Programming Parallel Computers 2019

Jukka Suomela · Jaakko Lehtinen · Samuli Laine

Aalto University

ppc.cs.aalto.fi

week 6

Quick recap

Programming modern CPUs & GPUs

Computers are massively parallel

- **Huge amounts of computing power available**
  - CPUs: ***hundreds of billions*** of operations per second
  - GPUs: even more
- **All new performance comes from parallelism**
  - > ***factor 100 difference*** between sequential and parallel performance
- **Memory is slow**
  - > ***factor 50 difference*** between memory bandwidth and arithmetic throughput

Parallel computing resources

CPU

- Pipelined arithmetic units:
  1 new operation per cycle
- ***Vector*** operations:
  8 similar operations
- Lots of arithmetic units:
  4 × 2 vector operations in parallel e.g. for FMA

GPU

- Pipelined arithmetic units:
  1 new operation per cycle
- "***Warp***" of "***threads***":
  32 similar operations
- Lots of arithmetic units:
  5 × 4 warps executed in parallel e.g. for FMA

Programmer's view

CPU

- ***Instruction-level parallelism important***
- Everything else is sequential unless explicitly parallelized

```
#pragma omp
float8_t
```

GPU

- Instruction-level parallelism not so important
- The ***only*** primitive that we can use is inherently parallel

```
f<<<blocks, threads>>>()
```

Key ideas

- Design algorithms so that there are lots of ***independent operations***
  - needed for **any kind of parallelism**
- Preferably lots of ***similar*** independent operations
  - needed for SIMD (vectors on CPUs)
  - needed for SIMT (warps on GPUs)
- Try to do ***lots of arithmetic operations per memory access***
  - otherwise the CPU will be mostly idle, **waiting for some data to process**

Algorithmic ideas

Designing algorithms that can be parallelized

Three concepts

- ***Computational problem***
  - specifies what we want
  - e.g.: **sort *n* numbers**
- ***Algorithm*** that solves it efficiently
  - tells how to solve it, on a somewhat abstract level
  - e.g.: **quicksort**
- Efficient ***implementation*** of the algorithm
  - actual C++ code that works well on real computers
  - e.g.: **std::sort** implementation in the GNU C++ Library

Three concepts

- ***Computational problem***
  - specifies what we want
  - e.g.: **sort *n* numbers**
- ***Parallel algorithm*** that solves it efficiently
  - tells how to solve it, on a somewhat abstract level
  - e.g.: **parallel quicksort**
- Efficient ***parallel implementation*** of the algorithm
  - actual C++ code that works well on real computers
  - e.g.: **__gnu_parallel::sort**

Three concepts

- ***Computational problem***
  - specifies what we want
  - e.g.: **sort *n* numbers**
- ***Parallel algorithm*** that solves it efficiently
  - tells how to solve it, on a somewhat abstract level
  - e.g.: **parallel quicksort**
- Efficient ***parallel implementation*** of the algorithm
  - actual C++ code that works well on real computers
  - e.g.: **__gnu_parallel::sort**

Independent operations, opportunities for parallelism

Caches, registers, ILP, AVX, OpenMP, CUDA …

We need new kinds of algorithms

- Some classical algorithms have opportunities for parallelism
  - example: many “divide and conquer” algorithms
- However, often we need to design entirely new algorithms!
- Wrong question:
  “how to implement this algorithm on a parallel computer?”
- Right question:
  “how to design a parallel algorithm for this problem?”

Parallel algorithms: terminology

- ***"Processor":***
  - any form of parallelism often is described as if we had *p* processors
  - abstraction — **shows what can be done independently in parallel**
  - practical realizations: superscalar execution, pipelining, CPU vector lanes, CPU threads, GPU threads, multiple GPUs, computing cluster …
- ***"Work":*** total number of operations by all processors
- ***"Depth":*** longest sequential dependency chain
  - how long does it take even if we had infinitely many processors

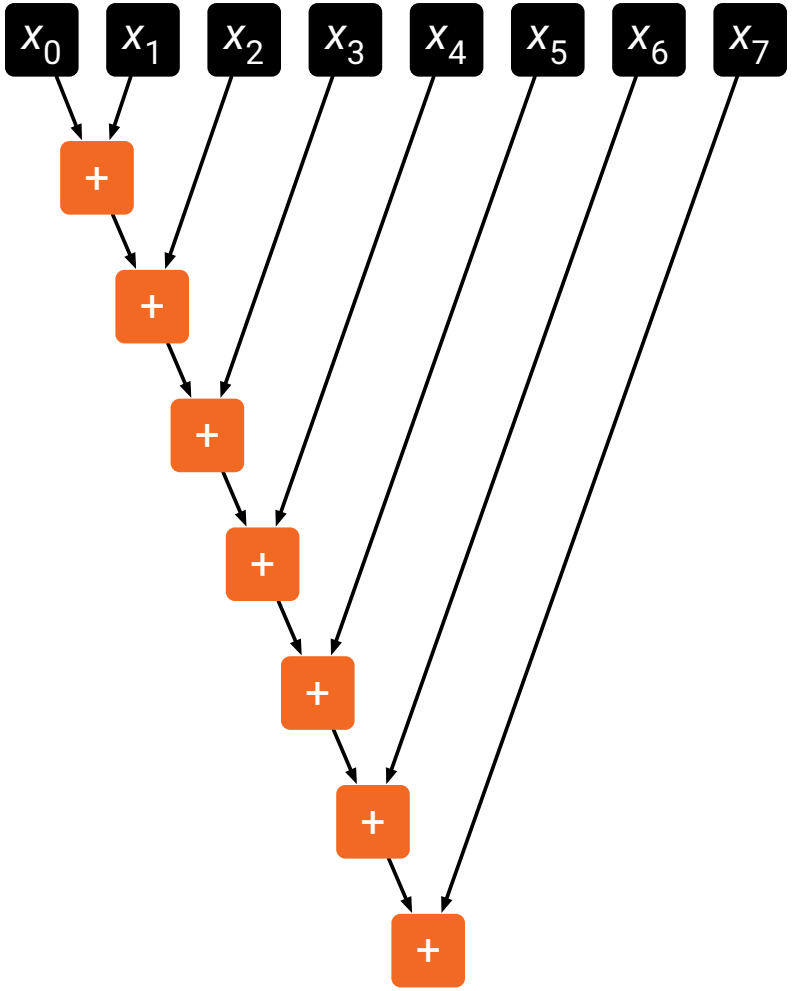
Sum

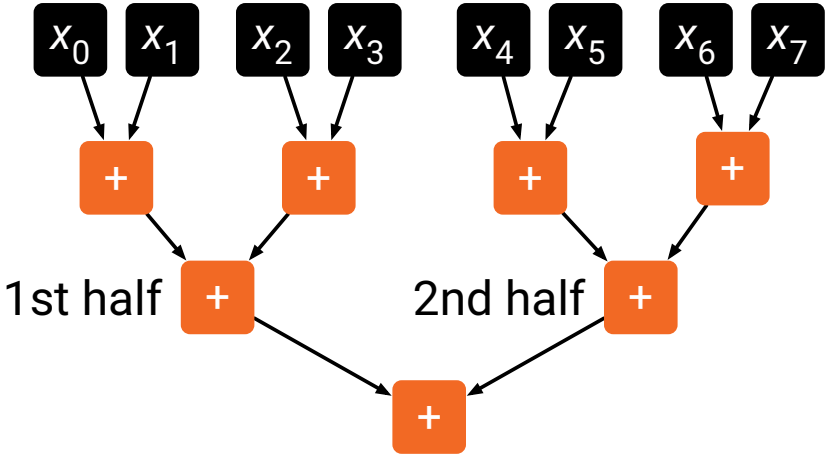
- Problem: calculate sum of $X = \{ x_0, x_1, \ldots, x_{n-1} \}$
- Trivial sequential algorithm
- Recursive parallel algorithm **sum(*X*)**:
  - if $|X| \leq 2$:
    - use sequential algorithm
  - if $|X| > 2$:
    - split *X* in two halves *A* and *B*
    - ***in parallel***, calculate *a* = **sum(*A*)** and *b* = **sum(*B*)**
    - return *a* + *b*

