

# Parallelism

---

## Overview and Concepts

# Outline

- Decomposition
  - Geometric decomposition
  - Task farm
  - Pipeline
  - Loop parallelism
- General parallelisation considerations
- Parallel code performance metrics and evaluation
- Parallel scaling models

# Why use parallel programming?

It is harder than serial so why bother?

# Why?

- Parallel programming is more difficult than its sequential counterpart
- However we are reaching limitations in uniprocessor design
  - Physical limitations to size and speed of a single chip
  - Developing new processor technology is very expensive
  - Some fundamental limits such as speed of light and size of atoms
- Parallelism is not a silver bullet
  - There are many additional considerations
  - Careful thought is required to take advantage of parallel machines

# Performance

- A key aim is to solve problems faster
  - To improve the time to solution
  - Enable new a new scientific problems to be solved
- To exploit parallel computers, we need to split the program up between different processors
- Ideally, would like program to run  $N$  times faster on  $N$  processors
  - Not all parts of program can be successfully split up
  - Splitting the program up may introduce additional overheads such as communication

# Parallel tasks

- How we split a problem up in parallel is critical
  1. Limit communication (especially the number of messages)
  2. Balance the load so all processors are equally busy
- Tightly coupled problems require lots of interaction between their parallel tasks
- Embarrassingly parallel problems require very little (or no) interaction between their parallel tasks
  - E.g. ensemble simulations
- In reality most problems sit somewhere between two extremes

# Decomposition

How do we split problems up to solve efficiently in parallel?

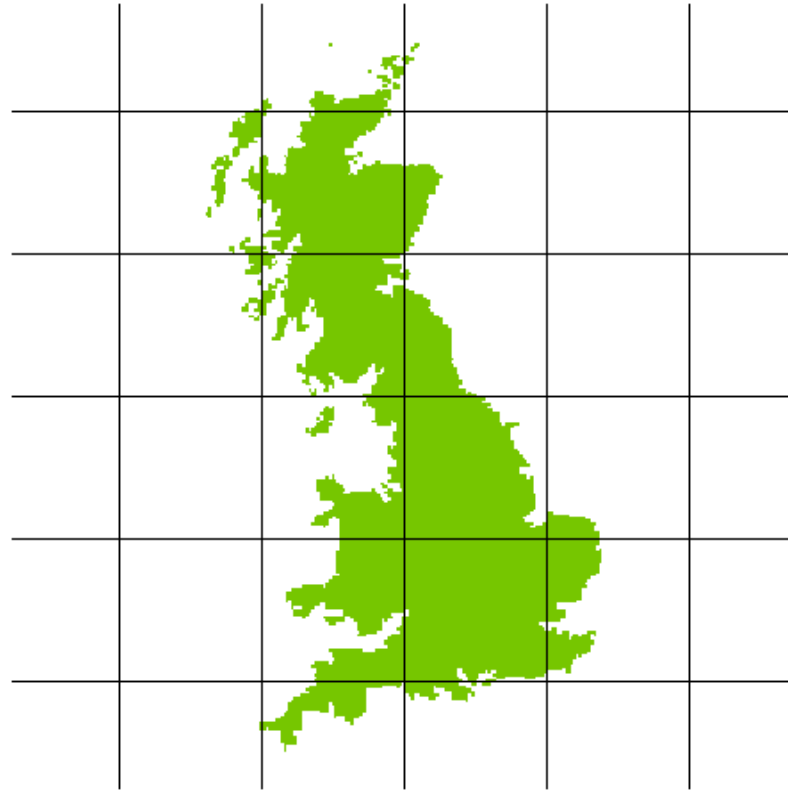
# Decomposition

- One of the most challenging, but also most important, decisions is how to split the problem up
- How you do this depends upon a number of factors
  - The nature of the problem
  - The amount of communication required
  - Support from implementation technologies
- We are going to look at some frequently used decompositions



