Parallel Computer Systems

Todd C. Mowry

CS 347

April 23 & 28, 1998

Topics

- Design Issues
- Current Parallel Architectures
- Parallel Programming
- Cache Coherence
- Memory Consistency Models

Motivation

Limits to Sequential Processing

- **Cannot push clock rates beyond technological limits**
- **Instruction-level parallelism gets diminishing returns**
  - 4-way superscalar machines only get average of 1.5 instructions / cycle
  - Branch prediction, speculative execution, etc. yield diminishing returns

Applications have Insatiable Appetite for Computing

- **Modeling of physical systems**
- **Virtual reality, real-time graphics, video**
- **Database search, data mining**

Many Applications can Exploit Parallelism

- **Work on multiple parts of problem simultaneously**
- **Synchronize to coordinate efforts**
- **Communicate to share information**

What is Parallel Architecture

Two components:

- **OLD: Instruction Set Architecture**
- **NEW: Communications Architecture**

Communications Architecture

Primitive communication abstractions that the hardware and software provide to the programmer.

- **e.g., shared memory model, message-passing model, etc.**

Implicit assumption:

- **the primitive operations are efficiently implemented.**

Goals are:

- **Broad applicability**
- **Programmability**
- **Scalability**
- **Cost**

Scalability Considerations

Both small-scale and large-scale machines have their place in the market.

Cost-performance-complexity issues, however, can be different depending on the scale.

Design Issues

Naming:

- **How is communicated data named?**

Latency:

- **What is the latency of communicating in a protected fashion?**

Bandwidth:

- **How much data can be communicated per second?**

Synchronization:

- **How can producers and consumers of data synchronize?**

Node Granularity:

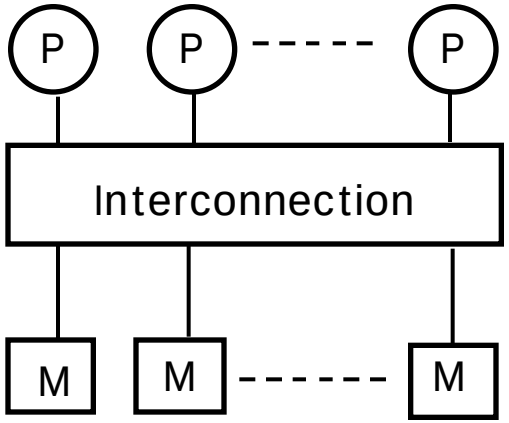
- **How to divide up the hardware resources efficiently?**
  - number and size of processors, distribution of memory, etc.

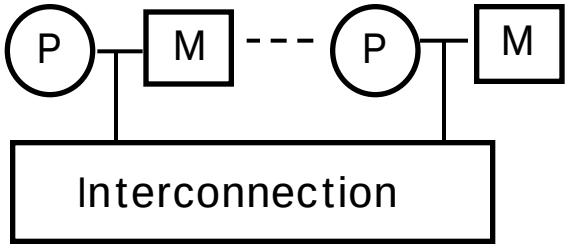
Current Parallel Architectures

- **Shared Memory**
- **Message Passing**
- **Data Parallel**
- **Dataflow**
- **Systolic**

Shared Memory Architectures

- **Key Feature: All processors in the system can directly access all memory locations in the system, thus providing a convenient and fast/cheap (?) mechanism for processors to communicate.**
  - convenient: (i) location transparency; (ii) abstraction supported is same as that on current day uniprocessors.
  - fast/cheap: as compared to other models (more later).

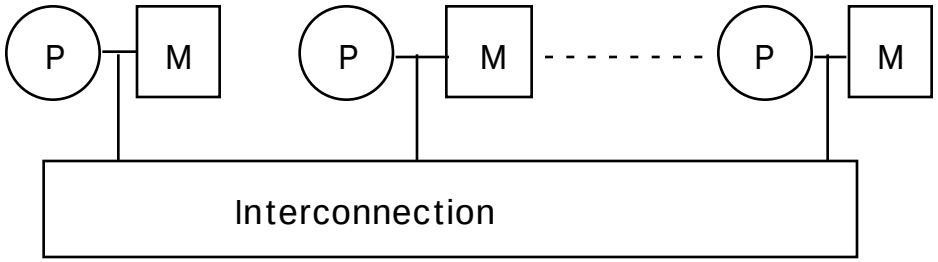




- **Memory can be centrally placed or distributed in such machines.**
- **Better name is Single Global Address Space machines.**
- **A problem traditionally cited with such machines is scalability.**

Message Passing Architectures

- **Processors can directly access only local memory, and all communication and synchronization happens via messages.**



Some issues:

- **Sending a message often incurs many overheads:**
  - building a header; copying data into network buffers; sending data; receiving data into buffers; copying data from kernel to user space**.**
    - » **Many of these steps require OS intervention.**
- **Synchronization using messages is based on handshake protocols.**
- **One of the main advantages is easy scalability.**

Shared Memory vs. Message Passing

From one perspective, single-address space machines can be viewed as message-passing machines.

- **Reading a remote location is simply sending a message to the remote node asking for the contents of that location.**
- **The difference boils down to efficiency and performance.**
  - » A single-address space machine is specialized to handle memory read/write messages in hardware and is thus faster at such operations.
  - » It also leaves the processor free to handle other computational tasks.

Data Parallel Architectures

Programming Model:

- assumes that there is a processor associated with each member of a collection of data and cheap global synchronization.

Example:

- Each PE contains an employee record with his/her salary
- Code:

```
If salary > 100K then
    salary = salary *1.05
else salary = salary *1.10
```

- Logically, the whole operation takes a single cycle.
- Notion of enabled processors.

Other examples:

- Document searching, graphics, image processing, ...

Machines:

- Maspar MP-1 and MP-2, Thinking Machines CM-2 and CM-5.

Data Parallel Architectures (Cont.)

Architectures supporting this model are often characterized by:

- **Single instruction stream**
- **Weak processors with limited local memory**
- **Powerful interconnection network**
- **Are used as an attached processor to a host**

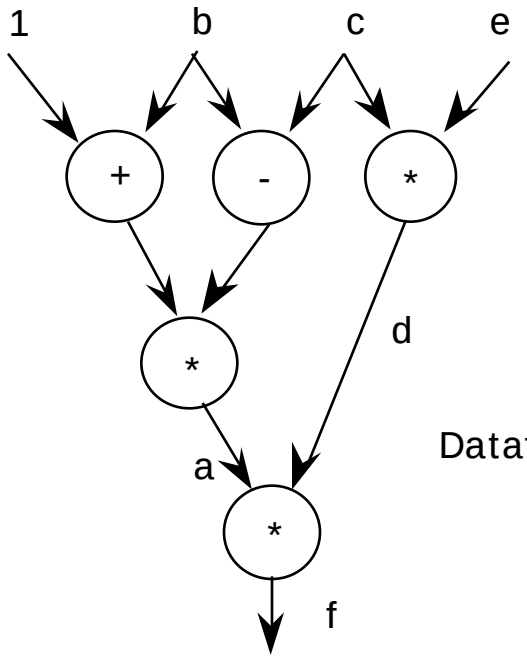
Some issues:

- **General applicability is not clear at this point in time**
- **MIMD machines can be just as effective at exploiting data parallelism (E.g., CM-5 from TMC)**

Dataflow Architectures

- **In the dataflow model, instructions are activated by the availability of data operands for instructions.**
- **In contrast, in control flow models, computation is a series of instructions with implicit or explicit sequencing between them.**

```
a=(b+1)*(b-c)
d=c*e
f=a*d
```

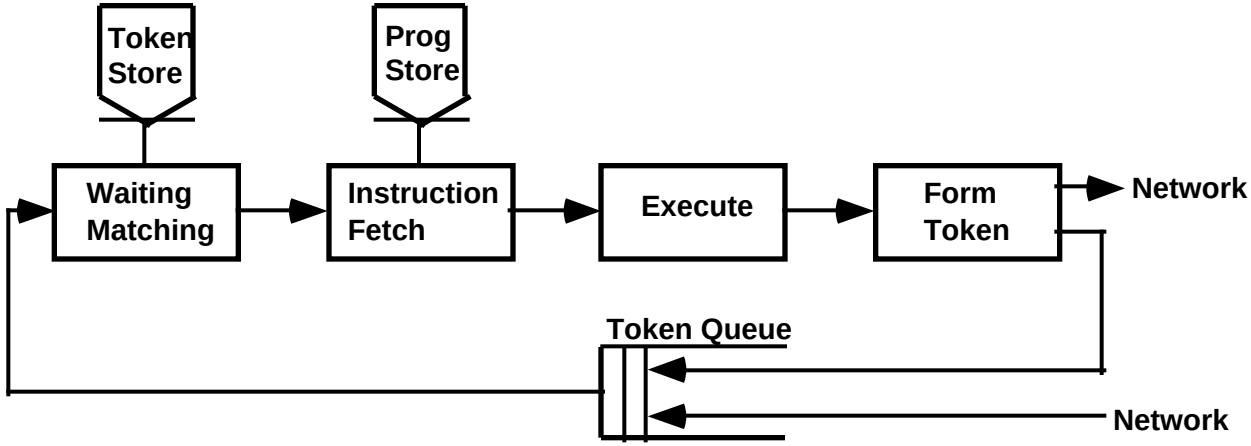


Dataflow graph

- **One of the advantages is that all dependencies are explicitly present in the dataflow graph, so parallelism is not hidden from hardware.**

Dataflow Architectures (Cont.)

General structure:

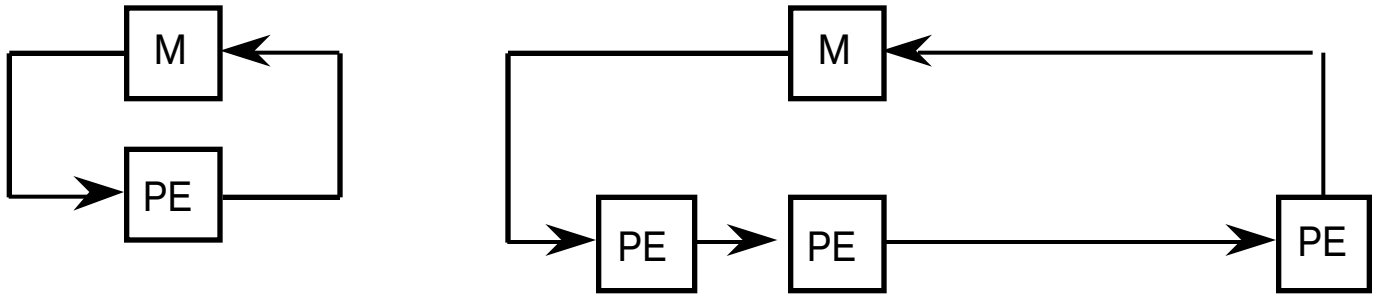


Some issues:

- granularity of operations (locality issues and macro dataflow)
- efficient handling of complex data structures like arrays
- complexity of matching store and memory units
- problems due to excess parallelism

Systolic Architectures

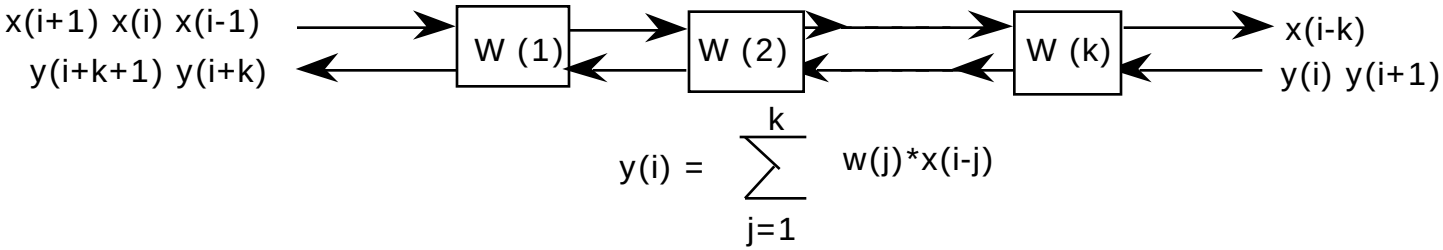
Basic principle: By replacing a single PE by a regular array of PEs, and by carefully orchestrating the flow of data between the PEs, high throughput can be achieved without increasing memory bandwidth requirements.



