Výkon při použití vláken: prakticky

Karel Richta a kol.

katedra počítačů FEL ČVUT v Praze

Programování v C++, B6B36PJC
12/2018, Lekce YY

https://cw.fel.cvut.cz/wiki/courses/a7b36pjc/start

Testovací stroj

- Všechna měření byla provedena na stejném serveru

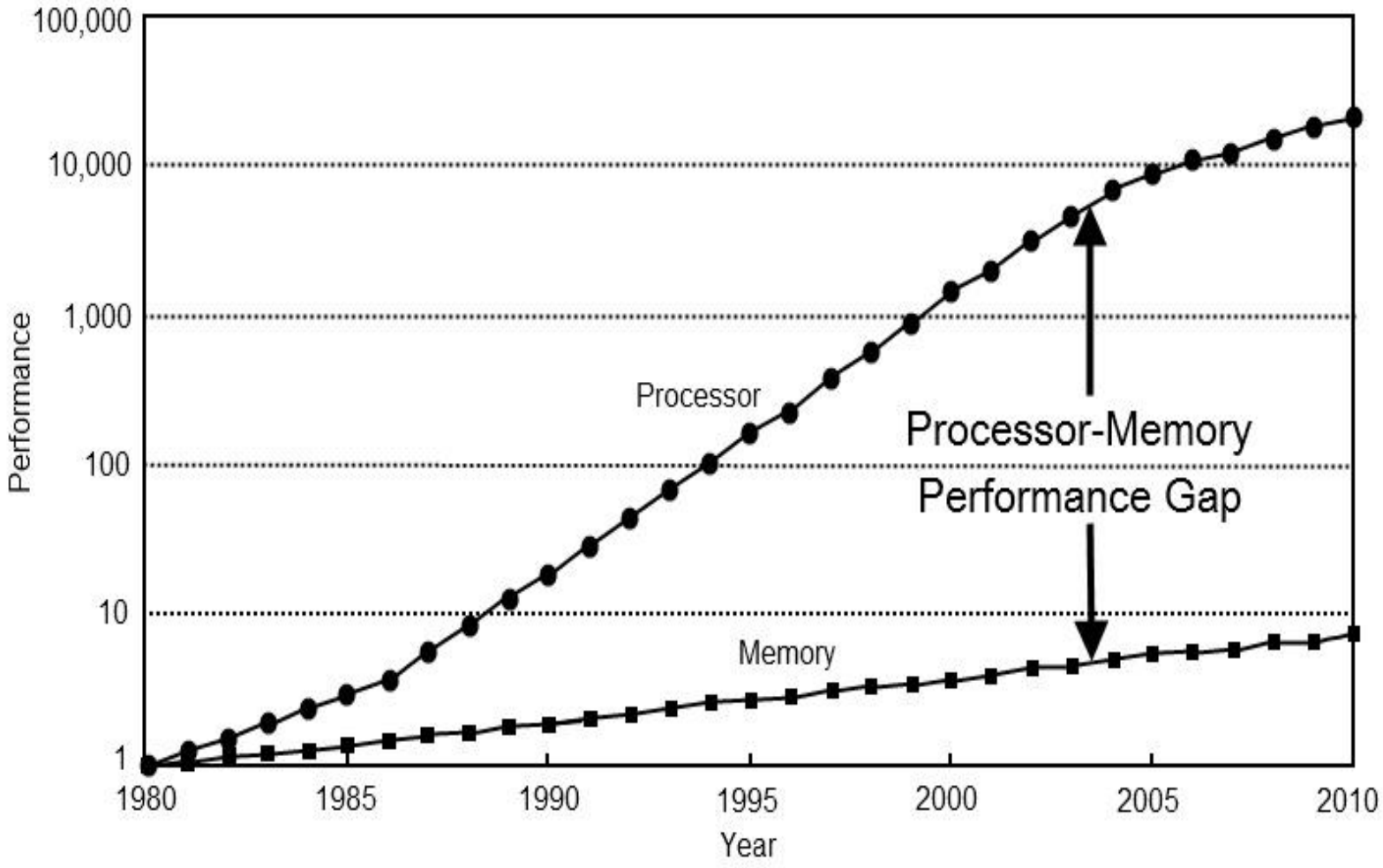
Konfigurace

- 2x Intel Xeon E5-2620 v2 (24 jader, 2.1 GHz)
- 64GB RAM
- Gentoo Linux
- Clang++ ve verzi 3.8

Co byste měli vědět

- Nesynchronizovaný přístup do sdílené paměti je chyba
  - Nesynchronizovaný znamená bez mutexů, atomických proměnných, atd.
- Vytváření vláken není zdarma
  - Ale ani není až tak drahé
- Zamykání není zdarma
  - Zvlášť když se o něj snaží více vláken najednou
- Procesory mají cache
  - Cache jsou tzv. „koherentní“, což má vliv na výkon kódu

Odbočka


100,000
10,000
1,000
100
10
1
Performance
Processor
Memory
Processor-Memory
Performance Gap
1980
1985
1990
1995
2000
2005
2010
Year

Odbočka (2)

- Dnešní procesor je řádově rychlejší než paměť
- Proto mají dnešní procesory tzv. cache
- V současné době má cache 3-4 úrovně s různou velikostí a rychlostí přístupu

Odbočka (3)

Akce	čas	cykly	poznámka
Přístup do L1 cache	2 ns	4	32 + 32 KB
Přístup do L2 cache	6 ns	12	256 KB
Přístup do L3 cache	20 ns	40	8 MB
Přístup do RAM	100 ns	200	??? GB

- Uvedené časy jsou pouze přibližné
- Přesné časy záleží na spoustě faktorů
  - Současná rychlost procesoru
  - Současná rychlost paměti
  - Fáze měsíce…

Odbočka (4)

- Z praktických důvodů cache pracují po blocích
  - Data, která jsou blízko u sebe, jsou potřeba brzy po sobě
- Pro intel i7 to bývá 64 bytů
- U CPU s více jádry musí cache komunikovat
  - Jinak by mohly obsahovat zastaralé hodnoty

Pravidla výkonu při použití vláken

- Minimalizujte množství přístupů k sdíleným a zapisovatelným datům
- Pozor na false sharing
- Minimalizujte množství práce v kritických sekcích
- Pozor na statické rozdělování práce

Přístup ke sdíleným datům

- Úkol: Máme pole čísel [0, 1) a zajímá nás, pro kolik z nich platí, že e^x > *threshold*.
- Řešení v jednom vláknu je jednoduché:

```
double threshold = ...;
auto higher = std::count_if(begin(data), end(data),
                            [=](double d) {
    return exp(d) > threshold);
});
```

Přístup ke sdíleným datům (2)

- Co když chceme řešení paralelizovat? Nabízí se jednoduchá myšlenka: rozdělíme vstupní data na *T* částí (kde *T* je počet vláken) a každé vlákno zkontroluje jednu část.