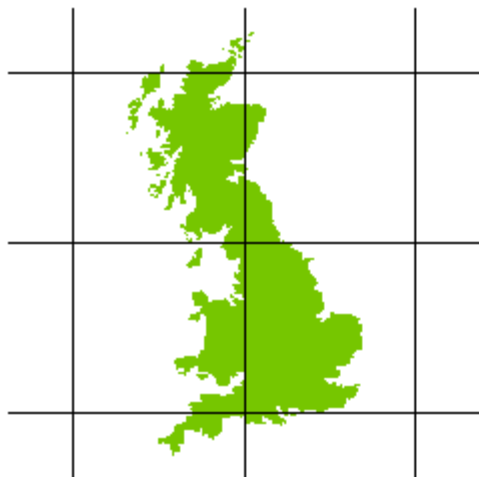
# Geometric decomposition

- Take advantage of the geometric properties of a problem



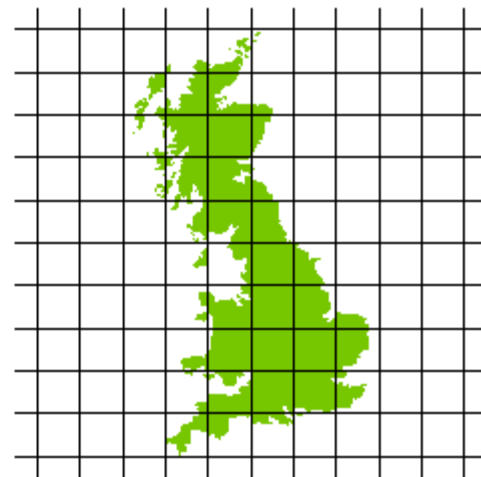
# Geometric decomposition

- Splitting the problem up does have an associated cost
  - Namely communication between processors
  - Need to carefully consider granularity
  - Aim to minimise communication and maximise computation



too large: little parallelism

Granularity  
Size of chunks of work



too small: communications rule

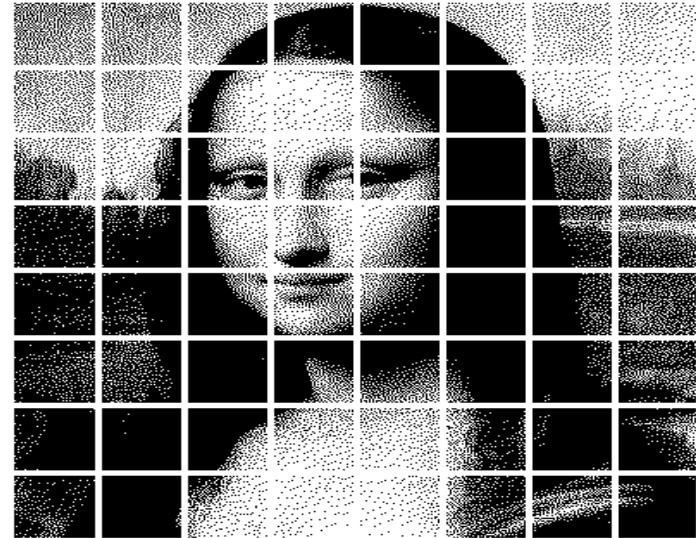
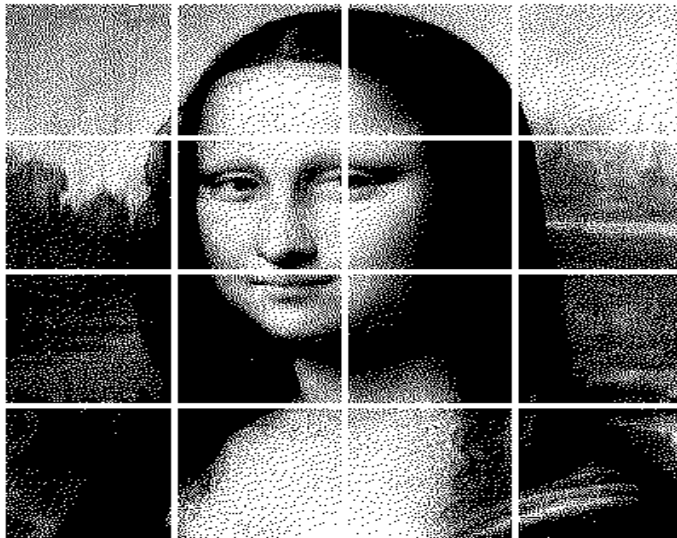
# Halo swapping

- Swap data in bulk at pre-defined intervals
- Often only need information on the boundaries
- Many small messages result in far greater overhead



