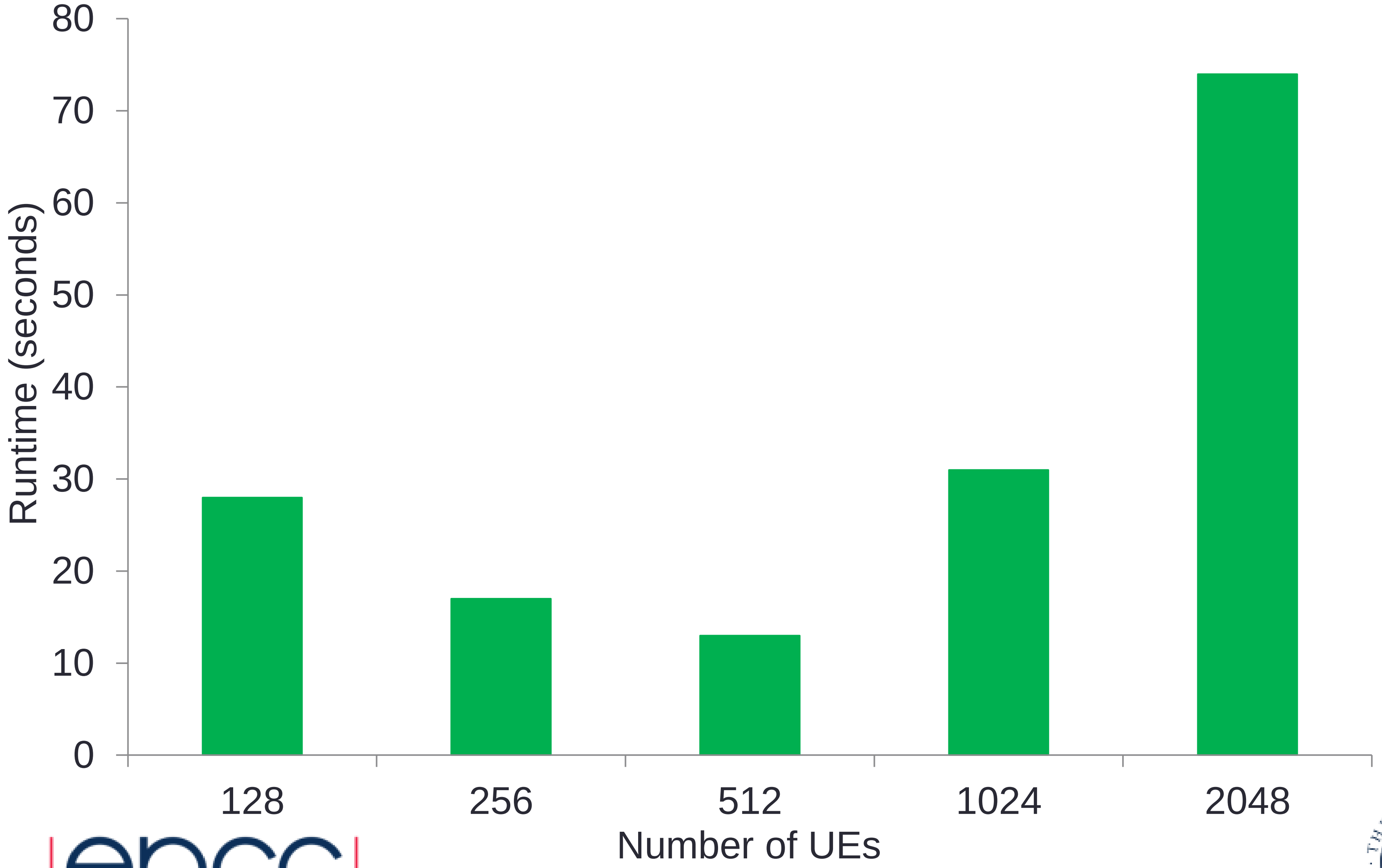
Parallel design patterns ARCHER course

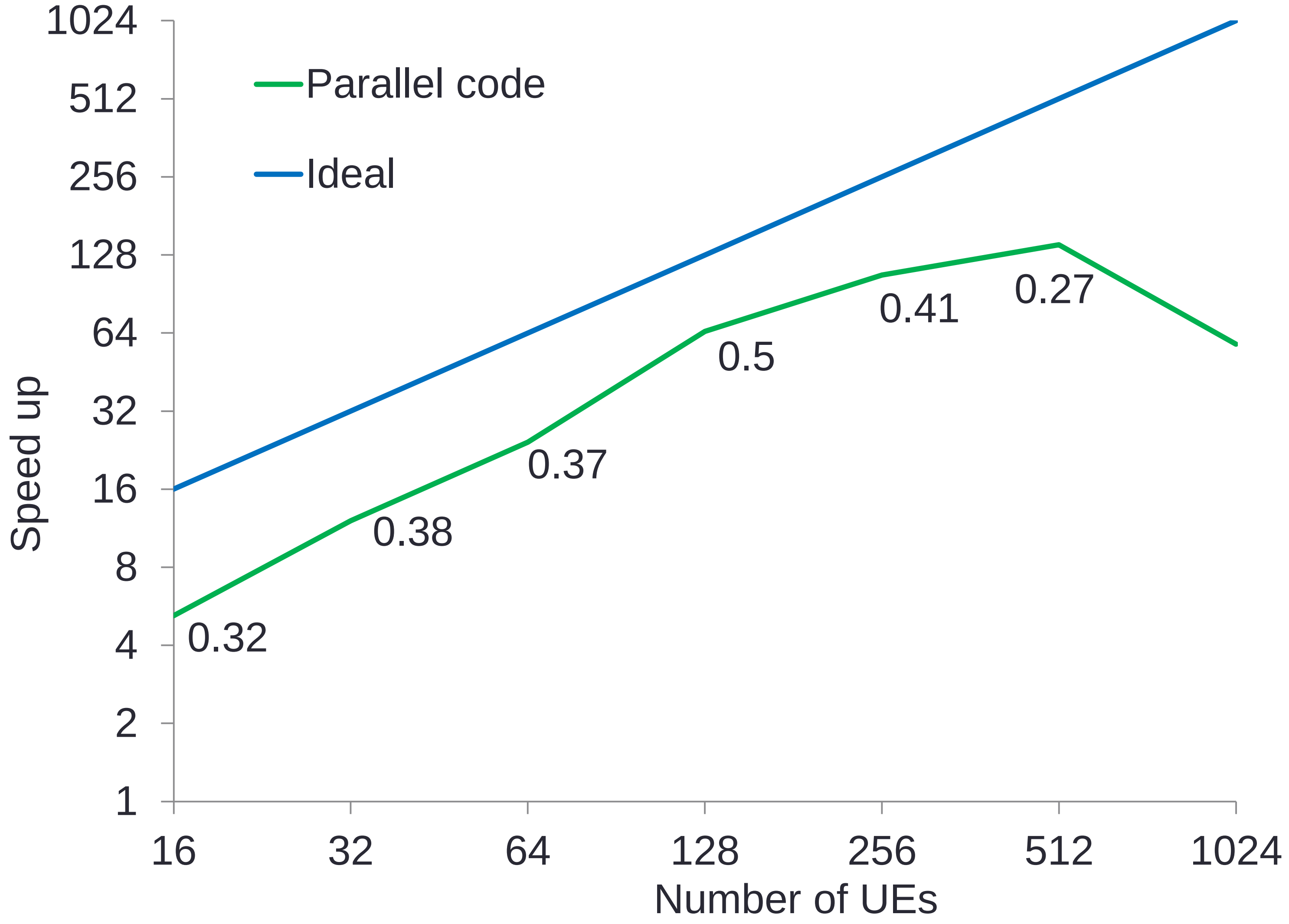
Comparing parallel algorithms

Reusing this material

What's the problem?



Speed up and parallel efficiency



$$S(n) = \frac{T1}{Tn} \qquad E(n) = \frac{S(n)}{P}$$

So what’s the issue?

- Empirical studies are fine, but it requires an existing parallel code to benchmark and study

- Be careful what you claim, how many data points is enough to make certain claims?
- How can we predict performance and scalability at higher core counts or with certain modifications made to the algorithm?
- How much insight can we really get (i.e. where is my bottleneck?)

So how can we talk sensibly about parallel algorithms (i.e. compare them) without explicit measurement?

Amdahl's law

- A fraction, α, is completely serial

- Parallel runtime $T(N,P) = \alpha T(N,1) + \frac{(1-\alpha)T(N,1)}{P}$
  - Assuming parallel part is 100% efficient

- Parallel speedup $S(N,P) = \frac{T(N,1)}{T(N,P)} = \frac{P}{\alpha P + (1-\alpha)}$

- We are fundamentally limited by the serial fraction
  - For $\alpha = 0$, $S = P$ as expected (i.e. *efficiency* = 100%)
  - Otherwise, speedup limited by 1/ α for any *P*
    - *For* α = 0.1; 1/0.1 = 10 therefore 10 times maximum speed up
    - *For* α = 0.1; S(N, 16) = 6.4, S(N, 1024) = 9.9

