CTU

CZECH TECHNICAL UNIVERSITY IN PRAGUE

Parallel and Distributed Computing (B4B36PDV)

Matěj Kafka, Michal Jakob

kafkamat@fel.cvut.cz

https://pdv.pages.fel.cvut.cz

A case study

PARALLEL SORTING ALGORITHMS

Parallel sorting algorithms

- Common and well-understood problem…
  - …with many interesting solutions.
- You should already know common algorithms from ALG.
- Comparison-based vs non-comparison-based sorting
  - We will focus on comparison-based algorithms.

- Good case study for designing parallel algorithms.
  - How to directly parallelize simple sorting algorithms?
  - How do real-world implementations sort?

Serial quicksort

We already saw this one last week.

```
template<typename It>
void quicksort(It begin, It end) {
    if (end - begin <= 1) {
        return;
    }

    // select a pivot and partition the data around it
    auto pivot_it = partition(begin, end);

    quicksort(begin, pivot_it);
    quicksort(pivot_it + 1, end);
}
```

Parallel quicksort

We already saw this one last week.

```
template<typename It>
void quicksort(It begin, It end, size_t task_count) {
    if (end - begin <= 1) return;

    if (task_count == 1) {
        quicksort_seq(begin, end);
        return;
    }

    // select a pivot and partition the data around it
    auto pivot_it = partition(begin, end);

    #pragma omp task
    quicksort(begin, pivot_it, task_count / 2);

    // run this branch in the current task
    quicksort(pivot_it + 1, end, task_count / 2);
}
```

Cutoff -> switch to serial version.

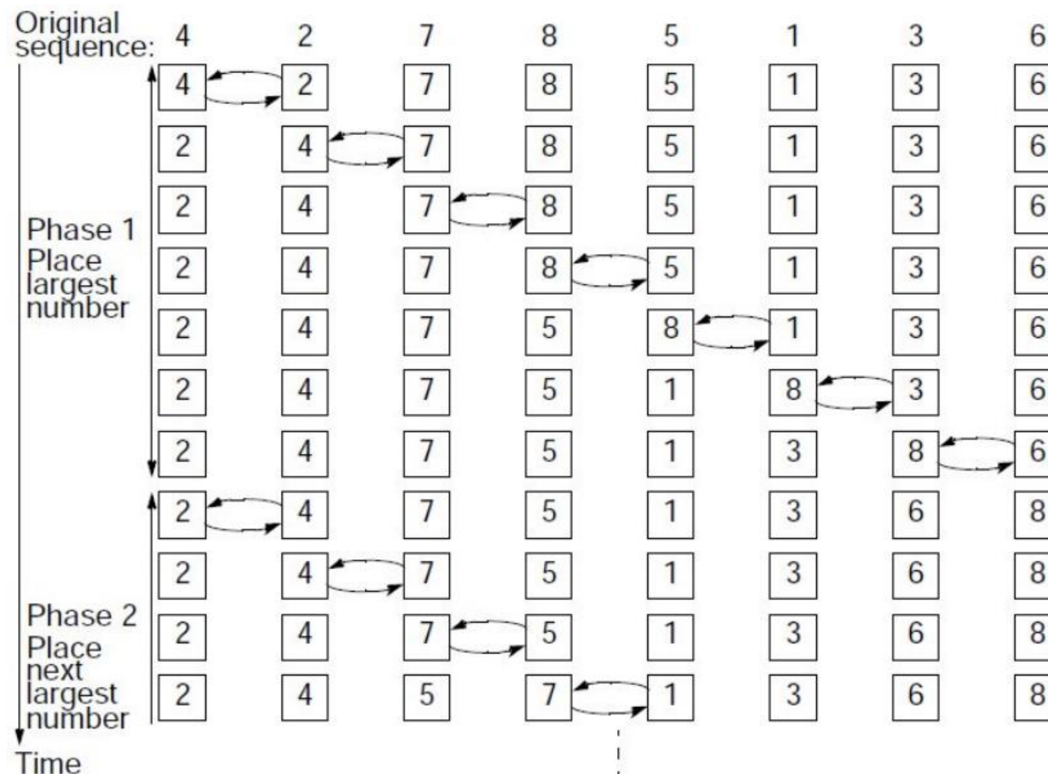
Create a task for one branch.

Avoid task overhead.

Bubble sort

The simplest usable sorting algorithm.

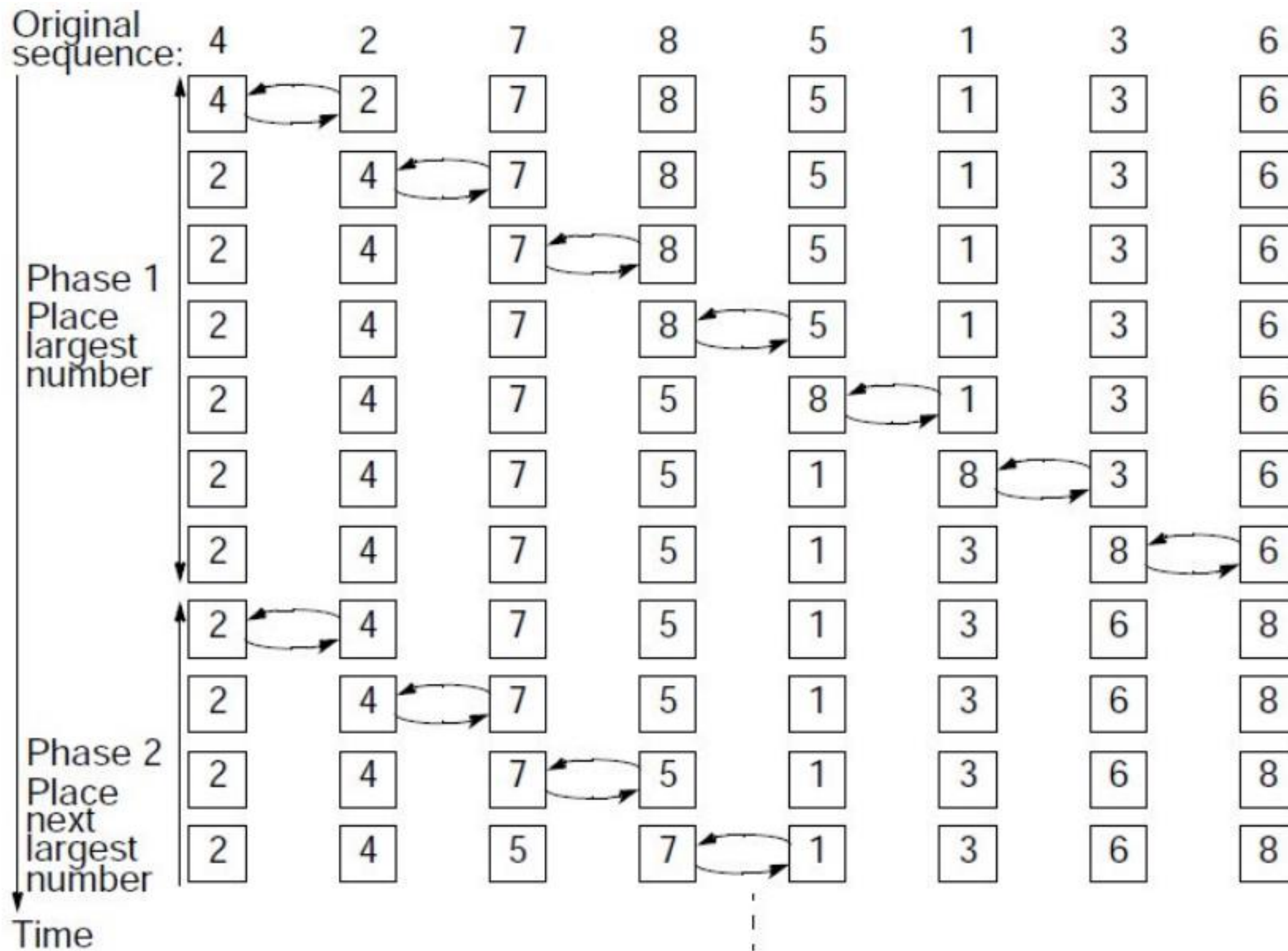
```
template<typename It>
void bubble_sort(It begin, It end) {
    for (end--; end != begin; end--) {
        for (auto it = begin; it != end; ++it) {
            minmax(*it, *(it + 1));
        }
    }
}
```



