Erlangen Regional Computing Center

UNIVERSITÄT GREIFSWALD
Wissen lockt. Seit 1456

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Winter term 2020/2021

Parallel Programming with OpenMP and MPI

Dr. Georg Hager
Erlangen Regional Computing Center (RRZE) at Friedrich-Alexander-Universität Erlangen-Nürnberg
Institute of Physics, Universität Greifswald

Lecture 2: Parallel computers

High Performance Computing

Outline of course

- Basics of parallel computer architecture
- Basics of parallel computing
- Introduction to shared-memory programming with OpenMP
- OpenMP performance issues
- Introduction to the Message Passing Interface (MPI)
- Advanced MPI
- MPI performance issues
- Hybrid MPI+OpenMP programming

Erlangen Regional Computing Center

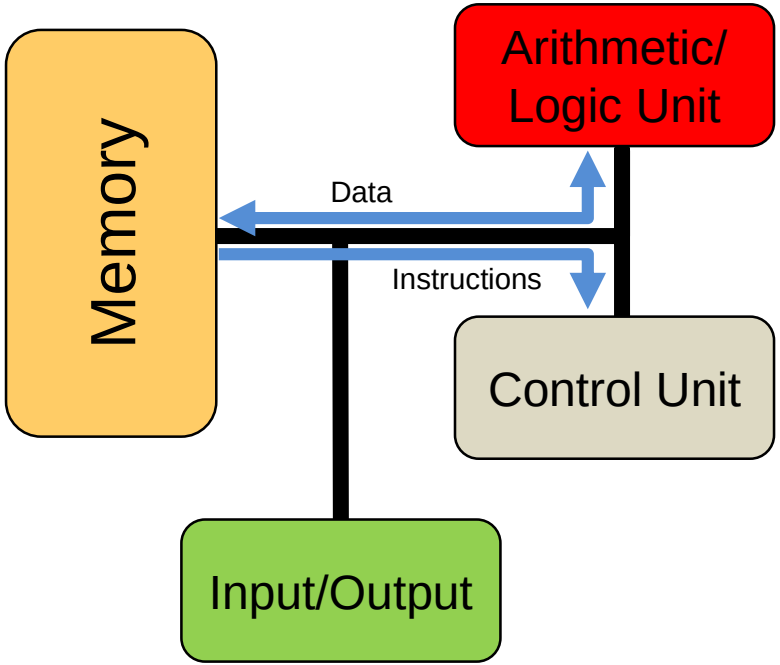
FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

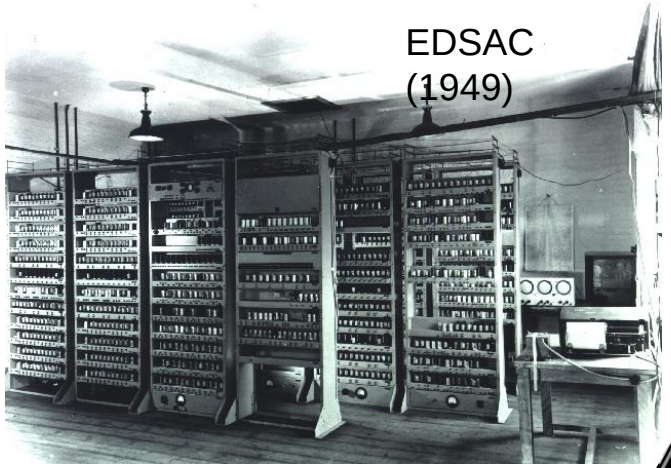
Single-core architecture

A very quick overview

High Performance Computing

At the core: the stored-program computer





Main performance limitation:
Memory access!

Basic resources: *Instruction execution and data movement*

1. Instruction execution

This is the primary resource of the processor. All efforts in hardware design are targeted towards increasing the instruction throughput.

Instructions are the concept of "work" as seen by processor designers.

Not all instructions count as "work" as seen by application developers!

Example: Adding two arrays `A(:)` and `B(:)`

```
do i=1, N
  A(i) = A(i) + B(i)
enddo
```

Processor work:

```
LOAD r1 = A(i)
LOAD r2 = B(i)
ADD r1 = r1 + r2
STORE A(i) = r1
INCREMENT i
BRANCH → top if i<N
```

User work:
`N` Flops (ADDs)

Basic resources: *Instruction execution and data movement*

2. Data transfer

Data transfers are a consequence of instruction execution and therefore a secondary resource. Maximum bandwidth is determined by the request rate of executed instructions and technical limitations (bus width, speed).

Example: Adding two arrays `A(:)` and `B(:)`

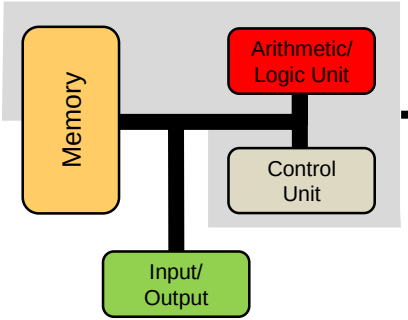
```
do i=1, N
  A(i) = A(i) + B(i)
enddo
```

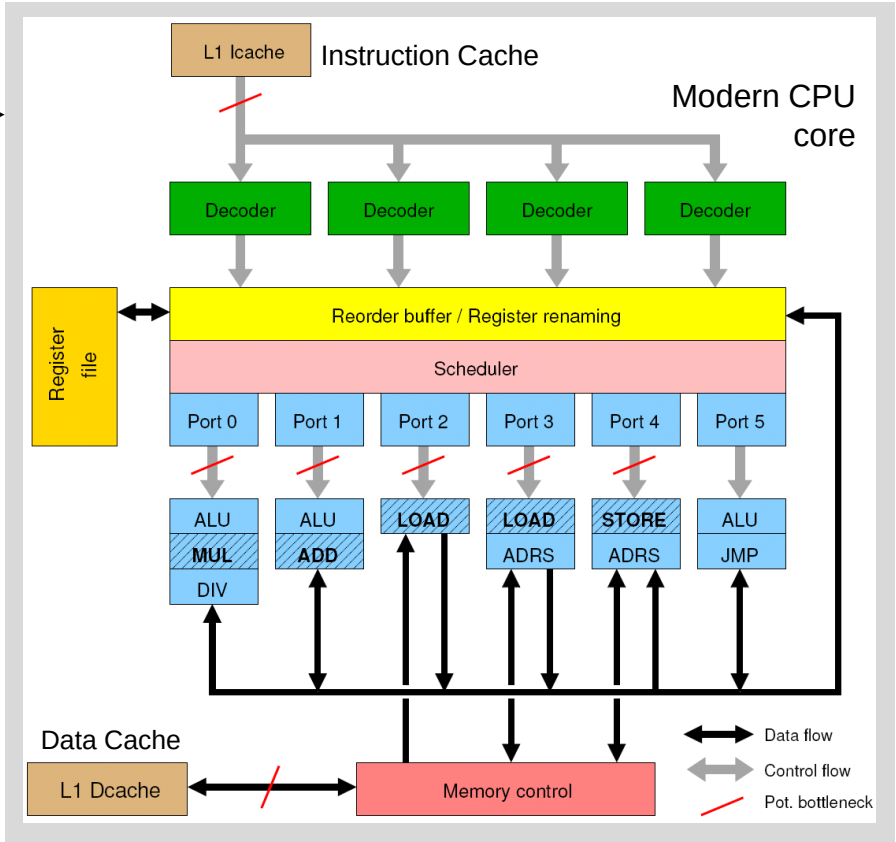
Data transfers:
8 byte: `LOAD r1 = A(i)`
8 byte: `LOAD r2 = B(i)`
8 byte: `STORE A(i) = r2`
Sum: **24 byte**