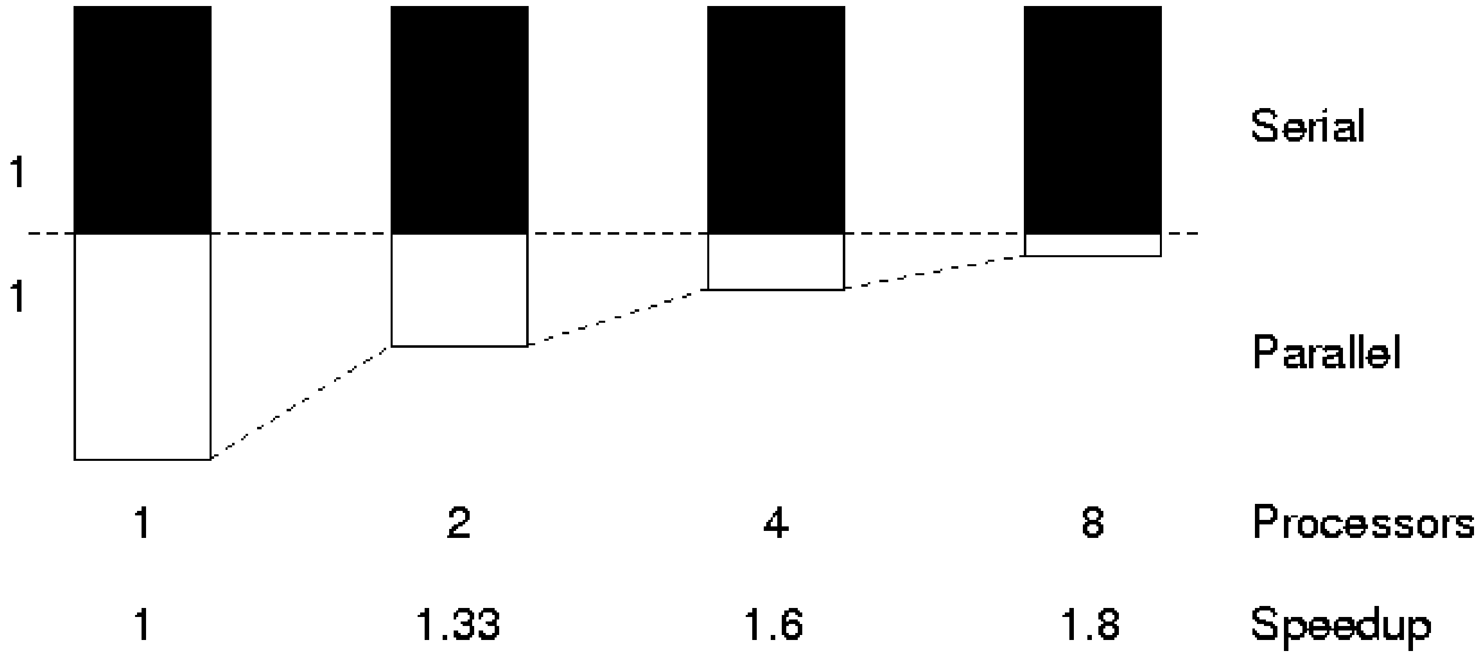
epcc

THE UNIVERSITY OF EDINBURGH

The serial section of code

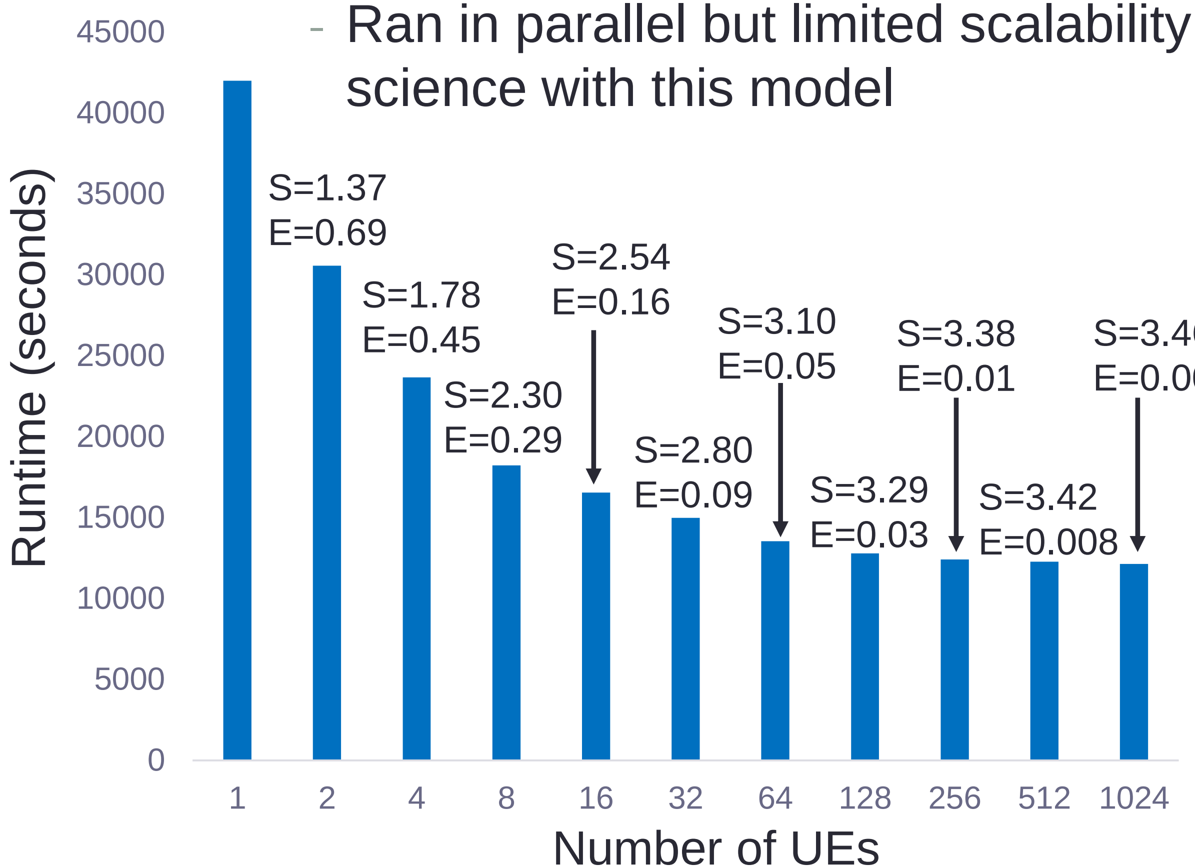
"The performance improvement to be gained by parallelisation is limited by the proportion of the code which is serial"

