HIGH PERFORMANCE COMPUTING FOR SCIENCE AND ENGINEERING

LECTURE 15

Fabian Wermelinger
Harvard University
CS205

Tuesday, March 28th 2023

LAST TIME

- Examples of parallel file I/O
- MPI I/O terminology
- MPI file views
- MPI file management
- Methods for MPI I/O data access
- File pointer manipulations
- Example for parallel I/O with HDF5
- What is left in the MPI standard?

TODAY

Main topic: ***Parallel scalability, performance metrics, strong and weak scaling***

Details:

- Performance metrics
- Using hardware performance counters to compute metrics
- Parallel speedup and parallel efficiency
- Amdahl's Law and strong scaling
- Gustafson's Law and weak scaling
- Weak efficiency and its relation to parallel overhead and compute/transfer overlap

AGENDA CHECK:

- Reading for Thursday: *"A New Golden Age for Computer Architecture"* (`pdf` in `git` repository as well)
- Project final presentations and deliverables are due on ***Thursday May 4th 2023 8:00AM*** (final presentations will be on the 4th and 5th of May).

HOW DO YOU KNOW YOU DID A GOOD JOB?

- You are now at the point where you can write parallel code using shared memory *and distributed memory*.
- The next thing you should wonder about is how well did you parallelize the code and how good are your memory access patterns given the data structures you chose (***recall:*** importance of *cache memories*).
- There are two metrics you want to look at:

1. ***Parallel scalability:*** How well does your parallelization *scale* when you add more processing units. *Communication overhead* will be the main concern here. If communication *latency* is not hidden well, your code will not scale well.
2. ***Node-level performance:*** How well does your parallel implementation exploit the different *forms of parallelism on the hardware* (e.g. one CPU only or full node with multiple sockets). Recall lecture 10, the forms of parallelism *on the hardware:*
   - Instruction-level parallelism (**ILP**; *example: processor pipelining*)
   - Data-level parallelism (**DLP**; *example: SIMD*)
   - Thread-level parallelism (**TLP**; *example: multicore*; we are not interested in communication overhead here but whether your cores are busy or not)

METRICS

- A *metric* is a ***quantitative*** measure used to characterize, evaluate and compare (parallel) performance.
- A metric can be *absolute* or *derived* from other metrics.
- Parallel metrics are used to understand the relationship of *scalability* and (parallel) performance.
- Some absolute metrics are:
  - Time or clock count measurements
  - Measurements of memory operations
  - Floating point operations (flop)
  - Integer operations
  - Number of sent or received messages
- Time measurements can further be divided into how much time the application spends in OS calls and actual work.
- Hardware metrics are available via performance counters. See for example https://icl.utk.edu/papi/

DERIVED METRICS

- The example metrics on the previous slide may be regarded as *raw metrics* as they originate from *actual measurements*.
- New metrics can be calculated from other metrics which may be a better characterization of performance *(often you cannot measure them directly)*.
- A ***derived metric*** is the result of an arithmetic metric expression.
- The probably best known derived metric is *flops per second*. This derived metric is a *rate* and is usually expressed in units flop/s *(flop per second)*.
- ***Note:*** sometimes you may see "*flops*" which is ambiguous notation as it is not clear whether the plural of the raw metric *flop* or the derived metric flop/s is meant. In these cases you must infer from the context.
- *Rate-based* derived metrics are easily calculated from any raw metrics and are often referred to as ***throughput***.
- ***Examples of derived metrics include:*** sustainable memory bandwidth, communication bandwidth, the ratio of compute-transfer overlap or operational intensity.

COMPARATIVE AND PARALLELISM METRICS

- The metrics discussed so far can be thought of as describing aspects of parallel execution performance where the ***problem size*** and other parameters are *fixed*.
- Such an execution instance is typically called a *performance experiment*.
- ***Example:*** the roofline analysis is a performance experiment for a fixed computational kernel on a fixed hardware. The derived metric of interest often is *flop/s*.
- Such experiments create *comparative metrics* to highlight certain *cross-experiment* performance aspects. ***For example:*** comparing two different compute kernels *on the same hardware (fixed parameter)* → *kernel comparison:* are they memory bound, compute bound, can they reach the nominal peaks, etc.
- A simple comparative metric would consider the changes as a single parameter varies. For example, increasing the problem size (changing the amount of calculations) while the number of parallel processors remains fixed. This study is called *problem size scaling*.
- The term *parallelism metrics* will be used to generally characterize performance effects as a *result of changes in parallel execution metrics* (e.g. number of threads). Similar to comparative metrics, parallelism metrics have implicit and explicit relevance to how parallelism is manifesting in performance variation. ***Example:*** *speedup*.

TIME MEASUREMENTS

- Parallel computing is generally concerned with reducing the amount of *time* necessary to perform a computational task.
- The most fundamental performance metric of interest to parallel computing is *execution time*.
- *Wall-clock* time refers to the time that would be measured on an external clock that records time-of-day between a *start* and *finish* event.
- The datum of the time measurement is not important since we compute the *difference* between the finish and start event.
- It is important that a wall-clock measurement is the same regardless if it was collected on a computer or on your wrist watch → *this means that the time a process is idle must also be measured.*
- The *elapsed wall-clock time* $T(n;p)$ is a function of the problem size n and the degree of parallelism p (often processors).

