Analysis

Efficiency

If we are lucky enough that S_p increases with p we can make our program get faster by adding more processors

Analysis

Efficiency

If we are lucky enough that S_p increases with p we can make our program get faster by adding more processors

But at what cost?

Analysis

Efficiency

If we are lucky enough that S_p increases with p we can make our program get faster by adding more processors

But at what cost?

If we can double the speed of a program using 4 processors we feel we are doing better than if we used a different approach that needed 8 processors for the same speedup

Analysis

Efficiency

If we are lucky enough that S_p increases with p we can make our program get faster by adding more processors

But at what cost?

If we can double the speed of a program using 4 processors we feel we are doing better than if we used a different approach that needed 8 processors for the same speedup

Efficiency measures this

Analysis

Efficiency

Efficiency is speedup per processor:

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

Analysis

Efficiency

Efficiency is speedup per processor:

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

Usually $0 \leq E_p \leq 1$, and is often written as a percentage

Analysis

Efficiency

Efficiency is speedup per processor:

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

Usually $0 \leq E_p \leq 1$, and is often written as a percentage

$E_p = 0.5$ (50%) means we are using only half of the processors' capabilities on our computation; half is lost in overheads or idling while waiting for something

Analysis

Efficiency

Efficiency is speedup per processor:

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

Usually $0 \leq E_p \leq 1$, and is often written as a percentage

$E_p = 0.5$ (50%) means we are using only half of the processors' capabilities on our computation; half is lost in overheads or idling while waiting for something

$E_p = 1$ (100%) means we are using all the processors all the time on our computation

Analysis

Efficiency

Efficiency is speedup per processor:

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

Usually $0 \leq E_p \leq 1$, and is often written as a percentage

$E_p = 0.5$ (50%) means we are using only half of the processors' capabilities on our computation; half is lost in overheads or idling while waiting for something

$E_p = 1$ (100%) means we are using all the processors all the time on our computation

$E_p > 1$ indicate superlinear speedup: we are using more than 100% of the processors!

Analysis

Efficiency

Efficiency is useful when we need to gauge the cost of a parallel system: the higher the efficiency the better the utilisation of the processors

Analysis

Efficiency

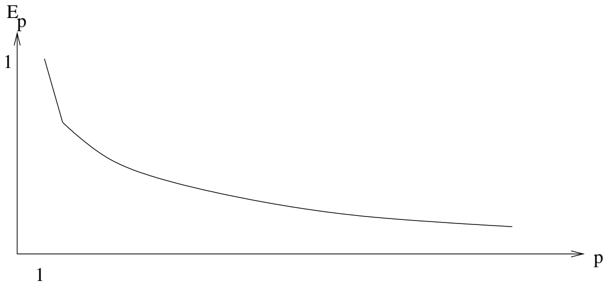
Efficiency is useful when we need to gauge the cost of a parallel system: the higher the efficiency the better the utilisation of the processors

When $E_p < 1$ this indicates that somewhere at some point a processor not working on the computation. Perhaps it is occupied in communication; or possibly just lying idle waiting

Analysis

Efficiency

Typical efficiency graph on a fixed size problem:



Efficiency graph dropoff

Analysis

Speedup and Efficiency

As an example of calculating speedup and efficiency we consider a pipeline (systolic array)

Systolic array

Analysis

Speedup and Efficiency

As an example of calculating speedup and efficiency we consider a pipeline (systolic array)

Systolic array

Data moves from one processor to the next being transformed at each stage: we assume one time step per transform

Analysis

Speedup and Efficiency

As an example of calculating speedup and efficiency we consider a pipeline (systolic array)

Systolic array

Data moves from one processor to the next being transformed at each stage: we assume one time step per transform

This could equally be a CPU instruction pipeline

Analysis

Speedup and Efficiency

A p-stage pipeline will take p time steps to fill; after that it produces one result per time step

Analysis

Speedup and Efficiency

A p-stage pipeline will take p time steps to fill; after that it produces one result per time step

So it can produce n results in $p + n - 1$ time steps

Analysis

Speedup and Efficiency

A p-stage pipeline will take p time steps to fill; after that it produces one result per time step

So it can produce n results in $p + n - 1$ time steps

A sequential system will take np time steps to do the p steps on the n computations

