Paralelní a distribuované výpočty (B4B36PDV)

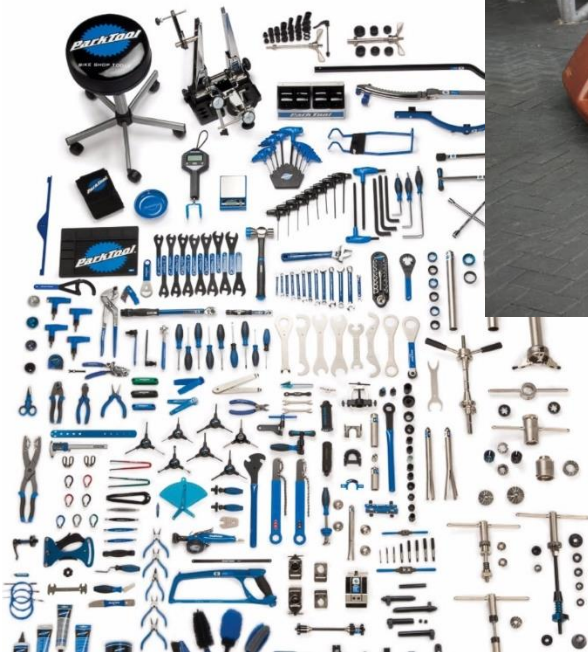
Branislav Bošanský, Michal Jakob

bosansky@fel.cvut.cz

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

Dnešní přednáška

Motivace



Dnešní přednáška

Techniky paralelizace

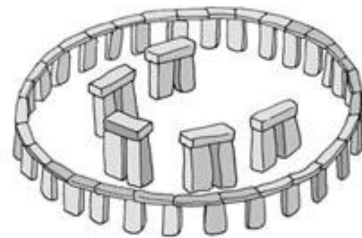
Chci paralelizovat algoritmus XY

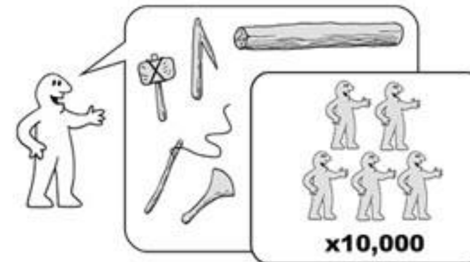
Jak na to?

Dnešní přednáška

Postup – Jak na to?