- **Distinguishing features from regular pipelined computers:**
  - Array structure can be non linear (e.g, hexagonal)
  - Pathways between PEs may be multidirectional
  - PEs may have local instruction and data memory, and are more complex than the stage of a pipelined computer.

Systolic Architectures (Cont.)

Example: Systolic array for 1-D convolution



Issues:

- System integration: Shipping data from host array and back
- Cell architecture and communication architecture (Warp, iWarp)
- Software for automatically mapping computations to systolic arrays
- General purpose systolic arrays

Architectural Issues

Naming

- **Single Global Linear-Address-Space**
  - (shared memory)

vs.

- **Single Global Segmented-Name-Space**
  - (global objects)

vs.

- **Multiple Local Address/Name Spaces**
  - (message passing)

Naming strategy has implications on:

- Programmer / Software
- Performance
- Design Complexity

Application Classes

Compute Servers

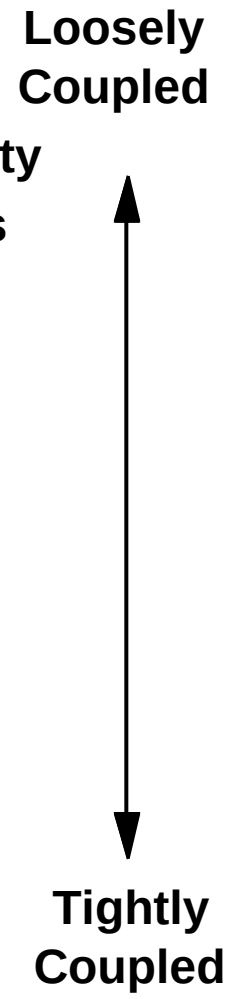
- **Number of independent users using single computing facility**
- **Only synchronization is to mediate use of shared resources**
  - Memory, disk sectors, file system

Database Servers

- **Users performing transactions on shared database**
  - E.g., bank records, flight reservations
- **Synchronization required to guarantee consistency**
  - Don't want two people to get last seat on flight

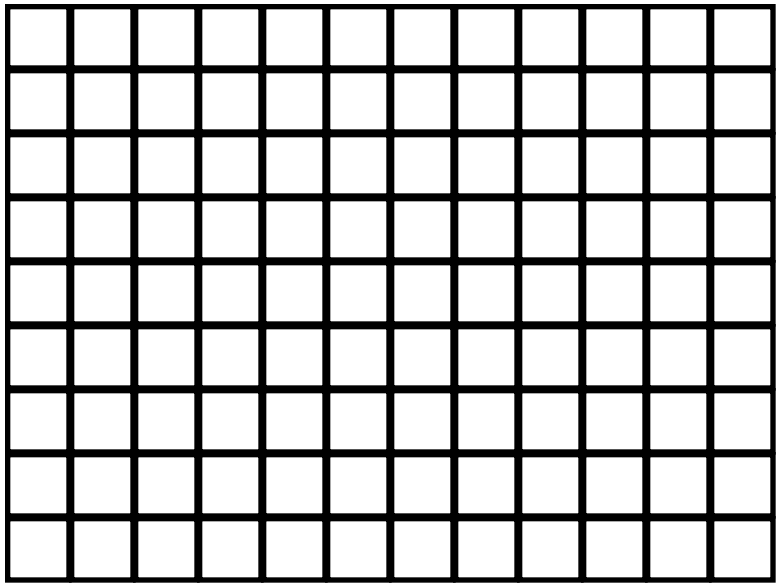
True Parallel Applications

- **Computationally intensive task exploiting multiple computing agents**
- **Synchronization required to coordinate efforts**



Parallel Application Example

Finite Element Model



Discrete representation of continuous system

- **Spatially: partition into mesh elements**
- **Temporally: Update state every dT time units**

Example Computation

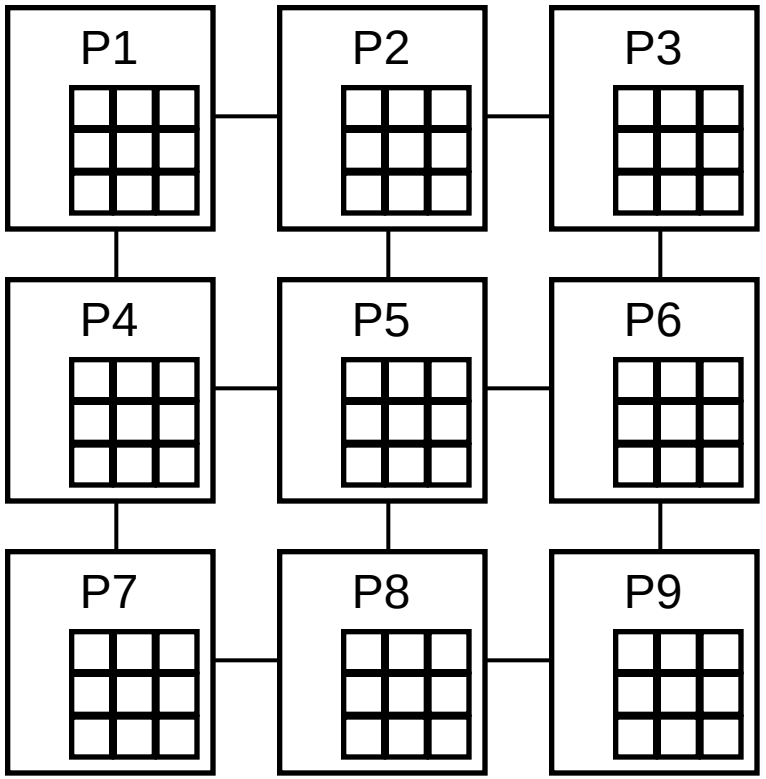
```
For time from 0 to maxT
   for each mesh element
      Update mesh value
```

Locality

- **Update depends only on values of adjacent elements**

Parallel Mapping

Spatial Partitioning



Partitioning

- **Divide mesh into regions**
- **Allocate different regions to different processors**

Computation for Each Processor

```
For time from 0 to maxT
  Get boundary values from
    neighbors
  For each mesh element
       Update mesh value
  Send boundary values to
    neighbors
```

Complicating Factors

Communication Overhead

- **N X N mesh, M processors**
- **Elements / processor = N^2 / M**
  - How much work is required per iteration
- **Boundary elements / processor ~ N / Sqrt(M)**
  - How much communication is required per iteration
- **Communication vs. computation load ~ Sqrt(M) / N**
  - Become communication limited as increase number of processors

Nonuniformities

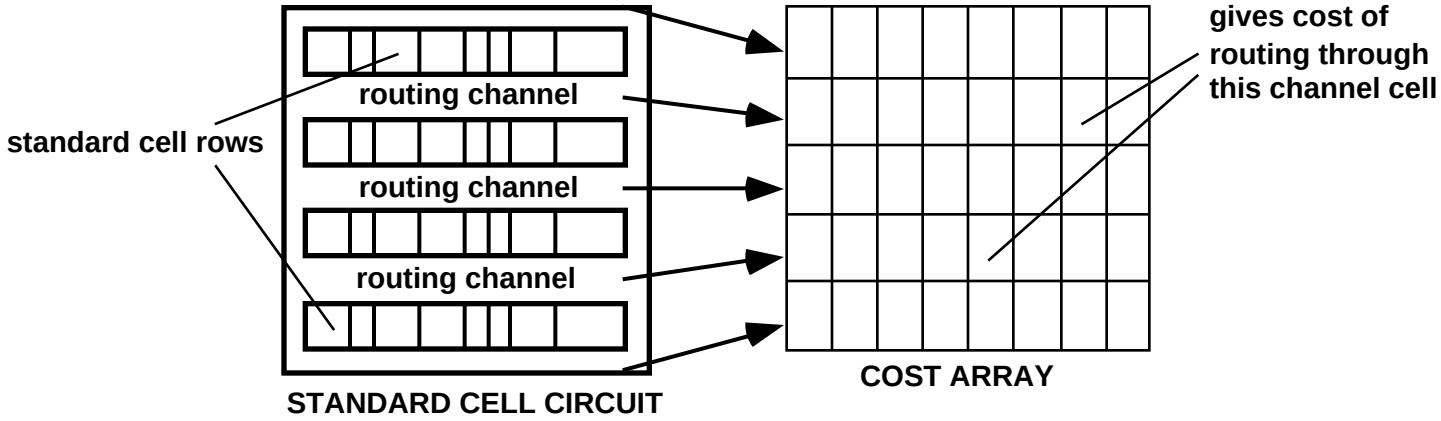
- **Irregular mesh, varying computing / mesh element**
- **Makes partioning & load balancing difficult**

Synchronization

- **Keeping all processors on same iteration**
- **Determining global properties such as convergence and time step**

Programmer / Software

- **Example: LocusRoute (standard cell router)**




```
while (route_density_improvement > threshold)
  {
    for (i = 1 to num_wires) do
       {
         - rip old wire route out
         - explore new routes
         - place wire using best new route
       }
  }
```

Shared-Memory Implementation

Shared memory algorithm:

- Divide cost-array into regions (assign regions to PEs)
- Assign wires to PEs based on the region in which center lies
- Do load balancing using stealing when local queue empty

Good points:

- Good load balancing
- Mostly local accesses
- High cache-hit ratio

Message-Passing Implementations

Solution-1:

- **Distribute wires and cost-array regions as in sh-mem implementation**
- **Big overhead when wire-path crosses to remote region**
  - send computation to remote PE, or
  - send messages to access remote data

Solution-2:

- **Wires distributed as in sh-mem implementation**
- **Each PE has copy of full cost array**
  - one owned region, plus potentially stale copy of others
  - send frequent updates so that copies not too stale
- **Consequences:**
  - waste of memory in replication
  - stale data => poorer quality results or more iterations

=> In either case, lots of thinking needed on the programmer's part

Performance Impact of Shared Memory

1. **Efficient naming**
   - virtual to physical using TLBs
   - ability to name only the relevant portion of objects
2. **Ease and efficiency of caching**
   - caching is natural and well understood
   - can be automated using hardware (fixed sized blocks)
3. **Communication overhead**
   - low since protection is built into the memory system
   - easy for hardware to packetize requests and replies
4. **Integration of latency toleration**
   - demand-driven: consistency models, prefetching, multiple-ctxt PEs
   - can be extended to push data to other PEs plus bulk transfer

Design Complexity

For single global address space:

- **Managing replication and coherence in hardware can be tricky**
- **Fault containment is more difficult**

Naming: Summary

Consensus is building towards single address space machines, though many are still not cache coherent.

