Performance Metrics & Measurements

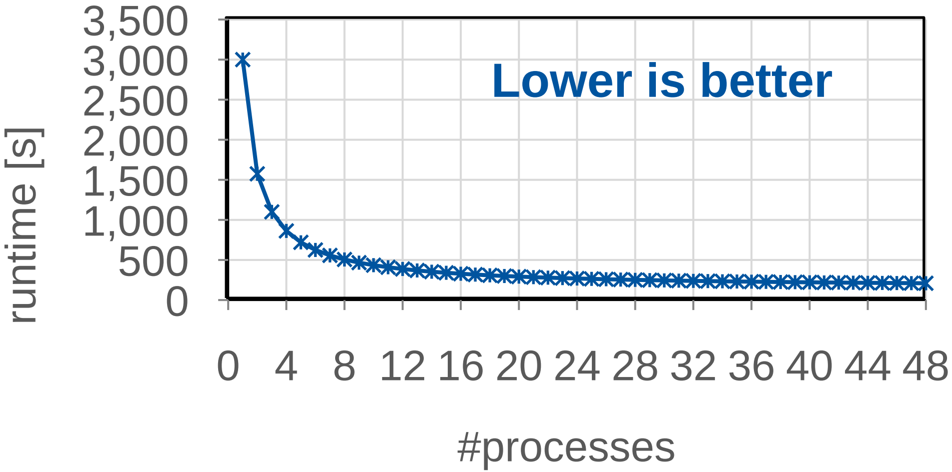
HPC Intro 2025

Felix Tomski

Performance Metrics

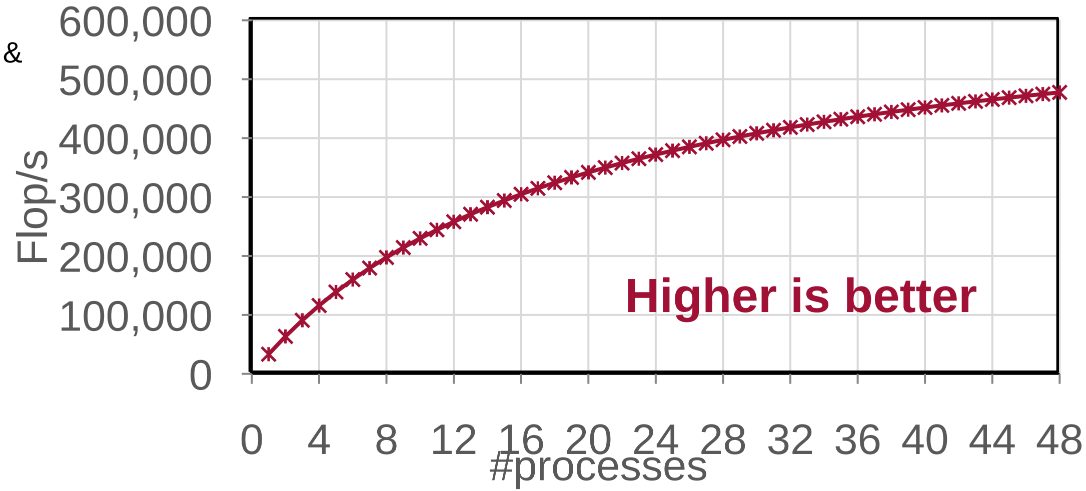
Runtime

- HPC is about reducing the runtime of an application & enabling the simulation of large data sets
  - Serial performance tuning
  - Parallel performance tuning
- Time metrics
  - **Wallclock time**: elapsed real time (such as a clock on the wall)
  - CPU time: accumulated time of all CPUs (cores) executing the application (instructions)
  - Derived → core-h: program run of 1 hour on 4 cores = 4 core-h
- Remarks
  - Complete application time
  - Kernel time
- Getting the runtime
  - Timers in code, or tools



Floating-Point Operations per Second

- Floating-point operations per second: Flop/s
  - Double precision (64-bit, e.g. double)
  - Single precision (32-bit, e.g. float)
  - Half precision (16-bit)
- Remarks
  - Typical for algorithm
  - Avoid „Macho-Flop/s“
  - Consider costs, e.g. energy consumption & efficiency
- Getting Flop/s
  - Runtime measurement
  - Theoretical calculation (algorithm)
  - Tools
- Typical application: Linpack (Top500)



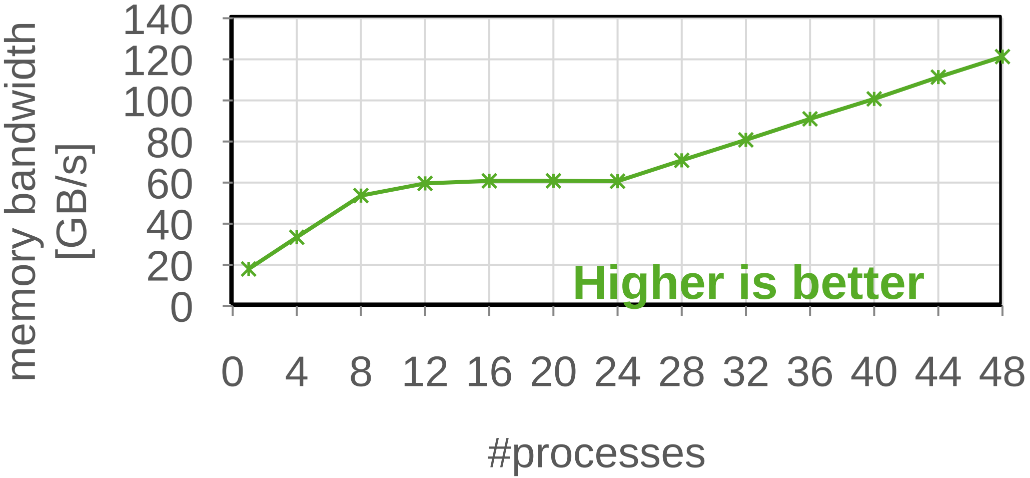
Bandwidth

- Bandwidth (throughput) in GB/s
  - Main memory bandwidth (node granularity)
  - Cache bandwidth (socket / core granularity)
  - Network bandwidth (cluster granularity)