Parallel bubble sort

How to parallelize it?

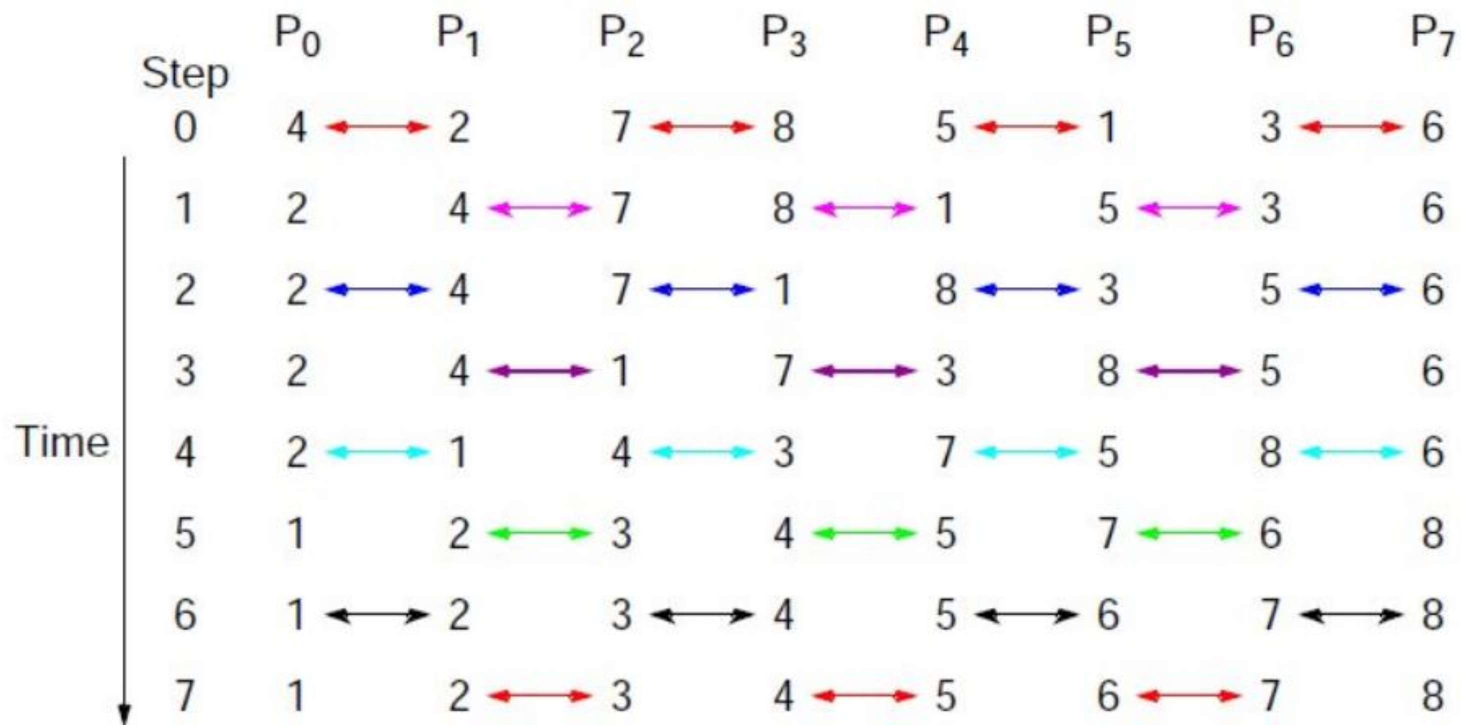
- Which steps can be done in parallel?



Odd-even sort

Bubble sort with a different comparison order.

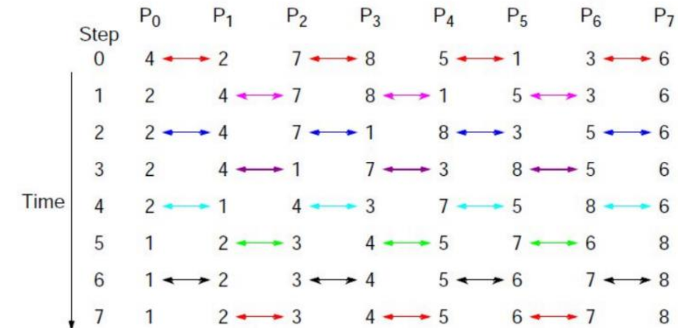
- Which steps can be done in parallel?
  - Remember heat diffusion from last lecture.



