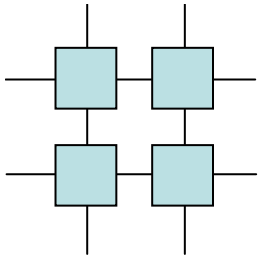
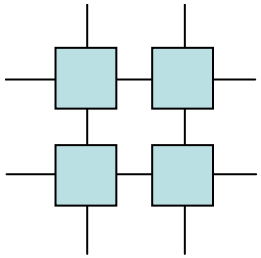


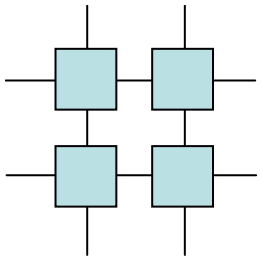
Distributed Shared Memory

History, fundamentals and a few examples



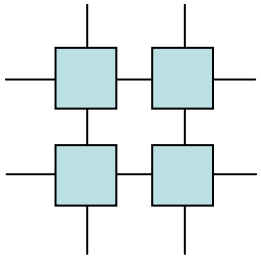
Coming up

- **The Purpose of DSM Research**
- **Distributed Shared Memory Models**
- **Distributed Shared Memory Timeline**
- **Three example DSM Systems**



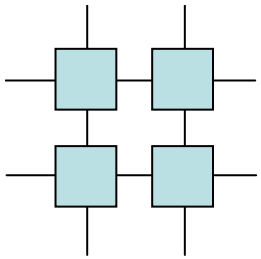
The Purpose of DSM Research

- **Building less expensive parallel machines**
- **Building larger parallel machines**
- **Eliminating the programming difficulty of MPP and Cluster architectures**
- **Generally break new ground:**
  - **New network architectures and algorithms**
  - **New compiler techniques**
  - **Better understanding of performance in distributed systems**



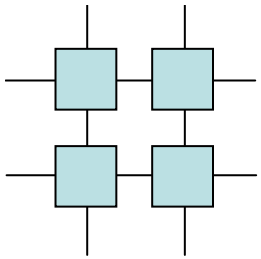
Distributed Shared Memory Models

- **Object based DSM**
- **Variable based DSM**
- **Structured DSM**
- **Page based DSM**
- **Hardware supported DSM**



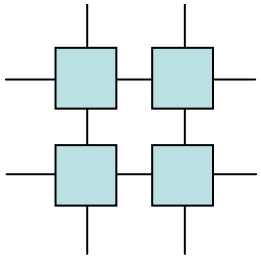
Object based DSM

- **Probably the simplest way to implement DSM**
- **Shared data must be encapsulated in an object**
- **Shared data may only be accessed via the methods in the object**
- **Possible distribution models are:**
  - **No migration**
  - **Demand migration**
  - **Replication**
- **Examples of Object based DSM systems are:**
  - **Shasta**
  - **Orca**
  - **Emerald**



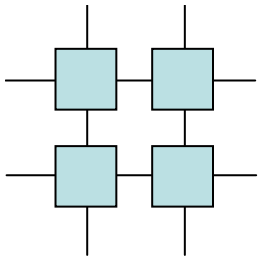
Variable based DSM

- **Delivers the lowest distribution granularity**
- **Closely integrated in the compiler**
- **May be hardware supported**
- **Possible distribution models are:**
  - **No migration**
  - **Demand migration**
  - **Replication**
- **Variable based DSM systems have never really matured into systems**



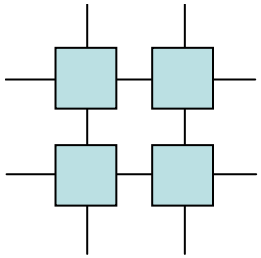
Structured DSM

- **Common denominator for a set of slightly similar DSM models**
- **Often tuple based**
- **May be implemented without hardware or compiler support**
- **Distribution is usually based on migration/read replication**
- **Examples of Structured DSM systems are:**
  - **Linda**
  - **Global Arrays**
  - **PastSet**



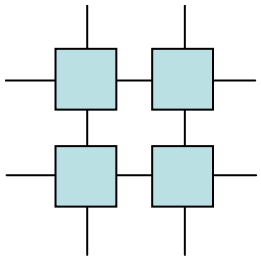
Page based DSM

- **Emulates a standard symmetrical shared memory multi processor**
- **Always hardware supported to some extend**
  - **May use customized hardware**
  - **May rely only on the MMU**
- **Usually independent of compiler, but may require a special compiler for optimal performance**



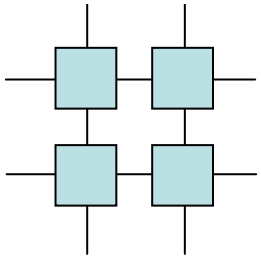
Page based DSM

- **Distribution methods are:**
  - **Migration**
  - **Replication**
- **Examples of Page based DSM systems are:**
  - **Ivy**
  - **Threadmarks**
  - **CVM**
  - **Shrimp-2 SVM**

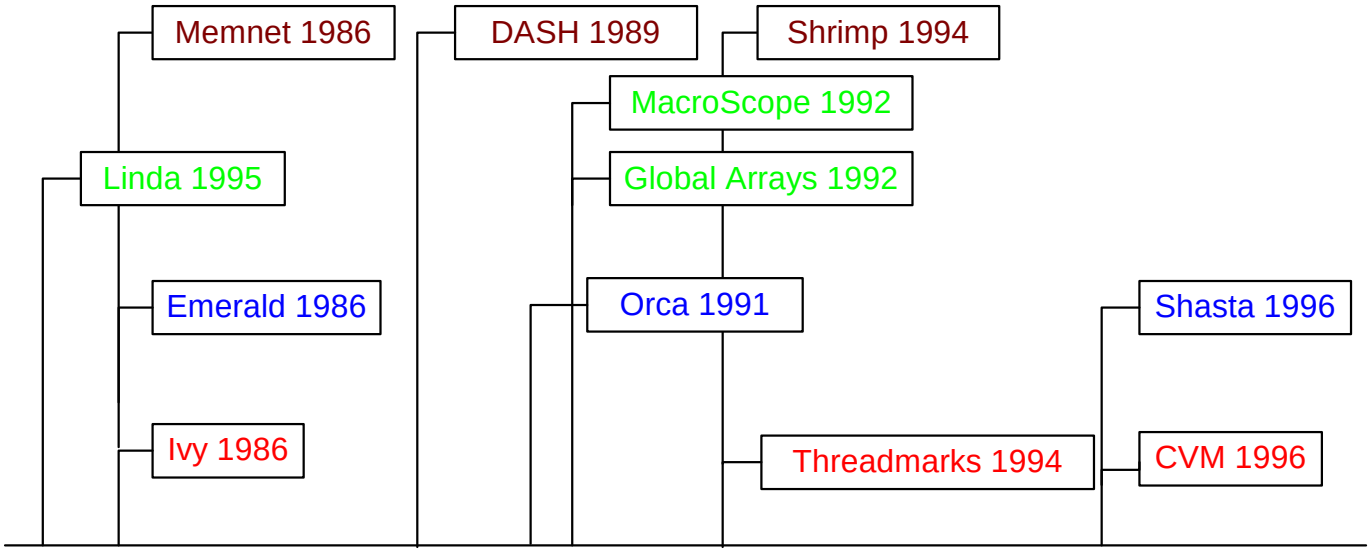


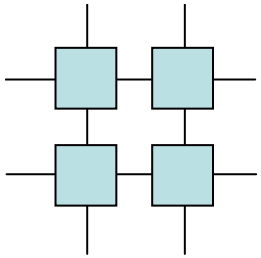
Hardware supported DSM

- **Uses hardware to eliminate software overhead**
- **May be hidden even from the operating system**
- **Usually provides sequential consistency**
- **May limit the size of the DSM system**
- **Examples of hardware based DSM systems are:**
  - **Shrimp**
  - **Memnet**
  - **DASH**
  - **Cray T3 Series**
  - **SGI Origin 2000**



