CDCS
CENTER FOR DATA AND COMPUTING
IN NATURAL SCIENCES

Parallel Computing I

Parallelism On CPUs

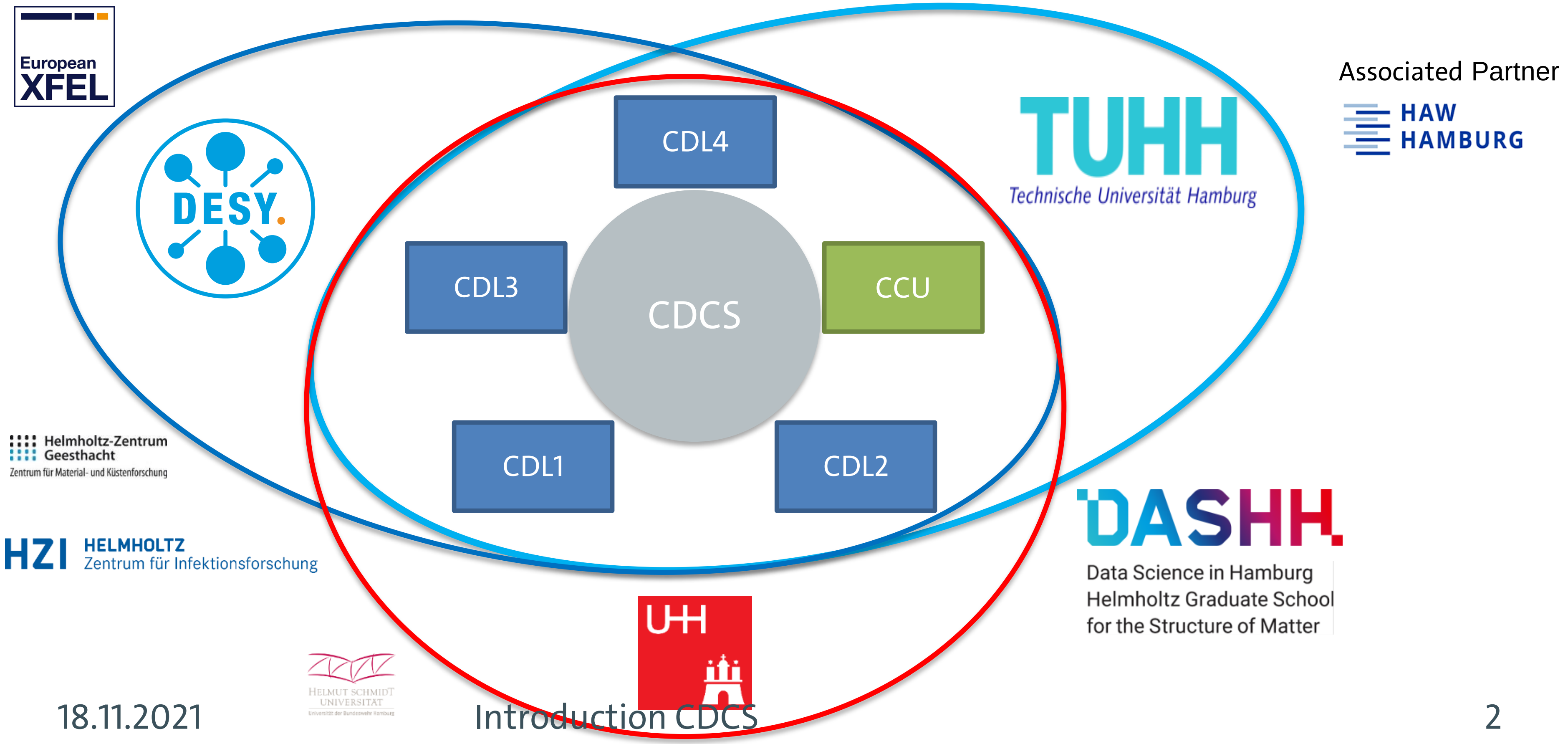
Annett Ungethüm, 20.01.2022

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

DESY.

TUHH
Technische Universität Hamburg-Harburg

CDCS Hamburg-X Project (BWFGB)


European XFEL
DESY.
CDL4
CDL3
CDCS
CCU
CDL1
CDL2
TUHH
Technische Universität Hamburg
Associated Partner
HAW HAMBURG
Helmholtz-Zentrum Geesthacht
Zentrum für Material- und Küstenforschung
HZI HELMHOLTZ
Zentrum für Infektionsforschung
DASHH
Data Science in Hamburg
Helmholtz Graduate School
for the Structure of Matter
UHH
Helmut Schmidt Universität
Universität der Bundeswehr Hamburg

CDCS Structure

UHH Project Coordinator

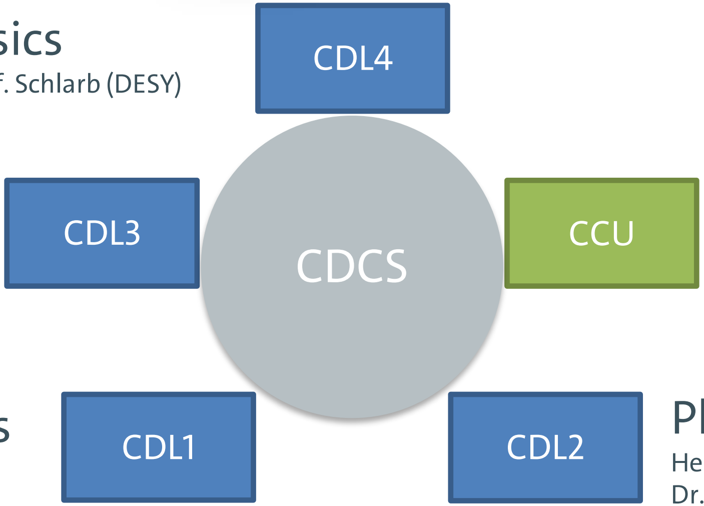
Prof. Dr. Matthias Rarey
UHH Computer Science
Spokesperson CDCS

DESY Project Coordinator

Prof. Dr. Nina Rohringer
DESY / UHH Physics

TUHH Project Coordinator

Prof. Dr. Sabine Le Borne
TUHH Mathematics



Accelerator Physics
Head: Prof. Fey (TUHH) / Prof. Schlarb (DESY)

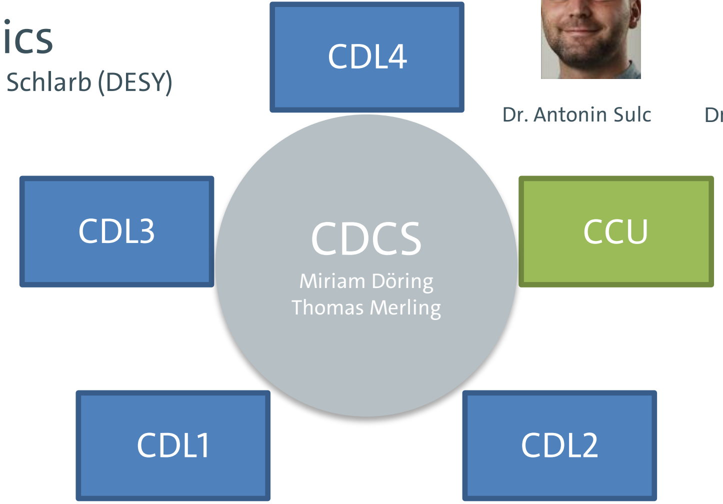
Systems Biology
Head: Prof. Grünewald (UHH)/
Prof. Baumbach (UHH)

Computational Core Unit
Head: Prof. Rarey (UHH)/ Prof. Knopp (TUHH)

Astro & Particle Physics
Head: Prof. Schleper (DESY)/
Dr. Gaede (UHH)

Photon Science
Head: Dr. Barty (DESY)/
Dr. White (DESY)

CDCS and CDLs in Detail



Accelerator Physics

Head: Prof. Fey (TUHH) / Prof. Schlarb (DESY)

Dr. Antonin Sulc

Dr. Ahmad Al-Zoubi

Systems Biology

Head: Prof. Grünewald (UHH)/
Prof. Baumbach (UHH)

Dr. Karen M.-Cantos

Dr. Khalique Newaz

Computational Core Unit

Head: Prof. Rarey (UHH)/ Prof. Knopp (TUHH)

Dr. Marie Tolkiehn
Scientific-administrative
Management

Dr. Florian Griese

Dr. Annett Ungethüm

Astro & Particle Physics

Head: Prof. Schleper (DESY)/
Prof. Gaede (UHH)

Dr. Patrick Connor

Dr. Janis Kummer

Dr. Lennart Rustige

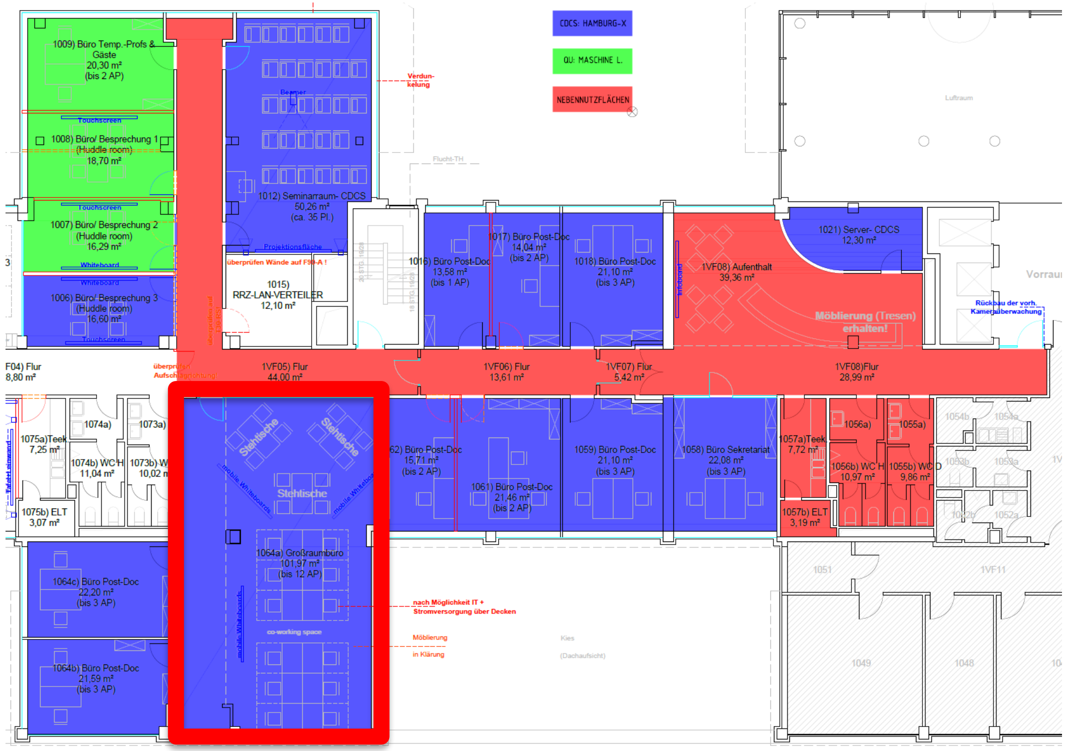
Photon Science

Head: Dr. Barty (DESY)/
Dr. White (DESY)