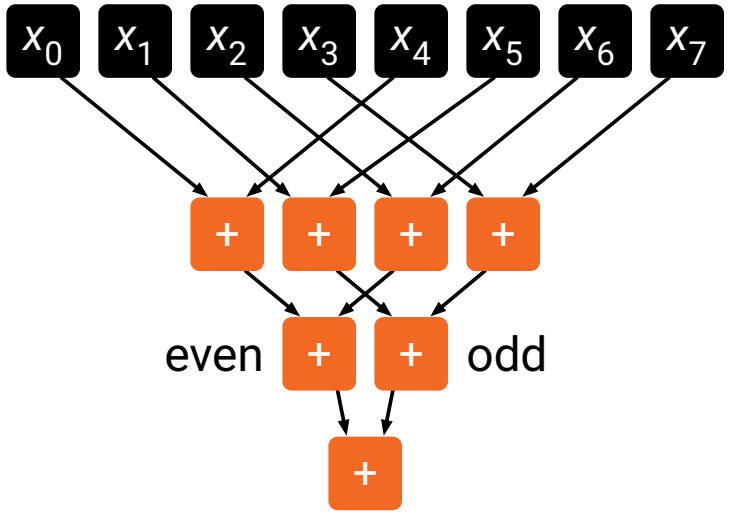
Some examples:

