Concurrent and Distributed Programming (CSC1101)

Concurrent & -> Distributed Architectures & Considerations

2024/2025

Graham Healy

These course slides are partly adapted from the original course slides prepared by:
Dr Martin Crane, Dr Rob Brennan and Dr Takfarinas Saber

Parallel & Concurrent Programming Types Discussed in this Course

- Shared Memory
  - Communication is through shared variables using ...
    - Mutexes, semaphores, monitors, ...
  - Java concurrency, Open MP, ...
  - But ... there are scalability constraints on a single machine!

- Distributed (e.g., Clusters)
  - Needs message passing for communication (overhead versus **scalability**)
    - Overhead: network latency, bandwidth, dealing with problems like lost packets, ...
  - OpenMPI, RabbitMQ, web services, REST, Java + RMI (old), ...

Problem Decomposition / Finding Concurrency

- Identify concurrency and decide at what level to exploit it
  - Tasks, data, pipelines e.g., task decomposition vs data decomposition

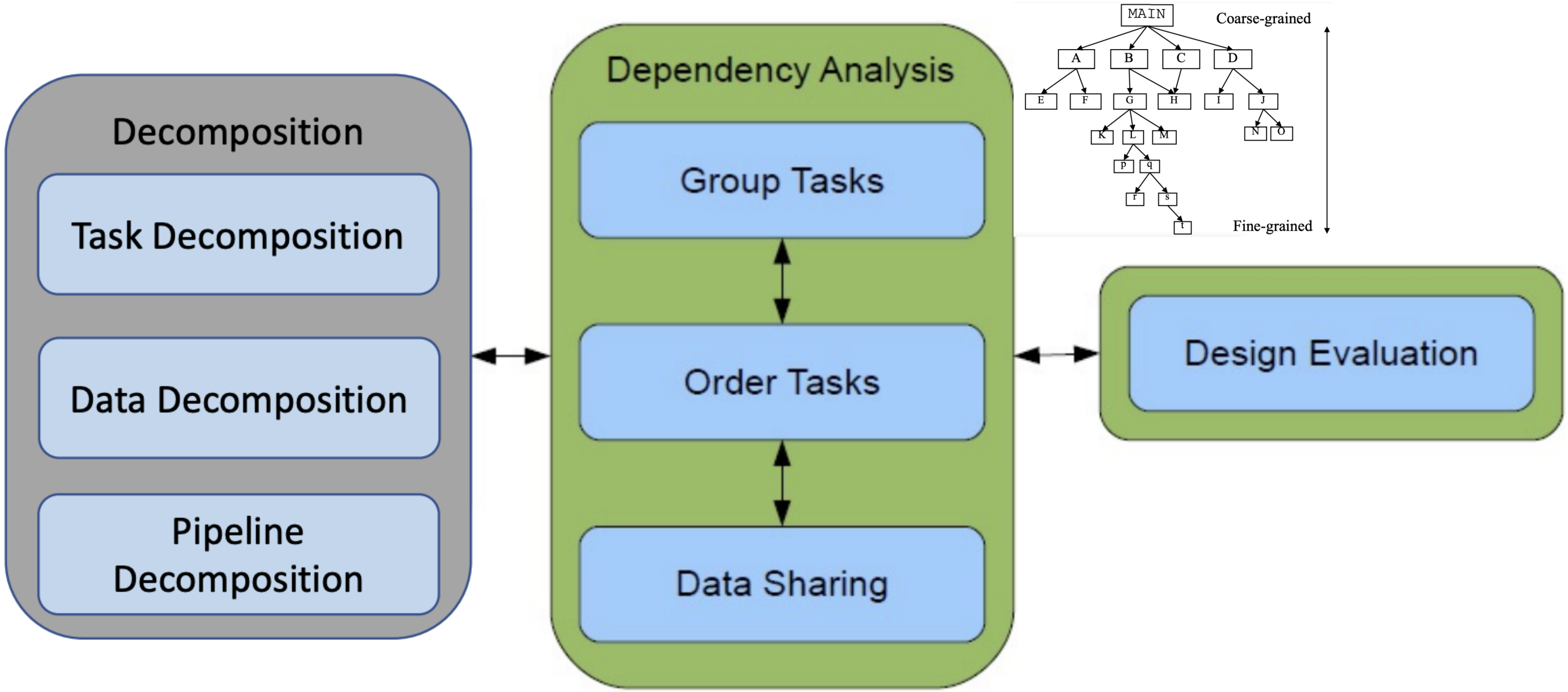
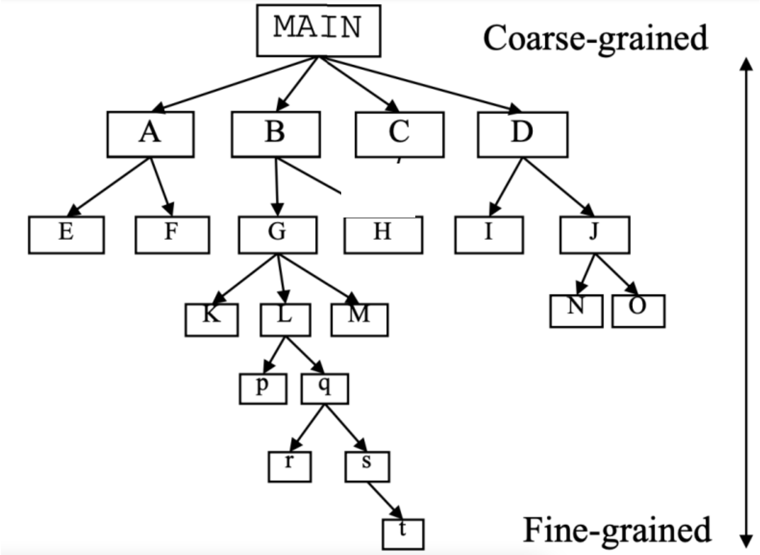


Figure based on ref [3]

Granularity

- Specify a mechanism to divide work among cores
  - load balance the work and minimise communication
- Programmers really worry about partitioning first
  - i.e., figuring out the parts of the application that they need to compose to make the application
- Preferable to keep program complexity down (so people often ignore highly complex solutions)



Task Decomposition

- Task decomposition is often harder than identifying data parallelism
- Requires a good understanding of the problem
- Goal, find independent coarse-grained computations/activities that are inherent to the algorithm
  - Ideally pick "natural" decompositions that fall out of the processing rather than forcing some pattern on them
- Most tasks follow from the way the programmer thinks of a problem
  - Often, it is easier to start with too many tasks and fuse them later rather than trying to split
- Task choices will impact software engineering decisions and implementation

Data Decomposition

- Key: Find where the same operations are applied to different data again and again

- Data decomposition is first when:
  - Main computation is organised around manipulation of a large data structure
  - Similar operations are applied to different parts of the data structure

- ... not really separate from task decomposition

Common Data Decompositions

- Array data structures
  - Decompose along rows, cols, blocks
- Recursive data structure e.g., tree
  - Subdivide into left/right
- Need to work out how to recombine the results
- This pattern is particularly useful when the application exhibits **locality of reference**

  i.e., when processors/threads can refer to their own partition only and need little or no communication with other processors/threads

Example

- Matrix-vector product
  $Ax = b$
- Matrix `A[][]` is partitioned into P horizontal blocks
- Each processor
  - operates on one block of `A[][]` and on a full copy of `x[]`
  - computes a portion of the result `b[]`

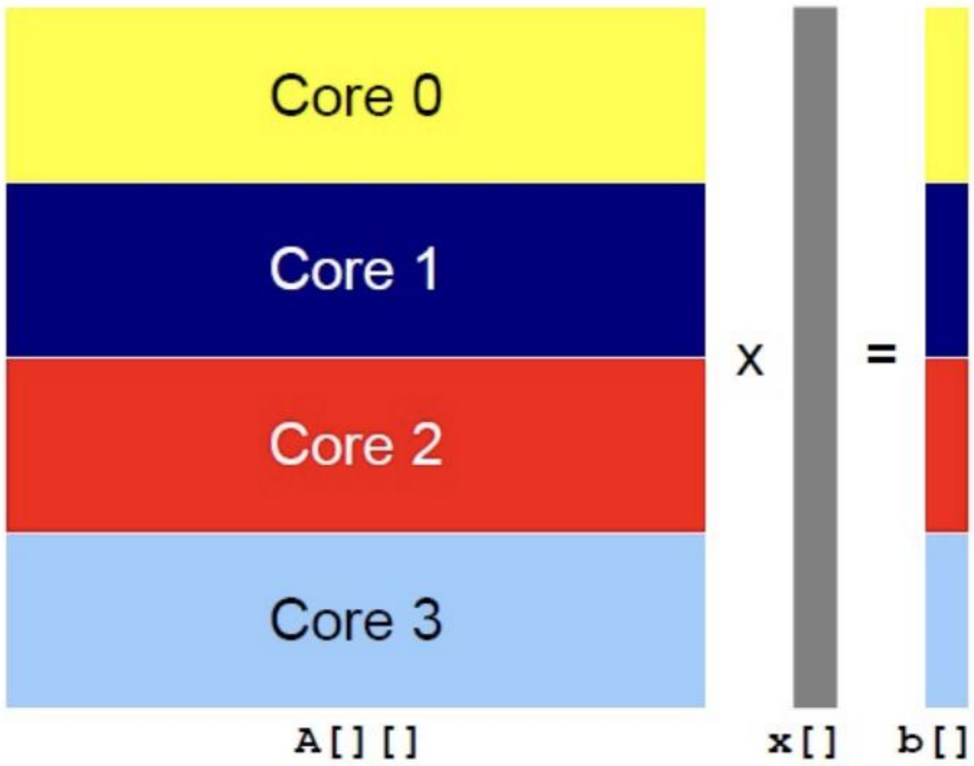


Figure based on ref [4]

Pipeline Decomposition

- Use where data is flowing through a sequence of stages
  - Assembly line is a good analogy
  - E.g., instruction pipeline in modern CPUs
  - E.g., pipes in Linux
  - E.g., signal processing