- Remarks
  - Many HPC applications are bound by memory bandwidth
  - Consider NUMA effects on node

- Getting GB/s
  - Runtime measurement
  - Theoretical calculation of Bytes
  - Tools

- Typical application: STREAM



Speedup

- Ratio between runtime t of some reference version ref and the (relevant) application version app

  - t is wallclock time

  - „app is S times faster than ref“:

$$\textit{Speedup}\ S = \frac{t_{ref}}{t_{app}}$$

- Remarks
  - Kernel speedup
  - Application speedup

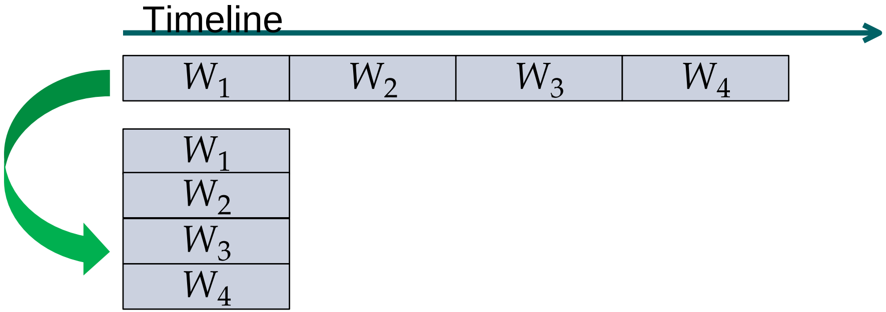
- Comparison examples
  - GPU vs. CPU version: $S = \frac{t_{CPU}}{t_{GPU}}$
  - Parallel vs. serial version: $S = \frac{t_{serial}}{t_{parallel}}$

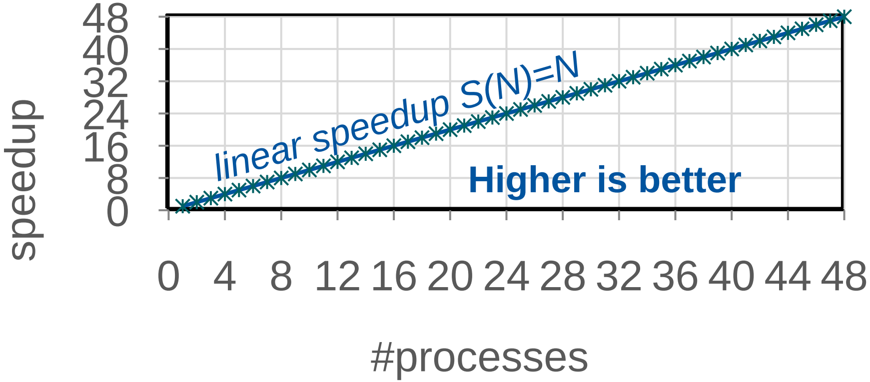
Strong Scaling

- In parallel computing: Indicator for relative performance improvement
- Assumption
  - Variation of number of processes N
  - Keep data set fixed

$$Speedup\ S(N) = \frac{t(1)}{t(N)}$$

- Ideal situation: All work is perfectly parallelizable —> Linear speedup
  - In general: Upper bound for parallel execution of programs

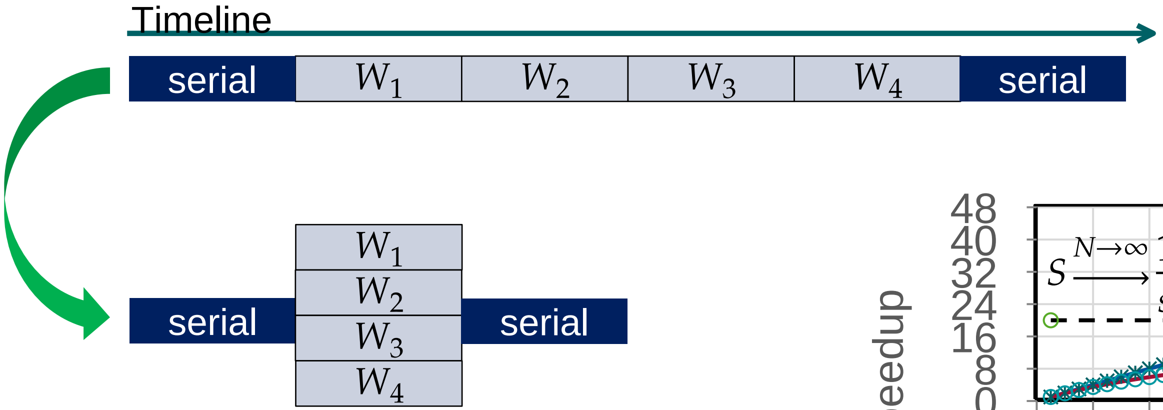




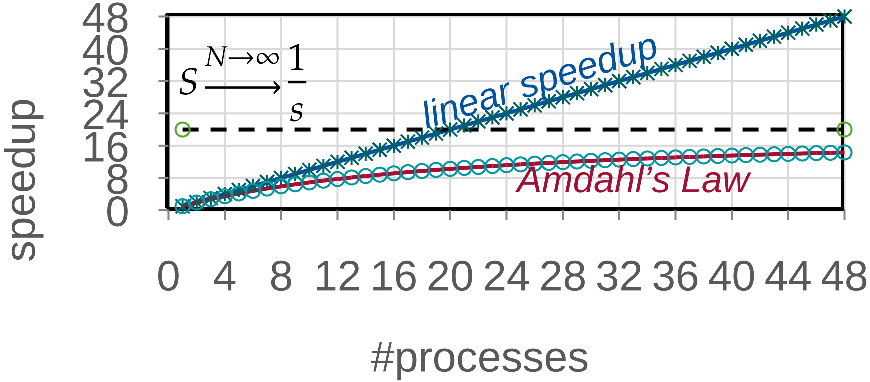
Strong Scaling

- Real-world limitations of scalability: serial parts in code
    - Serial portion s, parallel portion p
    - Refer to “Amdahl‘s Law”