HËNJ





80x

30x

30x

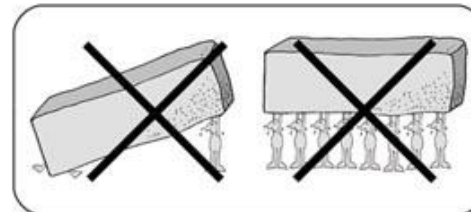
10x

5x

1x

3x

1



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

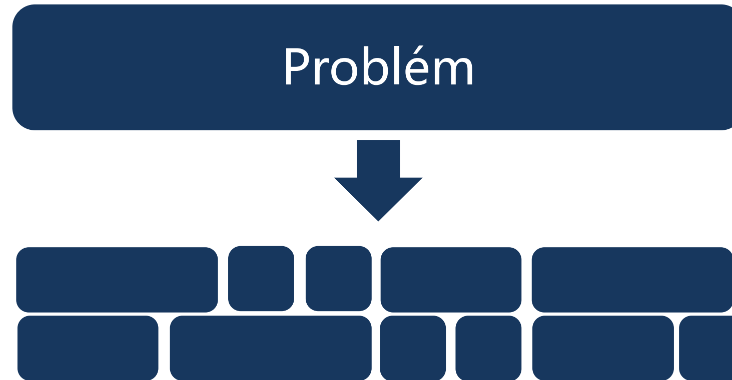
Paralelní programování

Náhled

Problém

Paralelní programování

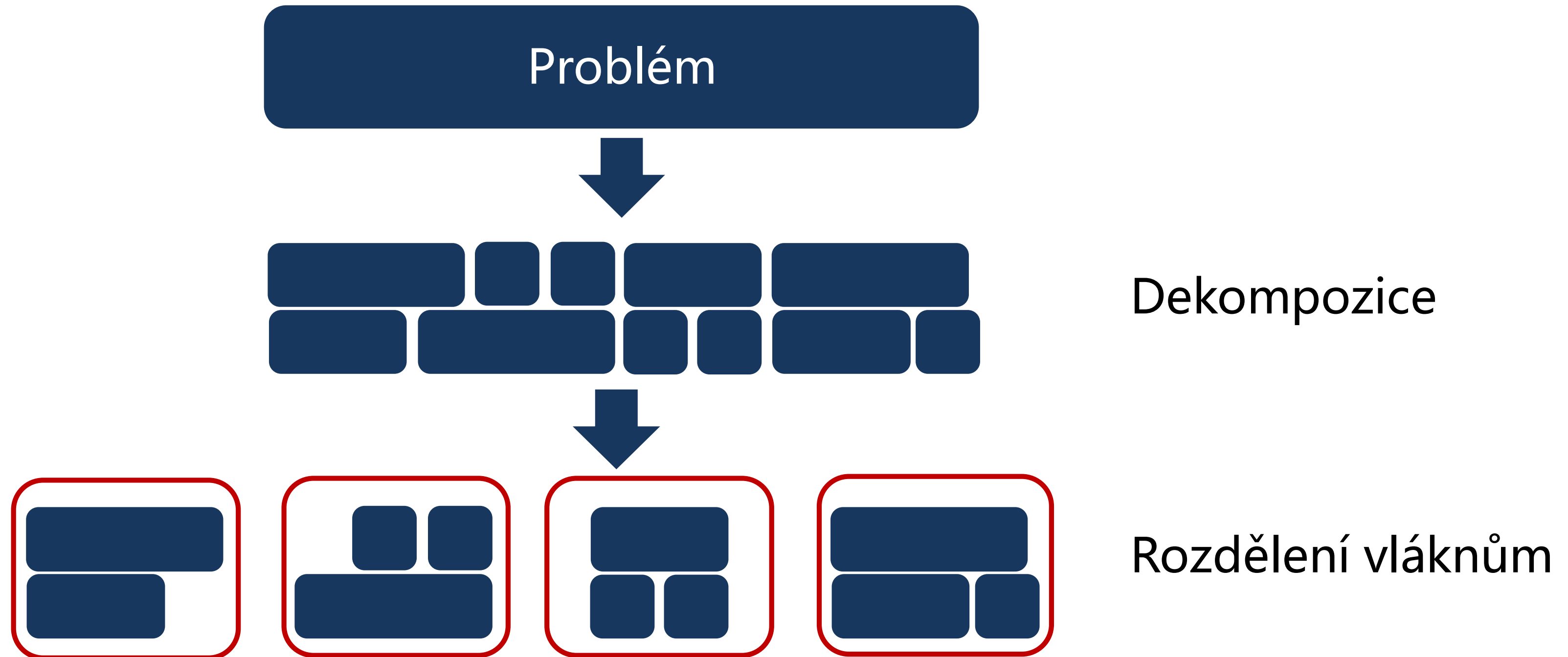
Náhled



Dekompozice

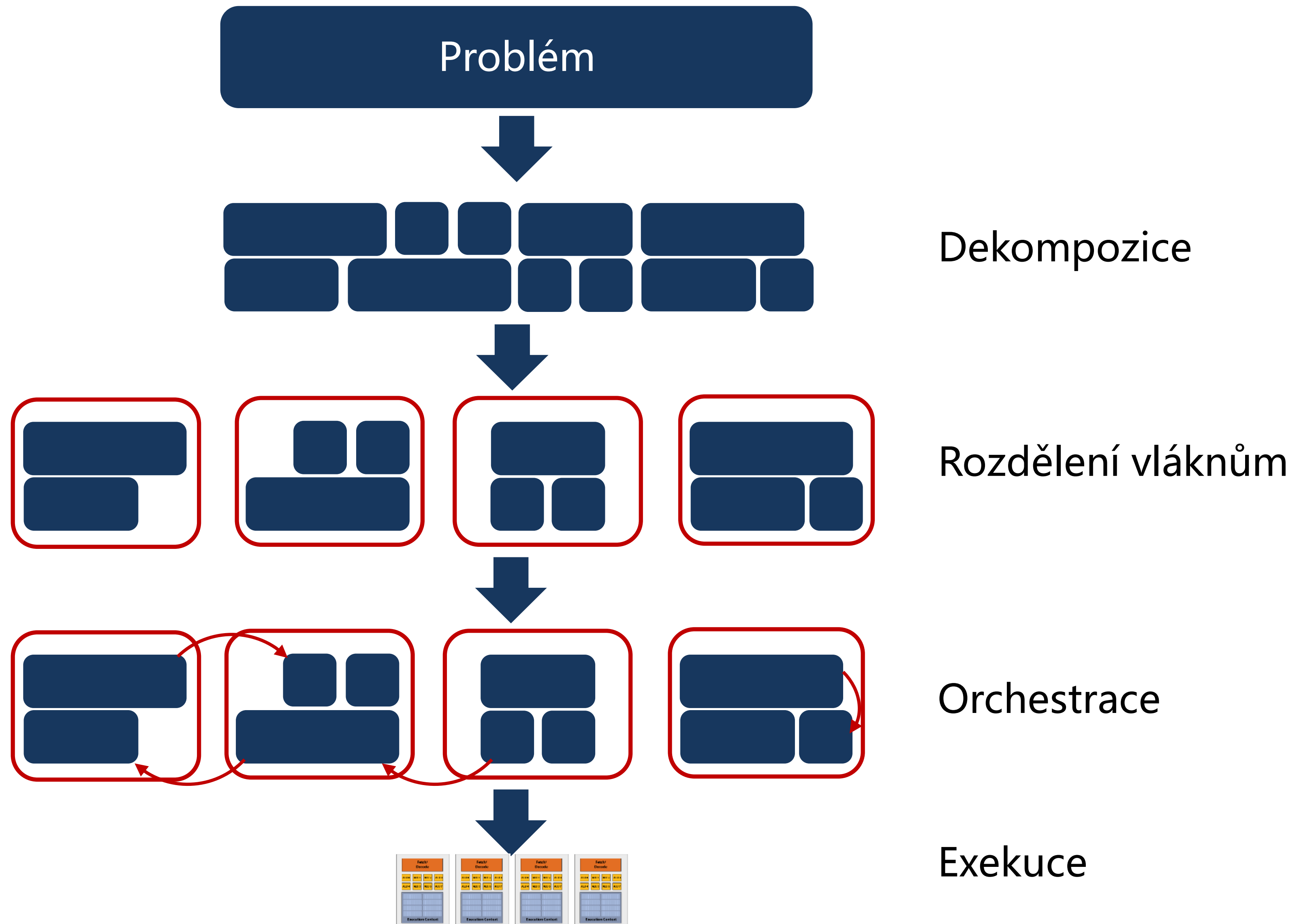
Paralelní programování

Náhled



Paralelní programování

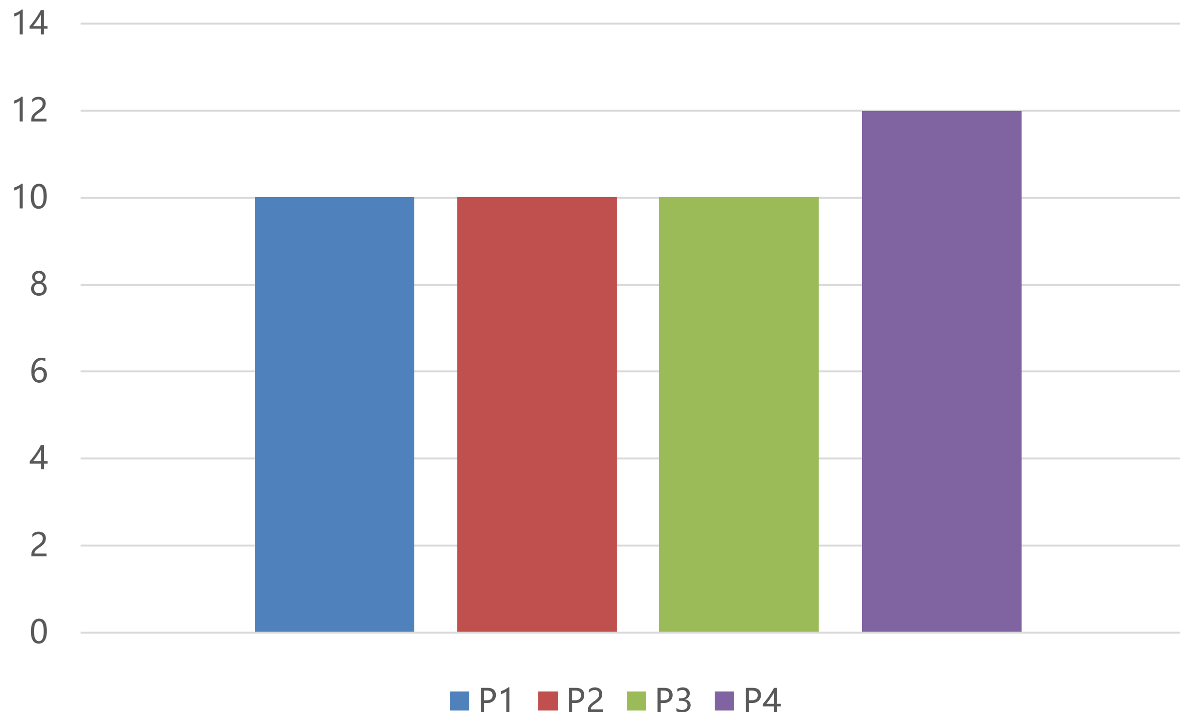
Náhled



Paralelní programování

Balancování

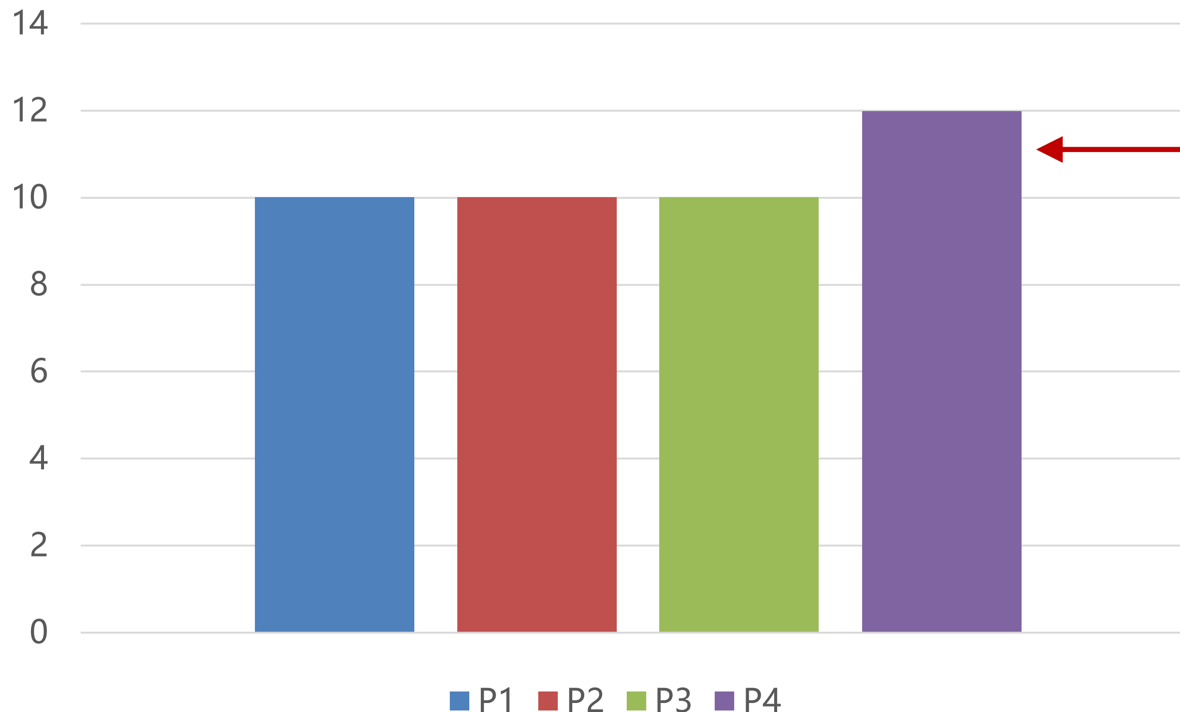
- Ideálně chceme, aby všechna vlákna/jádra pracovaly a skončily současně