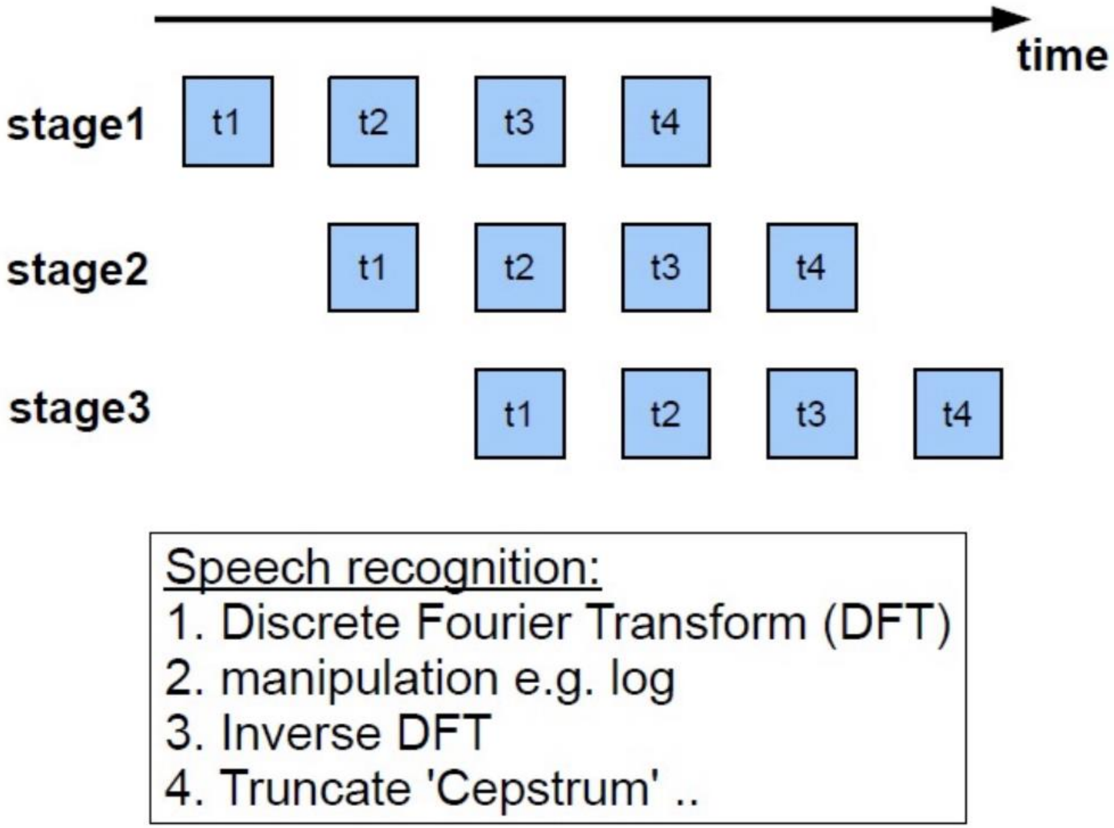
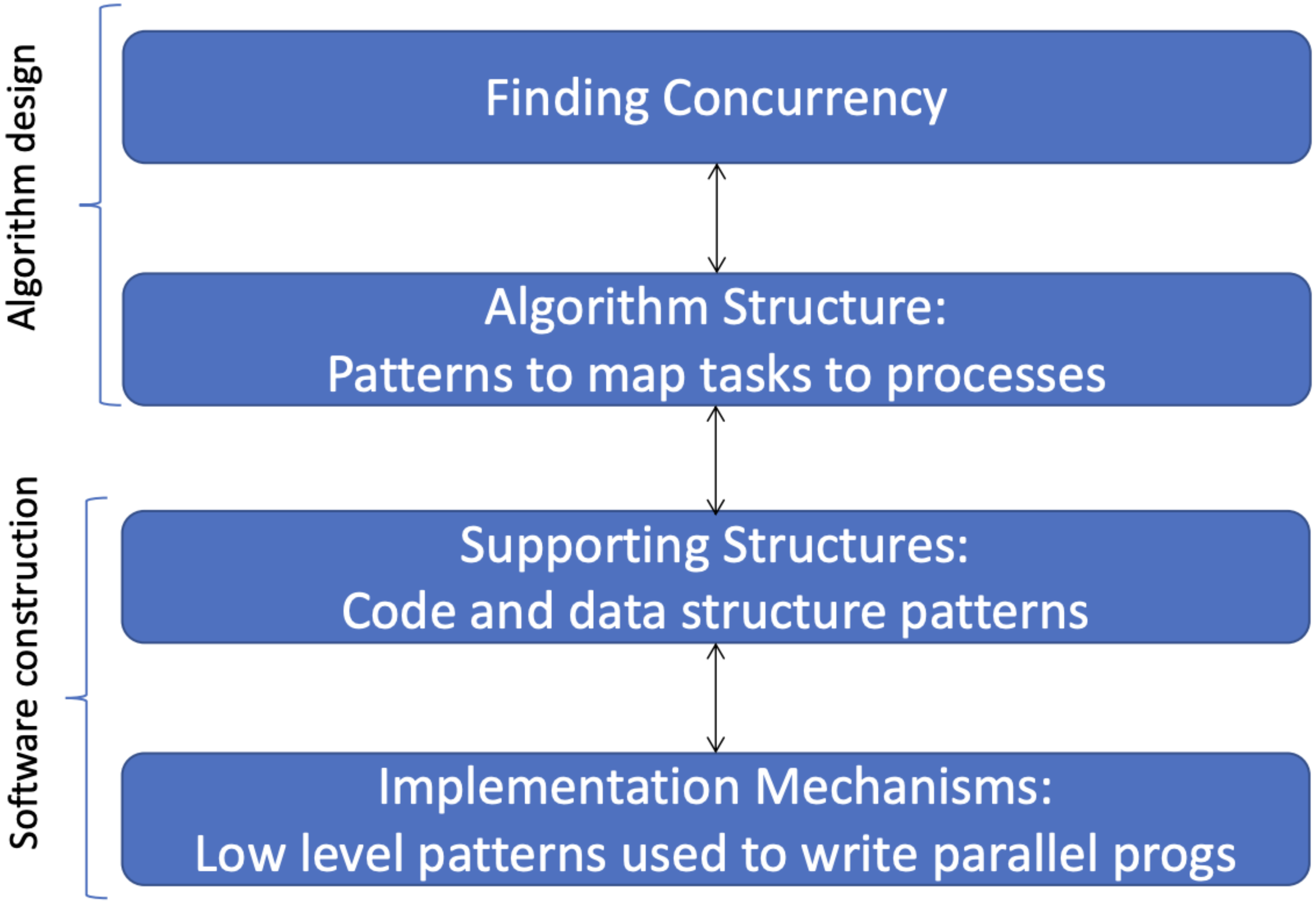


Figure based on ref [3]

Four Design Spaces



Problem Decomposition / Finding Concurrency

- Identify concurrency and decide at what level to exploit it
  - Tasks, data, pipelines e.g., task decomposition vs data decomposition

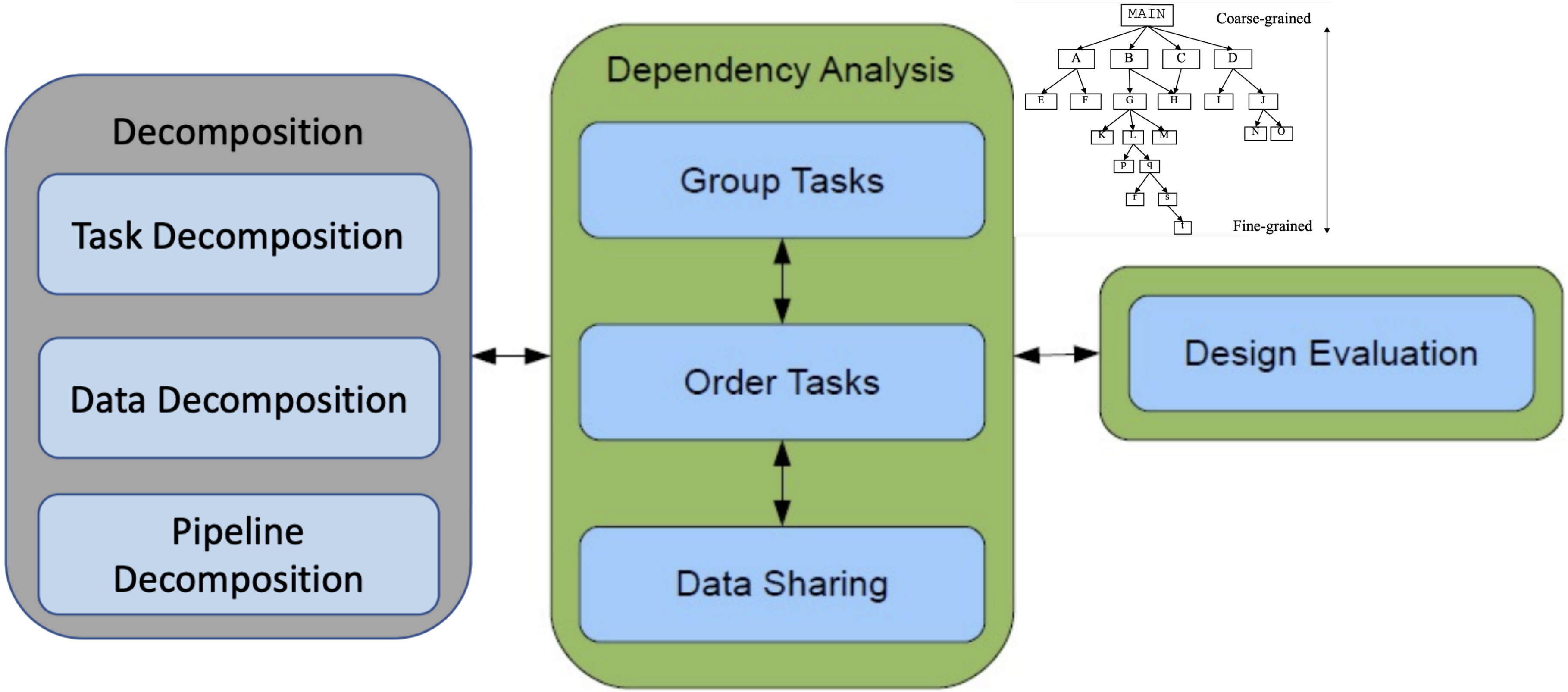


Figure based on ref [3]

