Programming Parallel Computers 2019

Jukka Suomela · Jaakko Lehtinen · Samuli Laine

Aalto University

ppc.cs.aalto.fi

week 2

Reminder: support available on Slack

- Please feel free to ask anything related to this course on Slack!
  - help with exercises: channels **#cp**, **#mf**, …
  - general questions about parallel programming and course practicalities: **#general**
  - help with non-Maari computers (e.g. macOS): **#tools**
  - more advanced topics: **#advanced**
- We would be happy to hear **feedback on the first week of the course**: please let us know on channel **#feedback** in Slack
  - especially if you had some difficulties solving the first week's exercises

Quick recap

Instruction-level parallelism

Parallel computing resources

- Typical modern desktop CPU: **Intel Core i5-6500** (4 cores)
- Operation: **single-precision floating-point multiplication**
- Latency: ***4 clock cycles***
- Sequential throughput: ***0.25 operations / cycle***
- Parallel throughput: ***64 operations / cycle***
  - we can have 256 operations simultaneously on the fly!
- ***200 billion*** operations per second (clock speed ≈ 3.3 GHz)

Parallel computing resources

- ***Multicore:*** factor **4**
  - 4 cores, each of them can run independent threads
- ***Superscalar:*** factor **2**
  - each core can initiate 2 multiplications per clock cycle
- ***Pipelining:*** factor **4**
  - no need to wait for operations to finish before starting a new one
- ***Vectorization:*** factor **8**
  - each multiplication can process 8-wide vectors

Parallel computing resources

- ***Multicore:*** factor **4**
  - 4 cores, each of them can run independent threads
- ***Superscalar:*** factor **2**
  - each core can initiate 2 multiplications per clock cycle
- ***Pipelining:*** factor **4**
  - no need to wait for operations to finish before starting a new one
- ***Vectorization:*** factor **8**
  - each multiplication can process 8-wide vectors

Instruction-level parallelism (week 1)

Parallel computing resources

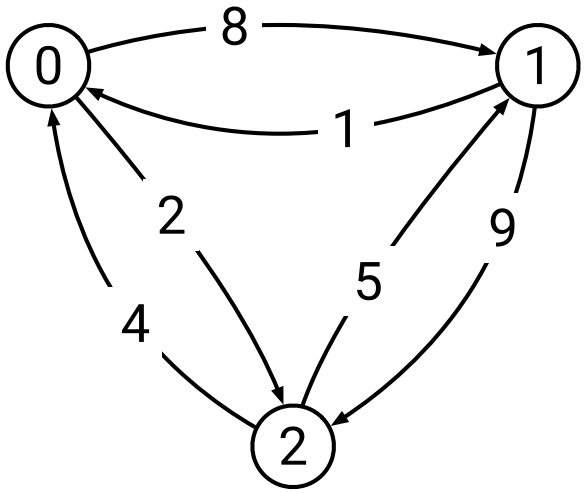
- ***Multicore:*** factor **4**
  - 4 cores, each of them can run independent threads
- ***Superscalar:*** factor **2**
  - each core can initiate 2 multiplications per clock cycle
- ***Pipelining:*** factor **4**
  - no need to wait for operations to finish before starting a new one
- ***Vectorization:*** factor **8**
  - each multiplication can process 8-wide vectors

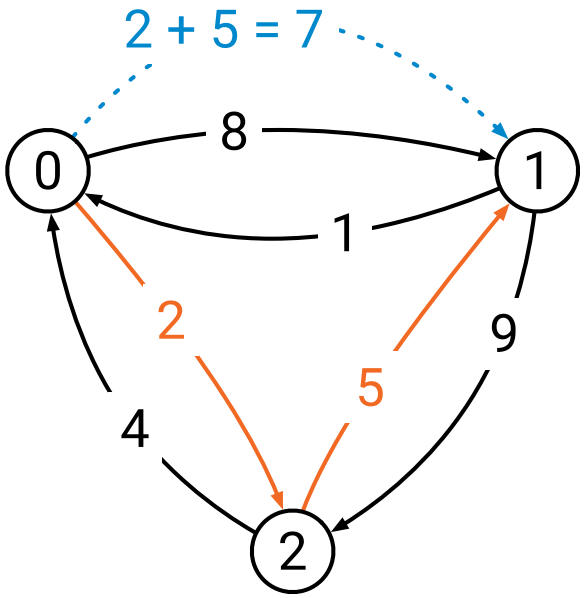
OpenMP (today)

Vector instructions (today)

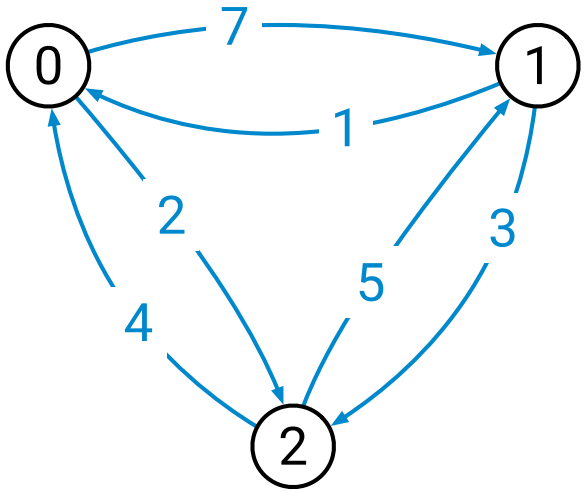
Sample application: cheapest 2-hop path

d (input):





r (output):




```
d[] = { 0, 8, 2,
        1, 0, 9,
        4, 5, 0 }
```

```
r[] = { 0, 7, 2,
        1, 0, 3,
        4, 5, 0 }
```

```
void step(float* r, const float* d, int n) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            float v = infinity;
            for (int k = 0; k < n; ++k) {
                float x = d[n*i + k];
                float y = d[n*k + j];
                float z = x + y;
                v = min(v, z);
            }
            r[n*i + j] = v;
        }
    }
}
```

Version 0

99 seconds

Memory access pattern

```
for (int k = 0; k < n; ++k) {
    float x = d[n*i + k]; // d[0], d[1], d[2], ...
    float y = d[n*k + j]; // d[0], d[4000], d[8000], ...
    float y = t[n*j + k]; // t[0], t[1], t[2], ...
    float z = x + y;
    v = min(v, z);
}
```

Version 1

71 seconds

```
float w[4] = ...
for (int k = 0; k < n/4; ++k) {
    for (int m = 0; m < 4; ++m) {
        float x = d[n*i + k*4 + m];
        float y = t[n*j + k*4 + m];
        float z = x + y;
        w[m] = min(w[m], z);
    }
}
v = min(w[0], w[1],
        w[2], w[3]);
```

Version 2

22 seconds

Instruction-level parallelism

- CPU will look at the instruction stream further ahead
- It will try to find operations that are ***ready for execution***
  - their operands are already known
  - there are execution units available for them
- Example – "`vminss`" instruction:
  - two execution ports in each CPU core that can run this operation
  - each of them can start a new operation at each clock cycle
  - if there are lots of ***independent*** "`vminss`" operations in the code, then we can get a throughput of 2 operations / clock cycle / core

