Friedrich-Alexander-Universität
Erlangen-Nürnberg

Introduction to Parallel Programming with MPI

Dr. Alireza Ghasemi, Dr. Georg Hager

Erlangen National High Performance Computing Center

Odds and Ends – what we have left out

What we have left out

- Point-to-point bells and whistles
  - Persistent communication
  - Message probing: MPI_Probe,…
  - One-sided communication: MPI_Put, MPI_Get, MPI_Accumulate,…
  - Partitioned communication
- Collectives bells and whistles
  - MPI_Reduce_scatter, MPI_Scan,…
- MPI I/O
- Virtual topologies
- MPI shared memory communication

Introduction to Parallel Programming with MPI

Dr. Alireza Ghasemi, Dr. Georg Hager

Erlangen National High Performance Computing Center

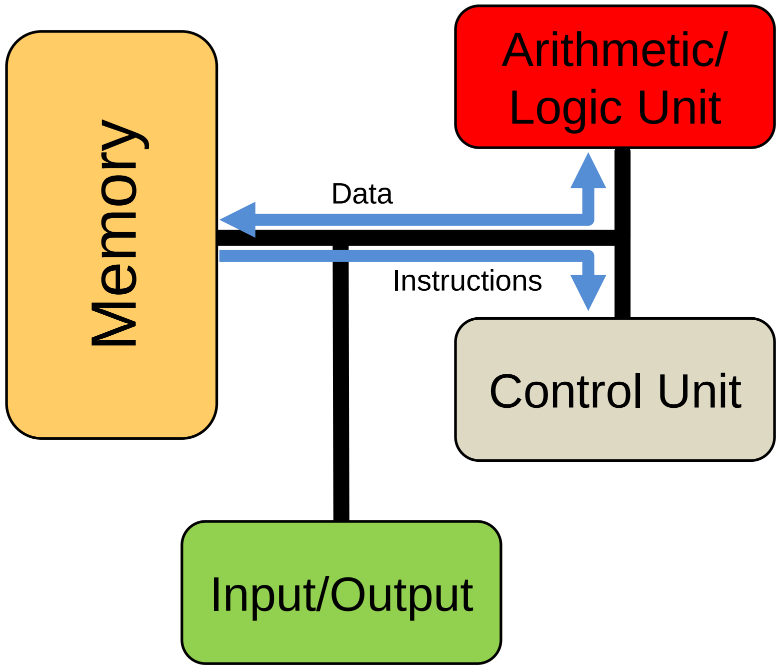
Computer Architecture and Performance issues
In MPI programming

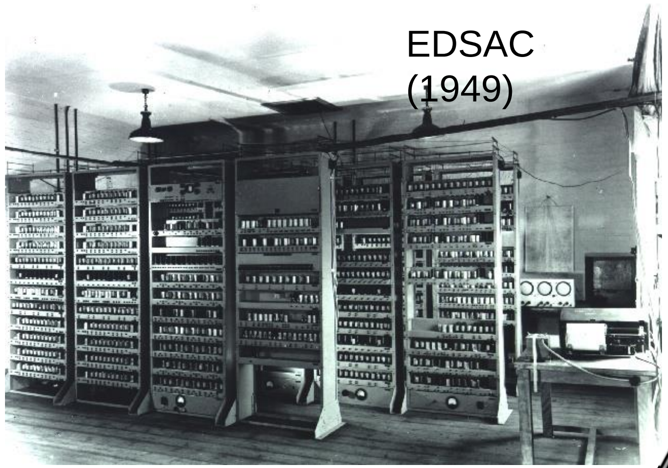
Performance issues – overview

- Basics of parallel computer architecture
- Affinity and pinning
- Simple scaling laws
- Benchmarking and performance assessment
- Tracing tools

Basics of parallel computer architecture

At the core: the stored-program computer

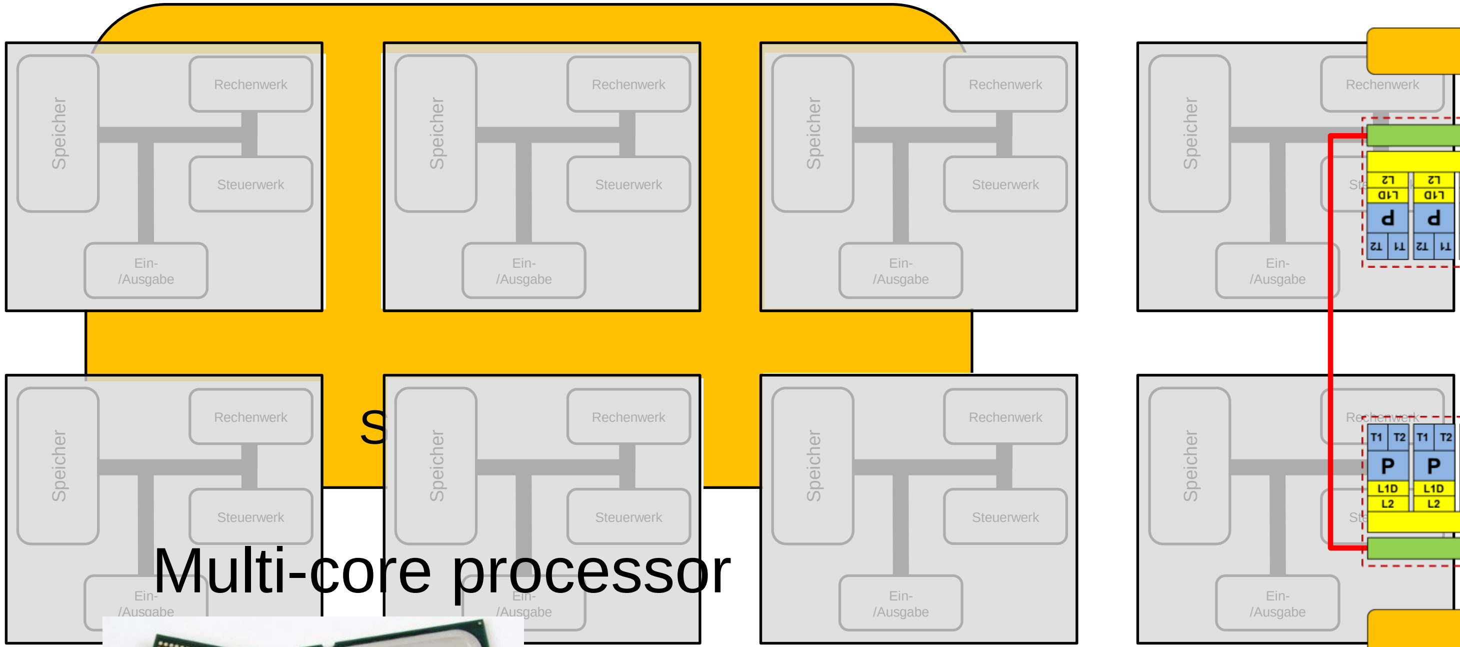




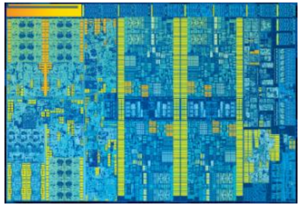
CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=432935

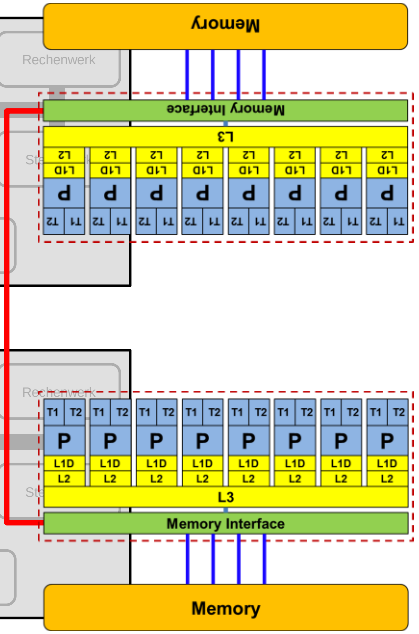
By Rafael Fernandez - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=51218006

