5DV118
Computer Organization and Architecture
Umeå University
Department of Computing Science

Stephen J. Hegner

Topic 7: Support for Parallelism

These slides are mostly taken verbatim, or with minor changes, from those prepared by

Mary Jane Irwin (www.cse.psu.edu/~mji)

of The Pennsylvania State University

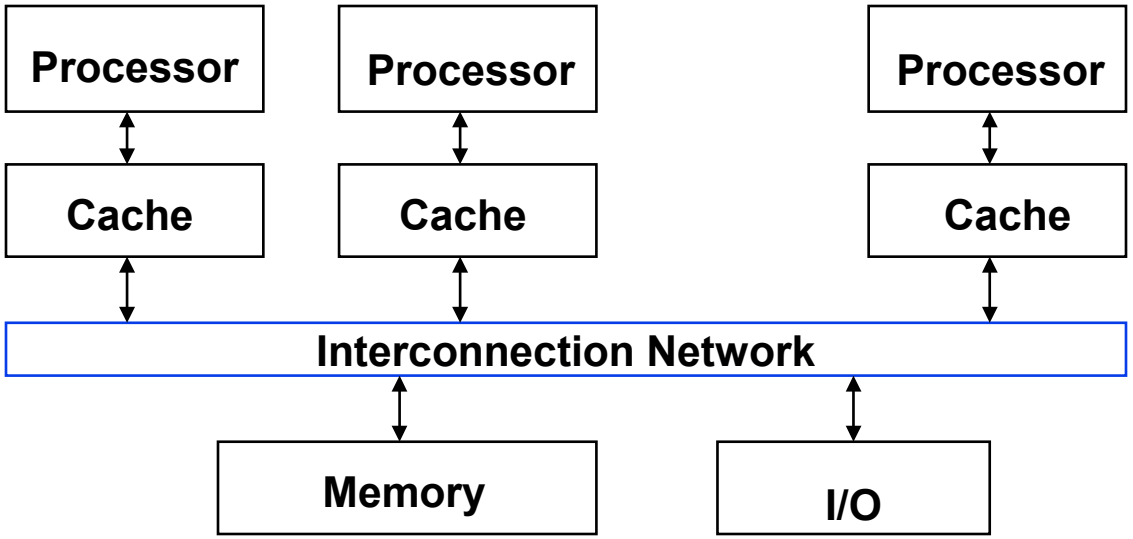
[Adapted from *Computer Organization and Design, 4th Edition*, Patterson & Hennessy, © 2008, MK]

Key to the Slides

- The source of each slide is coded in the footer on the right side:
  - Irwin CSE331 = slide by Mary Jane Irwin from the course CSE331 (Computer Organization and Design) at The Pennsylvania State University.
  - Irwin CSE431 = slide by Mary Jane Irwin from the course CSE431 (Computer Architecture) at The Pennsylvania State University.
  - Hegner UU = slide by Stephen J. Hegner at Umeå University.

The Big Picture: Where are We Now?

- Multiprocessor – a computer system with at least two processors



- Can deliver high throughput for independent jobs via job-level parallelism or process-level parallelism
- And improve the run time of a *single* program that has been specially crafted to run on a multiprocessor - a parallel processing program

Multicores Now Common

- The power challenge has forced a change in the design of microprocessors
  - Since 2002 the rate of improvement in the response time of programs has slowed from a factor of 1.5 per year to less than a factor of 1.2 per year
- Today's microprocessors typically contain more than one core – Chip Multicore microProcessors (CMPs) – in a single IC
  - The number of cores is expected to double every two years

Product	AMD Barcelona	Intel Nehalem	IBM Power 6	Sun Niagara 2
Cores per chip	4	4	2	8
Clock rate	2.5 GHz	~2.5 GHz?	4.7 GHz	1.4 GHz
Power	120 W	~100 W?	~100 W?	94 W

Other Multiprocessor Basics

- Some of the problems that need higher performance can be handled simply by using a cluster – a set of independent servers (or PCs) connected over a local area network (LAN) functioning as a single large multiprocessor
  - Search engines, Web servers, email servers, databases, …

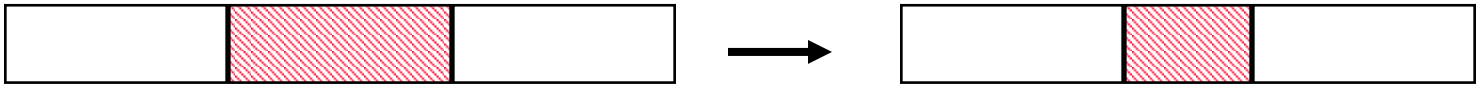
- A key challenge is to craft parallel (concurrent) programs that have high performance on multiprocessors as the number of processors increase – i.e., that scale
  - Scheduling, load balancing, time for synchronization, overhead for communication

Encountering Amdahl's Law

- Speedup due to enhancement E is

$$\text{Speedup w/ E} = \frac{\text{Exec time w/o E}}{\text{Exec time w/ E}}$$

- Suppose that enhancement E accelerates a fraction F (F <1) of the task by a factor S (S>1) and the remainder of the task is unaffected



$$\text{ExTime w/ E} = \text{ExTime w/o E} \times$$

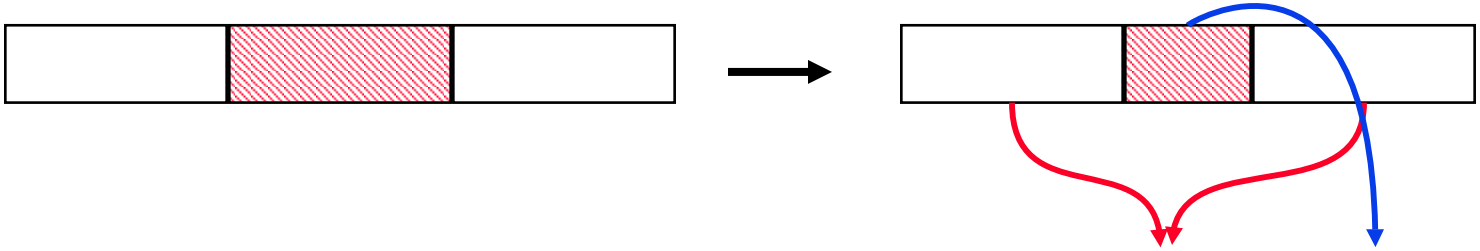
$$\text{Speedup w/ E} =$$

Encountering Amdahl's Law

- Speedup due to enhancement E is

$$\text{Speedup w/ E} = \frac{\text{Exec time w/o E}}{\text{Exec time w/ E}}$$

- Suppose that enhancement E accelerates a fraction F (F <1) of the task by a factor S (S>1) and the remainder of the task is unaffected



$$\text{ExTime w/ E} = \text{ExTime w/o E} \times ((1\text{-}F) + F/S)$$

$$\text{Speedup w/ E} = 1 / ((1\text{-}F) + F/S)$$

Example 1: Amdahl's Law

Speedup w/ E =

- Consider an enhancement which runs 20 times faster but which is only usable 25% of the time.

  Speedup w/ E =

- What if its usable only 15% of the time?

  Speedup w/ E =

- Amdahl's Law tells us that to achieve linear speedup with 100 processors, none of the original computation can be scalar!

- To get a speedup of 90 from 100 processors, the percentage of the original program that could be scalar would have to be 0.1% or less

  Speedup w/ E =

Example 1: Amdahl's Law

Speedup w/ E = 1 / ((1-F) + F/S)

- Consider an enhancement which runs 20 times faster but which is only usable 25% of the time.

  Speedup w/ E = 1/(.75 + .25/20) = 1.31

- What if its usable only 15% of the time?

  Speedup w/ E = 1/(.85 + .15/20) = 1.17

- Amdahl's Law tells us that to achieve linear speedup with 100 processors, none of the original computation can be scalar!

- To get a speedup of 90 from 100 processors, the percentage of the original program that could be scalar would have to be 0.1% or less

  Speedup w/ E = 1/(.001 + .999/100) = 90.99

Example 2: Amdahl's Law

Speedup w/ E = 1 / ((1-F) + F/S)

- Consider summing 10 scalar variables and two 10 by 10 matrices (matrix sum) on 10 processors

  Speedup w/ E =

- What if there are 100 processors ?

  Speedup w/ E =

- What if the matrices are100 by 100 (or 10,010 adds in total) on 10 processors?

  Speedup w/ E =

- What if there are 100 processors ?

  Speedup w/ E =

Example 2: Amdahl's Law

Speedup w/ E = 1 / ((1-F) + F/S)

- Consider summing 10 scalar variables and two 10 by 10 matrices (matrix sum) on 10 processors

  Speedup w/ E = 1/(.091 + .909/10) = 1/0.1819 = 5.5

- What if there are 100 processors ?

  Speedup w/ E = 1/(.091 + .909/100) = 1/0.10009 = 10.0

- What if the matrices are100 by 100 (or 10,010 adds in total) on 10 processors?

  Speedup w/ E = 1/(.001 + .999/10) = 1/0.1009 = 9.9

- What if there are 100 processors ?

  Speedup w/ E = 1/(.001 + .999/100) = 1/0.01099 = 91

Scaling

- To get good speedup on a multiprocessor while keeping the problem size fixed is harder than getting good speedup by increasing the size of the problem.
  - Strong scaling – when speedup can be achieved on a multiprocessor without increasing the size of the problem
  - Weak scaling – when speedup is achieved on a multiprocessor by increasing the size of the problem proportionally to the increase in the number of processors

- Load balancing is another important factor. Just a single processor with twice the load of the others cuts the speedup almost in half

Multiprocessor/Clusters Key Questions

- Q1 – How do they share data?
- Q2 – How do they coordinate?
- Q3 – How scalable is the architecture? How many processors can be supported?

