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

Use Cases

Via Theoretical Computer Science

A large part of theoretical computer science is devoted to the study of the limitations of parallelizability of algorithms, starting with the very clear question:

does every problem with a polynomial-time sequential algorithm also have an efficient parallel algorithm?

In particular, an algorithm solving the problem in time $O(\log^c n)$ using $O(n^k)$ parallel processors for some constants c and k.

In short, the answer is no.

Use Cases

Via Theoretical Computer Science

Let us consider the **P-Complete** problems i.e., decision problems that are in P and where every problem in P can be reduced to the P-Complete problem in logarithmic space (L). The prime examples of P-Complete problems are

- circuit evaluation: given a Boolean circuit and an input, is the output of the circuit *0*?
- linear programming: given a linear function subject to linear inequality constraints, is the minimum greater or equal to *0*?
- many graph problems, such as Lexicographically First Depth-first Search Ordering (LFDFS): given an undirected graph with fixed ordered adjacency lists, and two vertices *u* and *v*, is vertex *u* visited before vertex *v* in the depth-first search of the graph?
- many compression algorithms: given strings *s* and *t*, will compressing *s* with LZ78 add *t* to the dictionary?
- many problems related to Markov decision processes: is the minimum expected cost over all policies equal to 0 (for both finite, and long-run versions)?
- many tests of local optimality in combinatorial optimization: in an instance of Traveling Salesman, and a sequence of tours, is this a 2-Opt sequence?

Use Cases

Via Theoretical Computer Science

In contrast, Nick's class NC^c are decision problems solvable by a uniform family of Boolean circuits with polynomial size, depth $O(\log^c(n))$, and fan-in 2.

More usefully, a problem in NC with input of length n can be solved in time $O(\log^c n)$ using $O(n^k)$ parallel processors for some constants c and k. (Notice that for a constant k, $O(n^k)$ is a polynomial.

Thus, NC can be thought of as problems that can be efficiently solved on a parallel computer, and hence ``easier'' than P-Complete problems. For example:

- integer arithmetics (addition, multiplication and division),
- matrix arithmetics (multiplication, determinant, inverse, rank), or
- some graph problems (shortest path, maximal matching with some restrictions on the weights)

Use Cases

Via Theoretical Computer Science

Let us consider the problem of sorting an array of *n* elements.
Is it P-Complete?

Use Cases

Via Theoretical Computer Science

Let us consider the problem of sorting an array of *n* elements.
Is it P-Complete?

Algorithm	$p(n)$ Processors	Time
Sequential algorithms	1	$O(n \log n)$
Parallel divide and conquer	$O(1)$	$O(n \log n)$
	$O(\log n)$	$O\left(\frac{n(\log n)}{p(n)}\right)$
	$\omega(\log n)$	$O(n)$
Parallel Ranking	$O(n^2)$	$O(\log n)$

Use Cases

Via Theoretical Computer Science

Let us consider the problem of sorting an array of *n* elements.
Is it P-Complete?

Algorithm	$p(n)$ Processors	Time
Sequential algorithms	1	$O(n \log n)$
Parallel divide and conquer	$O(1)$	$O(n \log n)$
	$O(\log n)$	$O\left(\frac{n(\log n)}{p(n)}\right)$
	$\omega(\log n)$	$O(n)$
Parallel Ranking	$O(n^2)$	$O(\log n)$

As the table suggests, sorting is Nick's class: for n items, we can consider all pairs of items in parallel, compare them to obtain a binary value, and then for each item, obtain its rank in the sorted order by adding the binary values. This is known as the parallel ranking. There are many other parallel algorithms based on picking minimum and maximum from a small set, as well as algorithms based on hashing.

Use Cases

What to do in a particular use case?

In the second lecture, we have seen that within shared-memory parallel programming, we have broadly four options:

- **Confinement**: Do not share memory between threads.
- **Immutability**: Do not share any mutable data between threads.
- **Thread-safe code**: Use data types with additional guarantees for storing any mutable data shared between threads, or even better, use implementations of algorithms that are already parallelized and handle the concurrency issues for you.
- **Synchronization**: Use synchronization primitives to prevent accessing the variable at the same time.

