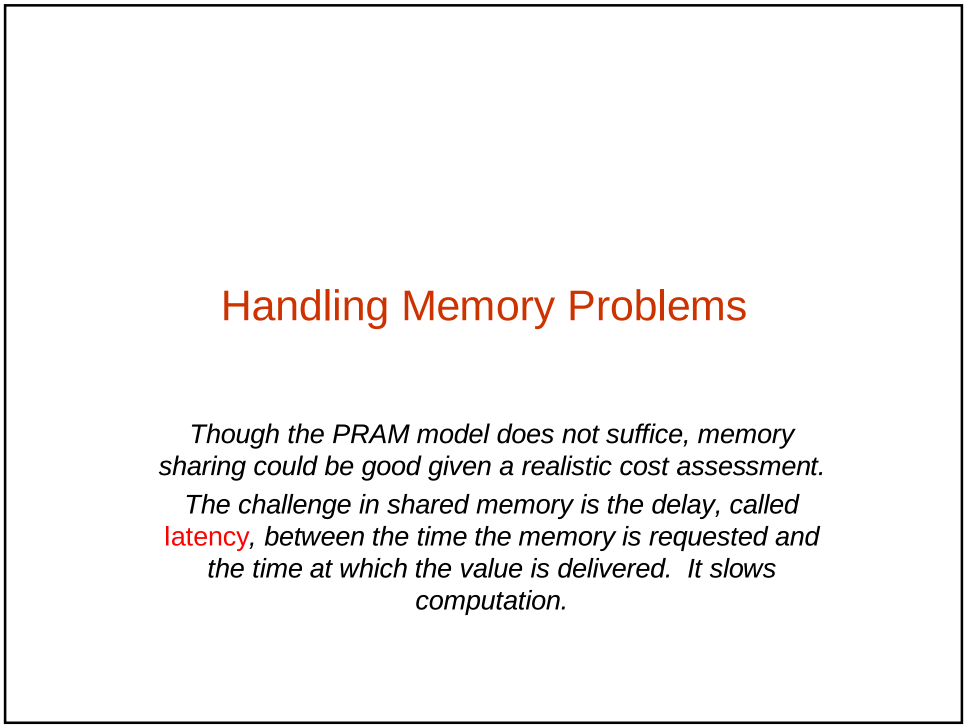
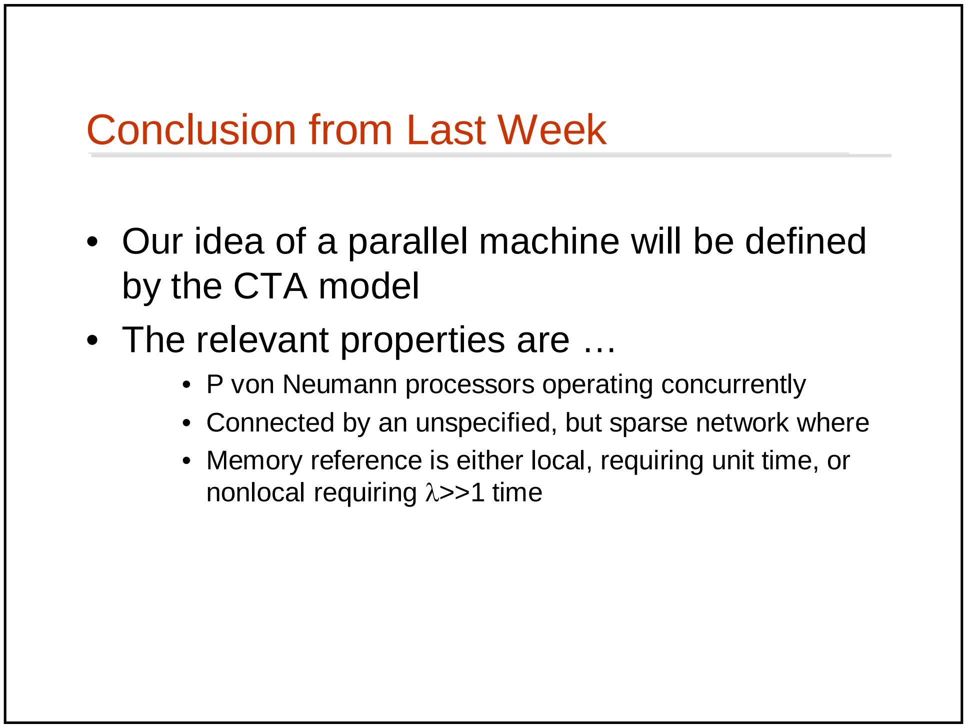


Handling Memory Problems

Though the PRAM model does not suffice, memory sharing could be good given a realistic cost assessment.