Paralelní programování

Balancování

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



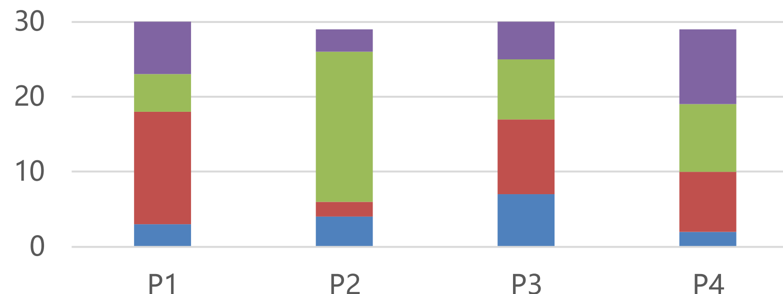
- Pokud 1 procesor pracuje o 20% déle, celý program pracuje o 20% déle
- Vzpomeňte si na Amdahlův zákon

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
    - Můžeme vyřešit optimálně pomocí rozvrhování (Constraint Satisfaction Programming)

Čas výpočtu (s) pro jednotlivé vlákna/procesory



Rozdělení práce

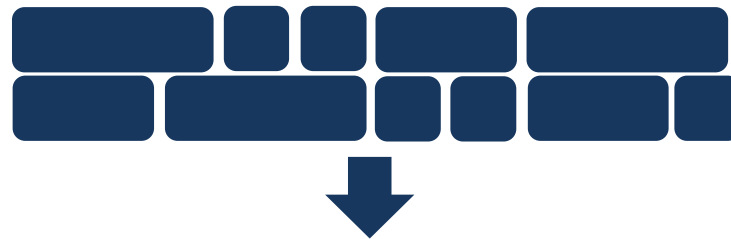
Dynamické rozdělení

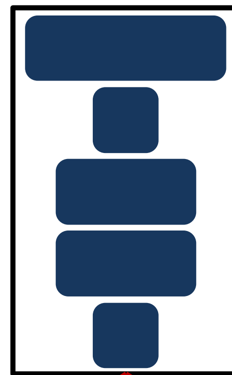
- Program přiděluje úkoly dynamicky na základě aktuálního vytížení jednotlivých vláken

Rozdělení práce

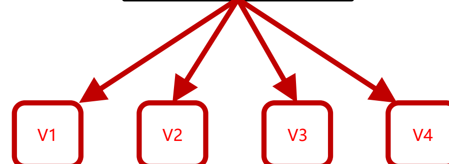
Dynamické rozdělení

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





Fronta úkolů

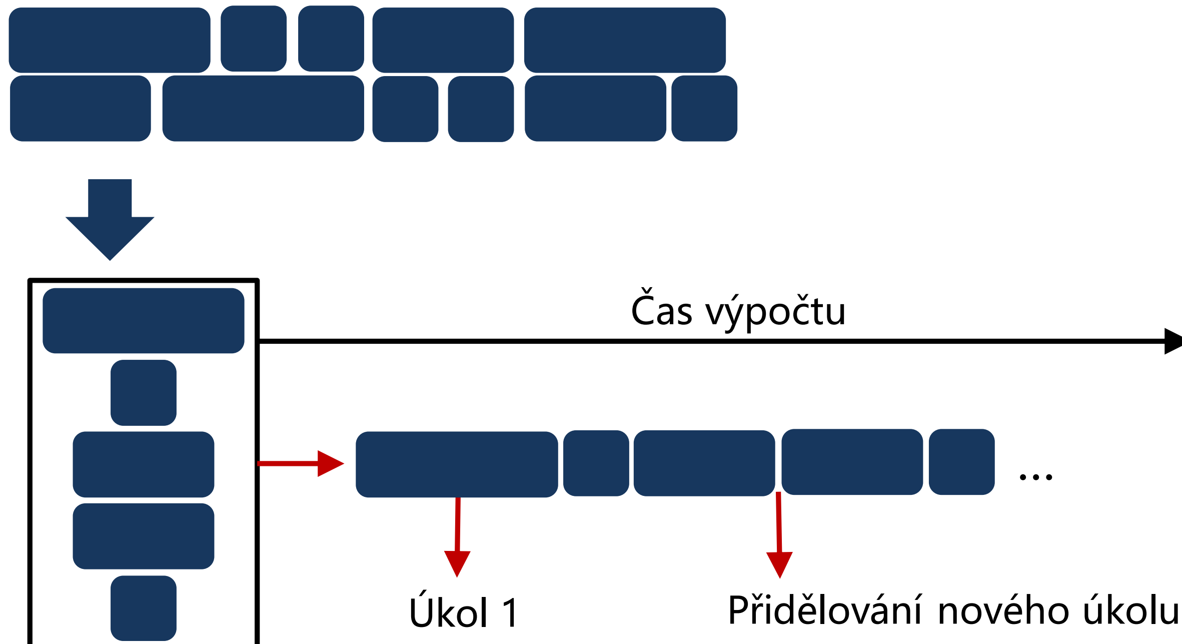


Vlákna berou úkoly z fronty.

Rozdělení práce

Dynamické rozdělení

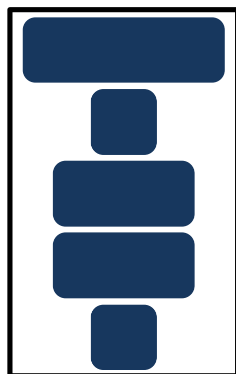
- Jak to bude vypadat z pohledu jednoho vlákna?
  - Threadpool a fronta úkolů



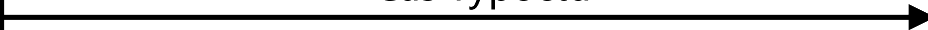
Rozdělení práce

Dynamické rozdělení

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



Čas výpočtu



…

Úkol 1

Přidělování nového úkolu.

Není v sériové variantě.

