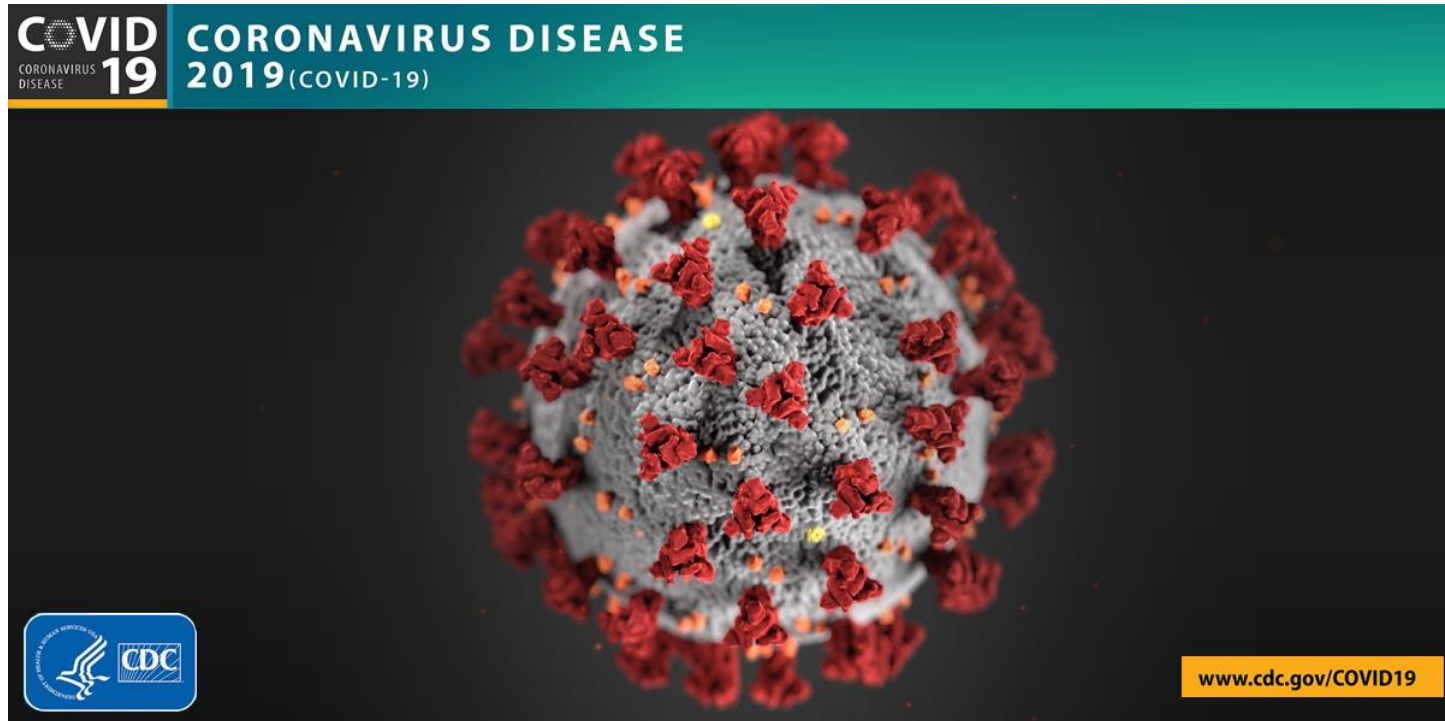


Paralelní a distribuované výpočty (B4B36PDV)



- Přihlašujte se pod Google účtem
- Pokud je to možné, používejte sluchátka
- Pokud nemluvíte, vypněte si mikrofon

Paralelní a distribuované výpočty (B4B36PDV)

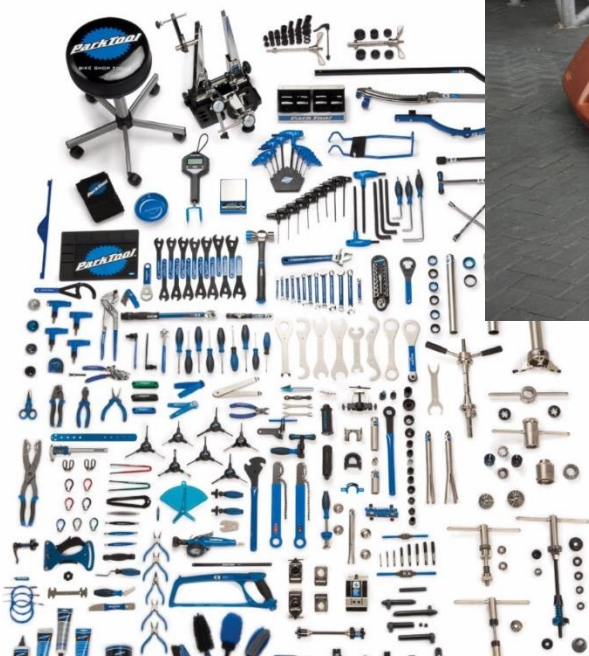
Branislav Bošanský, Michal Jakob

bosansky@fel.cvut.cz

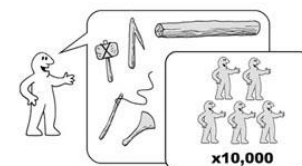
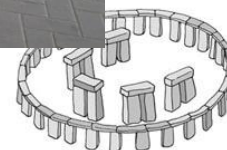
Artificial Intelligence Center
Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University in Prague

Dnešní přednáška

Motivace



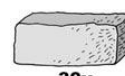
HĚNJ



80x



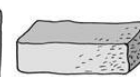
30x



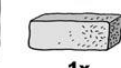
30x



10x



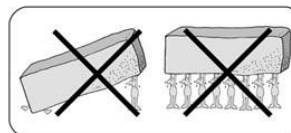
5x



1x



3x



Dnešní přednáška

Techniky paralelizace 2

Chci paralelizovat řadící algoritmus



Jak na to?

