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

Z předminulé přednášky

Co to vlastně dělá?

```cpp
#include <algorithm>
#include <chrono>
#include <execution>
#include <iostream>
#include <random>
#include <vector>

using namespace std::chrono;

int main() {
    const int N = 1000000;
    std::vector<int> v(N);
    std::mt19937 rng;
    rng.seed(std::random_device()());
    std::uniform_int_distribution<int> dist(0, 255);
    std::generate(begin(v), end(v), [&]() { return dist(rng); });

    auto start = high_resolution_clock::now();
    std::sort(std::execution::par, begin(v), end(v));
    auto finish = high_resolution_clock::now();
    auto duration = duration_cast<milliseconds>(finish - start);

    std::cout << "\nElapsed time = " << duration.count() << " ms\n";
    return 0;
}
```

Nad kódem z předminulé přednášky

Co to vlastně dělá?

- https://github.com/gcc-mirror/gcc/blob/16e2427f50c208dfe07d07f18009969502c25dc8/libstdc%2B%2B-v3/include/pstl/algorithm_impl.h#L2107

```
template <class _ExecutionPolicy, typename _RandomAccessIterator, typename _Compare, typename _LeafSort>
void
__parallel_stable_sort(_ExecutionPolicy&&, _RandomAccessIterator __xs, _RandomAccessIterator __xe, _Compare __comp,
                       _LeafSort __leaf_sort, std::size_t __nsort = 0)
{
    tbb::this_task_arena::isolate([=, &__nsort]() {
        //sorting based on task tree and parallel merge
        typedef typename std::iterator_traits<_RandomAccessIterator>::value_type _ValueType;
        typedef typename std::iterator_traits<_RandomAccessIterator>::difference_type _DifferenceType;
        const _DifferenceType __n = __xe - __xs;
        if (__nsort == __n)
            __nsort = 0; // 'partial_sort' becames 'sort'

        const _DifferenceType __sort_cut_off = _PSTL_STABLE_SORT_CUT_OFF;
        if (__n > __sort_cut_off)
        {
            __buffer<_ValueType> __buf(__n);
            __root_task<__stable_sort_func<_RandomAccessIterator, _ValueType*, _Compare, _LeafSort>> __root{
                    __xs, __xe, __buf.get(), true, __comp, __leaf_sort, __nsort, __xs, __buf.get()};
            __task::spawn_root_and_wait(__root);
            return;
        }
        //serial sort
        __leaf_sort(__xs, __xe, __comp);
    });
}
```

Nad kódem z předminulé přednášky

Co to vlastně dělá?

- https://godbolt.org/

```cpp
#include <algorithm>
#include <chrono>
#include <execution>
#include <iostream>
#include <random>
#include <vector>
using namespace std::chrono;
int main() {
    const int N = 1000000;
    std::vector<int> v(N);
    std::mt19937 rng;
    rng.seed(std::random_device()());
    std::uniform_int_distribution<int> dist(0, 255);
    std::generate(begin(v), end(v), [&]() { return dist(rng); });
    auto start = high_resolution_clock::now();
    std::sort(std::execution::par, begin(v), end(v));
  // std::reduce(std::execution::par, begin(v), end(v), 0.0, std::p
auto finish = high_resolution_clock::now();
auto duration = duration_cast<milliseconds>(finish - start);
std::cout << "\nElapsed time = " << duration.count() << " ms\n";
return 0; }
```

Nad kódem z předminulé přednášky

Co to vlastně dělá?

- https://github.com/gcc-mirror/gcc/blob/16e2427f50c208dfe07d07f18009969502c25dc8/libstdc%2B%2B-v3/include/pstl/parallel_backend_tbb.h#L1157
- "sorting based on task tree and parallel merge"
- tbb::task_scheduler_init init(...);
- std::thread::hardware_concurrency()
- std::hardware_constructive_interference_size; https://en.cppreference.com/w/cpp/thread/hardware_destructive_interference_size

Z minulé přednášky

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

```
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);
    }
}
```

Z minulé přednášky

Techniky rozděluj a panuj

- Intel Develop se chlubí následující variantou:
- Třícestné quicksort
- Kontrola setřízení
- taskgroup
- task untied mergeable

```
template<class RanIt, class _Pred>
void qsort3w(RanIt _First, RanIt _Last, _Pred compare) {
    if (_First >= _Last) return;

    std::size_t _Size = 0L;
    g_depth++;
    if ((_Size = std::distance(_First, _Last)) > 0) {
        RanIt _LeftIt = _First, _RightIt = _Last;
        bool is_swapped_left = false, is_swapped_right = false;
        typename std::iterator_traits<RanIt>::value_type _Pivot = *_First;

        RanIt _FwdIt = _First + 1;
        while (_FwdIt <= _RightIt) {
            if (compare(*_FwdIt, _Pivot)) {
                is_swapped_left = true;
                std::iter_swap(_LeftIt, _FwdIt);
                _LeftIt++;
                _FwdIt++;
            } else if (compare(_Pivot, *_FwdIt)) {
                is_swapped_right = true;
                std::iter_swap(_RightIt, _FwdIt);
                _RightIt--;
            } else _FwdIt++;
        }

        if (_Size >= cutoff) {
#pragma omp taskgroup
            {
#pragma omp task untied mergeable
                if ((std::distance(_First, _LeftIt) > 0) && (is_swapped_left))
                    qsort3w(_First, _LeftIt - 1, compare);

#pragma omp task untied mergeable
                if ((std::distance(_RightIt, _Last) > 0) && (is_swapped_right))
                    qsort3w(_RightIt + 1, _Last, compare);
            }
        } else {
#pragma omp task untied mergeable
            {
                if ((std::distance(_First, _LeftIt) > 0) && is_swapped_left)
                    qsort3w(_First, _LeftIt - 1, compare);

                if ((std::distance(_RightIt, _Last) > 0) && is_swapped_right)
                    qsort3w(_RightIt + 1, _Last, compare);
            }
        }
    }
```

https://software.intel.com/content/www/us/en/develop/articles/an-efficient-parallel-three-way-quicksort-using-intel-c-compiler-and-openmp-45-library.html

Nad kódem z minulé přednášky

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

```
template<class ForwardIt>
void quicksort(ForwardIt first, ForwardIt last) {
    if (first == last) return;
    std::size_t distance = std::distance(first, last);
    auto pivot = *std::next(first, distance / 2);
    ForwardIt middle1; ForwardIt middle2;
    if (distance < threshold) {
        middle1 = std::partition(std::execution::seq, first, last, [pivot](const auto &em) { return em < pivot; });
        middle2 = std::partition(std::execution::seq, middle1, last, [pivot](const auto &em) { return !(pivot < em);
});
    } else {
        middle1 = std::partition(std::execution::par, first, last, [pivot](const auto &em) { return em < pivot; });
        middle2 = std::partition(std::execution::par, middle1, last, [pivot](const auto &em) { return !(pivot < em);
});
    }
    quicksort(first, middle1);
    quicksort(middle2, last);
}
```

Dnešní přednáška

Jak kód vylepšovat?