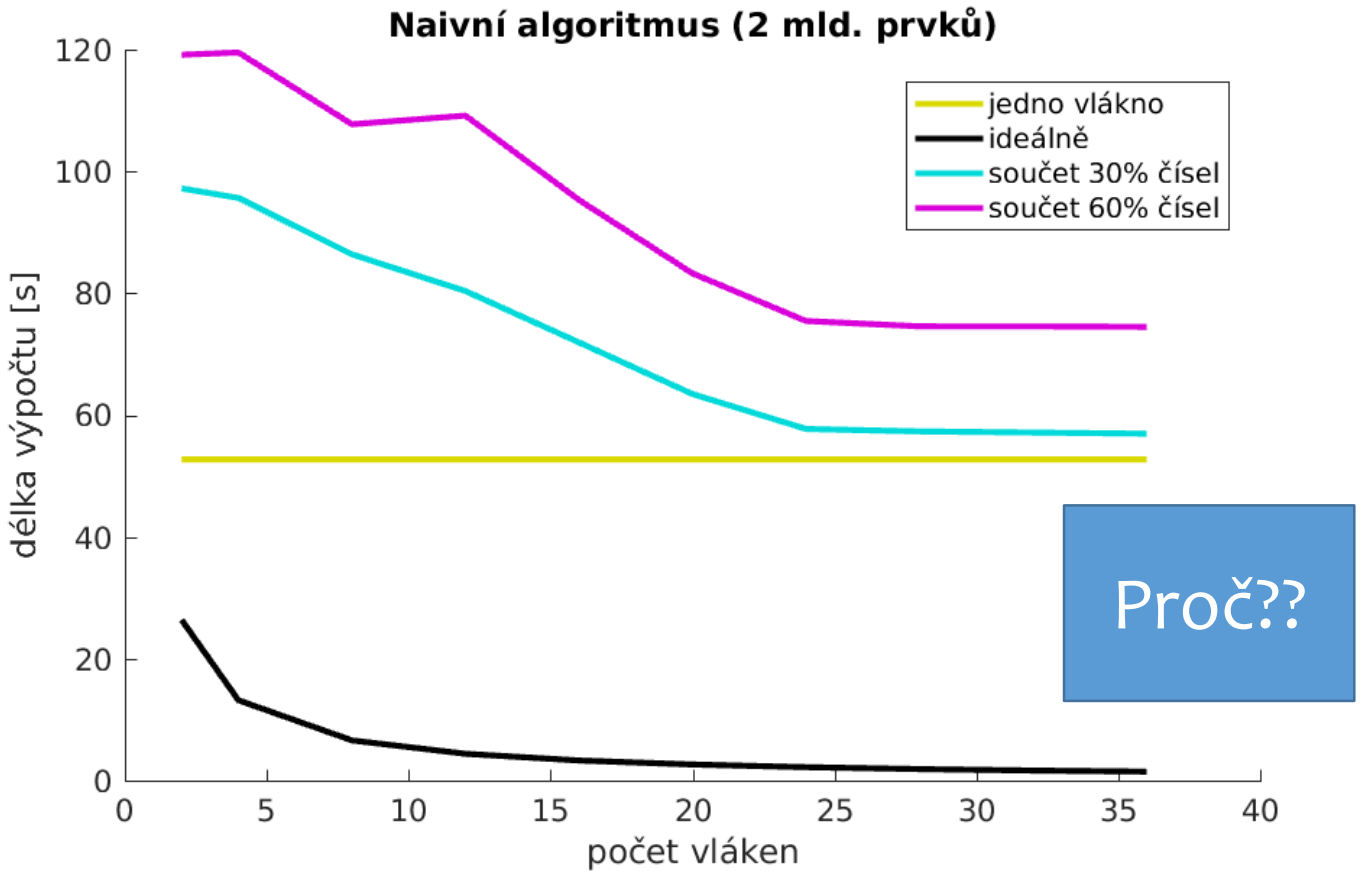
- Naivní implementace by vypadala asi takto:

```
std::atomic<int> result {0};
auto func1 = [&, threshold](size_t from, size_t to) {
    for (; from < to; ++from) {
        if (exp(data[from]) > threshold) {
            result++;
        }
    }
};
```

Jak správně napočítat `from` a `to` najdete v předchozích přednáškách

Přístup ke sdíleným datům (3)

- Naivní implementace ale nedopadne příliš dobře.

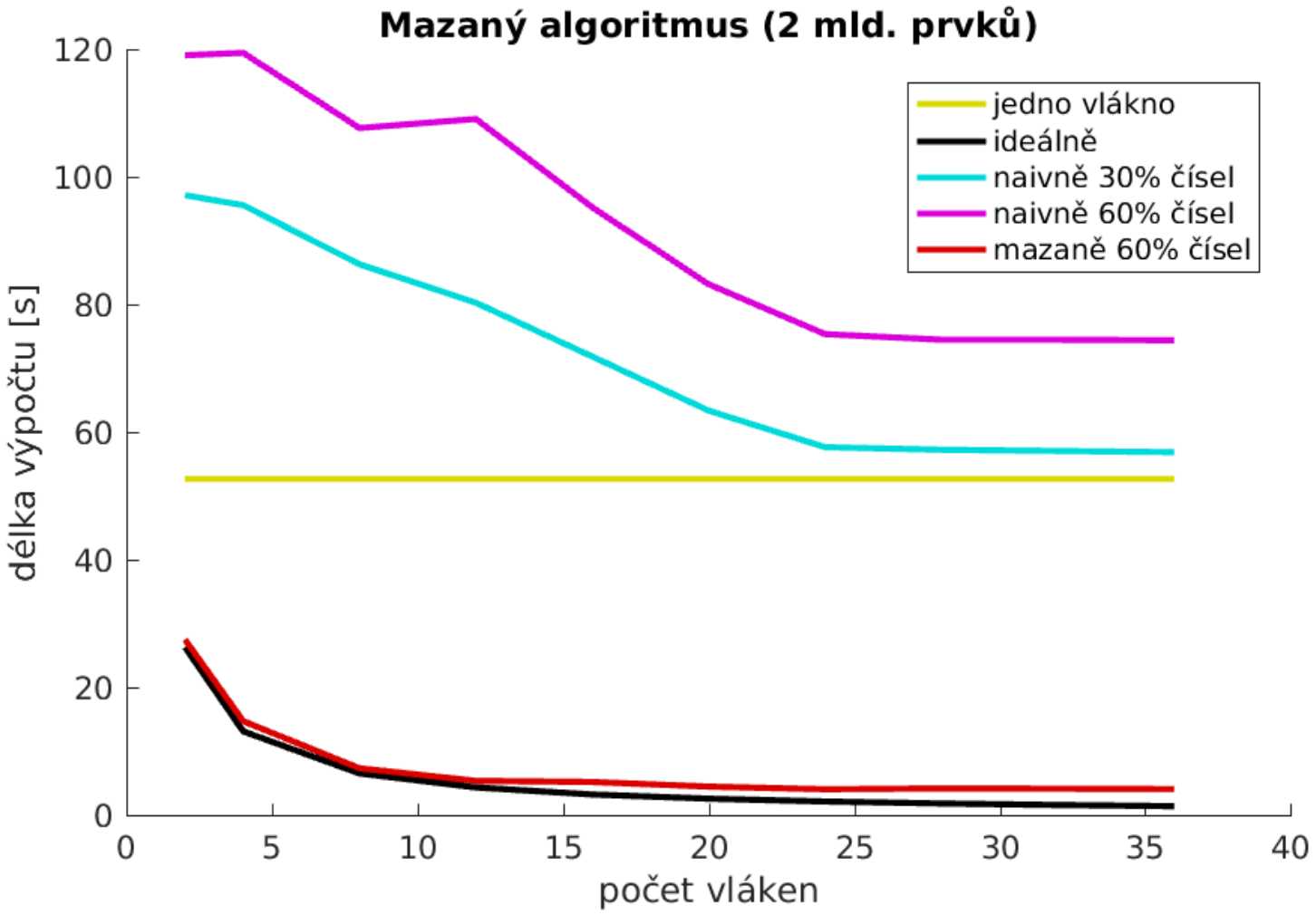


Přístup ke sdíleným datům (4)