Latency

Ability to deal with latency is critical with fast processors

Options for dealing with latency:

1. **Lower frequency of long latency events**
   - algorithmic changes, computation and data distribution, ...
2. **Reduce latency**
   - caching of shared data, network design, ...
3. **Tolerate latency**
   - message-passing overlaps communication and computation (under programmer control)
   - shared-memory overlaps access completion and computation using consistency model or prefetching

Bandwidth

Parallel programs have both *private* and *global* bandwidth requirements

Private bandwidth requirements can be supported by:

- **distributing main memory among PEs**
- **application changes, local caches, mem system design, ...**

Global bandwidth requirements can be supported by:

- **scalable interconnect technology**
- **distributed main memory and caches**
- **application changes**
- **hot-spots can be a major problem**

Synchronization

Coordination mainly takes three forms:

- mutual exclusion (e.g., spin-locks)
- event notification (e.g., producer-consumer interactions)
- global barriers (e.g., end of phase indication)

Issues:

- need large synchronization name space
- low latency, low serialization (hot spot handling)
- many proposals (test-&-set, compare-&-swap, ldLinked-stCond, F/E bits and traps, queue-based locks, fetch&op with combining, ...)

Commodity Parts vs. Custom Design

The economics of mass production are a major driving force in the multiprocessor market

- **e.g., the first processor costs > $100M, subsequent ones cost ~$1K**

Three interesting design points:

1. Commodity everything

- e.g., clusters of PCs, workstations, or small-scale MPs
- network: fast switched ethernet, Myrinet, etc.

  => cheap but communication is slow (OS overhead part of the problem)

2. Commodity processors & memory, custom system & network

- e.g., SGI Origin 2000, IBM SP-2, Cray T3E, HP-Convex Examplar, etc.

3. Custom processors and everything else (except memory)

- Thinking Machines CM-2, MIT J-machine, etc.
- why design your own processor?
  - » very fine-grained parallelism (e.g., 64K processors in CM-2)

Parallel Machines

Definition: "A parallel computer is a collection of processing elements that can cooperate and communicate to solve large problems fast." ---Almasi and Gottlieb.

"... a collection of PEs..."

- **How many? How powerful is each? Scalability?**
- **Few very powerful (e.g., Cray YMP) or many weak ones (e.g., CM-2). Cost is a factor.**

"... that can communicate ..."

- **How do PEs communicate? (e.g. shared memory vs message passing)**
- **Interconnection network architecture? (bus, crossbar, multistage, ...)**
- **Evaluation criteria: cost, latency, throughput, scalability, fault tolerance**

"... and cooperate ..."

- **Important issues are synchronization, granularity, and autonomy**
- **Synchronization allows sequencing of actions for correctness**
  - Variety of primitives (test&set, fetch&add, ...); scalability issues
- **Granularity corresponds to the size of subtasks in program**
  - Granularity ↓ ==> Parallelism ↑; Communication ↑; Overhead ↑
  - Typical grain sizes vs. number of instructions
  - program level 10^6+ instructions
  - task level 10^3-10^6 instructions
  - loop level 10-1000 instructions
  - stmt/instr level 2-10 instructions

  Important to be able to exploit parallelism at all granularities
- **As for autonomy, the tradeoff is between SIMD and MIMD machines**
  - MIMD machines are more general purpose, but overheads can be large if PEs need to synchronize frequently

"... to solve large problems fast ..."

- **General purpose vs. special purpose machines?**
  - Any machine can do well on some problems!!!
- **What applications are amenable to parallel processing:**
  - Highly (often embarrassingly) parallel applications
    - » Many scientific codes that exploit data parallelism (monte-carlo)
  - Medium parallel applications
    - » Many engineering applications (finite-element pgms, VLSI-CAD)
  - Not-so-parallel applications
    - » Examples are compilers, editors, etc.
- **Application and machine scaling issues**

Parallel Software Basics

Achieving High Performance

- **sources of poor performance**
- **partitioning tasks and data**
- **bag of tricks**

Case Studies

- **Ocean simulation**
  - regular parallelism
- **Barnes-Hut N-body simulation**
  - irregular parallelism

Parallel Programming Task

Break up computation into tasks

- assign tasks to processors

Break up data into chunks

- assign chunks to memories

Introduce synchronization for:

- mutual exclusion
- event ordering

Sources of Poor Parallel Performance

- **Load imbalance**
  - **Amdahl's Law**
- **Poor data locality**
  - **communication**
  - **contention**
- **Synchronization overhead**
- **Redundant computation**
- **Task management**

Goals of Partitioning Tasks & Data

- **Balance workload across processors**
- **Maximize data locality (within and across tasks)**
  - **cache locality**
    - temporal
    - spatial
  - **memory locality**
    - colocate tasks with data
  - **network locality**
- **Reduce synchronization and contention**

Partitioning / Scheduling Techniques

- **Static**
  - **e.g., regular, grid-based applications**
  - **low runtime overhead, use when possible**
- **Dynamic**
  - **repartition periodically based on changing characteristics**
  - **task queues and task stealing**
  - **higher overhead, but needed in irregular, dynamic computations**

High-level dependence analysis to reduce synchronization

Bag of Tricks

Exploit orthogonal axes of parallelism

- **multiplicative benefit**
- **e.g., VLSI routing**

Relax consistency requirements when appropriate

- **e.g., iterative equation solvers**

Use speculative parallelism

- **e.g., branch-and-bound travelling salesman problem**

Fine-grained synch. for overlap across computational phases

Bag of Tricks (Cont.)

Block for locality

- **e.g., matrix multiply**

Duplicate computation to reduce communication

Change data layout

- **reduce cache conflicts**
- **exploit spatial locality due to long cache lines**

Use distributed data structures for synchronization

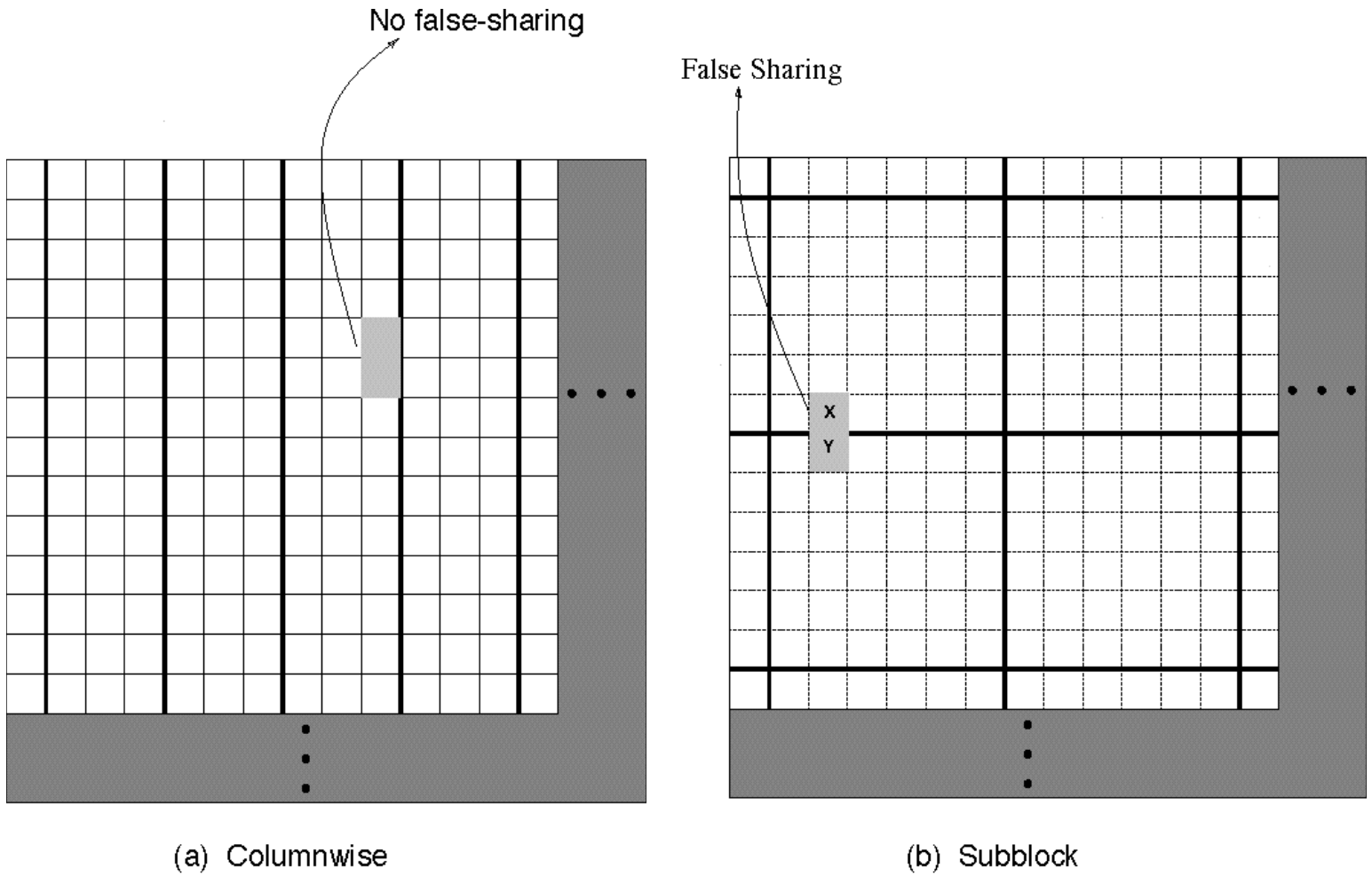
Case Study 1: Ocean Simulation

Put Laplacian of Ψ_1 in $W1_1$	Put Laplacian of Ψ_3 in $W1_3$	Copy Ψ_1, Ψ_3 into T_1, T_3	Put Ψ_1 - Ψ_3 in W2	Put computed Ψ_2 values in W3	Initialize γ_a and γ_b
Add f values to columns of $W1_1$ and $W1_3$		Copy Ψ_{1M}, Ψ_{3M} into Ψ_1, Ψ_3			Put Laplacian of Ψ_{1M}, Ψ_{3M} in $W7_{1,3}$
Put Jacobians of ($W1_1$, T_1), ($W1_3$, T_3) in $W5_1$, $W5_3$		Copy T_1, T_3 into Ψ_{1M}, Ψ_{3M}			Put Laplacian of $W7_{1,3}$ in $W4_{1,3}$
			Put Jacobian of (W2, W3) in W6		Put Laplacian of $W4_{1,3}$ in $W7_{1,3}$
U P D A T E T H E γ E X P R E S S I O N S					
S O L V E T H E E Q U A T I O N F O R Ψ_a A N D P U T T H E R E S U L T I N γ_a					
C O M P U T E T H E I N T E G R A L O F Ψ_a					
Compute $\Psi = \Psi_a + C(t)\Psi_b$ (note: Ψ_a and now Ψ are maintained in γ_a matrix)			Solve the equation for Φ and put result in γ_b		
Use Ψ and Φ to update Ψ_1 and Ψ_3					
Update streamfunction running sums and determine whether to end program					

Note: Every box is a computation on an entire grid(s). Horizontal lines represent synchronization points among all processes, and vertical lines spanning phases demarcate threads of dependence.