ARTICLE

Samplesort: A Sampling Approach to Minimal Storage Tree Sorting

Authors: W. D. Frazer, A. C. McKellar Authors Info & Affiliations

Publication: Journal of the ACM • July 1970 • https://doi.org/10.1145/321592.321600

intel Intrinsics Guide

The Intel Intrinsics Guide is an interactive reference tool for Intel intrinsic instructions, which are C style functions that provide access to many Intel instructions - including Intel® SSE, AVX, AVX-512, and more - without the need to write assembly code.

Technologies

- MMX
- SSE
- SSE2
- SSE3
- SSSE3
- SSE4.1
- SSE4.2
- AVX
- AVX2
- FMA
- AVX-512
- KNC

_mm_search

```
void _mm_2intersect_epi32 (__m128i a, __m128i b, __mmask8* k1, __mmask8* k2)                vp2intersectd
void _mm256_2intersect_epi32 (__m256i a, __m256i b, __mmask8* k1, __mmask8* k2)             vp2intersectd
void _mm512_2intersect_epi32 (__m512i a, __m512i b, __mmask16* k1, __mmask16* k2)           vp2intersectd
void _mm_2intersect_epi64 (__m128i a, __m128i b, __mmask8* k1, __mmask8* k2)                vp2intersectq
void _mm256_2intersect_epi64 (__m256i a, __m256i b, __mmask8* k1, __mmask8* k2)             vp2intersectq
void _mm512_2intersect_epi64 (__m512i a, __m512i b, __mmask8* k1, __mmask8* k2)             vp2intersectq
__m512i _mm512_4dpwssd_epi32 (__m512i src, __m512i a0, __m512i a1, __m512i a2, __m512i a3,  vp4dpwssd
__m128i * b)
__m512i _mm512_mask_4dpwssd_epi32 (__m512i src, __mmask16 k, __m512i a0, __m512i a1, __m512i a2,  vp4dpwssd
```

Dnešní přednáška

Techniky paralelizace 2

Chci paralelizovat řadící algoritmus

Jak na 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

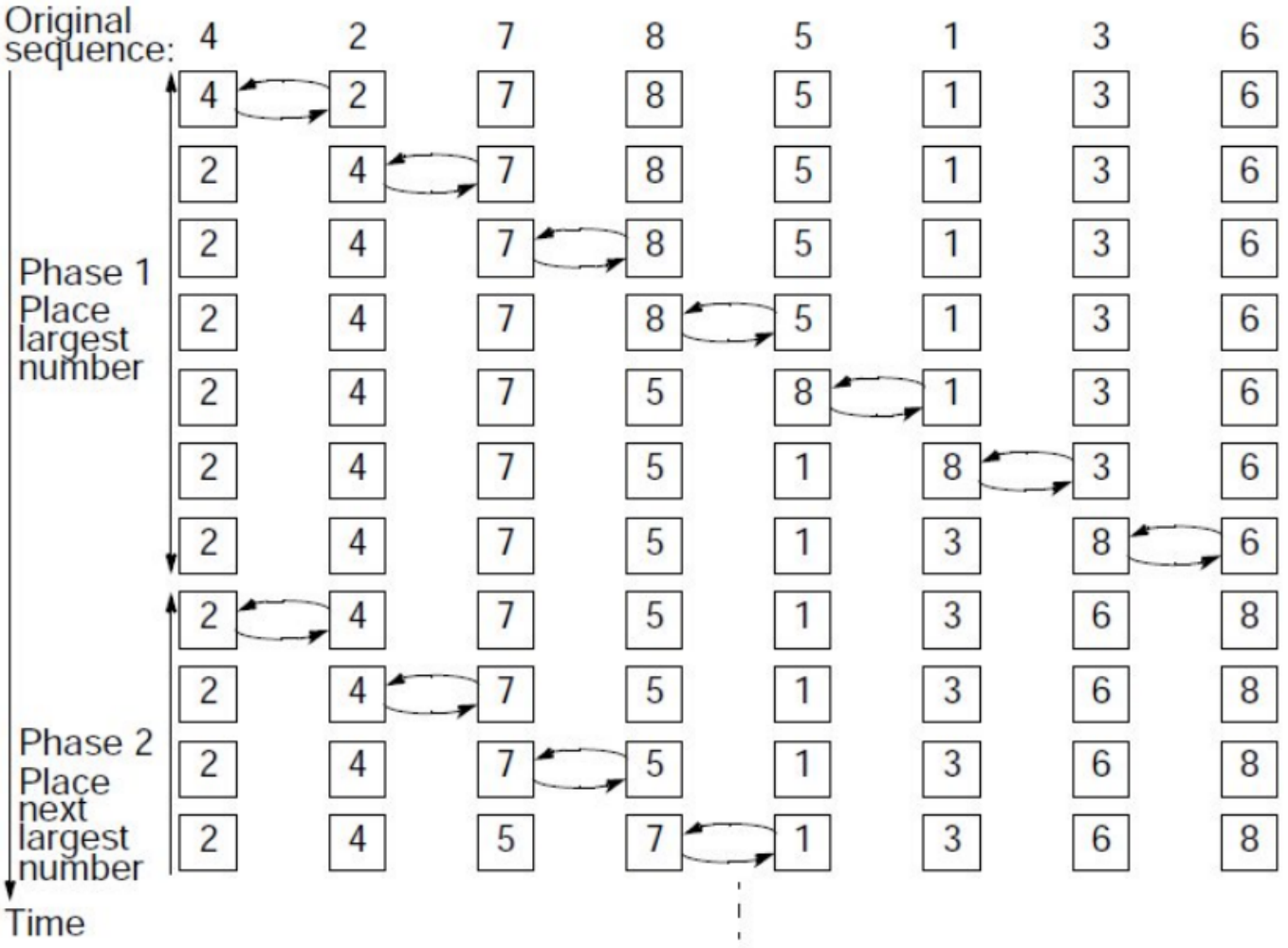
- Který je nejjednodušší řadící algoritmus, který jste se naučili jako první?

