NTNU – Trondheim
Norwegian University of
Science and Technology

Amdahl’s and Gustafson’s Laws

Speedup, efficiency, and scalability

Jan.Christian.Meyer@ntnu.no

Every parallel computation can be serialized

- Given a parallel computation where
  - Steps s_1, s_2, s_3, …, s_T all need to be taken
  - Some subset of steps s_i-s_j are evaluated simultaneously
- Evaluate those in some order (*e.g.* from i to j)
- If there is another subset of simultaneous operations, pick an order for it until you have ordered every step
- The parallel parts of a computation can't depend on any particular order, that's what makes them parallel

Not every sequential computation can be parallelized (probably)

- Given a serial computation where
  - Steps s_1, s_2, s_3, …, s_T all need to be taken
  - The input data of every step s_i contains an element from the output data of its predecessor s_{i-1} (for i>1)
- Even if we can begin to evaluate s_4 before s_3 is complete, it can't finish *first*, because it needs the result from s_3
- Some computations are *inherently sequential*

Nota Bene:
That was not a formal proof

- It was a common-sense argument
- We *can* formalize it, and say that
  - A problem is in the class P if it can be solved by a deterministic turing machine in polynomial time proportional to the problem size
  - A problem is in the class NC if it can be solved in parallel polylogarithmic time proportional to the number of processors
  - A problem x is P-complete if it is in P, and any other problem in P can be reduced to x in polylogarithmic time proportional to the problem size

...and then we get the “P = NC?” problem from complexity theory, which has been precisely as resistant to solution as its more famous “P = NP?” cousin

Some things just have to be done in a given order

- On paper, we don’t know whether all computations can be parallelized or not
- We know that the problem of simulating T steps of a von Neumann machine is P-Complete, though
- The trick with *starting* steps simultaneously and only *completing* them in requisite order can produce faster results (but not asymptotically faster)
- We can at least have an intuition about this issue until someone can prove us wrong

Ok, so *some* steps mandate a sequence. Now what?

- There is always at least one such *sequential dependency* in any program:
  - The final operation indirectly depends on the first, no activity can stop before it has started
  - Even if you write code where every statement could theoretically run simultaneously, it still has to launch and halt in practice
- Parallelizing a program amounts to discriminating between the sequentially dependent and the parallelizable steps

Total execution time

- Since every program will contain a mixture of sequentially dependent and parallel operations, we can define that
  - T is the sum of the time costs of all operations in the program,
  - *f* is the fraction of sequential operations it requires, so
  - *(1-f)* must be the fraction of operations that can be parallelized, and
  - T_s is the time it takes to run them all in sequence,

$$T_s = f \cdot T + (1 - f) \cdot T$$

  - We're just splitting the sum of operation costs into two parts that together add up to 1

Parallel execution time

- If we suppose that the parallel operations can be evenly distributed among p processors, the parallelizable part should only take 1/p as long:

$$T_p = f \cdot T + \frac{(1-f) \cdot T}{p}$$

Speedup

- Suppose we want to compare a slow and a fast solution to the same problem
- If the fast one takes ¼ as much time as the slow, we want to call it "4 times faster"
- That's the *speedup* of the fast solution relative to the slow one:

$$Speedup = \frac{T_{slow}}{T_{fast}}$$

Comparing the two

- If we assume the sequential run is T_{slow} and the parallel one is T_{fast}, we get speedup as a function of the number of processors:

$$S(p) = \frac{T_s}{T_p} = \frac{f \cdot T + (1-f) \cdot T}{f \cdot T + \frac{(1-f) \cdot T}{p}}$$

NTNU – Trondheim
Norwegian University of
Science and Technology

If we could parallelize everything

- If there were no sequential dependencies at all, f would be 0, and we get

$$\lim_{f \to 0} \frac{f \cdot T + (1-f) \cdot T}{f \cdot T + \frac{(1-f) \cdot T}{p}} = \frac{T}{\frac{T}{p}} = p$$

- S(p)=p is called *linear speedup*
- It would be nice if program performance were equally improved by every additional processor
- We can’t have it, because f can’t be exactly 0

NTNU – Trondheim
Norwegian University of Science and Technology

Let the processor count grow

- Let's pretend that we can have as many processors as we like without paying for them:

$$\lim_{p \to \infty} \frac{f \cdot T + (1-f) \cdot T}{f \cdot T + \frac{(1-f) \cdot T}{p}} = \frac{f \cdot T + (1-f) \cdot T}{f \cdot T} = \frac{T}{f \cdot T} = \frac{1}{f}$$

