Analysis

Speedup: Amdahl's Law

Now there is the natural upper bound of $S_p \leq p$: we wouldn't expect to get more speedup than the number of processors we have

Analysis

Speedup: Amdahl's Law

Now there is the natural upper bound of $S_p \leq p$: we wouldn't expect to get more speedup than the number of processors we have

But it turns out that the number of processors is generally not the limiting factor on speedup: there is another fundamental restriction on speedup that is often overlooked

Analysis

Speedup: Amdahl's Law

Now there is the natural upper bound of $S_p \leq p$: we wouldn't expect to get more speedup than the number of processors we have

But it turns out that the number of processors is generally not the limiting factor on speedup: there is another fundamental restriction on speedup that is often overlooked

Amdahl's Law reveals a natural upper bound on the speedup that is theoretically possible even before we add in implementation overheads

Analysis

Speedup: Amdahl's Law

Suppose we have a problem of which 90% can be run in parallel, leaving 10% sequential code

Analysis

Speedup: Amdahl's Law

Suppose we have a problem of which 90% can be run in parallel, leaving 10% sequential code

For example, we have to read data before we can process it: a necessary sequentiality. Similarly for writing after processing. Or the add after the square in $x^2 + 1$

Analysis

Speedup: Amdahl's Law

Suppose we have a problem of which 90% can be run in parallel, leaving 10% sequential code

For example, we have to read data before we can process it: a necessary sequentiality. Similarly for writing after processing. Or the add after the square in $x^2 + 1$

So there's always *some* sequentiality

Analysis

Speedup: Amdahl's Law

Suppose we have a problem of which 90% can be run in parallel, leaving 10% sequential code

For example, we have to read data before we can process it: a necessary sequentiality. Similarly for writing after processing. Or the add after the square in $x^2 + 1$

So there's always *some* sequentiality

But in the best possible case, using an unlimited number of processors, we might be able to get the parallel part down to as close to zero time as we like

Analysis

Speedup: Amdahl's Law

Suppose we have a problem of which 90% can be run in parallel, leaving 10% sequential code

For example, we have to read data before we can process it: a necessary sequentiality. Similarly for writing after processing. Or the add after the square in $x^2 + 1$

So there's always *some* sequentiality

But in the best possible case, using an unlimited number of processors, we might be able to get the parallel part down to as close to zero time as we like

We still have the 10% sequential part

Analysis

Speedup: Amdahl's Law

So the speedup is

$$S_\infty = \frac{\text{time on a sequential processor}}{\text{time on parallel processors}} = \frac{100}{10} = 10$$

A speedup of 10 even on an infinite number of processors

So the speedup is

$$S_\infty = \frac{\text{time on a sequential processor}}{\text{time on parallel processors}} = \frac{100}{10} = 10$$

A speedup of 10 even on an infinite number of processors

It doesn't even matter what the problem is, or what hardware we have

This is Amdahl's Law:

> *Every program has a natural limit on the maximum speedup it can attain, regardless of the number of processors used*

Analysis

Speedup: Amdahl's Law

We can quantify Amdahl's Law:

Let $T = T_{\text{seq}} + T_{\text{par}}$ be the time spent in the sequential and parallel parts of our problem on a sequential processor

Analysis

Speedup: Amdahl's Law

We can quantify Amdahl's Law:

Let $T = T_{\text{seq}} + T_{\text{par}}$ be the time spent in the sequential and parallel parts of our problem on a sequential processor

Then the *maximum* speedup S_p using p processors on the parallel part is

$$S_p \leq \frac{T_{\text{seq}} + T_{\text{par}}}{T_{\text{seq}} + T_{\text{par}}/p}$$

where we have perfectly parallelised the parallel part

Analysis

Speedup: Amdahl's Law

Thus there is a natural upper limit on how fast programs can go

Analysis

Speedup: Amdahl's Law

Thus there is a natural upper limit on how fast programs can go

Most do I/O, which must be serialised (made sequential)

Analysis

Speedup: Amdahl's Law

Thus there is a natural upper limit on how fast programs can go

Most do I/O, which must be serialised (made sequential)

Further, as $p \to \infty$, we get

$$S_\infty \leq \frac{T_{\mathrm{seq}} + T_{\mathrm{par}}}{T_{\mathrm{seq}}}$$

so there is a limit even given infinite processing power

Analysis

Speedup: Amdahl's Law

Thus there is a natural upper limit on how fast programs can go

Most do I/O, which must be serialised (made sequential)