Crucial question: What determines the runtime?

- Data transfer?
- Code execution?
- Something else?

From theory to reality: General-purpose (cache based) microprocessor core





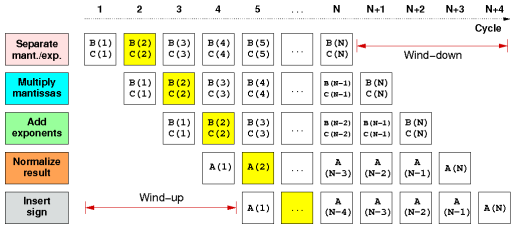
Measures to improve performance:

- Instruction execution is pipelined
- Instructions are executed out of program order (semantics permitting)
- Instructions can be inherently parallel (SIMD)
- Caches store often used data for quick reference

Important in-core features

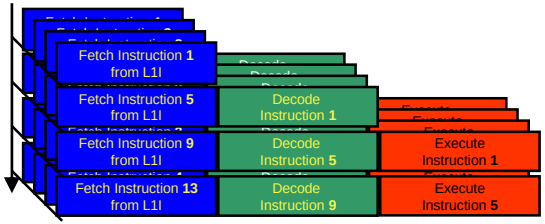
Pipelining:

Instruction execution
in multiple steps



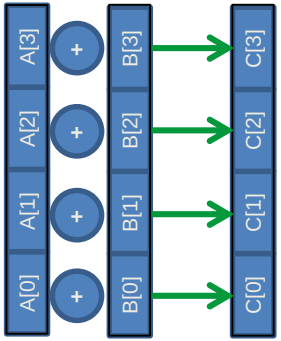
Superscalarity:

Multiple instructions
per cycle



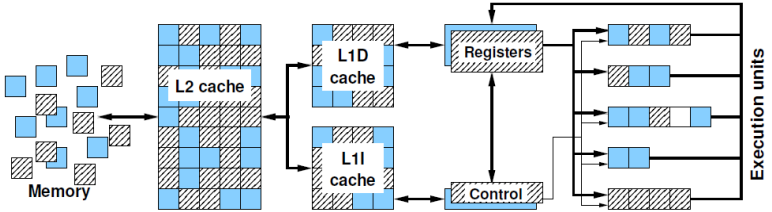
Single Instruction Multiple Data:

Multiple operations per instruction



Simultaneous Multi-Threading:

Multiple instruction sequences in parallel

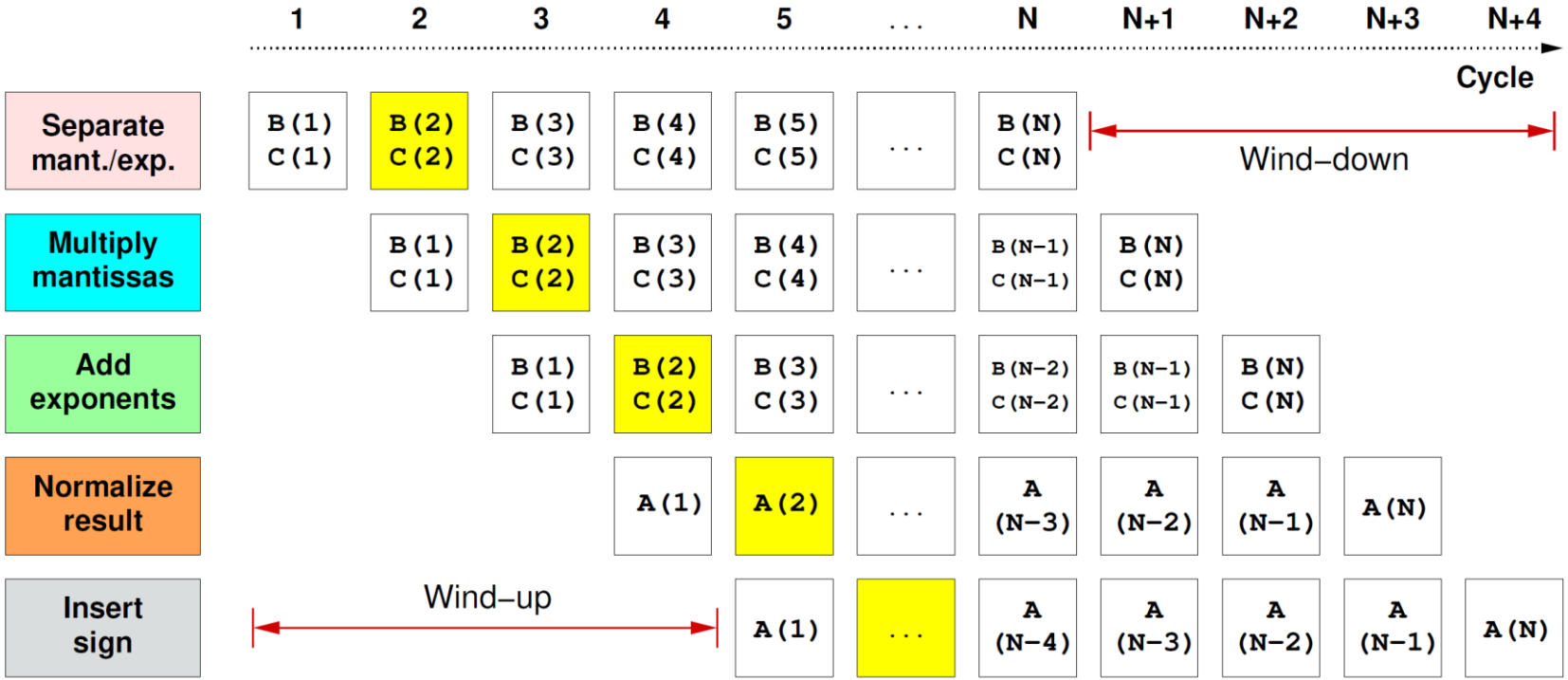


Pipelining of functional units

- Idea:
  - Split complex instruction into several simple / fast steps (stages)
  - Each step takes the same amount of time, e.g., a single cycle
  - Execute different steps on different instructions at the same time (in parallel)
- Benefits:
  - Core can work on several independent instructions simultaneously
  - One instruction finished each cycle after the pipeline is full
- Drawback:
  - Pipeline must be filled; large number of independent instructions required
  - Requires complex instruction scheduling by hardware (out-of-order execution) or compiler (software pipelining)

