Deciding how to Parallelise a Problem

Introduction

We are going to look at two separate, but closely related aspects of how to parallelise a problem:

- Parallel Decomposition
  - How to split the problem into portions that can be solved in parallel
- Parallel Communication Architecture
  - How to communicate between nodes to ensure that they have the data required to solve the problem

Parallel Decomposition

Introduction

- There are a vast number of different ways in which you can potentially split problems in order solve them in parallel
- Even for a given problem there is usually not a single way in which it might be split
  - Not even necessarily a single best way to split a problem
  - May depend on computer resources available:
    - Number of cores available
    - Amount of memory available of each node
    - Relative speed of processors vs communications
- We will therefore only be looking at a limited set of common ways to split a problem

Data Decomposition

Split the data between the processes:

- Split the output data
- Split the input data
- Split both the input and output data

Data Decomposition: Split the output data

- In many problems, given complete knowledge of the input data, different portions of the output data can be calculated independently
- Example of where it is appropriate
  - Matrix multiplication – Each element in the answer matrix can be calculated independently of the others
  - Communication is required to distribute the input data, but not between the nodes

Do Worksheet 6 Exercise 1

Data Decomposition
Split the input data

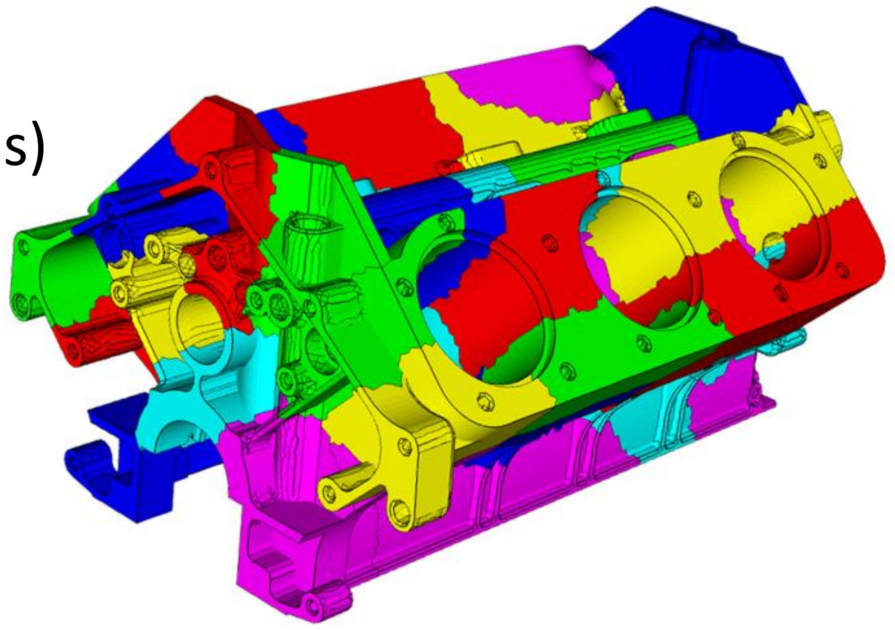
- Different portions of the input data assigned to different processes
- Contributing to a commonly held solution
  - Either globally known (e.g. a shared memory system) or known to a single primary node

Data Decomposition
Split both the input and output data

- Different portions of the input and output data in a problem may be closely related
  - E.g. When simulating a physical system there may be input data about a region or time, with some other properties of the system being calculated for those positions or times
- Very common approach in both distributed and shared memory systems
  - In distributed memory systems this will often require communication of information at the boundary of the data
  - In distributed memory system it restricts the need for blocks to memory associated with the edge of the data regions

Data Decomposition
Domain Decomposition

- Domain decomposition is a very commonly used example where both input and output data is split
- Used in the simulation of physical systems
  - The system is divided into a set of regions (domains)
  - Each process responsible for a different domain
  - Communication of data at the edge of domains