Paralelní řazení

Techniky rozděluj a panuj

- Potřebujeme seřadit pole (čísel) dané velikosti a využít k tomu techniky paralelizace
- Podobně jako při standardních řadících algoritmech – pro ilustraci myšlenek paralelizace se věnujeme i těm méně efektivním
- Připomenutí
 - 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);
    }
}
```

Paralelní řazení

- Který je nejjednodušší řadící algoritmus, který jste se naučili jako první?

Paralelní řazení

- Který je nejjednodušší řadící algoritmus, který jste se naučili jako první?
- Bubble Sort
 - porovnává dva za sebou následující prvky
 - pokud jsou v nesprávném pořadí, vymění je

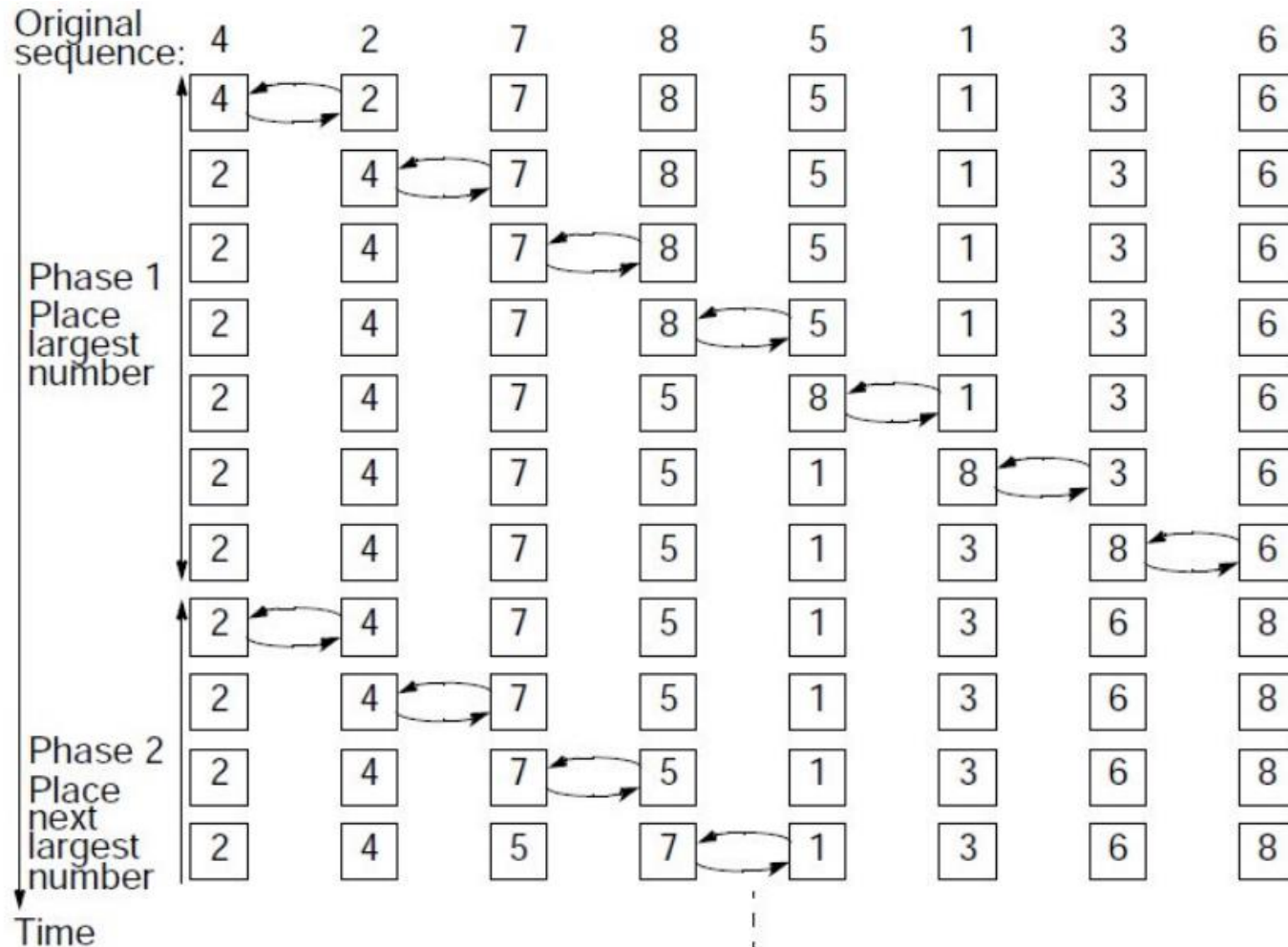
Paralelní řazení

- Který je nejjednodušší řadící algoritmus, který jste se naučili jako první?
- Bubble Sort
 - porovnává dva za sebou následující prvky
 - pokud jsou v nesprávném pořadí, vymění je

```
bool compare_swap(std::vector<int>& vector_to_sort, const int& val1, const int& val2) {  
    if (vector_to_sort[val1] > vector_to_sort[val2]) {  
        std::iter_swap(vector_to_sort.begin() + val1, vector_to_sort.begin() + val2);  
        return true;  
    }  
    return false;  
}  
  
void bubble(std::vector<int>& vector_to_sort, int from, int to) {  
    bool change = true;  
    while (change) {  
        change = false;  
        for (int i = from + 1; i < to; i++) {  
            change |= compare_swap(vector_to_sort, i - 1, i);  
        }  
    }  
}
```


Paralelní řazení

Bubble Sort



Paralelní řazení

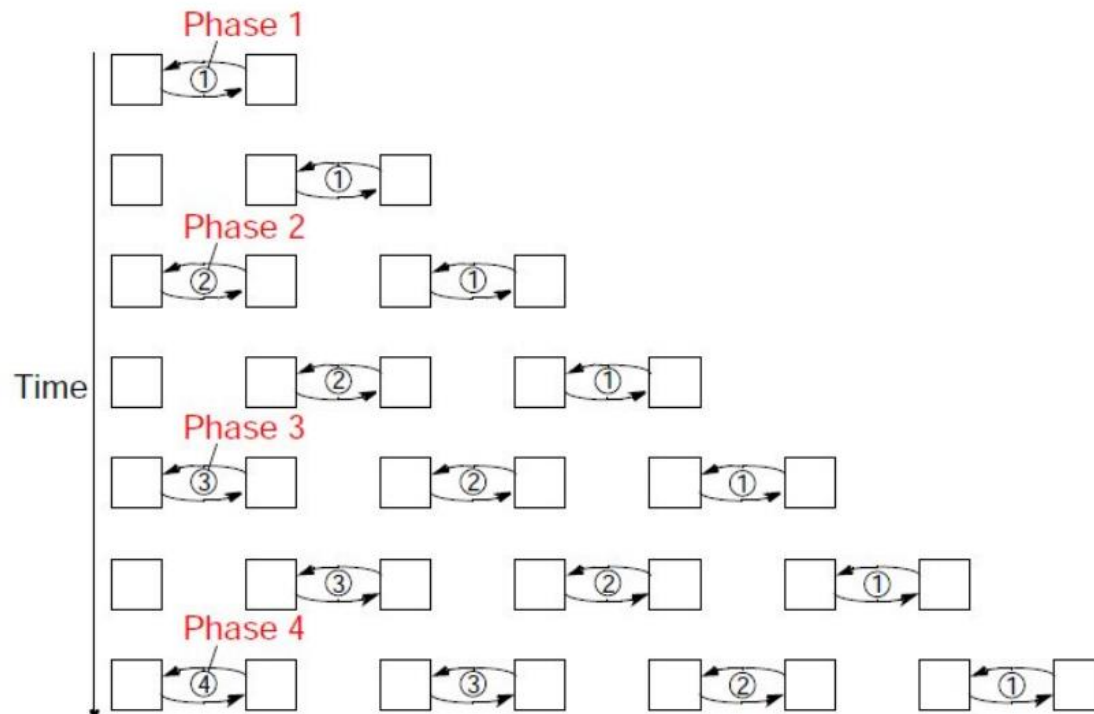
Bubble Sort

- Jak lze bubble sort paralelizovat?

Paralelní řazení

Bubble Sort

- Jak lze bubble sort paralelizovat?
- Varianta 1
 - Vzpomeňte si na paralelizaci úloh na CPU – pipelineing



Paralelní řazení

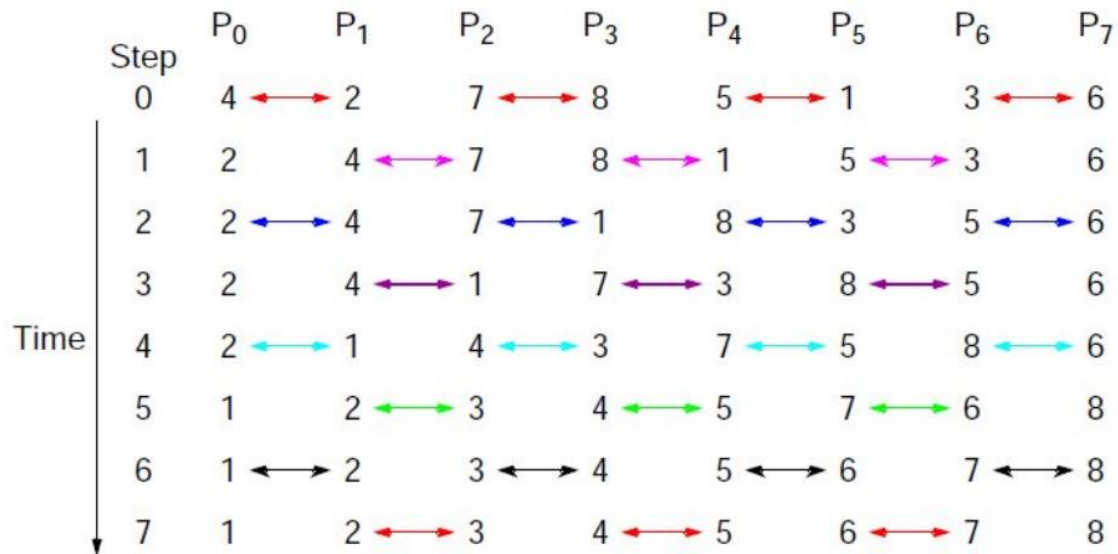
Bubble Sort

- Lze bubble sort paralelizovat ještě jinak?
- Které porovnání lze dělat paralelně bez konfliktu?

Paralelní řazení

Bubble Sort

- Lze bubble sort paralelizovat ještě jinak?
- Které porovnání lze dělat paralelně bez konfliktu?
 - Vzpomeňte si na dekompozici při výpočtu průměrů v matici
 - Porovnání dle lichých/sudých čísel



Paralelní řazení

Bubble Sort

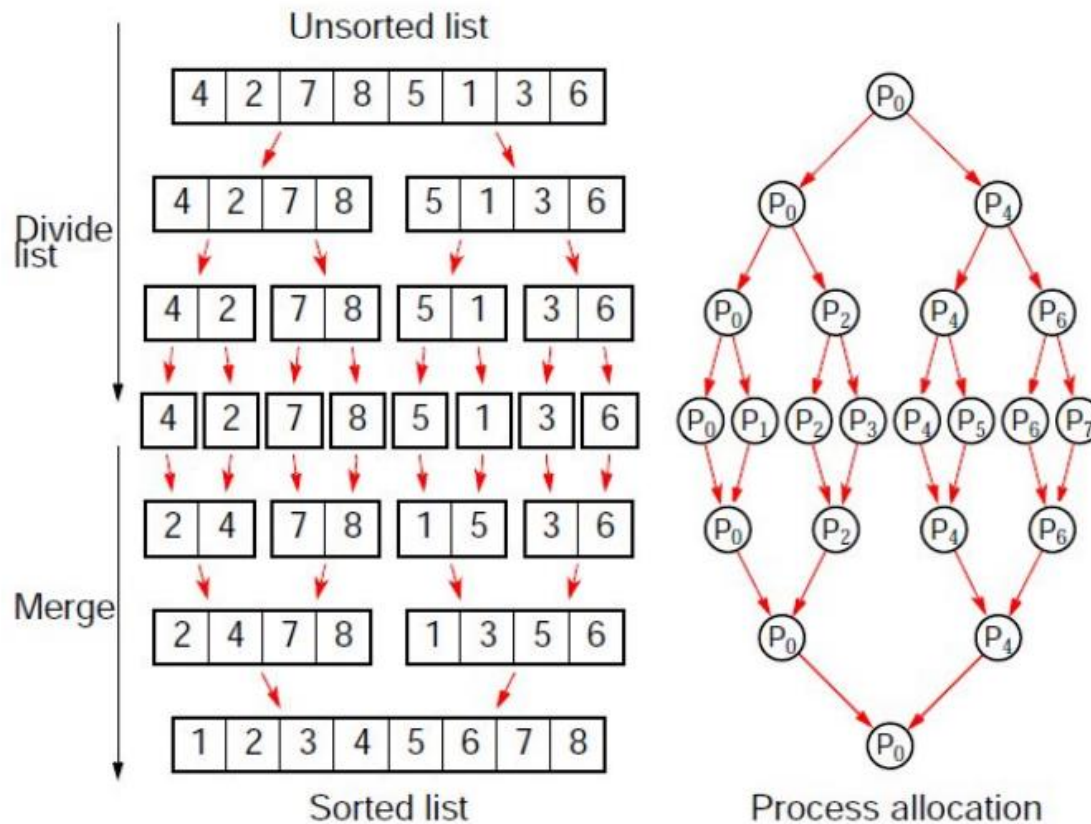
- Pro zvýšení paralelizace opět rozdělíme po blocích
- A můžeme paralelizovat

