Performance Scaling

How is my parallel code performing and scaling?

Performance metrics

- Measure the execution time *T*
  - how do we quantify performance improvements?

- Speed up
  - typically $S(N,P) < P$

$$S(N,P) = \frac{T(N,1)}{T(N,P)}$$

- Parallel efficiency
  - typically $E(N,P) < 1$

$$E(N,P) = \frac{S(N,P)}{P} = \frac{T(N,1)}{PT(N,P)}$$

- Serial efficiency
  - typically $E(N) <= 1$

$$E(N) = \frac{T_{best}(N)}{T(N,1)}$$

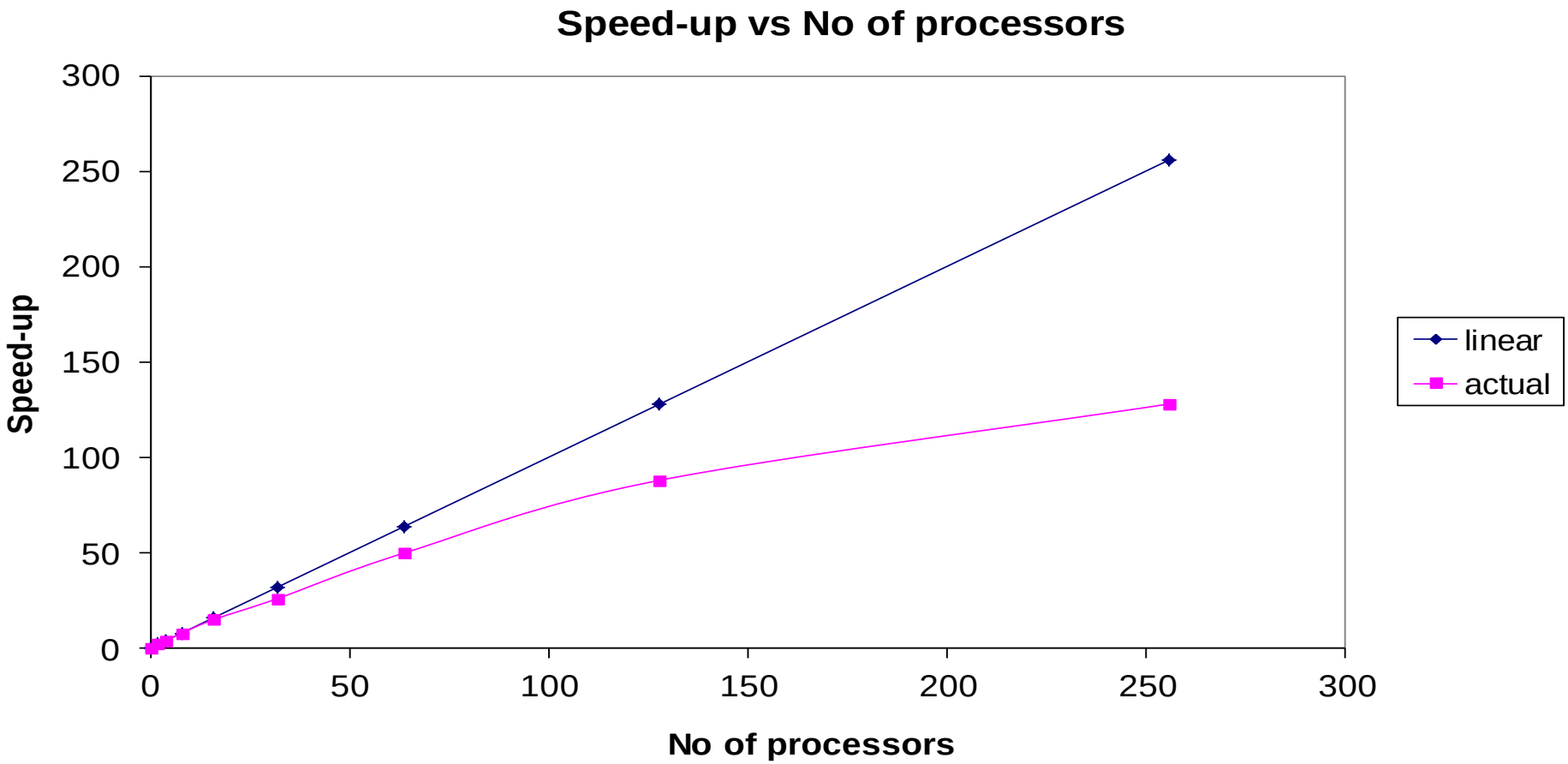
Where N is the size of the problem and P the number of processors

