Amdahl’s law is real: there is a natural limit on speedup \textit{for a given problem}
Analysis
Speedup: Gustafson’s Law

Amdahl’s law is real: there is a natural limit on speedup for a given problem.

Gustafson pointed out that in real life larger machines tend to attract larger problems.
Amdahl’s law is real: there is a natural limit on speedup *for a given problem*

Gustafson pointed out that in real life larger machines tend to attract larger problems

Amdahl assumes a fixed size of problem
Amdahl’s law is real: there is a natural limit on speedup for a given problem.

Gustafson pointed out that in real life larger machines tend to attract larger problems.

Amdahl assumes a fixed size of problem.

*Gustafson’s Law* (occasionally called *Gustafson-Barsis’s Law*) has an alternative point of view.
Suppose we have a problem of size $n$

$$S_p(n) \leq \frac{1}{x_n + (1 - x_n)/p}$$

where $S_p(n)$ is the speedup on $p$ processors for a problem of size $n$; $x_n$ is the fraction of the computation spent sequentially
Suppose we have a problem of size $n$

$$S_p(n) \leq \frac{1}{x_n + (1 - x_n)/p}$$

where $S_p(n)$ is the speedup on $p$ processors for a problem of size $n$; $x_n$ is the fraction of the computation spent sequentially.

Gustafson argues: as $n$ gets larger, the sequential part relatively decreases, so $x_n \rightarrow 0$ ($p$ is fixed).
Analysis

Speedup: Gustafson’s Law

Suppose we have a problem of size $n$

$$S_p(n) \leq \frac{1}{x_n + (1 - x_n)/p}$$

where $S_p(n)$ is the speedup on $p$ processors for a problem of size $n$; $x_n$ is the fraction of the computation spent sequentially.

Gustafson argues: as $n$ gets larger, the sequential part relatively decreases, so $x_n \to 0$ ($p$ is fixed).

So

$$S_p(\infty) \leq p$$

i.e., we now get a speedup limit that is the “perfect” speedup $p$ — on an infinitely sized problem.
Analysis

Speedup: Amdahl’s Law, Gustafson’ Law

Both Amdahl and Gustafson are correct: they just apply to different cases of scaling
Both Amdahl and Gustafson are correct: they just apply to different cases of scaling

Amdahl: fixed problem, scaling processing power (sometimes called *strong scaling*)
Analysis

Speedup: Amdahl’s Law, Gustafson’ Law

Both Amdahl and Gustafson are correct: they just apply to different cases of scaling

Amdahl: fixed problem, scaling processing power (sometimes called *strong scaling*)

Gustafson: fixed processing power, scaling problem
Both Amdahl and Gustafson are correct: they just apply to different cases of scaling

Amdahl: fixed problem, scaling processing power (sometimes called *strong scaling*)

Gustafson: fixed processing power, scaling problem

This should convince you that even a simple measure like speedup can be problematic!
Both Amdahl and Gustafson are correct: they just apply to different cases of scaling

Amdahl: fixed problem, scaling processing power (sometimes called *strong scaling*)

Gustafson: fixed processing power, scaling problem

This should convince you that even a simple measure like speedup can be problematic!

But it does re-emphasise the fact that parallelism is not about making things faster, but about making things larger
Analysis

Speedup

Speedup is a simple measure, often proving that your parallel program is slower than it ought to be.
Speedup is a simple measure, often proving that your parallel program is slower than it ought to be.

Sometimes it takes $p$ to be surprisingly large before you even catch up with the uniprocessor time with $S_p = 1$ (sometimes never!)
Analysis

Speedup

Very common is the low start, a modest increase, then a tailing off
Analysis

Speedup

Very common is the low start, a modest increase, then a tailing off

But taking it further

We might find adding processors makes it slower!
This is usually due to increased communications between the processors adding more overhead but not more speedup, perhaps due to Amdahl
This is usually due to increased communications between the processors adding more overhead but not more speedup, perhaps due to Amdahl.

Of course, it’s not always this bad, but it’s quite common!
Analysis

Speedup

This is usually due to increased communications between the processors adding more overhead but not more speedup, perhaps due to Amdahl.

Of course, it’s not always this bad, but it’s quite common!

It does mean there is often an optimum number of processors for a given size of problem that achieves the best speedup.
This is usually due to increased communications between the processors adding more overhead but not more speedup, perhaps due to Amdahl.

Of course, it’s not always this bad, but it’s quite common!