Analysis

Speedup and Efficiency

The speedup is

$$S_p = \frac{np}{p+n-1} = \frac{p}{(p-1)/n+1}$$

Analysis

Speedup and Efficiency

The speedup is

$$S_p = \frac{np}{p + n - 1} = \frac{p}{(p-1)/n + 1}$$

As time passes, the number of tasks n gets large, and $S_p \to p$

Analysis

Speedup and Efficiency

The speedup is

$$S_p = \frac{np}{p + n - 1} = \frac{p}{(p-1)/n + 1}$$

As time passes, the number of tasks n gets large, and $S_p \to p$

A p-stage pipeline has a speedup is less than p, but that gets closer to p as time progresses

Analysis

Speedup and Efficiency

The speedup is

$$S_p = \frac{np}{p+n-1} = \frac{p}{(p-1)/n+1}$$

As time passes, the number of tasks n gets large, and $S_p \to p$

A p-stage pipeline has a speedup is less than p, but that gets closer to p as time progresses

Also, the speedup starts low (for $n = 1$, $S_p = p/(p+1-1) = 1$) and increases over time, getting closer and closer to p

Analysis

Speedup and Efficiency

The efficiency is

$$E_p = \frac{S_p}{p} = \frac{n}{p+n-1} = \frac{1}{(p-1)/n+1}$$

Analysis

Speedup and Efficiency

The efficiency is

$$E_p = \frac{S_p}{p} = \frac{n}{p+n-1} = \frac{1}{(p-1)/n+1}$$

As n gets large, $E_p \to 1$

Analysis

Speedup and Efficiency

The efficiency is

$$E_p = \frac{S_p}{p} = \frac{n}{p+n-1} = \frac{1}{(p-1)/n+1}$$

As n gets large, $E_p \to 1$

Eventually we are (close to) using all the processors all the time: perfect efficiency!

Analysis

Speedup and Efficiency

The efficiency is

$$E_p = \frac{S_p}{p} = \frac{n}{p+n-1} = \frac{1}{(p-1)/n+1}$$

As n gets large, $E_p \to 1$

Eventually we are (close to) using all the processors all the time: perfect efficiency!

Also, the efficiency starts low (for $n = 1$, $E_p = 1/(p+1-1) = 1/p$) and increases over time

Analysis

Speedup and Efficiency

Pipelines are a really good way of making something parallel: both great speedup and great efficiency

Analysis

Speedup and Efficiency

Pipelines are a really good way of making something parallel: both great speedup and great efficiency

As long as we can keep the pipeline full: in a CPU each time we take a jump the instruction pipeline breaks, empties and needs to refill

Analysis

Speedup and Efficiency

Pipelines are a really good way of making something parallel: both great speedup and great efficiency

As long as we can keep the pipeline full: in a CPU each time we take a jump the instruction pipeline breaks, empties and needs to refill

To keep high efficiency we need to avoid this: thus the complications in the designs of modern processors that are aimed at keeping the pipeline full

Analysis

Speedup and Efficiency

Pipelines are a really good way of making something parallel: both great speedup and great efficiency

As long as we can keep the pipeline full: in a CPU each time we take a jump the instruction pipeline breaks, empties and needs to refill

To keep high efficiency we need to avoid this: thus the complications in the designs of modern processors that are aimed at keeping the pipeline full

(Things like speculative evaluation and branch prediction, using many transistors. . .)

Analysis

Other measures

Speedup and Efficiency are simple, but useful measures of a parallel system, as long as you take care over using them

Analysis

Other measures

Speedup and Efficiency are simple, but useful measures of a parallel system, as long as you take care over using them

There are many other measures that are occasionally used, but they are of lesser importance

Analysis

Other measures

Speedup and Efficiency are simple, but useful measures of a parallel system, as long as you take care over using them

There are many other measures that are occasionally used, but they are of lesser importance

Exercise Some people use the phrase “negative speedup” rather than “slowdown”. Why is that incorrect?