Odd-even sort

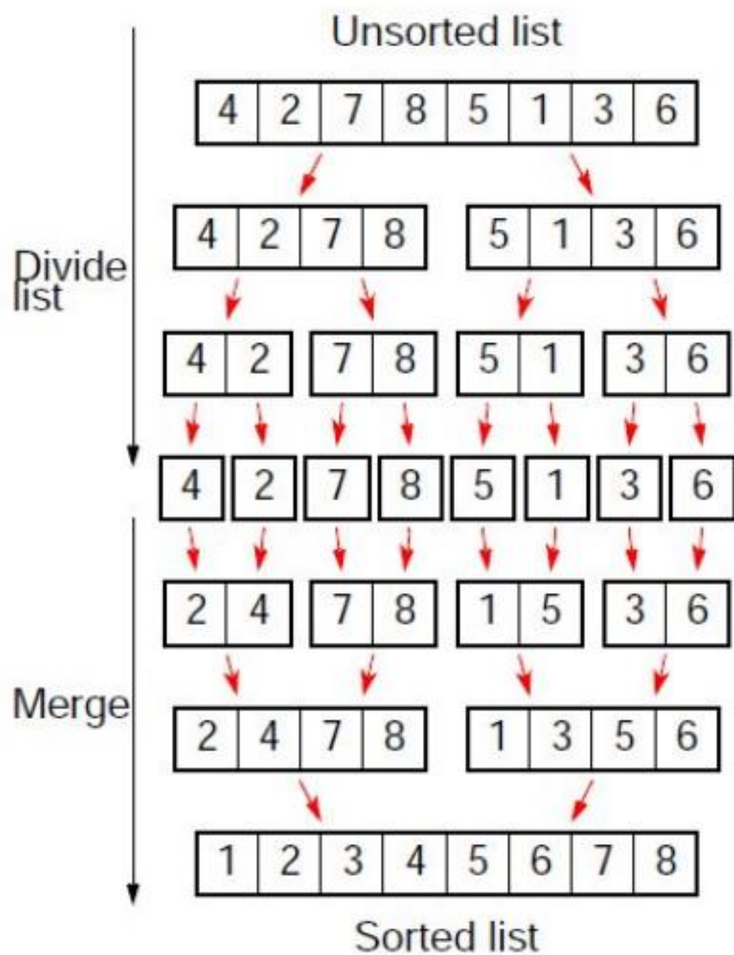
Bubble sort with a different comparison order.

```
template<typename It>
void odd_even_sort(It begin, It end) {
    end--;
    auto n = end - begin;

    #pragma omp parallel
    for (size_t i = 0; i < n; i++) {
        #pragma omp for
        for (auto it = begin; it < end; it += 2) {
            minmax(*it, *(it + 1));
        }

        #pragma omp for
        for (auto it = begin + 1; it < end; it += 2) {
            minmax(*it, *(it + 1));
        }
    }
}
```



Merge sort

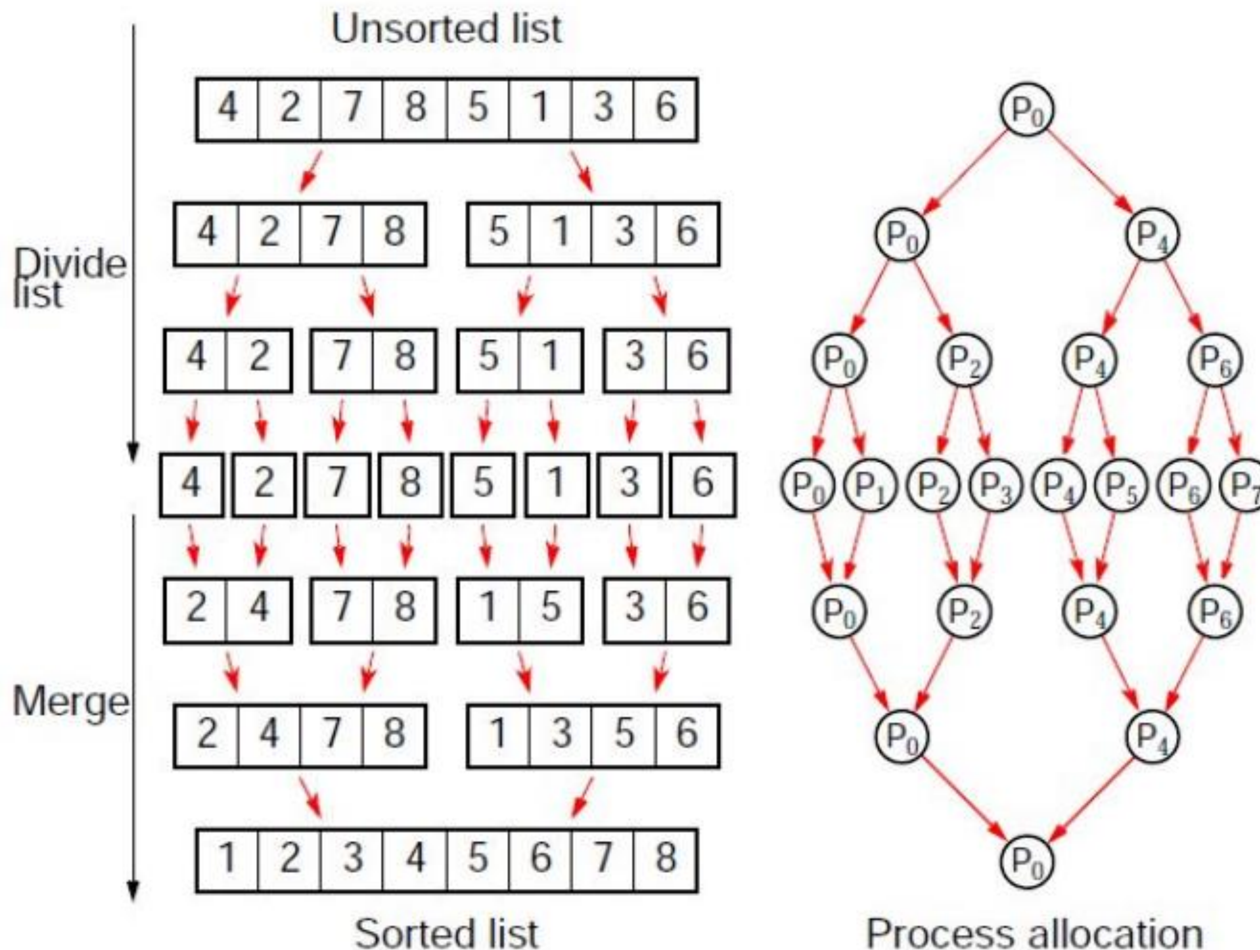
Another divide-and-conquer algorithm.



Parallel merge sort

Recursion? Tasks! (most likely)

- Could we also parallelize the merge?

