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 3: Parallel computing and its limits

HPC 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

# Simple but enlightening scalability models

High Performance Computing

# Metrics to quantify the efficiency of parallel computing

- $T(N)$: execution time of some fixed workload with $N$ workers
- How much faster than with a single worker?

  → parallel speedup: $S(N) = \frac{T(1)}{T(N)}$

- How efficiently do those $N$ workers do their work?

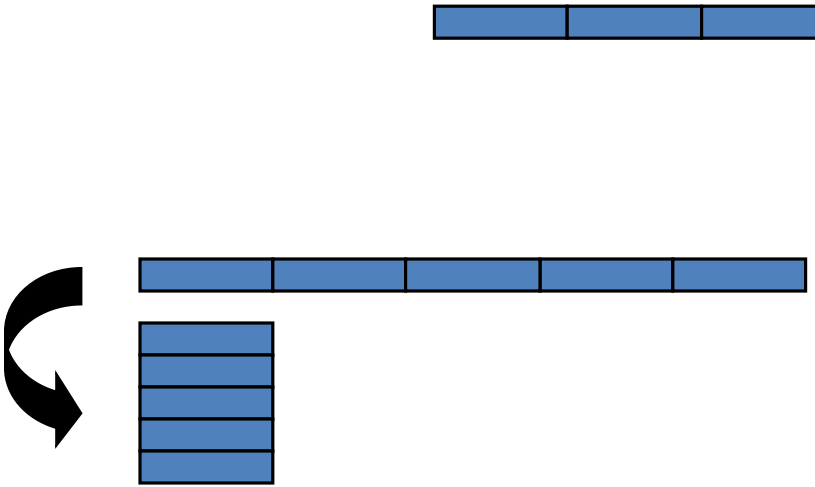
  → parallel efficiency: $\varepsilon(N) = \frac{S(N)}{N}$

Can we predict $S(N)$? Are there limits to it?

- Warning: These metrics are not performance metrics!

# Assumptions for basic scalability models

- Scalable hardware: $N$ times the iron can work $N$ times faster
- Work is either fully parallelizable or not at all
- For the time being, assume a constant workload



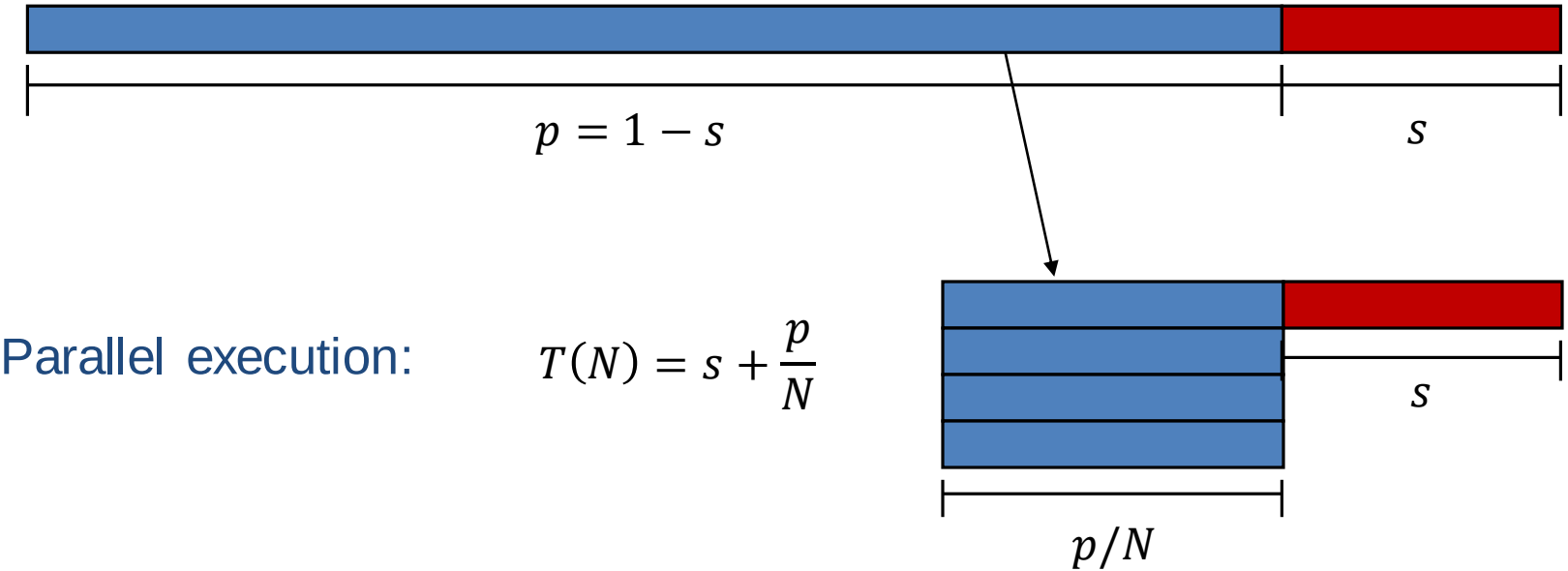
Ideal world:
All work is perfectly parallelizable
$S(N) = N, \qquad \varepsilon = 1$

# A simple speedup model for fixed workload

One worker normalized execution time: $T(1) = s + p = 1$

$s$: runtime of purely serial part

$p$: runtime of perfectly parallelizable part



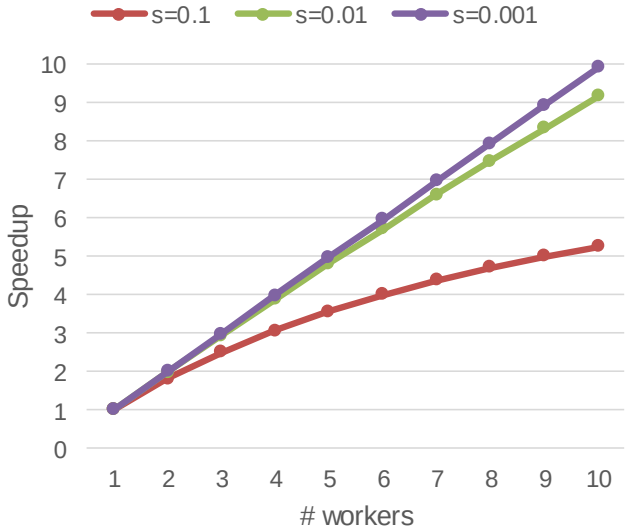

Parallel execution: $T(N) = s + \frac{p}{N}$

# Amdahl's Law (1967) – "Strong Scaling"

- Fixed workload speedup with $s$ being the fraction of nonparallelizable work

$$S(N) = \frac{T(1)}{T(N)} = \frac{1}{s + \frac{1-s}{N}}$$

- Parallel efficiency: $\varepsilon(N) = \frac{1}{s(N-1)+1}$




Gene M. Amdahl: *Validity of the single processor approach to achieving large scale computing capabilities*. In Proceedings of the April 18-20, 1967, spring joint computer conference (AFIPS '67 (Spring)). Association for Computing Machinery, New York, NY, USA, 483–485. DOI:10.1145/1465482.1465560

# Fundamental limits in Amdahl's Law

- Asymptotic speedup

$$\lim_{N \to \infty} S(N) = \frac{1}{s}$$

- Asymptotic parallel efficiency

$$\lim_{N \to \infty} \varepsilon(N) = 0$$

→ Asymptotically, nobody is doing anything except the worker that gets the serial work!

- In reality, it's even worse…

# Strong scaling plus overhead

- Let $c(N)$ be an overhead term that may include communication and/or synchronization

→ $T(N) = s + \frac{p}{N} + c(N)$

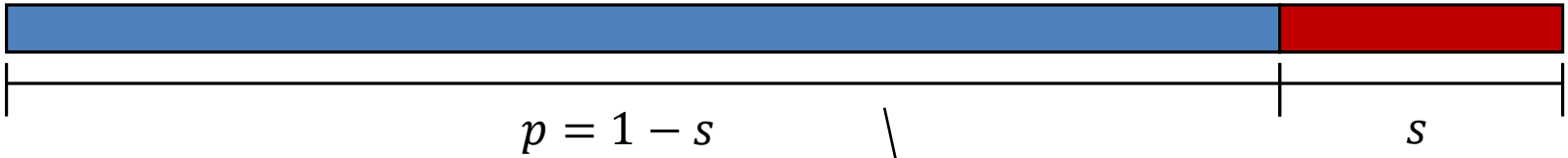
- What goes into $c(N)$?
  - Communication pattern
  - Synchronization strategy
  - Message sizes
  - Network structure
  - …

Typical examples: $c(N) =$

- $kN^2$ (all-to-all on bus network)
- $k \log N$ (optimal synchronization)
- $kN$ (one sends to all)
- $\lambda + kN^{-\frac{2}{3}}$ (Cartesian domain decomposition, nonblocking network)

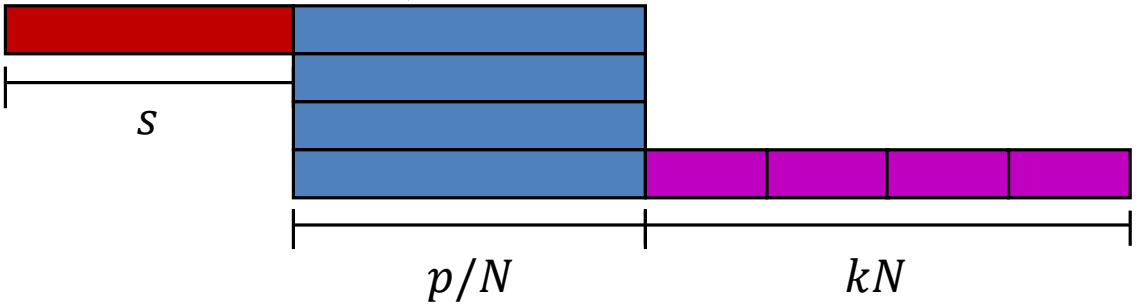
# Strong scaling with a particularly bad overhead model

Assume $c(N) = kN$ and $T(1) = s + p$, i.e., no communication with $N = 1$




Parallel execution:

$$T(N) = s + \frac{p}{N} + kN$$



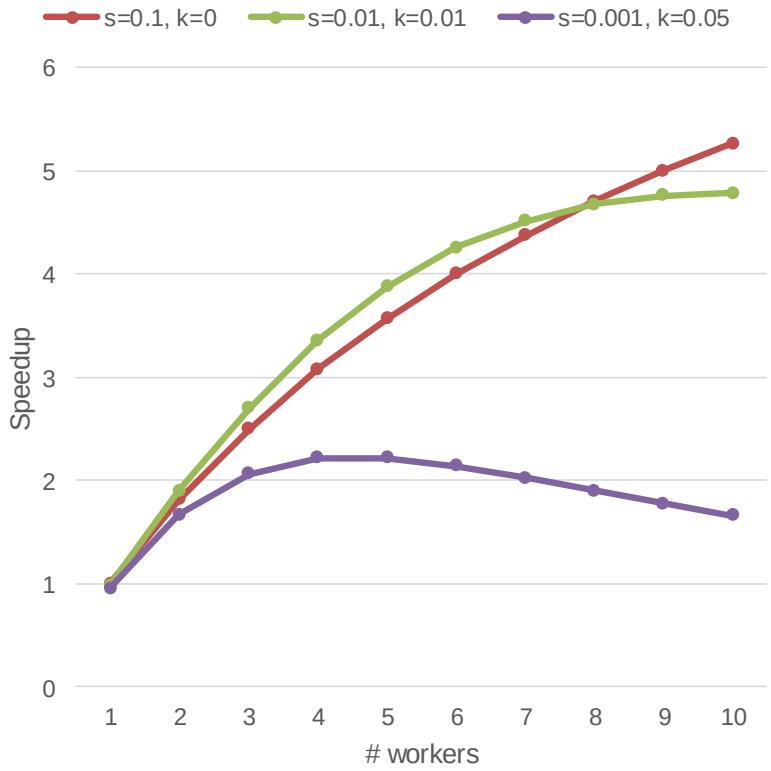
# Strong scaling with linear overhead

- Linear overhead is hazardous

$$S(N) = \frac{T(1)}{T(N)} = \frac{1}{s + \frac{1-s}{N} + kN}$$

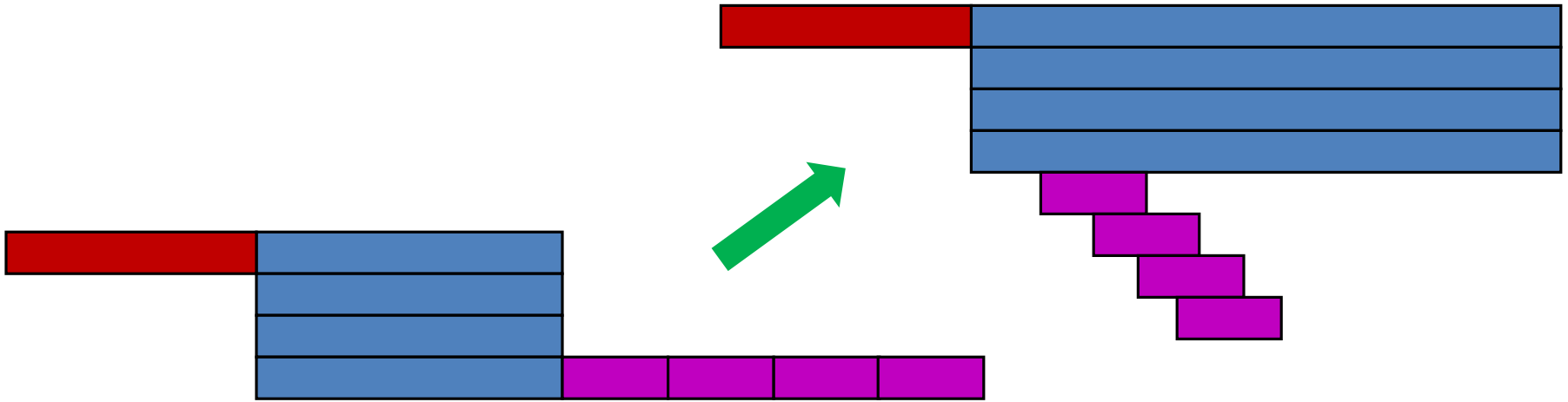
- Large-$N$ behavior

$$\rightarrow S_k(N) \xrightarrow{N \gg 1} \frac{1}{Nk}$$



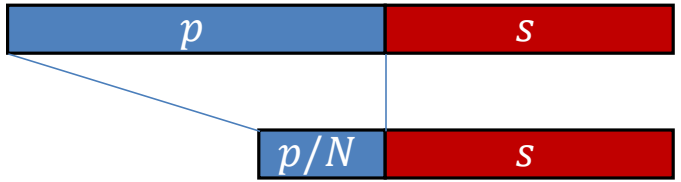
# So, all is lost? Not quite!

- Communication is not necessarily serial and/or non-overlapping
  - Nonblocking networks can transfer many messages concurrently
  - Communication may be overlapped with useful work for some algorithms
- Increasing the amount of parallel work can mitigate the impact of the serial work

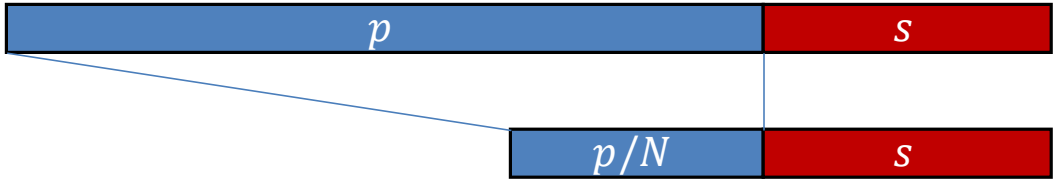


# A simple speedup model for scaled workload

- What if we could increase the parallel part of the work only?
  → the larger $p$, the larger the speedup




- This is not possible for all applications, but for some




- "Weak scaling"

# A simple speedup model for scaled workload

- Parallel workload grows linearly with $N$

  → $T(N) = s + \frac{pN}{N} = s + p$, i.e., runtime stays constant

- Scalability metric?
  → How much more work per second can be done with $N$ workers than with one worker?

$$S(N) = \frac{(s + pN)/(s + p)}{(s + p)/(s + p)} = s + (1 - s)N$$

Gustafson's Law ("weak scaling")

John L. Gustafson: *Reevaluating Amdahl's law*. Commun. ACM 31, 5 (May 1988), 532–533. DOI:10.1145/42411.42415

# Gustafson's Law for weak scaling

- Linear speedup (but not proportional unless $s = 0$) with $N$:

$$S(N) = s + (1 - s)N \quad \rightarrow \text{ unbounded speedup!}$$

- Weak scaling is the solution to the Amdahl dilemma: Why should we build massively parallel systems if all parallelism is limited by the serial fraction?

- Extension to communication?

$$\rightarrow T(N) = s + \frac{pN}{N} + c(N) = 1 + c(N)$$

$$\rightarrow S(N) = \frac{(s+pN)/(1+c(N))}{(s+p)/1} = \frac{s+(1-s)N}{1+c(N)}$$

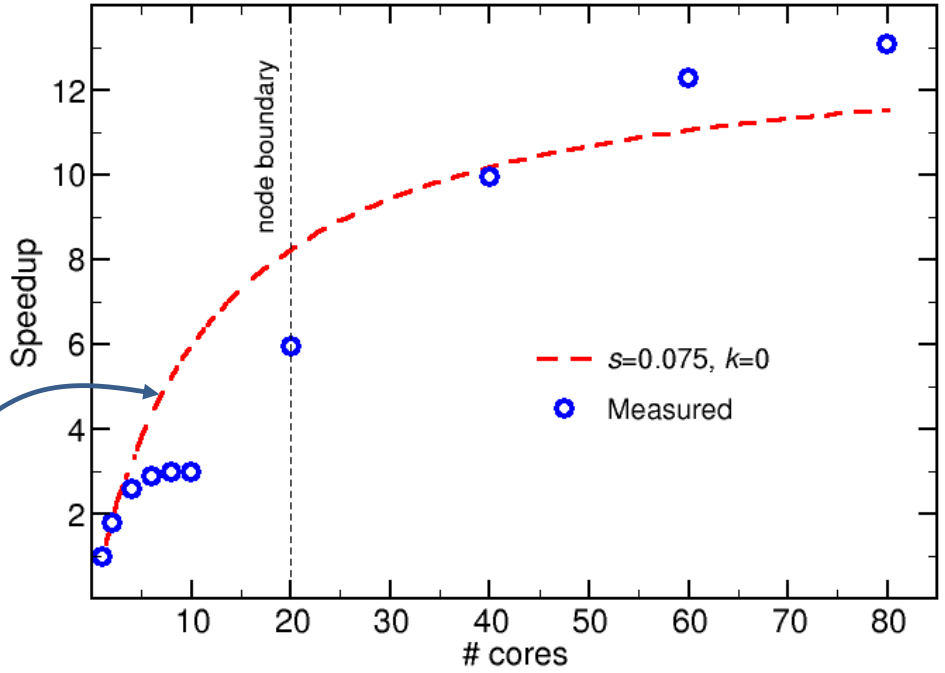
Much more relaxed conditions on $c(N)$

# How can we determine the model parameters?

- Manual analysis: Requires in-depth knowledge of hardware and program
- Curve fitting: Less insight, but also less cumbersome

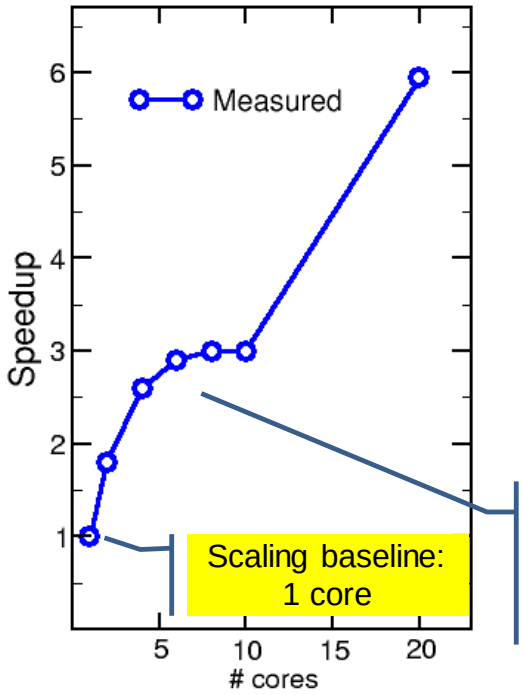
- Example: Strong scaling of hypothetical code on “Meggie” node @FAU (10 cores per socket, 2 sockets per node)
- Use “extended Amdahl’s” with $kN$ overhead

- Result:
  Best fit is not a good fit at all



# Separation of scaling baselines is key!

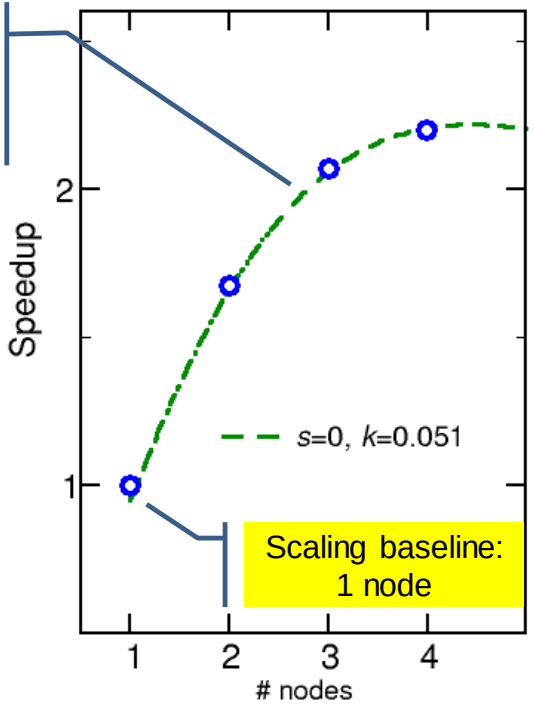
- Intra-socket scaling is not covered by the model
  - Model assumes “scalable resources”




Socket saturation due to memory bandwidth

Separating scaling baselines is important in modeling!

Model well suited for internode scaling!



Erlangen Regional Computing Center

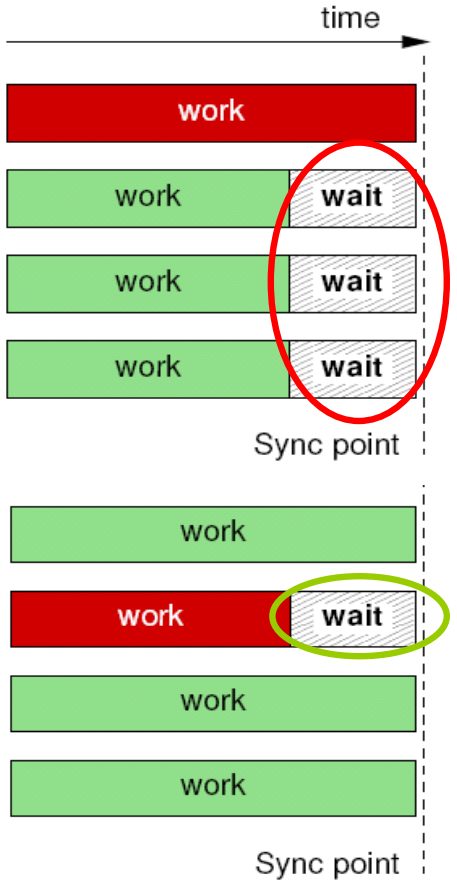
FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

# Scalability limitations beyond Amdahl's with communication

High Performance Computing

# Amdahl generalized: load imbalance

- Load imbalance at sync points
  - More specifically, execution time imbalance
  - $p/N$ assumption no longer valid in general
- Hard to model in general, but two corner cases:
  - A few "laggers" waste lots of resources
    - Single lagger → Amdahl's Law
  - A few "speeders" might be harmless
- Tuning advice
  - Avoid sync points
  - Turn laggers into speeders

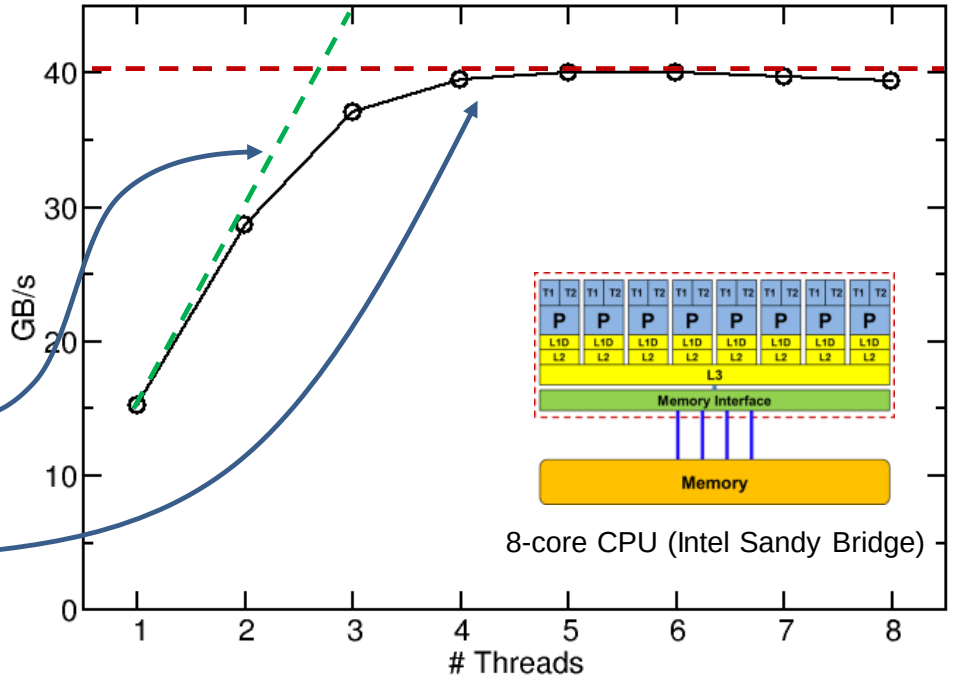


# Resource bottlenecks

- Amdahl's Law assumes perfect scalability of resources
- Reality: Computer architecture is plagued by bottlenecks!
- Example: array update loop

```
#pragma omp parallel for
for(i=0; i<10000000; ++i)
  a[i] = a[i] + s * c[i];
```

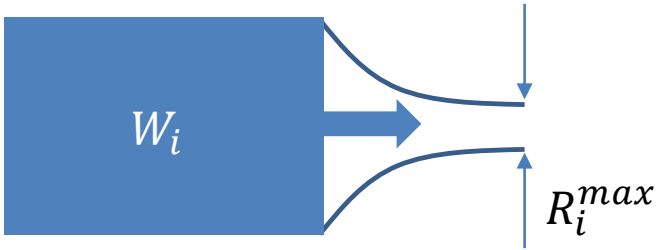
- Amdahl's: $s = 0, c(N) = 0$
  - Perfect speedup? No!
  - Saturation because of memory bandwidth exhaustion



# A more general view on resource bottlenecks

- What is the maximum performance when limited by a bottleneck?
- Resource bottleneck $i$ delivers resources at maximum rate $R_i^{max}$
- $W_i$ = needed amount of resources

- Minimum runtime: $T_i = \frac{W_i}{R_i^{max}}$



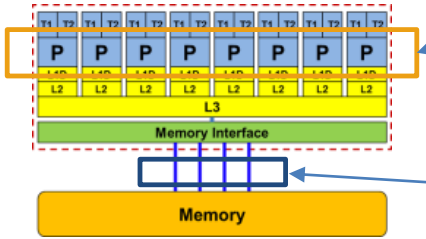

- Multiple bottlenecks → multiple minimum runtimes: $T_{\min} = f(T_1, \ldots T_n)$
- Overall performance:

$$P_{\max} = \frac{W}{T_{\min}}$$

# A bottleneck model of computing

- Example: two bottlenecks

```
#pragma omp parallel for
for(i=0; i<10^7; ++i)
  a[i] = a[i] + s * c[i];
```




$R_{flops}^{max} = 192 \frac{\text{Gflops}}{\text{s}}$

$R_{BW}^{max} = 40 \frac{\text{Gbyte}}{\text{s}}$

8-core CPU
(3 GHz Intel Sandy Bridge)

$W_{flops} = 2 \times 10^7 \text{ flops}$

$W_{BW} = 3 \times 8 \times 10^7 \text{ bytes}$

$$T_{flops} = \frac{2 \times 10^7 \text{ flops}}{192 \frac{\text{Gflops}}{\text{s}}} = 104 \, \mu s$$

$$T_{BW} = \frac{2.4 \times 10^8 \text{ bytes}}{40 \frac{\text{Gbyte}}{\text{s}}} = 6.0 \text{ ms}$$

# An *optimistic* bottleneck model

- How do we reconcile the multiple bottlenecks?
  I.e., what is the functional form of $f(T_1, \dots T_n)$?

  → optimistic model (full overlap): $f(T_1, \dots T_n) = \max(T_1, \dots T_n)$

- Application to example: $T_{\min} = \max(T_{flops}, T_{BW}) = 6\,\text{ms}$

- Maximum performance ("light speed"): $P_{\max} = \frac{2\times10^7}{6.0\times10^{-3}} \frac{\text{flops}}{s} = 3.3\,\text{Gflop/s}$

This is called the Roofline model. See also **https://youtu.be/IrkNZG8MJ64**

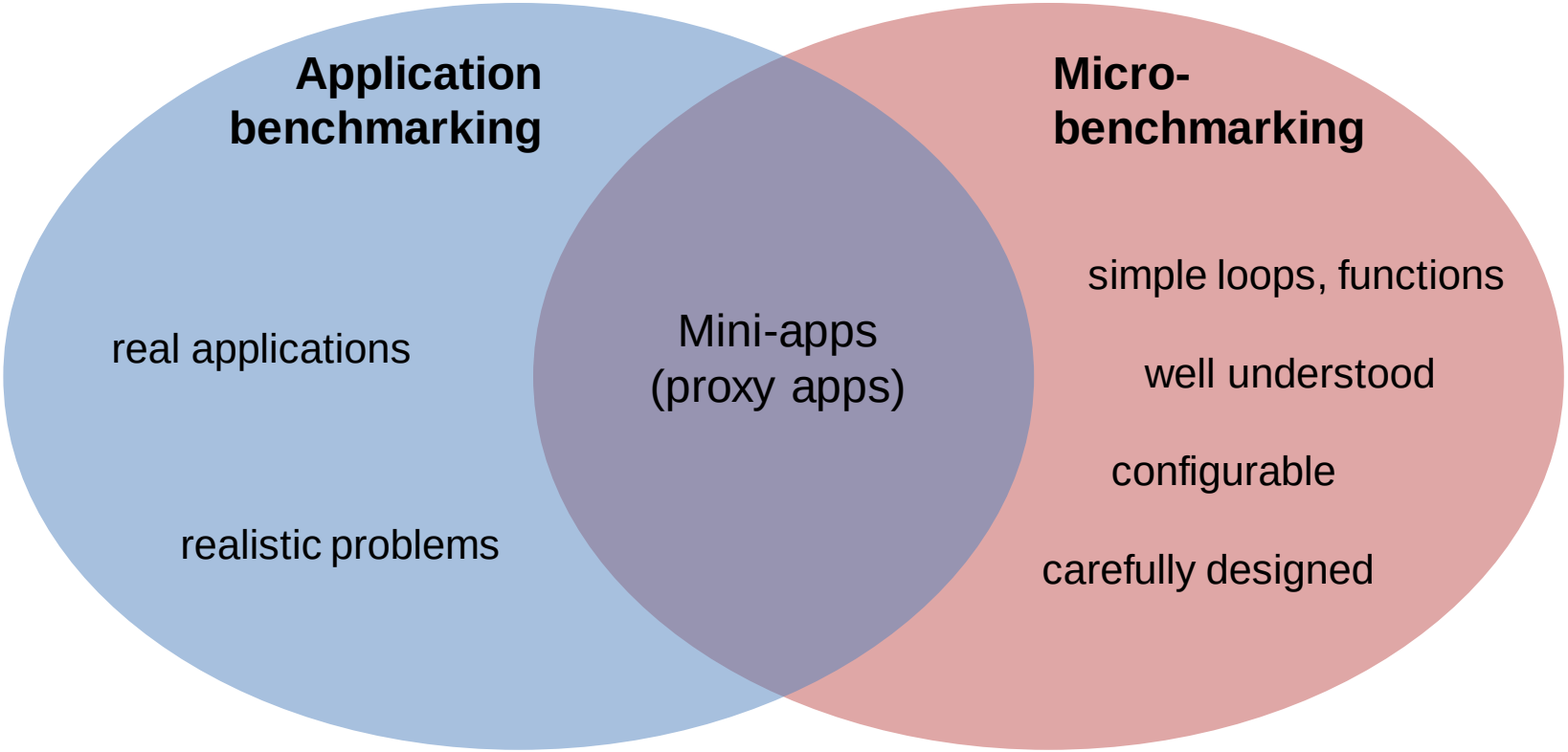
Erlangen Regional Computing Center

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

# Benchmarking: Measuring and presenting performance

High Performance Computing

# Benchmarking: two kinds (and a half)



# Proper definition of benchmark cases

Benchmarking is a vital part of development and performance analysis

1. Define proper benchmark case(s) (input data sets)
   - Reflect(s) “production” workload
   - Tolerable runtime (minutes at most)
2. Document system settings and execution environment
   - Software: compilers, compiler options, library versions, OS version, …
   - Hardware: CPU type, network, […many more …]
   - Runtime options: Threads/processes per node, affinity, large pages, […many more …]
3. Document measurement methodology
   - Number of repetitions, statistical variations, …

# Performance and time

- Performance is a "higher is better" metric: $P(N) = S(N) \times P(1)$
  - How much work can be done per time unit?
- Work: flops, iterations, "the problem," …
- Time: wall-clock time

- Measuring performance:

```
double s = get_walltime();
// do your work here
double e = get_walltime();
double p = work/(e-s);
```

- Caveat:
  Timer resolution is finite!

Return time stamp

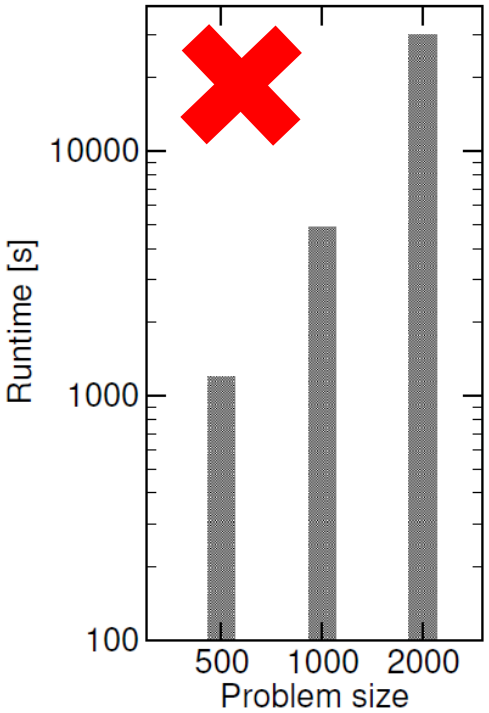
```
#if !defined(_POSIX_C_SOURCE)
#define _POSIX_C_SOURCE 199309L
#endif

#include <time.h>

double get_walltime() {
  struct timespec ts;
  clock_gettime(CLOCK_MONOTONIC, &ts);
   return (double)ts.tv_sec +
          (double)ts.tv_nsec * 1.e-9;
}
```

For Fortran

```
double get_walltime_() {
  return get_walltime();
}
```

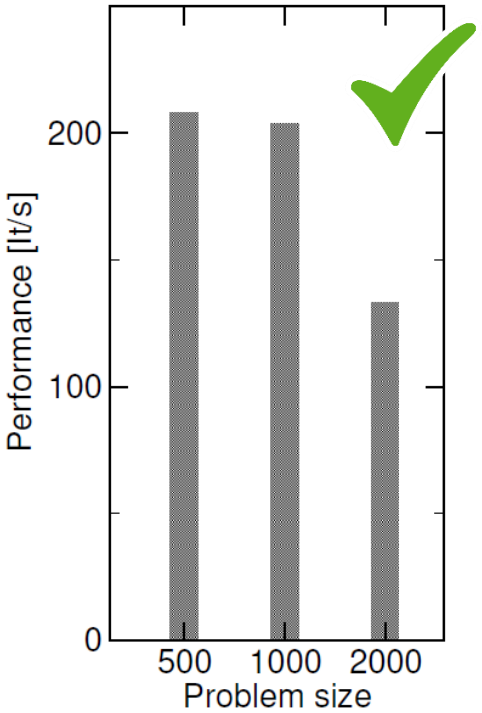
# Popular blunders: runtime != performance

- Just presenting runtime is almost always a bad idea!



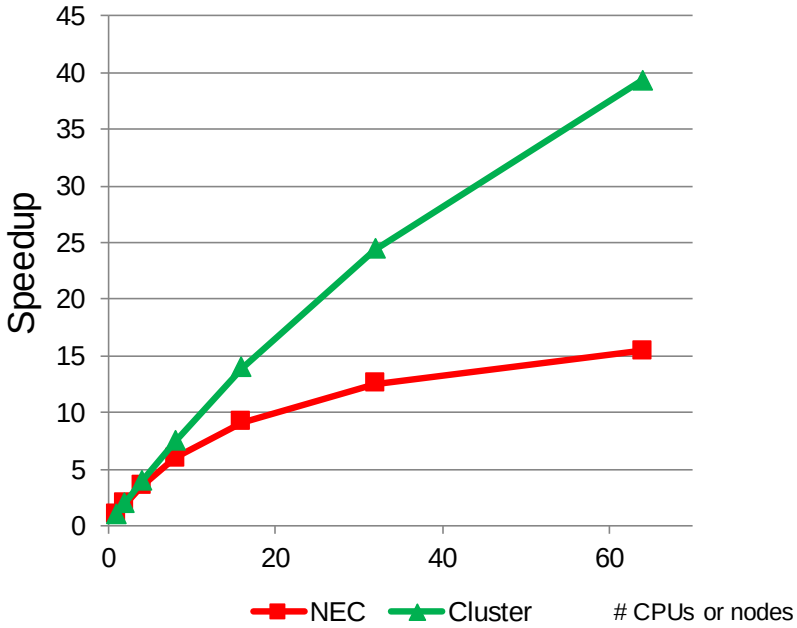

Insights hidden by trivial dependency: “larger problems need more time”

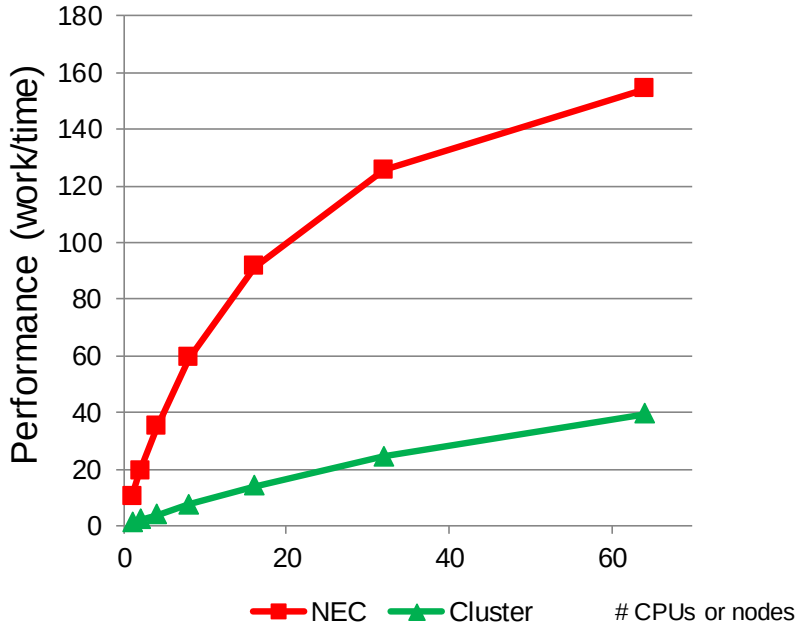
Performance metric reveals interesting behavior worth investigating!



# Popular blunders: speedup != performance

Speedup hides the “higher is better” quality when comparing different systems or cases



# Popular blunders: TMI bombs are no good

- Show only the data that is required to drive your point home
- You can always put the rest into an online repository (good thing!)

# More about how not to do it

Fooling the masses with performance results on parallel computers

https://blogs.fau.de/hager/archives/category/fooling-the-masses

https://blogs.fau.de/hager/files/2018/08/FTM-GridKa18-c.pdf

Erlangen Regional Computing Center

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

# Finding parallelism and mapping it to the hardware

High Performance Computing

# Finding parallelism

- … may be simple or might be a challenge.
  Example: summing up many numbers

$$\sum = s_1 + s_2 + s_3 + s_4 + s_5 + s_6 + \cdots + s_{999999} + s_{1000000}$$

$$\sum = ((\ldots((((s_1 + s_2) + s_3) + s_4) + s_5) + s_6) + \cdots + s_{999999}) + s_{1000000})$$

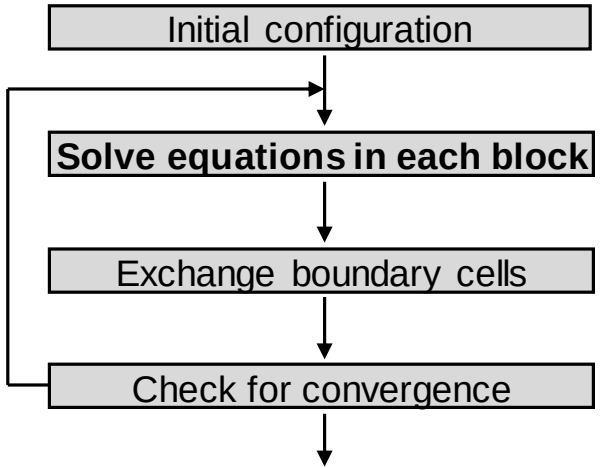
Sequential summation

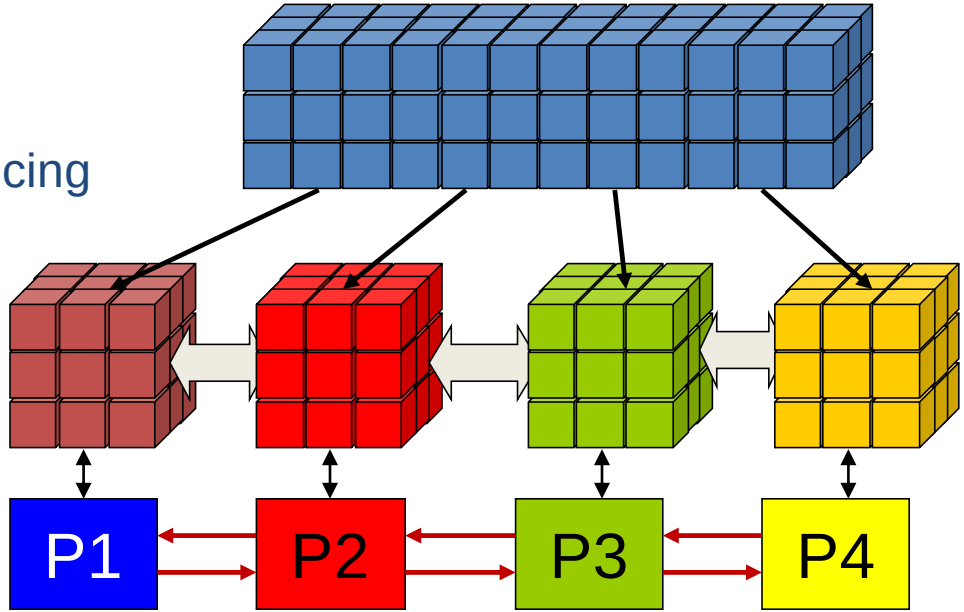
$$\sum = ((s_1 + s_2) + (s_3 + s_4)) + ((s_5 + s_6) + \cdots) + \cdots + (s_{999999} + s_{1000000}))$$

(Stepwise) parallel summation

# Finding parallelism: data parallelism on coarse level

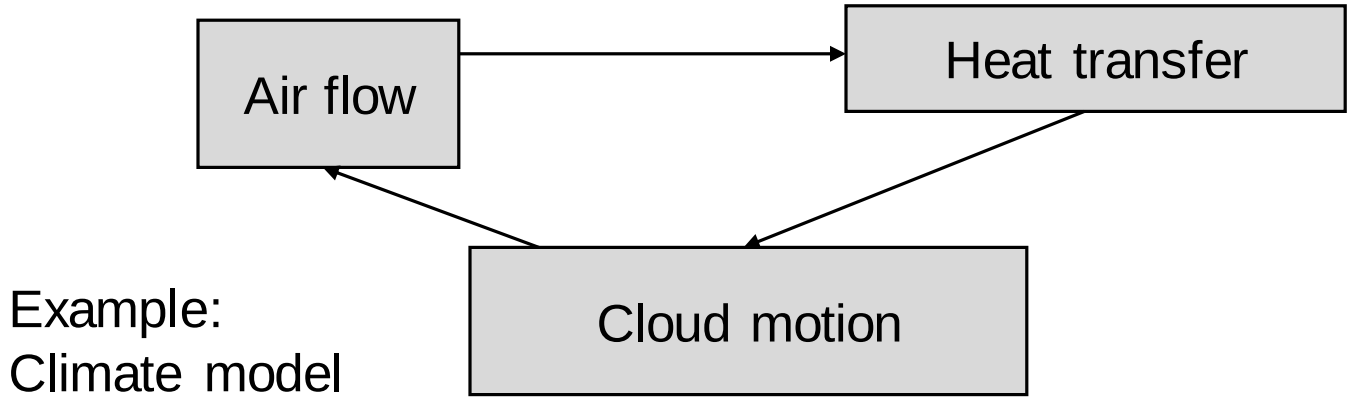
- Example: domain decomposition (e.g., in Computational Fluid Dynamics)
  - Mapping of 3D mesh to processes/threads
    - Cartesian/unstructured grid
    - Next-neighbor communication by message passing
  - Simple communication, load balancing



# Finding parallelism: functional parallelism on coarse level

- Example: functional decomposition (e.g., multi-physics codes)
  - Different functional units of a program are mapped to different processors
  - Every sub-task is different from the others and has different communication requirements
- Problem: load balancing

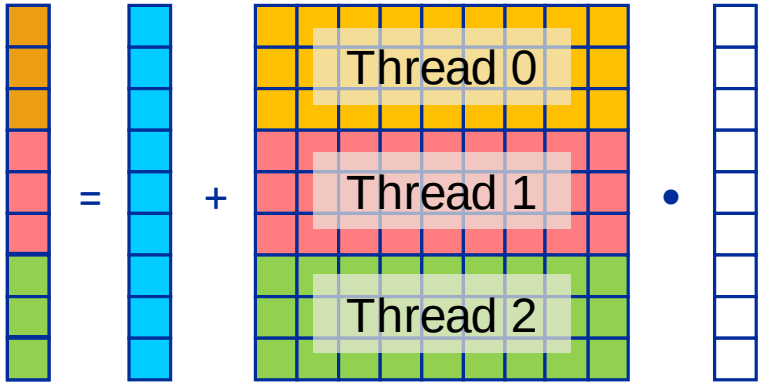



Example:
Climate model

# Finding parallelism: data parallelism on intermediate level

- Example: work sharing in shared memory via threading
  - Here: matrix-vector multiplication (dense MVM)

```
#pragma omp parallel for
for(int r=0; r<rows; ++r)
  for(int c=0; c<cols; ++c)
    y[r] += m[r][c] * x[c];
```




  - Execute a complete kernel (“solver”) on multiple threads, share data
  - “Loop parallelism”
- Programming techniques
  - OpenMP threading, or any other threading model (e.g., POSIX threads)
  - Auto-parallelizing compilers (don’t hold your breath)

# Finding parallelism: instruction and data parallelism on fine level

- Instruction-level parallelism exploits concurrency in instruction streams
- Example: dense MVM

```
for(int r=0; r<rows; ++r)
  for(int c=0; c<cols; ++c)
    y[r] += m[r][c] * x[c];
```

  - 2 loads + 1 FMA in inner loop
- Pipelining & superscalarity
  - Mostly automatic, done by hardware
  - Compiler can help

- SIMD parallelism exploits parallel data processing by instruction
- Example: dense MVM

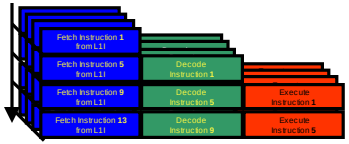
```
for(int r=0; r<rows; ++r) {
  y0 = y1 = 0.;
  for(int c=0; c<cols; c+=2) {
    y0 += m[r][c]   * x[c];
    y1 += m[r][c+1] * x[c+1];
  }
  y[r] += y0 + y1;
}
```

- Done by compiler or programmer
- Target: inner loops

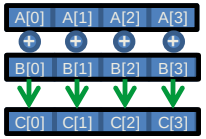
# Levels of parallelism in large parallel systems

fine

**Core**
ILP, pipelining, SIMD

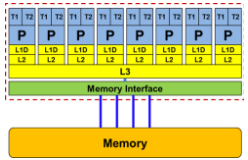


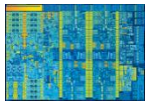




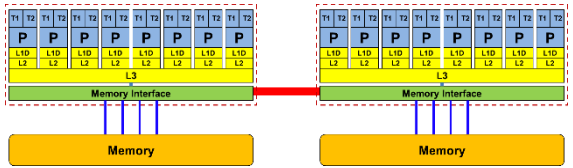

intermediate

**Chip**
cores



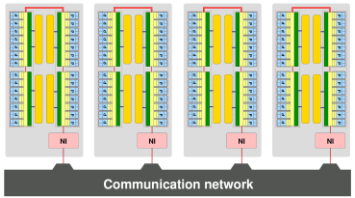



**Node**
chips, sockets, accelerators




coarse

**Cluster**
nodes, network




# Take-home messages

- The available parallelism is usually limited
  - Serial fraction, communication
- If you do it right, unlimited parallelism might be an option
  - Weak scaling, favorable communication
- Hardware bottlenecks are ubiquitous but constitute well-defined upper performance limits
  - A back-of-the-envelope calculation is better than nothing
- Measuring and presenting performance data is ridden with pitfalls
- Know your hardware and the parallelism it provides to your application