Instruction-level parallelism

Bad: dependent

```
a1 *= a0;
a2 *= a1;
a3 *= a2;
a4 *= a3;
a5 *= a4;
```

Good: independent

```
b1 *= a1;
b2 *= a2;
b3 *= a3;
b4 *= a4;
b5 *= a5;
```

Instruction-level parallelism

Bad: dependent

```
a1 = x[a0];
a2 = x[a1];
a3 = x[a2];
a4 = x[a3];
a5 = x[a4];
```

Good: independent

```
b1 = x[a1];
b2 = x[a2];
b3 = x[a3];
b4 = x[a4];
b5 = x[a5];
```

Instruction-level parallelism

Bad: dependent

```
a1 = min(b1, a0);
a2 = min(b2, a1);
a3 = min(b3, a2);
a4 = min(b4, a3);
a5 = min(b5, a4);
```

Good: independent

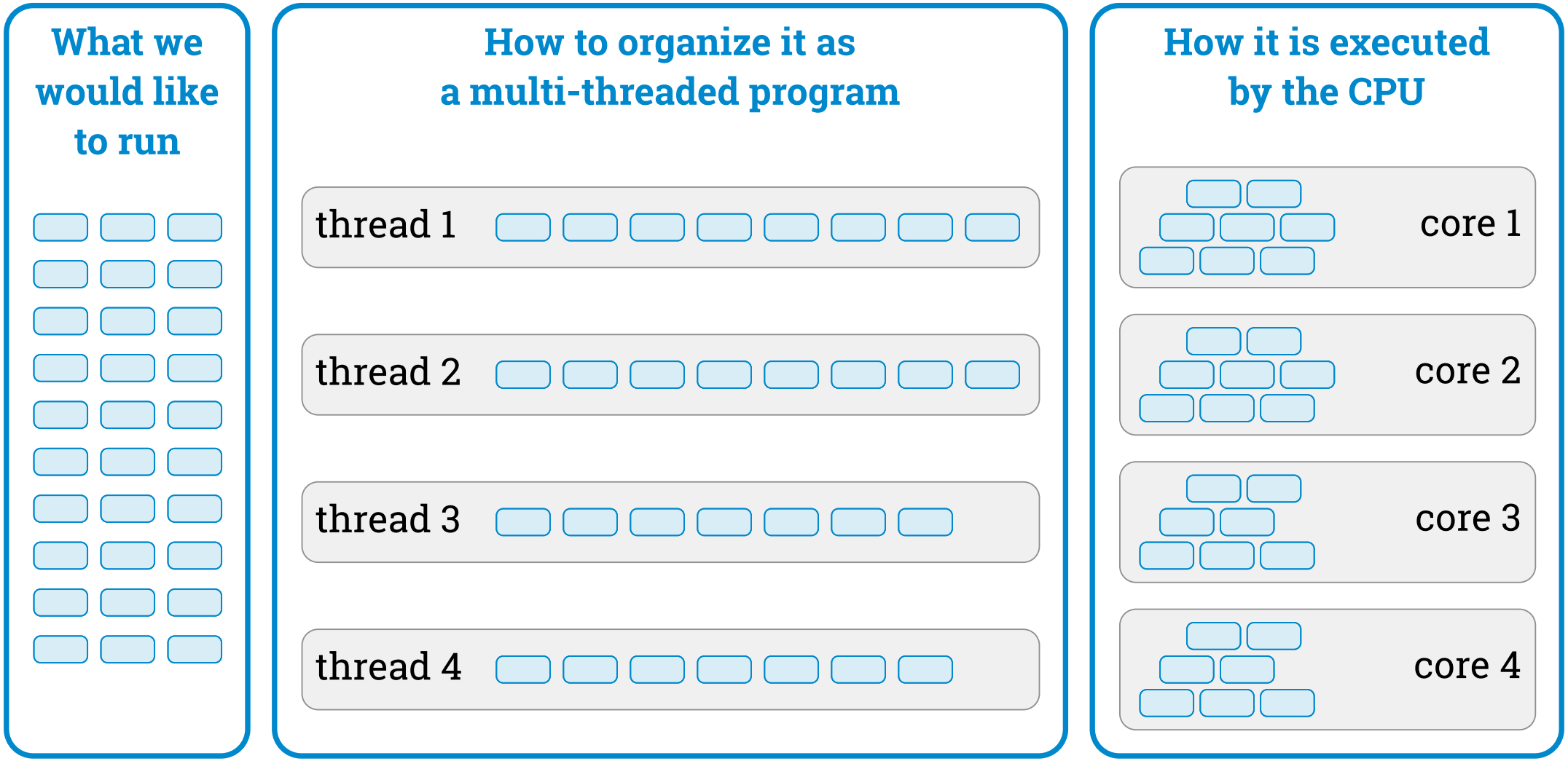
```
b1 = min(b1, a1);
b2 = min(b2, a2);
b3 = min(b3, a3);
b4 = min(b4, a4);
b5 = min(b5, a5);
```

Multicore parallelism

Multiple threads of execution

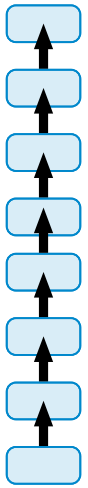
Different scales of parallelism

- **Instruction-level parallelism:**
  - very ***fine-grained***
  - executing machine language instructions in parallel
  - amount of work per independent unit: e.g. **1 multiplication**
  - CPU will take care of it ***automatically*** if there are opportunities
- **Multicore parallelism:**
  - very ***coarse-grained***
  - executing e.g. entire subroutines in parallel
  - amount of work per independent unit: e.g. **1 million multiplications**
  - nobody does it for you — you ***must create multiple threads*** explicitly



Independent operations: multi-threading + instruction-level parallelism easy

What we would like to run



How to organize it as a multi-threaded program

thread 1 

thread 2

thread 3

thread 4

How it is executed by the CPU

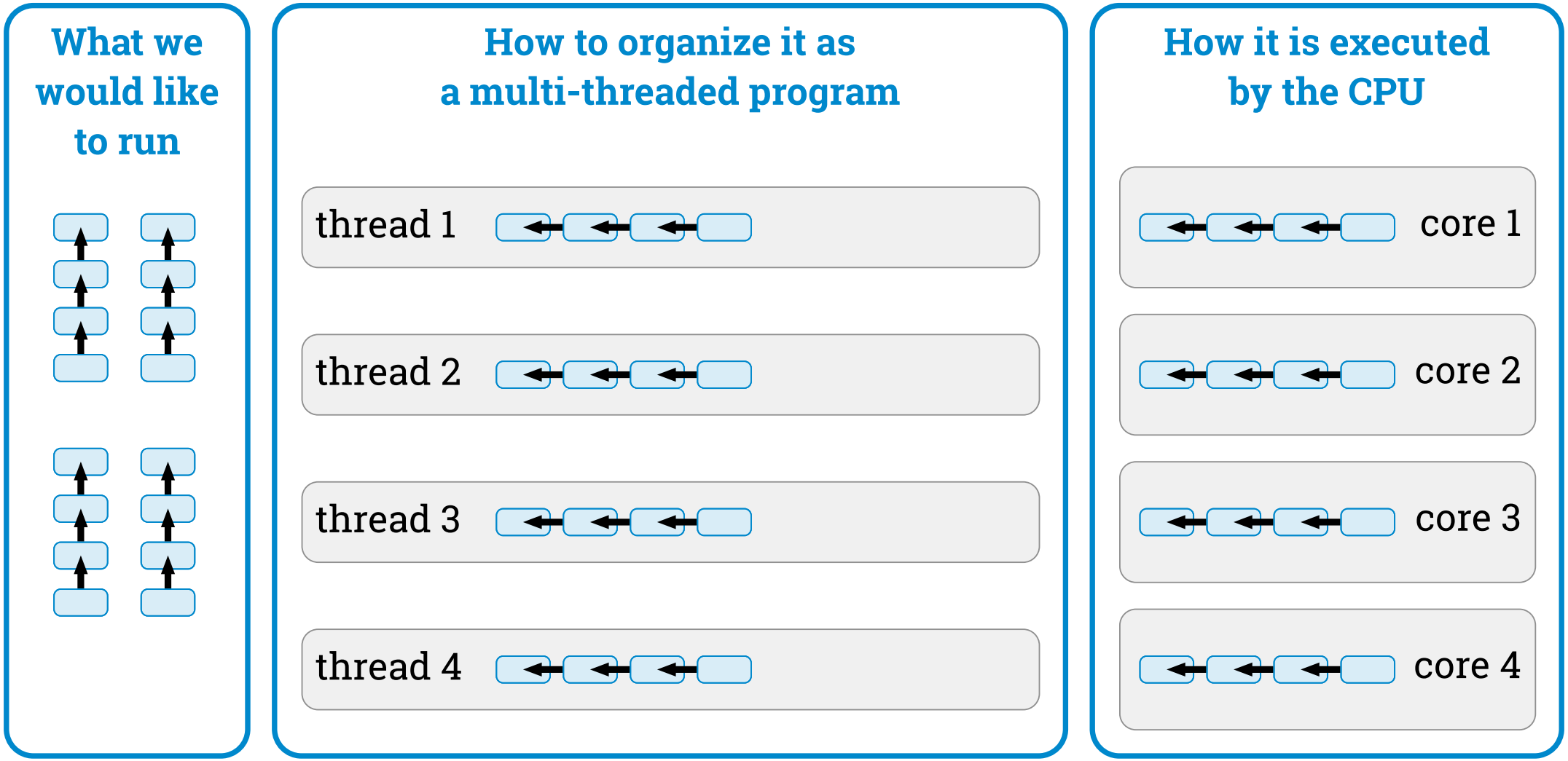
 core 1

core 2

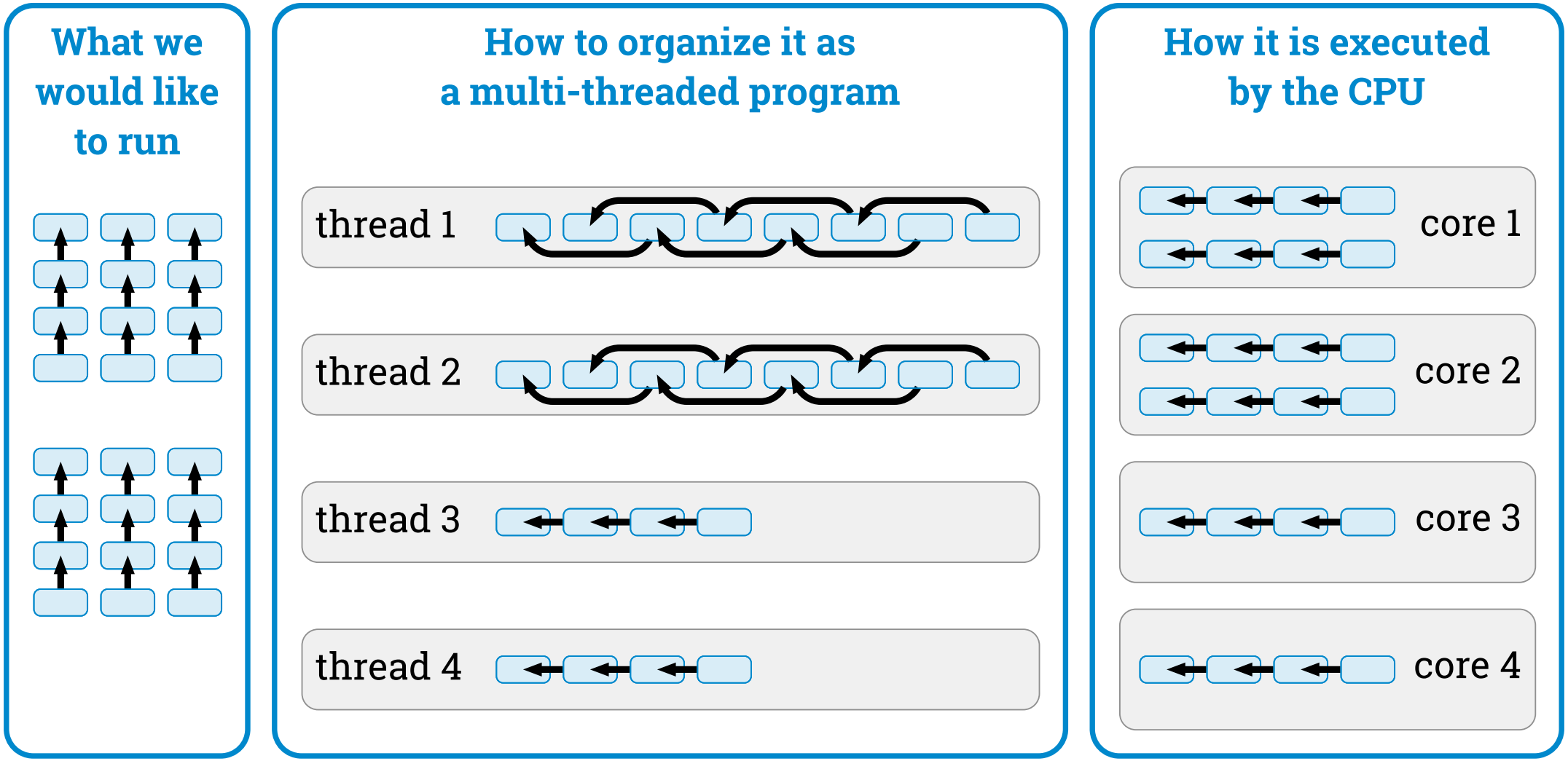
core 3

core 4

Inherently sequential ***dependency chain***: no opportunities for parallelism

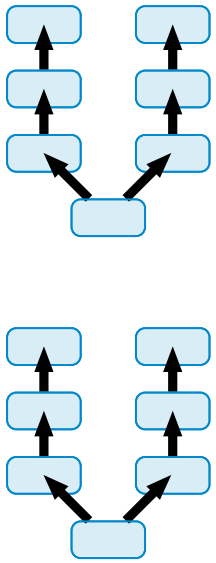


Multi-threading but no opportunities for instruction-level parallelism

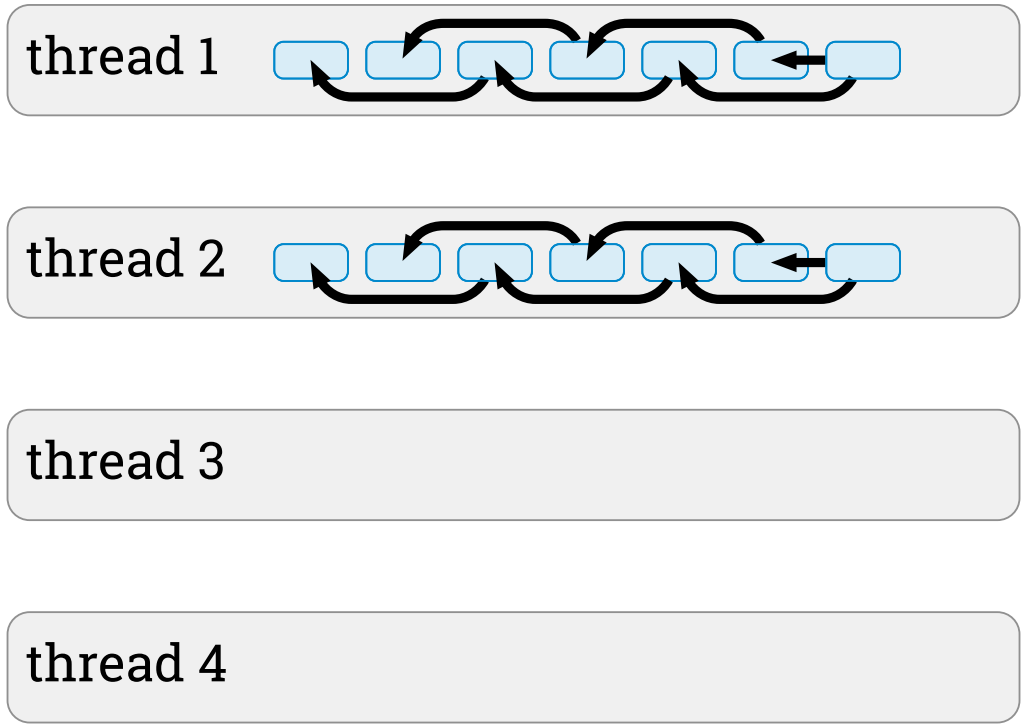


Multi-threading + some opportunities for instruction-level parallelism

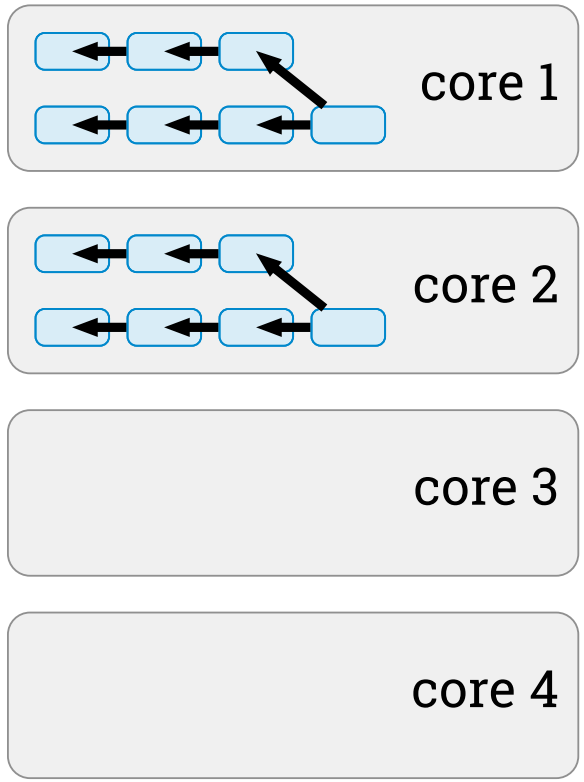
What we would like to run



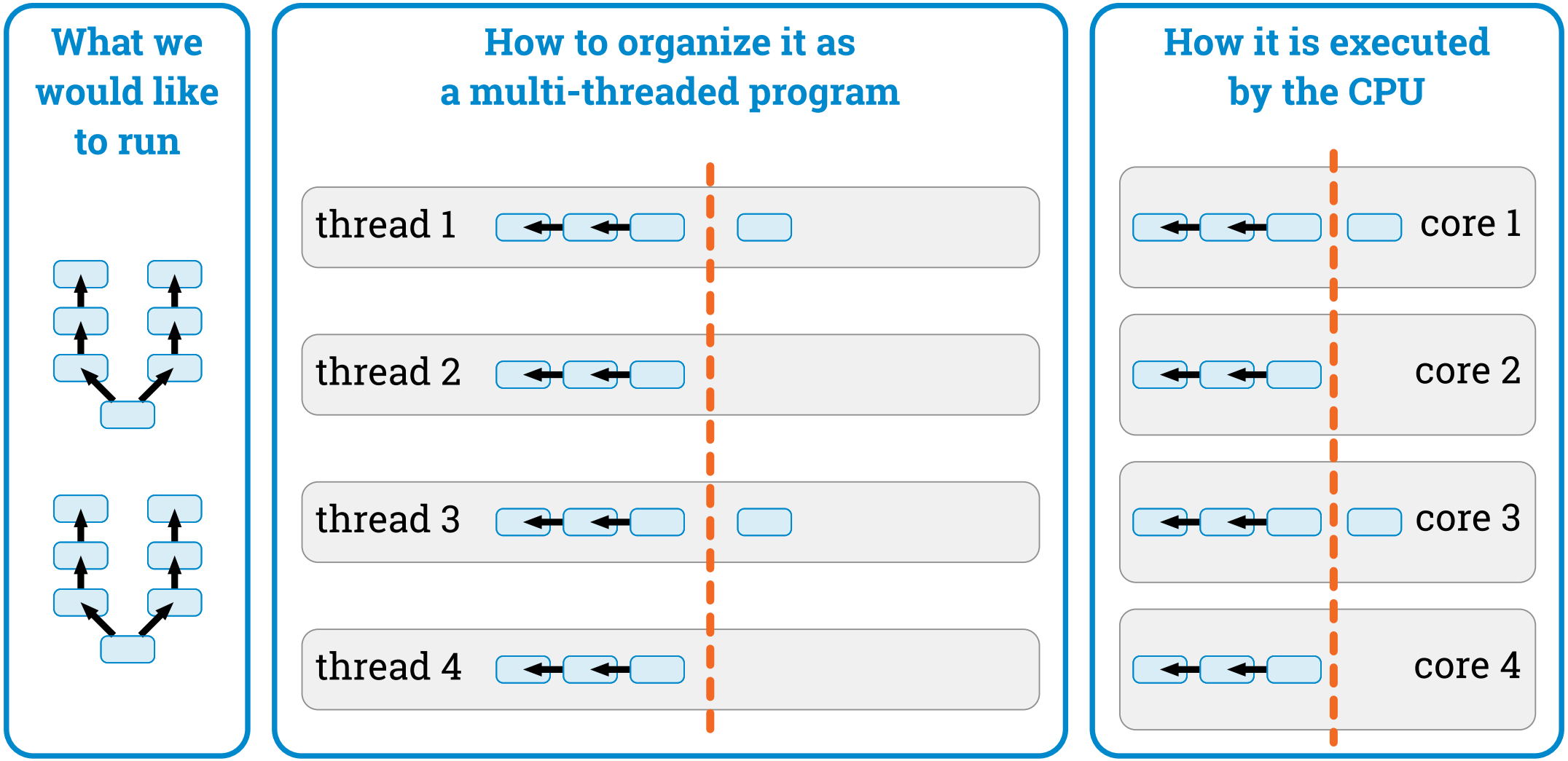
How to organize it as a multi-threaded program



How it is executed by the CPU



Multi-threading + instruction-level parallelism



Multi-threading — some ***synchronization*** between threads needed

Multicore & multithreading

- Assuming:
  - we have a computer with a ***4-core*** CPU
  - we have a program that creates ***4 threads***
  - no other program is active at the same time
- Then ***operating system*** will do the right thing:
  each CPU core will run one thread
  - resources fully utilized
  - at least until some of the threads finish their work…

Multicore & multithreading

- **More threads than cores?**
  - core 1 runs thread 1 for a short while
  - operating system makes a ***context switch***
  - core 1 runs thread 2 for a short while …
- **Fewer threads than cores?**
  - some cores are simply idle
  - there is no way to use 4 cores if you run 1 program with 1 thread

Multicore & multithreading

- How to split long-running computation among multiple threads?
- **Hard way:** use low-level primitives and do everything manually
  - pthreads
  - `std::thread` …
- **Easy way:** use high-level parallelization tools that do almost everything for you
  - ***OpenMP***
  - Intel TBB …

OpenMP

Multithreading made easy

OpenMP parallel for loop

```
for (int i = 0; i < 10; ++i) {
    c(i);
}
```

thread 0: `c(0)` `c(1)` `c(2)` `c(3)` `c(4)` `c(5)` `c(6)` `c(7)` `c(8)` `c(9)`

OpenMP parallel for loop

```
#pragma omp parallel for
for (int i = 0; i < 10; ++i) {
    c(i);
}
```

thread 0: `c(0)` `c(1)` `c(2)`

thread 1: `c(3)` `c(4)` `c(5)`

thread 2: `c(6)` `c(7)`

thread 3: `c(8)` `c(9)`

OpenMP parallel for loop

```
#pragma omp parallel for
for (int i = 0; i < 10; ++i) {
    c(i);
}
```

thread 0: `c(0)` `c(1)` `c(2)`

thread 1: `c(3)` `c(4)` `c(5)`

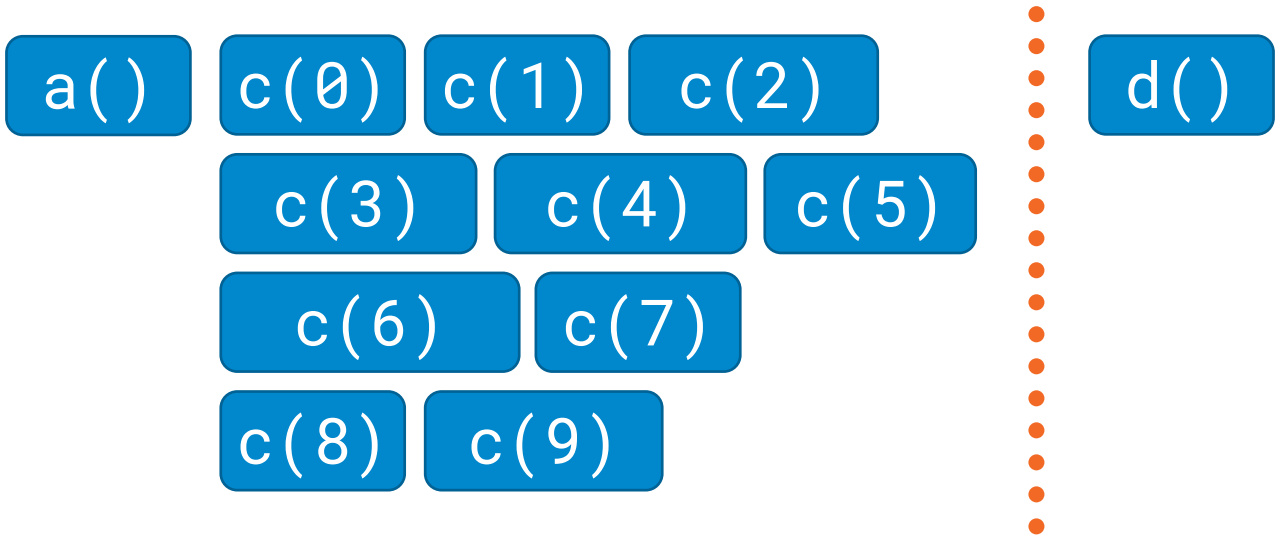
thread 2: `c(6)` `c(7)`

thread 3: `c(8)` `c(9)`

Threads might do different amounts of work

```
a();
#pragma omp parallel for
for (int i = 0; i < 10; ++i) {
    c(i);
}
d();
```

Start & end coordinated



Loop scheduling

Example:
4 threads
40 iterations

```
#pragma omp parallel for
```

- thread 0: iterations 0, 1, …, 9
- thread 1: iterations 10, 11, …, 19

```
#pragma omp parallel for schedule(static,1)
```

- thread 0: iterations 0, 4, 8, …, 36
- thread 1: iterations 1, 5, 9, …, 37

```
#pragma omp parallel for schedule(dynamic,1)
```

- iterations are waiting in a queue
- whenever a thread is available, process the next iteration

```
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        float v = infinity;
        for (int k = 0; k < n; ++k) {
            float x = d[n*i + k];
            float y = d[n*k + j];
            float z = x + y;
            v = min(v, z);
        }
        r[n*i + j] = v;
    }
}
```

Each iteration is independent of each other, could be done in parallel

```
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        float v = infinity;
        for (int k = 0; k < n; ++k) {
            float x = d[n*i + k];
            float y = d[n*k + j];
            float z = x + y;
            v = min(v, z);
        }
        r[n*i + j] = v;
    }
}
```

Each iteration is independent of each other, could be done in parallel

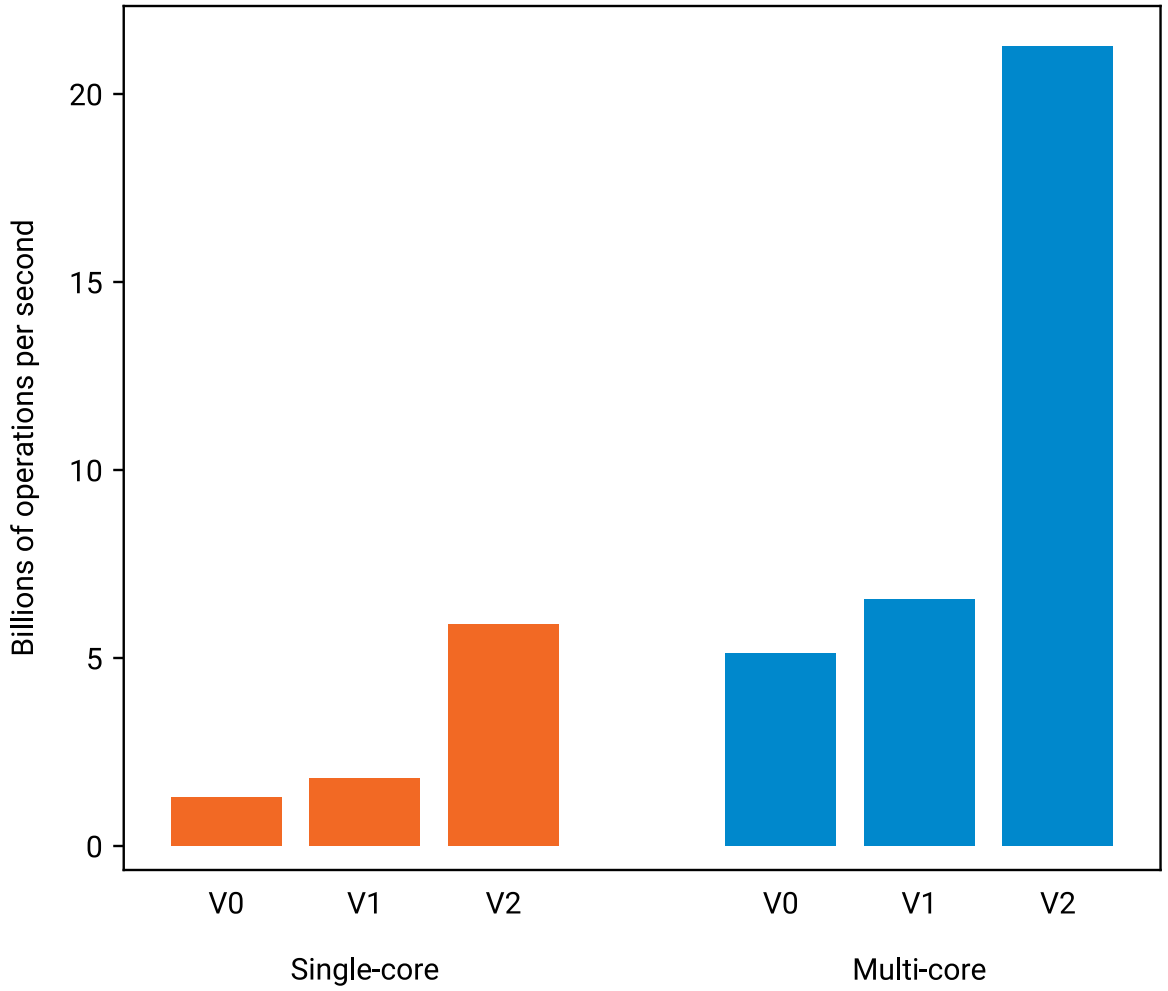
```
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        float v = infinity;
        for (int k = 0; k < n; ++k) {
            float x = d[n*i + k];
            float y = d[n*k + j];
            float z = x + y;
            v = min(v, z);
        }
        r[n*i + j] = v;
    }
}
```

That's all!
It works!

It works!

Multithreading with OpenMP helped by a ***factor of 3.6***

Overall 16 times faster than our starting point



Careful with OpenMP!

No data races!

```
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        float v = infinity;
        for (int k = 0; k < n; ++k) {
            float x = d[n*i + k];
            float y = d[n*k + j];
            float z = x + y;
            v = min(v, z);
        }
        r[n*i + j] = v;
    }
}
```

Private variables (one for each thread)

```
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        float v = infinity;
        for (int k = 0; k < n; ++k) {
            float x = d[n*i + k];
            float y = d[n*k + j];
            float z = x + y;
            v = min(v, z);
        }
        r[n*i + j] = v;
    }
}
```

Shared read-only variables

```
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        float v = infinity;
        for (int k = 0; k < n; ++k) {
            float x = d[n*i + k];
            float y = d[n*k + j];
            float z = x + y;
            v = min(v, z);
        }
        r[n*i + j] = v;
    }
}
```

Each thread writes different elements, no thread reads them

Rules

- Private data:
  - **OK:** everything
- Shared data:
  - **OK:** multiple threads read, nobody writes
  - **OK:** only one thread touches it
  - ***bad:*** one thread reads, another writes
  - ***bad:*** multiple threads write

} "Data race"

Cannot parallelize

```
for (int i = 0; i < 10; ++i) {
    x[i + 1] = f(x[i]);
}
```

Cannot parallelize

```
for (int i = 0; i < 10; ++i) {
    y[0] = f(x[i]);
}
```

OK

```
#pragma omp parallel for
for (int i = 0; i < 10; ++i) {
    y[i] = f(x[i]);
}
```

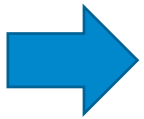
Vector instructions

SIMD – single instruction, multiple data

Modern CPUs are vector processors

- What are examples of ***cheap elementary operations*** that CPUs can do very efficiently with a high throughput?
- **40 years ago:** sum of two 8-bit integers
- **20 years ago:** multiplication of floating point numbers
- **Today:** multiplication of ***vectors*** of floating point numbers

$$x = [\, x_1, x_2, x_3, x_4 \,]$$
$$y = [\, y_1, y_2, y_3, y_4 \,]$$



$$[\, x_1 y_1,\ x_2 y_2,\ x_3 y_3,\ x_4 y_4 \,]$$

Modern CPUs are vector processors

- Sure, you can still do scalar operations:

```
float a = ...
float b = ...
float c = a * b;
```

- But internally the ***CPU will execute it using vector units***!
  - “store **a** to the first element of vector register 0”
  - “store **b** to the first element of vector register 1”
  - “multiply the first elements of vector registers 0 and 1”

Modern CPUs are vector processors

- Modern Intel CPUs have two kinds of registers:
  - `%rax`, `%rbx`, …: ***64-bit integers***
  - `%ymm0`, `%ymm1`, …: ***256-bit vectors***
- How compilers typically use these:
  - **integer registers:** memory addresses, array indexing, loop counters, integer arithmetic …
  - **vector registers:** floating point arithmetic
- But you can do much more with vector registers!

“Vector”: 4 × double or 8 × float

- `float` (single-precision floating-point number): ***32 bits***
- `double` (double-precision floating-point number): ***64 bits***
- Vector registers: ***256 bits***
  - enough space for 4 × double
  - enough space for 8 × float
  - enough space for 32 × byte

“Vector”: 4 × double or 8 × float

- `float` (single-precision floating-point number): ***32 bits***
- `double` (double-precision floating-point number): ***64 bits***
- Vector registers: ***256 bits***
  - enough space for 4 × double
  - enough space for 8 × float
  - enough space for 32 × byte

This text fits in one register!