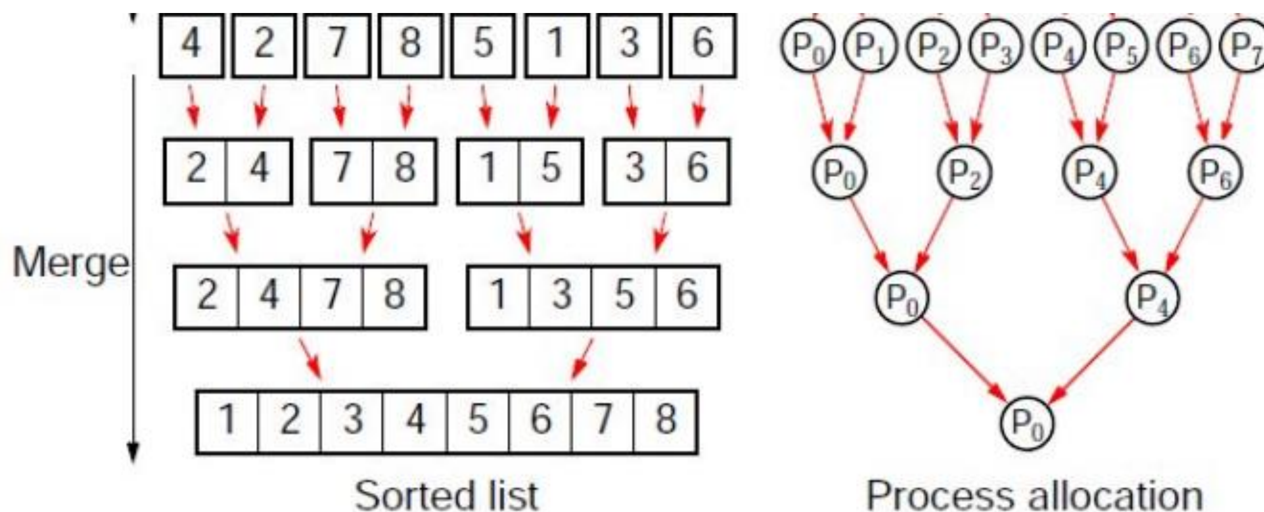


Parallel merge sort

Bottom-up formulation

We can also implement merge sort as an iterative algorithm by starting from the lowest level (merging pairs) and going up in powers of two, skipping the "divide" part of the recursion.

Can be easily implemented with 2 for loops, we can parallelize the inner loop.

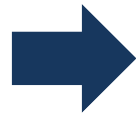


Sorting networks

Sorting small arrays quickly

- So far, we parallelized large arrays using CPU threads.
- How to efficiently sort small arrays in parallel?
  - CPU threads are too heavy, but we could implement the sorting algorithm in hardware (or an FPGA), or with lighter-weight "threads", e.g., CPU SIMD or GPUs (next lecture).
- We cannot easily branch control flow in hardware!
  - Could we avoid branching?

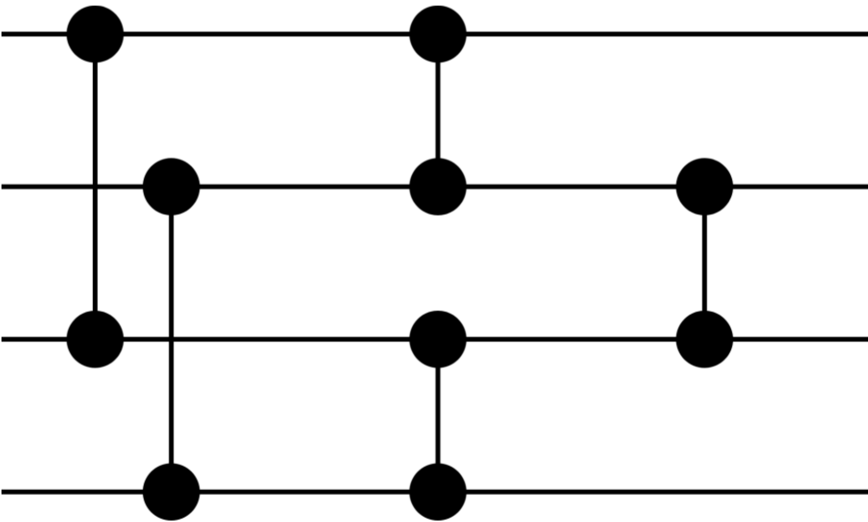
```
template<typename T>
void minmax(T& v1, T& v2) {
    if (v1 > v2) {
        std::swap(v1, v2);
    }
}
```



```
template<typename T>
void minmax(T& v1, T& v2) {
    auto min = std::min(v1, v2);
    auto max = std::max(v1, v2);
    v1 = min;
    v2 = max;
}
```

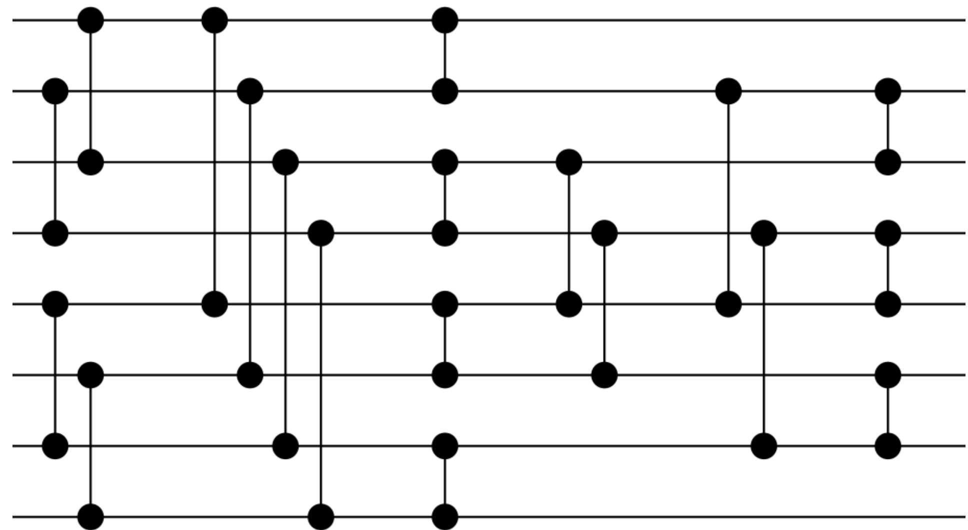
Sorting networks

Sorting small arrays quickly



Optimal sorting network for n=4

Optimal sorting network for n=8



Source: https://bertdobbelaere.github.io/sorting_networks.html

Sorting networks

Why are they useful?