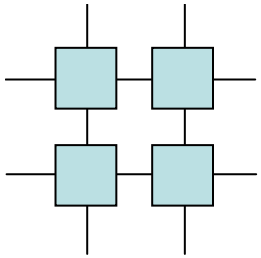
Distributed Shared Memory Timeline





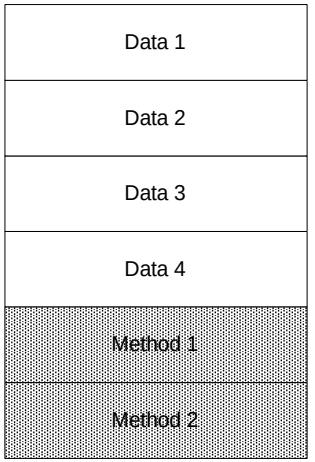
Three example DSM systems

- **Orca**
  Object based language and compiler sensitive system
- **Linda**
  Language independent structured memory DSM system
- **IVY**
  Page based system

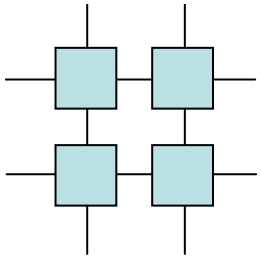


Orca

- **Three tier system**
  - **Language**
  - **Compiler**
  - **Runtime system**

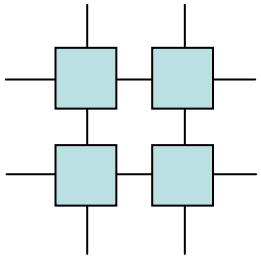


- **Closely associated with Amoeba**
- **Not fully object orientated but rather object based**



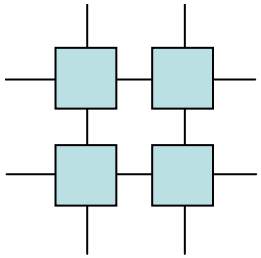
Orca

- **Claims to be be Modula-2 based but behaves more like Ada**
- **No pointers available**
- **Includes both remote objects and object replication and pseudo migration**
- **Efficiency is highly dependent of a physical broadcast medium - or well implemented multicast.**



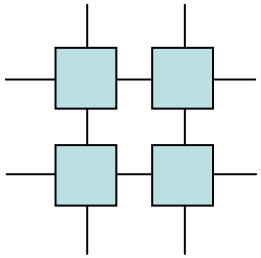
Orca

- **Advantages**
  - **Integrated operating system, compiler and runtime environment ensures stability**
  - **Extra semantics can be extracted to achieve speed**
- **Disadvantages**
  - **Integrated operating system, compiler and runtime environment makes the system less accessible**
  - **Existing application may prove difficult to port**



Orca Status

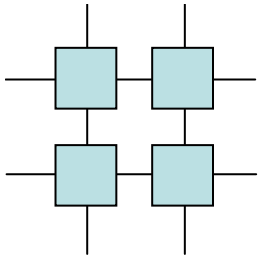
- **Alive and well**
- **Moved from Amoeba to BSD**
- **Moved from pure software to utilize custom firmware**
- **Many applications ported**



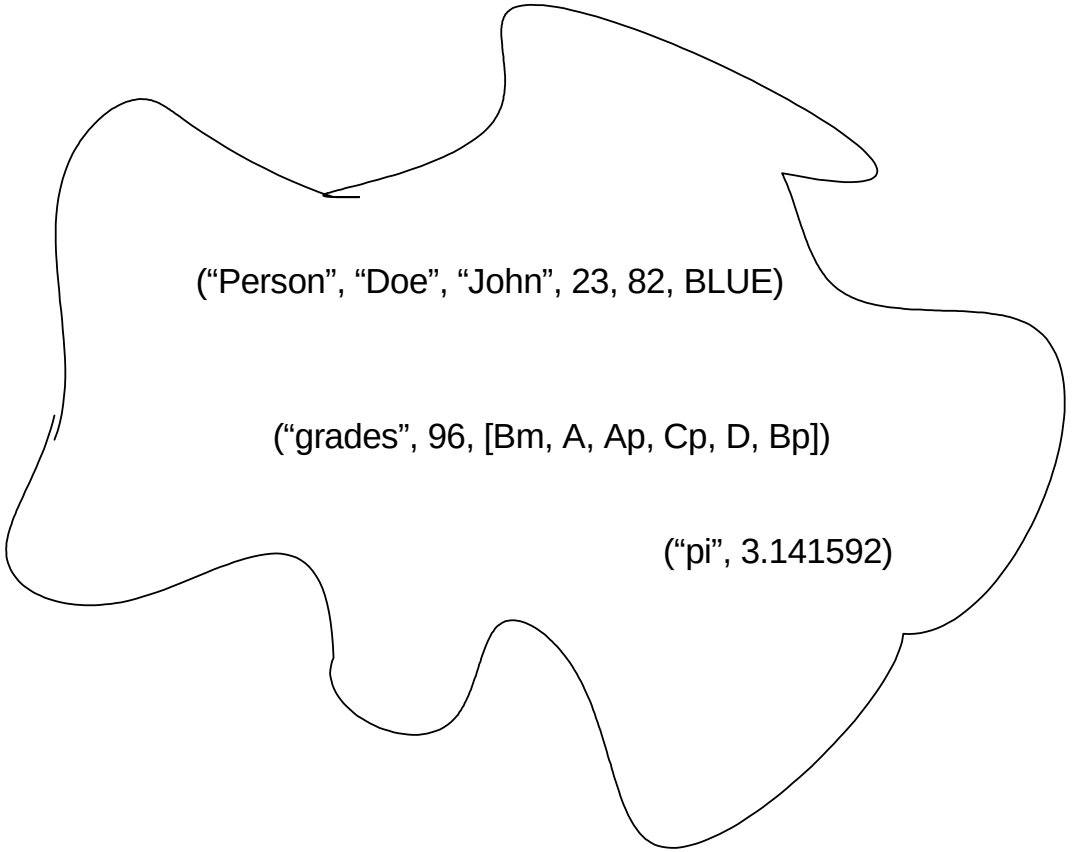
Linda

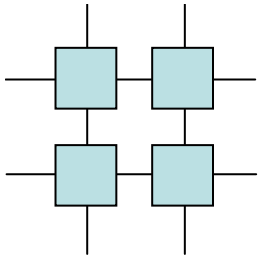
- **Tuple based**
- **Language independent**
- **Targeted at MPP systems but often used in NOW**
- **Structures memory in a tuple space**

```
(“Person”, “Doe”, “John”, 23, 82, BLUE)
(“pi”, 3.141592)
(“grades”, 96, [Bm, A, Ap, Cp, D, Bp])
```



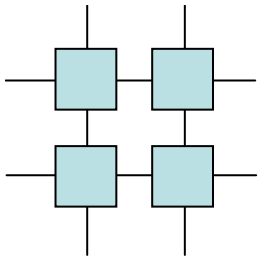
The Tuple Space





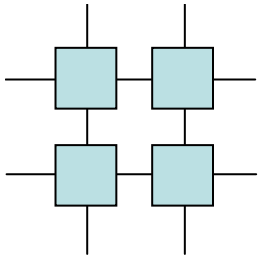
Linda

- **Linda consists of a mere 3 primitives**
  - **out - places a tuple in the tuple space**
  - **in - takes a tuple from the tuple space**
  - **read - reads the value of a tuple but leaves it in the tuple space**
- **No kind of ordering is guarantied, thus no consistency problems occur**