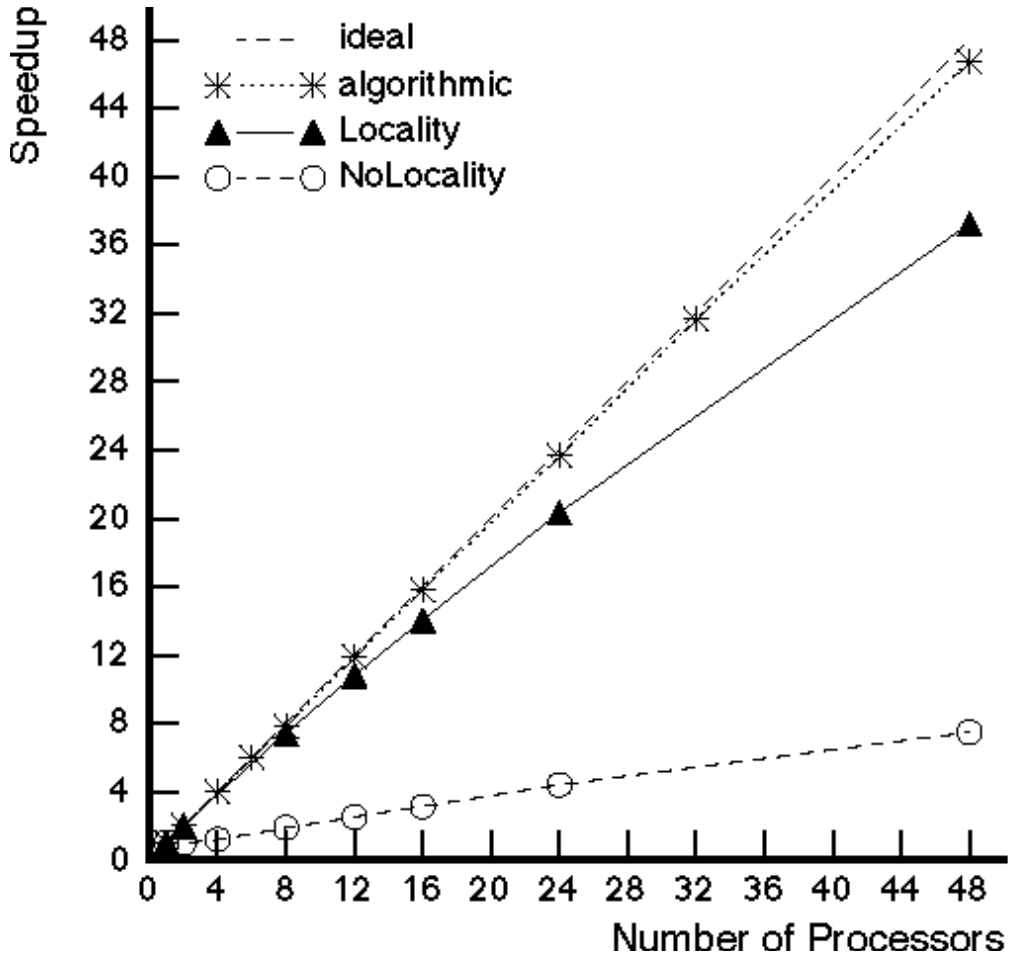
Computations in a Time-step

Ocean Simulation



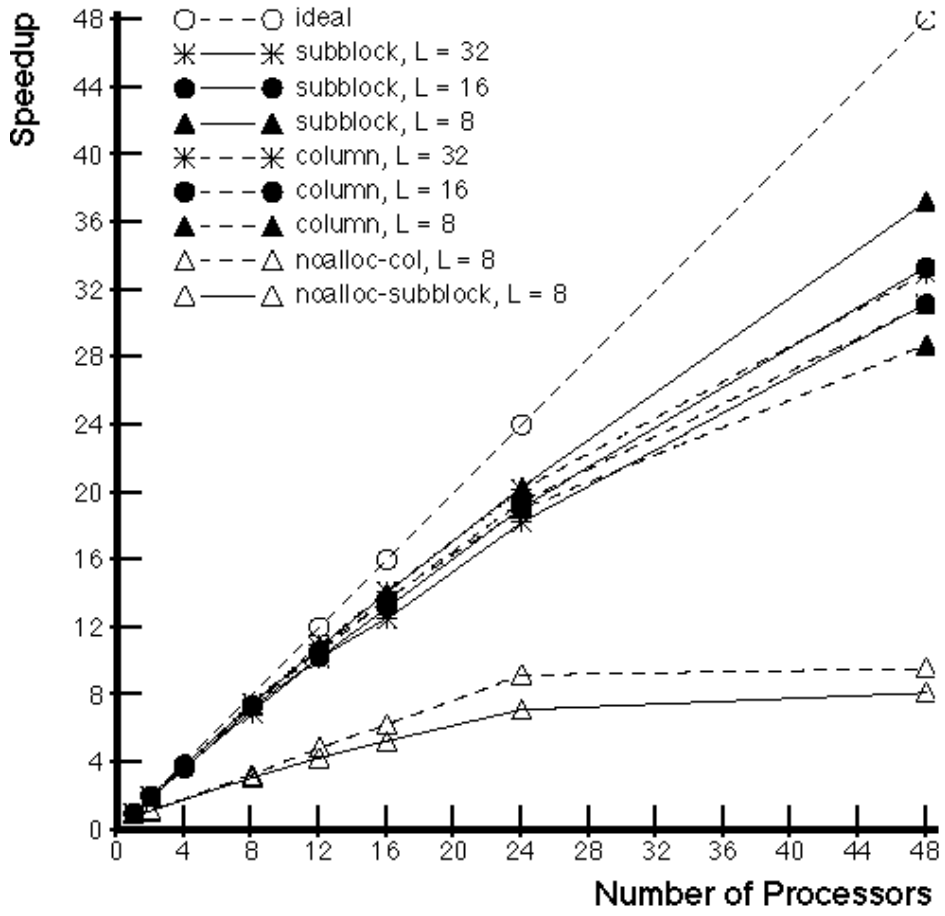
Two Static Partitioning Schemes

Impact of Memory Locality



algorithmic = perfect memory system; No Locality = dynamic assignment of columns to processors; Locality = static subgrid assigment (infinite caches)

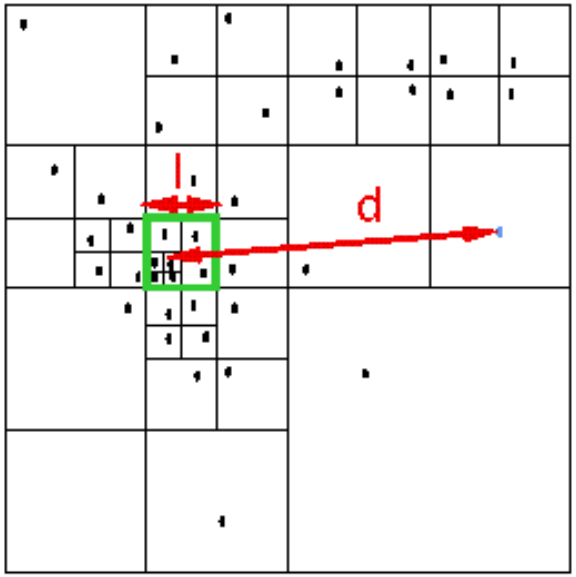
Impact of Line Size & Data Distribution



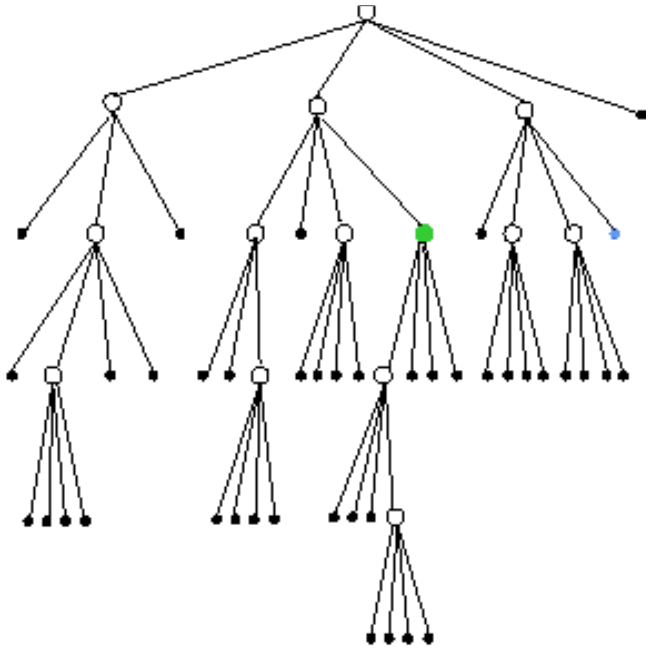
(a) 16 KByte Cache, Grid_98

no-alloc = round-robin page allocation; otherwise, data assigned to local memory.
L = cache line size.

Case Study 2: Barnes-Hut



2 d Spatial Domain



Quadtree Representation

Locality Goal:

- **particles close together in space should be on same processor**

Difficulties:

- **nonuniform, dynamically changing**

Load Balancing Challenges

- **Equal particles does mean equal work**

  Solution: assign costs to particles based on the work they do

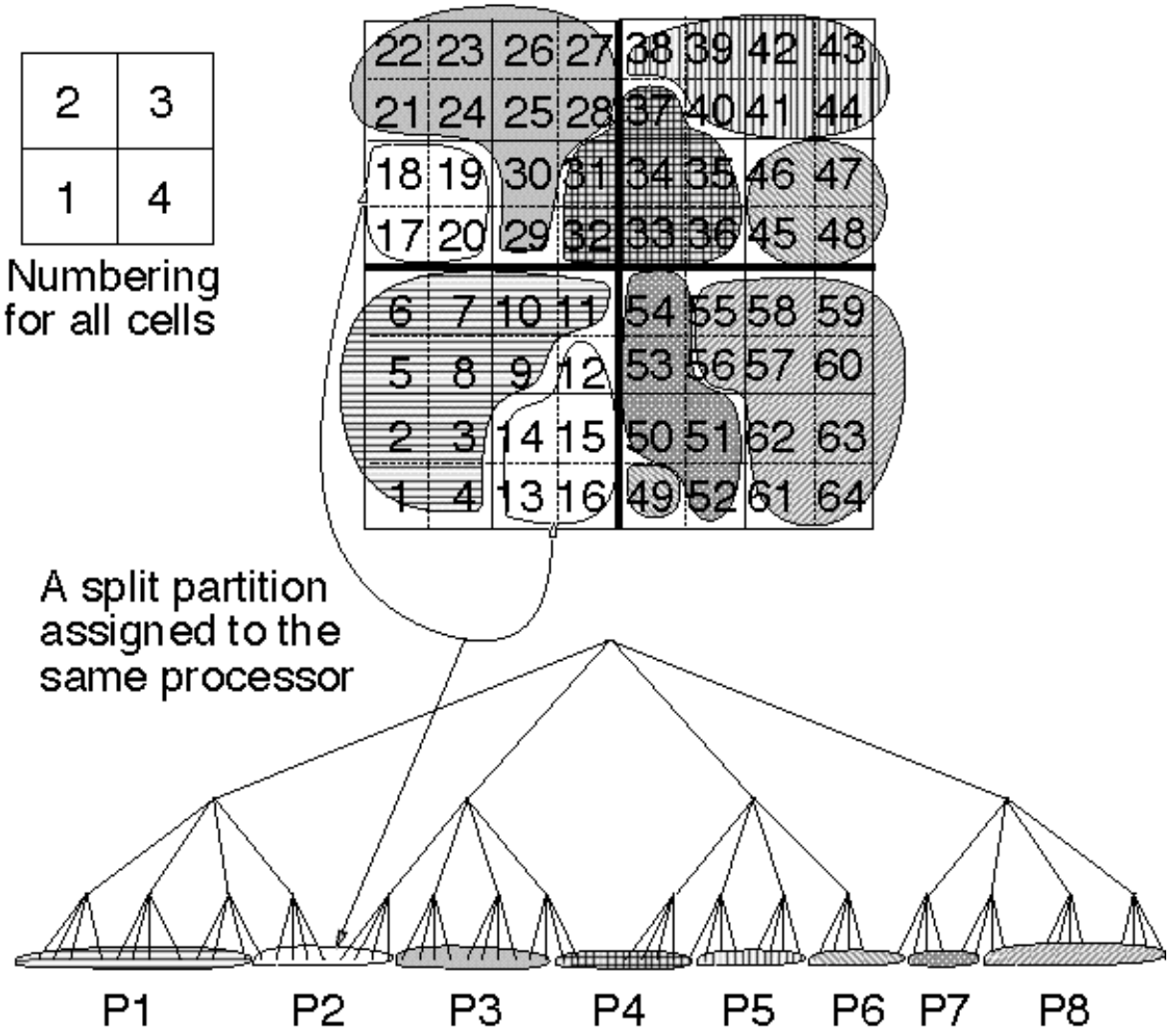
- **Work is unknown and changes with time-steps**

  Insight: system evolves slowly

  Solution: *count* work per particle, and use it as cost for next time-step

=> Powerful technique for evolving physical systems.