Analysis

Karp-Flatt

Sometimes people use the *Karp-Flatt metric* as a measure of an implementation to see how well it is doing

Analysis

Karp-Flatt

Sometimes people use the *Karp-Flatt metric* as a measure of an implementation to see how well it is doing

This is essentially an empirical measure of the sequential fraction of a computation (important for the Amdahl limit)

Analysis

Karp-Flatt

Sometimes people use the *Karp-Flatt metric* as a measure of an implementation to see how well it is doing

This is essentially an empirical measure of the sequential fraction of a computation (important for the Amdahl limit)

$$e = \frac{\frac{1}{S_p} - \frac{1}{p}}{1 - \frac{1}{p}}$$

where S_p is the measured speedup and p the number of processors

Analysis

Karp-Flatt

A larger e indicates a larger sequential part

Analysis

Karp-Flatt

A larger e indicates a larger sequential part

If we have perfect speedup, $S_p = p$, and $e = 0$

Analysis

Karp-Flatt

A larger e indicates a larger sequential part

If we have perfect speedup, $S_p = p$, and $e = 0$

If we have no speedup, $S_p = 1$, and $e = 1$

Analysis

Karp-Flatt

A larger e indicates a larger sequential part

If we have perfect speedup, $S_p = p$, and $e = 0$

If we have no speedup, $S_p = 1$, and $e = 1$

If we have slowdown, e.g., $S_p = 1/2$, and $e \approx 2$

Analysis

Karp-Flatt

A larger e indicates a larger sequential part

If we have perfect speedup, $S_p = p$, and $e = 0$

If we have no speedup, $S_p = 1$, and $e = 1$

If we have slowdown, e.g., $S_p = 1/2$, and $e \approx 2$

(If we have superlinear speedup, $S_p > p$, and $e < 0$)

Analysis

Karp-Flatt

A larger e indicates a larger sequential part

If we have perfect speedup, $S_p = p$, and $e = 0$

If we have no speedup, $S_p = 1$, and $e = 1$

If we have slowdown, e.g., $S_p = 1/2$, and $e \approx 2$

(If we have superlinear speedup, $S_p > p$, and $e < 0$)

Exercise Calculate Karp-Flatt for the pipeline. What does it tell us?

Analysis

Karp-Flatt

Note that Karp-Flatt will give you an estimate for the sequential time *for a given implementation*

Analysis

Karp-Flatt

Note that Karp-Flatt will give you an estimate for the sequential time *for a given implementation*

It does not tell us the sequential limit for the *problem*

Analysis

Karp-Flatt

Note that Karp-Flatt will give you an estimate for the sequential time *for a given implementation*

It does not tell us the sequential limit for the *problem*

After all, you might just have a poor implementation

Analysis

Karp-Flatt

Note that Karp-Flatt will give you an estimate for the sequential time *for a given implementation*

It does not tell us the sequential limit for the *problem*

After all, you might just have a poor implementation

A big Karp-Flatt value is often an indication you need to re-think your code

Analysis

Work Efficient

Next: a parallel algorithm is *work efficient* (*cost efficient*) if the number of operations it performs is no more than the sequential algorithm

Analysis

Work Efficient

Next: a parallel algorithm is *work efficient* (*cost efficient*) if the number of operations it performs is no more than the sequential algorithm

For example, a parallel algorithm might duplicate some operations on separate processors as it is more convenient, or reduces communications

Analysis

Work Efficient

Next: a parallel algorithm is *work efficient* (*cost efficient*) if the number of operations it performs is no more than the sequential algorithm

For example, a parallel algorithm might duplicate some operations on separate processors as it is more convenient, or reduces communications

The *parallel overhead* is

$$T_o = pT_p - T_s$$

where T_s is the sequential time and T_p is the parallel time

Analysis

Work Efficient

This measures the amount of extra work we are doing to get the parallelism

Analysis

Work Efficient

This measures the amount of extra work we are doing to get the parallelism

A measure of the extra energy expended in the parallel algorithm or implementation

Analysis

Work Efficient

This measures the amount of extra work we are doing to get the parallelism

A measure of the extra energy expended in the parallel algorithm or implementation

And the cost of the overheads (e.g., communication) when we measure a real implementation

Analysis

Work Efficient

This measures the amount of extra work we are doing to get the parallelism

A measure of the extra energy expended in the parallel algorithm or implementation

And the cost of the overheads (e.g., communication) when we measure a real implementation

Exercise Calculate the parallel overhead for the pipeline. What does it tell us?

