# 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 (more efficient PtP)
  - Message probing: MPI_Probe,… (is there a message waiting?)
  - One-sided communication: MPI_Put, MPI_Get, MPI_Accumulate,… (only one rank necessary to get data across)
  - Partitioned communication (better communication of threads are present)
- Collectives bells and whistles
  - MPI_Reduce_scatter, MPI_Scan, neighborhood collectives, …
- MPI I/O (reading and writing files through MPI, in parallel)
- Virtual topologies (make known to MPI who communicates with whom)
- MPI shared memory (more efficient intra-node communication)

NHR FAU

Friedrich-Alexander-Universität
Erlangen-Nürnberg

# Introduction to Parallel Programming with MPI

Dr. Alireza Ghasemi, Dr. Georg Hager

Erlangen National High Performance Computing Center

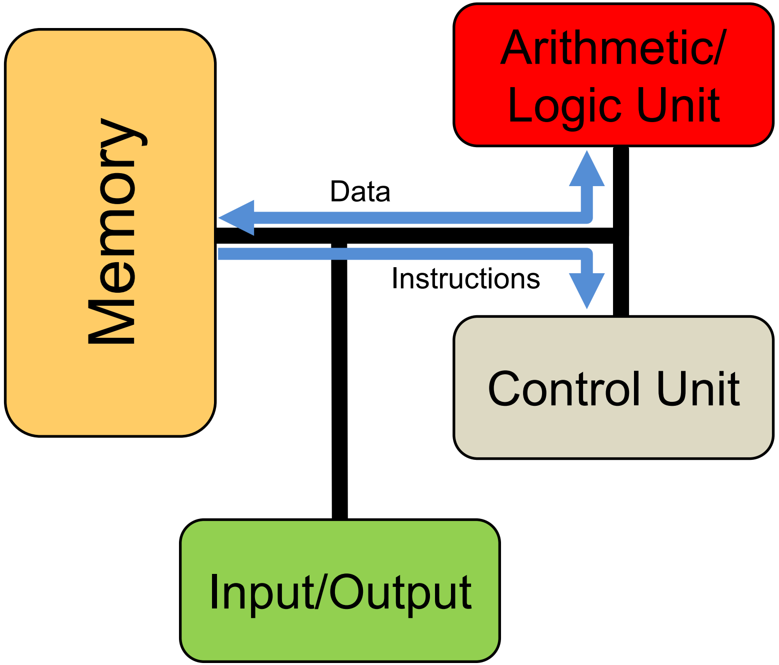
## Computer Architecture and Performance issues In MPI programming

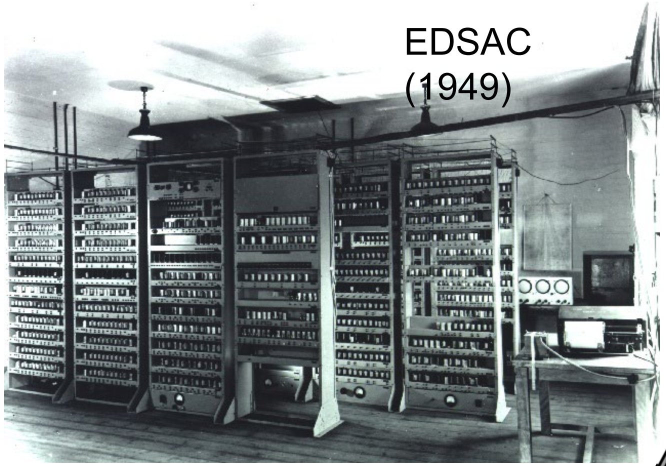
# Performance issues – overview

- 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

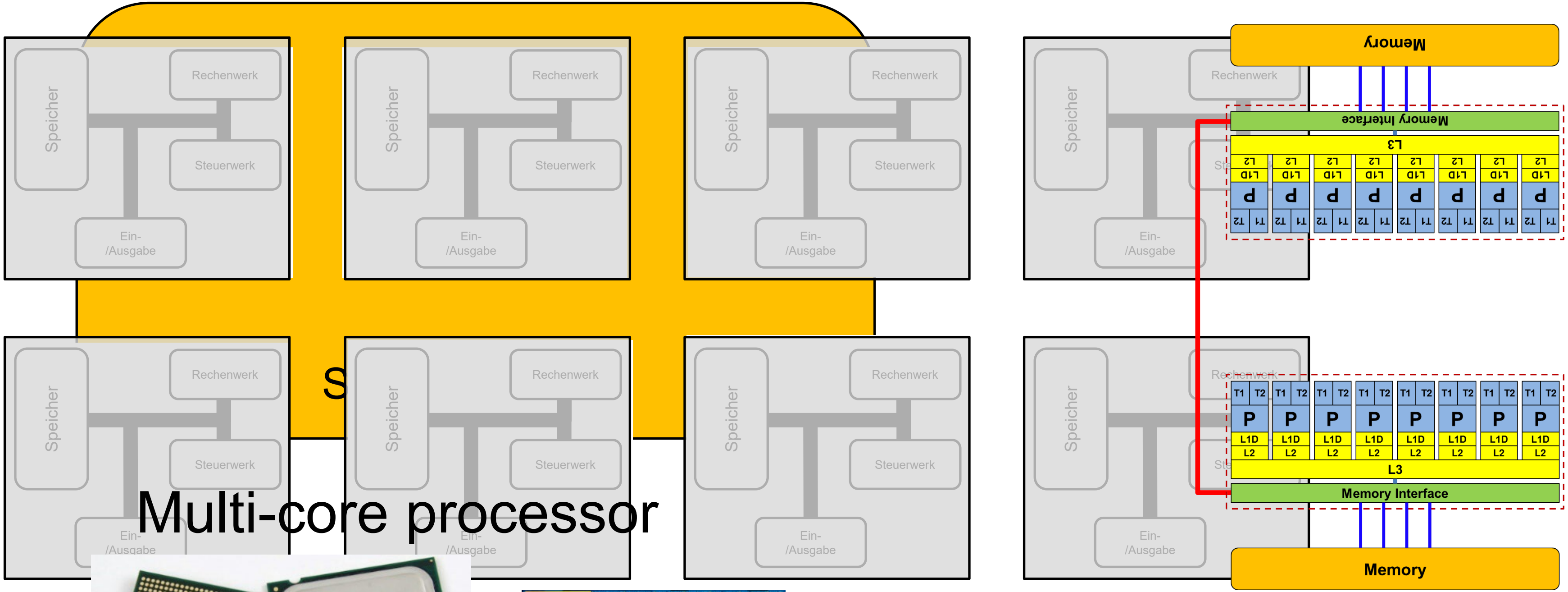




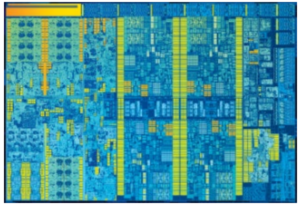
CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=432935

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

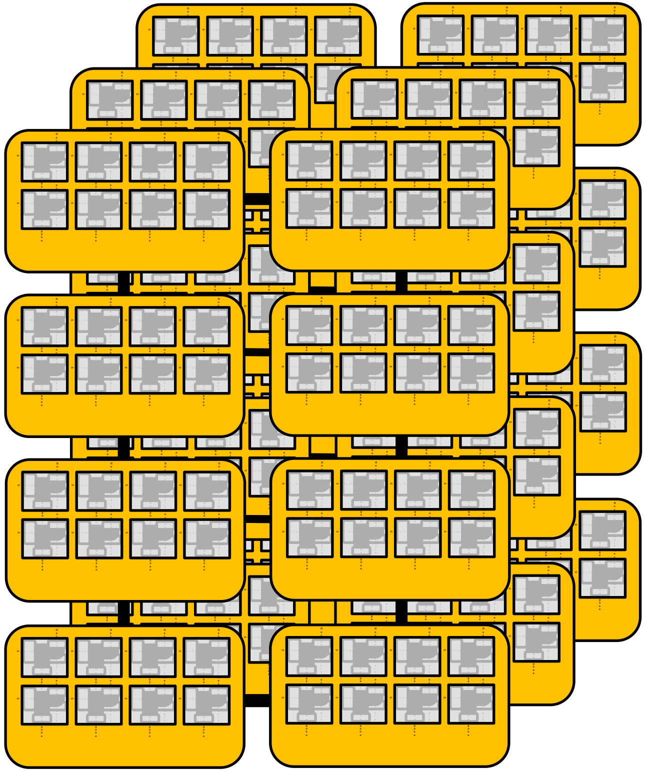


Multi-core processor



Multiple CPU chips per node

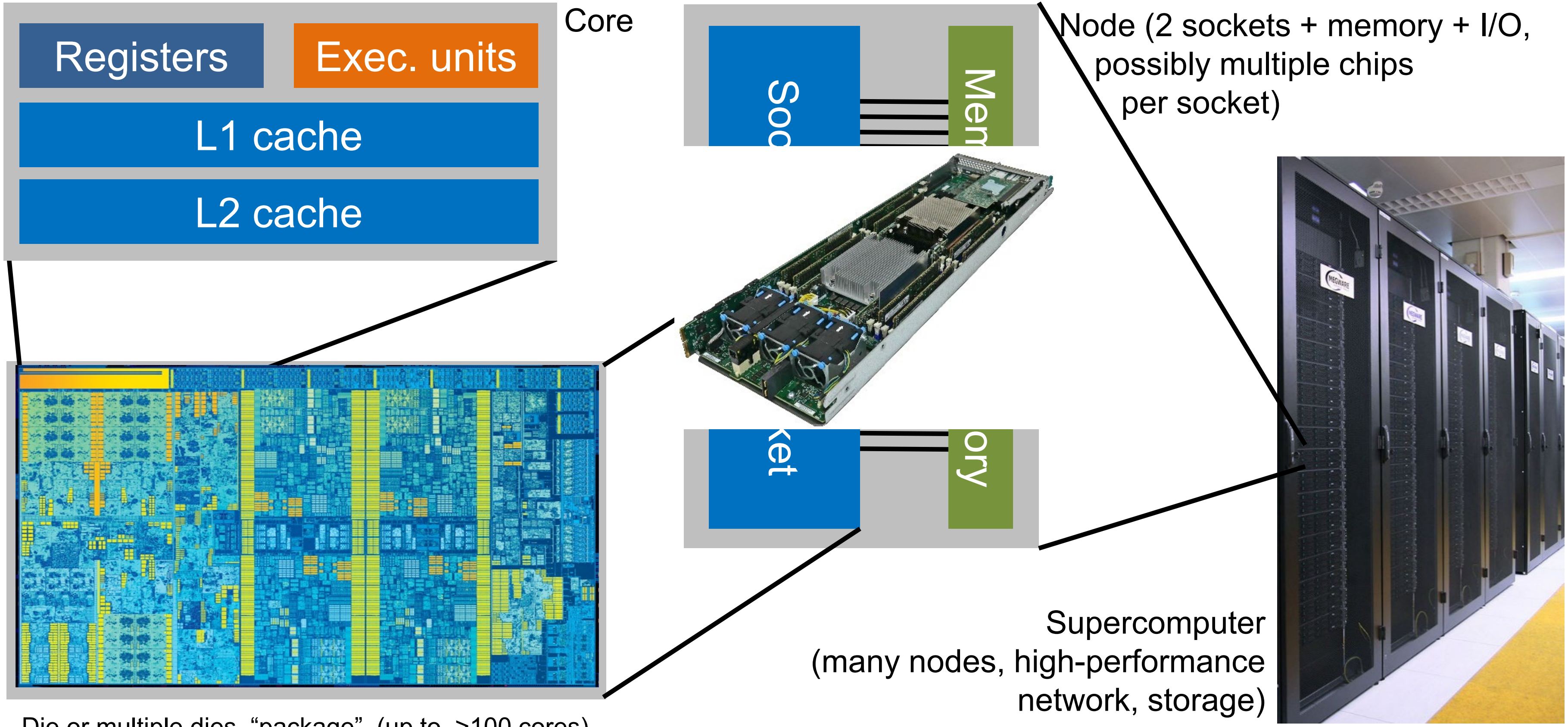
# Distributed memory: no cache-coherent single address space