Dr. Mads Jakobsen

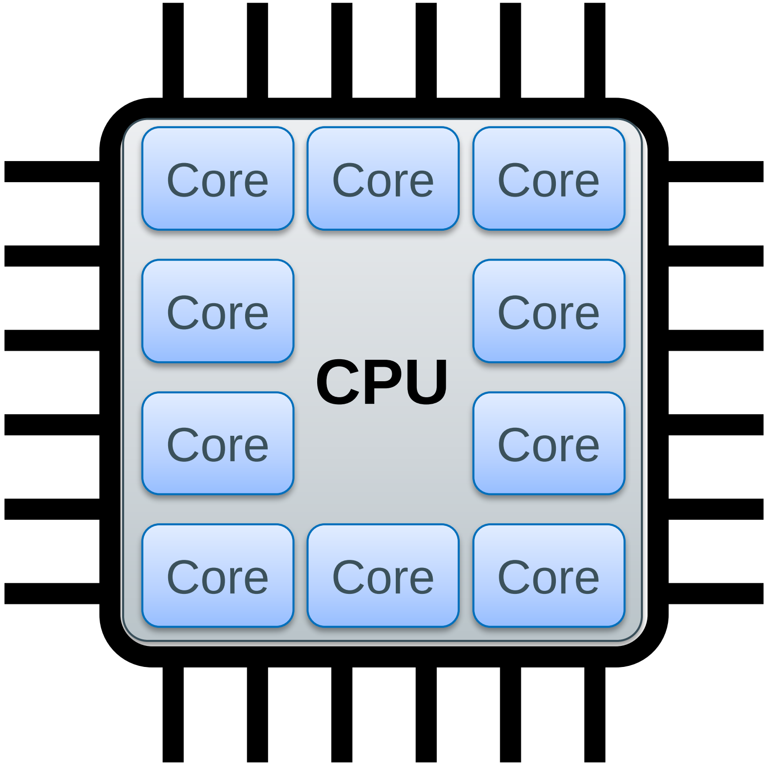
N.N

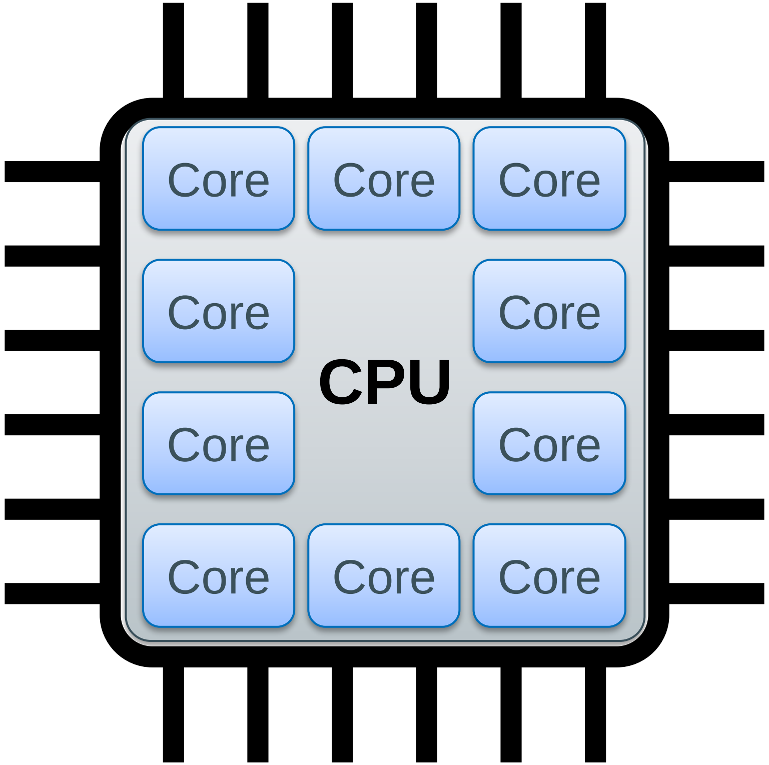
The CDCS Office Space


CDCS: HAMBURG-X
QU: MASCHINE L.
NEBENNUTZFLÄCHEN
1009) Büro Temp.-Profs & Gäste 20,30 m² (bis 2 AP)
1008) Büro/ Besprechung 1 (Huddle room) 18,70 m²
1007) Büro/ Besprechung 2 (Huddle room) 16,29 m²
1006) Büro/ Besprechung 3 (Huddle room) 16,60 m²
1012) Seminarraum- CDCS 50,26 m² (ca. 35 Pl.)
1015) RRZ-LAN-VERTEILER 12,10 m²
1016) Büro Post-Doc 13,58 m² (bis 1 AP)
1017) Büro Post-Doc 14,04 m² (bis 2 AP)
1018) Büro Post-Doc 21,10 m² (bis 3 AP)
1VF08) Aufenthalt 39,36 m²
1021) Server- CDCS 12,30 m²
Möblierung (Tresen) erhalten!
1VF05) Flur 44,00 m²
1VF06) Flur 13,61 m²
1VF07) Flur 5,42 m²
1VF08)Flur 28,99 m²
1064a) Großraumbüro 101,97 m² (bis 12 AP)
Stehtische
co-working space
1064c) Büro Post-Doc 22,20 m² (bis 3 AP)
1064b) Büro Post-Doc 21,59 m² (bis 3 AP)
1061) Büro Post-Doc 21,46 m² (bis 2 AP)
1059) Büro Post-Doc 21,10 m² (bis 3 AP)
1058) Büro Sekretariat 22,08 m² (bis 3 AP)
nach Möglichkeit IT + Stromversorgung über Decken
Möblierung in Klärung

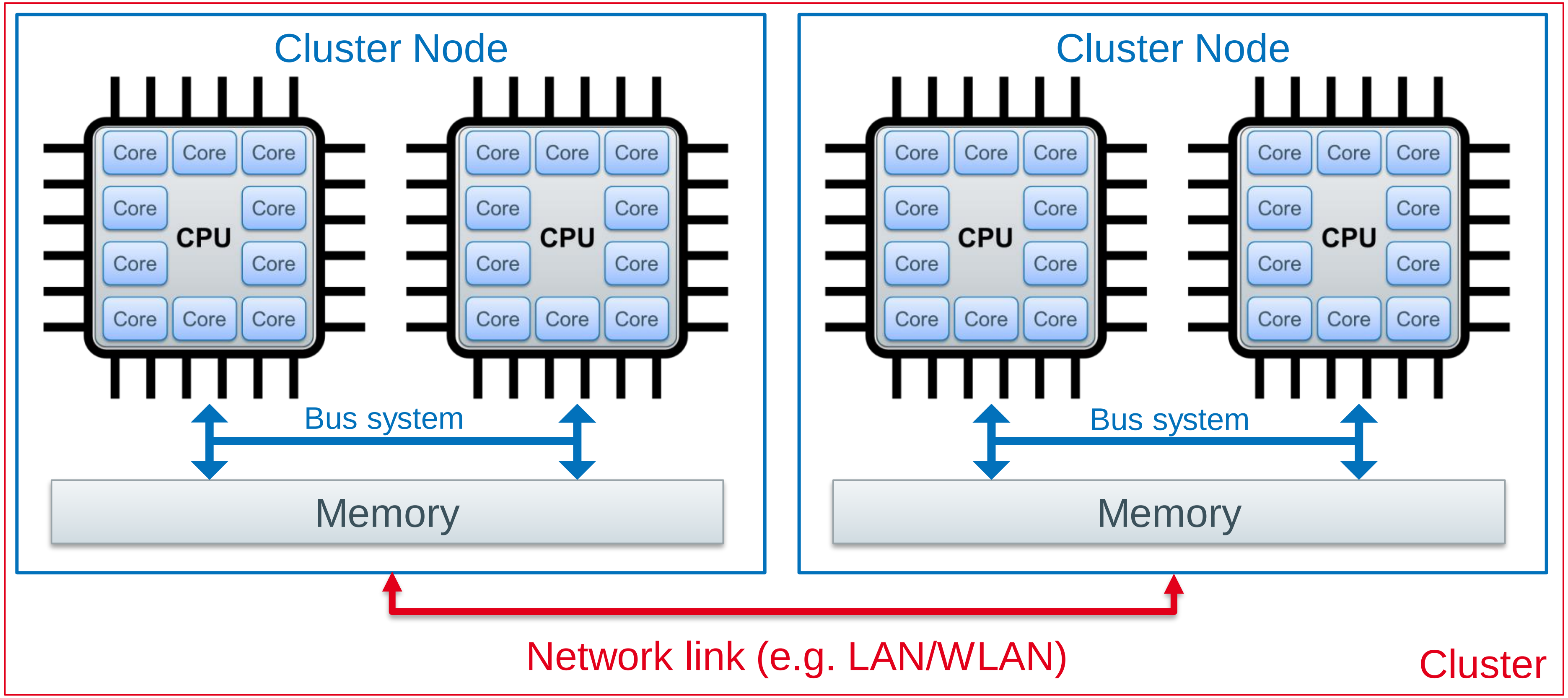
As a DASHH student you can get a transponder to the CDCS hot desk office space (room 1064)
Ask our secretary Miriam Döring:
miriam.doering@uni-hamburg.de