The challenge in shared memory is the delay, called latency*, between the time the memory is requested and the time at which the value is delivered. It slows computation.*



Conclusion from Last Week

- Our idea of a parallel machine will be defined by the CTA model
- The relevant properties are …
  - P von Neumann processors operating concurrently
  - Connected by an unspecified, but sparse network where
  - Memory reference is either local, requiring unit time, or nonlocal requiring λ>>1 time

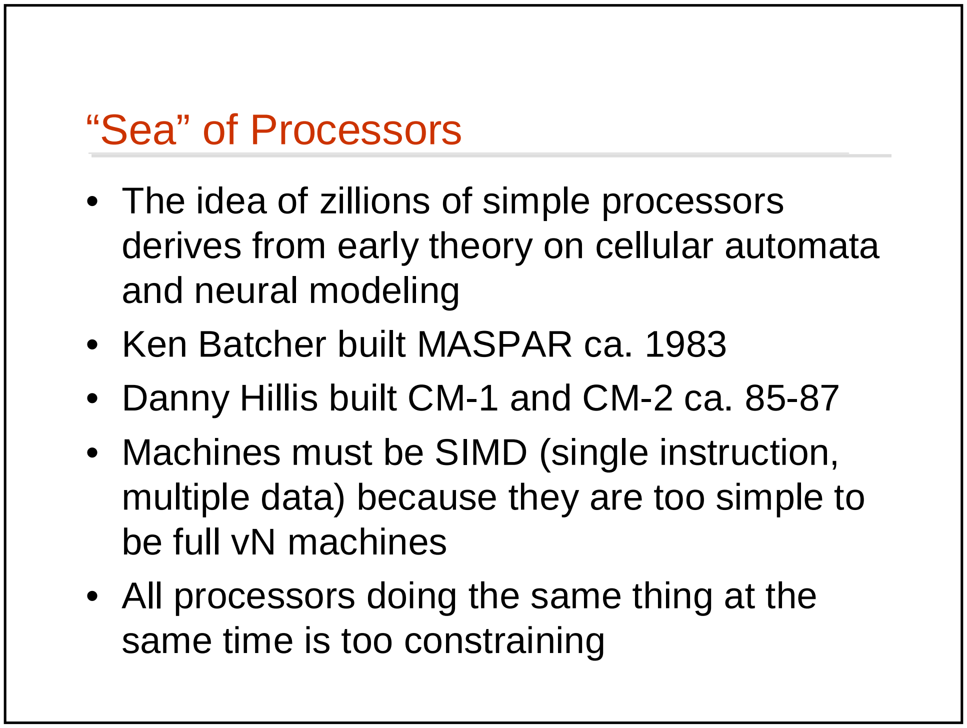
Sharing ...

- There are two aspects to sharing memory
  - (True) shared memory means every processor can reference every memory location of a “single coherent memory,” i.e every processor sees the same values
  - Shared address space means every processor can reference every memory location in the machine but the memory is not kept coherent

The problem: Suppose in time sequence
Processor A reads location 1000
Processor B writes location 1000
Processors A and C use 1000 in later computation
When A, C use values are they the same or different?