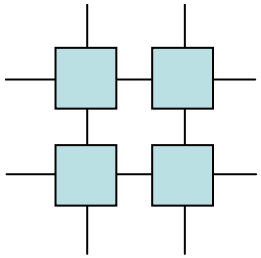
Linda

- **Advantages**
  - **No new language introduced**
  - **Easy to port trivial producer-consumer applications**
  - **Esthetic design**
  - **No consistency problems**
- **Disadvantages**
  - **Many applications are hard to port**
  - **Fine grained parallelism is not efficient**



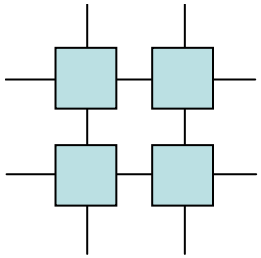
Linda Status

- **Alive but low activity**
- **Problems with performance**
- **Tuple based DSM improved by PastSet:**
  - **Introduced at kernel level**
  - **Added causal ordering**
  - **Added read replication**
  - **Drastically improved performance**



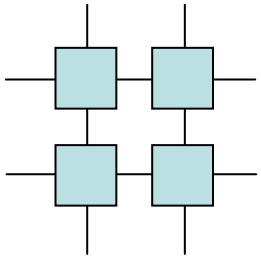
Ivy

- **The first page based DSM system**
- **No custom hardware used - only depends on MMU support**
- **Placed in the operating system**
- **Supports read replication**
- **Three distribution models supported**
  - **Central server**
  - **Distributed servers**
  - **Dynamic distributed servers**
- **Delivered rather poor performance**



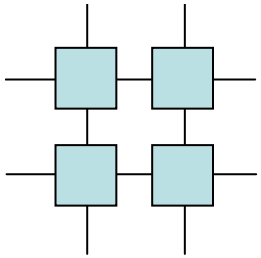
Ivy

- **Advantages**
  - **No new language introduced**
  - **Fully transparent**
  - **Virtual machine is a perfect emulation of an SMP architecture**
  - **Existing parallel applications runs without porting**
- **Disadvantages**
  - **Exhibits trashing**
  - **Poor performance**



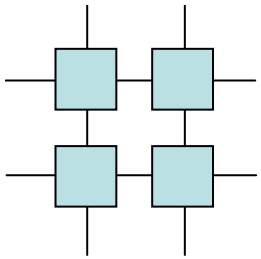
IVY Status

- **Dead!**
- **New SOA is Shrimp-2 SVM and CVM**
  - **Moved from kernel to user space**
  - **Introduced new relaxed consistency models**
  - **Greatly improved performance**
  - **Utilizing custom hardware at firmware level**



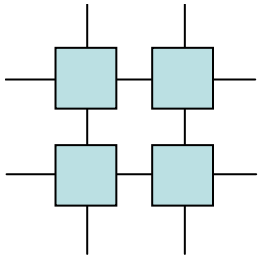
DASH

- **Flat memory model**
- **Directory Architecture keeps track of cache replica**
- **Based on custom hardware extensions**
- **Parallel programs run efficiently without change, trashing occurs rarely**



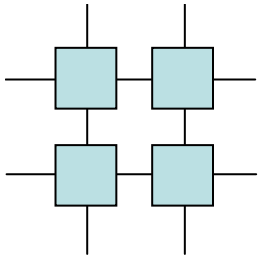
DASH

- **Advantages**
  - **Behaves like a generic shared memory multi processor**
  - **Directory architecture ensures that latency only grow logarithmic with size**
- **Disadvantages**
  - **Programmer must consider many layers of locality to ensure performance**
  - **Complex and expensive hardware**



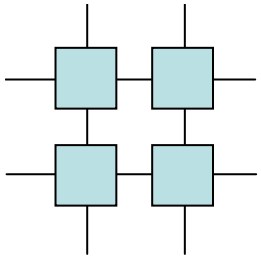
DASH Status

- **Alive**
- **Core people gone to SGI**
- **Main design can be found in the SGI Origin-2000**
- **SGI Origin designed to scale to 1024 processors**



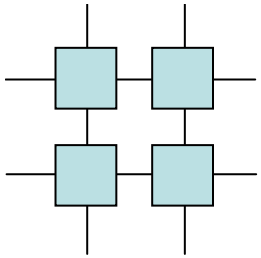
In depth problems to be presented later

- **Data location problem**
- **Memory consistency problem**



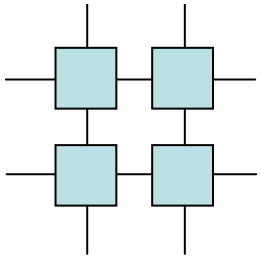
Consistency Models

Relaxed Consistency Models for Distributed Shared Memory



Presentation Plan

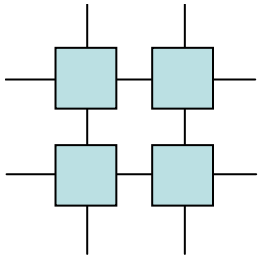
- Defining Memory Consistency
- Motivating Consistency Relaxation
- Consistency Models
- Comparing Consistency Models
- Working with Relaxed Consistency
- Summary



Defining Memory Consistency

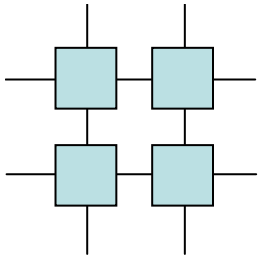
A Memory Consistency Model defines a set of constraints that must be meet by a system to conform to the given consistency model. These constraints define a set of rules that define how memory operations are viewed relative to:

- *Real time*
- *Each other*
- *Different nodes*



Why Relax the Consistency Model

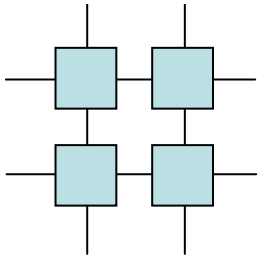
- To simplify bus design on SMP systems
  - More relaxed consistency models requires less bus bandwidth
  - More relaxed consistency requires less cache synchronization
- To lower contention on DSM systems
  - More relaxed consistency models allows better sharing
  - More relaxed consistency models requires less interconnect bandwidth



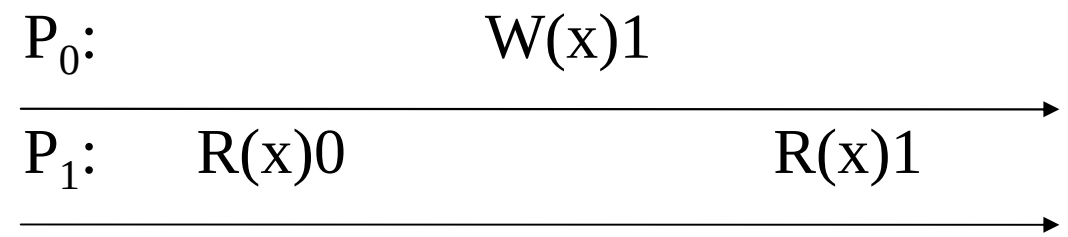
Strict Consistency

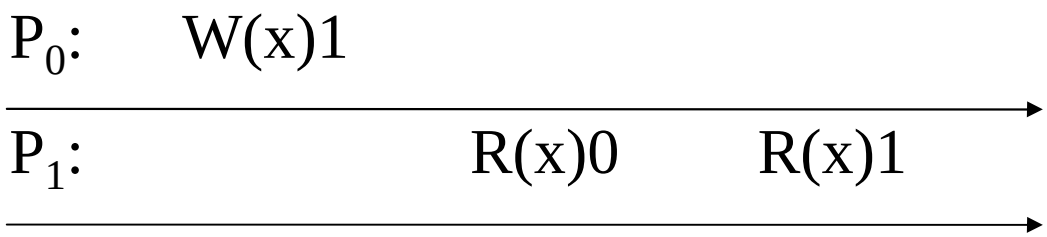
Any read to a memory location x returns the value stored by the most resent write to x.