Algorithm Structure Design Space

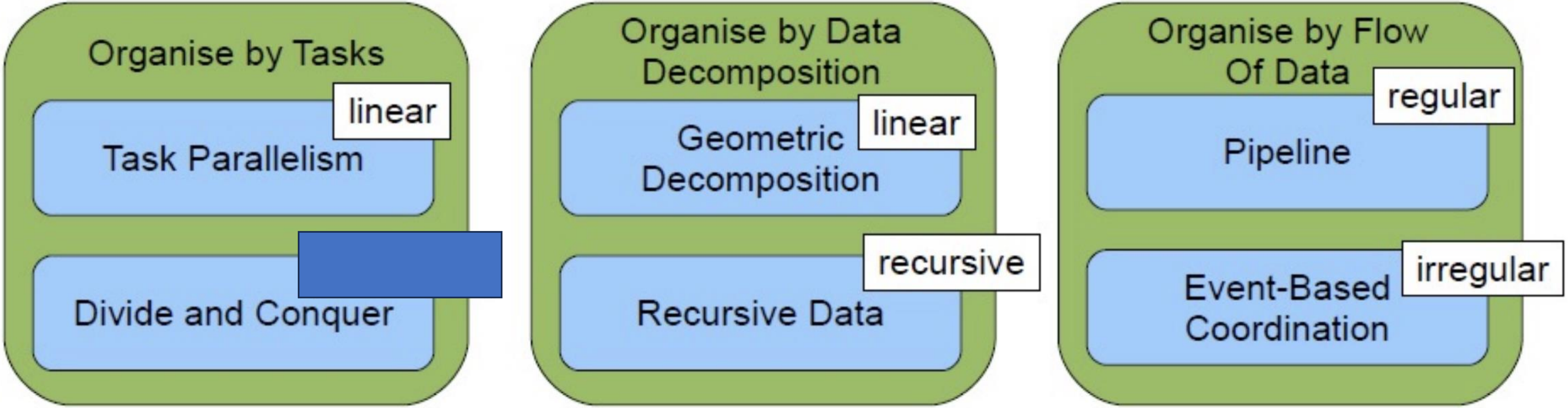


Figure based on ref [3]

Patterns for Supporting Structures

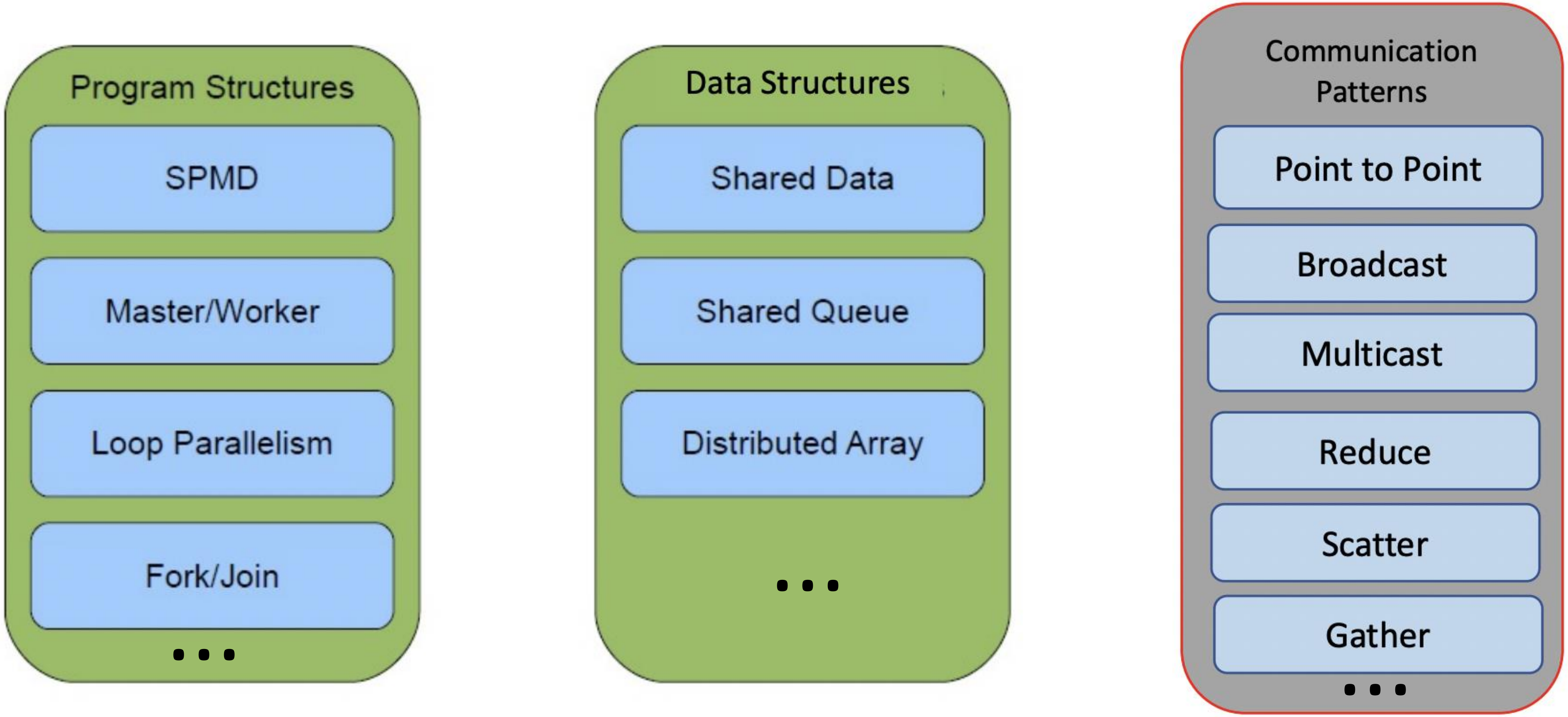


Figure based on ref [3]

Intro to Architectures in Concurrent & Dist’d Systems: S/w V System Architectures

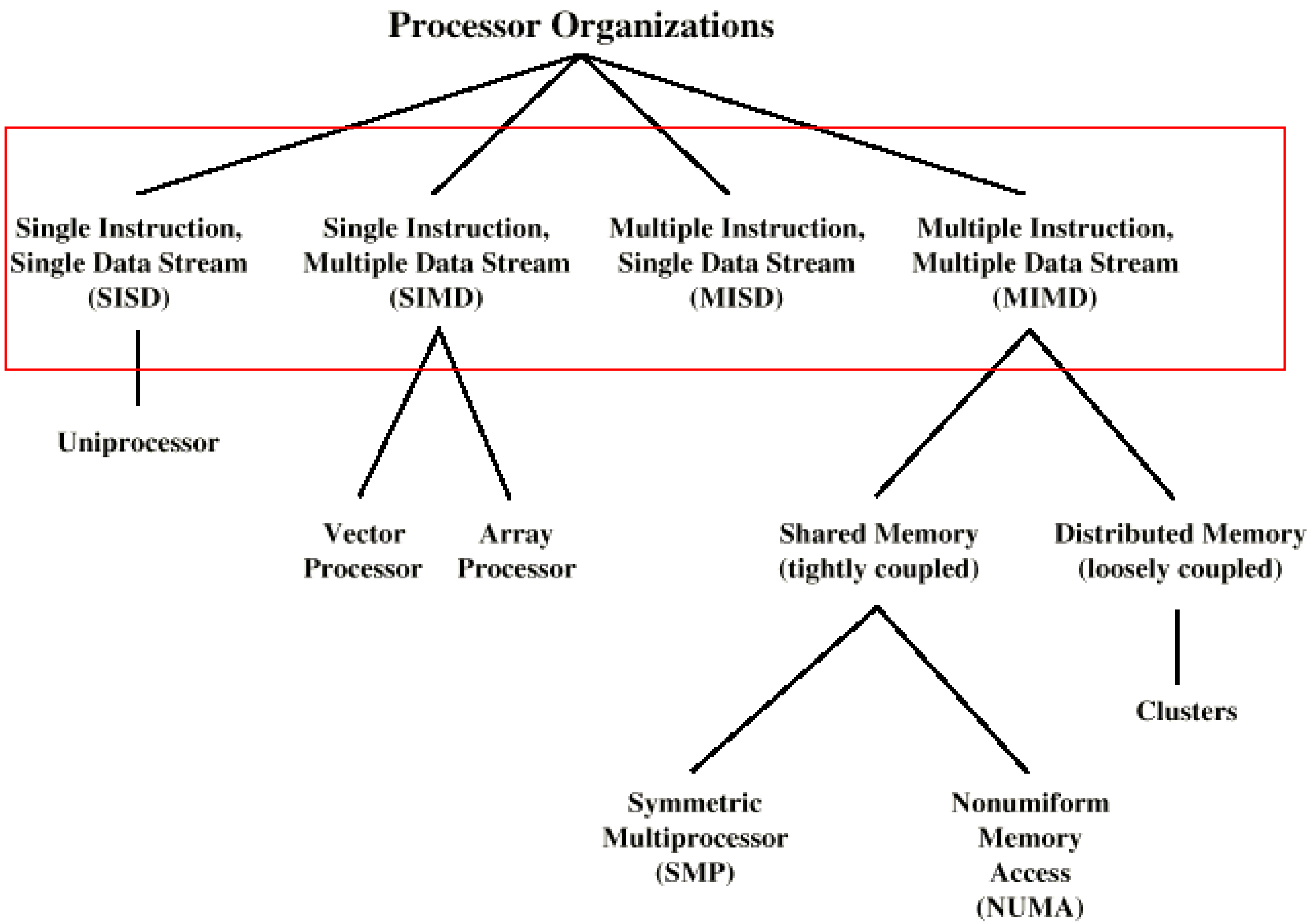
- Organizating concurrent & distributed systems is mostly about the software components making up the system.
- These *software architectures* (aka *Programming Models*) dictate the organization & interaction of the various s/w components.
- ... But the actual realization of a system requires instantiating and placing software components on real machines.
- There are many different choices that can be made in doing so.
- The final instantiation of a software architecture is referred to as a *system architecture* (aka *Machine Model*).

Parallel Programming Model

- *Definition:* Programming model comprises languages & libraries that create an abstract view of the machine.
  - Control
    - What orderings exist between operations?
    - How do different threads of control synchronize?
  - Data
    - What data is private vs. shared?
    - How is logically shared data accessed or communicated?
  - Synchronization
    - What operations can be used to coordinate parallelism?
    - What are the atomic (indivisible) operations?
  - …

