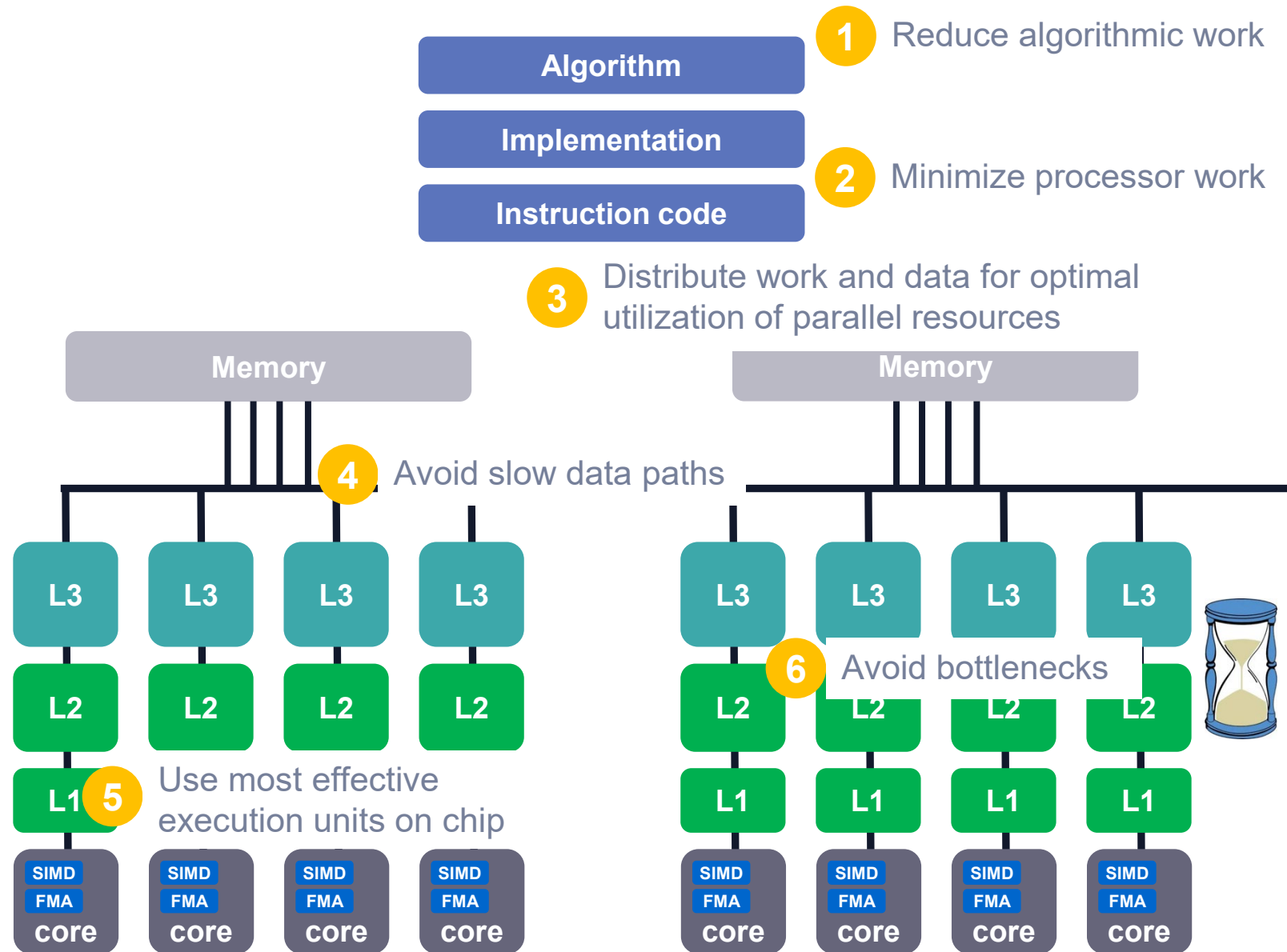


Performance Engineering

Basic skills and knowledge



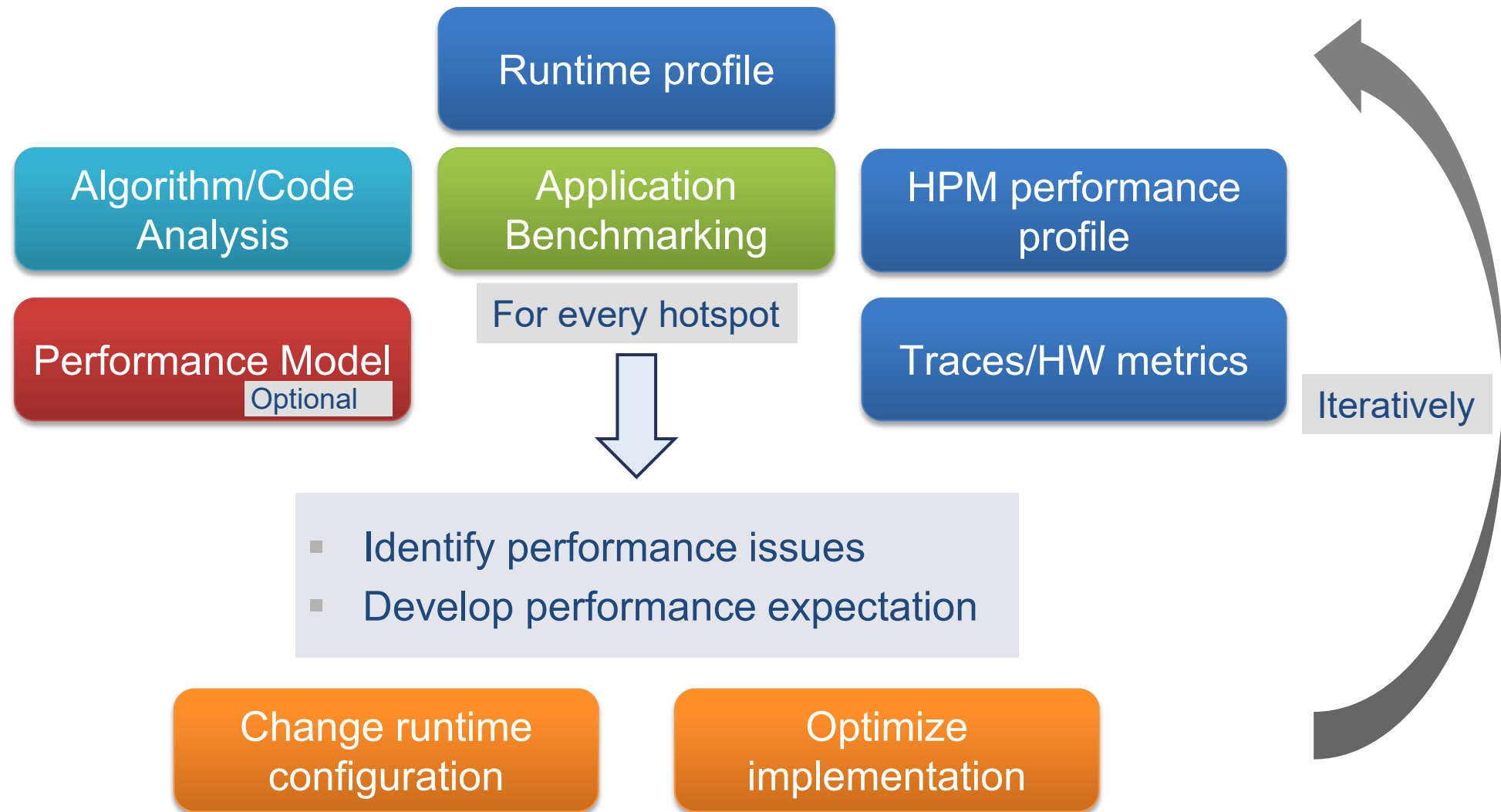
Optimizing code: The big Picture



Basic advice

- Software is complex
 - Keep it simple stupid! The simpler solution is the better solution.
 - Don't be afraid to refactor your code. Extract numerical code into simple kernels.
- Numerical algorithms on parallel computers is a nightmare:
 - To trust the results of a numerical computation is a painful journey.
 - Extensive test harness with automatic validation on different levels.
- Time to solution is all that matters!
 - Don't be guided by anything else: cache misses, speedup, flop rates
- Runtime contributions only matter if they appear on the critical path!
 - Critical path is the series of runtime contributions that do not overlap and form the total runtime

Performance Engineering process



Runtime profiling with gprof

Instrumentation based with gprof

Compile with **-pg** switch:

```
icc -pg -O3 -c myfile1.c
```

Execute the application. During execution a file **gmon.out** is generated.