$$\mathit{Speedup}\ S(N) = \frac{1}{s + \frac{p}{N}}$$

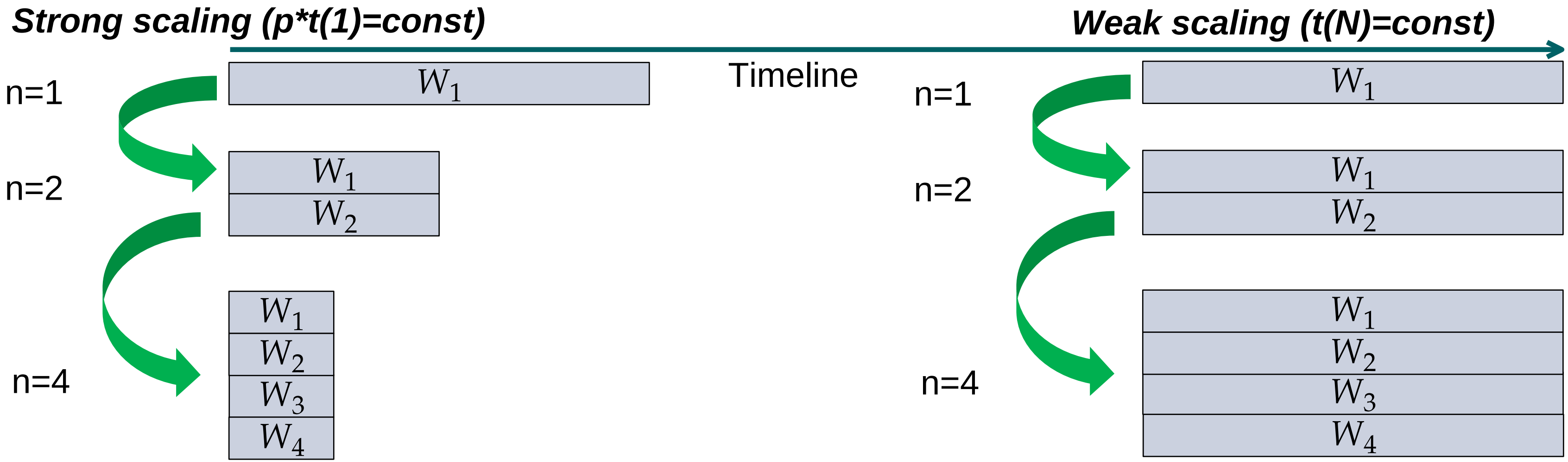


- Remarks
    - In reality, no task is perfectly parallelizable



Weak Scaling

- Why do we have big clusters if scalability is limited by Amdahl's Law?
    - Use bigger problem sizes!
- Assumption
    - Variation of number of processes N
    - Data set size changes with number of processes (e.g., doubling)