Shared memory: a single cache-coherent address space



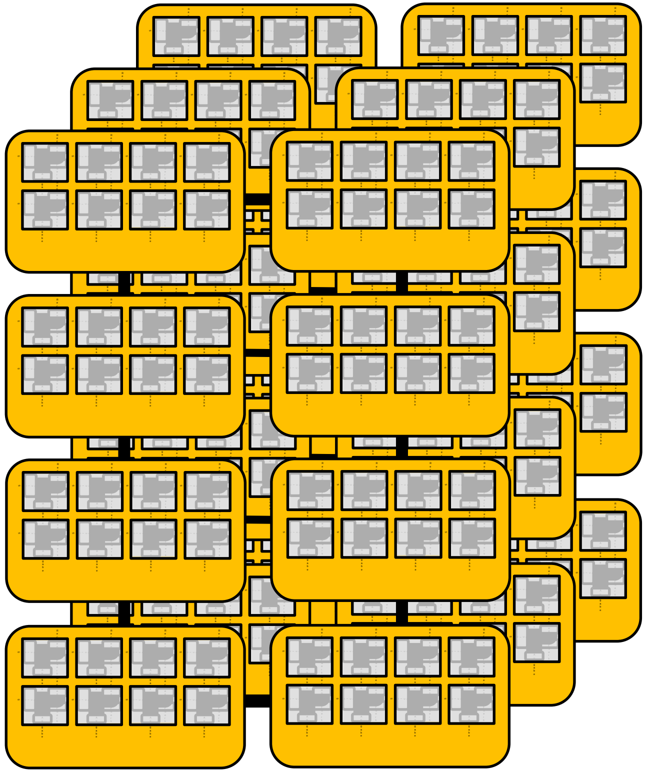
Multi-core processor





Multiple CPU chips per node

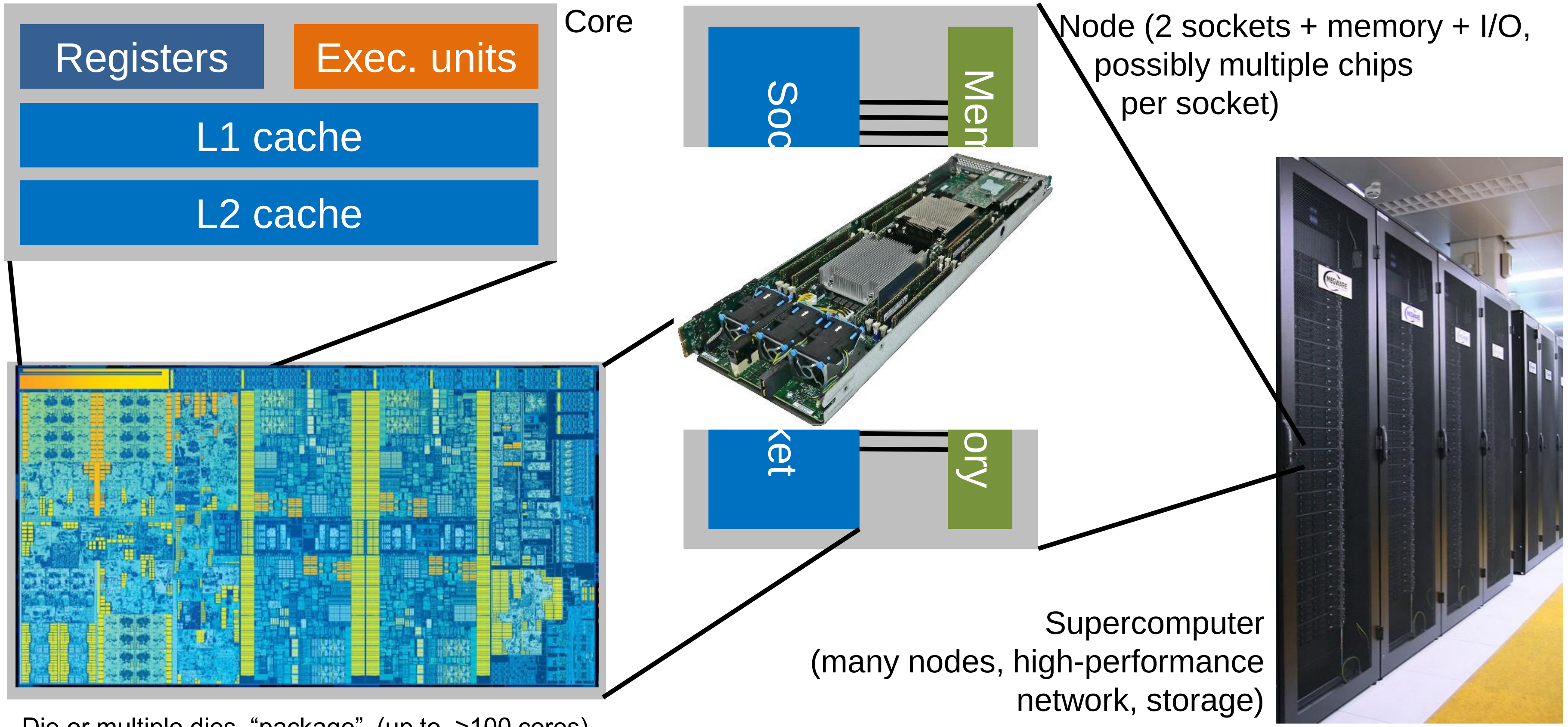
Distributed memory: no cache-coherent single address space



Cluster/
supercomputer

Modern supercomputers are shared-/distributed-memory hybrids

Parallelism in modern computers



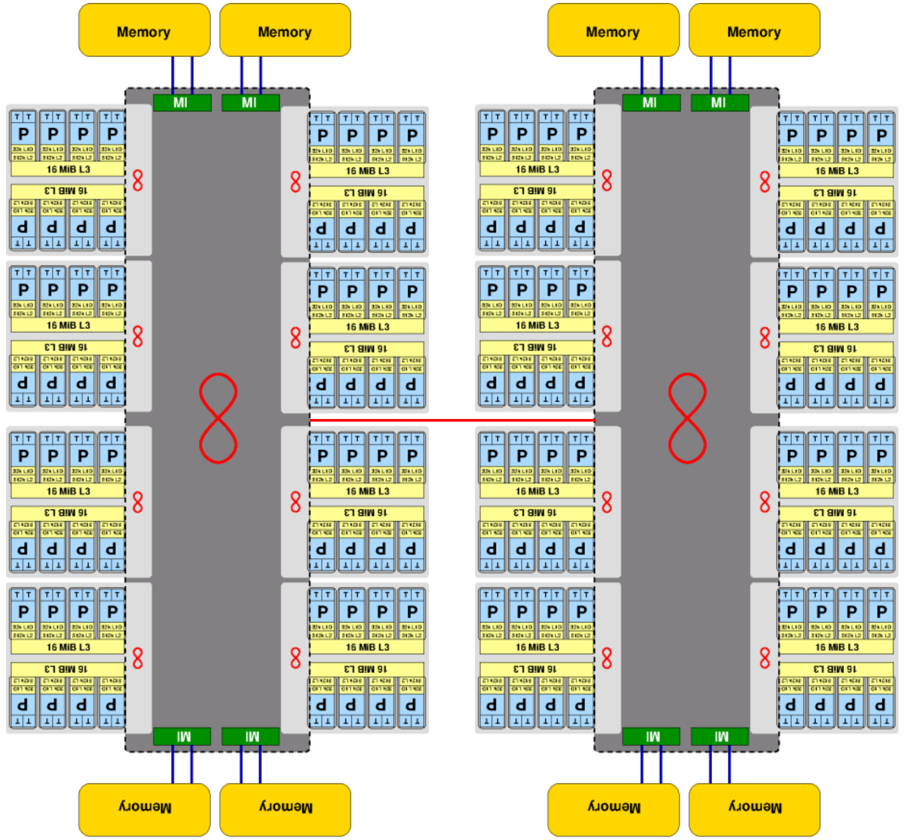
Core

Node (2 sockets + memory + I/O, possibly multiple chips per socket)

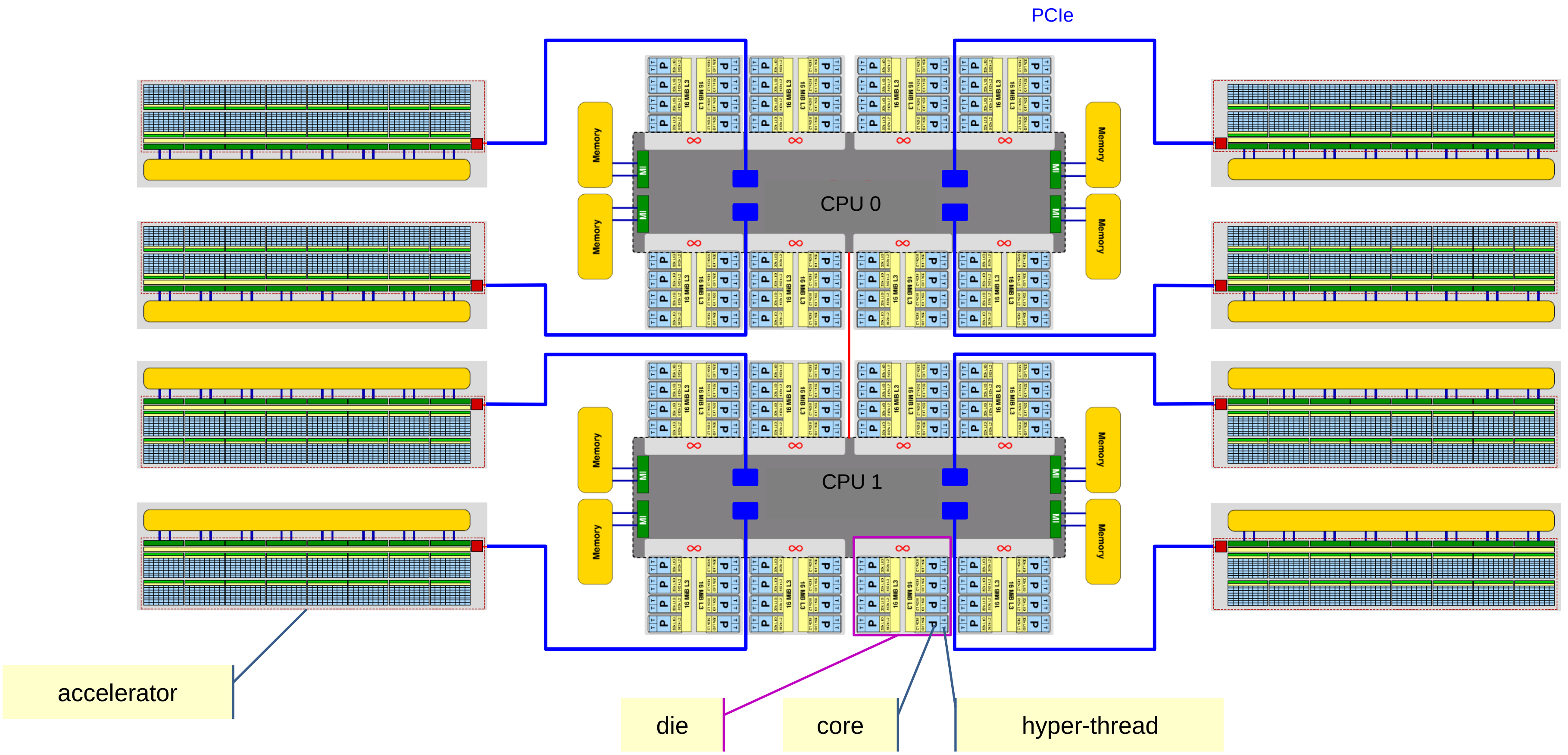
Supercomputer (many nodes, high-performance network, storage)

Die or multiple dies, “package” (up to >100 cores)