TIME MEASUREMENTS: WHAT TO MEASURE?

- As we will see below, the *speedup* is not a good metric to compare the parallel performance of an application that runs on *different machines* or uses *different algorithms* to arrive at the *same* solution.
- *Time to solution* of a specific task in your application is mostly the correct performance metric to use. *Time to solution must be defined* and could be the wall-time of a single time step or the total wall-time of one simulation for example.
- Wall-time should be measured in software and not with external tools such as the Linux `time` command when writing benchmarks.
- Different timers exist such as `omp_get_wtime` or `MPI_Wtime` for OpenMP or MPI specific tasks, respectively. Alternatively, the C++ `chrono` API can be used as well (see https://en.cppreference.com/w/cpp/chrono)
- It is often advisable to collect multiple measurements and compute a time metric with appropriate statistics. Moreover, computing statistics of time measurements collected on different MPI processes can also reveal valuable insights → it will tell you if one rank has issues and slows every else body down.

EXAMPLE: HARDWARE COUNTERS WITH PAPI

- The **P**erformance **A**pplication **P**rogramming **I**nterface (PAPI) is a tool to access low-level performance counters on supported hardware (see this overview).
- Can be used to count other hardware metrics such as the number of load/store requests for example → *Not all hardware supports the same counters. For example, counting the number of flops on Intel Haswell or later is not supported.*
- Calculating the operational intensity on paper with documented assumptions provides a good estimate for the kernel OI.
- PAPI is a library for advanced performance monitoring and often you only need it for a more detailed analysis in a second iteration.
- The library is installed on the academic cluster in `${SHARED_DATA}/local`.
- The precise value what a hardware counter counts must be looked up in the hardware manual of the vendor. For example, Intel CPUs: Software developer manual Volume 3, Chapter 19 (https://cdrdv2.intel.com/v1/dl/getContent/671447)

EXAMPLE: HARDWARE COUNTERS WITH PAPI

From HW2: BLAS level-1 DAXPY kernel:

```
#include "papi.h"
#include <vector>

typedef double Real;
void daxpy(const Real a, const Real *x, Real *y, const int n);

int main()
{
    int n = 1000000;
    std::vector<Real> x(n, 1.0);
    std::vector<Real> y(n, 0.001);

    // Initialize PAPI
    int event_set = PAPI_NULL;
    int events[4] = {PAPI_TOT_CYC, PAPI_TOT_INS, PAPI_LST_INS, PAPI_L1_DCM}; // specific events to count
    long long int counters[4];
    PAPI_library_init(PAPI_VER_CURRENT);
    PAPI_create_eventset(&event_set);
    PAPI_add_events(event_set, events, 4);

    PAPI_start(event_set); // start PAPI measurement
    daxpy(1.0, x.data(), y.data(), n); // run code to be measured
    PAPI_stop(event_set, counters); // stop PAPI and get counter values

    const long long total_cycles = counters[0];       // cpu cycles
    const long long total_instructions = counters[1]; // any
    const long long total_load_stores = counters[2];  // number of such instructions
    const long long total_l1d_misses = counters[3];   // number of access request to cache line

    return 0;
}
```

See https://code.harvard.edu/CS205/main/tree/master/lecture/code/lecture15

EXAMPLE: HARDWARE COUNTERS WITH PAPI

HW2 DAXPY kernel:

- All results obtained on a Intel E5-2683 v4 CPU (Broadwell architecture, see roofline lecture 10):

```
$ salloc -N1 -c32 -t 01:00:00
```

- Multiple versions are compiled using different `-O` flags for optimization. See the `Makefile`.
- The problem size for the results below is $n = 1\,000\,000$.

Analysis: The DAXPY kernel performs 2 flops and 3 memory operations (double precision) in each iteration. The expected operational intensity is $I(n) = I = \frac{1}{12} \approx 0.08333$.

`g++ -g -O0`

```
Total cycles:                   8562273
Total instructions:             26002174
Instructions per cycle (IPC):  3.03683
L1 cache size:                  32 KB
L2 cache size:                  256 KB
L3 cache size:                  40960 KB
Total problem size:             15625 KB
Total L1 data misses:           269700
Total load/store:               14001089
Operational intensity:          1.785575e-02
Performance [Gflop/s]:          3.715108e-01
Wall-time   [micro-seconds]:    5.383424e+03
```

`g++ -g -O1` and `g++ -g -O2`

```
Total cycles:                   2013374
Total instructions:             6002109
Instructions per cycle (IPC):  2.98112
L1 cache size:                  32 KB
L2 cache size:                  256 KB
L3 cache size:                  40960 KB
Total problem size:             15625 KB
Total L1 data misses:           255372
Total load/store:               3001020
Operational intensity:          8.330501e-02
Performance [Gflop/s]:          1.483574e+00
Wall-time   [micro-seconds]:    1.348096e+03
```

`g++ -g -O3`

```
Total cycles:                   1721165
Total instructions:             3502119
Instructions per cycle (IPC):  2.03474
L1 cache size:                  32 KB
L2 cache size:                  256 KB
L3 cache size:                  40960 KB
Total problem size:             15625 KB
Total L1 data misses:           252065
Total load/store:               1501027
Operational intensity:          1.665526e-01
Performance [Gflop/s]:          1.725375e+00
Wall-time   [micro-seconds]:    1.159168e+03
```

- L1 data misses are about the same in all cases.
- Total load/store operations are *different* for all three versions! We would expect $3n$ memory operations.