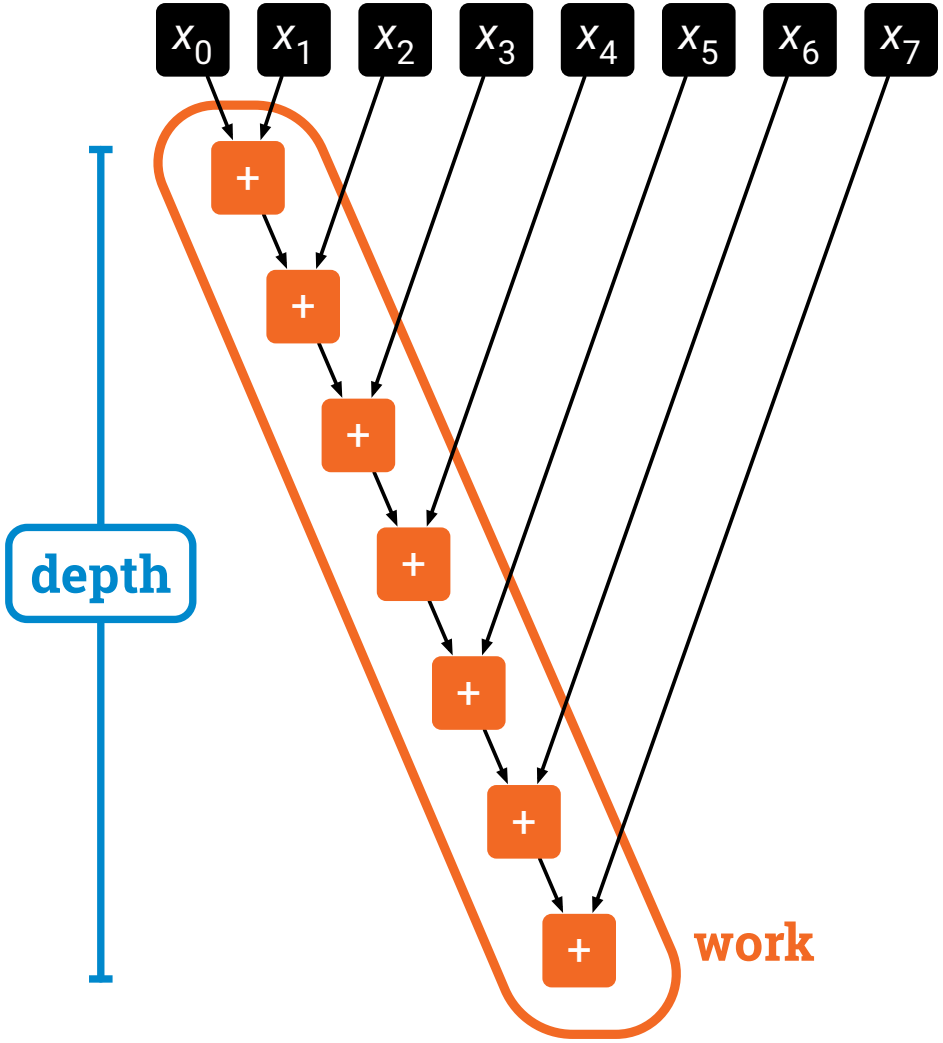
A = first half
B = second half

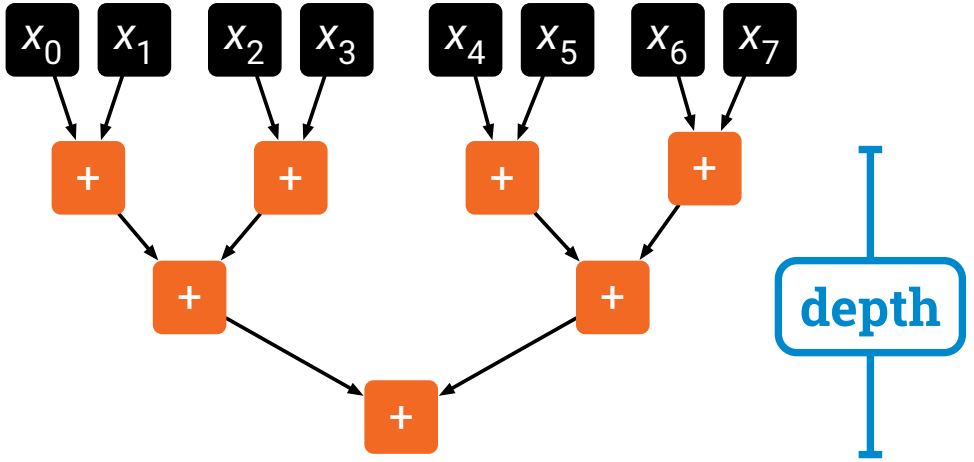
A = odd indexes
B = even indexes

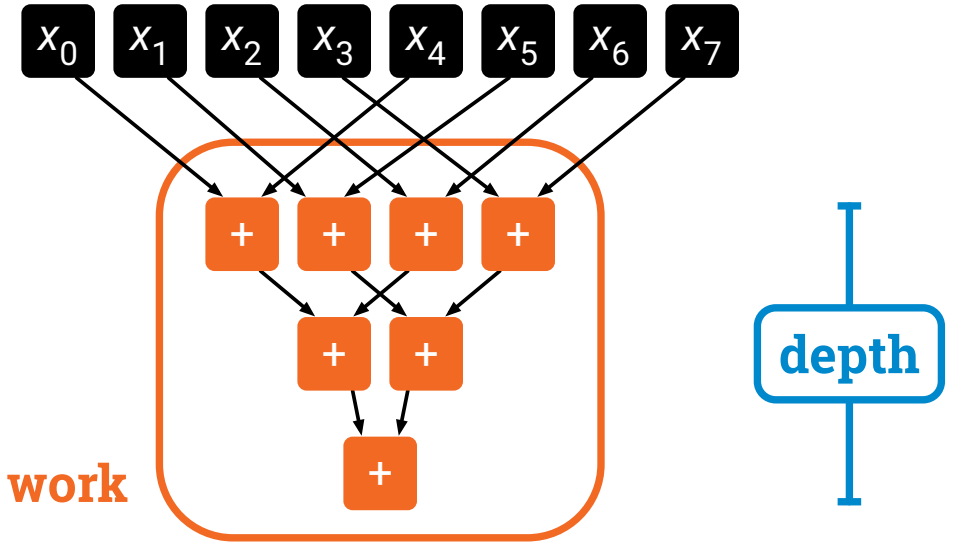

x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
+


x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
1st half
2nd half
+

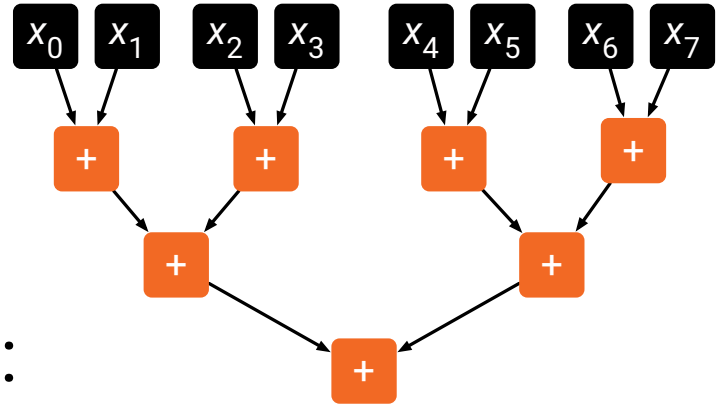

x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
even
odd
+


x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
+
depth
work


x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
+
depth

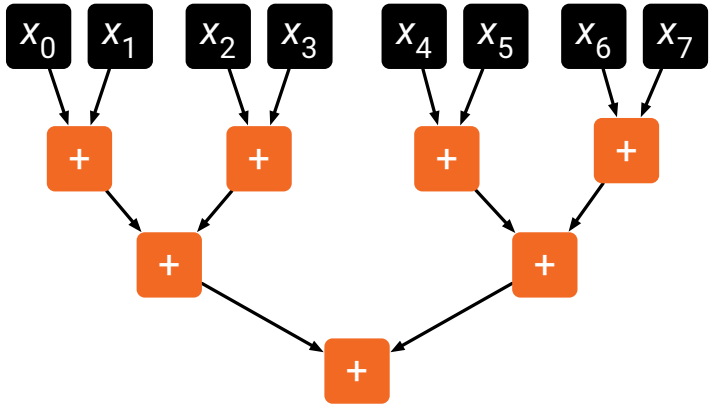

x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
+
work
depth

Sum



- ***In theory*** we could parallelize sums as follows:
  - $O(n)$ processors, $O(n)$ work, $O(\log n)$ depth
- ***In practice*** this shows that there is **lots of potential for parallelism**, without doing much extra work
  - ***do not*** try to implement the recursive algorithm directly, use it as a source of ideas of how you could reorganize computation
  - just use enough levels to fully utilize your hardware
    - e.g.: 3 levels for OpenMP, 3 levels for SIMD, 2 levels for ILP?
  - usually we don't have n "processors" but only e.g. 256

Sum



- The same idea works for any ***associative binary operation***:
  - sum
  - product
  - max
  - min
  - bitwise and, or, xor
  - matrix multiplication …

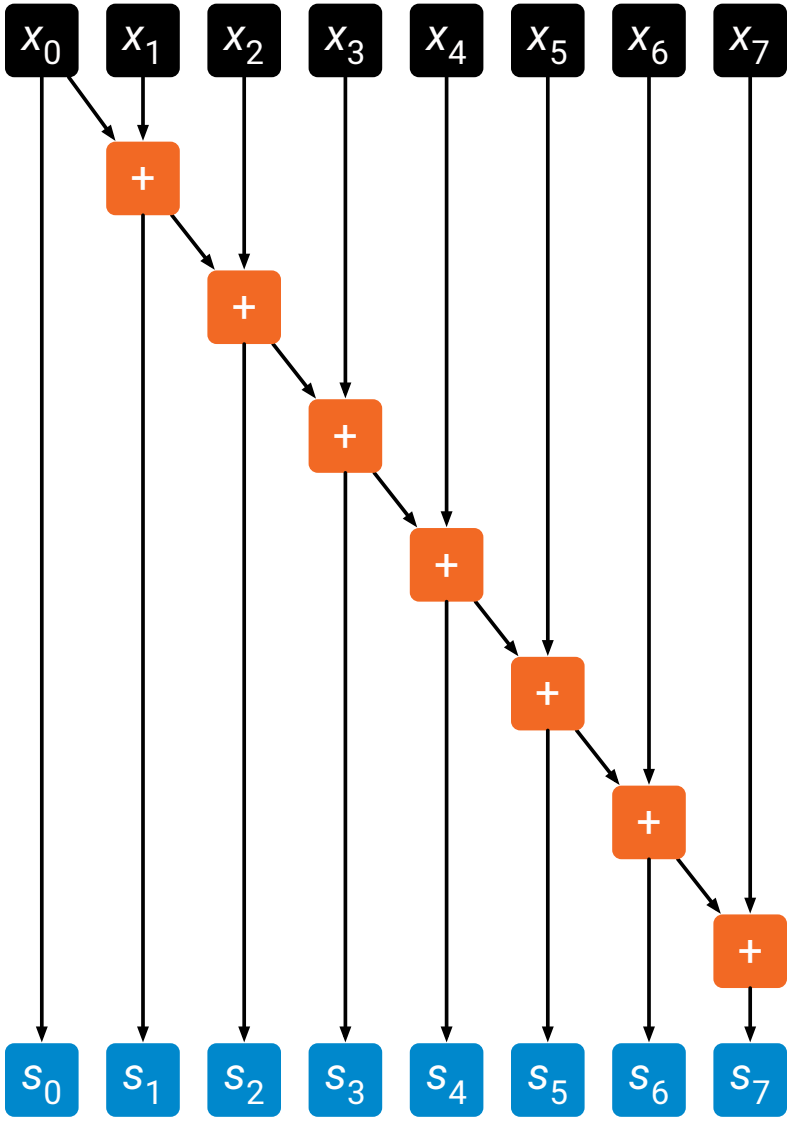
What can be parallized?