A je to exekuce v kritické sekci, která zpomaluje výpočet

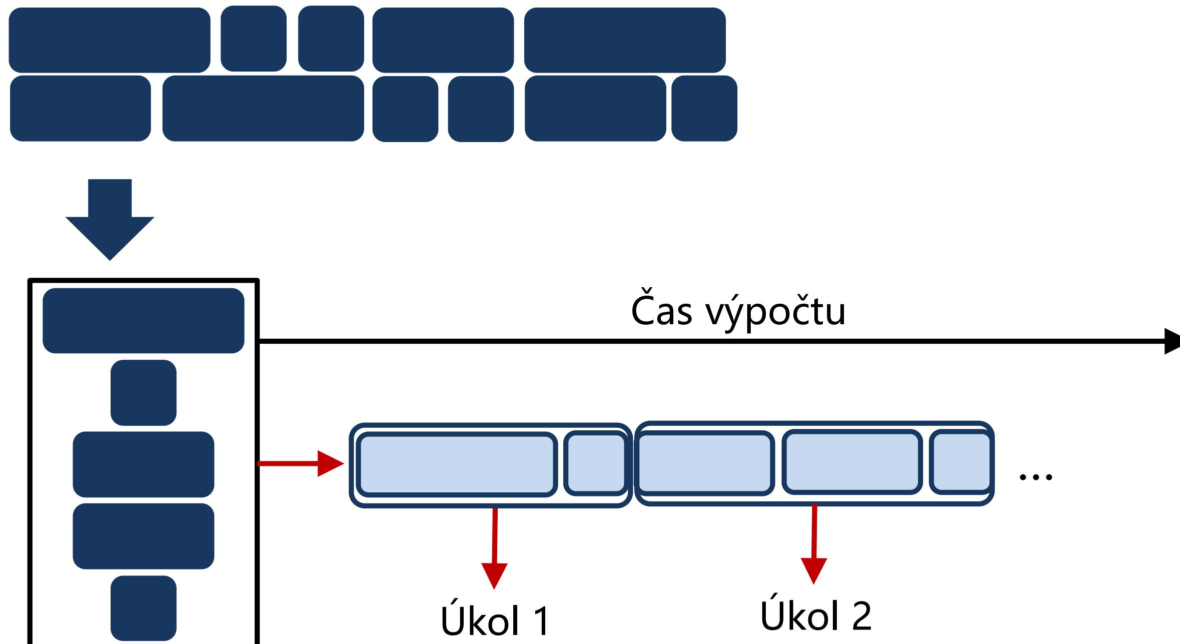
Více malých úkolů znamená dobré vybalancování mezi vlákna.

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

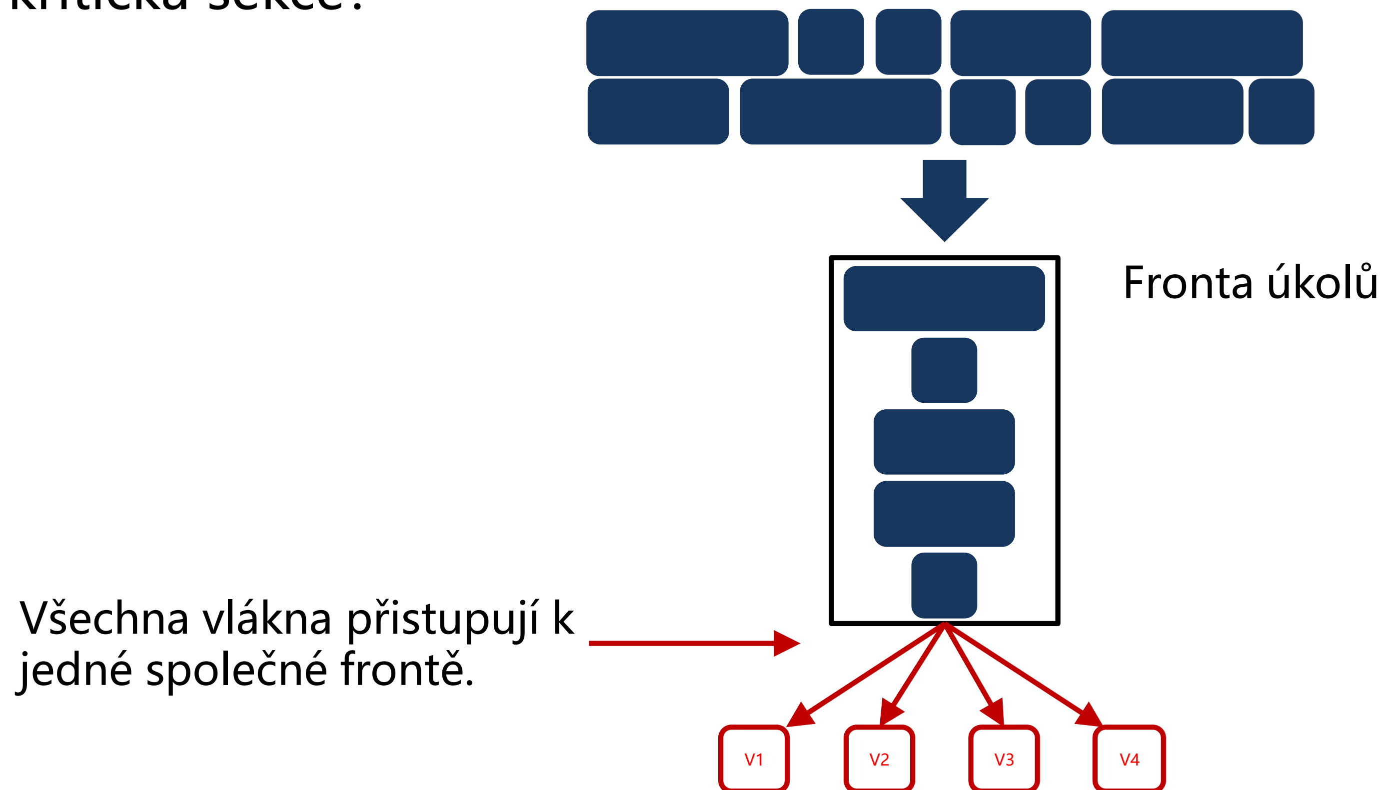
Dynamické rozdělení

- Jak zvolit správnou velikost úkolu?
- Neexistuje univerzální odpověď – závisí na problému/HW (#CPU) atd.
- Pokud lze (máme odhad), můžeme přiřazovat dlouhé úkoly nejdřív a pak krátké úkoly

Rozdělení práce

Dynamické rozdělení – problémy

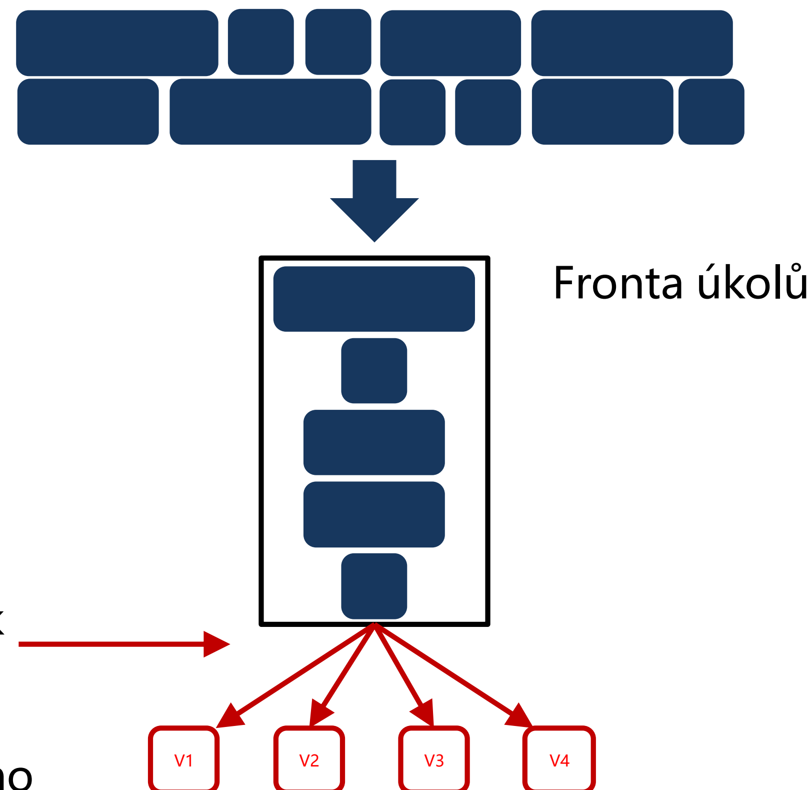
- Kde je kritická sekce?