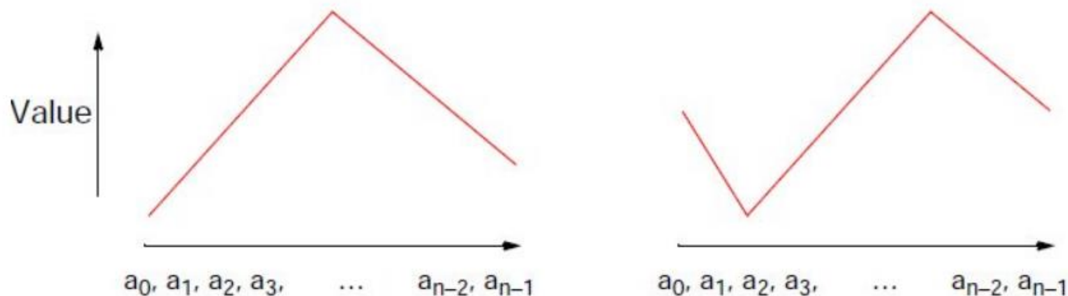
- We get a static tree of operations that's easy to parallelize on the small scale.
- Branchless code avoids penalties from branch misprediction on CPUs.
- Branchless code can be trivially implemented in hardware and in SIMD (next lecture).

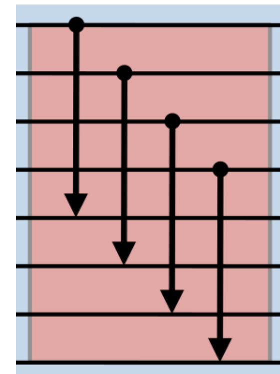
- We can construct sorting networks for small inputs by hand or using brute-force.
- What about larger inputs?

Bitonic merge

Branchless way to merge two sorted sequences.

- Let us go back to the idea of parallel merging.
- We have two ascending sequences of numbers.
  - Flip the second sequence (one ascending, one descending).
- What if we do parallel min/max between the two?
- The resulting sequences have 2 interesting properties:
  - The min sequence contains the lower half of all numbers.
  - Both sequences are "bitonic" = there is only a single "direction change", otherwise they're monotonic (or a cyclic shift of such a sequence).

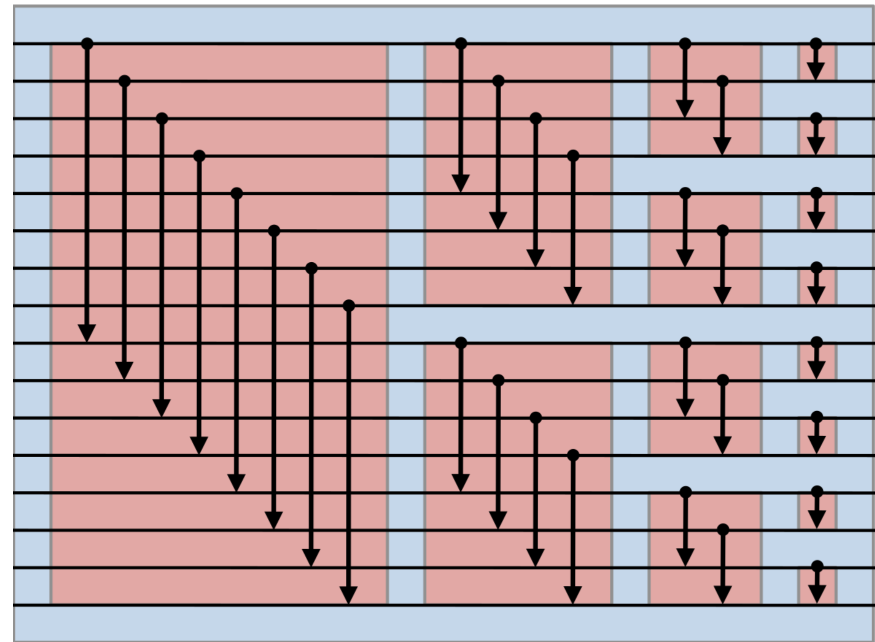




Bitonic merge

Branchless way to merge two sorted sequences.

- By recursive application of the min/max operation, we can convert a bitonic sequence to a sorted sequence.
- Proof is non-trivial, utilizing bitonicity and the zero-one principle.
- Intuitively, in each round, we move the values closer to the correct place, while the bitonic property ensures that a high value does not prevent a slightly lower value from moving to the upper half (and vice versa).

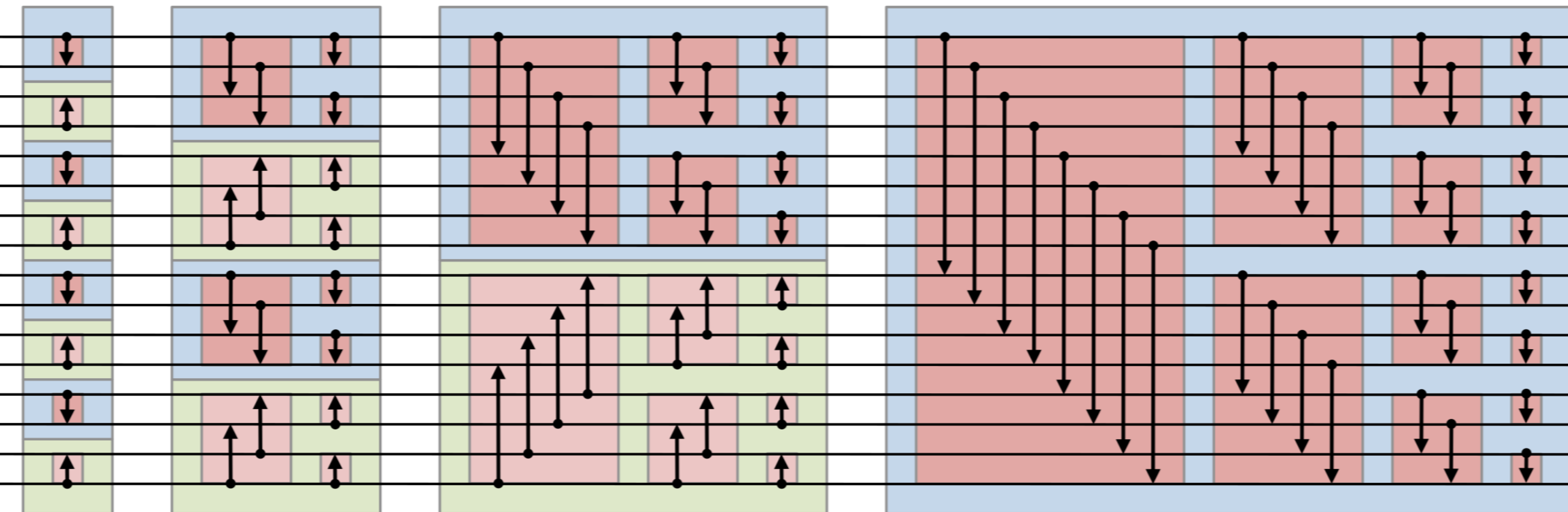


Source: https://commons.wikimedia.org/wiki/File:BitonicSort1.svg

Bitonic sort

An algorithm for constructing larger sorting networks.