Scaling

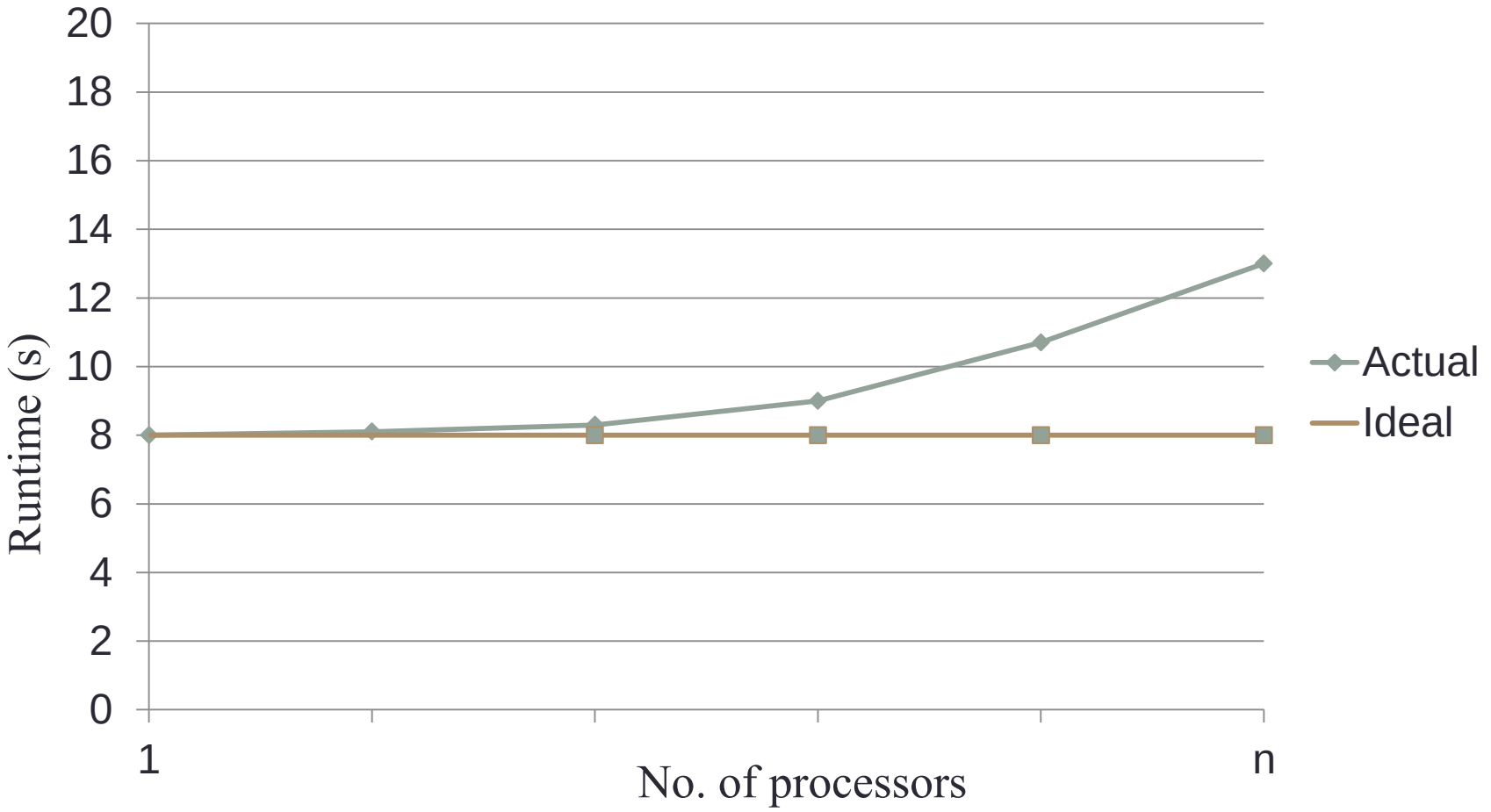
- *Scaling* is how the performance of a parallel application changes as the number of processors is increased