Computer Systems

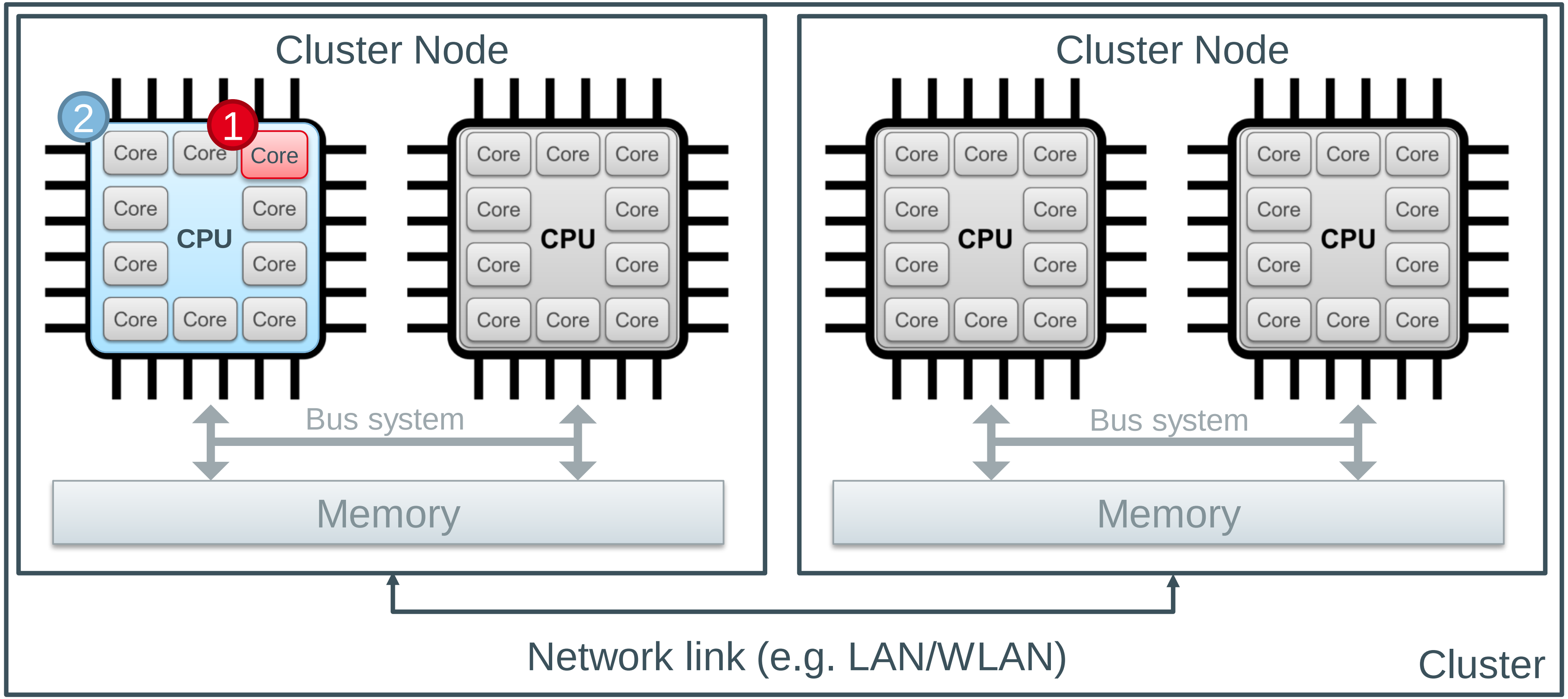




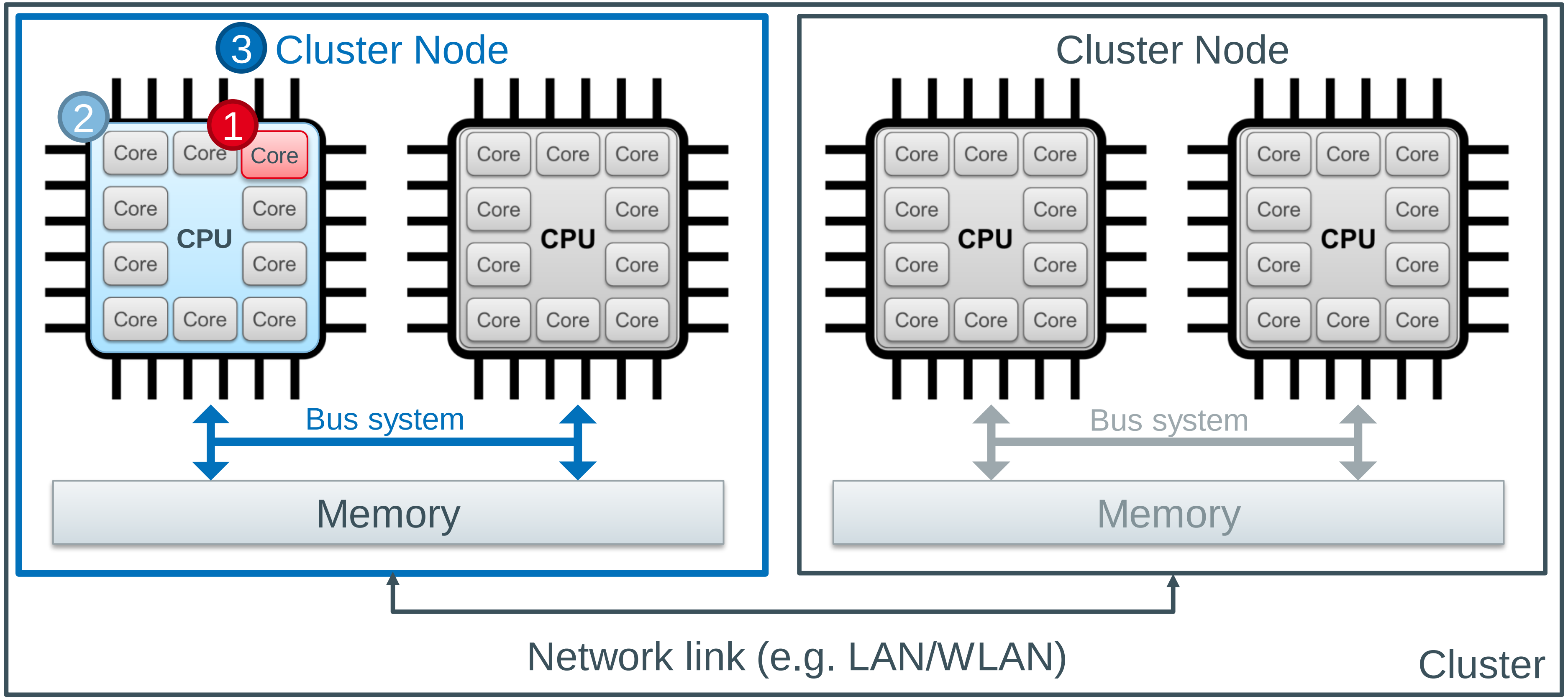
Computer Systems



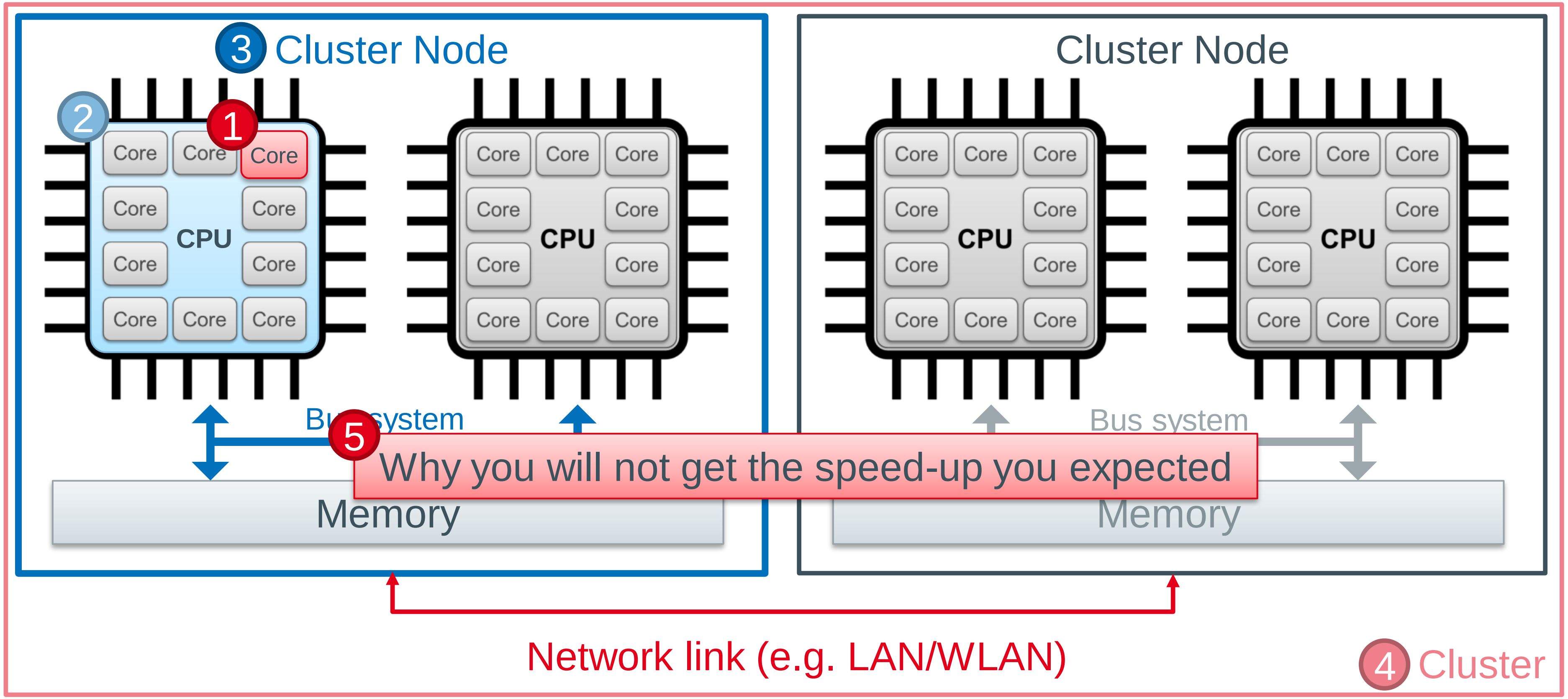
Roadmap


Cluster Node
2
1
Core
CPU
Bus system
Memory
Cluster Node
Core
CPU
Bus system
Memory
Network link (e.g. LAN/WLAN)
Cluster

Roadmap


3 Cluster Node
2
1
Core
CPU
Bus system
Memory
Cluster Node
Core
CPU
Bus system
Memory
Network link (e.g. LAN/WLAN)
Cluster

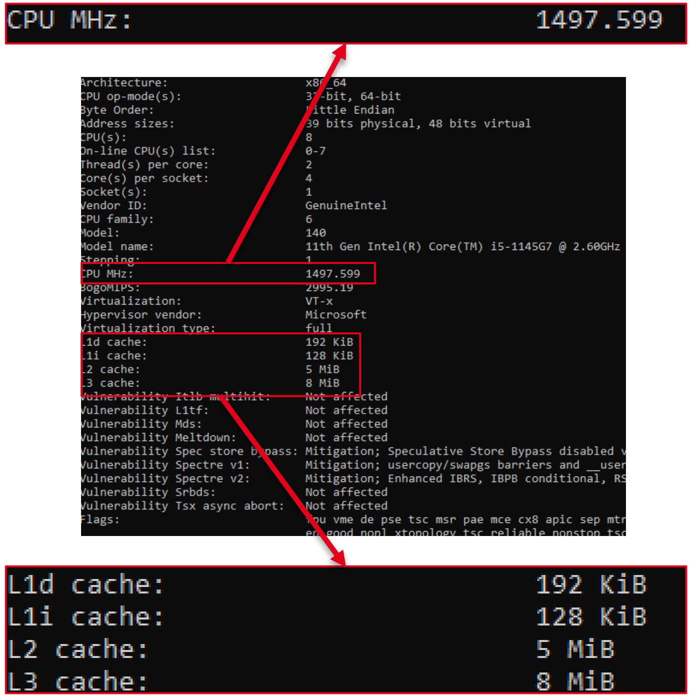
Roadmap


3 Cluster Node
2
1
Core
CPU
Cluster Node
Bus system
5 Why you will not get the speed-up you expected
Memory
Bus system
Memory
Network link (e.g. LAN/WLAN)
4 Cluster

Parallelism On A Single Core

CPU information

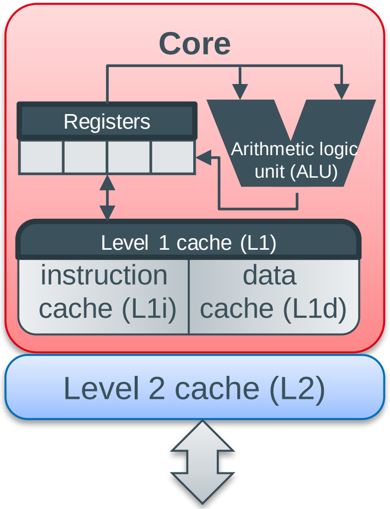
Linux: lscpu



```
CPU MHz:                         1497.599
```

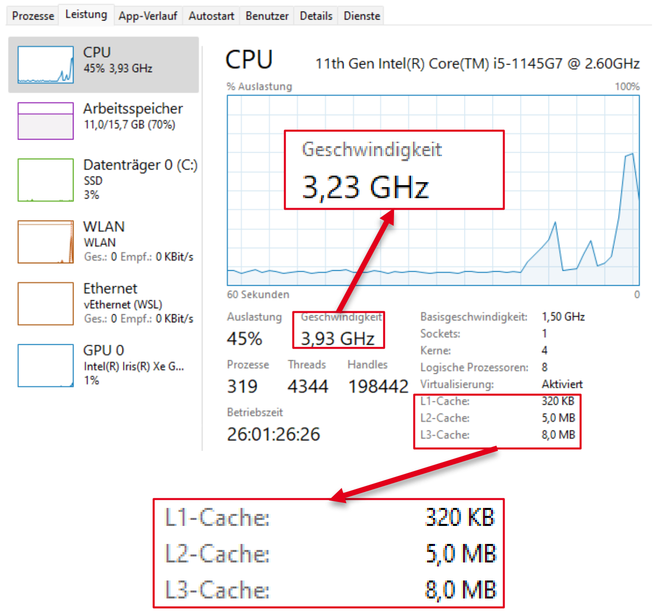
```
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   39 bits physical, 48 bits virtual
CPU(s):                          8
On-line CPU(s) list:             0-7
Thread(s) per core:              2
Core(s) per socket:              4
Socket(s):                       1
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           140
Model name:                      11th Gen Intel(R) Core(TM) i5-1145G7 @ 2.60GHz
Stepping:                        1
CPU MHz:                         1497.599
BogoMIPS:                        2995.19
Virtualization:                  VT-x
Hypervisor vendor:               Microsoft
Virtualization type:             full
L1d cache:                       192 KiB
L1i cache:                       128 KiB
L2 cache:                        5 MiB
L3 cache:                        8 MiB
Vulnerability Itlb multihit:     Not affected
Vulnerability L1tf:              Not affected
Vulnerability Mds:               Not affected
Vulnerability Meltdown:          Not affected
Vulnerability Spec store bypass: Mitigation; Speculative Store Bypass disabled v
Vulnerability Spectre v1:        Mitigation; usercopy/swapgs barriers and __user
Vulnerability Spectre v2:        Mitigation; Enhanced IBRS, IBPB conditional, RS
Vulnerability Srbds:             Not affected
Vulnerability Tsx async abort:   Not affected
Flags:                           fpu vme de pse tsc msr pae mce cx8 apic sep mtr
                                 er good nopl xtopology tsc reliable nonstop tsc
```

```
L1d cache:                       192 KiB
L1i cache:                       128 KiB
L2 cache:                        5 MiB
L3 cache:                        8 MiB
```



Bus system/
Last Level Cache (LLC/L3)

Windows: Task manager (ctrl + alt +del → task manager → Performance tab)



CPU 45% 3,93 GHz
Arbeitsspeicher 11,0/15,7 GB (70%)
Datenträger 0 (C:) SSD 3%
WLAN WLAN Ges.: 0 Empf.: 0 KBit/s
Ethernet vEthernet (WSL) Ges.: 0 Empf.: 0 KBit/s
GPU 0 Intel(R) Iris(R) Xe G... 1%

CPU 11th Gen Intel(R) Core(TM) i5-1145G7 @ 2.60GHz

Geschwindigkeit
3,23 GHz

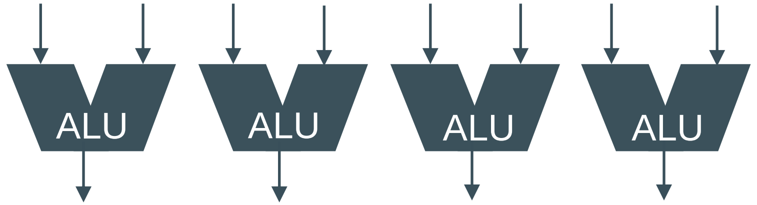
Auslastung	Geschwindigkeit	
45%	3,93 GHz	
Prozesse	**Threads**	**Handles**
319	4344	198442
Betriebszeit		
26:01:26:26		