- 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

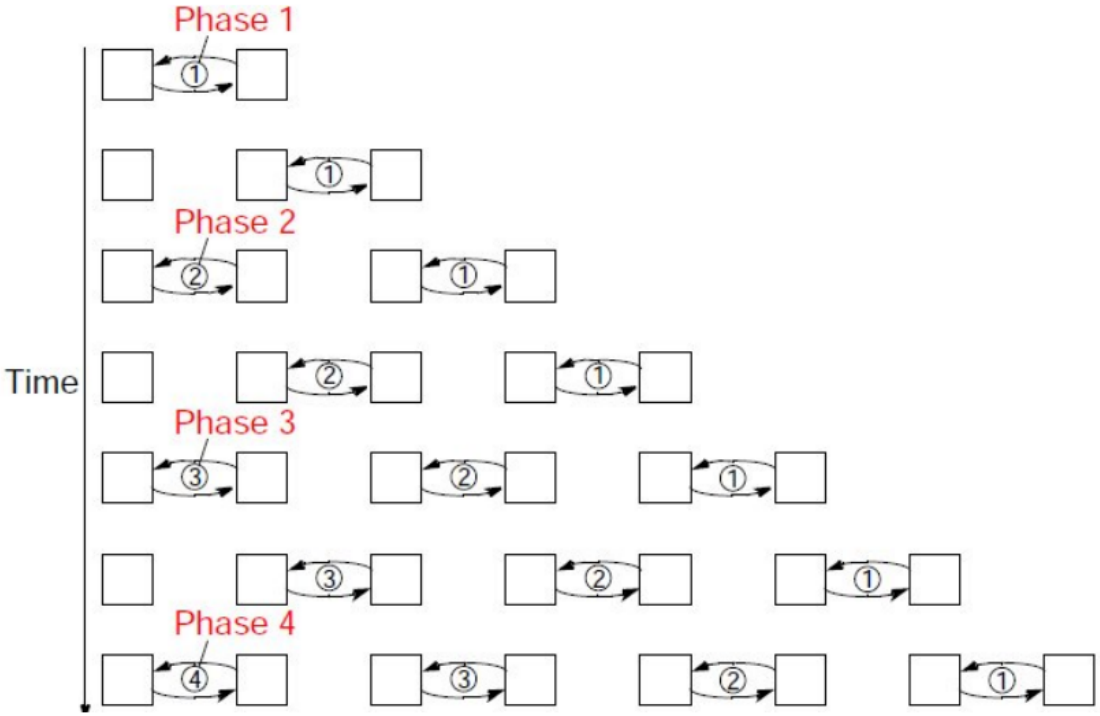
Bubble Sort

- 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

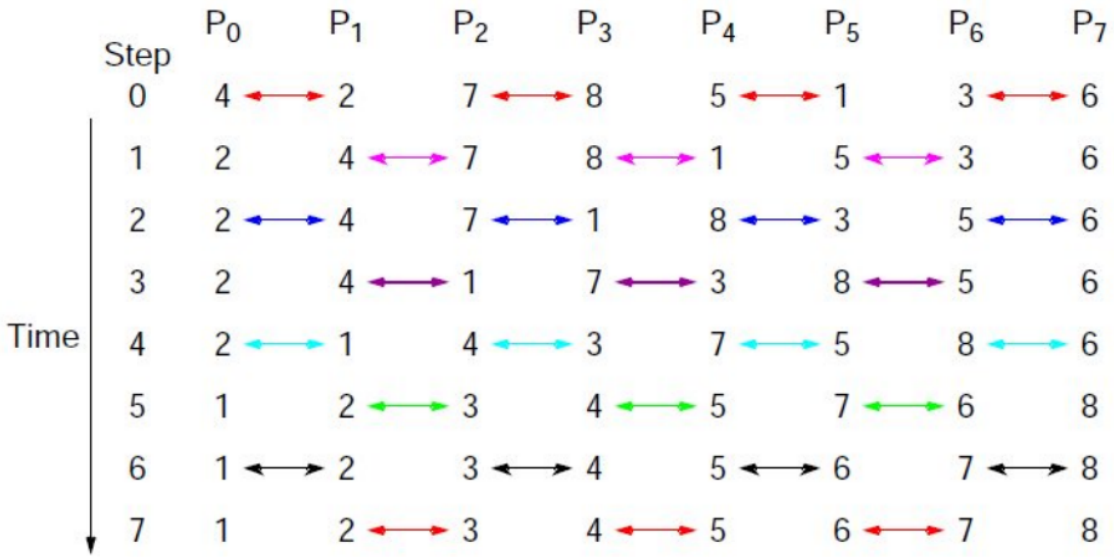
Bubble Sort

- 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 průměrování okolních prvků matice
  - 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

Co se skutečně používá

Algoritmy založené porovnání: multi-way mergesort, samplesort

- GCC C++ STL sort: multi-way mergesort
- GCC C++ STL stable_sort: quicksort
- GCC C++ PSTL, Intel Thread Building Blocks: "sorting based on task tree and parallel merge"