- A singleton array is trivially sorted.
- Using bitonic merge, we can combine two sorted sequences of the same length into a single sorted sequence.
- -> We just inductively described how to create a sorting network for any $N = 2^x$.

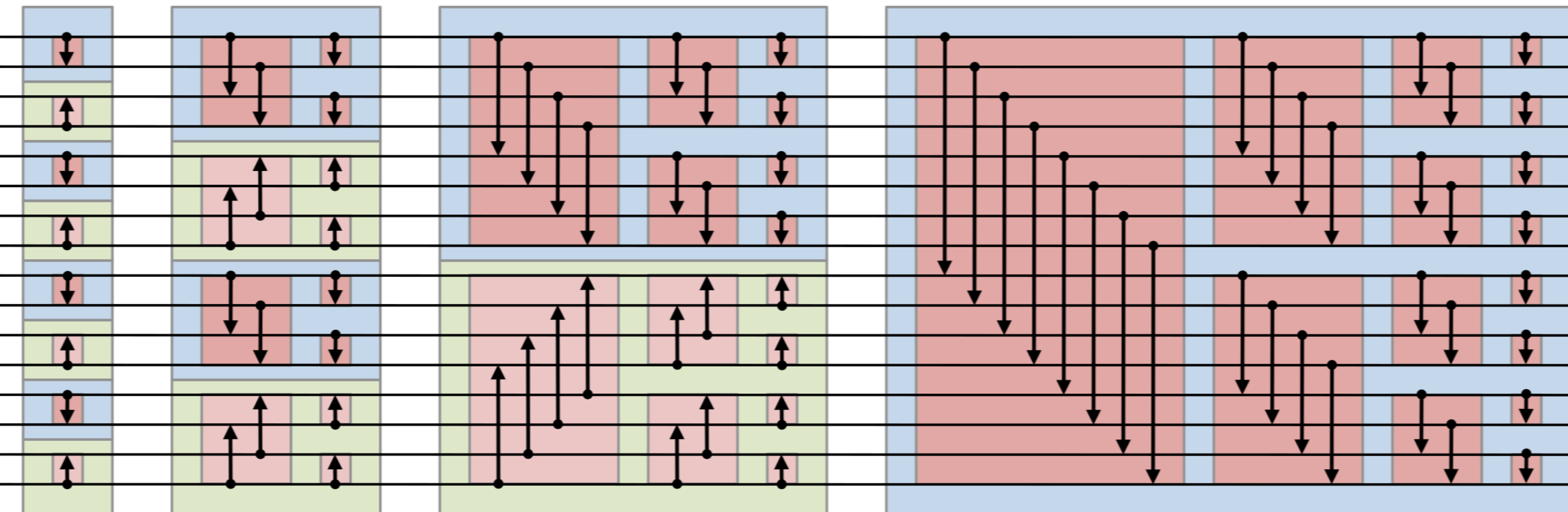


Source: https://commons.wikimedia.org/wiki/File:BitonicSort1.svg

Bitonic sort

Why is it interesting?

- Algorithm for creating larger sorting networks.
- Very amenable to parallelization, $O(\log^2 N)$ time complexity on sufficiently parallel hardware.
- Great fit for GPUs and SIMD.



Source: https://commons.wikimedia.org/wiki/File:BitonicSort1.svg

How does the C++ standard library sort?

SORTING IN THE REAL WORLD

Sorting in the real world

How does the C++ standard library sort?

- We already saw multiple parallel sorting algorithms.
- And we saw how OpenMP works internally.
- Let us apply both and see how widely-used libraries, such as the C++ standard library, sort things.

```cpp
auto vec = std::vector<uint32_t>{3, 4, 1, 5, 2};

// serial sort
std::sort(vec.begin(), vec.end());

// parallel sort
std::sort(std::execution::par_unseq, vec.begin(), vec.end());
```

C++ standard libraries

Not just a single C++ standard library.

- There are 3 mainstream implementations of the C++ standard library:
  - libstdc++ (GCC, most common on Linux)
  - libc++ (Clang, most common on macOS)
  - STL (MSVC, Windows-only)

- The parallel sort implementation in STL is the simplest and most readable (and somewhat slower than the other two), we'll explore it more in-depth.

- Warning: Stdlib code often looks ugly on first sight, you'll get used to it (if you look long enough).