Basisgeschwindigkeit:	1,50 GHz
Sockets:	1
Kerne:	4
Logische Prozessoren:	8
Virtualisierung:	Aktiviert
L1-Cache:	320 KB
L2-Cache:	5,0 MB
L3-Cache:	8,0 MB

L1-Cache:	320 KB
L2-Cache:	5,0 MB
L3-Cache:	8,0 MB

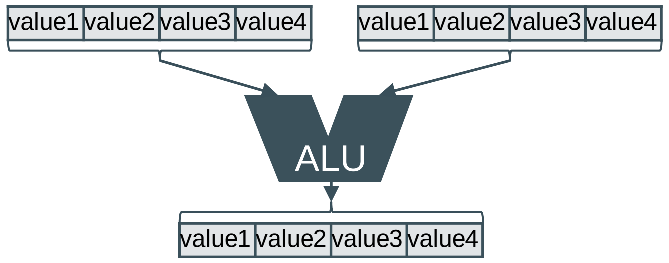
Parallelism On A Single Core

Instruction Parallelism



- Each ALU processes an execution pipeline
- Usually unnoticed by developers and users
- Ever wondered why some CPUs are faster than others despite having the same frequency and number of cores? → This is one of the reasons

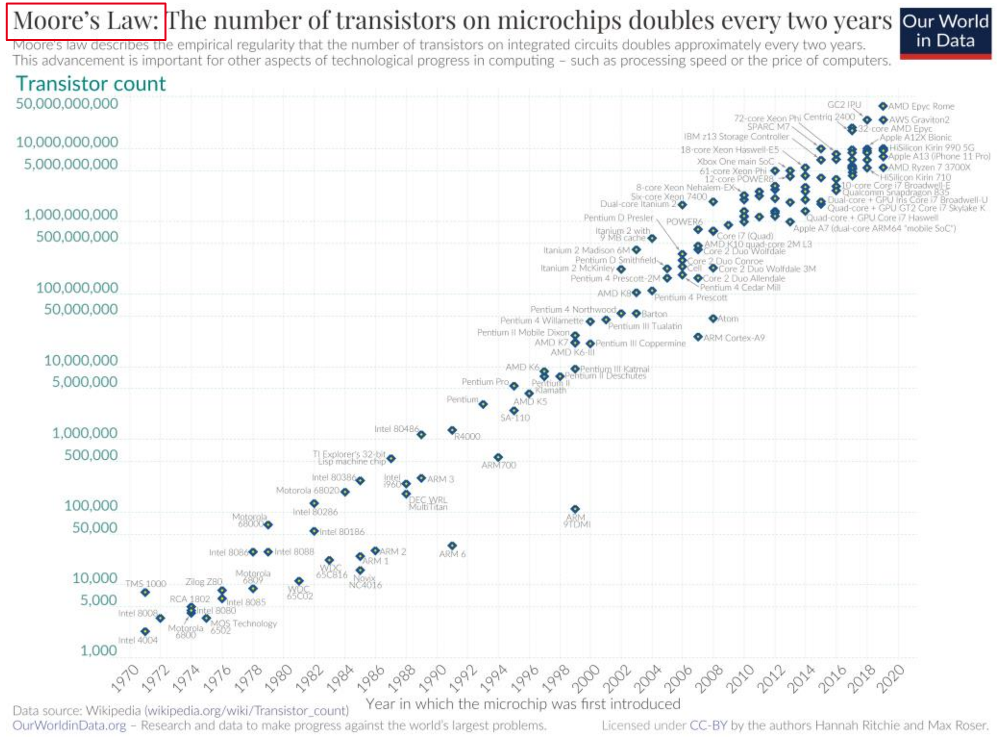
Data Parallelism



- Multiple values are processed at once by one ALU using one instruction
- Data is stored in vector registers, which can hold more than one value
- Not done automatically → Action by developer or compiler needed

Why We Have Data Parallelism On a Single Core?

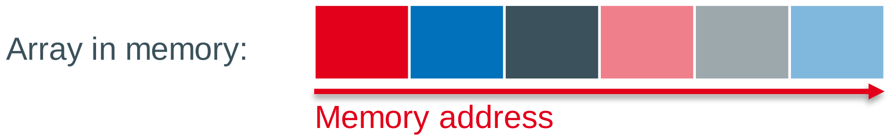
CDCS
CENTER FOR DATA AND COMPUTING
IN NATURAL SCIENCES

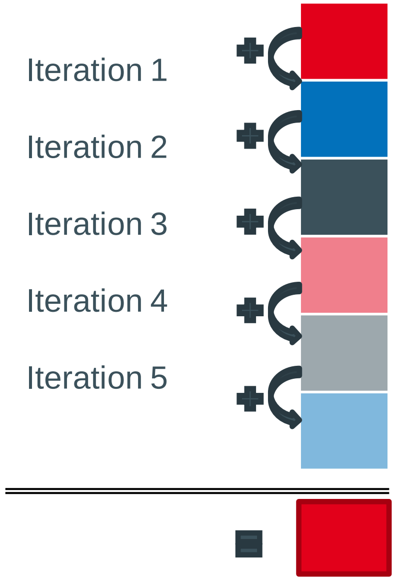


Number of Transistors per Core double every two years
- → But we cannot use all of them at the same time because of thermal constraints
- → Some transistors are switched off („Dark Silicon“[1])
- → Another scalar addition block doesn‘t make sense
- → „Dark“ parts of the chip are used for specialized instructions, e.g. for processing vectors

[1]H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam and D. Burger, "Dark Silicon and the End of Multicore Scaling," in IEEE Micro, vol. 32, no. 3, pp. 122-134, May-June 2012, doi: 10.1109/MM.2012.17.

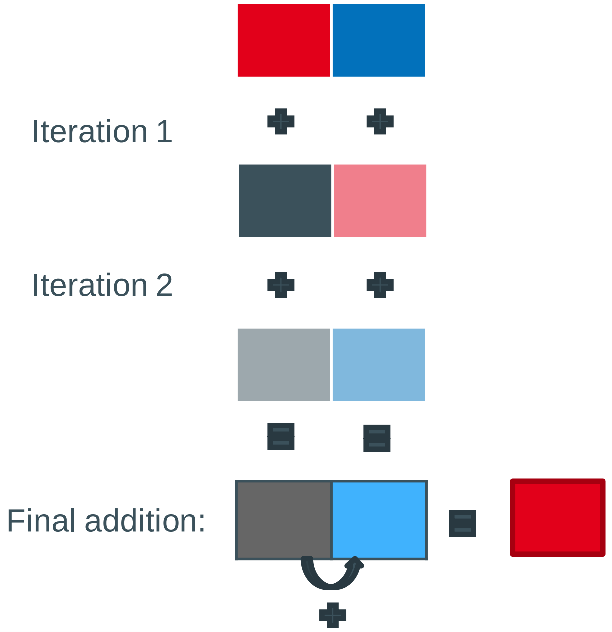
Vectorized Processing





Vectorization

Also called **S**ingle **I**nstruction **M**ultiple **D**ata (SIMD)



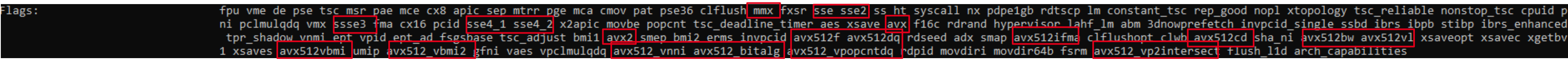
Scalar: 5 instructions
Vectorized: 3 instructions
→**High chances for performance gain**

Can My Potato CPU Do This?

YES!

CDCS
CENTER FOR DATA AND COMPUTING
IN NATURAL SCIENCES

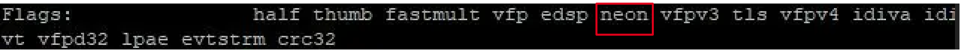
lscpu on an Intel or AMD CPU:



```
Flags:                      fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ss ht syscall nx pdpe1gb rdtscp lm constant_tsc rep_good nopl xtopology tsc_reliable nonstop_tsc cpuid p
                            ni pclmulqdq vmx ssse3 fma cx16 pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand hypervisor lahf_lm abm 3dnowprefetch invpcid_single ssbd ibrs ibpb stibp ibrs_enhanced
                             tpr_shadow vnmi ept vpid ept_ad fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid avx512f avx512dq rdseed adx smap avx512ifma clflushopt clwb avx512cd sha_ni avx512bw avx512vl xsaveopt xsavec xgetbv
                            1 xsaves avx512vbmi umip avx512_vbmi2 gfni vaes vpclmulqdq avx512_vnni avx512_bitalg avx512_vpopcntdq rdpid movdiri movdir64b fsrm avx512_vp2intersect flush_l1d arch_capabilities
```

mmx:	Extension from 1997
	64-bit registers → 2 x 32 float, 2 x 32 bit int, 4x 16 bit int, 8 x 8 bit int
sse*:	Extension from 1999
	128 bit registers → 2 x 64 bit integer, 2 x double, 4 x float, 4 x 32 bit integer,…
avx(2):	256 bit registers → 4 x 64 bit integer, 4 x double,… 32 x 8 bit integer
avx512*:	Different extensions with different functions
	512 bit registers → 8 x 64 bit registers…64 x 8 bit integer

lscpu on an Arm CPU (RaspberryPi, most Android phones,…)



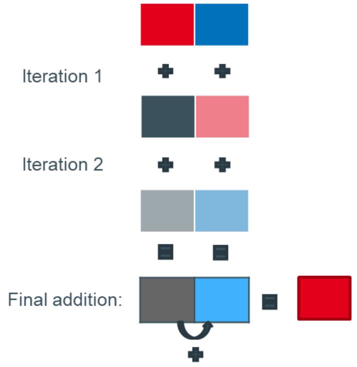
```
Flags:                half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idi
vt vfpd32 lpae evtstrm crc32
```

Neon(v2):	128 bit registers → 2 x 64 bit integer, 2 x double,…
SVE:	In newer Arm CPUs (Armv8), register size between 128 bit and 2048 bit

SIMD with Julia

Julia tries to apply SIMD automatically

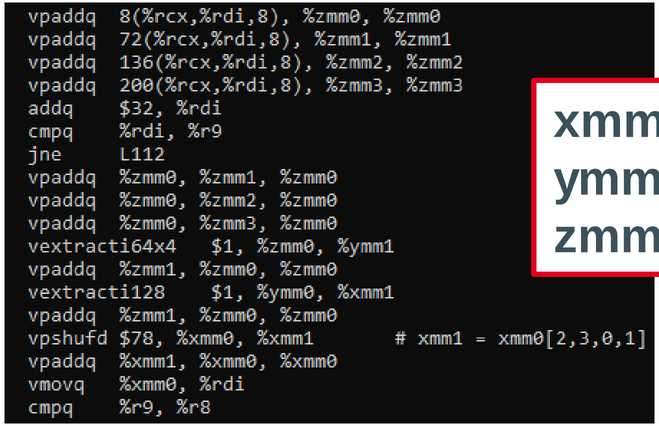
@code_native test(rand(Int64 , 100000)) ;



```
function test(myarray::Vector)
  res = 0
  for i in myarray
      res+=i
  end

  println(res)
end
```

Definition as vector required for SIMD to work



```
vpaddq  8(%rcx,%rdi,8), %zmm0, %zmm0
vpaddq  72(%rcx,%rdi,8), %zmm1, %zmm1
vpaddq  136(%rcx,%rdi,8), %zmm2, %zmm2
vpaddq  200(%rcx,%rdi,8), %zmm3, %zmm3
addq    $32, %rdi
cmpq    %rdi, %r9
jne     L112
vpaddq  %zmm0, %zmm1, %zmm0
vpaddq  %zmm0, %zmm2, %zmm0
vpaddq  %zmm0, %zmm3, %zmm0
vextracti64x4   $1, %zmm0, %ymm1
vpaddq  %zmm1, %zmm0, %zmm0
vextracti128    $1, %ymm0, %xmm1
vpaddq  %zmm1, %zmm0, %zmm0
vpshufd $78, %xmm0, %xmm1         # xmm1 = xmm0[2,3,0,1]
vpaddq  %xmm1, %xmm0, %xmm0
vmovq   %xmm0, %rdi
cmpq    %r9, %r8
```