Another question is “how scalable is this algorithm?”

Analysis

Isoefficiency

Another question is “how scalable is this algorithm?”

Here we ask for a relationship between p, the number of processors and n the size of the problem for a given efficiency

Analysis

Isoefficiency

Another question is “how scalable is this algorithm?”

Here we ask for a relationship between p, the number of processors and n the size of the problem for a given efficiency

If we increase p, how much to we have to increase n to maintain a given efficiency?

Increasing p will generally decrease efficiency (Amdahl)

Analysis

Isoefficiency

Increasing p will generally decrease efficiency (Amdahl)

Increasing n will generally increase efficiency (Gustafson)

Analysis

Isoefficiency

Increasing p will generally decrease efficiency (Amdahl)

Increasing n will generally increase efficiency (Gustafson)

A poorly scalable algorithm will need to increase n a lot to maintain efficiency as we increase p

Analysis

Isoefficiency

Increasing p will generally decrease efficiency (Amdahl)

Increasing n will generally increase efficiency (Gustafson)

A poorly scalable algorithm will need to increase n a lot to maintain efficiency as we increase p

This relationship is called the *isoefficiency*, and expresses n as a function of p

Analysis

Isoefficiency

Increasing p will generally decrease efficiency (Amdahl)

Increasing n will generally increase efficiency (Gustafson)

A poorly scalable algorithm will need to increase n a lot to maintain efficiency as we increase p

This relationship is called the *isoefficiency*, and expresses n as a function of p

It quantifies the balance between Amdahl and Gustafson

Analysis

Isoefficiency

Computing the isoefficiency can be a bit fiddly, but often it is easiest to start by looking at the parallel overhead

Analysis

Isoefficiency

Computing the isoefficiency can be a bit fiddly, but often it is easiest to start by looking at the parallel overhead

We have efficiency $E = T_s/pT_p$ and overhead $T_o = pT_p - T_s$. Combining these:

$$E = \frac{T_s}{p\left(\frac{T_o+T_s}{p}\right)} = \frac{T_s}{T_o + T_s} = \frac{1}{1 + T_o/T_s}$$

Analysis
Isoefficiency

Computing the isoefficiency can be a bit fiddly, but often it is easiest to start by looking at the parallel overhead

We have efficiency $E = T_s/pT_p$ and overhead $T_o = pT_p - T_s$. Combining these:

$$E = \frac{T_s}{p\left(\frac{T_o+T_s}{p}\right)} = \frac{T_s}{T_o + T_s} = \frac{1}{1 + T_o/T_s}$$

So to keep E constant, we need to keep T_o/T_s constant

Analysis

Isoefficiency

So we must have

$$T_s = cT_o$$

for some constant c

Analysis

Isoefficiency

So we must have

$$T_s = cT_o$$

for some constant c

As both T_s and T_o depend on n and p, this equation generally gives us enough to solve for n in terms of p

Analysis

Isoefficiency

Example. The p-stage pipeline had efficiency

$$E = n/(p + n - 1)$$

on a problem of size n

Analysis

Isoefficiency

Example. The p-stage pipeline had efficiency

$$E = n/(p + n - 1)$$

on a problem of size n

The overhead

$$T_o = pT_p - T_s = p(p + n - 1) - np = p^2 - p$$

independent of n

Analysis

Isoefficiency

Example. The *p*-stage pipeline had efficiency

$$E = n/(p + n - 1)$$

on a problem of size n

The overhead

$$T_o = pT_p - T_s = p(p + n - 1) - np = p^2 - p$$

independent of n

This fixed overhead again tells us it is a good idea to keep the pipeline full!