```
void parallel_bubble (std::vector<int>& vector_to_sort, unsigned int from, unsigned int to) {  
    while (change) {  
        change = false;  
        #pragma omp parallel for num_threads(thread_count) schedule(static) shared(vector_to_sort) reduction(|:change)  
        for (int i = from + 1; i < to; i += 2) {  
            change |= compare_swap(vector_to_sort, i - 1, i);  
        }  
  
        #pragma omp parallel for num_threads(thread_count) schedule(static) shared(vector_to_sort) reduction(|:change)  
        for (int i = from + 2; i < to; i += 2) {  
            change |= compare_swap(vector_to_sort, i - 1, i);  
        }  
    }  
}
```

Paralelní řazení

Merge Sort

- Jak paralelizujeme MergeSort?



Paralelní řazení

Merge Sort

- Jak paralelizujeme MergeSort?

```
void ms_serial(std::vector<int>& vector_to_sort, int from, int to) {
    if (to - from <= 1) {
        return;
    }
    int middle = (to - from)/2 + from;

    ms_serial(vector_to_sort, from, middle);
    ms_serial(vector_to_sort, middle, to);
    std::inplace_merge(vector_to_sort.begin()+from, vector_to_sort.begin()+middle, vector_to_sort.begin()+to);
}

void ms(std::vector<int>& vector_to_sort, int from, int to) {
    if (to - from <= base_size) {
        ms_serial(vector_to_sort, from, to);
        return;
    }
    int middle = (to - from)/2 + from;

    ms(vector_to_sort, from, middle);
    ms(vector_to_sort, middle, to);

    std::inplace_merge(vector_to_sort.begin()+from, vector_to_sort.begin()+middle, vector_to_sort.begin()+to);
}
```


Paralelní řazení

Merge Sort

- Jak paralelizujeme MergeSort?

```
void ms_serial(std::vector<int>& vector_to_sort, int from, int to) {
    if (to - from <= 1) {
        return;
    }
    int middle = (to - from)/2 + from;

    ms_serial(vector_to_sort, from, middle);
    ms_serial(vector_to_sort, middle, to);
    std::inplace_merge(vector_to_sort.begin()+from, vector_to_sort.begin()+middle, vector_to_sort.begin()+to);
}

void ms(std::vector<int>& vector_to_sort, int from, int to) {
    if (to - from <= base_size) {
        ms_serial(vector_to_sort, from, to);
        return;
    }
    int middle = (to - from)/2 + from;

    ms(vector_to_sort, from, middle);
    ms(vector_to_sort, middle, to);

    std::inplace_merge(vector_to_sort.begin()+from, vector_to_sort.begin()+middle, vector_to_sort.begin()+to);
}
```

Paralelní řazení

Merge Sort

- Která varianta je správná?

- A

```
#pragma omp task shared(vector_to_sort) firstprivate(from,middle)
ms(vector_to_sort, from, middle);
ms(vector_to_sort, middle, to);

#pragma omp taskwait
std::inplace_merge(vector_to_sort.begin()+from,vector_to_sort.begin()+middle,vector_to_sort.begin()+to);
```

- B

```
ms(vector_to_sort, from, middle);
#pragma omp task shared(vector_to_sort) firstprivate(from,middle)
ms(vector_to_sort, middle, to);

#pragma omp taskwait
std::inplace_merge(vector_to_sort.begin()+from,vector_to_sort.begin()+middle,vector_to_sort.begin()+to);
```

- C

```
#pragma omp task shared(vector_to_sort) firstprivate(from,middle)
ms(vector_to_sort, from, middle);
ms(vector_to_sort, middle, to);

std::inplace_merge(vector_to_sort.begin()+from,vector_to_sort.begin()+middle,vector_to_sort.begin()+to);
```



<https://goo.gl/a6BEMb>

Paralelní řazení

Merge Sort

- Jak paralelizujeme MergeSort?