It does mean there is often an optimum number of processors for a given size of problem that achieves the best speedup.

Of course, these are only typical behaviours: a given program may behave quite differently from all of this.
Analysis

Speedup

Exercise. Consider what might be the difference between a sequential implementation of something and a parallel implementation running on one processor.
You will get used to seeing $S_p < p$
You will get used to seeing $S_p < p$

On the other hand, it is possible that $S_p > p$
You will get used to seeing $S_p < p$

On the other hand, it is possible that $S_p > p$

This seemingly impossible condition is called superlinear speedup
You will get used to seeing $S_p < p$

On the other hand, it is possible that $S_p > p$

This seemingly impossible condition is called superlinear speedup.

It is quite rare in real life, but it really can happen that a program runs more than $p$ times as fast on $p$ processors.
Analysis
Superlinear Speedup

You will get used to seeing $S_p < p$

On the other hand, it is possible that $S_p > p$

This seemingly impossible condition is called superlinear speedup

It is quite rare in real life, but it really can happen that a program runs more than $p$ times as fast on $p$ processors

This can happen for a variety of reasons, some technological, and some more philosophical
Analysis
Superlinear Speedup

The first technological reason is due to cache memory
The first technological reason is due to cache memory.

Cache memory is a lot faster than main memory so if you can fit your problem entirely in cache, it will run faster.
The first technological reason is due to cache memory. Cache memory is a lot faster than main memory so if you can fit your problem entirely in cache, it will run faster.

For example, the Core i7: perhaps 200 cycles to access main memory, compared to 2 cycles for a L1 cache hit.
The first technological reason is due to cache memory

Cache memory is a lot faster than main memory so if you can fit your problem entirely in cache, it will run faster

For example, the Core i7: perhaps 200 cycles to access main memory, compared to 2 cycles for a L1 cache hit

$p$ processors might have $p$ times the cache of a single processor, so a problem spread across the processors might well fit in the larger amount of cache available
The first technological reason is due to cache memory

Cache memory is a lot faster than main memory so if you can fit your problem entirely in cache, it will run faster.

For example, the Core i7: perhaps 200 cycles to access main memory, compared to 2 cycles for a L1 cache hit.

$p$ processors might have $p$ times the cache of a single processor, so a problem spread across the processors might well fit in the larger amount of cache available.

Of course, this takes a certain kind of low-communication, easily dividable problem to work.
Note: modern CPUs tend to share cache across multiple cores, so it is unlikely $p$ cores has $p$ times as much cache.
Analysis
Superlinear Speedup

Note: modern CPUs tend to share cache across multiple cores, so it is unlikely \( p \) cores has \( p \) times as much cache

(This helps with cache coherence!)
Another (more philosophical) reason is due to the way speedup is defined:

\[ S_p = \frac{\text{time on a sequential processor}}{\text{time on } p \text{ parallel processors}} \]
Another (more philosophical) reason is due to the way speedup is defined

$$S_p = \frac{\text{time on a sequential processor}}{\text{time on } p \text{ parallel processors}}$$

What are we comparing against what?
Another (more philosophical) reason is due to the way speedup is defined

\[ S_p = \frac{\text{time on a sequential processor}}{\text{time on } p \text{ parallel processors}} \]

What are we comparing against what?

Here is an example to illustrate the issue
Another (more philosophical) reason is due to the way speedup is defined

\[ S_p = \frac{\text{time on a sequential processor}}{\text{time on } p \text{ parallel processors}} \]

What are we comparing against what?

Here is an example to illustrate the issue

We have bubblesort running on a uniprocessor: we wish to make it run on a parallel machine
Analysis

Superlinear Speedup

One way of doing this is:

- split the data into equal halves
- bubblesort each half in parallel
- merge the two sorted lists together
Analysis
Superlinear Speedup

One way of doing this is:

- split the data into equal halves
- bubblesort each half in parallel
- merge the two sorted lists together

This is 2-way parallelism
Analysis
Superlinear Speedup

One way of doing this is:

- split the data into equal halves
- bubblesort each half in parallel
- merge the two sorted lists together

This is 2-way parallelism

The middle step can be itself parallelised recursively: split into two, bubble and merge, giving 4-way parallelism
Analysis
Superlinear Speedup

One way of doing this is:

- split the data into equal halves
- bubblesort each half in parallel
- merge the two sorted lists together

This is 2-way parallelism

The middle step can be itself parallelised recursively: split into two, bubble and merge, giving 4-way parallelism

Depending on the number of processors we have, we can keep recursively dividing
Analysis
Superlinear Speedup

This seems like a reasonable way to implement bubblesort on a parallel machine.
This seems like a reasonable way to implement bubblesort on a parallel machine.