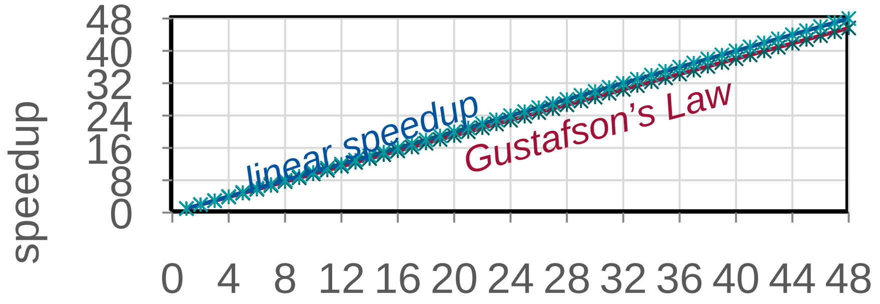


Weak Scaling

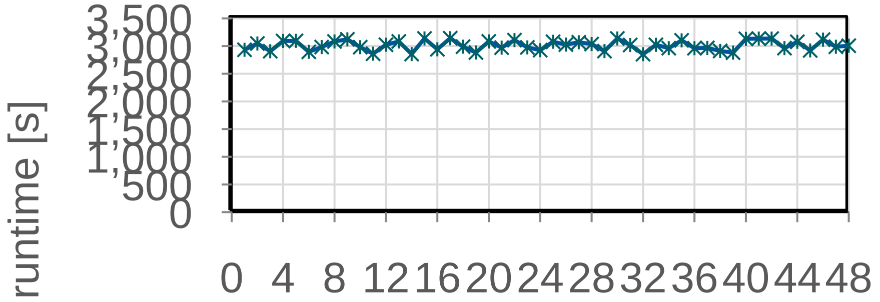
- Gustafson's Law

$$\mathit{Speedup}\ S(N) = Np + s$$

- Perfect weak scaling: roughly constant runtime among varying #processes



#processes



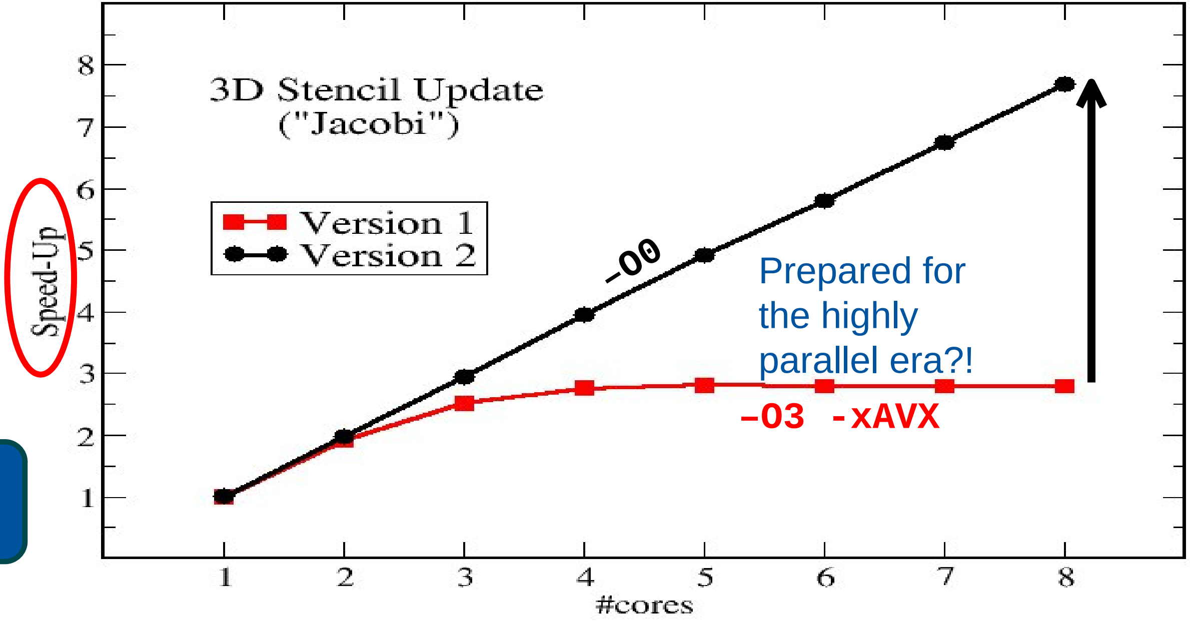
#processes

Performance Measurements

Scalability Myth: Code scalability is the key issue

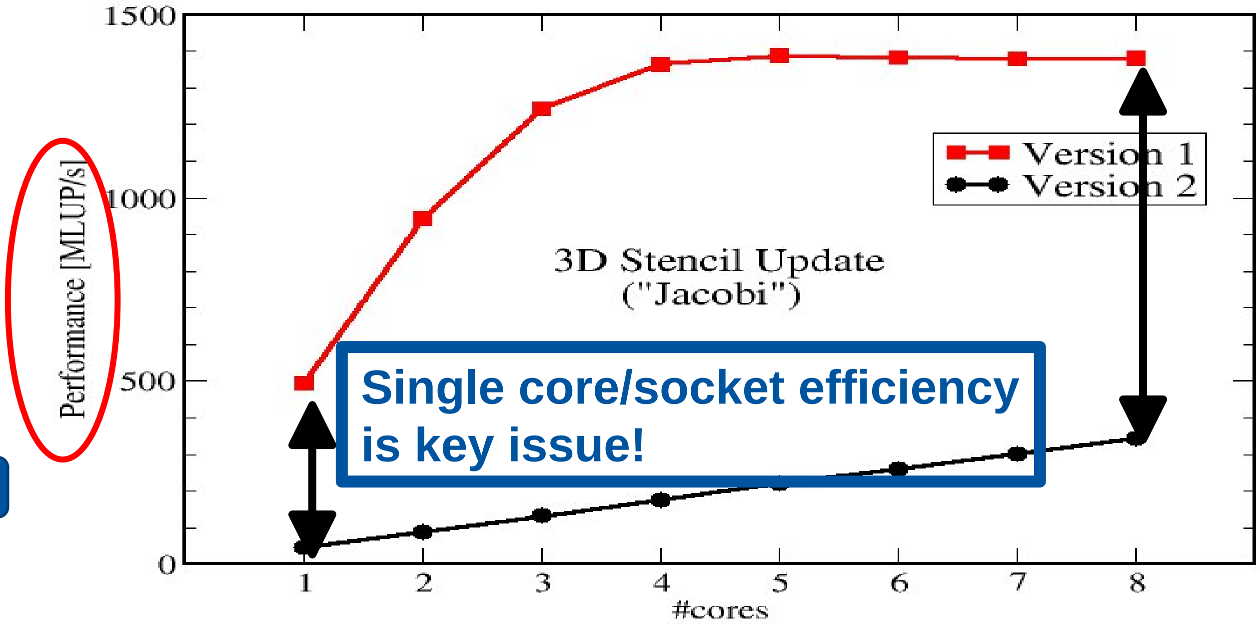
Changing only the compile options makes this code scalable on an 8-core chip

Parallel program is X times faster than serial program.



Prepared for the highly parallel era?!

Scalability Myth: Code scalability is the key issue



Single core/socket efficiency is key issue!