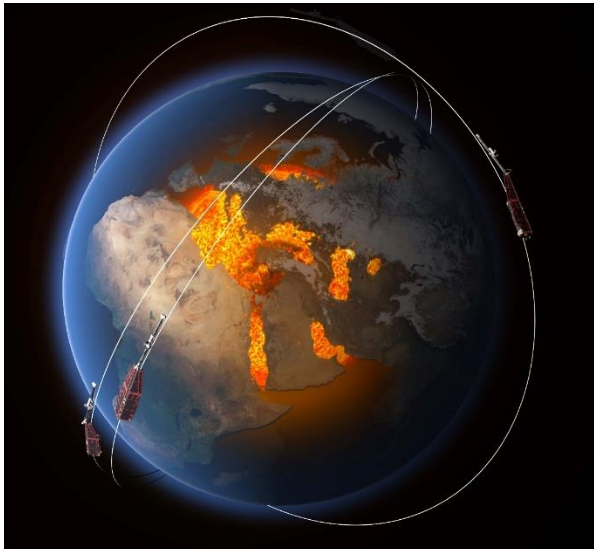
Gene Amdahl, 1967



eCSE project on BGS spline model

- Model for predicting the geomagnetic field lines of the earth
  - Ran in parallel but limited scalability and wanted to do more science with this model



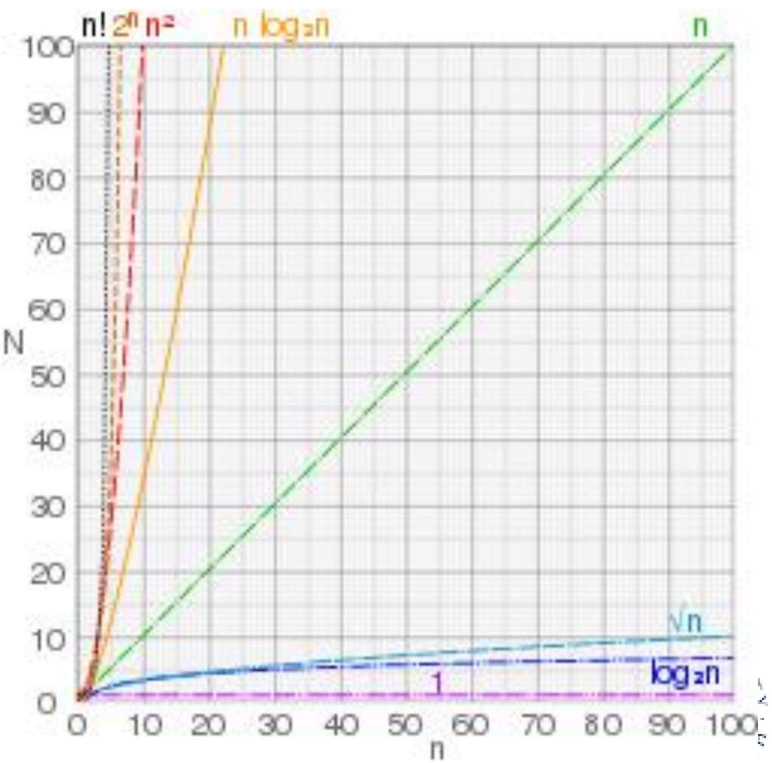


epcc

THE UNIVERSITY OF EDINBURGH

In the serial world

- Algorithm time complexity - The rough growth rate of resources (specifically runtime) with respect to the input size
- Estimated by counting the number of elementary operations the algorithm is required to perform
  - i.e. $8n + 12n^2$ where n is the number of input elements
  - The worst case time complexity is most commonly used and here it would be $O(n^2)$
- Provides a way to evaluate and compare sequential algorithms



Algorithm time complexity examples

```
for (i=0;i<50;i++) {
  result=result+a[0]
}
```

*50 * (2 + 3 + 1) = O(1)*

```
for (i=0;i<n;i++) {
  result=result+a[i]
}
```

*n * (2 + 3 + 1) = O(n)*

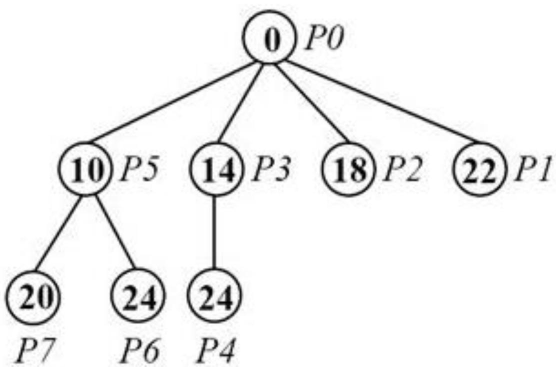
```
for (i=0;i<n;i++) {
  for (j=0;j<n;j++) {
    result=result+a[i]
  }
}
```