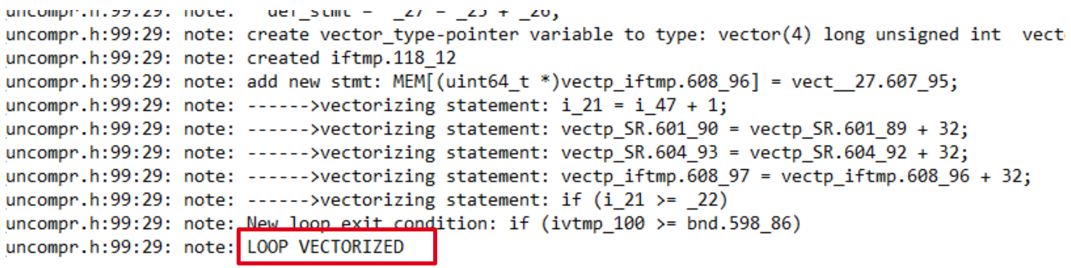
xmm 128 bit
ymm 256 bit
zmm 512 bit

Still not working? Consider explicit vectorization in Julia:
https://docs.julialang.org/en/v1/base/simd-types/

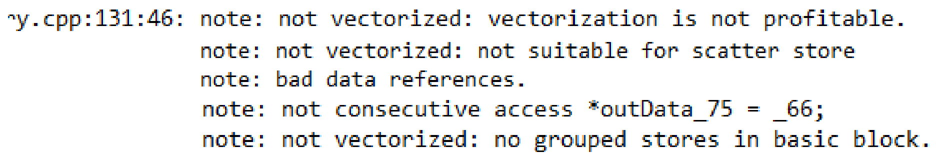
SIMD With C/C++

g++ tries to apply vectorization autoatically with -O2 and -O3

succesful compiler log


```
uncompr.h:99:29: note:   def_stmt = _27 = _25 + _26;
uncompr.h:99:29: note: create vector_type-pointer variable to type: vector(4) long unsigned int  vect
uncompr.h:99:29: note: created iftmp.118_12
uncompr.h:99:29: note: add new stmt: MEM[(uint64_t *)vectp_iftmp.608_96] = vect__27.607_95;
uncompr.h:99:29: note: ------>vectorizing statement: i_21 = i_47 + 1;
uncompr.h:99:29: note: ------>vectorizing statement: vectp_SR.601_90 = vectp_SR.601_89 + 32;
uncompr.h:99:29: note: ------>vectorizing statement: vectp_SR.604_93 = vectp_SR.604_92 + 32;
uncompr.h:99:29: note: ------>vectorizing statement: vectp_iftmp.608_97 = vectp_iftmp.608_96 + 32;
uncompr.h:99:29: note: ------>vectorizing statement: if (i_21 >= _22)
uncompr.h:99:29: note: New loop exit condition: if (ivtmp_100 >= bnd.598_86)
uncompr.h:99:29: note: LOOP VECTORIZED
```

unsuccesful compiler log


```
ry.cpp:131:46: note: not vectorized: vectorization is not profitable.
               note: not vectorized: not suitable for scatter store
               note: bad data references.
               note: not consecutive access *outData_75 = _66;
               note: not vectorized: no grouped stores in basic block.
```

Reasons can be next to unidentifyable.
Personal favourite: size_t as iterator type doesn‘t work, uint32_t works

Use explicit vectorization

Explicit SIMD with C/C++ → Step 1

CDCS
CENTER FOR DATA AND COMPUTING
IN NATURAL SCIENCES

Step 1: Getting data into a SIMD register

C-Style (scalar)

```
int arr[4] = {5,6,7,8};
int const* b=arr;
int a = 3;
int c = *b; //sets c to 5
```

~~__m128i a = 3;~~
~~__m128i c = *b;~~

This won‘t work

Vectorized

Uses load/store intrinsics

```
__m128i a = _mm_set1_epi64x(3);SSE  __m128i c = _mm_load_si128((__m128i*) b);
uint64x2_t a = vdupq_n_u64 (3);   NEON  uint64x2_t c = vld1q_u64((uint64x2_t*) b);
```

SIMD instrinsic

SIMD data types

Vector register *a:*

3	3

Vector register *c:*

5	6

There‘s more on Intel!
*Load unaligned data: _mm_loadu_*, Store (unaligned) data: _mm_store(u)_*, Stream load/store (bypass cache): _mm_stream_**

Explicit SIMD with C/C++ → Step 2 & 3

*Example is intel only

__m128i	A vector type (SSE)
_mm_*	An SSE intrinsic
resultVec	An alias for a vector register

Step 2: Add all values element-wise in a loop

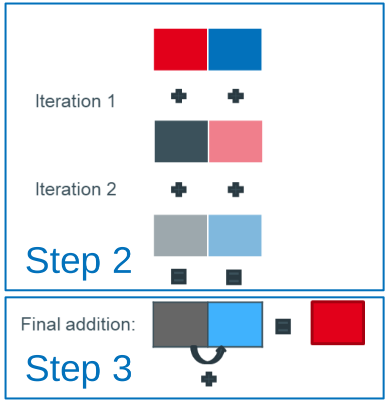
```
__m128i resultVec = _mm_setzero_si128( );
__m128i const * dataVecPtr = reinterpret_cast< __m128i * > b;

for( size_t i = 0; i < arraysize/2; ++i ) {
        resultVec = _mm_add_epi64( resultVec, _mm_load_si128( dataVecPtr+i ) );
}
```

The load intrinsic we just introduced

Step 3: Copy result vector to memory and aggregate ist elements

```
alignas(64) uint64_t resultArray[ 2 ];
_mm_store_si128( reinterpret_cast< __m128i * >( &resultArray ), resultVec );
int result = resultArray[ 0 ] + resultArray[ 1 ];
```



Explicit SIMD with C/C++ → Step 4

Step 4: Include the right headers and build it

`#include <mmintrin.h>` → Include file for SSE

g++ -O3 main.cpp –o myapp → Throws errors if g++ does not have your chosen SIMD extension in the defaults for your CPU

g++ -O3 –msse4.2 main.cpp –o myapp → Support for SSE

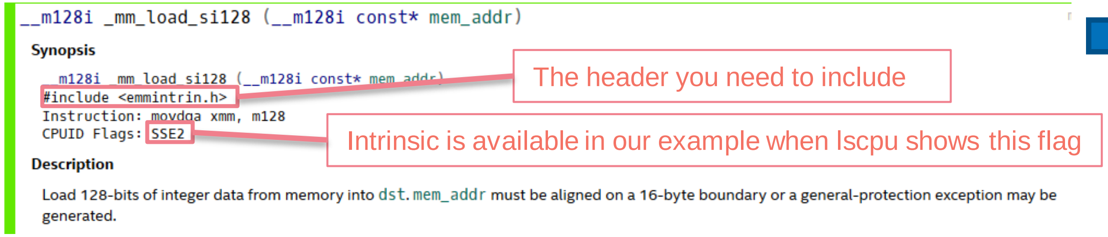
g++ -O3 –mavx512f main.cpp –o myapp → Support for AVX512 foundation

g++ -O3 -flax-vector-conversions main.cpp –o myapp

→ Support for Neon on most 64 bit Arm systems,
Use -mfpu=neon on 32 bit systems

Ressources for advanced C++ SIMD Programming

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



```
__m128i _mm_load_si128 (__m128i const* mem_addr)
```

Synopsis

```
__m128i _mm_load_si128 (__m128i const* mem_addr)
#include <emmintrin.h>
Instruction: movdqa xmm, m128
CPUID Flags: SSE2
```

The header you need to include

Intrinsic is available in our example when lscpu shows this flag

Description

Load 128-bits of integer data from memory into dst. mem_addr must be aligned on a 16-byte boundary or a general-protection exception may be generated.

→ Intel intrinsics guide (mmx, sse, avx(2), avx 512)

https://developer.arm.com/documentation/den0018/a/NEON-Intrinsics

→ Neon online documentation

https://developer.arm.com/documentation/ihi0073/h

→ Arm neon intrinsics reference

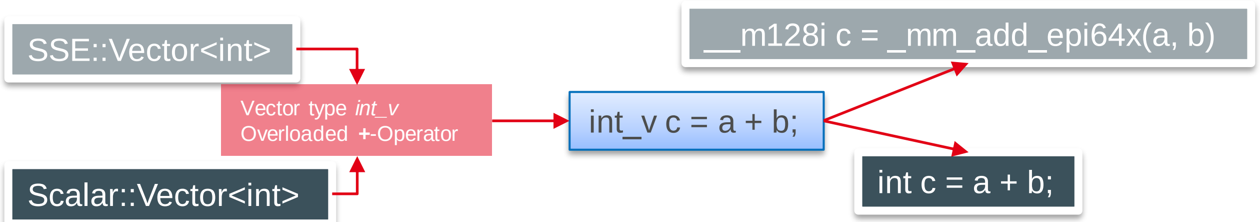
https://developer.arm.com/documentation/102476/0001

→ Arm SVE documentation

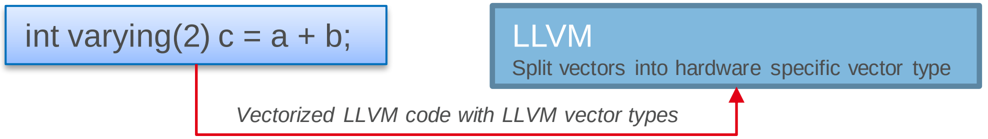
https://github.com/MorphStore/TVLLib

→ Library which abstracts the architecture
→ no architecture specific intrinsics needed

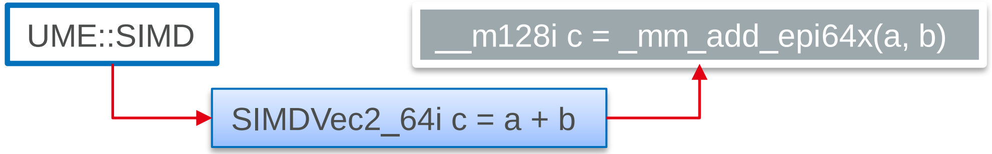
Vector Libraries



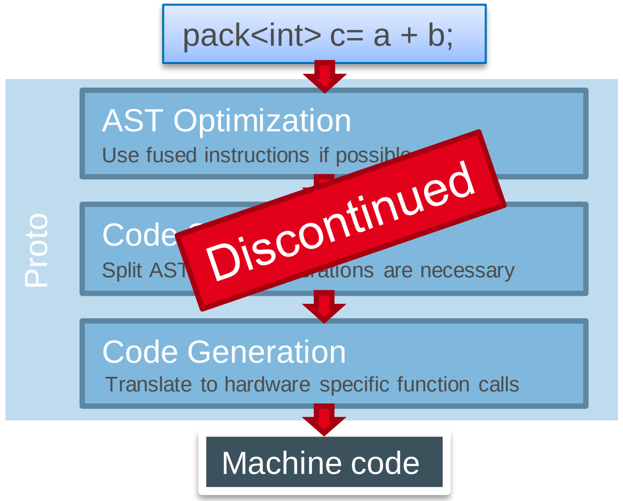
Matthias Kretz, Volker Lindenstruth: Vc: A C++ library for explicit vectorization. Softw., Pract. Exper. 42 (2012)


int varying(2) c = a + b;
LLVM
Split vectors into hardware specific vector type
Vectorized LLVM code with LLVM vector types

Roland Leißa, Immanuel Haffner, Sebastian Hack:
Sierra: a SIMD extension for C++. WPMVP 2014

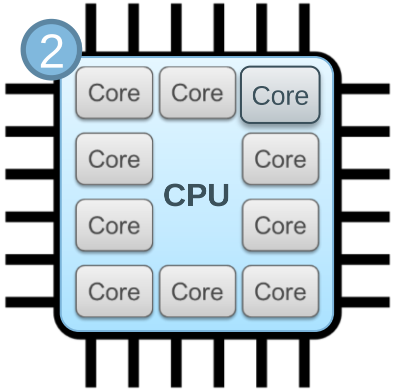


Przemyslaw Karpinski, John McDonald: A high-performance portable abstract interface for explicit SIMD vectorization. PMAM 2017



Pierre Estérie, Joel Falcou, Mathias Gaunard, Jean-Thierry Lapresté:
Boost.SIMD: generic programming for portable SIMDization. WPMVP 2014