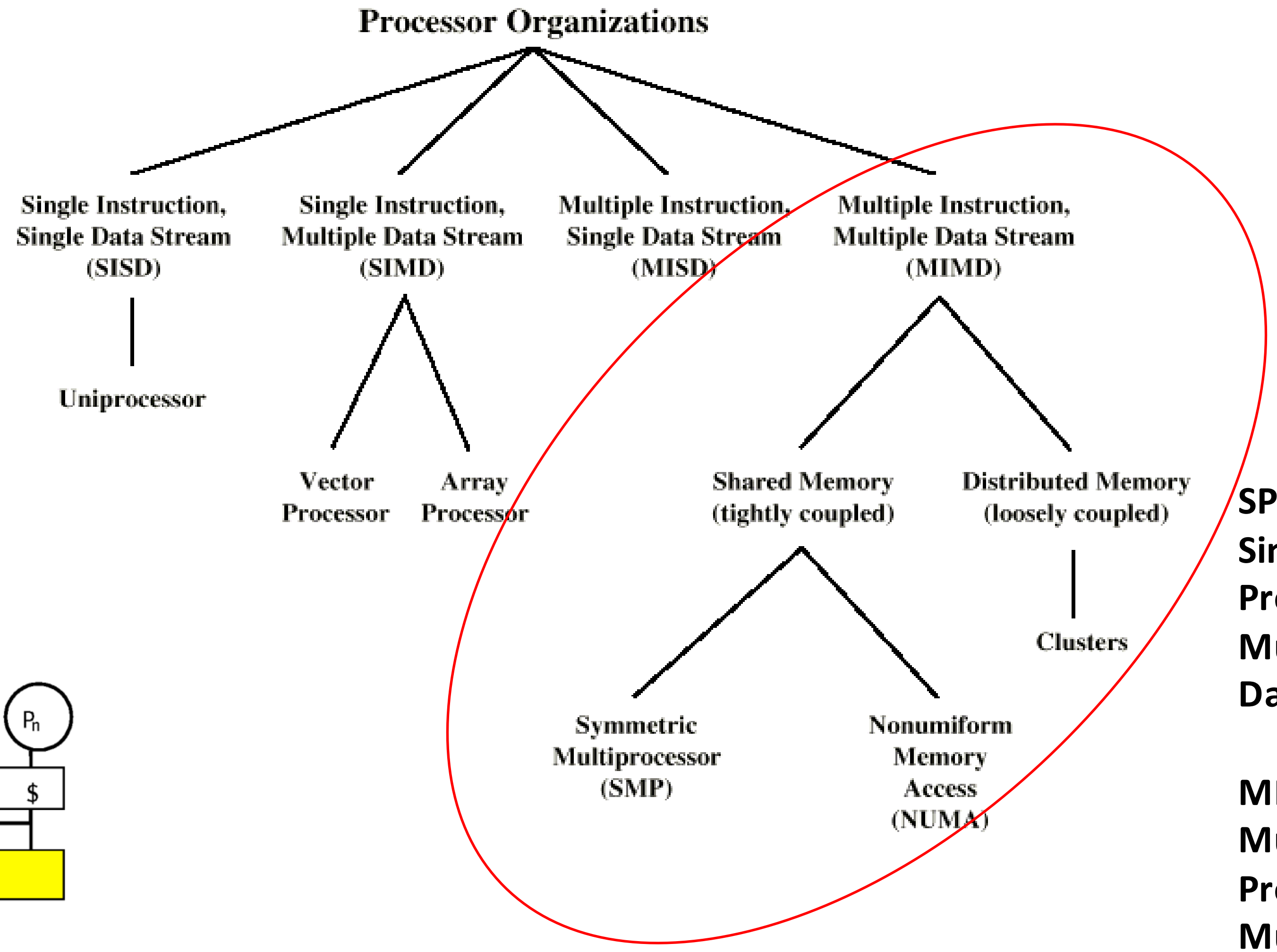
Concurrent Architecture Taxonomies

- Michael Flynn in 1966 classified machines into a taxonomy by the number of instruction and data streams
- We examine these from standpoint of concurrent architectures

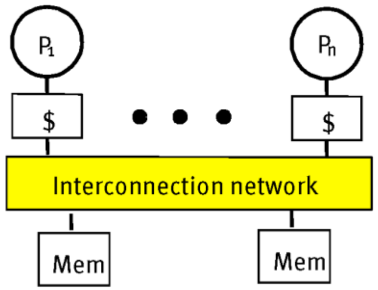


More on MIMD

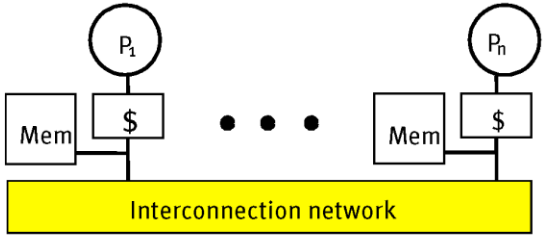
- *MIMD*
- General purpose processor
- Each can process all instructions necessary.
- Further classified by method of processor communication:
    - *Tight Coupling*
    - *Loose Coupling*



Scale



Centralized Memory



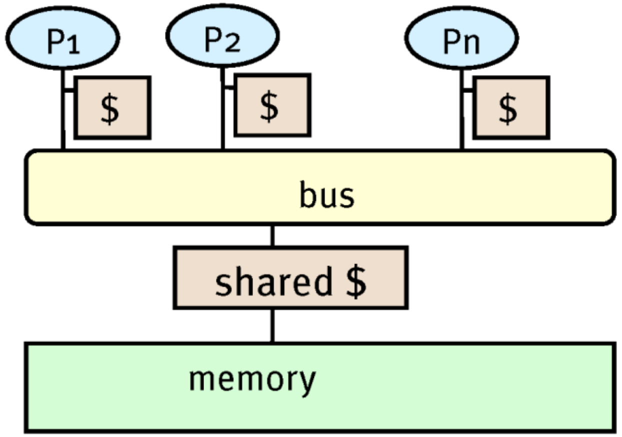
Distributed Memory

SPMD: Single Program Multiple Data

MPMD: Multiple Program Multiple Data

Concurrent Architectures

- *Machine Model #1: Shared Memory*
- Processors all connected to a large shared memory
  - Typically Symmetric Multiprocessors (SMPs) / Multicore chips
  - But
    - Shared memory can give issues with *race conditions*
    - Can be fixed by adding synchronisation (e.g. a lock), at performance cost



Note: $ = cache

Programming Models

Java Threads
OpenMP
POSIX Threads
C# TPL
multiprocessing library
...

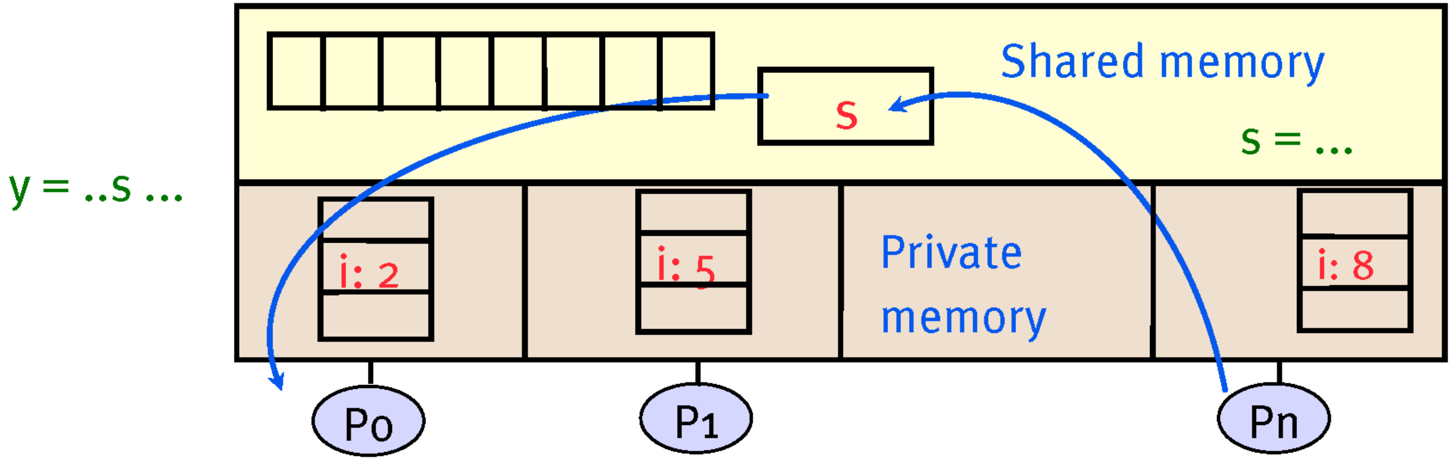
- *Programming Model # 1: Shared Memory*

  Program is a collection of threads of control.

  - Each thread has set of private variables, e.g., local stack variables

  & set of shared variables, e.g., static variables

  - Implicit comms between threads involves writing/reading shared variables
  - Threads coordinate by synchronizing on shared variables



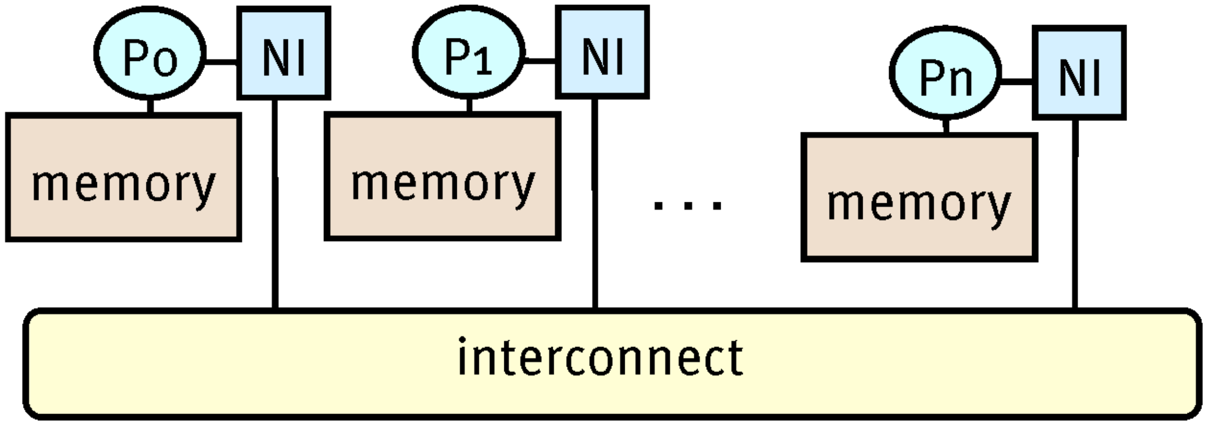
...

