sort, from, part2_start);
        }
    }
    if (to - part2_start > 1) {
            qs(vector_to_sort, part2_start, to);
    }
}
```



Dekompozice se závislostmi

- paralelizace QuickSortu byla snadná vzhledem k žádné závislosti mezi úkoly
- Co když jsou úkoly závislé?



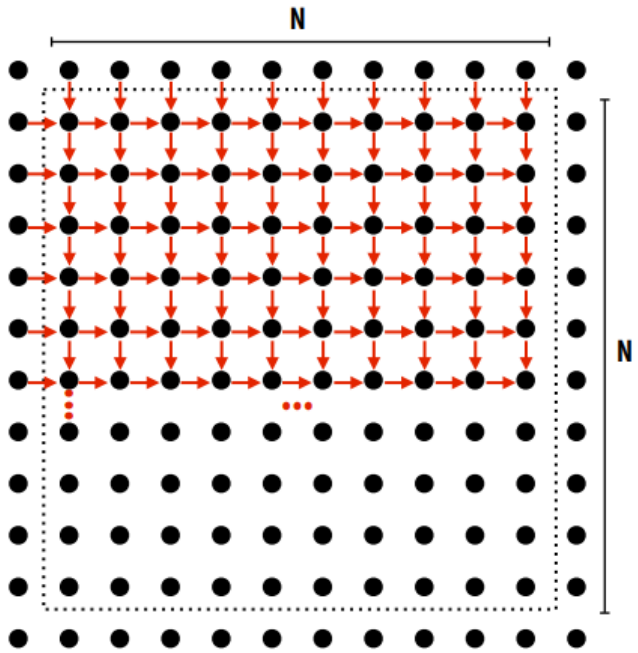
Problém:

- Chceme iterativně počítat průměr pro každé pole mřížky
  - $A[i,j] = 0.2 * (A[i-1,j] + A[i,j-1] + A[i,j] + A[i+1,j] + A[i,j+1])$
- Každou iteraci chceme projít celou matici z horního levého rohu
- Jaké jsou zde závislosti?

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

Dekompozice se závislostmi

- paralelizace QuickSortu byla snadná vzhledem k žádné závislosti mezi úkoly
- Co když jsou úkoly závislé?



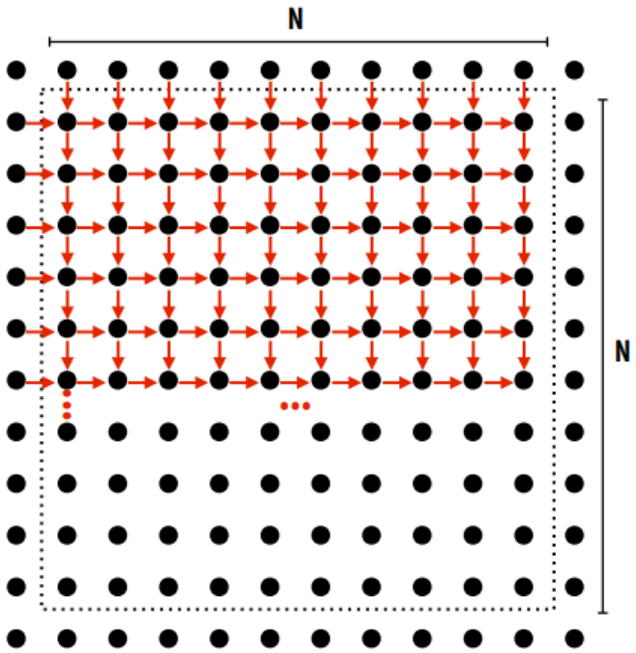
Problém:

- Chceme iterativně počítat průměr pro každé pole mřížky
  - $A[i,j] = 0.2 * (A[i-1,j] + A[i,j-1] + A[i,j] + A[i+1,j] + A[i,j+1])$