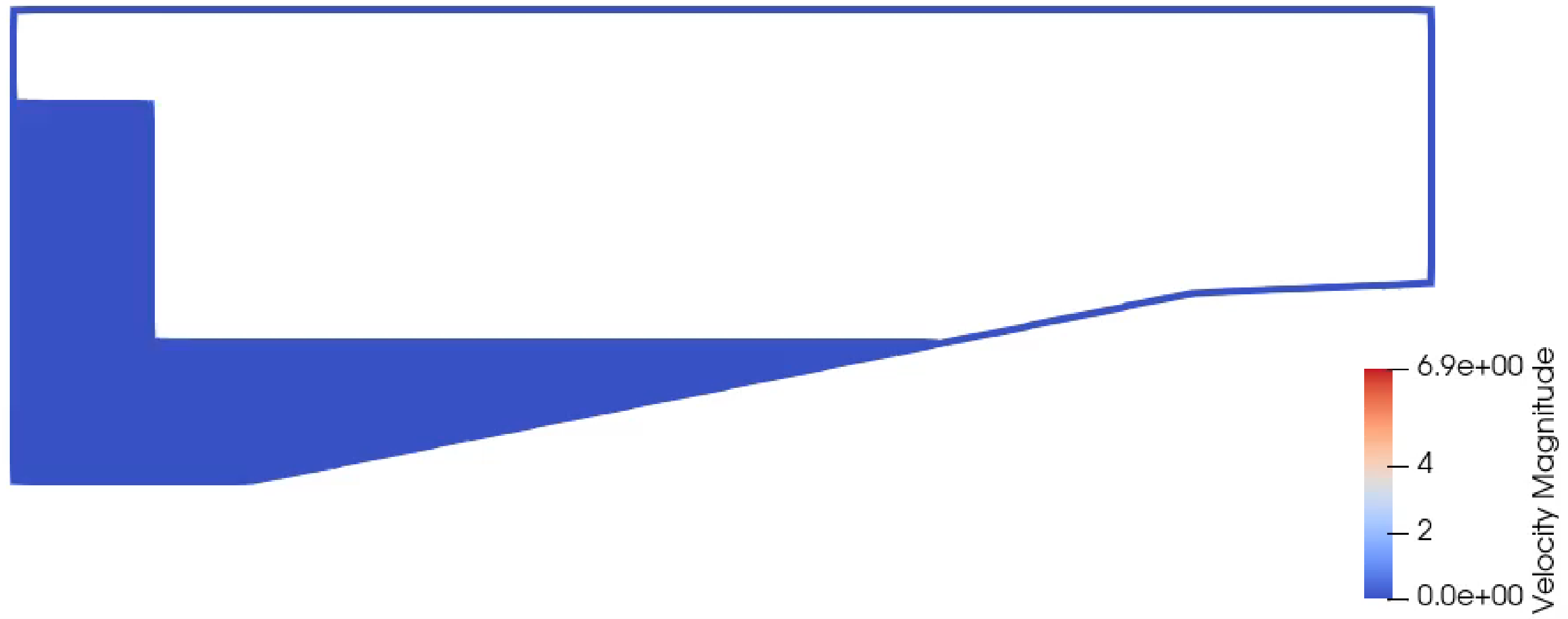


Domain decomposition will form the basis for the coursework

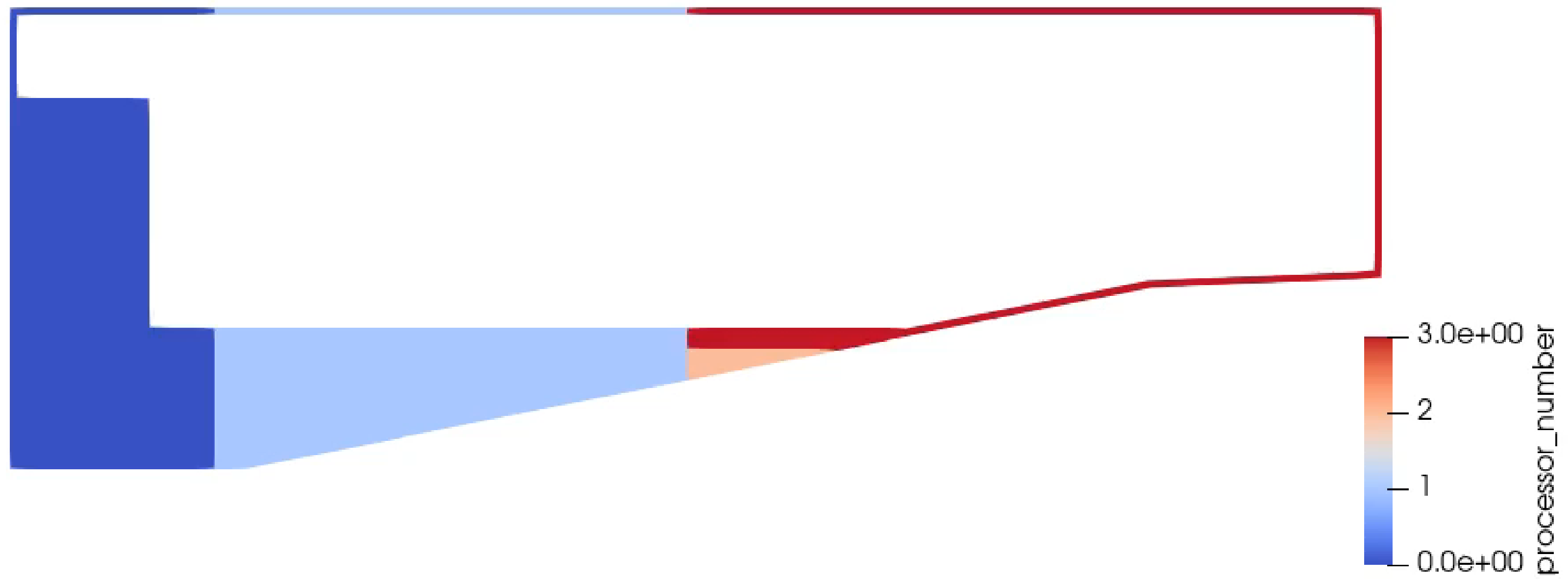
Data Decomposition
Load Balancing

- To ensure maximum parallel efficiency you don’t want some of the processes waiting idle while other processes work
- Need to spread the computational load – try to give each process the same amount of work to do
  - When splitting should err on giving one process slightly less to do than the others rather than slightly more
  - Typically want all processes to be responsible for the same number of degrees of freedom (e.g. the same number of nodes or elements)
  - If the simulation resolution is spatially constant this may be equivalent to evenly splitting the sizes of the regions
- Need to simultaneously try to keep communication to a minimum
  - Reduce the surface area of the regions
- Sometimes hard to assess the computational cost of different decompositions analytically
  - A tactic I sometimes use is to time how long a process is idle waiting for the communications on other processes to complete
    - Balance based on idle time – Give those waiting longest more to do