- There are two different types of scaling:
  - *Strong Scaling* – total problem size stays the same as the number of processors increases
  - *Weak Scaling* – the problem size increases at the same rate as the number of processors, keeping the amount of work per processor the same
- Strong scaling is generally more useful and more difficult to achieve than weak scaling

Strong scaling



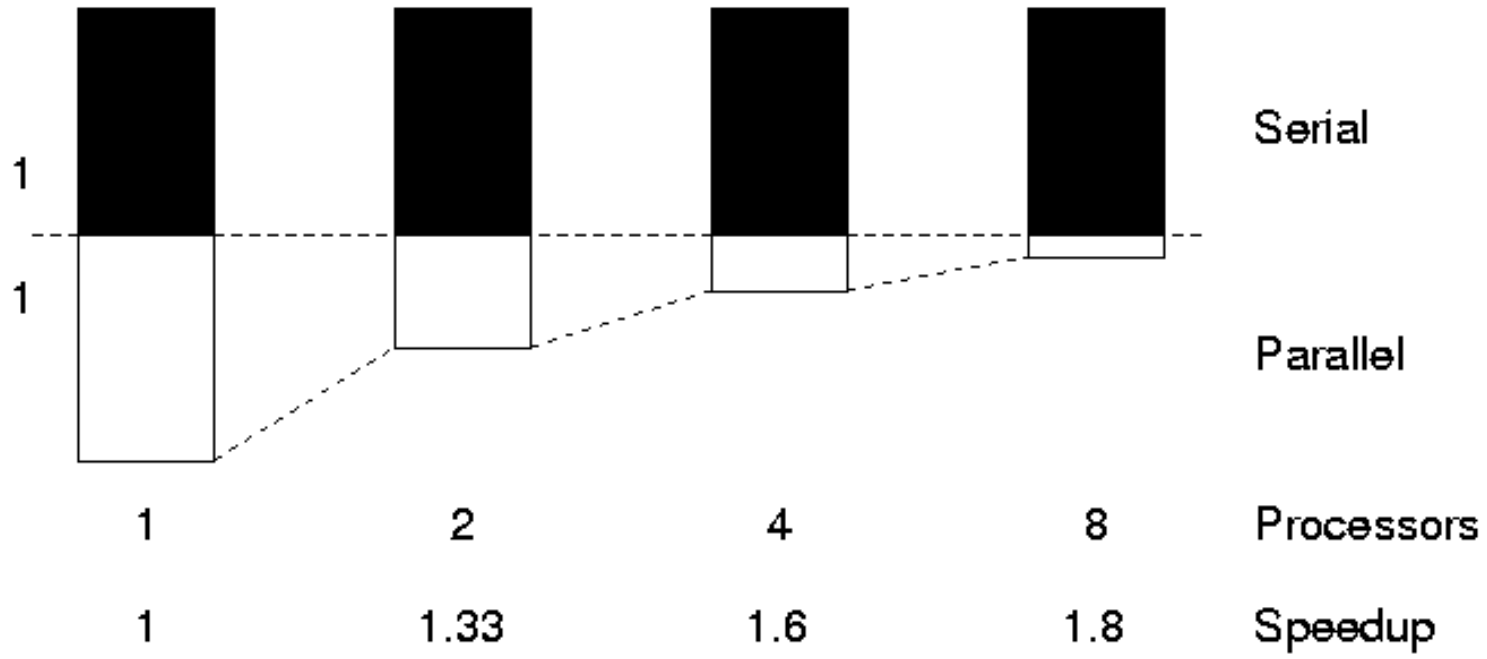
Weak scaling



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