Analyze the results with:

```
gprof ./a.out | less
```

The output contains three parts: A flat profile, the call graph, and an alphabetical index of routines.

The flat profile is what you are usually interested in.

Runtime profile with gprof: Flat profile

Time spent in routine itself

How often was it called

How much time was spent per call

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
66.86	26.14	26.14	502	0.05	0.05	ForceLJ::compute(Atom&, Neighbor&, Comm&, int)
30.77	38.17	12.03	26	0.46	0.46	Neighbor::build(Atom&)
1.43	38.73	0.56	1	0.56	38.46	Integrate::run(Atom&, Force*, Neighbor&, Comm&, Thermo&, Timer&)
0.36	38.87	0.14	2850	0.00	0.00	Atom::pack_comm(int, int*, double*, int*)
0.15	38.93	0.06	2850	0.00	0.00	Atom::unpack_comm(int, int, double*)
0.13	38.98	0.05	26	0.00	0.00	Atom::pbc()
0.10	39.02	0.04				__intel_sse3_rep_memcpy
0.08	39.05	0.03	25	0.00	0.00	Atom::sort(Neighbor&)
0.08	39.08	0.03	1	0.03	0.03	create_atoms(Atom&, int, int, int, double)
0.05	39.10	0.02	26	0.00	0.00	Comm::borders(Atom&)
0.00	39.10	0.00	1221559	0.00	0.00	Atom::pack_border(int, double*, int*)
0.00	39.10	0.00	1221559	0.00	0.00	Atom::unpack_border(int, double*)
0.00	39.10	0.00	131072	0.00	0.00	Atom::addatom(double, double, double, double, double, double)
0.00	39.10	0.00	1025	0.00	0.00	Timer::stamp(int)
0.00	39.10	0.00	502	0.00	0.00	Thermo::compute(int, Atom&, Neighbor&, Force*, Timer&, Comm&)
0.00	39.10	0.00	500	0.00	0.00	Timer::stamp()
0.00	39.10	0.00	475	0.00	0.00	Comm::communicate(Atom&)
0.00	39.10	0.00	26	0.00	0.00	Comm::exchange(Atom&)
0.00	39.10	0.00	25	0.00	0.00	Timer::stamp_extra_stop(int)
0.00	39.10	0.00	25	0.00	0.00	Timer::stamp_extra_start()
0.00	39.10	0.00	25	0.00	0.00	Neighbor::binatoms(Atom&, int)
0.00	39.10	0.00	7	0.00	0.00	Timer::barrier_stop(int)
0.00	39.10	0.00	1	0.00	0.00	create_box(Atom&, int, int, int, double)
0.00	39.10	0.00	1	0.00	0.00	create_velocity(double, Atom&, Thermo&)

Output is sorted according to total time spent in routine.

Sampling-based runtime profile with perf

Call executable with perf:

```
perf record -g ./a.out
```

Analyze the results with:

```
perf report
```

Advantages vs. gprof:

- Works on any binary without recompile
- Also captures OS and runtime symbols

```
Samples: 30K of event 'cycles:uppp', Event count (approx.): 20629160088
Overhead  Command      Shared Object      Symbol
 64.19%   miniMD-ICC    miniMD-ICC          [.] ForceLJ::compute
 31.54%   miniMD-ICC    miniMD-ICC          [.] Neighbor::build
  1.47%   miniMD-ICC    miniMD-ICC          [.] Integrate::run
  0.67%   miniMD-ICC    [kernel]            [k] irq_return
  0.40%   miniMD-ICC    miniMD-ICC          [.] Atom::pack_comm
  0.35%   mpiexec       [kernel]            [k] sysret_check
  0.21%   miniMD-ICC    miniMD-ICC          [.] create_atoms
  0.18%   miniMD-ICC    miniMD-ICC          [.] Atom::unpack_comm
  0.15%   miniMD-ICC    [kernel]            [k] sysret_check
  0.15%   miniMD-ICC    miniMD-ICC          [.] Comm::borders
  0.10%   miniMD-ICC    miniMD-ICC          [.] __intel_ssse3_rep_memcpy
  0.09%   miniMD-ICC    miniMD-ICC          [.] Atom::sort
  0.07%   miniMD-ICC    miniMD-ICC          [.] Neighbor::binatoms
```

Command line version of Intel Amplifier

Works out of the box for MPI/OpenMP parallel applications.