Analysis

Isoefficiency

We want $T_s = cT_o$ which is

$$np = c(p^2 - p)$$

Analysis

Isoefficiency

We want $T_s = cT_o$ which is

$$np = c(p^2 - p)$$

We solve for n

$$n = c(p - 1)$$

Analysis

Isoefficiency

We want $T_s = cT_o$ which is

$$np = c(p^2 - p)$$

We solve for n

$$n = c(p - 1)$$

Thus the isoefficiency is

$$n = O(p)$$

Analysis
Isoefficiency

This is linear in p: if we double p we need only double n to maintain efficiency

Analysis

Isoefficiency

This is linear in p: if we double p we need only double n to maintain efficiency

So this tells us pipelines are very scalable

Analysis

Measures Conclusion

There are many ways we can measure if our parallel program is performing well, or poorly

Analysis

Measures Conclusion

There are many ways we can measure if our parallel program is performing well, or poorly

But we do need to be careful that we are making meaningful comparisons of parallel and sequential algorithms

Analysis

Measures Conclusion

There are many ways we can measure if our parallel program is performing well, or poorly

But we do need to be careful that we are making meaningful comparisons of parallel and sequential algorithms

Exercise Compute these measures for summing n numbers using p processors

Analysis

Bandwidth and Latency

While we are thinking about measurement of parallel systems we need to make a quick comment about *bandwidth* and *latency* as they play an important role in the way we regard communications overhead

Analysis

Bandwidth and Latency

While we are thinking about measurement of parallel systems we need to make a quick comment about *bandwidth* and *latency* as they play an important role in the way we regard communications overhead

Bandwidth is the number of bytes per second transmitted over some medium

Analysis

Bandwidth and Latency

While we are thinking about measurement of parallel systems we need to make a quick comment about *bandwidth* and *latency* as they play an important role in the way we regard communications overhead

Bandwidth is the number of bytes per second transmitted over some medium

Latency is how long we have to wait for the data to arrive

Analysis

Bandwidth and Latency

Bandwidth is often mentioned in descriptions of things as it is easy to visualise (a rate of flow)

Analysis

Bandwidth and Latency

Bandwidth is often mentioned in descriptions of things as it is easy to visualise (a rate of flow)

However, latency is often just as important in parallel systems

Analysis

Bandwidth and Latency

Bandwidth is often mentioned in descriptions of things as it is easy to visualise (a rate of flow)

However, latency is often just as important in parallel systems

Bandwidths these days are pretty high: Mb and Gb rates are common

Analysis

Bandwidth and Latency

Bandwidth is often mentioned in descriptions of things as it is easy to visualise (a rate of flow)

However, latency is often just as important in parallel systems

Bandwidths these days are pretty high: Mb and Gb rates are common

Latencies of milliseconds may *seem* small, but relatively speaking they are the big problem

Analysis

Bandwidth and Latency

Example A memory bus (DDR5) might have 400Gb/sec bandwidth and latency 100ns.

Example A memory bus (DDR5) might have 400Gb/sec bandwidth and latency 100ns.

Fast, but processors are faster! Data might arrive at a prodigious rate when it does arrive, but a processor could do a lot of work while it was waiting for the first byte to arrive

Analysis

Bandwidth and Latency

Example A memory bus (DDR5) might have 400Gb/sec bandwidth and latency 100ns.

Fast, but processors are faster! Data might arrive at a prodigious rate when it does arrive, but a processor could do a lot of work while it was waiting for the first byte to arrive

This is why processors have lots of complex and clever caching to avoid going off-chip

Example A local network (10Gb Ethernet) might have bandwidth 10Gb/sec and latency 100μs

Analysis

Bandwidth and Latency

Example A local network (10Gb Ethernet) might have bandwidth 10Gb/sec and latency 100μs

This is how nodes in a cluster are often connected

Analysis

Bandwidth and Latency

Example A local network (10Gb Ethernet) might have bandwidth 10Gb/sec and latency 100μs

This is how nodes in a cluster are often connected

Again we are in the range of hundreds of thousands of instructions while waiting

Analysis

Bandwidth and Latency

Example A local network (10Gb Ethernet) might have bandwidth 10Gb/sec and latency 100μs

This is how nodes in a cluster are often connected

Again we are in the range of hundreds of thousands of instructions while waiting

And this does not include the CPU overhead of going through the Operating System to send and receive the packets from the network

The latency affects coding strongly: it may be worthwhile doing duplicate computations if that is faster than fetching a value

Analysis

Bandwidth and Latency

The latency affects coding strongly: it may be worthwhile doing duplicate computations if that is faster than fetching a value

In large parallel systems compute power is cheap and plentiful, but communications are slow and expensive

The latency affects coding strongly: it may be worthwhile doing duplicate computations if that is faster than fetching a value

In large parallel systems compute power is cheap and plentiful, but communications are slow and expensive

This is why when we implement parallel code we really need to concentrate on the communications more than the computations

Analysis

Bandwidth and Latency

It is quite easy to increase bandwidth

Analysis

Bandwidth and Latency

It is quite easy to increase bandwidth

Doubling the width of a bus will double the bandwidth, but do nothing to the latency

Analysis

Bandwidth and Latency

It is quite easy to increase bandwidth

Doubling the width of a bus will double the bandwidth, but do nothing to the latency

We might get a huge bandwidth by strapping a USB stick to a pigeon: the latency would not be so good, though!