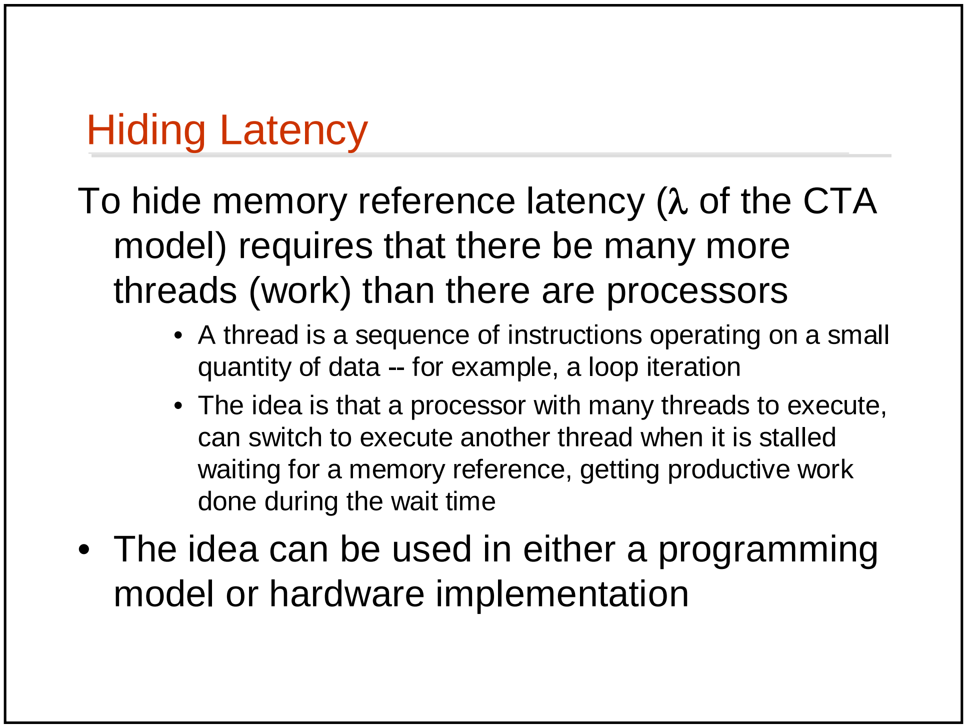
Memory Sharing on a CTA

- With many processors referencing a single memory image, the probability that a given reference is nonlocal is (P-1)/P
- So, by the CTA nearly all memory references will take λ time implying that each processor can do little work
- Responses
  - A “sea” of small processors that can be used “inefficiently”
  - Have many threads for processors to switch among
  - Find a way to greatly reduce λ
  - Change processors from vN to something else



“Sea” of Processors

- The idea of zillions of simple processors derives from early theory on cellular automata and neural modeling
- Ken Batcher built MASPAR ca. 1983
- Danny Hillis built CM-1 and CM-2 ca. 85-87
- Machines must be SIMD (single instruction, multiple data) because they are too simple to be full vN machines
- All processors doing the same thing at the same time is too constraining



Hiding Latency

To hide memory reference latency (λ of the CTA model) requires that there be many more threads (work) than there are processors

  - A thread is a sequence of instructions operating on a small quantity of data -- for example, a loop iteration
  - The idea is that a processor with many threads to execute, can switch to execute another thread when it is stalled waiting for a memory reference, getting productive work done during the wait time

- The idea can be used in either a programming model or hardware implementation

Latency Hiding In Programming Model

- Bulk-synchronous programming (BSP) is a solution by Valiant
- Computation executes in *supersteps*:
  - Assume threads execute 3-address code `a:=b op c;`

    [Load] Fetch operands from memory for many threads

    [Compute] For all threads having available operands compute `a:=b op c`

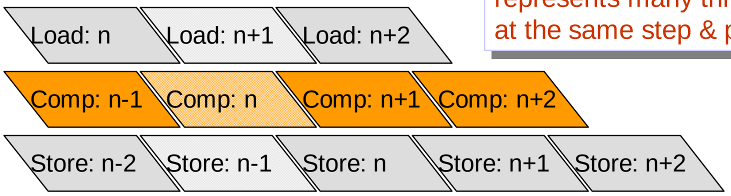
    [Store] Return the result to memory
- With many threads, there is compute work enough to hide the data transmission time

Each thread may not execute on each cycle

BSP Skews A Synchronous Step in Time

- Consider a series of steps

A parallelogram represents many threads at the same step & phase



Load: n

Load: n+1

Load: n+2

Comp: n-1

Comp: n

Comp: n+1

Comp: n+2

Store: n-2

Store: n-1

Store: n

Store: n+1

Store: n+2

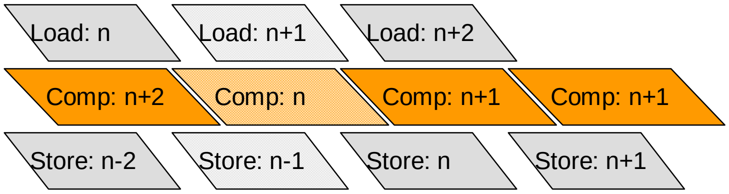
- Enough operands must be shipped on step n to *enable* enough threads at computation step n to cover the latency of the store of n-1 and the fetch of n+1