Cluster/
supercomputer

Modern supercomputers are
shared-/distributed-memory hybrids

# Parallelism in modern computers



Core

Registers

Exec. units

L1 cache

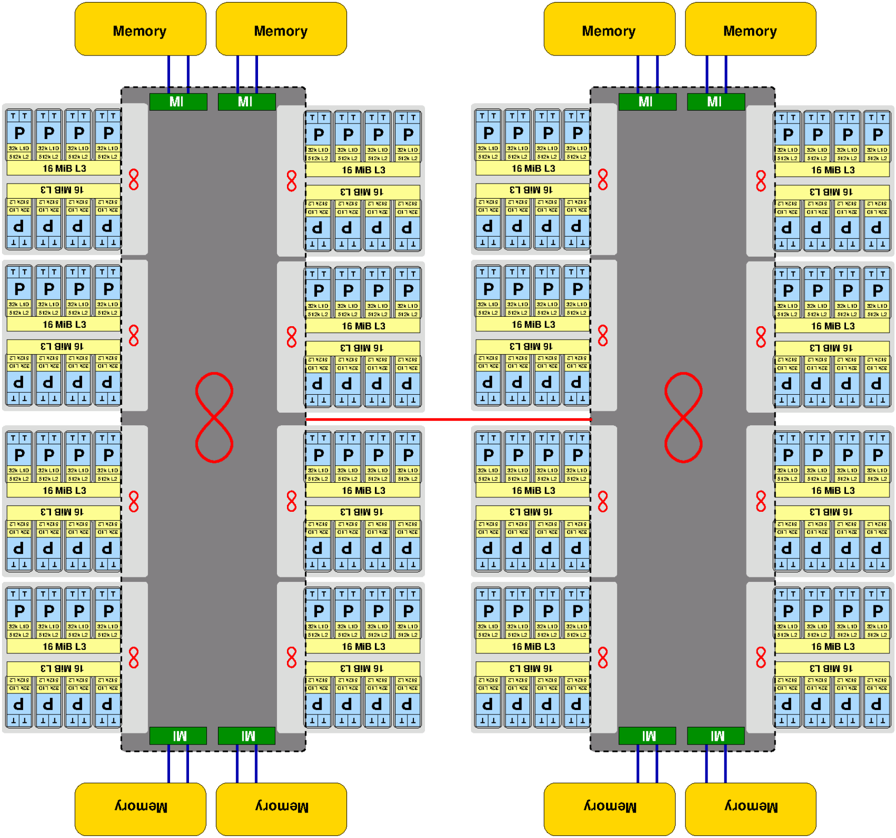
L2 cache

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

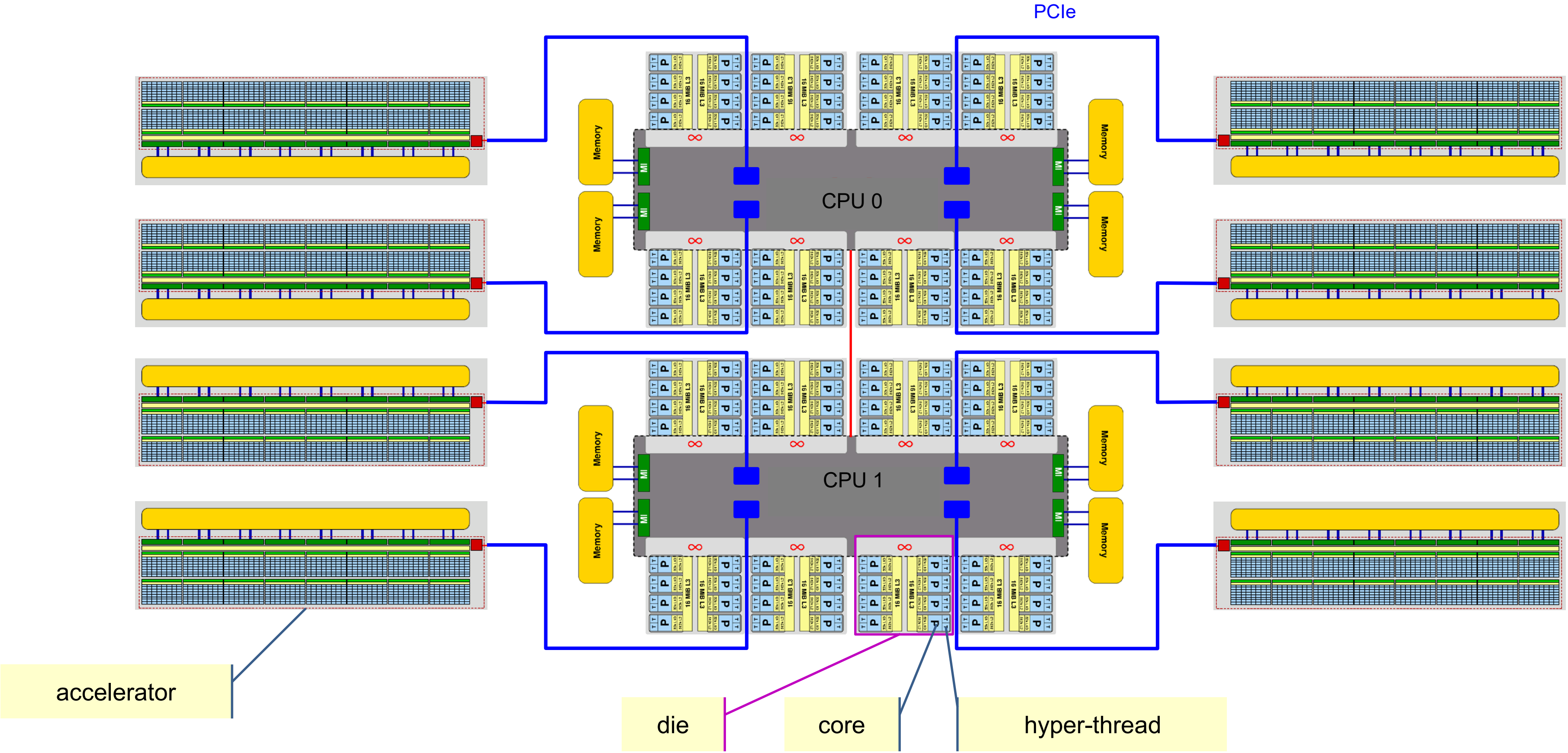
Node (2 sockets + memory + I/O, possibly multiple chips per socket)

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

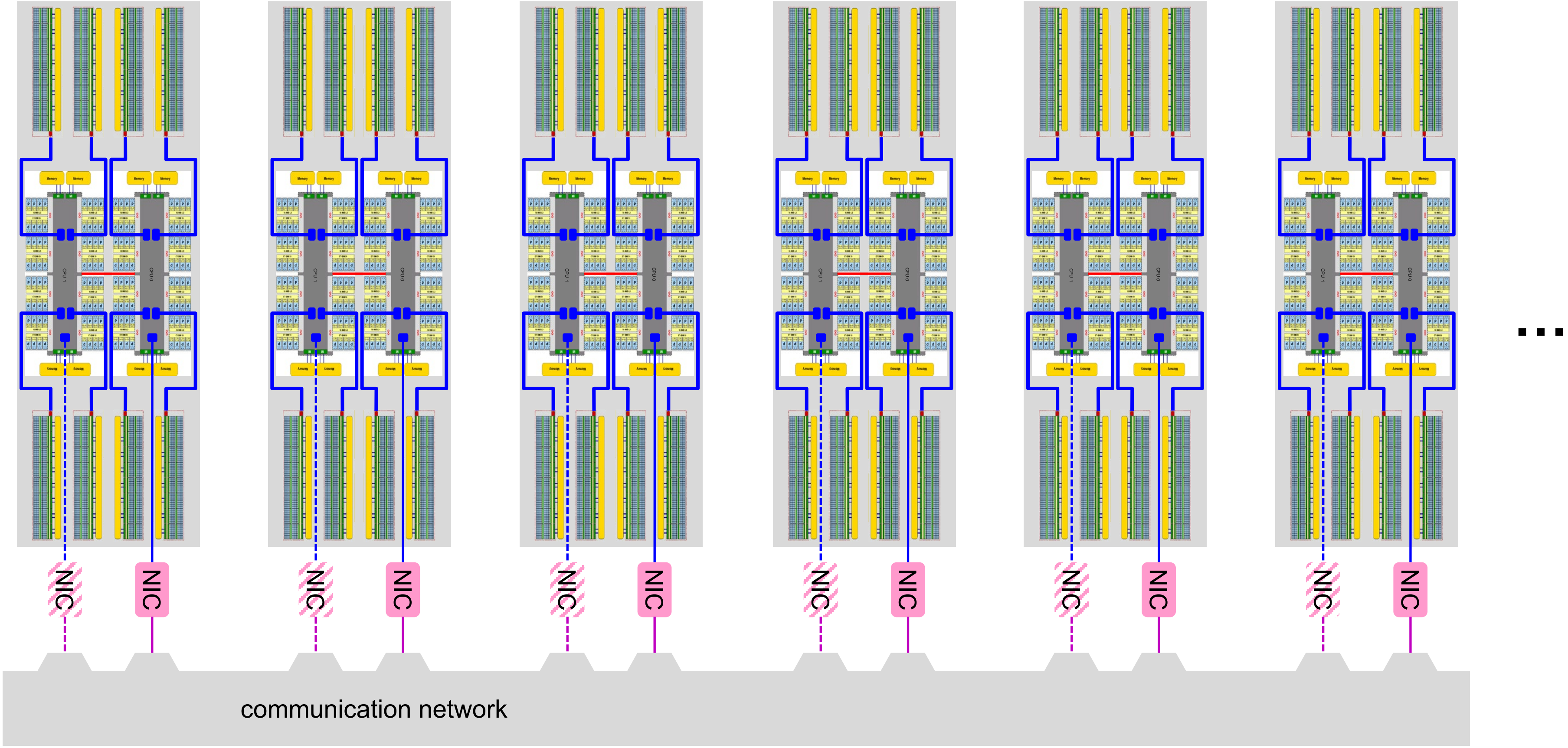
# A modern CPU compute node (AMD Zen2 “Rome”)



