Analysis

So we need to look at how to analyse parallel algorithms
Analysis

So we need to look at how to analyse parallel algorithms

Analysis of parallel algorithms is like analysis of sequential algorithms, just more complicated
Analysis

So we need to look at how to analyse parallel algorithms.

Analysis of parallel algorithms is like analysis of sequential algorithms, just more complicated.

Later we shall see statements like “this takes time $O(n^2)$ using $O(p)$ processors.”
Analysis

So we need to look at how to analyse parallel algorithms.

Analysis of parallel algorithms is like analysis of sequential algorithms, just more complicated.

Later we shall see statements like “this takes time $O(n^2)$ using $O(p)$ processors”.

But we shall start with a few simple measures that we can use to indicate how well our parallel algorithms are working.
So we need to look at how to analyse parallel algorithms

Analysis of parallel algorithms is like analysis of sequential algorithms, just more complicated

Later we shall see statements like “this takes time $O(n^2)$ using $O(p)$ processors”

But we shall start with a few simple measures that we can use to indicate how well our parallel algorithms are working

They are quite crude, but effective
They mostly measure the parallel algorithm in comparison with a corresponding sequential algorithm.
Analysis

Speedup

They mostly measure the parallel algorithm in comparison with a corresponding sequential algorithm.

Or a parallel *implementation* with a corresponding sequential implementation.

*Ideally we’d like $S_p = p$, but this never happens.*
Analysis

Speedup

They mostly measure the parallel algorithm in comparison with a corresponding sequential algorithm.

Or a parallel *implementation* with a corresponding sequential implementation.

We have seen that having $p$ processors won’t necessarily make our program run $p$ times as fast.
Analysis

Speedup

They mostly measure the parallel algorithm in comparison with a corresponding sequential algorithm

Or a parallel *implementation* with a corresponding sequential implementation

We have seen that having $p$ processors won’t necessarily make our program run $p$ times as fast

The *speedup* using $p$ processors is

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

Speedup

They mostly measure the parallel algorithm in comparison with a corresponding sequential algorithm

Or a parallel *implementation* with a corresponding sequential implementation

We have seen that having $p$ processors won’t necessarily make our program run $p$ times as fast

The *speedup* using $p$ processors is

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

Ideally we’d like $S_p = p$, but this never happens
Analysis

Speedup

Usually $S_p$ is much smaller than $p$ for several reasons
Analysis

Speedup

Usually $S_p$ is much smaller than $p$ for several reasons

Firstly, there is communications overheads between processors
Analysis

Speedup

Usually $S_p$ is much smaller than $p$ for several reasons.

Firstly, there is communications overheads between processors.

This might be fairly small for shared memory, or large for distributed memory, but it is present.
Analysis

Speedup

Usually $S_p$ is much smaller than $p$ for several reasons.

Firstly, there is communications overheads between processors.

This might be fairly small for shared memory, or large for distributed memory, but it is present.

Time spent communicating is time not spent computing.
So more communications (data movement) will tend to lead to smaller speedups.
So more communications (data movement) will tend to lead to smaller speedups

For example, speedups on distributed memory machines can be reduced as the cost of communications is quite high.
So more communications (data movement) will tend to lead to smaller speedups.

For example, speedups on distributed memory machines can be reduced as the cost of communications is quite high.

But speedups can improve for a larger computation where the relative cost of communications drops.
So more communications (data movement) will tend to lead to smaller speedups.

For example, speedups on distributed memory machines can be reduced as the cost of communications is quite high.

But speedups can improve for a larger computation where the relative cost of communications drops.

Remember clusters are used for large problems where the emphasis is on size, not speed.
In really bad cases, $S_p < 1$, i.e., our parallel program goes slower than our sequential program even though we’ve thrown all this expensive hardware at it!
Analysis

Slowdown

In really bad cases, $S_p < 1$, i.e., our parallel program goes slower than our sequential program even though we’ve thrown all this expensive hardware at it!

This is more common than we’d like.
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.
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* puts 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 $x^2 + 1$. 

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.
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
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.
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 \textit{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

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

so there is a limit even given infinite processing power.
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 \( x = \frac{T_{\text{seq}}}{T_{\text{seq}} + T_{\text{par}}} \) which is the proportion of the sequential part within the whole.
To see this consider the fraction \( x = \frac{T_{seq}}{T_{seq} + T_{par}} \) which is the proportion of the sequential part within the whole.

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

\[
S_p \leq \frac{1}{x + \frac{1 - x}{p}}
\]
Analysis
Speedup: Amdahl’s Law

To see this consider the fraction $x = \frac{T_{\text{seq}}}{(T_{\text{seq}} + T_{\text{par}})}$ 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
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

\[
\text{speedup} = p
\]

Amdahl’s limit
Analysis
Speedup: Amdahl’s Law

speedup = p

Amdahl’s limit

Actual speedup
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

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