```
void ms_serial(std::vector<int>& vector_to_sort, int from, int to) {
    if (to - from <= 1) {
        return;
    }
    int middle = (to - from)/2 + from;

    ms_serial(vector_to_sort, from, middle);
    ms_serial(vector_to_sort, middle, to);
    std::inplace_merge(vector_to_sort.begin()+from, vector_to_sort.begin()+middle, vector_to_sort.begin()+to);
}

void ms(std::vector<int>& vector_to_sort, int from, int to) {
    if (to - from <= base_size) {
        ms_serial(vector_to_sort, from, to);
        return;
    }
    int middle = (to - from)/2 + from;

    #pragma omp task shared(vector_to_sort) firstprivate(from, middle)
    ms(vector_to_sort, from, middle);

    ms(vector_to_sort, middle, to);

    #pragma omp taskwait
    std::inplace_merge(vector_to_sort.begin()+from, vector_to_sort.begin()+middle, vector_to_sort.begin()+to);
}
```

Paralelní řazení

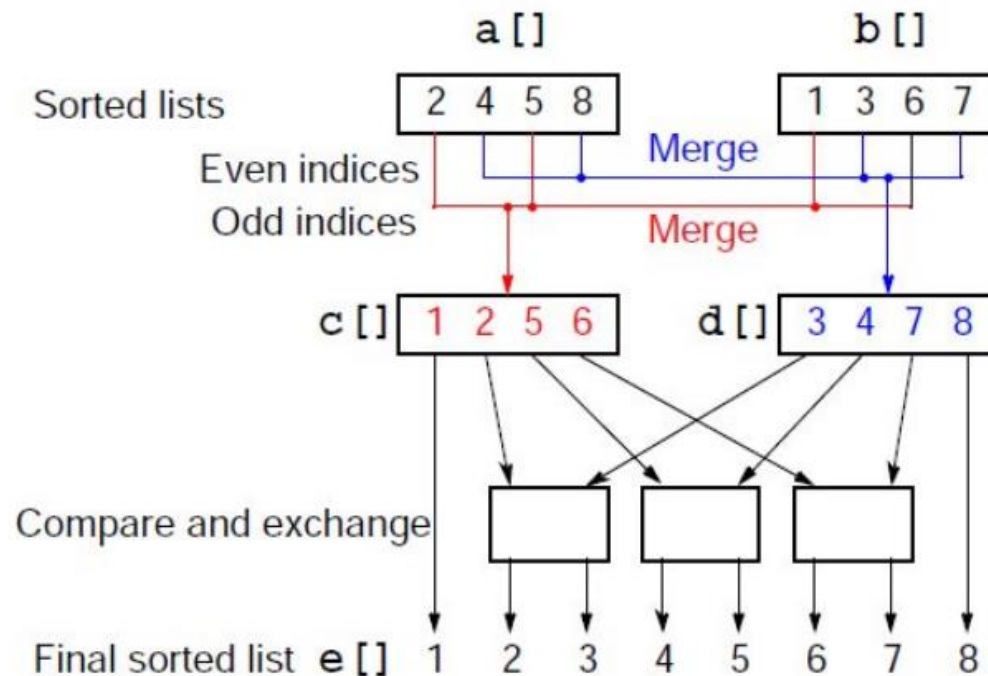
Merge Sort

- Lze merge sort paralelizovat lépe?

Paralelní řazení

Merge Sort

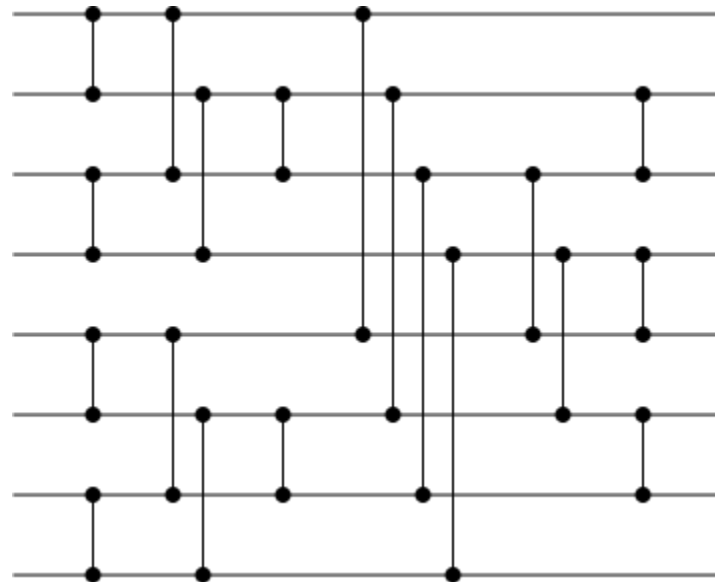
- Lze merge sort paralelizovat lépe?
- Také zde lze využít liché/sudé porovnání



Paralelní řazení

Odd-Even Merge Sort

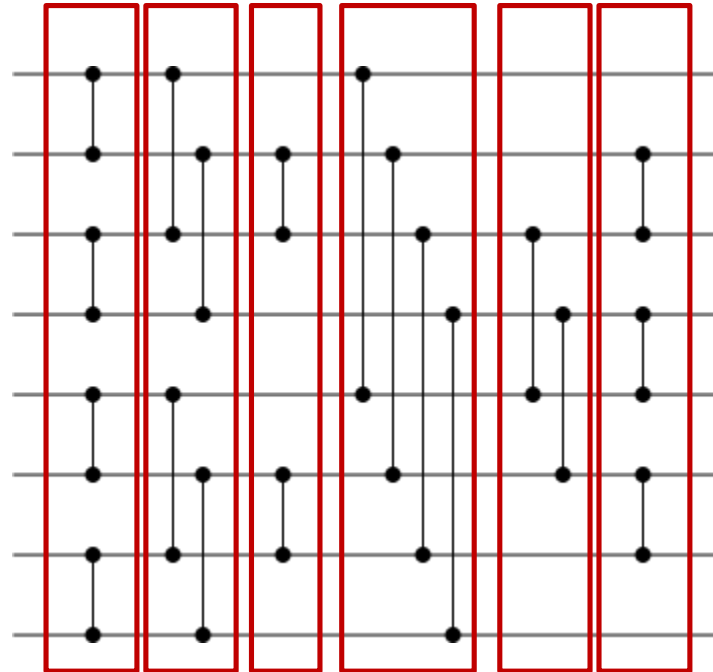
- Využíváme podobnou myšlenku jak v bubble sortu
 - Identifikujeme dvojice čísel, porovnání kterých lze dělat paralelně
- Jak to funguje?
 - Pro 8 prvků



Paralelní řazení

Odd-Even Merge Sort

- Využíváme podobnou myšlenku jak v bubble sortu
 - Identifikujeme dvojice čísel, porovnání kterých lze dělat paralelně
- Jak to funguje?
 - Pro 8 prvků

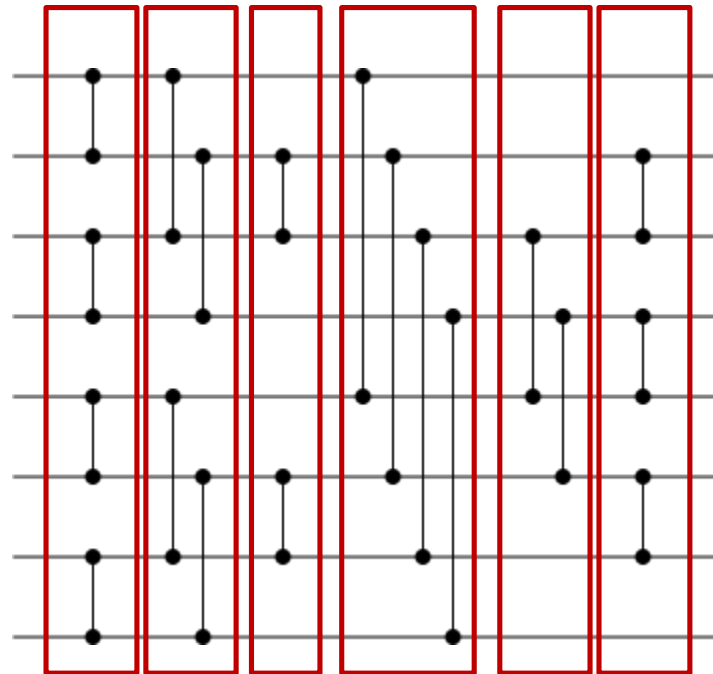


Paralelní řazení

Odd-Even Merge Sort

- Využíváme podobnou myšlenku jak v bubble sortu
 - Identifikujeme dvojice čísel, porovnání kterých lze dělat paralelně

- Jak to funguje?
 - Pro 8 prvků



- Obecně?

Paralelní řazení

Odd-Even Merge Sort

- Využíváme podobnou myšlenku jak v bubble sortu
 - Identifikujeme dvojice čísel, porovnání kterých lze dělat paralelně