# Load imbalance

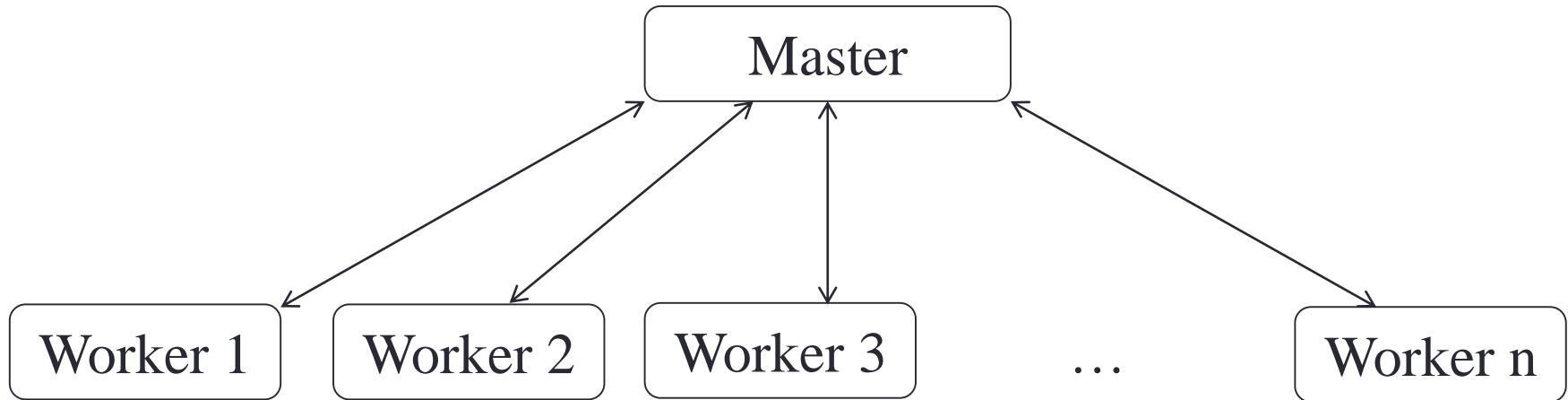
- Execution time determined by slowest processor
  - each processor should have (roughly) the same amount of work, i.e. they should be load balanced



- Address by multiple partitions per processor
  - Additional techniques such as work stealing available

# Task farm (master worker)

- Split the problem up into distinct, independent, tasks



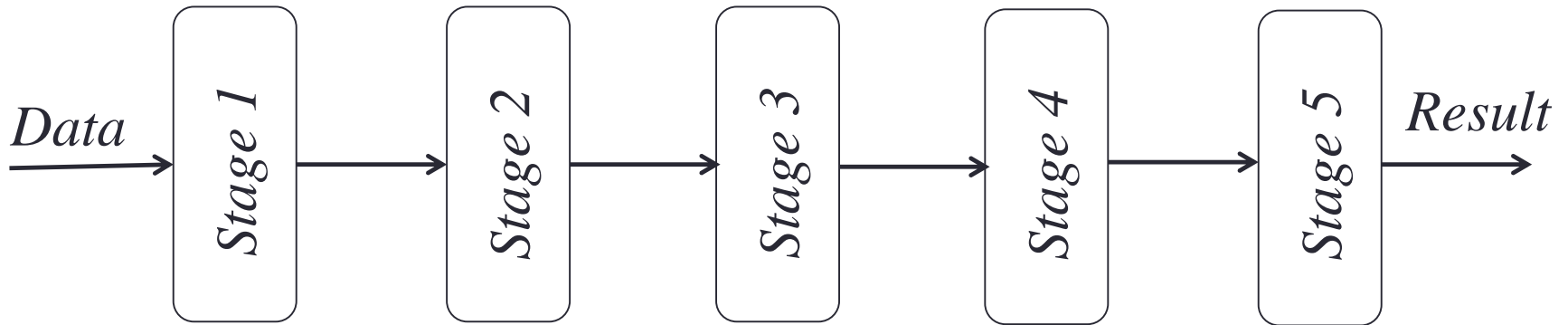
- Master process sends task to a worker
- Worker process sends results back to the master
- The number of tasks is often much greater than the number of workers and tasks get allocated to idle workers
- If known in advance, order jobs and send largest jobs first

# Task farm considerations

- Communication is between the master and the workers
  - Communication between the workers can complicate things
- The master process can become a bottleneck
  - Workers are idle waiting for the master to send them a task or acknowledge receipt of results
  - Potential solution: implement work stealing
- Resilience – what happens if a worker stops responding?
  - Master could maintain a list of tasks and redistribute that work's work

# Pipeline

- A problem involves operating on many pieces of data in turn. The overall calculation can be viewed as data flowing through a sequence of stages and being operated on at each stage.