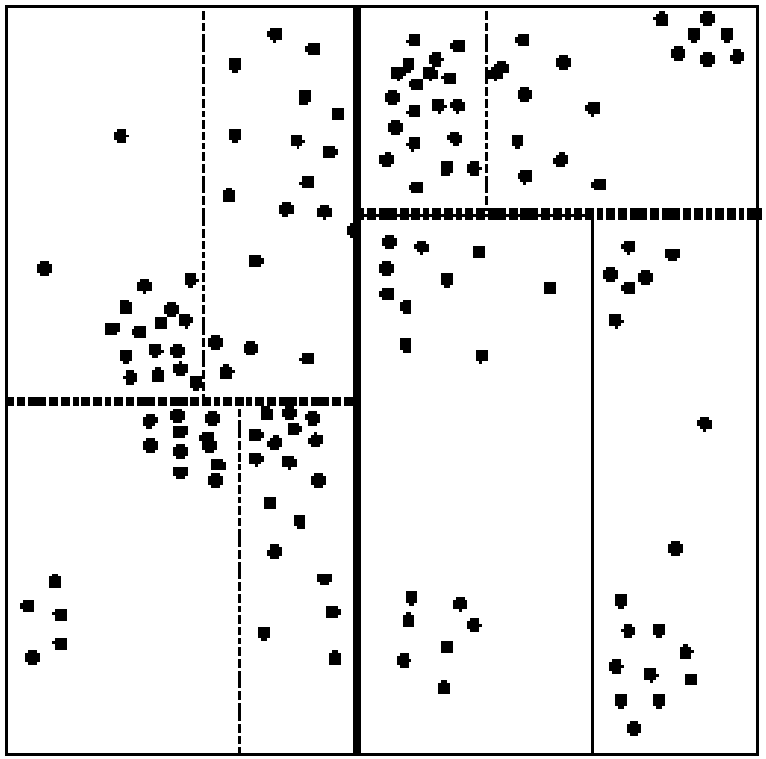
A Simple Partitioning: Costzones



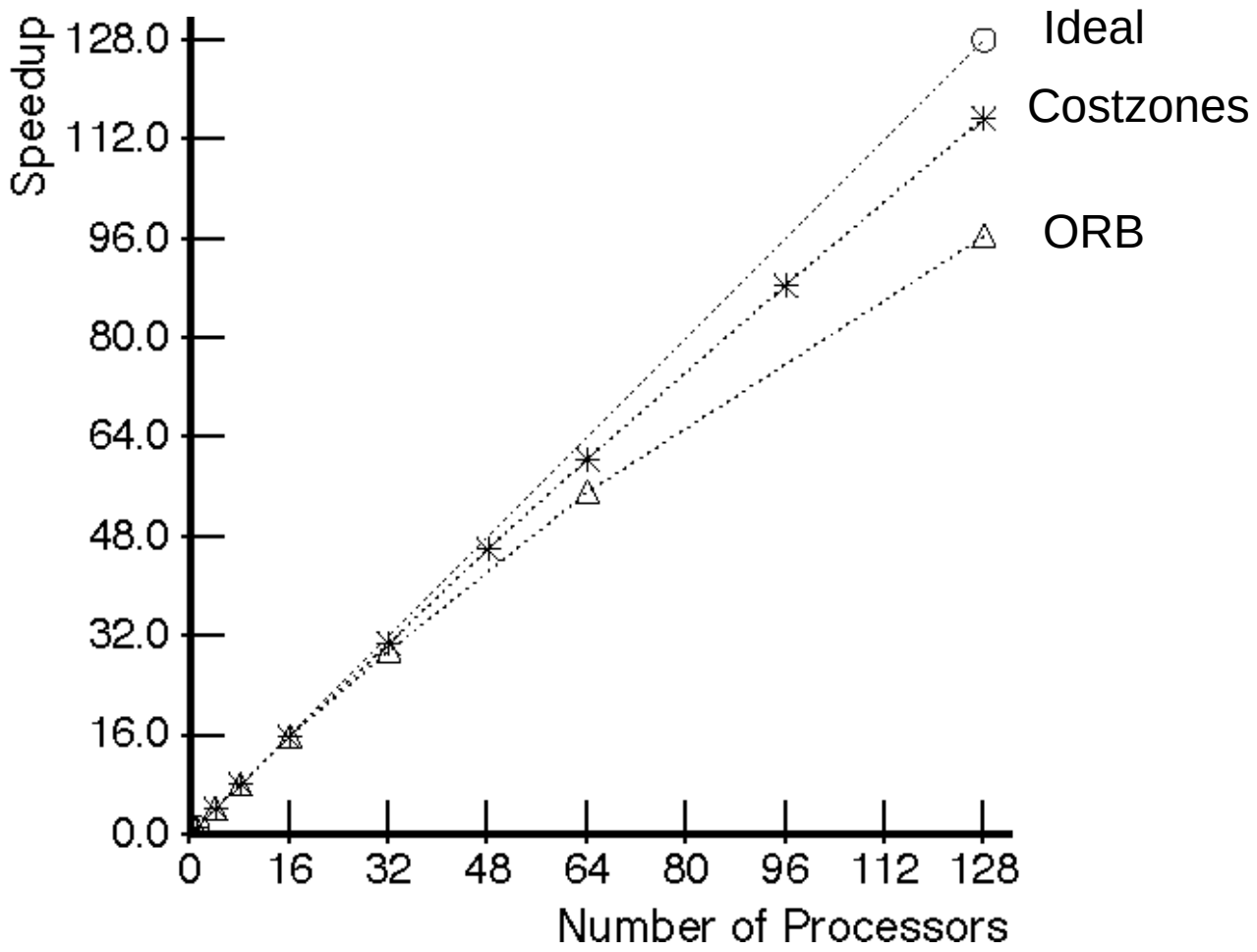
A Fancier Partitioning Scheme: ORB

Orthogonal Recursive Bisection (ORB):

- recursively bisect space into subspaces with equal work
- continue until one partition per processor



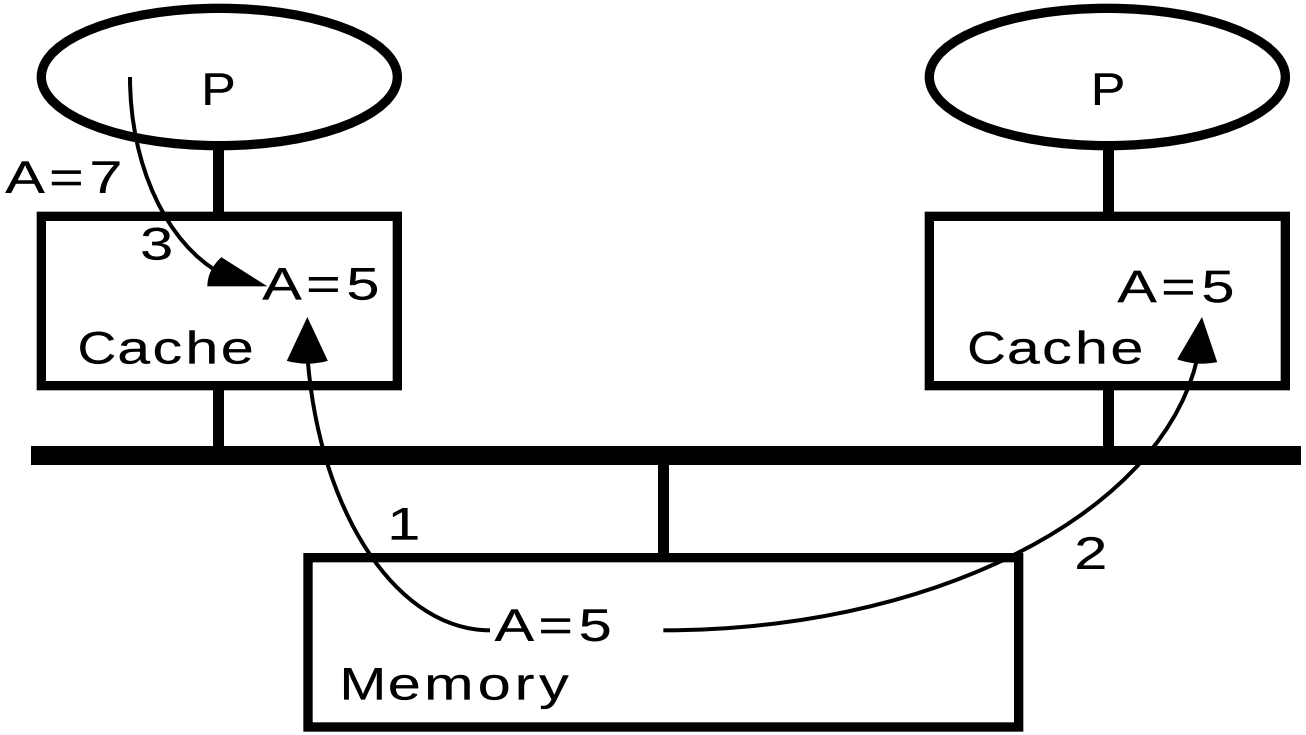
Barnes-Hut Performance



Speedups on simulated multiprocessor

The Cache Coherence Problem

- **Caches are critical to modern high-speed processors**
- **Multiple copies of a block can easily get inconsistent**
  - **• processor writes, I/O writes, ...**



Cache Coherence Solutions

Software Based:

- **typically used in clusters of workstations of PCs (e.g., “Treadmarks”)**
- **extend virtual memory system to perform more work on page faults**
  - send messages to remote machines if necessary

Hardware Based:

- **two most common variations:**
  - “snoopy” schemes
    - » rely on broadcast to observe all coherence traffic
    - » well suited for buses and small-scale systems
    - » example: SGI Challenge
  - directory schemes
    - » uses centralized information to avoid broadcast
    - » scales well to large numbers of processors
    - » example: SGI Origin 2000

Snoopy Cache Coherence Schemes

- **A distributed cache coherence scheme based on the notion of a snoop that watches all activity on a global bus, or is informed about such activity by some global broadcast mechanism.**
- **Most commonly used method in commercial multiprocessors.**
- **Examples: Encore Multimax, Sequent Symmetry, SGI Challenge, SUN Galaxy, ...**

Write-Through Schemes

All processor writes result in:

- **update of local cache and a global bus write that:**
  - updates main memory
  - invalidates/updates all other caches with that item

Examples:

- **early Sequent and Encore machines.**

Advantage:

- **simple to implement**

Disadvantages:

- **Since ~15% of references are writes, this scheme consumes tremendous bus bandwidth. Thus only a few processors can be supported.**

Write-Back/Ownership Schemes

- **When a single cache has ownership of a block, processor writes do not result in bus writes, thus conserving bandwidth.**
- **Most bus-based multiprocessors use such schemes these days.**
- **Many variants of ownership-based protocols exist:**
  - Goodman's write-once scheme
  - Berkeley ownership scheme
  - Firefly update protocol
  - ...