EXAMPLE: HARDWARE COUNTERS WITH PAPI

HW2 DAXPY kernel: ***L1 misses***

- The L1 cache misses are around 255'000 and about the same for all variants.
- These counts correspond to the number of *cache line* accesses that miss (see Intel manual).
- The DAXPY kernel has good spatial locality (unit stride accesses). We read x and y which are both n-vectors in *double precision*. A cache line is 64 bytes on this architecture which results in at least 250'000 cache lines that must be read due to *compulsory misses*.

HW2 DAXPY kernel: ***total memory operations***

- This metric is harder to understand. We expect $3n$ memory operations for the DAXPY kernel.
- The hardware counter tracks the number of load/store instructions executed (including stack!).
- The non-optimized code (`-O0`) has about one order of magnitude more memory accesses which is the reason the performance is about one order of magnitude less → *uses the stack heavily instead of registers.*
- We get the correct counts (i.e., 3'000'000) with `-O1` and `-O2`.
- With the `-O3` flag we get only *half* of what we would expect, *why?*
- The reason is because `-O3` enables `-ftree-vectorize` which enables *compiler auto-vectorization (SIMD)*. When you pass `-fno-tree-vectorize` you will get the expected count for the memory operations with `-O3` as well. *The assembly code will give the explanation.*

EXAMPLE: HARDWARE COUNTERS WITH PAPI

HW2 SGEMV kernel: what about the *L1 misses* for this kernel?

On paper we can make the assumption that the size n is such that *two* vectors (of size n each) fit into the L1 cache (the vector x and one row of the matrix A). This is only true for small problems of course. In general the assumption above will not hold. ***Let's check this***.

Small n:

- If we choose $n = 1000$, two n-vectors will require 8 kB of memory in L1 (size is 32 kB).
- For each element in y we must read one row of A, one element in x and one in y, i.e. a total of $M = n^2 + 2n$ reads.
- We expect at least 62'625 *compulsory misses*.

```
Total cycles:                2948457
Total instructions:          6010115
Instructions per cycle (IPC): 2.03839
L1 cache size:               32 KB
L2 cache size:               256 KB
L3 cache size:               40960 KB
Total problem size:          3914 KB
Total L1 data misses:        62801
Total load/store:            2003027
Operational intensity:       2.497470e-01
Performance [Gflop/s]:       1.011702e+00
Wall-time   [micro-seconds]: 1.977856e+03
```

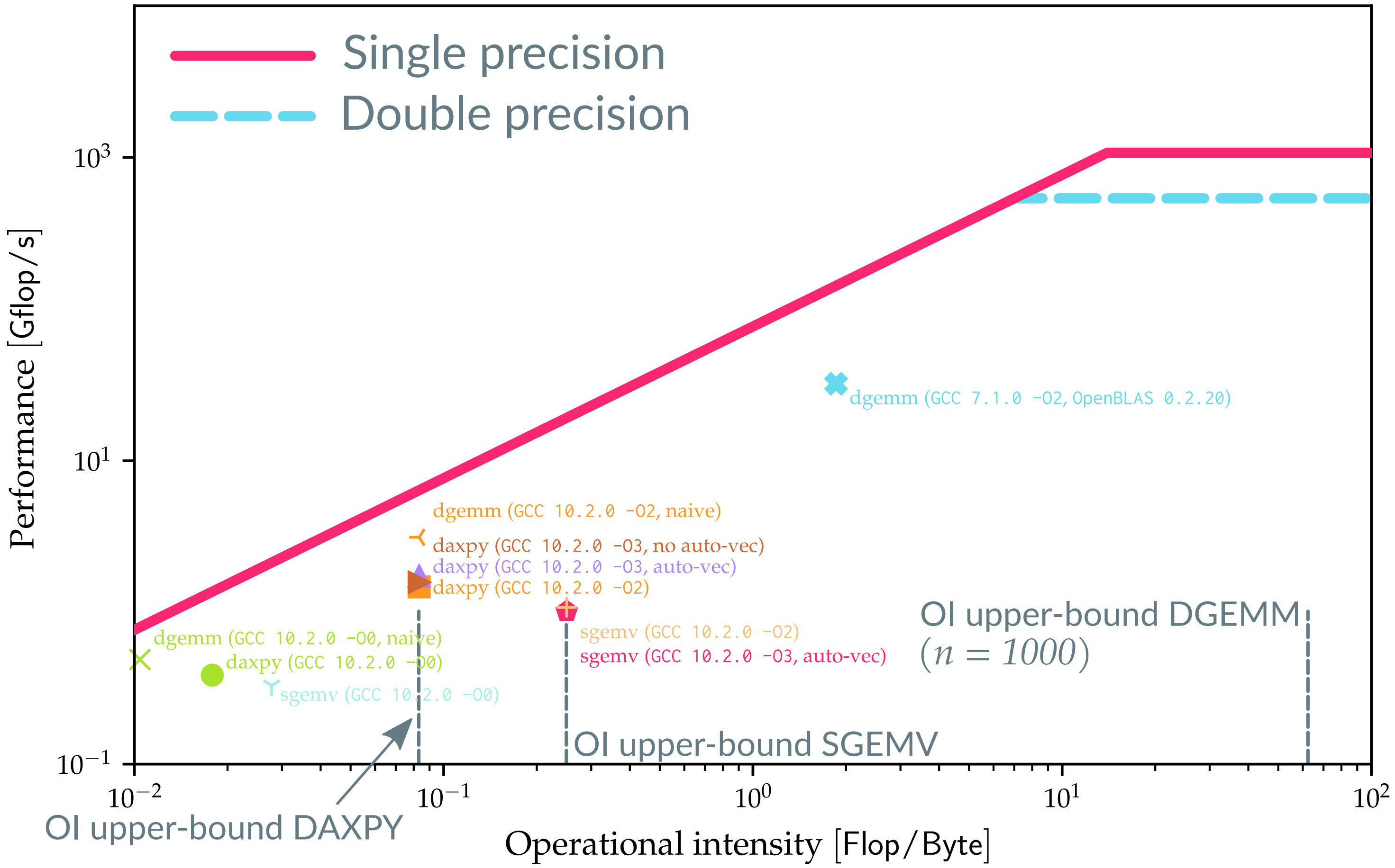
Large n:

- If we choose $n = 10000$, two n-vectors will require 80 kB of memory in L1 (size is 32 kB).
- For each element in y we must read one row of A, *the full vector* x and one element in y, i.e. a total of $M = 2n^2 + n$ reads.
- We expect at least 12'500'625 *compulsory* and *capacity misses*.

```
Total cycles:                318697168
Total instructions:          600082220
Instructions per cycle (IPC): 1.88292
L1 cache size:               32 KB
L2 cache size:               256 KB
L3 cache size:               40960 KB
Total problem size:          390703 KB
Total L1 data misses:        12542316
Total load/store:            200021821
Operational intensity:       2.499852e-01
Performance [Gflop/s]:       1.869667e+00
Wall-time   [micro-seconds]: 1.069763e+05
```

EXAMPLE: HARDWARE COUNTERS WITH PAPI

*The kernel variations in the roofline for the **Intel E5-2683 v4**:*



- All kernels are naive implementations (see the class `git` repo for the code).
- Single threaded.
- The theoretical upper-bounds computed on paper show good accuracy.
- No kernel can be at a higher OI than this bound. Lower is possible due to cache misses.
- The DGEMM kernel that uses BLAS performs best out-of-the box.

PARALLEL SPEEDUP

- *Speedup* is a *fundamental* metric for parallelism.
- Speedup is a *ratio of speeds* and expresses the performance improvement as parallelism increases.
- If $T_s = T(n;1)$ is the *sequential* execution time of a program to perform work W_s and $T_p = T(n;p)$ is the execution time when the program is run in *parallel* on p processors with work W_p, then the metric

$$S = S_p = \frac{W_p/T_p}{W_s/T_s}$$

expresses the *parallel speedup*.

- Note that the *rate* W/T is the *speed* at which work W is performed.
- In the case $W_p = W_s$ the speedup becomes a ratio of execution times:

$$S = S_p = \frac{T_s}{T_p}$$

- $S > 1$ conveys a performance *improvement* and $S < 1$ conveys a performance *degradation*.
- The *ideal* or perfect speedup is

$$S_p = p$$

and $S_p > p$ is referred to as *superlinear* speedup. See Gustafson, 1990.

PARALLEL EFFICIENCY

- Speedup metrics are based on rates which in turn depend on *time metrics.*
- It ***does not tell us*** how the speedup itself varies as the degree of parallelism changes.
- An *efficiency metric* is an example of a *utilization metric* and can be defined as the ratio of speedup to the degree of parallelism:

$$E_p = \frac{S_p}{p}$$

- ***Example:*** if your application reaches a speedup $S_p = 100$ but requires $p = 1000$ processors, the efficiency is less or equal to 10%
- Efficiency is useful to quantify *parallel overhead* in your application.
- Suppose the total time attributed to parallel overhead is $T_o = T_o(n; p)$. Running on p processors the equation

$$pT_p = T_s + T_o$$

must hold (assuming $W_p = W_s$).
- Using the definition of speedup and efficiency we can write

$$E_p = \frac{1}{1 + \frac{T_o}{T_s}}$$

- ***Note:*** it is possible that $T_o < 0$ (and therefore $E_p > 1$) for certain architecture/algorithm combinations resulting in *superlinear* speedup.