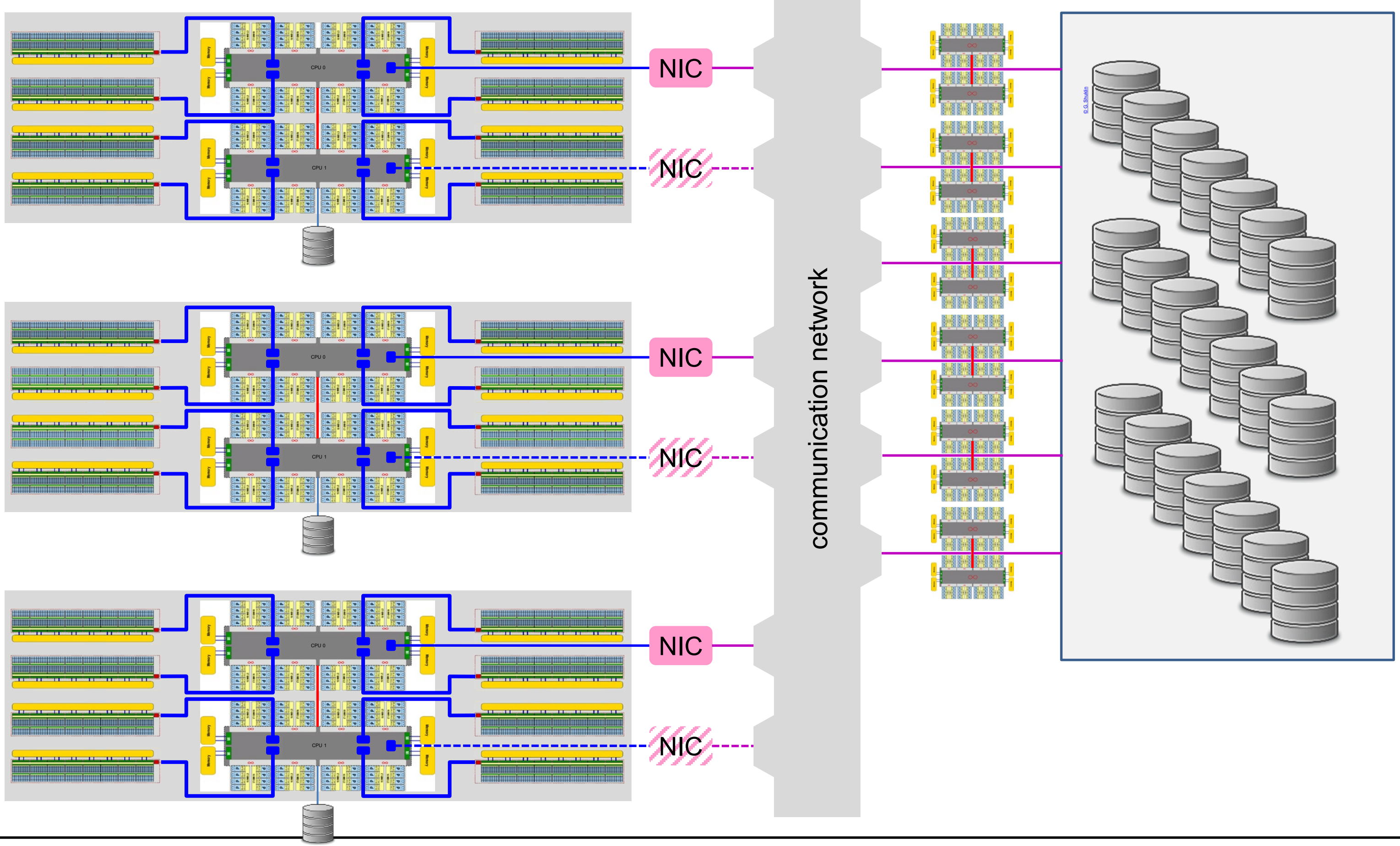
# Adding accelerators to the node



# Turning it into a cluster



# Adding permanent storage



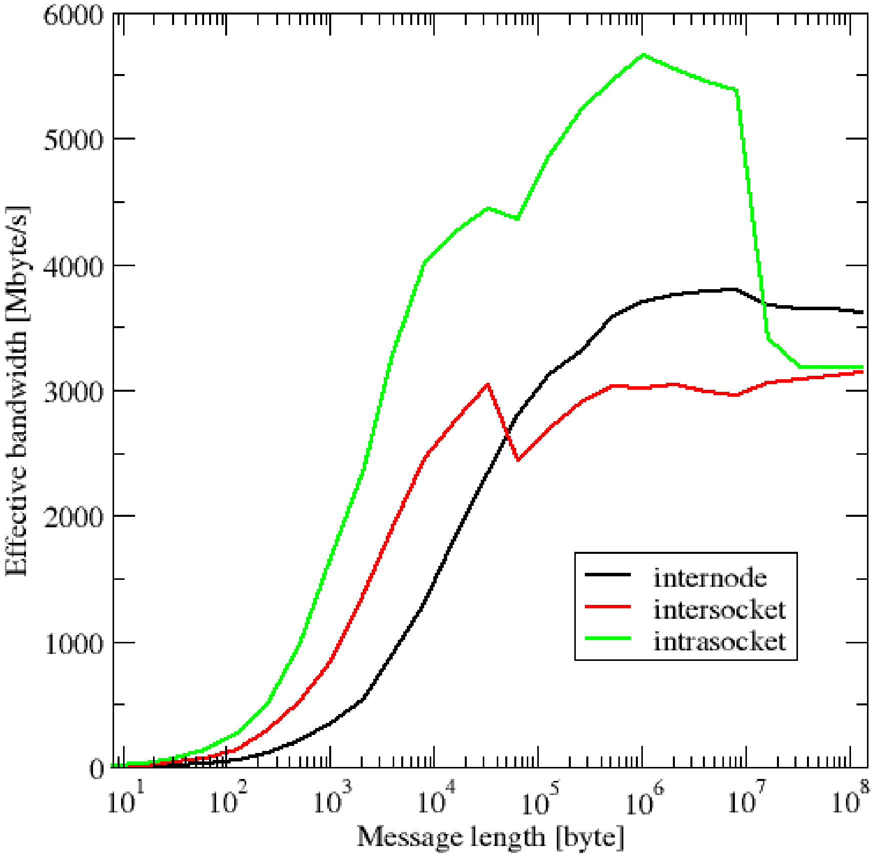
# Point-to-point data transmission performance

- Simple “Hockney model” for data transfer time

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

$\lambda$: latency, $b$: asymptotic BW

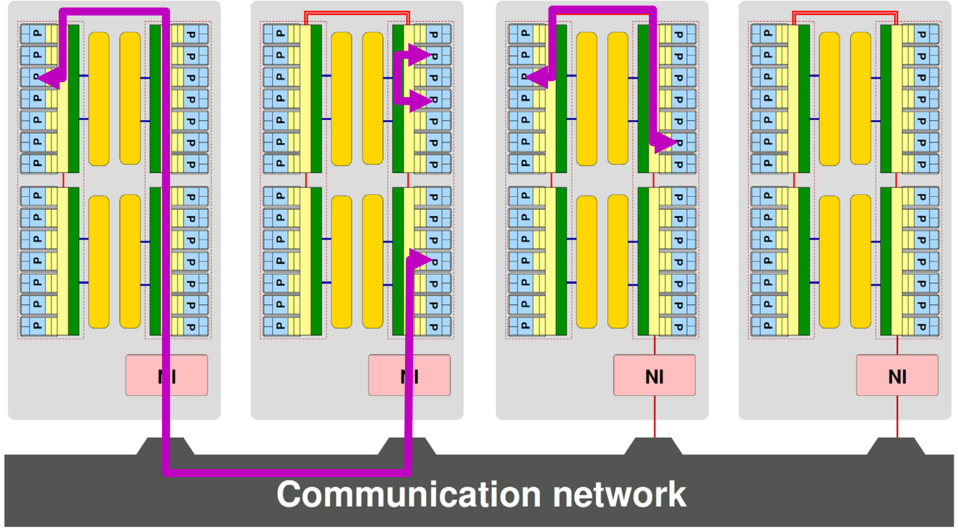
- Reality is more complicated
  - System topology
  - Caching effects
  - Contention effects
  - Protocol switches
  - Collective communication



# 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 $\lambda$ and $b$
  - On top: Effects from network structure

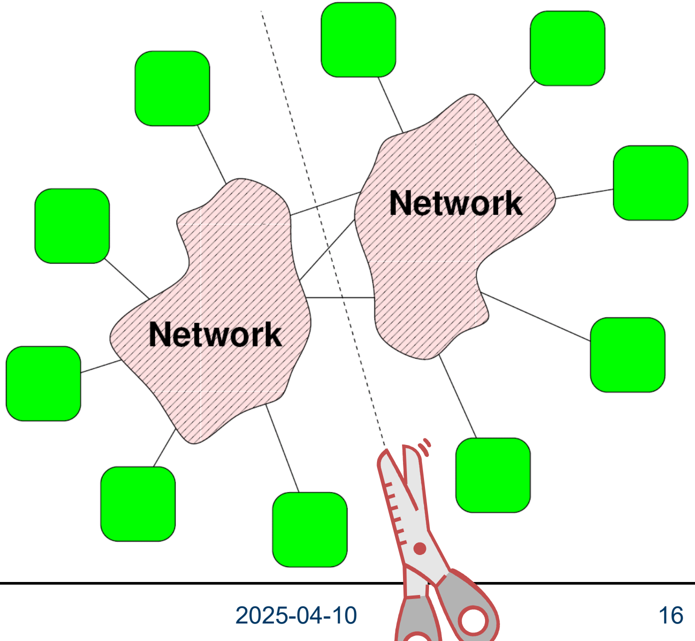
# Characterizing communication networks

- 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

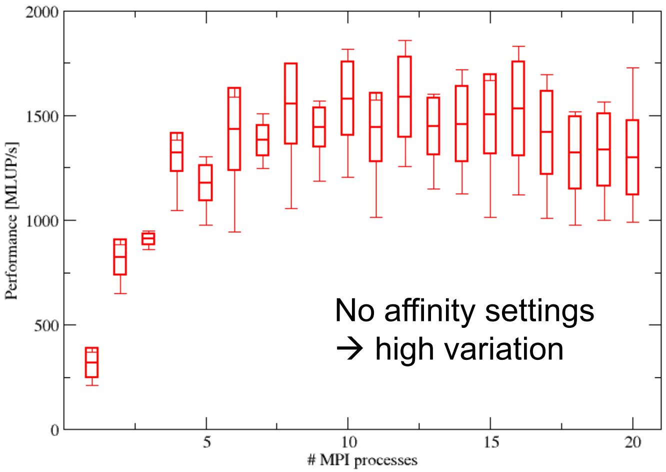
- More meaningful metric for system scalability: bisection BW per node: $B_b/N_{nodes}$

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

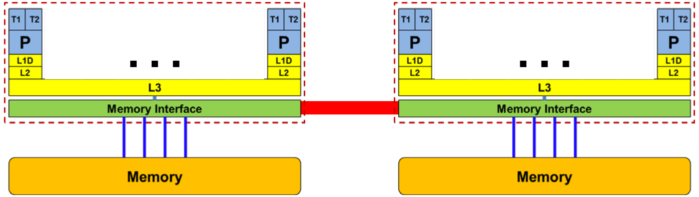


# Affinity control (pinning) of processes

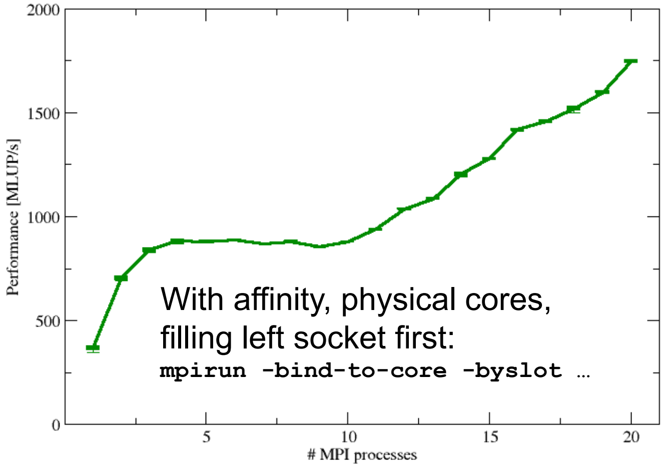
# Anarchy vs. affinity with a heat equation solver



