Parallel Communication Architectures

Different Types of Architecture

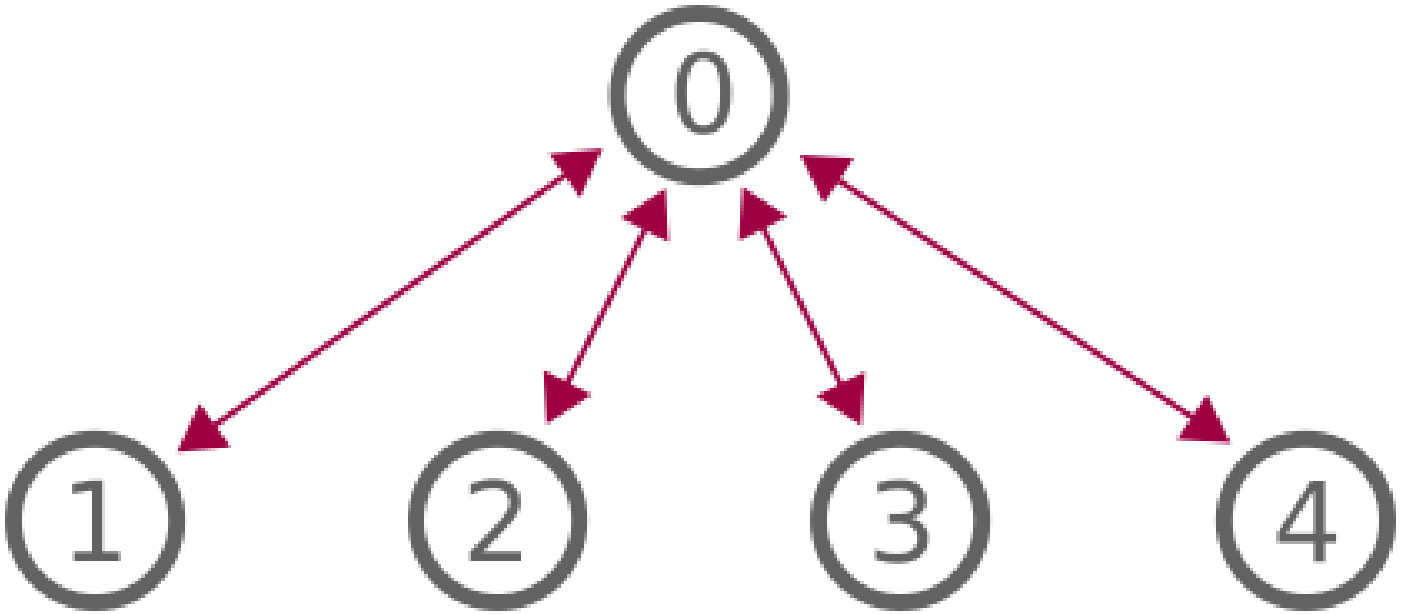
- As MPI simply provides the communication tools, it does not dictate the parallel architecture that will be used

Two main types of architecture

- Master/Slave
  - A master node controls the other nodes
  - …people are looking to change the name, but no agreement on replacement
    - Proposals include Master/Minion, Main/Secondary, or Primary/Replica
- Peer to Peer
  - All nodes are equivalent to one anther

Master/Slave

- What is a Master-Slave architecture?
  - All communications go through a master node that controls a set of nodes that do tasks for it



Master/Slave

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

Master/Slave

When might you consider using a master/slave architecture:

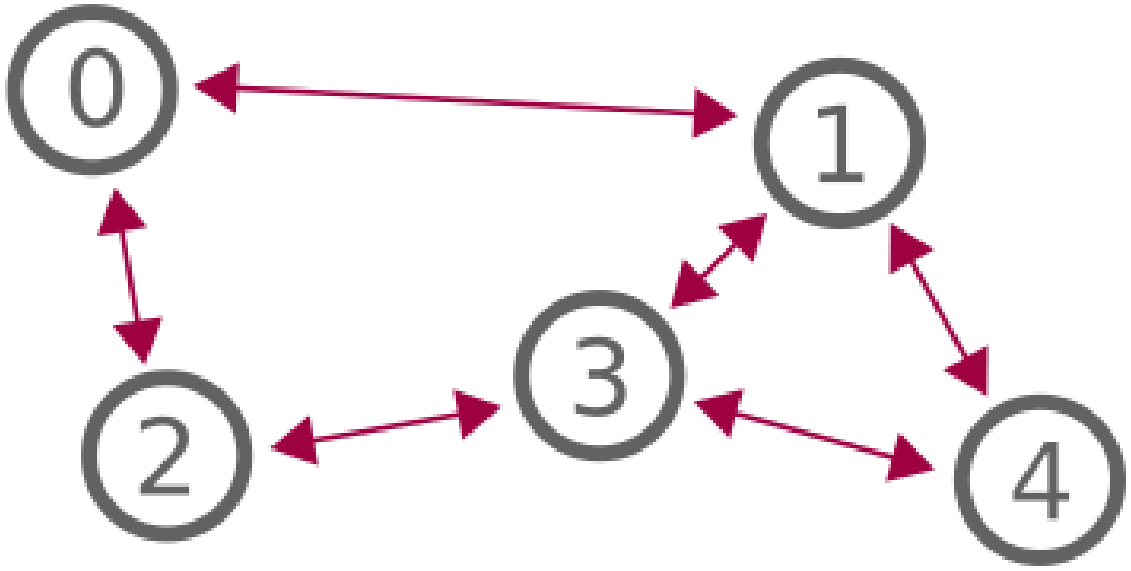
- Trivially parallel problems
  - Problems with sub-tasks that can be carried out completely independently
  - Master farms out the problems to all the slave nodes and waits for the answers to come back
- Problems where data from all the nodes need to be collated
  - Information not just exchanged between nodes but needs to be combined
  - Might form part of an otherwise peer-to-peer architecture – see hybrid architectures