- Jak to funguje?
 - Pro 8 prvků

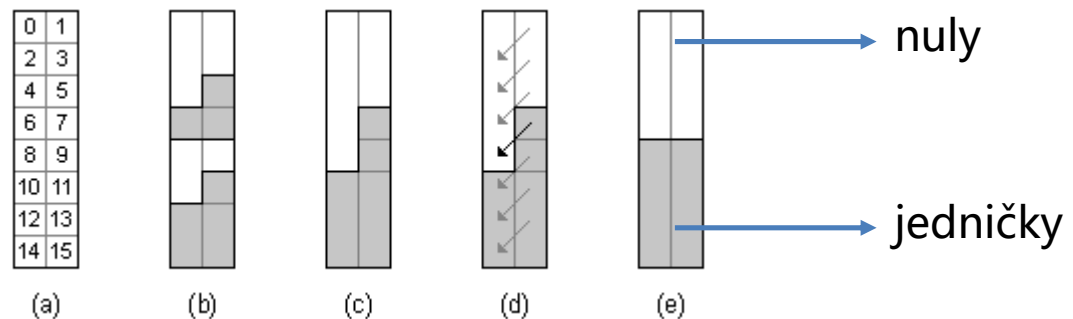
```
void odd-even-merge (std::vector<int>& vector_to_sort, int from, int to, int step) {  
    auto new_step = step * 2;  
    if (new_step < to - from) {  
        odd-even-merge(vector_to_sort, from, to, new_step);  
        odd-even-merge(vector_to_sort, from+step, to, new_step);  
        for (int i=from+step; i<to-step; i += new_step) {  
            compare_and_swap(vector_to_sort, i, i+step);  
        }  
    } else {  
        compare_and_swap(vector_to_sort, from, from+step);  
    }  
}
```

- Obecně?

Paralelní řazení

Odd-Even Merge Sort

- Proč to funguje?
 - Lze dokázat pomocí indukce a tzv. 0-1 principu
 - (pokud třídící síť dokáže setřídít libovolnou posloupnost nul a jedniček, dokáže setřídít libovolnou sekvenci libovolných celých čísel)
 - Předpokládejme (Indukční krok), že algoritmus funguje pro $n < k$



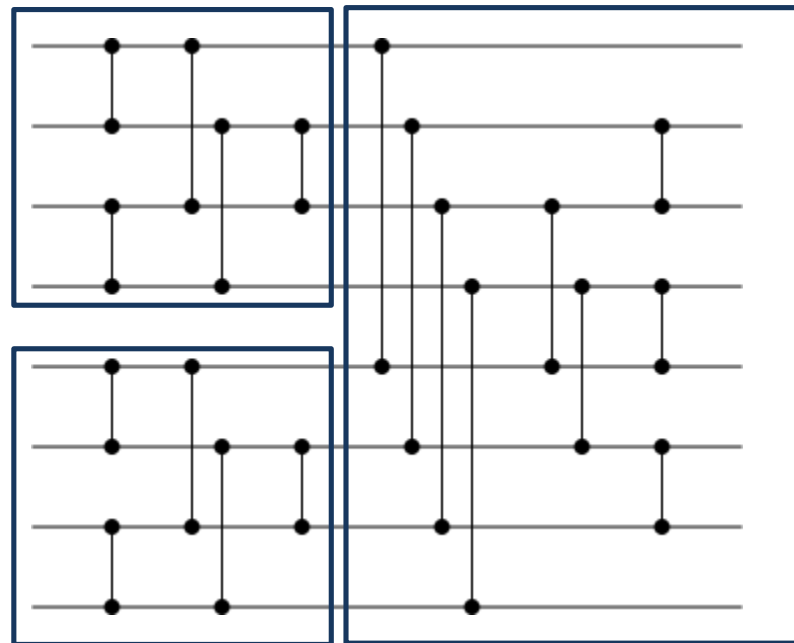
- Ideální pro HW/GPU implementaci
- $O(\log^2(n))$ paralelní výpočetní čas

Paralelní řazení

Bitonic Sort

- Bitonic Sort
- Vylepšená varianta Odd-Even Merge Sortu
- Pro paralelní slučování nepotřebujeme mít plně setříděné dílčí sekvence

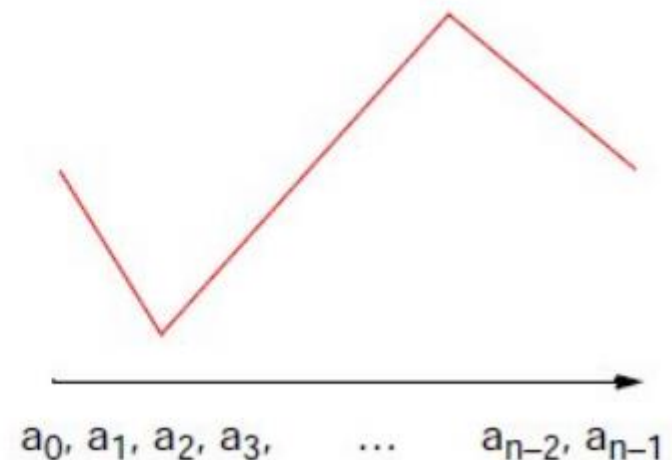
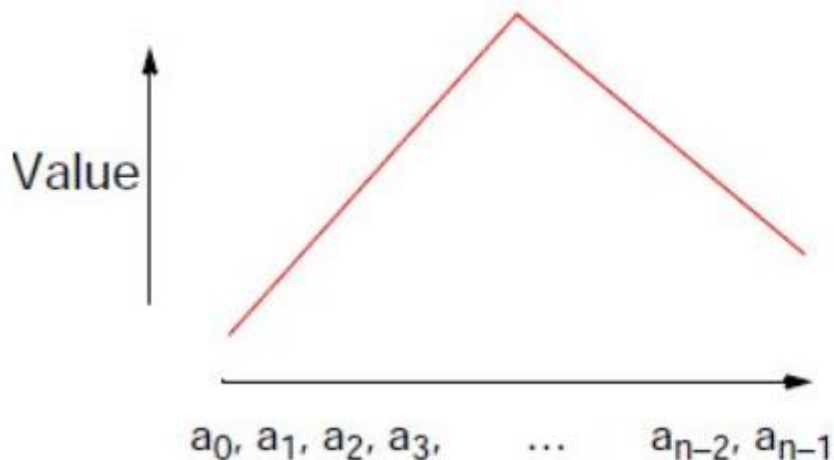
divide merge



Paralelní řazení

Bitonic Sort

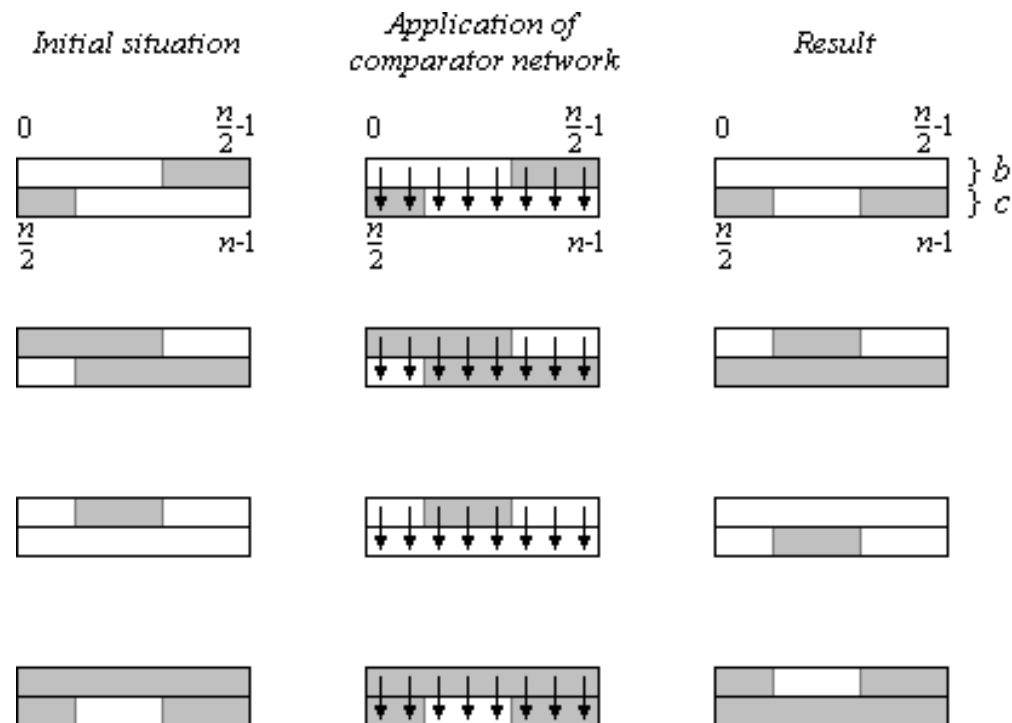
- Sekvence čísel je **bitonická**, pokud
 - obsahuje 2 podsekvence – jednu rostoucí a jednu klesající
 - tedy pro nějaké $(0 \leq i \leq n)$ platí
$$a_1 < a_2 < \dots < a_{i-1} < a_i > a_{i+1} > a_{i+2} > \dots > a_n$$
 - nebo lze dosáhnout této vlastnosti pomocí rotací prvků pole



Paralelní řazení

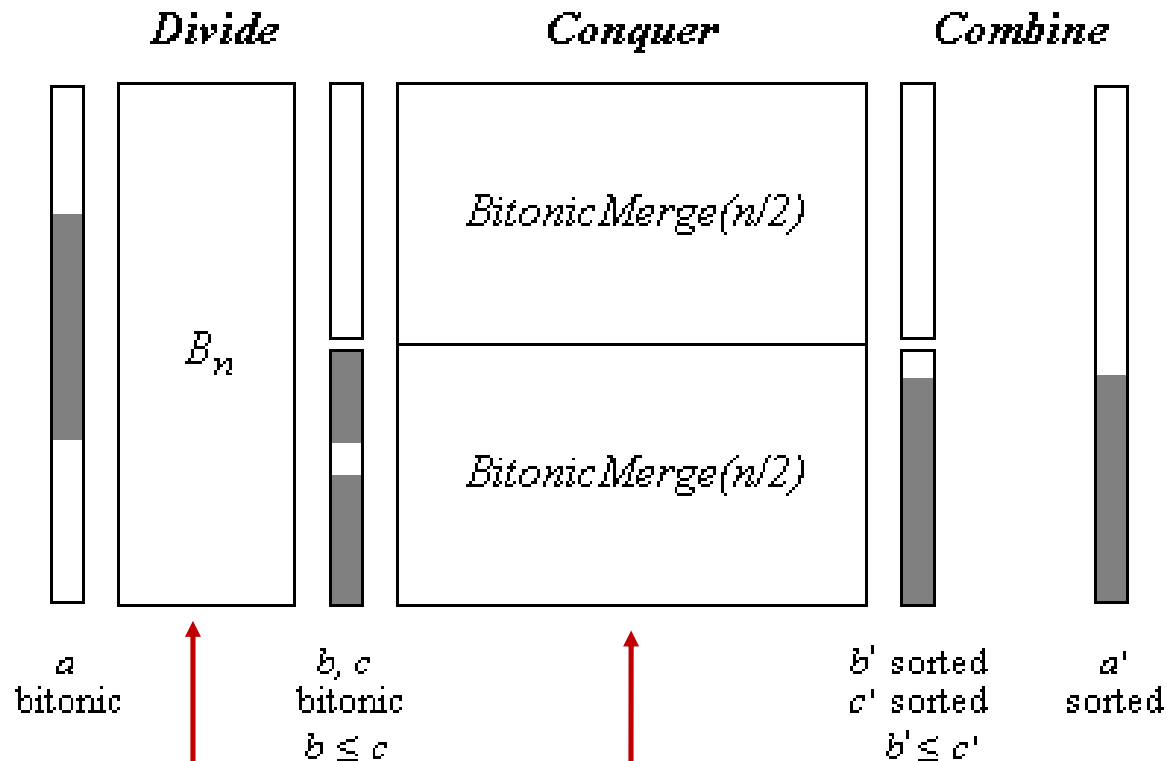
Bitonic Sort

- Párovým porovnáním prvků dvou částí bitonické sekvence dostaneme 2 bitonické sekvence



Paralelní řazení

Bitonic Sort



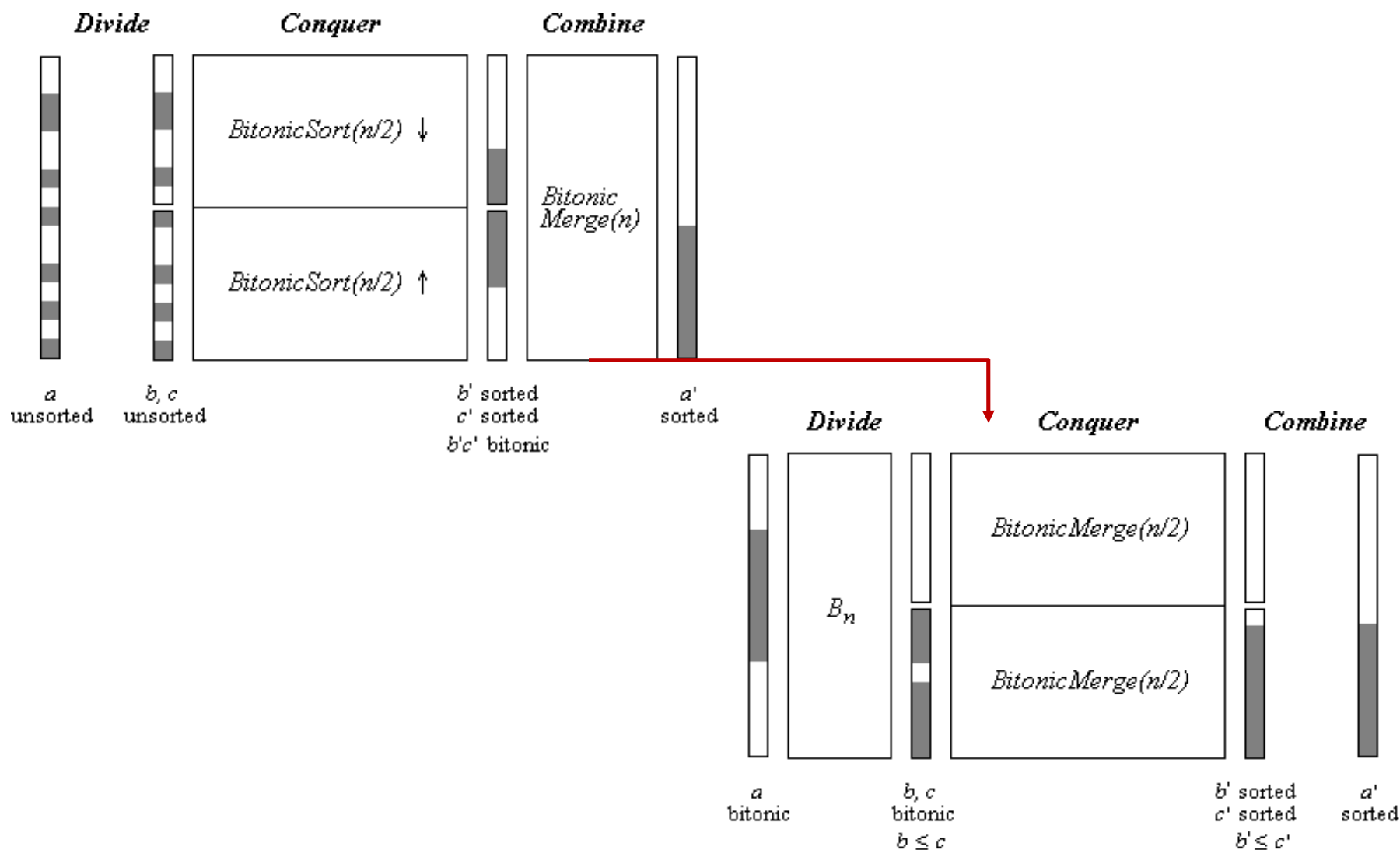
Párové porovnání

Rekurze