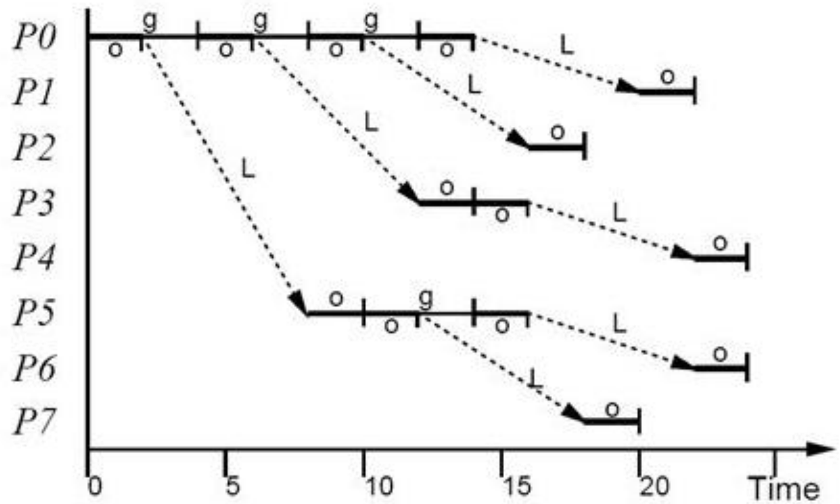
$n * (2 + n * (2 + 3 + 1)) = O(n*n)=O(n^2)$

- *Concerned with how the runtime grows as a function of the input size (n)*

But much more complex in the parallel world!

- A number of different ways of modelling this
- Log P is one common approach in the literature
  - L is the latency of the communication medium (cycles)
  - o is the overhead of sending and receiving messages (cycles)
  - g is the gap required between messages due to bandwidth limitations (cycles)
  - P is the number of UEs

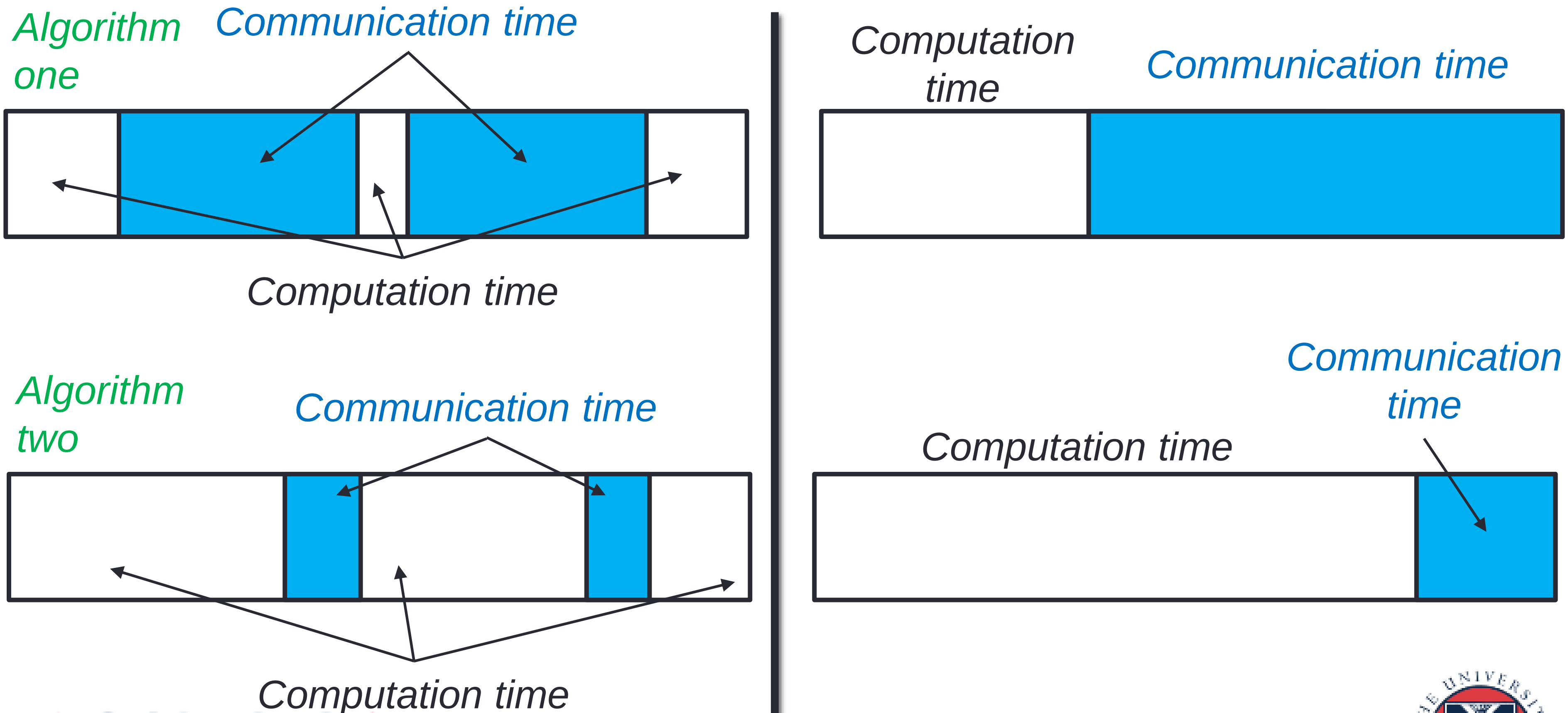




Example taken from http://slideplayer.com/slide/8123828/ where P=8, L=6, g=4, o=2