Load Balancing in Smoothed Particle Hydrodynamics (SPH)



- I do a lot of simulation work using SPH – Load balancing is one of the keys to efficient simulation
  - This is a simple simulation run on 4 cores – Colour on bottom video corresponds to processor number



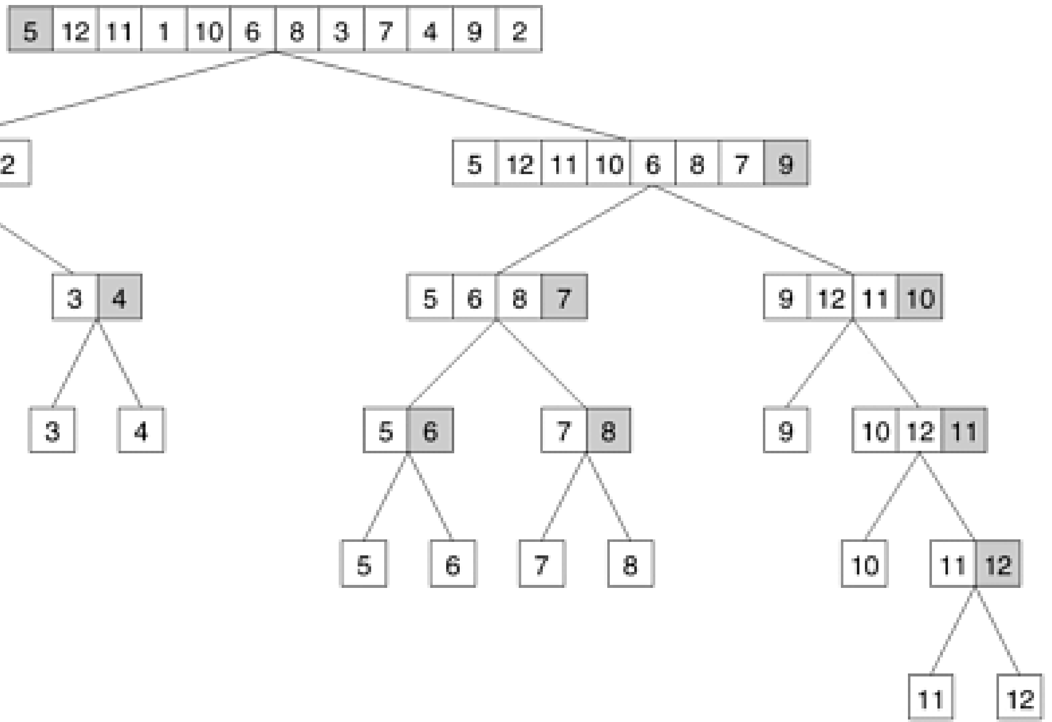
Do Worksheet 6 Exercise 2

Recursive Decomposition

- Progressively split a problem into smaller portions and then combine the solutions
  - Works best for problems where the computational cost of the solution increases faster than linearly with the size of the problem and where the cost of splitting and combining solutions is comparatively cheap
  - So called “divide and conquer” algorithms
  - Often used in serial algorithms, but can also be used as the basis for parallel decomposition

Recursive Decomposition - Quicksort

- Classic example of recursive decomposition is the quicksort algorithm
  - Brute force sorting of *n* items is O(n^2)
  - …,but you can split lists in O(*n*) time and combine them in O(*1*) time with the total time taken being approximately O(*n log(n)*)