- Each stage runs on a processor, each processor communicates with the processor holding the next stage
- One way flow of data

# Examples of pipeline

- CPU architectures
  - Fetch, decode, execute, write back
  - Intel Pentium 4 had a 20 stage pipeline
- Unix shell
  - i.e. `cat datafile | grep "energy" | awk '{print $2, $3}'`
- Graphics/GPU pipeline
- *A generalisation of pipeline (a workflow, or dataflow) is becoming more and more relevant to large, distributed scientific workflows*
- *Can combine the pipeline with other decompositions*



# Loop parallelism

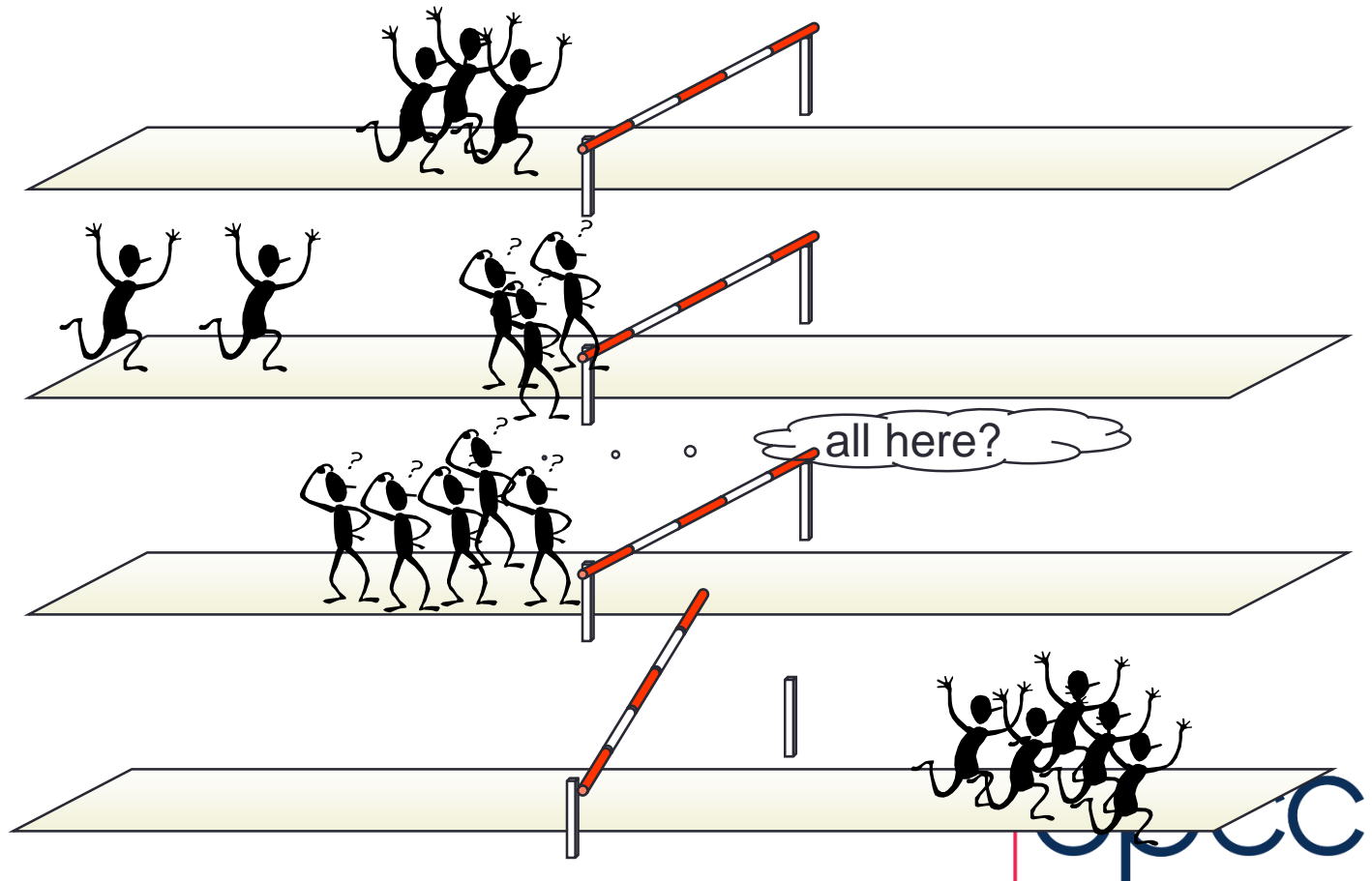
- Serial programs can often be dominated by computationally intensive loops.
- Can be applied incrementally, in small steps based upon a working code
  - This makes the decomposition very useful
  - Often large restructuring of the code is not required
- Tends to work best with small scale parallelism
  - Not suited to all architectures
  - Not suited to all loops
- If the runtime is not dominated by loops, or some loops can not be parallelised then these factors can dominate (Amdahl's law.)