- Každou iteraci chceme projít celou matici z horního levého rohu
- Jaké jsou zde závislosti?

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

Dekompozice se závislostmi

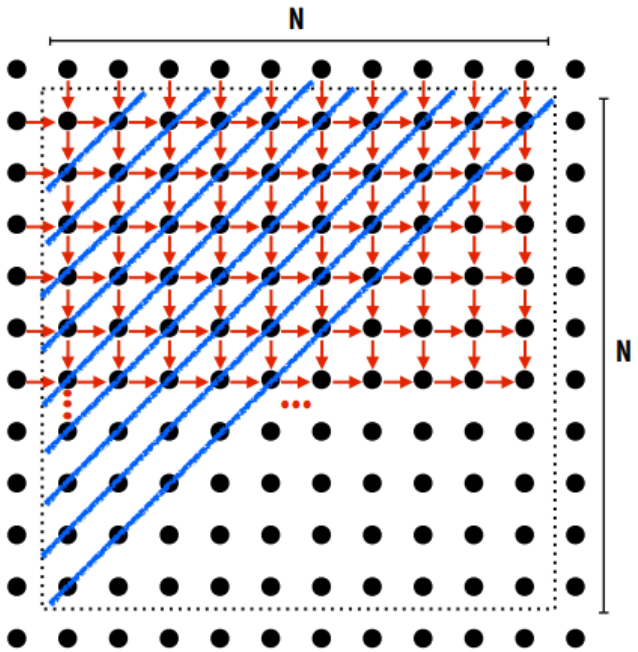
- Jakým způsobem můžeme tento problém paralelizovat?
- Zkusíme nalézt nezávislé úkoly
  - Které uzly lze aktualizovat paralelně?



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

Dekompozice se závislostmi

- Jakým způsobem můžeme tento problém paralelizovat?
- Zkusíme nalézt nezávislé úkoly
  - Které uzly lze aktualizovat paralelně?



Uzly na diagonále jsou nezávislé (mohou přistupovat ke stejné proměnné, ale pouze pro čtení).

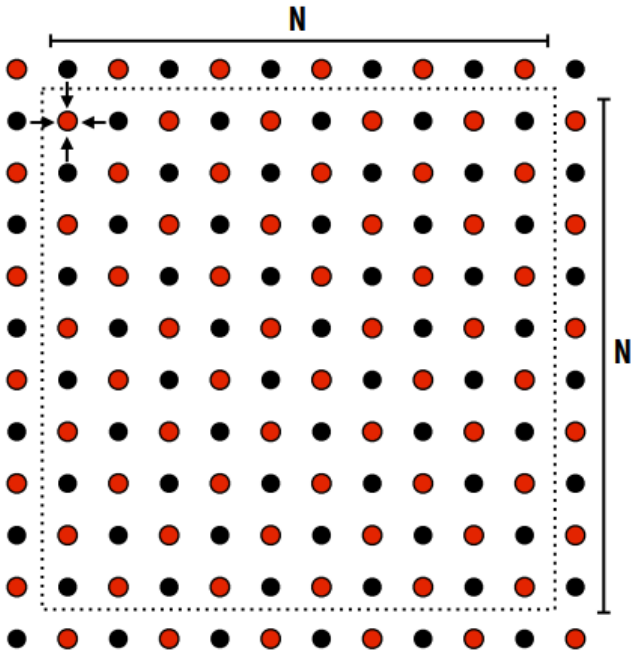
Problematické rozdělení na vlákna/procesory.

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

Dekompozice se závislostmi

- Existuje lepší způsob?