Notice that the strategy ignores locality

Many Threads = More Parallelism

When enabled-threads are numerous, the processors keep busy; “compute bound”

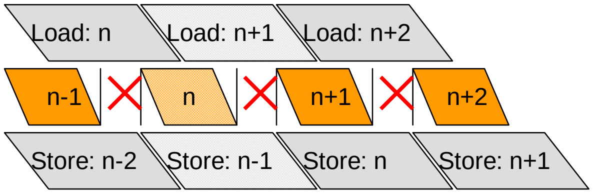
- More parallelism exists than can be used by processors
- Communication subsystem operates “below saturation”



MM is example: 2m values imply m^2 operations

Too Few Threads = Waiting

- When the enabled threads are too few to cover the latency, processors finish computing before next data arrives
  - Not enough parallelism
  - Communication subsystem may be less efficient



Theoretically, P log P threads are needed, minimum

Parallel Slackness

- Valiant called the amount of “excess” parallelism needed to cover latency *parallel slackness*
- In the “best case” a parallel slackness of logP is required because in the best case latency will be proportional to log P
- Any additional delays require further slackness

Problems With BSP

- Though threads are often very numerous, it is difficult always to have P log P available
- Other considerations
  - The network needs to have high bandwidth (explained in later lecture, but meaning O(n) bisection bandwidth)
  - Network congestion can be magnified when operating at peak capacity
  - Memory contention requires that a value be fetched from each processor’s memory at each cycle …what happens when multiple values are in the same memory unit?
  - What are the implications of slow memories and fast processors?

Reducing λ

- Reducing λ requires good engineering and some clever programming
- NYU's Ultracomputer Group tried both using 2-3 very slick ideas
- Though they were ultimately unsuccessful, the approach stands as a fair test of a true shared memory parallel system

J.T. Schwartz, *Ultracomputers*, ACM Transactions on Programming Languages and Systems 2(4):484-521, 1980

Historical caution ...

- Jack Schwartz wrote a paper called *Ultracomputer* in which he observed
  - The ideal parallel computer would be a PRAM (he called it a paracomputer), but it can't work
  - The realistic alterative would be an "ultracomputer" which was a specific (CTA-type architecture) using a shuffle-ex network, vN processors, etc.
  - Allen Gottlieb started a project to build an ultracomputer, but after getting started, i.e. after the computer was already named, decided they'd build a paracomputer

Although the original ultracomputer doesn't have a shared memory the machine that was built did … confusing

An Alternate Concurrency Primitive

- Rather than using the Test&Set to guard shared data, use Fetch&Add
  - Fetch&Add is an atomic read-modify-write operation on memory -- requires special hardware, to be discussed
  - Use Fetch&Add as a semaphore and as a scheduler
- Operation: Fetch&Add(V,e)
  - V is a memory location
  - e is an integer expression
  - Contents of V are returned
  - New value of V is V+e
  - Operation is atomic

V: 0

Fetch&Add(V,1)

V: 1

0 is returned

Concurrent Fetch&Adds

- When multiple Fetch&Adds are executed simultaneously, they are serializable
- Assume Fetch&Add(V, e1) and Fetch&Add(V, e2) are execute simultaneously
  - Assuming an initial value of e0
  - Final value is e0+e1+e2
  - The 1st process receives either e0 or e0+e2, implying it was first (e0) or second (e0+e2)
  - The 2nd process receives either e0 or e0+e1, implying it was second (e0+e1) or first (e0)

Suppose both execute Fetch&Add(I,1), then one gets I back, the other I+1, and final is I+2