- Pipelining is widely used in modern computer architectures

5-stage multiplication pipeline: `A(i)=B(i)*C(i) ; i=1,...,N`

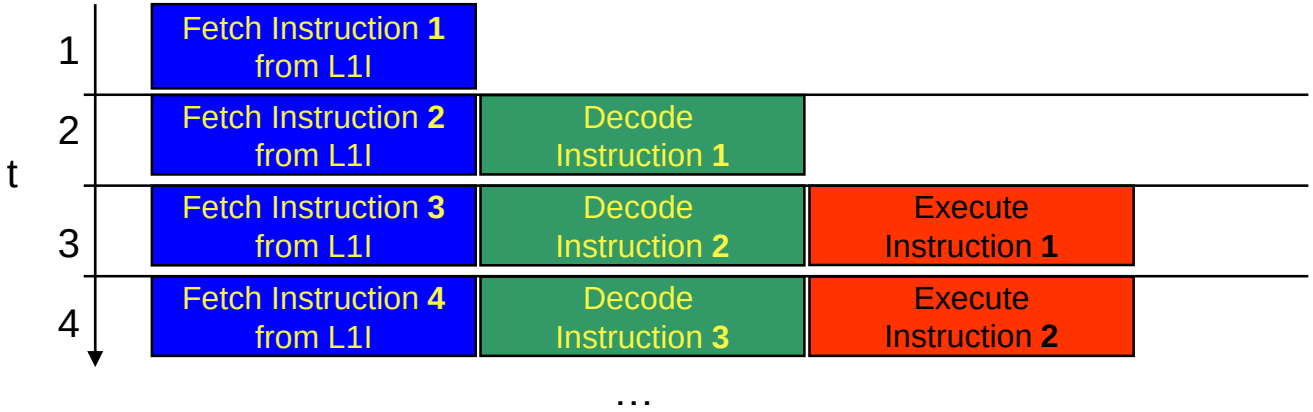


First result is available after 5 cycles (=latency of pipeline)!

Wind-up/-down phases: Empty pipeline stages

Pipelining: The instruction pipeline

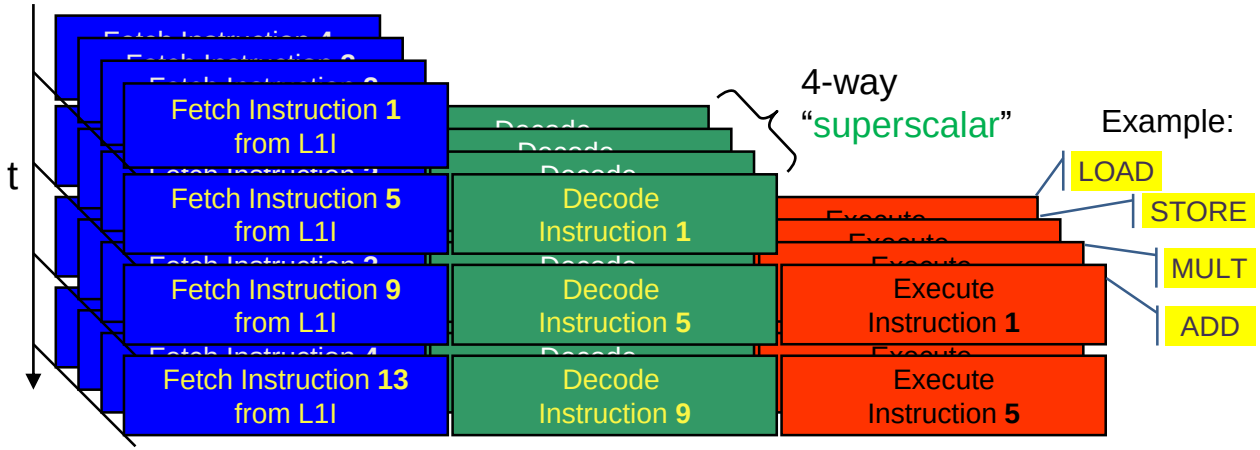
- Besides functional units, instruction execution itself is also pipelined with at least 3 steps:



- Branches can stall this pipeline! (speculative execution, predication)
- Each unit is pipelined itself (e.g., execute = multiply pipeline)
- Pipelines can be chained (e.g., LOAD/LOAD → MULT → STORE)