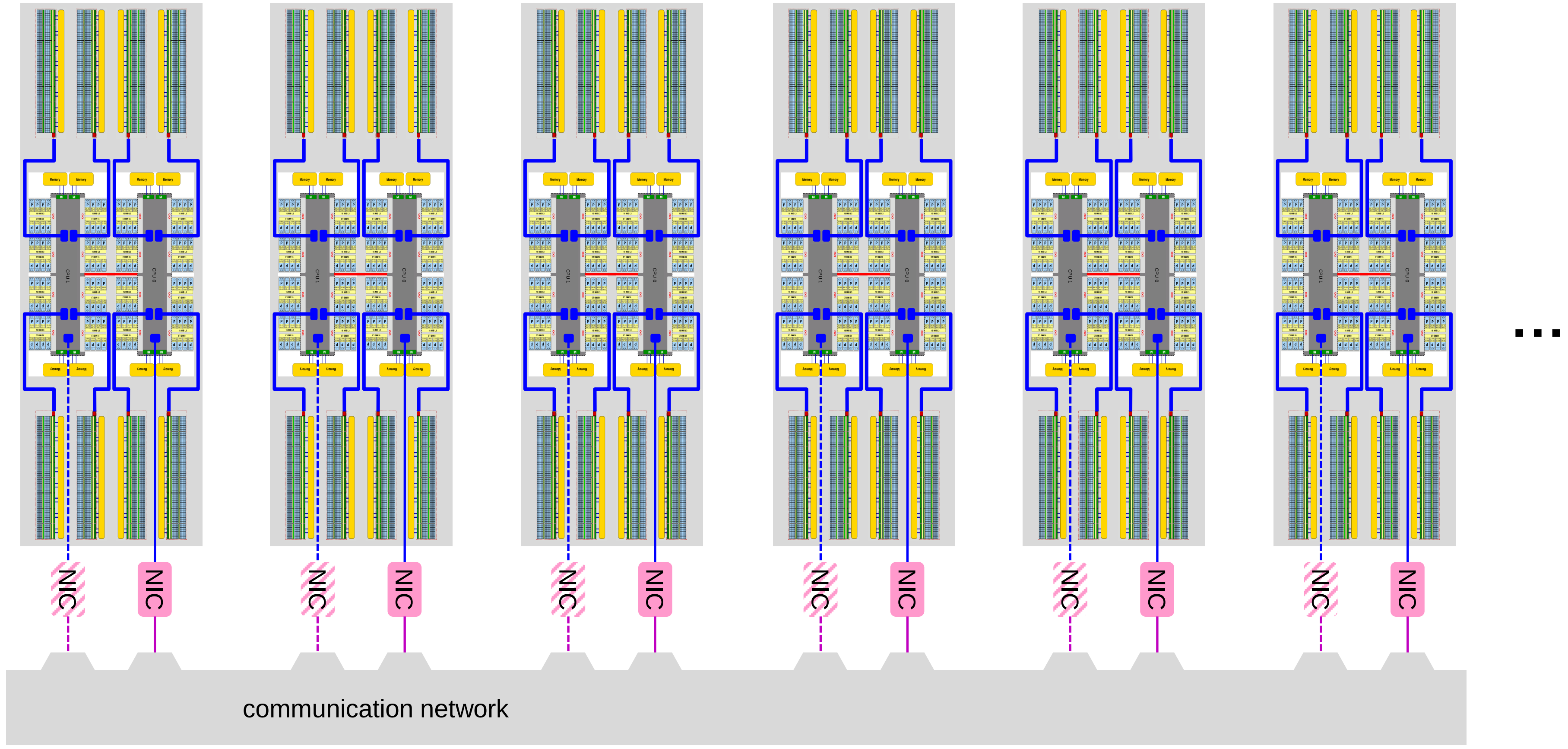
A modern CPU compute node (AMD Zen2 “Rome”)



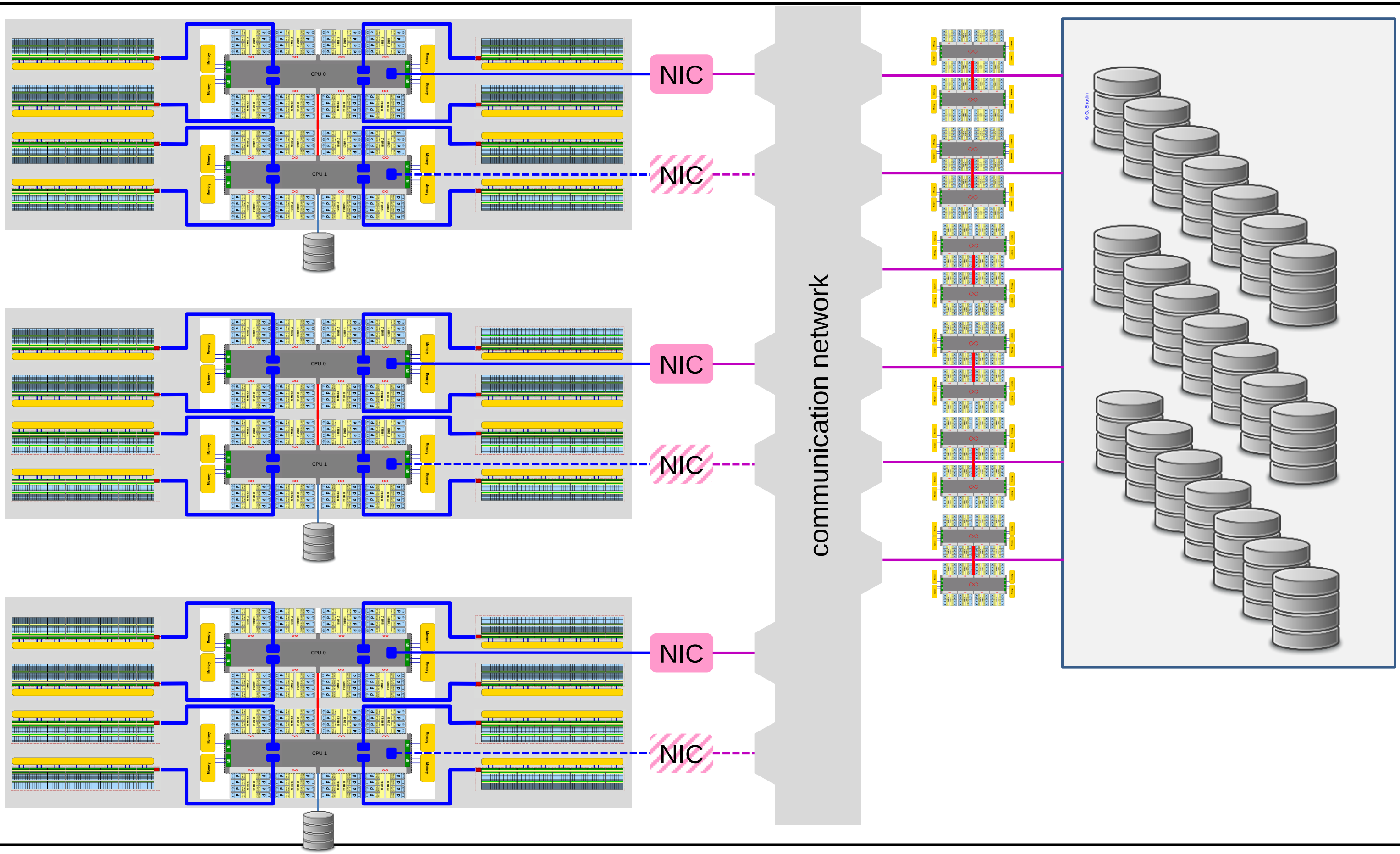
Adding accelerators to the node



Turning it into a cluster



Adding permanent storage



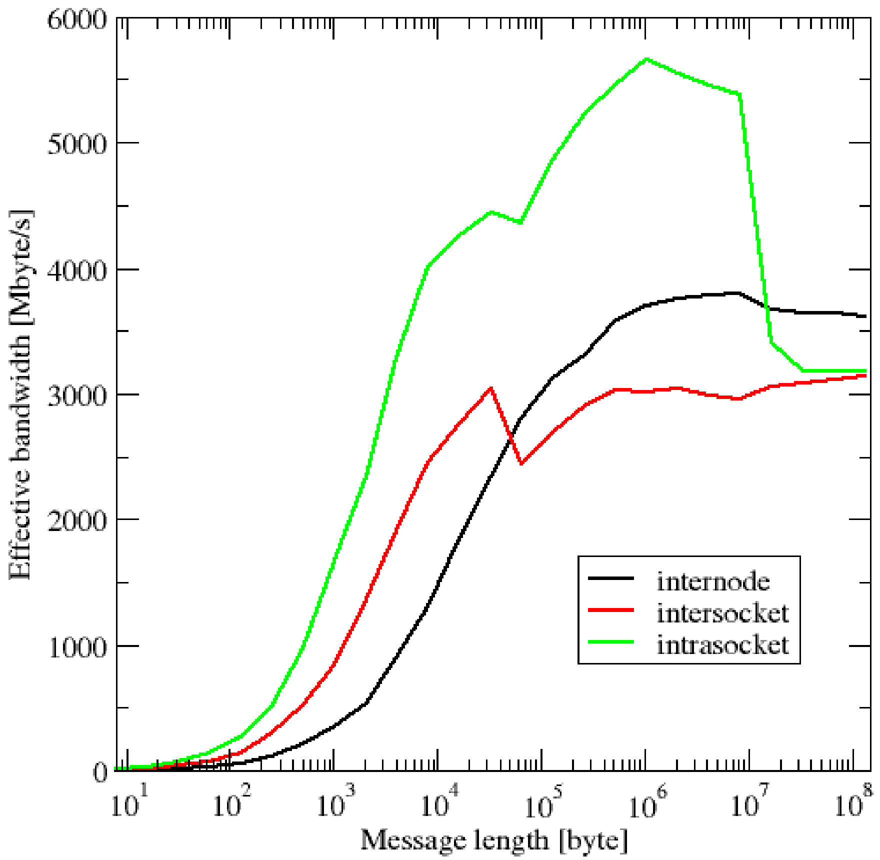
Point-to-point data transmission performance

- Simple "Hockney model" for data transfer time

$$T_{comm} = \lambda + \frac{V}{b}, \quad B_{\mathrm{eff}} = \frac{V}{T_{comm}}$$