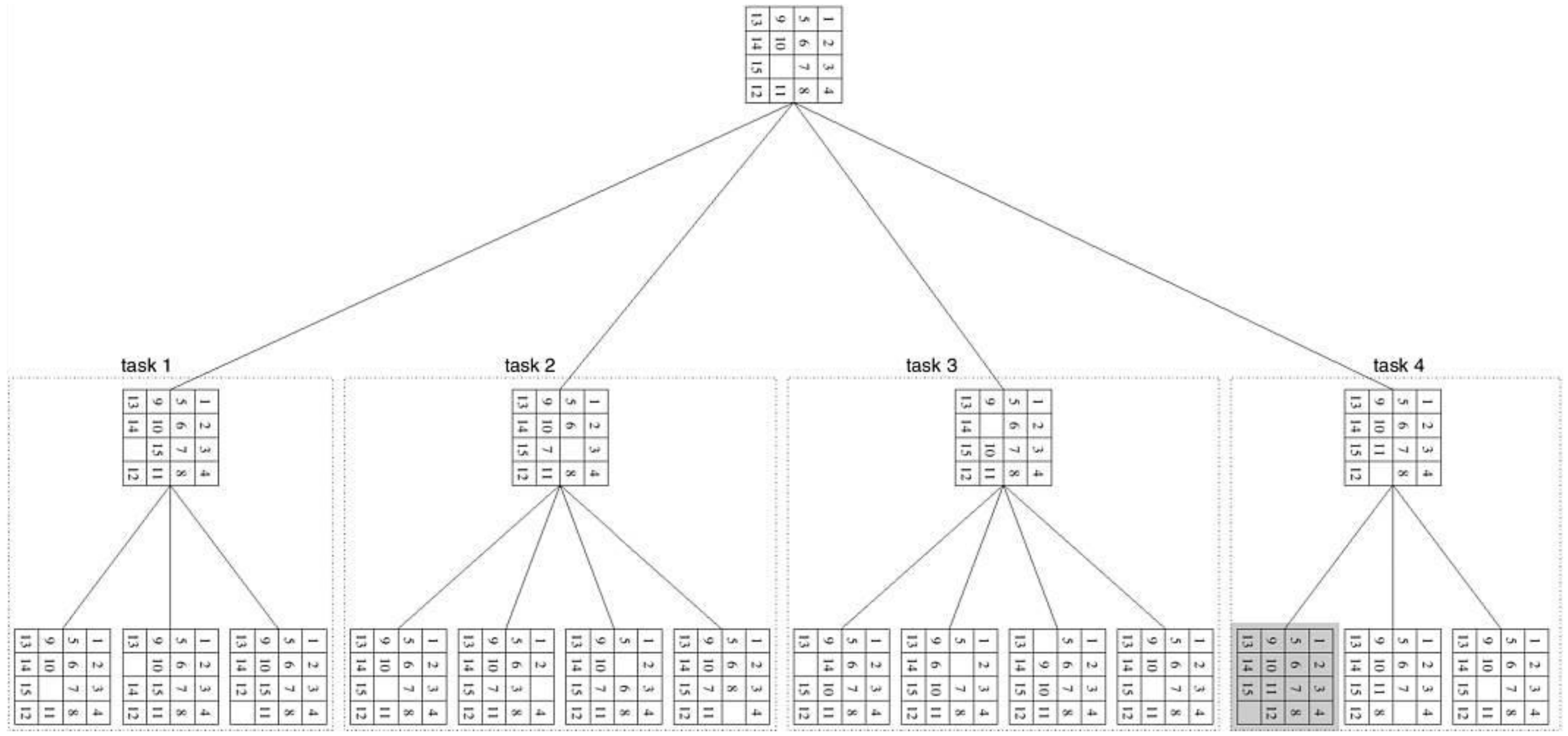
Do Worksheet 6 Exercise 3

Exploratory Decomposition

- Used for searching for solutions in multi-step problems
- Has some similarity to data decomposition in that the initial search space is split between processes
  - Note that this might not be a true split of data, but could be a split of a parameter space
- The next stage is that a new set of data/parameters are produced from the previous stage and these are then split
  - Can be split onto new processes if available, otherwise the current processes become responsible for more states
  - Could allow processes to become available again if their search hits a dead-end