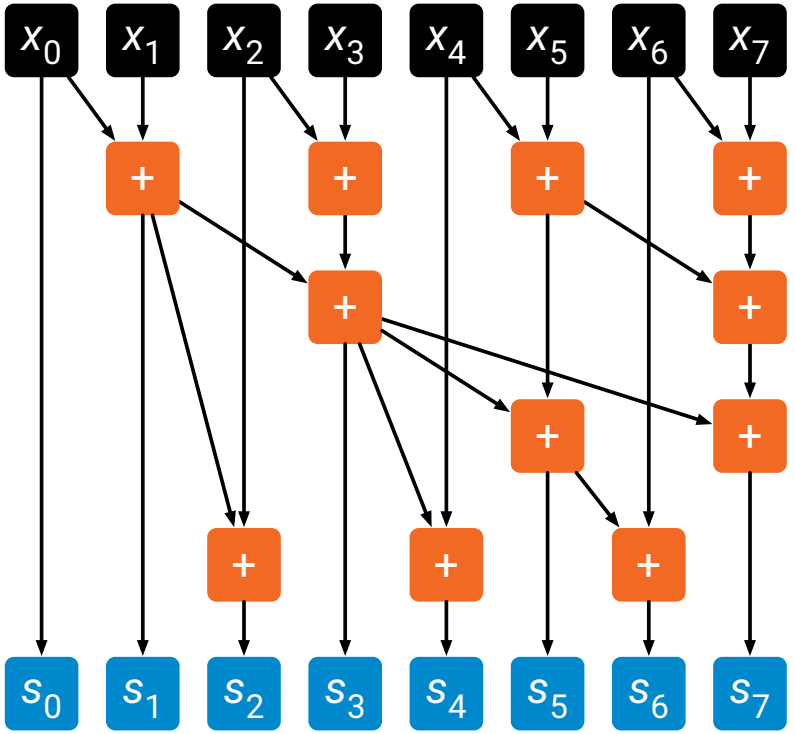
- Nobody knows yet!
- Efficient parallel algorithms exist for many problems
- Some evidence that some problems are very hard to parallelize
  - some useful keywords for further study: complexity class **NC**, **P-complete** problems, conjecture **P ≠ NC**
- Even if the problem is really hard to parallelize:
  - could you solve ***multiple instances*** in parallel?
  - if you are calculating e.g. $f(f(f(f(f(x)))))$, could you find opportunities for ***small-scale parallelism*** inside f ?

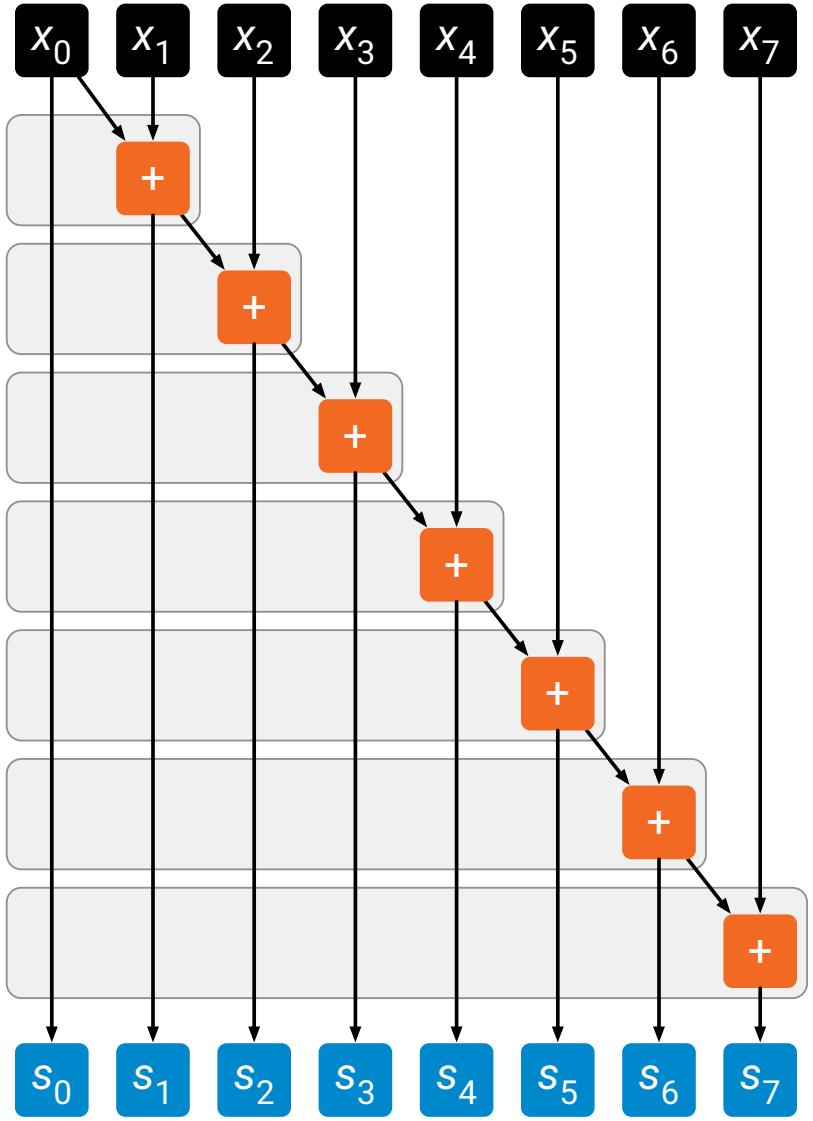
Prefix sum

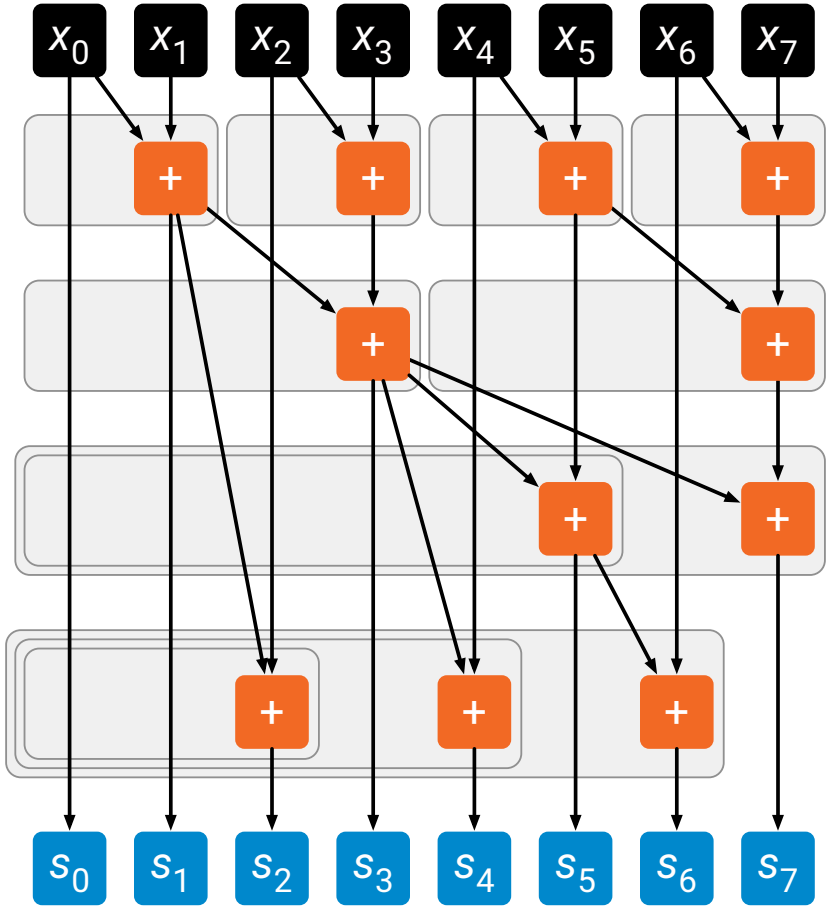
- Input: $x_0, x_1, \ldots, x_{n-1}$
- Output:
  - $s_0 = x_0$
  - $s_1 = x_0 + x_1$