Shared Memory Multiprocessor (SMP)

- Q1 – Single address space shared by all processors
- Q2 – Processors coordinate/communicate through shared variables in memory (via loads and stores)
  - Use of shared data must be coordinated via synchronization primitives (locks) that allow access to data to only one processor at a time
- They come in two styles
  - Uniform memory access (UMA) multiprocessors
  - Nonuniform memory access (NUMA) multiprocessors
- Programming NUMAs is harder
- But NUMAs can scale to larger sizes and have lower latency to local memory

Summing 100,000 Numbers on 100 Proc. SMP

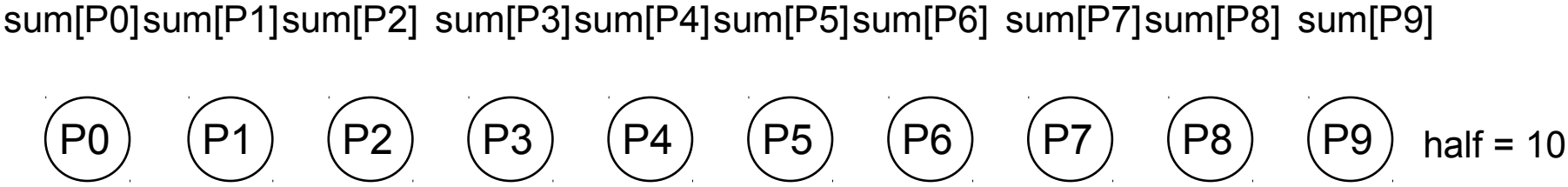
- Processors start by running a loop that sums their subset of vector `A` numbers (vectors `A` and `sum` are shared variables, `Pn` is the processor's number, `i` is a private variable)

```
sum[Pn] = 0;
for (i = 1000*Pn; i< 1000*(Pn+1); i = i + 1)
  sum[Pn] = sum[Pn] + A[i];
```

- The processors then coordinate in adding together the partial sums (`half` is a private variable initialized to `100` (the number of processors)) – reduction

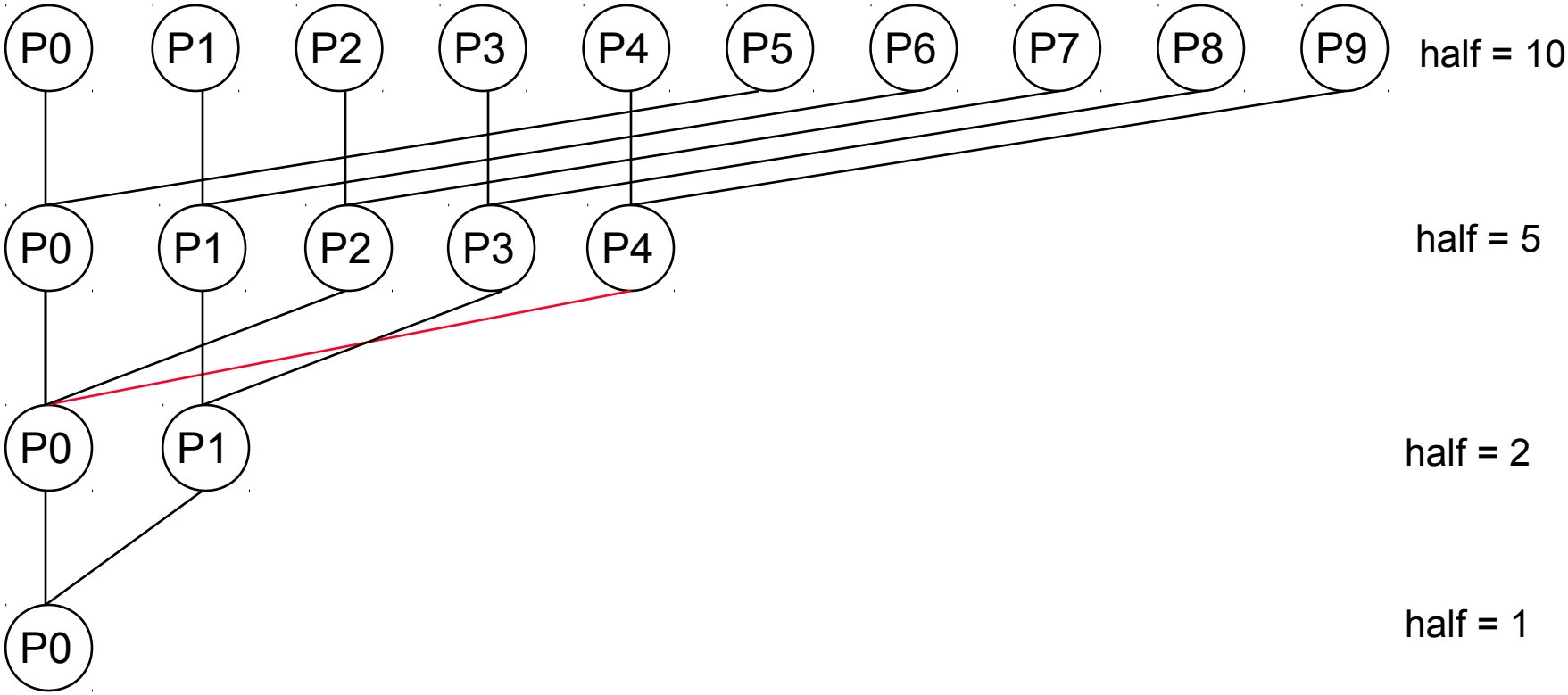
```
repeat
  synch();                    /*synchronize first
  if (half%2 != 0 && Pn == 0)
      sum[0] = sum[0] + sum[half-1];
  half = half/2
  if (Pn<half) sum[Pn] = sum[Pn] + sum[Pn+half]
until (half == 1);            /*final sum in sum[0]
```

An Example with 10 Processors



An Example with 10 Processors

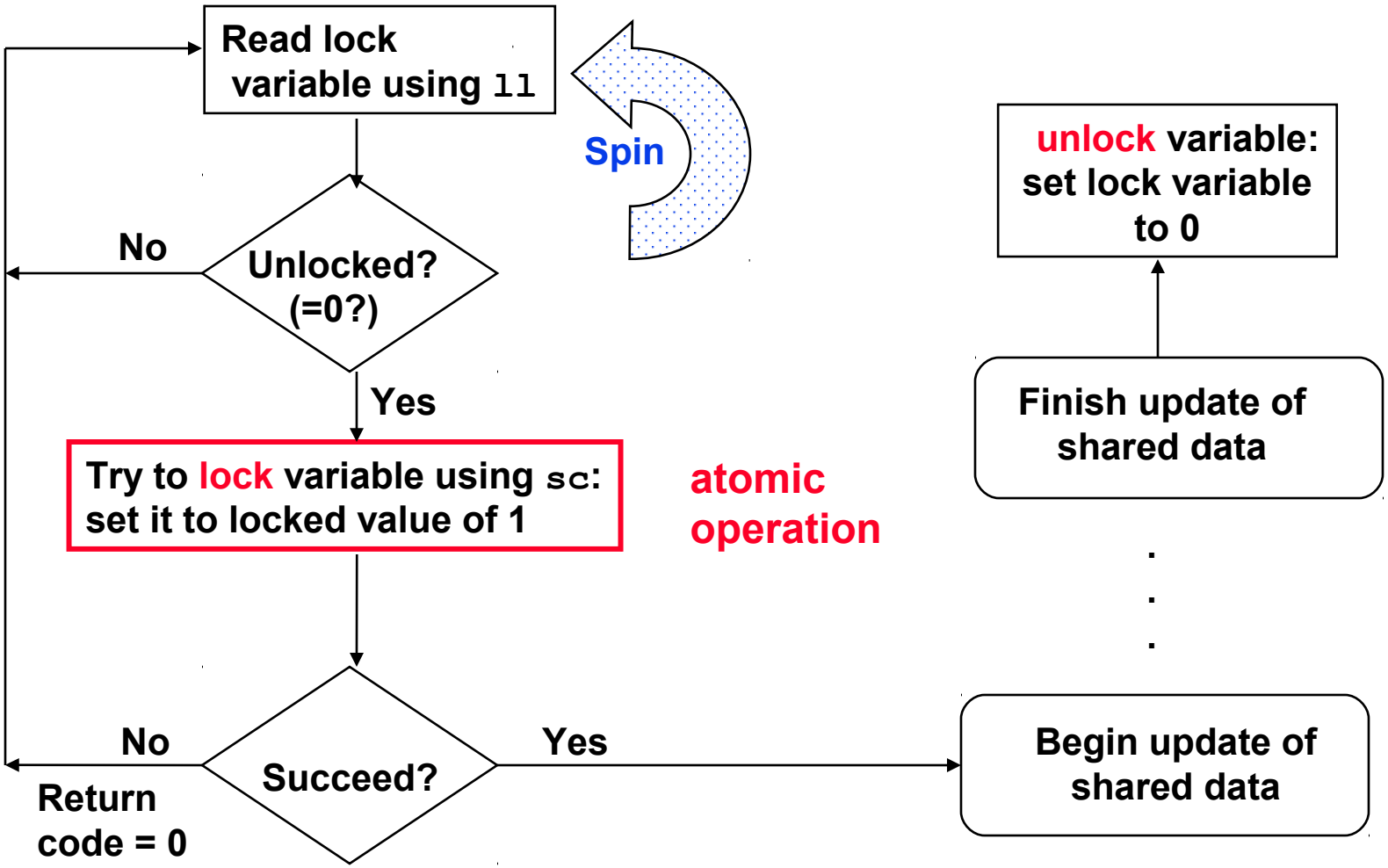
sum[P0]sum[P1]sum[P2] sum[P3]sum[P4]sum[P5]sum[P6] sum[P7]sum[P8] sum[P9]



Process Synchronization

- Need to be able to coordinate processes working on a common task
- Lock variables (semaphores) are used to coordinate or synchronize processes
- Need an architecture-supported arbitration mechanism to decide which processor gets access to the lock variable
  - Single bus provides arbitration mechanism, since the bus is the only path to memory – the processor that gets the bus wins
- Need an architecture-supported operation that locks the variable
  - Locking can be done via an atomic swap operation (on the MIPS we have `ll` and `sc` one example of where a processor can both read a location *and* set it to the locked state – test-and-set – in the same bus operation)

Spin Lock Synchronization



The *single* winning processor will succeed in writing a 1 to the lock variable - all others processors will get a return code of 0

Review: Summing Numbers on a SMP

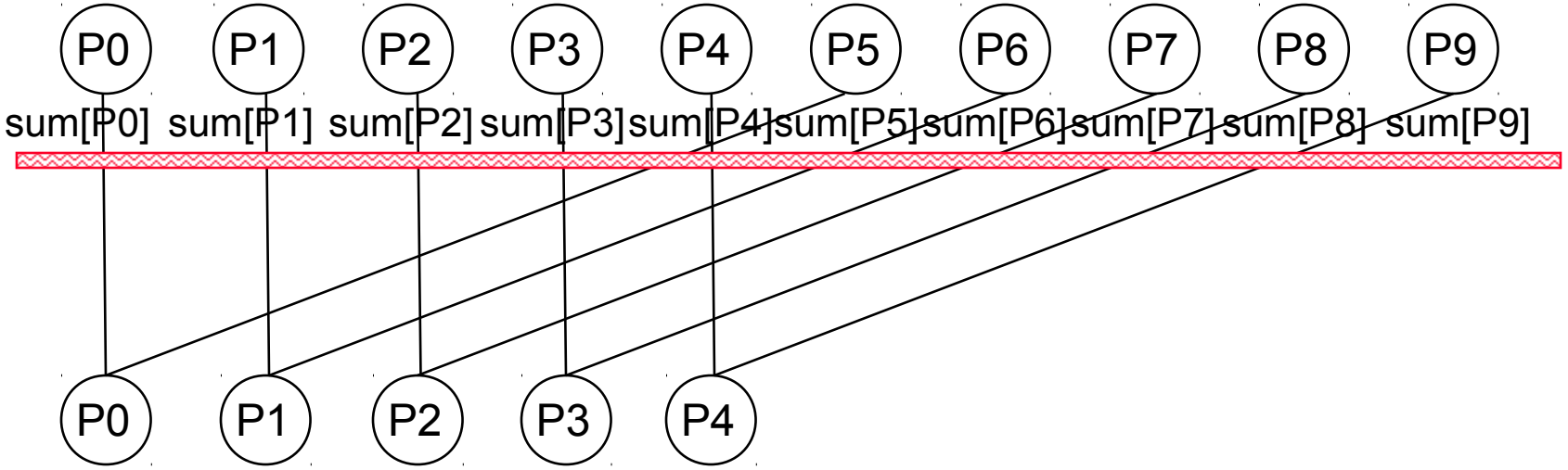
- Pn is the processor's number, vectors A and sum are shared variables, i is a private variable, half is a private variable initialized to the number of processors

```
 sum[Pn] = 0;
 for (i = 1000*Pn; i< 1000*(Pn+1); i = i + 1)
   sum[Pn] = sum[Pn] + A[i];
                              /* each processor sums its
                              /* subset of vector A

repeat                        /* adding together the
                              /* partial sums
   synch();                   /*synchronize first
   if (half%2 != 0 && Pn == 0)
        sum[0] = sum[0] + sum[half-1];
   half = half/2
   if (Pn<half) sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1);            /*final sum in sum[0]
```