Further, as $p \to \infty$, we get

$$S_\infty \leq \frac{T_{\text{seq}} + T_{\text{par}}}{T_{\text{seq}}}$$

so there is a limit even given infinite processing power

This limit is determined by the time taken in the sequential part of the computation

Analysis

Speedup: Amdahl's Law

To see this consider the fraction $x = T_{\text{seq}}/(T_{\text{seq}} + T_{\text{par}})$ which is the proportion of the sequential part within the whole

To see this consider the fraction $x = T_{\text{seq}}/(T_{\text{seq}} + T_{\text{par}})$ which is the proportion of the sequential part within the whole

Note that $0 \leq x \leq 1$, and that rearranging the above gives

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

To see this consider the fraction $x = T_{\text{seq}}/(T_{\text{seq}} + T_{\text{par}})$ which is the proportion of the sequential part within the whole

Note that $0 \leq x \leq 1$, and that rearranging the above gives

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

And so

$$S_\infty \leq \frac{1}{x}$$

is bounded

Analysis

Speedup: Amdahl's Law

Note that Amdahl **does not say anything about how the speedup varies with** p

Analysis

Speedup: Amdahl's Law

Note that Amdahl **does not say anything about how the speedup varies with** p

All Amdahl says is that an upper limit exists

Analysis

Speedup: Amdahl's Law

Note that Amdahl **does not say anything about how the speedup varies with** p

All Amdahl says is that an upper limit exists

Your program may not get anywhere close to that limit and often in real programs, does not

Analysis

Speedup: Amdahl's Law

In real programs, there are many other factors that affect speedup, so that the speedup may well vary all over the place as p increases

Analysis

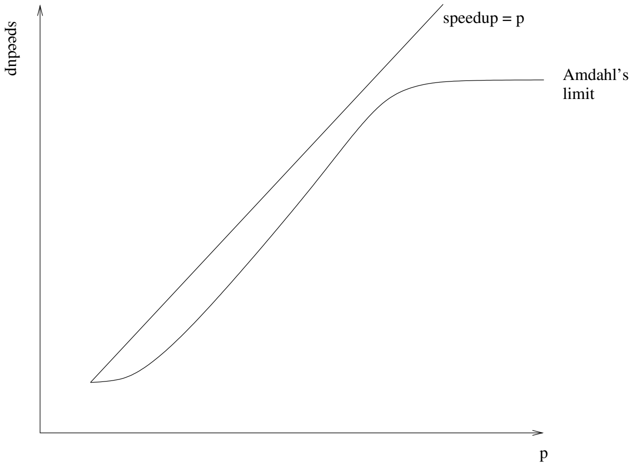
Speedup: Amdahl's Law

In real programs, there are many other factors that affect speedup, so that the speedup may well vary all over the place as p increases

It can even decrease as p gets larger

Analysis

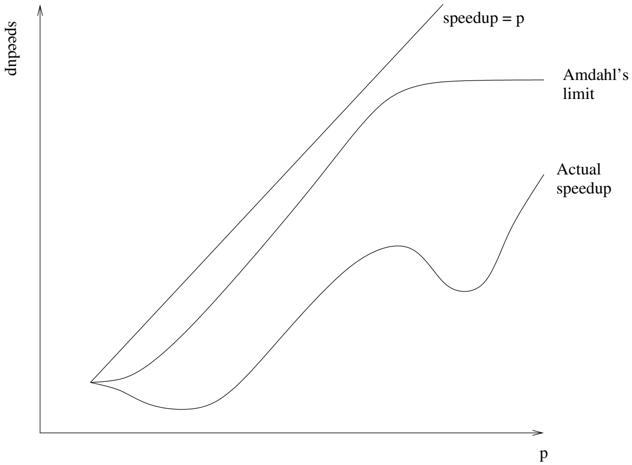
Speedup: Amdahl's Law



Speedup in theory

Analysis

Speedup: Amdahl's Law



Speedup in practice

Analysis

Speedup: Amdahl's Law

To emphasize: all we know is that actual speedup is below Amdahl's limit

To emphasize: all we know is that actual speedup is below Amdahl's limit

Exercise Show that if $0 \leq x \leq 1$, then

$$\frac{1}{x + (1 - x)/p} \leq p$$

Exercise What is the maximum speedup of a program that is 100% sequential?