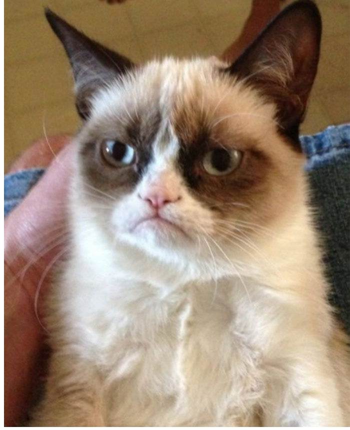
- In other words,

$$\lim_{p \to \infty} S(p) = \frac{1}{f}$$

- This observation is called *Amdahl's Law*

NTNU – Trondheim
Norwegian University of
Science and Technology

That is horrible!



- The amount of sequential work is built into the problem we're solving, not the machine
- If the problem happens to be 10% sequential, no number of processors can even theoretically speed it up more than 10 times
- This parallel computing stuff isn't very impressive
- **I Quit!**

The small print

- When Gene Amdahl made this argument back in 1967, he was working for IBM
- At the time, their business was to sell faster sequential computers, not more parallel ones
- What the argument doesn't mention, is that it assumes we try to solve the same, fixed problem on sequential and parallel computers
  - That's like replacing a car with a bus and complaining that you can't drive it any faster
  - Clearly, a waste of capacity

Total execution time (again)

- If you look at the Amdahl argument, we could actually cancel the total time T from every term in the speedup expression right away
  - The argument has to do with the ratio between two different ways to distribute the same amount of run time
  - the exact value of T doesn't really matter, we could just say it's 1
- I kept it in as a reminder of an assumption we started from:

  $T_s = fT + (1-f)T$

  says that we're calculating everything in proportion to an amount of work that always takes T time sequentially
- Amdahl's law assumes *constant serial time*

  i.e. we apply ever more processors to the same amount of work

The other way to do it

- If we assume *constant parallel time* instead, we get

  $T_p = fT + (1-f)T$

- For this to hold, we must assume that every time we add one to *p*, we also add another *(1-f)T* units of work to occupy the additional processor
- Since we're growing the size of the problem in proportion to the processor count, bigger problems will take longer to run sequentially:

  $T_s = fT + p(1-f)T$

Speedup (again)

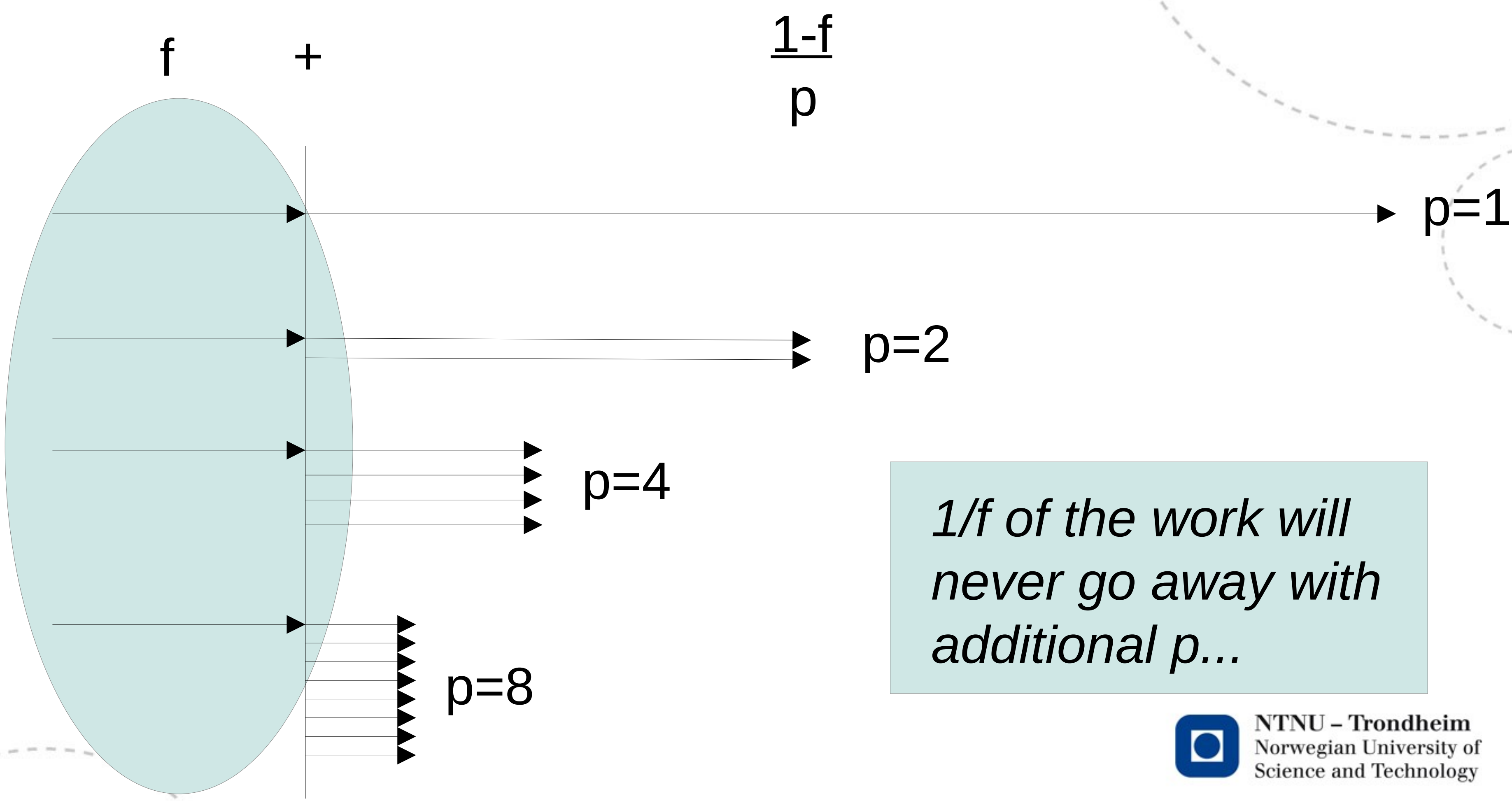
- Let us call this one "scaled speedup" S_s, so as not to confuse it with the other one:

$$S_s(p) = \frac{fT + p(1-f)T}{fT + (1-f)T} = \frac{f + p(1-f)}{f + 1 - f} = f + p(1-f)$$

- As you can see, $S_s(p)$ doesn't approach any limit as p grows
  - We're back in business!
    (...as long as we size up the problem in proportion to the machine)
- This observation is called *Gustafson's Law*

(time moves in this direction)

Amdahl, less formally



1/f of the work will never go away with additional p...

NTNU – Trondheim
Norwegian University of Science and Technology

(time moves in this direction)

Gustafson, less formally

(p-1) processors are idle for 1/f of the time

$$p-(p-1)f = f+p(1-f)$$

Mind the gap

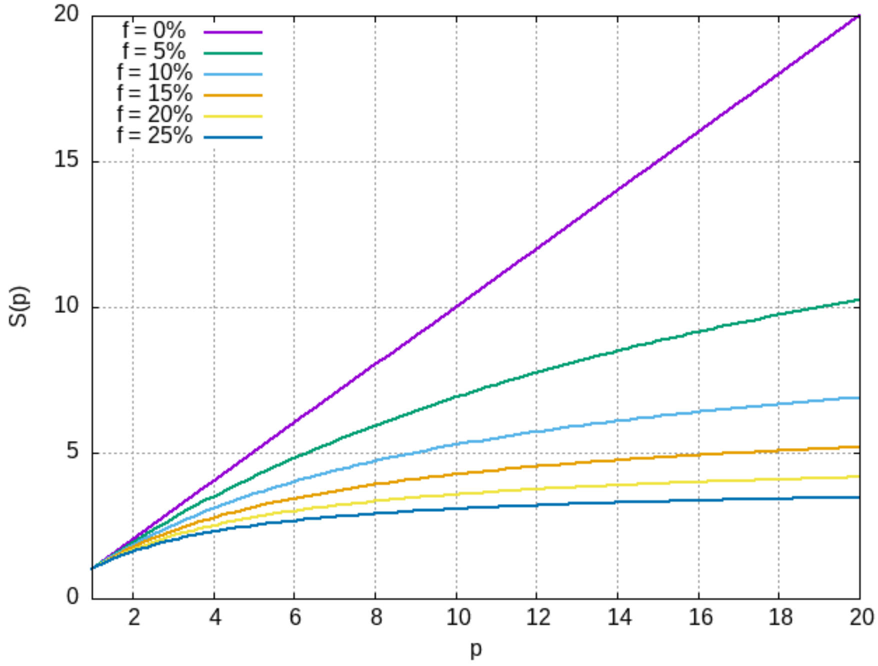
- The difference between speedup and scaled speedup is that