Example usage with MPI:

```
mpirun -np 2 amplxe-cl -collect hotspots -result-dir myresults -- a.out
```

- Compile with debugging symbols
- Can also resolve inlined C++ routines
- Many more collect modules available including hardware performance monitoring metrics

```
Elapsed Time: 8.650s
  CPU Time: 8.190s
    Effective Time: 8.190s
      Idle: 0.020s
      Poor: 8.170s
      Ok: 0s
      Ideal: 0s
      Over: 0s
    Spin Time: 0s
    Overhead Time: 0s
  Total Thread Count: 2
  Paused Time: 0s
```

Top Hotspots Function	Module	CPU Time
-----	-----	-----
ForceLJ::compute_fullneigh	miniMD-ICC	4.940s
Neighbor::build	miniMD-ICC	2.820s
Integrate::finalIntegrate	miniMD-ICC	0.100s
Integrate::initialIntegrate	miniMD-ICC	0.060s
__intel_ssse3_rep_memcpy	miniMD-ICC	0.040s
[Others]	N/A	0.230s

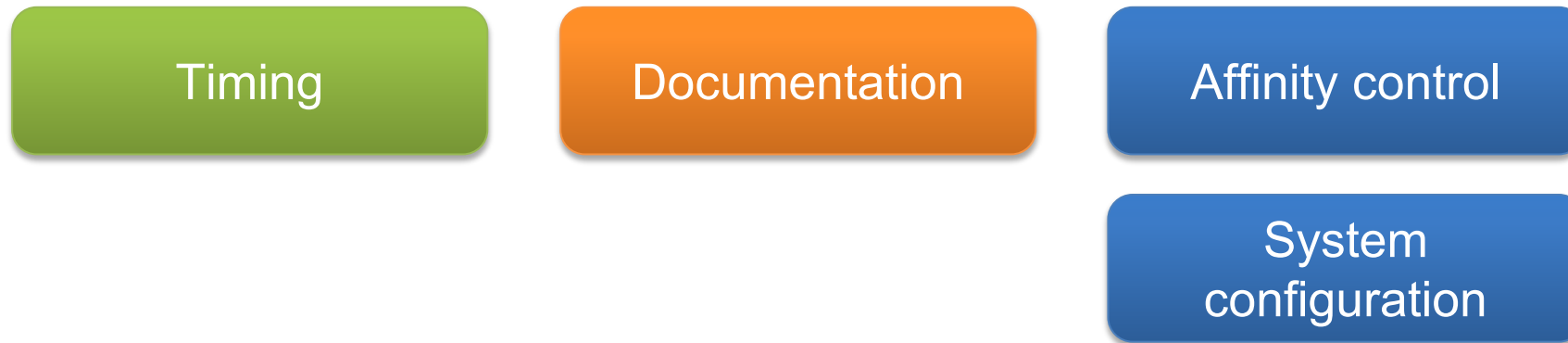
Application benchmarking preparation

- Discuss and prepare **relevant** benchmark **test case(s)**
 - Short turnaround time
 - **Representative of real production runs**
- For long term multi-site PE projects you may extract a **proxy application**
 - Simplified version of app (or a part of it) that still captures the relevant performance behavior
- Define an application-specific **performance metric**
 - Should avoid “trivial” dependencies on problem parameters (see later)
 - Common choice: **Useful work performed per time unit**

Application benchmarking components

Performance measurements must be accurate, deterministic and reproducible.

Components for application benchmarking:



Always run benchmarks on an **exclusive system!**

Timing within program code

For benchmarking, an accurate wall-clock timer (end-to-end stop watch) is required:

- `clock_gettime()` POSIX compliant timing function
- `MPI_Wtime()` and `omp_get_wtime()` Standardized programming-model-specific timing routines for MPI and OpenMP

```
#include <stdlib.h>
#include <time.h>

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

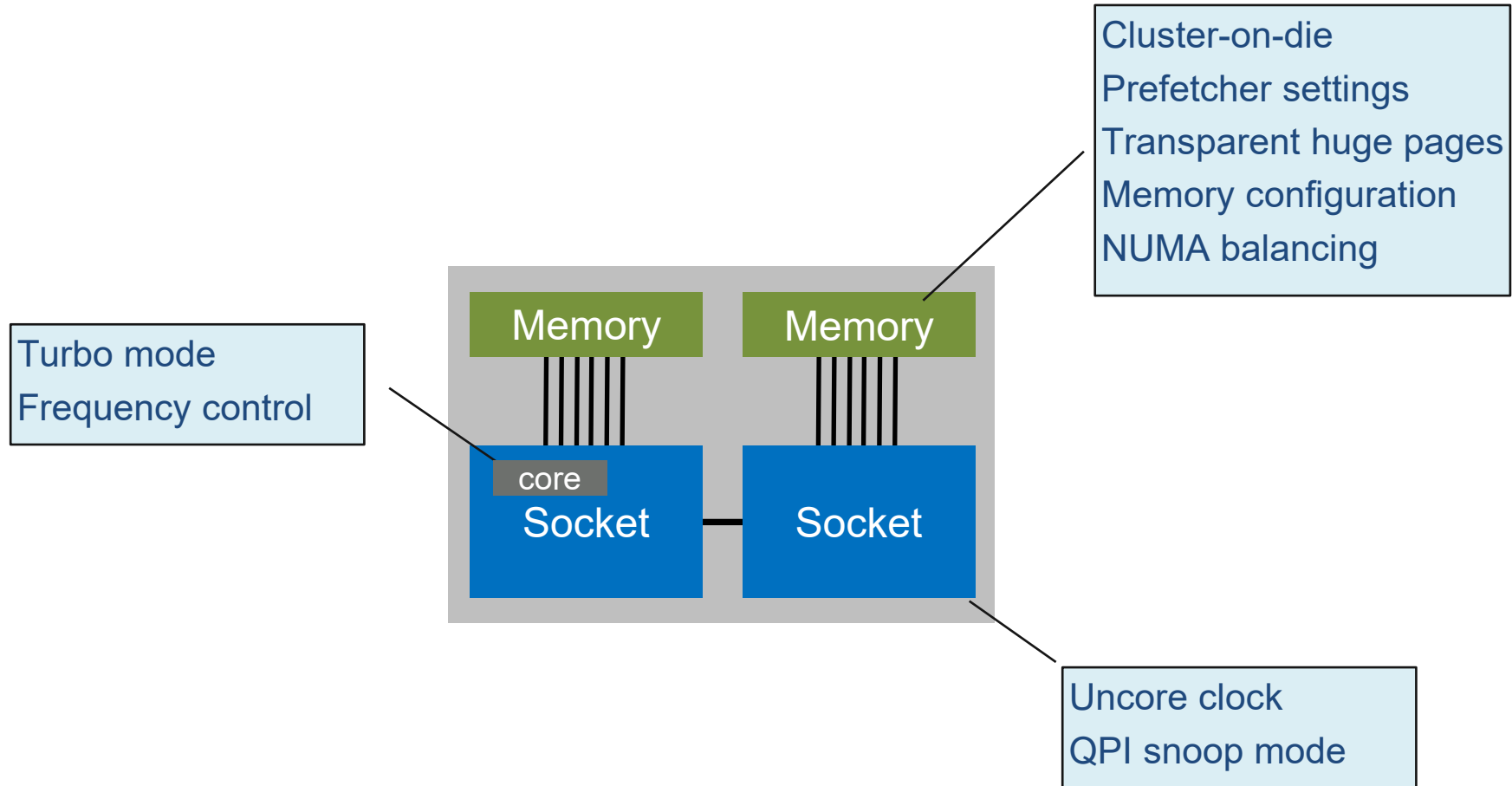
Usage:

```
double S, E;
S = getTimeStamp();
/* measured code region */
E = getTimeStamp();
return E-S;
```



<https://github.com/RRZE-HPC/TheBandwidthBenchmark/>

System configuration and clock frequency



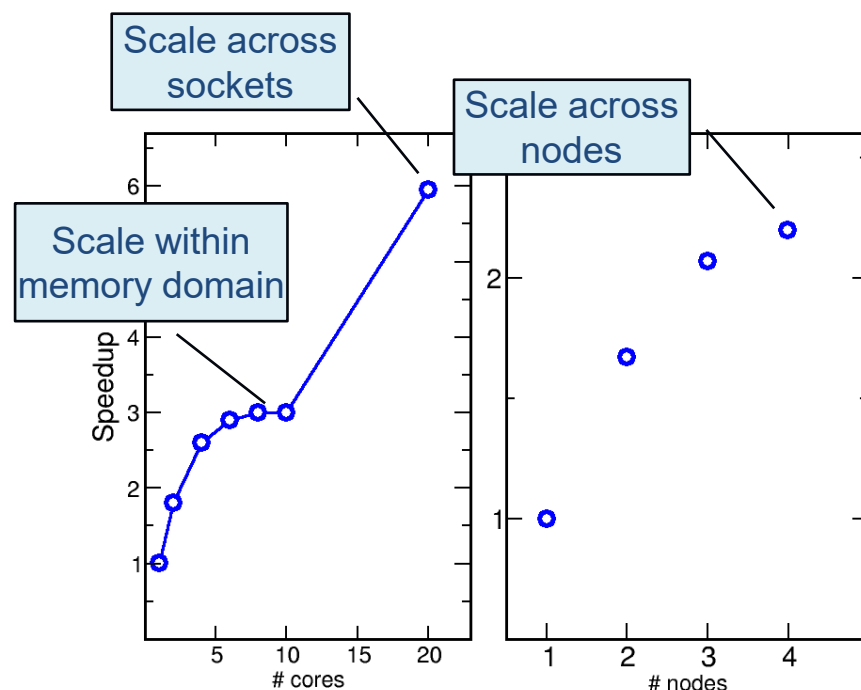
Tool for system state dump (requires Likwid tools):

<https://github.com/RRZE-HPC/MachineState>

Benchmark planning

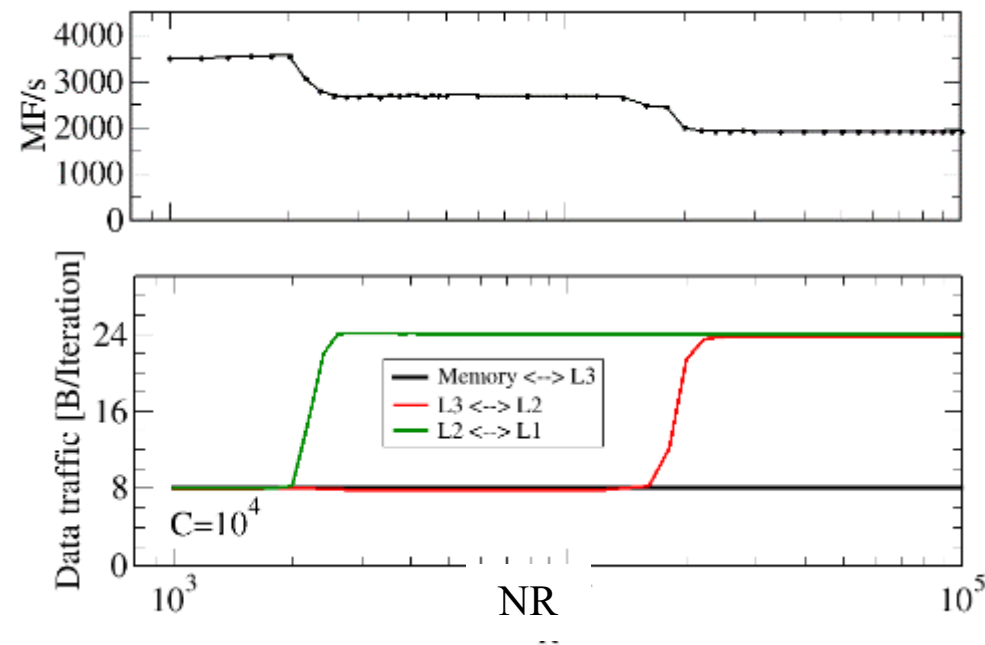
Two common variants:

Core/node/device count



Choosing the right scaling baseline

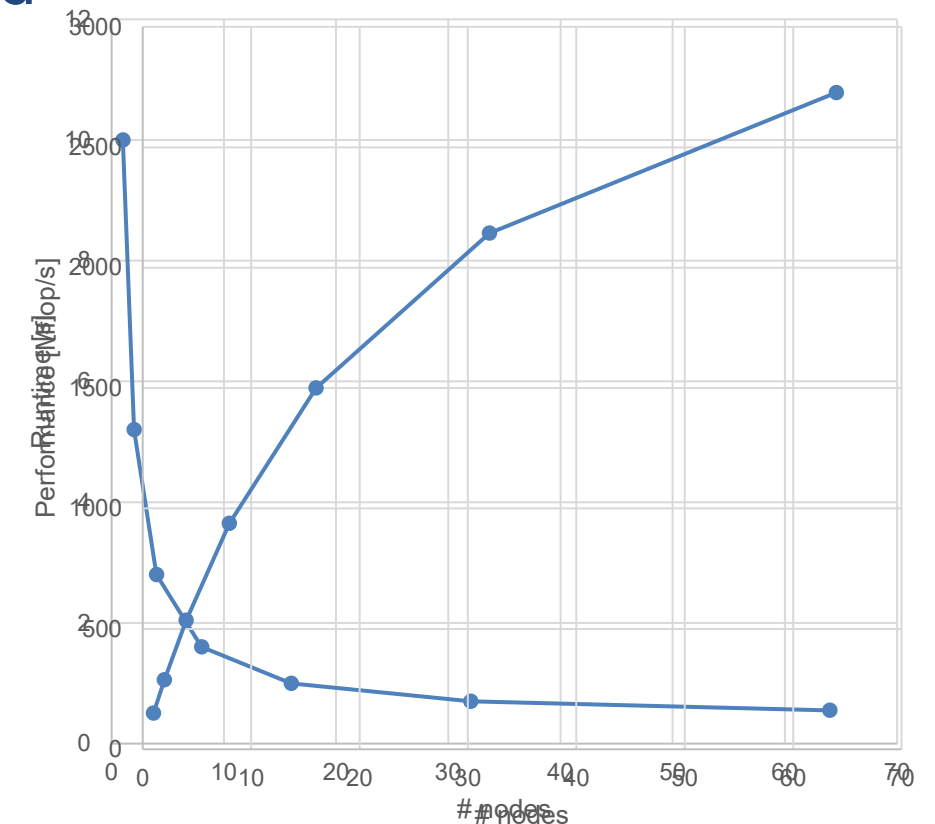
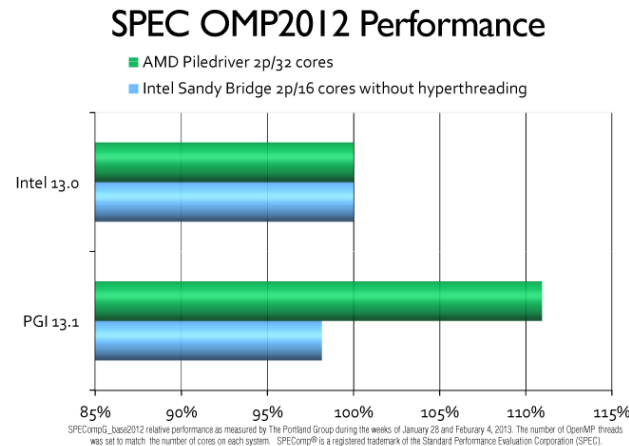
Dataset size



- Measure with one process (to start with)
- Scan dataset size in fine steps
- Verify the data volumes with a HPM tool

Common mistakes in result presentation

- Do not show time but performance (anything $1/t$) in scalability plots
- Add meaningful axis labels and self-contained captions
- Always use zero based y-axis range



The Performance Logbook

- **Manual** and knowledge collection how to build, configure and run application
- **Document** activities and results in a structured way
- Learn about **best practice guidelines** for performance engineering
- Serve as a well-defined and simple way to **exchange** and hand over performance projects

The logbook consists of a single **markdown** document, helper scripts, and directories for input, raw results, and media files.



<https://github.com/RRZE-HPC/ThePerformanceLogbook>