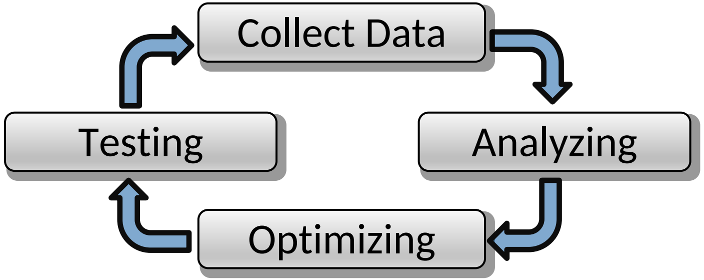
Absolute performance

First goal should be optimizing serial code before conduction parallel code tuning

Courtesy of Erlangen Regional Computing Center (RRZE)

Tuning Cycle

1. Find out <u>where</u> most of the <u>runtime</u> is spent
    - Usually starts with a hotspot analysis
2. Find out <u>why</u> most of the <u>runtime</u> is spent there (analyze data)
    - Determine which factors stall performance (e.g. by hardware counters)
3. Optimize your code to get a decreased runtime
4. Test the correctness of code & its performance
    - Use appropriate problem size
    - Start with step (1) if test not successful



Preamble: Performance Engineering

- Performance engineering depends on different levels

Cluster

Node

Core

Accelerator

- Some architectural levels may be shared resources (even in batch mode)
  - Example: Processes from different users may run on the same node
  - Possible impact: shared cache und memory channel utilization
  - If necessary: request node exclusively

```
#SBATCH --exclusive
```

- Efficient usage of hardware resources important
  - If you use exclusive nodes, try to leverage the available parallelism (e.g., multiple cores)
  - Otherwise: idling hardware, and money not well invested
  - Metrics, e .g., productivity $\frac{app.runs}{cost\ (TCO)}$, efficiency $\varepsilon(N) = \frac{S(N)}{N}$

Foto: Dieter Both, Bull GmbH

Preamble: Performance Engineering

Performance measurements and analysis heavily relies on a good test setup

- Data set
  - Find a *representative data set* (i.e., algorithmic & performance similarity to real data set)
  - Choose problem size *not too small* since performance behavior changes with the size of the memory consumption
  - Choose problem size *not too large* since tests need to be done quite often to compare tuning steps
  - To guarantee correct simulation results, use (automatic) correctness checks
- Interpreting performance data
  - Try to establish a “stable” testing environment to get *repeatable performance results* (e.g., use thread binding or exclusively-reserved nodes)
  - Repeat application runs to eliminate outlier behavior (if possible)
  - Use appropriate statistical data analysis of performance results (e.g., mean, standard deviation, significance)

Preamble: Hotspots

- A Hotspot is a source code region where a significant part of the runtime is spent.

90/10 law

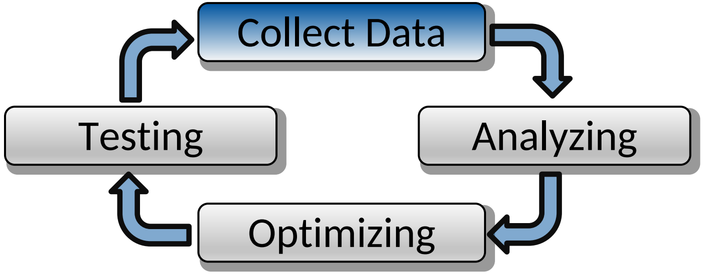
90% of the runtime in a program is spent in 10% of the code.

- Hotspots can indicate where to start with serial optimization or shared memory parallelization.

- Use a tool to identify hotspots. In many cases the results are surprising.

Collection of Performance Data

- Performance analysis tools are highly recommended to easily identify hotspots & collect performance data
  - Alternative: manual timing of code parts (limited)



Recording techniques

- Profiling
  - Retrieves summary information of a program's runtime behavior
  - Applies "instrumentation" or "sampling" for triggering
- Tracing
  - Time-ordered list of all the events that were recorded during program flow (event trace)

	Tracing	Profiling
Precision	exact information	accumulated information
Overhead	higher overhead (depends on #events)	lower runtime overhead
Space requirements	easily hundreds of MB or GB for larger applications (depends on #events)	smaller amount of space needed normally some MB

Collection of Performance Data: Function Profiling

Hotspot is function `f1`

gprof example

```
 %   cumulative   self              self     total
time   seconds   seconds    calls  ms/call  ms/call  name
86.65      0.62      0.62        1   615.21   615.21  f1
 9.94      0.69      0.07        1    70.60   685.81  f2
 4.26      0.72      0.03        1    30.26    30.26  f4
 0.00      0.72      0.00        1     0.00   615.21  f3
```

% of overall program runtime used exclusively by this function

#seconds used by this function (exclusive)

#calls of this function

Average number of ms per call that were spent in this function (inclusive)

Average number of ms per call that were spent in this function (exclusive)

- Profile information per function
  - Exclusive (not counting any callees of the function) or inclusive (including callees of function) runtimes
  - Flat profile or callgraph profile

- Profiling tools, e.g.
  - gprof (uses instrumenation + sampling)
  - Intel VTune Amplfier XE

gprof @ RWTH

Compile with -pg:

```
$ gcc –pg test.c –o a.out
```

Execute (will collect data in gmon.out)

```
$ ./a.out
```

Generate report

```
$ gprof a.out gmon.out > profile-data.txt
```

View report

```
$ cat profile-data.txt
```

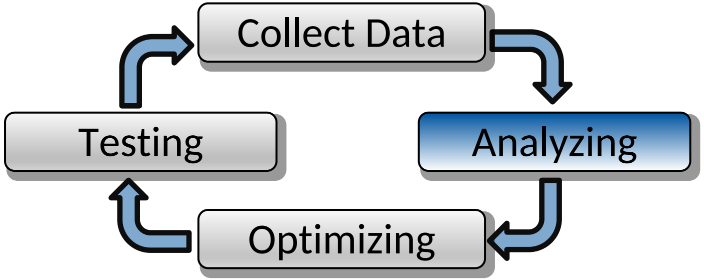
Intel VTune @ RWTH

```
$ module load VTune
$ vtune-gui
```