An Example with 10 Processors

- `synch()`: Processors must synchronize before the "consumer" processor tries to read the results from the memory location written by the "producer" processor
  - Barrier synchronization – a synchronization scheme where processors wait at the barrier, not proceeding until every processor has reached it



Barrier Implemented with Spin-Locks

- `n` is a shared variable initialized to the number of processors, `count` is a shared variable initialized to 0, `arrive` and `depart` are shared spin-lock variables where `arrive` is initially unlocked and `depart` is initially locked

```
procedure synch()
   lock(arrive);
     count := count + 1;    /* count the processors as
     if count < n           /* they arrive at barrier
         then unlock(arrive)
         else unlock(depart);


   lock(depart);
     count := count - 1;    /* count the processors as
     if count > 0           /* they leave barrier
         then unlock(depart)
         else unlock(arrive);
```

Spin-Locks on Bus Connected ccUMAs

- With a bus based cache coherency protocol (write invalidate), spin-locks allow processors to wait on a local copy of the lock in their caches
  - Reduces bus traffic – once the processor with the lock releases the lock (writes a 0) all other caches see that write and invalidate their old copy of the lock variable. Unlocking restarts the race to get the lock. The winner gets the bus and writes the lock back to 1. The other caches then invalidate their copy of the lock and on the next lock read fetch the new lock value (1) from memory.

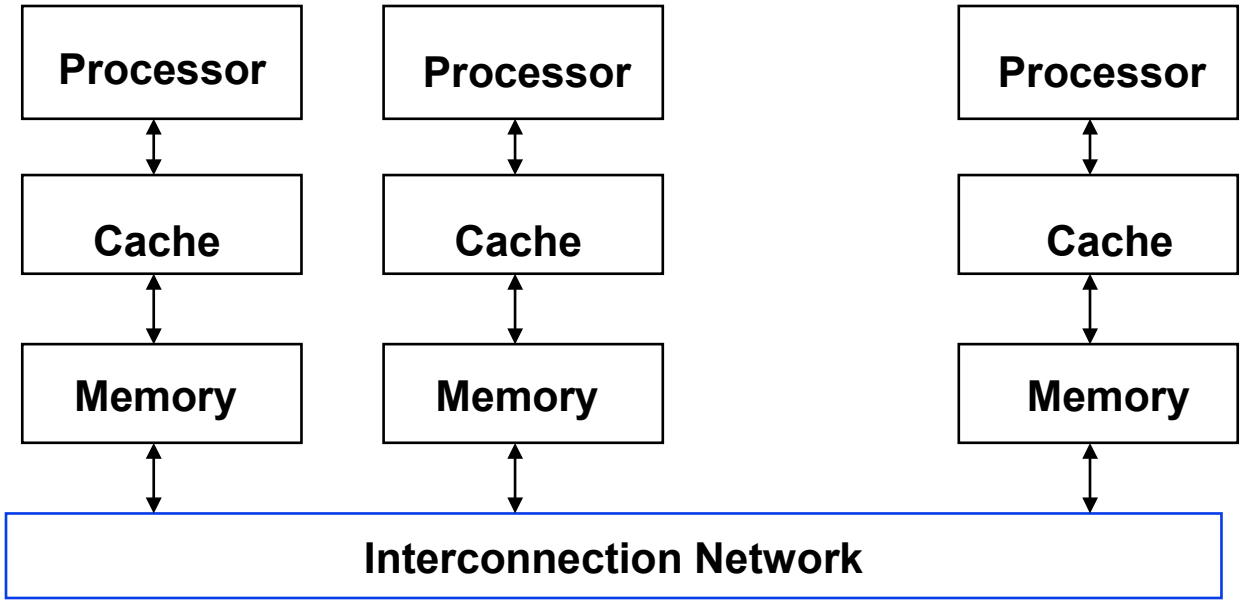
- This scheme has problems scaling up to many processors because of the communication traffic when the lock is released and contested

Aside: Cache Coherence Bus Traffic

	Proc P0	Proc P1	Proc P2	Bus activity	Memory
1	Has lock	Spins	Spins	None	
2	Releases lock (0)	Spins	Spins	Bus services P0's invalidate	
3		Cache miss	Cache miss	Bus services P2's cache miss	
4		Waits	Reads lock (0)	Response to P2's cache miss	Update lock in memory from P0
5		Reads lock (0)	Swaps lock (`ll,sc` of 1)	Bus services P1's cache miss	
6		Swaps lock (`ll,sc` of 1)	Swap succeeds	Response to P1's cache miss	Sends lock variable to P1
7		Swap fails	Has lock	Bus services P2's invalidate	
8		Spins	Has lock	Bus services P1's cache miss	

Message Passing Multiprocessors (MPP)

- Each processor has its own private address space
- Q1 – Processors share data by *explicitly* sending and receiving information (message passing)
- Q2 – Coordination is built into message passing primitives (message send and message receive)



Summing 100,000 Numbers on 100 Proc. MPP

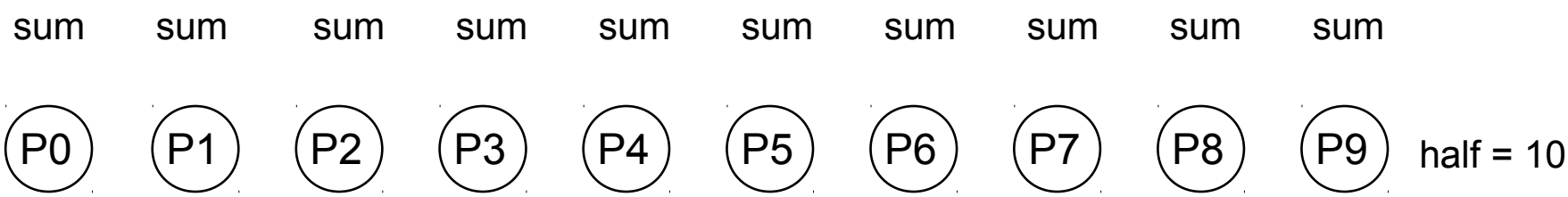
- Start by distributing 1000 elements of vector `A` to each of the local memories and summing each subset in parallel

```
sum = 0;
for (i = 0; i<1000; i = i + 1)
  sum = sum + Al[i];     /* sum local array subset
```

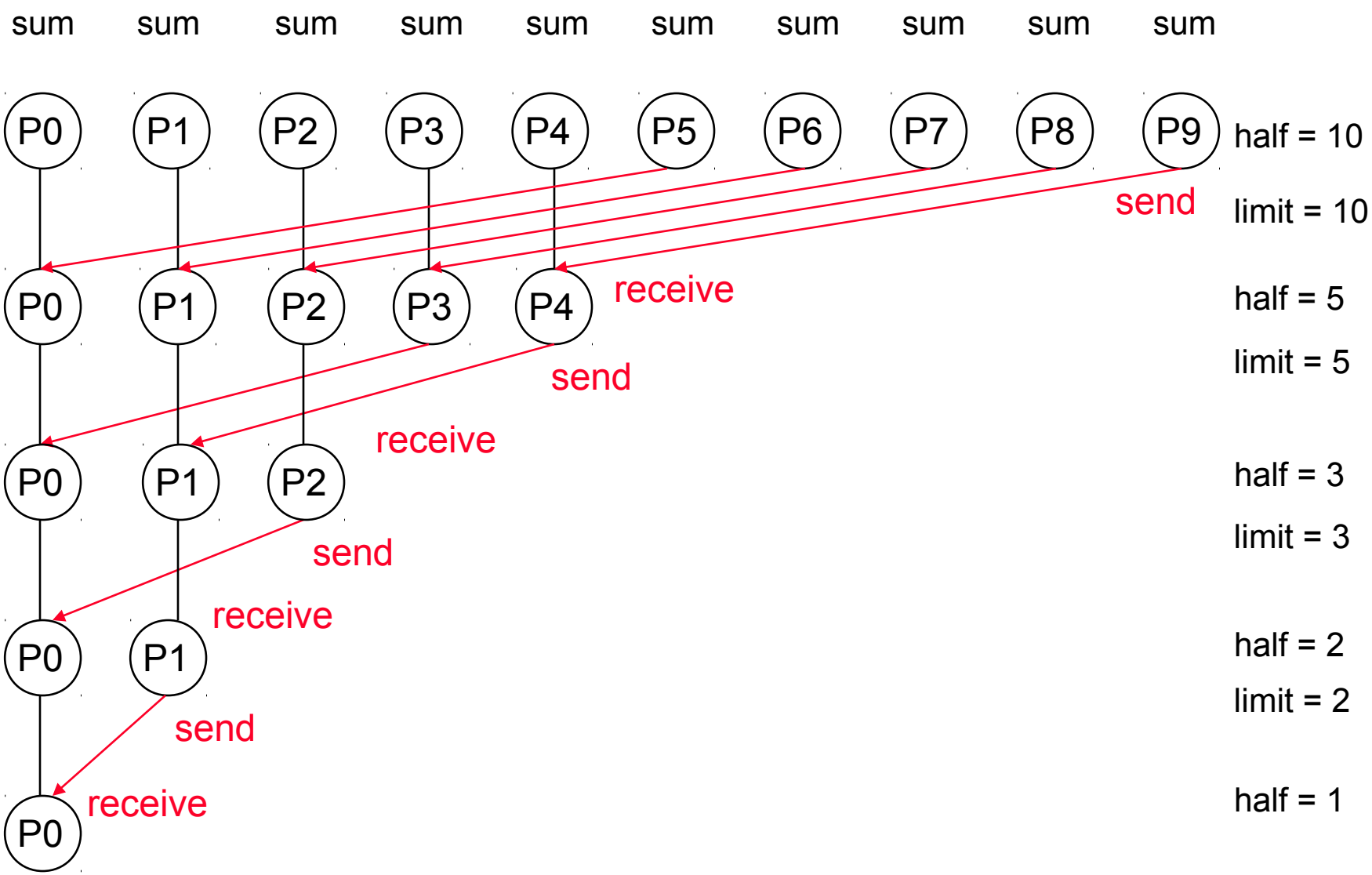
- The processors then coordinate in adding together the sub sums (`Pn` is the number of processors, `send(x,y)` sends value `y` to processor `x`, and `receive()` receives a value)

```
half = 100;
limit = 100;
repeat
  half = (half+1)/2;     /*dividing line
  if (Pn>= half && Pn<limit) send(Pn-half,sum);
  if (Pn<(limit/2)) sum = sum + receive();
  limit = half;
until (half == 1);       /*final sum in P0's sum
```