# Example of loop parallelism:

```
int main(int argc, char *argv[]) {  
    const int N = 100000;  
    int i, a[N];  
  
    #pragma omp parallel for  
    for (i = 0; i < N; i++)  
        a[i] = 2 * i;  
  
    return 0;  
}
```

- If we ignore all parallelisation directives then should just run in serial
- Technologies have lots of additional support for tuning this

# Barrier and Synchronization



# Performance metrics

How is my parallel code performing and scaling?

# Performance metrics

- A typical program has two categories of components
  - Inherently sequential sections: can't be run in parallel
  - Potentially parallel sections

- 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  $E(N) \leq 1$

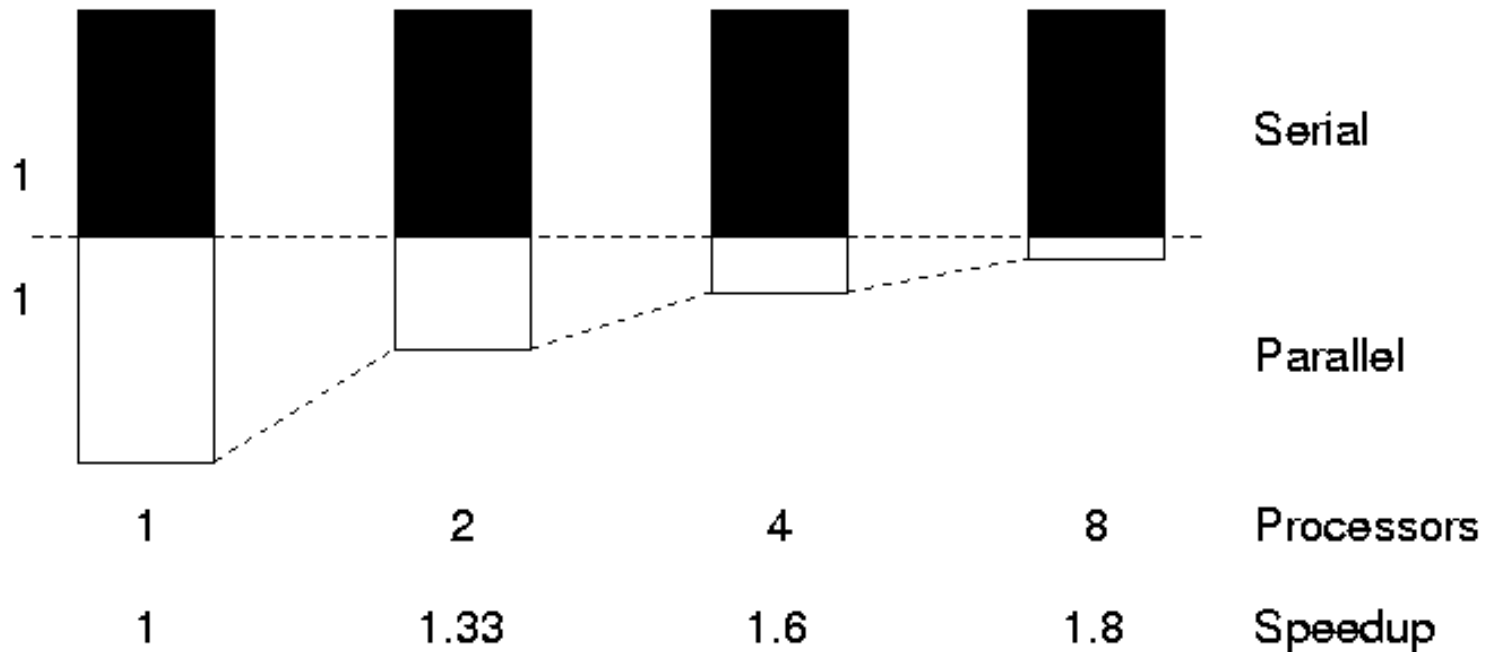