or use command line version

Performance Analysis

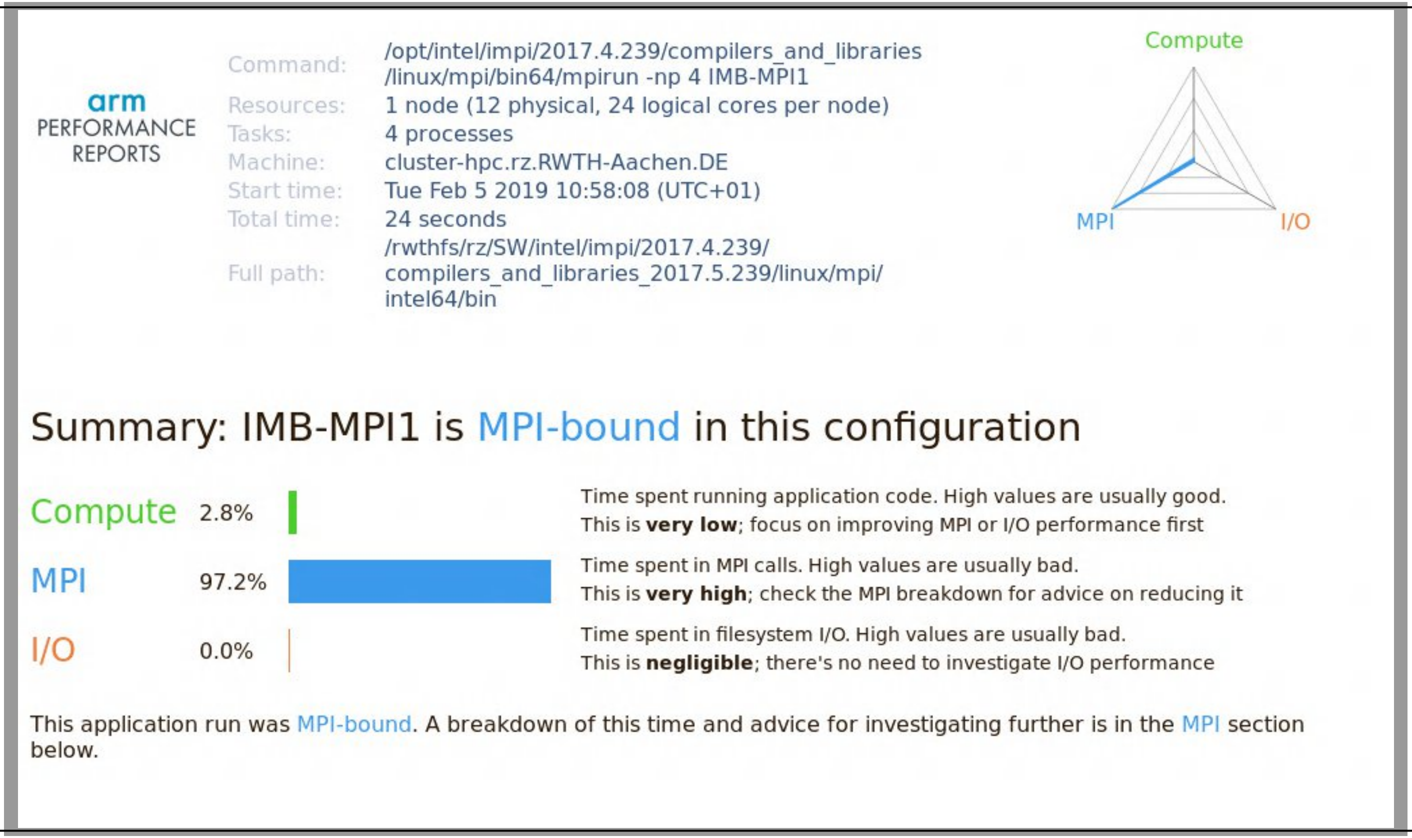
- Based on hardware performance counters
  - Special registers as part of hardware architecture
  - Count hardware-related information
  - Examples
    - Memory/ cache accesses
    - Floating-point operations
    - Cycles per instructions (CPI)
- Evaluations, e.g.,
  - Concurrency
  - Load Imbalance
  - Metrics: https://hpc-wiki.info/hpc/ProPE_PE_Process
- Performance analysis tools, e.g.,
  - Intel VTune Amplifier XE (medium-level)
  - LIKWID (low-level)
  - ARM/Linaro Performance Reports (high-level)
  - Intel Performance Snapshot (high-level)



LIKWID @ RWTH

```
$ module load GCC/11.3.0
$ module load likwid
$ likwid-perfctr <…>
```

Performance Analysis: Getting an High-level Overview



Command: /opt/intel/impi/2017.4.239/compilers_and_libraries /linux/mpi/bin64/mpirun -np 4 IMB-MPI1
Resources: 1 node (12 physical, 24 logical cores per node)
Tasks: 4 processes
Machine: cluster-hpc.rz.RWTH-Aachen.DE
Start time: Tue Feb 5 2019 10:58:08 (UTC+01)
Total time: 24 seconds
Full path: /rwthfs/rz/SW/intel/impi/2017.4.239/ compilers_and_libraries_2017.5.239/linux/mpi/ intel64/bin