- Když více vláken přistupuje ke stejné proměnné, jádra procesoru si to musí navzájem sdělit.
  - Všechna vlákna přistupují k proměnné `result` vždy, když se najde dostatečně vysoký exponent.
- Je tedy potřeba zmenšit množství přístupů ke sdílenné proměnné.

```
auto thread_func2 = [&, threshold](size_t from, size_t to) {
    int partial_result = 0;
    for (; from < to; ++from) {
        if (exp(data[from]) > threshold) {
            partial_result++;
        }
    }
    result += partial_result;
};
```

Přístup ke sdíleným datům (5)



Pravidla výkonu při použití vláken

- Minimalizujte množství přístupů k sdíleným a zapisovatelným datům
  - Potřebují synchronizaci, synchronizace je drahá
  - Komunikace mezi vlákny všeobecně implikuje synchronizaci
- Pozor na false sharing
- Minimalizujte množství práce v kritických sekcích
- Pozor na statické rozdělování práce

Pravidla výkonu při použití vláken

- Minimalizujte množství přístupů k sdíleným a zapisovatelným datům
  - Potřebují synchronizaci, synchronizace je drahá
  - Komunikace mezi vlákny všeobecně implikuje synchronizaci
- Pozor na false sharing
- Minimalizujte množství práce v kritických sekcích
- Pozor na statické rozdělování práce

False Sharing

- Vraťme se k předchozímu problému, ale upravme ho: nezajímá nás celkový počet, ale počet v každé části zvlášť
- Implementace v jednom vláknu je opět snadná:

```
std::vector<size_t> results(num_splits);

for (int split = 0; split < num_splits; ++split) {
    results[split] = std::count_if(start,
                                   start + split_size,
                                   [=](double d) {
        return exp(d) > threshold;
    });
    start += split_size;
}
```

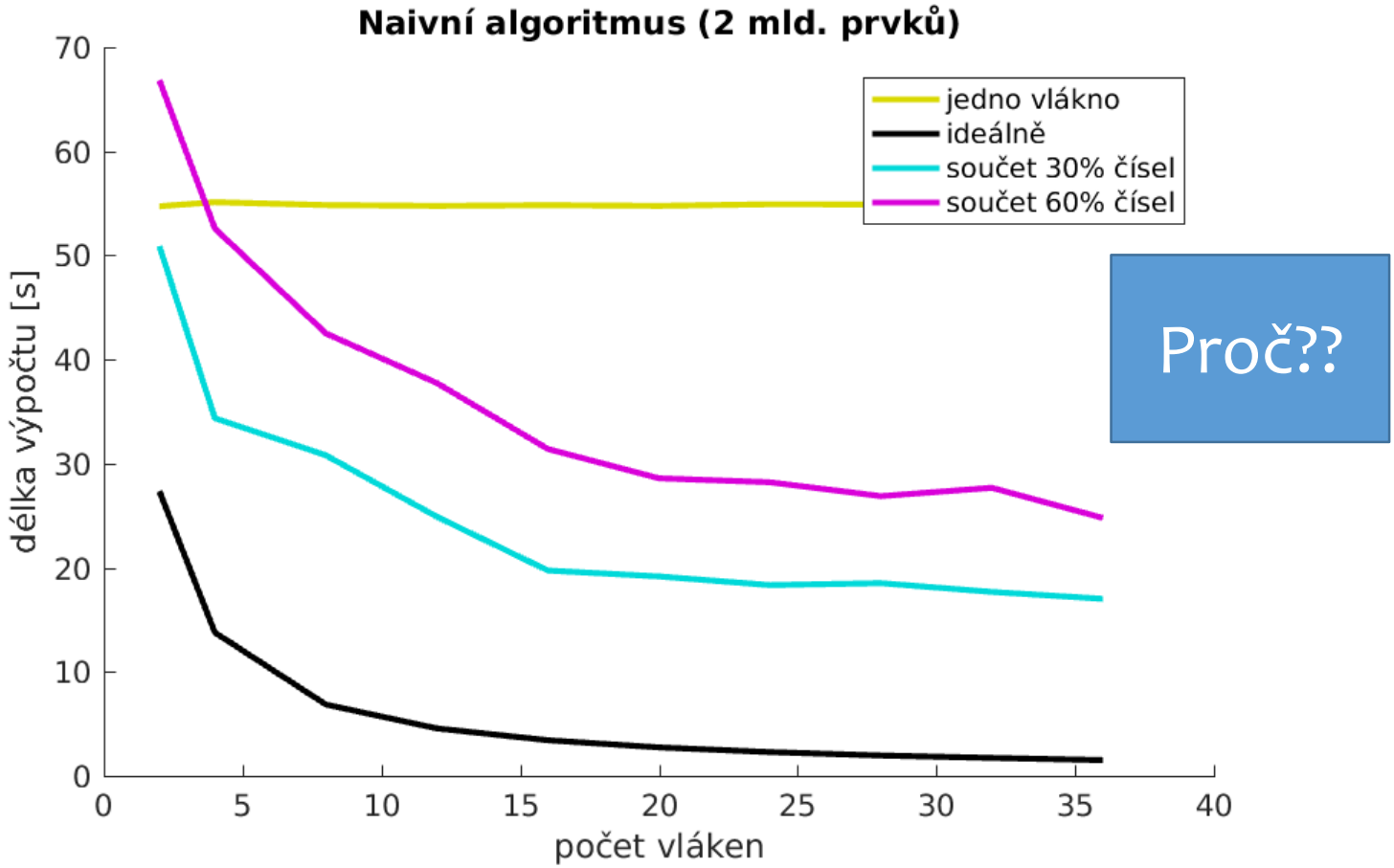
False Sharing (2)

- Implementace ve více vláknech též není složitá
  - Každé vlákno dostane část pole a index, kam má zapisovat
  - Protože nepřistupujeme ke stejné proměnné, tak nemusíme mít lokální mezisoučet

```
auto thread_func1 = [&, threshold](int order,
                                   size_t from, size_t to) {
    for (; from < to; ++from) {
        if (exp(data[from]) > threshold) {
            result[order]++;
        }
    }
};
```

False Sharing (3)

- Přímočará implementace opět nedopadne příliš dobře.



False Sharing (4)

- Jak již víme, měli bychom minimalizovat přístup ke sdíleným proměnným
- Občas se ale jako sdílené chovají i proměnné, které jsou různé
- Tomuto jevu se říká „False Sharing“ a mohou za něj cache
- Co s tím? Musíme zařídit, aby dané proměnné nesdílely řádek cache