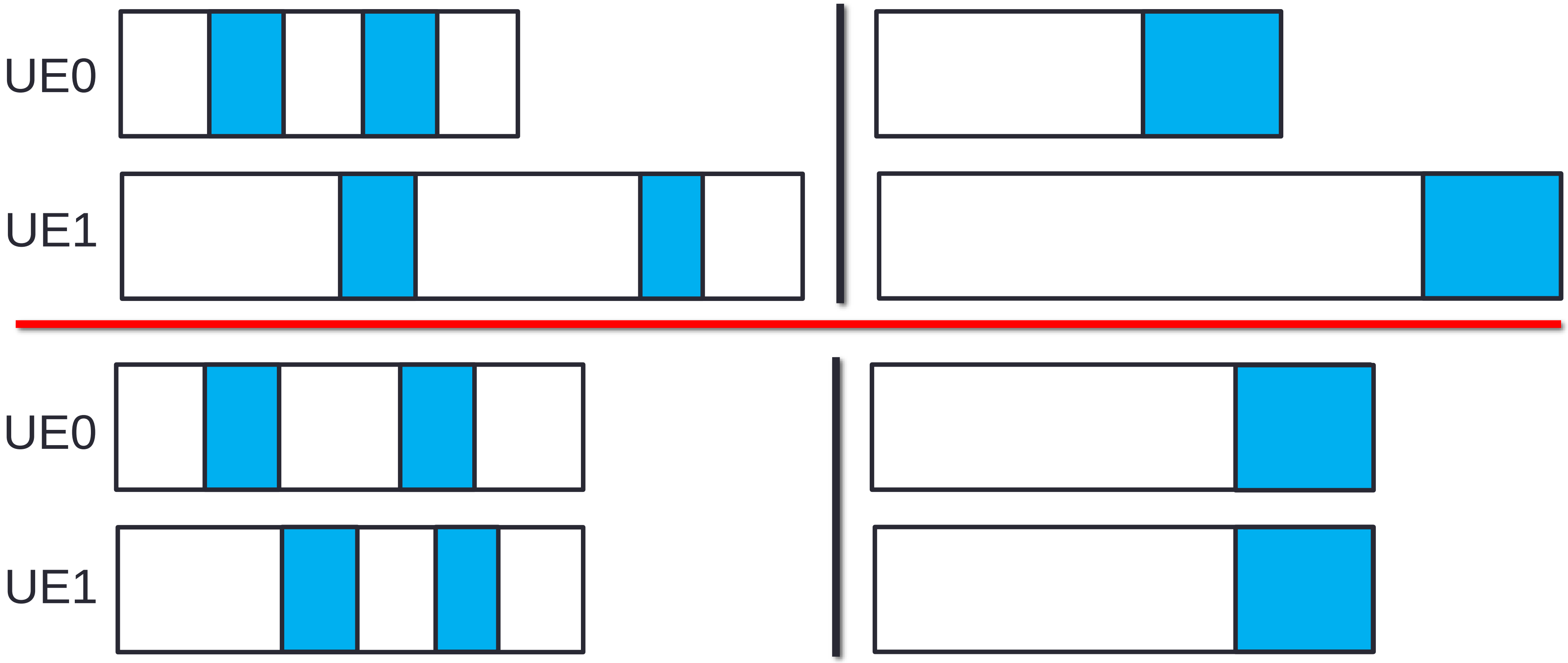
What are we looking to optimise

1. Communication to computation ratio



What are we looking to optimise

2. Load balance between processes

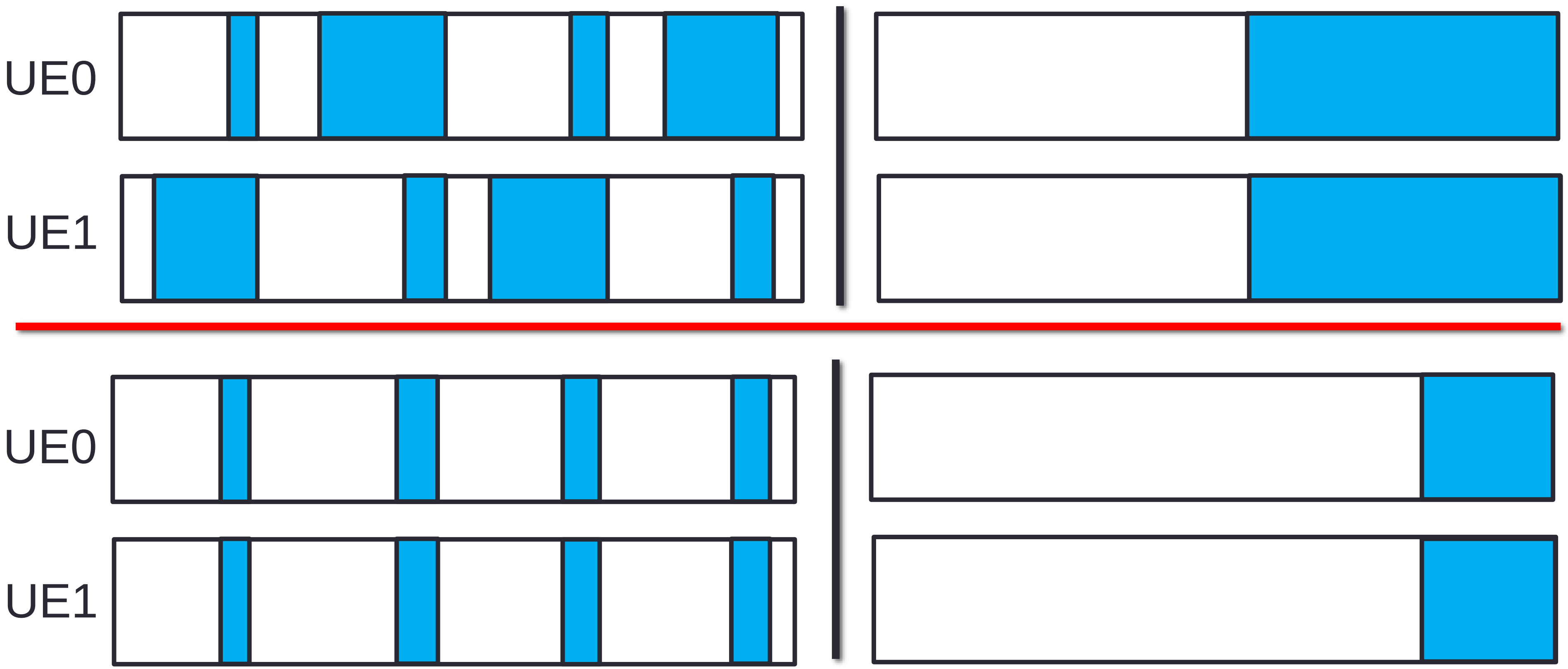


Load Imbalance Factor (LIF):
maximum load / average load

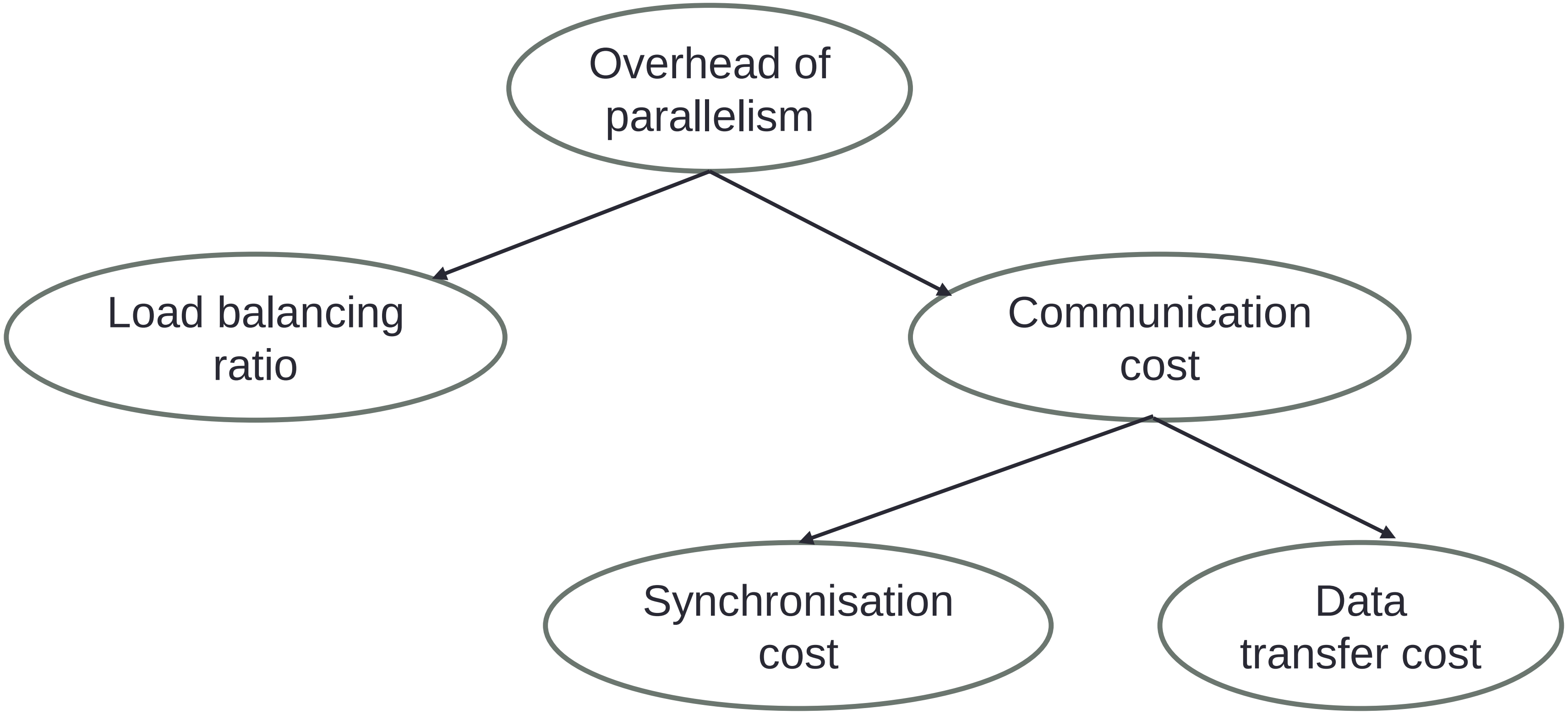
Where 1 is ideal

What are we looking to optimise

3. Synchronisation costs



Parallelism overhead



Can be obvious from code

```
if (rank == 0) {
  for (i=0;i<1000;i++) {
      a[i]=.........
  }
  send a to rank 1
  b=recv from rank 1
}
```

```
if (rank == 1) {
  b=recv from rank 0
  for (i=0;i<100;i++) {
      a[i]=.........
  }
  send a to rank 0
}
```