Goodman's Write-Once Scheme

One of the first write-back schemes proposed

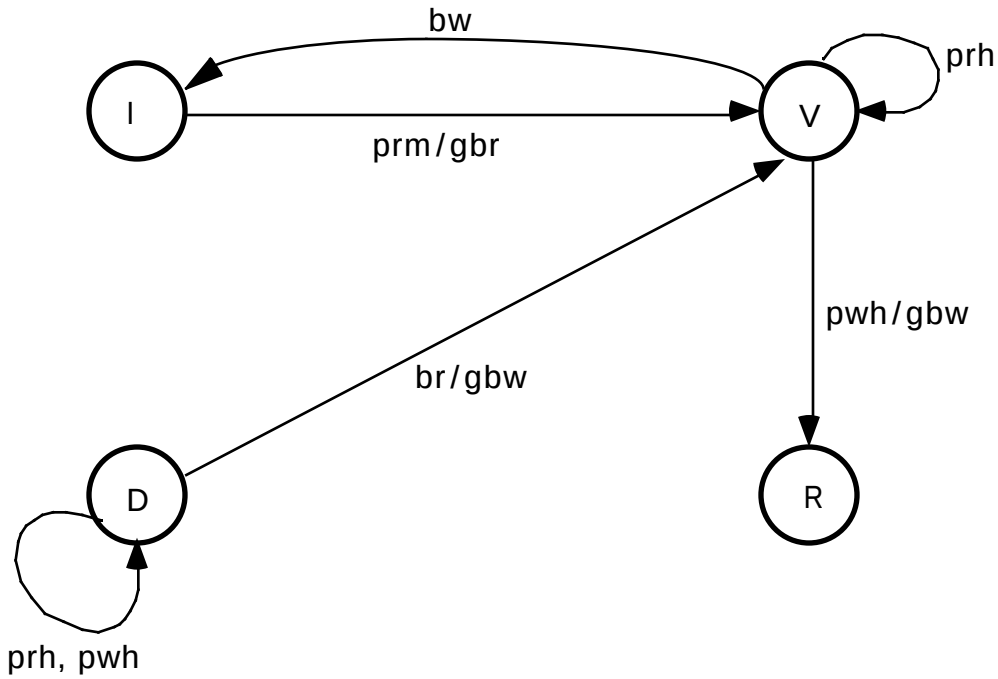
Classification: Write-back, invalidation-based

States:

- **I: invalid**
- **V: valid ==> data is clean and possibly in "V" state in multiple PEs**
- **R: reserved ==> owned by this cache, but main memory is up-to-date**
- **D: dirty ==> owned by this cache, and main memory is stale**

- **Cache sees transactions from two sides: (i) processor and (ii) bus.**

- **Terminology:**
  - prm, prh, pwm, pwh: processor read miss/hit, write miss/hit
  - gbr, gbri, gbw, gbi: generate bus read, read-with-inval, bus write, inval
  - br, bri, bw, bi: bus read, read-with-inval, write, inval observed



Illinois Scheme (J. Patel)

States:

- I, VE (valid-exclusive), VS (valid-shared), D (dirty)

Two features:

- The cache knows if it has an valid-exclusive (VE) copy. In VE state, no invalidation traffic on write-hits.
- If some cache has a copy, cache-cache transfer is used.

Advantages:

- closely approximates traffic on uniprocessor for sequential pgms
- in large cluster-based machines, cuts down latency (e.g., DASH)

Disadvantages:

- complexity of mechanism that determines exclusiveness
- memory needs to wait before sharing status is determined

DEC Firefly Scheme

Classification:

- Write-back, update

States:

- VE (valid exclusive): only copy and clean
- VS (valid shared): shared-clean copy. Write-hits result in updates to other caches and entry remains in this state
- D (dirty): dirty exclusive (only copy)

Used special "shared line" on bus to detect sharing status of cache line

Advantage:

- Supports producer-consumer model well

Disadvantage:

- What about sequential processes migrating between CPUs?

Invalidation vs. Update Strategies

Retention strategy: When to drop block from cache

- **1. Exclusive writer (inval-based): Write causes others to drop.**
- **2. Pack rat (update-based): Block dropped only on conflict.**

Exclusive writer is bad when:

- **single producer and many consumers of data (e.g., bound in TSP).**

Pack rat is bad when:

- **multiple writes by one PE before data is read by another PE (e.g., supernode-to-column update in panel cholesky).**
- **junk data accumulates in large caches (e.g., process migration).**

Overall, invalidation schemes are more popular as the default.

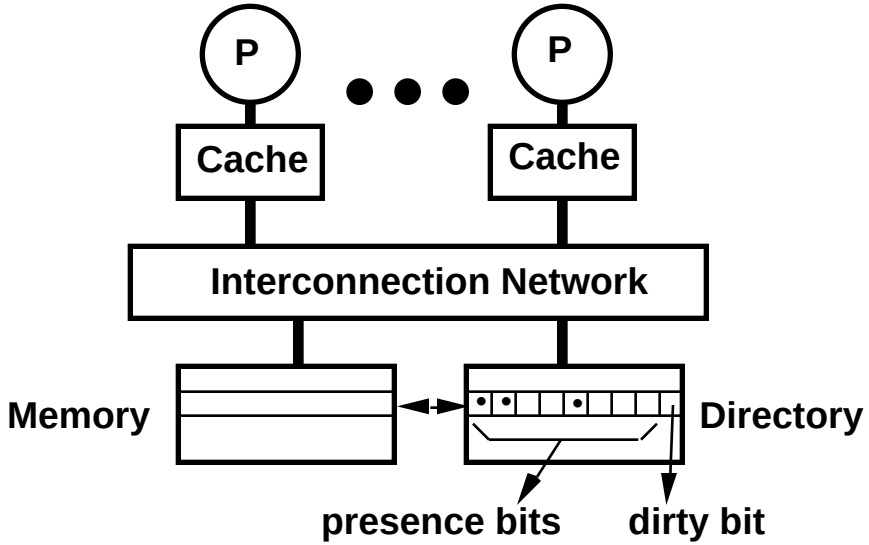
Motivation for Directory Schemes

Snoopy schemes do not scale because they rely on broadcast

Directory-based schemes allow scaling.

- **they avoid broadcasts by keeping track of all PEs caching a memory block, and then using point-to-point messages to maintain coherence**
- **they allow the flexibility to use any scalable point-to-point network**

Basic Scheme (Censier & Feautrier)



- **Assume "k" processors.**
- **With each cache-block in memory: k presence-bits, and 1 dirty-bit**
- **With each cache-block in cache: 1valid bit, and 1 dirty (owner) bit**

- **Read from main memory by PE-i:**
  - If dirty-bit is OFF then { read from main memory; turn p[i] ON; }
  - if dirty-bit is ON then { recall line from dirty PE (cache state to shared); update memory; turn dirty-bit OFF; turn p[i] ON; supply recalled data to PE-i; }
- **Write to main memory:**
  - If dirty-bit OFF then { supply data to PE-i; send invalidations to all PEs caching that block; turn dirty-bit ON; turn P[i] ON; ... }
  - …

Key Issues

Scaling of memory and directory bandwidth

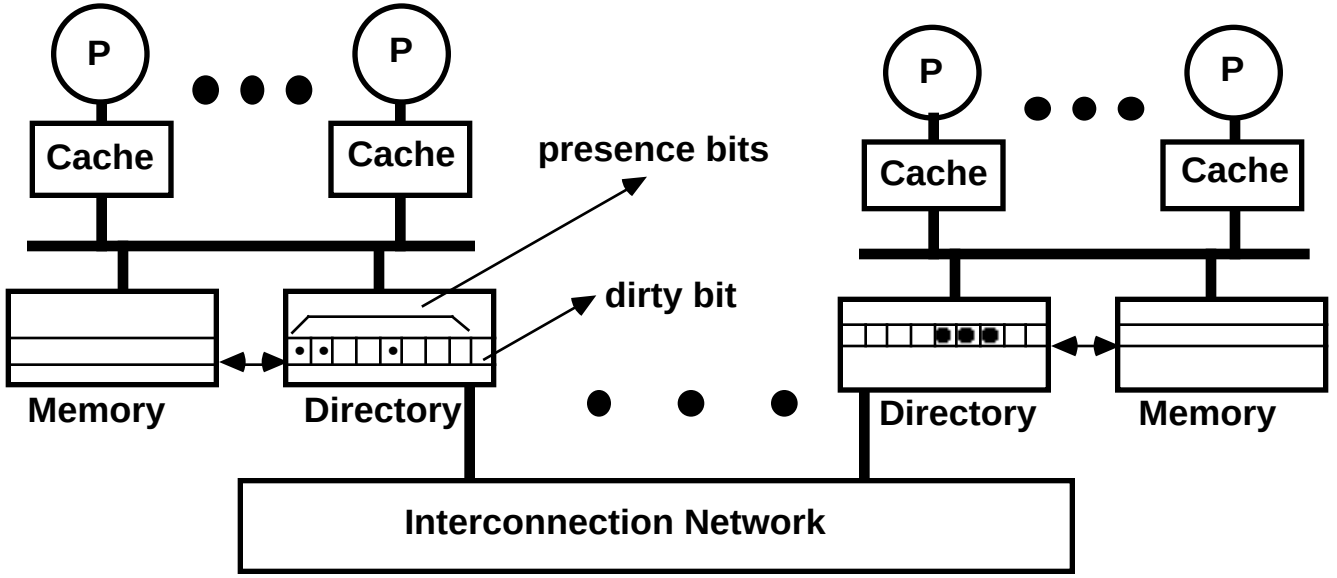
- **Can not have main memory or directory memory centralized**
- **Need a distributed cache coherence protocol**

As shown, directory memory requirements do not scale well

- **Reason is that the number of presence bits needed grows as the number of PEs**
- **In reality, there are many ways to get around this problem**
  - limited pointer schemes of many flavors