What is the speedup? We need to find out how long each version takes to run.
This seems like a reasonable way to implement bubblesort on a parallel machine.

What is the speedup? We need to find out how long each version takes to run.

Normal bubblesort takes time $n^2/2 + O(n)$ comparisons in the average case to sort $n$ items.
Analysis
Superlinear Speedup

This seems like a reasonable way to implement bubblesort on a parallel machine

What is the speedup? We need to find out how long each version takes to run

Normal bubblesort takes time $n^2/2 + O(n)$ comparisons in the average case to sort $n$ items

So bubblesorting the two halves (in parallel) takes time

$$(n/2)^2/2 + O(n/2) = n^2/8 + O(n)$$
Analysis
Superlinear Speedup

Merging $n$ values takes $O(n)$, giving a total of

$$n^2/8 + O(n) + O(n) = n^2/8 + O(n)$$

time
Merging $n$ values takes $O(n)$, giving a total of

$$n^2/8 + O(n) + O(n) = n^2/8 + O(n)$$

This gives speedup

$$S_2 = \frac{n^2/2 + O(n)}{n^2/8 + O(n)} \approx 4$$
Analysis
Superlinear Speedup

Merging $n$ values takes $O(n)$, giving a total of

$$n^2/8 + O(n) + O(n) = n^2/8 + O(n)$$

time

This gives speedup

$$S_2 = \frac{n^2/2 + O(n)}{n^2/8 + O(n)} \approx 4$$

Already superlinear!
On 4 processors we could repeat: the speedup we get is $S_4 \approx 16$
On 4 processors we could repeat: the speedup we get is $S_4 \approx 16$

Clearly this a wonderful algorithm
On 4 processors we could repeat: the speedup we get is $S_4 \approx 16$

Clearly this a wonderful algorithm

If we were to implement it, we would truly see these speedups
On 4 processors we could repeat: the speedup we get is $S_4 \approx 16$

Clearly this a wonderful algorithm

If we were to implement it, we would truly see these speedups

What is happening?
Consider the same subdividing algorithm on a *single processor*
Analysis
Superlinear Speedup

Consider the same subdividing algorithm on a *single processor*

Time to bubblesort halves: $2 \times (n^2/8 + O(n)) = n^2/4 + O(n)$;
time to merge $O(n)$; total $n^2/4 + O(n)$
Consider the same subdividing algorithm on a single processor

Time to bubblesort halves: \( 2 \times \left( \frac{n^2}{8} + O(n) \right) = \frac{n^2}{4} + O(n) \);
time to merge \( O(n) \); total \( \frac{n^2}{4} + O(n) \)

“Speedup”

\[
S_1 = \frac{n^2/2 + O(n)}{n^2/4 + O(n)} \approx 2
\]
Analysis
Superlinear Speedup

Consider the same subdividing algorithm on a single processor

Time to bubblesort halves: \( 2 \times (\frac{n^2}{8} + O(n)) = \frac{n^2}{4} + O(n) \);
time to merge \( O(n) \); total \( \frac{n^2}{4} + O(n) \)

“Speedup”

\[
S_1 = \frac{\frac{n^2}{2} + O(n)}{\frac{n^2}{4} + O(n)} \approx 2
\]

So we win even on a uniprocessor
Analysis
Superlinear Speedup

What is happening is that bubblesort is a really poor sorting algorithm.
What is happening is that bubblesort is a really poor sorting algorithm.

By subdividing and merging we are converting it into a different kind of sort: if we recurse all the way we have actually implemented a *merge* sort.
What is happening is that bubblesort is a really poor sorting algorithm.

By subdividing and merging we are converting it into a different kind of sort: if we recurse all the way we have actually implemented a *merge* sort.

Merge sort has complexity $O(n \log n)$.
Analysis
Superlinear Speedup

The point of this is that by converting bubblesort to be parallel in this way we are fundamentally changing it
The point of this is that by converting bubblesort to be parallel in this way we are fundamentally changing it.

This is an extreme case, but in general we must be care when computing speedups that we are comparing like with like.
The point of this is that by converting bubblesort to be parallel in this way we are fundamentally changing it.

This is an extreme case, but in general we must be care when computing speedups that we are comparing like with like.