Roadmap



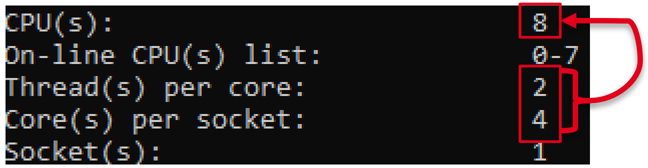
Core ≠ Core

Tasks are split into threads and distributed across the cores (=multithreading)

Core ≠ Core

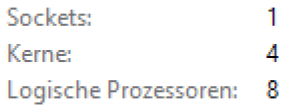
A physical core can (often) run more than one thread at the same time → logical cores through *hyperthreading*

lscpu:



```
CPU(s):                  8
On-line CPU(s) list:     0-7
Thread(s) per core:      2
Core(s) per socket:      4
Socket(s):               1
```

Windows Task Manager



Sockets:	1
Kerne:	4
Logische Prozessoren:	8

→ 4 physical cores à 2 threads = 8 logical cores

Hyperthreading

Advantages

- Threads on the same physical core use the same ressources
→ Fast exchange of data and fast ressource switching

- Lower probability of idle CPU time
→ Stalling cycles can be filled with computation by another thread

Disadvantages

- Threads on the same physical core use the same ressources
→ Threads have to share bandwidth and cache space

- Higher probability of contentions (both threads try to access the same ressource)

→ Useful when your threads exchange data frequently but do not share other ressources (e.g. they do not access the same array)
→ Not useful when all threads access the same ressources

Thread Pinning I

It is possible to enforce on which cores a thread is allowed to run

From the outside: numactl (might not be available on clusters because of security reasons)

```
numactl --physcpubind=0 myProgram
```

→ myProgram will run on core 0

```
cat /sys/devices/system/cpu/cpu0/topology/thread_siblings_list
```

→ Tells you which logical cores are on the same physical core as core 0
Can be used to disable hyperthreading

```
numactl --physcpubind=0,2 myProgram
```

→ myProgram can run on core 0 and 2

Thread Pinning II

From the inside: Depends on how you create your threads

→Search term: Thread affinity

Common approach: OpenMP

export GOMP_CPU_AFFINITY="0-2:2"   Binds the threads to core 0 and core 2

OR

export OMP_PLACES="{0}:2:2" OMP_PLACES overwrites GOMP_CPU_AFFINITY

➡ Works only if your implementation uses OpenMP

OpenMP - Loops

Simple way to parallelize C/C++ and Fortran

Equivalent to *export OMP_NUM_THREADS=8*
→ But can override OMP_NUM_THREADS

```
#include <omp.h>
int main(){
        omp_set_num_threads(8);

        #pragma omp parallel for
        for (int i=0; i<8; i++){
                //do something
        }
}
```

Requires linking with *–fopenmp*
→ Not linking will not throw errors but you will not get a parallel application

Parallelizes the following for-loop such that there is one thread per loop
- → If less than 8 cores are assigned, some threads run sequential insteadof parallel
- → Explicitly defining the number of threads is optional

OpenMP - Parallel Sections

CDCS
CENTER FOR DATA AND COMPUTING IN NATURAL SCIENCES

Define which pieces of code can run in parallel (in case it‘s not a loop)

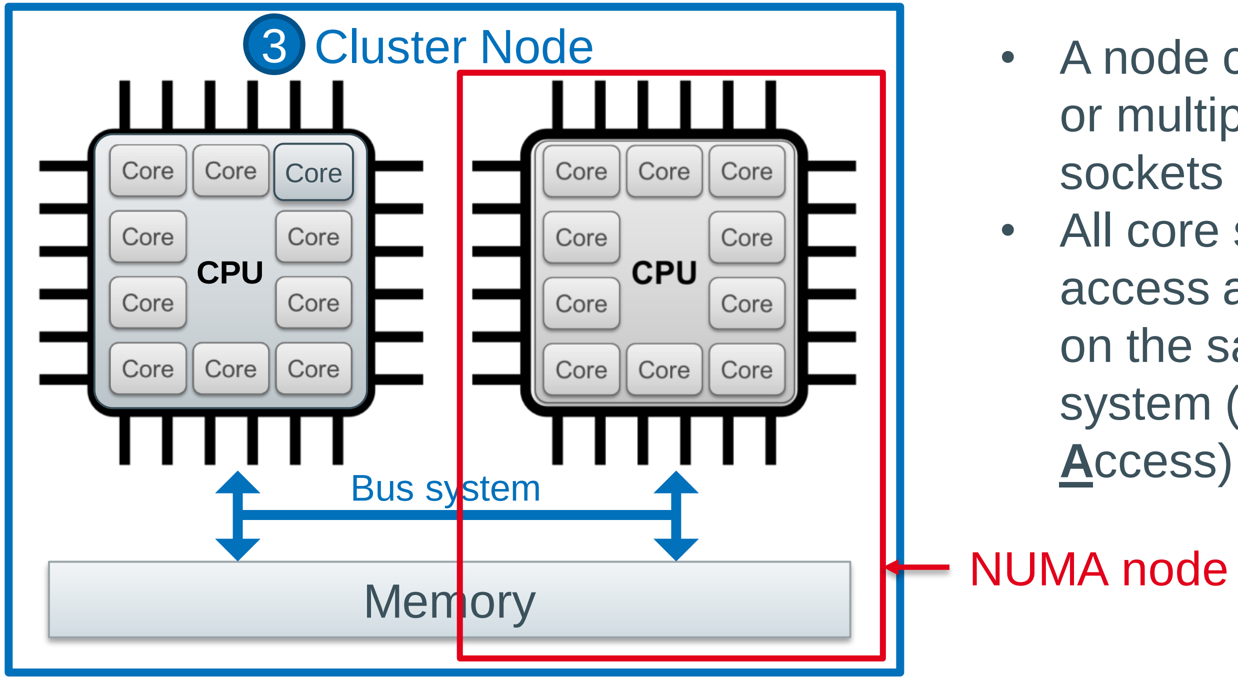
```
#pragma omp parallel sections
{
        #pragma omp section
        { //do something
        }
        #pragma omp section
        { //do something else in parallel
        }

}
```

Creates two threads which can run in parallel (if at least two cores are assigned)
→ There must be no dependencies between the sections, e.g. no commonly used variables

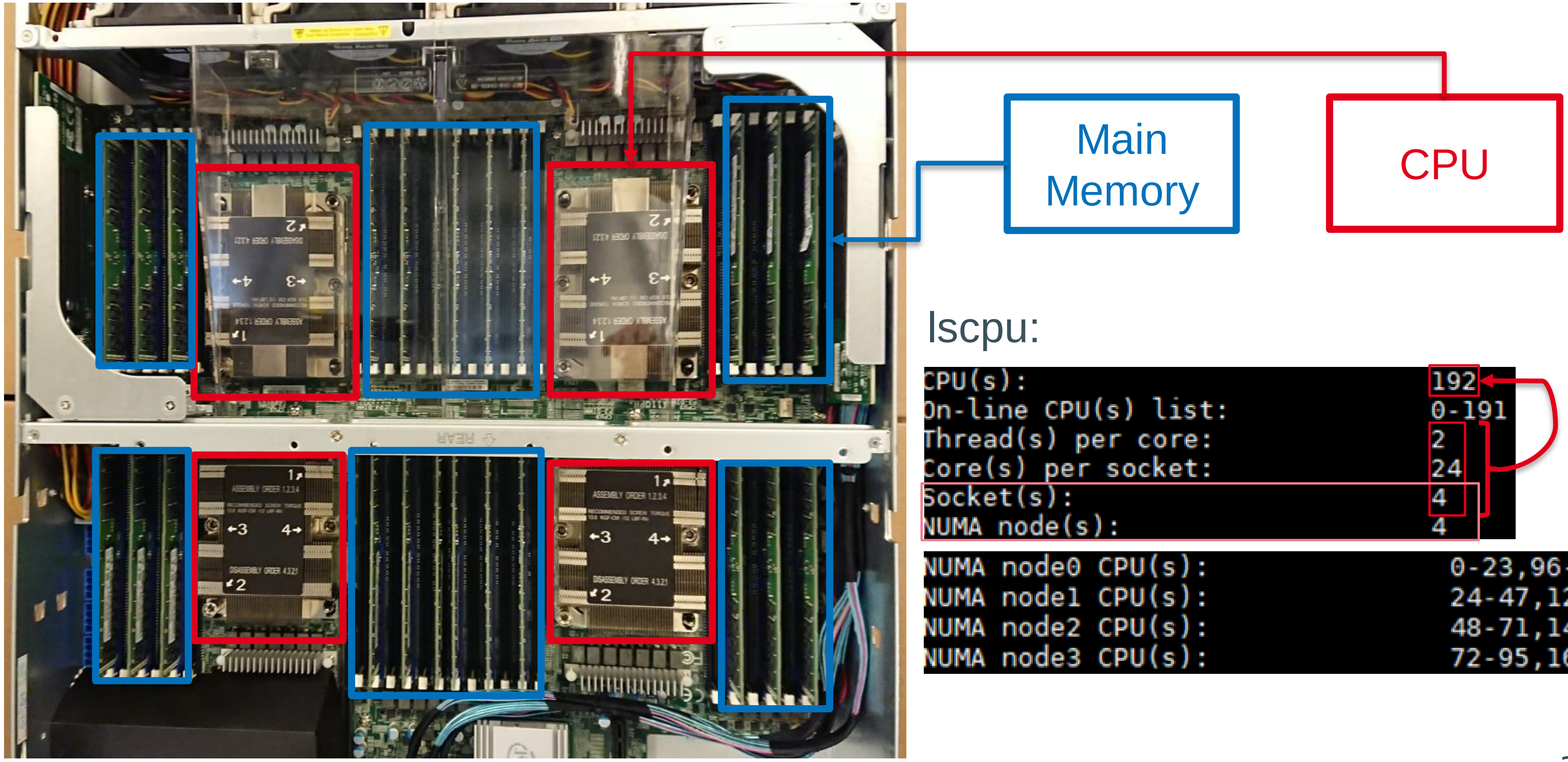
Multi-threading in Julia is very OpenMP-like: https://docs.julialang.org/en/v1/manual/multi-threading/#man-multithreading

Roadmap

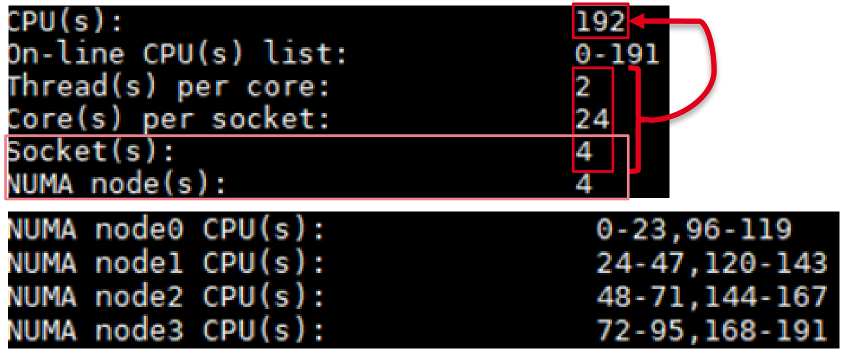


- A node can either feature one or multiple CPUs on multiple sockets
- All core son all CPUs can access all memory modules on the same node → NUMA system (**N**on **U**niform **M**emory **A**ccess)

NUMA Systems



lscpu:



```
CPU(s):                          192
On-line CPU(s) list:             0-191
Thread(s) per core:              2
Core(s) per socket:              24
Socket(s):                       4
NUMA node(s):                    4
```

```
NUMA node0 CPU(s):               0-23,96-119
NUMA node1 CPU(s):               24-47,120-143
NUMA node2 CPU(s):               48-71,144-167
NUMA node3 CPU(s):               72-95,168-191
```

Multithreading In NUMA Systems

Implementation-wise → Just like on a single CPU, you just have more cores available

Thread-pinning → It often makes sense to pin threads to the same NUMA node (if that node has enough cores)

```
numactl --cpunodebind=0 myProgram
```