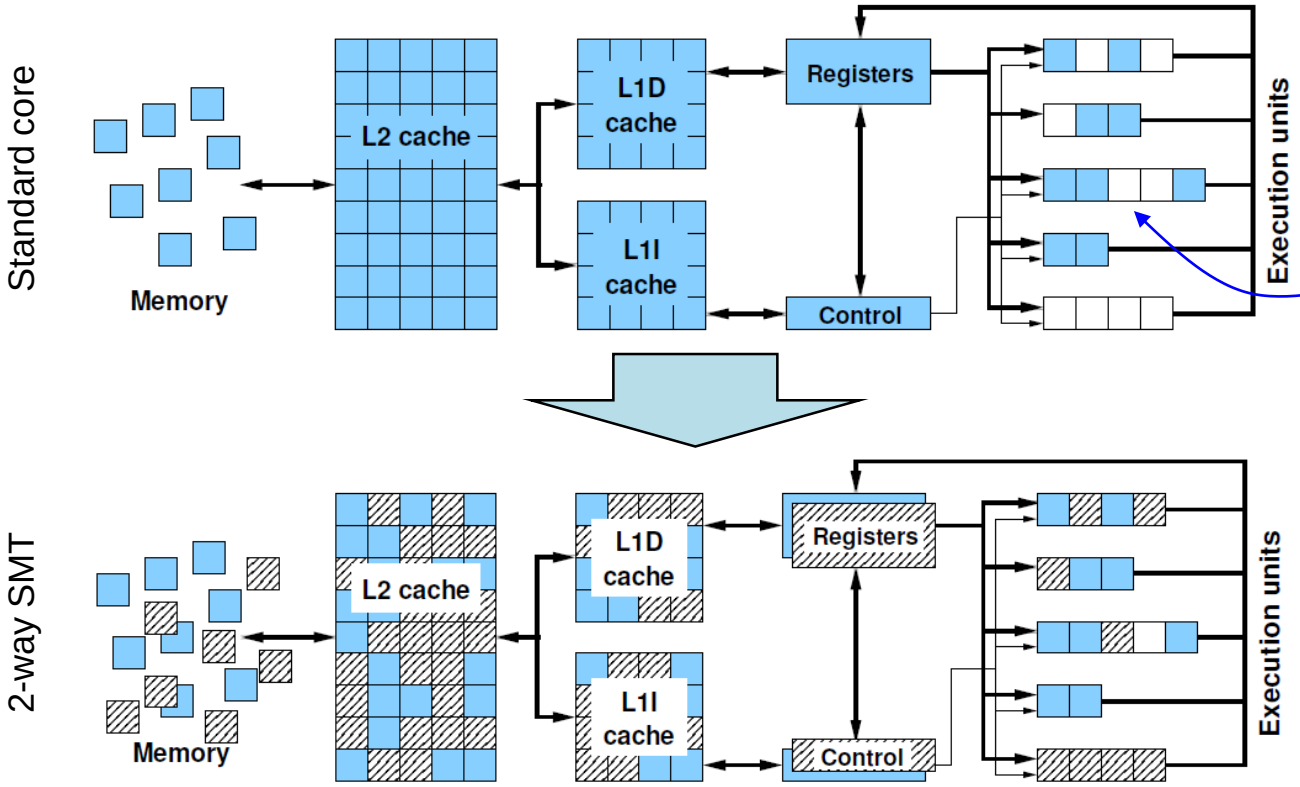
Instruction-level parallelism: Superscalar execution

- Multiple units enable use of Instruction Level Parallelism (ILP): Instruction stream is “parallelized” on the fly



- Issuing m concurrent instructions per cycle: m-way superscalar
- Modern processors are 3- to 6-way superscalar & can perform 2 floating point instructions per cycle

Simultaneous multi-threading (SMT)

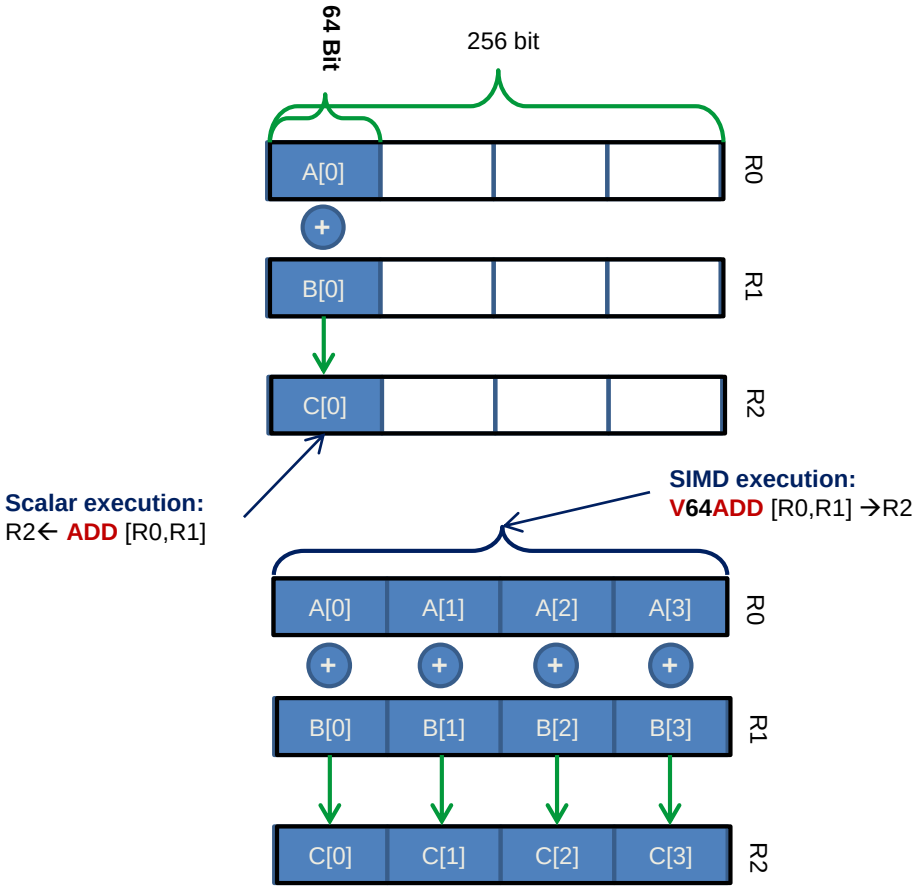


- Pipelines often underutilized due to dependencies, waiting times, etc.
- "Pipeline bubbles" mean wasted resources

- SMT can improve the utilization of pipelines
- It does not improve any other resources on the chip!
- Need to run multiple threads

SIMD processing

- Single Instruction Multiple Data (SIMD) instructions allow the execution of the same operation on “wide” registers from a single instruction
- x86 SIMD instruction sets:
  - SSE: register width = 128 Bit → 2 double precision floating point operands
  - AVX(2): register width = 256 Bit → 4 double precision floating point operands
  - AVX-512: … you guessed it!
- It is not specified if these operations are concurrent
  - They mostly are, though, on modern standard CPUs

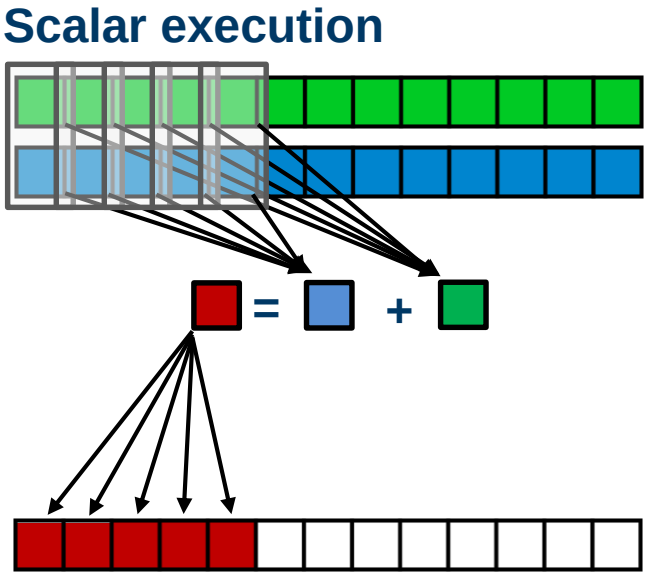


Scalar (non-SIMD) execution

```
double *A, *B, *C;
for (int j=0; j<size; j++){
    A[j] = B[j] + C[j];
}
```

