703650 VO Parallel Systems WS2019/2020
Measuring and Reporting Data

Philipp Gschwandtner

Overview

- what and how to measure
  - time
  - time-dependent (speedup, efficiency, …)
  - FLOPS
- use measurements to drive optimizations
  - Amdahl’s law
- how to report measurements

Motivation: Optimization

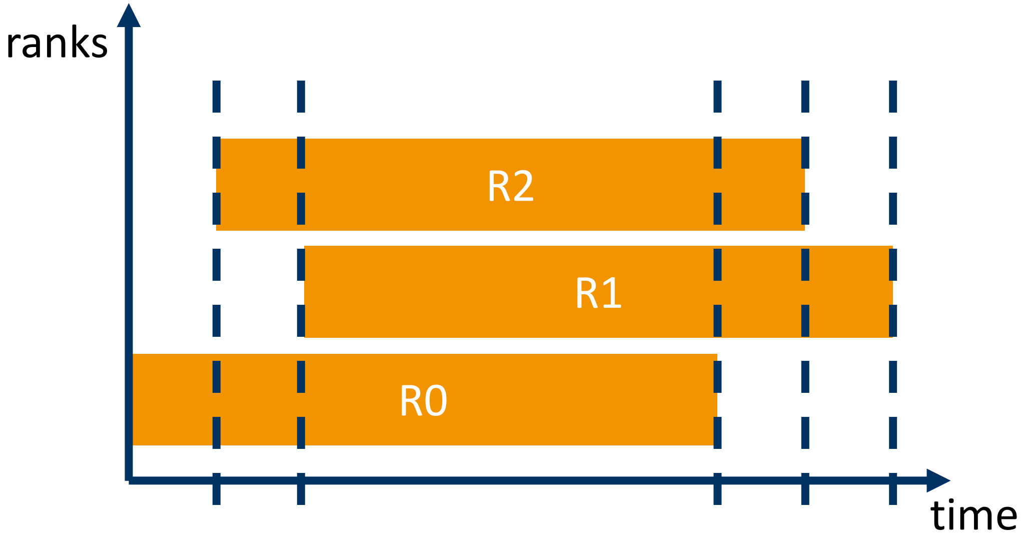
- difficult to optimize without measuring
  - measurements are crucial for everything in computer science
  - How do you know whether program performance improved or not?
- difficult to measure without knowing what is measured and how
  - know your metrics!
    - performance, time, efficiency, memory footprint, energy, FLOPs, cache misses, …
- tons of research on this topic…

Time-Related Metrics

Time

- *wall time*
  - time measured by looking at the wall clock
  - disregards degree of parallelism
  - the default when talking “time”

$$t_{\text{wall}} = \max_{0 \le i < N}\left(t_{\text{end},i}\right) - \min_{0 \le i < N}\left(t_{\text{start},i}\right)$$



- *cpu time*
  - wall time for each rank
  - cumulative over all ranks, varies with degree of parallelism

$$t_{\text{cpu}} = \sum_{i=0}^{N-1} t_i$$

Time is Parallel too

- synchronizing clocks over a large-scale system is very difficult
  - e.g. TSC: in-core timestamp counter, 1 clock cycle tick rate (< 1 ns)
  - compare to network latency: 100–1000 ns, best case
  - note: CPU clock frequency scaling (DVFS) today no longer an issue

- hard to compare time measurements between processes
  - no issue if moderate accuracy and granularity requirements
  - otherwise check synchronization first (e.g. `MPI_WTIME_IS_GLOBAL`)

- also: mind the timer granularity in general
  - e.g. don't measure time intervals of 1-2 milliseconds with a 1 millisecond granularity
  - good practice: 10x higher granularity than shortest interval to be measured

Speedup

- speed increase of a parallel program over the sequential version
- $\text{speedup}_p = \frac{t_s}{t_p}$
- ideal: $t_p = \frac{t_s}{p}$, hence $\text{speedup}_p = \frac{t_s}{\frac{t_s}{p}} = p$ (linear)
- super-linear speedup also possible
  - e.g. problem partitioning reduces memory footprint per processor
  - enables more efficient use of memory hierarchy (caches)

Absolute and Relative Speedup

- two kinds of speedup
  - absolute: reference t_s is the fastest sequential version
  - relative: reference t_s is the fastest parallel version run sequentially
  - rare and bad third option: reference is $t_{p'}$, with $p' < p$ (e.g. $p' = 16$ and $p = 128$)
  - always specify your reference!