Summary: IMB-MPI1 is MPI-bound in this configuration

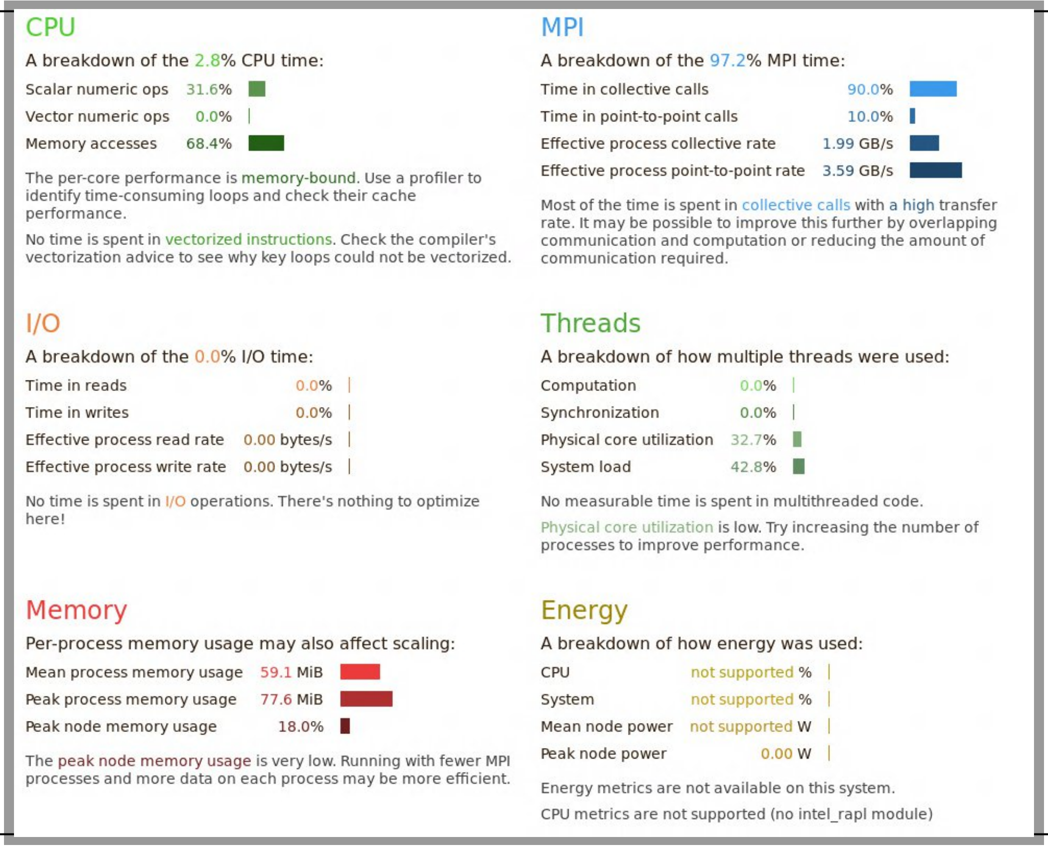
Compute 2.8% — Time spent running application code. High values are usually good. This is **very low**; focus on improving MPI or I/O performance first

MPI 97.2% — Time spent in MPI calls. High values are usually bad. This is **very high**; check the MPI breakdown for advice on reducing it

I/O 0.0% — Time spent in filesystem I/O. High values are usually bad. This is **negligible**; there's no need to investigate I/O performance

This application run was MPI-bound. A breakdown of this time and advice for investigating further is in the MPI section below.

Performance Analysis: Getting an High-level Overview



CPU

A breakdown of the 2.8% CPU time:

Scalar numeric ops	31.6%
Vector numeric ops	0.0%
Memory accesses	68.4%

The per-core performance is memory-bound. Use a profiler to identify time-consuming loops and check their cache performance.

No time is spent in vectorized instructions. Check the compiler's vectorization advice to see why key loops could not be vectorized.

MPI

A breakdown of the 97.2% MPI time:

Time in collective calls	90.0%
Time in point-to-point calls	10.0%
Effective process collective rate	1.99 GB/s
Effective process point-to-point rate	3.59 GB/s

Most of the time is spent in collective calls with a high transfer rate. It may be possible to improve this further by overlapping communication and computation or reducing the amount of communication required.

I/O

A breakdown of the 0.0% I/O time:

Time in reads	0.0%
Time in writes	0.0%
Effective process read rate	0.00 bytes/s
Effective process write rate	0.00 bytes/s

No time is spent in I/O operations. There's nothing to optimize here!

Threads

A breakdown of how multiple threads were used:

Computation	0.0%
Synchronization	0.0%
Physical core utilization	32.7%
System load	42.8%

No measurable time is spent in multithreaded code.

Physical core utilization is low. Try increasing the number of processes to improve performance.

Memory

Per-process memory usage may also affect scaling:

Mean process memory usage	59.1 MiB
Peak process memory usage	77.6 MiB
Peak node memory usage	18.0%

The peak node memory usage is very low. Running with fewer MPI processes and more data on each process may be more efficient.

Energy

A breakdown of how energy was used:

CPU	not supported %
System	not supported %
Mean node power	not supported W
Peak node power	0.00 W

Energy metrics are not available on this system.

CPU metrics are not supported (no intel_rapl module)