No affinity settings
→ high variation



2x 10-core Intel Ivy Bridge, OpenMPI



With affinity, physical cores,
filling left socket first:

```
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`**)

# Simple example (Intel MPI)

- 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

# 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?

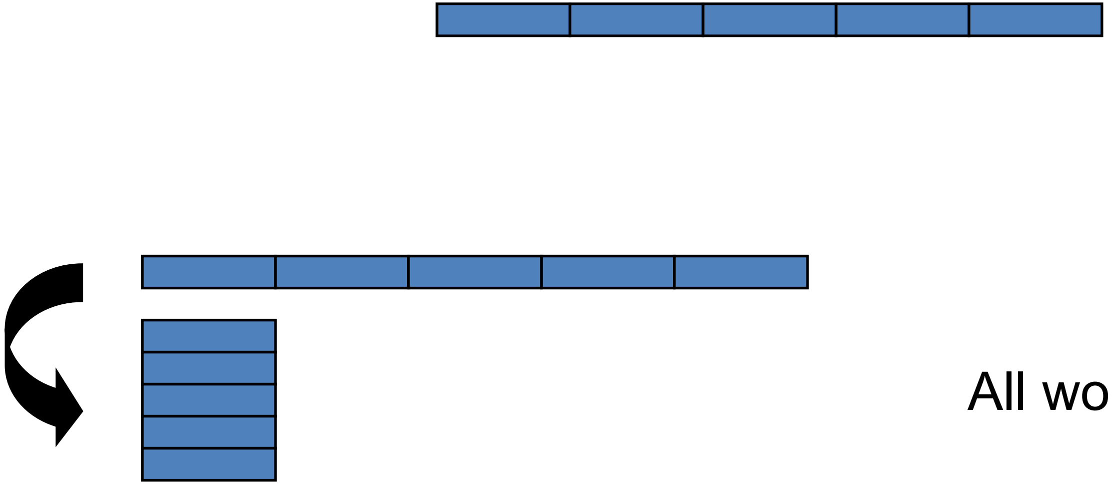
→ parallel efficiency: $\varepsilon(N) = \frac{S(N)}{N}$

- 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



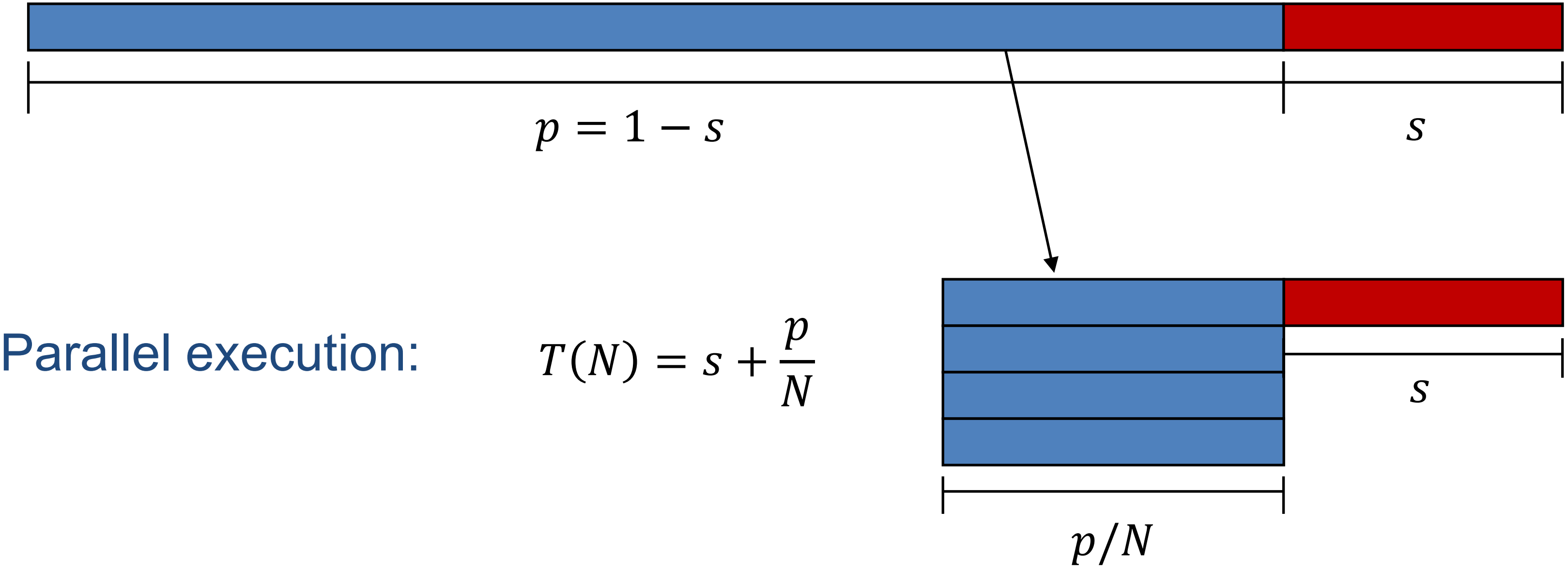
Ideal world:
All work is perfectly parallelizable
$S(N) = N, \qquad \varepsilon = 1$

# 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



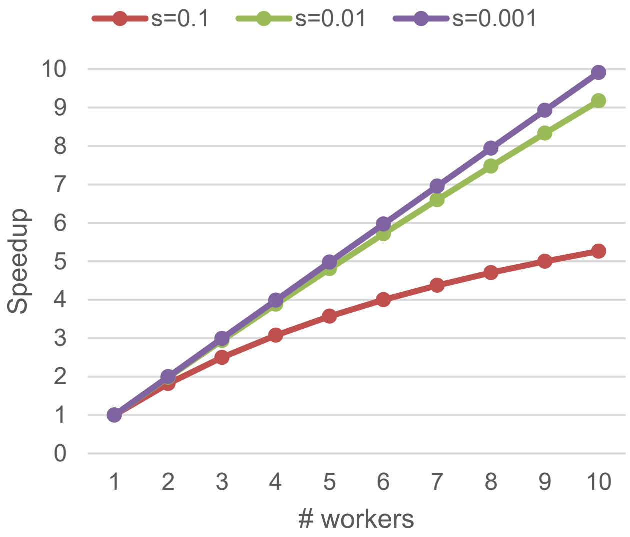
Parallel execution: $T(N) = s + \frac{p}{N}$

# 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

→ $$T(N) = s + \frac{p}{N} + c(N)$$

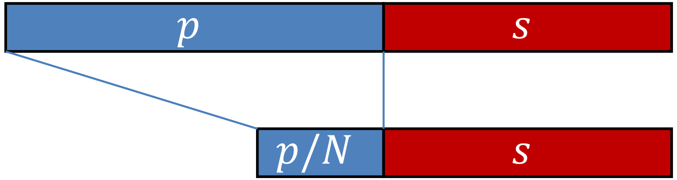
- What goes into $c(N)$?
  - Communication pattern
  - Synchronization strategy
  - Message sizes
  - Network structure
  - …

Typical examples: $c(N) =$

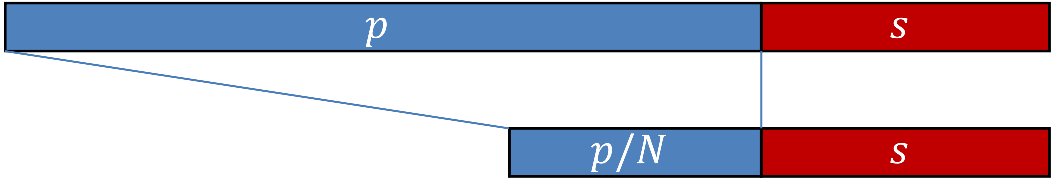
| | |
|---|---|
| $kN^2$ | (all-to-all on bus network) |
| $k \log N$ | (optimal synchronization) |
| $kN$ | (one sends to all) |
| $\lambda + kN^{-\frac{2}{3}}$ | (Cartesian domain decomposition, nonblocking network) |

# 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 $N$:

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

- 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?

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

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

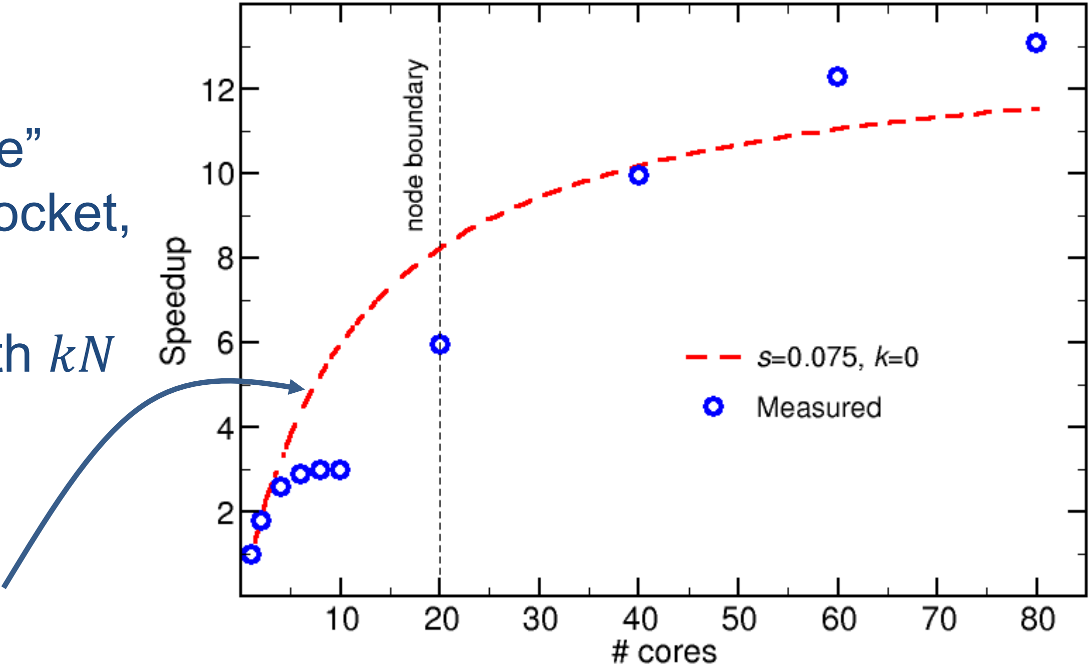
Much more relaxed conditions on $c(N)$

# 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

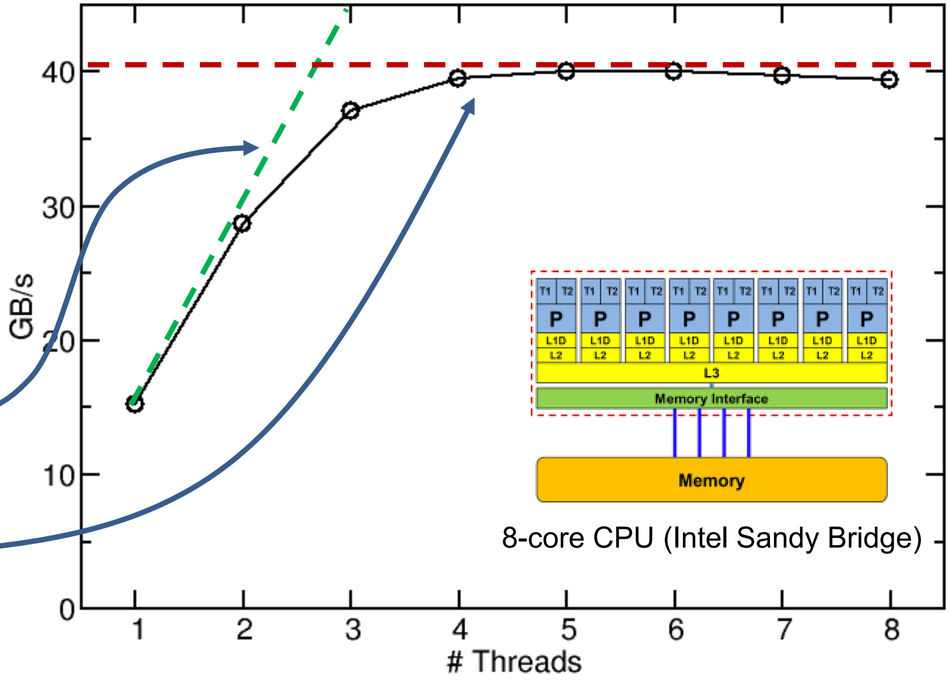


# Resource bottlenecks

- 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];
```