λ: latency, b: asymptotic BW

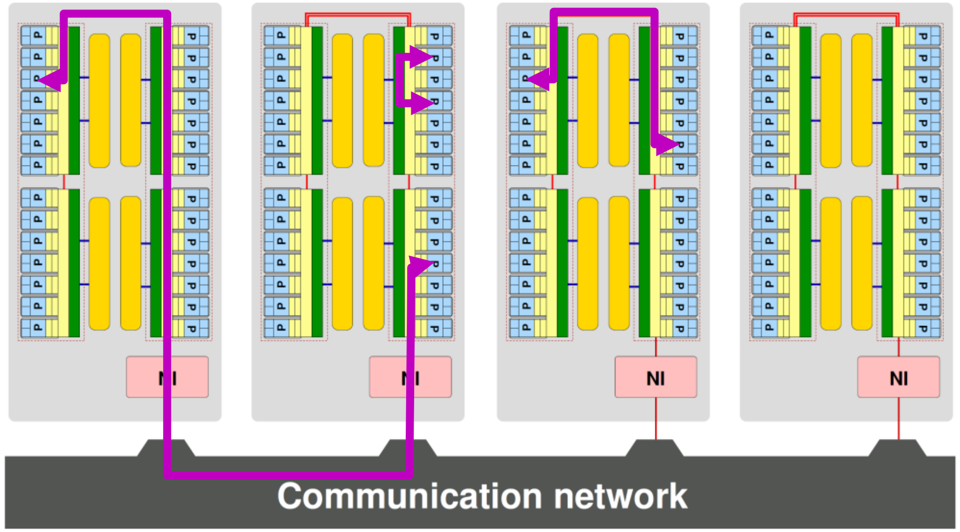
- Reality is more complicated
  - System topology
  - Caching effects
  - Contention effects
  - Protocol switches
  - Collective communication



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

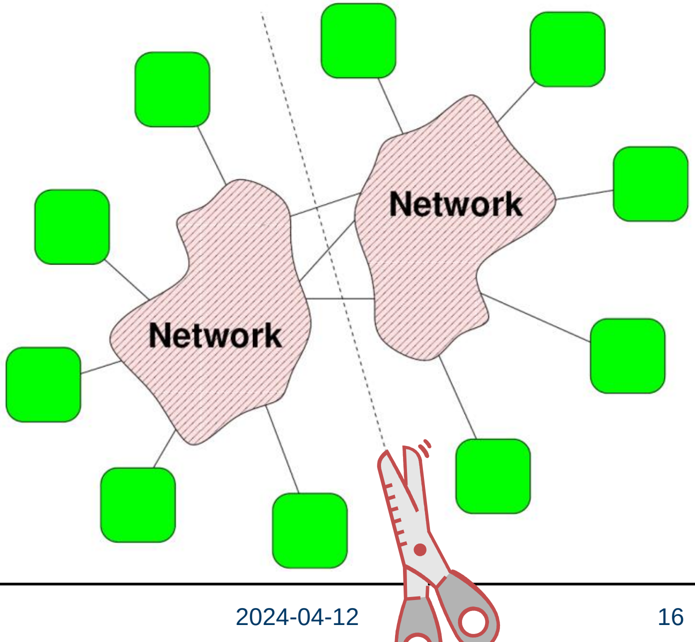
Characterizing communication networks

- Network bisection bandwidth B_b is a general metric for the data transfer "capability" of a system:

  Minimum sum of the bandwidths of all connections cut when splitting the system into two equal parts

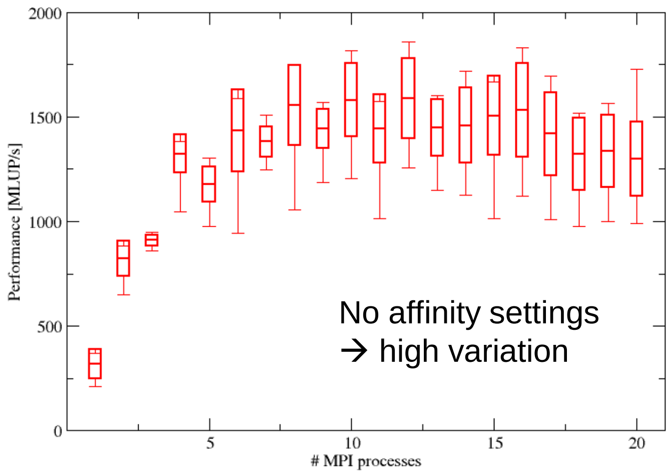
- More meaningful metric for system scalability: bisection BW per node: B_b / N_{nodes}

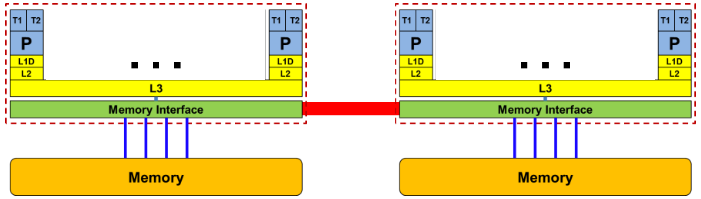
- Bisection BW depends on
  - Bandwidth per link
  - Network topology



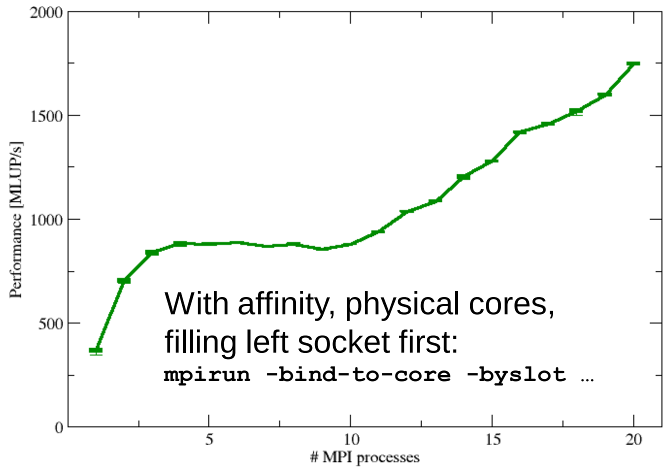
Affinity control (pinning) of processes

Anarchy vs. affinity with a heat equation solver





2x 10-core Intel Ivy Bridge, OpenMPI



```
mpirun -bind-to-core -byslot …
```

Reasons for caring about affinity:

- Eliminating performance variation
- Making use of architectural features
- Avoiding resource contention

Pinning of MPI processes

- Highly implementation and system dependent!
- Intel MPI: env variable **`I_MPI_PIN_PROCESSOR_LIST`** (MPI only) or **`I_MPI_PIN_DOMAIN`** (MPI+OpenMP)
- OpenMPI: choose between several mpirun options, e.g.,
  -bind-to-core, -bind-to-socket, -bycore, -byslot …
- Cray's aprun
- Platform-independent tools: **`likwid-mpirun`** (**`likwid-pin`**, **`numactl`**)

Which way to pin

- MPI-only code: `I_MPI_PIN_PROCESSOR_LIST`
- Many options

- Straightforward use:

```
$ mpirun -genv I_MPI_PIN_PROCESSOR_LIST=0-71  -np 144 ./a.out
```

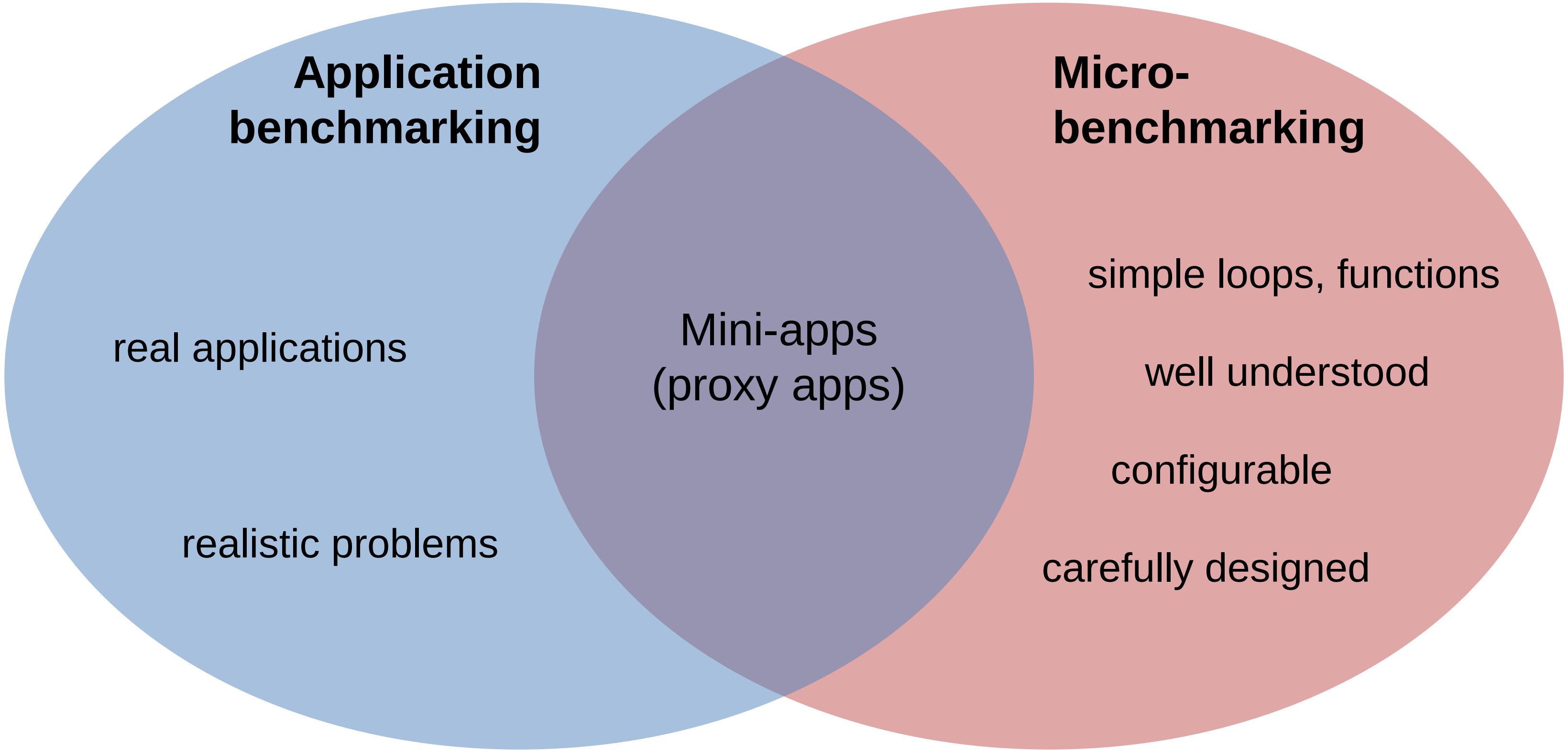
pins one process on each physical core

Benchmarking and performance assessment

More info:
Lecture “Experiments and Data Presentation in High Performance Computing”

https://youtu.be/y1n0IJZiPuw

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!