Coding session 1

EXPLORING STD::SORT IN MICROSOFT'S STL

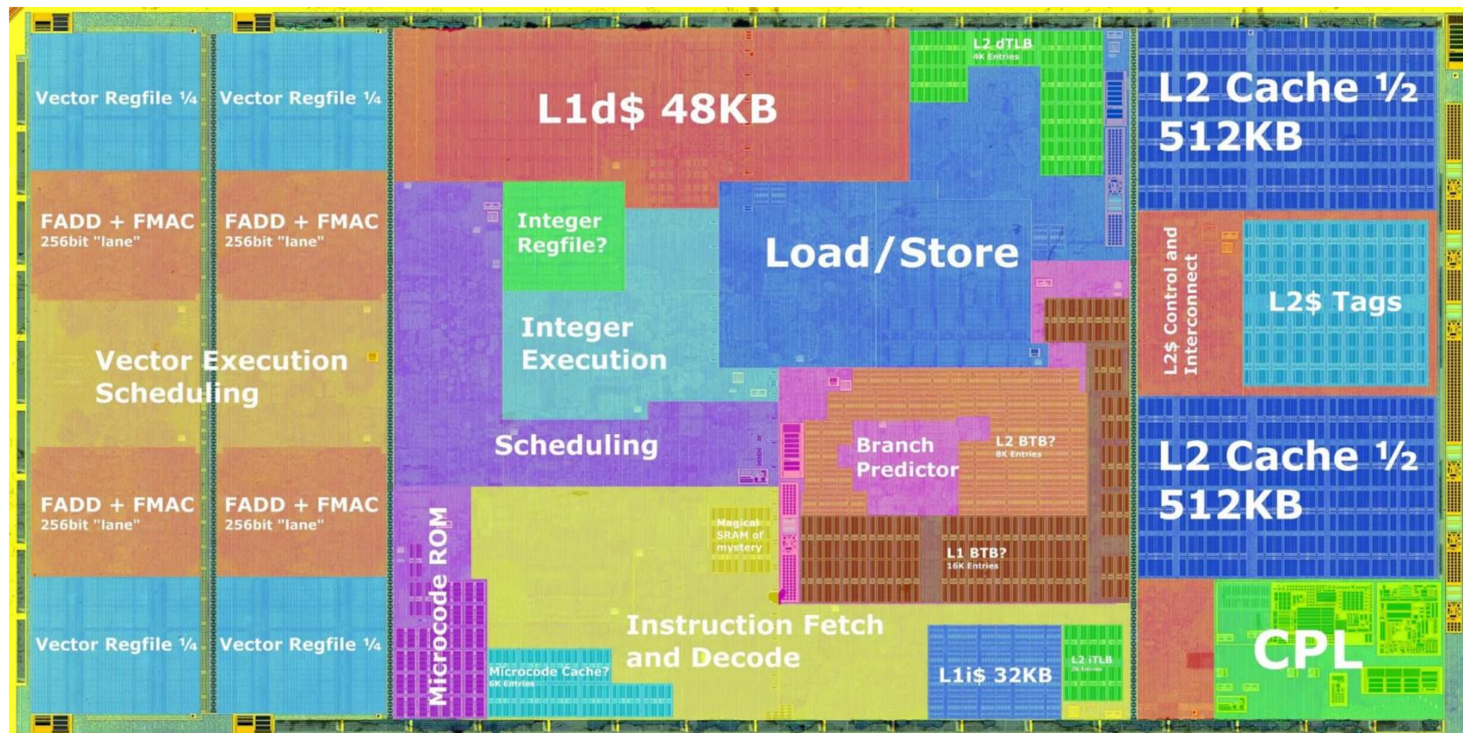
Small-scale parallelism

SIMD (VECTORIZATION)

SIMD

Lighter-weight parallelism

- The point of a CPU is to compute things.
- But most of the CPU is loading/storing data, scheduling instructions, predicting and handling branches,...



Source: https://nemez.net/die/

SIMD

Lighter-weight parallelism

- Could we somehow utilize more of the CPU for useful computation?
- Idea: Instructions that apply the same arithmetic operation to multiple values -> **vectorization**.
  - SIMD = single instruction, multiple data
  - We amortize the overhead of control flow, memory access, scheduling,… by only executing those steps once, but running multiple arithmetic operations.

`float` $x =$ | 0.5f |

`float` $y =$ | 1.2f |

`(float)` $x + y =$ | 1.7f |

`__m256` $x =$	0.5f	0.2f	0.6f	0.0f	1.5f	1.3f	2.5f	0.3f
`__m256` $y =$	1.2f	1.8f	0.2f	0.0f	1.2f	0.3f	2.4f	0.3f
`(__m256) _mm256_add_ps(`x`,`y`)` $=$	1.7f	2.0f	0.8f	0.0f	2.7f	1.6f	4.9f	0.6f

SIMD

Lighter-weight parallelism

- Implemented by most architectures as a set of CPU instructions that operate on a separate set of registers.
  - Typically available for fixed vector widths: 128, 256, 512 bits

```
; load next value
mov eax, [rdi]
mov ebx, [rsi]
; multiply values
mul ecx, eax, ebx
; add to accumulator
add r8d, r8d, ecx
; increment input iterators
; 32bit integer
add rdi, 4
add rsi, 4
; decrease remaining count
sub rdx, 1
```

```
; load next block of input
vmovdqu ymm1, [rdi]
vmovdqu ymm2, [rsi]
; multiply vectors
vpmulld ymm3, ymm1, ymm2
; add to accumulator
vpaddd ymm0, ymm0, ymm3
; increment input iterators
; (8 integers, each 4 bytes)
add rdi, 32
add rsi, 32
; decrease remaining count
sub rdx, 8
```

Available SIMD operations

What can we do with CPU vector instructions?

- load/store from memory
- fill vector with a scalar value (broadcast)
- basic arithmetic operations, including bitwise ops
- min/max
- shuffling (permuting values in a vector)
  - only predefined patterns
- masked operations (e.g., blending)
- comparisons

- …many other, often domain-specific instructions to speed up specific computations…

This is where we finished on 2025-03-26.

Use cases for SIMD

Where do you think SIMD could be useful?

Use cases for SIMD

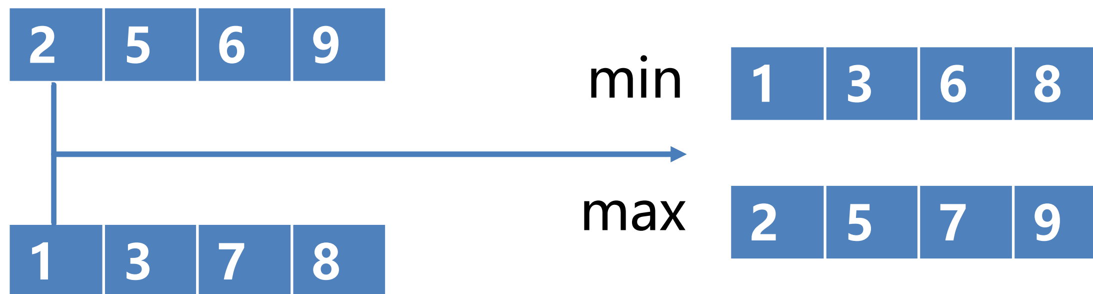
Everywhere...

- almost any string manipulation, memcpy,...
  - Heavily used in most `std::string` implementations.
- operations on data structures with continuous storage
  - Even trees, if you're clever.
- matrix and vector operations
  - As some of you will see next semester, a lot of things can be expressed using matrices.
- image processing (and most other GPU workloads)
- DSP (digital signal processing)
- sorting numeric values (e.g., bitonic sort)
- game simulations (e.g., Reversi from RPH)

Vector min/max

Example use case (used in sorting networks)

Vector min/max with large blocks:



What about smaller blocks? (e.g., pairs)

x3	x2	x1	x0
2	5	6	4

Vector min/max

Example use case (used in sorting networks)

What about smaller blocks? (e.g., pairs)

zero-extend

				x3	x2	x1	x0
0	0	0	0	2	5	6	4

shift right by 1

					x3	x2	x1
0	0	0	0	0	2	5	6

trim

	x3	x2	x1
0	2	5	6

compare with the original vector

x3	x2	x1	x0
2	5	6	4

we need min…

x3	x2	x1	x0
0	2	5	4

Vector min/max

Example use case (used in sorting networks)

What about smaller blocks? (e.g., pairs)

We need the minimum...

x3	x2	x1	x0
0	2	5	4

...but only at even positions.