Fetch&Add on Work List

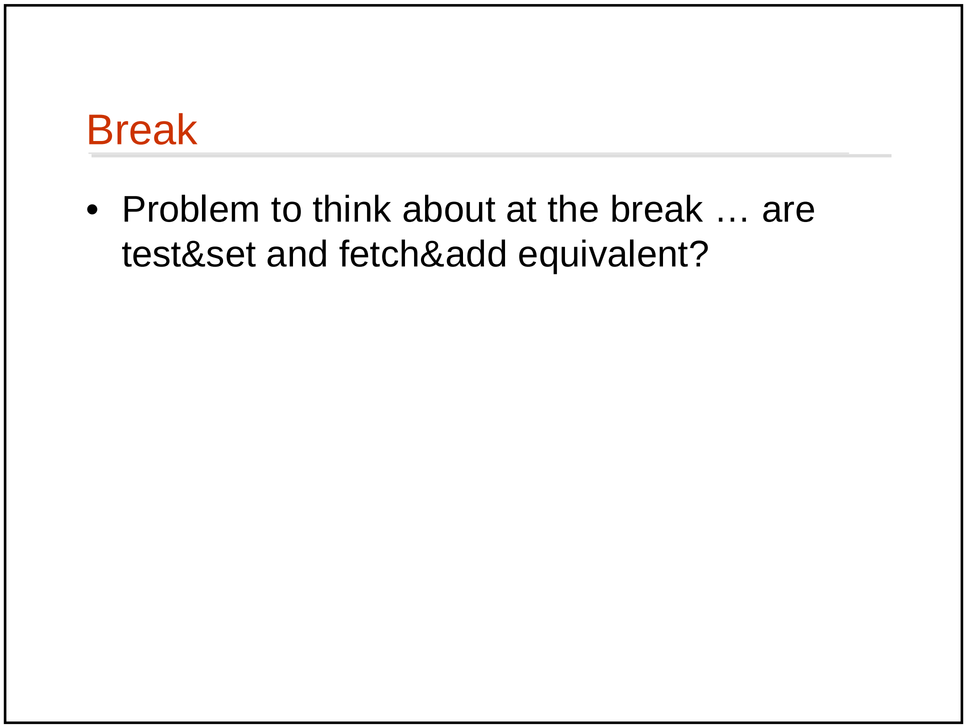
- Let `TD[1..n]` be a Todo list of TaskIDs
- Let `Next` be the index of the first unassigned task
- Processes execute
    - Fetch&Add(Next, 1)
    - Receive an index back
    - If index ≥ n, wrap around => Fetch&Add(Next,-n)

Such automatic scheduling can be useful for rows of an array or other ordered data structure

Generalizing Fetch&Add

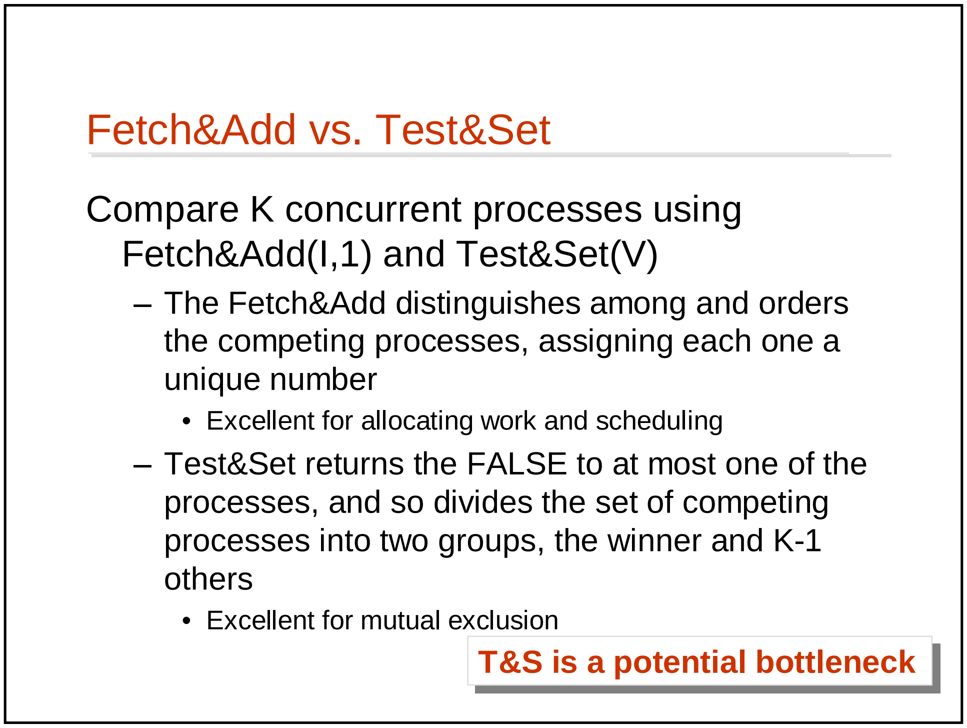
- Fetch&ϕ(V,e) is a generalization for arbitrary binary, associative operation ϕ
- Define ϕ(a,b) = a+b to define Fetch&Add( )
- If ϕ is associative, the final value is independent of the serializing order
- Test&Set(V) is just Fetch&Or(V, 1)
- Load and Store use π_1(a,b)=a, π_2(a,b)=b
    - Load R <-- Fetch&π_1(V,*)
    - Store * <-- Fetch&π_2(V,L)