Thread-safe Code in C++20

Use STL!

We have seen that the header execution defines objects std::execution::par and std::execution::par_unseq, which can be passed as the first argument of any standard algorithm:

```
#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;
}
```

Thread-safe Code in C++20

Use STL!

What does the code do?

```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;
}
```

Thread-safe Code in C++20

Use STL!

What does the code do?

One may imagine:

- templates
- iterators
- execution strategies.

Elegant code.

```
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);
}
```

Thread-safe Code in C++20

Use STL!

Reality can be much more messy:

```
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_ty
        ↪  _ValueType;
```

Thread-safe Code in C++20

Use STL!

Reality can be much more messy:

```
            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);
        });
    }
```

Thread-safe Code in C++20

Use STL!

In the previous slides, we have seen the implementation in Intel Thread Building Blocks (TBB) backend of the GCC. This uses:

- many megabytes of a library (TBB)
- ``sorting based on task tree and parallel merge”,

  while making use of several non-trivial tricks, including
- tbb::task_scheduler_init,
- std::thread::hardware_concurrency(),
- std::hardware_constructive_interference_size.

(Contrast this with the serial version of GCC **sort**, which uses a multi-way mergesort, and GCC **stable_sort**, which uses a quicksort.) We wish to make use of the STL, rather than redevelop it, in the first instance.

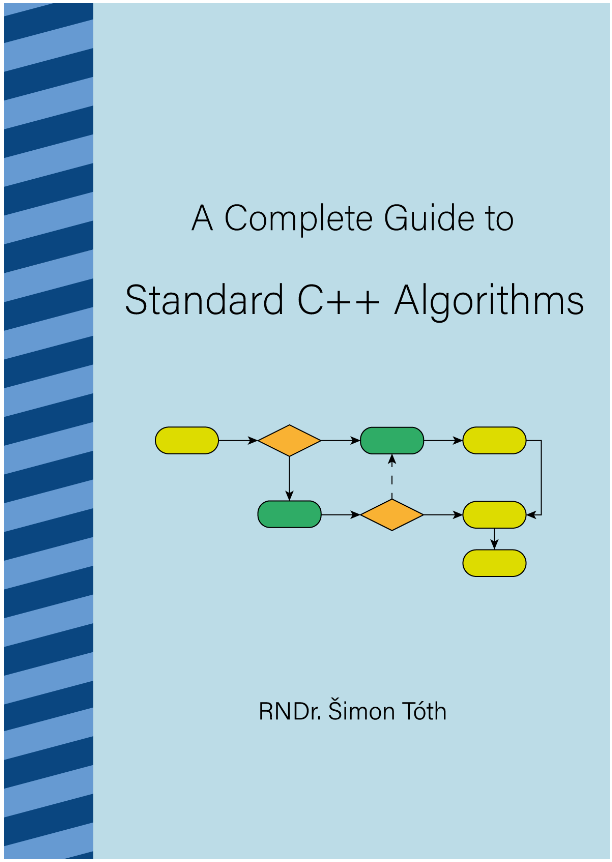
Thread-safe Code in C++20

Use STL!

Even making full use of the STL is quite non-trivial.

In the lecture notes, we present an overview of the sorting-related routines in verbatim from the fantastic book ``A Complete Guide to Standard C++ Algorithms'' of Simon Toth, in compliance with the license.

https://github.com/HappyCerberus/book-cpp-algorithms



Thread-safe Code in C++20

Use STL!

What is wrong with comparators:

```cpp
struct Point {

int x;
int y;

// pre-C++20 lexicographical less-than
friend bool operator<(const Point& left, const Point& right) {
    if (left.x != right.x)
        return left.x < right.x;
    return left.y < right.y;
}

// default C++20 spaceship version of lexicographical comparison
friend auto operator<=>(const Point&, const Point&) = default;

// manual version of lexicographical comparison using operator <=>
friend auto operator<=>(const Point& left, const Point& right) {
    if (left.x != right.x)
        return left.x <=> right.x;
    return left.y <=> right.y;
}

};
```

Thread-safe Code in C++20

Use STL!