Rozdělení práce

Dynamické rozdělení – problémy

- Kde je kritická sekce?

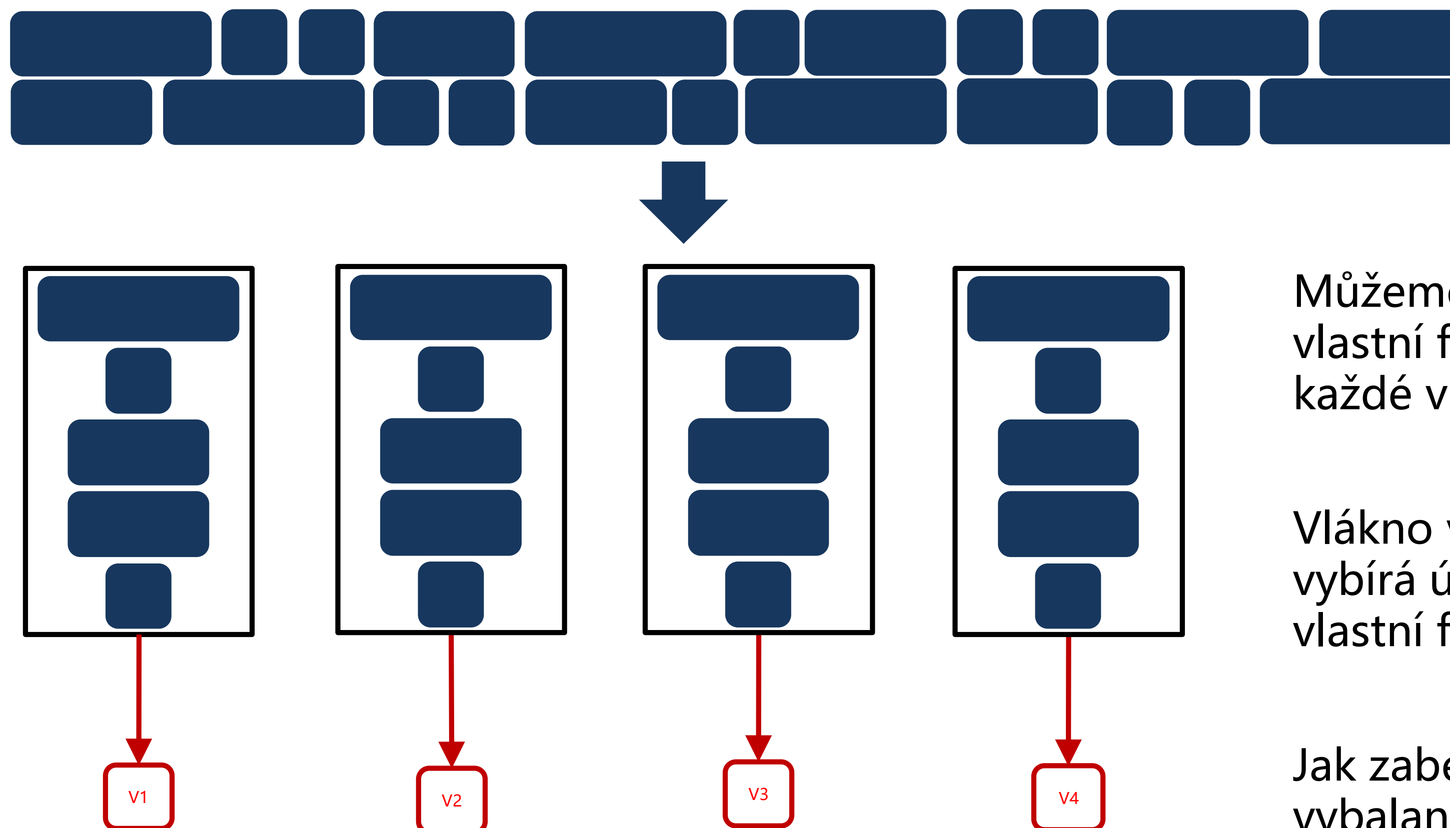


Všechna vlákna přistupují k jedné společné frontě.

Co kdyby mělo každé vlákno vlastní frontu?

Rozdělení práce

Dynamické rozdělení – vlastní fronty



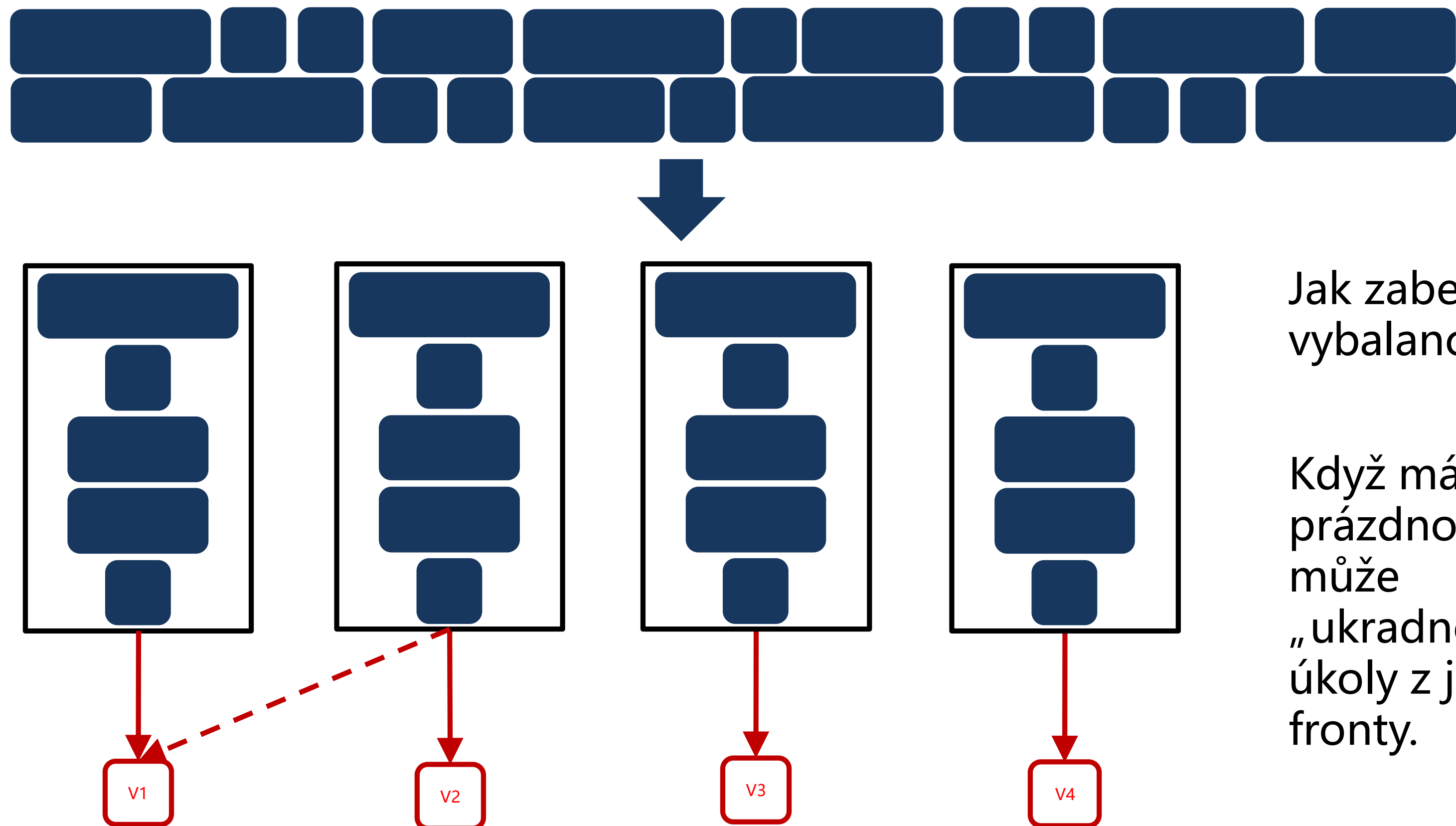
Můžeme vytvořit vlastní frontu pro každé vlákno

Vlákno vkládá a vybírá úkoly z vlastní fronty

Jak zabezpečíme vybalancování?