    …
  - $s_{n-1} = x_0 + x_1 + \ldots + x_{n-1}$
- Trivial sequential implementation
- Can be parallelized efficiently


x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
+
s_0 s_1 s_2 s_3 s_4 s_5 s_6 s_7


x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
+
s_0 s_1 s_2 s_3 s_4 s_5 s_6 s_7


x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
+
s_0 s_1 s_2 s_3 s_4 s_5 s_6 s_7


x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
+
s_0 s_1 s_2 s_3 s_4 s_5 s_6 s_7

Prefix sum

- Simple practical implementation for *p* threads:
  - split in *p* parts
  - ***in parallel:*** calculate ***y(i)*** = sum of part *i*
  - ***sequentially:*** calculate ***z(i)*** = sum of all parts up to *i*
    - using *y*(*i*) values that we just calculated
  - ***in parallel:*** calculate prefix sums for each part
    - part *i* uses z(*i*) as the initial value

x:	1	2	3	4	5	6	7	8	9	10	11	12
y:				**10**				**26**				**42**
z:				**10**				**36**				**78**
s:	**1**	**3**	**6**	**10**	**15**	**21**	**28**	**36**	**45**	**55**	**66**	**78**

p = 3 threads
n = 12 input values

Prefix sum

- Simple practical implementation for *p* threads:
  - split in *p* parts
  - ***in parallel:*** calculate ***y(i)*** = sum of part *i*
  - ***sequentially:*** calculate ***z(i)*** = sum of all parts up to *i*
    - using *y*(*i*) values that we just calculated
  - ***in parallel:*** calculate prefix sums for each part
    - part *i* uses z(*i*) as the initial value

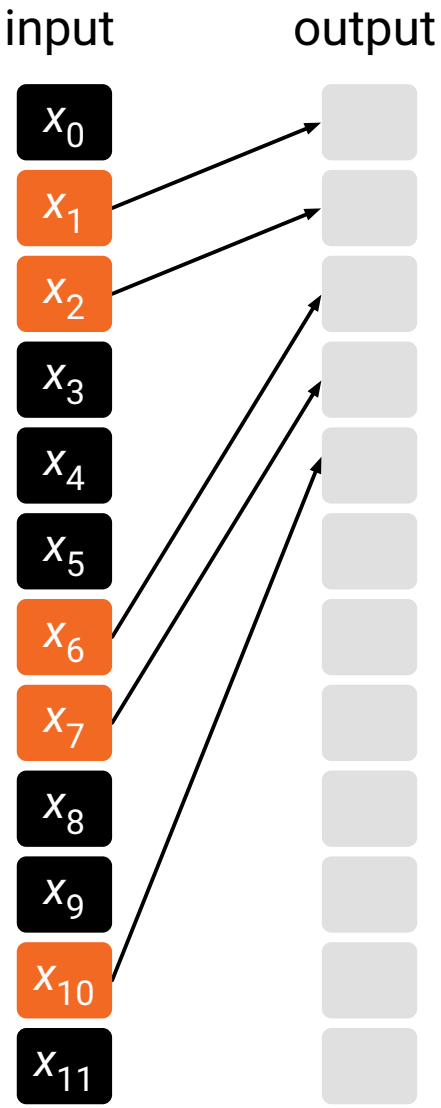
Smaller prefix sum calculation, could be further parallelized if needed

x:	1	2	3	4	5	6	7	8	9	10	11	12
y:				**10**				**26**				**42**
z:				**10**				**36**				**78**
s:	**1**	**3**	**6**	**10**	**15**	**21**	**28**	**36**	**45**	**55**	**66**	**78**

p = 3 threads
n = 12 input values

Select

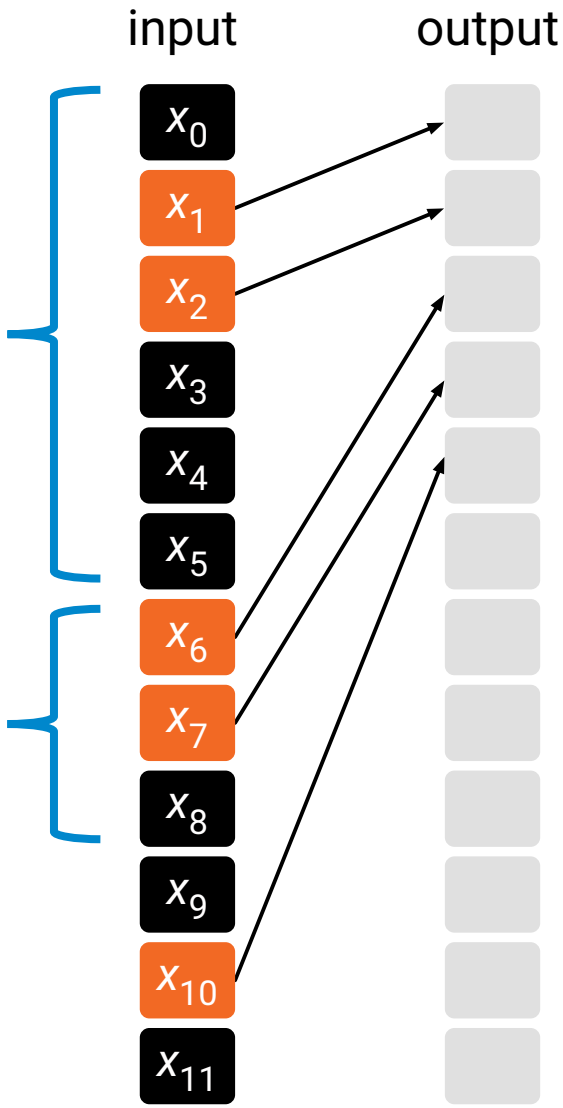
- Find all ***orange*** elements and put them in the output array
  - cf. **“partition”** in quicksort
- Trivial sequential algorithm
- How to parallelize?