Analysis

Speedup: Gustafson's Law

Amdahl's law is real: there is a natural limit on speedup *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*

But there's another point of view

Analysis

Speedup: Gustafson's Law

Amdahl's law is real: there is a natural limit on speedup *for a given problem*

But there's another point of view

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

Analysis

Speedup: Gustafson's Law

Amdahl's law is real: there is a natural limit on speedup *for a given problem*

But there's another point of view

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

Amdahl assumes a fixed size of problem

Analysis

Speedup: Gustafson's Law

Amdahl's law is real: there is a natural limit on speedup *for a given problem*

But there's another point of view

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*) gives us another limit

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

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)

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

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*)

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

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

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

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

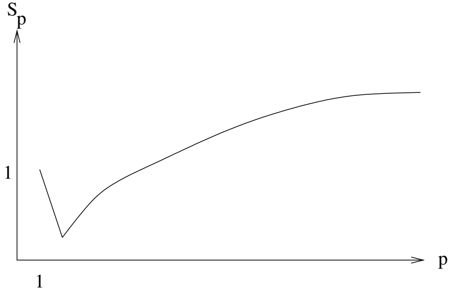
This should convince you that even a simple measure like speedup can be problematic!

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



Typical speedup curve

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

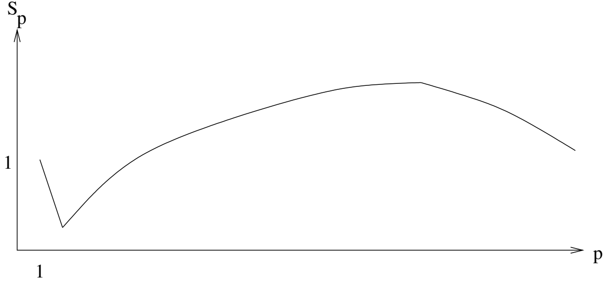
Very common is the low start, a modest increase, then a tailing off

Analysis

Speedup

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

But taking it further



Adding more processors

We might eventually find adding processors makes it slower!

Analysis

Speedup

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

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!

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

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

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

Exercise Consider what might be the difference between a sequential implementation of something and a parallel implementation running on one processor

Analysis

Superlinear Speedup

You will get used to seeing $S_p < p$

Analysis

Superlinear Speedup

You will get used to seeing $S_p < p$

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

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*

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

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

Analysis

Superlinear Speedup

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

Analysis

Superlinear Speedup

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

Analysis

Superlinear Speedup

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, a 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

Analysis

Superlinear Speedup

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, a 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; and the right hardware

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

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!)

Analysis

Superlinear Speedup

Another (more philosophical) reason is due to the way speedup is defined

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

Analysis

Superlinear Speedup

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?

Analysis

Superlinear Speedup

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

Analysis

Superlinear Speedup

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

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

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

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

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

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

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!

Analysis

Superlinear Speedup

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

Analysis

Superlinear Speedup

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

Clearly this a wonderful algorithm

Analysis

Superlinear Speedup

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

Analysis

Superlinear Speedup

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?

Analysis

Superlinear Speedup

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 $n^2/4 + O(n)$

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 $n^2/4 + O(n)$

"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: $2 \times (n^2/8 + O(n)) = n^2/4 + O(n)$; time to merge $O(n)$; total $n^2/4 + O(n)$

"Speedup"

$$S_1 = \frac{n^2/2 + O(n)}{n^2/4 + O(n)} \approx 2$$

So we win even on a uniprocessor

Analysis

Superlinear Speedup

What is happening is that bubblesort is a really poor sorting algorithm on average

Analysis

Superlinear Speedup

What is happening is that bubblesort is a really poor sorting algorithm on average

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

Analysis

Superlinear Speedup

What is happening is that bubblesort is a really poor sorting algorithm on average

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

Analysis

Superlinear Speedup

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

Analysis

Superlinear Speedup

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

Analysis

Superlinear Speedup

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

Analysis

Speedup

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

Analysis

Speedup

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

Analysis

Speedup

In a similar vein, another reason for getting superlinear speedups is that the original, sequential, program was poorly written

Analysis

Speedup

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

Analysis

Speedup

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”

Analysis

Speedup

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

Analysis

Speedup

And occasionally we see superlinear speedup due to randomness

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

Analysis

Speedup

And occasionally we see superlinear speedup due to randomness

If the data contains random numbers, or there is something that adds an elements 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

Analysis

Speedup

And occasionally we see superlinear speedup due to randomness

If the data contains random numbers, or there is something that adds an elements 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 elements 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

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

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

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

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!