Analysis

Bandwidth and Latency

It is quite easy to increase bandwidth

Doubling the width of a bus will double the bandwidth, but do nothing to the latency

We might get a huge bandwidth by strapping a USB stick to a pigeon: the latency would not be so good, though!

For a long time *sneakernet* was the best way to transmit large volumes of data

Analysis

Bandwidth and Latency

It is quite easy to increase bandwidth

Doubling the width of a bus will double the bandwidth, but do nothing to the latency

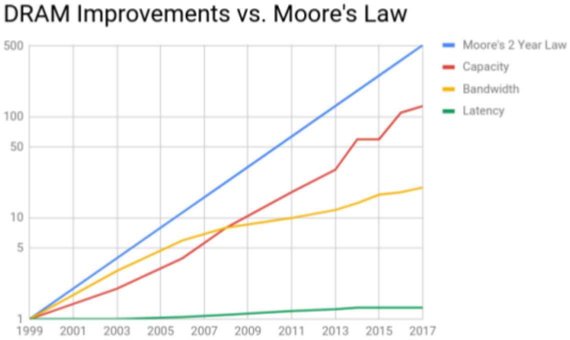
We might get a huge bandwidth by strapping a USB stick to a pigeon: the latency would not be so good, though!

For a long time *sneakernet* was the best way to transmit large volumes of data

Exercise Read about how data was transmitted to generate the recent (2019) image of a black hole

Analysis

Note: Moore says sizes of RAM are increasing, but latencies are far behind



Sizes of RAM over time

Graph from Kevin K. Chang, PhD., CMU 2017

Analysis

Bandwidth and Latency

Latency is often limited by Physics: the speed of light is a big factor on latency these days

Analysis

Bandwidth and Latency

Latency is often limited by Physics: the speed of light is a big factor on latency these days

Thus, like Amdahl, latency is another natural limit on parallel computation

Analysis

Bandwidth and Latency

Latency is often limited by Physics: the speed of light is a big factor on latency these days

Thus, like Amdahl, latency is another natural limit on parallel computation

Particularly on distributed architectures

Shared Memory Systems

We now move on to look at shared memory and distributed memory systems in more detail, in particular the issues that arise in software and programming

Shared Memory Systems

We now move on to look at shared memory and distributed memory systems in more detail, in particular the issues that arise in software and programming

We start with shared memory MIMD as people think it seems more similar to SISD than distributed memory is, and so is "easier"

Shared Memory Systems

We now move on to look at shared memory and distributed memory systems in more detail, in particular the issues that arise in software and programming

We start with shared memory MIMD as people think it seems more similar to SISD than distributed memory is, and so is "easier"

We will look at simple programs that have multiple *threads of control*, i.e., parts of the process are running simultaneously on separate processors

Shared Memory Systems

Note: a single program might use several processes, and each process might contain several threads

Shared Memory Systems

Note: a single program might use several processes, and each process might contain several threads

Separate processes have separate (virtual) memory address spaces (my memory location `42` is different from your memory location `42`)

Shared Memory Systems

Note: a single program might use several processes, and each process might contain several threads

Separate processes have separate (virtual) memory address spaces (my memory location `42` is different from your memory location `42`)

Multiple threads in the same process (generally) share the same (virtual) address space (my memory location `42` is the same as your memory location `42`)

Shared Memory Systems

Note: a single program might use several processes, and each process might contain several threads

Separate processes have separate (virtual) memory address spaces (my memory location 42 is different from your memory location 42)

Multiple threads in the same process (generally) share the same (virtual) address space (my memory location 42 is the same as your memory location 42)

Here we consider the shared part, i.e., threads within a process