Register width:

- 1 operand (scalar)



Data-parallel execution (SIMD)

```
double *A, *B, *C;
for (int j=0; j<size; j++){
    A[j] = B[j] + C[j];
}
```

Register widths (double prec.):

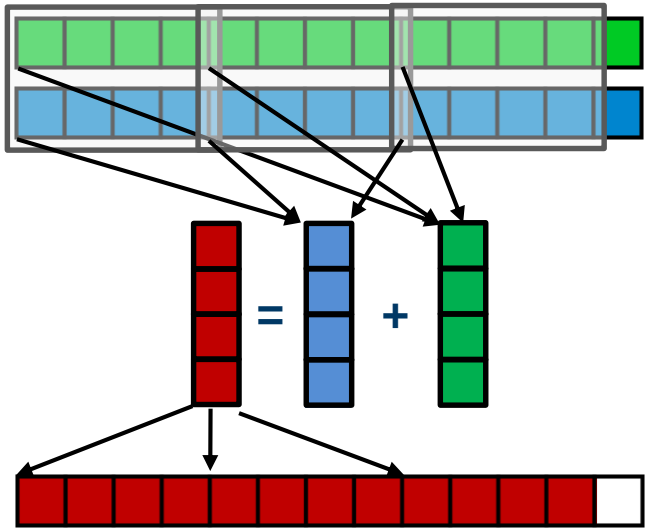
- 1 operand

- 2 operands (SSE)

- 4 operands (AVX)

- 8 operands (AVX512)

SIMD execution



SIMD by compiler

Steps (done by the compiler) for “SIMD processing”

```
for(int i=0; i<n;i++)
            C[i]=A[i]+B[i];
```

“Loop unrolling”

This should not be done by hand!

```
for(int i=0; i<n;i+=4){
            C[i]   =A[i]   +B[i];
            C[i+1]=A[i+1]+B[i+1];
            C[i+2]=A[i+2]+B[i+2];
            C[i+3]=A[i+3]+B[i+3];}
//remainder loop handling
```

Load 256 Bits starting from address of `A[i]` to register `R0`

Add the corresponding 64 Bit entries in `R0` and `R1` and store the 4 results to `R2`

Store `R2` (256 Bit) to address starting at `C[i]`

```
LABEL1:
            VLOAD R0 ← A[i]
            VLOAD R1 ← B[i]
            V64ADD[R0,R1] → R2
            VSTORE R2 → C[i]
            i←i+4
            i<(n-4)? JMP LABEL1
//remainder loop handling
```

What is the peak performance of a core?

$$P_{core} = n_{super}^{FP} \cdot n_{FMA} \cdot n_{SIMD} \cdot f$$

Super-scalarity | FMA factor | SIMD factor | Clock Speed

Typical representatives	n_{super}^{FP} [inst./cy]	n_{FMA}	n_{SIMD} [ops/inst.]	@market	Ex. model	f [Gcy/s]	P_{core}[GF/s]
Nehalem	2	1	2	Q1/2009	X5570	2.93	11.7
Westmere	2	1	2	Q1/2010	X5650	2.66	10.6
Sandy Bridge	2	1	4	Q1/2012	E5-2680	2.7	21.6
Ivy Bridge	2	1	4	Q3/2013	E5-2660 v2	2.2	17.6
Haswell	2	2	4	Q3/2014	E5-2695 v3	2.3	36.8
Broadwell	2	2	4	Q1/2016	E5-2699 v4	2.2	35.2
Skylake	2	2	8	Q3/2017	Gold 6148	2.4	76.8
AMD Zen	2	2	2	Q1/2017	Epyc 7451	2.3	18.4
AMD Zen2	2	2	4	Q4/2019	Epyc 7642	2.3	36.8
IBM POWER8	2	2	2	Q2/2014	S822LC	2.93	23.4

Erlangen Regional Computing Center

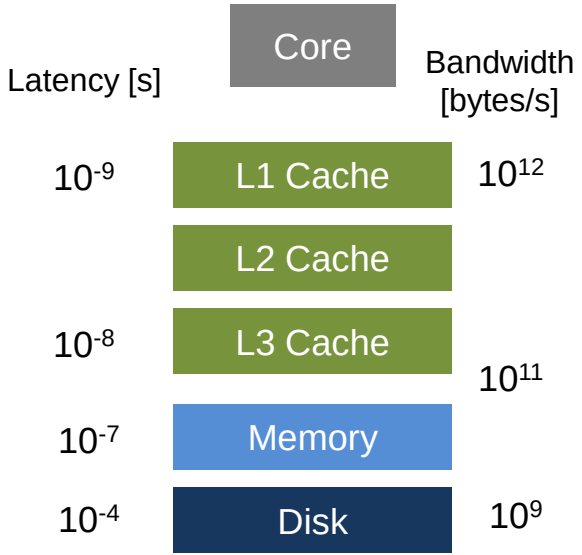
FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

The memory hierarchy

High Performance Computing

Memory hierarchy

- Data transfers are the #1 limiting factor in computing
  - Main memory is too slow to keep up with the CPU's hunger for data
- You can either build a small and fast memory or a large and slow memory
  - Caches hold often-used data for fast reference
  - Multiple levels (the larger the slower)
  - Data transfers occur in "bursts" of single cache lines (typically 64 bytes)
- The purpose of many optimizations is to avoid slow data paths