* means value doesn’t matter



Break

- Problem to think about at the break … are test&set and fetch&add equivalent?



Fetch&Add vs. Test&Set

Compare K concurrent processes using Fetch&Add(I,1) and Test&Set(V)

- The Fetch&Add distinguishes among and orders the competing processes, assigning each one a unique number
  - Excellent for allocating work and scheduling
- Test&Set returns the FALSE to at most one of the processes, and so divides the set of competing processes into two groups, the winner and K-1 others
  - Excellent for mutual exclusion

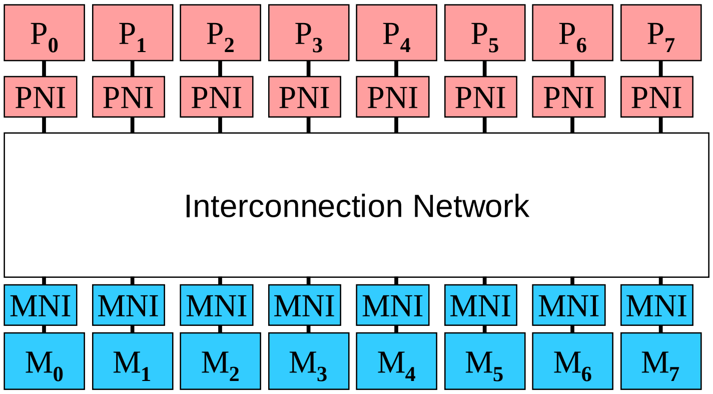
T&S is a potential bottleneck

Fetch&Add Exploits Sharing

- Though earlier solutions attempt to reduce sharing to reduce the amount of invalidation and acknowledgment, Fetch&Add does better with greater sharing
- Sharing is used to schedule or allocate, which is then independent activity
    - Sharing is concentrated in a few variables
    - Fine grain size is possible
- Since load/store, Test&Set, etc. are implementable, it is a sufficient primitive

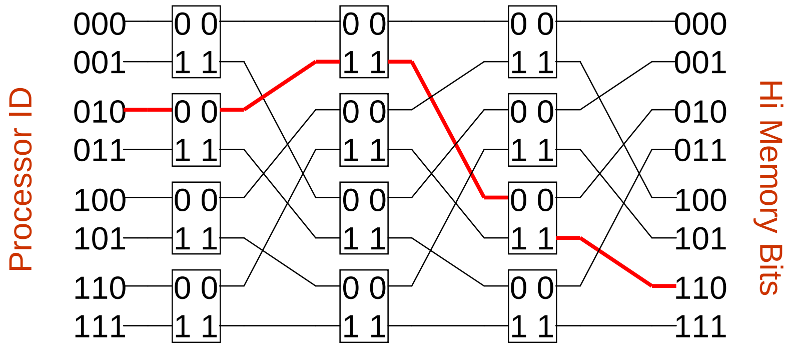
Implementing Fetch&Add

- Fetch&Add assumes a flat shared memory as implemented by a “dance hall architecture”



Omega Network

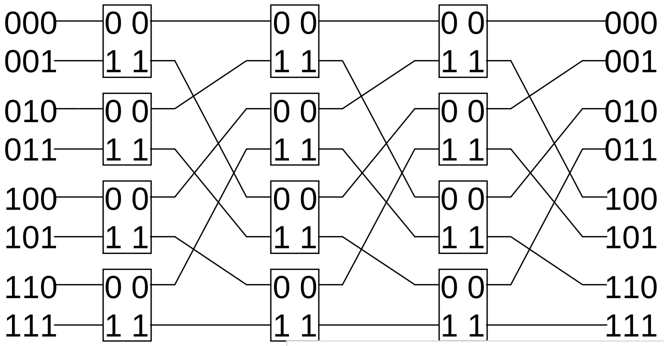
- The interconnection network is an Ω-network