Rozdělení práce

Dynamické rozdělení – vlastní fronty



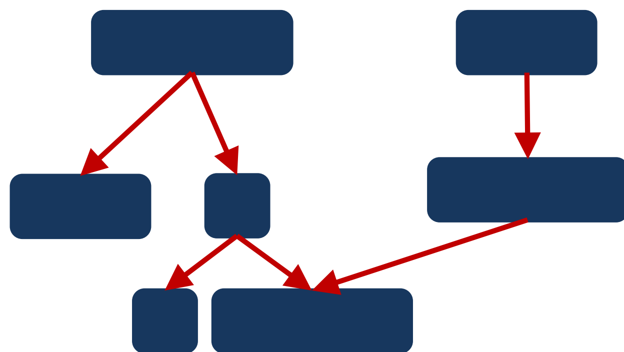
Jak zabezpečíme vybalancování?

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 – 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

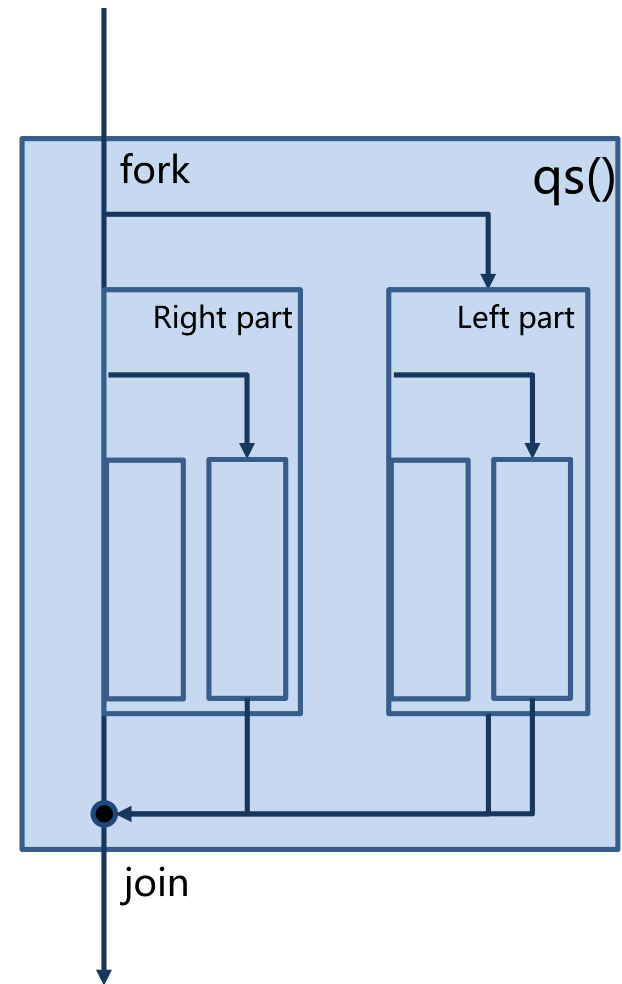
- Omezíme minimální velikost, aby nedocházelo k false-sharingu
- Můžeme asynchronně volat rekurzivní úkoly

```
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

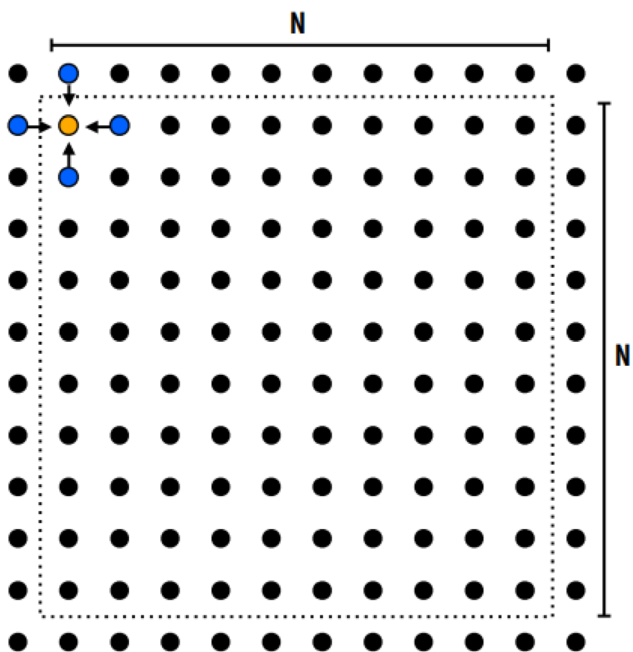
Quick Sort

```
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?



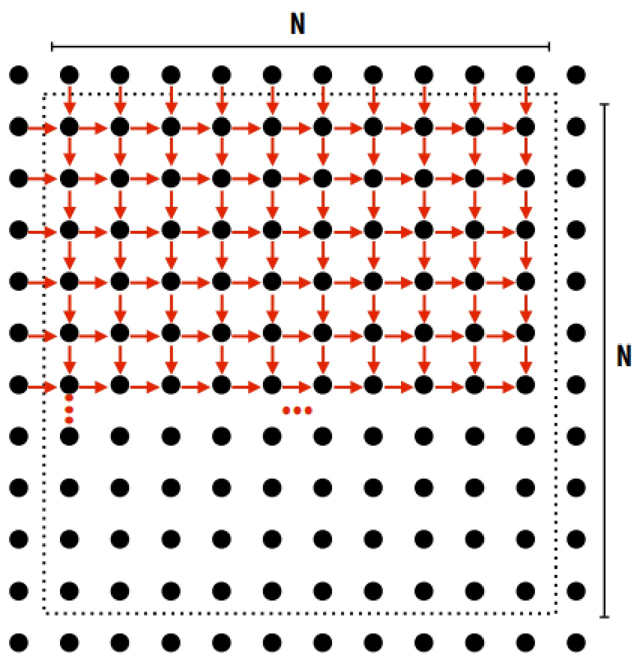
Problém:

- 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?