Characterization of data paths

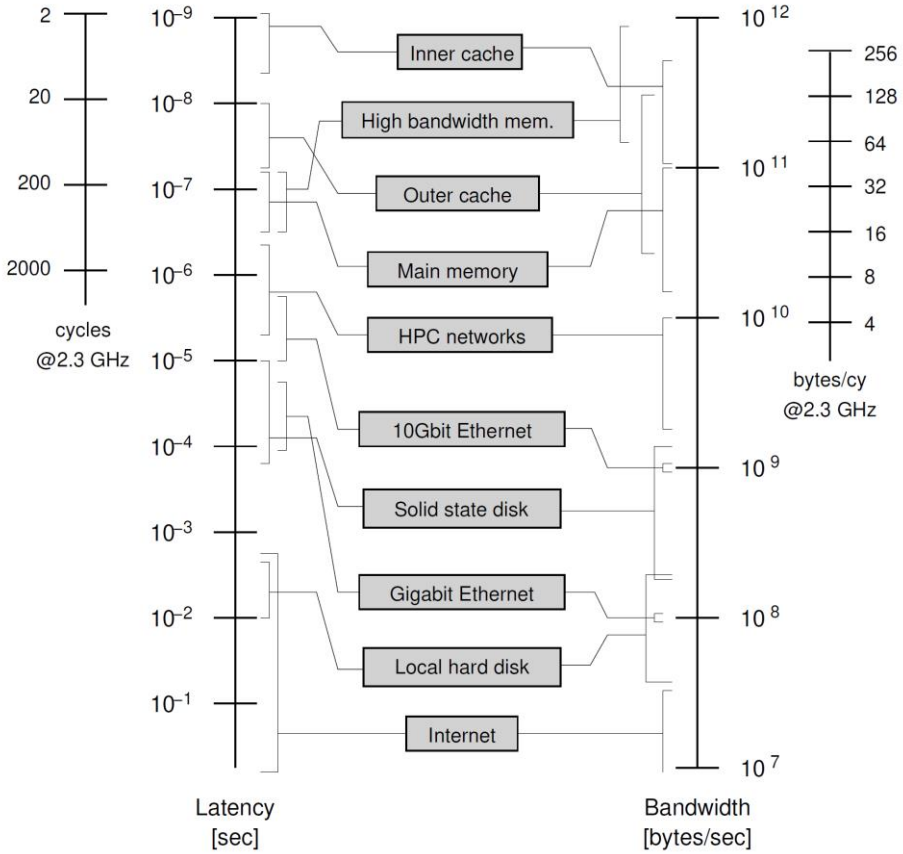
- Basic model: Latency & bandwidth
  Transfer time for message (N bytes)

$$T = \lambda + \frac{N}{b}$$

λ: latency (set-up time) [s]
b: bandwidth of data path [byte/s]

- Effective bandwidth of message transfer:

$$B_{\text{eff}} = \frac{N}{T} = \frac{N}{\lambda + \frac{N}{b}}$$



Single core: Summary

- A single CPU core is still a stored-program computer
- Multiple hardware optimizations to boost performance
  - Pipelining
  - Out-of-order execution and superscalarity
  - Simultaneous multi-threading
  - SIMD (main driver of peak performance today)
  - Caches

- Parallelism is already built into the single core
  - Mostly addressed by compiler
  - Manual optimizations may still be useful sometimes

Erlangen Regional Computing Center

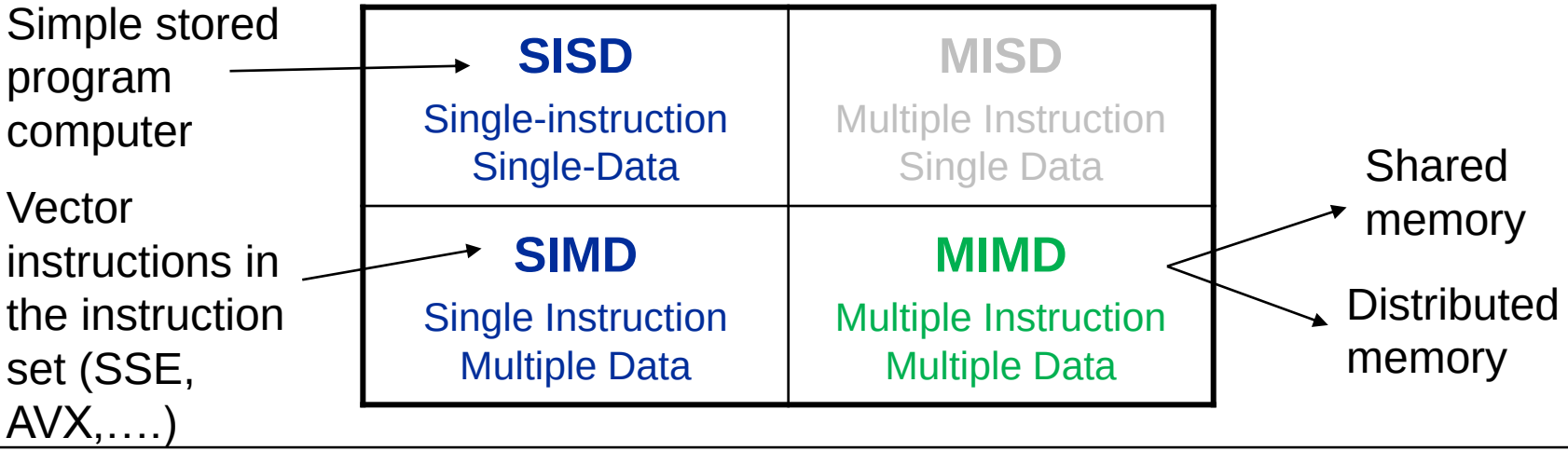
FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Parallel computer architecture

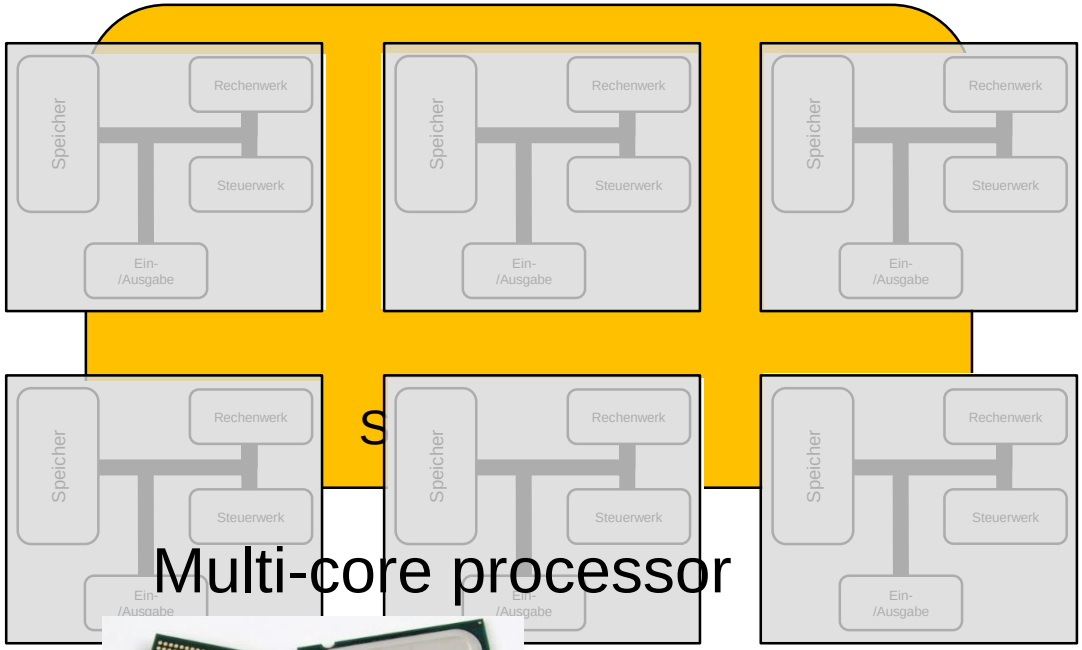
High Performance Computing

Classification of parallel computers

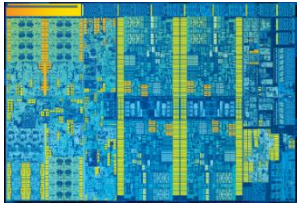
- Parallel Computing: A number of compute elements solve a problem in a cooperative way
- Parallel Computer: A number of compute elements connected in such a way as to do parallel computing for a large set of applications
- Classification according to Flynn (1972) DOI: 10.1109/TC.1972.5009071