  "the program runs x times faster"

  has different meanings for each
- Speedup x says
  - "I can do the same work in 1/x time"
- Scaled speedup x says
  - "I can do x times the work in the same amount of time"
- You can relate them to each other if you want, but
- They're not directly comparable without a little bit of arithmetic in between

Strong scaling curves

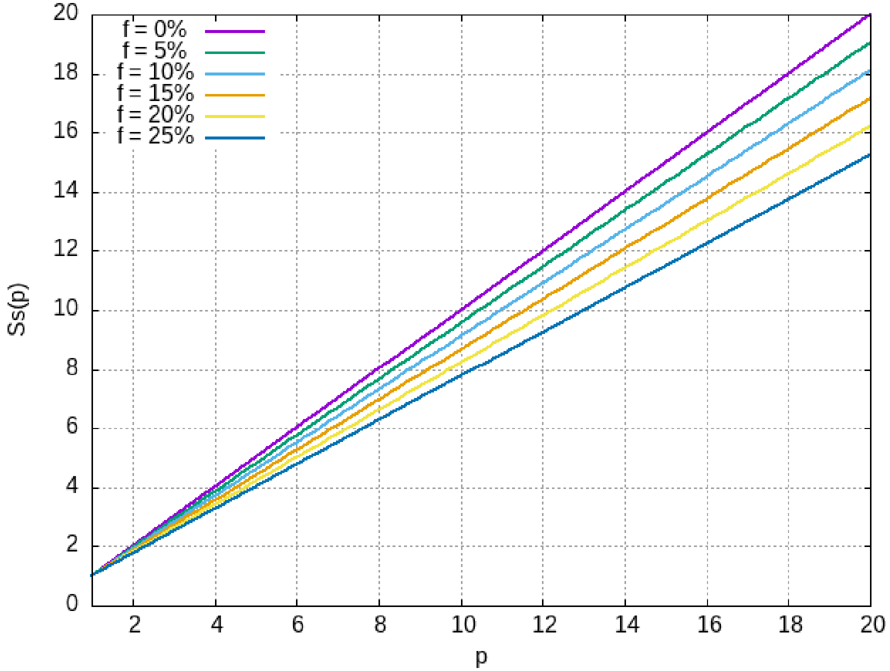
- Amdahl-flavor speedup curves look like this:



  – At best, they can be almost-linear for a while in the beginning, but they always level off and approach their asymptote toward infinity

Weak scaling curves

- Gustafson-flavor speedup curves look like this:



  - They're linear, but their gradient is not quite 1

Efficiency

- The formula for *parallel efficiency* is mercifully simple:

$$E(p) = \frac{S(p)}{p}$$

- The amount of speedup divided evenly among the processors that produced it gives us a sense of how much each one contributed
- When speedup is linear, efficiency is 1

  (In practice it drops with growing p, but hopefully as little as possible)

"Scalability"

- This word unfortunately has far too many definitions, and most of them are imprecise*
  (my favourite one is *"ability to imagine a slightly larger computer"*)
- It intuitively has something to do with capacity increases, though
- We can usefully refer to some related terms

NTNU – Trondheim
Norwegian University of
Science and Technology

* ***"What is scalability?"*** **, M.D. Hill, ACM SIGARCH Computer Architecture News, Vol. 18, No. 4, 1990**

Modes of scalability

- *Strong scaling*

  Studies of how speedup changes with processor count are said to be carried out in the strong mode

- *Weak scaling*

  Studies of how scaled speedup changes with processor count are said to be carried out in the weak mode

- “Strong” and “weak” are just some names, they tell you different things about system performance

  (It is a mistake to think that strong is better than weak, but the terms can have that effect on people who are not familiar with them)

Quantity *vs.* quality

- *Horizontal scaling*
  - Means to upgrade your system with more of the same components you had from before
  - Improves upon the parallel part of execution (when it works)
- *Vertical scaling*
  - Means to upgrade your system with more powerful components than the ones you had from before
  - Improves upon the sequential part of execution (when it works)

Theory & practice

- In theory, theory and practice are the same
  - In practice, they aren't
- We're pretending that the parallel part can always be *evenly* distributed among any number of processors
  - This is very rarely the case
- Gustafson pretends that we can grow the parallel workload without increasing the sequential
  - This is also very rare, but sequential growth is often small in comparison to the parallel part
- We *can* actually observe *superlinear* speedup S(p)>p
  - We'll return to the conditions required for this to happen