- Boost: Samplesort ad. https://www.boost.org/doc/libs/develop/libs/sort/doc/html/sort/parallel.html

Vektorizované algoritmy založené na výběru minima a maxima

Algoritmy založené na hashování

Paralelní řazení

Merge Sort

RESEARCH-ARTICLE

Efficient implementation of sorting on multi-core SIMD CPU architecture

Authors: Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, Pradeep Dubey Authors Info & Affiliations

Publication: Proceedings of the VLDB Endowment • August 2008 • https://doi.org/10.14778/1454159.1454171

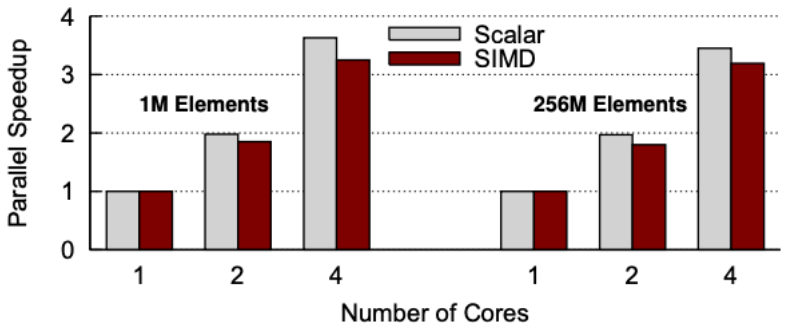
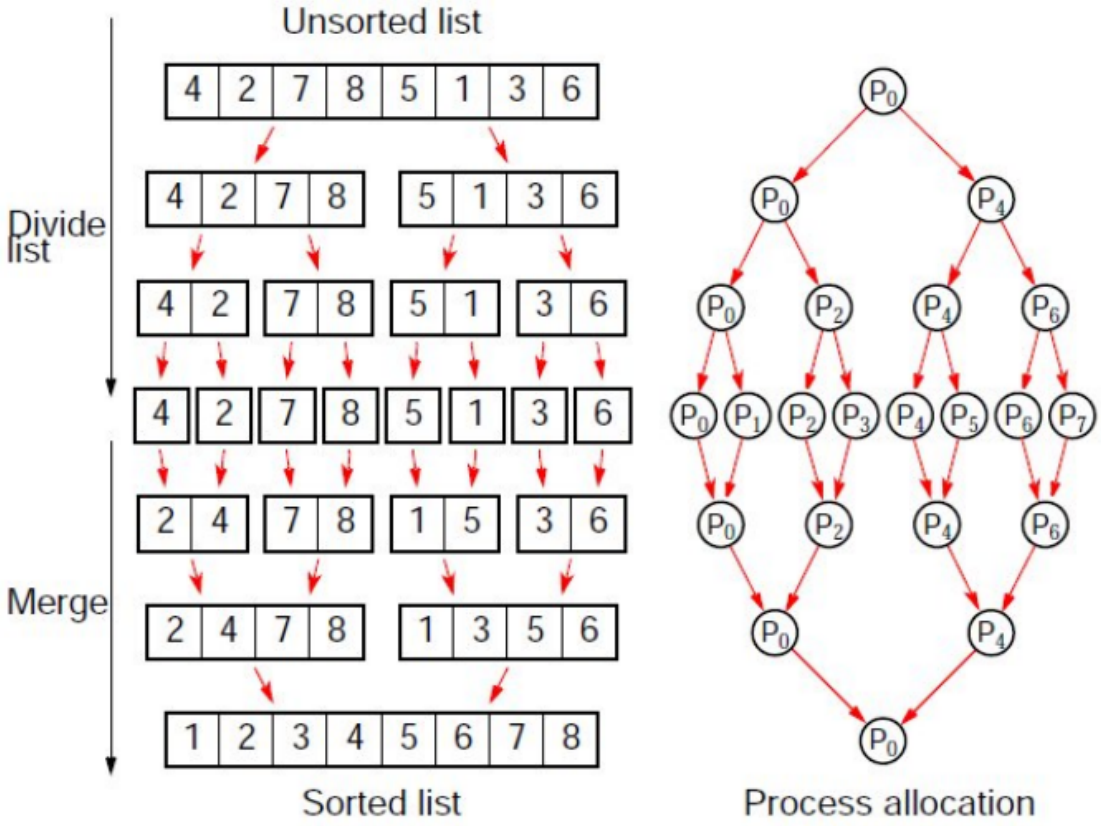