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

#include <time.h>
```

$\left.\text{Return time stamp}\right\}$

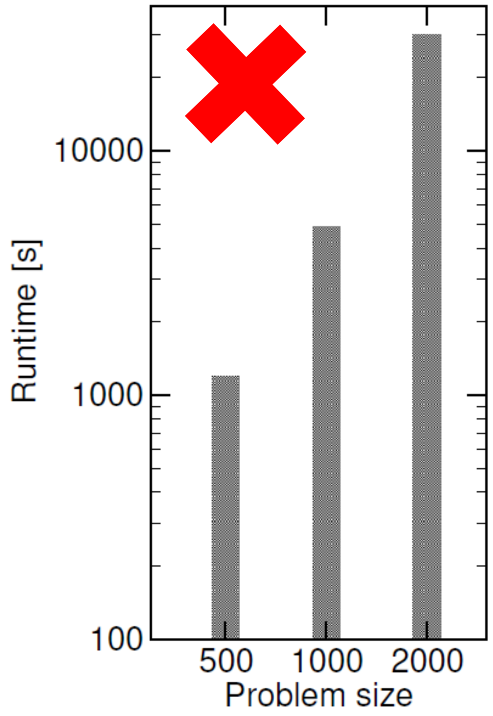
```
double get_walltime() {
  struct timespec ts;
  clock_gettime(CLOCK_MONOTONIC, &ts);
   return (double)ts.tv_sec +
          (double)ts.tv_nsec * 1.e-9;
}
```

$\left.\text{For Fortran}\right\}$

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

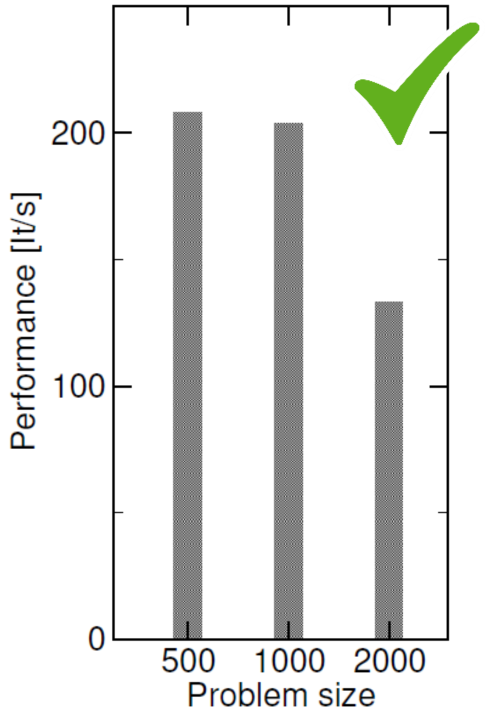
Popular blunders: runtime != performance

- Just presenting runtime is almost always a bad idea!



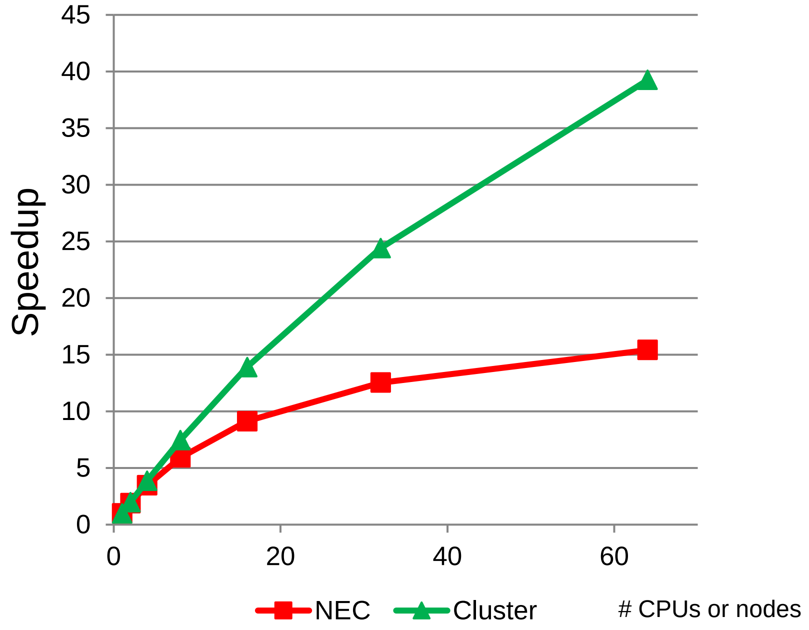
Insights hidden by trivial dependency: “larger problems need more time”

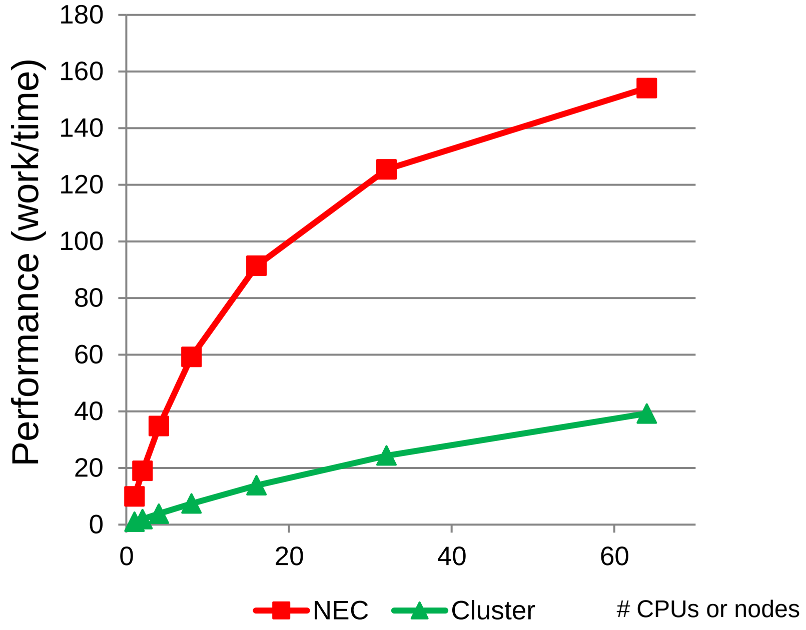
Performance metric reveals interesting behavior worth investigating!



Popular blunders: speedup != performance

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





Limits of parallelism: simple scaling laws

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?

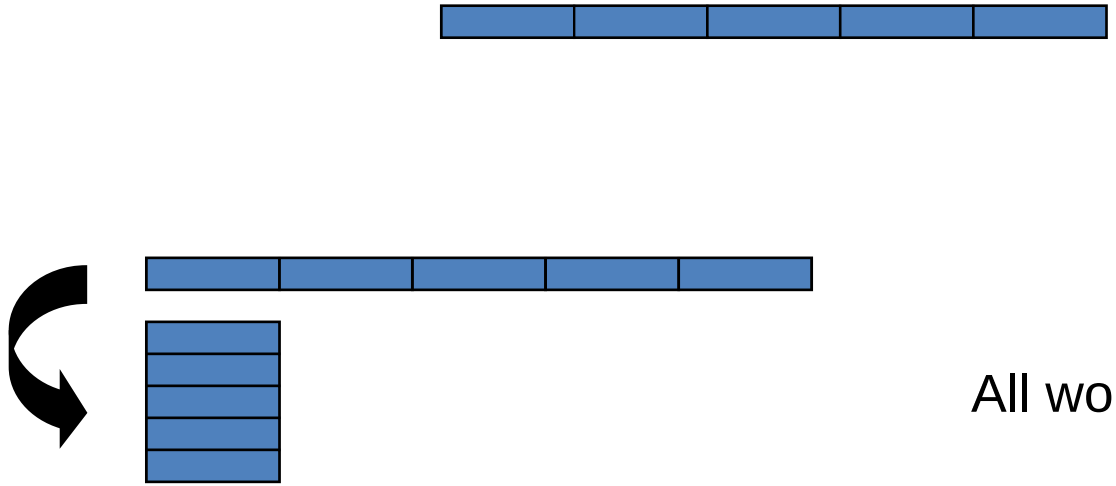
  → parallel efficiency: $\varepsilon(N) = \frac{S(N)}{N}$

- Warning: These metrics are not performance metrics!

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

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



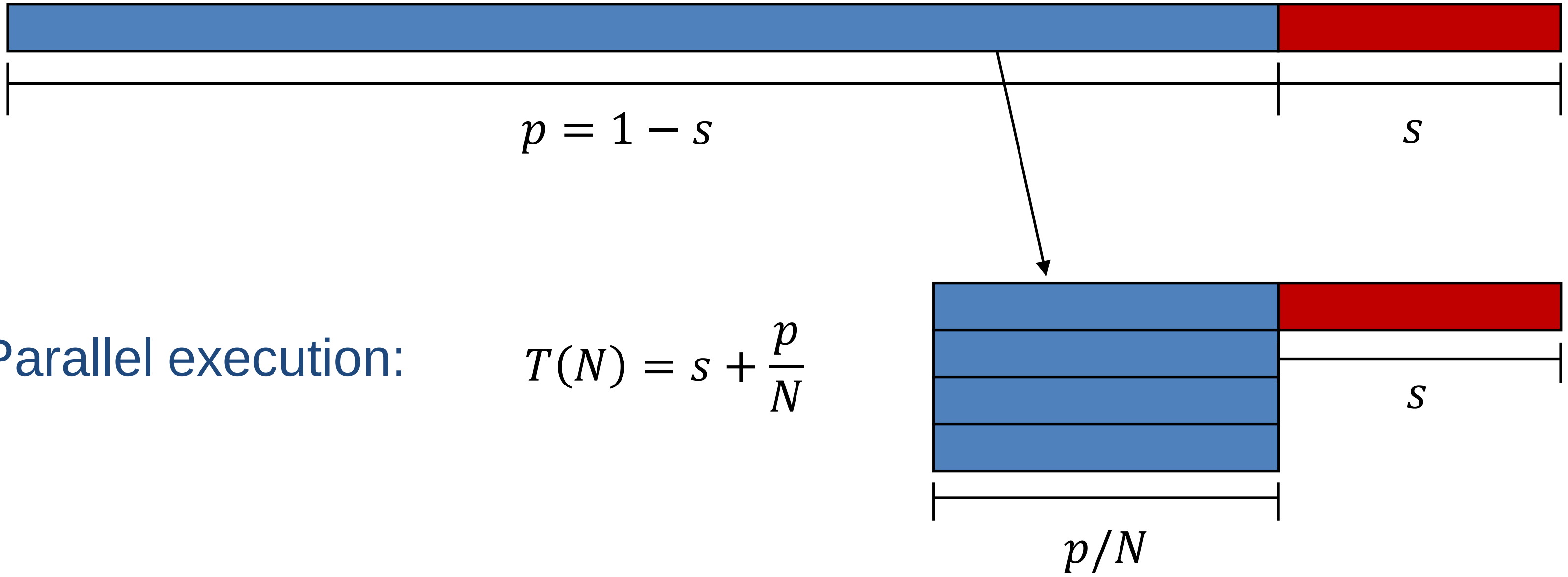