archer

epcc

THE UNIVERSITY OF EDINBURGH

Amdahl's law

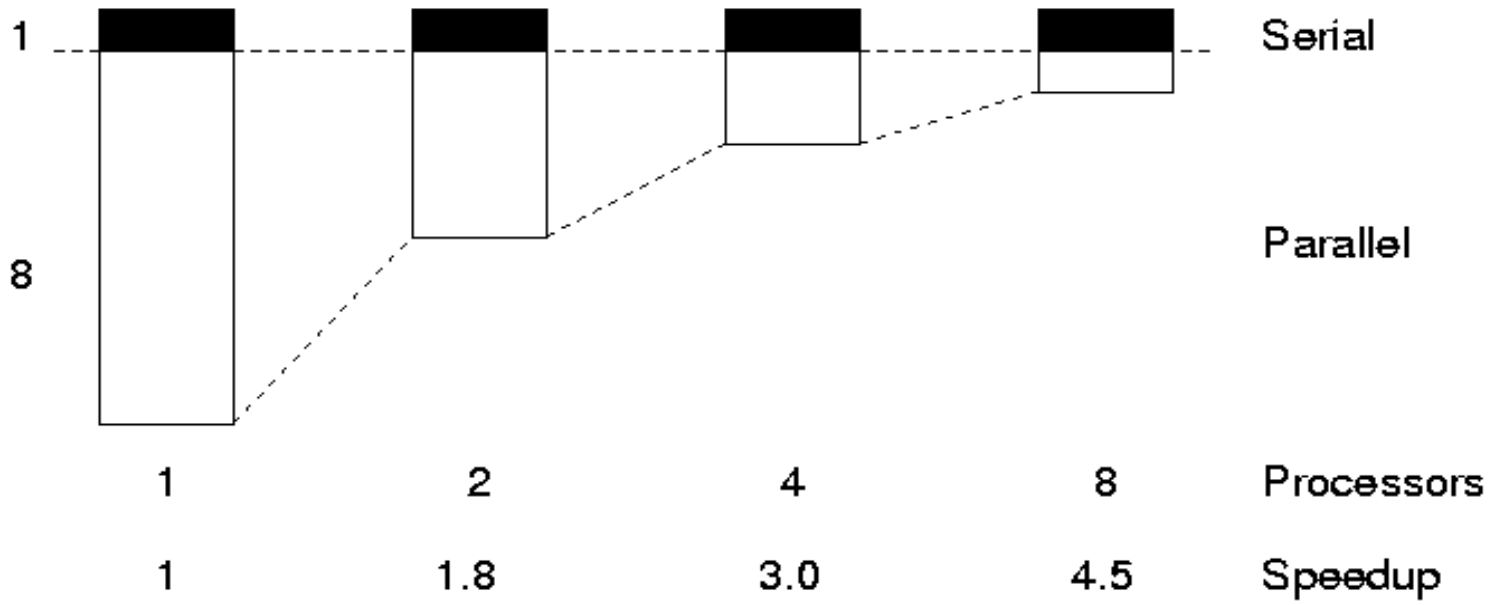
- A typical program has two categories of components
  - Inherently sequential sections: can't be run in parallel
  - Potentially parallel sections
- A fraction, α, is completely serial
- Assuming parallel part is 100% efficient:

- Parallel runtime $$T(N,P) = aT(N,1) + \frac{(1-a)T(N,1)}{P}$$
- Parallel speedup $$S(N,P) = \frac{T(N,1)}{T(N,P)} = \frac{P}{aP + (1-a)}$$