- non-trivial problem: parallelism might entail algorithmic changes and/or overheads (e.g. communication)

Efficiency

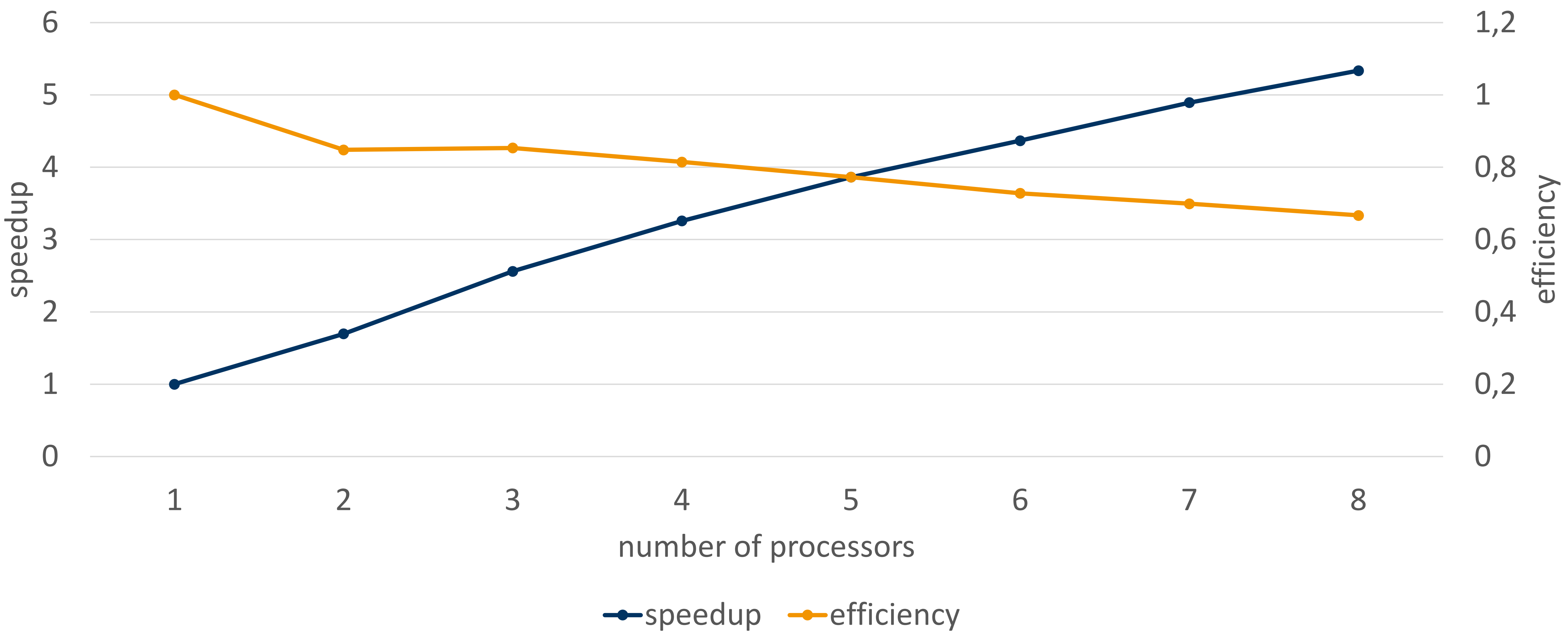
- measure of parallelization overhead
  - value range given between 0 and 1 or as a percentage

- $\text{efficiency}_p = \dfrac{\text{speedup}_p}{p}$

- ideal: $\text{efficiency}_p = 1$ (= linear speedup)