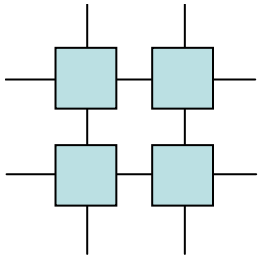
- Performs correctly with race conditions
- Can’t be implemented in systems with more than one CPU



Strict Consistency



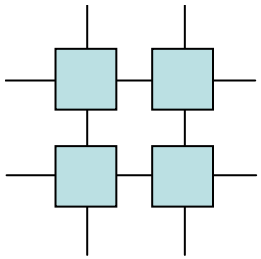




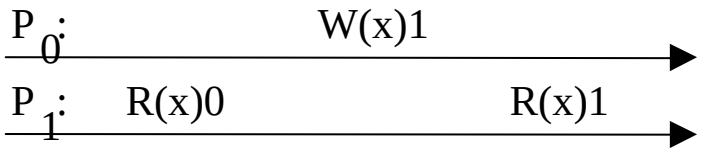
Sequential Consistency

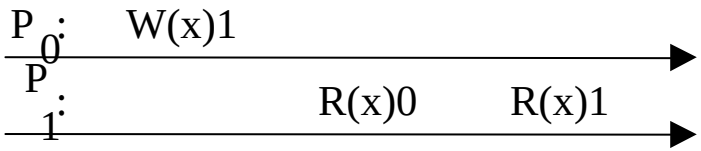
[A multiprocessor system is sequentially consistent if] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appears in this sequence in the order specified by its program.

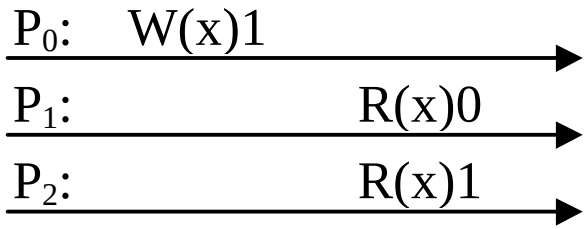
- Handles all correct code, except race conditions
- Can be implemented with more than one CPU

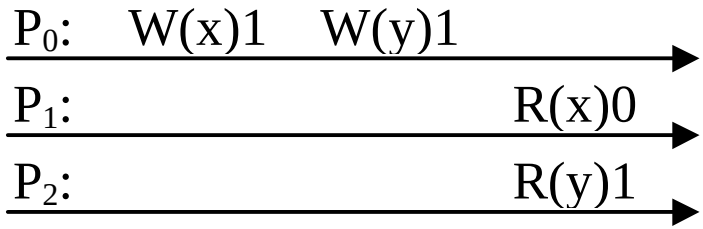


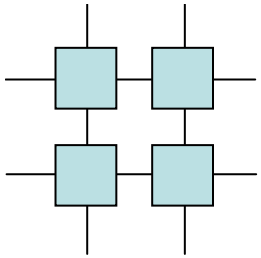
Sequential Consistency







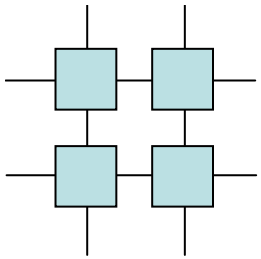




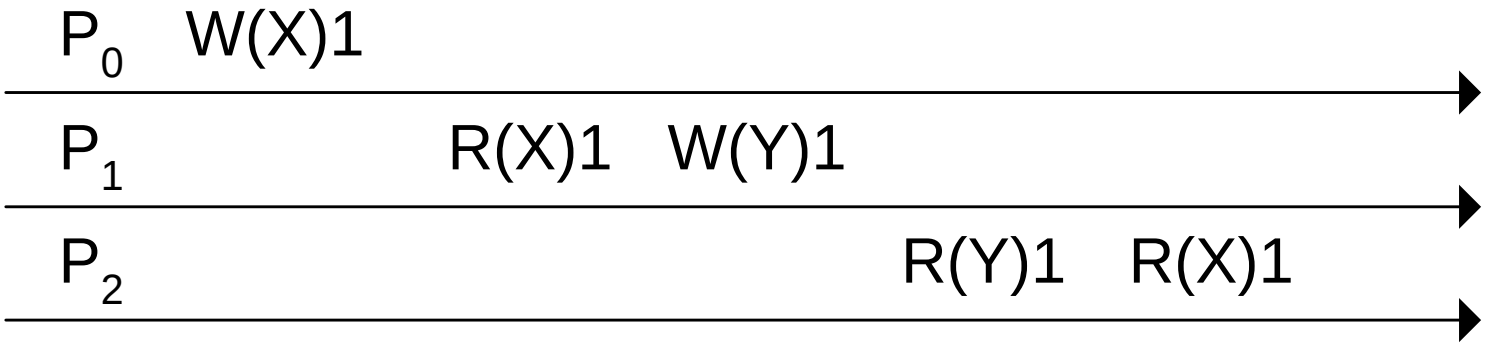
Causal Consistency

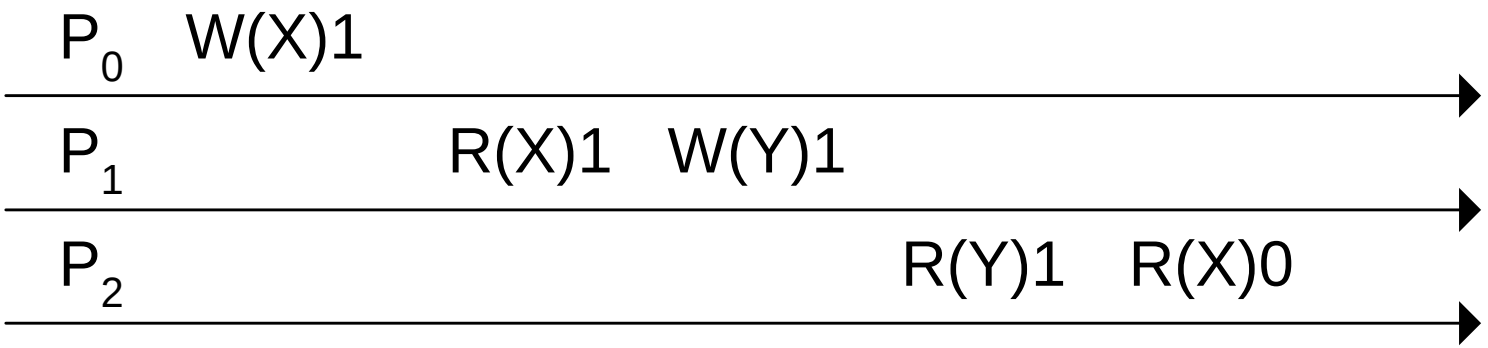
Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines.