Ideal world:
All work is perfectly parallelizable
$S(N) = N, \qquad \varepsilon = 1$

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



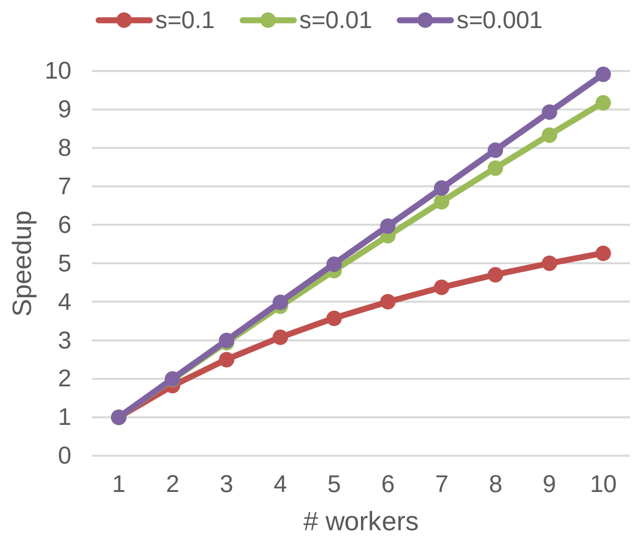
Parallel execution: $T(N) = s + \frac{p}{N}$

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

$$\rightarrow \quad T(N) = s + \frac{p}{N} + c(N)$$

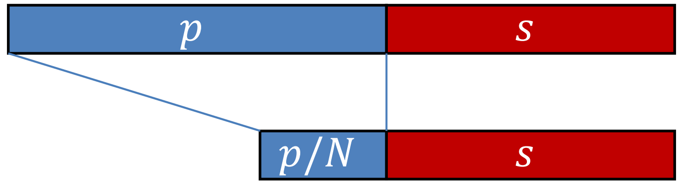
- What goes into $c(N)$?
  - Communication pattern
  - Synchronization strategy
  - Message sizes
  - Network structure
  - …

Typical examples: $c(N) =$

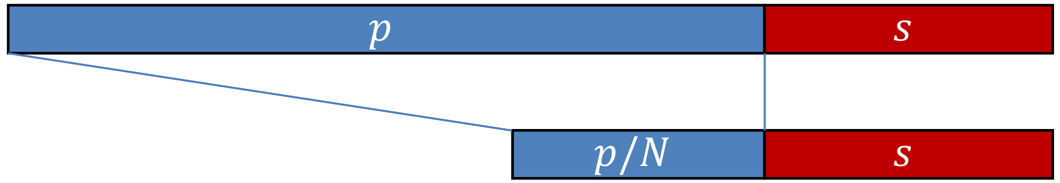
kN^2	(all-to-all on bus network)
$k \log N$	(optimal synchronization)
kN	(one sends to all)
$\lambda + kN^{-\frac{2}{3}}$	(Cartesian domain decomposition, nonblocking network)

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:

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

- 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?

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

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

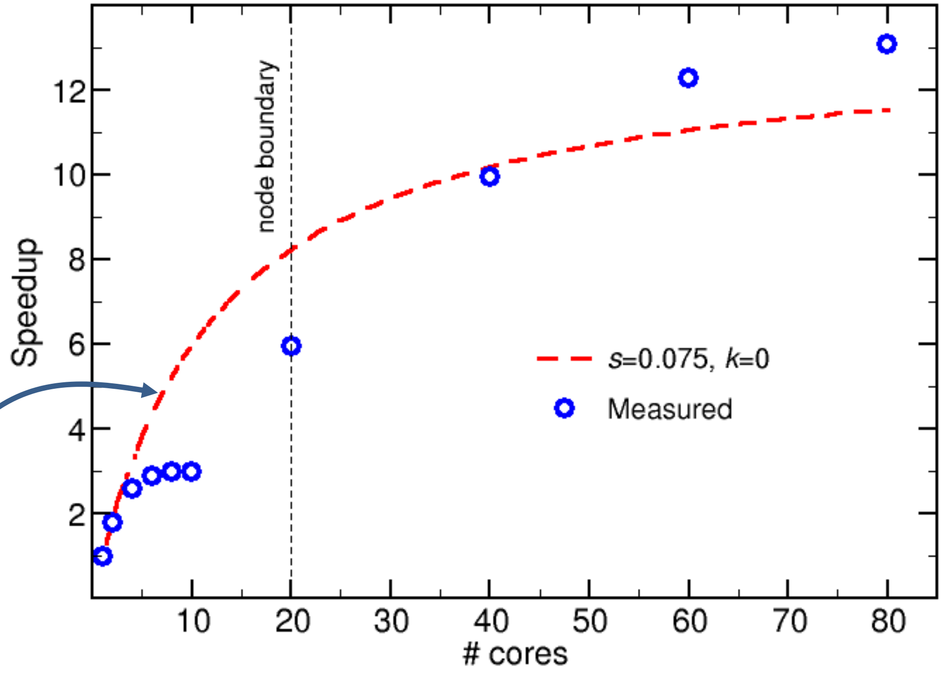
Much more relaxed conditions on $c(N)$

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

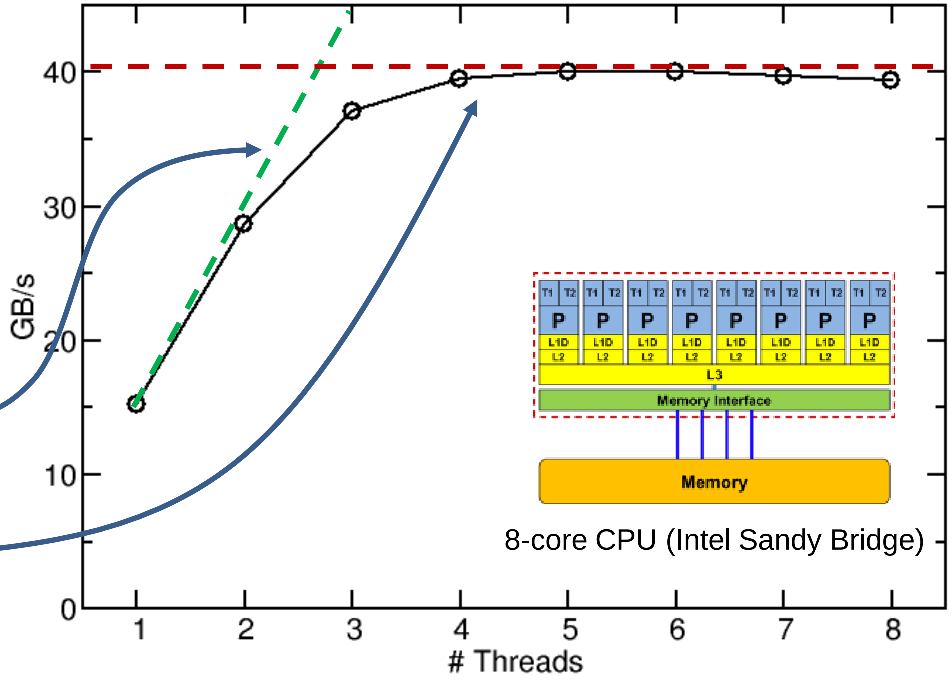


Resource bottlenecks

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