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

Gustafson's Law

- We need larger problems for larger numbers of CPUs



- Whilst we are still limited by the serial fraction, it becomes less important

Utilising Large Parallel Machines

- Assume parallel part is proportional to N
  - serial part is independent of N
  - time

$$T(N,P) = T_{serial}(N,P) + T_{parallel}(N,P)$$

$$= aT(1,1) + \frac{(1-a)\,N\,T(1,1)}{P}$$

$$T(N,1) = aT(1,1) + (1-a)\,N\,\mathrm{T}(1,1)$$

  - speedup

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

- Scale problem size with CPUs, i.e. set $N = P$ (weak scaling)
  - speedup $S(P,P) = \alpha + (1-\alpha)\,P$
  - efficiency $E(P,P) = \alpha/P + (1-\alpha)$

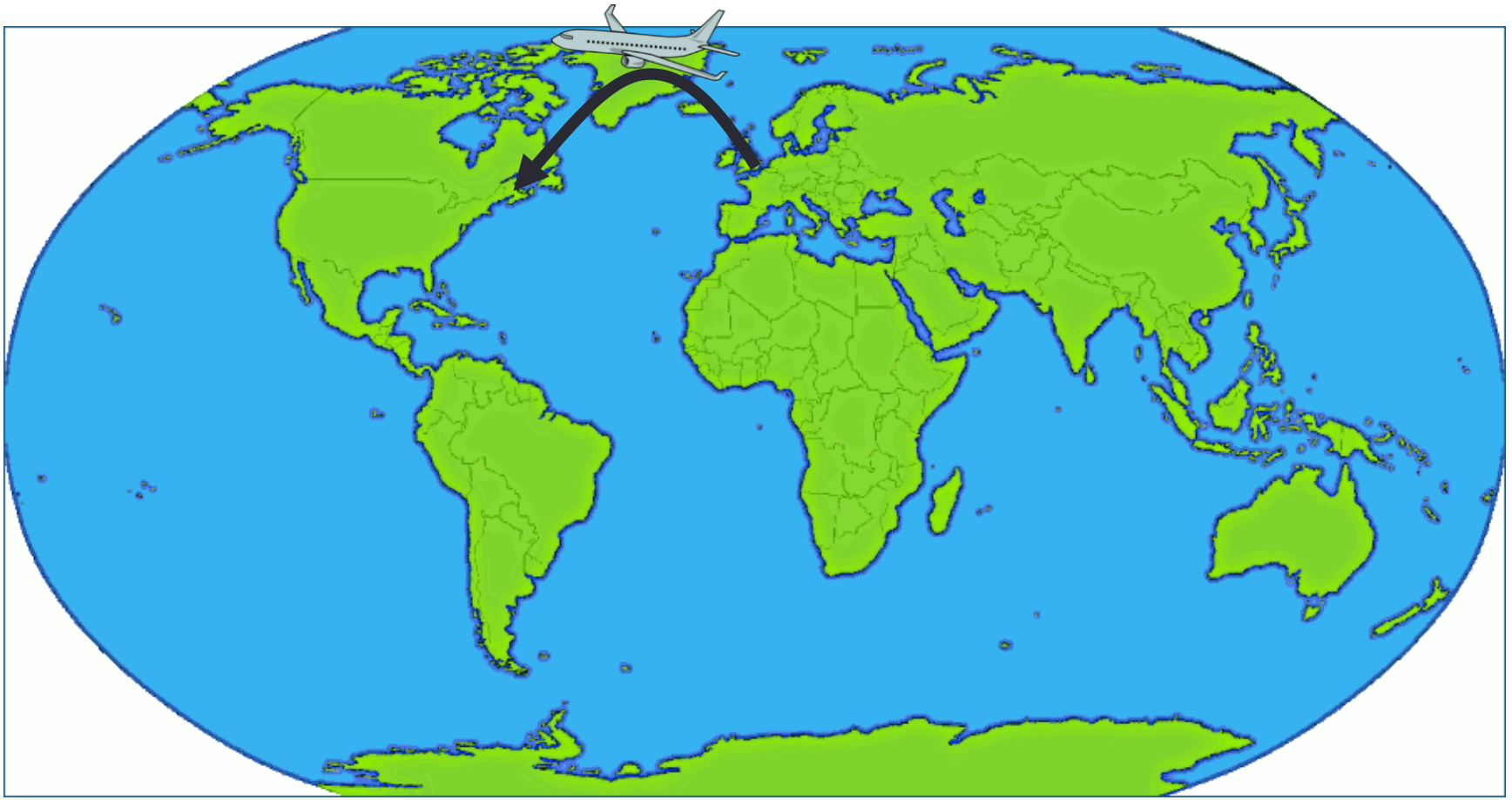
Gustafson's Law

- If you increase the amount of work done by each parallel task then the serial component will not dominate
  - Increase the problem size to maintain scaling
  - Can do this by adding extra complexity or increasing the overall problem size

Number of processors	Strong scaling (Amdahl's law)	Weak scaling (Gustafson's law)
16	6.4	14.5
1024	9.9	921.7