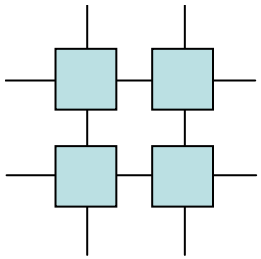
- Still fits programmers idea of sequential memory accesses
- Hard to make an efficient implementation



Causal Consistency



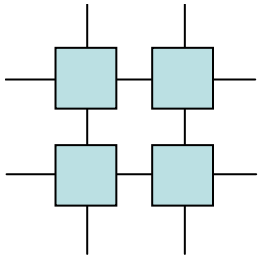




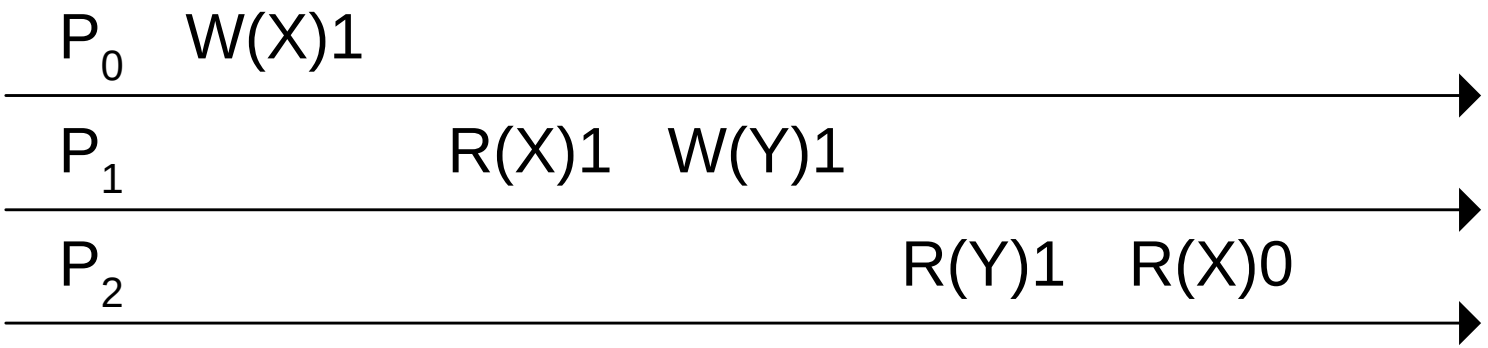
PRAM Consistency

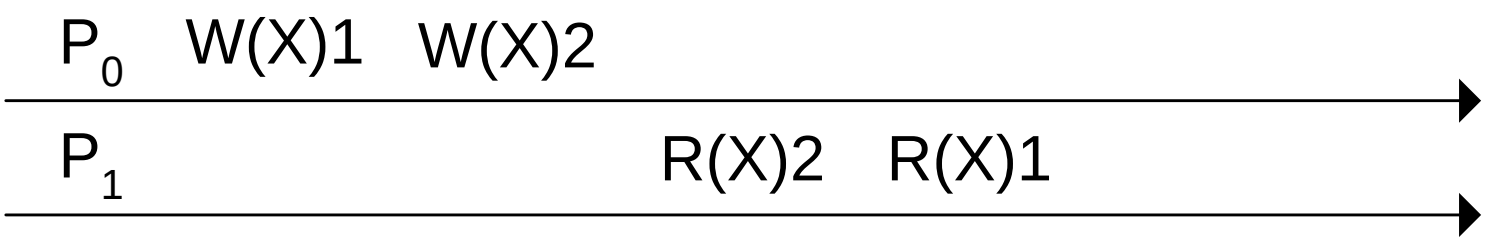
Writes done by a single process are received by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes.

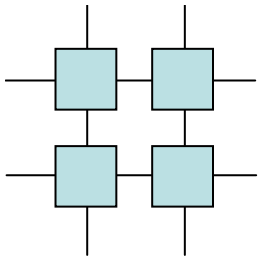
- Operations from one node can be grouped for better performance
- Does not comply with ordinary memory conception



PRAM Consistency



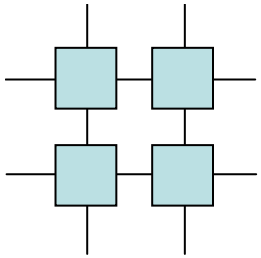




Processor Consistency

1. *Before a read is allowed to perform with respect to any other processor, all previous reads must be performed.*
2. *Before a write is allowed to perform with respect to any other processor, all other accesses (read and write) must be performed.*

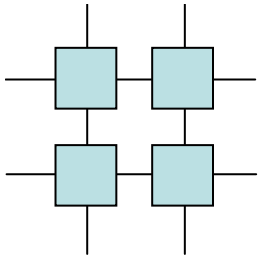
- Slightly stronger than PRAM
- Slightly easier than PRAM



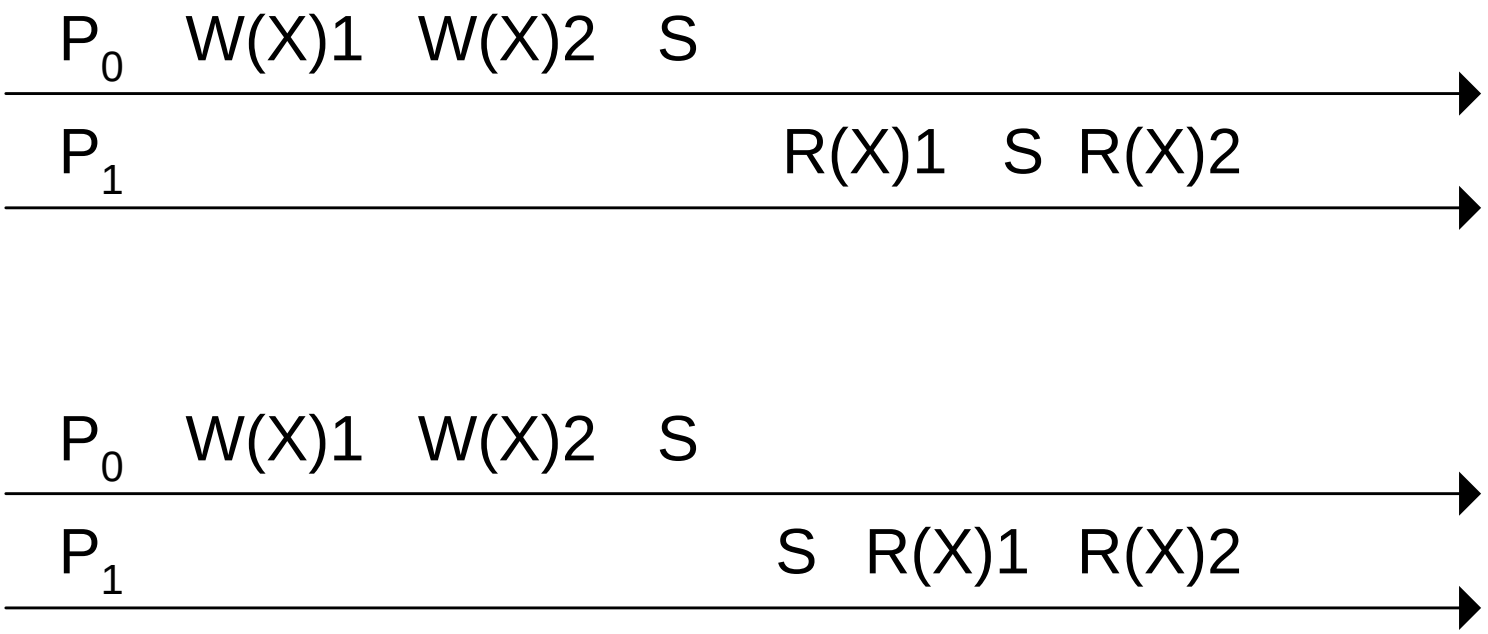
Weak Consistency

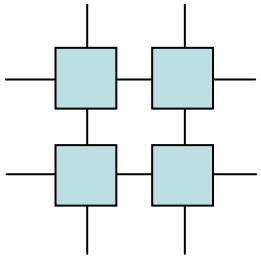
1. *Accesses to synchronization variables are sequentially consistent.*
2. *No access to a synchronization variable is allowed to be performed until all previous writes have completed everywhere.*
3. *No data access (read or write) is allowed to be performed until all previous accesses to synchronization variables have been performed.*

- Synchronization variables are different from ordinary variables
- Lends itself to natural synchronization based parallel programming