The Stanford DASH Architecture

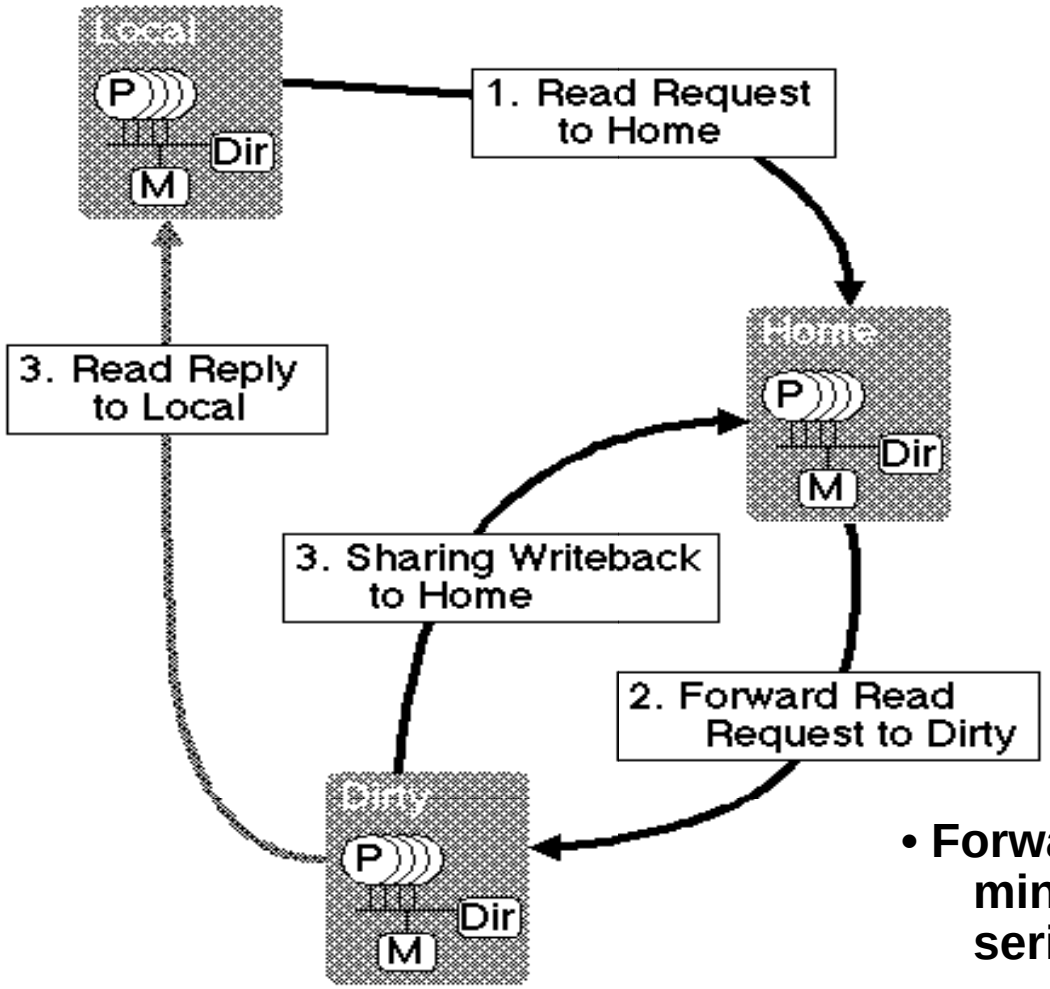
DASH ==> Directory Architecture for SHared memory



- **Nodes connected by scalable interconnect**
- **Partitioned shared memory**
- **Processing nodes are themselves multiprocessors**
- **Distributed directory-based cache coherence**

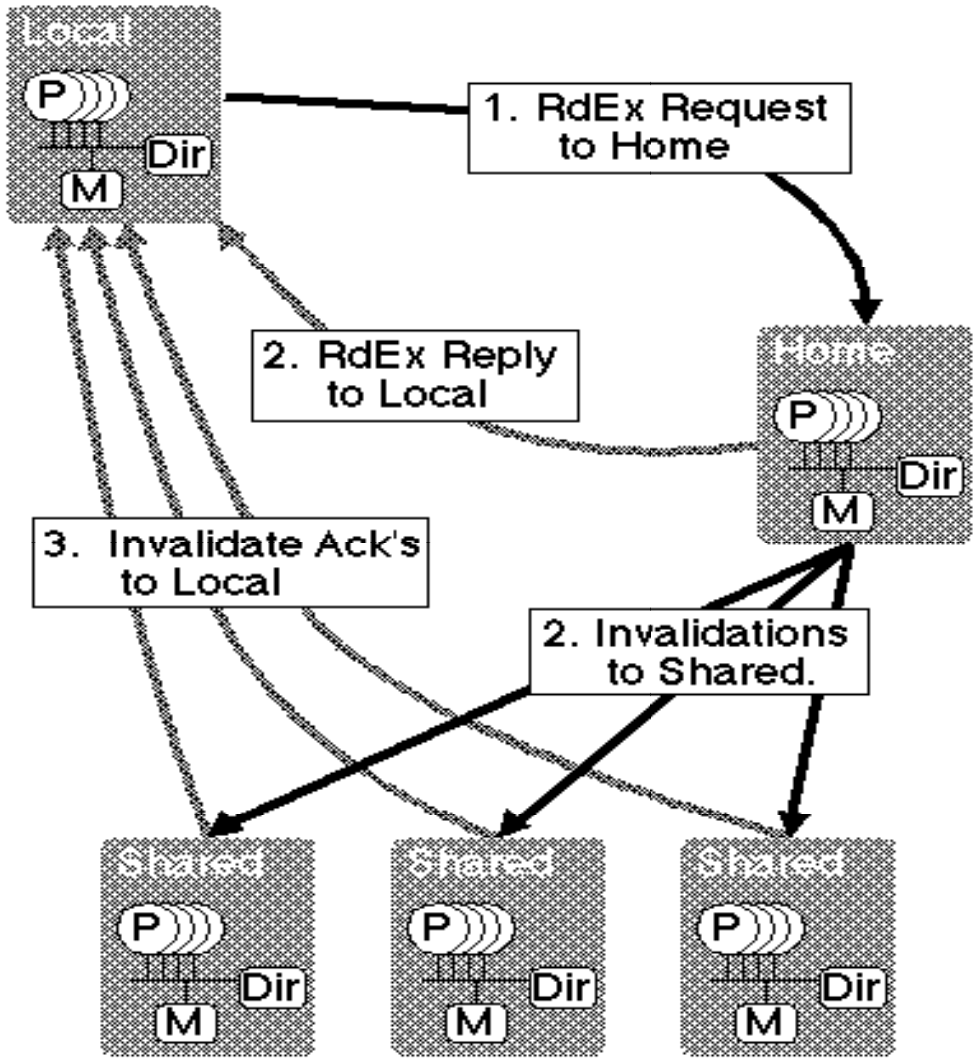
Directory Protocol Examples

- **Read of remote-dirty data**



- **Forwarding strategy minimizes latency and serialization**

Write (Read-Exclusive) to Shared Data

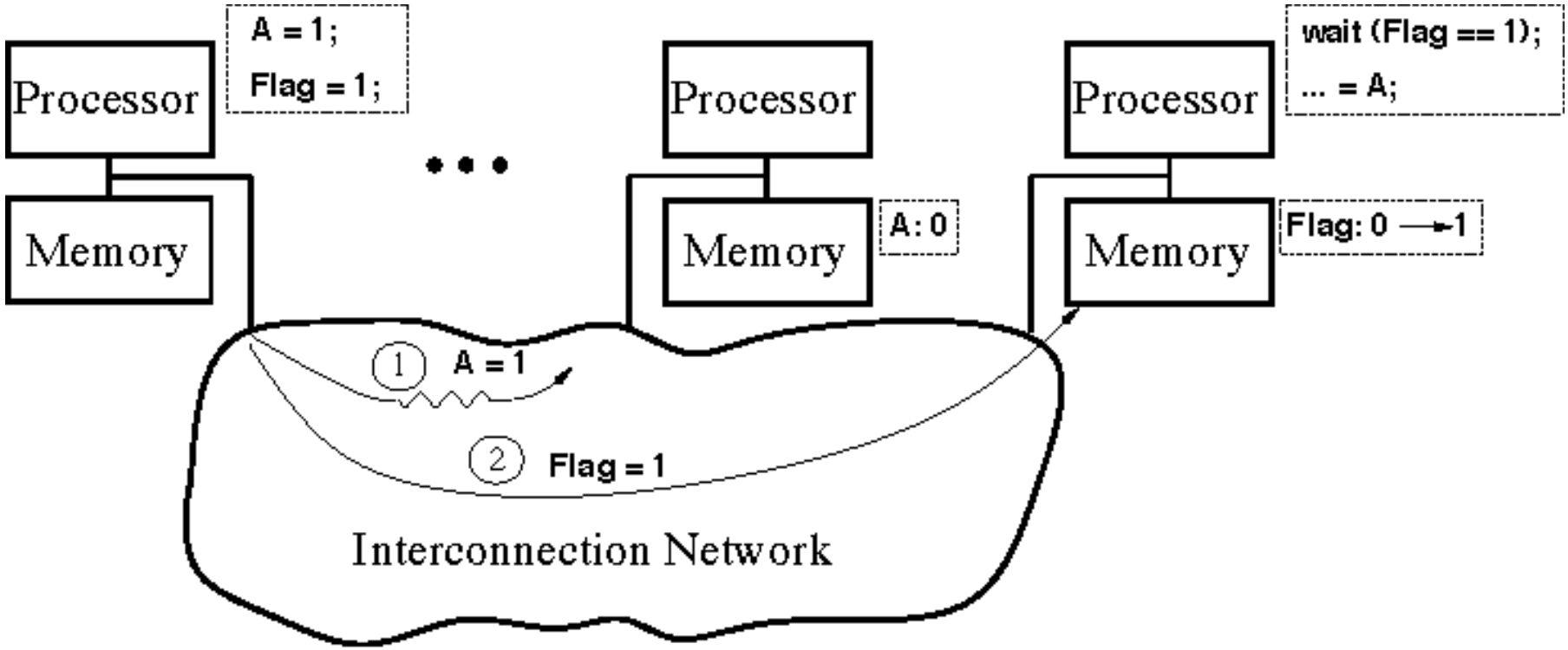


Memory Consistency Models

- **Order between accesses to different locations becomes important**

P1	P2
A = 1;	
Flag = 1;	wait (Flag ==
	... = A;