Connection between 2 and 6 … follow bits to destination lsb to msb

Notice Details

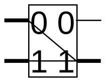
- The Ω-Network requires O(P log P) routers
- The given network uses 2x2 but $2^b x 2^b$ work
- Wiring is consistent at each stage



Long Wires Are Necessary

Routing In Ω-Network

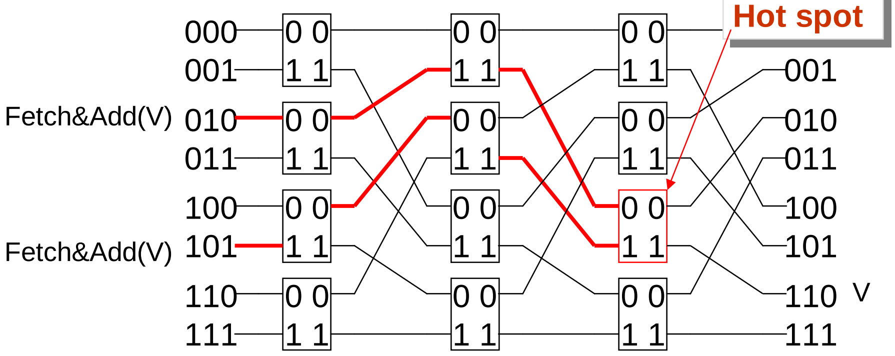
- The network is pipelined
- There is a unique path between any processor and memory port pair
- Conflicts are possible because there exist permutations in which packets collide
- What happens when two packets collide at a router?
  - Packet is delayed, leaving its “file”
  - Pipelining is affected, here comes more



The separate packets must be serialized

Two Processors Make Fetch&Add(V,1)

- Simultaneous requests collide in network

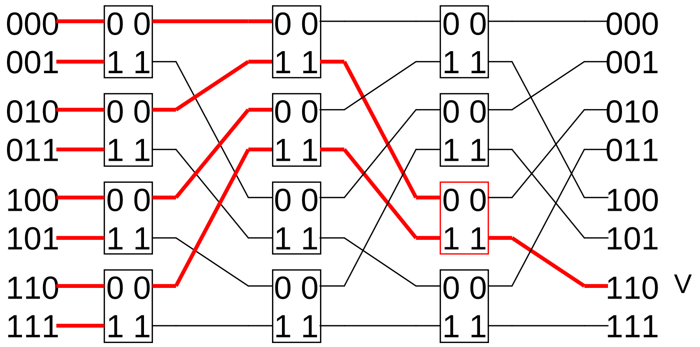


Hot spot

Fetch&Add increases potential for collisions

The Bright Idea: Combine Requests

Idea: Combine requests for same dest. In the limit all nodes could be referencing same loc.

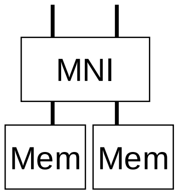


Combining Loads and Stores

- At a switch combine loads and stores to a common location as follows
  - Load/Load -- forward one of the loads towards the memory, and when the value is returned, satisfy both
  - Load/Store -- forward the store, and when the ACK arrives back at the switch, return value to satisfy load
  - Store/Store -- forward one of the stores, and when the ACK arrives back at the switch, return it for both
- Processors are restricted to having only one outstanding request at a time to a given location

Implementing Fetch&Add

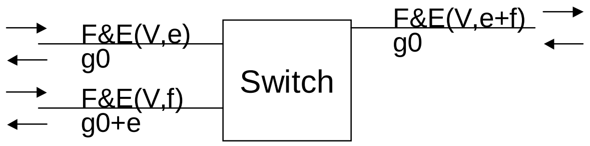
- Include an adder with the Memory Network Interface chips
- For Fetch&Add(V,e)
  - Fetch the value of V, say e0
  - Return e0 to processor requesting
  - Add e0+e
  - Store e0+e back into V
- It is probably necessary to do these concurrently