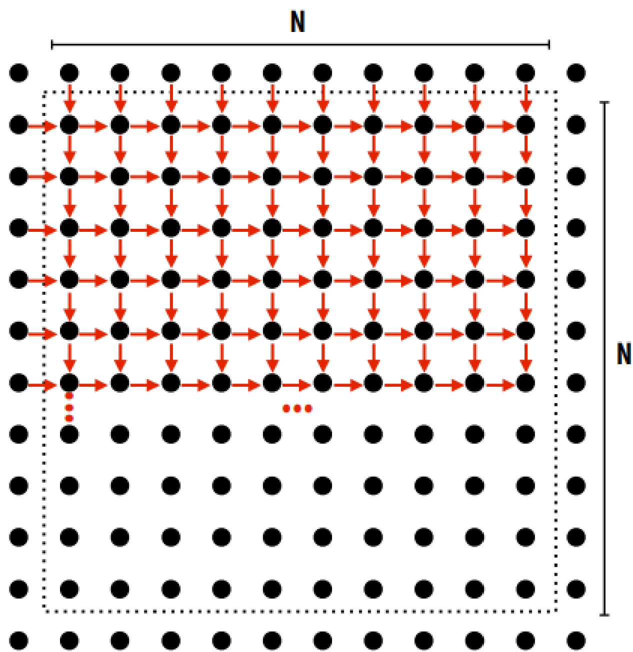
Problém:

- 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

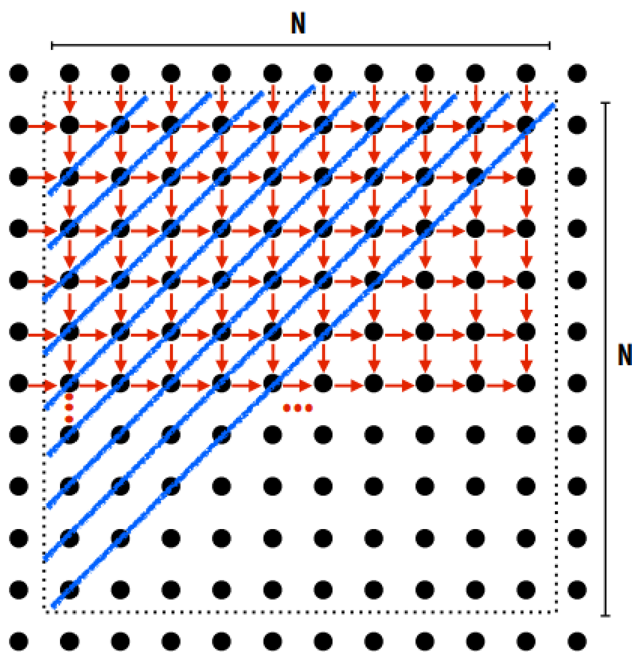
- Jakým způsobem můžeme tento problém paralelizovat?
- Zkusíme nalézt nezávislé úkoly
  - Které uzly lze aktualizovat paralelně?



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).

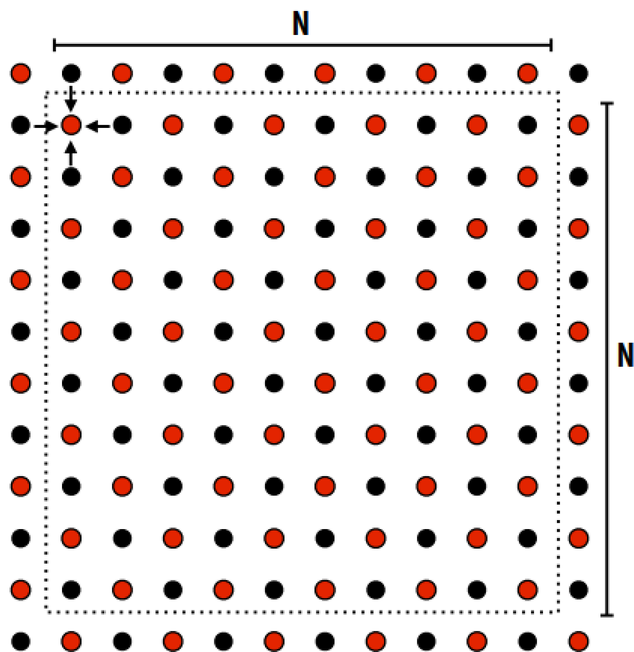
Problematické rozdělení na vlákna/procesory.

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



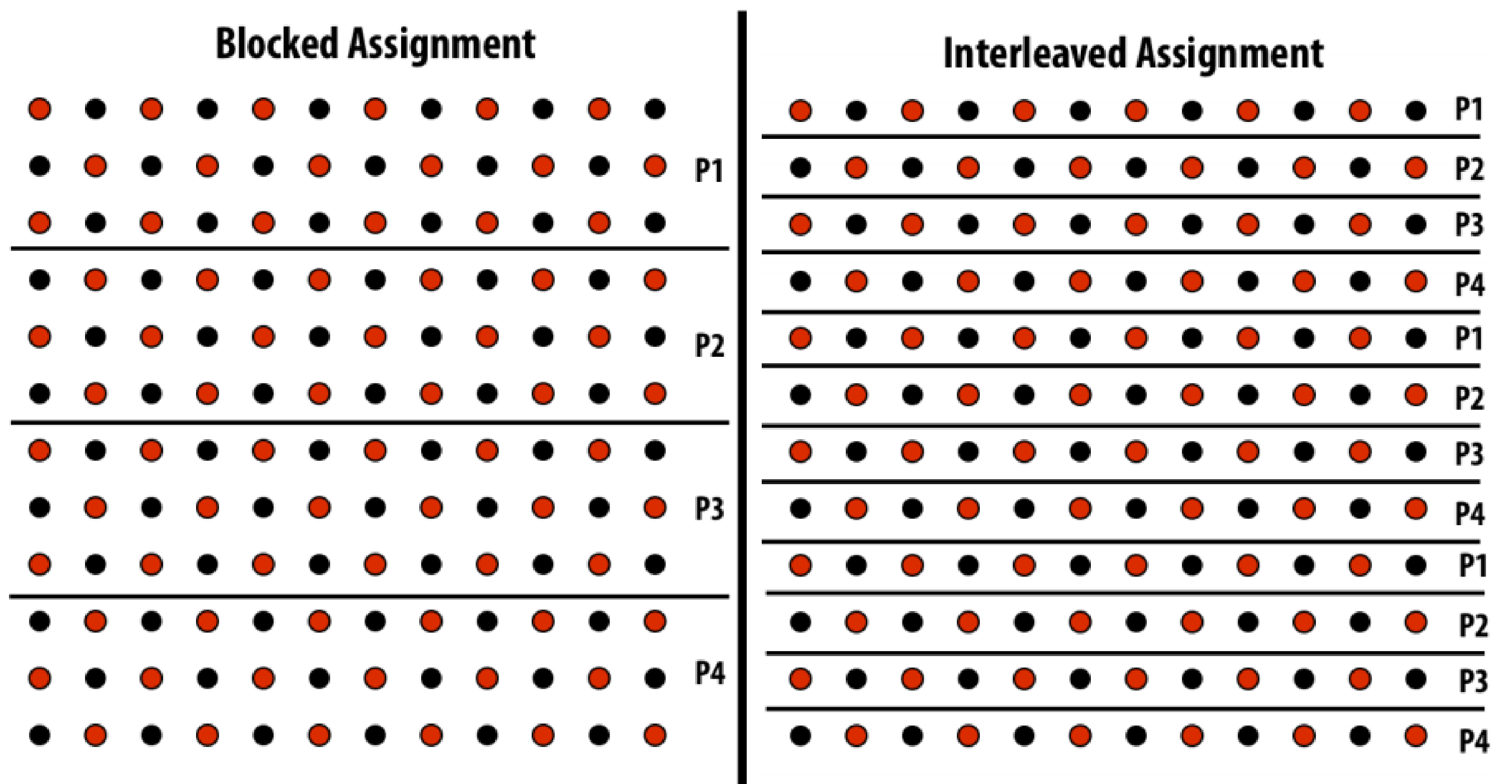
Jednoduchá paralelizace a rozdělení úkolu vláknům.