A co když není vstupní
sekvence bitonická?

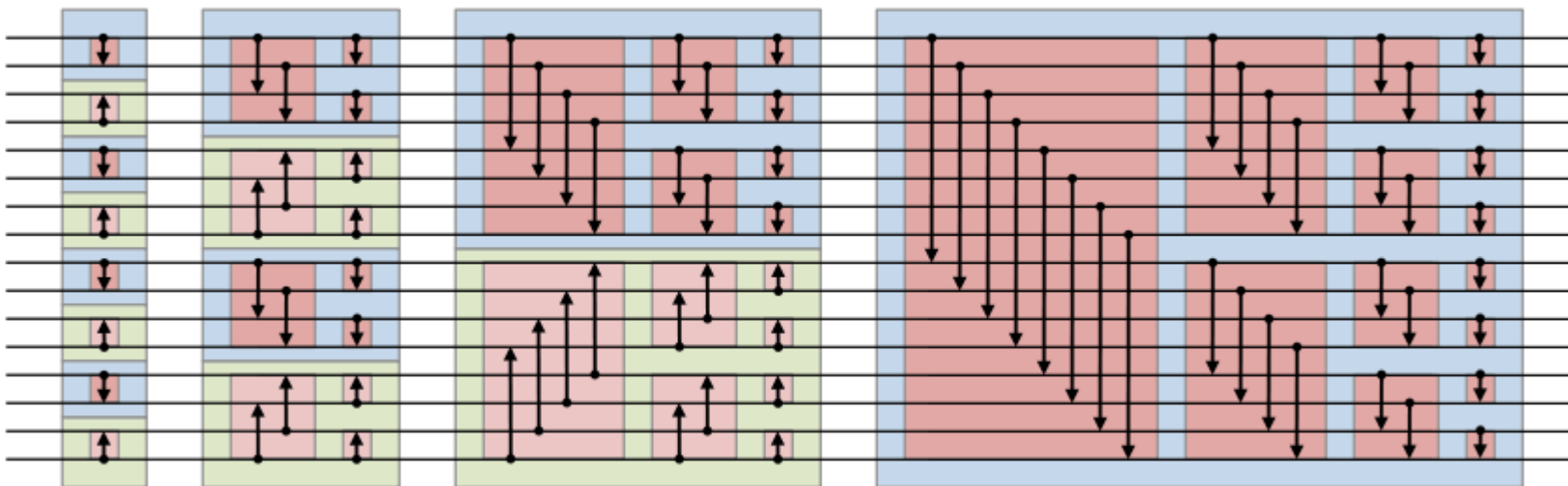
Paralelní řazení

Bitonic Sort



Paralelní řazení

Bitonic Sort



Paralelní řazení

Bitonic Sort

- Jak efektivně implementovat?
- Chceme provést vybranou skupinu porovnání zároveň
 - SIMD typ kroku – chci porovnání a případnou výměnu prvků na vícero datech současně
 - Ideální pro implementaci na GPUs
- Co když GPU nemáme?
- Můžeme využít vektorizaci na dnešních CPUs

Krátký úvod do vektorových instrukcí

- Myšlenka – jednotlivá čísla polí budeme representovat pomocí vektoru čísel
- Použitím přístupných datových struktur a metod řekneme procesoru, které operace se mohou vykonat paralelně (SIMD)


	Datové typy	
SSE	__m128	128 bitový vektor, obsahuje 4x float
	__m128d	128 bitový vektor, obsahuje 2x double
	__m128i	128 bitový vektor, obsahuje celá čísla
	__m256	256 bitový vektor, obsahuje 8x float
AVX	__m256d	256 bitový vektor, obsahuje 4x double
	__m256i	256 bitový vektor, obsahuje celá čísla
	...	

překládejte s přepínači **-march=native -mavx**

Krátký úvod do vektorových instrukcí

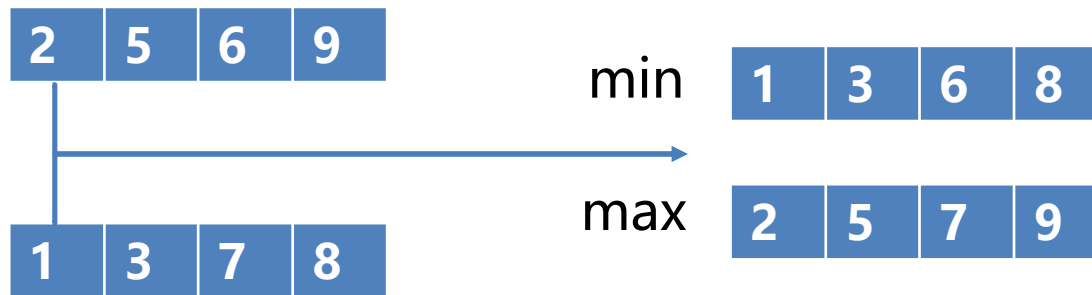
- Počet celých čísel záleží na typu
- Ve `__m256i` může být:
 - 32 char
 - 16 short
 - 8 int
 - 4 long

Datové typy	
<code>__m128</code>	128 bitový vektor, obsahuje 4x float
<code>__m128d</code>	128 bitový vektor, obsahuje 2x double
<code>__m128i</code>	128 bitový vektor, obsahuje celá čísla
<code>__m256</code>	256 bitový vektor, obsahuje 8x float
<code>__m256d</code>	256 bitový vektor, obsahuje 4x double
<code>__m256i</code>	256 bitový vektor, obsahuje celá čísla
...	

- Pole je reprezentované v obráceném pořadí
- `float[4] {0f,1f,2f,3f}` 

Krátký úvod do vektorových instrukcí

- Klíčová operace v bitonic sortu
 - Párové porovnání (a případná výměna) prvků v dvou polích
- Jak na to?



- porovnání 4 (8) čísel se provede zároveň

Krátký úvod do vektorových instrukcí

Párové porovnání 2 vektorů

```
#include ...
#include <immintrin.h>

int main() {

    std::vector<int> vec1 = std::vector<int>(SIZE);
    std::vector<int> vec2 = std::vector<int>(SIZE);

    for (int i=0; i<SIZE; i++) {
        vec1[i] = rand() % 10000;
        vec2[i] = rand() % 10000;
    }

    auto t_start = std::chrono::high_resolution_clock::now();

    __m256i v1;
    __m256i v2;
    __m256i r1,r2;

    for (int i=0; i<SIZE; i += 8) {
        v1 = _mm256_loadu_si256((__m256i *) &vec1[i]);
        v2 = _mm256_loadu_si256((__m256i *) &vec2[i]);
        r1 = _mm256_min_epi32(v1, v2);
        r2 = _mm256_max_epi32(v1, v2);
        _mm256_storeu_si256((__m256i *) &vec1[i], r1);
        _mm256_storeu_si256((__m256i *) &vec2[i], r2);
    }

    auto t_end = std::chrono::high_resolution_clock::now();
    double elapsed = std::chrono::duration<double, std::milli>(t_end-t_start).count()/1000.0;

    std::cout << "compared in " << elapsed << " s" << std::endl;
    return 0;
}
```

Krátký úvod do vektorových instrukcí

Párové porovnání 2 vektorů

```
#include ...
#include <immintrin.h>

int main() {

    std::vector<int> vec1 = std::vector<int>(SIZE);
    std::vector<int> vec2 = std::vector<int>(SIZE);

    for (int i=0; i<SIZE; i++) {
        vec1[i] = rand() % 10000;
        vec2[i] = rand() % 10000;
    }

    auto t_start = std::chrono::high_resolution_clock::now();

    __m256i v1;
    __m256i v2;
    __m256i r1,r2;

    for (int i=0; i<SIZE; i += 8) {
        v1 = _mm256_loadu_si256((__m256i *) &vec1[i]);
        v2 = _mm256_loadu_si256((__m256i *) &vec2[i]);
        r1 = _mm256_min_epi32(v1, v2);
        r2 = _mm256_max_epi32(v1, v2);
        _mm256_storeu_si256((__m256i *) &vec1[i], r1);
        _mm256_storeu_si256((__m256i *) &vec2[i], r2);
    }

    auto t_end = std::chrono::high_resolution_clock::now();
    double elapsed = std::chrono::duration<double, std::milli>(t_end-t_start).count()/1000.0;

    std::cout << "compared in " << elapsed << " s" << std::endl;
    return 0;
}
```

Načtení dat do
vektorové
reprezentace

Krátký úvod do vektorových instrukcí

Párové porovnání 2 vektorů

```
#include ...
#include <immintrin.h>

int main() {

    std::vector<int> vec1 = std::vector<int>(SIZE);
    std::vector<int> vec2 = std::vector<int>(SIZE);

    for (int i=0; i<SIZE; i++) {
        vec1[i] = rand() % 10000;
        vec2[i] = rand() % 10000;
    }

    auto t_start = std::chrono::high_resolution_clock::now();

    __m256i v1;
    __m256i v2;
    __m256i r1,r2;

    for (int i=0; i<SIZE; i += 8) {
        v1 = _mm256_loadu_si256((__m256i *) &vec1[i]);
        v2 = _mm256_loadu_si256((__m256i *) &vec2[i]);
        r1 = _mm256_min_epi32(v1, v2);
        r2 = _mm256_max_epi32(v1, v2);
        _mm256_storeu_si256((__m256i *) &vec1[i], r1);
        _mm256_storeu_si256((__m256i *) &vec2[i], r2);
    }

    auto t_end = std::chrono::high_resolution_clock::now();
    double elapsed = std::chrono::duration<double, std::milli>(t_end-t_start).count()/1000.0;

    std::cout << "compared in " << elapsed << " s" << std::endl;
    return 0;
}
```

2 operace
porovnání
(lze i pomocí
jednoho
porovnání a 1 xor)

Krátký úvod do vektorových instrukcí

Párové porovnání 2 vektorů

```
#include ...
#include <immintrin.h>

int main() {

    std::vector<int> vec1 = std::vector<int>(SIZE);
    std::vector<int> vec2 = std::vector<int>(SIZE);

    for (int i=0; i<SIZE; i++) {
        vec1[i] = rand() % 10000;
        vec2[i] = rand() % 10000;
    }

    auto t_start = std::chrono::high_resolution_clock::now();

    __m256i v1;
    __m256i v2;
    __m256i r1,r2;

    for (int i=0; i<SIZE; i += 8) {
        v1 = _mm256_loadu_si256((__m256i *) &vec1[i]);
        v2 = _mm256_loadu_si256((__m256i *) &vec2[i]);
        r1 = _mm256_min_epi32(v1, v2);
        r2 = _mm256_max_epi32(v1, v2);
        _mm256_storeu_si256((__m256i *) &vec1[i], r1);
        _mm256_storeu_si256((__m256i *) &vec2[i], r2);
    }

    auto t_end = std::chrono::high_resolution_clock::now();
    double elapsed = std::chrono::duration<double, std::milli>(t_end-t_start).count()/1000.0;

    std::cout << "compared in " << elapsed << " s" << std::endl;
    return 0;
}
```

Uložení výsledků

Krátký úvod do vektorových instrukcí

- Párové porovnání (a případná výměna) prvků v poli (např. sousedních)
 - $x_0 ? x_1$ (a případně vyměnit tak, aby x_0 byla menší)
 - $x_2 ? x_3$ (a případně vyměnit tak, aby x_2 byla menší)

x_3	x_2	x_1	x_0
2	5	6	9

- Jak na to?

Krátký úvod do vektorových instrukcí

- Párové porovnání (a případná výměna) prvků v poli (např. sousedních)
 - $x_0 ? x_1$ (a případně vyměnit tak, aby x_0 byla menší)
 - $x_2 ? x_3$ (a případně vyměnit tak, aby x_2 byla menší)

x3	x2	x1	x0
2	5	6	9

- Jak na to?
- Vytvoříme posunutou kopii vektoru a porovnáme

Krátký úvod do vektorových instrukcí

- Vytvoříme posunutou kopii vektoru a porovnáme
- Metoda alignr – spojí 2 vektory, umožní posun a ořízne

doplníme

				x3	x2	x1	x0
0	0	0	0	2	5	6	9

Krátký úvod do vektorových instrukcí

- Vytvoříme posunutou kopii vektoru a porovnáme
- Metoda alignr – spojí 2 vektory, umožní posun a ořízne

doplníme

				x3	x2	x1	x0
0	0	0	0	2	5	6	9

Shift 1 do prava

					x3	x2	x1
0	0	0	0	0	2	5	6

Krátký úvod do vektorových instrukcí

- Vytvoříme posunutou kopii vektoru a porovnáme
- Metoda alignr – spojí 2 vektory, umožní posun a ořízne

doplníme

				x3	x2	x1	x0
0	0	0	0	2	5	6	9

Shift 1 do prava

					x3	x2	x1
0	0	0	0	0	2	5	6

ořízneme

	x3	x2	x1
0	2	5	6

Porovnáme s
původním
vektorem

x3	x2	x1	x0
2	5	6	9

Krátký úvod do vektorových instrukcí

- Vytvoříme posunutou kopii vektoru a porovnáme
- Metoda alignr – spojí 2 vektory, umožní posun a ořízne

doplníme

				x3	x2	x1	x0
0	0	0	0	2	5	6	9

Shift 1 do prava

					x3	x2	x1
0	0	0	0	0	2	5	6

ořízneme

	x3	x2	x1
0	2	5	6
x3	x2	x1	x0
2	5	6	9

Porovnáme s
původním
vektorem



Zajímají nás
minima

x3	x2	x1	x0
0	2	5	6

Krátký úvod do vektorových instrukcí

- Vytvoříme posunutou kopii vektoru a porovnáme

Zajímají nás
minima

3	2	1	0
0	2	5	6

Krátký úvod do vektorových instrukcí

- Vytvoříme posunutou kopii vektoru a porovnáme

Zajímají nás
minima

3	2	1	0
0	2	5	6



Ale pouze sudé pozice

- $\min(x_0, x_1)$ je na pozici 0
- $\min(x_2, x_3)$ je na pozici 2
- ...

Vynulujeme pomocí masky

3	2	1	0
0	2	0	6

Krátký úvod do vektorových instrukcí

- Vytvoříme posunutou kopii vektoru a porovnáme

Zajímají nás
minima

3	2	1	0
0	2	5	6



Ale pouze sudé pozice

- $\min(x_0, x_1)$ je na pozici 0
- $\min(x_2, x_3)$ je na pozici 2
- ...

Vynulujeme pomocí masky

3	2	1	0
0	2	0	6

Podobně získáme
maxima z
porovnání a
uložíme je na
liché pozice



3	2	1	0
5	0	9	0

Výsledek je OR
těchto vektorů

Krátký úvod do vektorových instrukcí

```
int SIZE = 8;
std::vector<int> vec1 = std::vector<int>(SIZE);

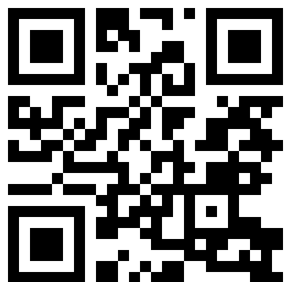
for (int i=0; i<SIZE; i++) {
    vec1[i] = rand() % 10000;
    std::cout << vec1[i] << " ";
}

__m128i mask_llhllhh = _mm_set_epi32(0xffffffff,0,0xffffffff,0);
__m128i mask_hhllhll = _mm_set_epi32(0,0xffffffff,0,0xffffffff);

__m128i v1;
__m128i v2;
__m128i r1,r2;

for (int i=0; i<SIZE; i += 4) {
    v1 = _mm_loadu_si128((__m128i *) &vec1[i]);
    v2 = _mm_alignr_epi8(_mm_setzero_si128(), v1 ,1*4);
    r1 = _mm_min_epi32(v1, v2);
    r1 = _mm_and_si128(r1,mask_hhllhll);
    v2 = _mm_alignr_epi8(v1, _mm_setzero_si128(),3*4);
    r2 = _mm_max_epi32(v1, v2);
    r2 = _mm_and_si128(r2,mask_llhllhh);
    r1 = _mm_or_si128(r1,r2);
    _mm_storeu_si128((__m128i *) &vec1[i], r1);
}
```

Hlasování



<https://goo.gl/a6BEMb>