An Example with 10 Processors



An Example with 10 Processors



Pros and Cons of Message Passing

- Message sending and receiving is *much* slower than addition, for example
- But message passing multiprocessors and much easier for hardware designers to design
  - Don't have to worry about cache coherency for example
- The advantage for programmers is that communication is explicit, so there are fewer "performance surprises" than with the implicit communication in cache-coherent SMPs.
  - Message passing standard MPI-2 (www.mpi-forum.org)
- However, its harder to port a sequential program to a message passing multiprocessor since every communication must be identified in advance.
  - With cache-coherent shared memory the hardware figures out what data needs to be communicated

Networks of Workstations (NOWs) Clusters

- Clusters of off-the-shelf, whole computers with multiple private address spaces connected using the I/O bus of the computers
  - lower bandwidth than multiprocessor that use the processor-memory (front side) bus
  - lower speed network links
  - more conflicts with I/O traffic
- Clusters of N processors have N copies of the OS limiting the memory available for applications
- Improved system availability and expandability
  - easier to replace a machine without bringing down the whole system
  - allows rapid, incremental expandability
- Economy-of-scale advantages with respect to costs

Commercial (NOW) Clusters

	Proc	Proc Speed	# Proc	Network
Dell PowerEdge	P4 Xeon	3.06GHz	2,500	Myrinet
eServer IBM SP	Power4	1.7GHz	2,944	
VPI BigMac	Apple G5	2.3GHz	2,200	Mellanox Infiniband
HP ASCI Q	Alpha 21264	1.25GHz	8,192	Quadrics
LLNL Thunder	Intel Itanium2	1.4GHz	1,024*4	Quadrics
Barcelona	PowerPC 970	2.2GHz	4,536	Myrinet

Multithreading on A Chip

- Find a way to "hide" true data dependency stalls, cache miss stalls, and branch stalls by finding instructions (from other process threads) that are independent of those stalling instructions
- Hardware multithreading – increase the utilization of resources on a chip by allowing multiple processes (threads) to share the functional units of a single processor
  - Processor must duplicate the state hardware for each thread – a separate register file, PC, instruction buffer, and store buffer for each thread
  - The caches, TLBs, BHT, BTB, RUU can be shared (although the miss rates may increase if they are not sized accordingly)
  - The memory can be shared through virtual memory mechanisms
  - Hardware must support *efficient* thread context switching

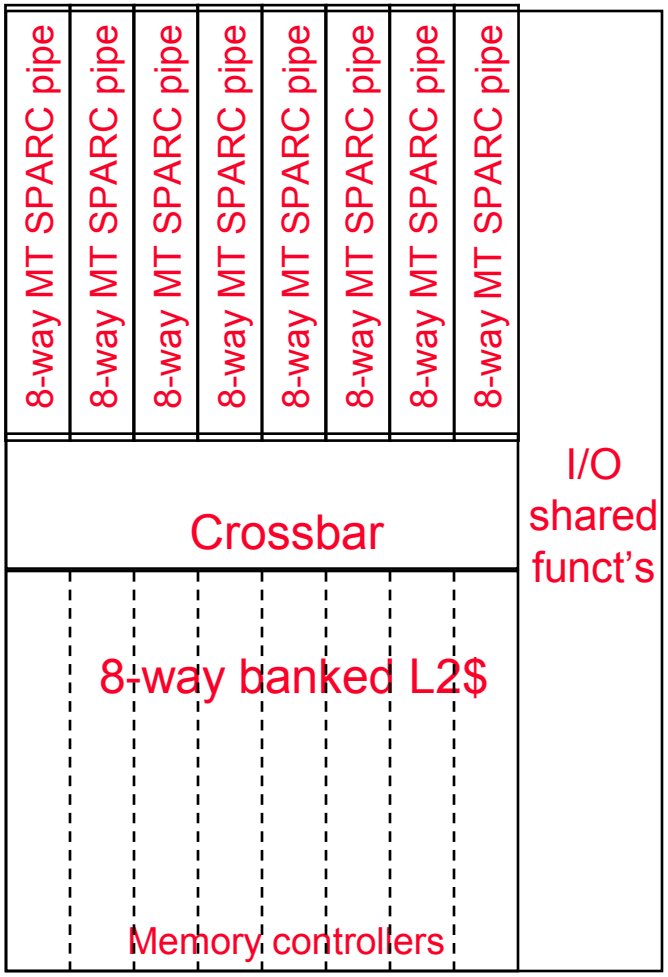
Types of Multithreading

- Fine-grain – switch threads on every instruction issue
  - Round-robin thread interleaving (skipping stalled threads)
  - Processor must be able to switch threads on every clock cycle
  - Advantage – can hide throughput losses that come from both short and long stalls
  - Disadvantage – slows down the execution of an individual thread since a thread that is ready to execute without stalls is delayed by instructions from other threads
- Coarse-grain – switches threads only on costly stalls (e.g., L2 cache misses)
  - Advantages – thread switching doesn't have to be essentially free and much less likely to slow down the execution of an individual thread
  - Disadvantage – limited, due to pipeline start-up costs, in its ability to overcome throughput loss
    - Pipeline must be flushed and refilled on thread switches

Multithreaded Example: Sun's Niagara (UltraSparc T2)

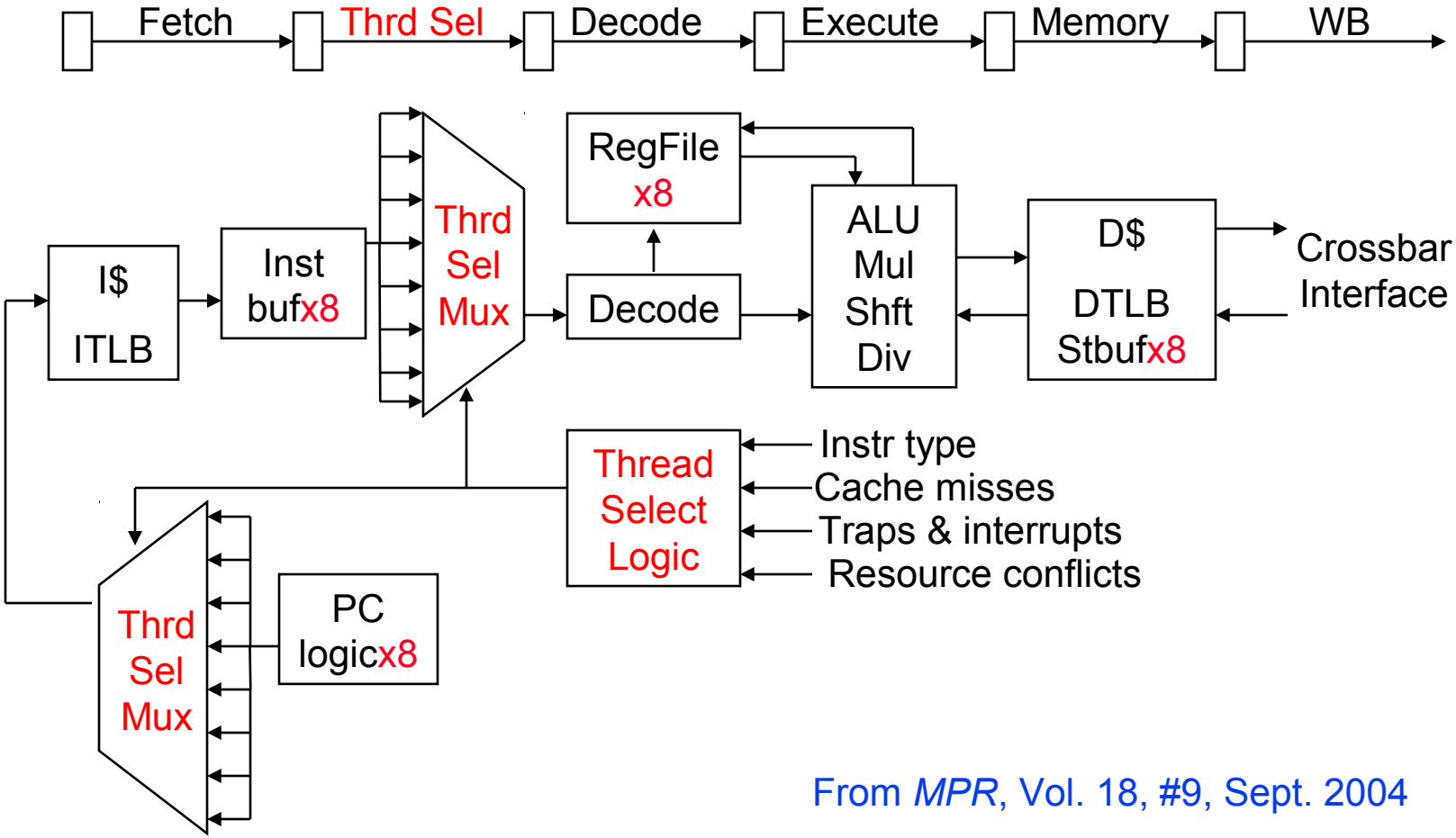
- Eight fine grain multithreaded single-issue, in-order cores (no speculation, no dynamic branch prediction)

	Niagara 2
Data width	64-b
Clock rate	1.4 GHz
Cache (I/D/L2)	16K/8K/4M
Issue rate	1 issue
Pipe stages	6 stages
BHT entries	None
TLB entries	64I/64D
Memory BW	60+ GB/s
Transistors	??? million
Power (max)	<95 W



Niagara Integer Pipeline

- Cores are simple (single-issue, 6 stage, no branch prediction), small, and power-efficient