- worst case: $\lim_{p \to \infty} \text{efficiency}_p = 0$

Example Data for Speedup and Efficiency



Amdahl's Law

- one of the oldest (1967) and still most important laws in parallel programming

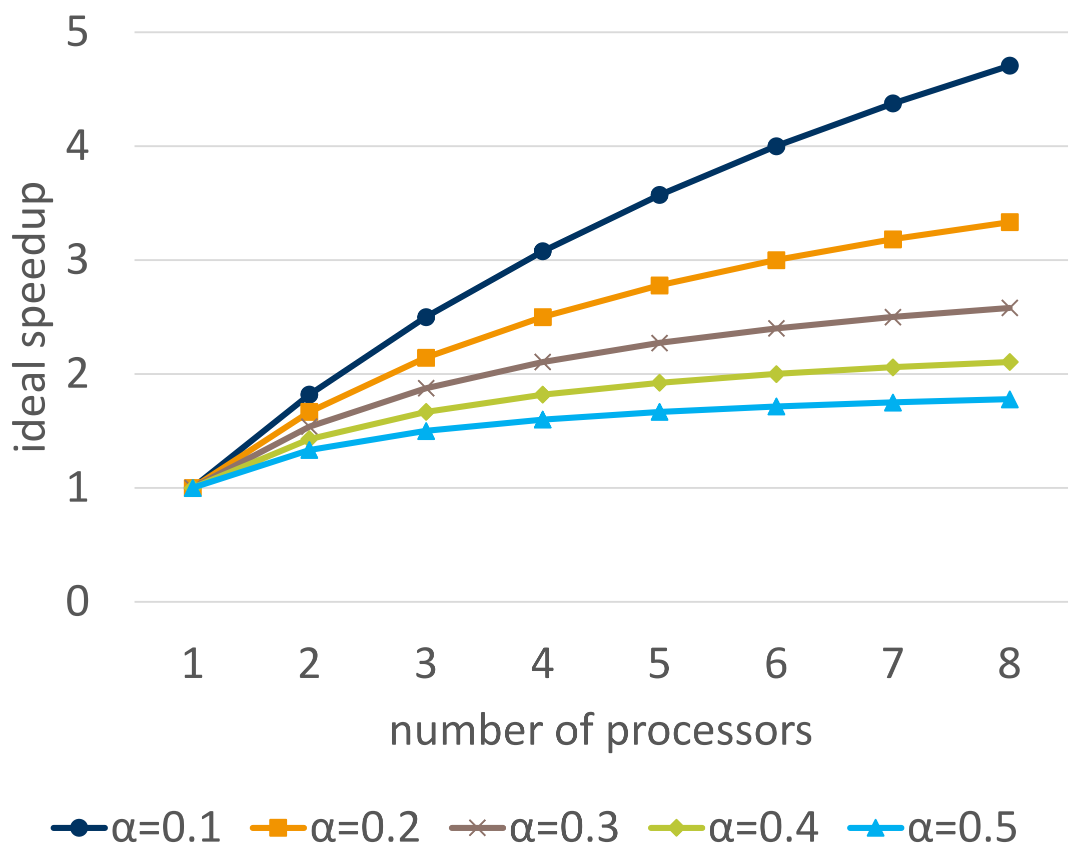
- defines (upper limit on) speedup when dividing a parallel program into a sequential part (r_s) and a parallel part (r_p), with $r_s + r_p = 1$

- idea: r_s cannot be improved through parallelism
  - hence it must somehow pose a limiting factor for any potential speedup
  - all improvement must originate solely from improving r_p

- note: idealized perspective, does not include e.g. hardware bottlenecks

Amdahl's Law cont'd

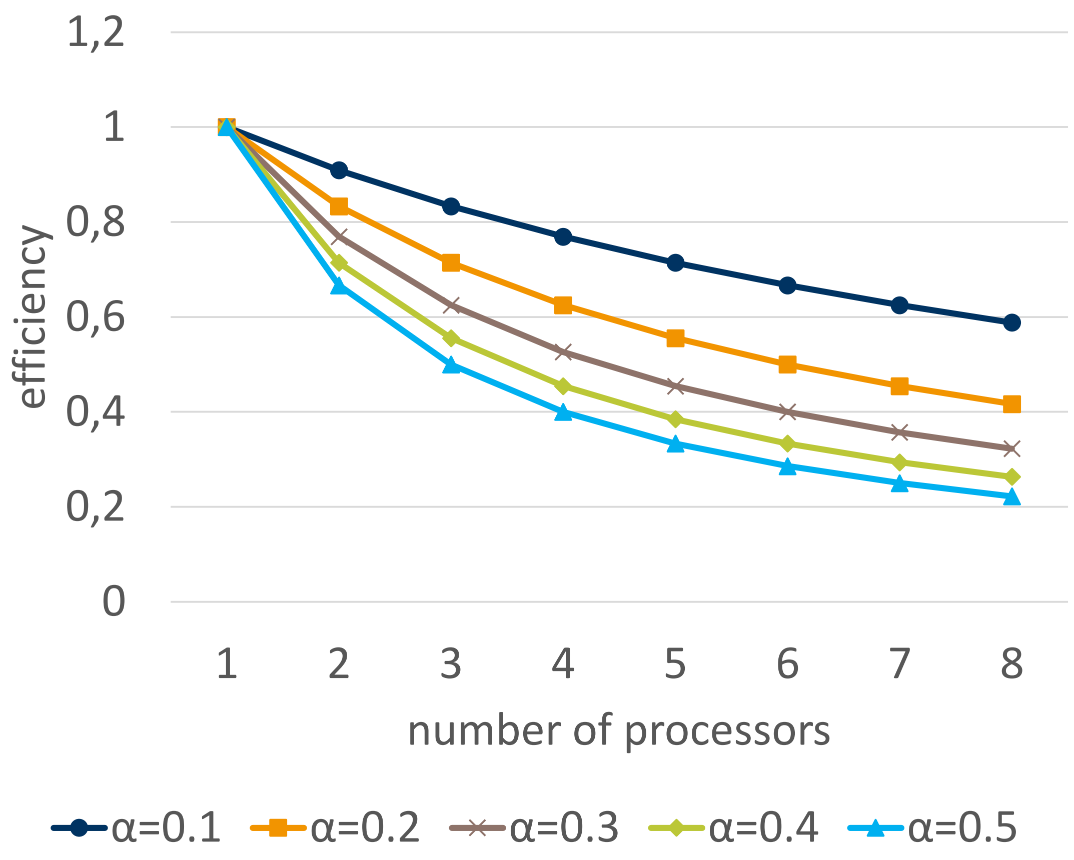
- law: $\text{speedup}_p = \frac{1}{r_s + \frac{r_p}{n}} = \frac{1}{\alpha + \frac{1-\alpha}{n}}$
- severely limits potential speedup, even for infinite parallelism
- example: $\alpha = 0.2$ (=20 % sequential)
  - ideal speedup on 8 processors is 3.33
  - ideal speedup on ∞ processors is 5