A slight improvement….

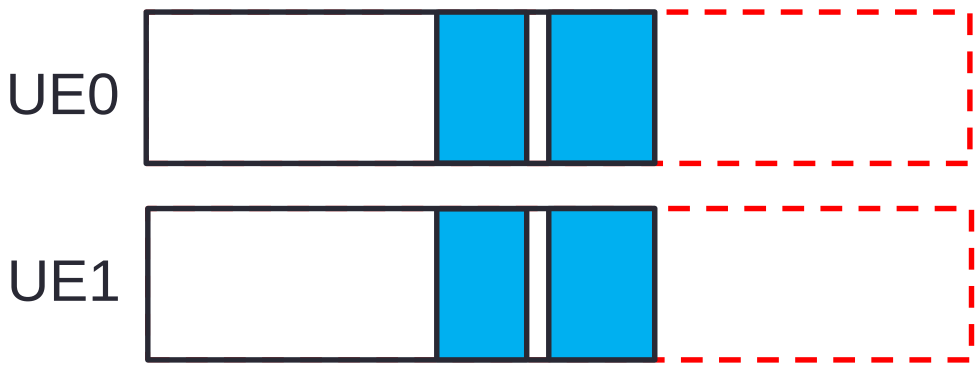
```
if (rank == 0) {
  for (i=0;i<1000;i++) {
      a[i]=………
  }
  send a to rank 1
  b=recv from rank 1
}
```

```
if (rank == 1) {
  for (i=0;i<100;i++) {
      a[i]=………
  }
  b=recv from rank 0
  send a to rank 0
}
```

More of an improvement….

```
if (rank == 0) {
  for (i=0;i<550;i++) {
      a[i]=………
  }
  send a to rank 1
  b=recv from rank 1
}
```

```
if (rank == 1) {
  for (i=0;i<550;i++) {
      a[i]=………
  }
  b=recv from rank 0
  send a to rank 0
}
```

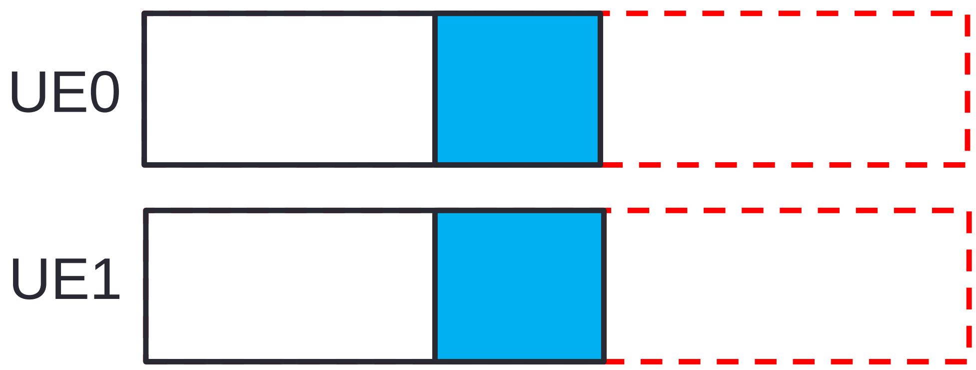


|epcc|

Potentially even better

```
if (rank == 0) {
  b=nonblocking recv from rank 1
  for (i=0;i<550;i++) {
      a[i]=………
  }
  nonblocking send a to rank 1
  wait on all comms
}
```

```
if (rank == 1) {
  b=nonblocking recv from rank 0
  for (i=0;i<550;i++) {
      a[i]=………
  }
  nonblocking send a to rank 0
  wait on all comms
}
```



But we don't get this for free!

- Efficiency
  - Speed, memory, storage
- Scalability
  - Large machines, large problems
- Simplicity of the code
  - Development, debugging, verification, modification, maintenance
- Portability
  - Software nearly always outlives its original target platform

- There is rarely *one right answer* and a good design often boils down to a number of *tradeoffs*
- Parallel optimisations can increase sequential time complexity