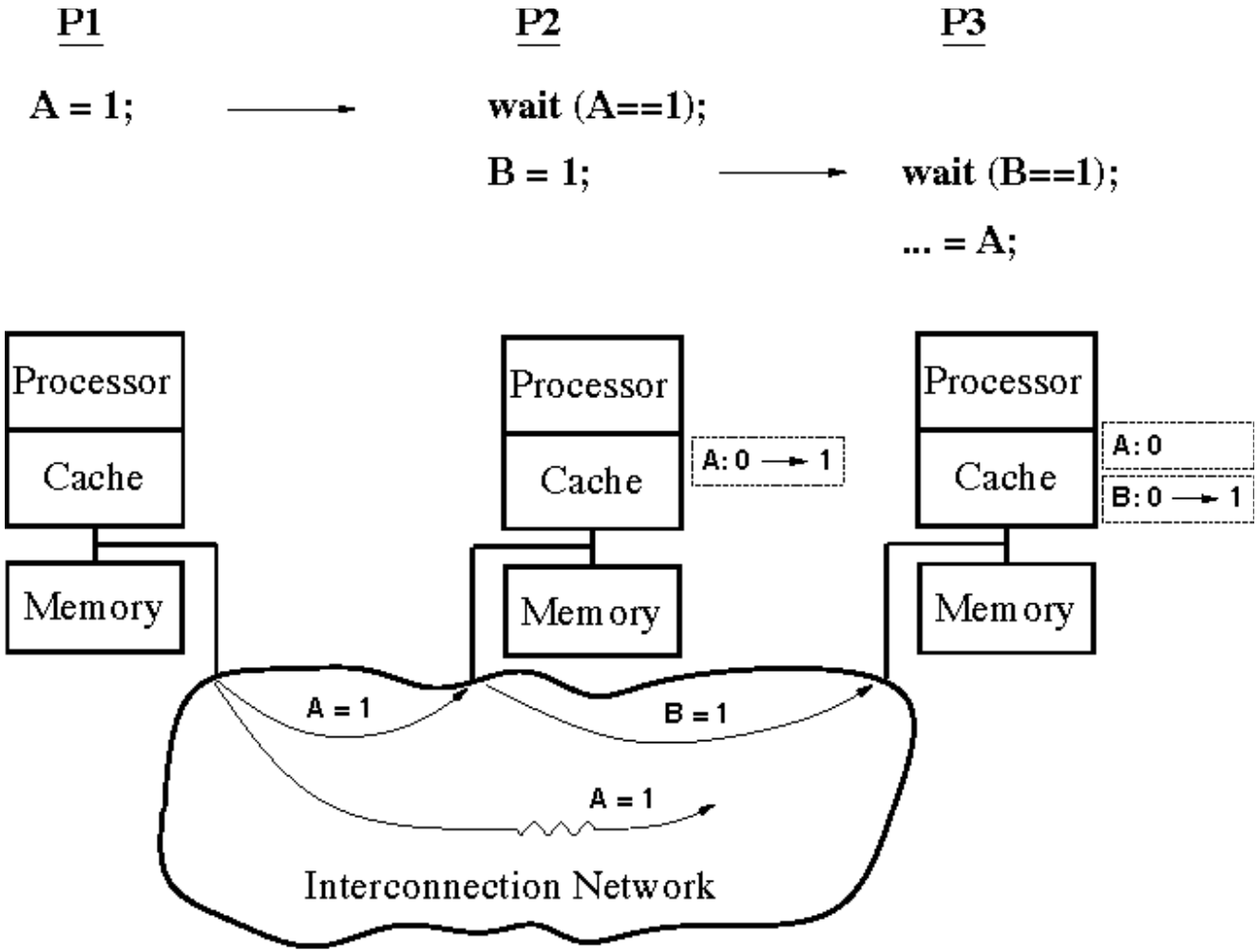
How Unsafe Reordering Can Happen



- **Distribution of memory resources**
  - access issued in order may be observed out of order

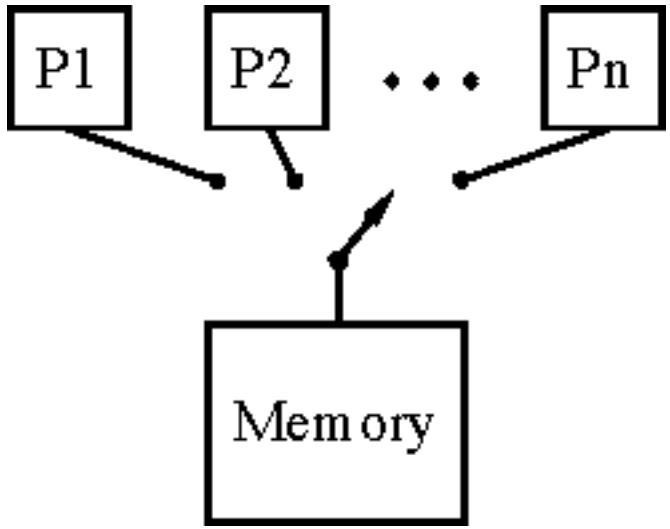
Caches Complicate Things More

- **Multiple copies of the same location**



Sequential Consistency

- **Formalized by Lamport (1979):**
  - accesses within each processor performed in program order
  - accesses across processors correspond to some sequential interleaving

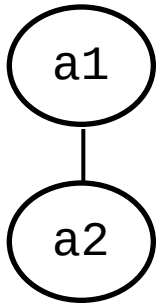


- **Any order implicitly assumed by the programmer is maintained**

Sequential Consistency Example

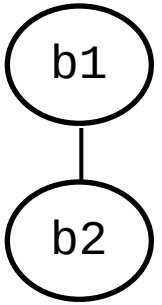
Process A

```
a1:  x = 1
a2: if (y == 0) …
```

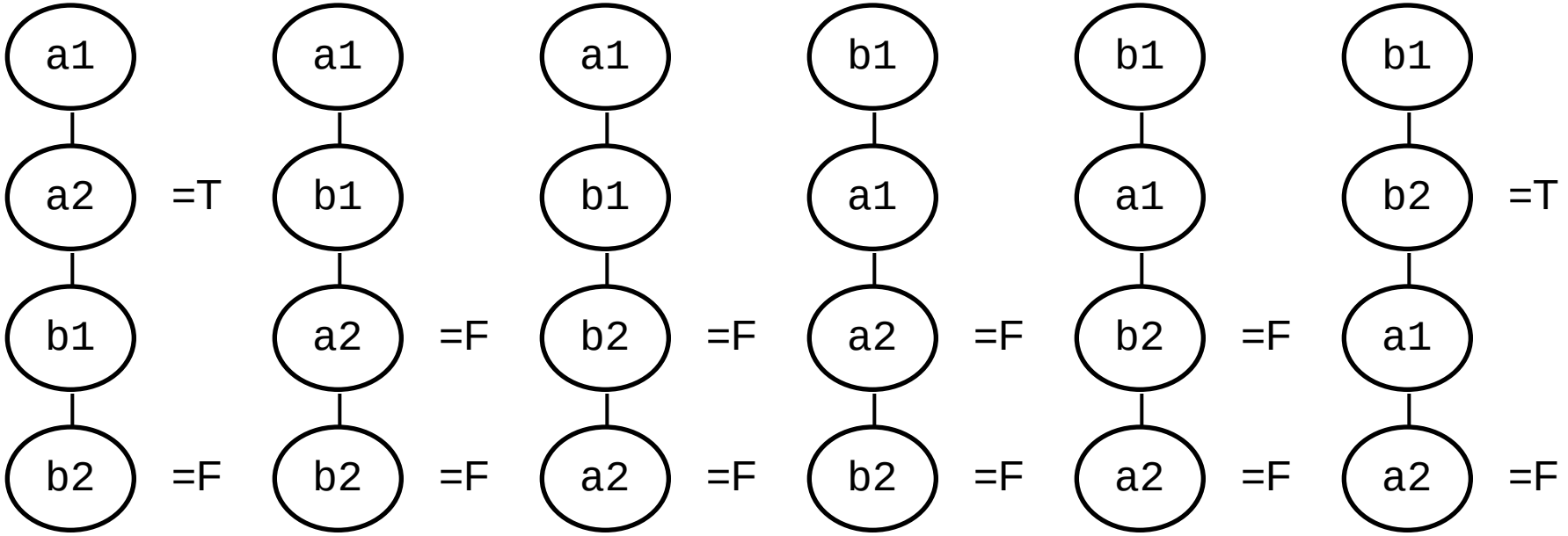


Process B

```
b1:  y = 1
b2: if (x == 0) …
```



Possible Interleavings



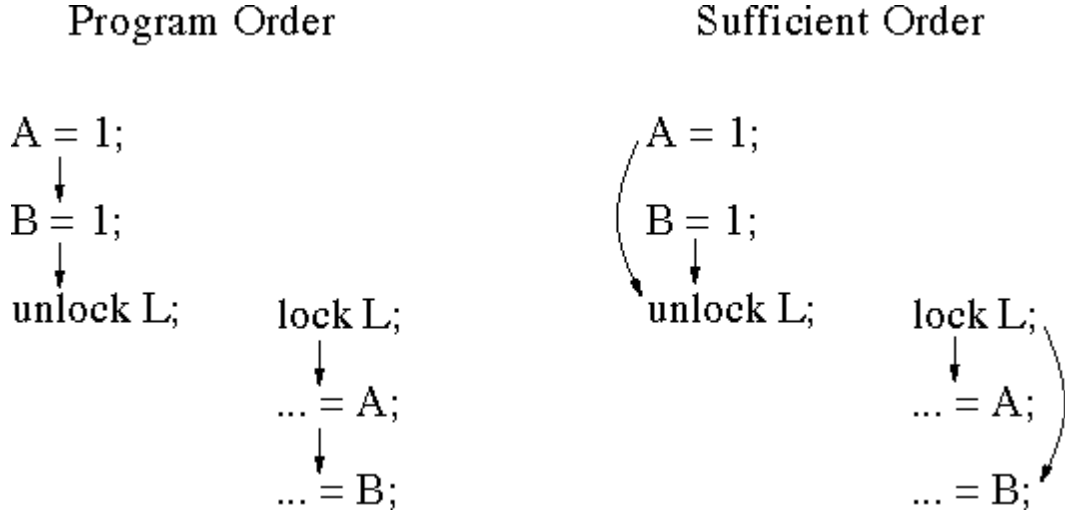
Relaxed Consistency Models

Motivation:

- **while Sequential Consistency (SC) is intuitive for programmers, it severely inhibits hardware flexibility and performance**
  - even in 10-year-old processor technology

Relaxing the ordering constraints:

- **"correctness" = same result as SC**
- **most programs don't require strict ordering for "correctness"**



Consistency: How It Affects You

Most modern processors *do not* support sequential consistency!

- **the MIPS R10000 is a rare exception**

Instead, you must properly synchronize your program

- **i.e. use locks, barriers, etc. whenever inter-processor ordering is necessary**
- **these synchronization primitives are visible to the hardware, and it knows how to do the right thing when it seems them**
  - e.g., flush all outstanding stores before an unlock, stall all loads until after a lock completes, etc.

If you use regular memory locations for synch. rather than explicit synch. primitives, it is unlikely that your program will function properly.

Lesson#1: Use Commodity Microprocessor

Build on their Economy of Scale

- **Thousands of engineering person-years**
- **Efficient manufacturing**

Failures

- **KSR: 15 MHz processor, designed by 3 person team**
- **Tera: > 2 years late shipping first system**

Successes

- **Cray T3D, T3E uses Alpha, fastest processor made**
- **SMP's based on MIPS, SPARC, PentiumPro, …**
  - Snoopy protocol hardware integrated into chip

Lesson #2: Provide Shared Memory Interface

Natural Programming Model for Many Applications

- **Shared database**
  - Can access different parts of data base by memory references
- **Task scheduling**
  - Processors grab tasks from shared queue
  - Get code & data via memory references

Can Simulate Other Models Effectively

- **Message-passing**
  - Store data in buffer allocated in distant memory

Exploits Memory Management Technology

- **OS involved in setting up & allocating resources**
- **Once allocated, communication proceeds without OS intervention**

Attack of the Killer Clusters?

Workstations Good for High Performance Computing

- **Easy to program**
- **Low cost per compute cycle**
- **Low unit cost allows frequent upgrades**

Parallel Processors at Disadvantage

- **Takes work to make application run well on parallel machine**
- **Parallel machines big & expensive**
- **Higher unit costs & longer development times**
  - Hard to upgrade processors
  - Even harder to upgrade network or bus

Viable Strategies

- **Minimal extension of workstation**
- **Stick with applications requiring supercomputer performance**