Pro konvergenci nemusíme nutně postupovat sekvenčně z jednoho rohu – uzly rozdělíme do dvou skupin a aktualizujeme nejdřív jednu, pak druhou



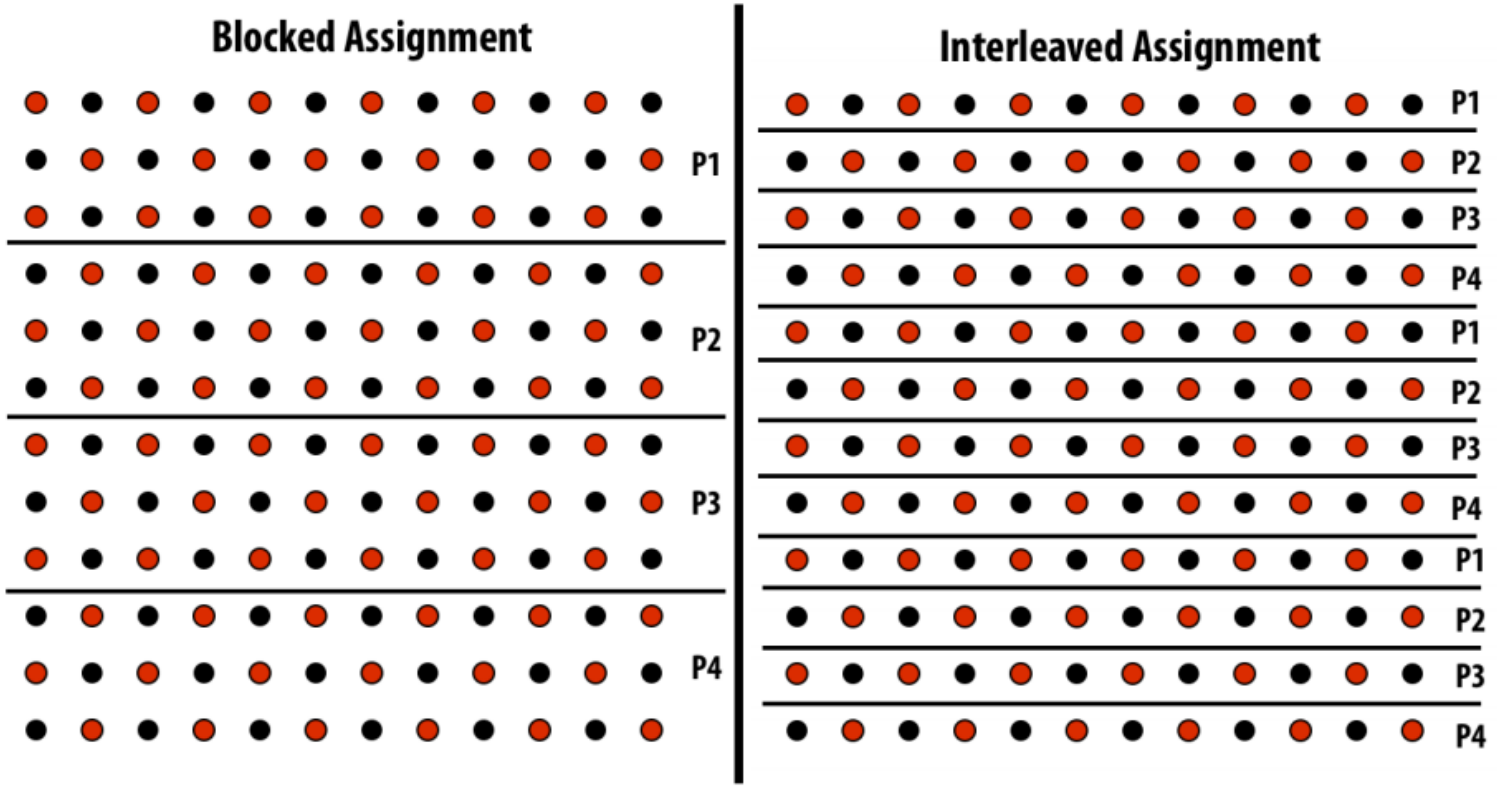
Jednoduchá paralelizace a rozdělení úkolu vláknům.

Musíme vědět, že si to můžeme dovolit (znalost problému/domény).

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

Dekompozice se závislostmi

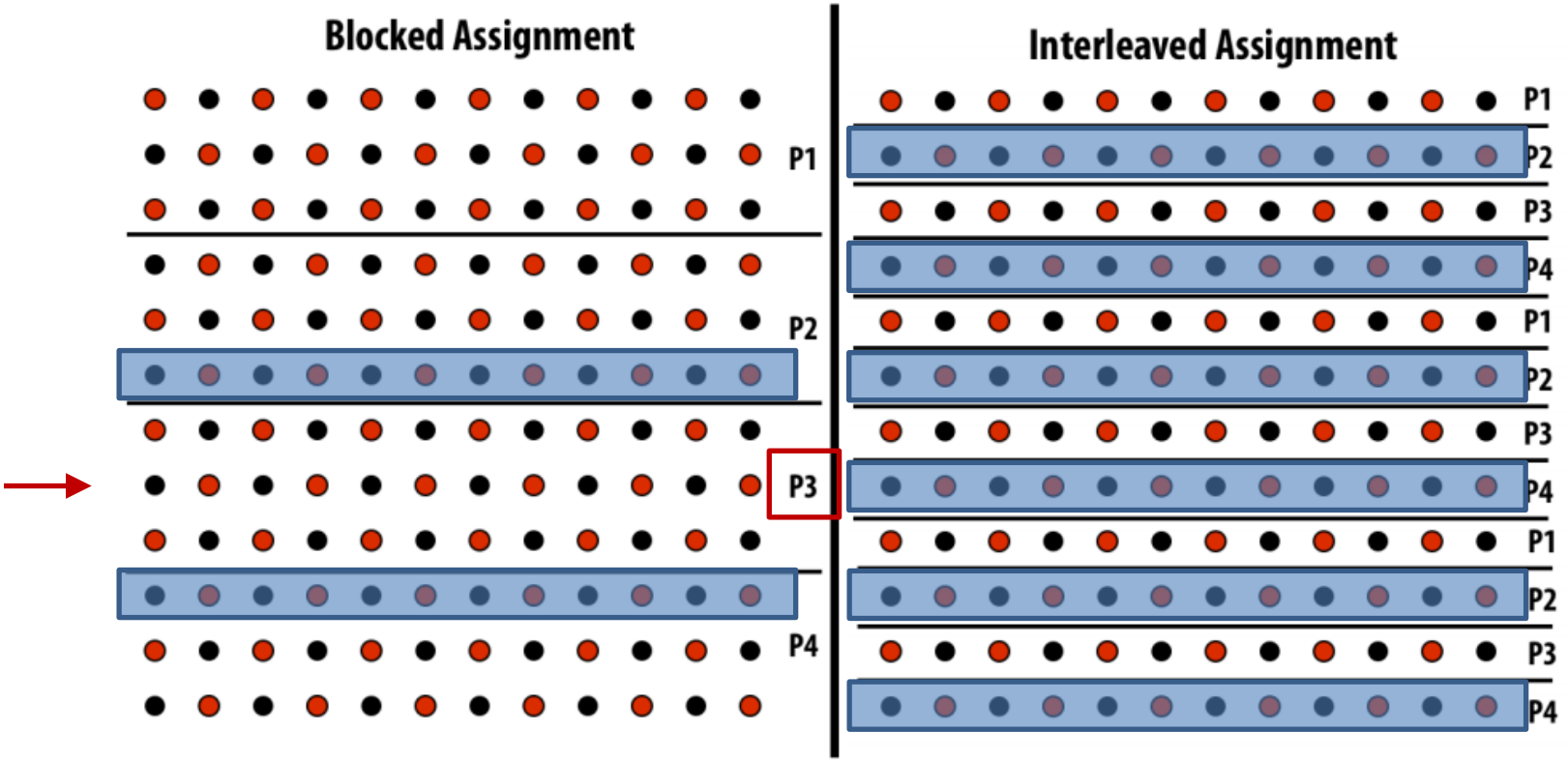
- Existuje lepší způsob?
- Jak můžeme rozdělit na úkoly pro vlákna/procesory?



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

Dekompozice se závislostmi

- Který je lepší?
- Které části jsou privátní a které sdílené?



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

Hledání prvočísel

Eratostenovo síto

- Problém: Chceme zjistit počet prvočísel mezi 0 a X (např. 10^9)
- Jaký je sériový algoritmus?

Hledání prvočísel

Eratostenovo síto

- Problém: Chceme zjistit počet prvočísel mezi 0 a X (např. 10^9)
- Jaký je sériový algoritmus?