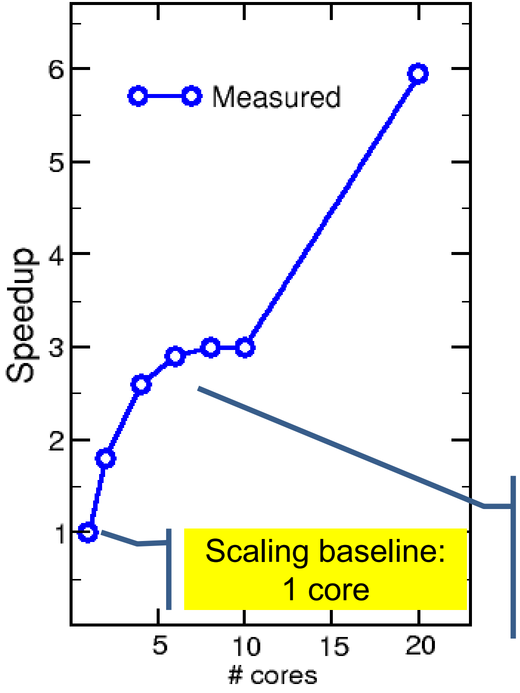
- Amdahl’s: $s = 0, c(N) = 0$
  - Perfect speedup? No!
  - Saturation because of memory bandwidth exhaustion



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”

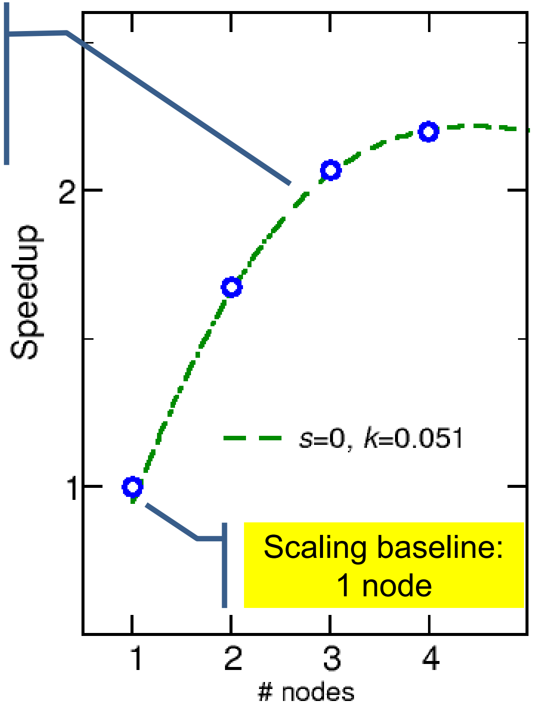


Scaling baseline: 1 core

Socket saturation due to memory bandwidth

Separating scaling baselines is important in modeling!

Model well suited for internode scaling!



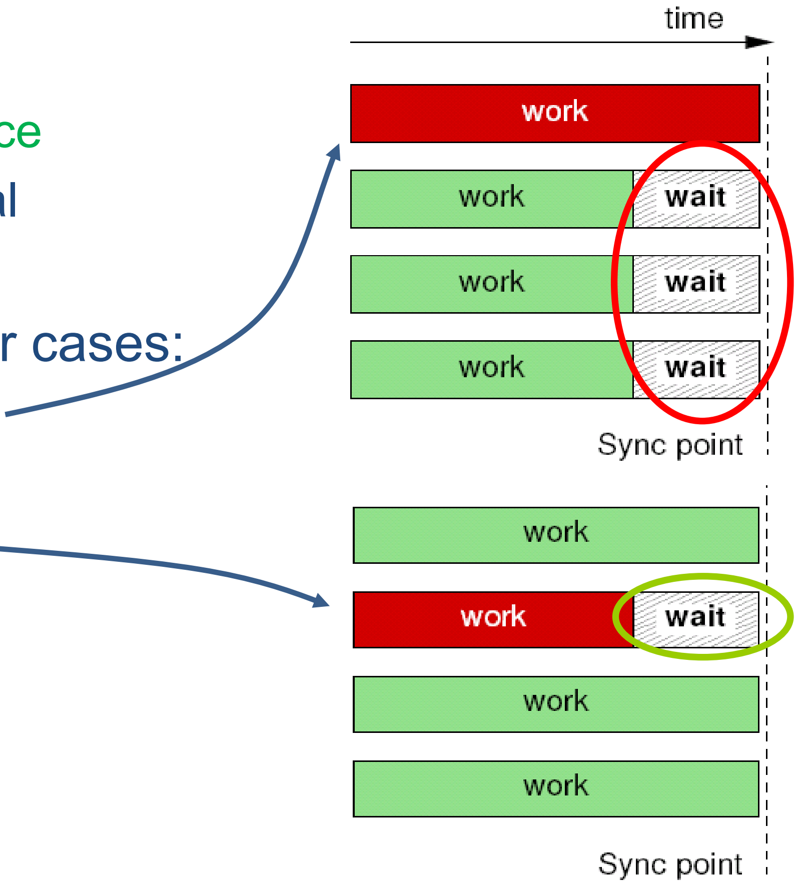
Scaling baseline: 1 node

# 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



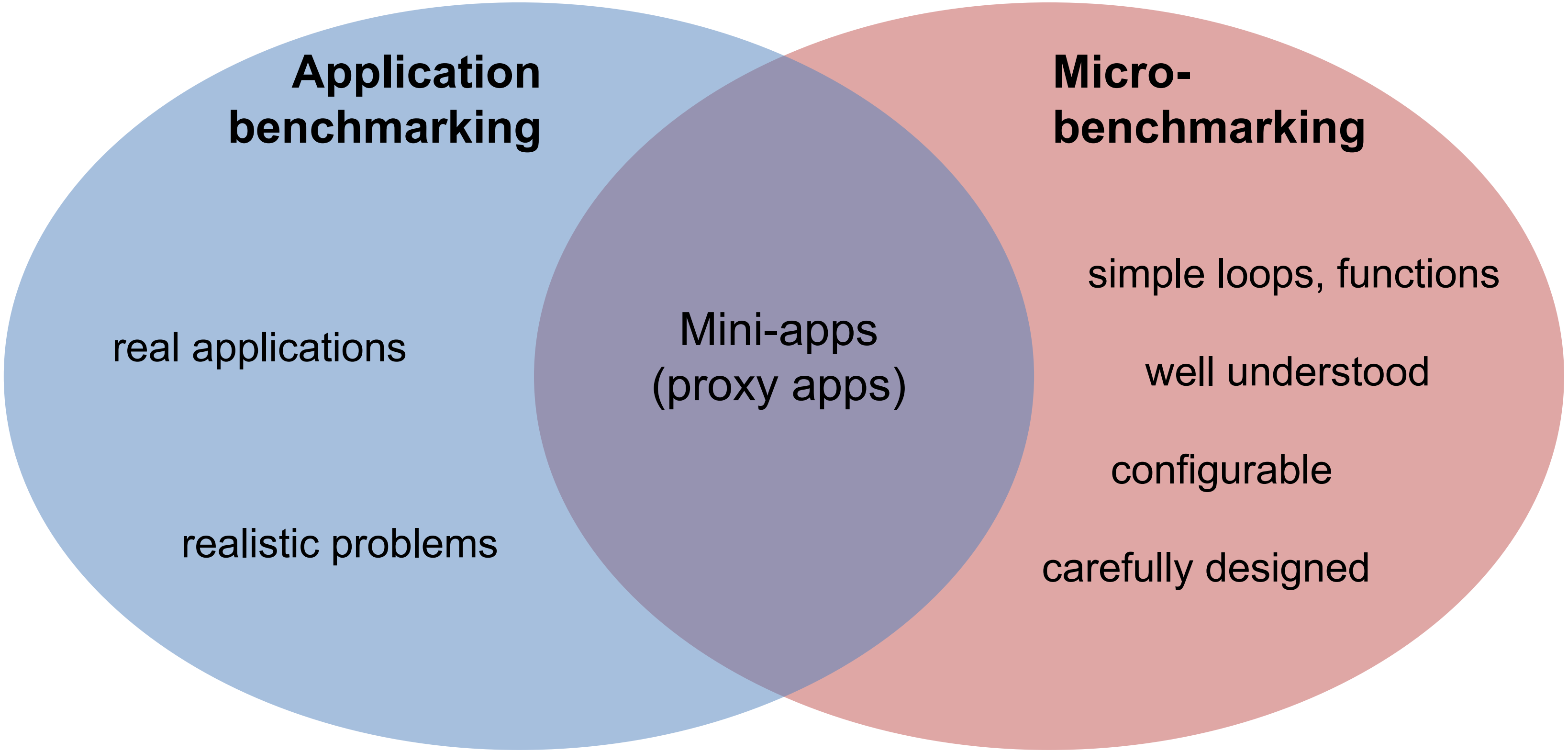
Friedrich-Alexander-Universität
Erlangen-Nürnberg

# 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)



**Application benchmarking**

real applications

realistic problems

Mini-apps (proxy apps)