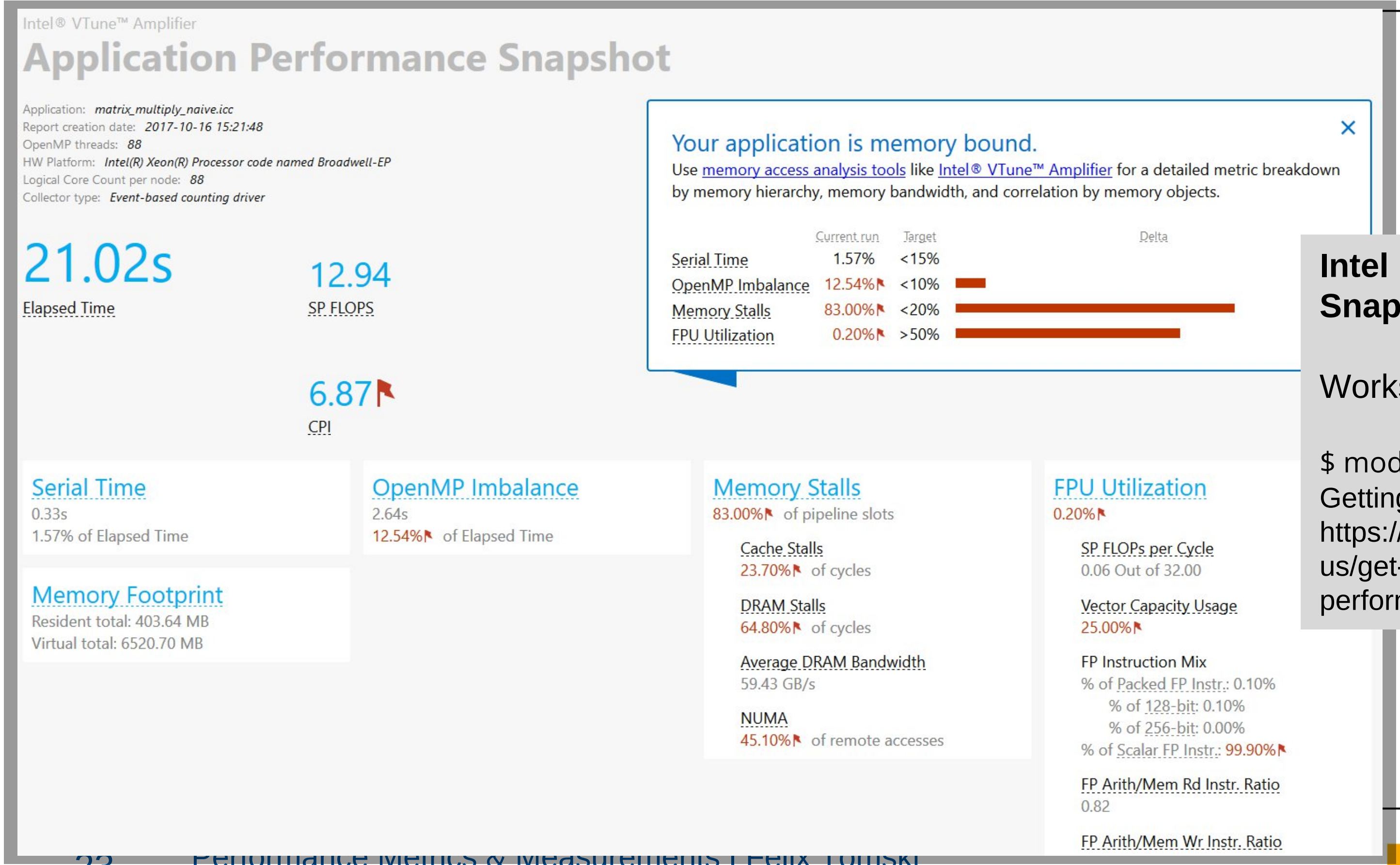
ARM Performance Reports @ RWTH

Limited number of licenses

Just execute your application with perf-report:

```
$ module load ARMForge/22.0.4
$ perf-report $MPI_BINDIR/mpirun -np 4  a.out
$ firefox IMB-MPI1_4p_1n_1t_2019-02-05_10-58.html
```

Performance Analysis: Getting an High-level Overview



Intel® VTune™ Amplifier

Application Performance Snapshot

Application: *matrix_multiply_naive.icc*
Report creation date: *2017-10-16 15:21:48*
OpenMP threads: *88*
HW Platform: *Intel(R) Xeon(R) Processor code named Broadwell-EP*
Logical Core Count per node: *88*
Collector type: *Event-based counting driver*

21.02s
Elapsed Time

12.94
SP FLOPS

6.87
CPI

Your application is memory bound.
Use memory access analysis tools like Intel® VTune™ Amplifier for a detailed metric breakdown by memory hierarchy, memory bandwidth, and correlation by memory objects.

	Current run	Target	Delta
Serial Time	1.57%	<15%	
OpenMP Imbalance	12.54%	<10%	
Memory Stalls	83.00%	<20%	
FPU Utilization	0.20%	>50%	

Serial Time
0.33s
1.57% of Elapsed Time

OpenMP Imbalance
2.64s
12.54% of Elapsed Time

Memory Footprint
Resident total: 403.64 MB
Virtual total: 6520.70 MB

Memory Stalls
83.00% of pipeline slots
Cache Stalls
23.70% of cycles
DRAM Stalls
64.80% of cycles
Average DRAM Bandwidth
59.43 GB/s
NUMA
45.10% of remote accesses

FPU Utilization
0.20%
SP FLOPs per Cycle
0.06 Out of 32.00
Vector Capacity Usage
25.00%
FP Instruction Mix
% of Packed FP Instr.: 0.10%
% of 128-bit: 0.10%
% of 256-bit: 0.00%
% of Scalar FP Instr.: 99.90%
FP Arith/Mem Rd Instr. Ratio
0.82
FP Arith/Mem Wr Instr. Ratio

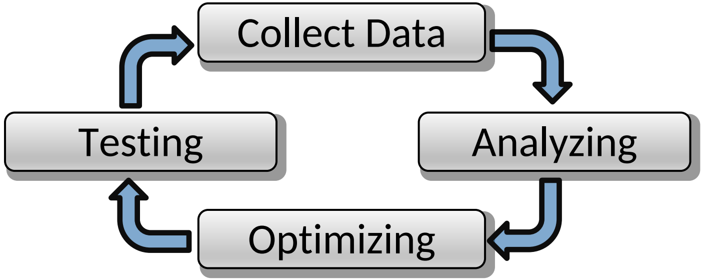
Intel Performance Snapshot @ RWTH

Works only with Intel MPI

$ module load VTune
Getting started:
https://software.intel.com/en-us/get-started-with-application-performance-snapshot

Summary

- HPC goal: reduce application runtime
  - Serial and parallel performance optimization
- Performance metrics
  - Absolute metrics: runtime, Flop/s, GB/s
  - Relative metrics: speedup (strong/ weak scaling)
- Performance measurements
  - Use requested HPC resources efficiently
  - Start with simple performance measurements like hotspot analyses and then focus on these hotspots
  - Performance analysis tools help to collect and analyze performance data



Performance Engineering: Tuning Cycle