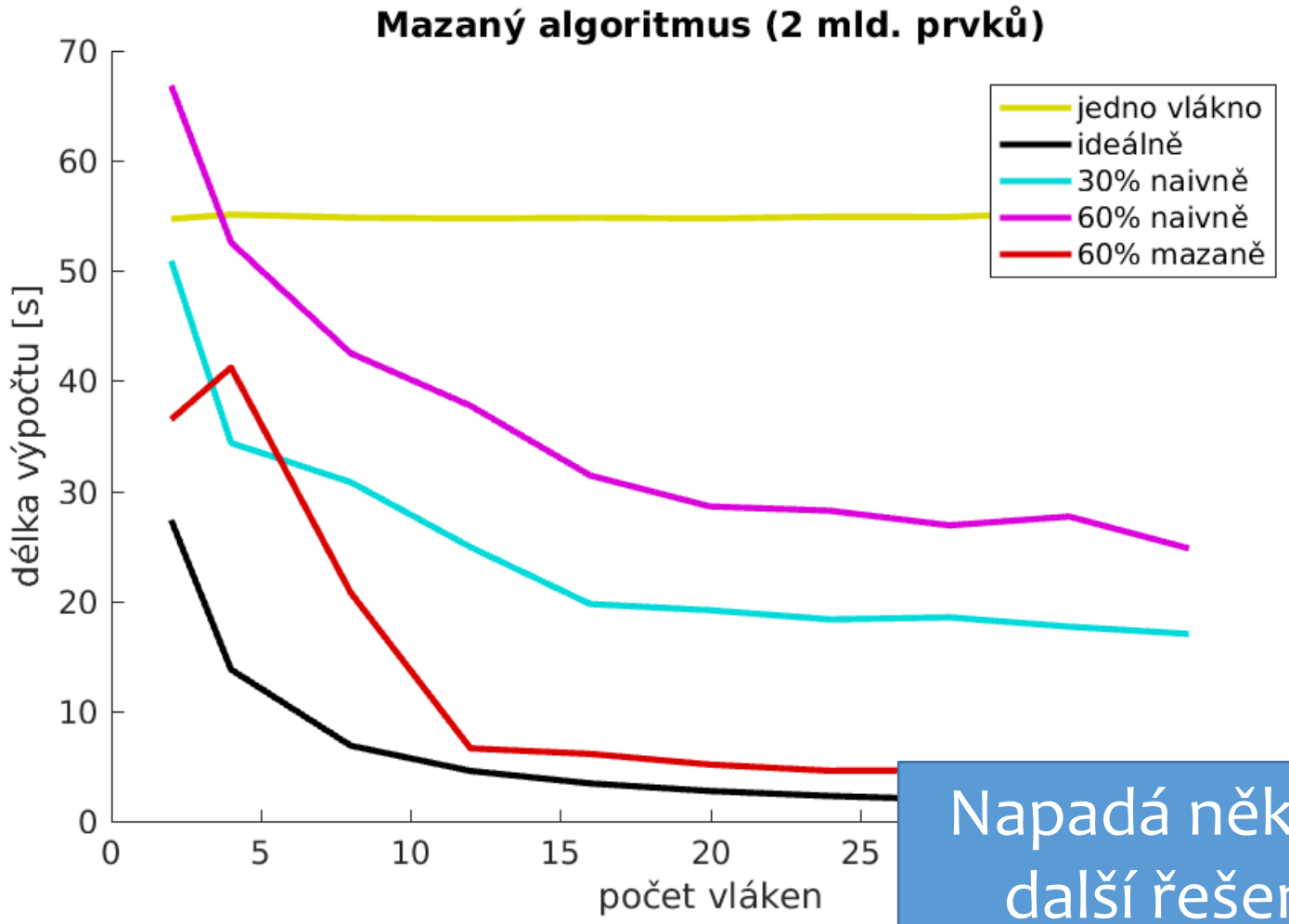
False Sharing (5)

- Nejsnazší možnost je zavést nový typ a ten patřičně nafouknout.
  - V současné době má x86 cache line 64 bytů

```
struct padded_result {
    padded_result(int res = 0):res{ res } {}
    int res;
    std::array<char,64 - sizeof(padded_result::res)> padding;
};
```

```
auto thread_func2 = [&, threshold](int order,
                                   size_t from, size_t to) {
    for (; from < to; ++from) {
        if (exp(data[from]) > threshold) {
            result[order].res++;
        }
    }
};
```

False Sharing (6)



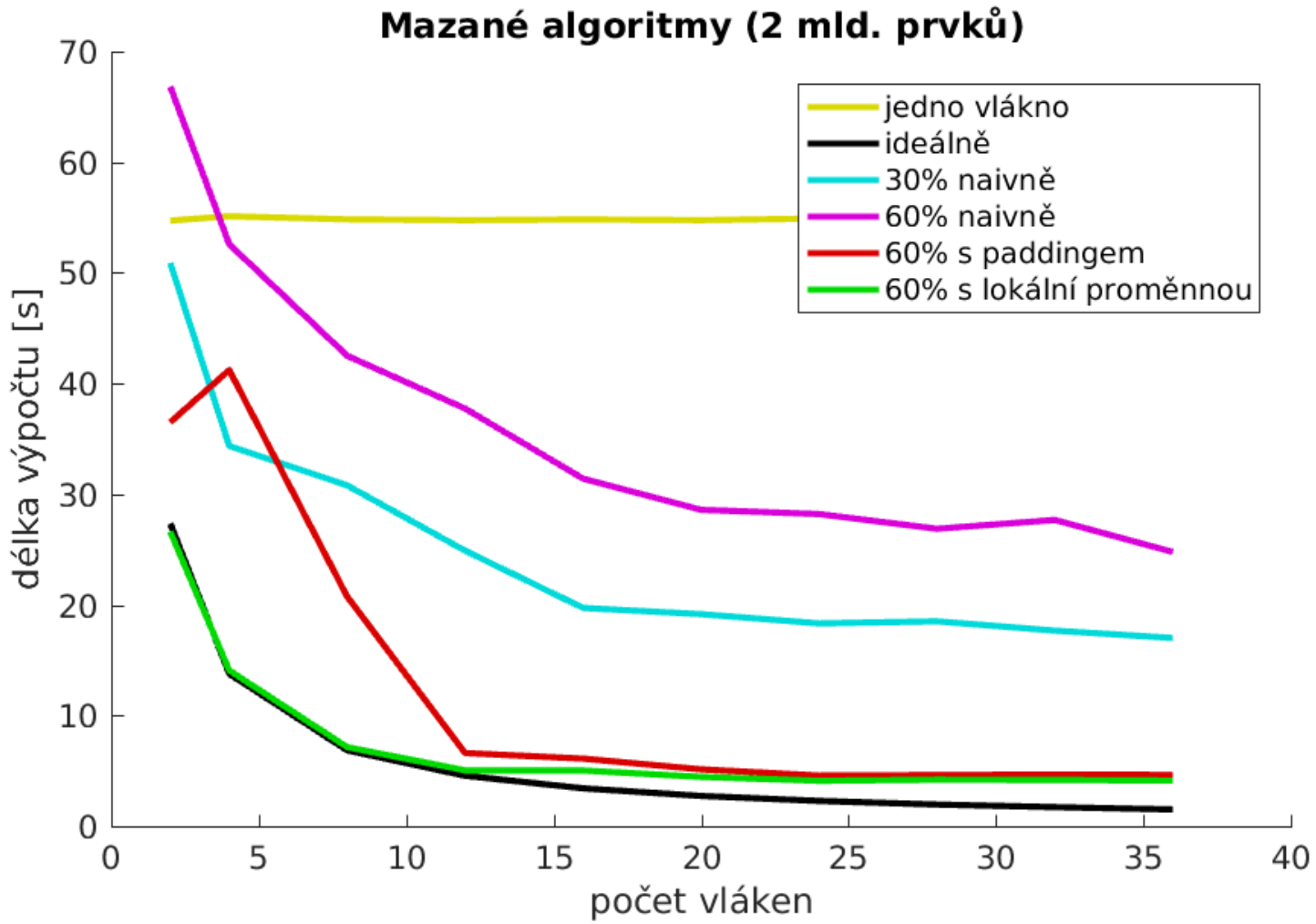
Napadá někoho další řešení?