```
long result = 0;

   for (int i = 2; i < MAXSQRT; i++) {
      if (primes[i] == 1) {
         for (int j = i * i; j < MAXNUMBER; j += i) {
            primes[j] = 0;
         }
      }
   }
   for (int i = 0; i<MAXNUMBER; i++)
      result += primes[i];

   return result;
```

Jak na to?

Hledání prvočísel

Eratostenovo síto

- Zkusíme paralelizovat hlavní for cyklus
- Můžeme paralelizovat druhý for cyklus pro součet

```
long result = 0;

#pragma omp parallel num_threads(thread_count)
   {
#pragma omp for schedule(static)
      for (int i = 2; i < MAXSQRT; i++) {
         if (primes[i] == 1) {
            for (int j = i * i; j < MAXNUMBER; j += i) {
               primes[j] = 0;
            }
         }
      }
   }

#pragma omp parallel for reduction(+:result)
   for (int i = 0; i<MAXNUMBER; i++)
      result += primes[i];

   return result;
```

Jak nám to bude fungovat?

Hledání prvočísel

Eratostenovo síto

```
long result = 0;

#pragma omp parallel num_threads(thread_count)
   {
#pragma omp for schedule(static)
      for (int i = 2; i < MAXSQRT; i++) {
         if (primes[i] == 1) {
            for (int j = i * i; j < MAXNUMBER; j += i) {
               primes[j] = 0;
            }
         }
      }
   }

#pragma omp parallel for reduction(+:result)
   for (int i = 0; i<MAXNUMBER; i++)
      result += primes[i];