$$E(N) = \frac{T_{best}(N)}{T(N, 1)}$$

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

# 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*



# Amdahl's law

- A fraction,  $\alpha$ , is completely serial

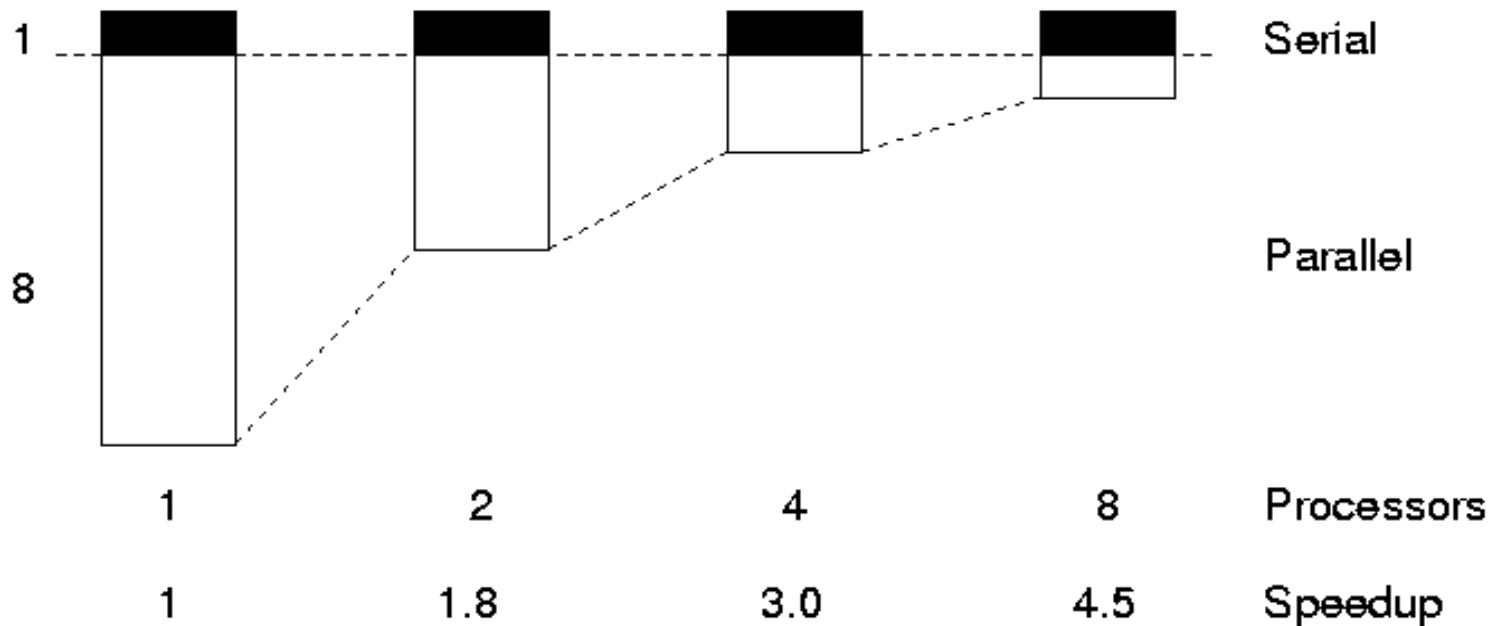
- Parallel runtime 
$$T(N, P) = \alpha T(N, 1) + \frac{(1-\alpha)T(N, 1)}{P}$$
  - Assuming parallel part is 100% efficient

- 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  $1/\alpha$  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



# Gustafson's Law

- If you can increase the amount of work done by each process/task then the serial component will not dominate
  - Increase the problem size to maintain scaling
  - This can be in terms of adding extra complexity or increasing the overall problem size.

*Due to the scaling of  $N$ ,  
effectively the serial  
fraction becomes  $\propto 1/P$*

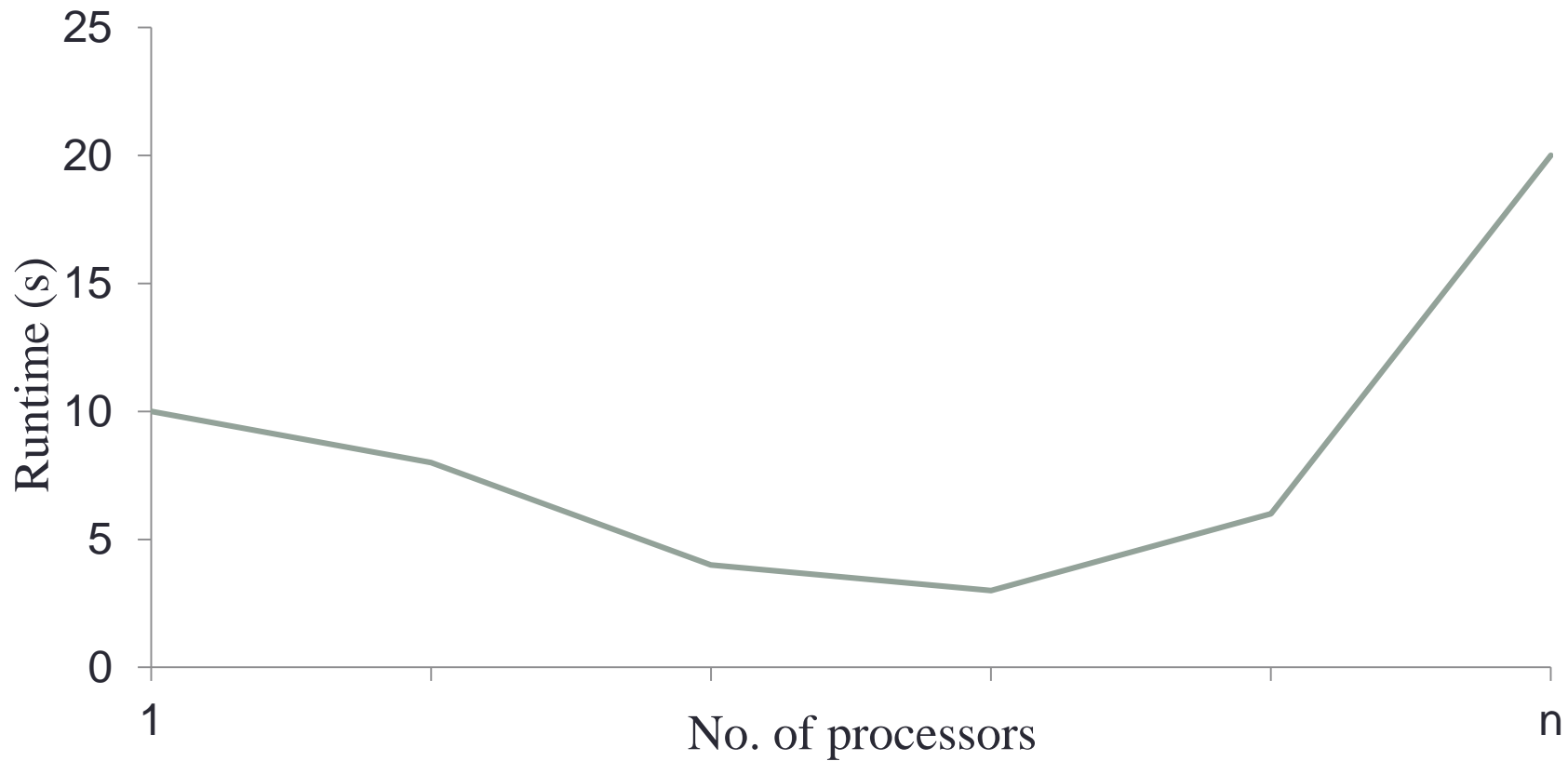
- For instance,  $\approx 0.1$ 
  - $S(16 \times N, 16) = 14.5$
  - $S(1024 \times N, 1024) = 921.7$