Weak Consistency

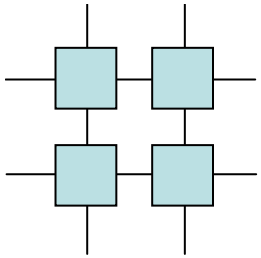




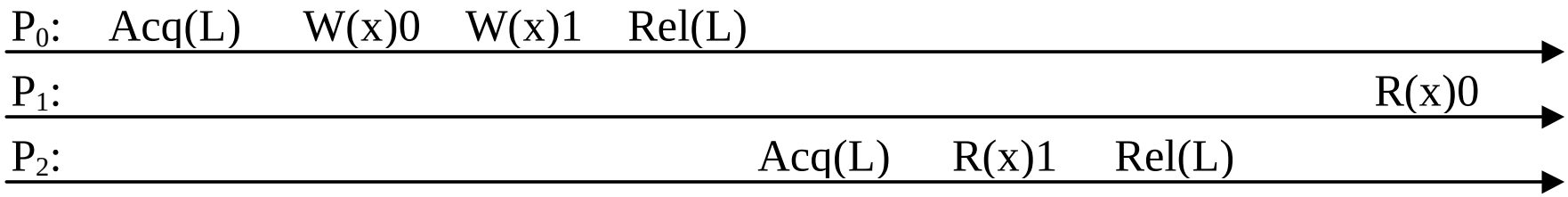
Release Consistency

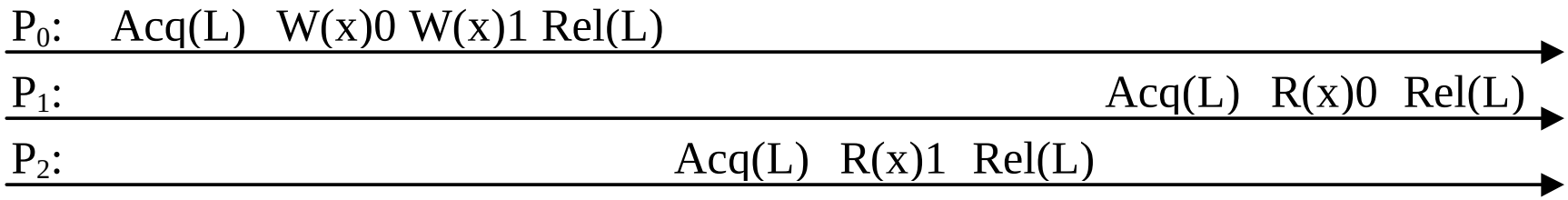
1. *Before an ordinary access to a shared variable is performed, all previous acquires done be the process must have completed successfully.*
2. *Before a release is allowed to be performed, all previous reads and writes done by the process must have completed.*
3. *The acquire and release accesses must be processor consistent.*

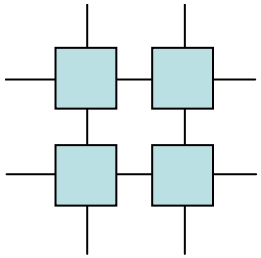
- Synchronization's now differ between Acquire and Release
- Lends itself directly to semaphore synchronized parallel programming



Release Consistency

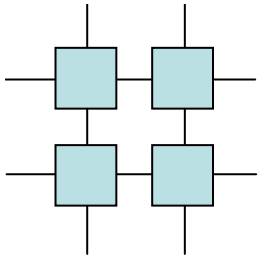






Lazy Release Consistency

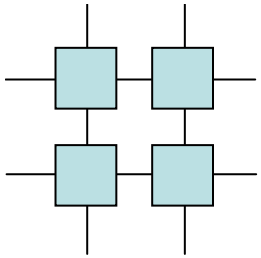
- Differs only slightly from Release Consistency
- Release dependent variables are not propagated at release, but rather at the following acquire
- This allows Release Consistency to be used with smaller granularity



Entry Consistency

1. *An acquire access of a synchronization variable is not allowed to perform with respect to a process until all updates to the guarded shared data have been performed with respect to that process.*
2. *Before an exclusive mode access to a synchronization variable by a process is allowed to perform with respect to that process, no other process may hold the synchronization variable, not even in non-exclusive mode.*
3. *After an exclusive mode access to a synchronization variable has been performed, any other process' next non-exclusive mode access to that synchronization variable may not be performed until it has been performed with respect to that variable's owner.*

- Associates specific synchronization variables with specific data variables

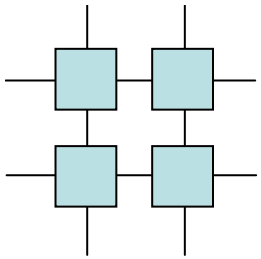


Automatic Update

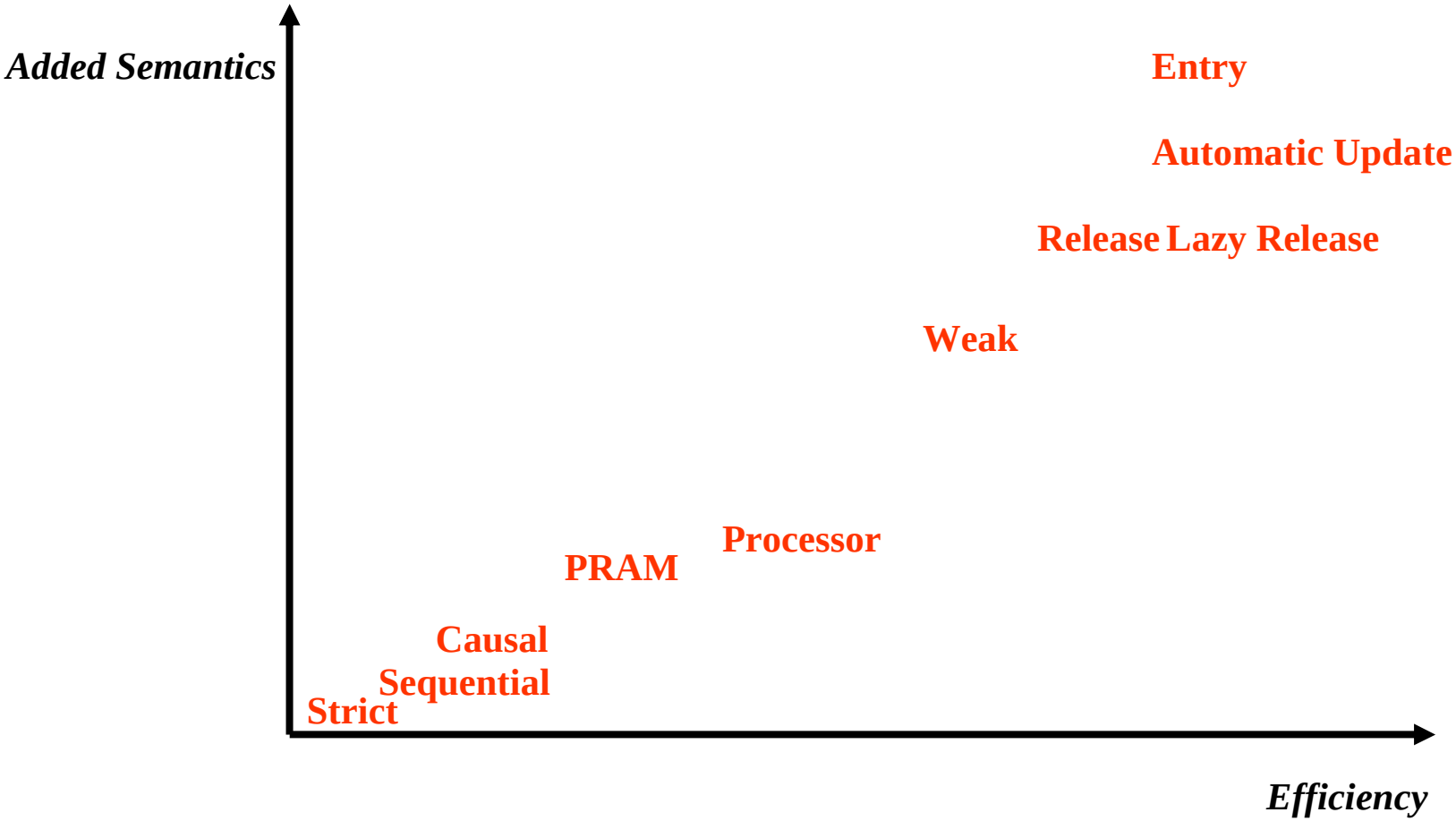
Automatic update consistency has the same semantics as lazy release consistency, and adding:

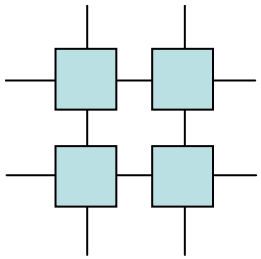
> *Before performing a release all automatic updates must be performed.*

- Lends itself to hardware support
- Efficient when two nodes are sharing the same data often



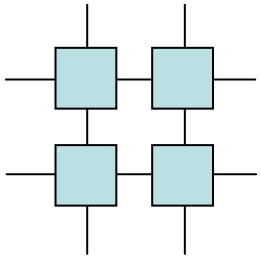
Comparing Consistency models





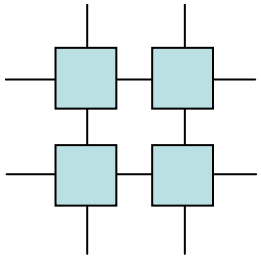
Working with Relaxed Consistency Models

- Natural tradeoff between efficiency and added work
- Anything beyond Causal Consistency requires the consistency model to be explicitly known
- Compiler knowledge of the consistency model can hide the relaxation from the programmer



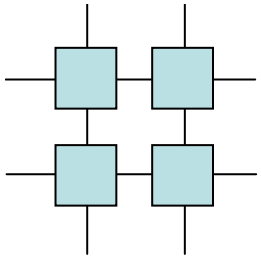
Summary

- Relaxing memory consistency is necessary for any system with more than one processor
- Simple relaxation can be hidden
- Strong relaxation can achieve better performance



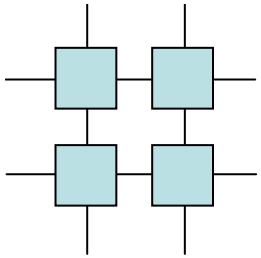
Data Location

Finding the data in Distributed Shared Memory Systems.



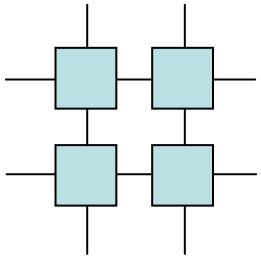
Coming Up

- Data Distribution Models
- Comparing Data Distribution Models
- Data Location
- Comparing Data Location Models



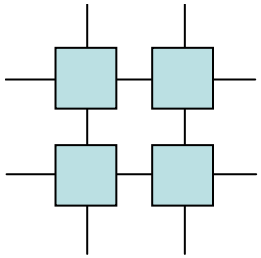
Data Distribution

- Fixed Location
- Migration
- Read Replication
- Full Replication
- Comparing Distribution Models



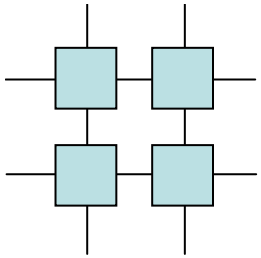
Fixed Location

- Trivial to implement via RPC
- Can be handled at compile time
- Easy to debug
- Efficiency depends on locality
- Lends itself to Client-Server type of applications



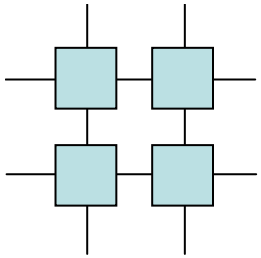
Migration

- Programs are written for local data access
- Accesses to non present data is caught at runtime
- Invisible at compile time
- Can be hardware supported
- Efficiency depends on several elements
  - Spatial Locality
  - Temporal Locality
  - Contention



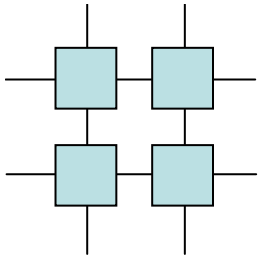
Read Replication

- As most data that exhibits contention are read only data the idea of read-replication is intuitive
- Very similar to copy-on-write in UNIX fork() implementations
- Can be hardware supported
- Natural problem is when to invalidate mutable read replicas to allow one node to write

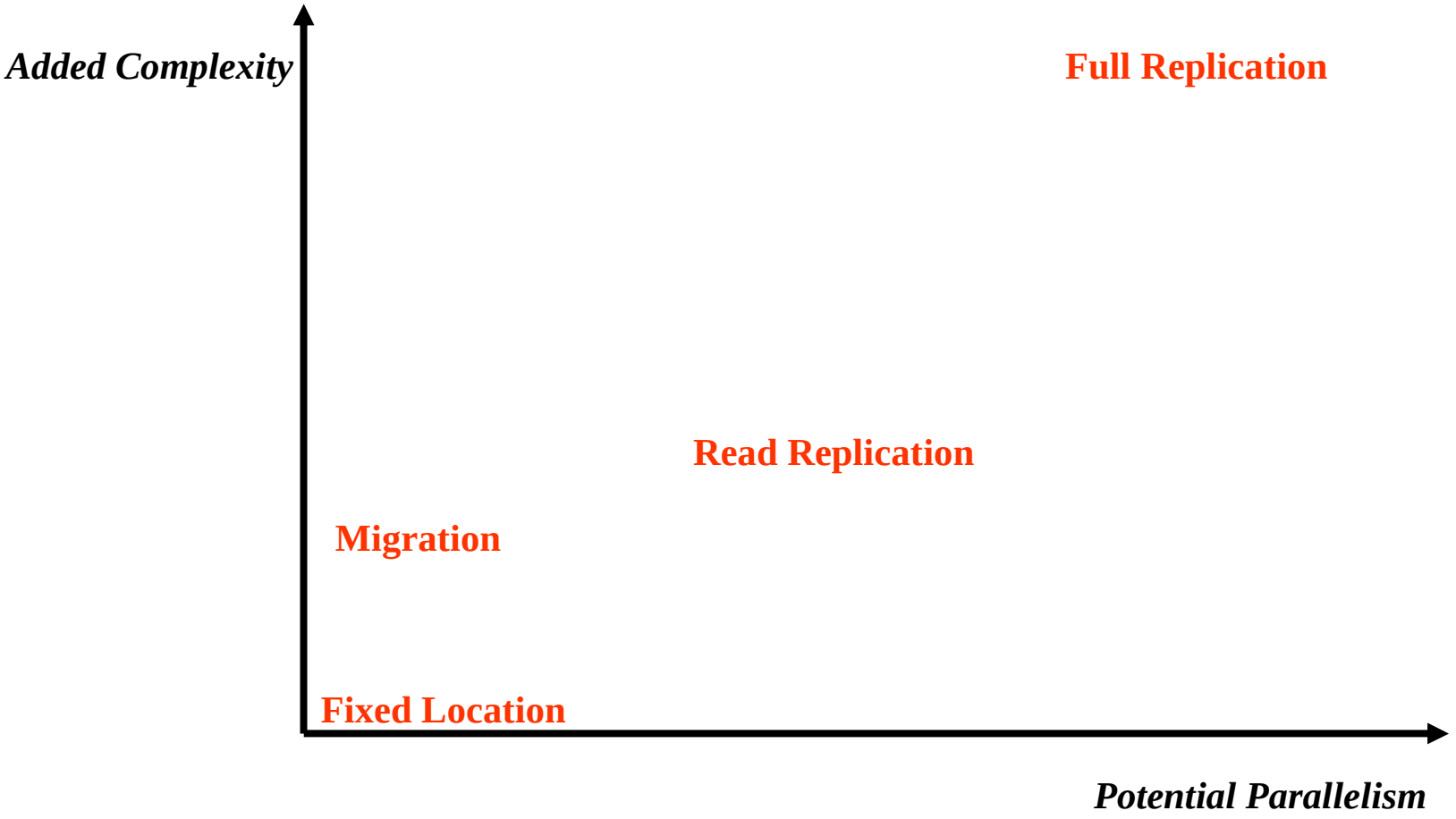


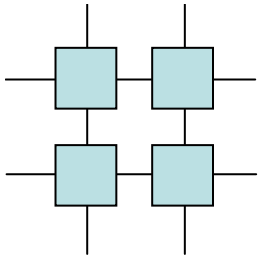
Full Replication

- Migration+Read replication+Write replication
- Write replication requires four phases
  - Obtain a copy of the data block and make a copy of that
  - Perform writes to one of the copies
  - On releasing the data create a log of performed writes
  - Assembling node checks that no two nodes has written the same position
- Showed to be of little interest