Exploratory Decomposition Example

- Finding solution to 15 puzzle problem



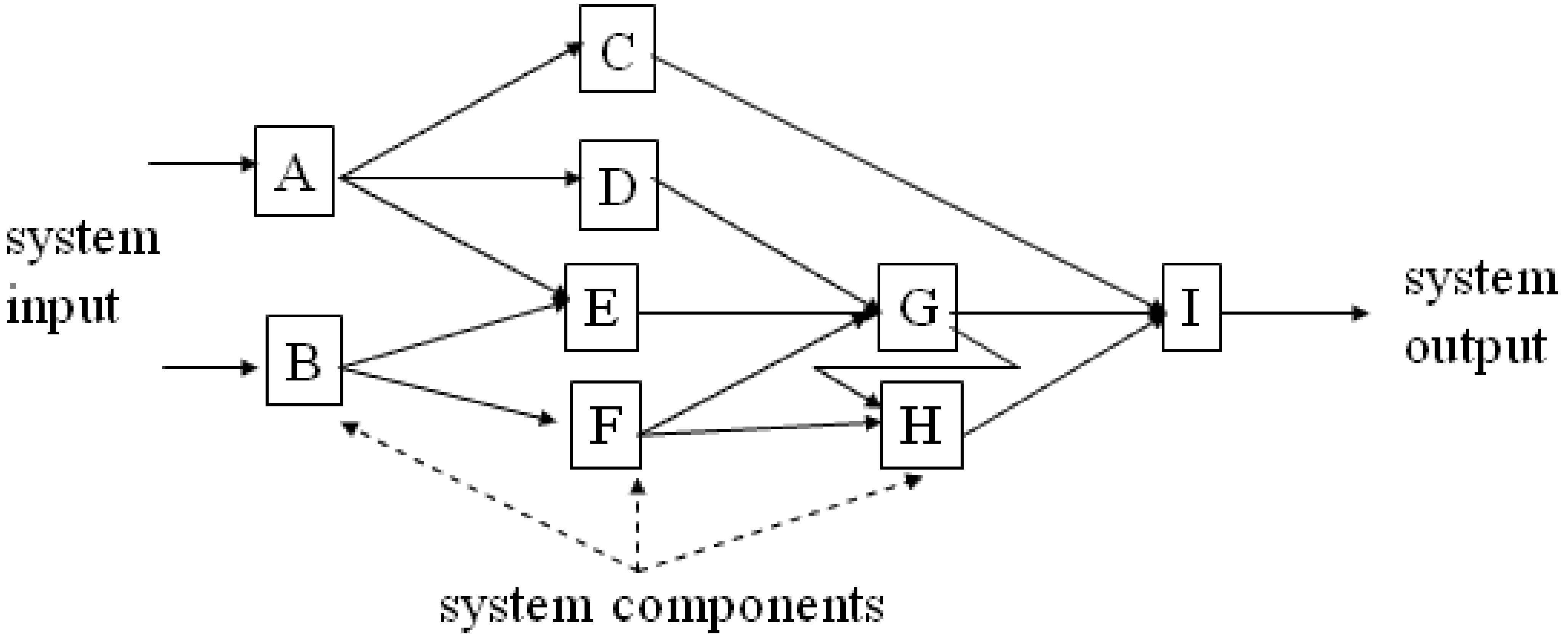
This is the Workshop Exercise

Speculative Decomposition

- Can be used in problems where subsequent tasks depend on the outcome of earlier tasks
  - E.g. two different tasks to complete depending on whether the solution to an earlier task is true or false
- In speculative decomposition you carry out all the subsequent tasks without waiting for the result from the earlier task
- Particularly advantageous if the earlier tasks take a larger or similar amount of time compared to all of the subsequent tasks

Speculative Decomposition Example

- Simulation of system with multiple interacting components



Parallel Communication Architectures

Different Types of Architecture

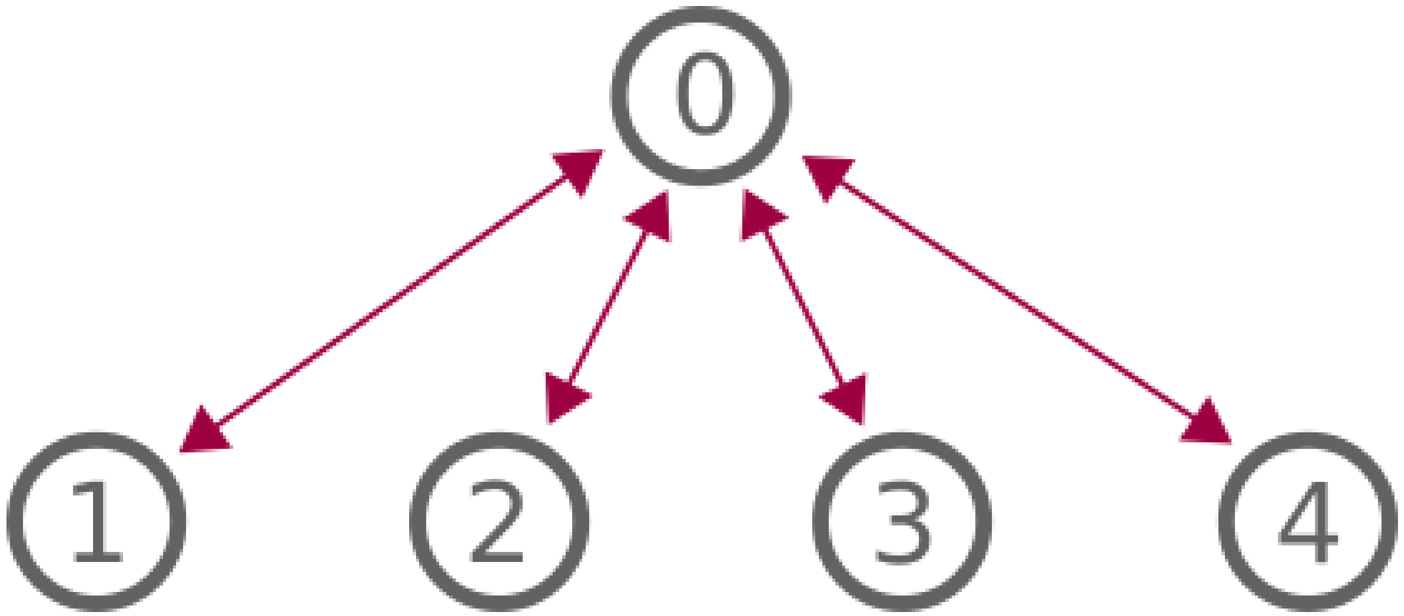
- As MPI simply provides the communication tools, it does not dictate the parallel architecture that will be used