```
// MPI-parallel
for(i=0; i<10000000; ++i)
  a[i] = a[i] + s * c[i];
```

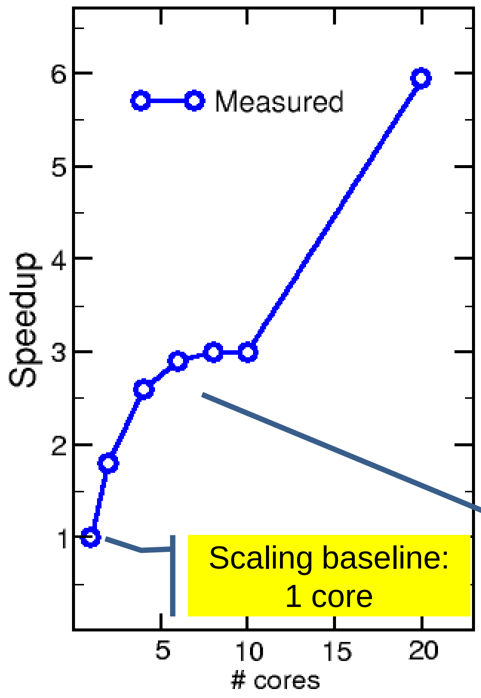
- Amdahl's: $s = 0,\ c(N) = 0$
  - Perfect speedup? No!
  - Saturation because of memory bandwidth exhaustion



8-core CPU (Intel Sandy Bridge)

Separation of scaling baselines is key!

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

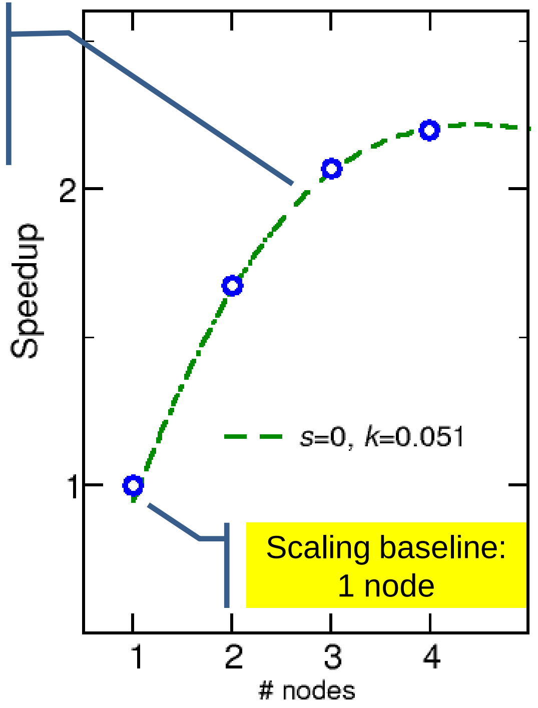


Scaling baseline: 1 core

Socket saturation due to memory bandwidth

Separating scaling baselines is important in modeling!

Model well suited for internode scaling!



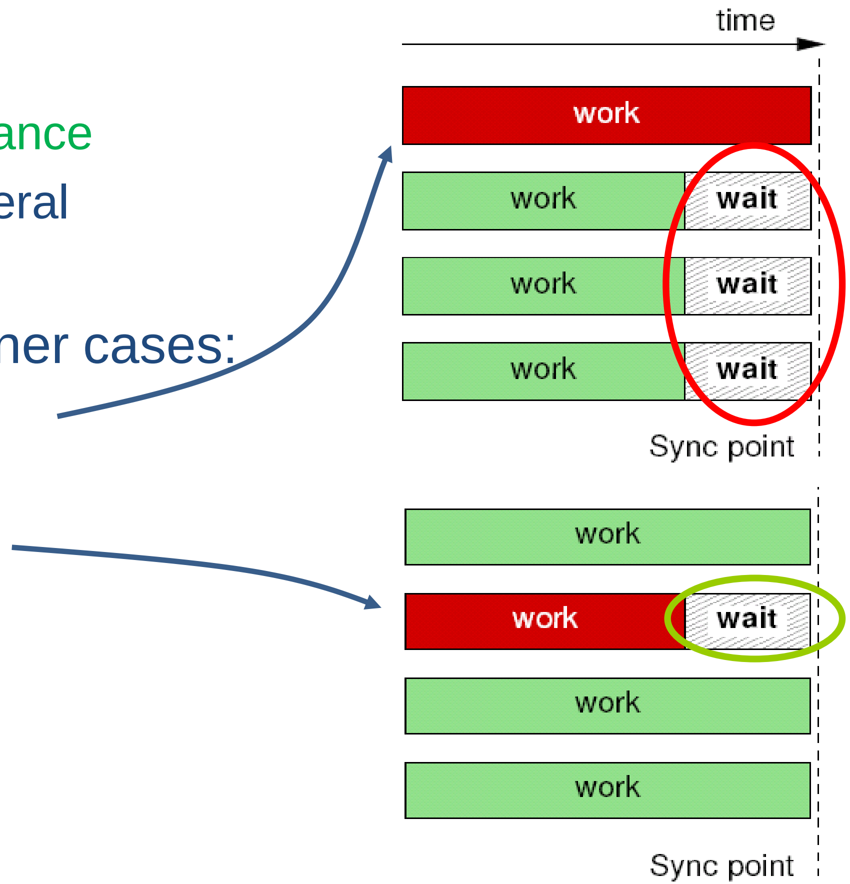
Scaling baseline: 1 node

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



MPI tracing tools

MPI tracing tools

- Allow the user to track events and statistics pertaining to MPI communication and code execution
- Popular tools
  - Intel Trace Analyzer and Collector (ITAC)
  - VAMPIR (commercial)
  - Paraver

- Powerful tools
- Potential to produce massive amounts of data
- Danger of “drowning in data”

Intel Trace Analyzer and Collector

Environment preparation

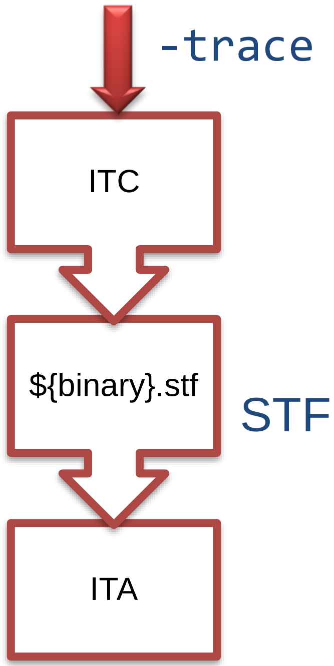
```
$ module load itac intel intelmpi
```

Collector: compilation, linking and run