What is wrong with projections:

```
struct Account {
    double value() { return value_; }
    double value_;
};

std::vector<Account> accounts{{0.1}, {0.3}, {0.01}, {0.05}};
std::ranges::sort(accounts, std::greater<>{}, &Account::value);
// accounts = { {0.3}, {0.1}, {0.05}, {0.01} }
```

Thread-safe Code in C++20

If you know your problem and STL well, you may benefit from reformulating the problem, e.g., to partial sort:

Example of using `std::partial_sort` to sort the first three elements of a range.

```
std::vector<int> data{9, 8, 7, 6, 5, 4, 3, 2, 1};
std::partial_sort(data.begin(), data.begin()+3, data.end());
// data == {1, 2, 3, -unspecified order-}

std::ranges::partial_sort(data, data.begin()+3, std::greater<>());
// data == {9, 8, 7, -unspecified order-}
```

Synchronization

Now, let us move to our own parallel sorting algorithms.

We will see:

- Bubble sort with OpenMP
- Quick sort variants with task construct in OpenMP
- Merge sort variants with task construct in OpenMP
- Many variants, including intrinsics.

Synchronization

Bubble Sort

Bubble sort is essentially using loops:

```
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);
        }
    }
}
```

Synchronization

Bubble Sort

Bubble sort is essentially using loops, which are easy to parallize:

```
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);
        }
    }
}
```

Synchronization

Quick Sort

Quick sort may benefit from, OpenMP construct **task**

```
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;
    }

    // cf. the pivot (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);
    }
}
```

Synchronization

Quick Sort

One can improve upon this:

- using three-way sort
- using task mergeable

(As suggested by Intel.)

```
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++;
        }
```

Synchronization

Quick Sort

One can improve upon this:

- using three-way sort
- using task mergeable

```
        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))
```

Synchronization

Quick Sort

One can improve upon this:

- using three-way sort
- using task mergeable

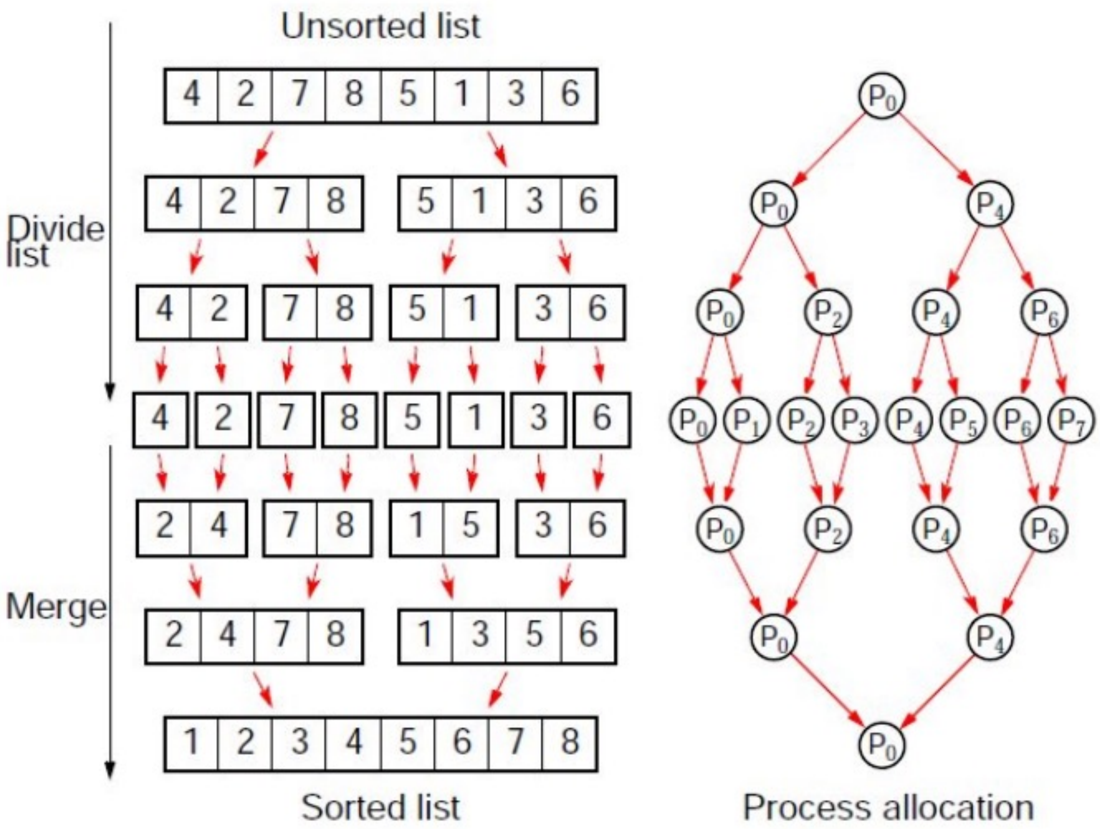
```
                    qsort3w(_RightIt + 1, _Last, compare);
            }
        } else {
#pragma omp task untied mergeable
            {
                if ((std::distance(_First, _LeftIt) > 0) && is_swapped_left)
                    qsort3w(_First, _LeftIt - 1, compare);

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

Synchronization

Merge Sort

Similarly, one can use task to parallelise merge sort:



Synchronization

Merge Sort

Similarly, one can use task to parallelise merge sort:

```cpp
void ms_parallel(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;

```

Synchronization

Merge Sort

Similarly, one can use task to parallelise merge sort:

```
#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);
}
```

Synchronization

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

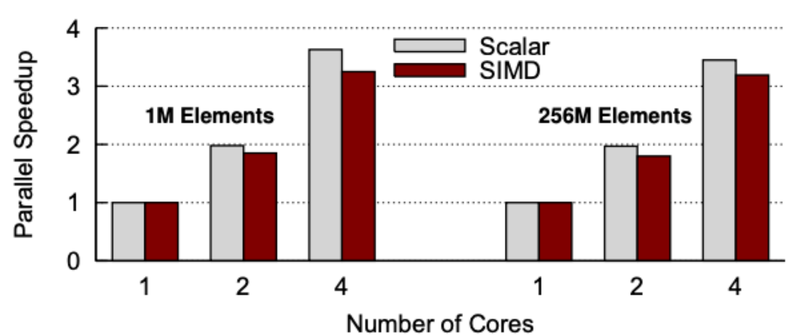
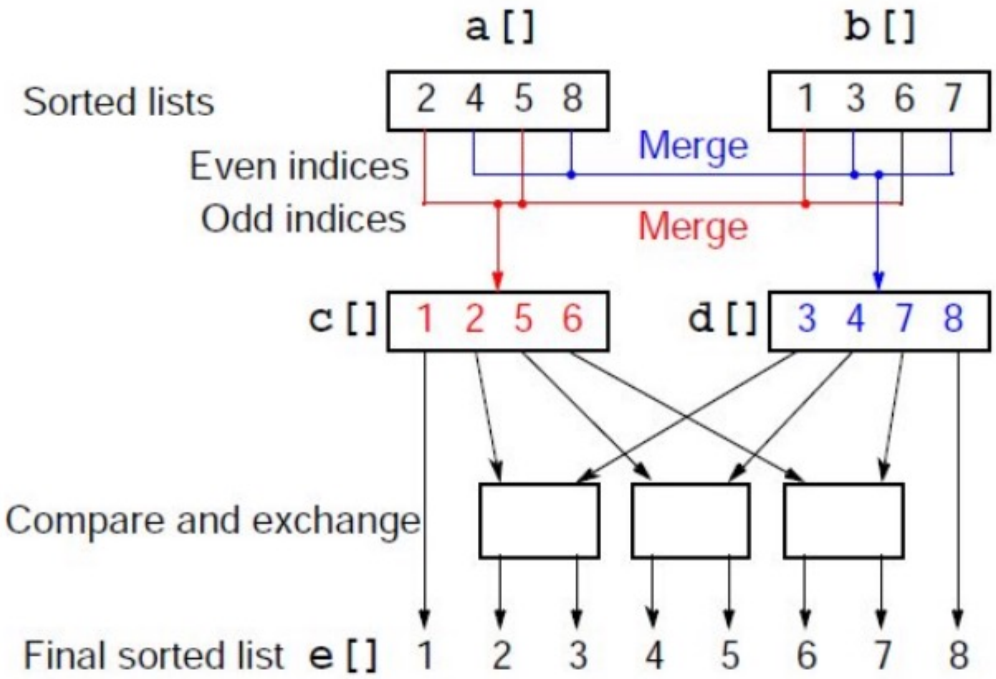


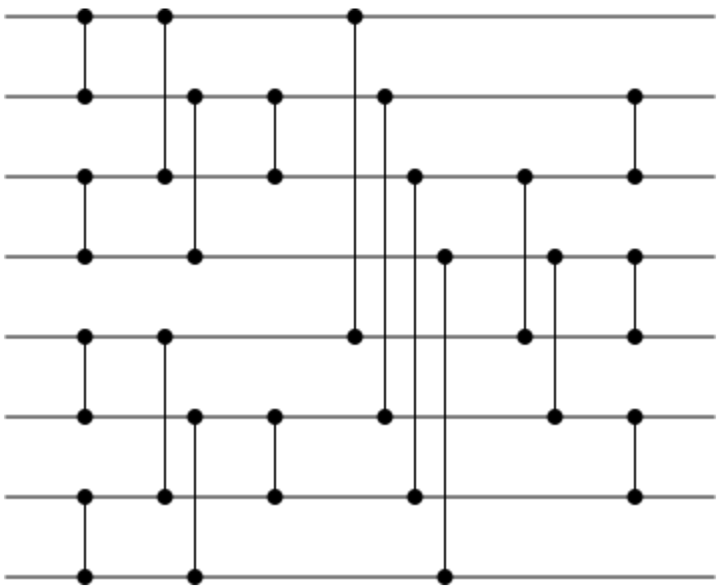
Figure 8: Parallel performance of the scalar and SIMD implementations.

Synchronization

Going beyond Merge Sort

Odd-Even Merge Sort:

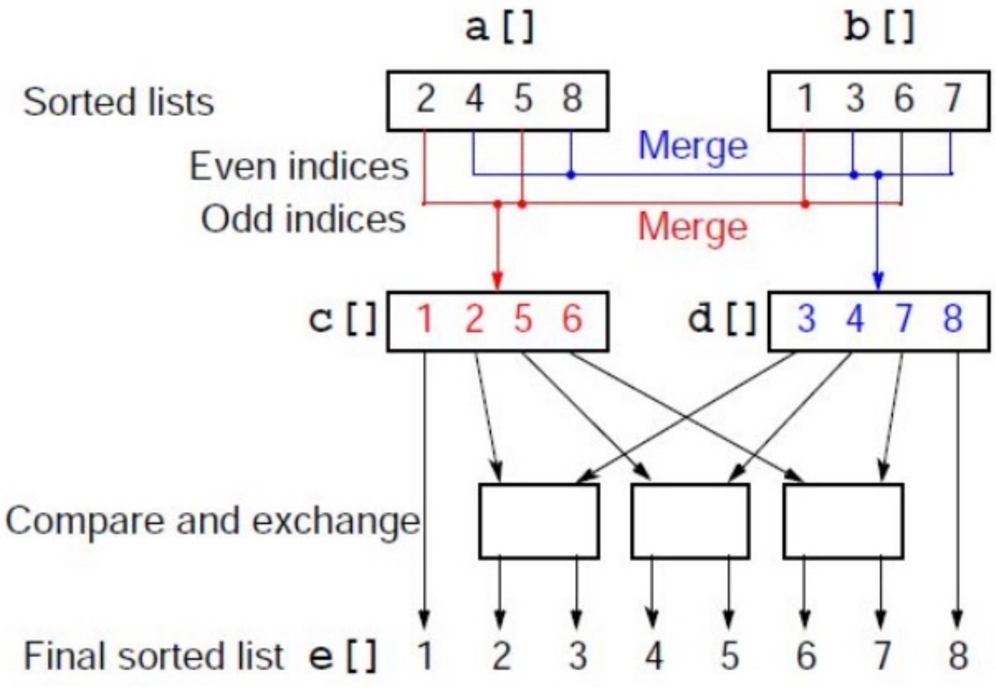