SPEEDUP THE UNIVERSAL METRIC?

- The speedup metric is often used to judge the parallel performance of (different) algorithms running on (possibly) different architectures for a particular problem.
- If you use a *"slow"* processor → achieving good scaling and speedup is easy (***Recall*** the ridge point in the roofline model).
- If you argue with speedup, the question is always ***over what***? When arguing with speedup, you should generally keep the algorithm fixed and be careful when comparing parallel speedups computed on different architectures.
- When comparing performance on different architectures it is usually better to use a derived metric based on a rate (*benchmarking*). ***Examples are:*** the number of computational cells per second (throughput) or how many time steps per second.
- → *A proper reference (sequential application) to compute T_s is important:*
  - Ideally it is the same algorithm but without any MPI calls.
  - In case of a hybrid MPI+OpenMP code, the reference could have threads enabled *(be clear when you document it!)*.
  - A reference implementation must always be optimized w/r/t memory accesses *(cache locality)*.
  - ***For example:*** "My GPU implementation is 100x faster than the sequential CPU code" has no meaning if you did not make sure the reference is already optimal. *We know from how bandwidth and peak performance differs between CPUs and GPUs that this implies the reference was not optimal and the claim is unfair and not accurate.*

AMDAHL'S LAW

- There are two well-known speedup formulations depending on how *execution time* is modeled.
- Amdahl's law describes the *maximum expected improvement* to an *overall system* when only part of the system is improved. If the ***problem size remains fixed*** and the number of parallel processors is increased in the *parallel part* of an application, then Amdahl's law expresses the *maximum speedup theoretically possible.*
- Let f be the *serial fraction* of an application (part of code that *cannot be parallelized*). The parallel execution time for Amdahl's law is

$$\boxed{T_p = fT_s + (1-f)\frac{T_s}{p}}$$

assuming negligible parallel overhead ($T_o = 0$*).*

- The speedup for this model is given by

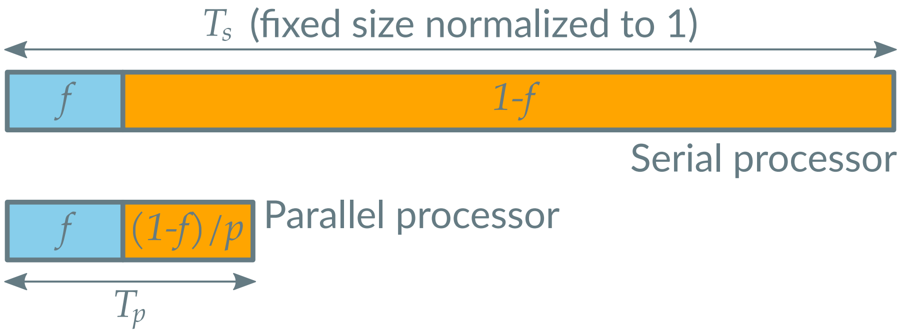
$$\boxed{S_p = \frac{T_s}{T_p} = \frac{1}{f + \frac{1-f}{p}}}$$

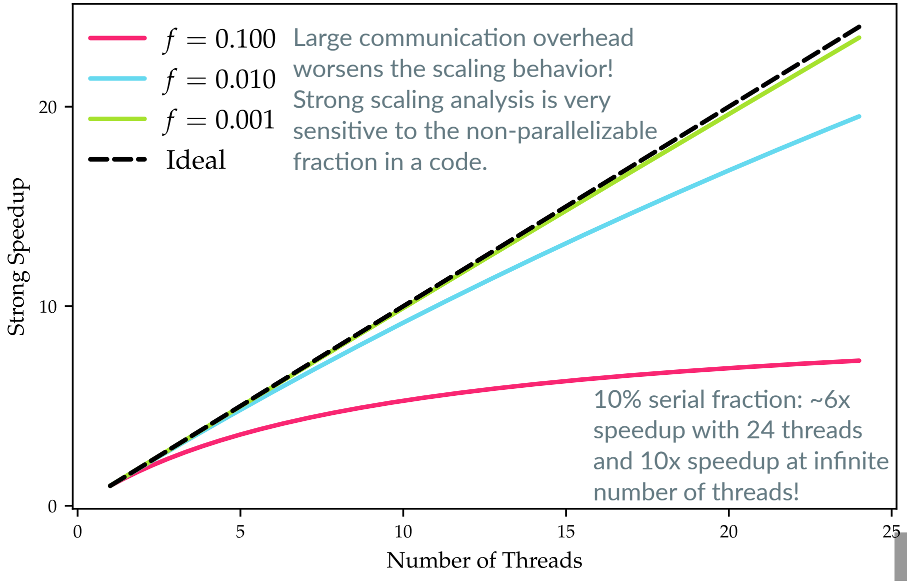
- Note that the maximum speedup is bounded by the *serial fraction* since $\lim_{p\to\infty} S_p = \frac{1}{f}$.