Two main types of architecture

- Primary/Replica
  - A primary node controls the other nodes
  - …historically this has been referred to as a Master/Slave architecture
    - There is no consensus on a replacement name, but Primary/Replica is the name used by Microsoft and Amazon and so it is what I will be using
    - Other names used include Master/Minion, Primary/Secondary, or Chief/Worker
- Peer to Peer
  - All nodes are equivalent to one anther

Primary/Replica

- What is a Primary/Replica architecture?
  - All communications go through a primary node that controls a set of nodes that do tasks for it



Primary/Replica

- Advantages:
  - Often relatively straightforward to implement
  - A single process has access to all the data
- Disadvantages:
  - Scalability issues as communications into the primary node can become a bottleneck

Primary/Replica

When might you consider using a primary/replica architecture:

- Trivially parallel problems
  - Problems with sub-tasks that can be carried out completely independently
  - Primary farms out the problems to all the replica nodes and waits for the answers to come back

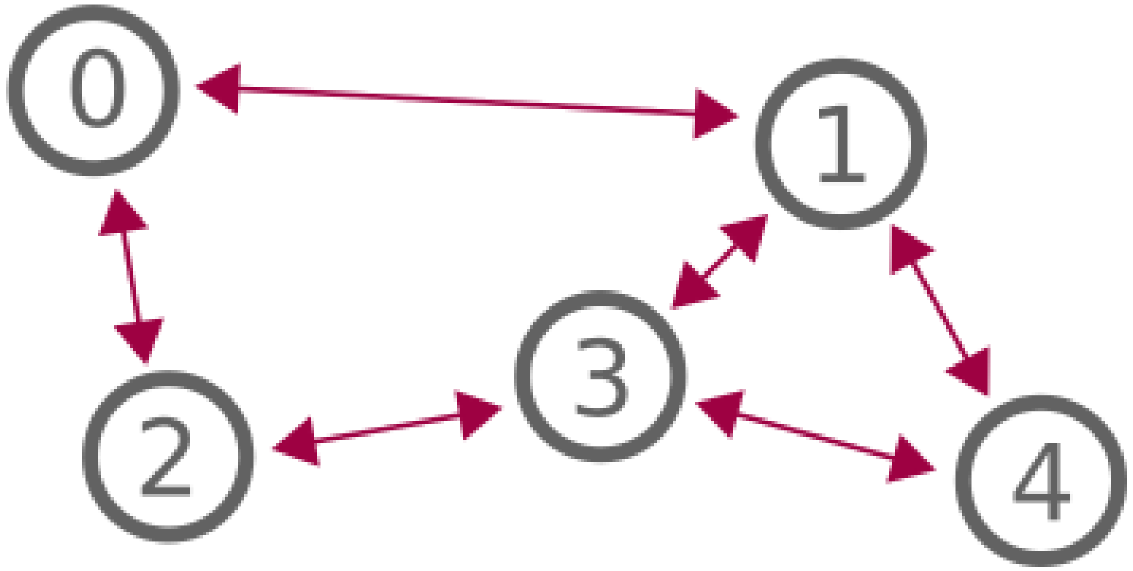
- Problems where data from all the nodes need to be collated
  - Information not just exchanged between nodes but needs to be combined
  - Might form part of an otherwise peer-to-peer architecture – see hybrid architectures

Primary/Replica

- Types of communications to be used –
    - A Primary/Replica architecture will usually rely heavily on collective communications
        - This will spread some of the communication load away from the primary node
    - Scatter/Gather or Scatter/Reduce the usual communication methods
    - Can use non-blocking point to point if you wish to respond to individual replica nodes as they complete
        - Send new data to a node without waiting for all the nodes to complete