Concurrent Architectures

- *Machine Model #2: Distributed Memory*

Processors have own memory but typically fast interconnect

- Each processor has its own memory and cache but cannot directly access another processor's memory.
- Each "node" has a Network Interface (NI) for all communication and synchronization.
- Example: Beowolf Cluster



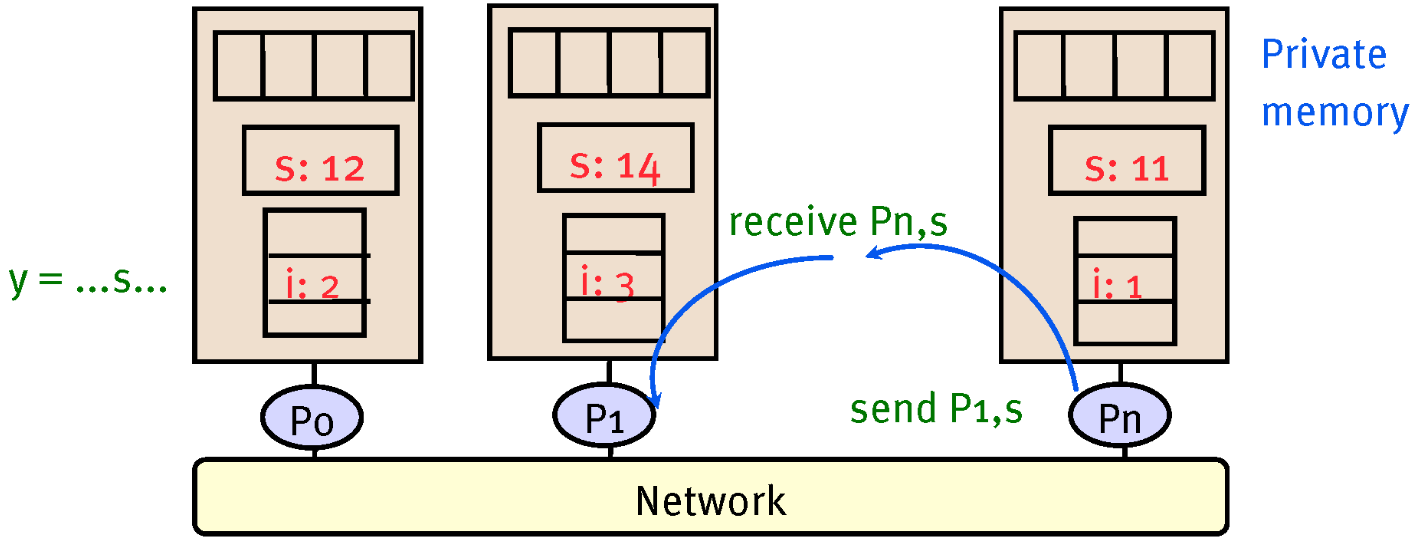
Programming Models

OpenMPI
RabbitMQ
ActiveMQ
Apache Kafka
…

- *Programming Model # 2: Message Passing*

  Program consists of a collection of named processes.

  - Usually fixed at program startup time
  - Thread of control plus local address space—NO shared data.
  - Data is partitioned over processes.

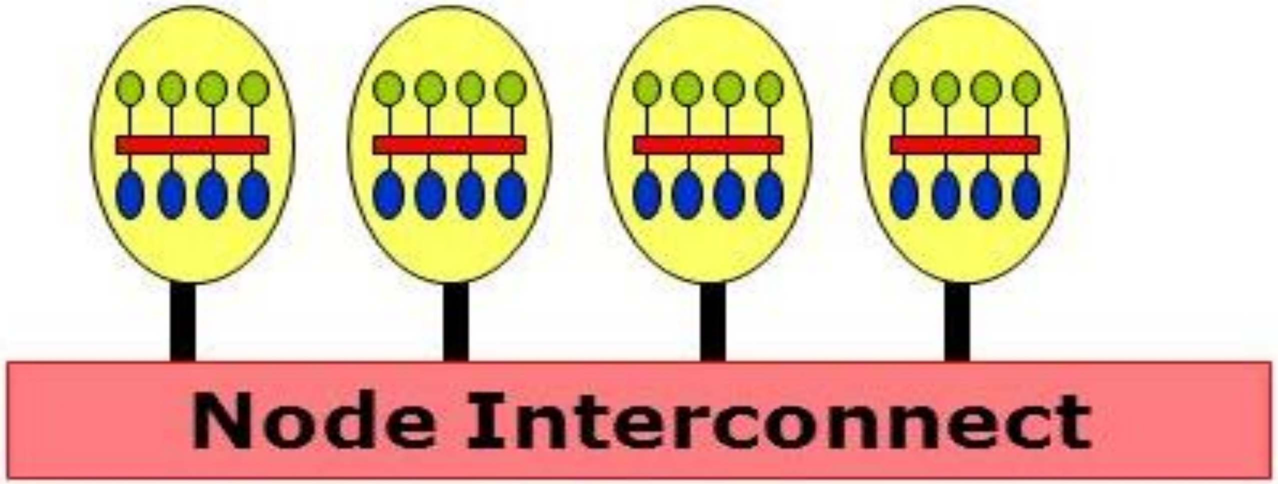


Concurrent Architectures

- *Machine Model #3: Clusters*

Used for computation-intensive purposes, (Vs for IO operations such as web service or DBs.)

- Emerged due to trends e.g. low-cost cores, high speed n/ws & s/w for HP distributed computing.
- Wide applicability from small biz clusters to fastest supercomputers



Programming Models

- *Programming Model # 3: Hybrids*

Need to run "same/similar computation" on many nodes very fast

- Common model: Hybrid MPI + OpenMP
  - Each SMP node = 1 MPI process, w MPI comm on node interconnect
  - OpenMP inside of each SMP node
- Maybe gives the highest performance?
  - Advantage: Could be good for heavyweight comps & lightweight threads within a node
  - Disadvantages:
    - Difficult to start with OpenMP and modify for MPI
    - Difficult to program, debug, modify and maintain
    - Generally, cannot do MPI calls within OpenMP parallel regions
    - Requires experience to use this mixed prog model

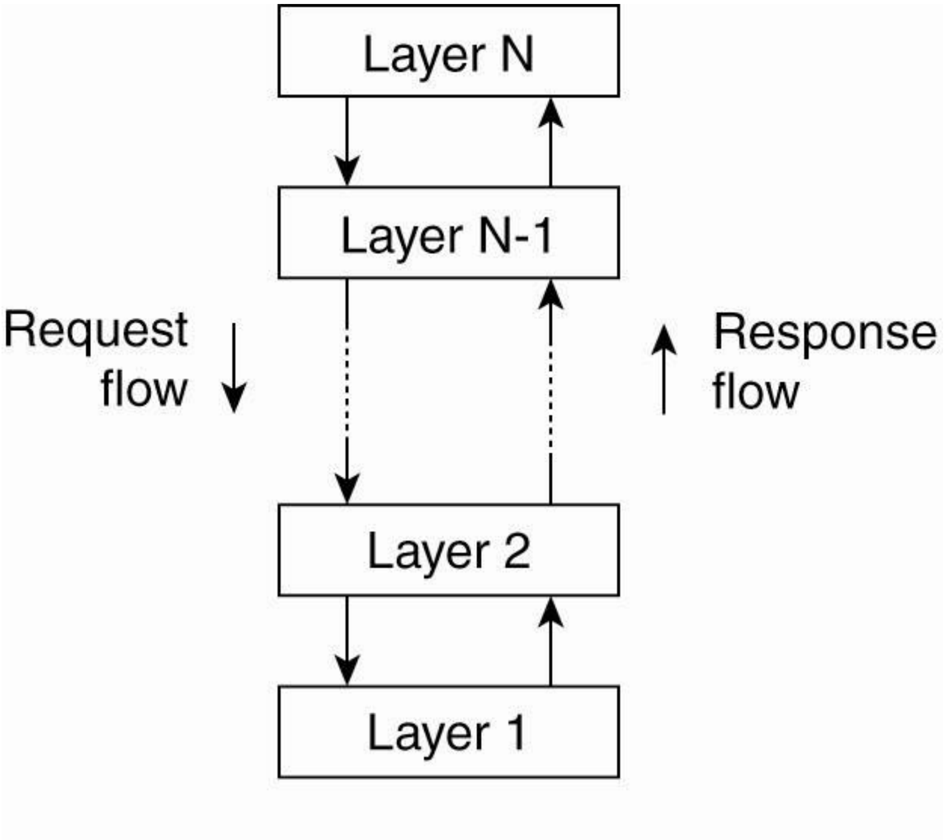
Software Architectures for Distributed Systems

Architectures for *Distributed* Systems

- *Introduction*
- Common: traditional *centralized distributed systems* architectures - where 1 server implements most s/w components (the functionality)
- Remote clients access the server using simple communication means.
- But we also have ... *decentralized* architectures in which machines more or less play equal roles, or .... hybrid organizations.
- Adopting such a layer is an important architectural decision: main purpose is *distribution transparency*.
- However, trade-offs must be made to have transparency, leading to various techniques to make middleware adaptive.

Distributed Architectural Styles

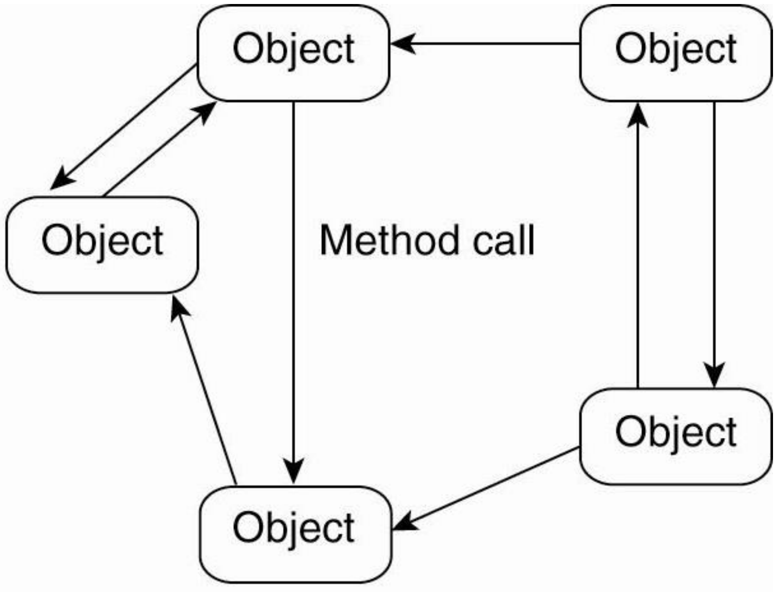
- *#1 Layered Architectures*
  - Basic idea is simple: organize components in layers
  - Component at layer N can call those at underlying layer $N-1$ (but not vice versa)
  - This so-called *application layering* is shown in the diagram
  - A key observation is that control generally flows from layer to layer
  - E.g. requests go down the hierarchy whereas the results flow upward.
  - This model has been widely adopted by the networking community



(a)