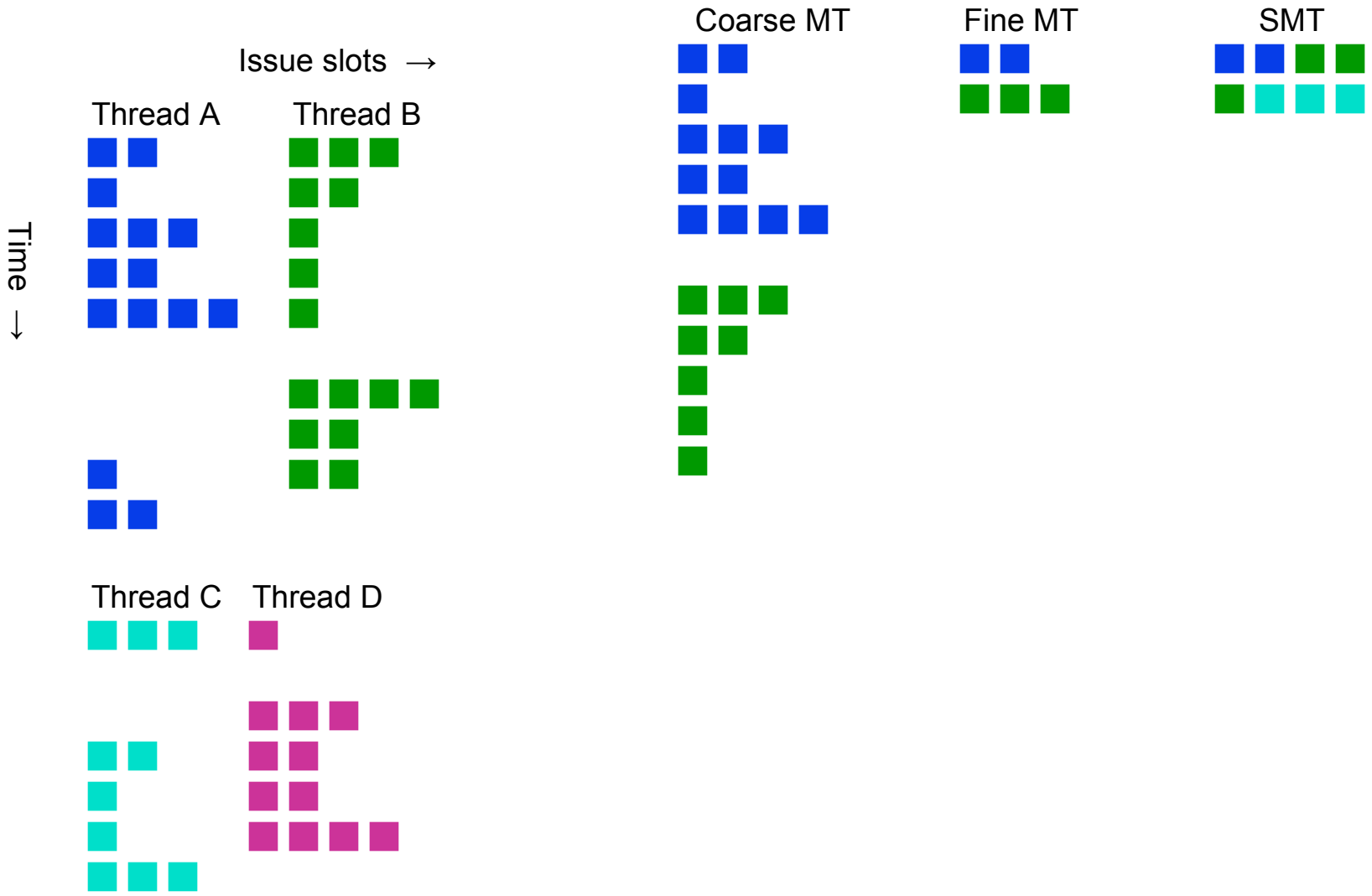


From *MPR*, Vol. 18, #9, Sept. 2004

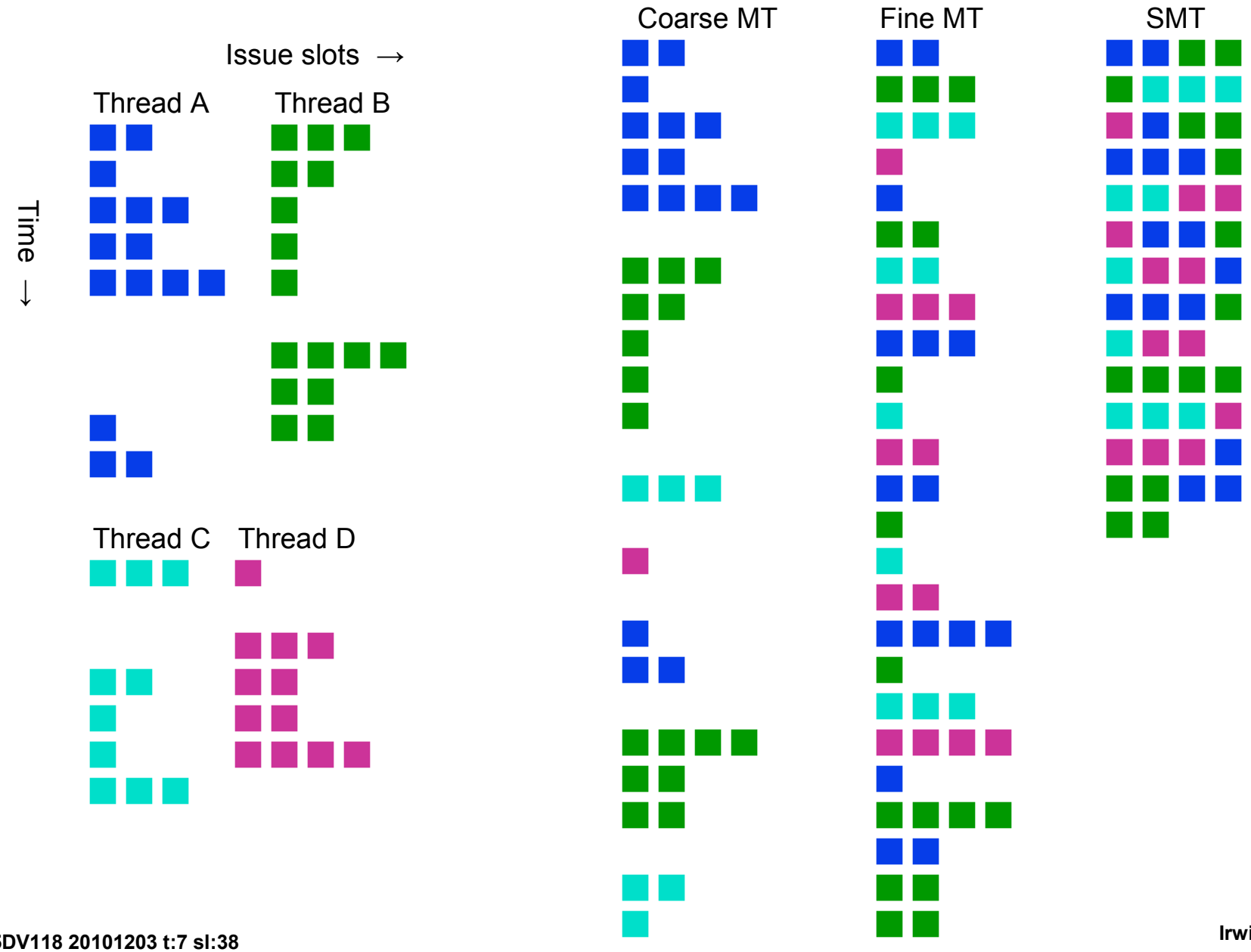
Simultaneous Multithreading (SMT)

- A variation on multithreading that uses the resources of a multiple-issue, dynamically scheduled processor (superscalar) to exploit both program ILP and thread-level parallelism (TLP)
  - Most SS processors have more machine level parallelism than most programs can effectively use (i.e., than have ILP)
  - With register renaming and dynamic scheduling, multiple instructions from independent threads can be issued without regard to dependencies among them
    - Need separate rename tables (RUUs) for each thread or need to be able to indicate which thread the entry belongs to
    - Need the capability to commit from multiple threads in one cycle
- Intel's Pentium 4 SMT is called hyperthreading
  - Supports just two threads (doubles the architecture state)

Threading on a 4-way SS Processor Example



Threading on a 4-way SS Processor Example



Review: Multiprocessor Basics

- Q1 – How do they share data?
- Q2 – How do they coordinate?
- Q3 – How scalable is the architecture? How many processors?

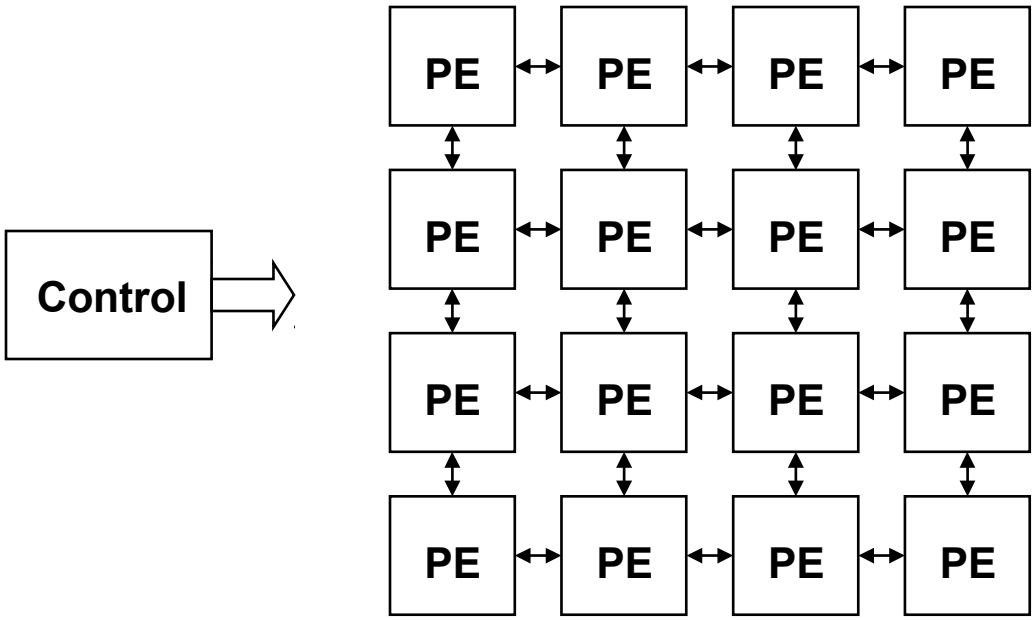
			# of Proc
Communication model	Message passing		8 to 2048
	Shared address	NUMA	8 to 256
		UMA	2 to 64
Physical connection	Network		8 to 256
	Bus		2 to 36

Flynn's Classification Scheme

- SISD – single instruction, single data stream
  - aka uniprocessor - what we have been talking about all semester
- SIMD – single instruction, multiple data streams
  - single control unit broadcasting operations to multiple datapaths
- MISD – multiple instruction, single data
  - no such machine (although some people put vector machines in this category)
- MIMD – multiple instructions, multiple data streams
  - aka multiprocessors (SMPs, MPPs, clusters, NOWs)

- Now obsolete terminology except for . . .

SIMD Processors



- Single control unit (one copy of the code)
- Multiple datapaths (Processing Elements – PEs) running in parallel
  - Q1 – PEs are interconnected (usually via a mesh or torus) and exchange/share data as directed by the control unit
  - Q2 – Each PE performs the same operation on its own local data

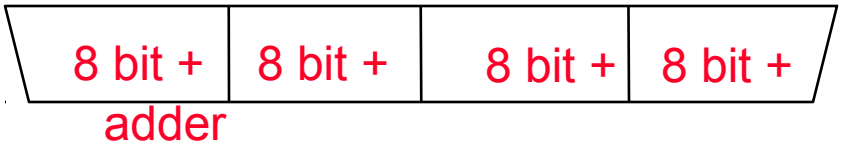
Example SIMD Machines

	Maker	Year	# PEs	# b/ PE	Max memory (MB)	PE clock (MHz)	System BW (MB/s)
Illiac IV	UIUC	1972	64	64	1	13	2,560
DAP	ICL	1980	4,096	1	2	5	2,560
MPP	Goodyear	1982	16,384	1	2	10	20,480
CM-2	Thinking Machines	1987	65,536	1	512	7	16,384
MP-1216	MasPar	1989	16,384	4	1024	25	23,000

- Did SIMDs die out in the early 1990s ??

Multimedia SIMD Extensions

- The most widely used variation of SIMD is found in almost every microprocessor today – as the basis of MMX and SSE instructions added to improve the performance of multimedia programs
  - A single, wide ALU is partitioned into many smaller ALUs that operate in parallel



  - Loads and stores are simply as wide as the widest ALU, so the same data transfer can transfer one 32 bit value, two 16 bit values or four 8 bit values

- There are now hundreds of SSE instructions in the x86 to support multimedia operations

Vector Processors

- A vector processor (e.g., Cray) pipelines the ALUs to get good performance at lower cost. A key feature is a set of vector registers to hold the operands and results.
  - Collect the data elements from memory, put them in order into a large set of registers, operate on them sequentially in registers, and then write the results back to memory
  - They formed the basis of supercomputers in the 1980's and 90's

- Consider extending the MIPS instruction set (VMIPS) to include vector instructions, e.g.,
  - `addv.d` to add two double precision vector register values
  - `addvs.d` and `mulvs.d` to add (or multiply) a scalar register to (by) each element in a vector register
  - `lv` and `sv` do vector load and vector store and load or store an entire vector of double precision data

MIPS vs VMIPS DAXPY Codes: Y = a × X + Y

```
        l.d     $f0,a($sp)      ;load scalar a
        addiu   r4,$s0,#512     ;upper bound to load to
loop:   l.d     $f2,0($s0)      ;load X(i)
        mul.d   $f2,$f2,$f0     ;a × X(i)
        l.d     $f4,0($s1)      ;load Y(i)
        add.d   $f4,$f4,$f2     ;a × X(i) + Y(i)
        s.d     $f4,0($s1)      ;store into Y(i)
        addiu   $s0,$s0,#8      ;increment X index
        addiu   $s1,$s1,#8      ;increment Y index
        subu    $t0,r4,$s0      ;compute bound
        bne     $t0,$zero,loop  ;check if done
```

MIPS vs VMIPS DAXPY Codes: Y = a × X + Y

```
        l.d     $f0,a($sp)      ;load scalar a
        addiu   r4,$s0,#512     ;upper bound to load to
loop:   l.d     $f2,0($s0)      ;load X(i)
        mul.d   $f2,$f2,$f0     ;a × X(i)
        l.d     $f4,0($s1)      ;load Y(i)
        add.d   $f4,$f4,$f2     ;a × X(i) + Y(i)
        s.d     $f4,0($s1)      ;store into Y(i)
        addiu   $s0,$s0,#8      ;increment X index
        addiu   $s1,$s1,#8      ;increment Y index
        subu    $t0,r4,$s0      ;compute bound
        bne     $t0,$zero,loop  ;check if done
```

```
        l.d     $f0,a($sp)      ;load scalar a
        lv      $v1,0($s0)      ;load vector X
        mulvs.d $v2,$v1,$f0     ;vector-scalar multiply
        lv      $v3,0($s1)      ;load vector Y
        addv.d  $v4,$v2,$v3     ;add Y to a × X
        sv      $v4,0($s1)      ;store vector result
```

Vector verus Scalar

- Instruction fetch and decode bandwidth is dramatically reduced (also saves power)
  - Only six instructions in VMIPS versus almost 600 in MIPS for 64 element DAXPY
- Hardware doesn't have to check for data hazards within a vector instruction. A vector instruction will only stall for the first element, then subsequent elements will flow smoothly down the pipeline. And control hazards are nonexistent.
  - MIPS stall frequency is about 64 times higher than VMIPS for DAXPY
- Easier to write code for data-level parallel app's
- Have a known access pattern to memory, so heavily interleaved memory banks work well. The cost of latency to memory is seen only once for the entire vector

Example Vector Machines

	Maker	Year	Peak perf.	# vector Processors	PE clock (MHz)
STAR-100	CDC	1970	??	113	2
ASC	TI	1970	20 MFLOPS	1, 2, or 4	16
Cray 1	Cray	1976	80 to 240 MFLOPS		80
Cray Y-MP	Cray	1988	333 MFLOPS	2, 4, or 8	167
Earth Simulator	NEC	2002	35.86 TFLOPS	8	

- Did Vector machines die out in the late 1990s ??

Graphics Processing Units (GPUs)

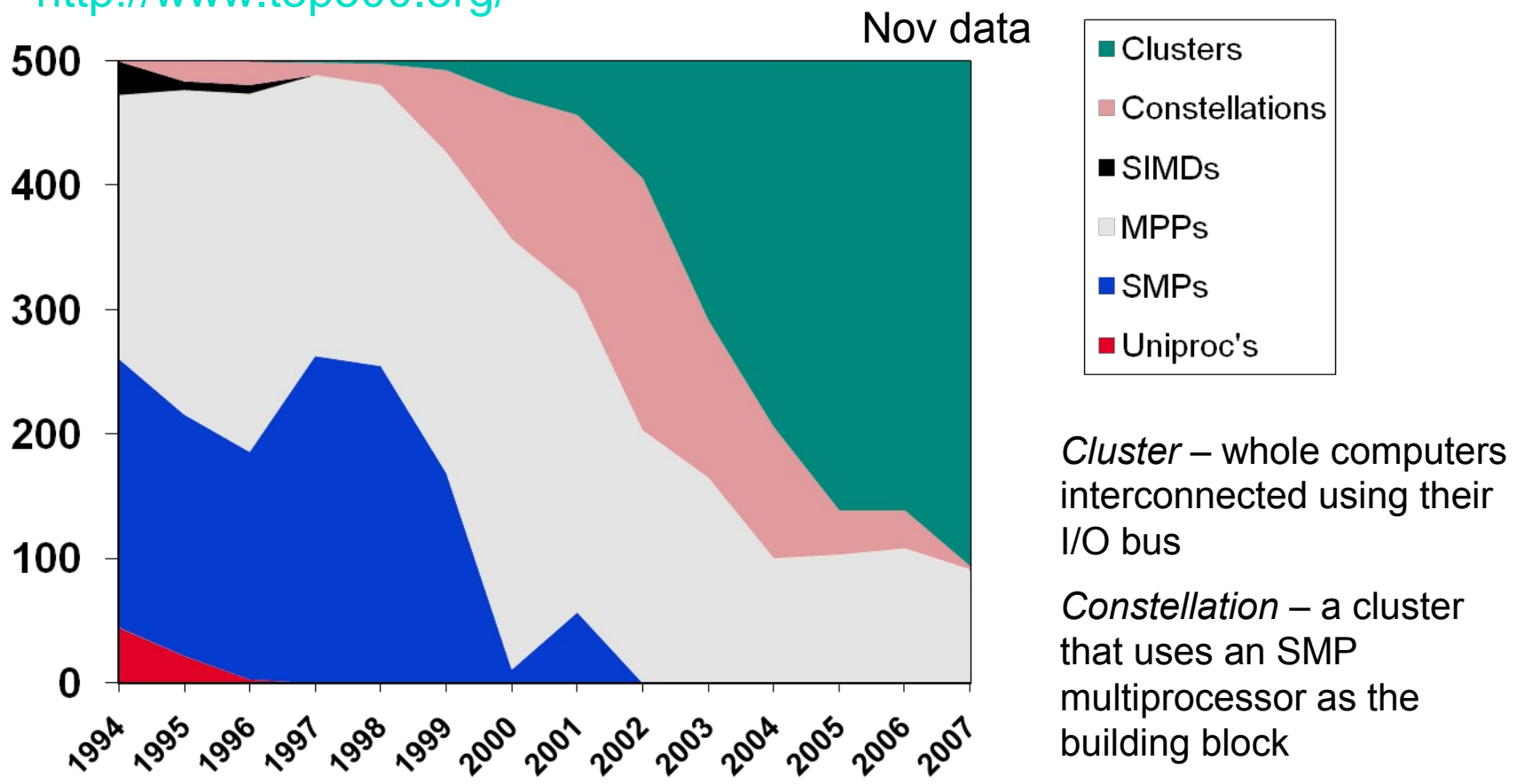
- GPUs are accelerators that supplement a CPU so they do not need to be able to perform all of the tasks of a CPU. They dedicate *all* of their resources to graphics
  - CPU-GPU combination – heterogeneous multiprocessing
- Programming interfaces that are free from backward binary compatibility constraints resulting in more rapid innovation in GPUs than in CPUs
  - Application programming interfaces (APIs) such as OpenGL and DirectX coupled with high-level graphics shading languages such as NVIDIA's Cg and CUDA and Microsoft's HLSL
- GPU data types are vertices (x, y, z, w) coordinates and pixels (red, green, blue, alpha) color components
- GPUs execute many threads (e.g., vertex and pixel shading) in parallel – *lots* of data-level parallelism

Typical GPU Architecture Features

- Rely on having enough threads to hide the latency to memory (not caches as in CPUs)
  - Each GPU is highly multithreaded
- Use extensive parallelism to get high performance
  - Have extensive set of SIMD instructions; moving towards multicore
- Main memory is bandwidth, not latency driven
  - GPU DRAMs are wider and have higher bandwidth, but are typically smaller, than CPU memories
- Leaders in the marketplace (in 2008)
  - NVIDIA GeForce 8800 GTX (16 multiprocessors each with 8 multithreaded processing units)
  - AMD's ATI Radeon and ATI FireGL
  - Watch out for Intel's Larrabee

Supercomputer Style Migration (Top500)

http://www.top500.org/



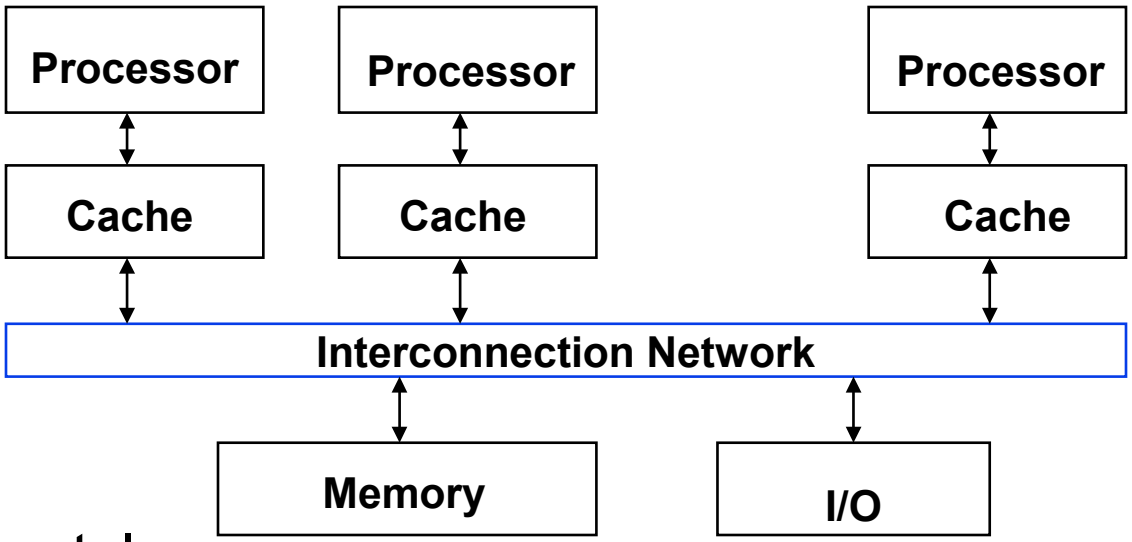
Cluster – whole computers interconnected using their I/O bus

Constellation – a cluster that uses an SMP multiprocessor as the building block

- Uniprocessors and SIMDs disappeared while Clusters and Constellations grew from 3% to 80%. Now it is 98% Clusters and MPPs.

Review: Shared Memory Multiprocessors (SMP)

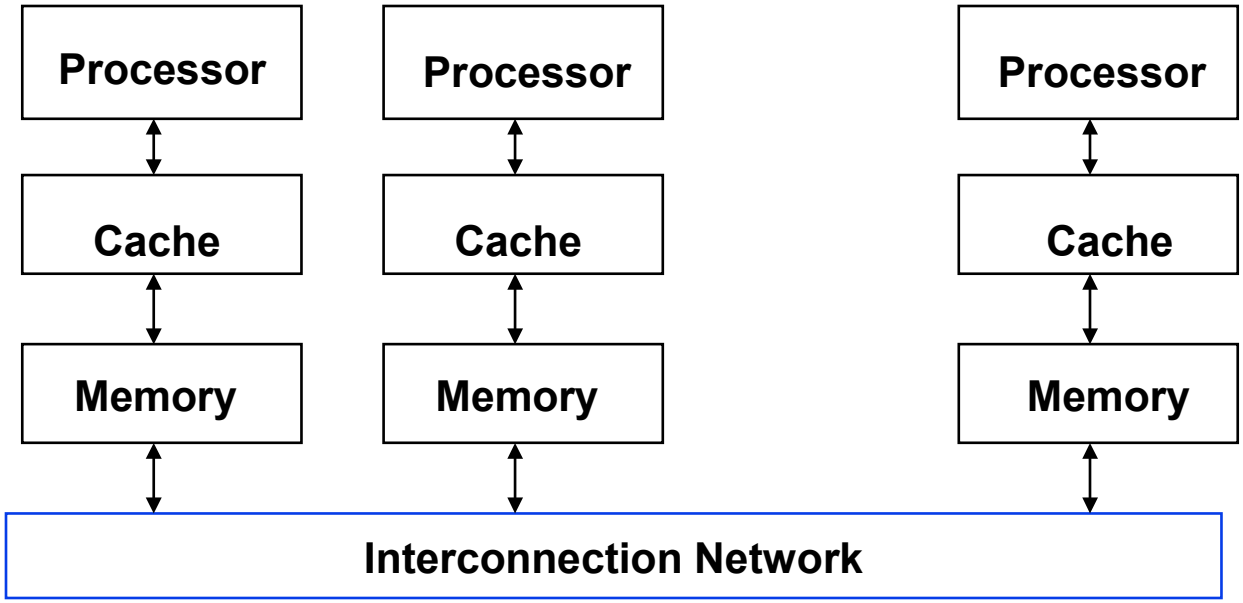
- Q1 – Single address space shared by all processors
- Q2 – Processors coordinate/communicate through shared variables in memory (via loads and stores)
  - Use of shared data must be coordinated via synchronization primitives (locks) that allow access to data to only one processor at a time



- They come in two styles
  - Uniform memory access (UMA) multiprocessors
  - Nonuniform memory access (NUMA) multiprocessors

Message Passing Multiprocessors (MPP)

- Each processor has its own private address space
- Q1 – Processors share data by *explicitly* sending and receiving information (message passing)
- Q2 – Coordination is built into message passing primitives (message send and message receive)



Communication in Network Connected Multi's

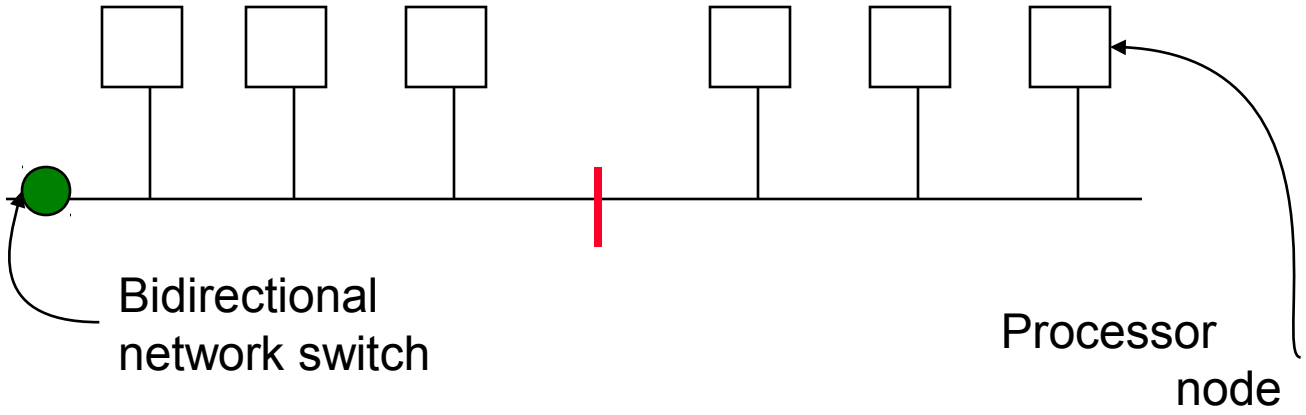
- Implicit communication via loads and stores
  - hardware designers have to provide coherent caches and process (thread) synchronization primitive (like `ll` and `sc`)
  - lower communication overhead
  - harder to overlap computation with communication
  - more efficient to use an address to remote data when *needed* rather than to send for it in case it *might* be used

- Explicit communication via sends and receives
  - simplest solution for hardware designers
  - higher communication overhead
  - easier to overlap computation with communication
  - easier for the programmer to optimize communication

IN Performance Metrics

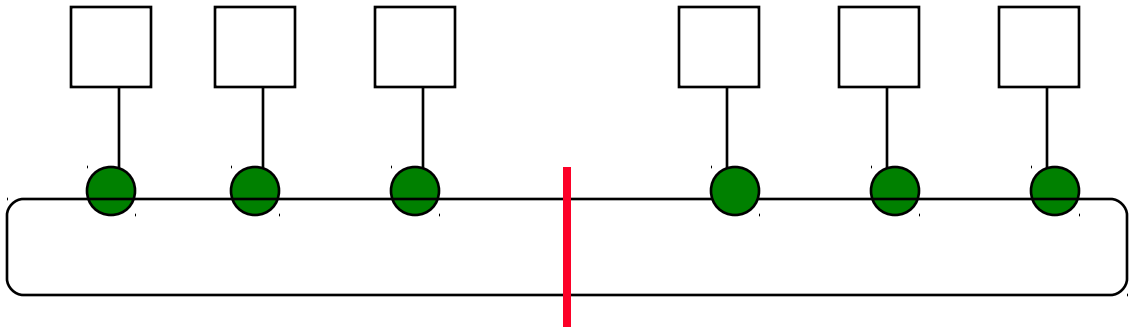
- Network cost
  - number of switches
  - number of (bidirectional) links on a switch to connect to the network (plus one link to connect to the processor)
  - width in bits per link, length of link wires (on chip)
- Network bandwidth (NB) – represents the best case
  - bandwidth of each link * number of links
- Bisection bandwidth (BB) – closer to the worst case
  - divide the machine in two parts, each with half the nodes and sum the bandwidth of the links that cross the dividing line
- Other IN performance issues
  - latency on an unloaded network to send and receive messages
  - throughput – maximum # of messages transmitted per unit time
  - # routing hops worst case, congestion control and delay, fault tolerance, power efficiency

Bus IN



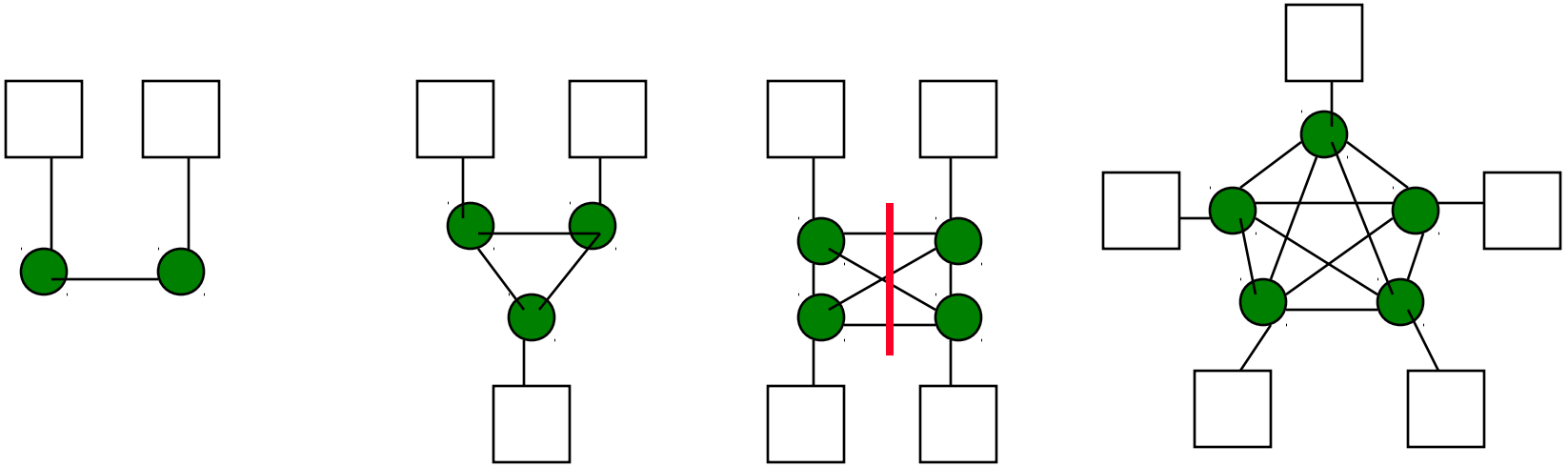
- N processors, 1 switch ( ), 1 link (the bus)
- Only 1 simultaneous transfer at a time
  - NB = link (bus) bandwidth * 1
  - BB = link (bus) bandwidth * 1

Ring IN



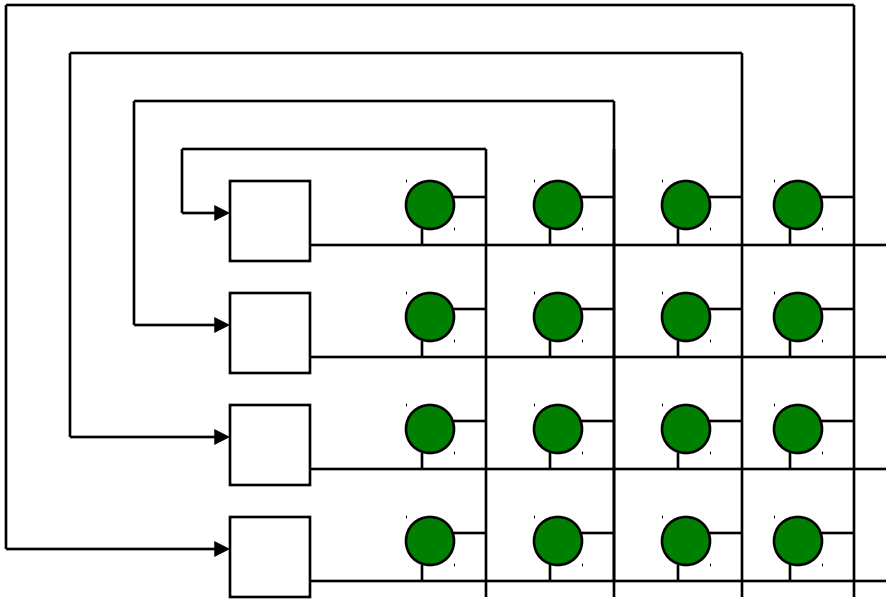
- N processors, N switches, 2 links/switch, N links
- N simultaneous transfers
  - NB = link bandwidth * N
  - BB = link bandwidth * 2
- If a link is as fast as a bus, the ring is only twice as fast as a bus in the worst case, but is N times faster in the best case