Vector operations in CPU

- Example: `vaddps %ymm0, %ymm1, %ymm2`
- Behaves like this:

```
z[0] = x[0] + y[0];
z[1] = x[1] + y[1];
z[2] = x[2] + y[2];
z[3] = x[3] + y[3];
z[4] = x[4] + y[4];
z[5] = x[5] + y[5];
z[6] = x[6] + y[6];
z[7] = x[7] + y[7];
```

```
float x[8] ≈ %ymm0
float y[8] ≈ %ymm1
float z[8] ≈ %ymm2
```

Vector operations in C++

- **Hard way:**
  - use “intrinsic functions”
  - code looks like this: `z = _mm256_add_ps(x, y);`
- **Easy way:**
  - use “vector types”
  - code looks like this: `z = x + y;`

Vector types

GCC syntax for saying that “`float8_t`” = vector of 8 × float:

```
typedef float float8_t
  __attribute__ ((vector_size (8 * sizeof(float))));
```

Just copy & paste it whenever you need it…

Vector types

```
float8_t a, b, c;

a = ...;
b = ...;
c = a + b;
```

≈

```
float a[8], b[8], c[8];

a = ...;
b = ...;
c[0] = a[0] + b[0];
c[1] = a[1] + b[1];
c[2] = a[2] + b[2];
c[3] = a[3] + b[3];
c[4] = a[4] + b[4];
c[5] = a[5] + b[5];
c[6] = a[6] + b[6];
c[7] = a[7] + b[7];
```

Similar behavior, but much more efficient code: one vector addition

Vector types

- You can refer to entire vectors — compiler will generate efficient code in which you do element-wise operations:

```
x = (a + b) * c;
```

- You can mix scalars and vectors:

```
x = 3 * a + 2;
```

- You can also refer to individual vector elements if needed:

```
x[0] = 3 * a[1] + 2;
```

Vector types

- You can imagine that vector types are a class or struct that contains 8 floats
  - happens to support convenient overloaded “+”, “*”, etc. operations
- You can freely pass vectors around in ***function parameters***, ***return values***, etc.
  - they are typically kept in registers
- You can allocate ***small arrays of vectors in stack***

Vector types

```
float8_t example(float8_t a, float8_t b) {
    float8_t c[2];
    c[0] = a + b;
    c[1] = a - b;
    float8_t d = c[0] * c[1];
    return d;
}
```

Works fine!

Memory alignment

- Just one complication: care needed with memory allocation!
- Any ***pointer*** to **`float8_t`** has to be ***properly aligned***
  - memory address has to be a multiple of 32 bytes
  - `malloc`, new, etc. do not guarantee that!
- All of these are ***seriously broken***:
  - `float8_t* p = (float8_t*)malloc(n * sizeof(float8_t));`
  - `float8_t* p = new float8_t[n];`
  - `std::vector<float8_t> p(n);`

Program might crash with 50% probability!

Memory alignment

- Always use `posix_memalign` for dynamic memory allocation
  - instead of **`malloc`**, **`new`**, **`std::vector`**, etc.
- See the course material for more details & examples
- See **`common/vector.h`** in your Git repository for helpful definitions that you can directly use
- Remember that local variables, small arrays in stack, function parameters, return values etc. do not need any special care
  - compiler knows about their alignment requirements and does the right job (and often keeps those in registers anyway)

How to use them?

- Some creativity needed!
- Keep in mind that you have access to super-cheap operations that do e.g. “**8 floating point multiplications in parallel**”
- It is ***your job*** as a programmer to figure out how to best use them to speed up your program

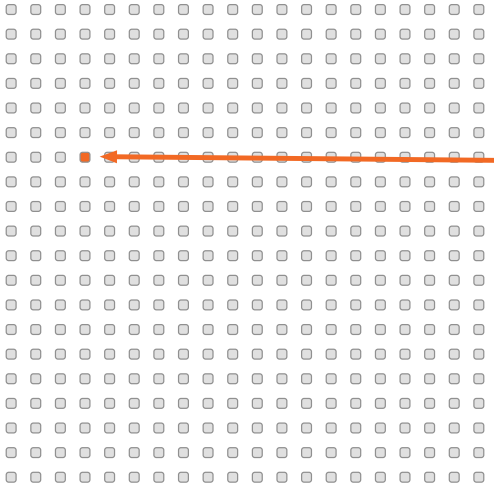
How to use them?

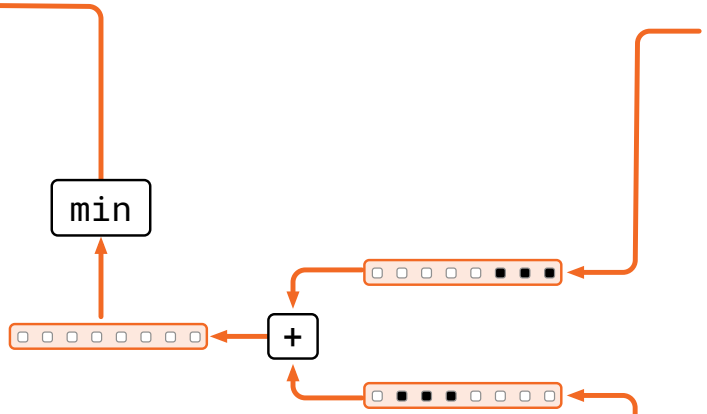
- Typical idea:
  - preprocess your data
  - “***pack***” individual data elements to vectors
  - add ***padding*** if input size not multiple of 8
  - do vector operations
  - “unpack” results from vectors
  - if needed, do some post-processing to turn vector results into normal results
- Make sure you do enough arithmetic operations so that all of this extra work is worth it!

How to use them?

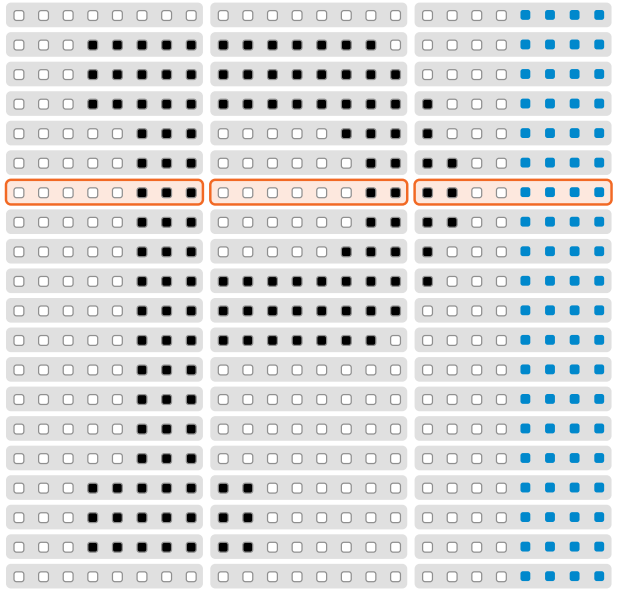
- Packing data, some examples:
  - vector = multiple elements from the same row of input
  - vector = one element from each row of input
  - vector = (R, G, B) triple in image processing
  - vector = one sample from each input channel in audio processing
  - vector = 256 pixels of a monochromatic image
  - vector = 32 characters of text
- Make sure you are mostly doing ***similar operations*** for each vector element
  - e.g. elementwise addition, elementwise multiplication

Output:

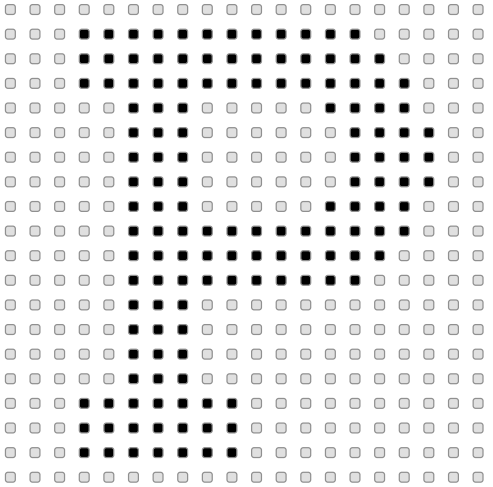




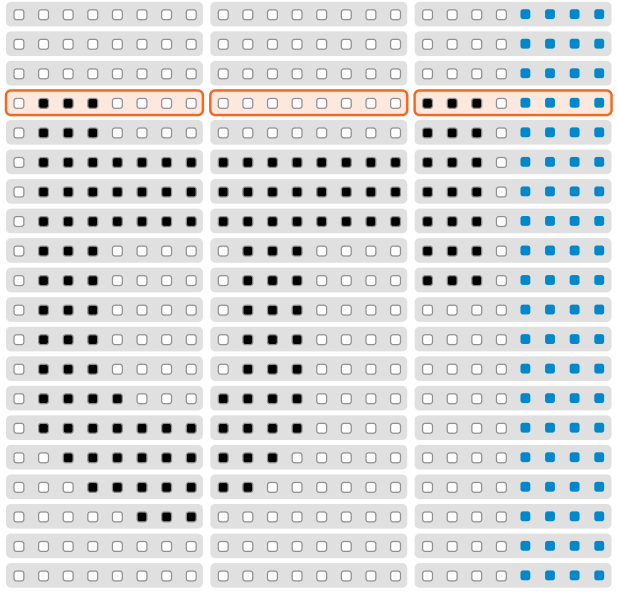
Converted to vectors, padded:



Input:



Transposed:



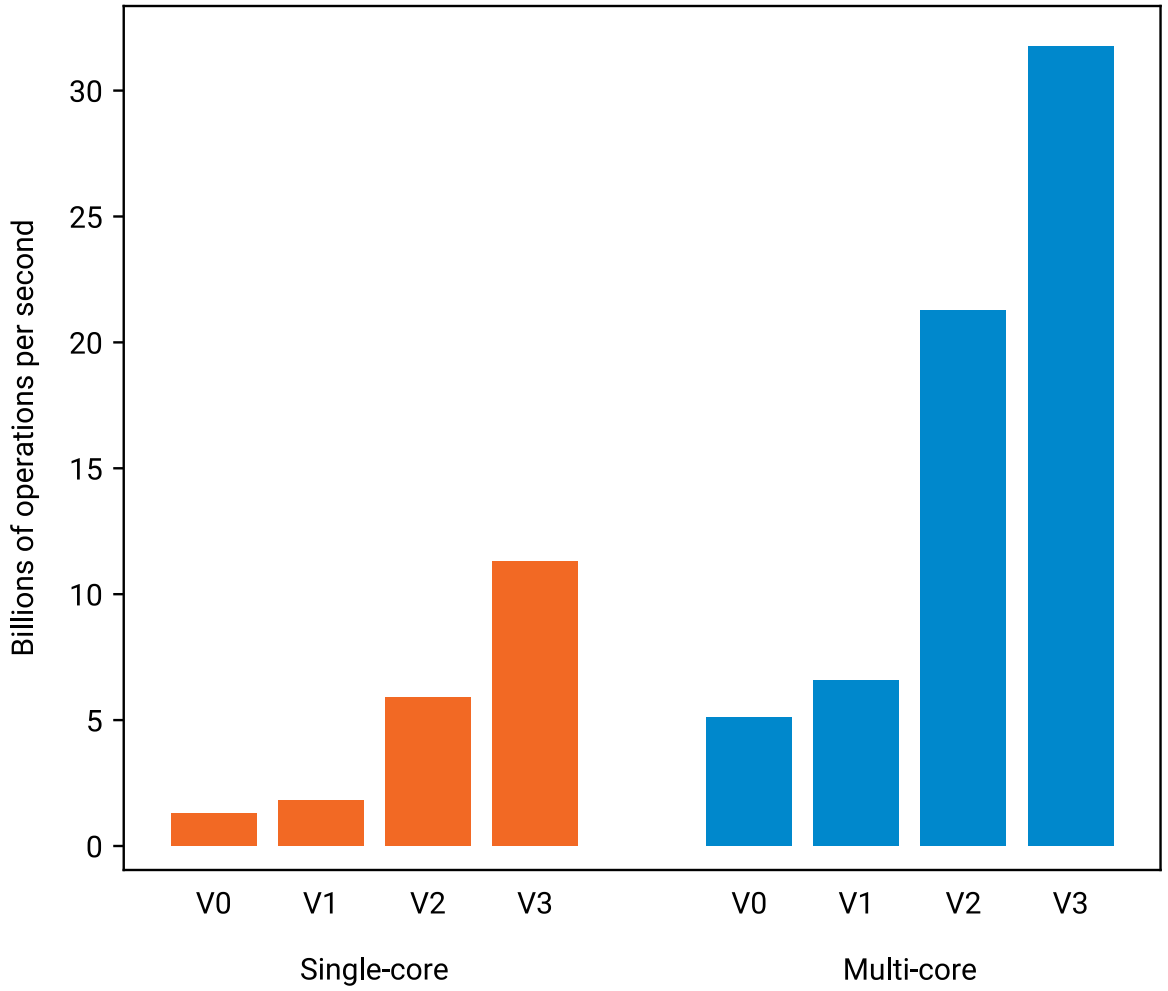
Example: Chapter 2, V3

Vectorization

V2: instruction-level parallelism

V3: vectorization

Running time improved from **99 s** to **4 s**



Summary

All forms of parallelism, simultaneously

Parallel computing resources

- ***Multicore:*** factor **4**
  - 4 cores, each of them can run independent threads
- ***Superscalar:*** factor **2**
  - each core can initiate 2 multiplications per clock cycle
- ***Pipelining:*** factor **4**
  - no need to wait for operations to finish before starting a new one
- ***Vectorization:*** factor **8**
  - each multiplication can process 8-wide vectors

You will need all of this

- ***Multithreading***
  - otherwise you are using only 1 CPU core
- ***Vector instructions***
  - otherwise you are not making good use of vector instructions
- ***Instruction-level parallelism***
  - you need to make sure that in each thread there are lots of independent vector operations that can be executed

Data reuse will be necessary

- Performance of a typical 4-core CPU:
  - could do ***64 floating-point additions*** per clock cycle
  - main memory bandwidth: can fetch enough data for ≈ ***1.25 floating-point additions*** per clock cycle
  - we can only afford to fetch **2%** of our input from main memory!
- Lots of data reuse needed:
  - reusing what you have got from main memory to ***caches***
  - reusing what you have got from caches to ***registers***
- **More about this next week!**