Combining Fetch&Adds at Switch

Suppose Fetch&Add(V,e) and Fetch&Add(V,f) arrive at a switch together …

- Form the sum f+e
- Send Fetch&Add(V,f+e) on to the memory
- Store e locally
- When g0 is returned by the memory
  - Return g0 as response to Fetch&Add(V,e)
  - Return g0+e as response to Fetch&Add(V,f)



Combine Fetch&Add with other requests

- Combining can apply to all memory traffic to a location V
- Consider the following cases
  - Fetch&Add/Fetch&Add -- as just described
  - Fetch&Add/Load -- Treat Load as Fetch&Add(V,0)
  - Fetch&Add/Store -- If Fetch&Add(V,e) meets Store(V,f) send Store(V,e+f) to memory; when ACK is received, return f as value of F&A
- Conclusion -- it is possible to combine all requests to the same memory location

Will It Work?

- Potential Problems …
  - Network routing is driven entirely by performance, so a complicated switch is usually a problem
  - Routers typically forward non-blocked packets in <= 3 tix
  - Matching to recognize that two requests collide is an “add” operation
  - Combining is an “add” operation *after* the previous add
  - Combining relies on the requests getting to the switch simultaneously, or at worst, before the forwarded packet leaves … this is improbable
  - Most traffic is non-combinable -- head for different places

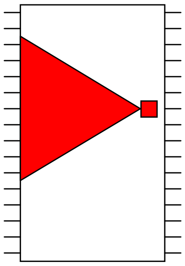
  - A combining router was created by Susan Dickey

A Backup Strategy

- If the network switch is too slow then …
  - Do not combine at every stage … so that some stages can be fast
  - Use two networks, one fast and one that does combining -- it can handle the sharing requests
  - Combine only like requests, e.g. loads/loads
  - Limit combining at a node to two requests
  - As it happened
    - Only like requests have ever been implemented in switch
    - IBM used the two network solution in the RP3

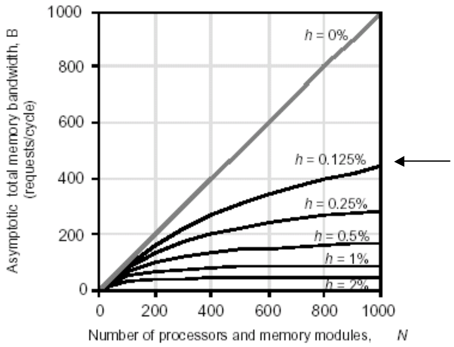
More Globally

- Norton and Pfister discovered in simulation for the RP3 computer that the Ω-Network develops hot-spots
- It was thought that combining would remove the hot-spots … it seemed to for 64-way network
- The problem is that once a node becomes hot, a “back-up” tree forms “behind” the node



More Globally

- But Lee, Kruskal and Kuck showed by simulation and also proved it won’t work
  - LKK discovered and named the “back-up tree”
  - Showed in simulation that the 64-way network is a lucky case
  - Combining doesn’t help because it is the other traffic that is really the problem



0.125% traffic directed at a hot spot

Assessment

- It was a good idea but it didn’t work
- What was good and bad
  - Good
    - Fetch&Add is clever -- a primitive with good properties
    - Shifting from protecting data to allocating work is better
    - Computation at memory is powerful, worth doing
  - Bad
    - Pipelined multistage networks probably just don’t work
    - Complexity in a switch is wrong -- speed is essential
    - Failed to exploit locality -- caching essentially impossible

Can other designs include the good ideas?

Summary

- We introduced the concept of the Fetch&Add
- Showed how F&A can be used to implement many basic parallel operations
- Considered the implementation
  - Ω-network
  - Switches and interface structure
- Introduced combining at switches w/ examples
- Combining may fix hotspots, but its slow performance can slow down other operations

Homework

Write two procedures -- **`Put_Task()`** and **`Get_Task()`** -- in any language, e.g. C, and using Fetch&Add for synchronization as follows

- Define `TD[1..n]`, the ToDo array; `n=c*Processors` for some `c`
- Define `FF`, for first free, pointing in `TD` to first free cell
- Define `NA`, for next available, pointing to next task in `TD`
- `Put_Task(a)` takes a task as input, and places it in `TD`; `Get_Task()` returns the next task from `TD` if there is one
- The management of `TD` is completely decentralized

Submit solution by email to Adam prior to class