False Sharing (7)

- Další alternativa je opět ukládat až finální výsledek

```
auto thread_func3 = [&, threshold](int order,
                                   size_t from, size_t to) {
    int partial_result = 0;
    for (; from < to; ++from) {
        if (exp(data[from]) > threshold) {
            partial_result += 1;
        }
    }
    result[order] = partial_result;
};
```

False Sharing (8)



Pravidla výkonu při použití vláken

- Minimalizujte množství přístupů k sdíleným a zapisovatelným datům
  - Potřebují synchronizaci, synchronizace je drahá
  - Komunikace mezi vlákny všeobecně implikuje synchronizaci
- Pozor na false sharing
  - CPU nevidí data granulárně, ale po blocích
  - Nesouvisející proměnné mohou být „sdílené“
- Minimalizujte množství práce v kritických sekcích
- Pozor na statické rozdělování práce

Pravidla výkonu při použití vláken

- Minimalizujte množství přístupů k sdíleným a zapisovatelným datům
  - Potřebují synchronizaci, synchronizace je drahá
  - Komunikace mezi vlákny všeobecně implikuje synchronizaci
- Pozor na false sharing
  - CPU nevidí data granulárně, ale po blocích
  - Nesouvisející proměnné mohou být „sdílené“
- Minimalizujte množství práce v kritických sekcích
- Pozor na statické rozdělování práce

Práce v kritické sekci

- Kritická sekce je část programu, kde v jednu chvíli může být pouze jedno vlákno.
- Je jasné, že pro dobrý výkon programu je potřeba mít kritické sekce co nejkratší, aby ostatní vlákna nečekala.
- Ale práce se do kritické sekce může dostat nečekanými způsoby
- Řekněme, že používáme v programu MPMC (Multiple Producer, Multiple Consumer) frontu
- Co se stane, když použijeme naivní implementaci?

Práce v kritické sekci (2)

- Naivní implementace MPMC fronty vypadá takto

```
template <typename T>
class MPMC_queue {
public:
    void push(const T& elem) {
        std::unique_lock<std::mutex> lock(m_mutex);
        m_data.push(elem);
        m_cvar.notify_one();
}
...
```

Práce v kritické sekci (2 pokrač.)

```
template <typename T>
class MPMC_queue {
    ...
    void pop(T& elem) {
        std::unique_lock<std::mutex> lock(m_mutex);
        m_cvar.wait(lock, [&](){
                return !m_data.empty();
        });
        elem = std::move_if_noexcept(m_data.front());
        m_data.pop();
    }

private:
    std::queue<T> m_data;
    std::mutex m_mutex;
    std::condition_variable m_cvar;
};
```

Práce v kritické sekci (3)

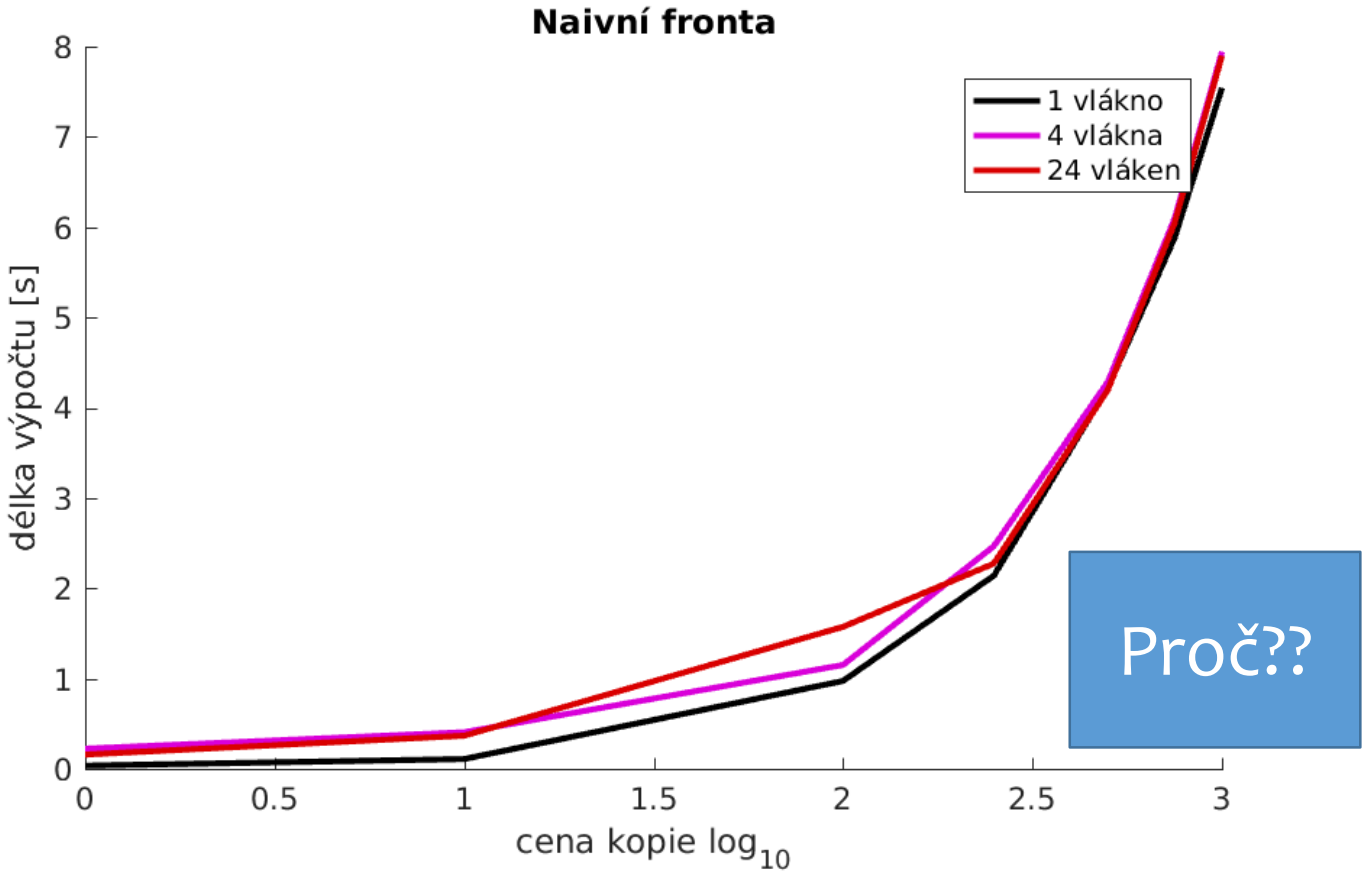
- A tento objekt posíláme skrze frontu, abychom simulovali objekt s nákladnou kopií:

```
class waiter {
public:
    explicit waiter(int);
    waiter(const waiter&);
    waiter& operator=(const waiter& rhs);

private:
    void slowdown() const {
        for (int i = 0; i < m_waitfactor; ++i) {
            auto func = [=](int d) { return i / (d + 1); };
            dump = func(dump);
        }
    }
    int m_waitfactor;
};
```

Práce v kritické sekci (4)

- Jak dlouho trvá dostat 320 tisíc prvků skrz frontu?



Práce v kritické sekci (5)

- Žádného zrychlení jsme nedosáhli, protože dochází k velkému množství práce pod kritickou sekcí a tudíž vlákna pracují jedno po druhém
- Problém je způsobený frontou založenou na poli (deque), kde není možné vytáhnout prvek bez zamknutí celé fronty.
- Fronta založená na spojovém seznamu toto umí.
  - Ale sami si další spojový seznam psát nechceme, co s tím?
  - Pomůžeme si standardní knihovnou

list_queue (1)

```
template <typename T>
class list_queue {
public:

    void push(const T& elem) {
        std::list<T> temp_data;
        temp_data.push_back(elem);

        { // Enter critical section
            std::unique_lock<std::mutex> lock(m_mutex);
            m_data.splice(m_data.end(), temp_data);
            m_cvar.notify_one();
        }
    }
    ...
```

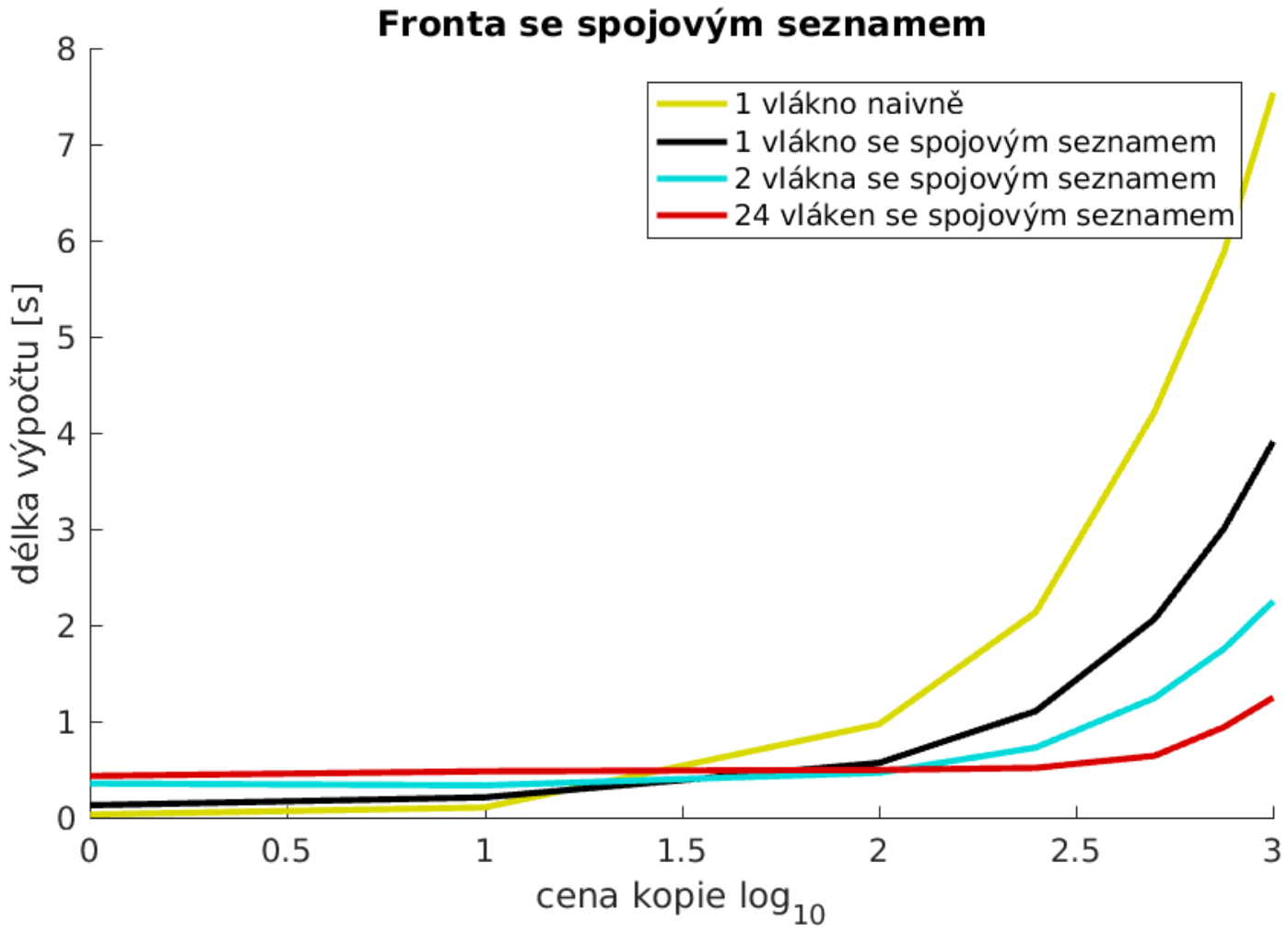
```
private:
  std::list<T> m_data;
  std::mutex m_mutex;
  std::condition_variable m_cvar;
};
```

list_queue (2)

```
template <typename T>
class list_queue {
public:
    ...
    void pop(T& elem) {
        std::list<T> temp;

        { // Enter critical section to slice an element
            std::unique_lock<std::mutex> lock(m_mutex);
            m_pop_var.wait(lock, [&]() {
                return !m_data.empty();
            });
            temp_data.splice(temp_data.end(), m_data,
                                    m_data.begin());
        }
        elem = std::move_if_noexcept(temp.front());
    }
    ...
```

Práce v kritické sekci (6)



Pravidla výkonu při použití vláken

- Minimalizujte množství přístupů k sdíleným a zapisovatelným datům
  - Potřebují synchronizaci, synchronizace je drahá
  - Komunikace mezi vlákny všeobecně implikuje synchronizaci
- Pozor na false sharing
  - CPU nevidí data granulárně, ale po blocích
  - Nesouvisející proměnné mohou být „sdílené“
- Minimalizujte množství práce v kritických sekcích
  - Pozor, kopírování je taky práce
- Pozor na statické rozdělování práce

Pravidla výkonu při použití vláken

- Minimalizujte množství přístupů k sdíleným a zapisovatelným datům
  - Potřebují synchronizaci, synchronizace je drahá
  - Komunikace mezi vlákny všeobecně implikuje synchronizaci
- Pozor na false sharing
  - CPU nevidí data granulárně, ale po blocích
  - Nesouvisející proměnné mohou být „sdílené“
- Minimalizujte množství práce v kritických sekcích
  - Pozor, kopírování je taky práce
- Pozor na statické rozdělování práce

Static Scheduling

- Vraťme se znovu k problému, kdy máme fixní množství práce, které chceme paralelizovat.
- Místo pole exponentů budeme mít pole `waiter` objektů.
- Tentokrát ale nebudou všechny objekty reprezentovat stejné množství práce.
  - 1/*množství vláken* objektů bude reprezentovat x-násobek práce

Upravený waiter

```
volatile int dump;

class waiter {
public:
    ...
    int process() const {
        slowdown();
        return m_waitfactor;
    }
private:
    void slowdown() const {
        for (int i = 0; i < m_waitfactor; ++i) {
            auto func = [=](int d) { return i / (d + 1); };
            dump = func(dump);
        }
    }
    int m_waitfactor;
};
```

Static Scheduling (2)