   return result;
```

Pro $X=10^9$

Sériová varianta	První paralelizace (4 vlákna)
11.7188 s	13.0681 s

Hledání prvočísel

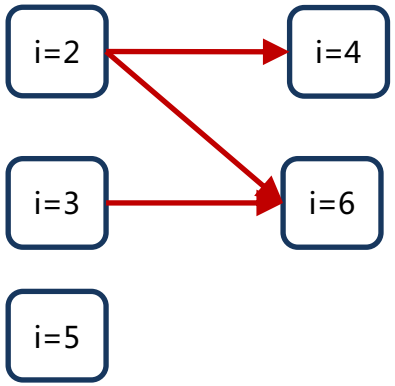
Eratostenovo síto

- Co se stane když paralelizujeme hlavní cyklus?
  - Např. vlákno 0 bude zpracovávat iteraci i=2, vlákno 2 bude zpracovávat iteraci i=4
  - Vlákno 2 dělá úplně zbytečnou práci – informace o tom, že číslo 4 není prvočíslo se k němu nemusí dostat včas

Hledání prvočísel

Eratostenovo síto

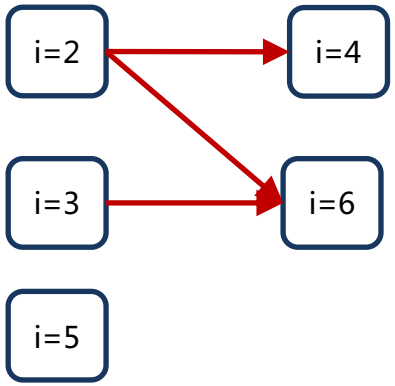
- Co se stane když paralelizujeme hlavní cyklus?
  - Např. vlákno 0 bude zpracovávat iteraci i=2, vlákno 2 bude zpracovávat iteraci i=4
  - Vlákno 2 dělá úplně zbytečnou práci – informace o tom, že číslo 4 není prvočíslo se k němu nemusí dostat včas
- Jaká je závislost mezi úkoly?



Hledání prvočísel

Eratostenovo síto

- Co se stane když paralelizujeme hlavní cyklus?
  - Např. vlákno 0 bude zpracovávat iteraci i=2, vlákno 2 bude zpracovávat iteraci i=4
  - Vlákno 2 dělá úplně zbytečnou práci – informace o tom, že číslo 4 není prvočíslo se k němu nemusí dostat včas
- Jaká je závislost mezi úkoly?

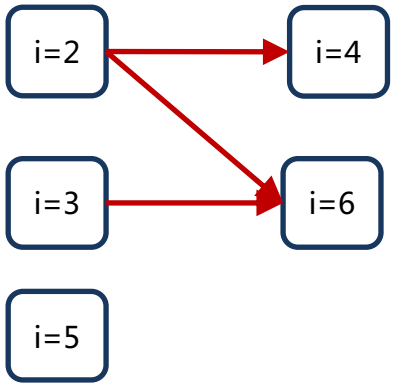


Na to abychom identifikovaly správné pořadí musíme vyřešit vlastní problém

Hledání prvočísel

Eratostenovo síto

- Co se stane když paralelizujeme hlavní cyklus?
  - Např. vlákno 0 bude zpracovávat iteraci i=2, vlákno 2 bude zpracovávat iteraci i=4