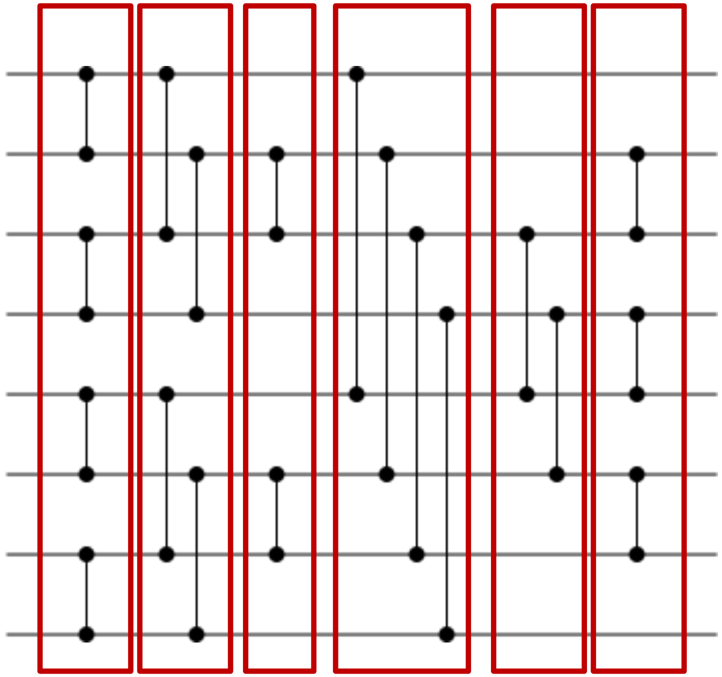


Synchronization

Going beyond Merge Sort

Odd-Even Merge Sort:





Synchronization

Going beyond Merge Sort

Odd-Even Merge Sort:

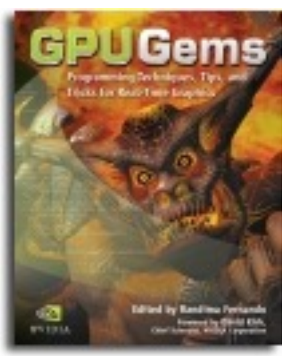
```
void odd-even-merge (std::vector<int>& vector_to_sort, int from, int to, int step) {
    auto new_step = step * 2;
    if (new_step < to - from) {
        odd-even-merge(vector_to_sort,from,to,new_step);
        odd-even-merge(vector_to_sort,from+step,to,new_step);
        for (int i=from+step; i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