Figure 8: Parallel performance of the scalar and SIMD implementations.

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);
```

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

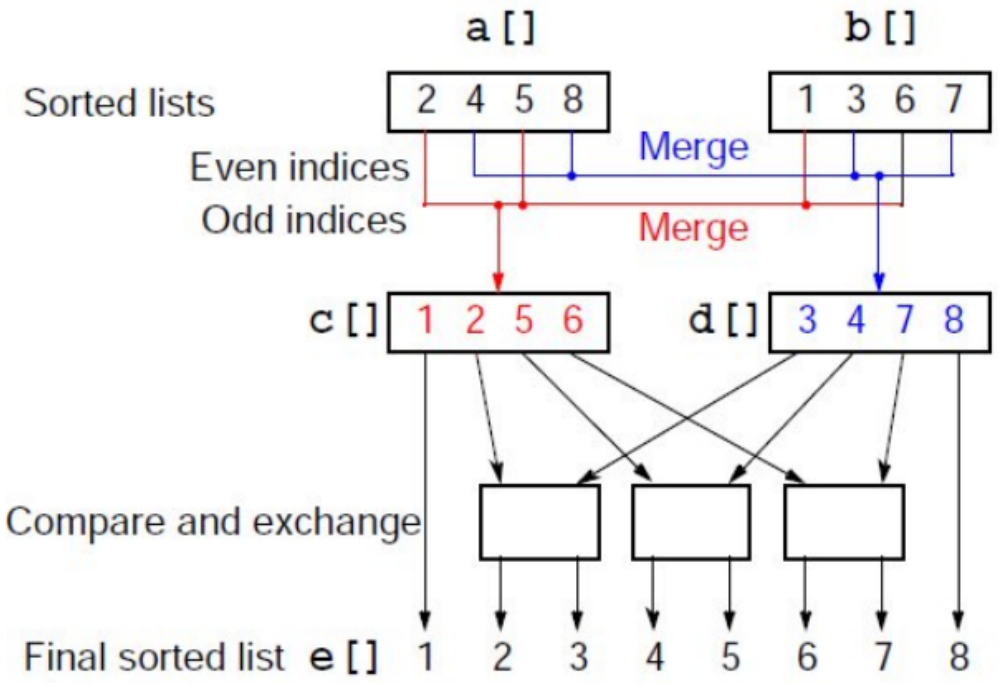
Merge Sort

- Lze merge sort paralelizovat lépe?

Paralelní řazení

Merge Sort

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

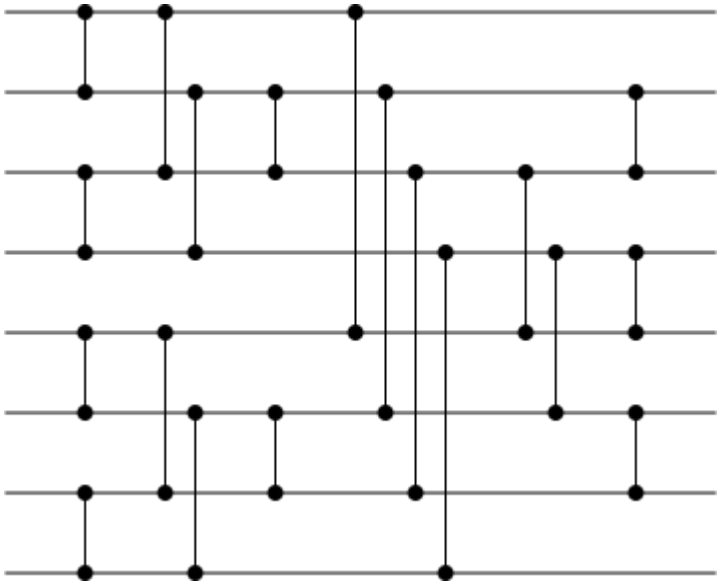


Paralelní řazení

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

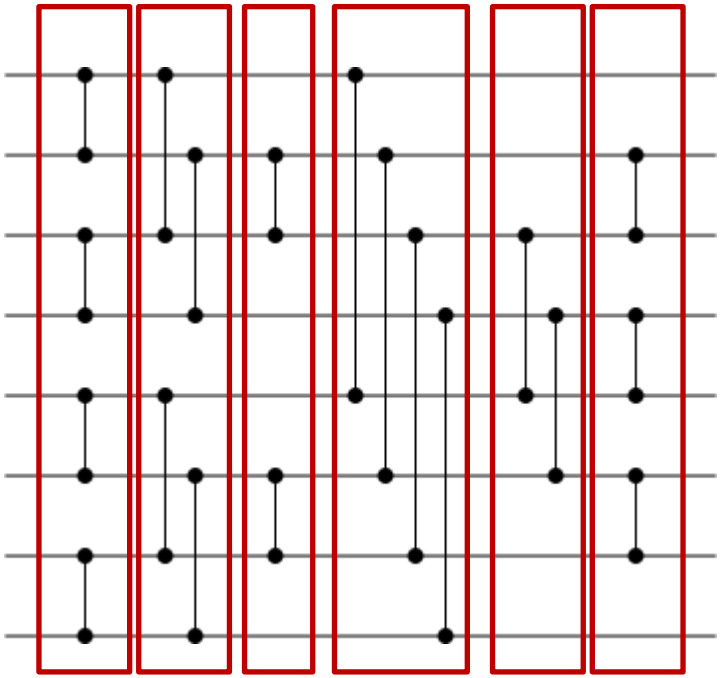


Paralelní řazení

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

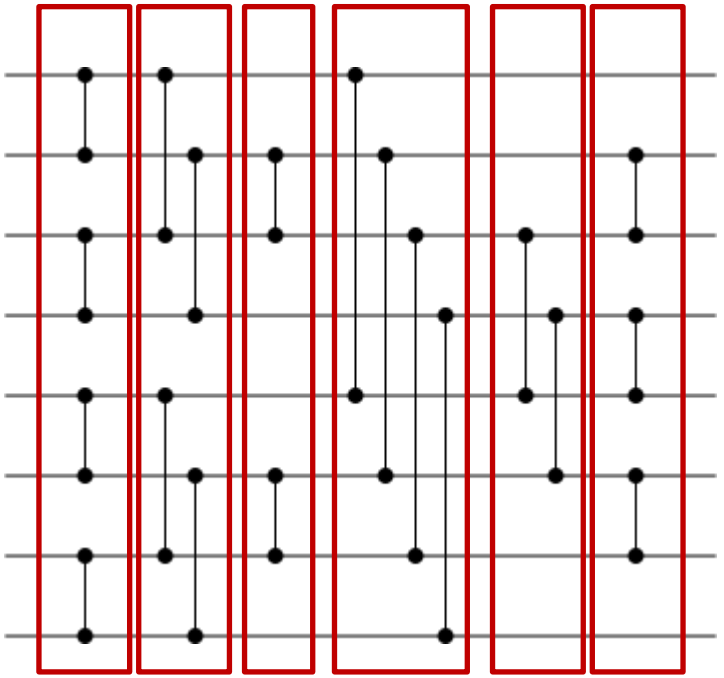


Paralelní řazení

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

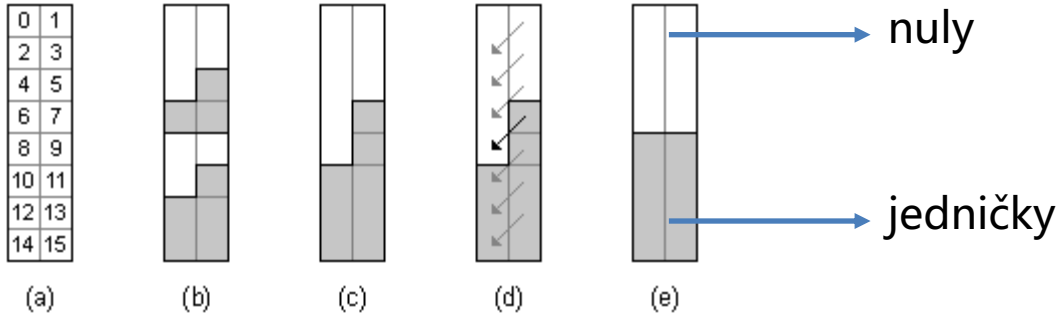
```
void odd-even-merge (std::vector<int>& vector_to_sort, int from, int to, int step) {
    auto new_step = step * 2;
    if (new_step < to - from) {
        odd-even-merge(vector_to_sort,from,to,new_step);
        odd-even-merge(vector_to_sort,from+step,to,new_step);
        for (int i=from+step; i<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 setřídit libovolnou posloupnost nul a jedniček, dokáže setřídit libovolnou sekvenci libovolných celých čísel)
  - Předpokládejme (Indukční krok), že algoritmus funguje pro n<k

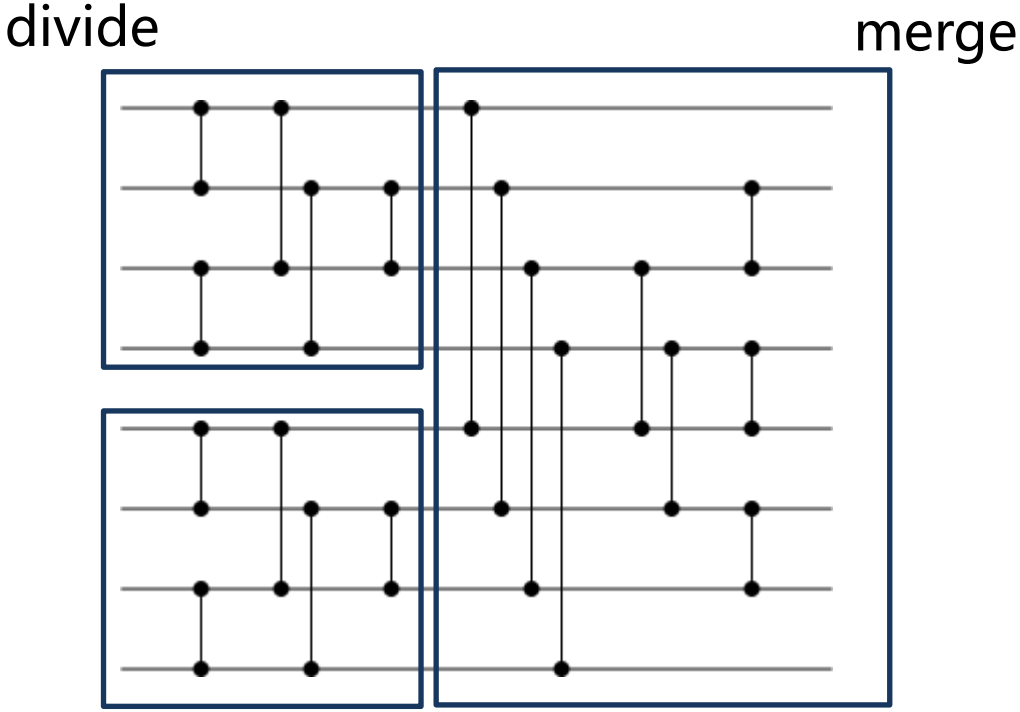


- Ideální pro HW/GPU implementaci
- $O(\log^2(n))$ paralelní výpočetní čas

Paralelní řazení

Bitonic Sort

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

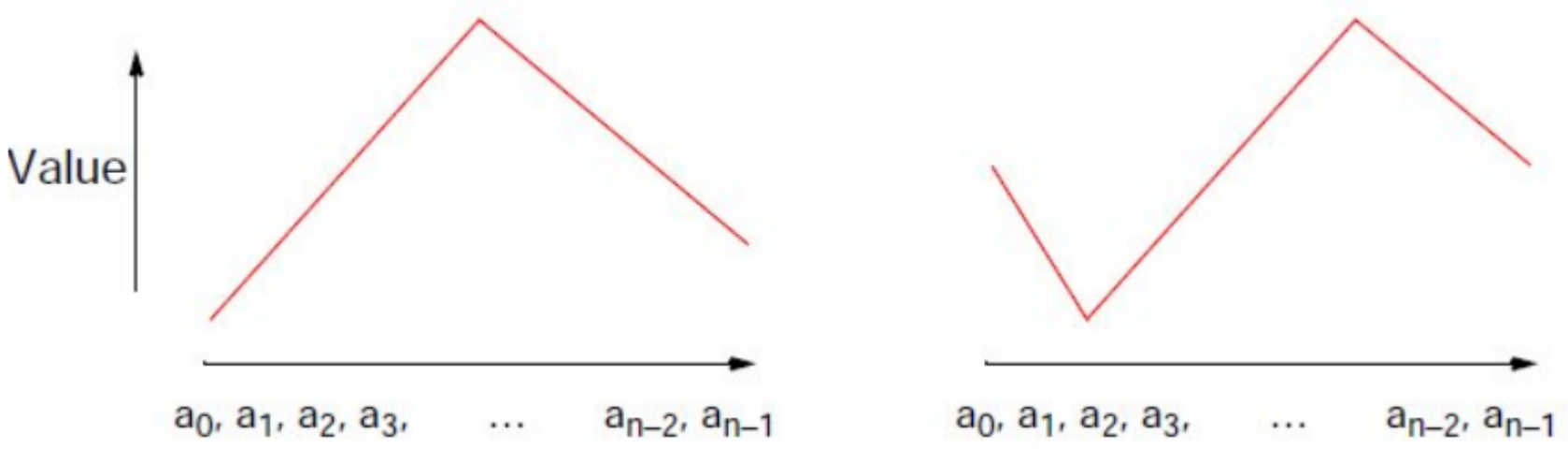


Paralelní řazení

Bitonic Sort

- Sekvence čísel je **bitonická**, pokud
  - obsahuje 2 podsekvence – jednu rostoucí a jednu klesající
    - tedy pro nějaké ($0 \leq i \leq n$) platí

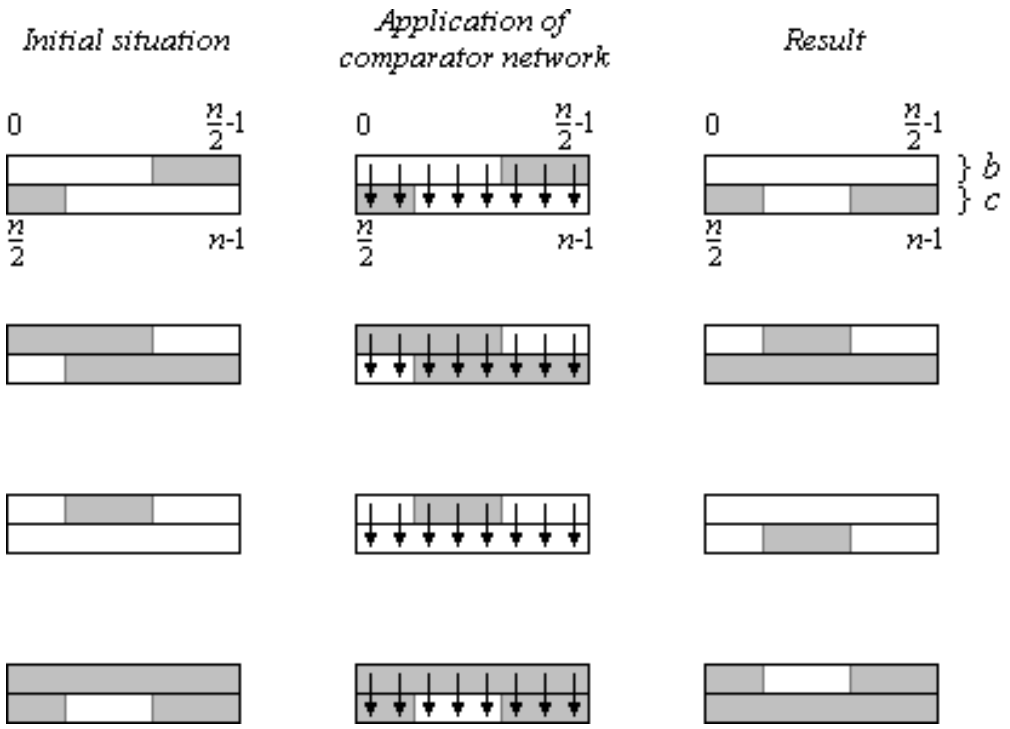
      $a1 < a2 < \ldots < ai-1 < ai > ai+1 > ai+2 > \ldots > an$
  - nebo lze dosáhnout této vlastnosti pomocí rotací prvků pole



Paralelní řazení

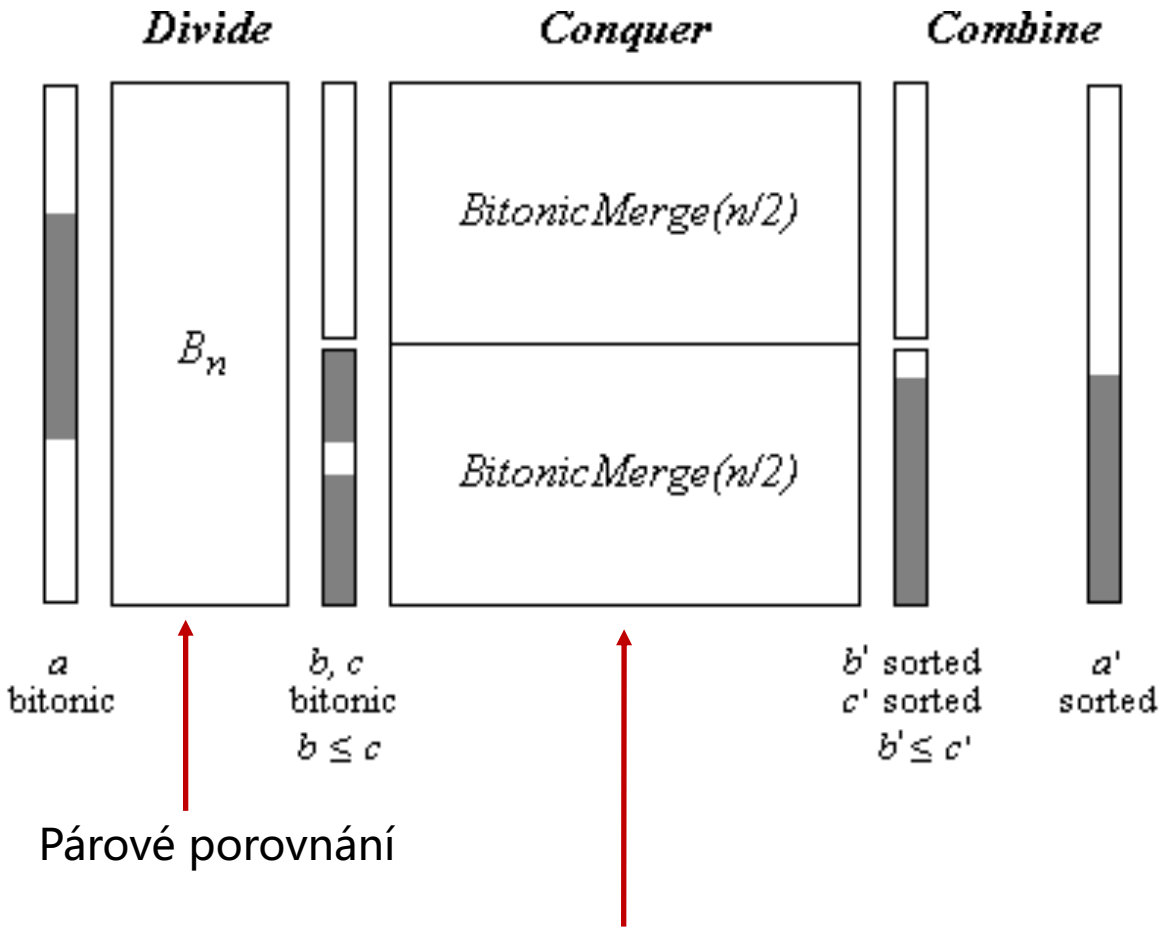
Bitonic Sort

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



Paralelní řazení

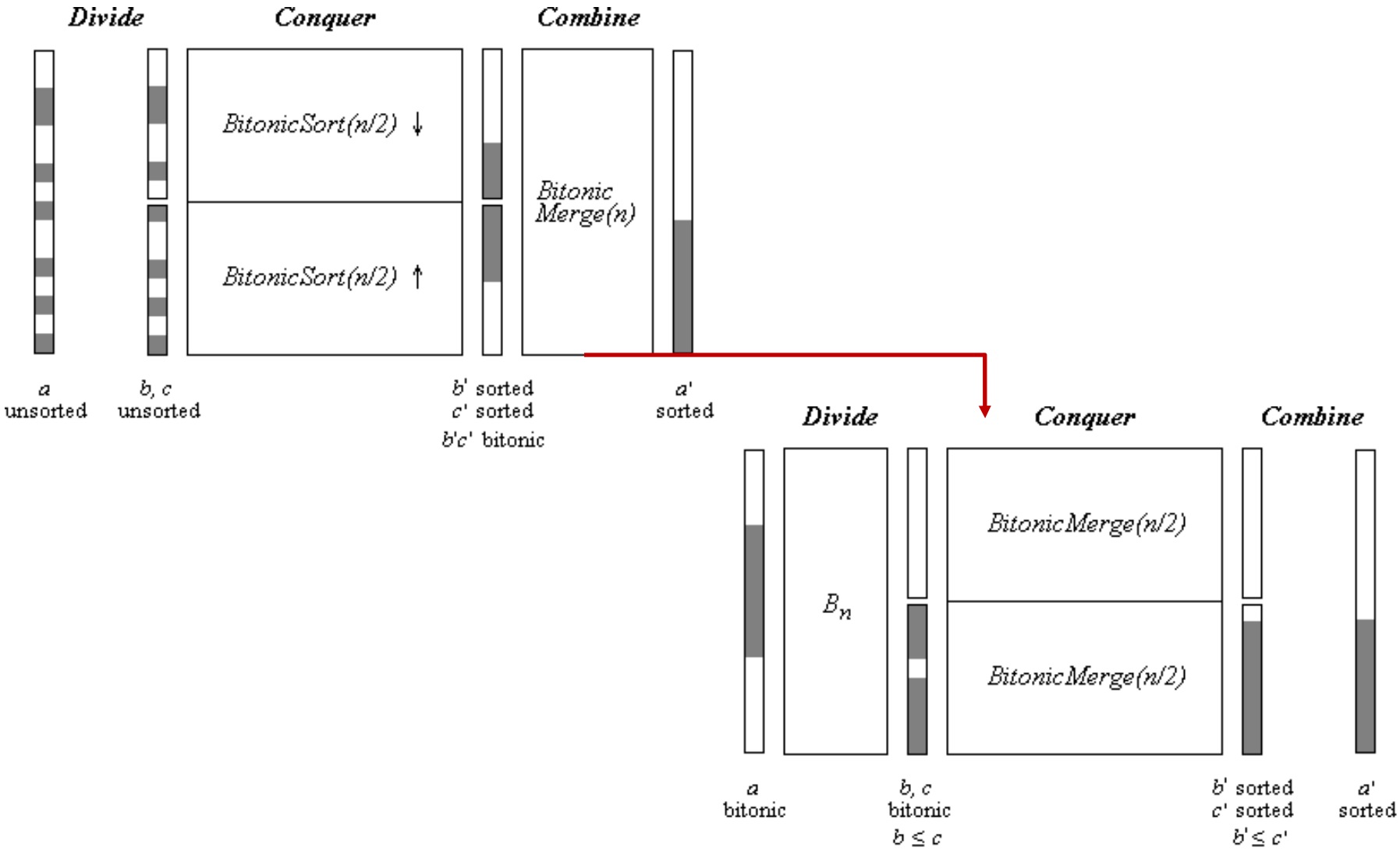
Bitonic Sort



A co když není vstupní sekvence bitonická?