```
$ mpiicc -trace myProg.c -o ${binary}
$ mpirun -n 4 ./${binary}
```

Analyzer

```
$ traceanalyzer ${binary}.stf &
```

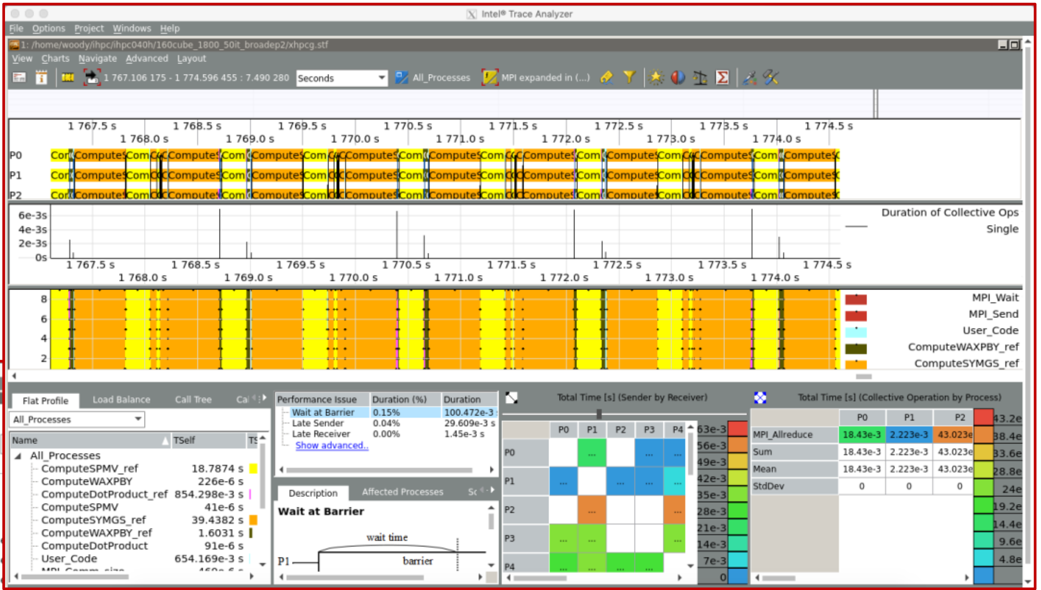


format

STF: Structured Trace File

Basic features of ITAC

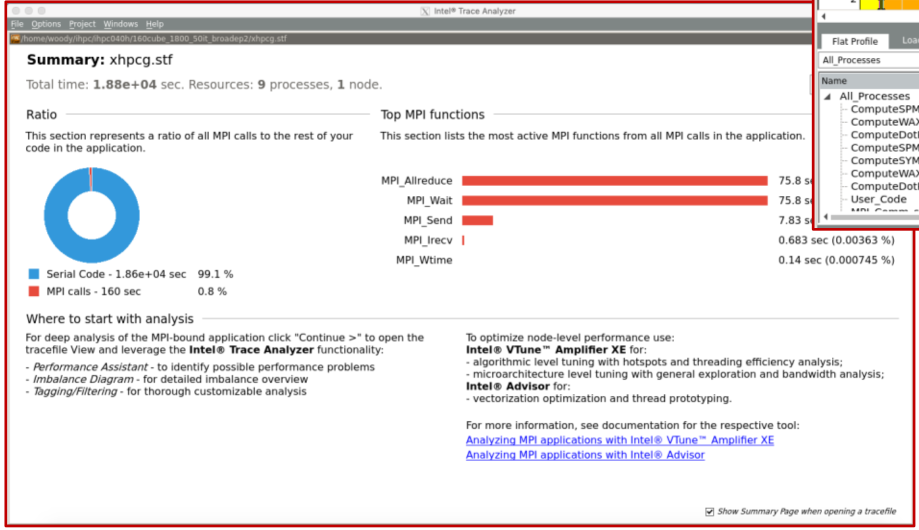
Event-based approach that record
- user function calls
- MPI communication calls



Name	TSelf
All_Processes	
ComputeSPMV_ref	18.7874 s
ComputeWAXPBY	226e-6 s
ComputeDotProduct_ref	854.298e-3 s
ComputeSPMV	41e-6 s
ComputeSYMGS_ref	39.4382 s
ComputeWAXPBY_ref	1.6031 s
ComputeDotProduct	91e-6 s
User_Code	654.169e-3 s

Performance Issue	Duration (%)	Duration
Wait at Barrier	0.15%	100.472e-3
Late Sender	0.04%	29.609e-3 s
Late Receiver	0.00%	1.45e-3 s
Show advanced..		

	P0	P1	P2
MPI_Allreduce	18.43e-3	2.223e-3	43.023e
Sum	18.43e-3	2.223e-3	43.023e
Mean	18.43e-3	2.223e-3	43.023e
StdDev	0	0	0


Intel® Trace Analyzer
File Options Project Windows Help
/home/woody/ihpc/ihpc040h/160cube_1800_50it_broadep2/xhpcg.stf

Summary: xhpcg.stf

Total time: **1.88e+04** sec. Resources: **9** processes, **1** node.

Ratio

This section represents a ratio of all MPI calls to the rest of your code in the application.

🟦	Serial Code - 1.86e+04 sec	99.1 %
🟥	MPI calls - 160 sec	0.8 %

Top MPI functions

This section lists the most active MPI functions from all MPI calls in the application.

MPI_Allreduce	75.8 s
MPI_Wait	75.8 s
MPI_Send	7.83 s
MPI_Irecv	0.683 sec (0.00363 %)
MPI_Wtime	0.14 sec (0.000745 %)

Where to start with analysis

For deep analysis of the MPI-bound application click "Continue >" to open the tracefile View and leverage the **Intel® Trace Analyzer** functionality:

- *Performance Assistant* - to identify possible performance problems
- *Imbalance Diagram* - for detailed imbalance overview
- *Tagging/Filtering* - for thorough customizable analysis

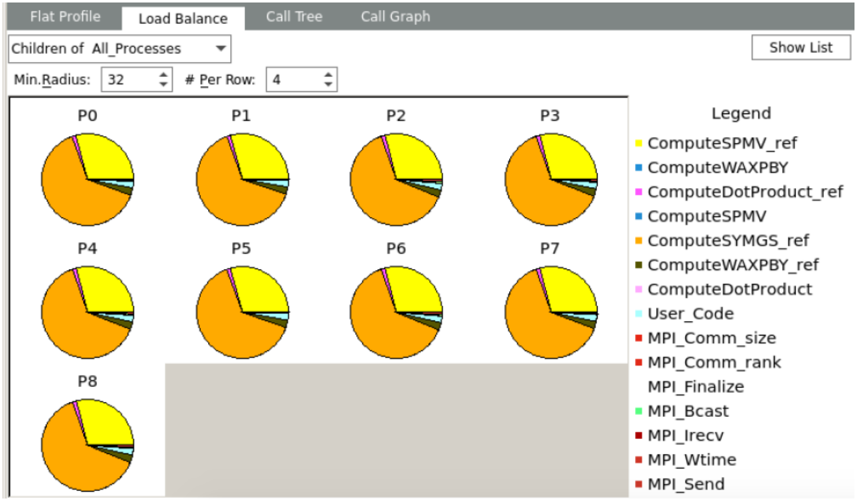
To optimize node-level performance use:
Intel® VTune™ Amplifier XE for:
- algorithmic level tuning with hotspots and threading efficiency analysis;
- microarchitecture level tuning with general exploration and bandwidth analysis;

Intel® Advisor for:
- vectorization optimization and thread prototyping.

For more information, see documentation for the respective tool:
Analyzing MPI applications with Intel® VTune™ Amplifier XE
Analyzing MPI applications with Intel® Advisor

☑ *Show Summary Page when opening a tracefile*

Some features of ITAC



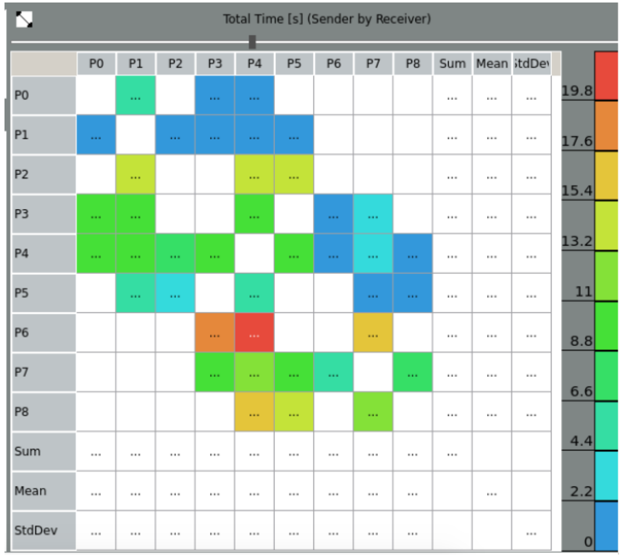
Children of All_Processes

Min.Radius: 32 # Per Row: 4

Legend

- 🟨 ComputeSPMV_ref
- 🟦 ComputeWAXPBY
- 🟪 ComputeDotProduct_ref
- 🟦 ComputeSPMV
- 🟧 ComputeSYMGS_ref
- 🟫 ComputeWAXPBY_ref
- 🟪 ComputeDotProduct
- ⬜ User_Code
- 🟥 MPI_Comm_size
- 🟥 MPI_Comm_rank
- MPI_Finalize
- 🟩 MPI_Bcast
- 🟥 MPI_Irecv
- 🟥 MPI_Wtime
- 🟥 MPI_Send

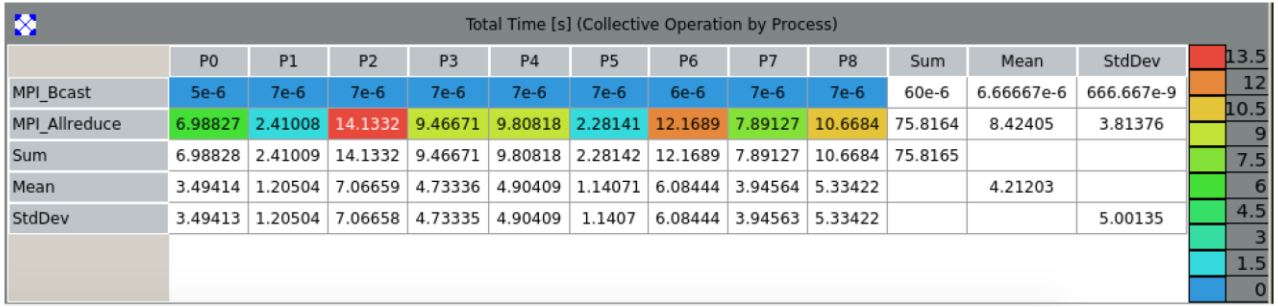
Receiver



Total Time [s] (Sender by Receiver)

	P0	P1	P2	P3	P4	P5	P6	P7	P8	Sum	Mean	StdDev
P0		...		...	...					...	...	...
P1	...		...	...	...	...				...	...	...
P2		...			...	...				...	...	...
P3	...	...			...		...	...		...	...	...
P4	...	...	...	...		...	...	...	...	...	...	...
P5		...	...		...			...	...	...	...	...
P6				...	...			...		...	...	...
P7				...	...	...	...		...	...	...	...
P8					...	...		...		...	...	...
Sum	...	...	...	...	...	...	...	...	...	...		
Mean	...	...	...	...	...	...	...	...	...		...	
StdDev	...	...	...	...	...	...	...	...	...			...

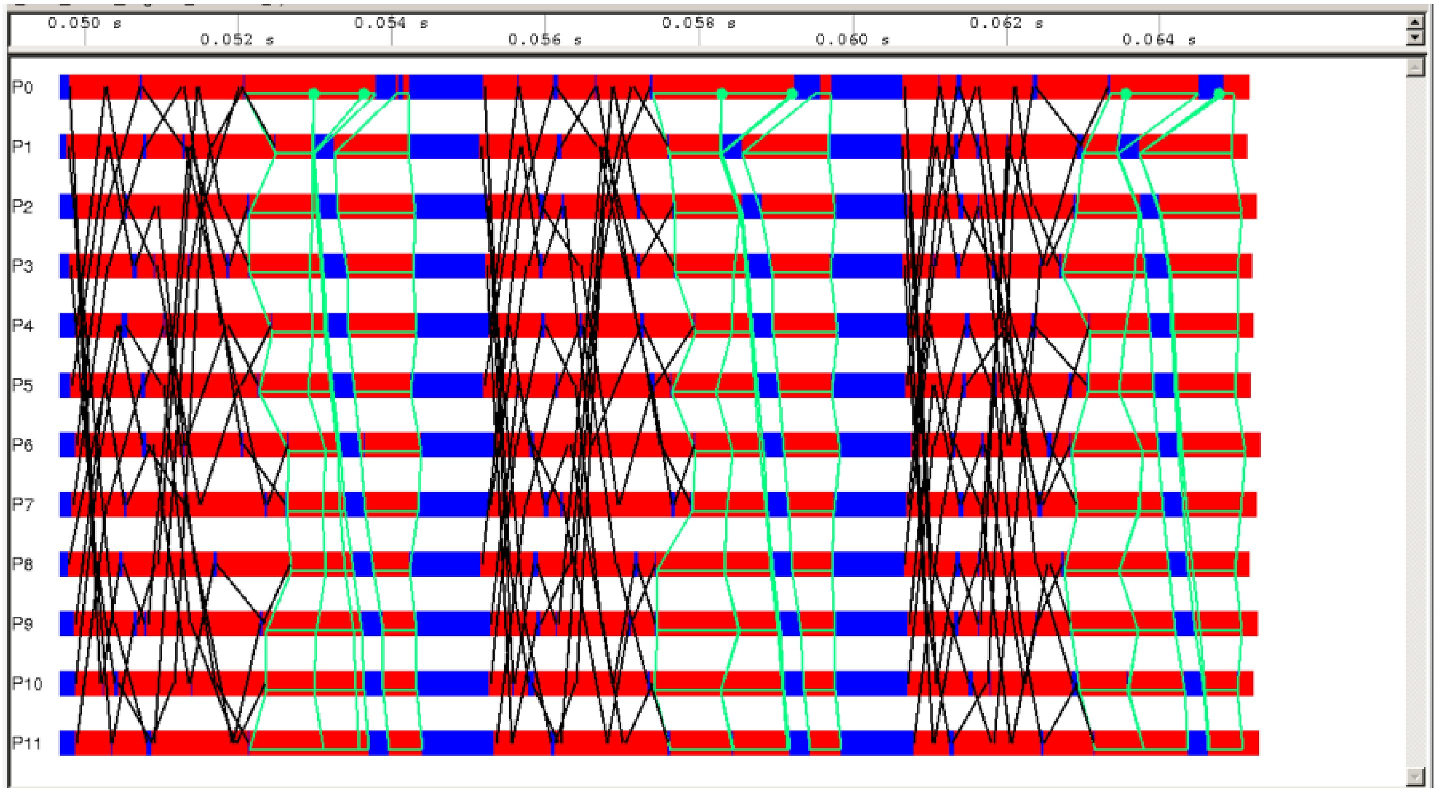
Sender



Total Time [s] (Collective Operation by Process)

	P0	P1	P2	P3	P4	P5	P6	P7	P8	Sum	Mean	StdDev
MPI_Bcast	5e-6	7e-6	7e-6	7e-6	7e-6	7e-6	6e-6	7e-6	7e-6	60e-6	6.66667e-6	666.667e-9
MPI_Allreduce	6.98827	2.41008	14.1332	9.46671	9.80818	2.28141	12.1689	7.89127	10.6684	75.8164	8.42405	3.81376
Sum	6.98828	2.41009	14.1332	9.46671	9.80818	2.28142	12.1689	7.89127	10.6684	75.8165		
Mean	3.49414	1.20504	7.06659	4.73336	4.90409	1.14071	6.08444	3.94564	5.33422		4.21203	
StdDev	3.49413	1.20504	7.06658	4.73335	4.90409	1.1407	6.08444	3.94563	5.33422			5.00135

Timeline view


0.050 s
0.052 s
0.054 s
0.056 s
0.058 s
0.060 s
0.062 s
0.064 s
P0
P1
P2
P3
P4
P5
P6
P7
P8
P9
P10
P11