  - Vlákno 2 dělá úplně zbytečnou práci – informace o tom, že číslo 4 není prvočíslo se k němu nemusí dostat včas
- Jaká je závislost mezi úkoly?

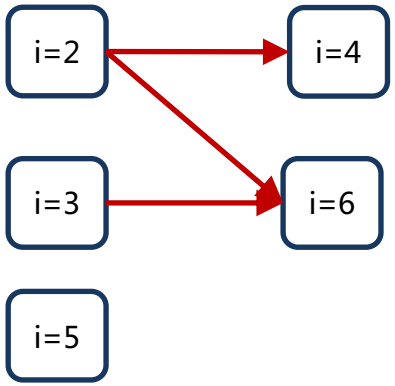


Na to abychom identifikovaly správné pořadí musíme vyřešit vlastní problém

Hledání prvočísel

Eratostenovo síto

- Co se stane když paralelizujeme hlavní cyklus?
  - Např. vlákno 0 bude zpracovávat iteraci i=2, vlákno 2 bude zpracovávat iteraci i=4
  - Vlákno 2 dělá úplně zbytečnou práci – informace o tom, že číslo 4 není prvočíslo se k němu nemusí dostat včas
- Jaká je závislost mezi úkoly?

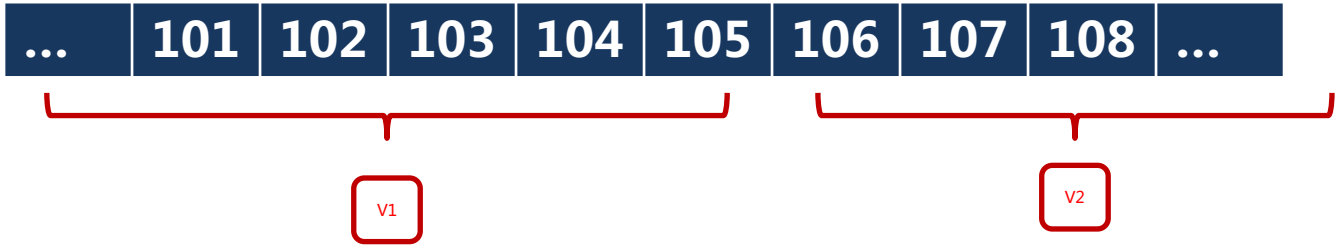


Na to abychom identifikovaly správné pořadí musíme vyřešit vlastní problém

Hledání prvočísel

Eratostenovo síto

- Jak můžeme snížit závislost?
  - Každé vlákno může kontrolovat pouze podinterval čísel

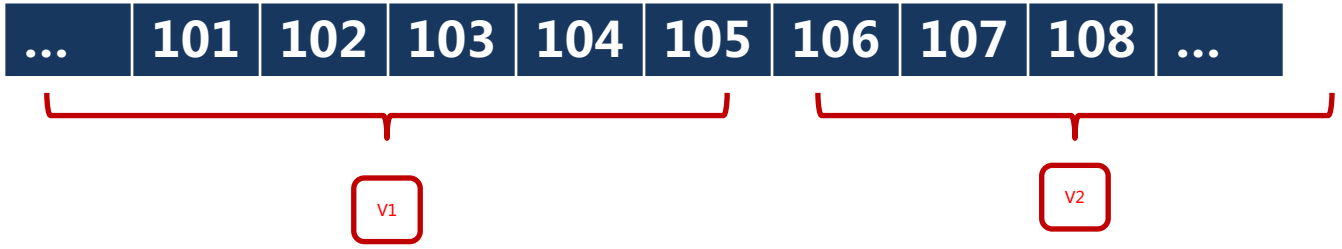


Úkol pro vlákno: označit čísla, které nejsou prvočísly v daném podintervalu

Hledání prvočísel

Eratostenovo síto

- Jak můžeme snížit závislost?
  - Každé vlákno může kontrolovat pouze podinterval čísel

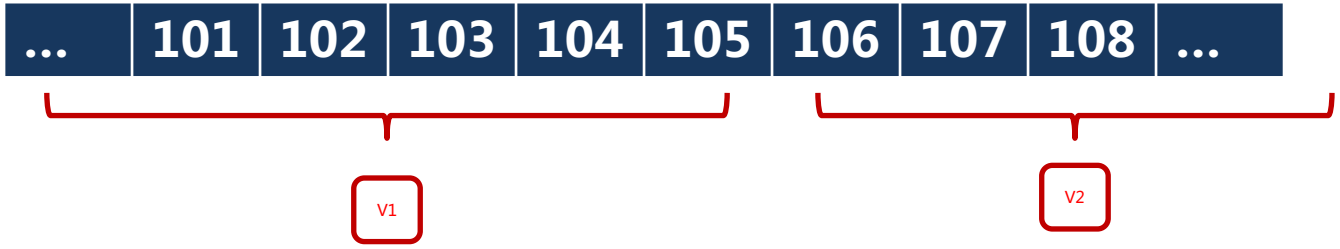


Úkol pro vlákno: označit čísla, které nejsou prvočísly v daném podintervalu

Hledání prvočísel

Eratostenovo síto

- Jak můžeme snížit závislost?
  - Každé vlákno může kontrolovat pouze podinterval čísel



Úkol pro vlákno: označit čísla, které nejsou prvočísly v daném podintervalu

...

Hledání prvočísel

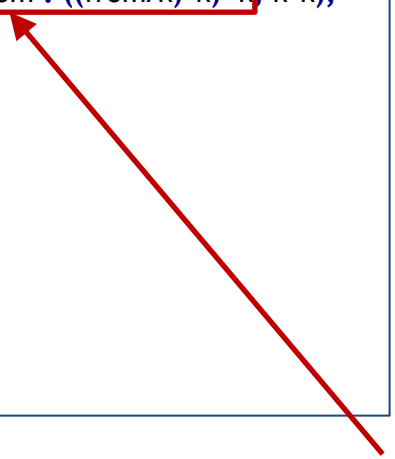
Eratostenovo síto