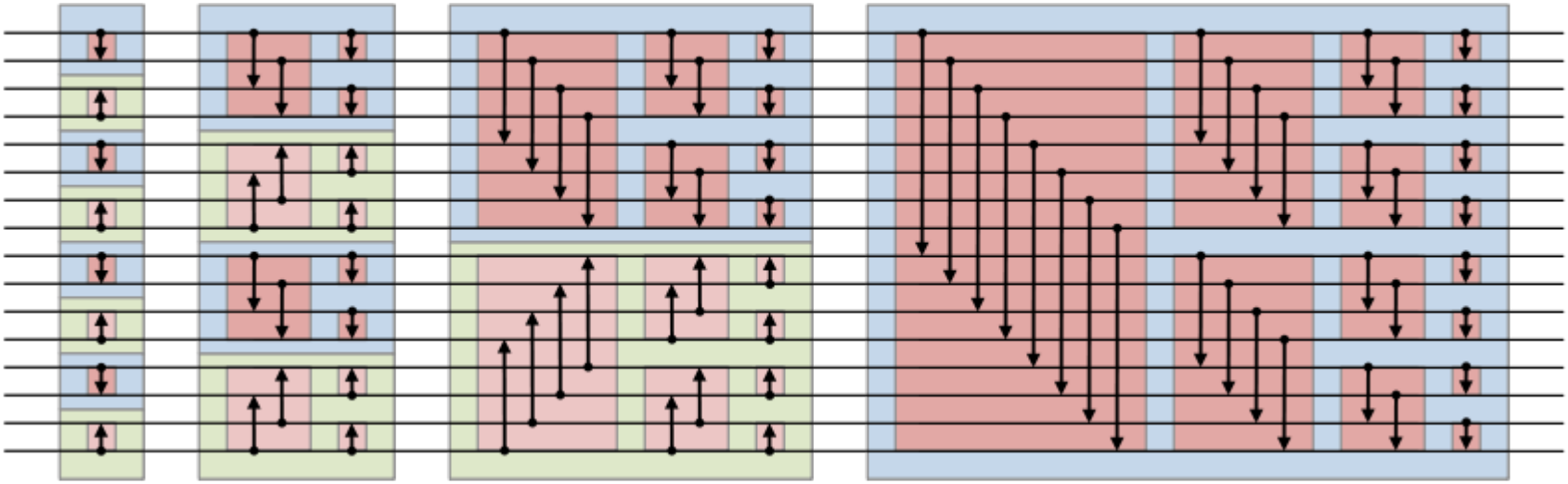
Paralelní řazení

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
  - Můžeme využít GPU nebo vektorizaci na CPUs

- Vektorizace pomocí instrukcí a intrinsics Intel: https://software.intel.com/sites/landingpage/IntrinsicsGuide/

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
AVX	__m256	256 bitový vektor, obsahuje 8x **float**
	__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

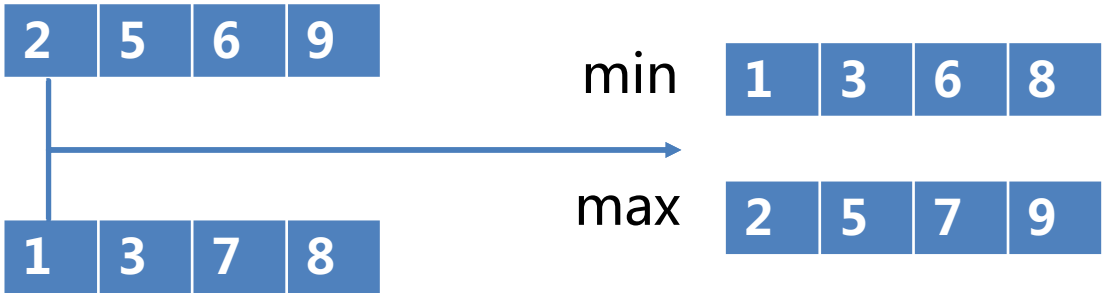
- Počet celých čísel záleží na typu
- Ve __m256i může být:
  - 32 char
  - 16 short
  - 8 int
  - 4 long

Datové typy	
__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**
__m256d	256 bitový vektor, obsahuje 4x **double**
__m256i	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

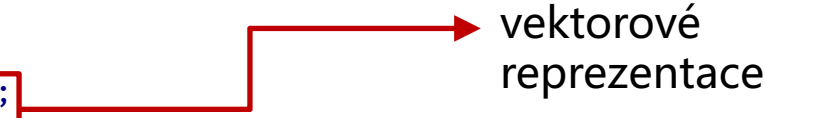
Párové porovnání 2 vektorů

```
#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;
}
```



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)
  - x0 ? x1 (a případně vyměnit tak, aby x0 byla menší)
  - x2 ? x3 (a případně vyměnit tak, aby x2 byla menší)

x3	x2	x1	x0
2	5	6	9

- Jak na to?

Krátký úvod do vektorových instrukcí

- Párové porovnání (a případná výměna) prvků v poli (např. sousedních)
  - x0 ? x1 (a případně vyměnit tak, aby x0 byla menší)
  - x2 ? x3 (a případně vyměnit tak, aby x2 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

Porovnáme s původním vektorem

x3	x2	x1	x0
2	5	6	9

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(x0,x1) je na pozici 0
- min(x2,x3) 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(x0,x1) je na pozici 0
- min(x2,x3) je na pozici 2
- …

Vynulujeme pomocí masky

3	2	1	0
0	2	0	6

Výsledek je OR těchto vektorů

Podobně získáme maxima z porovnání a uložíme je na liché pozice

3	2	1	0
5	0	9	0

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_llhhllhh = _mm_set_epi32(0xffffffff,0,0xffffffff,0);
  __m128i  mask_hhllhhll = _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_hhllhhll);
     v2 = _mm_alignr_epi8(v1, _mm_setzero_si128(),3*4);
     r2 = _mm_max_epi32(v1, v2);
     r2 = _mm_and_si128(r2,mask_llhhllhh);
     r1 = _mm_or_si128(r1,r2);
     _mm_storeu_si128((__m128i *) &vec1[i], r1);
  }
```

Pointa vtipu

https://arxiv.org/pdf/2009.13569.pdf

Engineering In-place (Shared-memory) Sorting Algorithms

MICHAEL AXTMANN, Karlsruhe Institute of Technology
SASCHA WITT, Karlsruhe Institute of Technology
DANIEL FERIZOVIC, Karlsruhe Institute of Technology
PETER SANDERS, Karlsruhe Institute of Technology

Type	Distribution	IPS^{4}o	PBBS	PS^{4}o	MCSTLmwm	MCSTLbq	TBB	RegionSort	PBBR	RADULS2	ASPaS
double	Sorted	1.42	10.96	2.02	15.47	13.36	**1.06**				42.23
double	ReverseSorted	**1.06**	1.34	1.98	1.76	11.00	3.01				5.34
double	Zero	1.54	12.83	1.80	14.55	166.67	**1.06**				41.78
double	Exponential	**1.00**	1.82	1.97	2.60	3.20	10.77				4.97
double	Zipf	**1.00**	1.96	2.12	2.79	3.55	11.56				5.33
double	RootDup	**1.00**	1.54	2.22	2.52	3.88	5.54				6.28
double	TwoDup	**1.00**	1.93	1.88	2.45	2.99	5.52				4.44
double	EightDup	**1.00**	1.82	2.01	2.48	3.19	10.37				5.02
double	AlmostSorted	**1.00**	1.73	2.40	5.12	2.18	3.54				6.37
double	Uniform	**1.00**	2.00	1.85	2.53	2.99	9.16				4.39
Total		**1.00**	1.82	2.06	2.83	3.10	7.46				5.21
Rank		1	2	3	4	5	7				6

Table 4. Average slowdowns of parallel algorithms for different data types and input distributions. The slowdowns average over the machines and input sizes with at least $2^{21}t$ bytes.

Závěr

- Cílem bylo vyzkoušet si techniky paralelizace
- Klíčový je návrh algoritmu, “thinking out of the box”
- Plná implementace (i jen) paraleního třídícího algoritmu pomocí vektorových instrukcí může být dost pracná: https://xhad1234.github.io/Parallel-Sort-Merge-Join-in-Peloton/