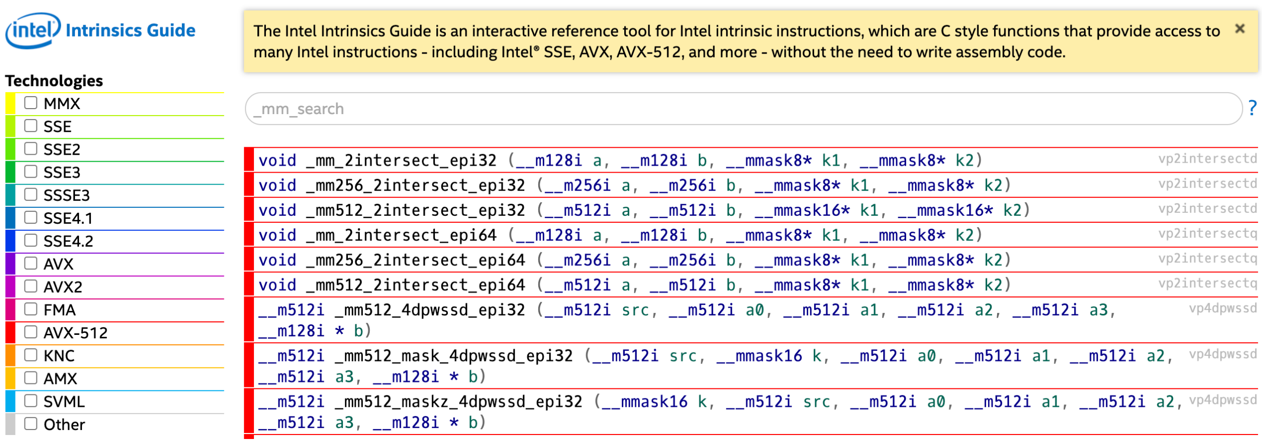
One can go even further with bitonic sort.