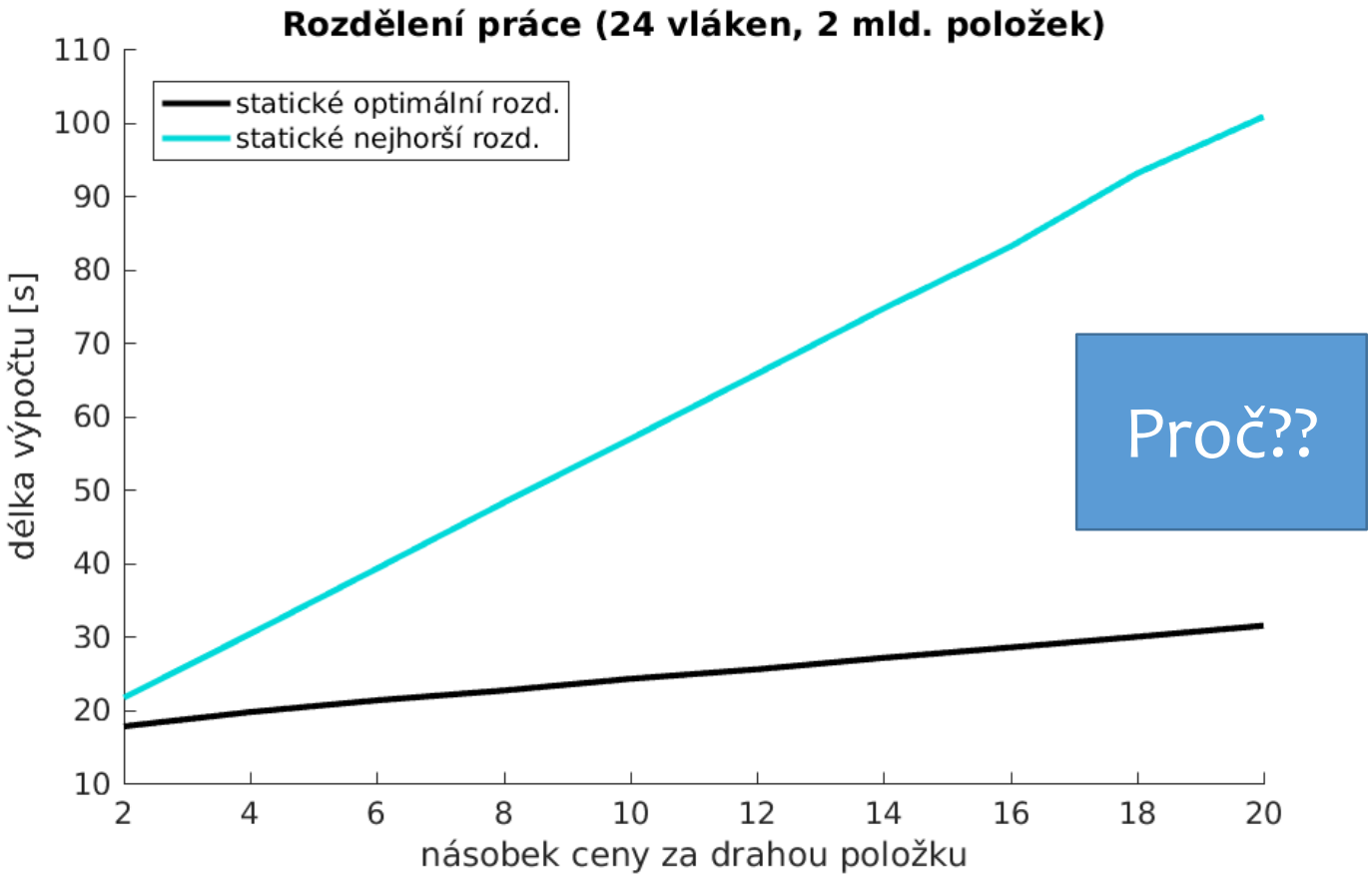
- Jednoduchá implementace by vypadala takto

```
auto split_size = data.size() / num_threads;
for (int i = 0; i < num_threads; ++i) {
    threads.emplace_back([&](int from, int to) {
        for (; from < to; ++from) {
            data[from].process();
        }
    }, i * split_size, (i + 1) * split_size);
}
```

- Jak dobře poběží?
- Záleží na pořadí objektů na vstupu?

Static Scheduling (3)

- Nastavíme `waitfactor` na 50.



Static Scheduling (4)

- Problém je v tom, že pokud rozdělíme práci na začátku, tak se čeká na nejpomalejší vlákno.
- Nejpomalejší vlákno je to, které dostane nejtěžší práci.
- Řešením tohoto problému je rozdělit práci dynamicky
  - Každé vlákno si vždy dojde pro nový index práce, dokud nedostanu index za konec pole.

Dynamic Scheduling

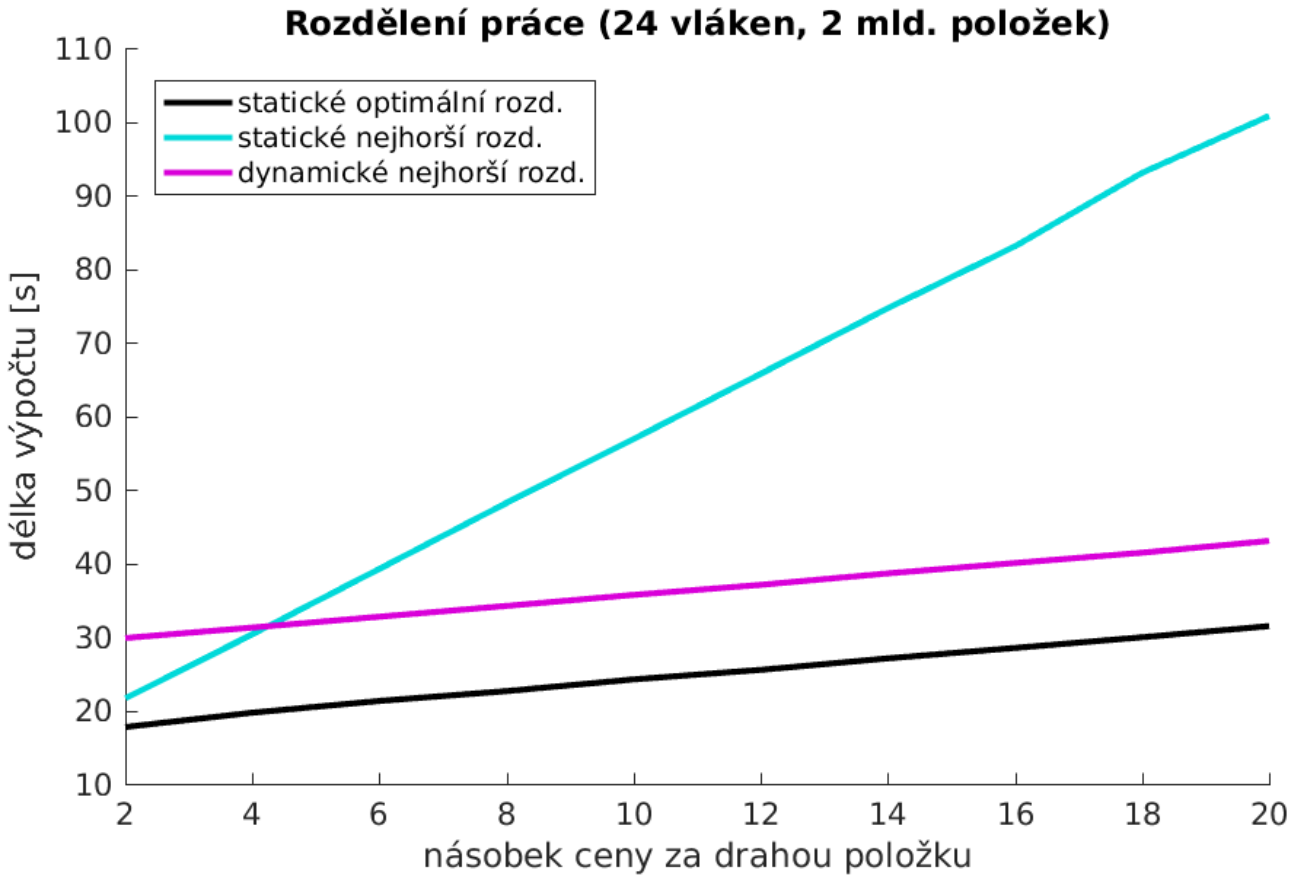
- Základní implementace je znovu jednoduchá

```
std::atomic<size_t> current_index{ 0 };
for (int i = 0; i < num_threads; ++i) {
    threads.emplace_back([&](size_t limit) {
        size_t taken = 0;
        while (taken < limit) {
            taken = current_index++;
            if (taken < limit) {
                data[taken].process();
            }
        }
    }, data.size());
}
```

- Umíme ji zlepšit?