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

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

Dekompozice se závislostmi

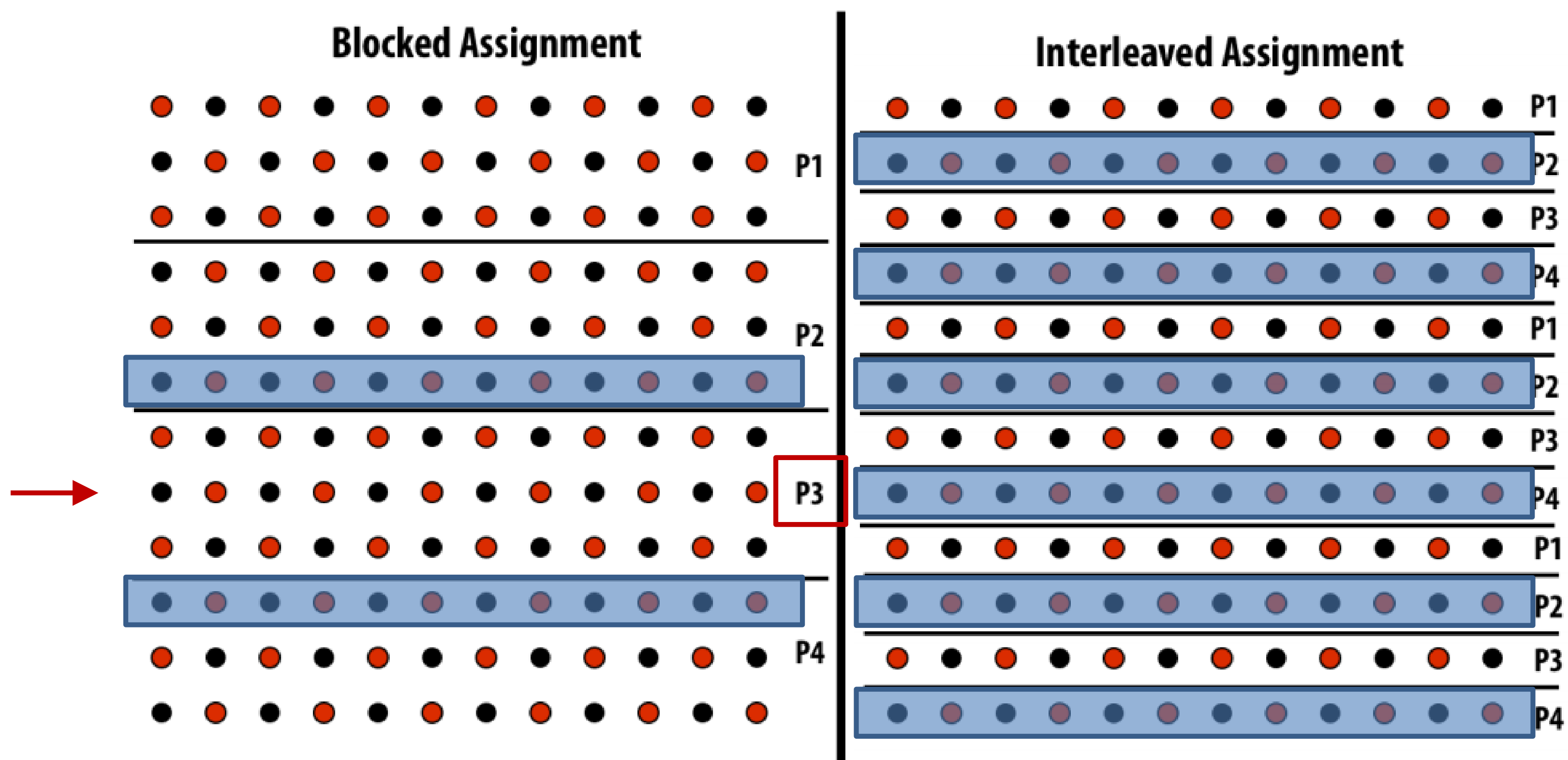
- Existuje lepší způsob?
- Jak můžeme rozdělit na úkoly pro vlákna/procesory?



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

Dekompozice se závislostmi

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



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

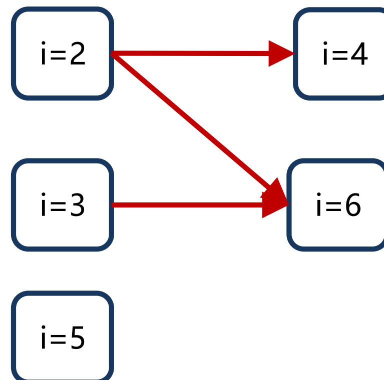
Eratostenovo síto

- 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

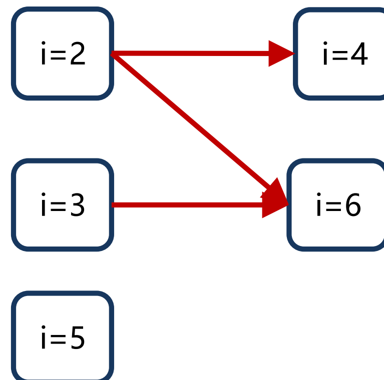
- Co se stane když paralelizujeme hlavní cyklus?
  - Např. vlákno 0 bude zpracovávat iteraci i=2, vlákno 2 bude zpracovávat iteraci i=4
  - Vlákno 2 dělá úplně zbytečnou práci – informace o tom, že číslo 4 není prvočíslo se k němu nemusí dostat včas
- Jaká je závislost mezi úkoly?



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?

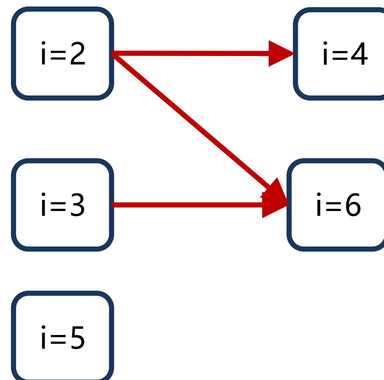


Na to abychom identifikovaly správné pořadí musíme vyřešit vlastní problém

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?

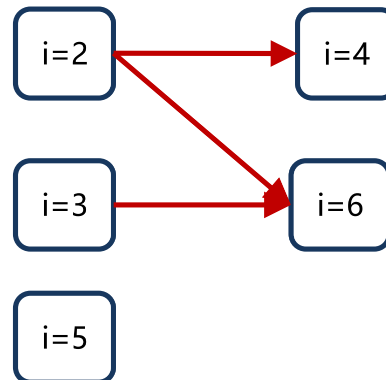


Na to abychom identifikovaly správné pořadí musíme vyřešit vlastní problém

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?

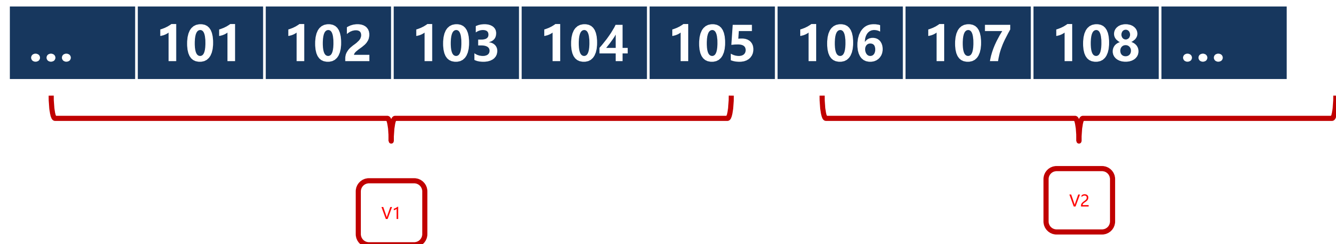


Na to abychom identifikovaly správné pořadí musíme vyřešit vlastní problém

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

```
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

Hlasování

https://goo.gl/a6BEMb