**Micro-benchmarking**

simple loops, functions

well understood

configurable

carefully designed

# 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>
```

Return time stamp

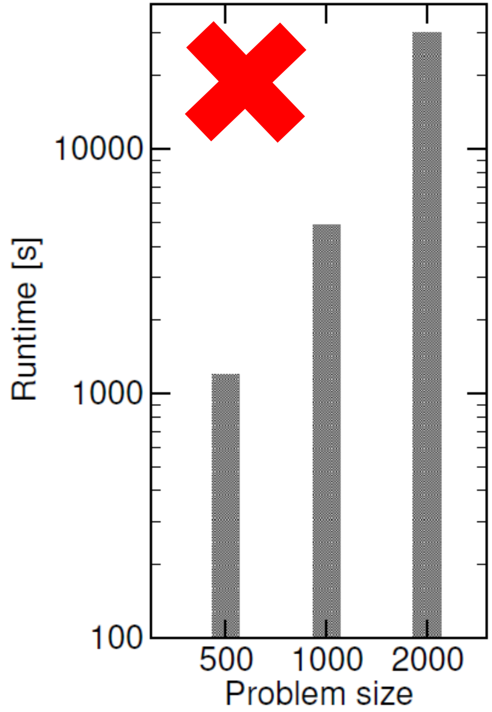
```
double get_walltime() {
  struct timespec ts;
  clock_gettime(CLOCK_MONOTONIC, &ts);
   return (double)ts.tv_sec +
          (double)ts.tv_nsec * 1.e-9;
}
```

For Fortran

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

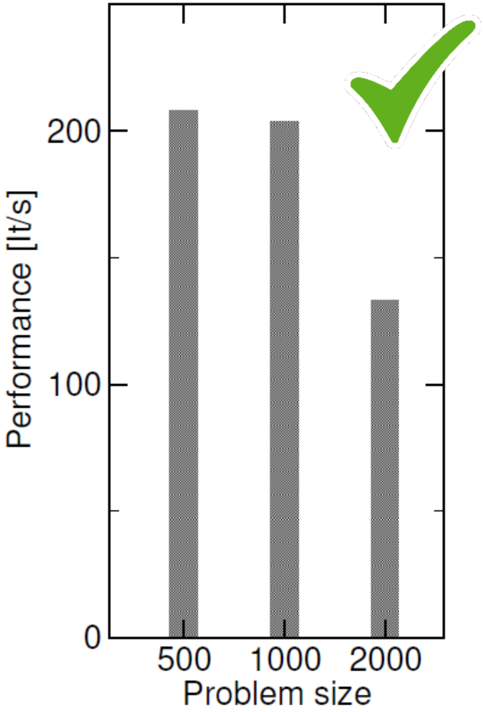
# Popular blunders: runtime != performance

- Just presenting runtime is almost always a bad idea!



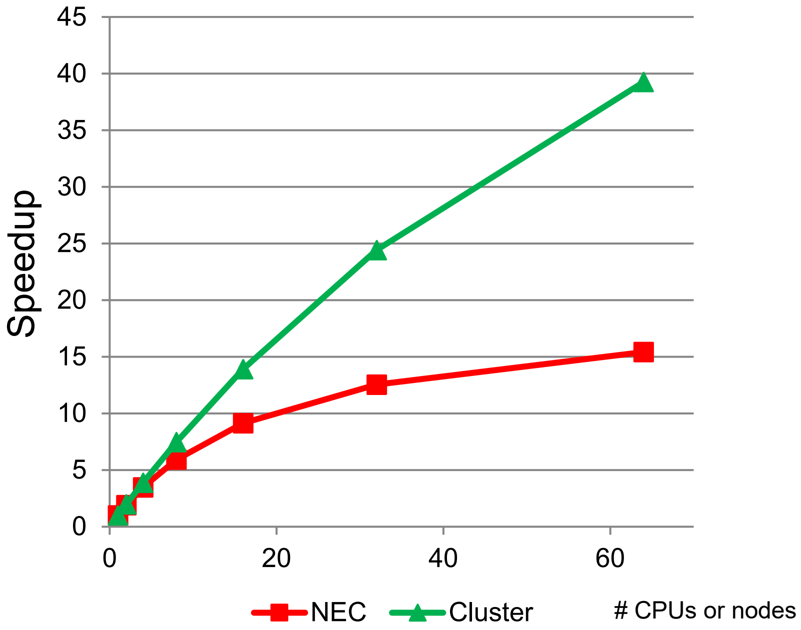
Insights hidden by trivial dependency: “larger problems need more time”

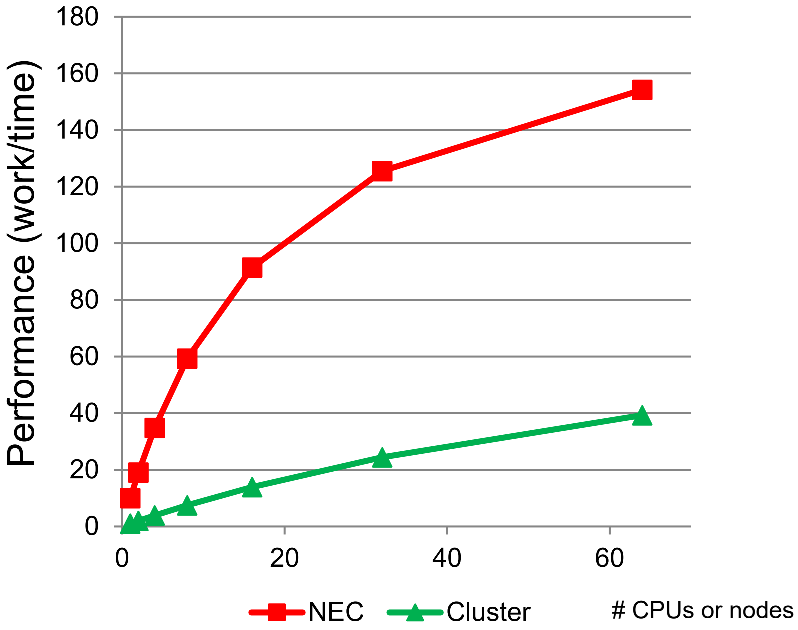
Performance metric reveals interesting behavior worth investigating!



# Popular blunders: speedup != performance

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





NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

# MPI tracing tools

# MPI tracing tools

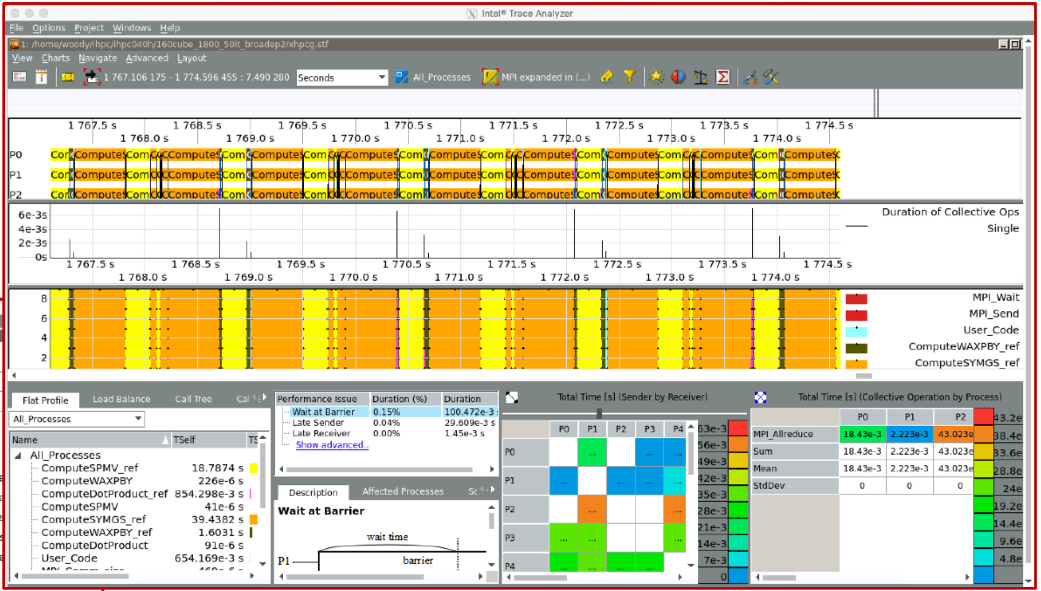
- Allow the user to track events and statistics pertaining to MPI communication and code execution
- Popular tools
  - Intel Trace Analyzer and Collector (ITAC)
  - VAMPIR (commercial)
  - Paraver

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

# Intel Trace Anayzer and Collector

Event-based tool recording user function calls and MPI communication calls

GUI for advanced visualization



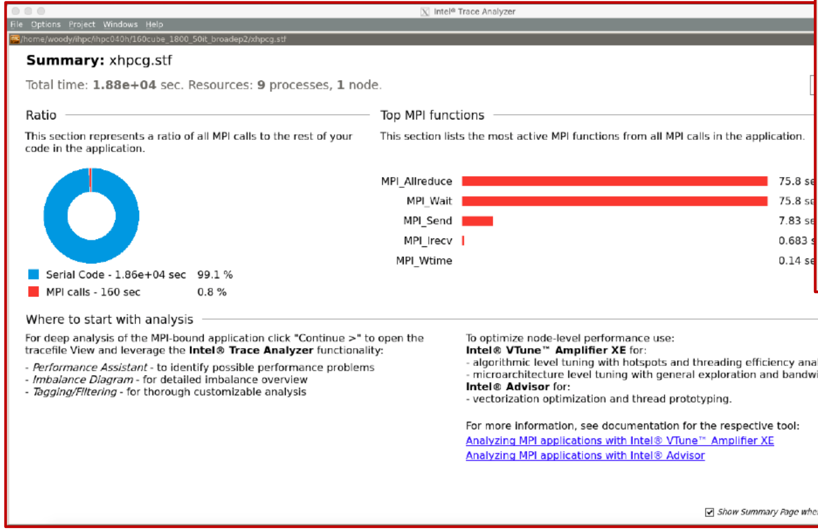
All_Processes

| Name | TSelf | TS |
|---|---|---|
| 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 | P3 | P4 |
|---|---|---|---|---|---|
| P0 | | ... | | ... | ... |
| P1 | ... | | ... | ... | ... |
| P2 | | ... | | | ... |
| P3 | ... | ... | | | ... |
| P4 | ... | ... | ... | ... | |

| | 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 se |
| MPI_Wait | 75.8 se |
| MPI_Send | 7.83 se |
| MPI_Irecv | 0.683 s |
| MPI_Wtime | 0.14 se |

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:

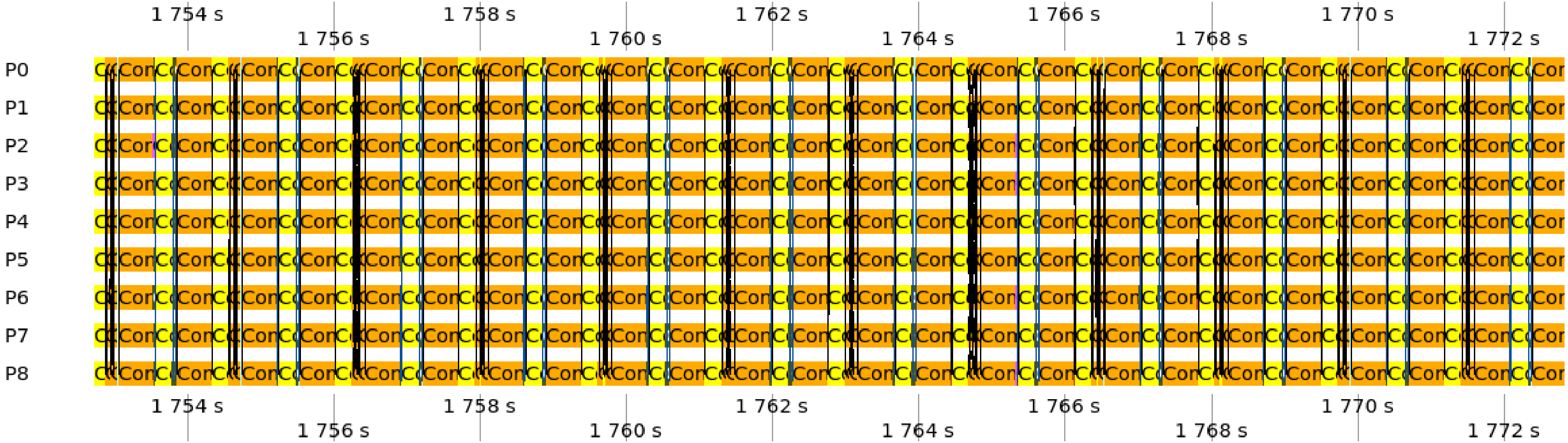
- *Performance Assistant* - to identify possible performance problems
- *Imbalance Diagram* - for detailed imbalance overview
- *Tagging/Filtering* - for thorough customizable analysis

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

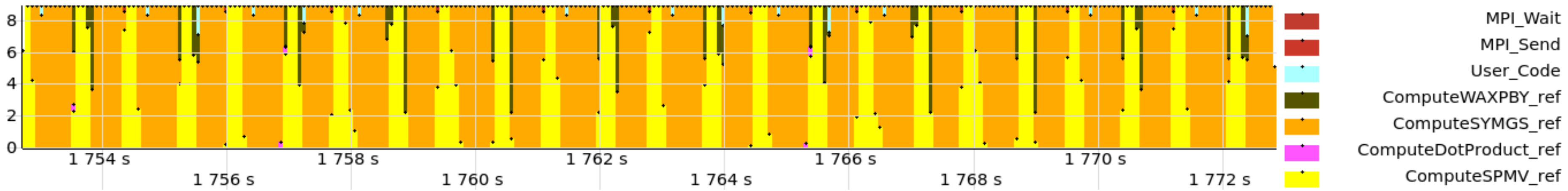
☑ Show Summary Page when opening a tracefile

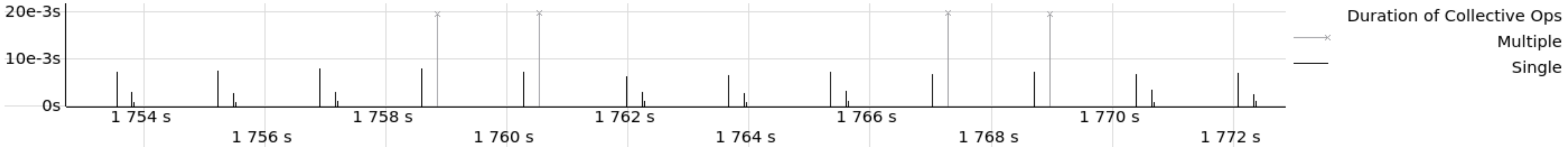
# Event timeline view



- Timeline of MPI and user function execution
- Message visualization
- Context menu provides details on functions/messages
- Zoom/pan

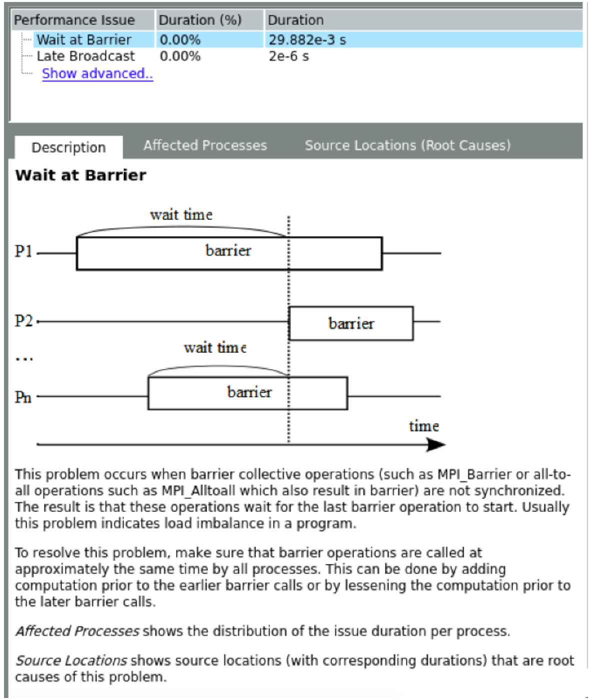
# Quantitative and qualitative timelines





- Time spent in different MPI/user functions across processes
- Duration of certain things (collectives, PtP)

# Performance advice



| Performance Issue | Duration (%) | Duration |
|---|---|---|
| Wait at Barrier | 0.00% | 29.882e-3 s |
| Late Broadcast | 0.00% | 2e-6 s |
| Show advanced.. | | |

Description | Affected Processes | Source Locations (Root Causes)

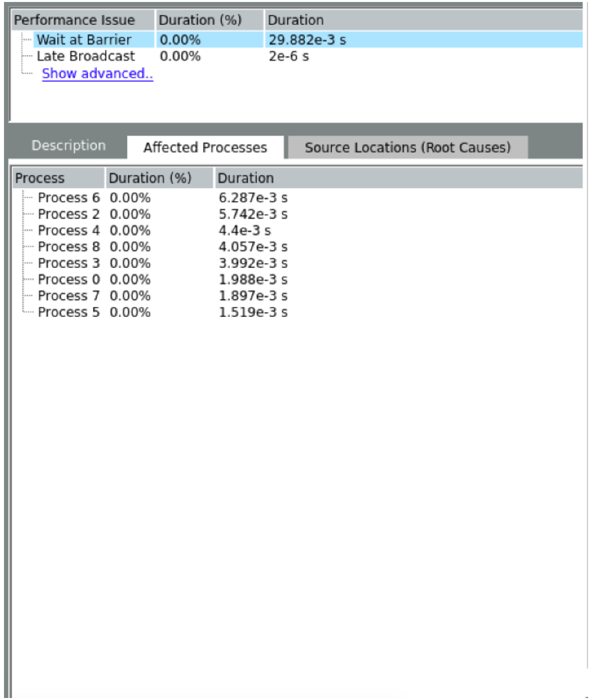
**Wait at Barrier**

This problem occurs when barrier collective operations (such as MPI_Barrier or all-to-all operations such as MPI_Alltoall which also result in barrier) are not synchronized. The result is that these operations wait for the last barrier operation to start. Usually this problem indicates load imbalance in a program.