# 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

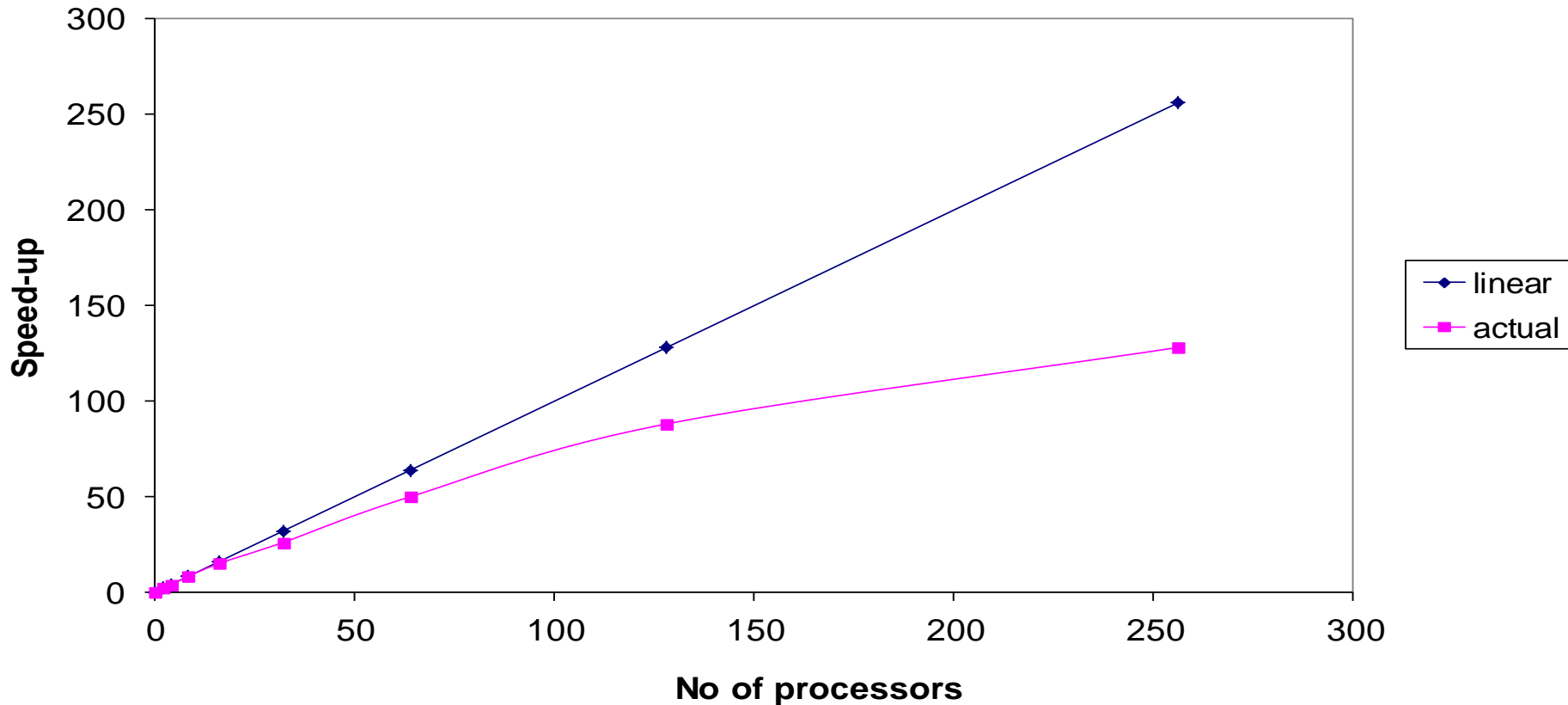
# Strong scaling

Example runtime vs No. of processors



# Weak scaling

Speed-up vs No of processors



# Summary

- There are a variety of considerations when parallelising code
- Scaling is important, as the more a code scales the larger a machine it can take advantage of
- Metrics exist to give you an indication of how well your code performs and scales
- A variety of patterns exist that can provide well known approaches to parallelising a serial problem