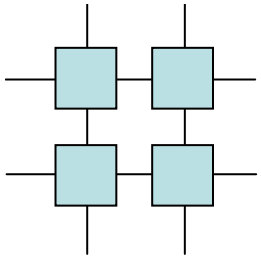
Comparing Distribution Models





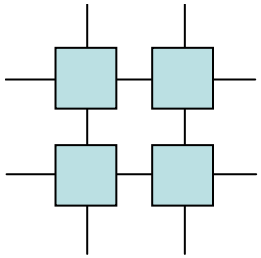
Data Location

- Central Server
- Distributed Servers
- Dynamic Distributed Servers
- Home Base Location
- Directory Based Location
- Comparing Location Models



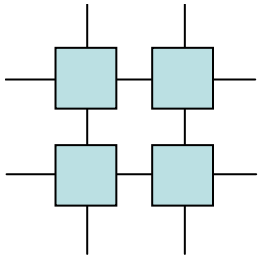
Central Server

- All data location is know a one place
- Simple to implement
- Low overhead at the client nodes
- Potential bottleneck
- The server could be dedicated for data serving



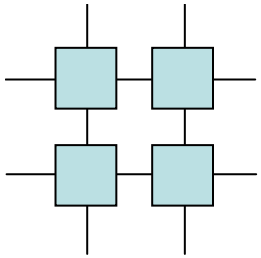
Distributed Servers

- Data is placed at node once
- Relatively simple to implement
- Location problem can be solved in two ways
  - Static mapping
  - Locate once
- No possibility to adapt to locality patterns



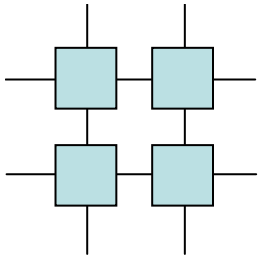
Dynamic Distributed Servers

- Data block handling can migrate during execution
- More complex implementation
- Location may be done via
  - Broadcasting
  - Location log
  - Node investigation
- Possibility to adapt to locality patterns
- Replica handling becomes inherently hard



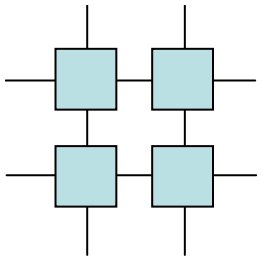
Home Base Location

- The Home node always hold a coherent version of the data block
- Otherwise very similar to distributed server
- Advanced distribution models such as shared write don’t have to elect a leader for data merging.

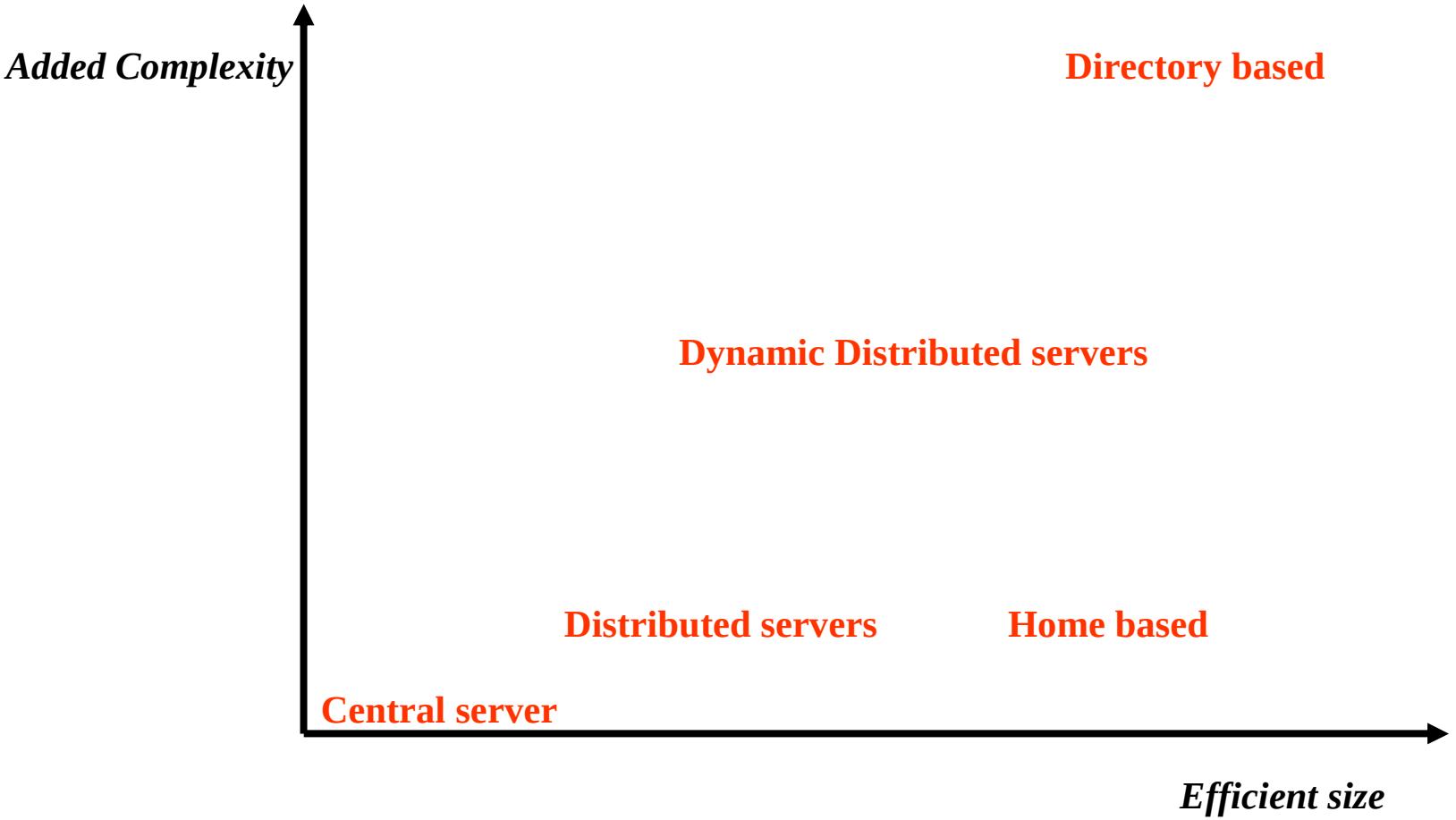


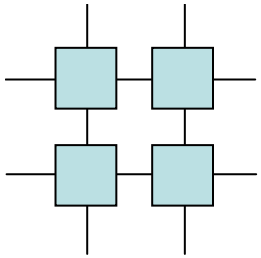
Directory Based Location

- Specially suited for non-flat topologies
- Nodes only has to consider their immediate server
- Servers provide a view as a ’virtual’ instance of the remaining system
- Servers may connect to servers in the same invisible way
- Usually hardware based



Comparing Location Models





Summary

- Distribution aspects differ widely, but high complexity don’t always pay of
- Data location can be solved in various ways, but each solution behaves best for a given number of nodes