Peer to Peer

- What is a Peer to Peer architecture?
  - Every process has equal precedent and communicates directly with the other process (or, more usually, a subset of them)



Peer to Peer

- Advantages
  - Very good for problems that are strongly coupled, especially if each node needs to only communicate with a subset of neighbours
  - Good scalability as the amount of communication into a particular node will typically be independent of or only a weak function of the number of processes used if the number of neighbours communicating with one another is system size independent
    - Often the case in, for instance, domain decomposition problems
- Disadvantages
  - Often harder to code as no node is in charge of the system
  - Typically, no node will know the entire solution
    - Need to post-process results from different nodes

Peer to Peer

- Types of communications to be used –
  - Will mostly be reliant on non-blocking point to point communications - MPI_Isend and MPI_Irecv
  - Most appropriate choice if nodes only communicate with a subset of the other nodes and these sets of communications are fully interlinked
  - May sometimes need to communicate data between all the nodes (e.g. obtaining a single timestep when using a dynamic timestep)
    - MPI_Allgather or MPI_Allreduce most appropriate

Hybrid Architectures

- It is not required that a program stick religiously to single communication architecture
- An example of where a hybrid architecture is appropriate might be domain decomposition with dynamic load balancing
  - The main calculation loop is most efficiently done using peer-to-peer communications as each process only needs to communicate with the processes responsible for the neighbouring domains
  - For dynamic load balancing each processes needs to calculate their own computational and communication load, but this information needs to be collated at a single process in order for a new domain distribution to be calculated and distributed – a primary/replica type communication arrangement

Parallel Performance

Assessing the Performance of Parallel Code

Two main measures of performance:

- Speedup ratio:
  - How many times quicker the code is in parallel relative to the serial code