Dynamic Scheduling (2)

- Stále měřeno pro `waitfactor` 50



Dynamic Scheduling – vylepšení

- Víme, že by více vláken nemělo sdílet jednu atomickou proměnnou
- Víme, že by více vláken nemělo sdílet proměnné vedle sebe
- Co kdyby tedy každé vlákno zpracovávalo svojí část, až do doby než skončí a pak šlo na pomoc dalším?

Sdílení indexů

```
class index_drawer {
public:
    ...
    bool get(uint32_t& index) {
        if (next >= last) { return false; }

        uint32_t temp = next++;
        if (temp >= last) { return false; }
        index = temp;
        return true;
    }
    // Nastaví next a last na předané hodnoty.
    void reset(uint32_t from, uint32_t to);

private:
    std::atomic<uint32_t> next; std::array<uint8_t, 60> pad;
    uint32_t last; std::array<uint8_t, 60> pad2;
};
```

Sdílení indexů (2)

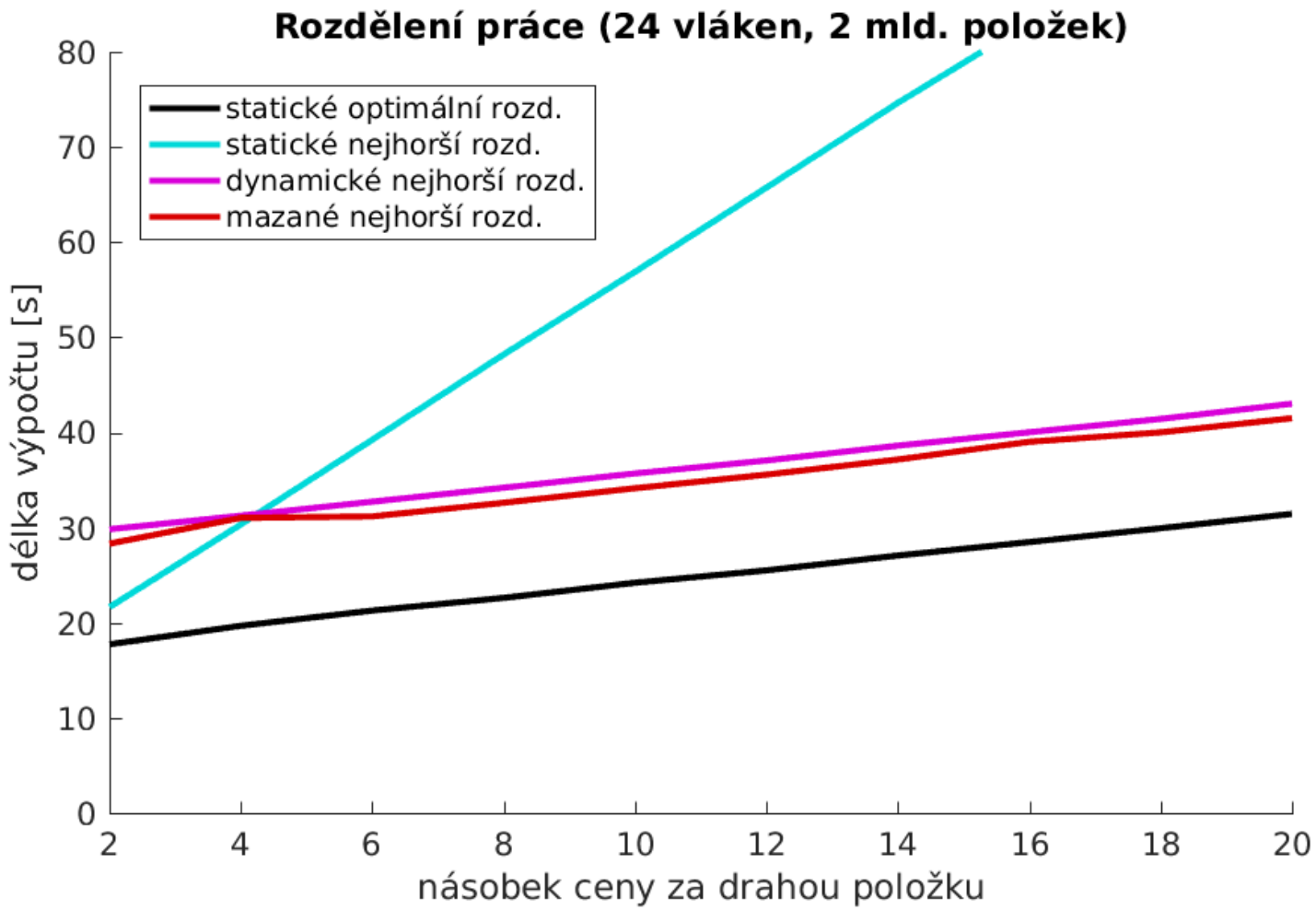
```
class index_cupboard {
public:
    index_cupboard(uint32_t num_drawers, uint32_t up_to);
    bool get(int first, uint32_t& index, int& last) {
        uint32_t drawer = first;
        for (int i = 0; i < number_of_drawers; ++i) {
            if (drawers[drawer].get(index)) {
                last = drawer;
                return true;
            }
            if (++drawer >= number_of_drawers) drawer = 0;
        }
        return false;
    }
private:
    std::unique_ptr<index_drawer[]> drawers;
    std::size_t number_of_drawers = 0;
};
```

Sdílení indexů (3)

- Pak ještě musíme uzpůsobit vlákna, aby pracovala s naší novou třídou `index_cupboard`.

```
index_cupboard cupboard(num_threads, data.size());
for (int i = 0; i < num_threads; ++i) {
    threads.emplace_back([&](int cupboard_idx) {
        uint32_t data_idx = 0;
        while (cupboard.get(cupboard_idx,
                            data_idx, cupboard_idx)) {
            data[data_idx].process();
        }
    }, i);
}
```

Sdílení indexů (4)



Pravidla výkonu při použití vláken

- Minimalizujte množství přístupů k sdíleným a zapisovatelným datům
  - Potřebují synchronizaci, synchronizace je drahá
  - Komunikace mezi vlákny všeobecně implikuje synchronizaci
- Pozor na false sharing
  - CPU nevidí data granulárně, ale po blocích
  - Nesouvisející proměnné mohou být „sdílené“
- Minimalizujte množství práce v kritických sekcích
  - Pozor, kopírování je taky práce
- Pozor na statické rozdělování práce
  - Různé problémy mohou běžet různou dobu

Pravidla výkonu při použití vláken

- Minimalizujte množství přístupů k sdíleným a zapisovatelným datům
  - Potřebují synchronizaci, synchronizace je drahá
  - Komunikace mezi vlákny všeobecně implikuje synchronizaci
- Pozor na false sharing
  - CPU nevidí data granulárně, ale po blocích
  - Nesouvisející proměnné mohou být „sdílené“
- Minimalizujte množství práce v kritických sekcích
  - Pozor, kopírování je taky práce
- Pozor na statické rozdělování práce
  - Různé problémy mohou běžet různou dobu

Děkuji za pozornost.

Picture credits

- Rychlost CPU vs RAM bylo vzato z

Computer Architecture: A Quantitative Approach by John L. Hennessy, David A. Patterson, Andrea C. Arpaci-Dusseau