- min(x0, x1) at position 0
- min(x2, x3) at position 2
- ...

Use a mask to zero out useless values.

x3	x2	x1	x0
0	2	0	4

Result is the OR of these two vectors.

Similarly, find maximum and store it at odd positions.

x3	x2	x1	x0
5	0	6	0

Using SIMD from C/C++

Assembly is somewhat old-school…

- We could use inline assembly, but compiler does not understand the operations and cannot optimize around them, which is a major issue.
  - Also quite cumbersome to write.
- Compilers provide so-called "intrinsics" – arch-specific functions that correspond to specific CPU instructions.
  - Compilers also provide types to represent vectors of various width and types.

```
__m256 exp_vec(__m256 x) {
    __m256 three = _mm256_set1_ps(3.0f);
    __m256 addthree = _mm256_add_ps(x, three);
    __m256 subthree = _mm256_sub_ps(x, three);
    return _mm256_div_ps(
        _mm256_add_ps(_mm256_mul_ps(addthree, addthree), three),
        _mm256_add_ps(_mm256_mul_ps(subthree, subthree), three)
    );
}
```

https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

Using SIMD from C/C++

Autovectorization

- The compiler can also sometimes automatically optimize scalar code to use SIMD instructions = "autovectorization".
- Clang is somewhat good at it, GCC and MSVC not so much.
- Not a panacea, often we need to help the compiler by rearranging the computation, changing data structures,…
- We need to specify the microarchitecture to compile for, defaults are very conservative.

```
void multiply_vec(uint32_t multiplier,
        std::vector<uint32_t>& vec) {
    for (auto& n : vec) {
        n = n * multiplier;
    }
}
```

```
.L4:
vpmulld ymm0, ymm1, YMMWORD PTR [rax]
add     rax, 32
vmovdqu YMMWORD PTR [rax-32], ymm0
cmp     rdx, rax
jne     .L4
```

https://godbolt.org/z/eWf5dEjeP

Using SIMD from modern C++

Intrinsics are not exactly readable.

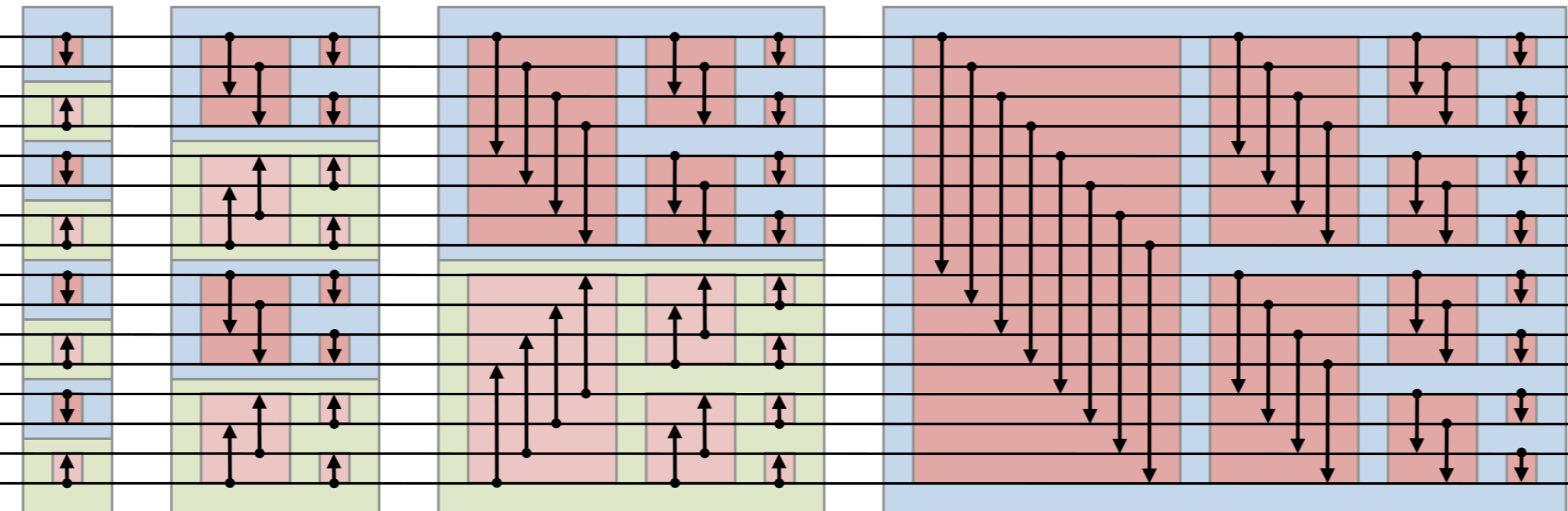
- There is an experimental C++ standard library extension that provides **cross-platform** SIMD vectors.
- `std::experimental::simd`
- Seems on track to be accepted into C++26. For the experimental version, you must use quite modern GCC or Clang (**please update, you'll need it next week**).

```
using vec_f32 = std::experimental::native_simd<float>;

static vec_f32 exp_vec_cpp(vec_f32 x) {
    vec_f32 three{3.0f};
    vec_f32 x_plus_3 = x + three;
    vec_f32 x_minus_3 = x - three;
    return (x_plus_3 * x_plus_3 + three)
         / (x_minus_3 * x_minus_3 + three);
}
```

C++ SIMD example

Vector min/max again

```
size_t half = N / 2;
for (size_t i = 0; i < half; i += vec_f32::size()) {
    vec_f32 low = vec_f32{&data[i], element_aligned};
    vec_f32 high = vec_f32{&data[half + i], element_aligned};
    min(low, high).copy_to(&data[i], element_aligned);
    max(low, high).copy_to(&data[half + i], element_aligned);
}
```



Source: https://commons.wikimedia.org/wiki/File:BitonicSort1.svg

Using SIMD from modern C++

`std::experimental::simd`

- Due to being cross-platform, the exposed operations are somewhat limited compared to what, e.g., modern x64 provides, but good-enough for our purposes.
- Note that available SIMD instructions differ a lot between different architectures, and even a single architecture (e.g., x64, ARM64) provides different SIMD instructions in different generations of CPUs (SSE, AVX, AVX2, AVX-512,...).
  - We'll discuss more in the next lecture.
- For many performance-sensitive workloads, the SIMD code is handwritten for a specific microarchitecture that it will run on, with manually optimized instructions to maximize throughput and saturate all available ALUs.
  - There are many tricks to do common operations quickly, mostly out of scope for PDV.