Distributed Architectural Styles

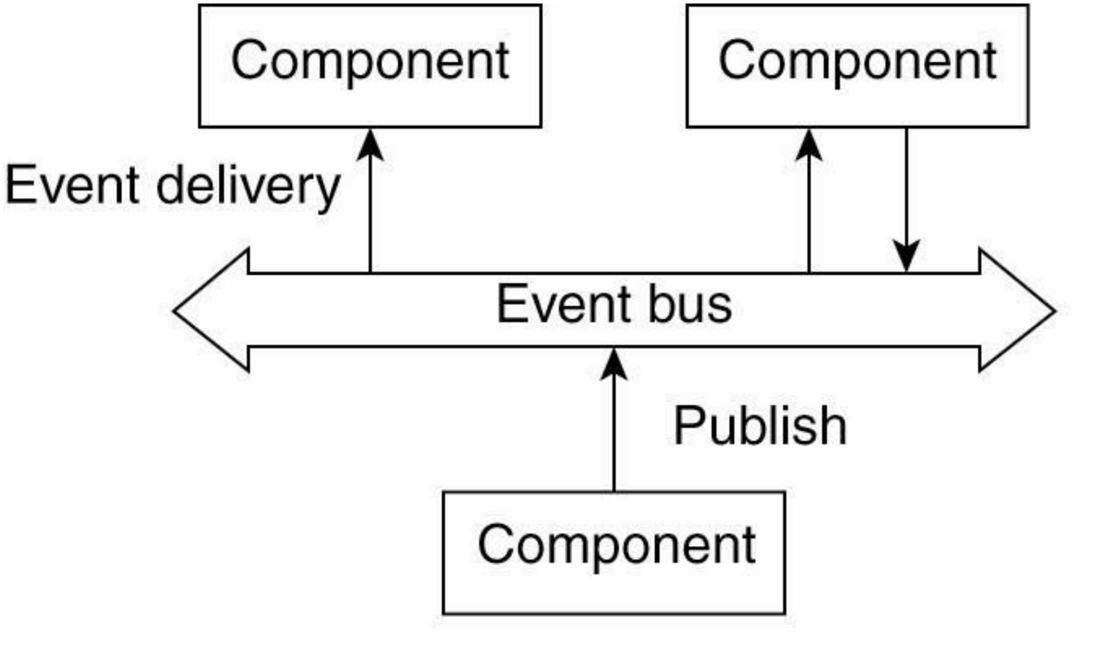
- *#2 Object-Based Architectures*
  - A far looser organization is followed in object-based architectures,
  - These components are connected through a *(remote) procedure call* mechanism.
  - Layered & object-based architectures are one style for large s/w systems



(b)

Distributed Architectural Styles

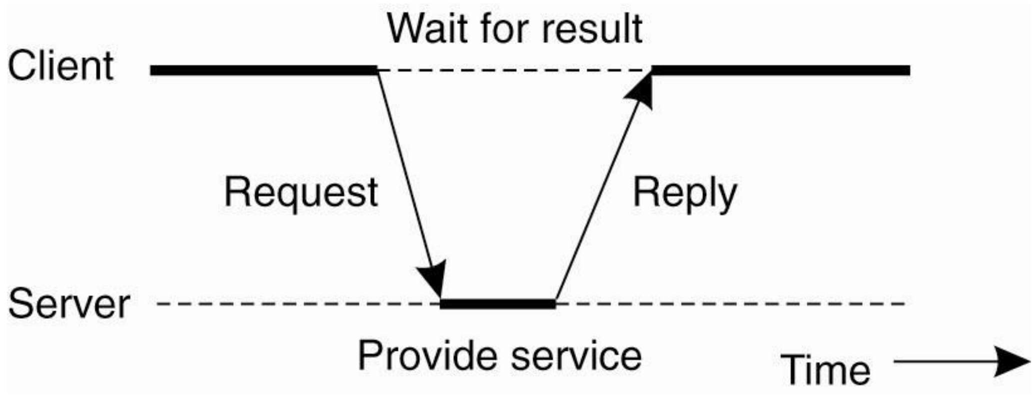
- *#3 Event-Based Architectures*
  - Processes communicate thro event propagation, optionally with data.
  - For DS, event propagation usually associated with so-called *publish/subscribe.*
  - Idea: processes publish events & m/w ensures only subscribed processes receive them.
  - The main advantage of such systems is *loose coupling* of processes
  - Needn't refer to each other explicitly.



System Architectures: Centralized Architectures

- *Basic Client–Server Model Characteristics*
  - There are processes offering services (*servers*)
  - There are processes that use services (*clients*)
  - Clients and servers can be on different machines

  - Clients follow request/reply model with respect to using services
  - Thinking in terms of Clients requesting Services from Servers aids in the understanding of Distributed Systems

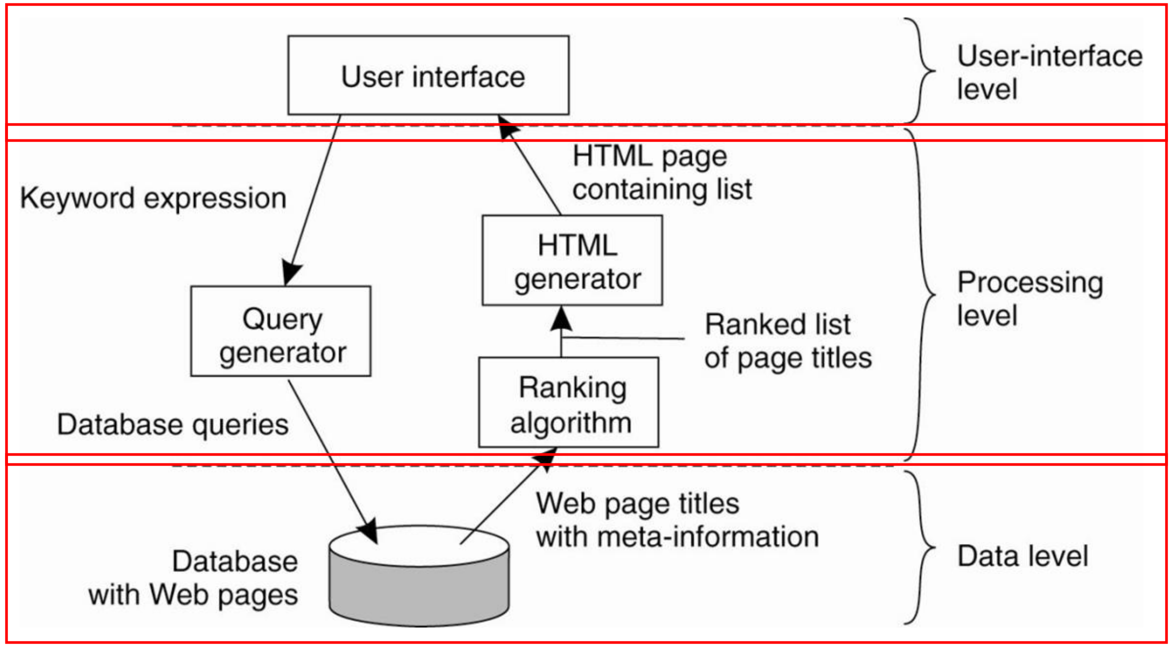


Plain Ol' Client-Server

System Architectures: Centralized Architectures

- *Application Layering: Traditional three-layered view*
  1. *User-interface layer* contains units for an application's user interface
  2. *Processing layer* contains the functions of an application, i.e. no specific data
  3. *Data layer* contains data client wants to process thro application components
  - Found in many distributed info systems, using traditional DB technology and accompanying applications.

Typical Web Browser Architecture



Core Functionality Transforming user keywords into DB queries & ranking results on return

System Architectures: Decentralized Architectures

- Example ...

- In the last couple of years there has been a tremendous growth in such *peer-to-peer (P2P)* systems:
  - *Structured P2P*: nodes are organized following a specific distributed data structure (usually a **Distributed Hash Table**)
  - *Unstructured P2P*: nodes have randomly selected neighbours. Each node has a list of neighbours which is constructed in a random way.
  - *Hybrid P2P*: some nodes are appointed special functions in a well-organized fashion

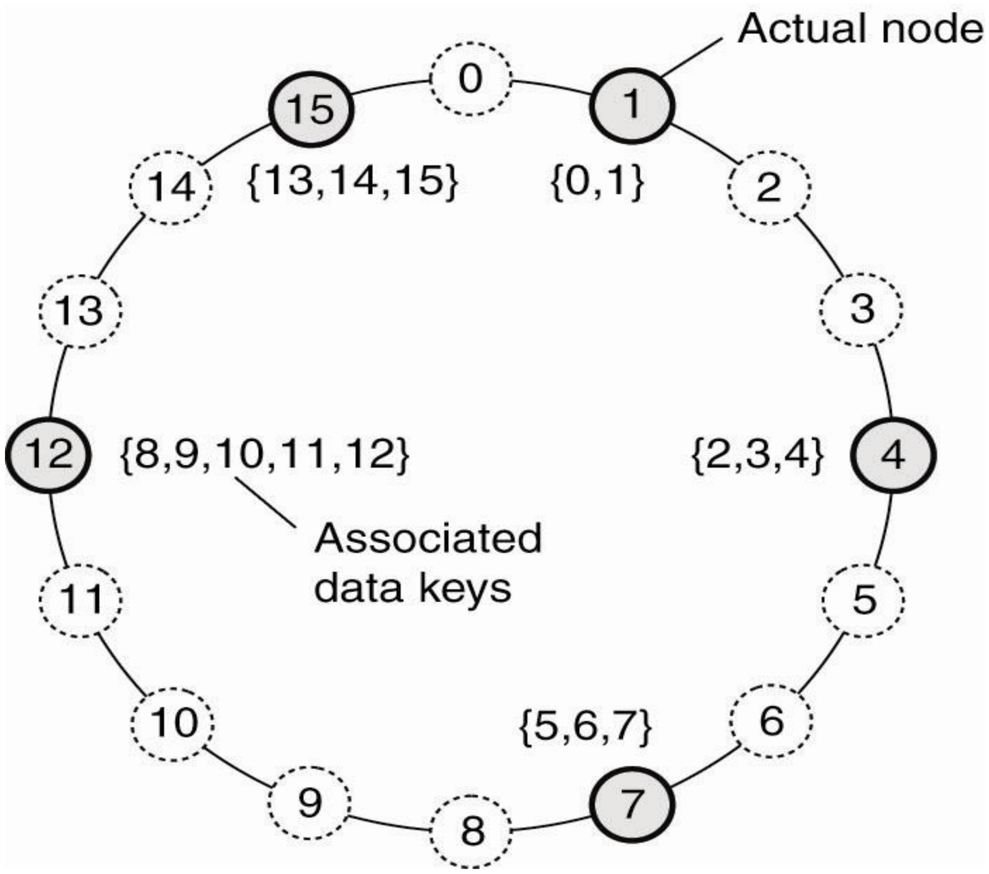
Decentralized Architectures (/2): Structured P2P Systems

- In virtually all cases for Structured P2P, have *overlay networks*
  - Data is routed over connections setup between nodes.

- As processes can't communicate directly with others, available communication channel must be used (a.k.a. *Application-level Multicasting*)
  - ALM is offered by middleware (in contrast to low-level TCP/IP Multicasting)
  - Basic idea is to organize nodes in a structured overlay n/w such as a logical ring.
  - Specific nodes are made responsible for services based only on their ID.
- For example,
  - Random key is assigned to a data item from a large (eg 128 bit) identifier space
  - The system provides an operation e.g. *LOOKUP(key)* that will efficiently route the lookup request to the associated node.
  - When the key is returned, the network address of node responsible for the data (known as the *successor*) item stored is returned.