STRONG SCALING

- A speedup analysis formulated using Amdahl's Law is called ***strong scaling***.
- It assumes the problem size is fixed (defined by the sequential problem) and you add more parallel processors to *arrive at the solution faster*.
- The serial fraction in a code becomes the dominant factor in this model. *Serial fraction is hard to determine in general.*
- Achieving good strong scaling speedup for a large scale system with thousands of processors is ***very hard***.
- Strong scaling analysis makes sense for systems with few processors and negligible communication overhead → shared memory.

$$S_p = \frac{T_s}{T_p} = \frac{1}{f + \frac{1-f}{p}}$$





IMPLICIT ASSUMPTIONS IN AMDAHL'S LAW

Fixed problem size

In the past 20 years the implicit assumption of fixed problem size has been overlooked. It was valid back then when HPC systems were small or target systems had few CPUs only. Strong scaling a problem leads to the solution faster but reduces the resource utilization on individual compute nodes (smaller memory footprints) and therefore wasting resources.

Communication cost

Amdahl formulated his Law in 1967 where floating point arithmetic was much slower than today. Assuming *negligible* communication overhead among parallel processors was valid then but is no longer today. Strong scaling cannot express the impact of the communication overhead T_o accurately.

All-or-none parallelism

There are only two levels of concurrency in Amdahl's law: p-parallel or serial. A more realistic formulation of the model includes more fine grained fractions that express sections that can be executed with i processors. The speedup in this case becomes

$$S_p = \frac{1}{\sum_{i=1}^{p} \frac{f_i}{i}}$$

with $\sum_{i=1}^{p} f_i = 1$.

Problems for which these assumptions are reasonable can use this model for scaling analysis. We call this type of scaling analysis ***strong scaling***.

EXAMPLE APPLICATIONS OF AMDAHL'S LAW

Example 1: *Web server*

Suppose that we want to enhance the processor used for a web server. The new processor can do the computation in the web serving application 10 times faster than the old one. Assuming that the old processor is busy with computation 40% of the time and is waiting for I/O 60% of the time, what is the overall speedup you can expect when buying the new processor?

The serial fraction is $f = 0.6$ and the parallel fraction is $1 - f = 0.4$. The new processor can do work 10 times faster so $p = 10$:

$$S_{10} = \frac{1}{0.6 + \frac{0.4}{10}} \approx 1.56.$$

The efficiency in this case is 15.6% which will allow you to make a judgement whether it is reasonable to buy the new processor given its current price.

EXAMPLE APPLICATIONS OF AMDAHL'S LAW

Example 2: *Arithmetic for a larger problem*

Suppose you want to perform two sums in parallel:

- sum 10 scalar variables (they do not scale → sequential)
- sum two matrices $A + B$ with dimension $n \times n$

Assume you are interested in problems for $n = 10$ and $n = 100$ and you have two machines with 10 and 100 processors, respectively. What speedup do you observe?

The total amount of work is quantified by the number of flop (additions) performed $W = n^2 + 10$, where the 10 comes form the sequential additions. The problem size W is *fixed*. The serial fraction is $f = \frac{10}{W}$ which equates to a speedup and efficiency

$$S_p = \frac{10 + n^2}{10 + \frac{n^2}{p}}, \qquad E_p = \frac{10 + n^2}{10p + n^2},$$

respectively.

Small problem: $n = 10$

$p = 10$	$S_{10} = 5.5$	$E_{10} = 0.55$
$p = 100$	$S_{100} = 10.0$	$E_{100} = 0.1$

Large problem: $n = 100$

$p = 10$	$S_{10} = 9.9$	$E_{10} = 0.99$
$p = 100$	$S_{100} = 91.0$	$E_{100} = 0.91$

Running this problem on 10 processors is more efficient. *More processors is not*

EXAMPLE APPLICATIONS OF AMDAHL'S LAW

Example 2: *Arithmetic for a larger problem (continued: load imbalance)*

In the previous example we have assumed that the work W was *perfectly balanced* among the processors. *Load balance* means that in case of the 100 processors, each processor performs exactly 1% of the total work W. How does the speedup change if 1 out of p processors has increased loads of 2% and 5%, respectively?

Let l_1 be the load fraction of one out of p processors. The *parallel* fraction in Amdahl's Law is now constrained by

$$\max\left(\frac{l_1 n^2}{1}, \frac{(1-l_1)n^2}{p-1}\right)$$

and the speedup becomes

$$S_p = \frac{10+n^2}{10+\max\left(\frac{l_1 n^2}{1}, \frac{(1-l_1)n^2}{p-1}\right)}.$$

Large problem: $n = 100, l_1 = 0.02$

$p = 10$	$S_{10} = 9.1$	$E_{10} = 0.91$
$p = 100$	$S_{100} = 47.7$	$E_{100} = 0.48$

Large problem: $n = 100, l_1 = 0.05$

$p = 10$	$S_{10} = 9.4$	$E_{10} = 0.94$
$p = 100$	$S_{100} = 19.6$	$E_{100} = 0.20$

Load imbalance has a *dramatic* effect on systems with many parallel processors!

GUSTAFSON'S ARGUMENT

- Amdahl's Law was long the standard model to assess parallel performance.
- Parallel computers have become much more powerful and Amdahl's Law becomes inappropriate for massively parallel systems (see Gustafson 1988).
- Gustafson and Barsis suggested that instead of fixing the computational work, it should be allowed to *increase* as the degree of parallelism is increased. In this approach the ***execution time per process*** is fixed instead of problem size. This allows problems to scale to much larger systems.
- Suppose the *normalized* parallel execution time is

$$T_p = a(n) + b(n) = 1,$$

where $a(n)$ and $b(n)$ are the serial and parallel fractions, respectively, for a problem of size n.
- The sequential execution time is given by

$$T_s = a(n) + p\big(1 - a(n)\big).$$

- The speedup in this model then is

$$S_p = \frac{T_s}{T_p} = f + p(1 - f).$$

- As the *total problem* size increases, the serial fraction diminishes and the speedup becomes $\lim_{n\to\infty} S_p = p$.

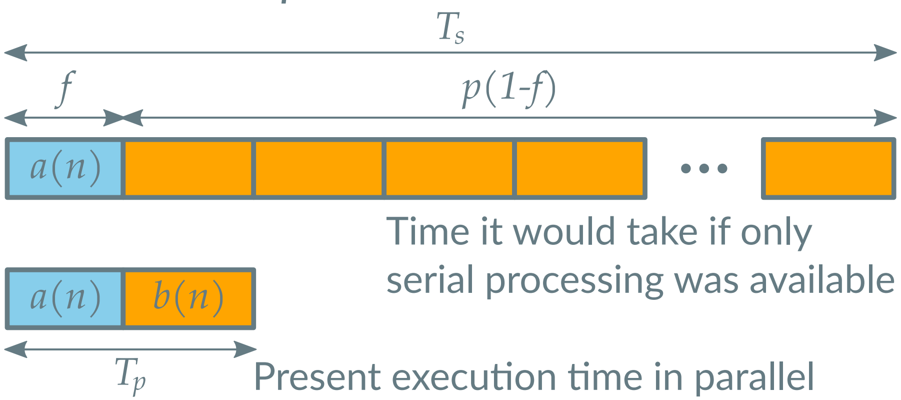
WEAK SCALING

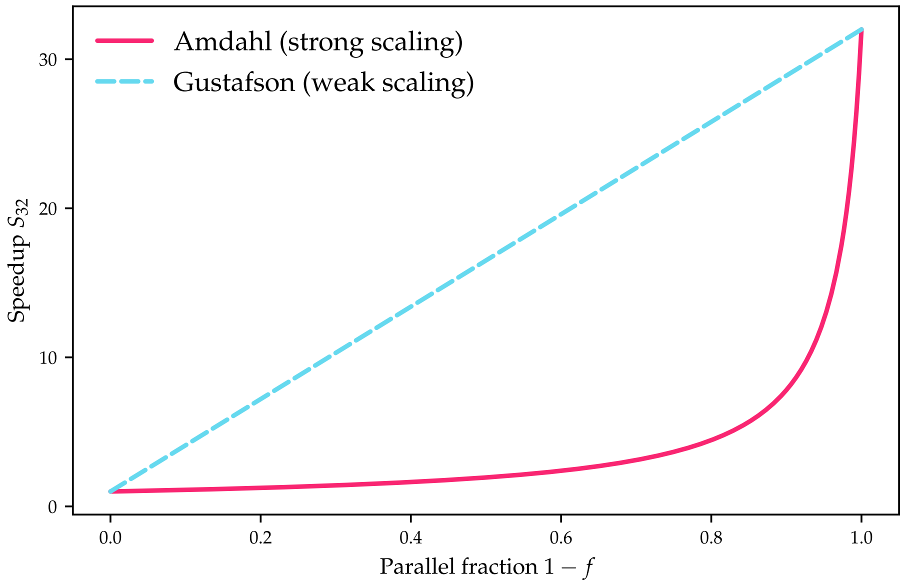
- A speedup analysis formulated with this model is called ***weak scaling***.
- It assumes the execution time per processor (the work per processor) is fixed and you add more parallel processors to increase the problem size to arrive at a larger solution *in the same time* as it would take for one processor working on a p-times smaller problem.
- The argument is equivalent to a work size of the parallel problem that is p times the serial problem, i.e., $W_p = pW_s$. The speedup is

$$S_p = \frac{W_p/T_p}{W_s/T_s} = p\frac{T_s}{T_p} = p\frac{T_s}{T_s + T_o},$$

where this assumes a diminishing serial fraction for $n \to \infty$ but non-zero parallel overhead as $T_o = T_o(n; p)$.

Fixed parallel execution time:





Observable speedup for 32 processors as a function of the *parallel fraction* in a code, expressed in terms of Amdahl's Law and Gustafson's Law.

WEAK EFFICIENCY

- Given the relation for the speedup in the weak argument (previous slide)

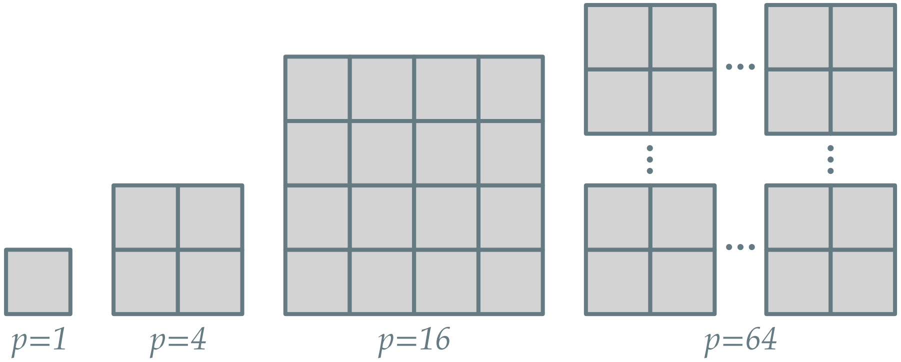
$$S_p = \frac{W_p/T_p}{W_s/T_s} = p\frac{T_s}{T_p} = p\frac{T_s}{T_s + T_o},$$

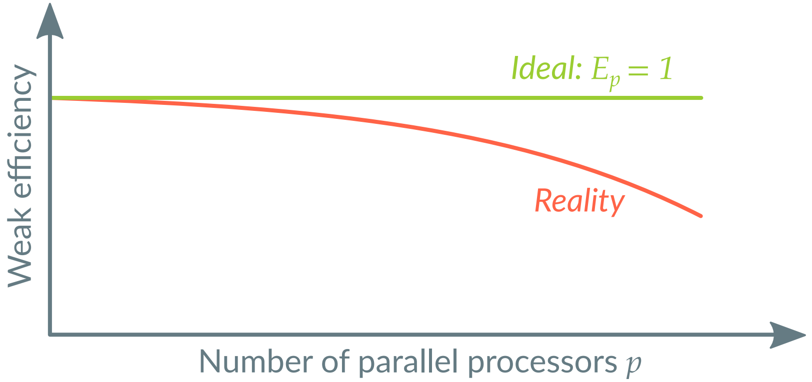
we obtain the weak efficiency

$$\boxed{E_p = \frac{S_p}{p} = \frac{T_s}{T_p} = \frac{1}{1 + \frac{T_o}{T_s}}}$$

- This defines a metric that allows us to quantify the *total parallel overhead* T_o and therefore provides us with insight on *how well parallel overhead is **hidden** in a code*.
- Since communication overhead is the main contributor to the total parallel overhead, the weak efficiency is a good indicator for the quality of compute/transfer overlap in an application.

Fixed size per processor:





Reasons for efficiency reduction:

- Blocking semantics (*replace with non-blocking semantics*)
- Many MPI calls (*combine MPI+OpenMP*)
- Collective primitives(*algorithm/problem dependent*)
- Message sizes (*pack them into bigger messages*)
- Network contention (*play with MPI configuration*)

WEAK SCALING ANALOGIES

There are many aspects of technology where an enhancement for ***time reduction*** actually turns out to be an enhancement for ***what can be accomplished in the same time*** instead.

Commuting time

People seem to have an internal threshold for what is an acceptable commuting time. This time seems to average around 30 minutes and has been like that since we were commuting with horses or much faster trains as of today. The situation today has improved nevertheless because the reach for possible work with a train is larger than with a horse considering both commute for 30 minutes. This is an example of Gustafson's Law applied to commuting trips at *fixed time*.

Human brain

If the human brain would be governed by Amdahl's Law, then the human intelligence would be similar to that of a starfish. The human brain has about 100 Billion neurons that operate in parallel, to avoid passing the point of diminishing returns in Amdahl's Law, the serial fraction to process a task in the human brain would have to be about 10^{-14} *(which it is not!)*. If that was the case we would be able to compute extremely complex tasks within nanoseconds. Gustafson's Law says that on a time-averaged basis, the human brain will accomplish *vastly more complex tasks* than what a starfish can accomplish.

RECAP

- Wall-time is an important performance metric used to derive other metrics.
- Hardware performance counters can be used to collect metrics like memory accesses or number of floating point instructions executed.
- Computing the operational intensity on paper with suitable (cache) assumptions often leads to accurate approximations for which the large overhead associated with accessing performance counters is not necessary.
- Parallel speedup is not a universal metric to express parallel performance.
- Strong scaling is useful to evaluate shared memory code *(few number of processors)*.
- Weak scaling is preferable when scaling to large machines with thousands of processors.
- Weak efficiency is useful to asses the quality compute/transfer overlap

Further reading:

- Hockney, "*The Science of Computer Benchmarking*", SIAM, 1996, https://doi.org/10.1137/1.9780898719666
- A. Y. Grama et al., "*Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures*", IEEE, 1993
- J. L. Gustafson, "*Fixed Time, Tiered Memory and Superlinear Speedup*", Proceedings of the 5th Distributed Memory Computing Conference, 1990
- J. L. Gustafson, "*Reevaluating Amdahl's Law*", ACM, 1988
- G. M. Amdahl, "*Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities*", AFIPS Press, 1967