To resolve this problem, make sure that barrier operations are called at approximately the same time by all processes. This can be done by adding computation prior to the earlier barrier calls or by lessening the computation prior to the later barrier calls.

*Affected Processes* shows the distribution of the issue duration per process.

*Source Locations* shows source locations (with corresponding durations) that are root causes of this problem.



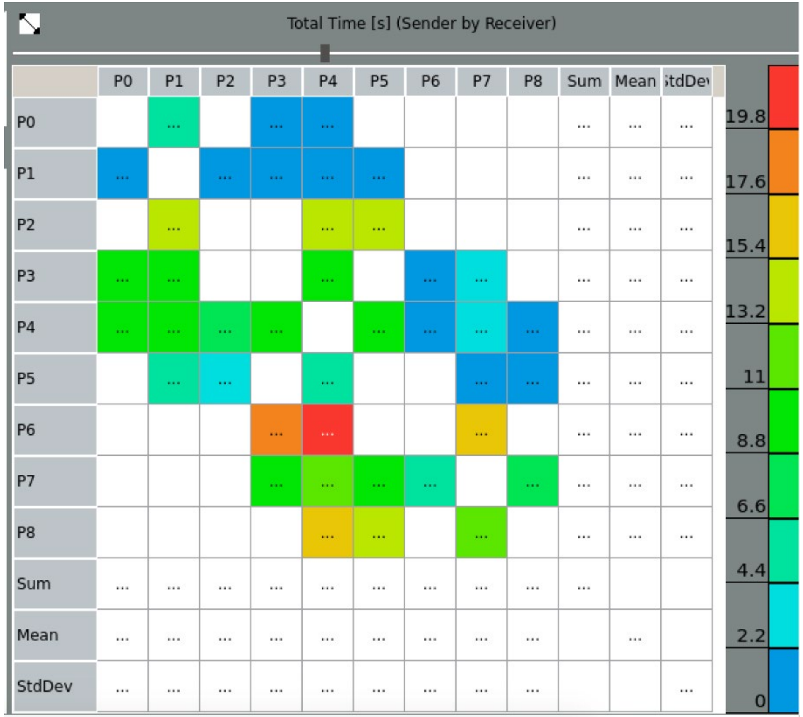
| Performance Issue | Duration (%) | Duration |
|---|---|---|
| Wait at Barrier | 0.00% | 29.882e-3 s |
| Late Broadcast | 0.00% | 2e-6 s |
| Show advanced.. | | |

Description | Affected Processes | Source Locations (Root Causes)

| Process | Duration (%) | Duration |
|---|---|---|
| Process 6 | 0.00% | 6.287e-3 s |
| Process 2 | 0.00% | 5.742e-3 s |
| Process 4 | 0.00% | 4.4e-3 s |
| Process 8 | 0.00% | 4.057e-3 s |
| Process 3 | 0.00% | 3.992e-3 s |
| Process 0 | 0.00% | 1.988e-3 s |
| Process 7 | 0.00% | 1.897e-3 s |
| Process 5 | 0.00% | 1.519e-3 s |

Context-sensitive advice on typical performance patterns

# Message profile



Total Time [s] (Sender by Receiver)

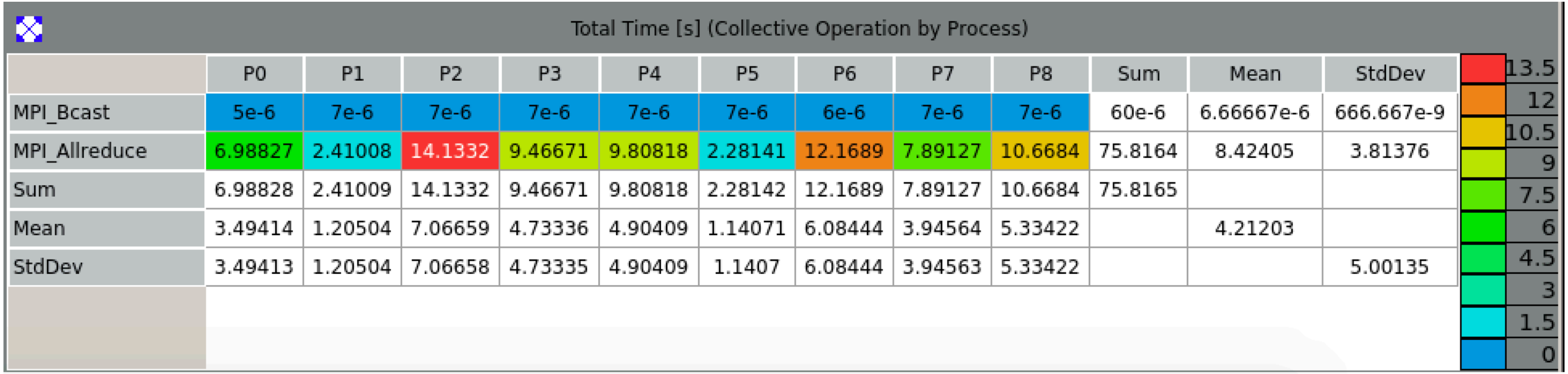
| | P0 | P1 | P2 | P3 | P4 | P5 | P6 | P7 | P8 | Sum | Mean | StdDev |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| P0 | | ... | | ... | ... | | | | | ... | ... | ... |
| P1 | ... | | ... | ... | ... | ... | | | | ... | ... | ... |
| P2 | | ... | | | ... | ... | | | | ... | ... | ... |
| P3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

Receiver

Sender

- Who sends how much to whom?
- How long does it take?
- Effective bandwidth?

# Collective operations profile

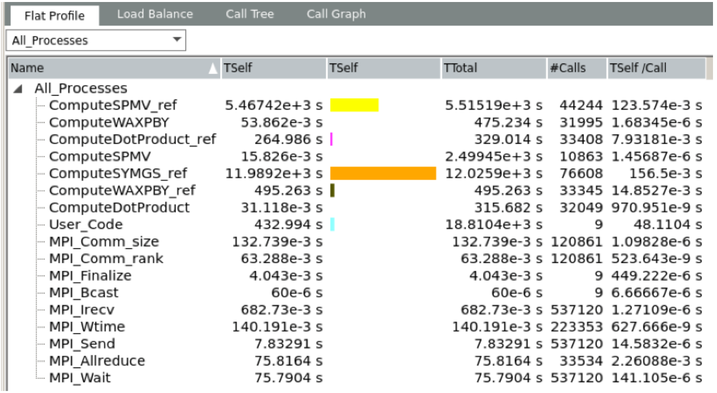


Total Time [s] (Collective Operation by Process)

| | 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 |

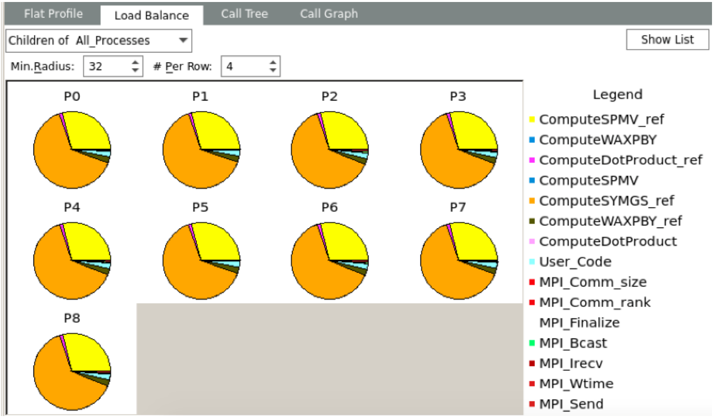
- Time spent in collective call
- Data volume sent/received

# Functions profile, call tree/graph, load imbalance



All_Processes

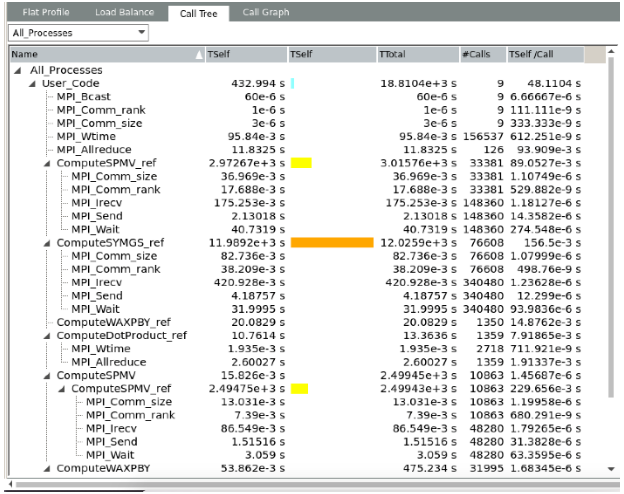
| Name | TSelf | TSelf | TTotal | #Calls | TSelf /Call |
|---|---|---|---|---|---|
| All_Processes | | | | | |
| ComputeSPMV_ref | 5.46742e+3 s | | 5.51519e+3 s | 44244 | 123.574e-3 s |
| ComputeWAXPBY | 53.862e-3 s | | 475.234 s | 31995 | 1.68345e-6 s |
| ComputeDotProduct_ref | 264.986 s | | 329.014 s | 33408 | 7.93181e-3 s |
| ComputeSPMV | 15.826e-3 s | | 2.49945e+3 s | 10863 | 1.45687e-6 s |
| ComputeSYMGS_ref | 11.9892e+3 s | | 12.0259e+3 s | 76608 | 156.5e-3 s |
| ComputeWAXPBY_ref | 495.263 s | | 495.263 s | 33345 | 14.8527e-3 s |
| ComputeDotProduct | 31.118e-3 s | | 315.682 s | 32049 | 970.951e-9 s |
| User_Code | 432.994 s | | 18.8104e+3 s | 9 | 48.1104 s |
| MPI_Comm_size | 132.739e-3 s | | 132.739e-3 s | 120861 | 1.09828e-6 s |
| MPI_Comm_rank | 63.288e-3 s | | 63.288e-3 s | 120861 | 523.643e-9 s |
| MPI_Finalize | 4.043e-3 s | | 4.043e-3 s | 9 | 449.222e-6 s |
| MPI_Bcast | 60e-6 s | | 60e-6 s | 9 | 6.66667e-6 s |
| MPI_Irecv | 682.73e-3 s | | 682.73e-3 s | 537120 | 1.27109e-6 s |
| MPI_Wtime | 140.191e-3 s | | 140.191e-3 s | 223353 | 627.666e-9 s |
| MPI_Send | 7.83291 s | | 7.83291 s | 537120 | 14.5832e-6 s |
| MPI_Allreduce | 75.8164 s | | 75.8164 s | 33534 | 2.26088e-3 s |
| MPI_Wait | 75.7904 s | | 75.7904 s | 537120 | 141.105e-6 s |



Children of All_Processes

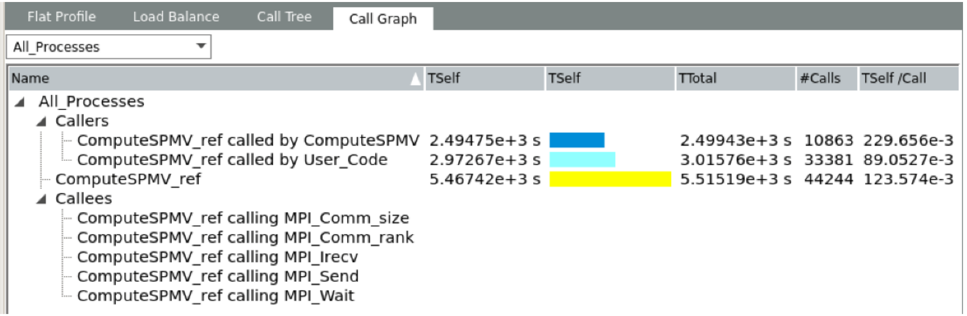
Show List

Min.Radius: 32   # Per Row: 4



All_Processes