Select

If we know how many orange elements are here…

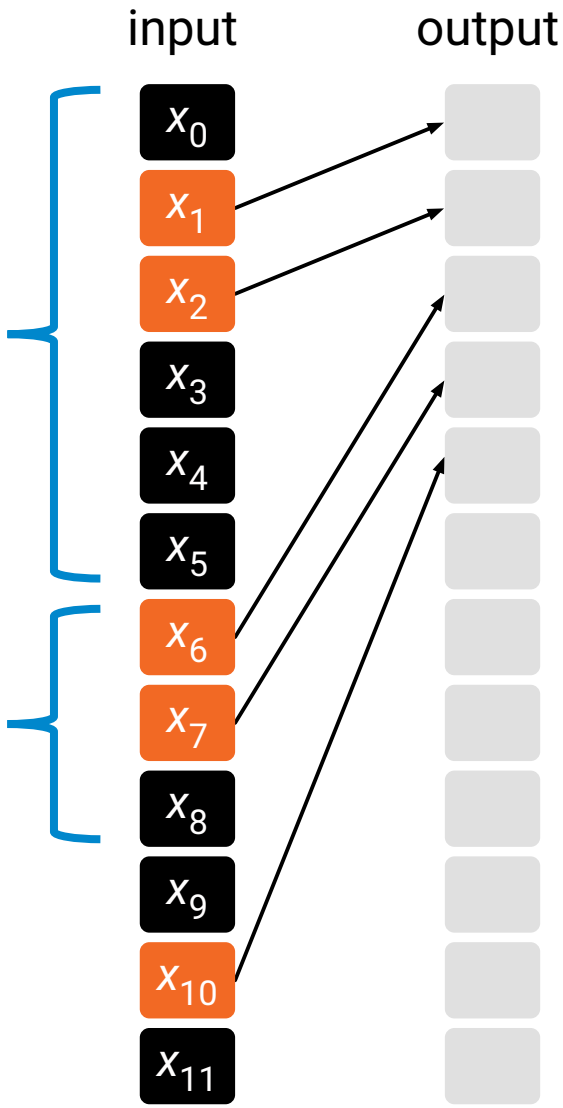
… we know where to put these elements



Select

If we know how many orange elements are here…

… we know where to put these elements

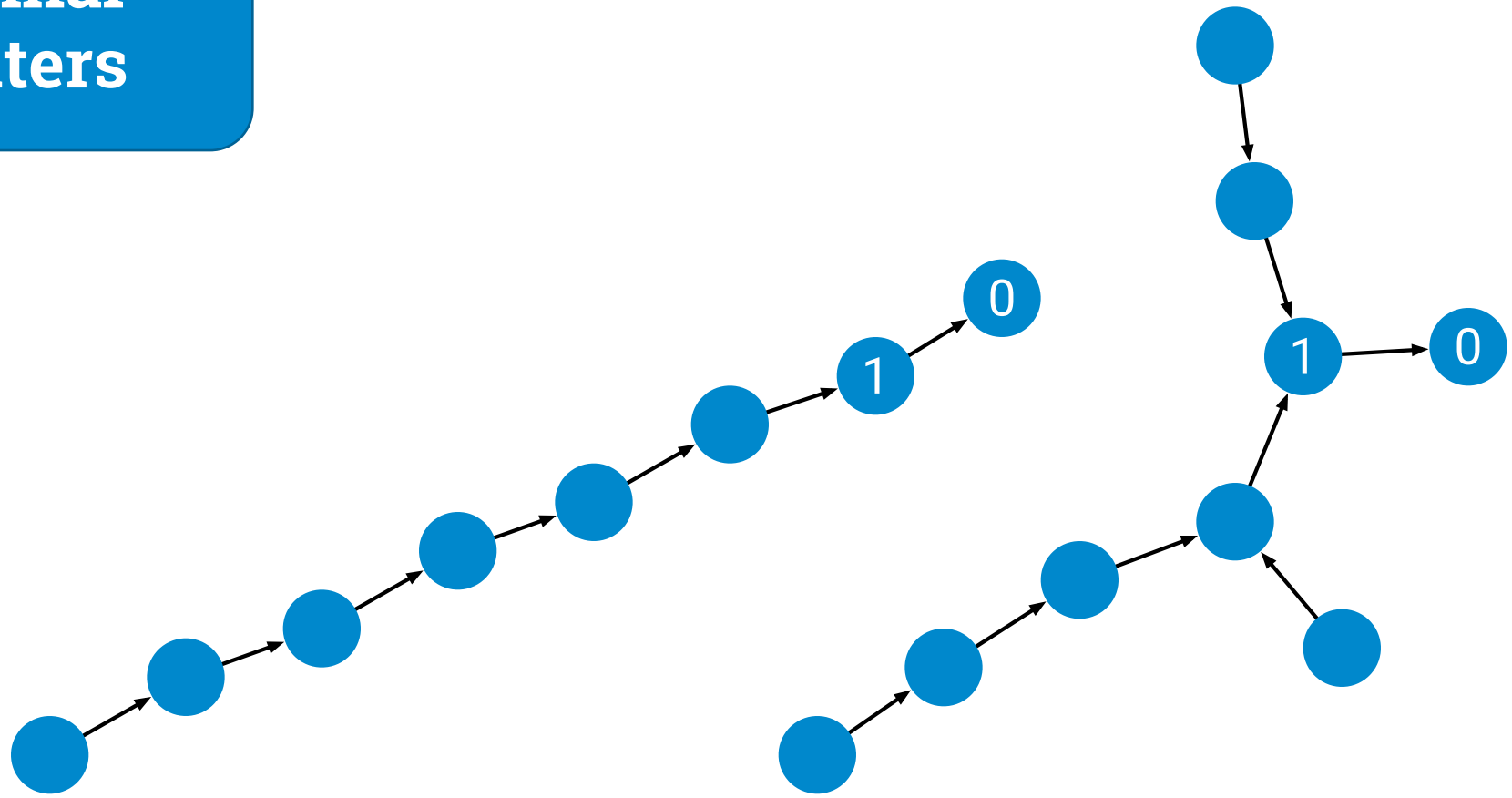


Special case of prefix sum

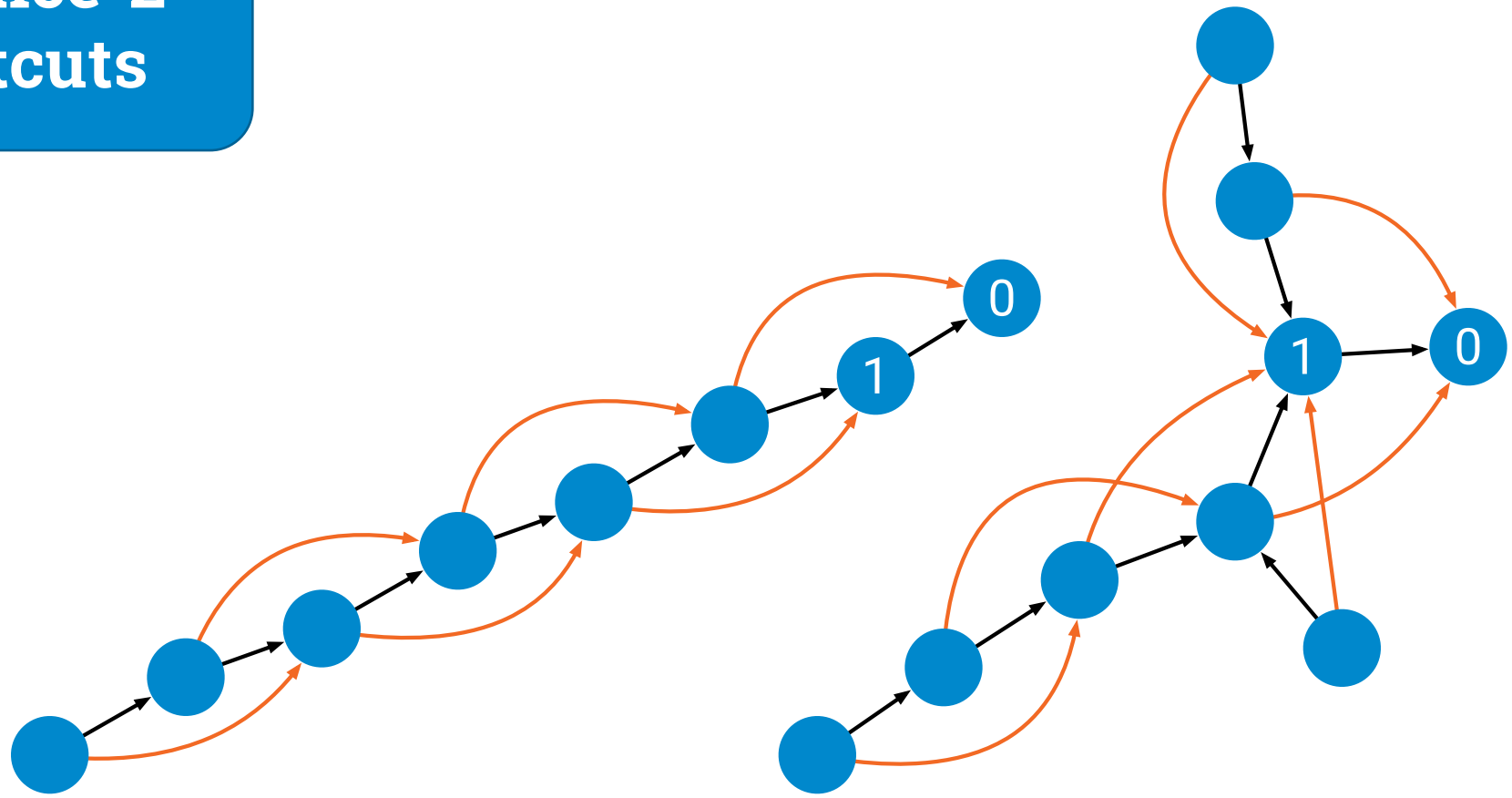
Pointer jumping

- Simple and efficient technique for handling “linked” data
- Basic idea:
  - $p[x]$ = successor of element x
  - ***in parallel:*** set $q[x] = p[p[x]]$ for all x
  - now we have shortcuts: **“1 hop of q” = “2 hops of p”**
  - ***repeat:*** shortcuts of length 2, 4, 8, …
- Examples:
  - “how far am I from the end of a linked list?”
  - “how far am I from the root of a tree?”