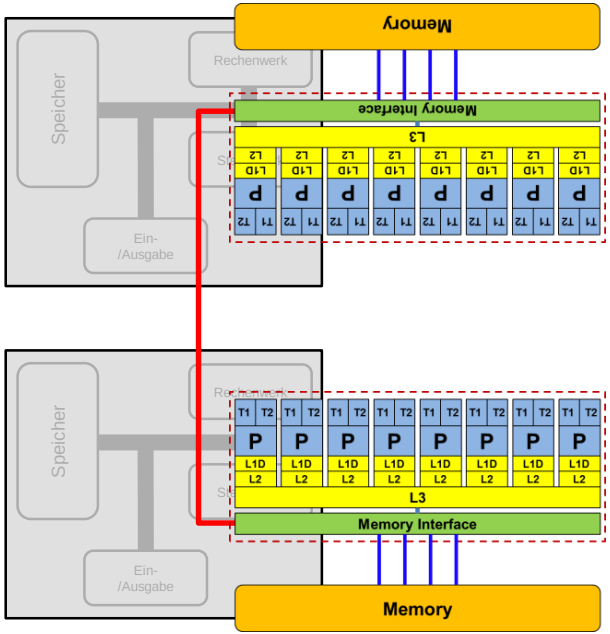


Shared memory: a single cache-coherent address space



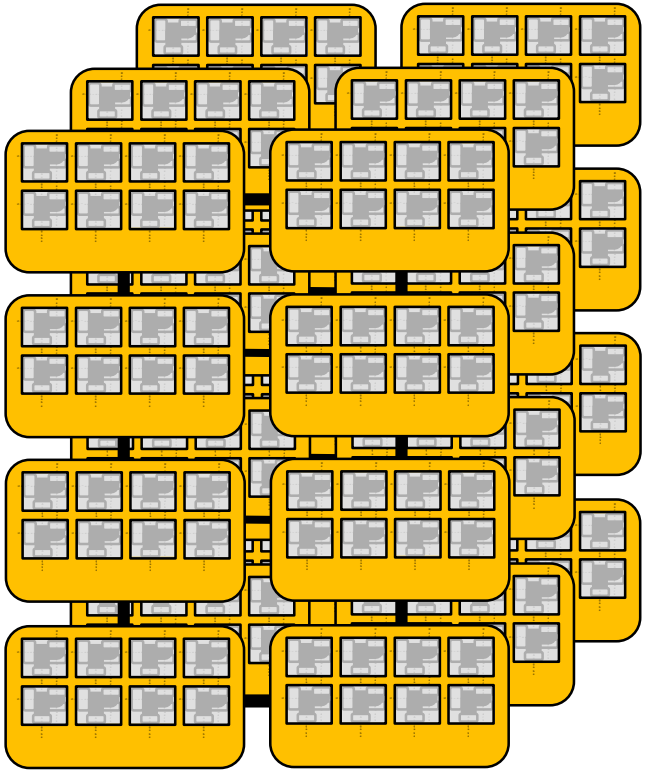
Multi-core processor





Multiple CPU chips per node

Distributed memory: no cache-coherent single address space



Cluster/
supercomputer

Modern supercomputers are
shared-/distributed-memory hybrids

Erlangen Regional Computing Center

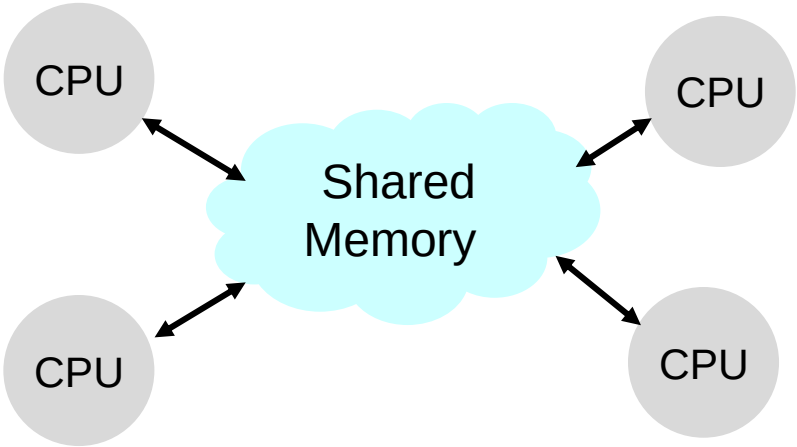
FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Shared-memory parallel computers

High Performance Computing

Shared memory

- Single address space for all processors/cores
- Cache coherent, i.e., changes in one cache will be communicated to all others for consistency

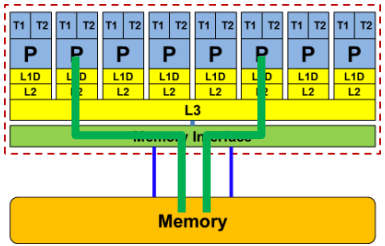


- Two basic variants: UMA and ccNUMA

UMA vs. ccNUMA

[cache-coherent]
Uniform Memory Access

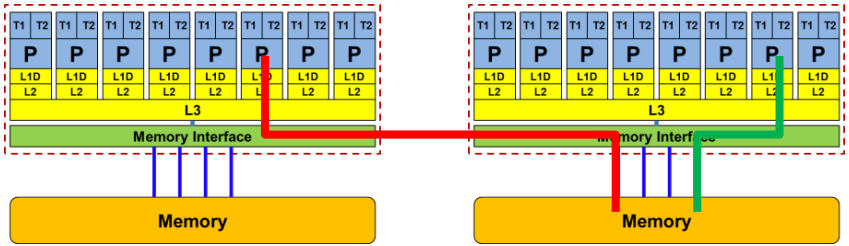
All memory accessible by all cores with equal latency and bandwidth