```
Name                        TSelf          TSelf  TTotal        #Calls  TSelf /Call
◢ All_Processes
  ◢ User_Code               432.994 s             18.8104e+3 s       9  48.1104 s
      MPI_Bcast             60e-6 s               60e-6 s            9  6.66667e-6 s
      MPI_Comm_rank         1e-6 s                1e-6 s             9  111.111e-9 s
      MPI_Comm_size         3e-6 s                3e-6 s             9  333.333e-9 s
      MPI_Wtime             95.84e-3 s            95.84e-3 s    156537  612.251e-9 s
      MPI_Allreduce         11.8325 s             11.8325 s        126  93.909e-3 s
    ◢ ComputeSPMV_ref       2.97267e+3 s          3.01576e+3 s   33381  89.0527e-3 s
        MPI_Comm_size       36.969e-3 s           36.969e-3 s    33381  1.10749e-6 s
        MPI_Comm_rank       17.688e-3 s           17.688e-3 s    33381  529.882e-9 s
        MPI_Irecv           175.253e-3 s          175.253e-3 s  148360  1.18127e-6 s
        MPI_Send            2.13018 s             2.13018 s     148360  14.3582e-6 s
        MPI_Wait            40.7319 s             40.7319 s     148360  274.548e-6 s
    ◢ ComputeSYMGS_ref      11.9892e+3 s          12.0259e+3 s   76608  156.5e-3 s
        MPI_Comm_size       82.736e-3 s           82.736e-3 s    76608  1.07999e-6 s
        MPI_Comm_rank       38.209e-3 s           38.209e-3 s    76608  498.76e-9 s
        MPI_Irecv           420.928e-3 s          420.928e-3 s  340480  1.23628e-6 s
        MPI_Send            4.18757 s             4.18757 s     340480  12.299e-6 s
        MPI_Wait            31.9995 s             31.9995 s     340480  93.9836e-6 s
      ComputeWAXPBY_ref     20.0829 s             20.0829 s       1350  14.8762e-3 s
    ◢ ComputeDotProduct_ref 10.7614 s             13.3636 s       1359  7.91865e-3 s
        MPI_Wtime           1.935e-3 s            1.935e-3 s      2718  711.921e-9 s
        MPI_Allreduce       2.60027 s             2.60027 s       1359  1.91337e-3 s
    ◢ ComputeSPMV           15.826e-3 s           2.49945e+3 s   10863  1.45687e-6 s
      ◢ ComputeSPMV_ref     2.49475e+3 s          2.49943e+3 s   10863  229.656e-3 s
          MPI_Comm_size     13.031e-3 s           13.031e-3 s    10863  1.19958e-6 s
          MPI_Comm_rank     7.39e-3 s             7.39e-3 s      10863  680.291e-9 s
          MPI_Irecv         86.549e-3 s           86.549e-3 s    48280  1.79265e-6 s
          MPI_Send          1.51516 s             1.51516 s      48280  31.3828e-6 s
          MPI_Wait          3.059 s               3.059 s        48280  63.3595e-6 s
    ◢ ComputeWAXPBY         53.862e-3 s           475.234 s      31995  1.68345e-6 s
```



All_Processes

```
Name                                             TSelf         TSelf  TTotal        #Calls  TSelf /Call
◢ All_Processes
  ◢ Callers
      ComputeSPMV_ref called by ComputeSPMV      2.49475e+3 s         2.49943e+3 s   10863  229.656e-3
      ComputeSPMV_ref called by User_Code        2.97267e+3 s         3.01576e+3 s   33381  89.0527e-3
    ComputeSPMV_ref                              5.46742e+3 s         5.51519e+3 s   44244  123.574e-3
  ◢ Callees
      ComputeSPMV_ref calling MPI_Comm_size
      ComputeSPMV_ref calling MPI_Comm_rank
      ComputeSPMV_ref calling MPI_Irecv
      ComputeSPMV_ref calling MPI_Send
      ComputeSPMV_ref calling MPI_Wait
```

# Options for taking traces

- Caveat: Tracing can generate vast amounts of data!
- Compiler switches (only works with legacy Intel compiler and wrappers [mpiicc, mpiicpc, mpiifort])
  - `-trace` # record MPI calls (also possible with mpirun/mpiexec)
  - `-tcollect -trace` # record MPI and user code function calls
    # potential of large overhead and large trace size
  - `-tcollect-filter=func.txt -tcollect –trace` # filter file

func.txt example

```
'.*' OFF
'.*ComputeDotProduct.*' ON
'.*ComputeSYMGS.*' ON
'.*ComputeSPMV.*' ON
'.*ComputeWAXPBY.*' ON
```

# More (important) configuration options

| Environment variable | Default | Description |
|---|---|---|
| **VT_FLUSH_PREFIX** | ... depends | directory for temporary flush files |
| **VT_LOGFILE_PREFIX** | current working directory | directory for physical trace information files |
| **VT_LOGFILE_FORMAT** | STF | `SINGLESTF`: rolls all trace files into one file (.single.stf) |
| **VT_LOGFILE_NAME** | ${binary}.stf | control the name for the trace file |
| VT_MEM_BLOCKSIZE | 64 KB | trace data in chunks of main memory |
| VT_MEM_FLUSHBLOCKS | 1024 | flushing is started when the number of blocks in memory exceeds this threshold |
| VT_MEM_MAXBLOCKS | 1024 | maximum number of blocks in main memory, if exceed the application is stopped until AUTOFLUSH/ MEM-OVERWRITE/ stop recording trace info |
| VT_CONFIG_RANK | 0 | control the process that reads and parses the configuration file |

- Avoid rapid-fire dumping trace data into shared filesystems!
- Your fellow cluster users will hate you for it.

# Alternatives

- ITAC is deprecated by Intel and will not be further developed (as of 2025)
  - Intel recommends VTune as a replacement, but this is not competitive

- Other tools with similar functionality
  - Vampir (commercial, scalable) https://vampir.eu/
  - Scalasca (for highly scalable programs, no trace view) https://www.scalasca.org/
  - Paraver https://tools.bsc.es/paraver

  - Jumpshot
    Don’t even bother.