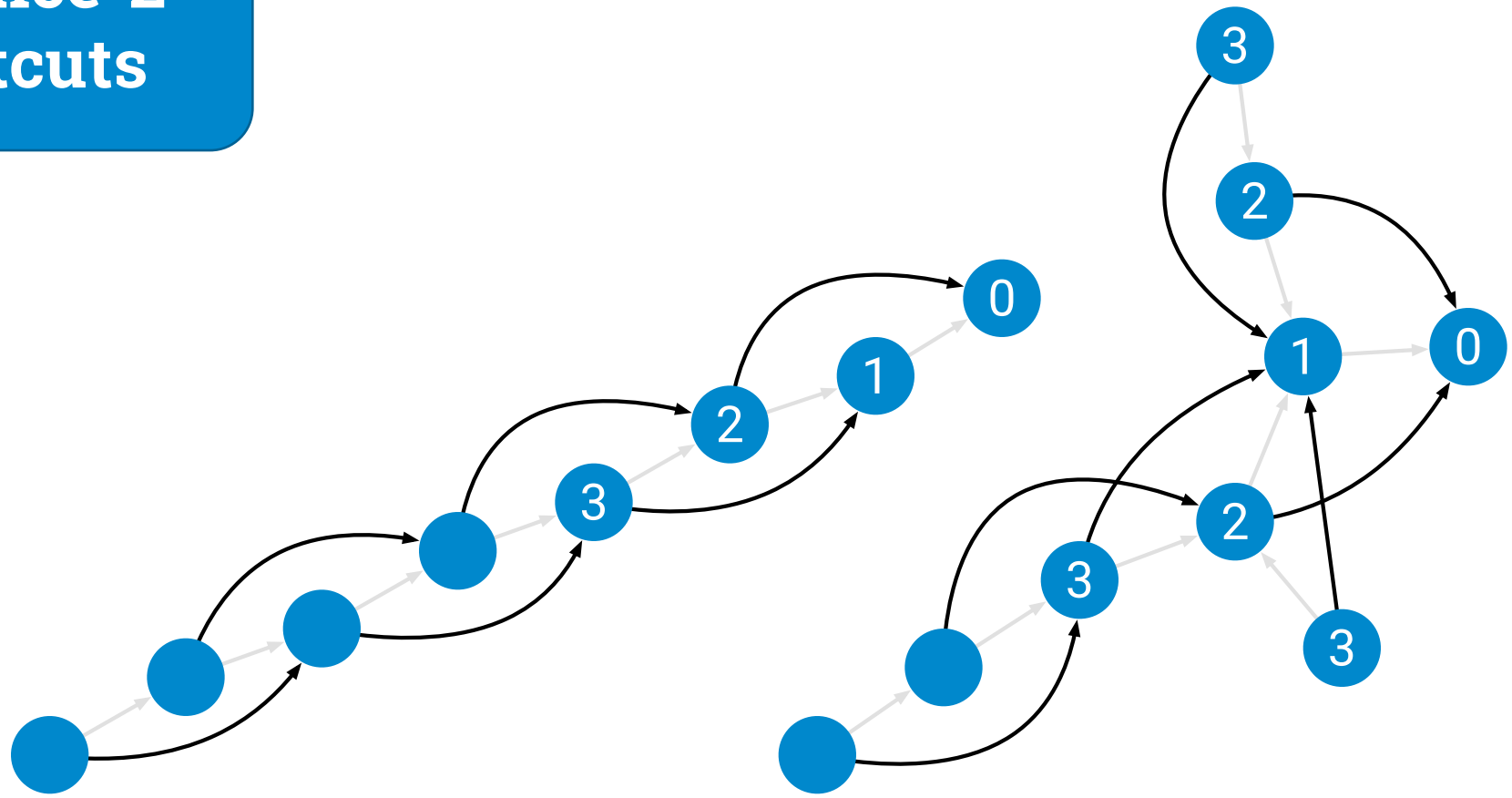
Original pointers



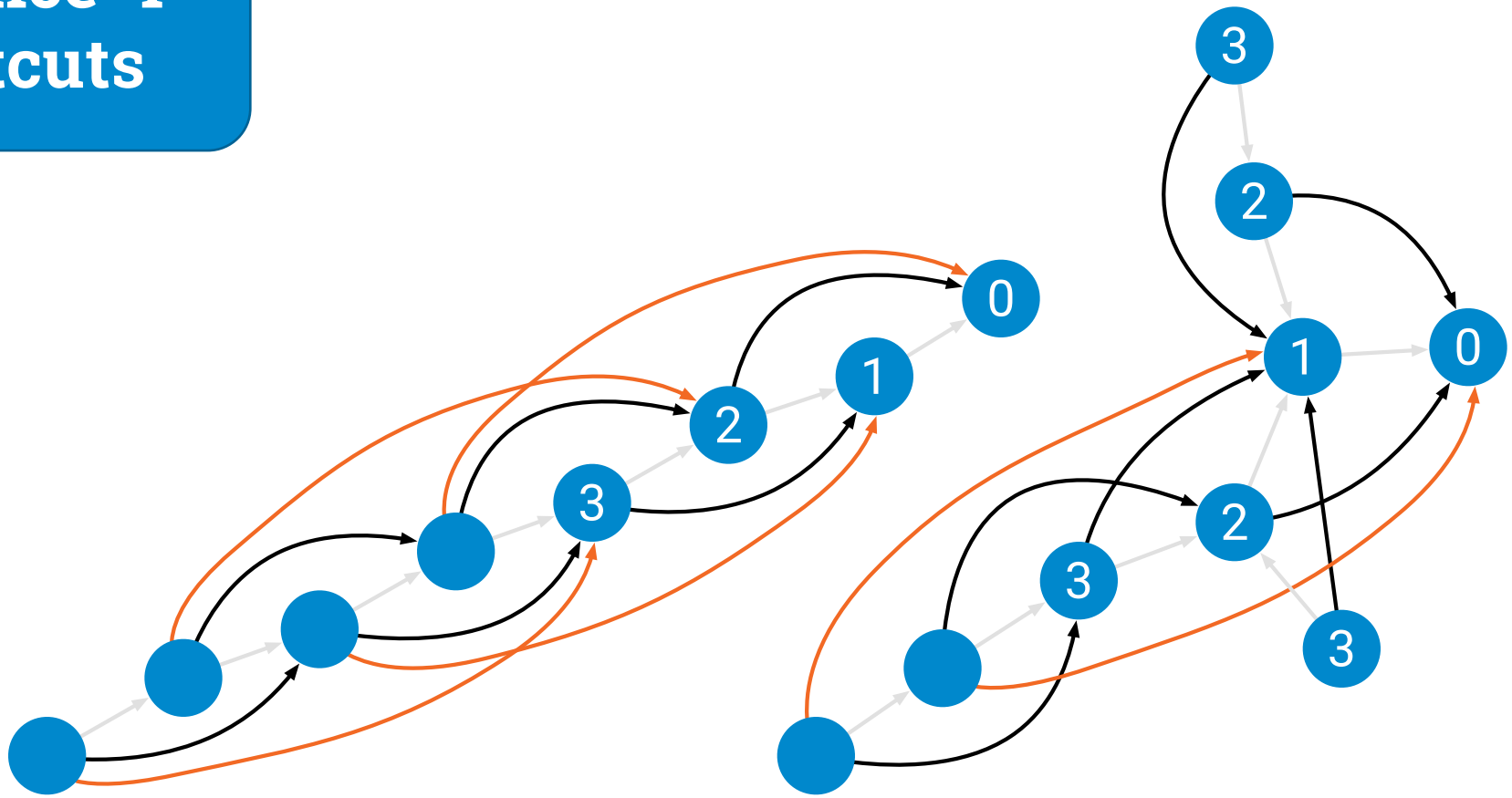
Distance-2 shortcuts



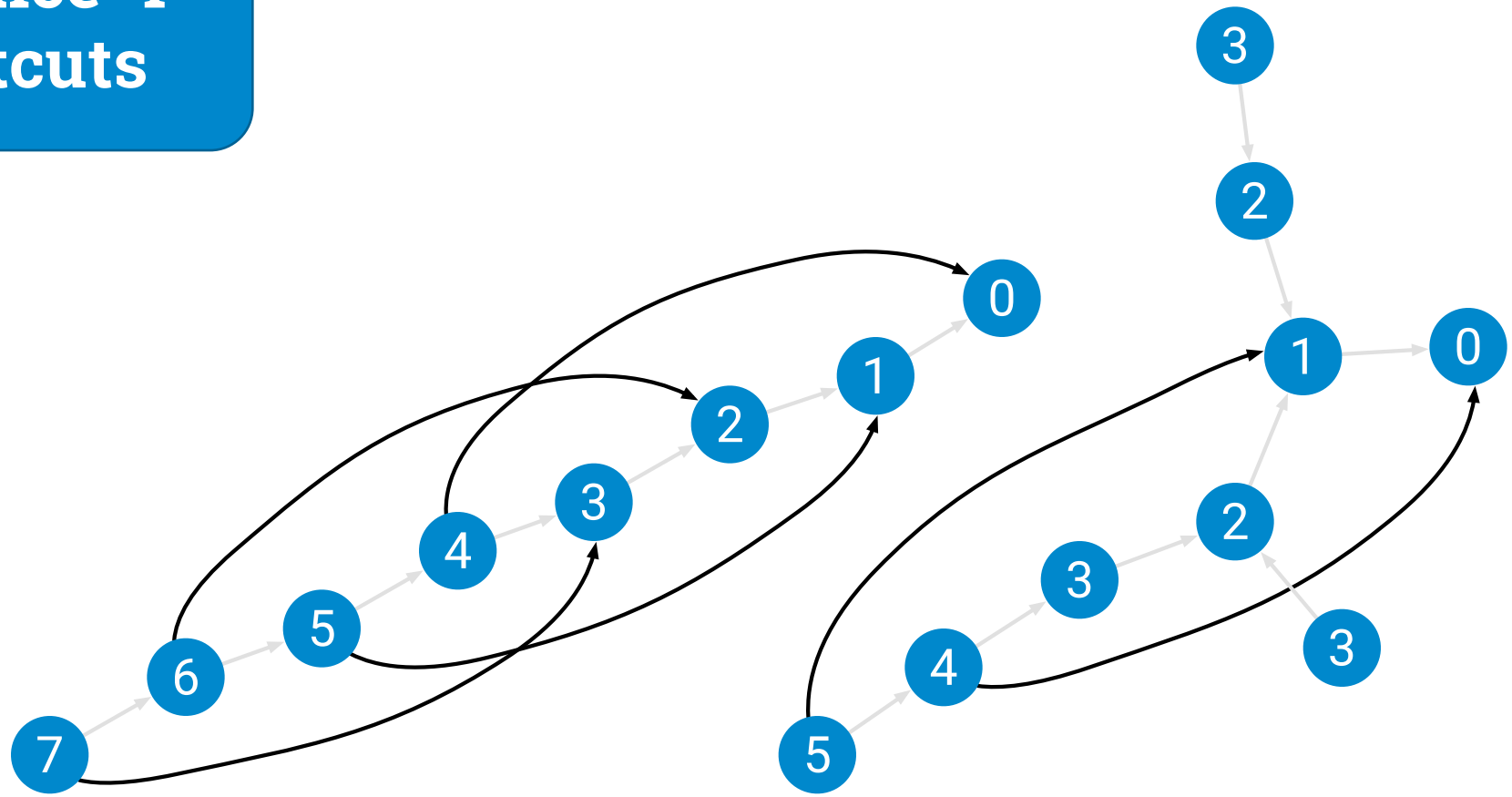
Distance-2 shortcuts



Distance-4 shortcuts



Distance-4 shortcuts



Future

What next?

What is happening to hardware

- ***Wider vector units***
  - Intel CPUs with **AVX-512** already available
- ***GPU-like auxiliary processors***
  - Google's "Tensor Processing Unit":
    special hardware for **matrix multiplications**
- ***Low-precision floating point numbers***
  - NVIDIA's "Tensor cores":
    4 × 4 matrix multiplication of **16-bit floats**

What is happening to hardware

- ***Transactional memory***
  - you can use memory a bit like transactional databases:
    - **begin transaction**
    - read and write memory (without any coordination)
    - try to **commit**
    - **rollback** if conflicts
  - some hardware support available in recent Intel CPUs

What is happening to hardware

- ***Virtualization***
  - many virtual machines running on one physical computer
  - yet another abstraction layer between your program and hardware
- ***Cloud computing***
  - somebody else owns the computers, you buy CPU time
  - launch Linux virtual machines using a web interface,
    you will have root access, pay based on the number of hours
  - easy to experiment with e.g. massively multicore computers,
    high-end GPUs

What next?

- ***Write code, lots of code***
  - learn about parallel computing capabilities in your programming tools
- **Practical path:**
  - computer architecture, computer hardware, compilers, programming languages, cloud computing, distributed computing, computer networks, internet protocols, mobile computing …
- **Theory path:**
  - algorithm design & analysis, computational complexity, parallel algorithms, distributed algorithms, concurrency theory, formal verification & synthesis …

Questions?

That's all, many thanks!

- This is the last week of the course
  - two exercise sessions as usual
  - deadline for all exercises: **26 May 2019**
- Please remember to give ***course feedback!***
  - you should have received an email with a link to the feedback form
  - deadline **6 June 2019**, but you can do it already now!
  - **+1 point** for everyone who gives course feedback
  - extremely important for the development of the course, **please do it** even if you do not need extra points!