```
long sieve() {
   int step = MAXNUMBER/thread_count/500;
   long result = 0;

#pragma omp parallel num_threads(thread_count) reduction(+:result)
   {
#pragma omp for schedule(static)
      for (int i = 2; i < MAXNUMBER; i += step) {
         int from = i;
         int to = (i + step < MAXNUMBER) ? i + step : MAXNUMBER;

         for (int k = 2; k < MAXSQRT; k++) {
            if (primes[k] == 1) {
               int start = std::max((from % k == 0) ? from : ((from/k)*k)+k, k*k);
               for (int j = start; j < to; j += k) {
                  primes[j] = 0;
               }
            }
         }
         for (int k = from; k < to; k++)
            result += primes[k];
      }
   }
   return result;
}
```

Hledání prvočísel

Eratostenovo síto

```
long sieve() {
   int step = MAXNUMBER/thread_count/500;
   long result = 0;

#pragma omp parallel num_threads(thread_count) reduction(+:result)
   {
#pragma omp for schedule(static)
      for (int i = 2; i < MAXNUMBER; i += step) {
         int from = i;
         int to = (i + step < MAXNUMBER) ? i + step : MAXNUMBER;

         for (int k = 2; k < MAXSQRT; k++) {
            if (primes[k] == 1) {
               int start = std::max((from % k == 0) ? from : ((from/k)*k)+k, k*k);
               for (int j = start; j < to; j += k) {
                  primes[j] = 0;
               }
            }
         }
         for (int k = from; k < to; k++)
            result += primes[k];
      }
   }
   return result;
}
```



první násobek **k** v intervalu [from,to]

Hledání prvočísel

Eratostenovo síto

```
long sieve() {
    int step = MAXNUMBER/thread_count/500;
    long result = 0;

#pragma omp parallel num_threads(thread_count) reduction(+:result)
    {
#pragma omp for schedule(static)
        for (int i = 2; i < MAXNUMBER; i += step) {
            int from = i;
            int to = (i + step < MAXNUMBER) ? i + step : MAXNUMBER;

            for (int k = 2; k < MAXSQRT; k++) {
                if (primes[k] == 1) {
                    int start = std::max((from % k == 0) ? from : ((from/k)*k)+k, k*k);
                    for (int j = start; j < to; j += k) {
                        primes[j] = 0;
                    }
                }
            }
            for (int k = from; k < to; k++)
                result += primes[k];
        }
    }
    return result;
}
```

nulujeme od druhé mocniny **k**

Hledání prvočísel

Eratostenovo síto

```
long sieve() {
   int step = MAXNUMBER/thread_count/500;
   long result = 0;

#pragma omp parallel num_threads(thread_count) reduction(+:result)
   {
#pragma omp for schedule(static)
      for (int i = 2; i < MAXNUMBER; i += step) {
         int from = i;
         int to = (i + step < MAXNUMBER) ? i + step : MAXNUMBER;

         for (int k = 2; k < MAXSQRT; k++) {
            if (primes[k] == 1) {
               int start = std::max((from % k == 0) ? from : ((from/k)*k)+k, k*k);
               for (int j = start; j < to; j += k) {
                  primes[j] = 0;
               }
            }
         }
         for (int k = from; k < to; k++)
            result += primes[k];
      }
   }
   return result;
}
```

Pro $X=10^9$

1 vlákno	paralelizace (4 vlákna)
3.99 s	1.74 s