Master/Slave

- Types of communications to be used –
  - A Master/Slave architecture will usually rely heavily on collective communications
    - This will spread some of the communication load away from the master node
  - Scatter/Gather or Scatter/Reduce the usual communication methods
  - Can use non-blocking point to point if you wish to respond to individual slave 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 master slave 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

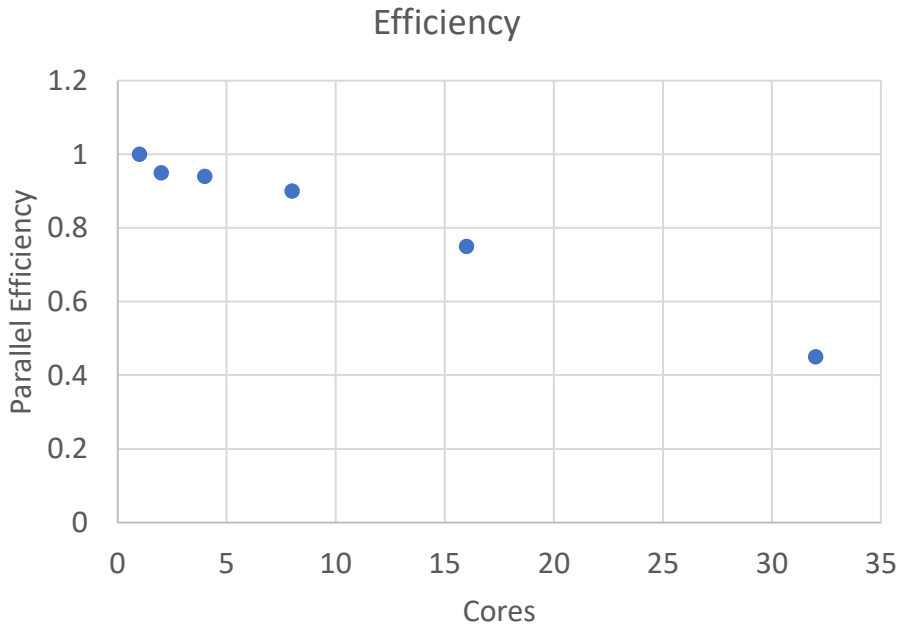
$$S = \frac{T_1}{T_N}$$

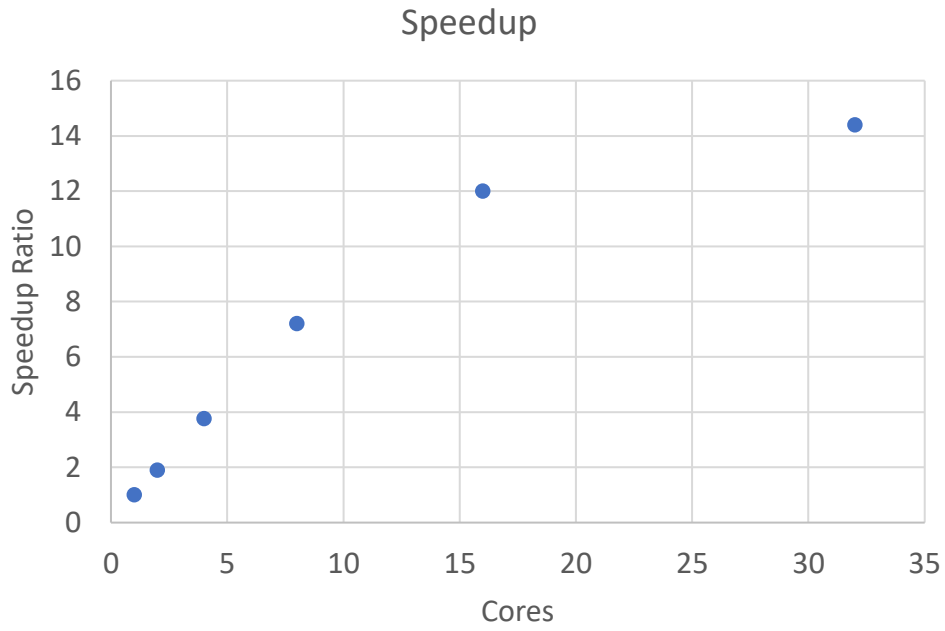
- Parallel Efficiency
  - How fast is the code relative to an ideal speedup

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

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 are often 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 domain 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 domain and the communications with the perimeter of the domain

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

$$E_{3D} \approx \frac{1}{1 + k\left(\frac{V}{N}\right)^{-\frac{1}{3}}} \qquad E_{2D} \approx \frac{1}{1 + k\left(\frac{V}{N}\right)^{-\frac{1}{2}}}$$

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