Decentralized Architectures (/3): Structured P2P Systems: Chord Case Study

- *Details of Chord Algorithm*
  1. Assign random key (*m-bit identifier*) to data item & random number (*m-bit identifier*) to node in system,
  2. Implement an efficient & deterministic system to map a data item to a node based on some distance metric,
  3. This means that data item should physically be as close to node as possible
  4. *LOOKUP(key)* ≡ returning network address of node responsible for that data item,
  5. Do this by routing a request for the data item to responsible node (*successor*).
  6. Node with key k falls under the jurisdiction of node with smallest $id \geq k$
  7. This process of looking up node's name (& any info stored there) called *name resolution*



Decentralized Architectures (/3): Structured P2P Systems: Chord Case Study

- ***Principle of Operation of Chord***
  - Membership management in Chord doesn't follow a logical organization of nodes in a ring as shown in diagram (previous).
  - Lookups on keys can be done in $O(\log_2 N)$ steps.
  - Each node p maintains a finger table $FT_p[i]$ with at most m entries:
$$FT_p[i] = succ(p + 2^{i-1})$$
  - Note: $FT_p[i]$ points to the first node succeeding p by at least 2^{i-1}
  - This is because Chord is an algorithm based on binary (will look at higher order algorithms later)
  - To look up a key k, node p forwards the request to node with index j satisfying
$$q = FT_p[j] \leq k < FT_p[j+1]$$
  - If $p < k < FT_p[1]$ the request is also forwarded to $FT_p[1]$

Decentralized Architectures (/4): Structured P2P Systems: Chord Case Study

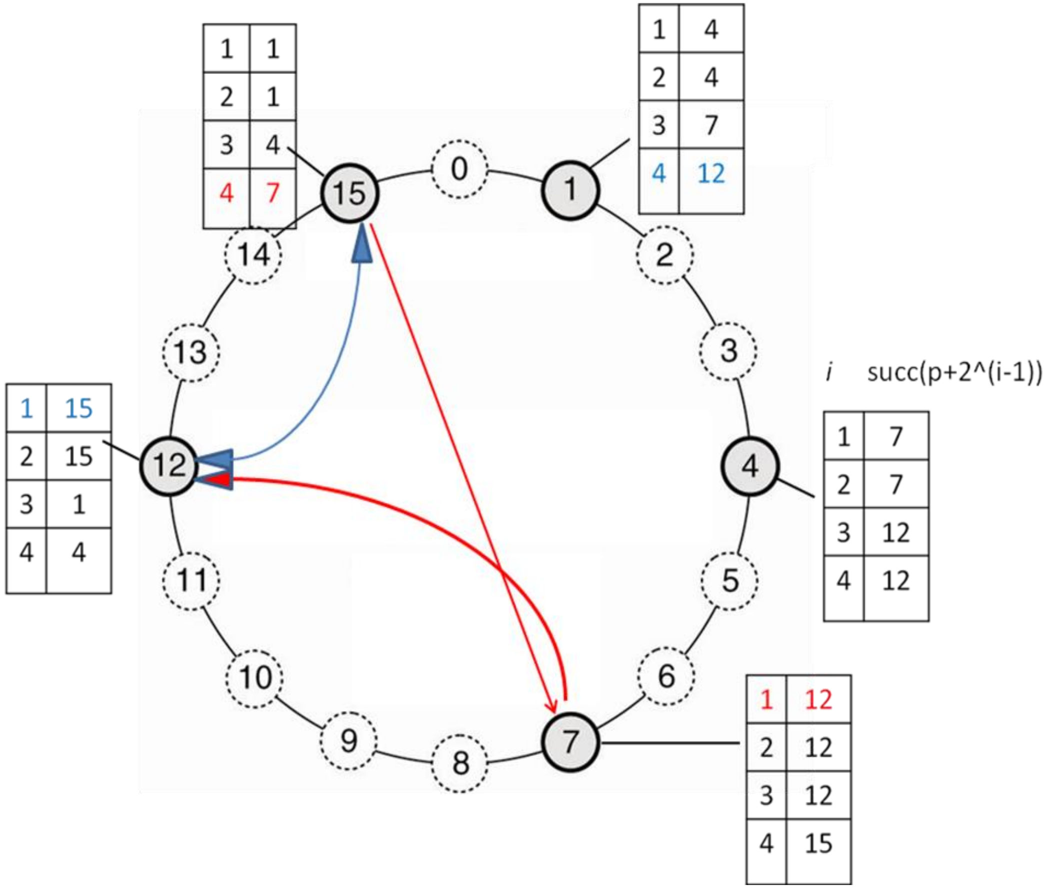
- *Building Finger Tables in Chord*

Some calculations for Finger tables in the diagram:

$$FT_1[1] = succ(1 + 2^0) = succ(2) = 4$$
$$FT_1[2] = succ(1 + 2^1) = succ(3) = 4$$
$$FT_1[3] = succ(1 + 2^2) = succ(5) = 7$$
$$FT_1[4] = succ(1 + 2^3) = succ(9) = 12$$

$$FT_4[1] = succ(4 + 2^0) = succ(5) = 7$$
$$FT_4[2] = succ(4 + 2^1) = succ(6) = 7$$
$$FT_4[3] = succ(4 + 2^2) = succ(8) = 12$$
$$FT_4[4] = succ(4 + 2^3) = succ(12) = 12$$

$$FT_{15}[1] = succ(15 + 2^0) = succ(16) = succ(0) = 1$$
$$FT_{15}[2] = succ(15 + 2^1) = succ(17) = succ(1) = 1$$
$$FT_{15}[3] = succ(15 + 2^2) = succ(19) = succ(3) = 4$$
$$FT_{15}[4] = succ(15 + 2^3) = succ(23) = succ(7) = 7$$



Finger table of node 15:

1	1
2	1
3	4
4	7

Finger table of node 1:

1	4
2	4
3	7
4	12

Finger table of node 12:

1	15
2	15
3	1
4	4

Finger table of node 4:

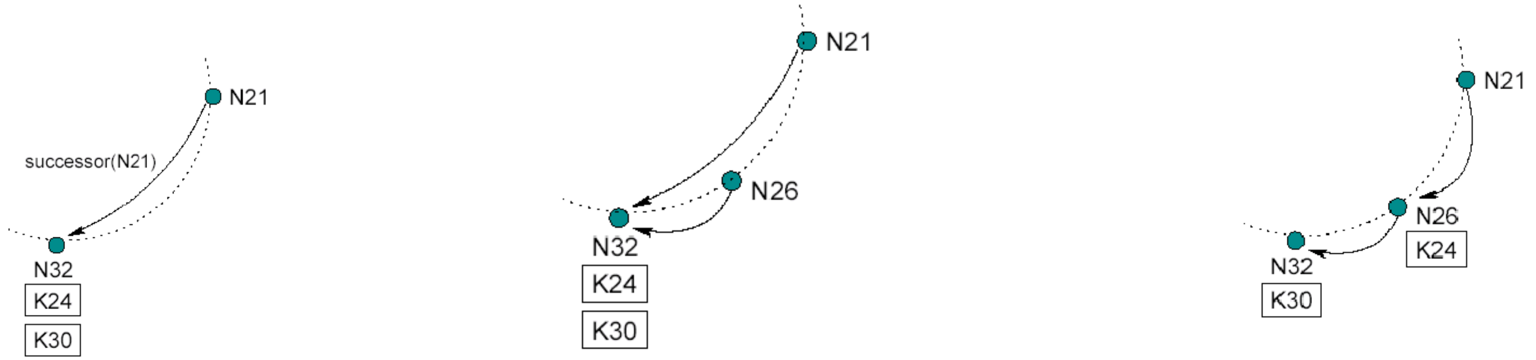
1	7
2	7
3	12
4	12

Finger table of node 7:

1	12
2	12
3	12
4	15

Decentralized Architectures (/5): Structured P2P Systems: Chord Case Study

- *Principle of Joining a System in Chord*
  - Node wanting to join system starts by generating random identifier $id = 26$ (or hashes to get id)
    - Then node simply contacts an arbitrary node & does a lookup on id,
    - Returns address of $succ(id) = 32,$ node responsible for looking after id
    - Next, node simply contacts $succ(id)$ & it's predecessor & inserts self in ring
    - This consists of updating the finger tables.
    - Insertion also yields that each data item whose key is now associated with node id, is transferred from $succ(id)$.
  - Chord scheme requires that each node also stores info on its predecessor(s) – *r* nodes forward and back.



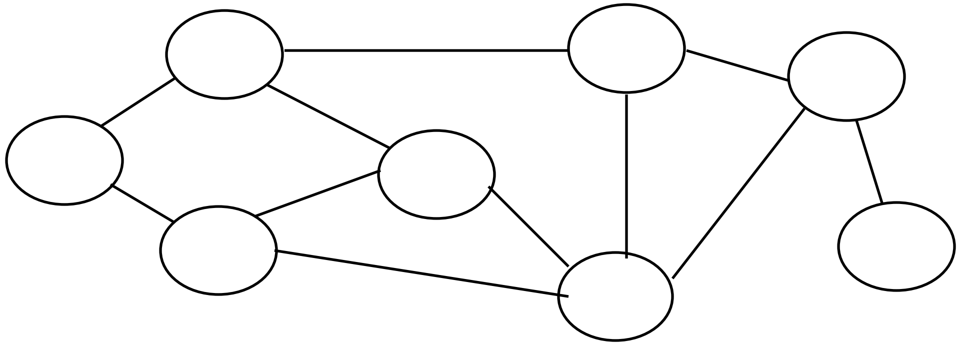
Decentralized Architectures (/6): Structured P2P Systems: Chord Case Study

- *Problems in Chord*
  - Logical organization of overlay nodes may lead to erratic msg transfers in underlying Internet: node k, node $\mathrm{succ}(k)$ may be far apart.
    - *Topology-aware node assignment:*
      - When assigning an ID to a node, make sure that nodes close in the ID space are also close in the network.
      - Could be difficult …
    - *Proximity routing:*
      - Maintain more than one possible successor, and forward to the closest.
      - Example: in Chord $FT_p[i]$ points to first node in $[p + 2^{i-1}, p + 2^i - 1]$.
      - Node p can also store pointers to other nodes in the interval.
    - *Proximity neighbour selection:*
      - When there is a choice of selecting who your neighbour will be (not in Chord), pick the closest one.

(more) Algorithms for Distributed Systems

Asynchronous Heartbeat Algorithms

- Heartbeat algorithms are a typical type of process interaction between *peer* processes connected by channels.
- Called heartbeat algorithms as each process' actions akin to a heart;
  - first expanding, sending information out;
  - then contracting, gathering new information in.
- This behaviour is repeated for several iterations.
- An example of an asynchronous heartbeat algorithm is the algorithm for computing the topology of a network.