Amdahl's Law: Efficiency

- the same principle applies to parallel efficiency

- example: $\alpha = 0.2$ (=20 % sequential)
  - max. efficiency on 8 processors is 0.42
  - max. efficiency on ∞ processors?



Amdahl's Law: Implications

- first order of business: minimize α!
  - change algorithm to reduce amount of sequential code
    - careful, changes reference for relative speedup
  - optimize sequential code as much as possible
- don't spend excessive effort on optimizing parallel portion
- consider a sensible maximum degree of parallelism
  - don't allocate 1024 processors when using 128 is equally fast
  - also consider other users or power consumption!

Amdahl’s Law: Shortcomings

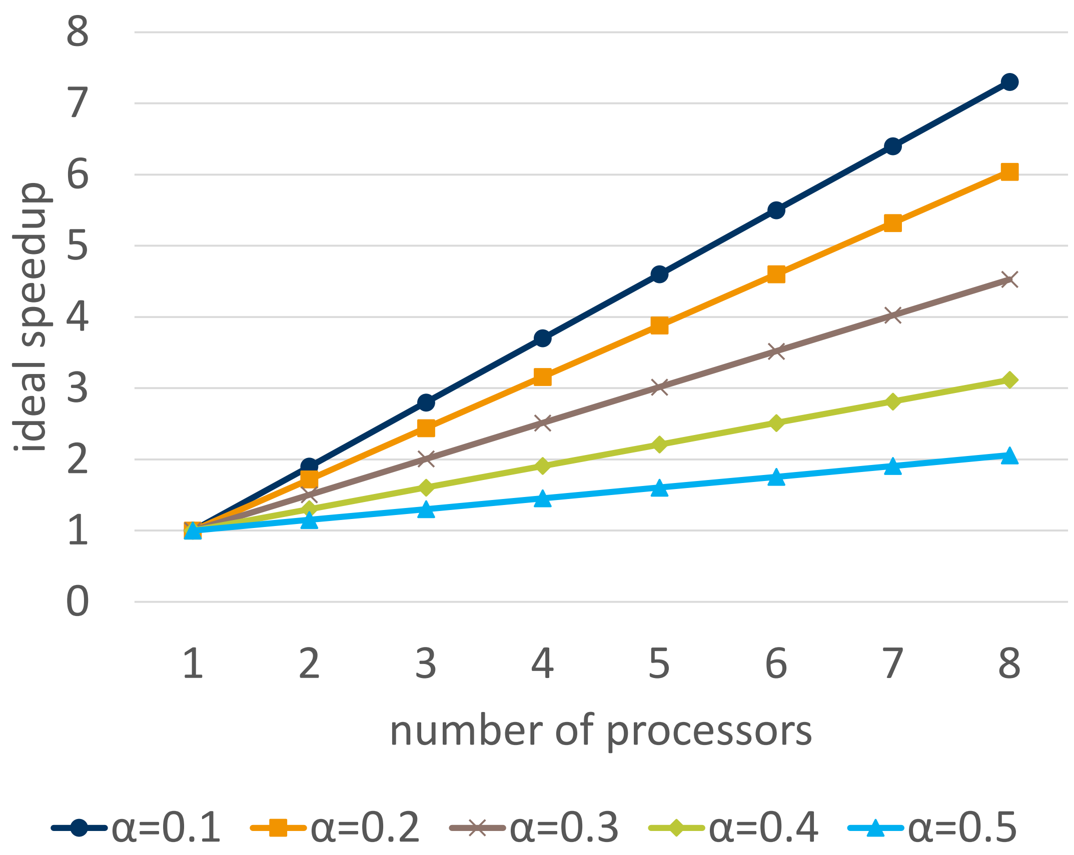
- Noticed, that it assumes the amount of work to be fixed?
  - not realistic for many use cases, problem size often scales with machine size
  - some problems are solved in real-time (e.g. rendering)
    but at the highest problem size (e.g. image quality) possible
  - more resources (=processors) can indeed help here

- John Gustafson reevaluated this issue in 1988

  ➔ Gustafson’s Law

Gustafson's Law cont'd

- Assume a sequential portion α and a parallel portion $(1-\alpha)$
- Assume that α is fixed or scales very slowly with the number of processors
- $\text{speedup}_p = \alpha + (1-\alpha) \times P$



Strong vs. Weak Scalability

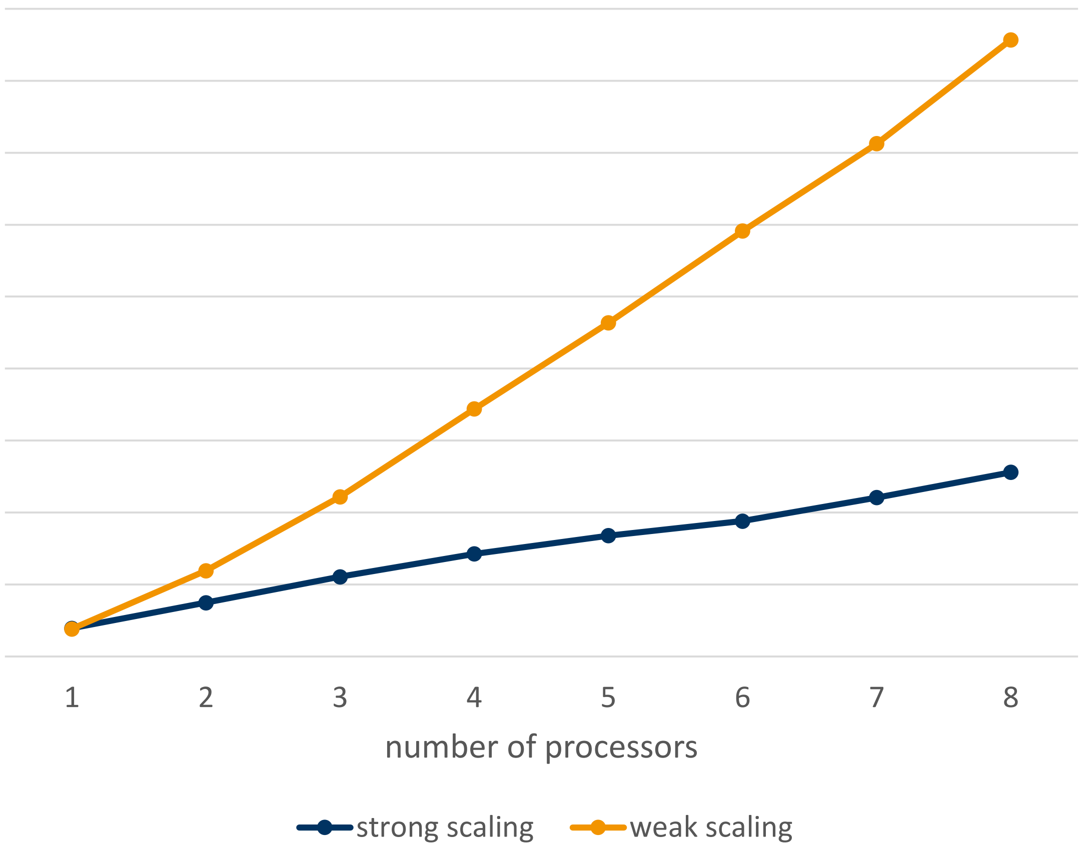
- *scalability* is (sort-of) a synonym for (good) speedup and efficiency
  - “program scales (linearly)” = program achieves linear speedup
- strong scalablity
  - how the program scales with a fixed problem size
- weak scalability
  - how the program scales when keeping the problem size proportional
  - important: how to scale the problem in proportion?

Weak Scalability

- consider an application with a two-dimensional domain of size n x m (e.g. 2D heat stencil)
  - scale n, or m, or both?
  - scale timesteps?

- consider a 3D problem?

- consider a blackbox problem?



Additional Metrics

FLOPS

- floating point operations (FLOPS/s: per second)
  - only operations with floating point semantics
  - do not count load/store/bitwise/…
- main measure for useful performance of software and hardware (work processed per time)
  - time not necessarily useful: can be wasted in waiting states…
- not applicable to every problem
  - consider integer-heavy problems (e.g. combinational logic, dynamic programming, …)

FLOPS cont'd

- What's the number of floating point instructions in x86 on the right?
  - 2 operations: mul, add
  - 1 instruction: fused multiply-add (FMA)

- Don't confuse *operations* and *instructions*
  - number of instructions is hardware- and compiler-dependent
  - know your hardware, compiler, and environment (e.g. compiler flags)
  - do not blindly trust hardware peak FLOPS

```
double foo(double A, double B,
           double C, double D) {
    D = A * B + C;
    return D;
}
```

```
x86:   vfmadd132sd xmm0, xmm2, xmm1
```

Common way of counting FLOPS: Multiply + Add

```
D = A * B + C;          // 2 Ops, 1 Ins
D = A     + C;          // 1 Op,  1 Ins
D = A * B    ;          // 1 Op,  1 Ins
D = A * A * B + C;      // 3 Ops, 2 Ins
D = A * B + C * A + B   // 4 Ops, 2 Ins
```

Measuring FLOPS

- instrumentation and profiling
  - `perf`, `PAPI`, etc.
  - requires detailed knowledge of the source code, hardware, and compiler
- modeling
  - know your algorithm and implementation
  - there are also tools to automatically build models, of varying quality…

```
FP_ARITH_INST_RETIRED.SCALAR_DOUBLE
FP_ARITH_INST_RETIRED.SCALAR_SINGLE
FP_ARITH_INST_RETIRED.128B_PACKED_DOUBLE
FP_ARITH_INST_RETIRED.128B_PACKED_SINGLE
FP_ARITH_INST_RETIRED.256B_PACKED_DOUBLE
FP_ARITH_INST_RETIRED.256B_PACKED_SINGLE
INST_RETIRED.X87
```

(a few floating point counters available on Intel Skylake CPUs)

FLOPS/s: Rmax vs. Rpeak

- Rmax: achieved by software
  - high performance linpack (HPL) benchmark
  - linear algebra stress-testing

- Rpeak: achievable by hardware
  - product of: number of FP units per CPU, their FP instructions per cycle, clock frequency, and number of CPUs

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Summit** - IBM Powe Volta GV100, Dual-ra DOE/SC/Oak Ridge I United States	2,414,592	148,600.0	200,794.9	10,096
2	**Sierra** - IBM Power Volta GV100, Dual-ra DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
3	**Sunway TaihuLight** Sunway , NRCPC National Supercomp China	10,649,600	93,014.6	125,435.9	15,371
4	**Tianhe-2A** - TH-IVB Express-2, Matrix-2 National Super Com	4,981,760	61,444.5	100,678.7	18,482

Application Throughput

- FLOPS/s are not everything
- consider what the application does
  - lagrangian simulation (no grid, move particles around): particles/s
  - eulerian simulation (no particles, compute properties at grid points): cells/s
  - chemistry applications: molecules/s
  - business applications: stock prices/s
  - unknown application: elements/s
- often much more useful to users than highly technical metrics

Compute Bound vs. Memory Bound (vs. Communication Bound)

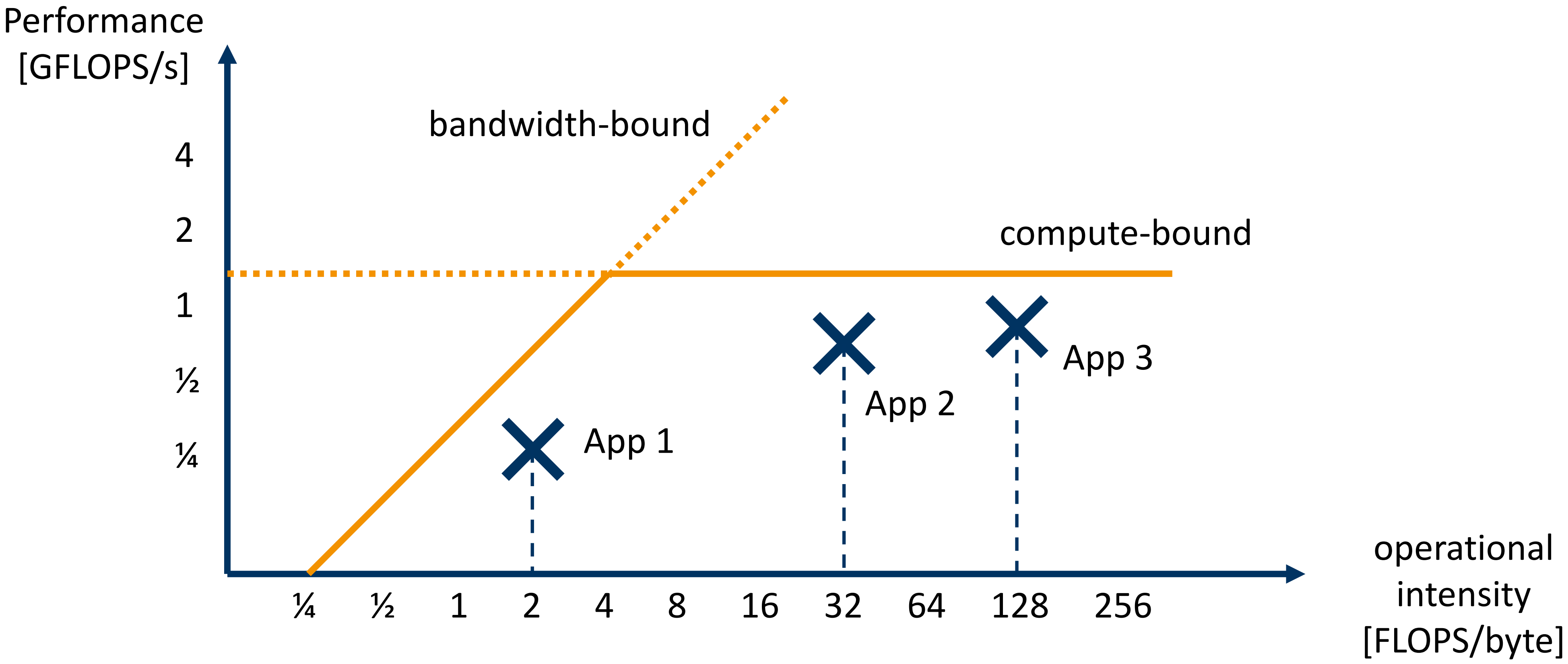
- what is the bottleneck of your code X on hardware Y
  - memory accesses
  - computational throughput
  - data transfer
  - …
- compute bound
  - no benefit in speeding up memory accesses, execution units are 100% busy
- memory bound
  - no benefit in speeding up computation, memory bus is 100% busy

Roofline Model

- visual performance model for intra-node optimization
  - illustrates performance potential of a given hardware architecture
  - illustrates how much of this potential is used by an application
  - indicates which optimizations to apply

- indicates whether one is compute bound or memory bound
  - compute bound: execution units are always busy with computation
  - memory bound: memory bus is always busy transferring data

Roofline Model cont’d



Driving Optimizations

- Roofline model says “bandwidth bound”
  - check cache optimality of algorithm
  - check NUMA mapping/bindings
  - check software prefetching

- Roofline model says “compute bound”
  - check vectorization (SIMD)
  - check hardware cache usage (data locality)
  - check software caching

Performance Counters

- originally added to CPUs for post-silicon functional debugging
  - nowadays re-purposed for performance debugging
- can be read with tools
  - command line: perf
  - library: PAPI
  - tons of additional performance tools out there…
- can be extremely useful if you know your hardware well

Performance Counters Example

```
[c703429@login.lcc2 ~]$ perf stat ./heat_stencil_1D_seq
                               ...
28,826,239,136 cycles:u            #   2.471 GHz
35,220,856,783 instructions:u      #   1.22  insn per cycle
 6,711,849,029 branches:u          # 575.356 M/sec
     1,295,209 branch-misses:u     #   0.02% of all branches
         1,044 LLC-load-misses:u
            26 LLC-store-misses:u
    15,312,122 L1-dcache-load-misses:u
   476,440,489 L1-dcache-store-misses:u
```

Properly Reporting Data

How to Report Metrics? (Non-Exhaustive)

- How many repetitions of an experiment do I need to run?
  - A single run is enough, right?
  - Three runs is enough for averaging, right?
- Should I run on multiple systems?
  - Should their architecture differ?
- Should my experiments be reproducible?
  - What information is required to enable reproducibility?
- Do I show all collected data?
  - Do I select data, and if so, how?
  - Do I aggregate data, and if so, how?
  - Do I report absolute or relative values?
- Do I quantify the reliability of my data?
  - If so, how?
- …

Experiment Repetitions

- Report whether your problem is deterministic!

- if deterministic
  - single run sufficient (should be – ideally – exactly reproducible)
  - Are performance measurements deterministic?

- if non-deterministic
  - multiple runs are required
  - report statistical measures
    - arithmetic/geometric/harmonic means
    - confidence intervals
    - …

How Many Repetitions are Required?

- use confidence interval (CI)
  - interval around sample mean, e.g. $\bar{x} = 100\ [95; 105]$
  - e.g. a 95 % CI means there's a 95 % probability it contains the true mean

- for normally distributed data
  - directly compute the number of measurements required to satisfy error e
  - $n = \left(\frac{s \times t\left(n-1, \frac{\alpha}{2}\right)}{e\bar{x}}\right)^2$
  - check https://en.wikipedia.org/wiki/Confidence_interval#Basic_steps for details on computing CI

- for non-normally distributed data
  - re-compute CI every k experiments (choose k depending on experiment effort), stop when happy
  - at least e.g. 5 runs to compute meaningful CI

Running on Multiple Systems

- depends on how high you aim
  - “I have shown in a proof-of-concept that this might work in some selected cases”

    vs.
  - “I have shown that this generally works in all cases”

- more hardware coverage is always better
  - but trade-off between porting effort and benefits

Data Collection and Selection

- document your efforts and backup your data and documentation
- if reporting only a subset of data, clearly specify the reason and motivation
  - e.g. only 8 to 16 nodes; only 4, 9, 16, 25, and 36 nodes
  - e.g. only using half the cores per node
  - e.g. applications 1, 2, and 3 on hardware A and B, but data for 3 on B is missing
  - e.g. only measuring parts of an application

Data Aggregation

- costs
  - usually atomic units, linear relationship
  - can directly aggregate using arithmetic mean
  - e.g. execution time (seconds), memory footprint (megabyte), energy consumption (joule)
- rates
  - if able, aggregate costs first and then compute rate; otherwise use harmonic mean
  - e.g. FLOPS/s, MB/s, etc.
- ratios
  - if able, don’t aggregate at all; if required, aggregate base data instead
  - e.g. “A is 50 % higher than B”

Reproducibility

- publish all required information, but not more than that
  - source code
  - compiler, version, flags
  - hardware architecture (detail depends on use case)
  - external factors (e.g. other users, load of the system, temperature, …)
  - reference data to compare against
  - aggregation methods, if used

- note: doesn’t mean you should provide a VM or container…

Four Steps to Creating an Optimized Parallel Program

- **<u>1.</u>** devise and implement the simplest, most straight-forward parallel solution that you can think of

- **<u>2.</u>** measure the performance (or whatever metric you are interested in) to obtain a point of reference

- **<u>3.</u>** improve your program based on your measurements

- **<u>4.</u>** while(!☺) { step2(); step3(); }

Sequential Equivalence

- strong sequential equivalence
  - bitwise identical results
  - potentially big impact on performance
    (associativity, collective communication patterns, …)
  - requires preserving the order of computations compared to sequential version

- weak sequential equivalence
  - mathematically equivalent but not bitwise identical
    (IEEE 754 float arithmetic is neither associative, nor commutative)
  - does not require preserving the order of computations

- Always check your requirements!
  - If your algorithm doesn't require a specific order, why should its implementation?

Portability

- functional portability
  - program runs on different hardware and produces correct results
  - hardware architecture, compiler, libraries, etc.
  - usually not hard to achieve (mostly x86 + GPUs)

- performance (also “non-functional”) portability
  - program runs efficiently on different hardware and is considered “fast”
  - no exact definition, topic of ongoing research

Summary

- lots of metrics to choose from
  - classics are speedup and efficiency
- always consider Amdahl’s law for everything you do
- measure before you optimize
  - and when you optimize, do it data-driven and not based on hunches

Image Sources

- Rmax/Rpeak: https://www.top500.org/lists/2019/06/