cache-coherent
Non-Uniform Memory Access

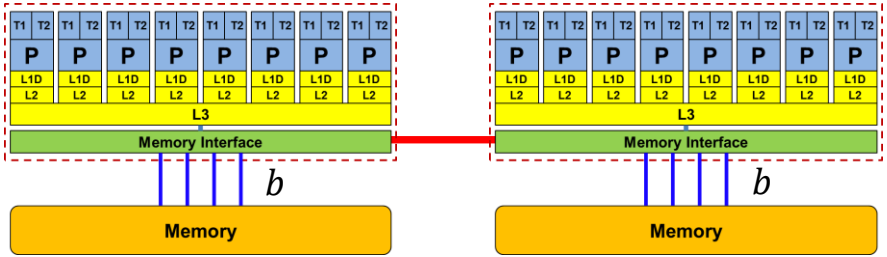
Latency and bandwidth vary depending on mutual position of core and memory

But why???



Why ccNUMA?

- Many algorithms rely on high Memory bandwidth:



$$b = \frac{V}{T}$$

V data transferred over memory bus [byte]

T wallclock time [s]

- Advantage: Easier (cheaper) to build multiple domains with smaller bandwidth than one UMA domain with high bandwidth
- Disadvantage: Adds “topology” (non-uniformity in memory access, need to know where my threads are running)

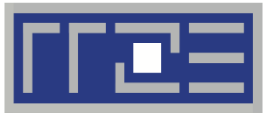
Shared-memory parallel programming

- Many programming models exist
- Popular in scientific computing: OpenMP (https://openmp.org)
- Source code directives interpreted by compiler (+ small API)
- Example:

```
double s = 0.0, *a, *b, *c;
...
#pragma omp parallel for reduction(+:s)
for(int i=0; i<n; i++) {
  a[i] = b[i] + c[i];
  s = s + a[i];
}
```

OpenMP has

- a directive sentinel
- parallelization directives
- work-sharing directives
- clauses
- … and much more

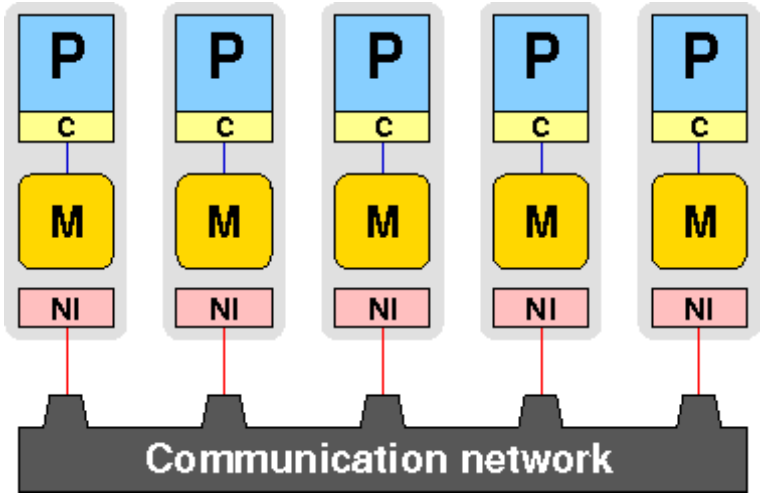


Distributed-memory parallel computers

Distributed-memory systems “back in the day”

“Pure” distributed-memory system:

- Individual processors with exclusive local memory (M) and a network interface (NI) → one “node” == one processor core
- Dedicated communication network
- Parallel program == one process per node
- Data exchange via “message passing” over the network

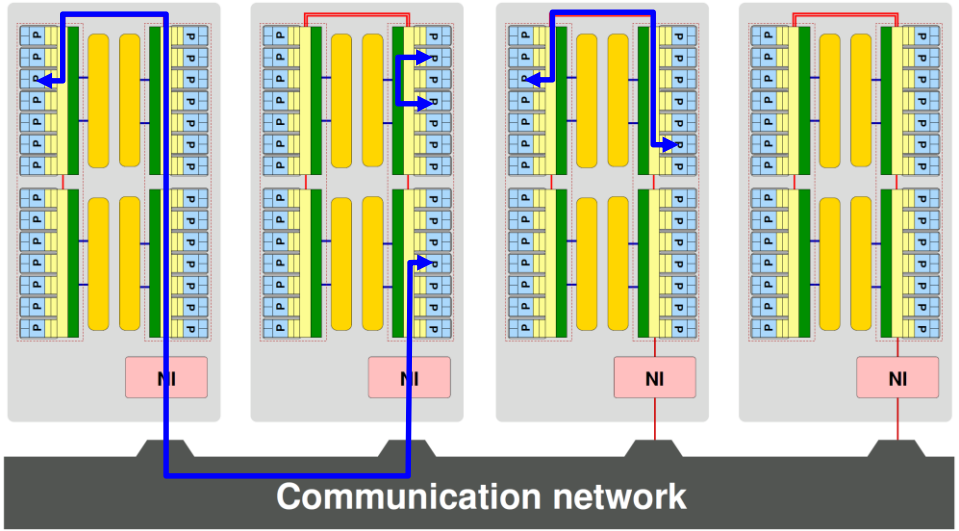


- This was a thing not so long ago…

Distributed-memory systems today

"Hybrid" distributed-/shared-memory systems

- Cluster of networked shared-memory nodes
- ccNUMA architecture per node
- Multiple cores per ccNUMA domain



- Expect strong topology effects in communication performance
  - Intra-socket, inter-socket, inter-node, all have different λ and b
  - On top: Effects from network structure

Distributed-memory parallel programming

- Many programming models exist
- Dominant in scientific computing: MPI, the Message Passing Interface (https://mpi-forum.org)
- Library standard, several open & commercial implementations (Intel, OpenMPI,…)
- Processes communicating via message transfers
- Hundreds of functions
- Significantly more complex than OpenMP
- Can use MPI on shared memory!

```
include <mpi.h>

int main(int argc, char** argv) {
   int rank, size;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   printf("Hello World! I am %d of %d\n",
          rank,size);

   MPI_Finalize();
}
```

Summary on parallel computer architecture

- Modern systems exhibit parallelism on multiple levels
  - Multi-core, multi-NUMA-domain, multi-node
- Mixture of shared- and distributed-memory architecture
  - Shared memory on the node, distributed between nodes

Programming models: There are ten a penny, but…:

- OpenMP
  - Shared-memory programming
  - Compiler directives, thread based
- Message Passing Interface (MPI)
  - Distributed-memory programming
  - Library calls, process based