It may not always be possible to have a suitable parallel version of an algorithm: in such a case “speedup” is not meaningful.
The point of this is that by converting bubblesort to be parallel in this way we are fundamentally changing it.

This is an extreme case, but in general we must be care when computing speedups that we are comparing like with like.

It may not always be possible to have a suitable parallel version of an algorithm: in such a case “speedup” is not meaningful.

In most real cases we don’t get this effect, but it’s worth being aware that it can happen.
Some people go further and define speedup as

$$S_p = \frac{\text{time of the best possible sequential algorithm}}{\text{time on } p \text{ parallel processors}}$$

but this has its own problems, not least that we might not know the best possible sequential way of doing things.
Some people go further and define speedup as

\[ S_p = \frac{\text{time of the best possible sequential algorithm}}{\text{time on } p \text{ parallel processors}} \]

but this has its own problems, not least that we might not know the best possible sequential way of doing things.

And we now might be comparing two completely unrelated algorithms.
In a similar vein, another reason for getting superlinear speedups is that the original, sequential, program was poorly written.
In a similar vein, another reason for getting superlinear speedups is that the original, sequential, program was poorly written.

Perhaps the programmer spent more time thinking about the parallel version, or gained more experience from writing the sequential version, making it substantially better code than the sequential version.
In a similar vein, another reason for getting superlinear speedups is that the original, sequential, program was poorly written.

Perhaps the programmer spent more time thinking about the parallel version, or gained more experience from writing the sequential version, making it substantially better code than the sequential version.

This is much the same as the “transform bad algorithm to better algorithm” above, but is now “transform bad code to better code.”
In a similar vein, another reason for getting superlinear speedups is that the original, sequential, program was poorly written

Perhaps the programmer spent more time thinking about the parallel version, or gained more experience from writing the sequential version, making it substantially better code than the sequential version

This is much the same as the “transform bad algorithm to better algorithm” above, but is now “transform bad code to better code”

So, again, we are not really comparing like with like
Analysis

Speedup

And occasionally we see superlinear speedup due to randomness
And occasionally we see superlinear speedup due to randomness.

If the data contains random numbers, or there is something that adds an element of randomness to the run time, we can get a superlinear speedup.
And occasionally we see superlinear speedup due to randomness

If the data contains random numbers, or there is something that adds an element of randomness to the run time we can get a superlinear speedup

This time due to the parallel version “getting lucky” and hitting a special case that finishes early relative to your measured sequential version
And occasionally we see superlinear speedup due to randomness

If the data contains random numbers, or there is something that adds an element of randomness to the run time we can get a superlinear speedup

This time due to the parallel version “getting lucky” and hitting a special case that finishes early relative to your measured sequential version

So also not comparing like with like
Analysis

Speedup

And occasionally we see superlinear speedup due to randomness.

If the data contains random numbers, or there is something that adds an element of randomness to the run time we can get a superlinear speedup.

This time due to the parallel version “getting lucky” and hitting a special case that finishes early relative to your measured sequential version.

So also not comparing like with like.

You would need to ensure each run had the same randomness to be properly comparable; or run many times and take an average time.
Analysis

Speedup

In conclusion: speedup is a nice and simple, easy to understand measure: but we have to take care over what it is telling us
In conclusion: speedup is a nice and simple, easy to understand measure: but we have to take care over what it is telling us.

Some problems are *pathologically parallel*, meaning they fall easily into parallel parts that have a minimum of communication.
In conclusion: speedup is a nice and simple, easy to understand measure: but we have to take care over what it is telling us.

Some problems are *pathologically parallel*, meaning they fall easily into parallel parts that have a minimum of communication.

For such problems it is easy to get good speedups.
In conclusion: speedup is a nice and simple, easy to understand measure: but we have to take care over what it is telling us.

Some problems are *pathologically parallel*, meaning they fall easily into parallel parts that have a minimum of communication.

For such problems it is easy to get good speedups.

E.g., graphics rendering, weather forecasting, parameter sweeping, etc. Often they are data parallel problems.
Analysis

Speedup

In conclusion: speedup is a nice and simple, easy to understand measure: but we have to take care over what it is telling us

Some problems are *pathologically parallel*, meaning they fall easily into parallel parts that have a minimum of communication

For such problems it is easy to get good speedups

E.g., graphics rendering, weather forecasting, parameter sweeping, etc. Often they are data parallel problems

Other problems fare less well — in terms of speed — from parallelisation!