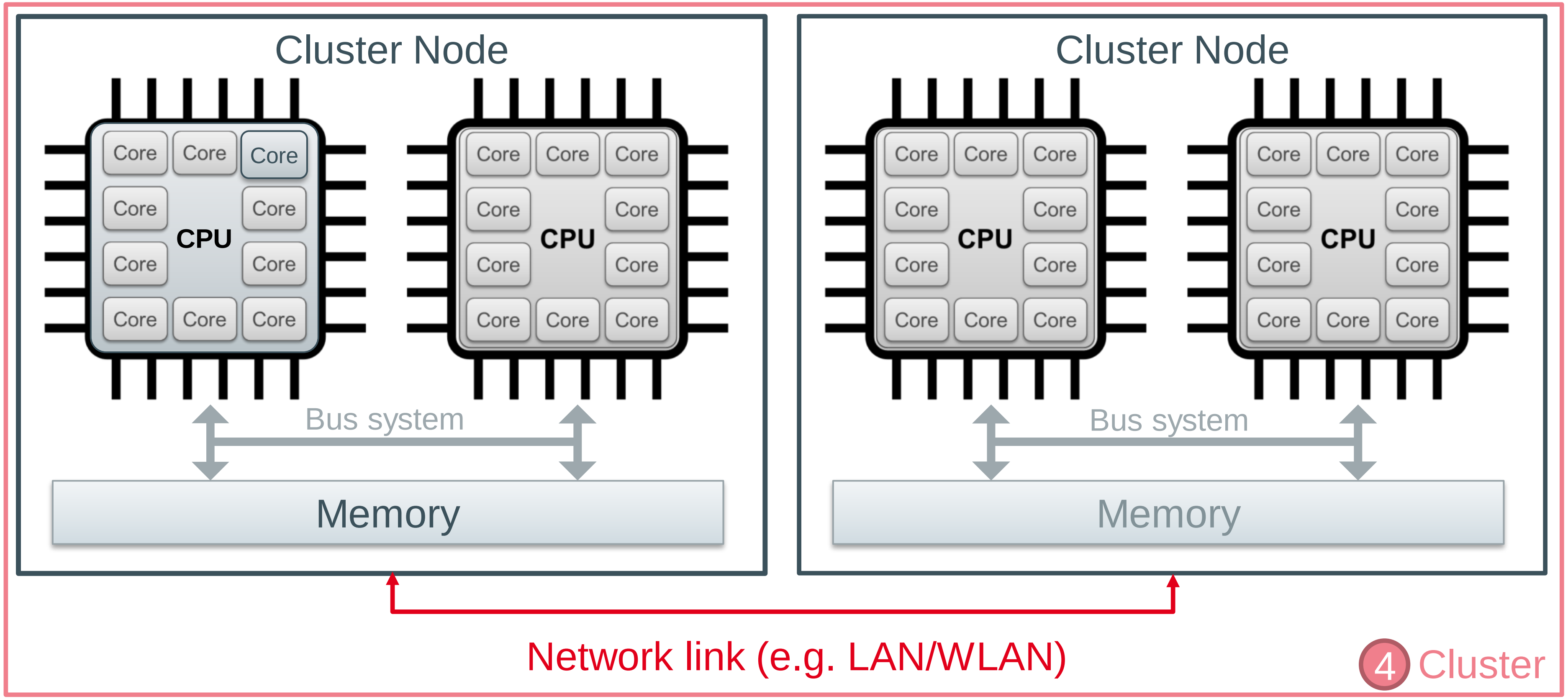
→ myProgram will run on node 0

No knowledge necessary about the ID of the cores on each node

```
numactl --cpunodebind=0 --membind=0 myProgram
```

→ myProgram will run on node 0 AND only allocate memory on node 0

Roadmap


Cluster Node
Core
CPU
Bus system
Memory
Cluster Node
Core
CPU
Bus system
Memory
Network link (e.g. LAN/WLAN)
4 Cluster

Message Passing Interface (MPI)

Popular solution for clusters

```
include <mpi.h>
int main(){
        MPI_init (NULL,NULL);
        int id;
        int err = MPI_Comm_rank(MPI_COMM_WORLD, &id);

        if (id==0) then{
                //do something
        }
        If (id==1) then{
                //do something else
        }
        MPI_Finalize();
}
```

- Build with *mpicc main.c - myProgram*

→ If building with another compiler MPI library must be linked explicitly

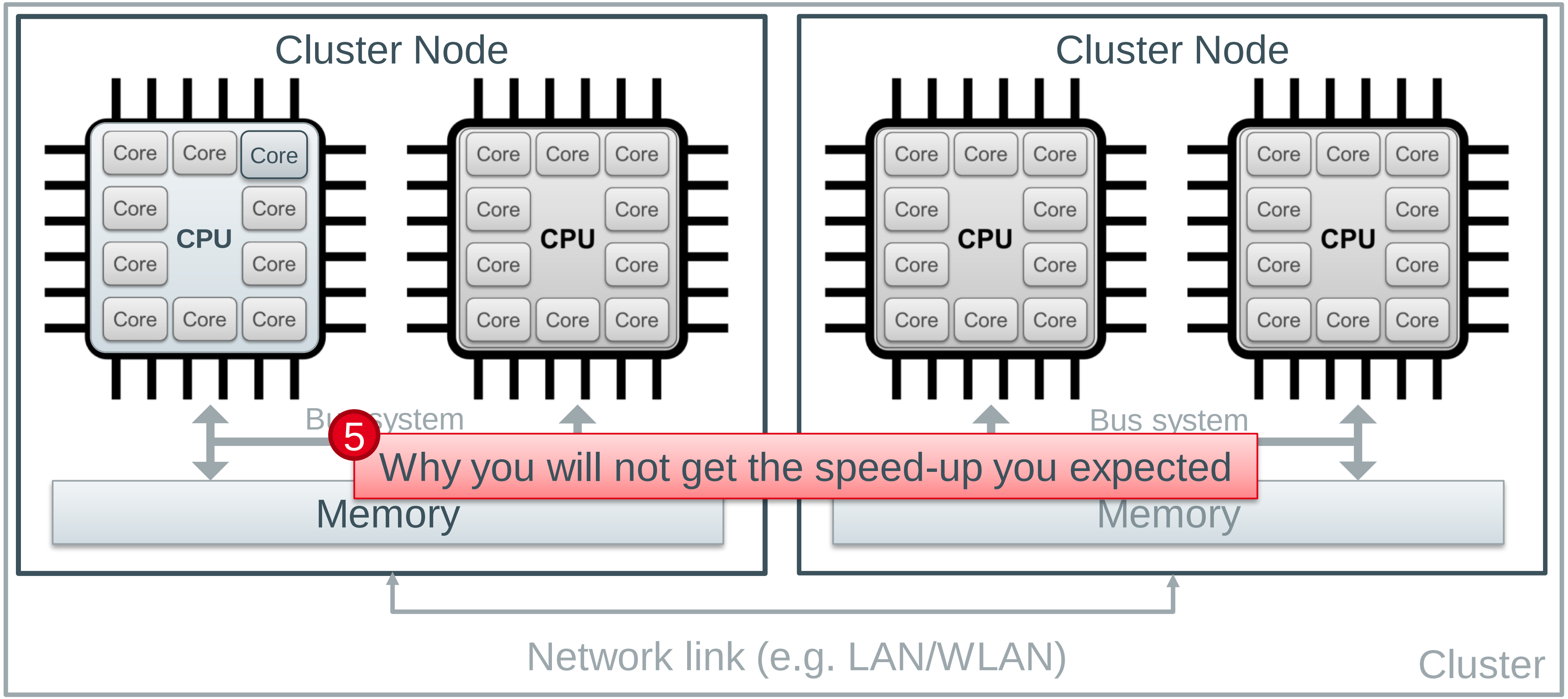
- Run with *mpirun –np 2 myProgram*

→ Runs myProgram in two diffeent processes

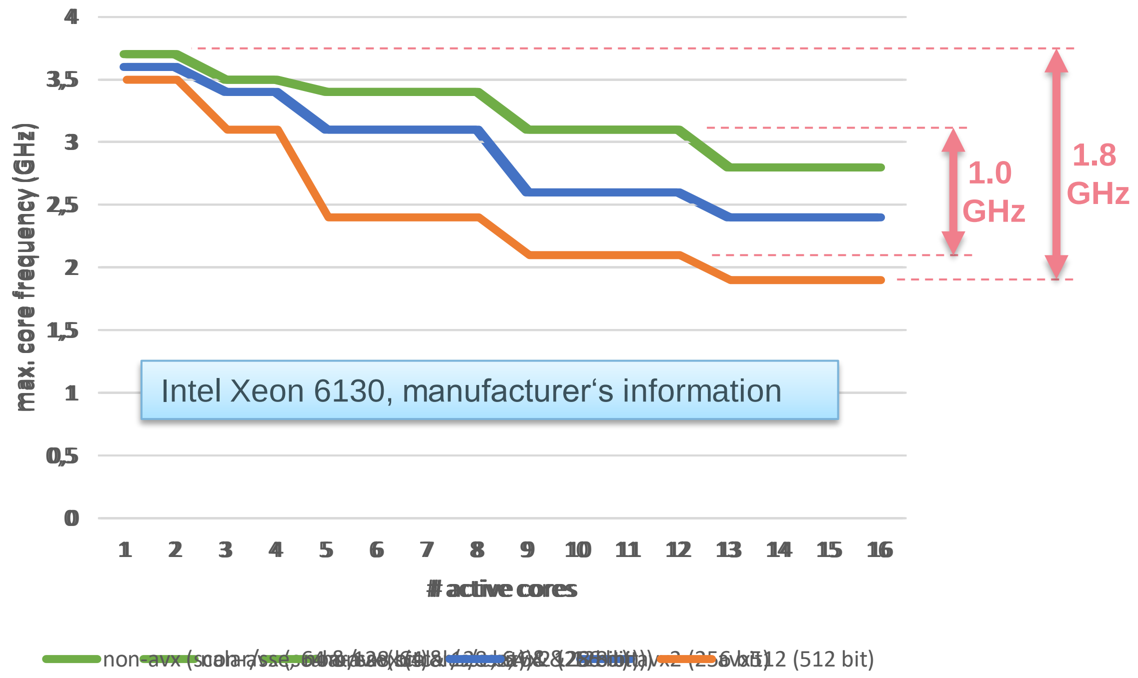
Communication between processes via MPI_Send and MPI_Recv.
→ MPI_Recv blocks a process until data transfer is done

Available for many programming languages, e.g. C/C++, Fortran, Pythn, R, Haskell, Julia…
Also works on a single node, but might be overkill

Roadmap


Cluster Node
Core
CPU
Bus system
Memory
Cluster Node
Core
CPU
Bus system
Memory
Network link (e.g. LAN/WLAN)
Cluster
5
Why you will not get the speed-up you expected

Voltage Scaling



Active transistors produce heat

→ Depends on density of active transistors and frequency

→ Physical constraints limit the ability to conduct heat away

→ Core frequency is regulated down (via voltage) to reduce heat

→ Speed-up is not proportional to #cores

Context Switching

More threads than cores: Instructions and data of a thread are swapped out of the CPU to load another thread → Context Switching

→ As we know, bandwidth for this swap is limited

→ Frequent swaps block your data bus

Operating system place threads on cores according to their own heuristics, e.g. round robin after fixed amount of time → This is done to prevent CPU from overheating in one region, but:

→ It introduces additional context switches

→ The new core might be further away from where the data is written originally

Never spawn more threads than available cores & pin your threads to the same CPU(s) or NUMA node(s) if possible

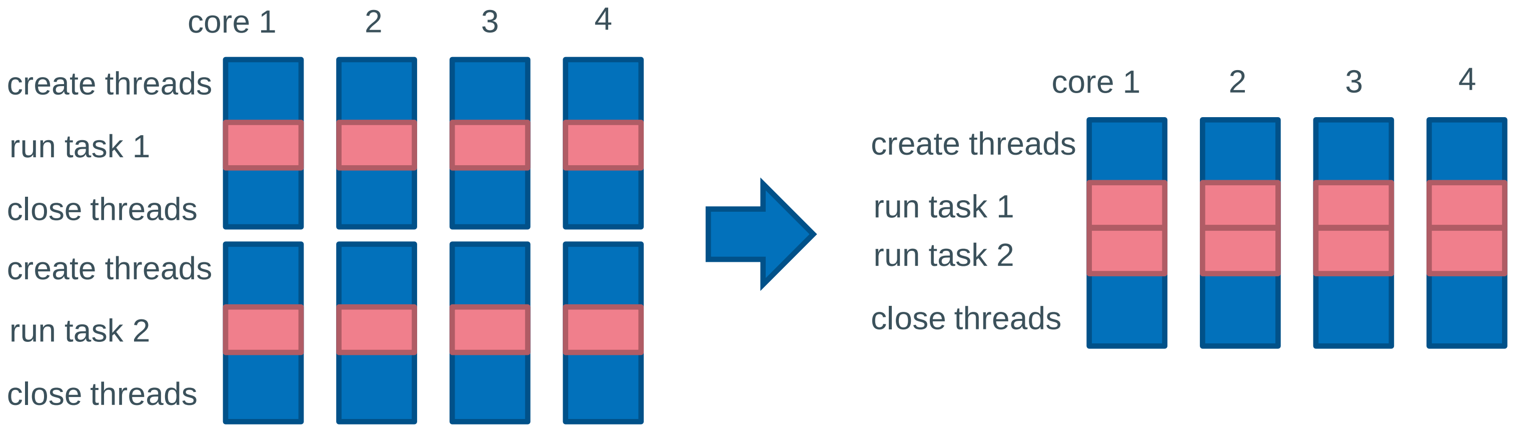
Spawn Overhead

On CPUs, spawning of threads comes with an overhead

→ Short running threads, e.g. only a few ms, do not profit from multithreading

→ Create a thread pool and reuse the same threads for different tasks instead of constantly spawning new threads