Synchronization

Bitonic Sort

- Efficient implementations can vectorize the compare and swap
- This is the most efficient approach on GPGPUs
- One can also experiment with intrinsics: https://software.intel.com/sites/landingpage/IntrinsicsGuide/





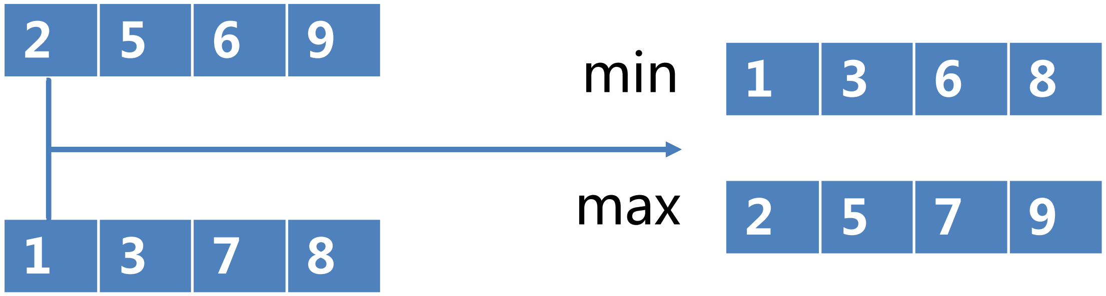
An Aside

Intrinsics

- Modern processors imlement vector instructions (SIMD)
- On Intel and AMD, there are Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX) incl. 512-bit vectors.
- In SSE, __m128d stores 2 doubles, 4 ints, 16 chars, but in the reverse order float[4] {0f,1f,2f,3f}

- You can compile for this with –march=native-mavx

- You can run pairwise sorting using minima and maxima:

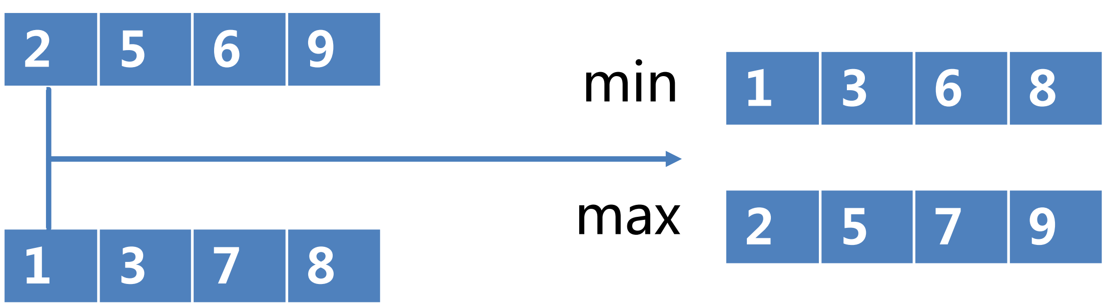


An Aside

Intrinsics

```
    __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);
    }
```



An Aside

Intrinsics

- We can also pad, shift, truncate. (Illustrations by Brano Bosansky.)

Pad:

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

Shift to the right:

					x3	x2	x1
0	0	0	0	0	2	5	6

Truncate:

	x3	x2	x1
0	2	5	6

Compare:

x3	x2	x1	x0
2	5	6	9

An Aside

Intrinsics

- We can also pad, shift, truncate. (Illustrations by Brano Bosansky.)

```
__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);
}
```

An Aside

Intrinsics

- We can also pad, shift, truncate.
  (Illustrations by Brano Bosansky.)

- See https://xhad1234.github.io/Parallel-Sort-Merge-Join-in-Peloton/
  for a comprehensive illustration.

The Upshot

Intrinsics need not win

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,
__m128i * b)                                                                         vp4dpwssd
__m512i _mm512_mask_4dpwssd_epi32 (__m512i src, __mmask16 k, __m512i a0, __m512i a1, __m512i a2,   vp4dpwssd
```

The Upshot

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.

Conclusions

- First use case in parallelization
- Highlights importance of “thinking out of the box”