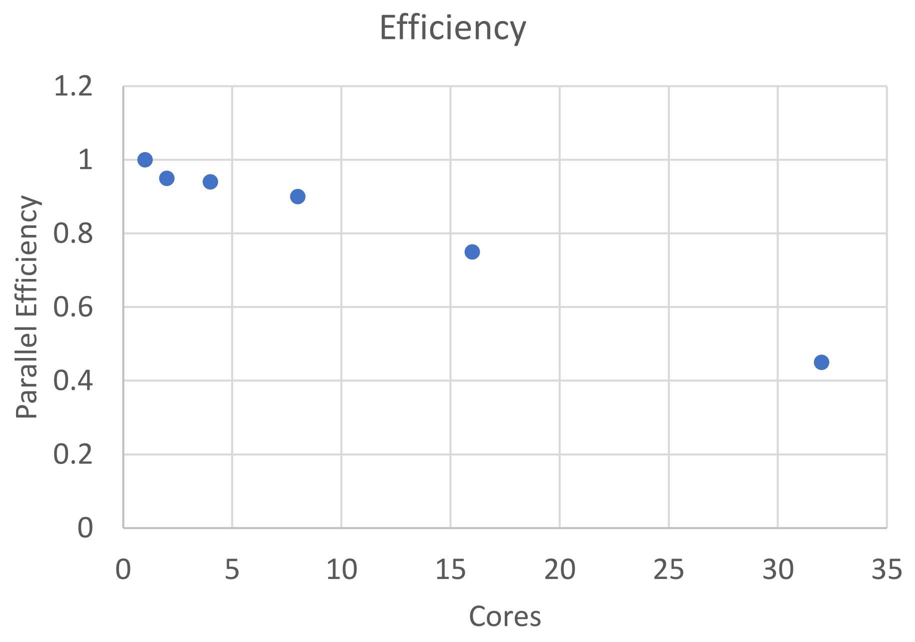
$$S = \frac{T_1}{T_N}$$

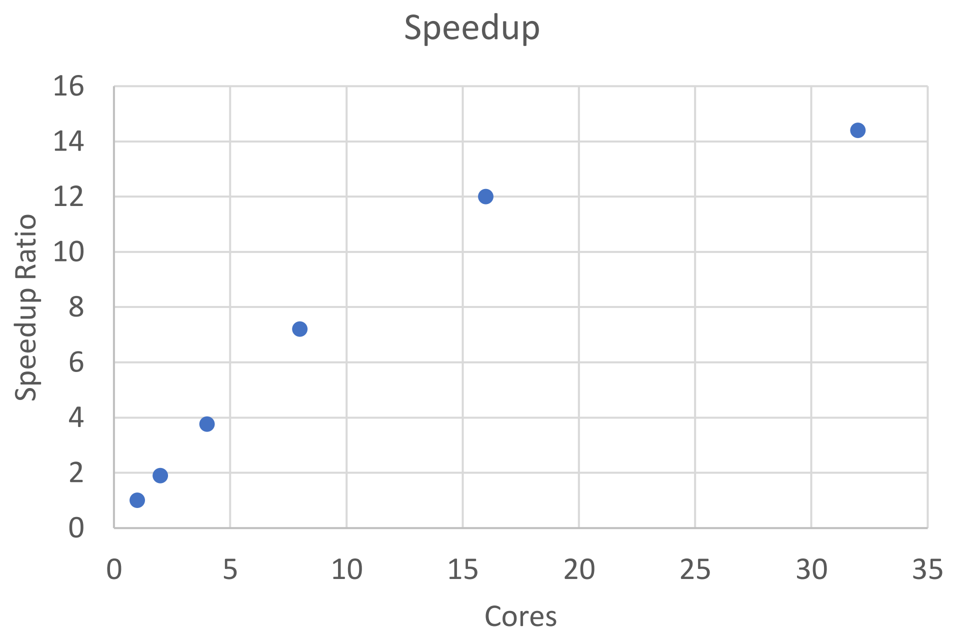
- Parallel Efficiency
  - How fast is the code relative to an ideal speedup

$$E = \frac{T_1}{N\,T_N} = \frac{S}{N}$$

Speedup and Parallel Efficiency

- Parallel efficiency will usually drop as the number of cores used increases
  - It is possible to have super-linear speedup (efficiencies greater than one), but quite rare and will usually be caused by smaller tasks having more efficient cache usage





Amdahl’s Law

- Based on the idea that part of the solution is in parallel and part is in serial
    - f is the fraction of the code that is executing in parallel
        - The fraction of the code based on execution time
    - Assumes that the parallel portion has an efficiency of 1

$$T_N = (1-f)T_1 + \frac{f}{N}T_1 = T_1\left(1 - f + \frac{f}{N}\right)$$

$$S = \frac{1}{1 - f + \frac{f}{N}} \qquad E = \frac{1}{N(1-f)+f}$$

- This implies that the speedup can never be greater than $\frac{1}{1-f}$ irrespective of the number of cores used

Communication and Parallel Efficiency

- In distributed memory codes, most of the calculations are in parallel
  - Inefficiency often comes from the relative amount of time spent transferring data (or waiting for communications) relative to the amount of time spend doing calculations
  - Communicated data will often involve repeating the same calculations on more than one processor (a serial like behaviour)
  - In some codes not quite as clear cut as this as you can do calculations while communicating

$$T_{Total} = T_{Calculate} + T_{Communicate}$$

- If we assume that the problem is of size P and that we can perfectly decompose the problem then

$$T_{Calculate} \propto \frac{P}{N}$$

- The communications (and any duplicated calculations) are associated with the "edges" of the data

$$T_{Communicate} \propto \left(\frac{P}{N}\right)^{n}$$

- Where n will typically be between zero and one and will often depend on the dimensionality of the data

Communication and Parallel Efficiency

- We can combine these to estimate how speedup and parallel efficiency might be expected to change with problem size and the number of cores used
    - Note that these are just approximations, but can be used to get a feel for the expected trends

$$E \approx \frac{1}{1 + kP^{n-1}N^{1-n}}$$

- Where k is problem specific and is related to the relative cost of communication and computation
- Given that $0 < n < 1$ this equation implies that:
    - For a given problem size, P, the efficiency drops as N increases
    - It also implies that bigger problems will have a higher efficiency when using the same number of cores

Domain Decomposition
Efficiency of domain decomposition

- If we assume a constant resolution then, for a 3D system, computational time will be roughly proportional to the volume of a region and communication to the surface area of the domain (volume of the domain raised to the power 2/3)
  - In 2D system the equivalent is the computational time varying with the area of the region and the communications with the perimeter of the region

For domain decomposition $n \approx \frac{d-1}{d}$

$$E_{3D} \approx \frac{1}{1 + k\left(\frac{V}{N}\right)^{-\frac{1}{3}}}$$

$$E_{2D} \approx \frac{1}{1 + k\left(\frac{V}{N}\right)^{-\frac{1}{2}}}$$

A note on parallel efficiency

- It is virtually always more efficient to have tasks carried out in parallel relative to being data parallel
  - E.g. if you have 10 large simulations to complete, have parallel code to carry out the simulations and have 100 cores available to you:
  - Two main options:
    - Carry out each of the simulations on 100 cores, doing this for each of the 10 simulations
    - Carry out all 10 simulations at the same time, each using 10 cores.

  - The second option is typically the best one
    - The parallel efficiency of carrying out the individual simulations drops as the number of cores increase
    - The exception to this heuristic will be when some of the simulations to be carried out are computationally much more expensive than others

  - Of course there is the issue of when you get your first results back
    - Running fewer jobs on more cores each gets you your first results quicker than if you ran more jobs on fewer cores, though you will have to wait longer for all the jobs to complete