→ Put all tasks into a queue and let the threads pull a task when they are done with the last one



The Memory Hierarchy

Size	Type	Bandwidth
16 general purpose 64 bit registers on an x86	Registers	At CPU clock speed
<1MB	L1 cache	A few 100 – a few 1000 GB/s
KB to MB r	L2 cache	A few 100 GB/s
GB to TB r		s per module
		d 5 GB/s
		SD Express), in likely < 1 GB/s
	Disc (hdd)	< 200 MB/s

The maximum degree of (useful) parallelism is limited by memory access!
→ You cannot keep 20 cores busy if they all rely on frequent disc access
→ Random memory access increases the issue by an order of magnitude

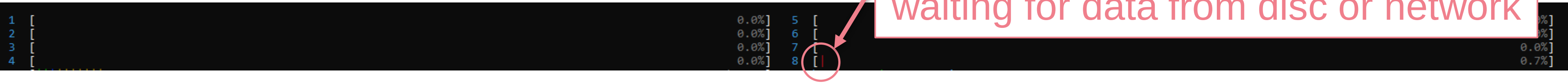
Persistent memory

The Memory Bottleneck

CDCS
CENTER FOR DATA AND COMPUTING
IN NATURAL SCIENCES

htop:

This is all your CPU can do while waiting for data from disc or network



```
1  [                                                          0.0%]   5  [                                                   %]
2  [                                                          0.0%]   6  [                                                   %]
3  [                                                          0.0%]   7  [                                                0.0%]
4  [                                                          0.0%]   8  [|                                               0.7%]
```

Solution: Copy your data to main memory before accessing it

```
loop over lines {
        open file
        read line from file
        do something with line
        close file
}
```

```
open file
data = get file contents
close file
loop over lines{
        read line from data
        do something with line
}
```

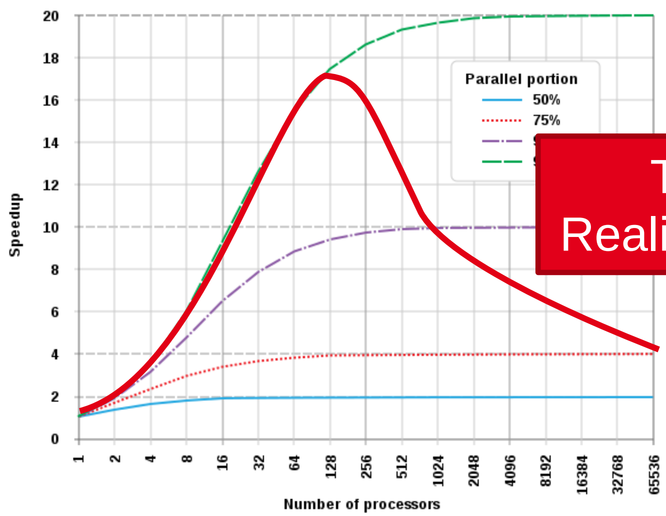
The same goes for writing
→ Collect data in an array, struct, or dataframe
→ Write data to disc when everything is finished

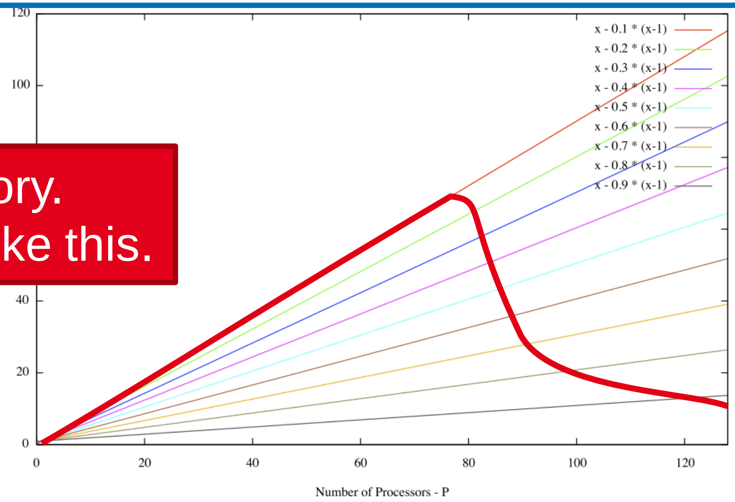
Amdahl‘s Law And Gustafson‘s Law

Not every part of a problem can be parallelized

Amdahl‘s Law: Fixed problem size

Gustafson‘s Law: Growing Problem size





This was theory.
Reality can look like this.

Minimum execution time will not go below execution time of the not parallelizable part, regardless of how many cores we use.

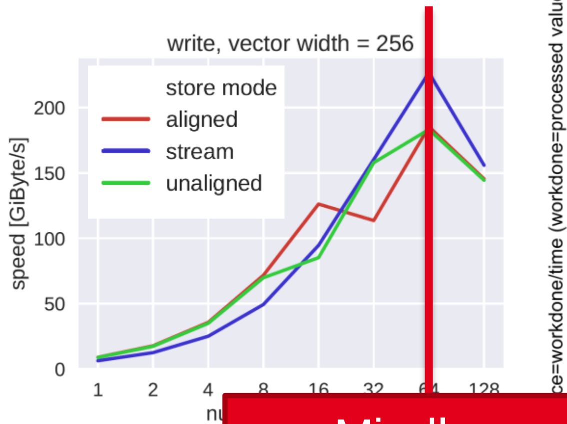
The speedup increases along with the number of cores if the workload (problem size) of the parallelizable part increases, too

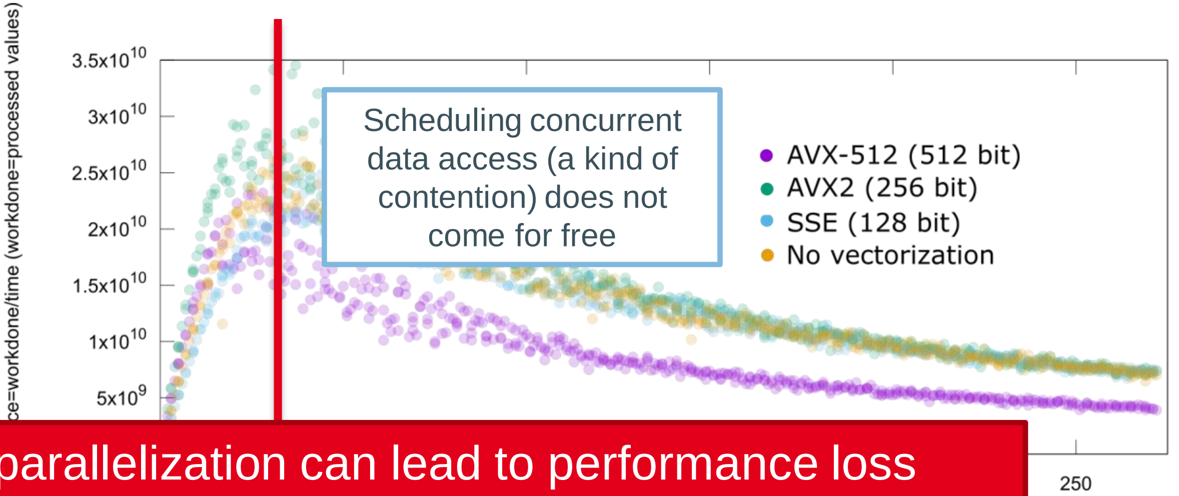
*graphs from wikipedia

Too Many Threads

Main memory can deliver data to more threads in parallel than disc, but it is still limited

Performance decreases when bandwidth is saturated → Hyperthreads not useful





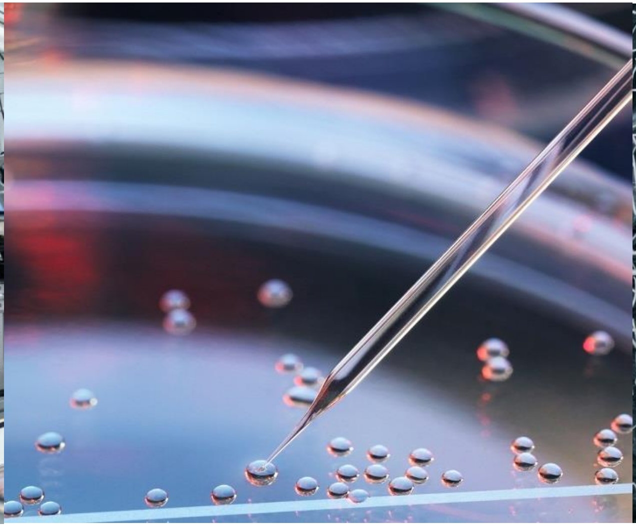
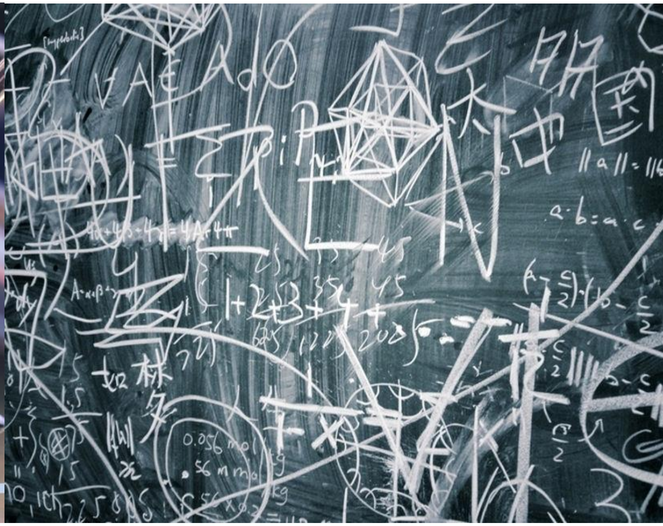
Scheduling concurrent data access (a kind of contention) does not come for free

Sequential write w
Xeon Gold (the sa
Maxwell, 2 thread

re L2 cache)

Mindless parallelization can lead to performance loss
→ Before starting the 20 h run: Try a minimal example of your solution with different thread counts

CDCS
CENTER FOR DATA AND COMPUTING
IN NATURAL SCIENCES

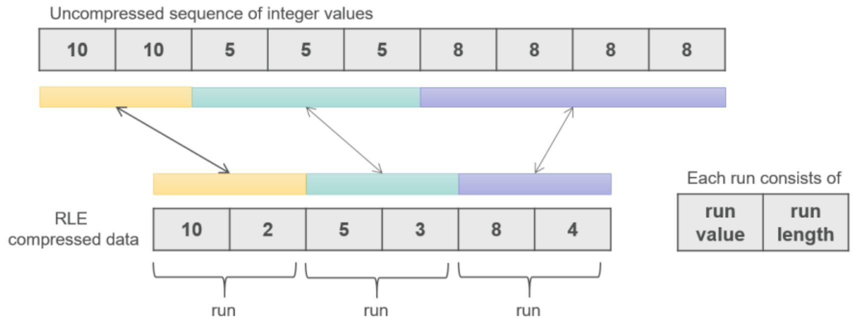
Exercise

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

DESY.

TUHH
Technische Universität Hamburg-Harburg

Exercise: Parallelizing Run Length Encoding



1. Read data from a single file into an array
2. Iterate over array and count duplicates
3. Write all run values and run lengths into a file on disc

a) Which of these tasks do you think are parallelizable?

b) Which parts are parallelizable but will not profit much from parallelization?

c) Vectorization, hyperthreading (multiple threads on a core), multithreading (multiple threads on a CPU in general), or message passing (multiple nodes)?