Due to the scaling of N, the serial fraction effectively becomes α/P

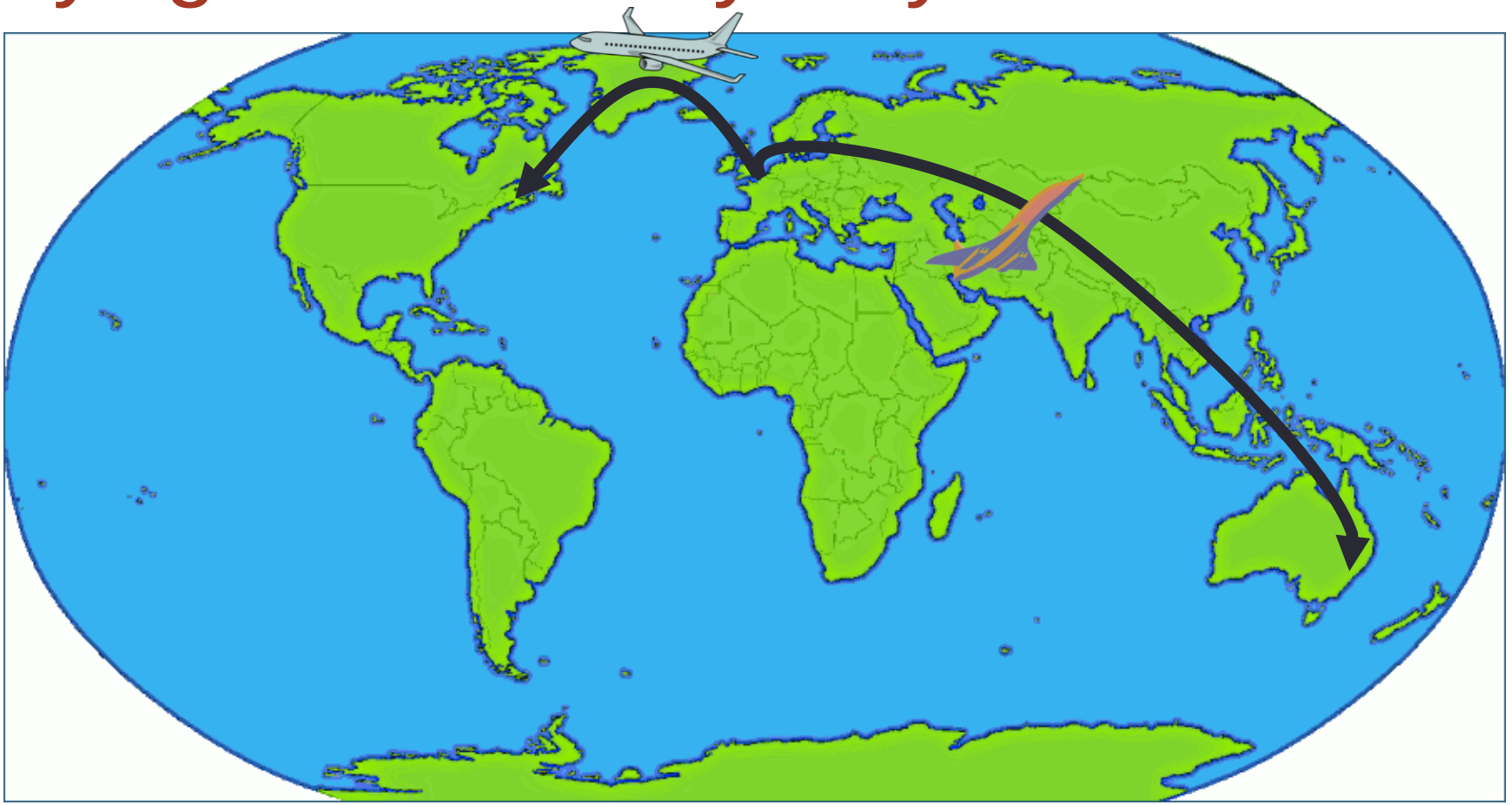
Analogy: Flying London to New York



Buckingham Palace to Empire State

- By Jumbo Jet
  - distance: 5600 km; speed: 700 kph
  - time: 8 hours ?
- No!
  - 1 hour by tube to Heathrow + 1 hour for check in etc.
  - 1 hour immigration + 1 hour taxi downtown
  - fixed overhead of 4 hours; total journey time: 4 + 8 = 12 hours
- Triple the flight speed with Concorde to 2100 kph
  - total journey time = 4 hours + 2 hours 40 mins = 6.7 hours
  - speedup of 1.8 not 3.0
- Amdahl's law! α = 4/12 = 0.33; max speedup = 3 (i.e. 4 hours)

Flying London to Sydney



archer

epcc

THE UNIVERSITY OF EDINBURGH

Buckingham Palace to Sydney Opera

- By Jumbo Jet
  - distance: 16800 km; speed: 700 kph; flight time; 24 hours
  - serial overhead **stays the same:** total time: 4 + 24 = 28 hours
- Triple the flight speed
  - total time = 4 hours + 8 hours = 12 hours
  - speedup = 2.3 (as opposed to 1.8 for New York)
- Gustafson's law!
  - bigger problems scale better
  - increase **both** distance (i.e. N) **and** max speed (i.e. P) by three
  - maintain same balance: 4 "serial" + 8 "parallel"

Load Imbalance

- These laws all assumed all processors are equally busy
  - what happens if some run out of work?
- Specific case
  - four people pack boxes with cans of soup: 1 minute per box

Person	Anna	Paul	David	Helen	Total
# boxes	6	1	3	2	12

  - takes 6 minutes as everyone is waiting for Anna to finish!
  - if we gave everyone same number of boxes, would take 3 minutes
- Scalability isn't everything
  - make the best use of the processors at hand before increasing the number of processors

Quantifying Load Imbalance

- Define Load Imbalance Factor

$$LIF = maximum\ load\ /\ average\ load$$

  - for perfectly balanced problems $LIF = 1.0$, as expected
  - in general, $LIF > 1.0$
  - LIF tells you how much faster your calculation could be with balanced load

- Box packing
  - LIF = 6/3 = 2
  - initial time = 6 minutes
  - best time = LIF / 2 = 3 minutes

Summary

- There are many considerations when parallelising code
- A variety of patterns exist that can provide well known approaches to parallelising a serial problem
  - You will see examples of some of these during the practical sessions
- Scaling is important, as the more a code scales the larger a machine it can take advantage of
  - can consider weak and strong scaling
  - in practice, overheads limit the scalability of real parallel programs
  - Amdahl's law models these in terms of serial and parallel fractions
  - larger problems generally scale better: Gustafson's law
- Load balance is also a crucial factor
- Metrics exist to give you an indication of how well your code performs and scales