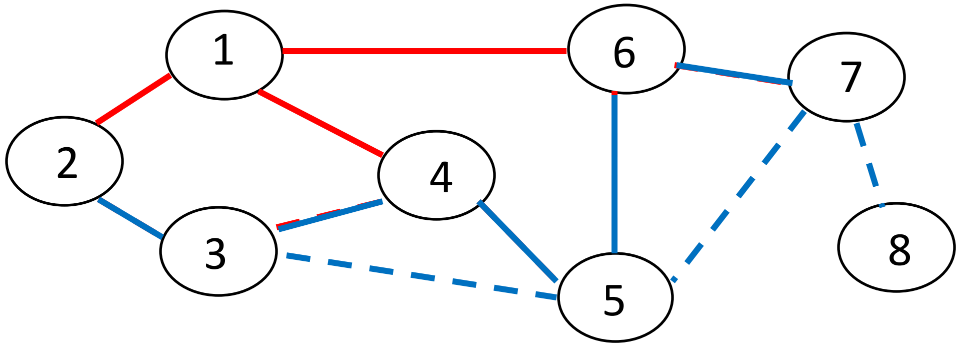


Ex 2: Asynchronous Heartbeat Algorithm for Computing Network Topology

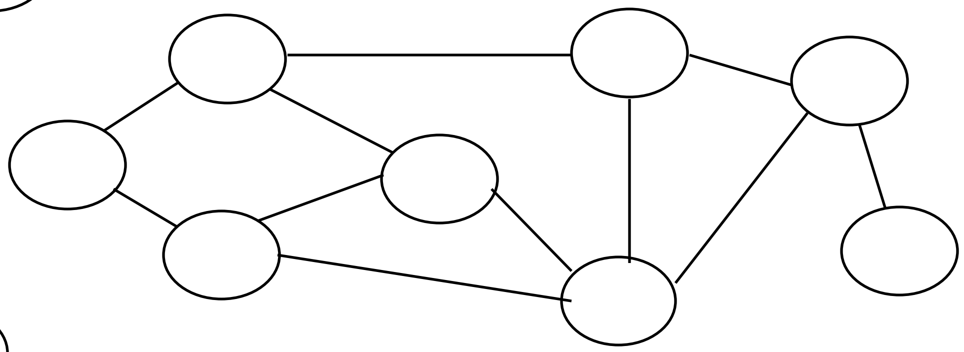
- Each node has a processor & initially only knows about the other nodes to which it is directly connected.
- Algorithm goal: each node has to determine the overall n/w topology.
- Two phases of the heartbeat algorithm:
    1. transmit current knowledge of network to all neighbours, and
    2. receive the neighbours' knowledge of the network.
- After iteration 1. a node is aware of nodes connected to its neighbours, (i.e. within two links of itself.)
- After 2 it has sent (to neighbours) all nodes within 2 links of itself; & got info about all nodes within 2 links of its neighbours, (i.e. 3 links of itself).
- In general, after i iterations knows about all nodes within $(i + 1)$ links of itself.

Ex 2: Asynchronous Heartbeat Algorithm for Computing Network Topology: Algorithm Operation

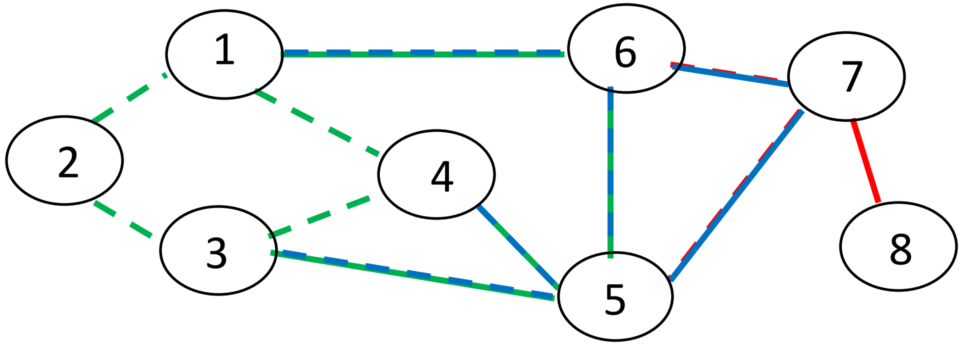
Firstly from Node 1's Point of View



....and Node 1 is done!



Next from Node 8's Point of View



....and Node 8 is done!

Ex 2: Asynchronous Heartbeat Algorithm for Network Topology(/2)

- How many iterations are necessary? (when do we know when to stop)
- As network is connected, every node has at least one neighbour.
- If known network topology at any given stage is stored in an n x n matrix **`top`** where

  `top`$[i, j]$ = true if a link exists between node i and j,

  then a node knows the complete network topology when every row in **`top`**$[i, j]$ has at least one true value.

- At this point the node must perform one more iteration of the algorithm
- This is to transmit any new information received from one neighbour to its other neighbours.

Ex 2: Asynchronous Heartbeat Algorithm for Network Topology(/3)

- If m is the max number of neighbours any node has, & D the n/w diameter[1], then number of messages exchanged must be less than $2n \times m \times (D + 1)$.
- A centralised algorithm, in which `top` was held in memory shared by each process, requires only $2n$ messages.
- If m & D are small relative to n then relatively few extra messages.
- Heartbeat algorithm requires more messages, but these can be exchanged in parallel.

[1] i.e. the max. value of the minimum number of links between any two nodes

Ex 2: Asynchronous Heartbeat Algorithm for Network Topology(/4)

- All heartbeat algorithms have the same basic structure: send messages to neighbours, and then receive messages from them.

- A major difference between the different algorithms is termination.
  - If the termination condition can be determined locally, as above, then each process can terminate itself.
  - If however, condition depends on a global condition, each process must do a worst-case number of iterations,

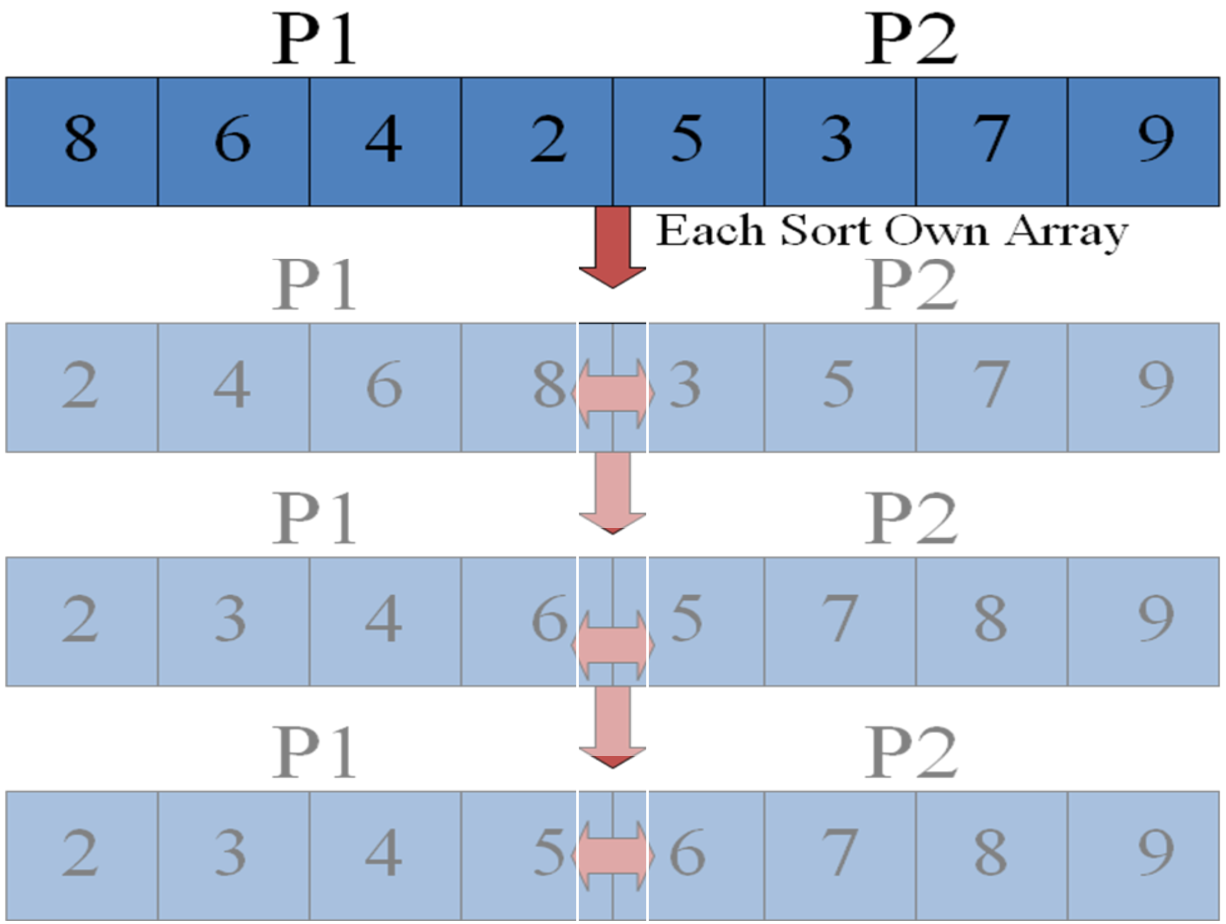
  - we could... Alternative is to communicate with a central controller monitoring the global state of the algorithm,
  - This then issues a termination message to each process when required.

Ex 3: Synchronous Heartbeat Algorithm: Parallel Sorting

- To sort an array of n values in parallel using a synchronous heartbeat algorithm, must partition n values among processes.
- Assume that we have 2 processes, P_1 and P_2, and that n is even.
- Each process initially has $n/2$ values and sorts these values into non descending order, using a sequential sort algorithm.
- Then, each iteration, P_1 swaps its largest value with P_2's smallest
- Then both processes place new values into correct place in their own sorted list of numbers.
- Note: as both sending & receiving block in synchronous message passing, P_1 and P_2 can't execute send, receive primitives in same order (as could in asynchronous case).

Ex 3: Synchronous Heartbeat Algorithm: Parallel Sorting: Algorithm Operation (/2)

- Demonstration of Odd/Even Sort for 2 Processes:



Ex 3: Synchronous Heartbeat Algorithm: Parallel Sorting: (/3)

- Can extend this to k processes by initially dividing array to give each process n/k values to sort using a sequential algorithm.
- Then sort n elements by repeated applications of the two process compare and exchange algorithm.
- The algorithm for exchange sort on n processes can be terminated in many ways; two of which are:

1. Have a separate controller process who is informed by each process, each round, if they have modified their n/k values.
    - If no process has modified its list then the central controller replies with a message to terminate.
    - This adds an extra $2k$ messages overhead per round.
2. Execute enough iterations to guarantee that the list will be sorted ...

Example 3(a): Exchange Sort: Algorithm Operation

- Demonstration of *Exchange Sort* for k Processes: