Musk explains why SpaceX prefers clusters of small engines

"*It’s sort of like the way modern computer systems are set up.*"

The company's development of the Falcon 9 rocket, with nine engines, had given Musk confidence that SpaceX could scale up to 27 engines in flight, and he believed this was a better overall solution for the thrust needed to escape Earth's gravity. To explain why, the former computer scientist used a computer metaphor.

"It’s sort of like the way modern computer systems are set up," Musk said.

"**With Google or Amazon they have large numbers of small computers, such that if one of the computers goes down it doesn’t really affect your use of Google or Amazon. That’s different from the old model of the mainframe approach, when you have one big mainframe and if it goes down, the whole system goes down.**"

Cook analogy

- We want to prepare food for several banquets, each of which requires many dinners.
- We have two positions we can fill:
  - The boss (control), who gets all the ingredients and tells the chef what to do
  - The chef (datapath), who does all the cooking
- ILP is analogous to:
  - One ultra-talented boss with many hands
  - One ultra-talented chef with many hands

Cook analogy

- We want to prepare food for several banquets, each of which requires many dinners.
- But one boss and one chef isn't enough to do all our cooking.
- What are our options?

Chef scaling

- What's the cheapest way to cook more?
- Is it easy or difficult to share (ingredients, cooked food, etc.) between chefs?
- Which method of scaling is most flexible?

Flynn's Classification Scheme

- SISD – single instruction, single data stream
  - Uniprocessors
- 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)

Performance beyond single thread ILP

- There can be much higher natural parallelism in some applications (e.g., database or scientific codes)
- Explicit Thread Level Parallelism or Data Level Parallelism
- Thread: process with own instructions and data
  - Thread may be a subpart of a parallel program (“thread”), or it may be an independent program (“process”)
  - Each thread has all the state (instructions, data, PC, register state, and so on) necessary to allow it to execute
  - Many kitchens, each with own boss and chef
- Data Level Parallelism: Perform identical operations on data, and lots of data
  - 1 kitchen, 1 boss, many chefs

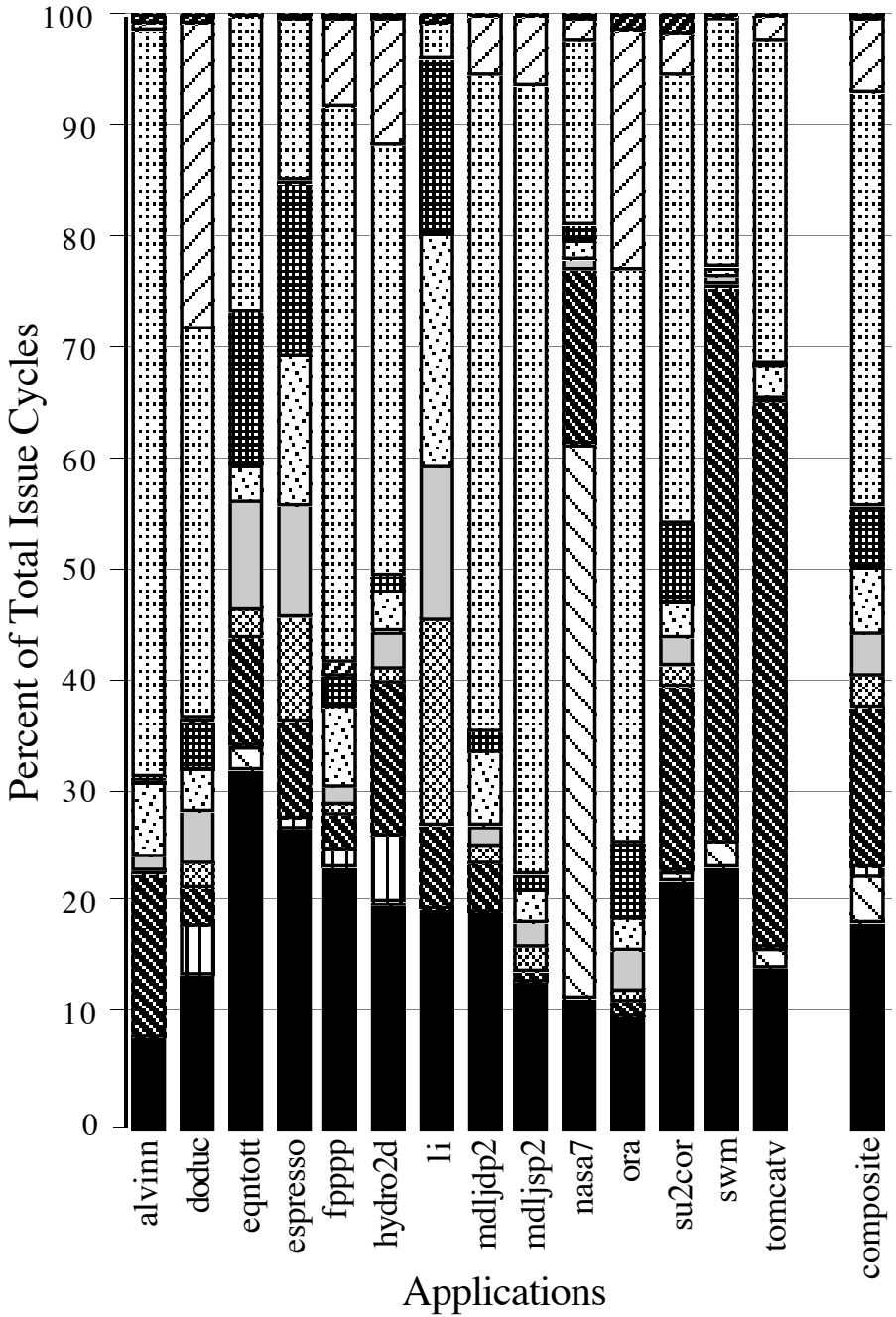
Continuum of Granularity

- "Coarse"
  - Each processor is more powerful
  - Usually fewer processors
  - Communication is more expensive between processors
  - Processors are more loosely coupled
  - Tend toward MIMD
- "Fine"
  - Each processor is less powerful
  - Usually more processors
  - Communication is cheaper between processors
  - Processors are more tightly coupled
  - Tend toward SIMD

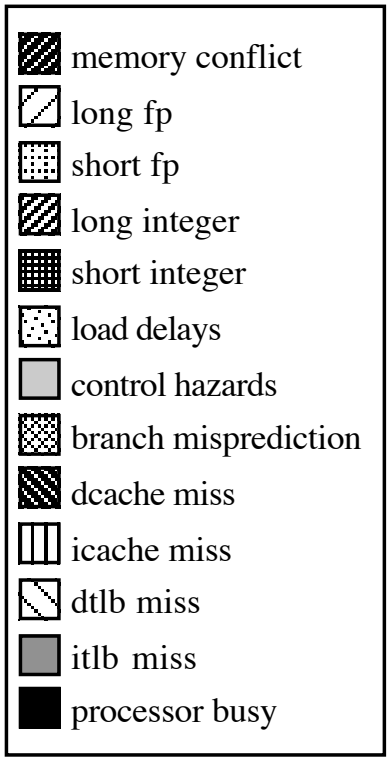
Thread Level Parallelism

- ILP exploits implicit parallel operations within a loop or straight-line code segment
- TLP explicitly represented by the use of multiple threads of execution that are inherently parallel
  - ***You must rewrite your code to be thread-parallel.***
- Goal: Use multiple instruction streams to improve
  - Throughput of computers that run many programs
  - Execution time of multi-threaded programs
- TLP could be more cost-effective to exploit than ILP

For most apps, most execution units lie idle



[8-way superscalar]

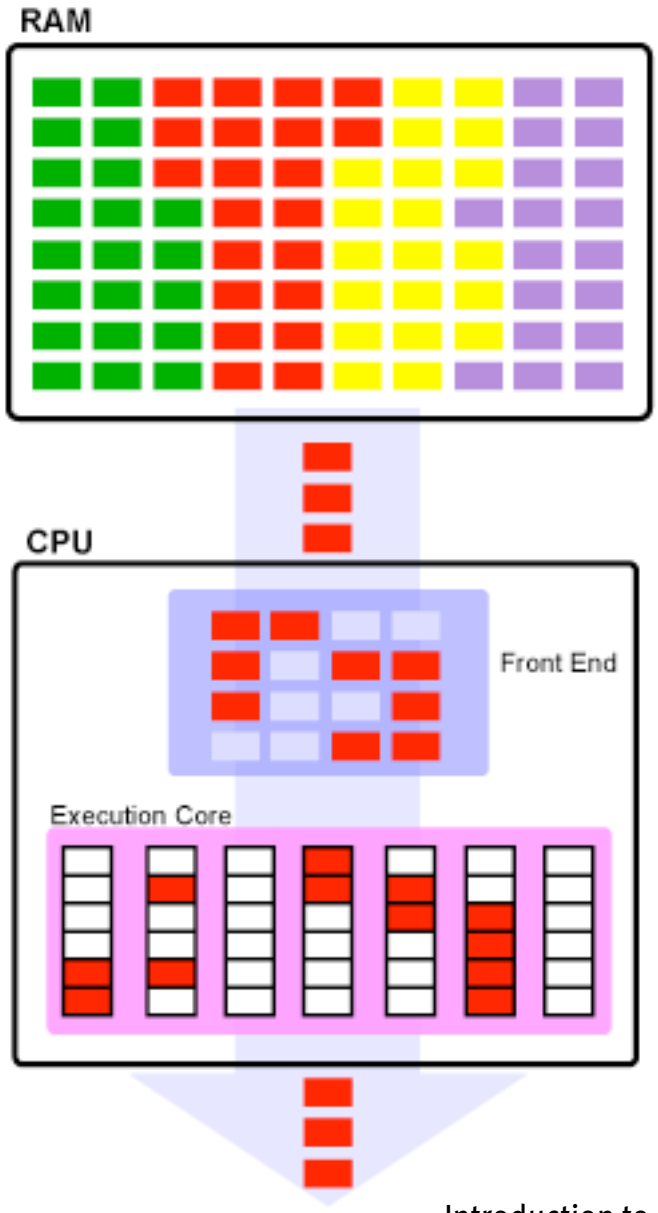


From: Tullsen, Eggers, and Levy, "Simultaneous Multithreading: Maximizing On-chip Parallelism, ISCA 1995.

Source of Wasted Slots

Source of Wasted Issue Slots	Possible Latency-Hiding or Latency-Reducing Technique
instruction tlb miss, data tlb miss	decrease the TLB miss rates (e.g., increase the TLB sizes); hardware instruction prefetching; hardware or software data prefetching; faster servicing of TLB misses
I cache miss	larger, more associative, or faster instruction cache hierarchy; hardware instruction prefetching
D cache miss	larger, more associative, or faster data cache hierarchy; hardware or software prefetching; improved instruction scheduling; more sophisticated dynamic execution
branch misprediction	improved branch prediction scheme; lower branch misprediction penalty
control hazard	speculative execution; more aggressive if-conversion
load delays (first-level cache hits)	shorter load latency; improved instruction scheduling; dynamic scheduling
short integer delay	improved instruction scheduling
long integer, short fp, long fp delays	(multiply is the only long integer operation, divide is the only long floating point operation) shorter latencies; improved instruction scheduling
memory conflict	(accesses to the same memory location in a single cycle) improved instruction scheduling

Single-threaded CPU

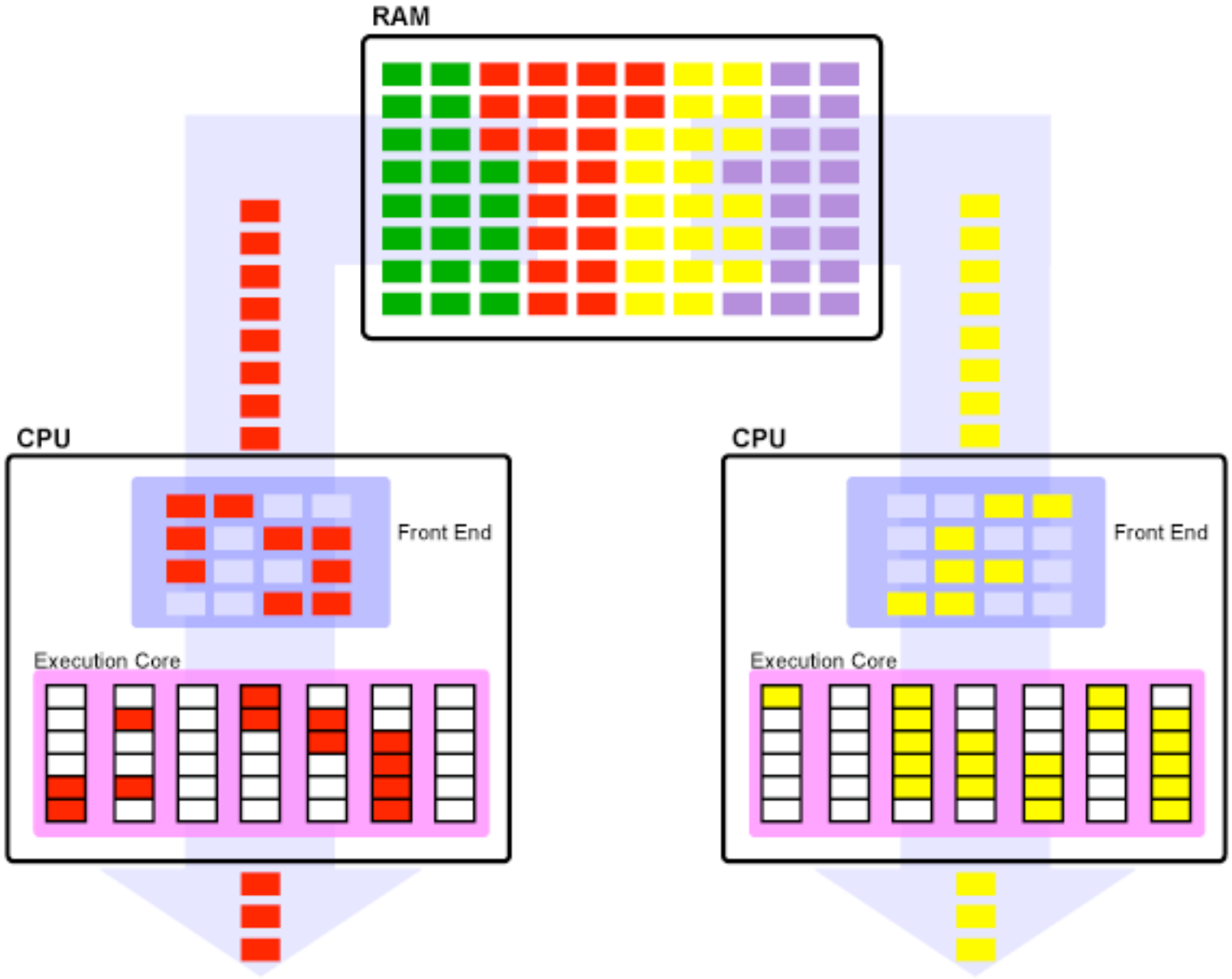


Introduction to Multithreading, Superthreading and Hyperthreading
By Jon Stokes
http://arstechnica.com/articles/paedia/cpu/hyperthreading.ars

We can add more CPUs …

- … and we'll talk about this later in the class
- Note we have multiple CPUs reading out of the same instruction store
- Is this more efficient than having one CPU?

Symmetric Multiprocessing



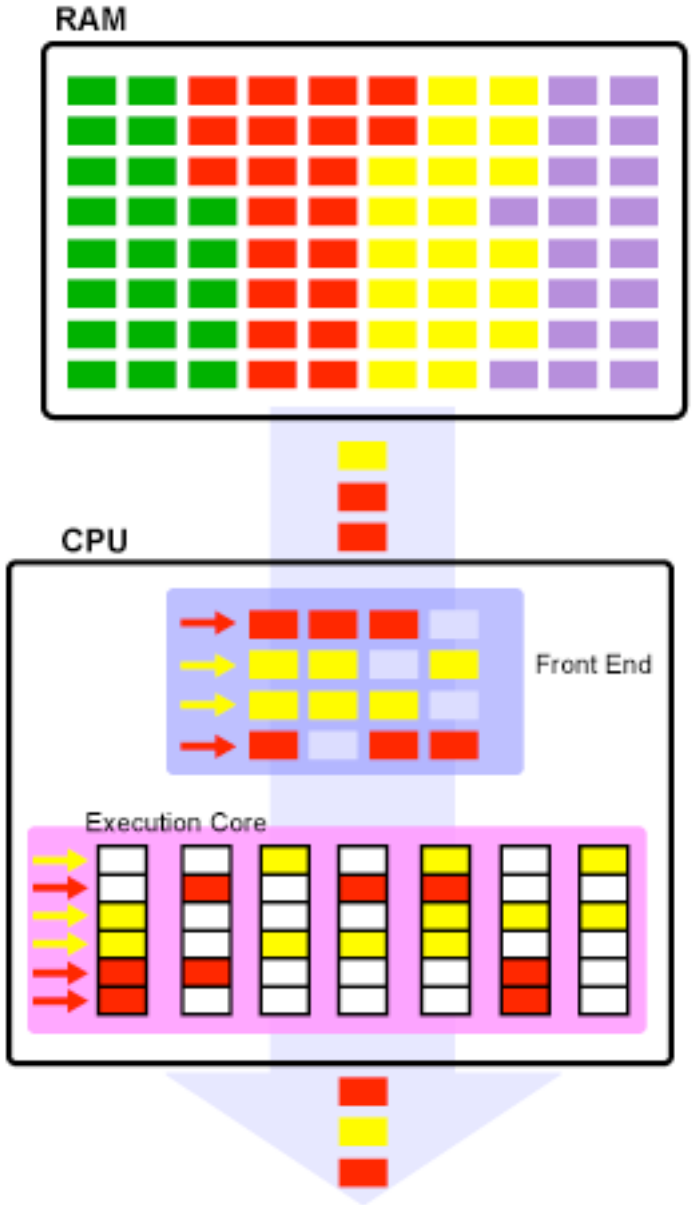
Conventional Multithreading

- How does a microprocessor run multiple processes / threads “at the same time”?
  - How does one program interact with another program?
- What is preemptive multitasking vs. cooperative multitasking?

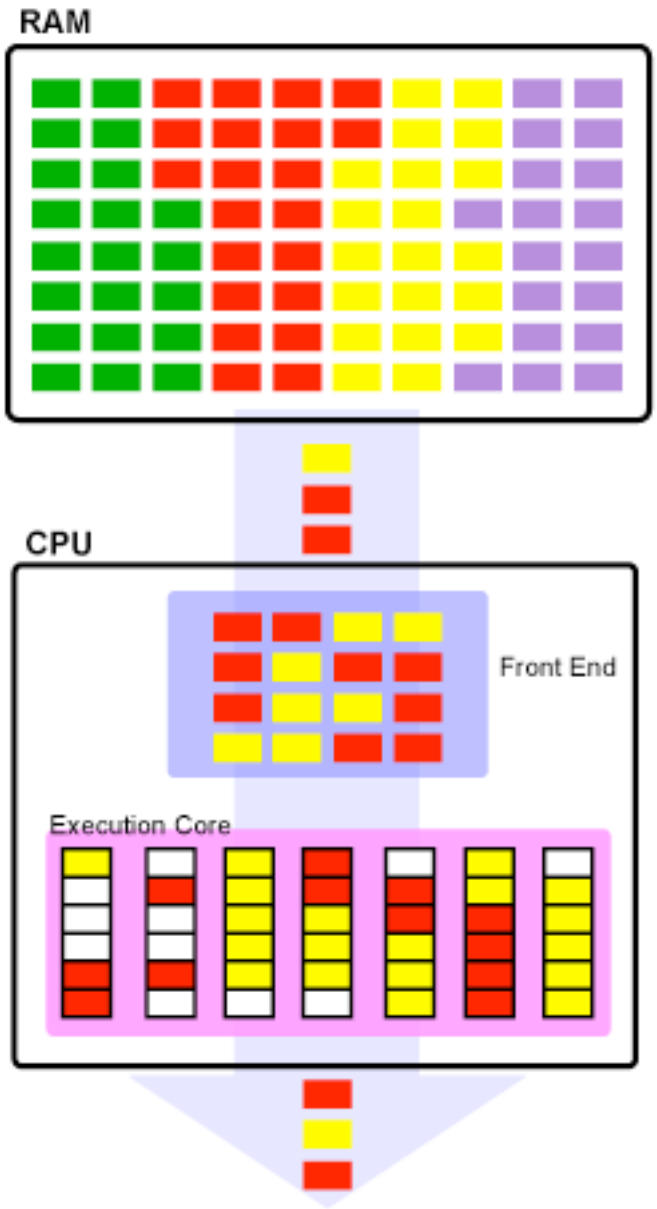
New Approach: Multithreaded Execution

- Multithreading: multiple threads to share the functional units of 1 processor via overlapping
  - processor must duplicate independent state of each thread e.g., a separate copy of register file, a separate PC, and for running independent programs, a separate page table
  - memory shared through the virtual memory mechanisms, which already support multiple processes
  - HW for fast thread switch; much faster than full process switch ≈ 100s to 1000s of clocks

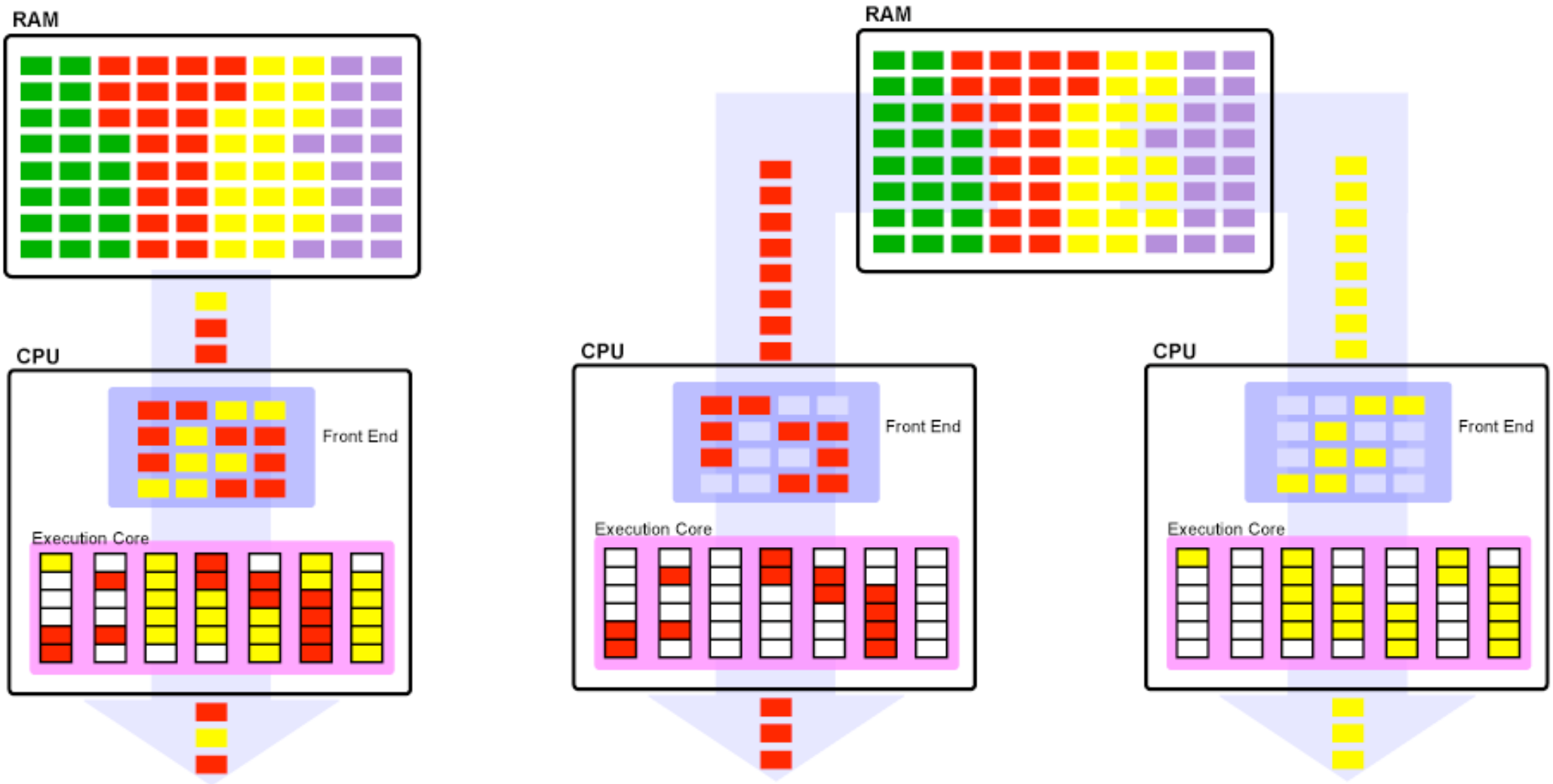
Superthreading



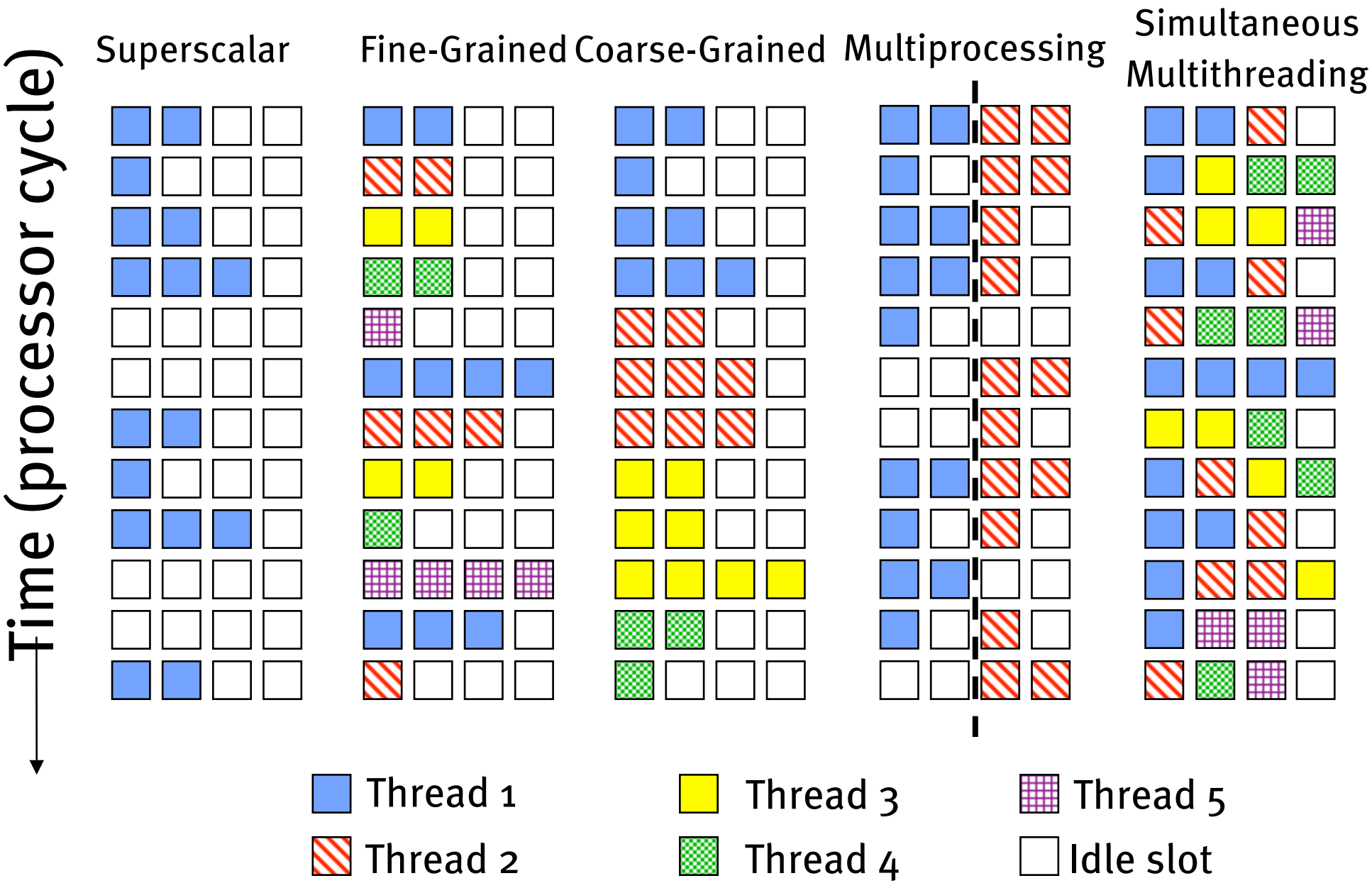
Simultaneous multithreading (SMT)



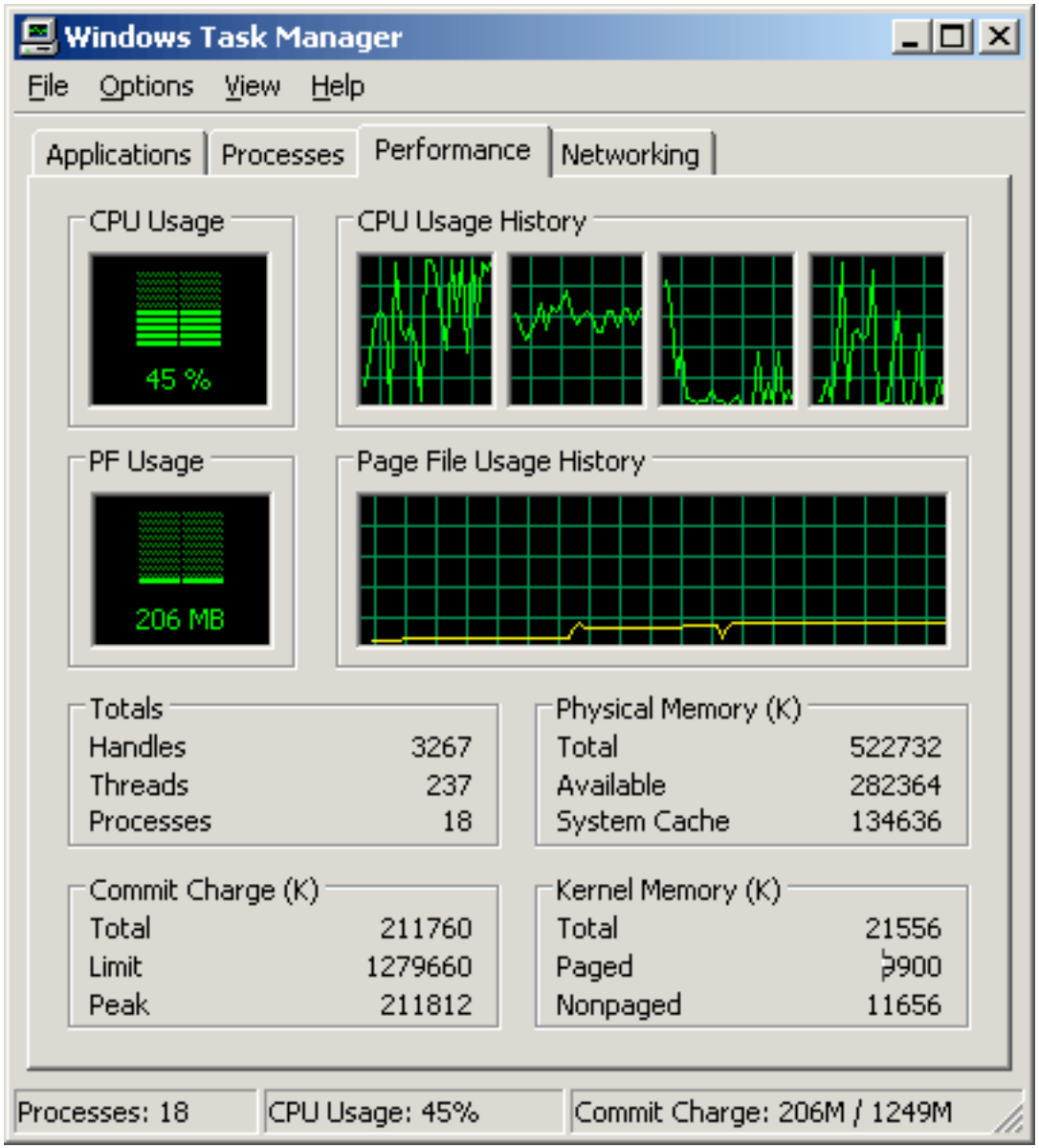
Simultaneous multithreading (SMT)



Multithreaded Categories



“Hyperthreading”



http://www.2cpu.com/Hardware/ht_analysis/images/taskmanager.html

Multithreaded Execution

- When do we switch between threads?
  - Alternate instruction per thread (fine grain)
  - When a thread is stalled, perhaps for a cache miss, another thread can be executed (coarse grain)

Fine-Grained Multithreading

- Switches between threads on each instruction, causing the execution of multiple threads to be interleaved
- Usually done in a round-robin fashion, skipping any stalled threads
- CPU must be able to switch threads every clock
- Advantage is it can hide both short and long stalls, since instructions from other threads executed when one thread stalls
- Disadvantage is it slows down execution of individual threads, since a thread ready to execute without stalls will be delayed by instructions from other threads
- Used on Sun’s Niagara (will see later)

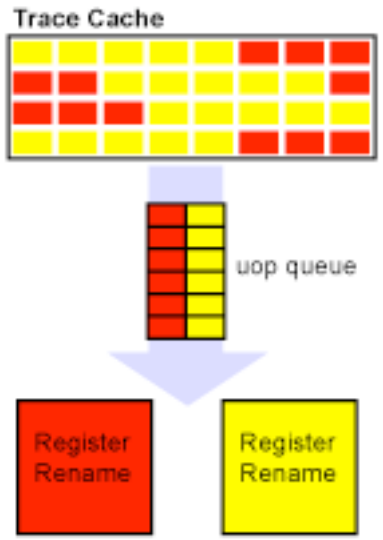
Coarse-Grained Multithreading

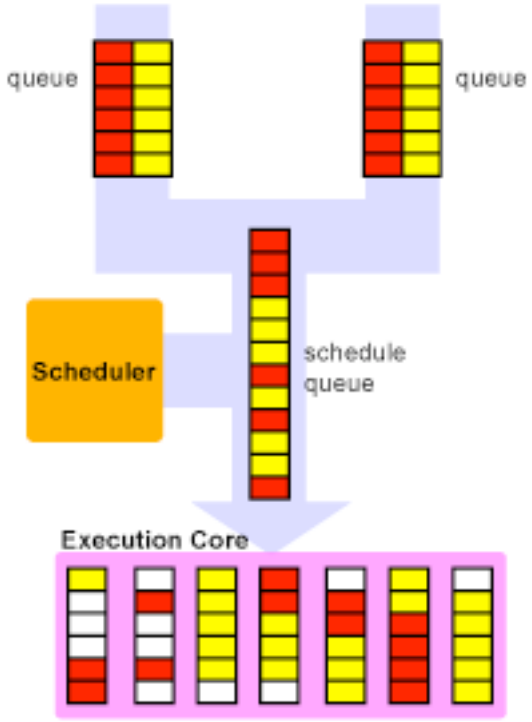
- Switches threads only on costly stalls, such as L2 cache misses
- Advantages
  - Relieves need to have very fast thread-switching
  - Doesn't slow down thread, since instructions from other threads issued only when the thread encounters a costly stall
- Disadvantage is hard to overcome throughput losses from shorter stalls, due to pipeline start-up costs
  - Since CPU issues instructions from 1 thread, when a stall occurs, the pipeline must be emptied or frozen
  - New thread must fill pipeline before instructions can complete
- Because of this start-up overhead, coarse-grained multithreading is better for reducing penalty of high cost stalls, where pipeline refill << stall time
- Used in IBM AS/400

P4Xeon Microarchitecture

- Replicated
  - Register renaming logic
  - Instruction pointer, other architectural registers
  - ITLB
  - Return stack predictor
- Partitioned
  - Reorder buffers
  - Load/store buffers
  - Various queues: scheduling, uop, etc.
- Shared
  - Caches (trace, L1/L2/L3)
  - Microarchitectural registers
  - Execution units
- If configured as single-threaded, all resources go to one thread

Partitioning: Static vs. Dynamic





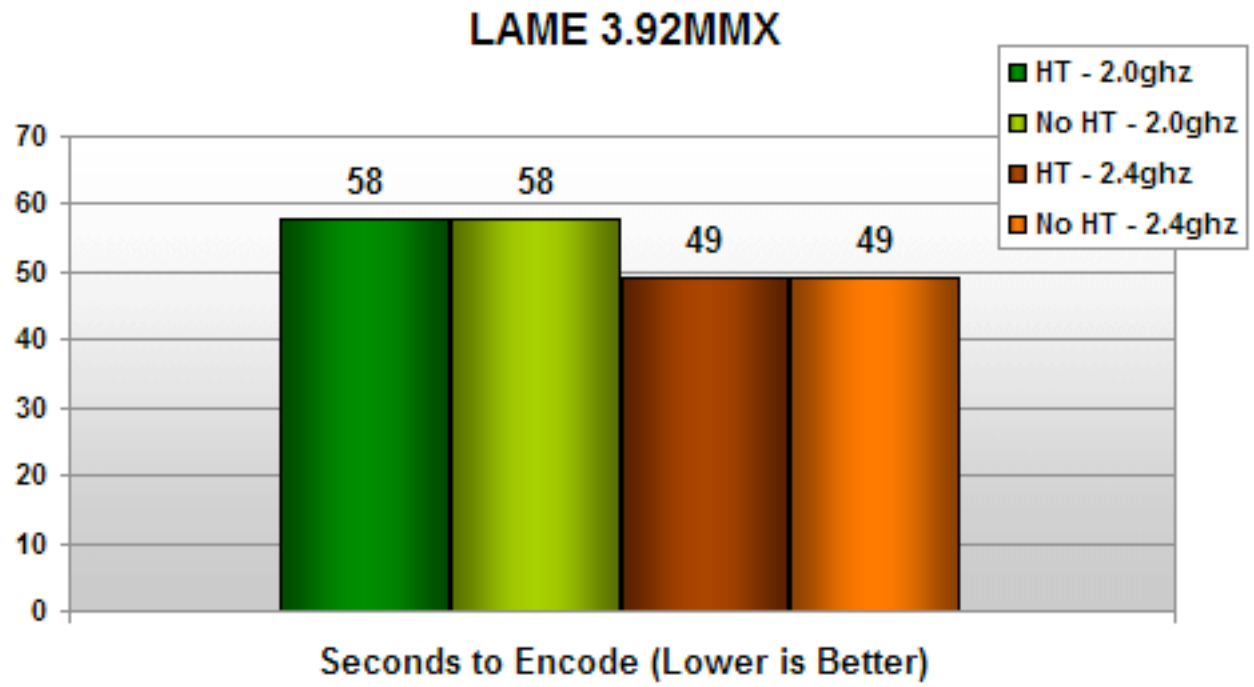
Design Challenges in SMT

- Since SMT makes sense only with fine-grained implementation, impact of fine-grained scheduling on single thread performance?
  - A preferred thread approach sacrifices neither throughput nor single-thread performance?
  - Unfortunately, with a preferred thread, the processor is likely to sacrifice some throughput, when preferred thread stalls
- Larger register file needed to hold multiple contexts
- Not affecting clock cycle time, especially in
  - Instruction issue—more candidate instructions need to be considered
  - Instruction completion—choosing which instructions to commit may be challenging
- Ensuring that cache and TLB conflicts generated by SMT do not degrade performance

Problems with SMT

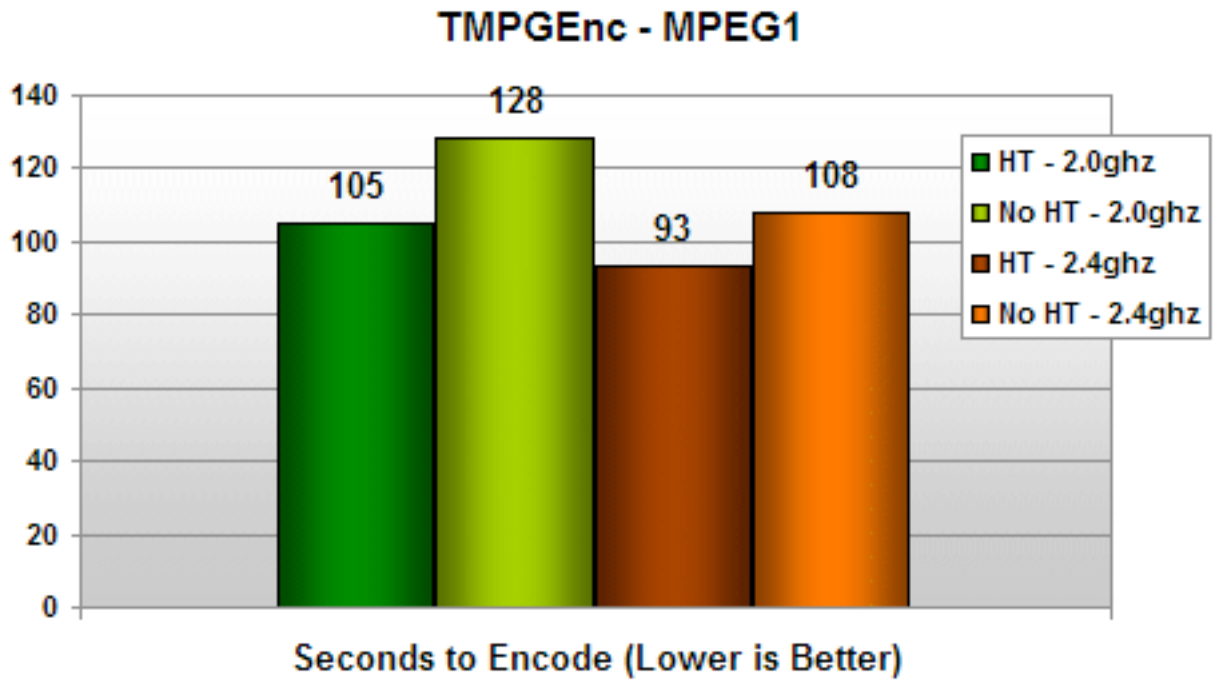
- One thread monopolizes resources
  - Example: One thread ties up FP unit with long-latency instruction, other thread tied up in scheduler
- Cache effects
  - Caches are unaware of SMT—can't make warring threads cooperate
  - If both warring threads access different memory and have cache conflicts, constant swapping

Hyperthreading Neutral!



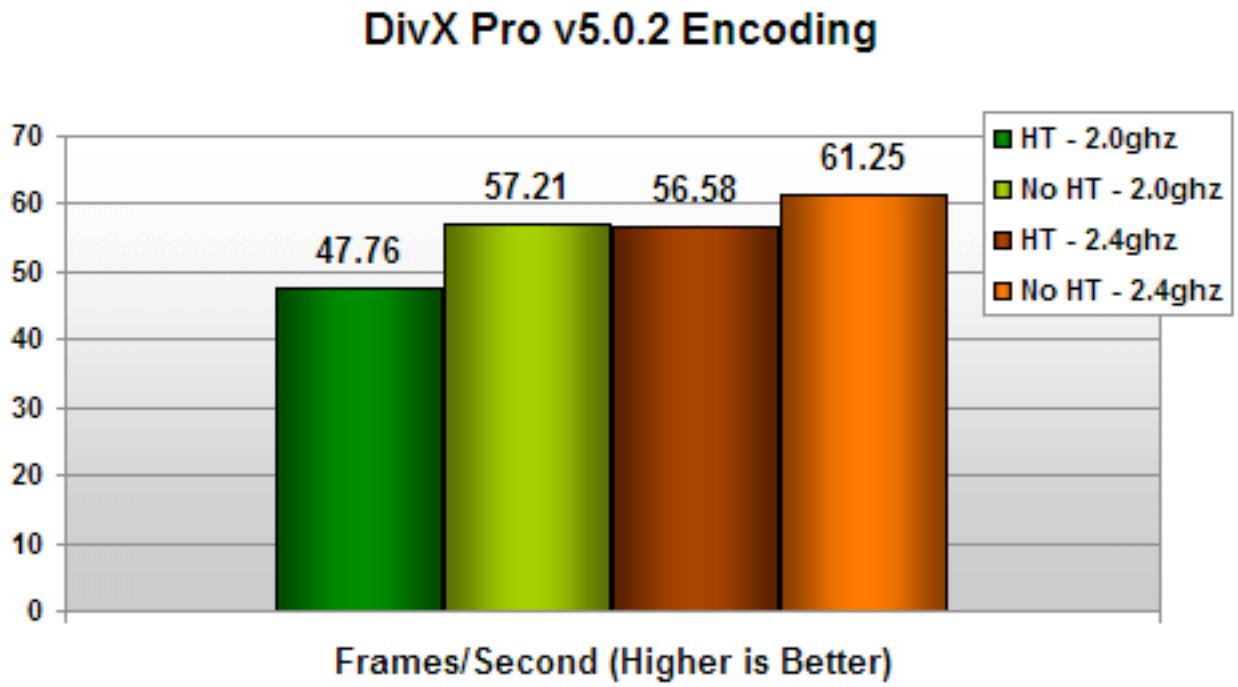
http://www.2cpu.com/articles/43_1.html

Hyperthreading Good!



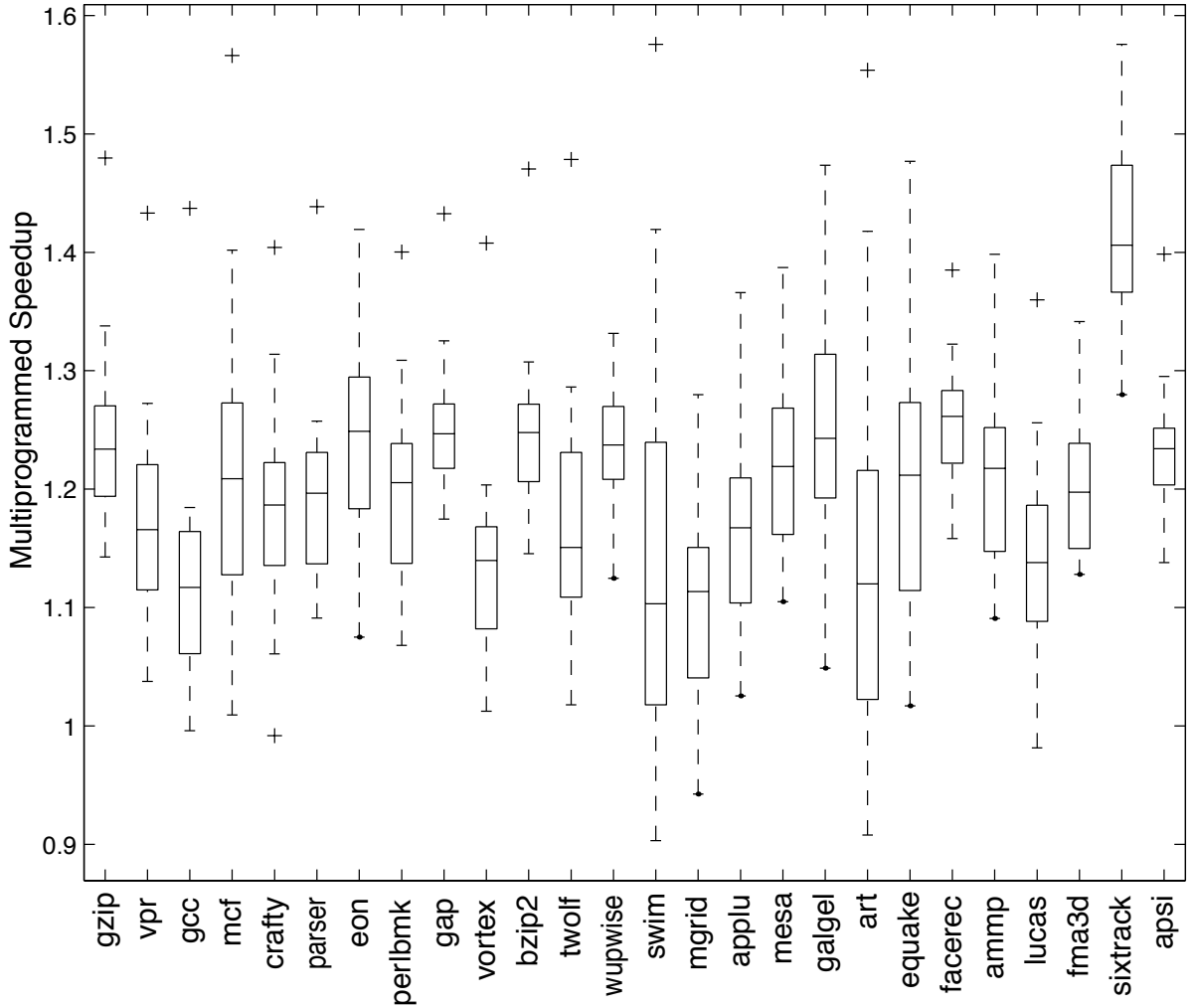
http://www.2cpu.com/articles/43_1.html

Hyperthreading Bad!



http://www.2cpu.com/articles/43_1.html

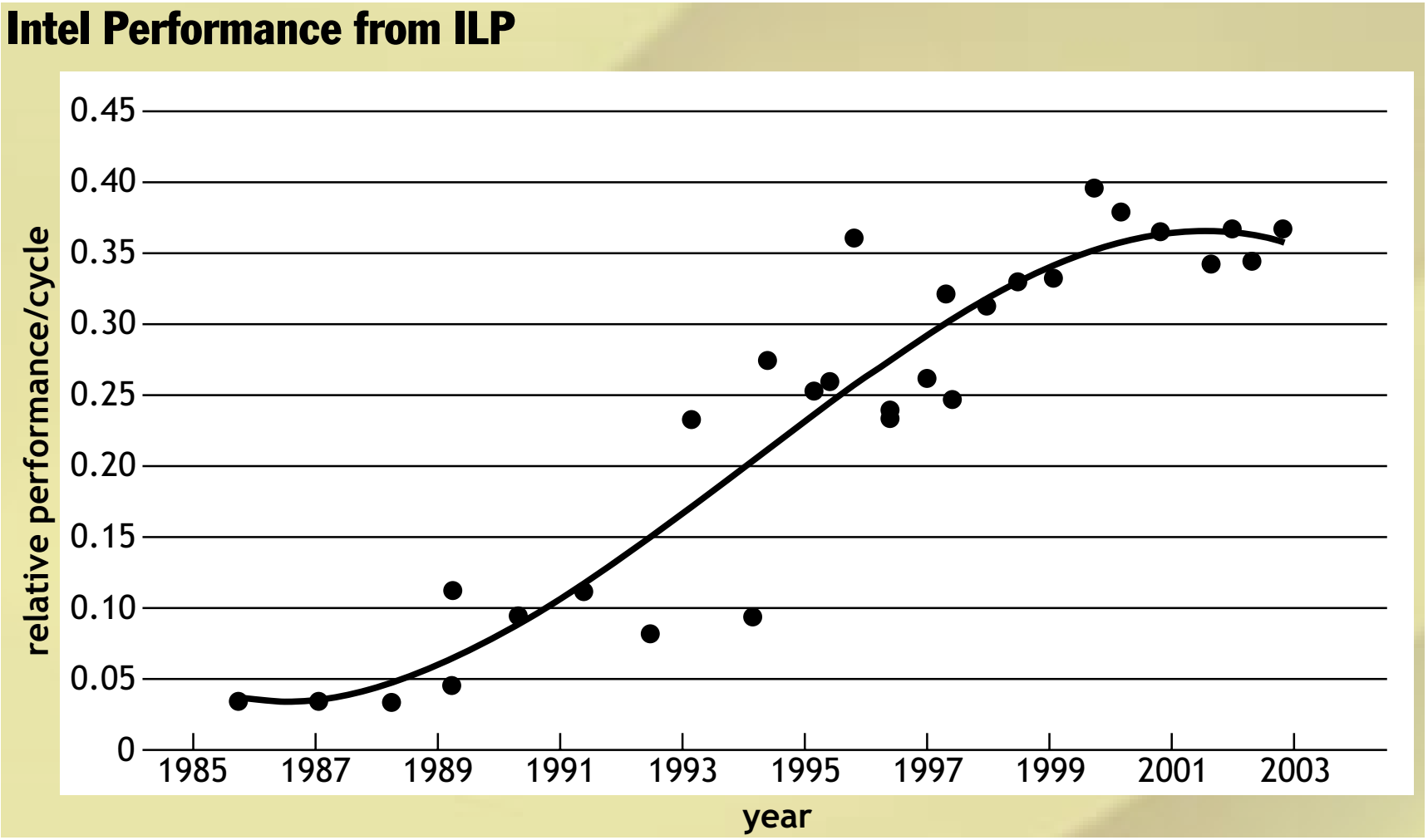
SPEC vs. SPEC (PACT '03)



- Avg. multithreaded speedup 1.20 (range 0.90–1.58)

"Initial Observations of the Simultaneous Multithreading Pentium 4 Processor", Nathan Tuck and Dean M. Tullsen (PACT '03)

ILP reaching limits



- Olukotun and Hammond, “The Future of Microprocessors”, *ACM Queue*, Sept. 2005

Olukotun's view

- "With the exhaustion of essentially all performance gains that can be achieved for 'free' with technologies such as superscalar dispatch and pipelining, we are now entering an era where programmers must switch to more parallel programming models in order to exploit multi-processors effectively, if they desire improved single-program performance."

Olukotun (pt. 2)

- “This is because there are only three real ‘dimensions’ to processor performance increases beyond Moore’s law: clock frequency, superscalar instruction issue, and multiprocessing. We have pushed the first two to their logical limits and must now embrace multiprocessing, even if it means that programmers will be forced to change to a parallel programming model to achieve the highest possible performance.”

Google's Architecture

- "Web Search for a Planet: The Google Cluster Architecture"
  - Luiz André Barroso, Jeffrey Dean, Urs Hölzle, Google
- Reliability in software not in hardware
- 2003: 15k commodity PCs
- July 2006 (estimate): 450k commodity PCs
  - $2M/month for electricity

Goal: Price/performance

- “We purchase the CPU generation that currently gives the best performance per unit price, not the CPUs that give the best absolute performance.”
  - Google rack: 40–80 x86 servers
    - “Our focus on price/performance favors servers that resemble mid-range desktop PCs in terms of their components, except for the choice of large disk drives.”
    - 4-processor motherboards: better perf, but not better price/perf
    - SCSI disks: better perf and reliability, but not better price/perf
  - Depreciation costs: $7700/month; power costs: $1500/month
    - Low-power systems must have equivalent performance

Google power density

- Mid-range server, dual 1.4 GHz Pentium III: 90 watts
  - 55 W for 2 CPUs
  - 10 W for disk drive
  - 25 W for DRAM/motherboard
    - so 120 W of AC power (75% efficient)
- Rack fits in 25 ft^2
  - 400 W/ft^2; high end processors 700 W/ft^2
  - Typical data center: 70–150 W/ft^2
  - Cooling is a big issue

Google Workload (1 GHz P3)

Table 1. Instruction-level measurements on the index server.

Characteristic	Value
Cycles per instruction	1.1
Ratios (percentage)	
Branch mispredict	5.0
Level 1 instruction miss*	0.4
Level 1 data miss*	0.7
Level 2 miss*	0.3
Instruction TLB miss*	0.04
Data TLB miss*	0.7

* Cache and TLB ratios are per instructions retired.

Details of workload

- “Moderately high CPI” (P3 can issue 3 instrs/cycle)
  - “Significant number of difficult-to-predict branches”
  - Same workload on P4 has “nearly twice the CPI and approximately the same branch prediction performance”
- “In essence, there isn’t that much exploitable instruction-level parallelism in the workload.”
- “Our measurements suggest that the level of aggressive out-of-order, speculative execution present in modern processors is already beyond the point of diminishing performance returns for such programs.”

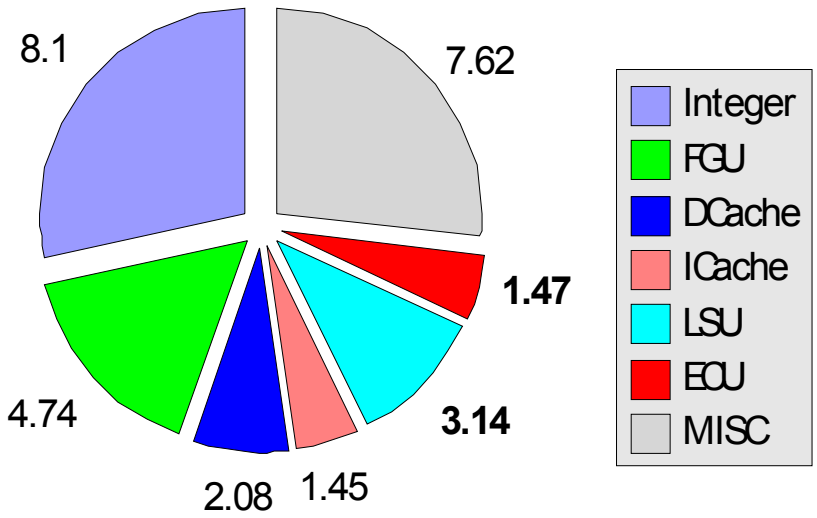
Google and SMT

- “A more profitable way to exploit parallelism for applications such as the index server is to leverage the trivially parallelizable computation.”
- “Exploiting such abundant thread-level parallelism at the microarchitecture level appears equally promising. Both simultaneous multithreading (SMT) and chip multiprocessor (CMP) architectures target thread-level parallelism and should improve the performance of many of our servers.”
- “Some early experiments with a dual-context (SMT) Intel Xeon processor show more than a 30 percent performance improvement over a single-context setup.”

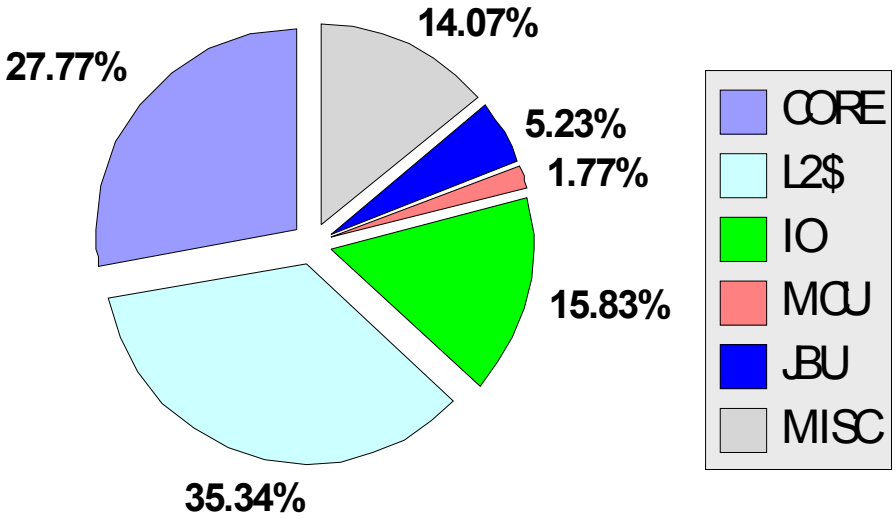
CMP: Chip Multiprocessing

- First CMPs: Two or more conventional superscalar processors on the same die
  - UltraSPARC Gemini, SPARC64 VI, Itanium Montecito, IBM POWER4
- One of the most important questions: What do cores share and what is not shared between cores?

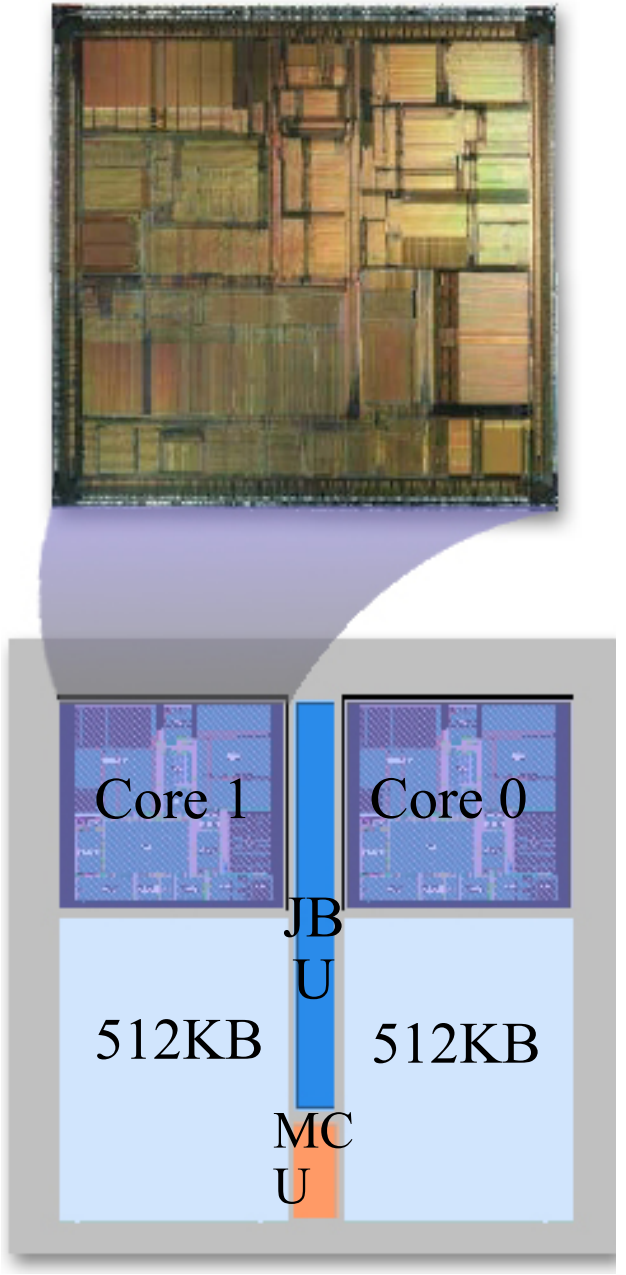
UltraSPARC Gemini



Core Area = 28.6 mm^2

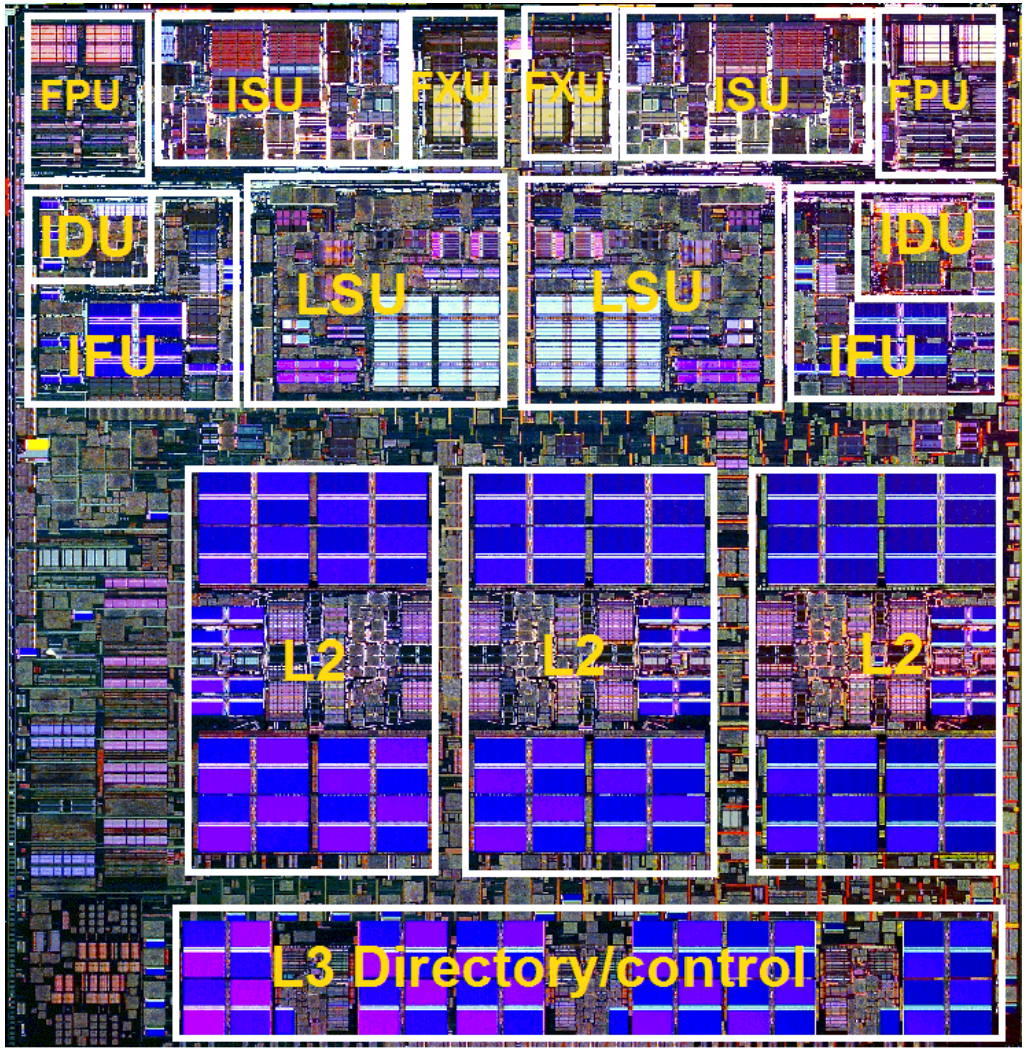


CPU Area = 206 mm^2



POWER5

- Technology: 130nm lithography, Cu, SOI
- Dual processor core
- 8-way superscalar
- Simultaneous multithreaded (SMT) core
  - Up to 2 virtual processors per real processor
  - 24% area growth per core for SMT
  - Natural extension to POWER4 design



CMP Benefits

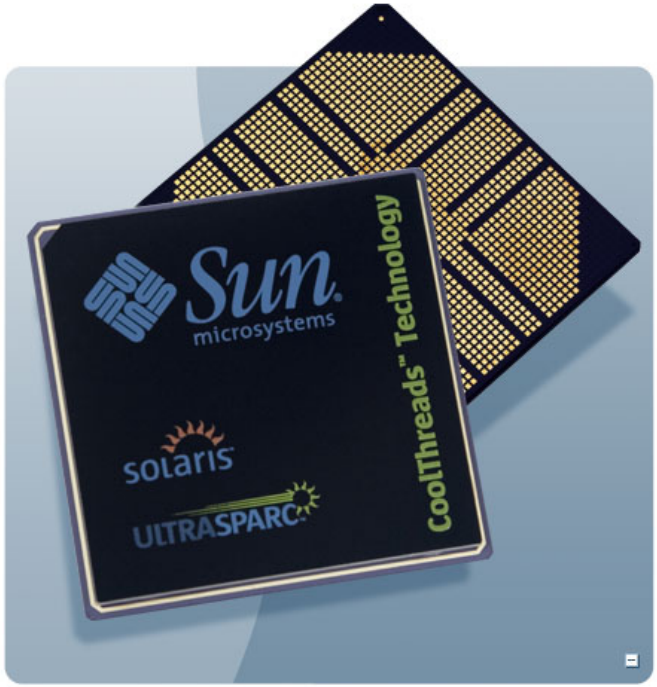
- Volume: 2 processors where 1 was before
- Power: All processors on one die share a single connection to rest of system

CMP Power

- Consider a 2-way CMP replacing a uniprocessor
- Run the CMP at half the uniprocessor's clock speed
  - Each request takes twice as long to process …
  - … but slowdown is less because request processing is likely limited by memory or disk
  - If there's not much contention, overall throughput is the same
- Half clock rate -› half voltage -› quarter power per processor, so 2x savings overall

Sun T1 (“Niagara”)

- Target: Commercial server applications
  - High thread level parallelism (TLP)
    - Large numbers of parallel client requests
  - Low instruction level parallelism (ILP)
    - High cache miss rates
    - Many unpredictable branches
    - Frequent load-load dependencies



- Power, cooling, and space are major concerns for data centers
- Metric: Performance/Watt/Sq. Ft.
- Approach: Multicore, Fine-grain multithreading, Simple pipeline, Small L1 caches, Shared L2

T1 Fine-Grained Multithreading

- Each core supports four threads and has its own level one caches (16 KB for instructions and 8 KB for data)
- Switching to a new thread on each clock cycle
- Idle threads are bypassed in the scheduling
  - Waiting due to a pipeline delay or cache miss
  - Processor is idle only when all 4 threads are idle or stalled
- Both loads and branches incur a 3 cycle delay that can only be hidden by other threads
- A single set of floating point functional units is shared by all 8 cores
  - Floating point performance was not a focus for T1

Microprocessor Comparison

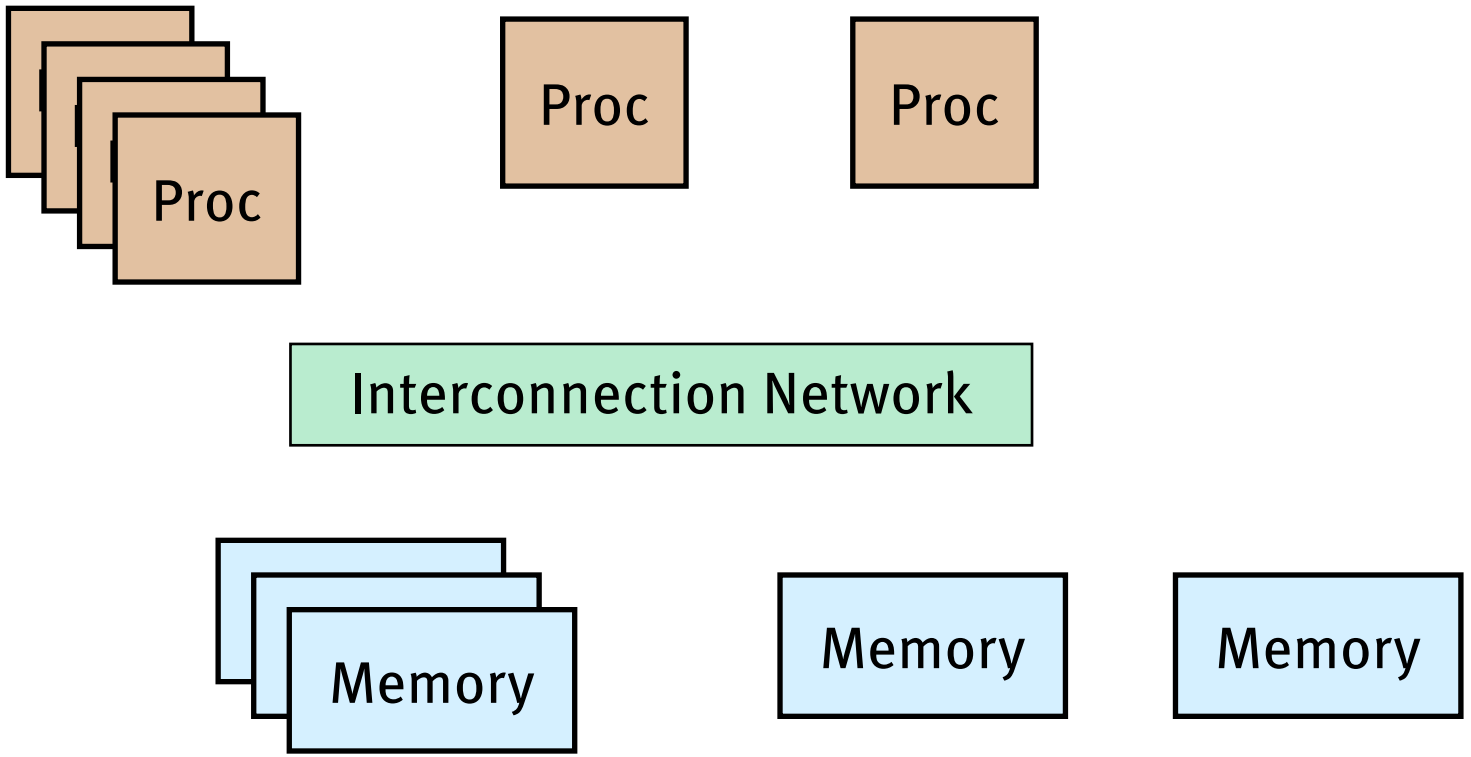
Processor	SUN T1	Opteron	Pentium D	IBM Power 5
Cores	8	2	2	2
Instruction issues / clk / core	1	3	3	4
Peak instr. issues / chip	8	6	6	8
Multithreading	Fine-grained	No	SMT	SMT
L1 I/D in KB per core	16/8	64/64	12K uops/16	64/32
L2 per core/shared	3 MB shared	1 MB / core	1 MB/ core	1.9 MB shared
Clock rate (GHz)	1.2	2.4	3.2	1.9
Transistor count (M)	300	233	230	276
Die size (mm^2)	379	199	206	389
Power (W)	79	110	130	125

Niagara 2 (October 2007)

- Improved performance by increasing # of threads supported per chip from 32 to 64
  - 8 cores * 8 threads per core [now has 2 ALUs/core, 4 threads/ALU]
- Floating-point unit for each core, not for each chip
- Hardware support for encryption standards EAS, 3DES, and elliptical-curve cryptography
- Added 1 8x PCI Express interface directly into the chip in addition to integrated 10 Gb Ethernet XAU interfaces and Gigabit Ethernet ports.
- Integrated memory controllers will shift support from DDR2 to FB-DIMMs and double the maximum amount of system memory.
- Niagara 3 rumor: 45 nm, 16 cores, 16 threads/core

Kevin Krewell, “Sun's Niagara Begins CMT Flood—The Sun UltraSPARC T1 Processor Released”. Microprocessor Report, January 3, 2006

A generic parallel architecture

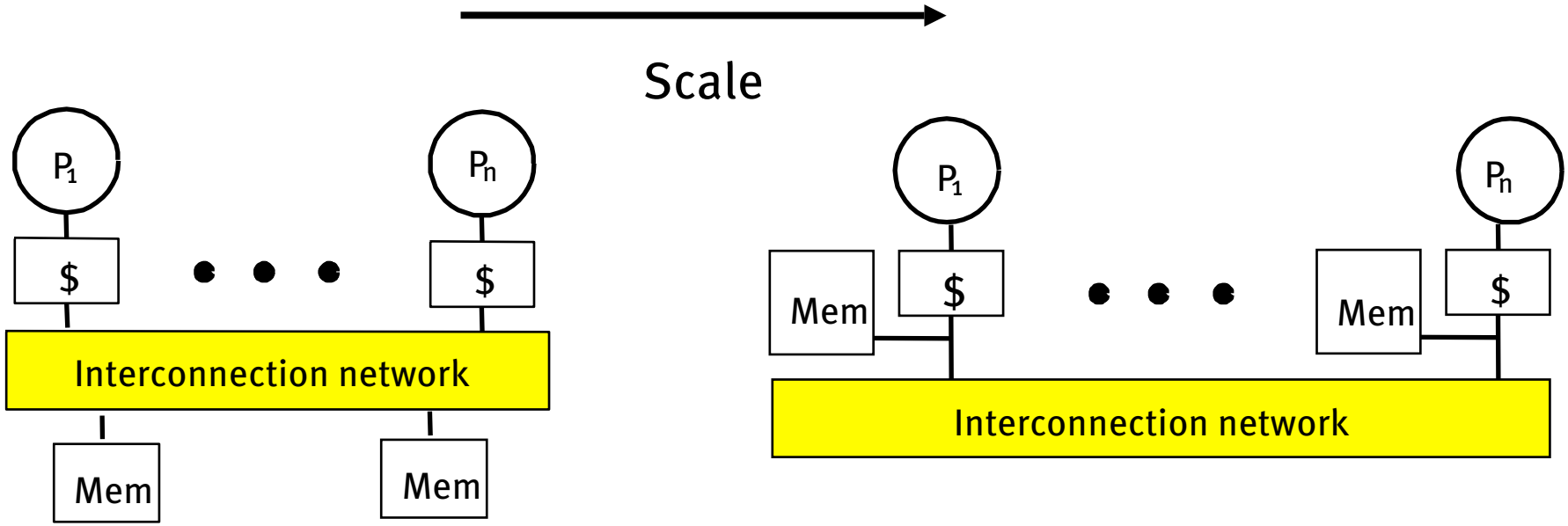


Where is the memory physically located?

Is it connected directly to processors?

What is the connectivity of the network?

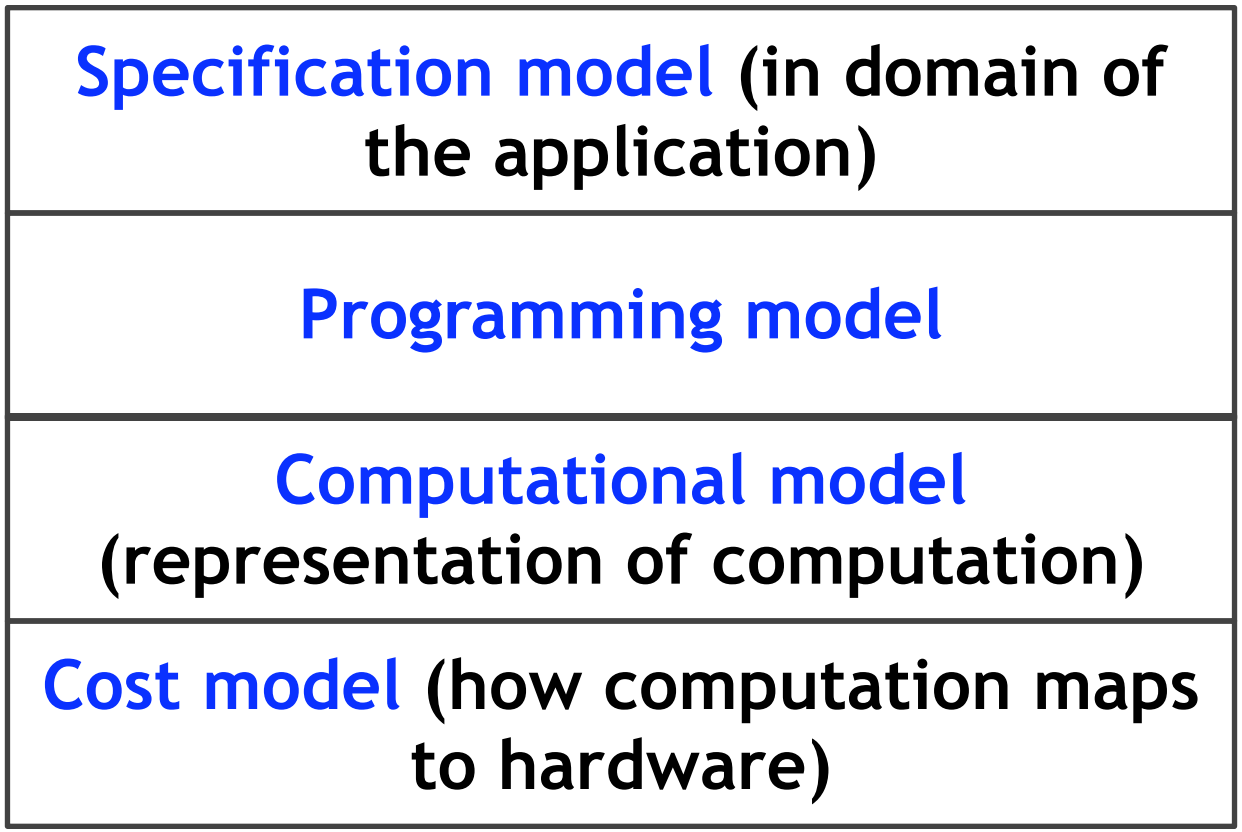
Centralized vs. Distributed Memory



Centralized Memory

Distributed Memory

What is a programming model?



- Is a programming model a language?
  - Programming models allow you to express ideas in particular ways
  - Languages allow you to put those ideas into practice

Writing Parallel Programs

- ***Identify*** **concurrency in task**
  - Do this in your head
- ***Expose*** **the concurrency when writing the task**
  - Choose a programming model and language that allow you to express this concurrency
- ***Exploit*** **the concurrency**
  - Choose a language and hardware that together allow you to take advantage of the concurrency

Parallel Programming Models

- Programming model is made up of the languages and libraries that create an abstract view of the machine
- Control
  - How is parallelism created?
  - What orderings exist between operations?
  - How do different threads of control synchronize?

Parallel Programming Models

- Programming model is made up of the languages and libraries that create an abstract view of the machine
- Data
  - What data is private vs. shared?
  - How is logically shared data accessed or communicated?

Parallel Programming Models

- Programming model is made up of the languages and libraries that create an abstract view of the machine
- Synchronization
  - What operations can be used to coordinate parallelism?
  - What are the atomic (indivisible) operations?
    - Next slides

Segue: Atomicity

- Swaps between threads can happen any time
- Communication from other threads can happen any time
- Other threads can access shared memory any time
- Think about how to grab a shared resource (lock):
  - Wait until lock is free
  - When lock is free, grab it
  - while (*ptrLock == 0) ;
    *ptrLock = 1;

Segue: Atomicity

- Think about how to grab a shared resource (lock):
  - Wait until lock is free
  - When lock is free, grab it
  - while (*ptrLock == 0) ;
    *ptrLock = 1;
- Why do you want to be able to do this?
- What could go wrong with the code above?
- How do we fix it?

Parallel Programming Models

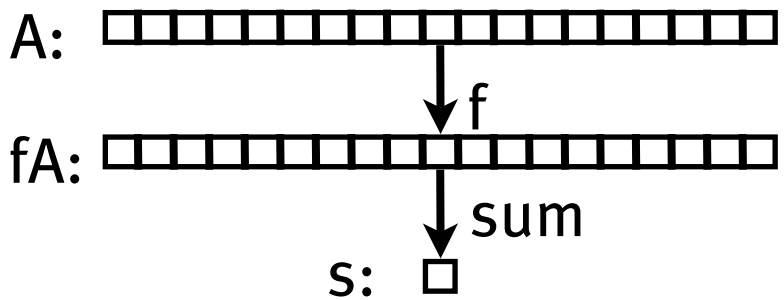
- Programming model is made up of the languages and libraries that create an abstract view of the machine
- Cost
  - How do we account for the cost of each of the above?

Simple Example

- Consider applying a function f to the elements of an array A and then computing its sum:

$$\sum_{i=0}^{n-1} f(A[i])$$

A = array of all data
fA = f(A)
s = sum(fA)

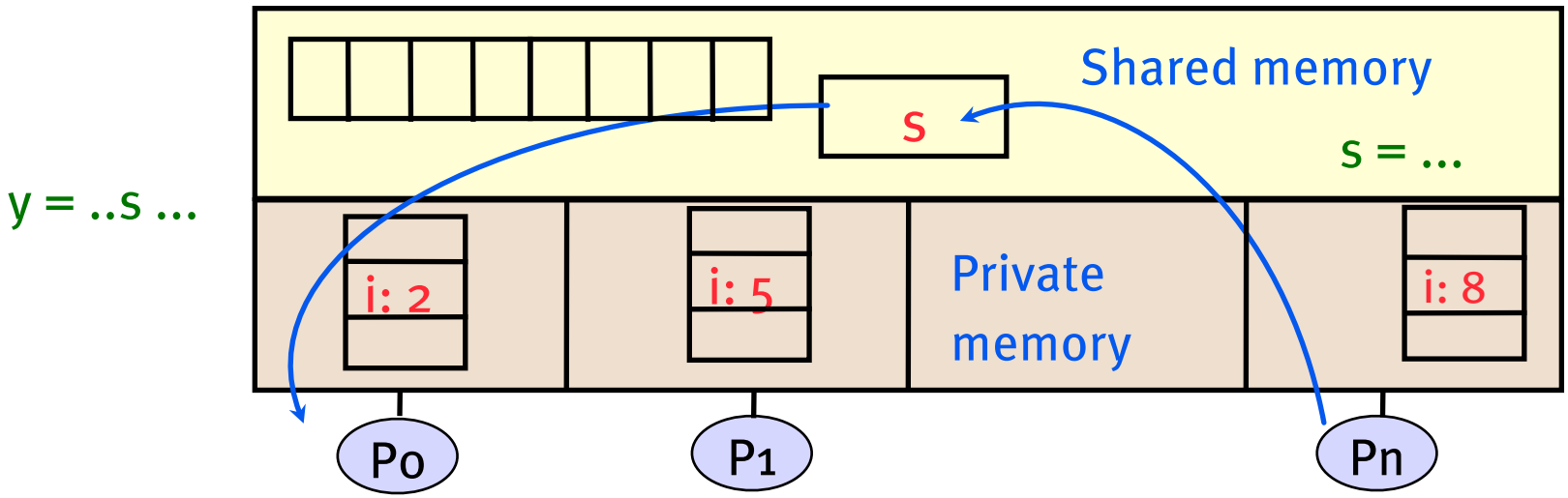


- Questions:
  - Where does A live? All in single memory? Partitioned?
  - How do we divide the work among processors?
  - How do processors cooperate to produce a single result?

Programming Model 1: Shared Memory

- Program is a collection of threads of control.
  - Can be created dynamically, mid-execution, in some languages
- Each thread has a set of private variables, e.g., local stack variables
- Also a set of shared variables, e.g., static variables, shared common blocks, or global heap.
  - Threads communicate implicitly by writing and reading shared variables.
  - Threads coordinate by synchronizing on shared variables

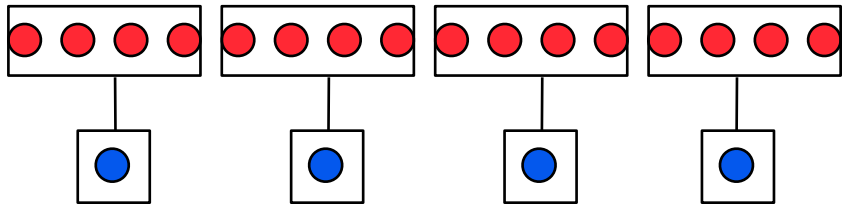
Shared Memory



Simple Example

- Shared memory strategy:
  - small number p << n=size(A) processors
  - attached to single memory

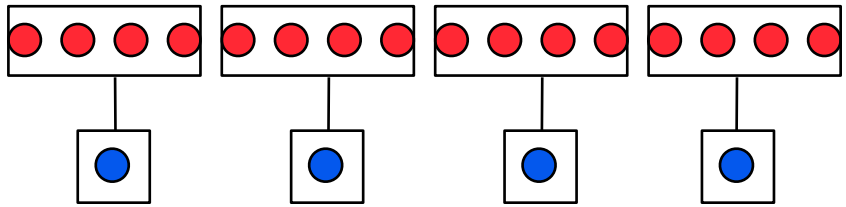
$$\sum_{i=0}^{n-1} f(A[i])$$



- Parallel Decomposition:
  - Each evaluation and each partial sum is a task.
- Assign n/p numbers to each of p procs
  - Each computes independent “private” results and partial sum.
  - Collect the p partial sums and compute a global sum.

Simple Example

$$\sum_{i=0}^{n-1} f(A[i])$$



- Two Classes of Data:
  - Logically Shared
    - The original n numbers, the global sum.
  - Logically Private
    - The individual function evaluations.
    - What about the individual partial sums?

Shared Memory "Code" for Computing a Sum

```
static int s = 0;
```

```
Thread 1

  for i = 0, n/2-1
      s = s + f(A[i])
```

```
Thread 2

  for i = n/2, n-1
      s = s + f(A[i])
```

- Each thread is responsible for half the input elements
- For each element, a thread adds that element to the a shared variable s
- When we're done, s contains the global sum

Shared Memory "Code" for Computing a Sum

```
static int s = 0;
```

```
Thread 1

for i = 0, n/2-1
    s = s + f(A[i])
```

```
Thread 2

for i = n/2, n-1
    s = s + f(A[i])
```

- Problem is a race condition on variable s in the program
- A race condition or data race occurs when:
  - Two processors (or two threads) access the same variable, and at least one does a write.
  - The accesses are concurrent (not synchronized) so they could happen simultaneously

Shared Memory Code for Computing a Sum

A | 3 | 5 | f = square | static int s = 0;

Thread 1		Thread 2	
….		…	
compute f([A[i]) and put in rego	9	compute f([A[i]) and put in rego	25
reg1 = s	0	reg1 = s	0
reg1 = reg1 + rego	9	reg1 = reg1 + rego	25
s = reg1	9	s = reg1	25
…		…	

- Assume A = [3,5], f is the square function, and s=0 initially
- For this program to work, s should be 34 at the end
- but it may be 34, 9, or 25 (how?)
- The atomic operations are reads and writes
- += operation is not atomic
- All computations happen in (private) registers

Improved Code for Computing a Sum

```
Thread 1

  local_s1= 0
  for i = 0, n/2-1
    local_s1 = local_s1 + f(A[i])
  s = s + local_s1
```

```
static int s = 0;
```

```
Thread 2

  local_s2 = 0
  for i = n/2, n-1
    local_s2 = local_s2 + f(A[i])
  s = s + local_s2
```

- Since addition is associative, it's OK to rearrange order
- Most computation is on private variables
- Sharing frequency is also reduced, which might improve speed
- But there is still a race condition on the update of shared s

Improved Code for Computing a Sum

```
Thread 1

  local_s1= 0
  for i = 0, n/2-1
     local_s1 = local_s1 + f(A[i])
  lock(lk);
  s = s + local_s1
  unlock(lk);
```

```
static int s = 0;
static lock lk;
```

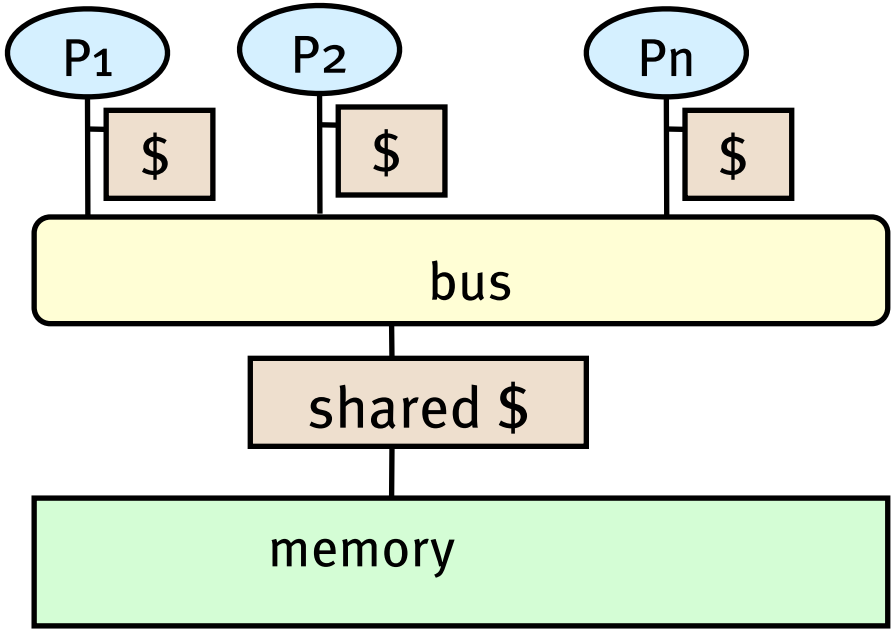
```
Thread 2

  local_s2 = 0
  for i = n/2, n-1
     local_s2= local_s2 + f(A[i])
  lock(lk);
  s = s +local_s2
  unlock(lk);
```

- Since addition is associative, it's OK to rearrange order
- Most computation is on private variables
- Sharing frequency is also reduced, which might improve speed
- But there is still a race condition on the update of shared s
- The race condition can be fixed by adding locks (only one thread can hold a lock at a time; others wait for it)

Machine Model 1a: Shared Memory

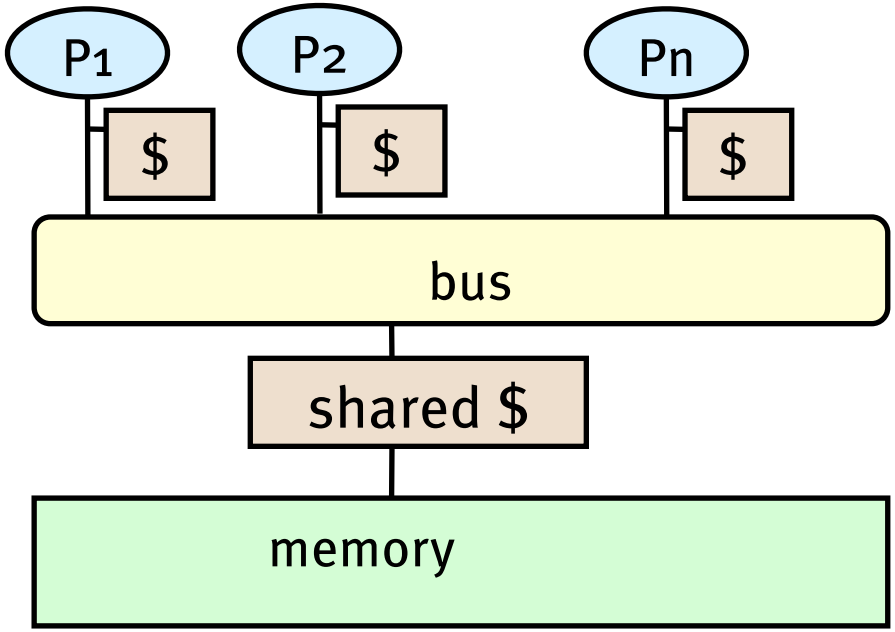
- Processors all connected to a large shared memory
  - Typically called Symmetric Multiprocessors (SMPs)
  - SGI, Sun, HP, Intel, IBM SMPs (nodes of Millennium, SP)
  - Multicore chips, except that caches are often shared in multicores



Note: $ = cache

Machine Model 1a: Shared Memory

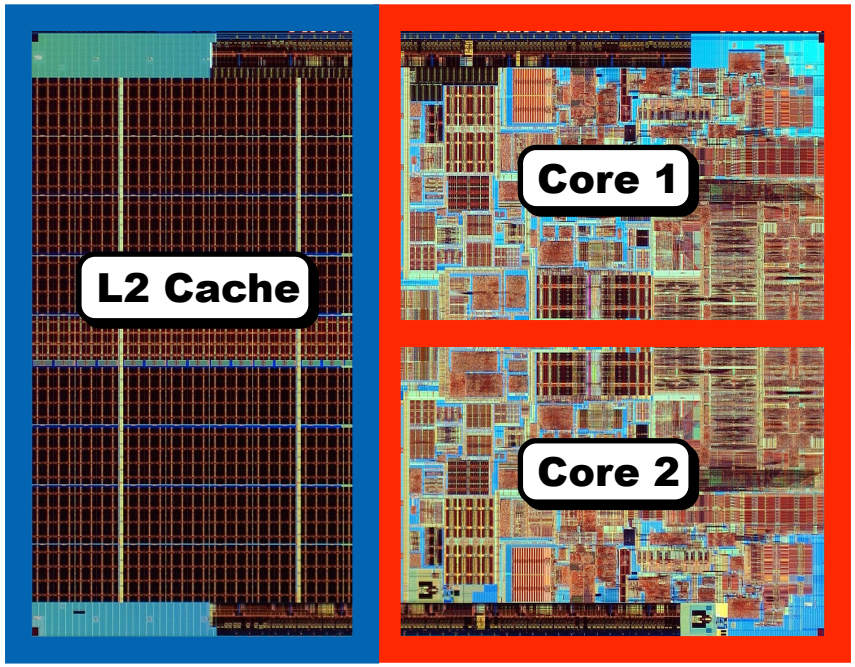
- Difficulty scaling to large numbers of processors
  - <= 32 processors typical
- Advantage: uniform memory access (UMA)
- Cost: much cheaper to access data in cache than main memory.



Note: $ = cache

Intel Core Duo

- Based on Pentium M microarchitecture
  - Pentium D dual-core is two separate processors, no sharing
- Private L1 per core, shared L2, arbitration logic
- Saves power
  - Share data w/o bus
  - Only one access bus, share



Problems Scaling Shared Memory Hardware

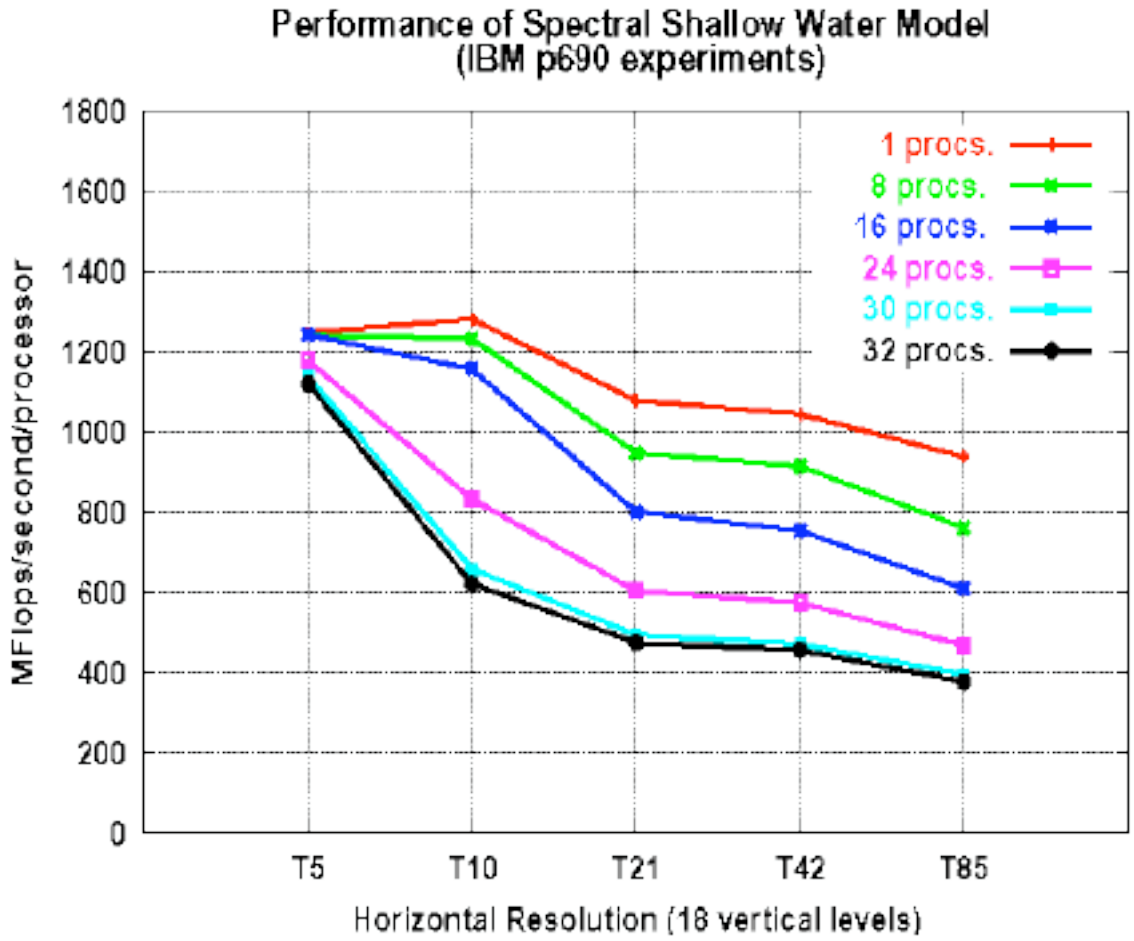
- Why not put more processors on (with larger memory?)
  - The memory bus becomes a bottleneck
    - We're going to look at interconnect performance in a future lecture. For now, just know that "busses are not scalable".
  - Caches need to be kept coherent

Problems Scaling Shared Memory Hardware

- Example from a Parallel Spectral Transform Shallow Water Model (PSTSWM) demonstrates the problem
  - Experimental results (and slide) from Pat Worley at ORNL
  - This is an important kernel in atmospheric models
    - 99% of the floating point operations are multiplies or adds, which generally run well on all processors
    - But it does sweeps through memory with little reuse of operands, so uses bus and shared memory frequently
  - These experiments show serial performance, with one “copy” of the code running independently on varying numbers of procs
    - The best case for shared memory: no sharing
    - But the data doesn’t all fit in the registers/cache

Example: Problem in Scaling Shared Memory

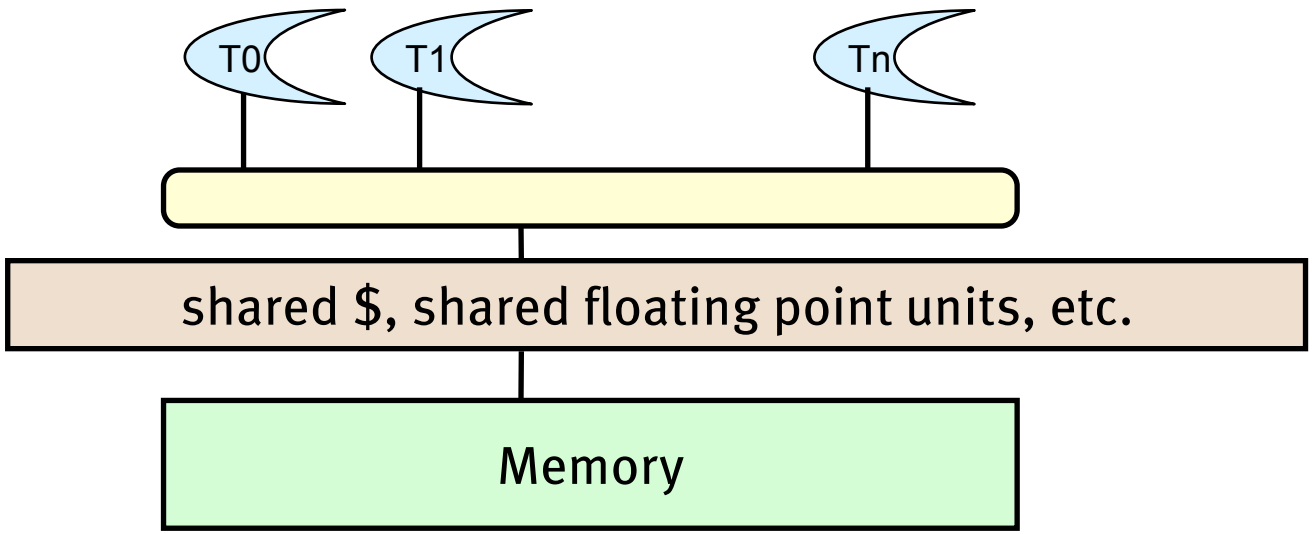
- Performance degradation is a “smooth” function of the number of processes.
- No shared data between them, so there should be perfect parallelism.
- (Code was run for a 18 vertical levels with a range of horizontal sizes.)
- From Pat Worley, ORNL via Kathy Yelick, UCB



Process scaling on IBM p690

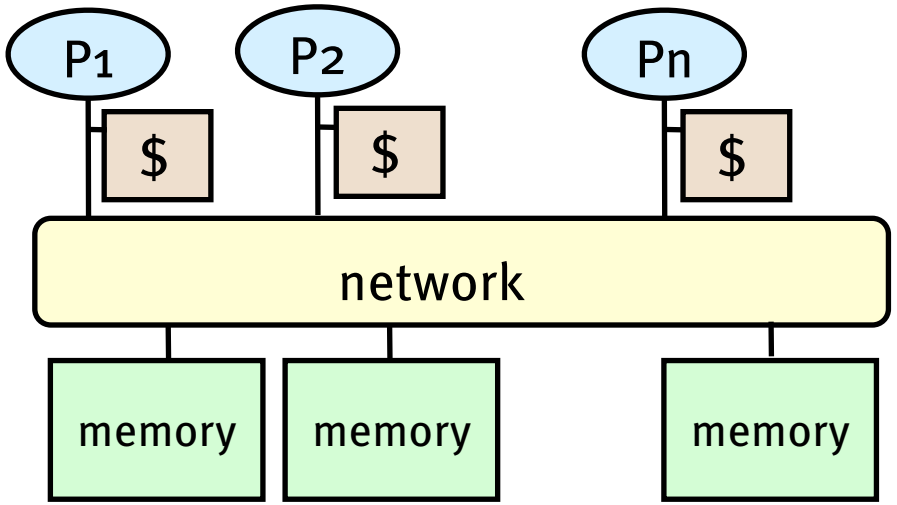
Machine Model 1b: Multithreaded Processor

- Multiple thread “contexts” without full processors
- Memory and some other state is shared
- Sun Niagara processor (for servers)
  - Up to 32 threads all running simultaneously
  - In addition to sharing memory, they share floating point units
  - Why? Switch between threads for long-latency memory operations
- Cray MTA and Eldorado processors (for HPC)



Machine Model 1c: Distributed Shared Memory

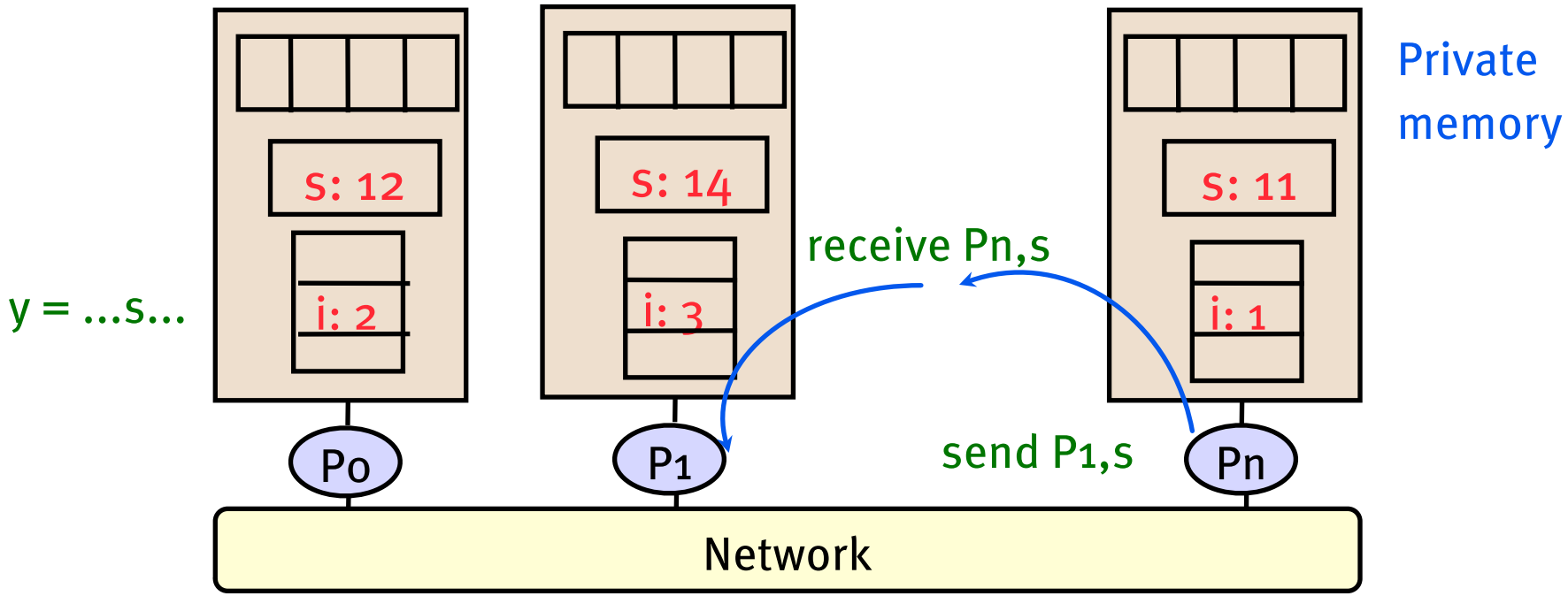
- Memory is logically shared, but physically distributed
  - Any processor can access any address in memory
  - Cache lines (or pages) are passed around machine
- SGI Origin is canonical example (+ research machines)
  - Scales to 512 (SGI Altix (Columbia) at NASA/Ames)
  - Limitation is cache coherency protocols—how to keep cached copies of the same address consistent



Cache lines (pages) must be large to amortize overhead—locality is critical to performance

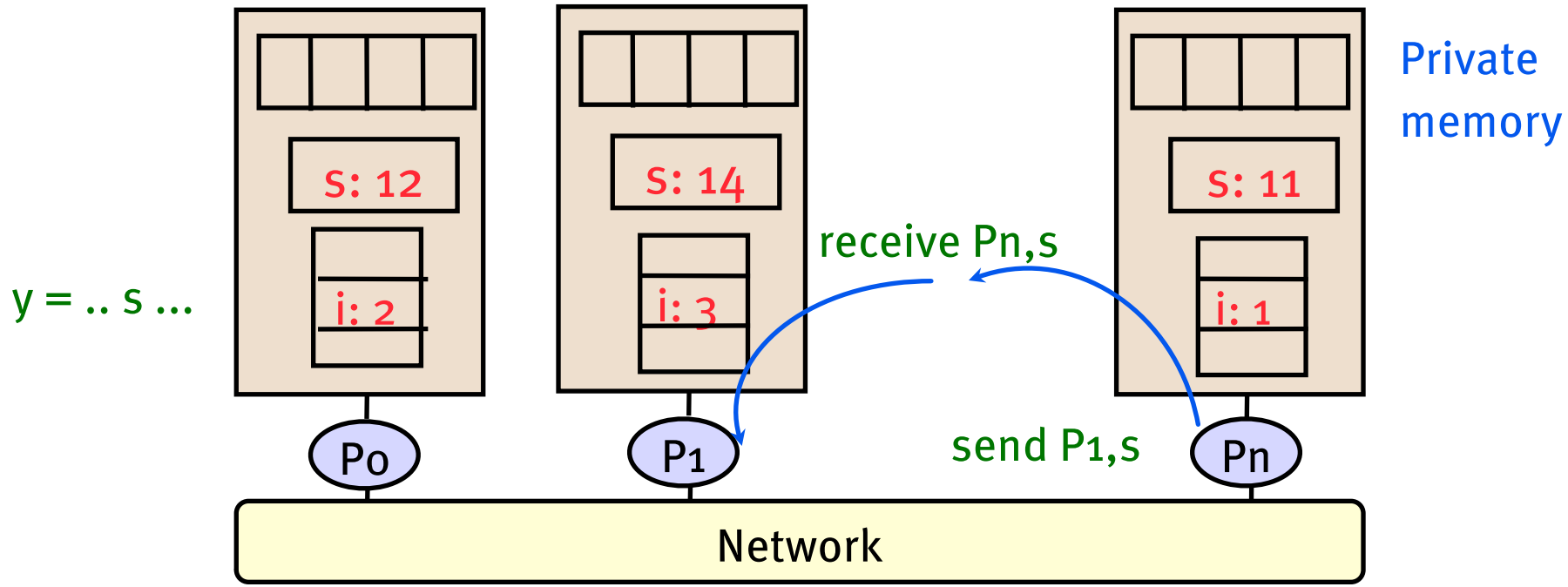
Programming Model 2: Message Passing

- Program consists of a collection of named processes.
  - Usually fixed at program startup time
  - Thread of control plus local address space—NO shared data.
  - Logically shared data is partitioned over local processes.



Programming Model 2: Message Passing

- Processes communicate by explicit send/receive pairs
  - Coordination is implicit in every communication event.
  - MPI (Message Passing Interface) is the most commonly used SW



Computing s = A[1]+A[2] on each processor

- First possible solution—what could go wrong?

Processor 1	Processor 2
xlocal = A[1]	xlocal = A[2]
send xlocal, proc2	send xlocal, proc1
receive xremote, proc2	receive xremote, proc1
s = xlocal + xremote	s = xlocal + xremote

- If send/receive acts like the telephone system? The post office?
- Second possible solution

Processor 1	Processor 2
xlocal = A[1]	xlocal = A[2]
send xlocal, proc2	receive xremote, proc1
receive xremote, proc2	send xlocal, proc1
s = xlocal + xremote	s = xlocal + xremote

- What if there are more than 2 processors?

MPI—the de facto standard

- MPI has become the de facto standard for parallel computing using message passing
- Pros and Cons of standards
  - MPI created finally a standard for applications development in the HPC community ⇢ portability
  - The MPI standard is a least common denominator building on mid-80s technology, so may discourage innovation
- Programming Model reflects hardware!

MPI Hello World

```
int main(int argc, char *argv[])
{
  char idstr[32];
  char buff[BUFSIZE];
  int numprocs;
  int myid;
  int i;
  MPI_Status stat;

  MPI_Init(&argc,&argv); /* all MPI programs start with MPI_Init; all 'N' processes
exist thereafter */
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs); /* find out how big the SPMD world is */
  MPI_Comm_rank(MPI_COMM_WORLD,&myid); /* and this processes' rank is */

  /* At this point, all the programs are running equivalently, the rank is used to
     distinguish the roles of the programs in the SPMD model, with rank 0 often used
     specially... */
```

MPI Hello World

```
if(myid == 0)
{
  printf("%d: We have %d processors\n", myid, numprocs);
  for(i=1;i<numprocs;i++)
  {
    sprintf(buff, "Hello %d! ", i);
    MPI_Send(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD);
  }
  for(i=1;i<numprocs;i++)
  {
    MPI_Recv(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD, &stat);
    printf("%d: %s\n", myid, buff);
  }
}
```

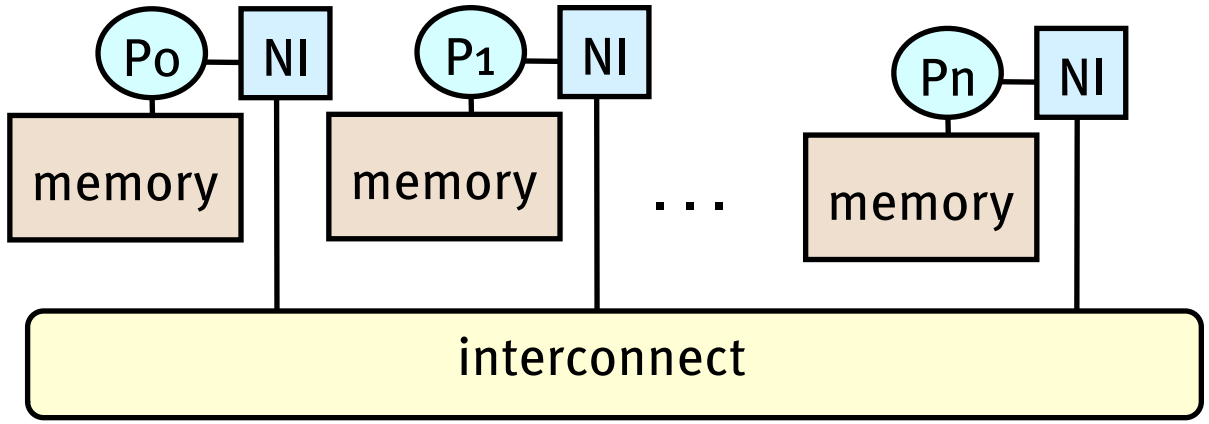
MPI Hello World

```
  else
  {
    /* receive from rank 0: */
    MPI_Recv(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD, &stat);
    sprintf(idstr, "Processor %d ", myid);
    strcat(buff, idstr);
    strcat(buff, "reporting for duty\n");
    /* send to rank 0: */
    MPI_Send(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD);
  }

  MPI_Finalize(); /* MPI Programs end with MPI Finalize; this is a weak
synchronization point */
  return 0;
}
```

Machine Model 2a: Distributed Memory

- Cray T3E, IBM SP2
- PC Clusters (Berkeley NOW, Beowulf)
- IBM SP-3, Millennium, CITRIS are distributed memory machines, but the nodes are SMPs.
- Each processor has its own memory and cache but cannot directly access another processor's memory.
- Each "node" has a Network Interface (NI) for all communication and synchronization.



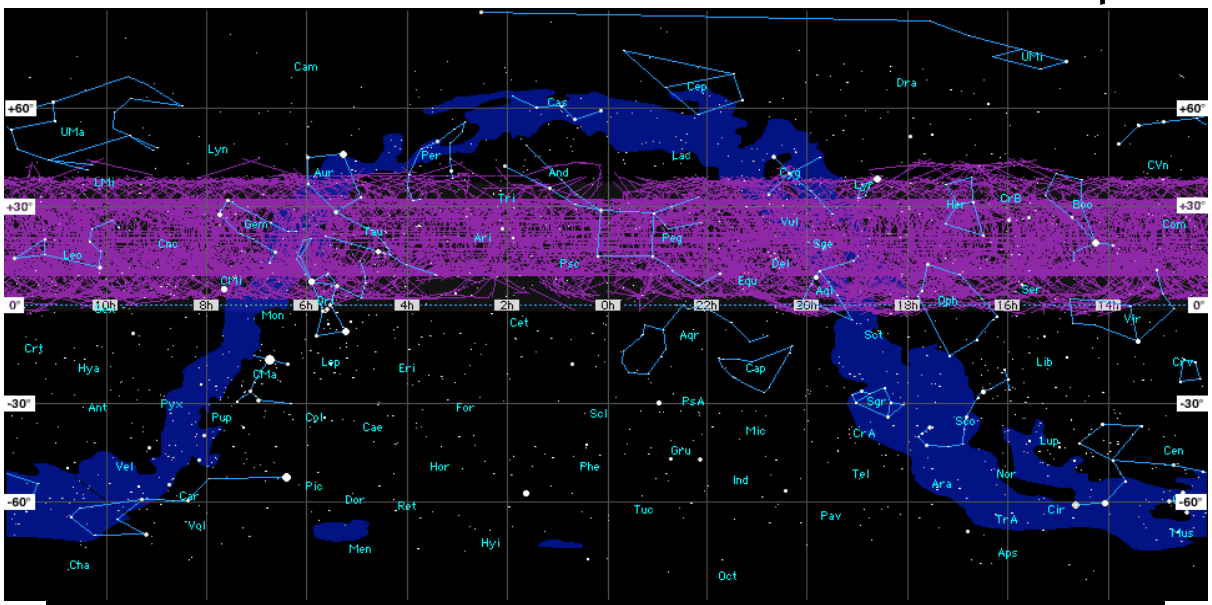
Tflop/s Clusters

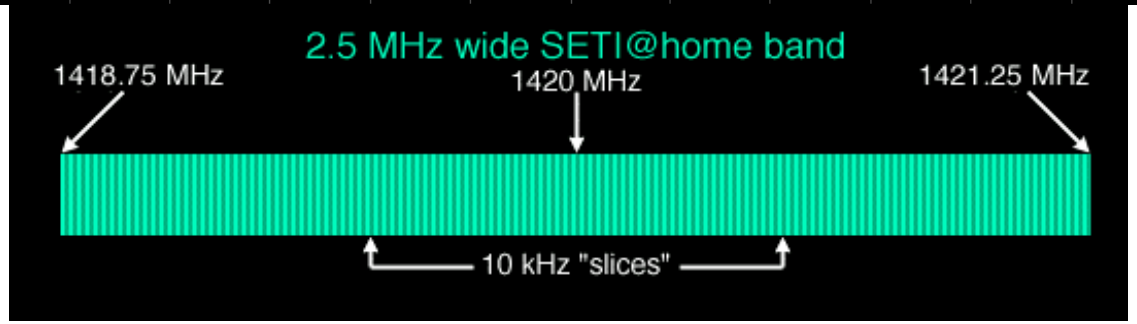
- The following are examples of clusters configured out of separate networks and processor components
  - 72% of Top 500 (Nov 2005), 2 of top 10
- Dell cluster at Sandia (Thunderbird) is #4 on Top 500
  - 8000 Intel Xeons @ 3.6GHz
  - 64TFlops peak, 38 TFlops Linpack
  - Infiniband connection network
- Walt Disney Feature Animation (The Hive) is #96
  - 1110 Intel Xeons @ 3 GHz
  - Gigabit Ethernet
- Saudi Oil Company is #107
- Credit Suisse/First Boston is #108

Machine Model 2b: Internet/Grid Computing

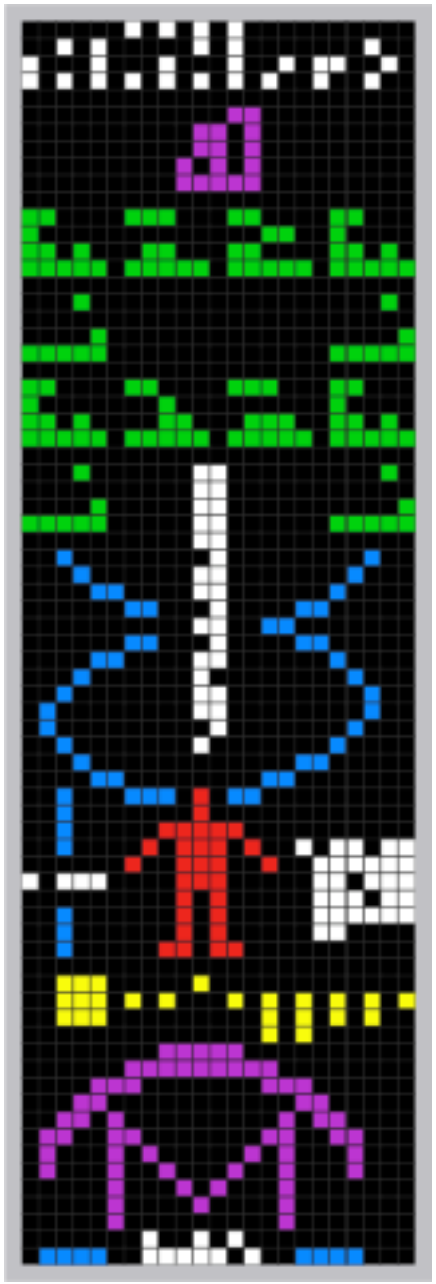
- SETI@Home: Running on 500,000 PCs
  - ~1000 CPU Years per Day, 485,821 CPU Years so far
- Sophisticated Data & Signal Processing Analysis
- Distributes Datasets from Arecibo Radio Telescope

Next Step—
Allen Telescope
Array





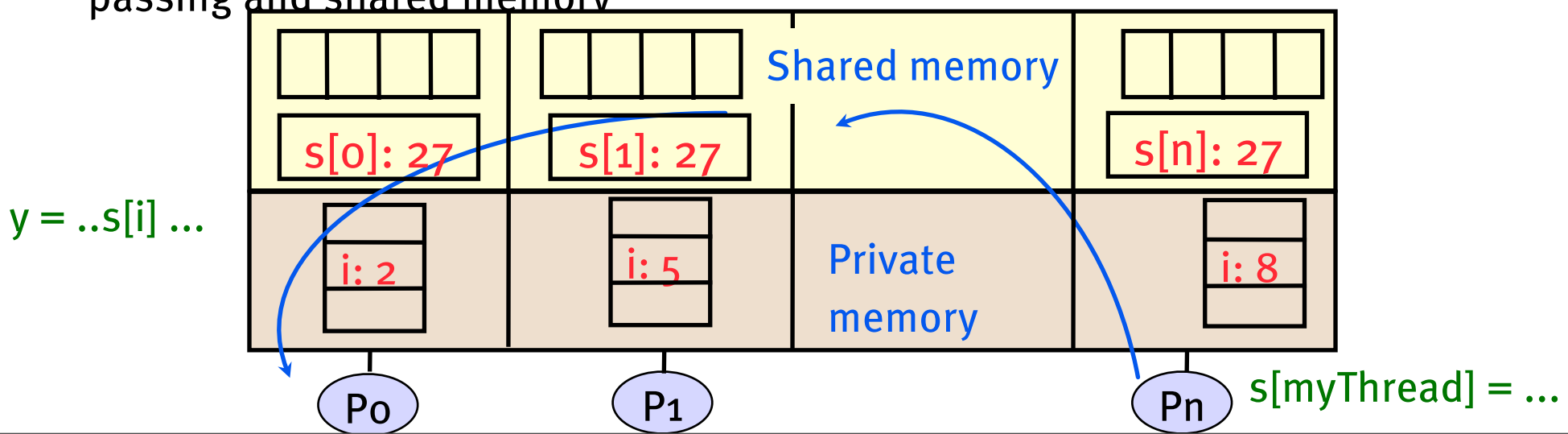
Arecibo message



http://en.wikipedia.org/wiki/Image:Arecibo_message.svg

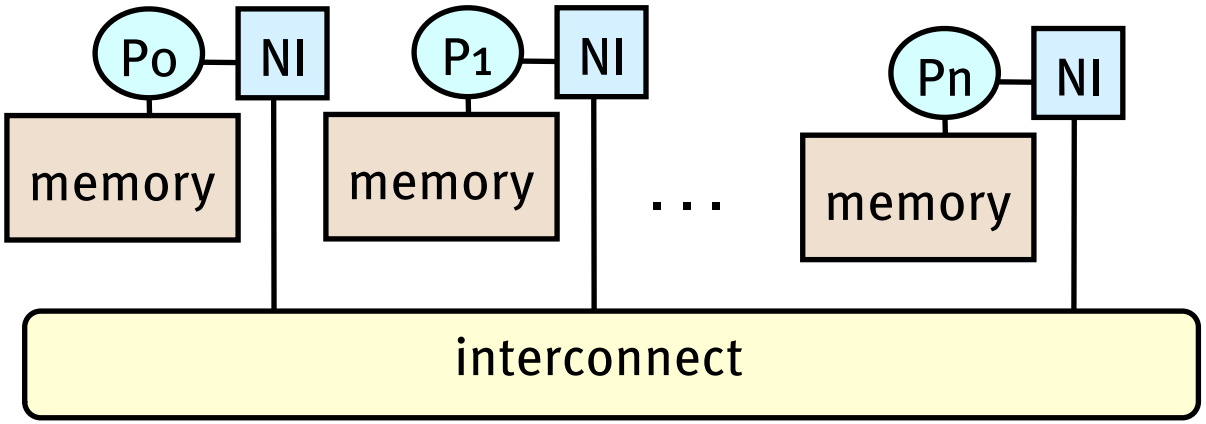
Programming Model 2c: Global Address Space

- Program consists of a collection of named threads.
  - Usually fixed at program startup time
  - Local and shared data, as in shared memory model
  - But, shared data is partitioned over local processes
  - Cost model says remote data is expensive
- Examples: UPC, Titanium, Co-Array Fortran
- Global Address Space programming is an intermediate point between message passing and shared memory



Machine Model 2c: Global Address Space

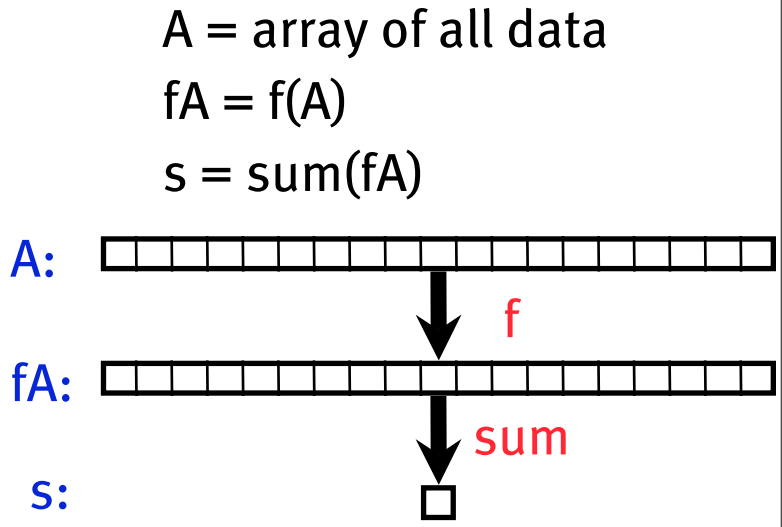
- Cray T3D, T3E, X1, and HP Alphaserver cluster
- Clusters built with Quadrics, Myrinet, or Infiniband
- The network interface supports RDMA (Remote Direct Memory Access)
  - NI can directly access memory without interrupting the CPU
  - One processor can read/write memory with one-sided operations (put/get)
  - Not just a load/store as on a shared memory machine
    - Continue computing while waiting for memory op to finish
  - Remote data is typically not cached locally



Global address space may be supported in varying degrees

Programming Model 3: Data Parallel

- Single thread of control consisting of parallel operations.
- Parallel operations applied to all (or a defined subset) of a data structure, usually an array
  - Communication is implicit in parallel operators
  - Elegant and easy to understand and reason about
  - Coordination is implicit—statements executed synchronously
  - Similar to Matlab language for array operations
- Drawbacks:
  - Not all problems fit this model
  - Difficult to map onto coarse-grained machines



Programming Model 4: Hybrids

- These programming models can be mixed
  - Message passing (MPI) at the top level with shared memory within a node is common
  - New DARPA HPCS languages mix data parallel and threads in a global address space
  - Global address space models can (often) call message passing libraries or vice versa
  - Global address space models can be used in a hybrid mode
    - Shared memory when it exists in hardware
    - Communication (done by the runtime system) otherwise

Machine Model 4: Clusters of SMPs

- SMPs are the fastest commodity machine, so use them as a building block for a larger machine with a network
- Common names:
  - CLUMP = Cluster of SMPs
  - Hierarchical machines, constellations
- Many modern machines look like this:
  - Millennium, IBM SPs, ASCI machines
- What is an appropriate programming model for #4?
  - Treat machine as “flat”, always use message passing, even within SMP (simple, but ignores an important part of memory hierarchy).
  - Shared memory within one SMP, but message passing outside of an SMP.

Challenges of Parallel Processing

- Application parallelism ⇒ primarily via new algorithms that have better parallel performance
- Long remote latency impact ⇒ both by architect and by the programmer
- For example, reduce frequency of remote accesses either by
  - Caching shared data (HW)
  - Restructuring the data layout to make more accesses local (SW)
- Today's lecture on HW to help latency via caches

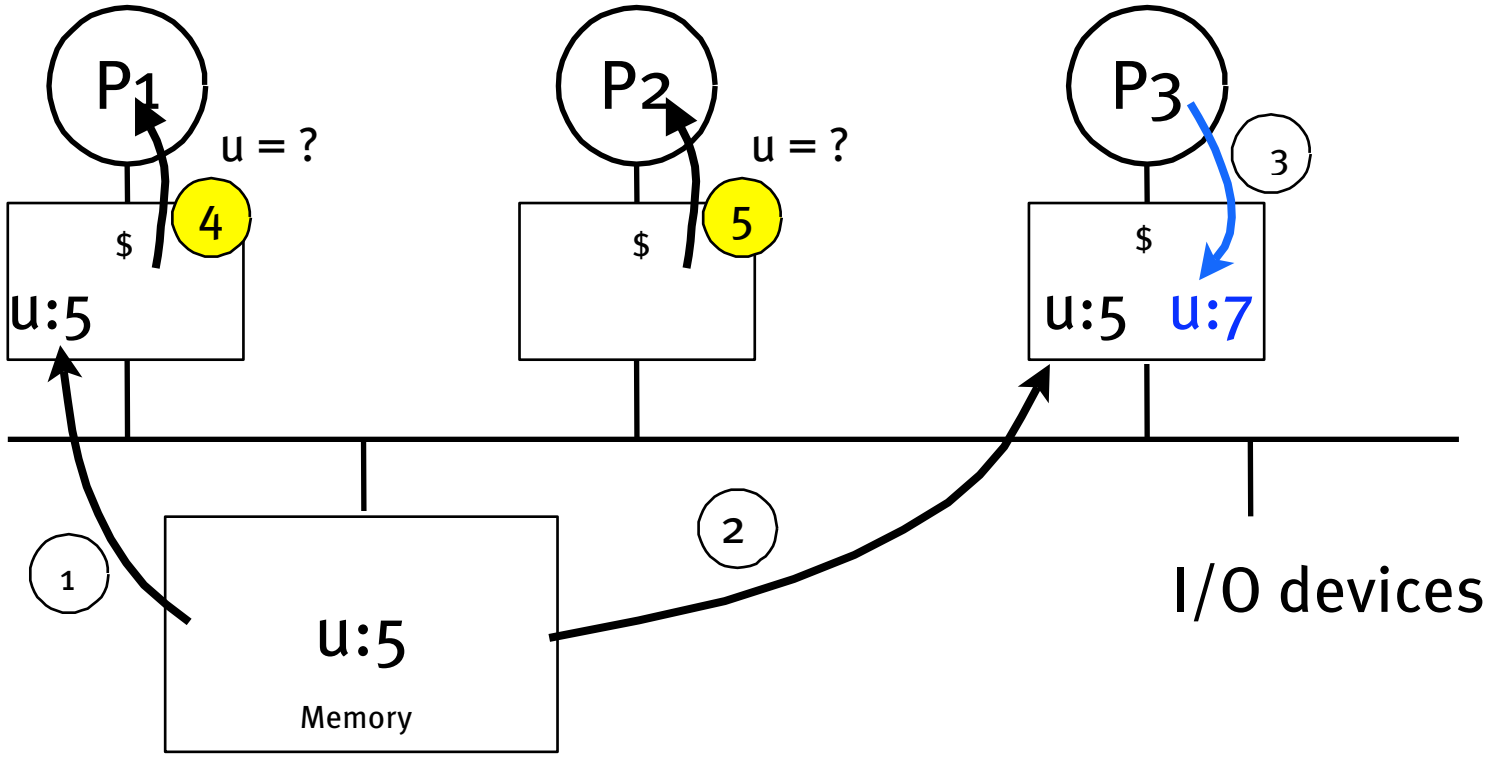
Fundamental Problem

- Many processors working on a task
- Those processors share data, need to communicate, etc.
- For efficiency, we use caches
- This results in multiple copies of the data
- Are we working with the right copy?

Symmetric Shared-Memory Architectures

- From multiple boards on a shared bus to multiple processors inside a single chip
- Caches:
  - Private data are used by a single processor
  - Shared data are used by multiple processors
- Caching shared data:
  - reduces latency to shared data, memory bandwidth for shared data, and interconnect bandwidth
  - introduces a cache coherence problem

Example Cache Coherence Problem



- Processors see different values for u after event 3
- With write back caches, value written back to memory depends on happenstance of which cache flushes or writes back value when
  - Processes accessing main memory may see very stale value
- Unacceptable for programming, and it's frequent!

Intuitive Memory Model

- Reading an address should return the last value written to that address
  - Easy in uniprocessors, except for I/O
- Too vague and simplistic; 2 issues
  - Coherence defines values returned by a read
  - Consistency determines when a written value will be returned by a read
  - Coherence defines behavior for same processor, Consistency defines behavior for other processors

Defining Coherent Memory System

- Preserve Program Order: A read by processor P to location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P
  - P writes D to X
  - Nobody else writes to X
  - P reads X -› always gives D

Defining Coherent Memory System

- Coherent view of memory: Read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses
  - P1 writes D to X
  - Nobody else writes to X
  - … wait a while …
  - P2 reads X, should get D

Defining Coherent Memory System

- Write serialization: 2 writes to same location by any 2 processors are seen in the same order by all processors
  - If not, a processor could keep value 1 since saw as last write
  - For example, if the values 1 and then 2 are written to a location, processors can never read the value of the location as 2 and then later read it as 1

Write Consistency

- For now assume
  - A write does not complete (and allow the next write to occur) until all processors have seen the effect of that write
  - The processor does not change the order of any write with respect to any other memory access
- ⇒ if a processor writes location A followed by location B, any processor that sees the new value of B must also see the new value of A
- These restrictions allow the processor to reorder reads, but forces the processor to finish writes in program order

Basic Schemes for Coherence with Performance

- Program on multiple processors will normally have copies of the same data in several caches
  - Unlike I/O, where it's rare
- SMPs use a HW protocol to maintain coherent caches
  - Migration and Replication key to performance of shared data

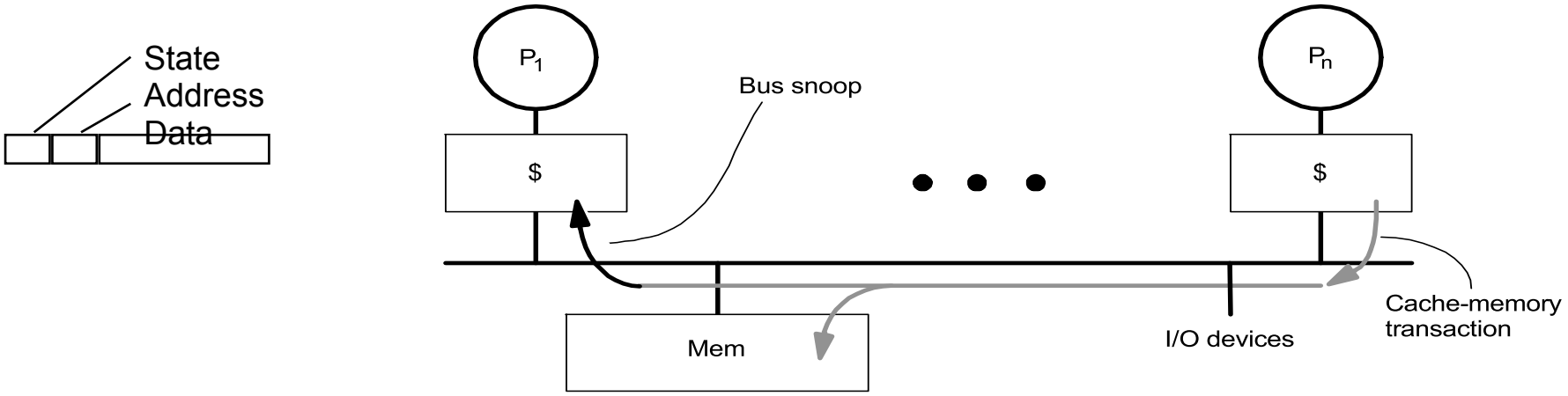
Basic Schemes for Coherence with Performance

- Migration—data can be moved to a local cache and used there in a transparent fashion
  - Reduces both latency to access shared data that is allocated remotely and bandwidth demand on the shared memory
- Replication—for reading shared data simultaneously, since caches make a copy of data in local cache
  - Reduces both latency of access and contention for read shared data

2 Classes of Cache Coherence Protocols

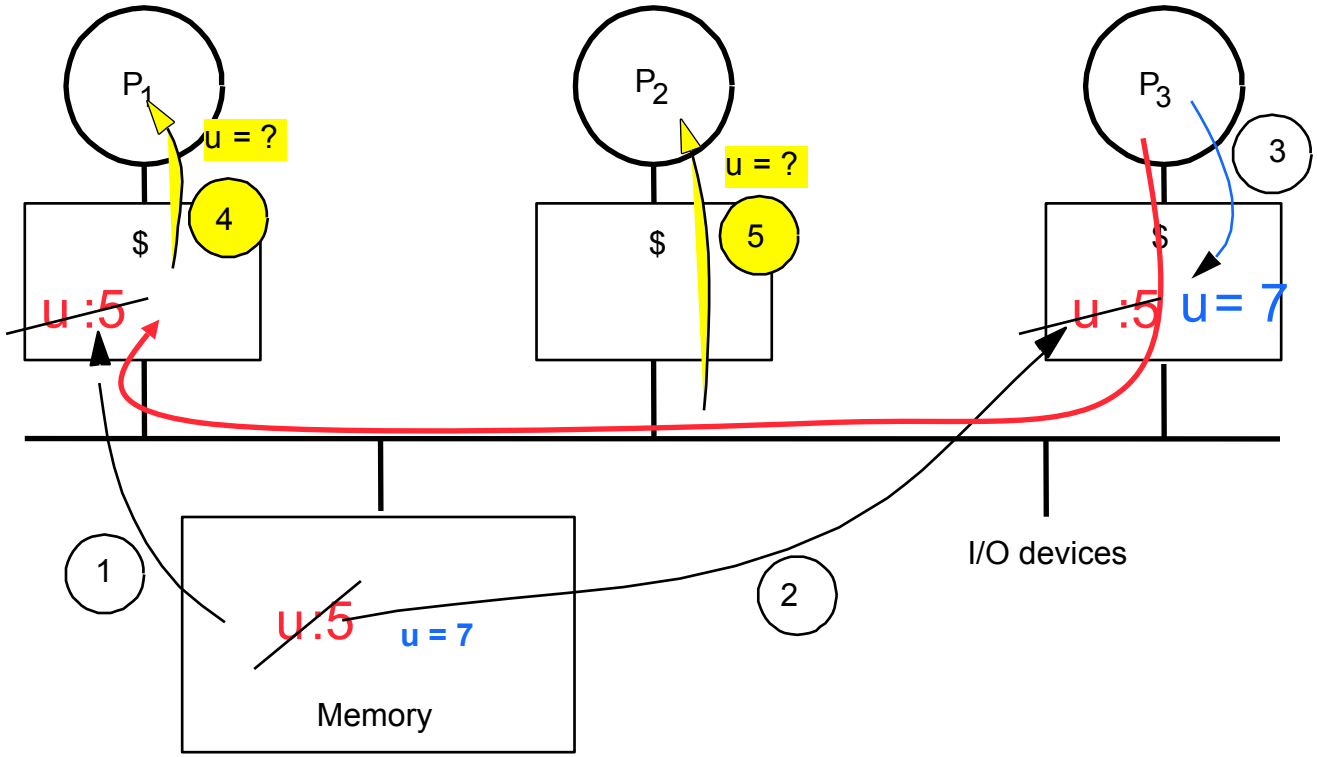
- Directory based — Sharing status of a block of physical memory is kept in just one location, the directory
- Snooping — Every cache with a copy of data also has a copy of sharing status of block, but no centralized state is kept
  - All caches are accessible via some broadcast medium (a bus or switch)
  - All cache controllers monitor or snoop on the medium to determine whether or not they have a copy of a block that is requested on a bus or switch access

Snoopy Cache-Coherence Protocols



- Cache Controller "snoops" all transactions on the shared medium (bus or switch)
  - Does this transaction concern data that I have?
  - If so, take action to ensure coherence
    - invalidate (my val), update (my val), or supply (my value) (when, when, and when?)
  - depends on state of the block and the protocol
- Either get exclusive access before write via write invalidate or update all copies on write

Example: Write-thru Invalidate



- Must invalidate before step 3
- Write update uses more broadcast medium BW
  ⇒ all recent MPUs use write invalidate

Architectural Building Blocks

- Cache block state transition diagram
  - FSM specifying how disposition of block changes
    - invalid, valid, exclusive
- Broadcast Medium Transactions (e.g., bus)
  - Fundamental system design abstraction
  - Logically single set of wires connect several devices
  - Protocol: arbitration, command/address, data
  - Every device observes every transaction

Architectural Building Blocks

- Broadcast medium enforces serialization of read or write accesses
  ⇒ Write serialization
  - 1st processor to get medium invalidates others copies
  - Implies cannot complete write until it obtains bus
  - All coherence schemes require serializing accesses to same cache block
- Also need to find up-to-date copy of cache block (on read for instance)

Locate up-to-date copy of data

- Write-through: get up-to-date copy from memory
  - Write through simpler if enough memory BW
- Write-back harder
  - Most recent copy can be in a cache
- Can use same snooping mechanism
  - Snoop every address placed on the bus
  - If a processor has dirty copy of requested cache block, it provides it in response to a read request and aborts the memory access
  - Complexity from retrieving cache block from cache, which can take longer than retrieving it from memory
- Write-back needs lower memory bandwidth
  ⇒ Support larger numbers of faster processors

  ⇒ Most multiprocessors use write-back

Performance of Symmetric Shared-Memory Multiprocessors

- Cache performance is combination of
  - Uniprocessor cache miss traffic
  - Traffic caused by communication
    - Results in invalidations and subsequent cache misses
- 4th C: coherence miss
  - Joins Compulsory, Capacity, Conflict

Coherency Misses

- True sharing misses arise from the communication of data through the cache coherence mechanism
  - Invalidates due to 1st write to shared block
  - Reads by another CPU of modified block in different cache
  - Miss would still occur if block size were 1 word
- False sharing misses when a block is invalidated because some word in the block, other than the one being read, is written into
  - Invalidation does not cause a new value to be communicated, but only causes an extra cache miss
  - Block is shared, but no word in block is actually shared
    ⇒ miss would not occur if block size were 1 word

Example: True v. False Sharing v. Hit?

- Assume x1 and x2 in same cache block.
  P1 and P2 both read x1 and x2 before.

Time	P1	P2	True, False, Hit? Why?
1	Write x1		True miss; invalidate x1 in P2
2		Read x2	False miss; x1 irrelevant to P2
3	Write x1		False miss; x1 irrelevant to P2
4		Write x2	False miss; x1 irrelevant to P2
5	Read x2		True miss; invalidate x2 in P1

Review

- Caches contain all information on state of cached memory blocks
- Snooping cache over shared medium for smaller MP by invalidating other cached copies on write
- Sharing cached data $\Rightarrow$ Coherence (values returned by a read), Consistency (when a written value will be returned by a read)

A Cache Coherent System Must:

- Provide set of states, state transition diagram, and actions
- Manage coherence protocol
  - (o) Determine when to invoke coherence protocol
  - (a) Find info about state of block in other caches to determine action
    - whether need to communicate with other cached copies
  - (b) Locate the other copies
  - (c) Communicate with those copies (invalidate/update)
- (o) is done the same way on all systems
  - state of the line is maintained in the cache
  - protocol is invoked if an “access fault” occurs on the line
- Different approaches distinguished by (a) to (c)

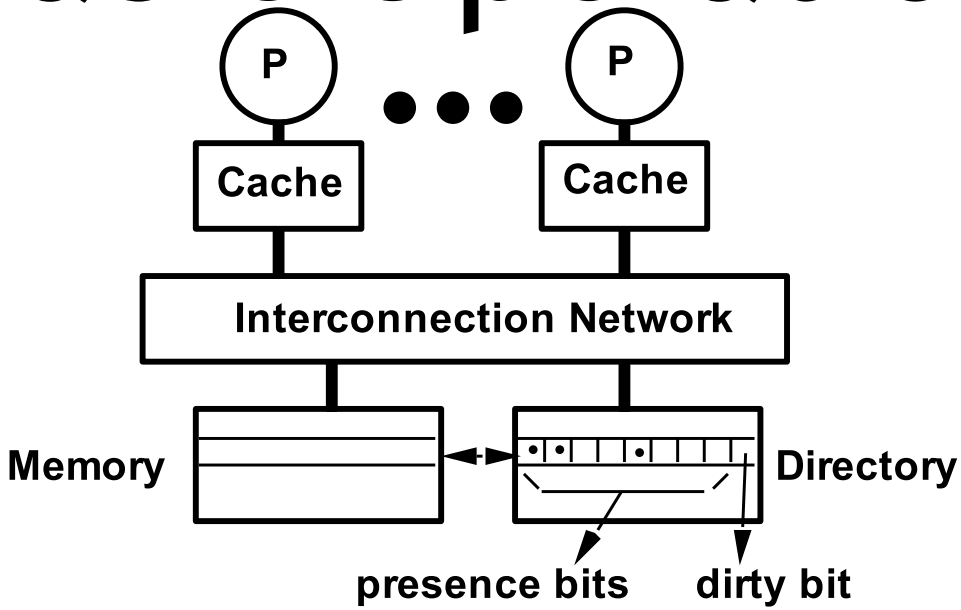
Bus-based Coherence

- All of (a), (b), (c) done through broadcast on bus
  - faulting processor sends out a “search”
  - others respond to the search probe and take necessary action
- Could do it in scalable network too
  - broadcast to all processors, and let them respond
- Conceptually simple, but broadcast doesn’t scale with p
  - on bus, bus bandwidth doesn’t scale
  - on scalable network, every fault leads to at least p network transactions
- Scalable coherence:
  - can have same cache states and state transition diagram
  - different mechanisms to manage protocol

Scalable Approach: Directories

- Every memory block has associated directory information
  - keeps track of copies of cached blocks and their states
  - on a miss, find directory entry, look it up, and communicate only with the nodes that have copies if necessary
  - in scalable networks, communication with directory and copies is through network transactions
- Many alternatives for organizing directory information

Basic Operation of Directory



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

- Read from main memory by processor i:
  - If dirty-bit OFF then { read from main memory; turn p[i] ON; }
  - If dirty-bit ON then { recall line from dirty proc (cache state to shared); update memory; turn dirty-bit OFF; turn p[i] ON; supply recalled data to i;}
- Write to main memory by processor i:
  - If dirty-bit OFF then { supply data to i; send invalidations to all caches that have the block; turn dirty-bit ON; turn p[i] ON; ... }
  - • ...

Another MP Issue: Memory Consistency Models

- What is consistency? When must a processor see the new value? e.g., seems that

-

P1:	A = 0;	P2:	B = 0;
	.....		.....
	A = 1;		B = 1;
L1:	if (B == 0) ...	L2:	if (A == 0) ...

- Impossible for both if statements L1 & L2 to be true?
  - What if write invalidate is delayed & processor continues?

Another MP Issue: Memory Consistency Models

- Memory consistency models:
  what are the rules for such cases?
- Sequential consistency: result of any execution is the same as if the accesses of each processor were kept in order and the accesses among different processors were interleaved ⇒ assignments before ifs above
  - SC: delay all memory accesses until all invalidates done

Memory Consistency Model

- Schemes faster execution to sequential consistency
- Not an issue for most programs; they are synchronized
  - A program is synchronized if all access to shared data are ordered by synchronization operations

    write (x)
    ...
    release (s) {unlock}
    ...
    acquire (s) {lock}
    ...
    read(x)

- Only those programs willing to be nondeterministic are not synchronized: “data race”: outcome f(proc. speed)
- Several Relaxed Models for Memory Consistency since most programs are synchronized; characterized by their attitude towards: RAR, WAR, RAW, WAW to different addresses

Relaxed Consistency Models: The Basics

- Key idea: allow reads and writes to complete out of order, but to use synchronization operations to enforce ordering, so that a synchronized program behaves as if the processor were sequentially consistent
  - By relaxing orderings, may obtain performance advantages
  - Also specifies range of legal compiler optimizations on shared data
  - Unless synchronization points are clearly defined and programs are synchronized, compiler could not interchange read and write of 2 shared data items because might affect the semantics of the program

Relaxed Consistency Models: The Basics

- 3 major sets of relaxed orderings:
- W⇢R ordering (all writes completed before next read)
  - Because retains ordering among writes, many programs that operate under sequential consistency operate under this model, without additional synchronization. Called processor consistency
- W ⇢ W ordering (all writes completed before next write)
- R ⇢ W and R ⇢ R orderings, a variety of models depending on ordering restrictions and how synchronization operations enforce ordering
- Many complexities in relaxed consistency models; defining precisely what it means for a write to complete; deciding when processors can see values that it has written

Mark Hill observation

- Instead, use speculation to hide latency from strict consistency model
  - If processor receives invalidation for memory reference before it is committed, processor uses speculation recovery to back out computation and restart with invalidated memory reference
- 1. Aggressive implementation of sequential consistency or processor consistency gains most of advantage of more relaxed models
- 2. Implementation adds little to implementation cost of speculative processor
- 3. Allows the programmer to reason using the simpler programming models