Fully Connected IN



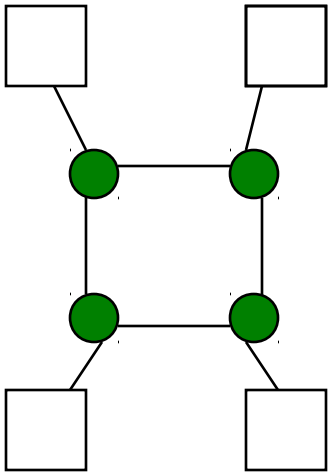
- N processors, N switches, N-1 links/switch, (N*(N-1))/2 links
- N simultaneous transfers
  - NB = link bandwidth * (N * (N-1))/2
  - BB = link bandwidth * $(N/2)^2$

Crossbar (Xbar) Connected IN

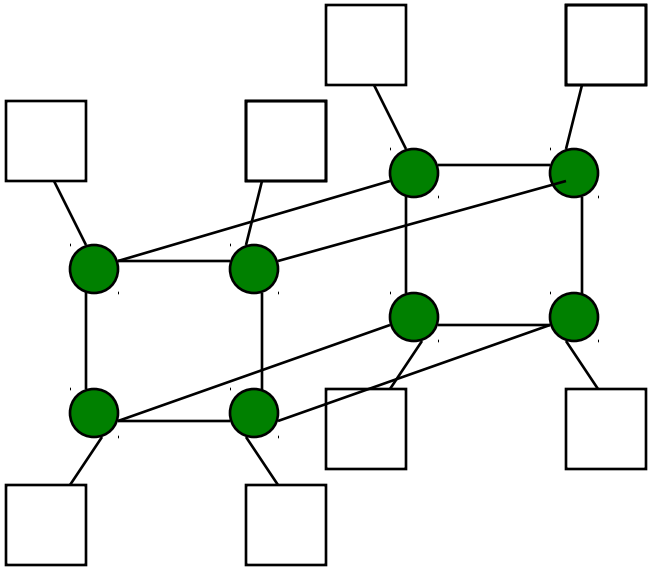


- N processors, N^2 switches (unidirectional), 2 links/switch, N^2 links
- N simultaneous transfers
  - NB = link bandwidth * N
  - BB = link bandwidth * N/2

Hypercube (Binary N-cube) Connected IN



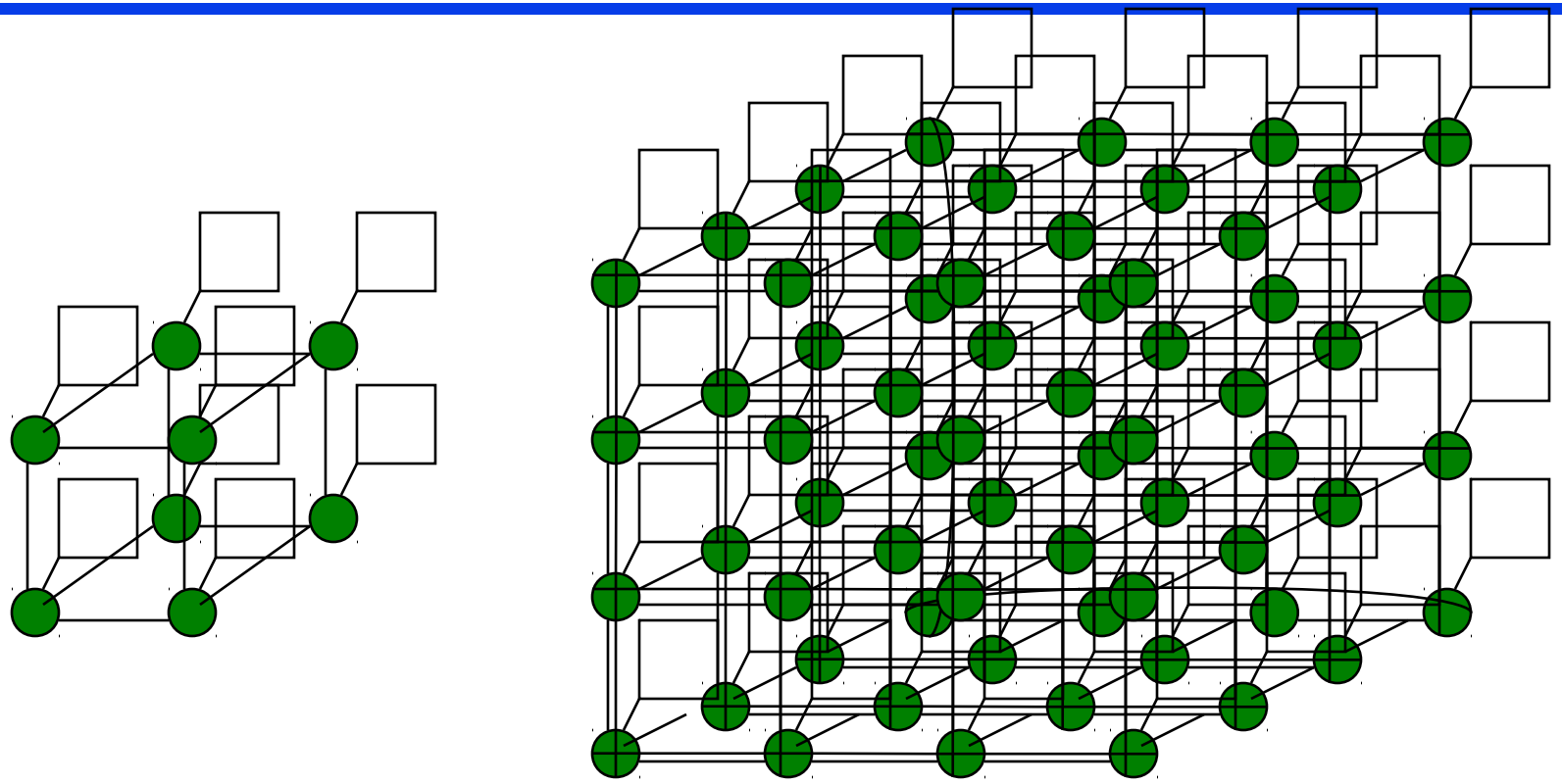
2-cube



3-cube

- N processors, N switches, logN links/switch, (NlogN)/2 links
- N simultaneous transfers
  - NB = link bandwidth * (NlogN)/2
  - BB = link bandwidth * N/2

2D and 3D Mesh/Torus Connected IN



- N processors, N switches, 2, 3, 4 (2D torus) or 6 (3D torus) links/switch, 4 N/2 links or 6 N/2 links
- N simultaneous transfers
  - NB = link bandwidth * 4N or link bandwidth * 6N
  - BB = link bandwidth * 2 $N^{1/2}$ or link bandwidth * 2 $N^{2/3}$

IN Comparison

- For a 64 processor system

	Bus	Ring	Torus	6-cube	Fully connected
Network bandwidth	1				
Bisection bandwidth	1				
Total # of switches	1				
Links per switch					
Total # of links (bidi)	1				

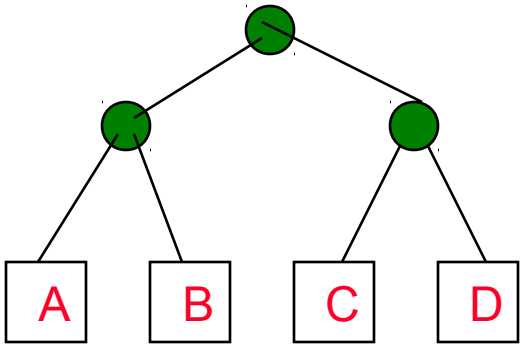
IN Comparison

- For a 64 processor system

	Bus	Ring	2D Torus	6-cube	Fully connected
Network bandwidth	1	64	256	192	2016
Bisection bandwidth	1	2	16	32	1024
Total # of switches	1	64	64	64	64
Links per switch		2+1	4+1	6+7	63+1
Total # of links (bidi)	1	64+64	128+64	192+64	2016+64

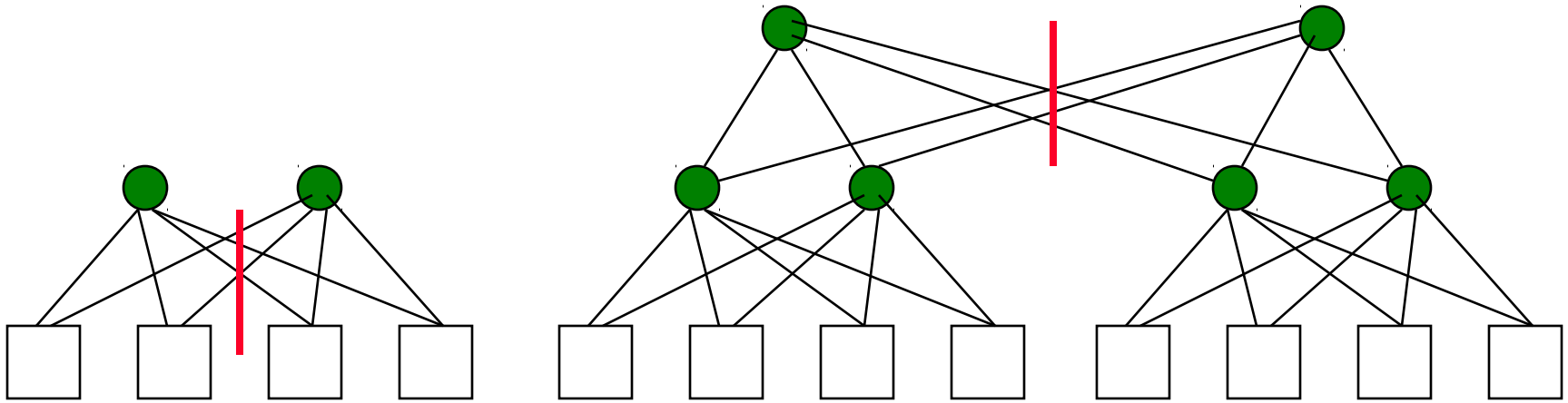
"Fat" Trees

- Trees are good structures. People in CS use them all the time. Suppose we wanted to make a tree network.



- Any time A wants to send to C, it ties up the upper links, so that B can't send to D.
  - The bisection bandwidth on a tree is horrible - 1 link, at all times
- The solution is to 'thicken' the upper links.
  - Have more links as you work towards the root of the tree increases the bisection bandwidth
- Rather than design a bunch of N-port switches, use pairs of switches

Fat Tree IN



- N processors, log(N-1) * logN switches, 2 up + 4 down = 6 links/switch, N * logN links
- N simultaneous transfers
  - NB = link bandwidth * NlogN
  - BB = link bandwidth * 4