Shared Memory Programming:

Threads and OpenMP

Lecture 6

James Demmel

www.cs.berkeley.edu/~demmel/cs267_Spr12/

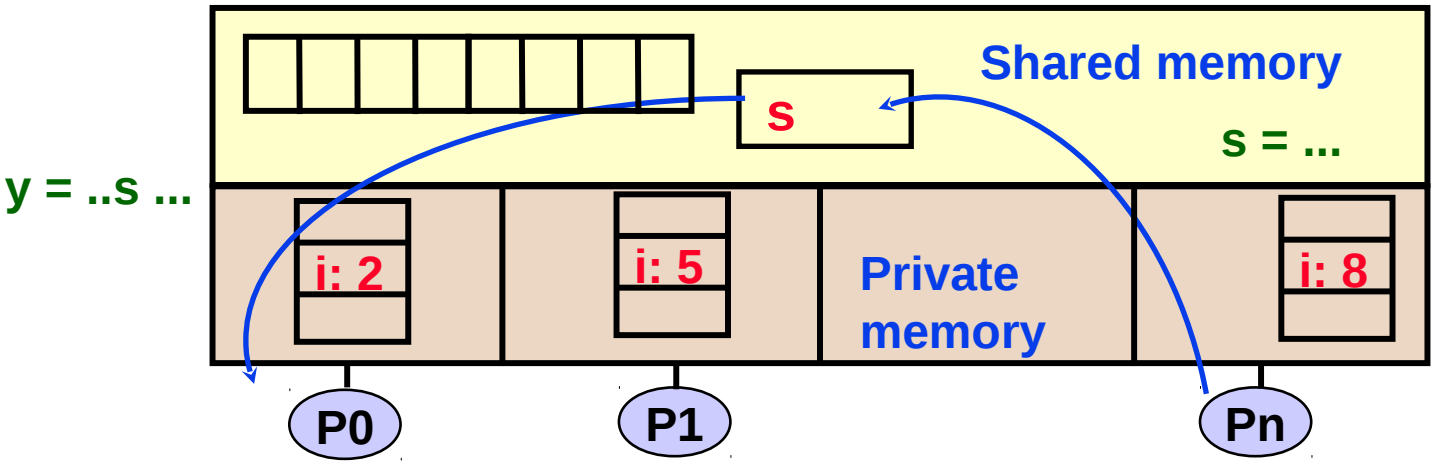
Outline

- Parallel Programming with Threads
- Parallel Programming with OpenMP
  - See http://www.nersc.gov/nusers/help/tutorials/openmp/
  - Slides on OpenMP derived from: U.Wisconsin tutorial, which in turn were from LLNL, NERSC, U. Minn, and OpenMP.org
    - See tutorial by Tim Mattson and Larry Meadows presented at SC08, at OpenMP.org; includes programming exercises
- (There are other Shared Memory Models: CILK, TBB…)
- Performance comparison
- Summary

Parallel Programming with Threads

Recall 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 Programming

Several Thread Libraries/systems

- PTHREADS is the POSIX Standard
  - Relatively low level
  - Portable but possibly slow; relatively heavyweight
- OpenMP standard for application level programming
  - Support for scientific programming on shared memory
  - http://www.openMP.org
- TBB: Thread Building Blocks
  - Intel
- CILK: Language of the C “ilk”
  - Lightweight threads embedded into C
- Java threads
  - Built on top of POSIX threads
  - Object within Java language

Common Notions of Thread Creation

- cobegin/coend

```
cobegin
    job1(a1);
    job2(a2);
coend
```

- Statements in block may run in parallel
- cobegins may be nested
- Scoped, so you cannot have a missing coend

- fork/join

```
tid1 = fork(job1, a1);
job2(a2);
join tid1;
```

- Forked procedure runs in parallel
- Wait at join point if it's not finished

- future

```
v = future(job1(a1));
… = …v…;
```

- Future expression evaluated in parallel
- Attempt to use return value will wait

- Cobegin cleaner than fork, but fork is more general
- Futures require some compiler (and likely hardware) support

Overview of POSIX Threads

- POSIX: ***P****ortable* ***O****perating* ***S****ystem* ***I****nterface*
  - Interface to Operating System utilities
- PThreads: The POSIX threading interface
  - System calls to create and synchronize threads
  - Should be relatively uniform across UNIX-like OS platforms
- PThreads contain support for
  - Creating parallelism
  - Synchronizing
  - No explicit support for communication, because shared memory is implicit; a pointer to shared data is passed to a thread

Forking Posix Threads

Signature:

```
int pthread_create(pthread_t *,
                   const pthread_attr_t *,
                   void * (*)(void *),
                   void *);
```

Example call:

```
errcode = pthread_create(&thread_id; &thread_attribute
                         &thread_fun; &fun_arg);
```

- thread_id is the thread id or handle (used to halt, etc.)
- thread_attribute various attributes
  - Standard default values obtained by passing a NULL pointer
  - Sample attribute: minimum stack size
- thread_fun the function to be run (takes and returns void*)
- fun_arg an argument can be passed to thread_fun when it starts
- errorcode will be set nonzero if the create operation fails

Simple Threading Example

```
void* SayHello(void *foo) {
  printf( "Hello, world!\n" );
  return NULL;
}

int main() {
  pthread_t threads[16];
  int tn;
  for(tn=0; tn<16; tn++) {
    pthread_create(&threads[tn], NULL, SayHello, NULL);
  }
  for(tn=0; tn<16 ; tn++) {
    pthread_join(threads[tn], NULL);
  }
  return 0;
}
```

Compile using gcc –lpthread

Loop Level Parallelism

- Many scientific application have parallelism in loops
  - With threads:

```
… my_stuff [n][n];
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    … pthread_create (update_cell[i][j], …,
                      my_stuff[i][j]);
```

- But overhead of thread creation is nontrivial
  - update_cell should have a significant amount of work
  - 1/p-th if possible

Some More Pthread Functions

- **`pthread_yield();`**
  - Informs the scheduler that the thread is willing to yield its quantum, requires no arguments.
- **`pthread_exit(void *value);`**
  - Exit thread and pass value to joining thread (if exists)
- **`pthread_join(pthread_t *thread, void **result);`**
  - Wait for specified thread to finish. Place exit value into *result.

Others:

- **`pthread_t me; me = pthread_self();`**
  - Allows a pthread to obtain its own identifier pthread_t thread;
- **`pthread_detach(thread);`**
  - Informs the library that the threads exit status will not be needed by subsequent pthread_join calls resulting in better threads performance. For more information consult the library or the man pages, e.g., man -k pthread.

Shared Data and Threads

- Variables declared outside of main are shared
- Object allocated on the heap may be shared (if pointer is passed)
- Variables on the stack are private: passing pointer to these around to other threads can cause problems

- Often done by creating a large “thread data” struct
  - Passed into all threads as argument
  - Simple example:

```
char *message = "Hello World!\n";

pthread_create( &thread1,
      NULL,
      (void*)&print_fun,
      (void*) message);
```

Setting Attribute Values

- Once an initialized attribute object exists, changes can be made. For example:
  - To change the stack size for a thread to 8192 (before calling pthread_create), do this:
    - pthread_attr_setstacksize(&my_attributes, (size_t)8192);
  - To get the stack size, do this:
    - size_t my_stack_size;
      pthread_attr_getstacksize(&my_attributes, &my_stack_size);
- Other attributes:
  - Detached state – set if no other thread will use pthread_join to wait for this thread (improves efficiency)
  - Guard size – use to protect against stack overfow
  - Inherit scheduling attributes (from creating thread) – or not
  - Scheduling parameter(s) – in particular, thread priority
  - Scheduling policy – FIFO or Round Robin
  - Contention scope – with what threads does this thread compete for a CPU
  - Stack address – explicitly dictate where the stack is located
  - Lazy stack allocation – allocate on demand (lazy) or all at once, “up front”

Recall Data Race Example

```
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

Basic Types of Synchronization: Barrier

Barrier -- global synchronization

- Especially common when running multiple copies of the same function in parallel
  - SPMD "Single Program Multiple Data"
- simple use of barriers -- all threads hit the same one

```
work_on_my_subgrid();
barrier;
read_neighboring_values();
barrier;
```

- more complicated -- barriers on branches (or loops)

```
if (tid % 2 == 0) {
  work1();
  barrier
} else { barrier }
```

- barriers are not provided in all thread libraries

Creating and Initializing a Barrier

- To (dynamically) initialize a barrier, use code similar to this (which sets the number of threads to 3):

```
pthread_barrier_t b;
pthread_barrier_init(&b,NULL,3);
```

- The second argument specifies an attribute object for finer control; using NULL yields the default attributes.

- To wait at a barrier, a process executes:

```
pthread_barrier_wait(&b);
```

Basic Types of Synchronization: Mutexes

Mutexes -- mutual exclusion aka locks

- threads are working mostly independently
- need to access common data structure

```
lock *l = alloc_and_init();    /* shared */
acquire(l);
 access data
release(l);
```

- Locks only affect processors using them:
  - If a thread accesses the data without doing the acquire/release, locks by others will not help
- Java and other languages have lexically scoped synchronization, i.e., synchronized methods/blocks
  - Can't forgot to say "release"
- Semaphores generalize locks to allow k threads simultaneous access; good for limited resources

Mutexes in POSIX Threads

- To create a mutex:

```
#include <pthread.h>
pthread_mutex_t amutex = PTHREAD_MUTEX_INITIALIZER;
// or pthread_mutex_init(&amutex, NULL);
```

- To use it:

```
int pthread_mutex_lock(amutex);
int pthread_mutex_unlock(amutex);
```

- To deallocate a mutex

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- Multiple mutexes may be held, but can lead to problems:

thread1	thread2
lock(a)	lock(b)
lock(b)	lock(a)

- Deadlock results if both threads acquire one of their locks, so that neither can acquire the second

Summary of Programming with Threads

- POSIX Threads are based on OS features
  - Can be used from multiple languages (need appropriate header)
  - Familiar language for most of program
  - Ability to shared data is convenient
- Pitfalls
  - Data race bugs are very nasty to find because they can be intermittent
  - Deadlocks are usually easier, but can also be intermittent
- Researchers look at transactional memory an alternative
- OpenMP is commonly used today as an alternative

Parallel Programming in OpenMP

Introduction to OpenMP

- What is OpenMP?
  - Open specification for Multi-Processing
  - “Standard” API for defining multi-threaded shared-memory programs
  - openmp.org – Talks, examples, forums, etc.

- High-level API
  - Preprocessor (compiler) directives (~ 80%)
  - Library Calls (~ 19%)
  - Environment Variables (~ 1%)

A Programmer's View of OpenMP

- OpenMP is a portable, threaded, shared-memory programming *specification* with "light" syntax
  - Exact behavior depends on OpenMP *implementation*!
  - Requires compiler support (C or Fortran)

- OpenMP will:
  - Allow a programmer to separate a program into *serial regions* and *parallel regions,* rather than T concurrently-executing threads.
  - Hide stack management
  - Provide synchronization constructs

- OpenMP will not:
  - Parallelize automatically
  - Guarantee speedup
  - Provide freedom from data races

Motivation – OpenMP

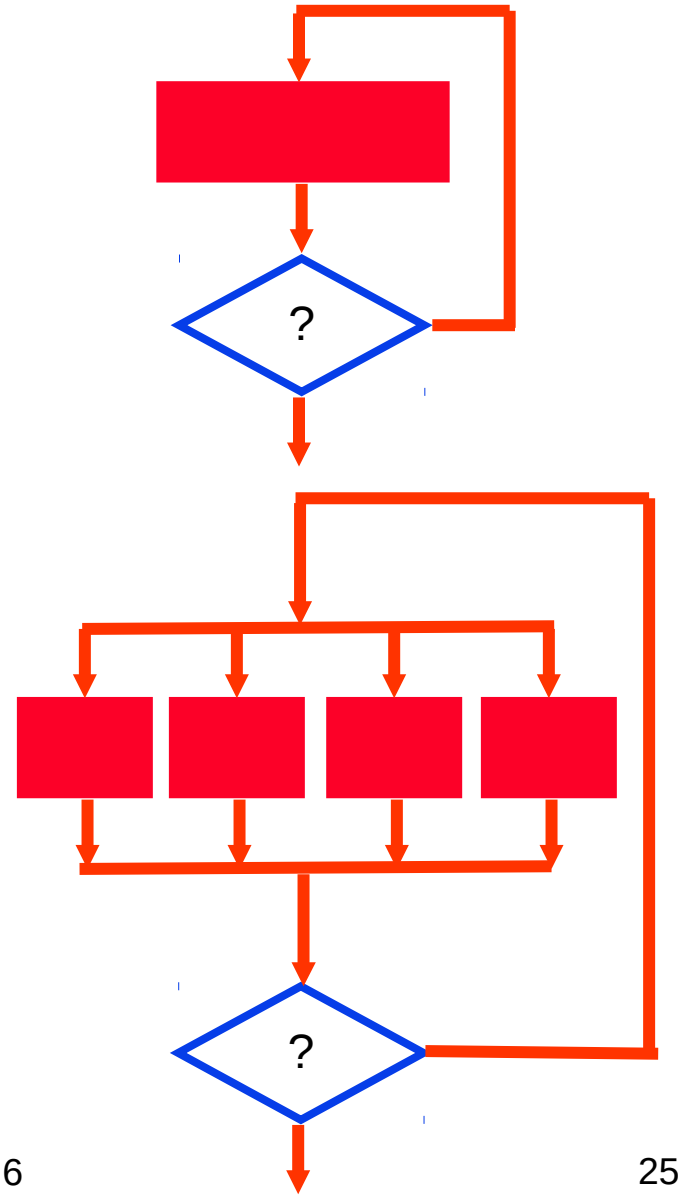
```
int main() {


  // Do this part in parallel


  printf( "Hello, World!\n" );


  return 0;
}
```

Motivation – OpenMP

```
int main() {

  omp_set_num_threads(16);

  // Do this part in parallel
  #pragma omp parallel
  {
    printf( "Hello, World!\n" );
  }

  return 0;
}
```

Programming Model – Concurrent Loops

- OpenMP easily parallelizes loops
  - Requires: No data dependencies (reads/write or write/write pairs) between iterations!
- Preprocessor calculates loop bounds for each thread directly from *serial* source




```
#pragma omp parallel for

for( i=0; i < 25; i++ )
{
  printf("Foo");
}
```

Programming Model – Loop Scheduling

- `schedule` clause determines how loop iterations are divided among the thread team
  - **`static([chunk])`** divides iterations statically between threads
    - Each thread receives `[chunk]` iterations, rounding as necessary to account for all iterations
    - Default **`[chunk]`** is **`ceil( # iterations / # threads )`**
  - **`dynamic([chunk])`** allocates **`[chunk]`** iterations per thread, allocating an additional **`[chunk]`** iterations when a thread finishes
    - Forms a logical work queue, consisting of all loop iterations
    - Default **`[chunk]`** is 1
  - **`guided([chunk])`** allocates dynamically, but **`[chunk]`** is exponentially reduced with each allocation

Programming Model – Data Sharing

- Parallel programs often employ two types of data
  - Shared data, visible to all threads, similarly named
  - Private data, visible to a single thread (often stack-allocated)
- PThreads:
  - Global-scoped variables are shared
  - Stack-allocated variables are private
- OpenMP:
  - **`shared`** variables are shared
  - **`private`** variables are private

```
// shared, globals
int bigdata[1024];

void* foo(void* bar) {
  int tid;
  int tid, stack
  int tid;
  #pragma omp parallel \
  /* Calculation goes */ \
   shared ( bigdata ) \
   private ( tid )
  here */
} {
    /* Calc. here */
  }
}
```

Programming Model - Synchronization

- OpenMP Synchronization
  - OpenMP Critical Sections
    - Named or unnamed
    - No *explicit* locks / mutexes
  - Barrier directives
  - Explicit Lock functions
    - When all else fails – may require `flush` directive
  - Single-thread regions *within* parallel regions
    - **`master, single`** directives

```
#pragma omp critical
{
  /* Critical code here */
}

#pragma omp barrier

omp_set_lock( lock l );
/* Code goes here */
omp_unset_lock( lock l );

#pragma omp single
{
  /* Only executed once */
}
```

Microbenchmark: Grid Relaxation (Stencil)

```
for( t=0; t < t_steps; t++) {

  #pragma omp parallel for \

   shared(grid,x_dim,y_dim) private(x,y)

  for( x=0; x < x_dim; x++) {
    for( y=0; y < y_dim; y++) {
      grid[x][y] = /* avg of neighbors */
    }
  }

   // Implicit Barrier Synchronization

  temp_grid = grid;
  grid = other_grid;
}
  other_grid = temp_grid;
```

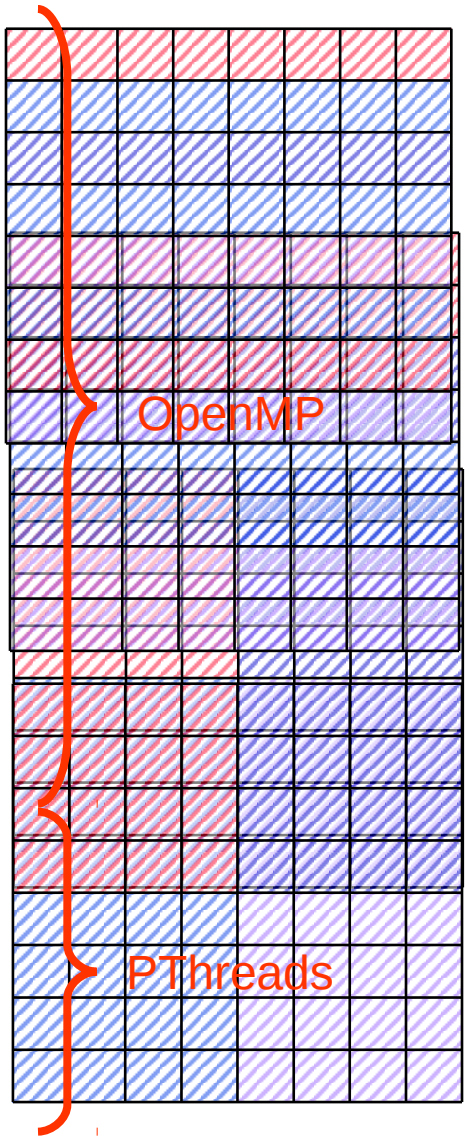
Microbenchmark: Structured Grid

- **`ocean_dynamic`** – Traverses entire ocean, row-by-row, assigning row iterations to threads with `dynamic` scheduling.

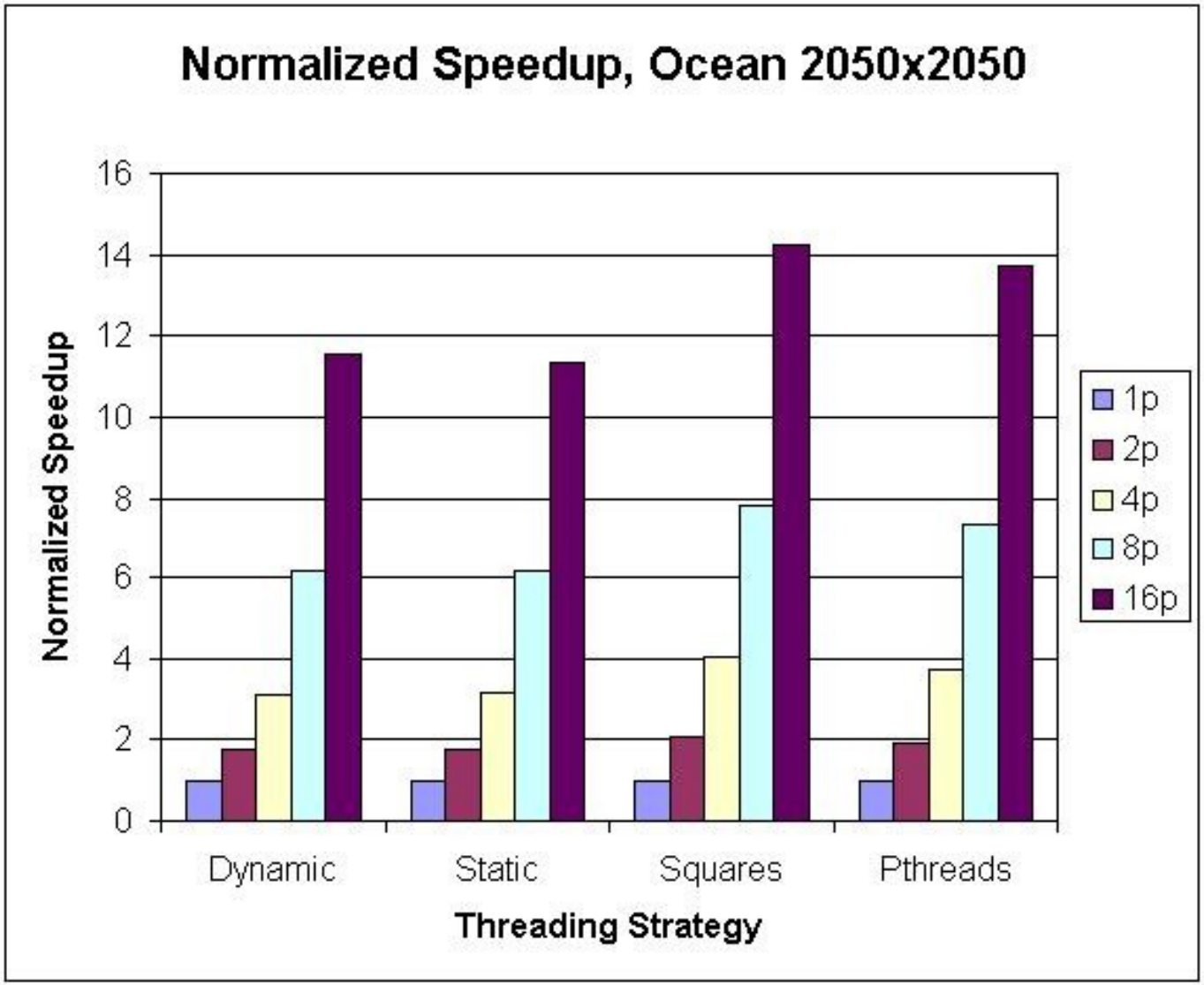
- **`ocean_static`** – Traverses entire ocean, row-by-row, assigning row iterations to threads with `static` scheduling.

- **`ocean_squares`** – Each thread traverses a square-shaped section of the ocean. Loop-level scheduling not used—loop bounds for each thread are determined explicitly.

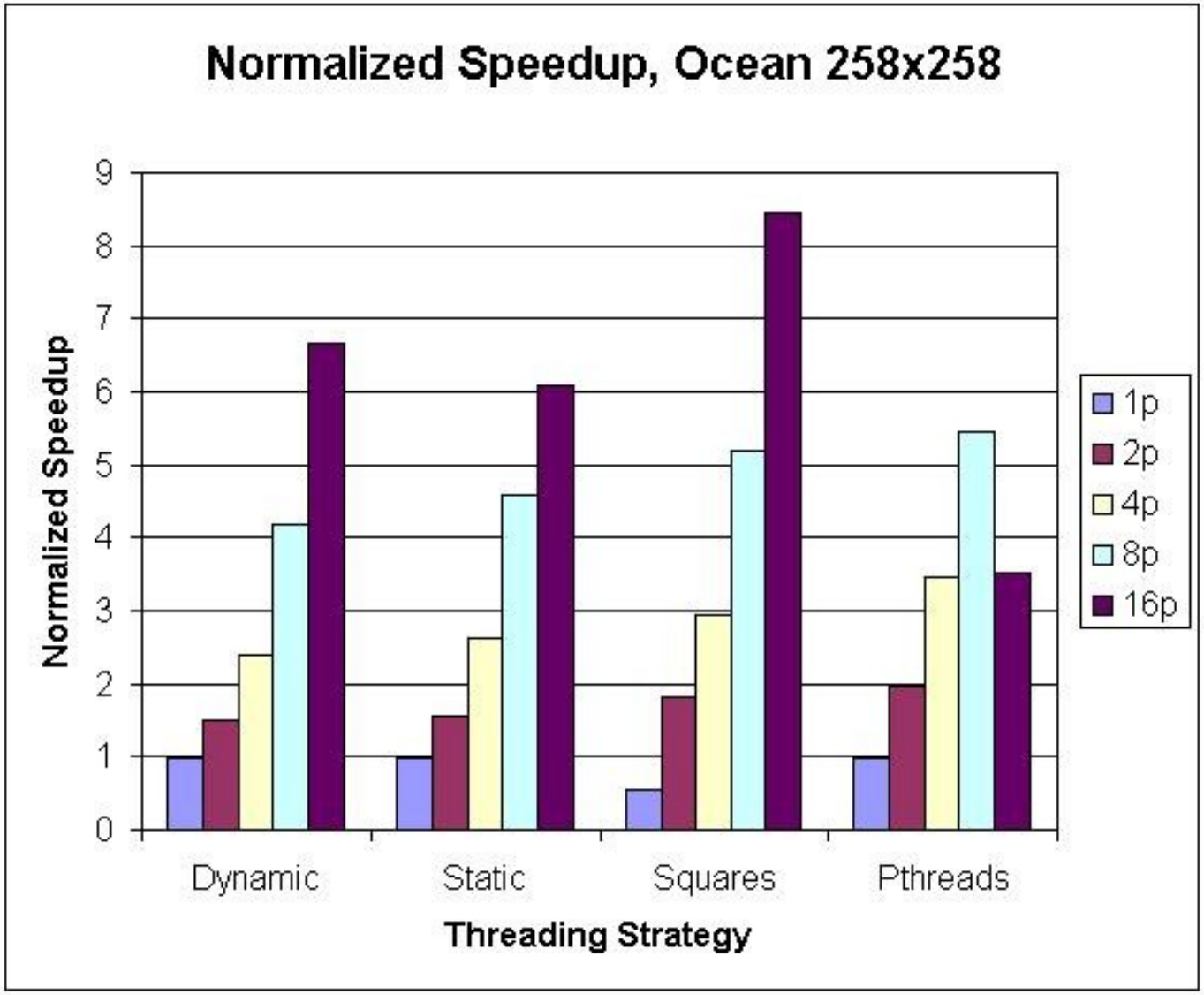
- **`ocean_pthreads`** – Each thread traverses a square-shaped section of the ocean. Loop bounds for each thread are determined explicitly.



Microbenchmark: Ocean



Microbenchmark: Ocean



Microbenchmark: GeneticTSP

- Genetic heuristic-search algorithm for approximating a solution to the Traveling Salesperson Problem (TSP)
  - Find shortest path through weighted graph, visiting each node once
- Operates on a *population* of possible TSP paths
  - Forms new paths by combining known, good paths (*crossover*)
  - Occasionally introduces new random elements (*mutation*)
- Variables:
  - `Np` – Population size, determines search space and working set size
  - `Ng` – Number of generations, controls effort spent refining solutions
  - `rC` – Rate of crossover, determines how many new solutions are produced and evaluated in a generation
  - `rM` – Rate of mutation, determines how often new (random) solutions are introduced

Microbenchmark: GeneticTSP

```
while( current_gen < Ng ) {
  Breed rC*Np new solutions:
    Select two parents
    Perform crossover()
    Mutate() with probability rM
    Evaluate() new solution

  Identify least-fit rC*Np solutions:
    Remove unfit solutions from population

  current_gen++
}

return the most fit solution found
```

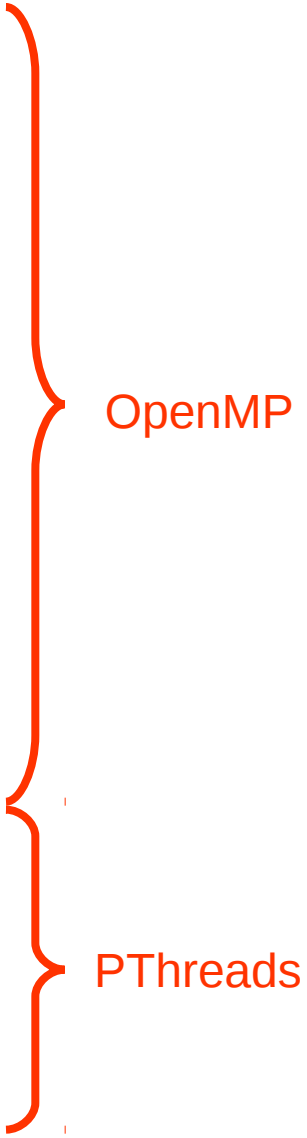
Outer loop has data dependence between iterations, as the population is not a loop invariant.

Can generate new solutions in parallel, but crossover(), mutate(), and evaluate() have varying runtimes.

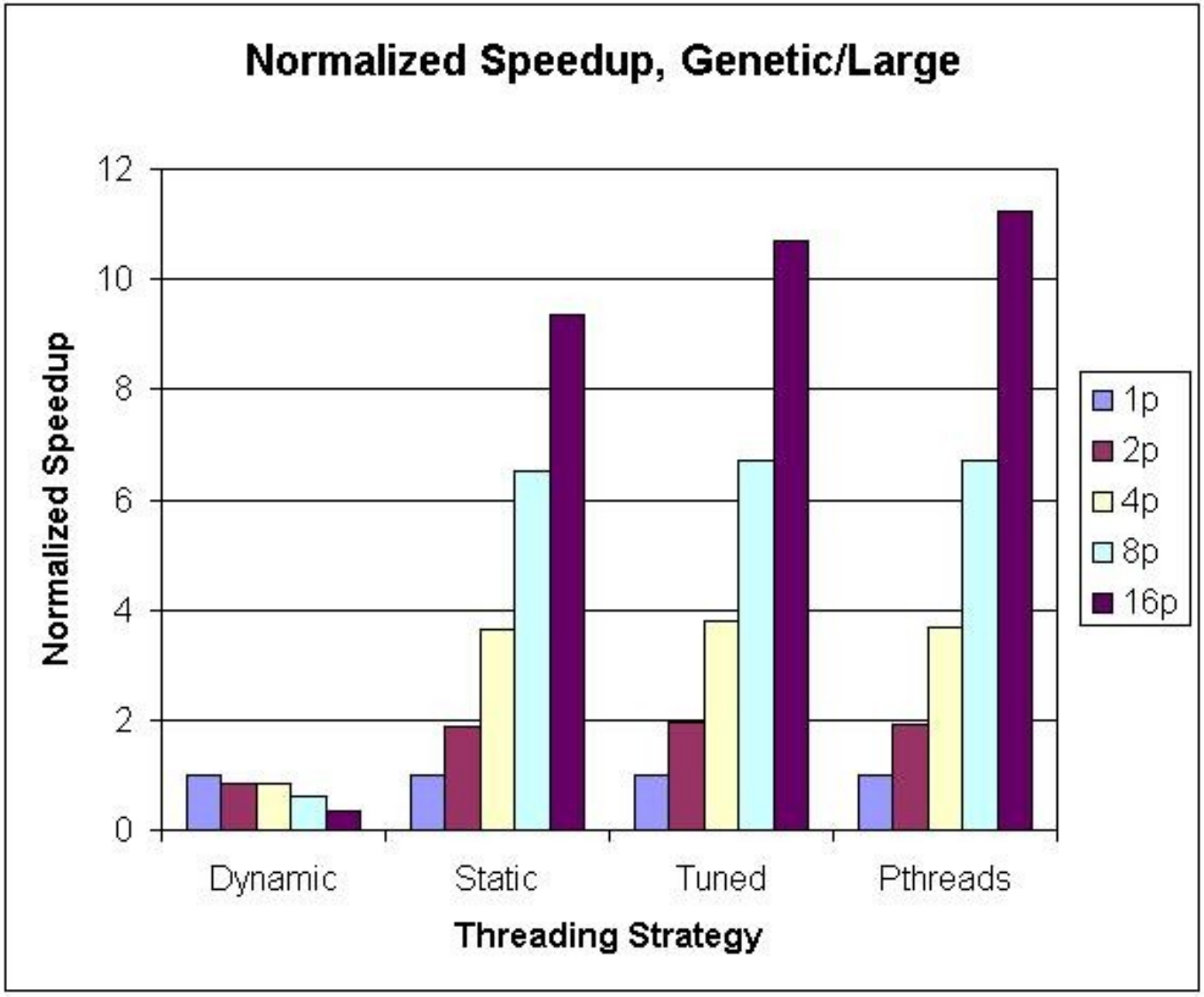
Threads can find least-fit population members in parallel, but only one thread should actually delete solutions.

Microbenchmark: GeneticTSP

- **`dynamic_tsp`** – Parallelizes both breeding loop and survival loop with OpenMP's `dynamic` scheduling
- **`static_tsp`** – Parallelizes both breeding loop and survival loop with OpenMP's `static` scheduling
- **`tuned_tsp`** – Attempt to tune scheduilng. Uses `guided` (exponential allocation) scheduling on breeding loop, `static` predicated scheduling on survival loop.
- **`pthreads_tsp`** – Divides iterations of breeding loop evenly among threads, conditionally executes survival loop in parallel



Microbenchmark: GeneticTSP



Evaluation

- OpenMP scales to 16-processor systems
  - Was overhead too high?
    - In some cases, yes
  - Did compiler-generated code compare to hand-written code?
    - Yes!
  - How did the loop scheduling options affect performance?
    - **`dynamic`** or **`guided`** scheduling helps loops with variable iteration runtimes
    - **`static`** or predicated scheduling more appropriate for shorter loops
- OpenMP is a good tool to parallelize (at least some!) applications

SpecOMP (2001)

- Parallel form of SPEC FP 2000 using Open MP, larger working sets
  - www.spec.org/omp
  - Aslot et. Al., Workshop on OpenMP Apps. and Tools (2001)

- Many of CFP2000 were “straightforward” to parallelize:
  - *ammp* (Computational chemistry): 16 Calls to OpenMP API, 13 `#pragmas`, converted linked lists to vector lists
  - *Applu* (Parabolic/elliptic PDE solver): 50 directives, mostly `parallel` or `do`
  - *Fma3d* (Finite element car crash simulation): 127 lines of OpenMP directives (60k lines total)
  - *mgrid* (3D multigrid): automatic translation to OpenMP
  - *Swim* (Shallow water modeling): 8 loops parallelized

OpenMP Summary

- OpenMP is a compiler-based technique to create concurrent code from (mostly) serial code
- OpenMP can enable (easy) parallelization of loop-based code
  - Lightweight syntactic language extensions
- OpenMP performs comparably to manually-coded threading
  - Scalable
  - Portable
- Not a silver bullet for all applications

More Information

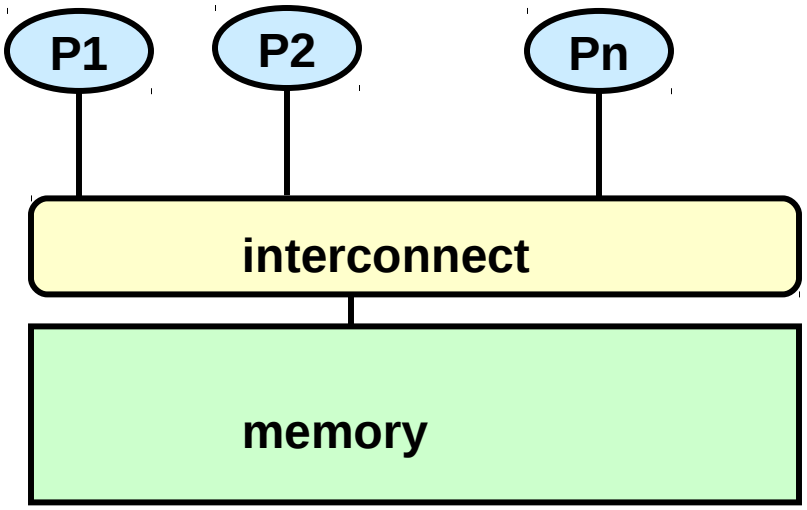
- openmp.org
  - OpenMP official site
- www.llnl.gov/computing/tutorials/openMP/
  - A handy OpenMP tutorial
- www.nersc.gov/assets/Uploads/XE62011OpenMP.pdf
  - Another OpenMP tutorial and reference

Extra Slides

Shared Memory Hardware and Memory Consistency

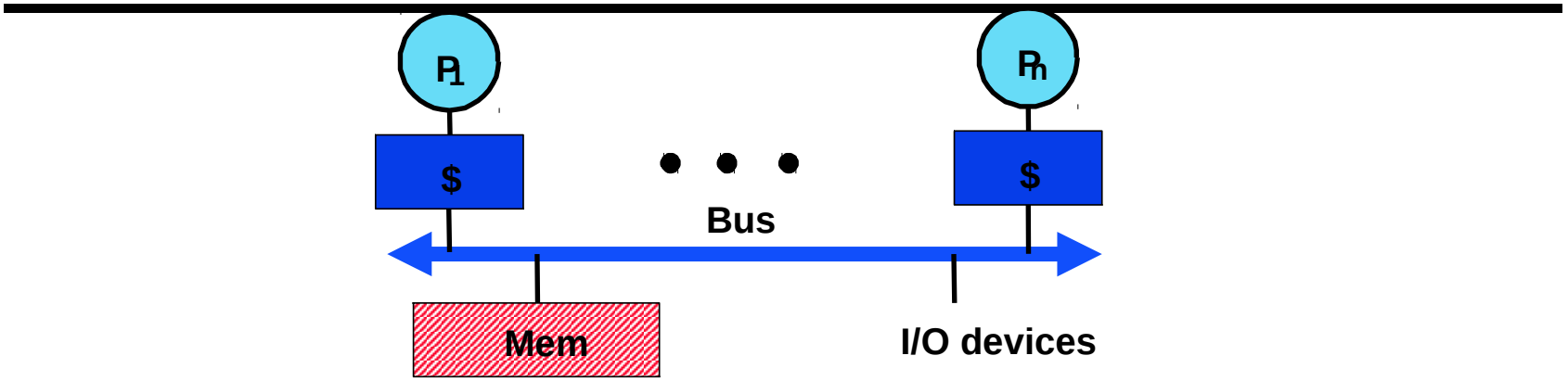
Basic Shared Memory Architecture

- Processors all connected to a large shared memory
  - Where are caches?



- Now take a closer look at structure, costs, limits, programming

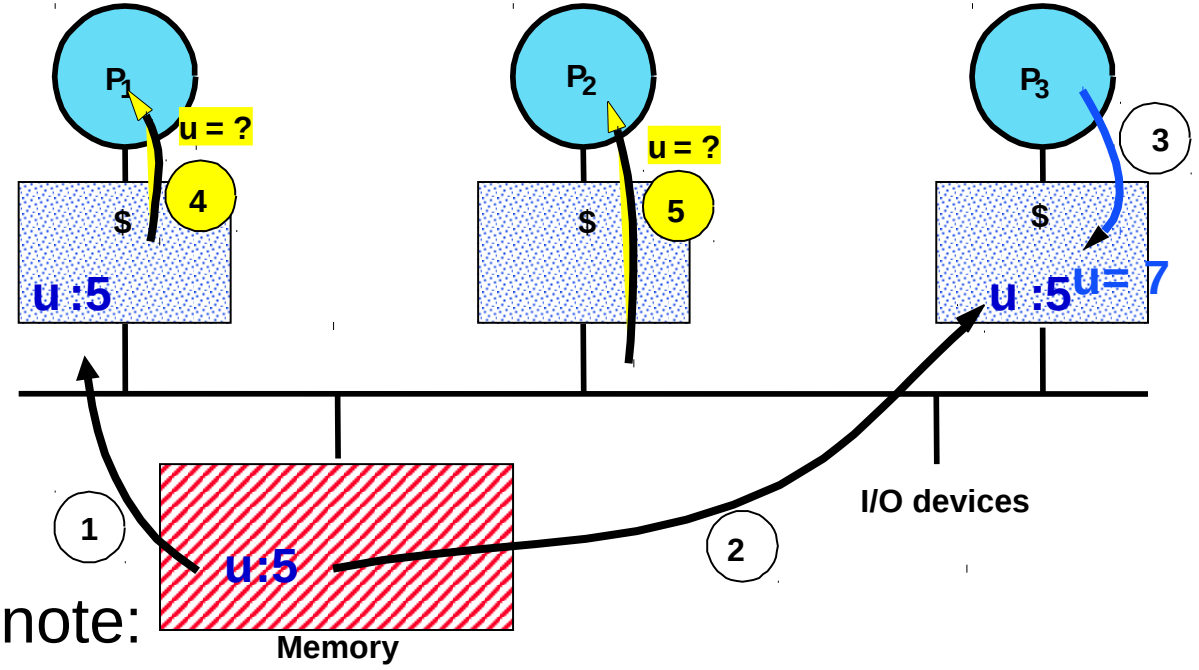
What About Caching???



- Want high performance for shared memory: Use Caches!
  - Each processor has its own cache (or multiple caches)
  - Place data from memory into cache
  - Writeback cache: don't send all writes over bus to memory
- Caches reduce average latency
  - Automatic replication closer to processor
  - *More* important to multiprocessor than uniprocessor: latencies longer
- Normal uniprocessor mechanisms to access data
  - Loads and Stores form very low-overhead communication primitive
- Problem: Cache Coherence!

Slide source: John Kubiatowicz

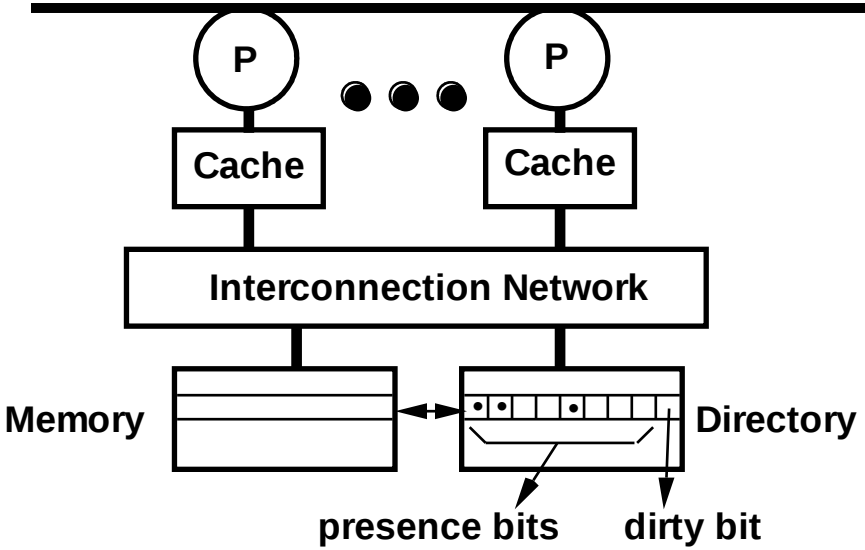
Example Cache Coherence Problem



- Things to note:
  - Processors could 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
- How to fix with a bus: Coherence Protocol
  - Use bus to broadcast writes or invalidations
  - Simple protocols rely on presence of broadcast medium
- Bus not scalable beyond about 64 processors (max)
  - Capacity, bandwidth limitations

Slide source: John Kubiatowicz

Scalable Shared Memory: Directories



- **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**

- 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
- Each Reader recorded in directory
- Processor asks permission of memory before writing:
  - Send invalidation to each cache with read-only copy
  - Wait for acknowledgements before returning permission for writes

Slide source: John Kubiatowicz

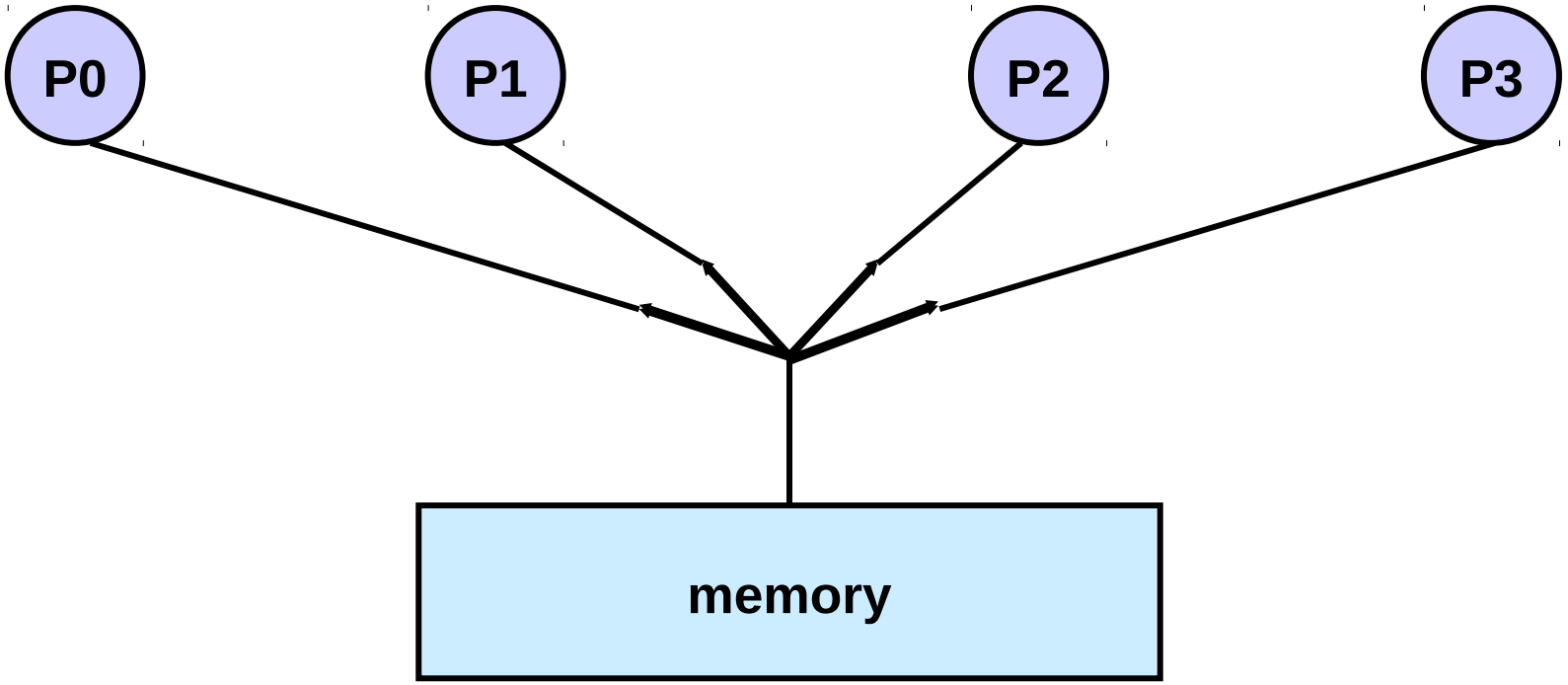
Intuitive Memory Model

- Reading an address should return the last value written to that address
- Easy in uniprocessors
  - except for I/O
- Cache coherence problem in MPs is more pervasive and more performance critical
- More formally, this is called sequential consistency:

  “A multiprocessor is *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.” [Lamport, 1979]

Sequential Consistency Intuition

- Sequential consistency says the machine *behaves as if* it does the following



Memory Consistency Semantics

What does this imply about program behavior?

- **No process ever sees "garbage" values, i.e., average of 2 values**
- **Processors always see values written by some processor**
- **The value seen is constrained by program order on all processors**
  - **Time always moves forward**
- **Example: *spin lock***
  - **P1 writes data=1, then writes flag=1**
  - **P2 waits until flag=1, then reads data**

If P2 sees the new value of flag (=1), it must see the new value of data (=1)

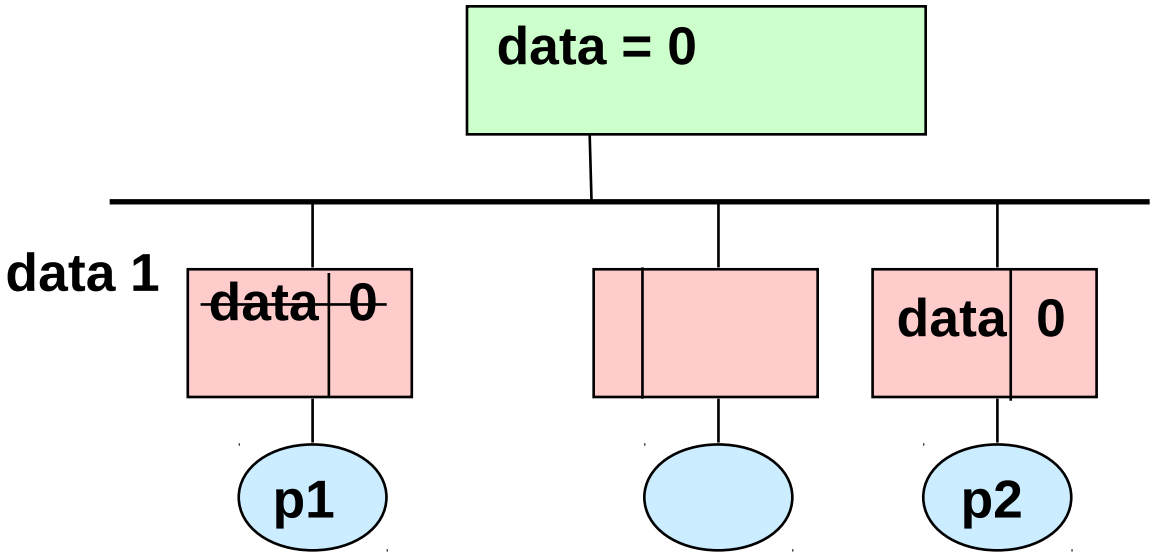
initially:	flag=0 data=0

P1	P2
data = 1 flag = 1	10: if flag=0, goto 10 …= data

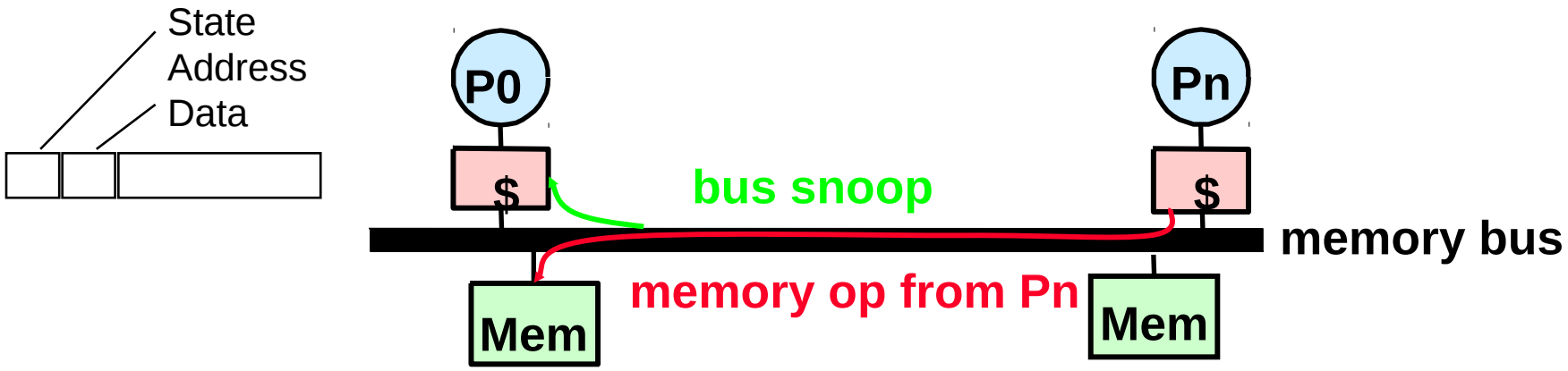
If P2 reads flag	Then P2 may read data
0	1
0	0
1	1

Are Caches "Coherent" or Not?

- Coherence means different copies of same location have same value, incoherent otherwise:
- p1 and p2 both have cached copies of data (= 0)
- p1 writes data=1
  - May "write through" to memory
- p2 reads data, but gets the "stale" cached copy
  - This may happen even if it read an updated value of another variable, flag, that came from memory

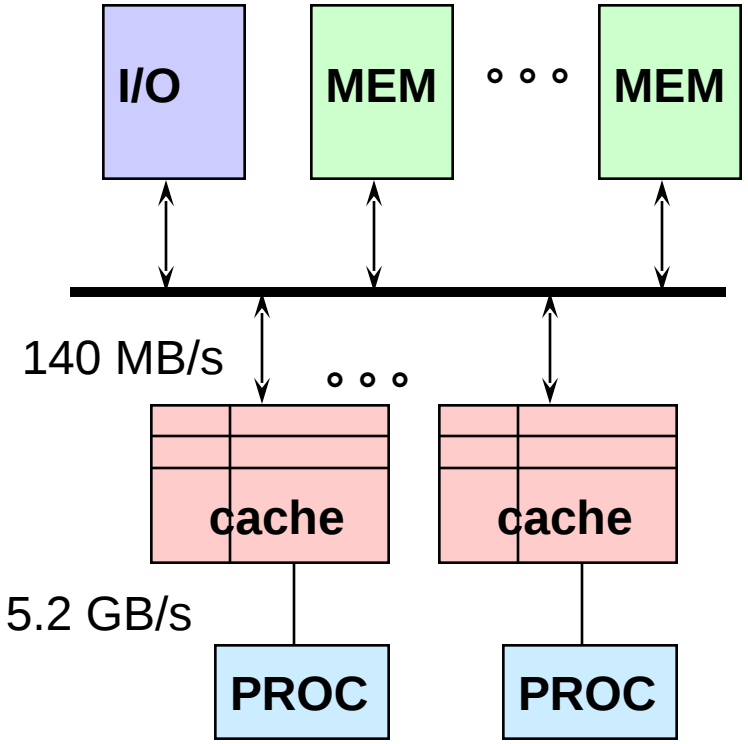


Snoopy Cache-Coherence Protocols



- Memory bus is a broadcast medium
- Caches contain information on which addresses they store
- Cache Controller "snoops" all transactions on the bus
  - A transaction is a relevant transaction if it involves a cache block currently contained in this cache
  - Take action to ensure coherence
    - invalidate, update, or supply value
  - Many possible designs (see CS252 or CS258)

Limits of Bus-Based Shared Memory



Assume:

1 GHz processor w/o cache

=> 4 GB/s inst BW per processor (32-bit)

=> 1.2 GB/s data BW at 30% load-store

Suppose 98% inst hit rate and 95% data hit rate

=> 80 MB/s inst BW per processor

=> 60 MB/s data BW per processor

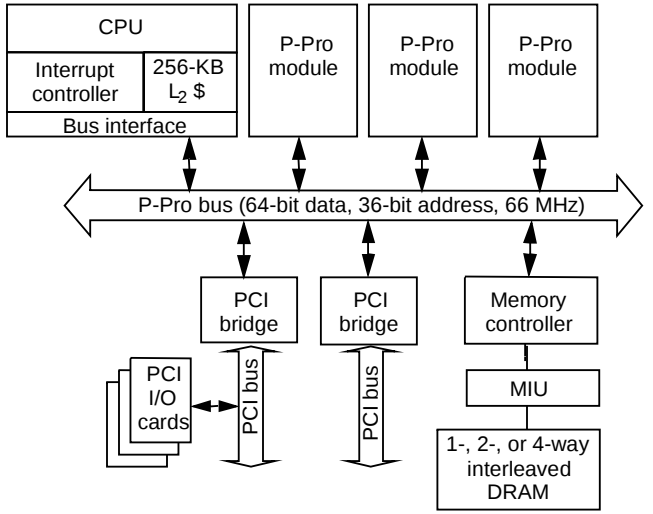
⇒140 MB/s combined BW

Assuming 1 GB/s bus bandwidth

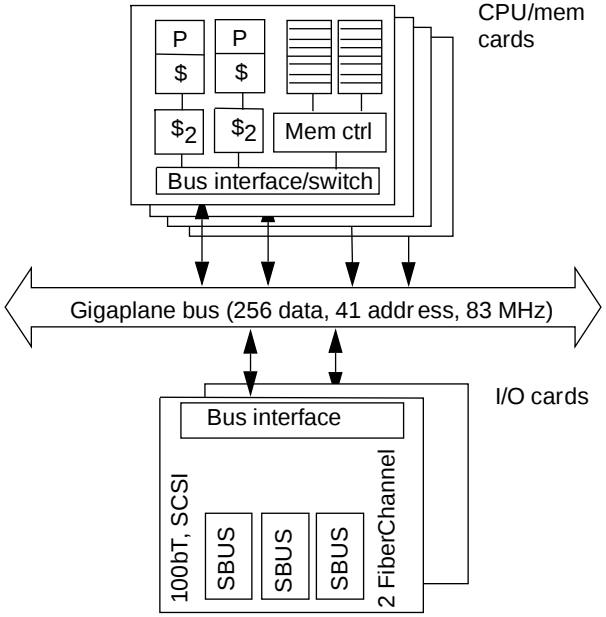
∴ 8 processors will saturate bus

Sample Machines

- Intel Pentium Pro Quad
  - Coherent
  - 4 processors



- Sun Enterprise server
  - Coherent
  - Up to 16 processor and/or memory-I/O cards

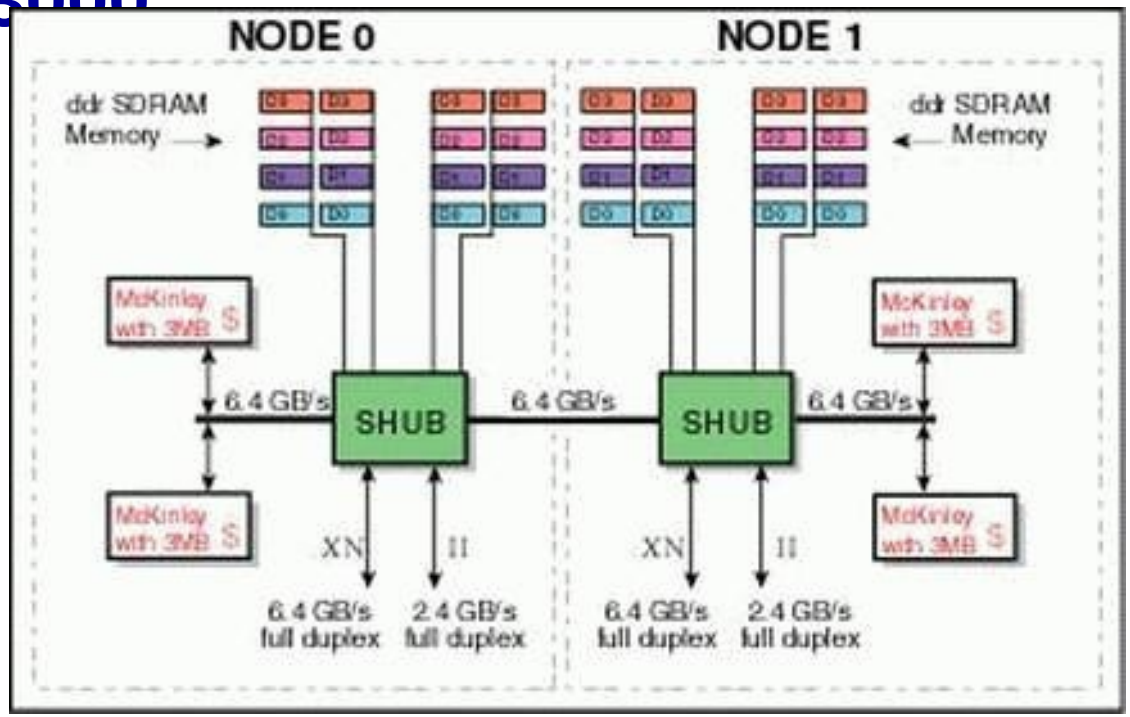


- IBM Blue Gene/L
  - L1 not coherent, L2 shared

Directory Based Memory/Cache Coherence

- Keep Directory to keep track of which memory stores latest copy of data
- Directory, like cache, may keep information such as:
  - Valid/invalid
  - Dirty (inconsistent with memory)
  - Shared (in another caches)
- When a processor executes a write operation to shared data, basic design choices are:
  - With respect to memory:
    - Write through cache: do the write in memory as well as cache
    - Write back cache: wait and do the write later, when the item is flushed
  - With respect to other cached copies
    - Update: give all other processors the new value
    - Invalidate: all other processors remove from cache
- See CS252 or CS258 for details

SGI Altix 3000



- **A node contains up to 4 Itanium 2 processors and 32GB of memory**
- **Network is SGI's NUMAlink, the NUMAflex interconnect technology.**
- **Uses a mixture of snoopy and directory-based coherence**
- **Up to 512 processors that are cache coherent (global address space is possible for larger machines)**

Sharing: A Performance Problem

- True sharing
  - Frequent writes to a variable can create a bottleneck
  - OK for read-only or infrequently written data
  - Technique: make copies of the value, one per processor, if this is possible in the algorithm
  - Example problem: the data structure that stores the freelist/heap for malloc/free
- False sharing
  - Cache block may also introduce artifacts
  - Two distinct variables in the same cache block
  - Technique: allocate data used by each processor contiguously, or at least avoid interleaving in memory
  - Example problem: an array of ints, one written frequently by each processor (many ints per cache line)

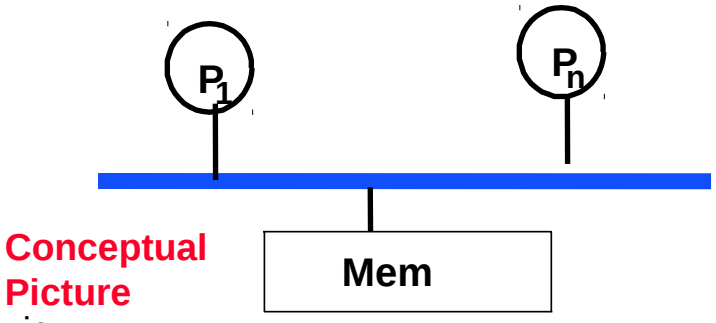
Cache Coherence and Sequential Consistency

- There is a lot of hardware/work to ensure coherent caches
  - Never more than 1 version of data for a given address in caches
  - Data is always a value written by some processor
- But other HW/SW features may break sequential consistency (SC):
  - The compiler reorders/removes code (e.g., your spin lock, see next slide)
  - The compiler allocates a register for flag on Processor 2 and spins on that register value without ever completing
  - Write buffers (place to store writes while waiting to complete)
    - Processors may reorder writes to merge addresses (not FIFO)
    - Write X=1, Y=1, X=2 (second write to X may happen before Y's)
  - Prefetch instructions cause read reordering (read data before flag)
  - The network reorders the two write messages.
  - The write to flag is nearby, whereas data is far away.
  - Some of these can be prevented by declaring variables "volatile"
- Most current commercial SMPs give up SC
  - A correct program on a SC processor may be incorrect on one that is not

Example: Coherence not Enough

P_1	P_2
/*Assume initial value of A and ag is 0*/	
`A = 1;`	`while (flag == 0);` /*spin idly*/
`flag = 1;`	`print A;`

- Intuition not guaranteed by coherence
- expect memory to respect order between accesses to *different* locations issued by a given process
  - to preserve orders among accesses to same location by different processes
- Coherence is not enough!
  - pertains only to single location
  - Need statement about ordering between multiple locations.



Conceptual Picture

Slide source: John Kubiatowicz

Programming with Weaker Memory Models than SC

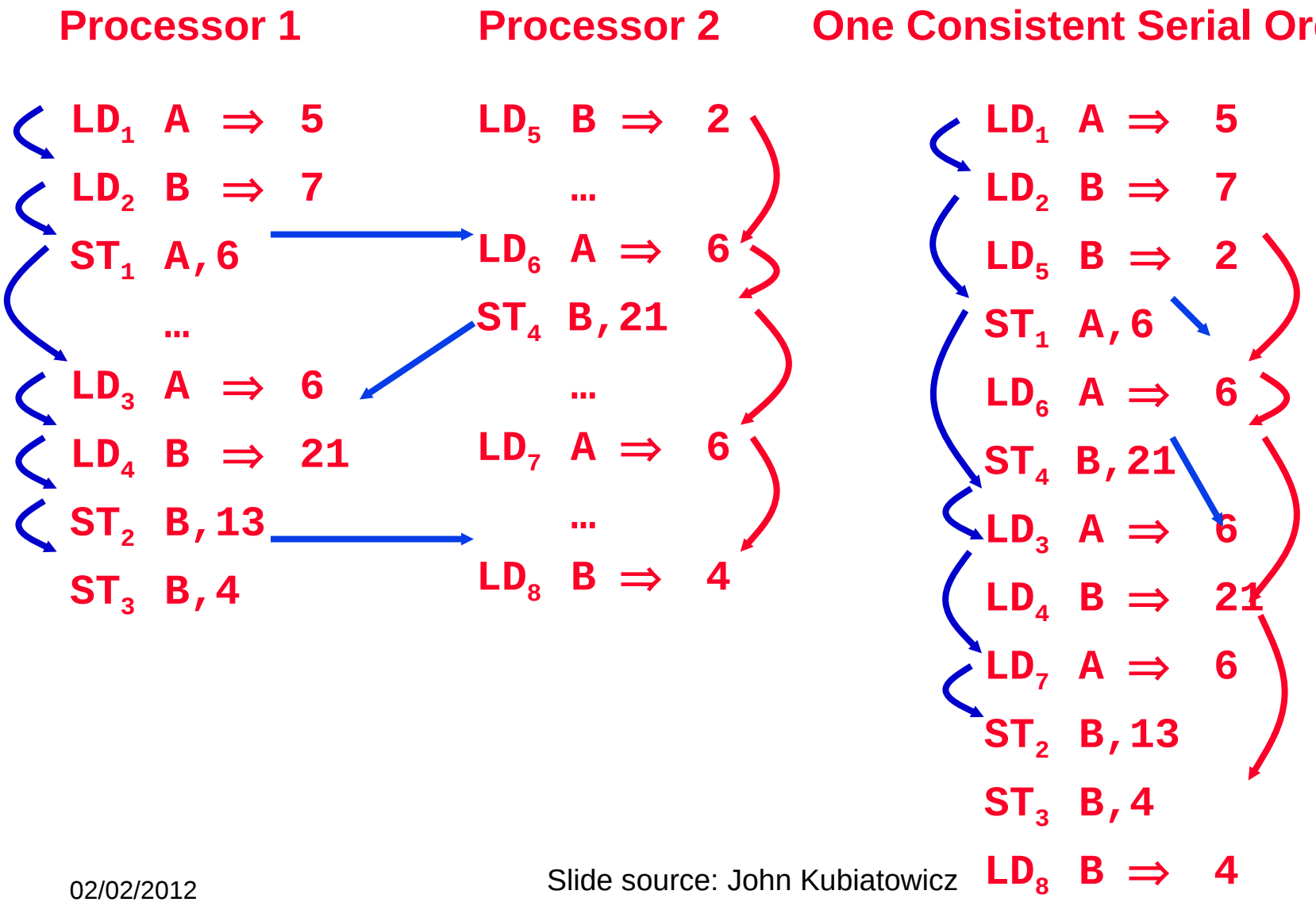
- Possible to reason about machines with fewer properties, but difficult
- Some rules for programming with these models
  - Avoid race conditions
  - Use system-provided synchronization primitives
  - At the assembly level, may use “fences” (or analogs) directly
- The high level language support for these differs
  - Built-in synchronization primitives normally include the necessary fence operations
    - lock (), … only one thread at a time allowed here…. unlock()
    - Region between lock/unlock called **critical region**
  - For performance, need to keep critical region short

What to Take Away?

- Programming shared memory machines
  - May allocate data in large shared region without too many worries about where
  - Memory hierarchy is critical to performance
    - Even more so than on uniprocessors, due to coherence traffic
  - For performance tuning, watch sharing (both true and false)
- Semantics
  - Need to lock access to shared variable for read-modify-write
  - Sequential consistency is the natural semantics
    - Write race-free programs to get this
  - Architects worked hard to make this work
    - Caches are coherent with buses or directories
    - No caching of remote data on shared address space machines
  - But compiler and processor may still get in the way
    - Non-blocking writes, read prefetching, code motion…
    - Avoid races or use machine-specific fences carefully

Extra Slides

Sequential Consistency Example



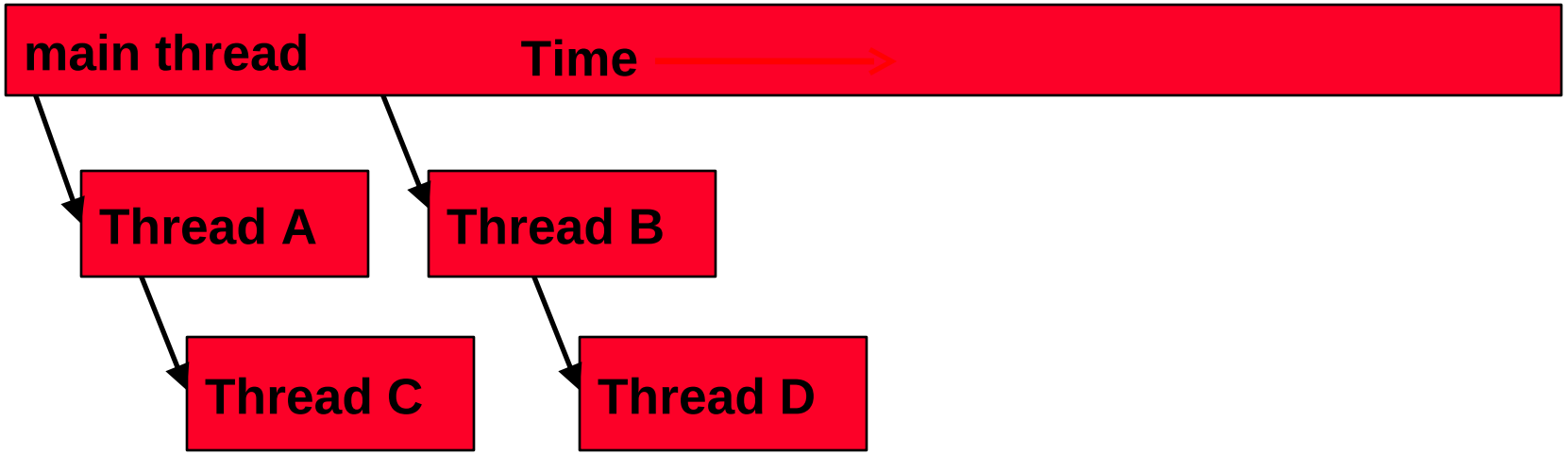
Slide source: John Kubiatowicz

Multithreaded Execution

- Multitasking operating system:
  - Gives “illusion” that multiple things happening at same time
  - Switches at a course-grained time quanta (for instance: 10ms)
- Hardware Multithreading: multiple threads share processor simultaneously (with little OS help)
  - Hardware does switching
    - HW for fast thread switch in small number of cycles
    - much faster than OS switch which is 100s to 1000s of clocks
  - Processor duplicates 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
- When to 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)

Slide source: John Kubiatowicz

Thread Scheduling



- Once created, when will a given thread run?
  - It is up to the Operating System or hardware, but it will run eventually, even if you have more threads than cores
  - But – scheduling may be non-ideal for your application
- Programmer can provide hints or affinity in some cases
  - E.g., create exactly P threads and assign to P cores
- Can provide user-level scheduling for some systems
  - Application-specific tuning based on programming model
  - Work in the ParLAB on making user-level scheduling easy to do (Lithe)

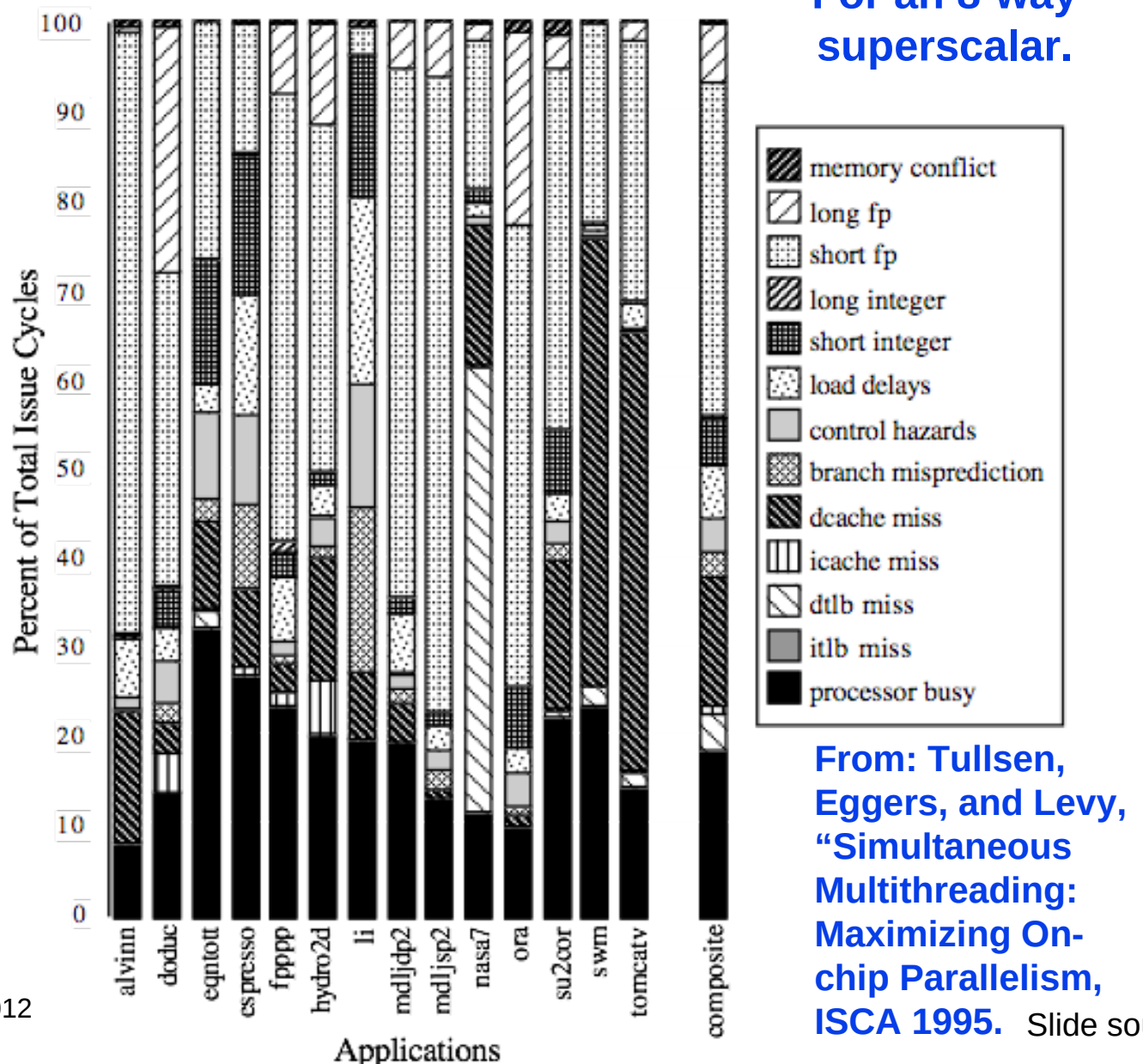
Slide source: John Kubiatowicz

What about combining ILP and TLP?

- TLP and ILP exploit two different kinds of parallel structure in a program
- Could a processor oriented at ILP benefit from exploiting TLP?
  - functional units are often idle in data path designed for ILP because of either stalls or dependences in the code
  - TLP used as a source of independent instructions that might keep the processor busy during stalls
  - TLP be used to occupy functional units that would otherwise lie idle when insufficient ILP exists
- Called “Simultaneous Multithreading”
  - Intel renamed this “Hyperthreading”

Slide source: John Kubiatowicz

Quick Recall: Many Resources IDLE!

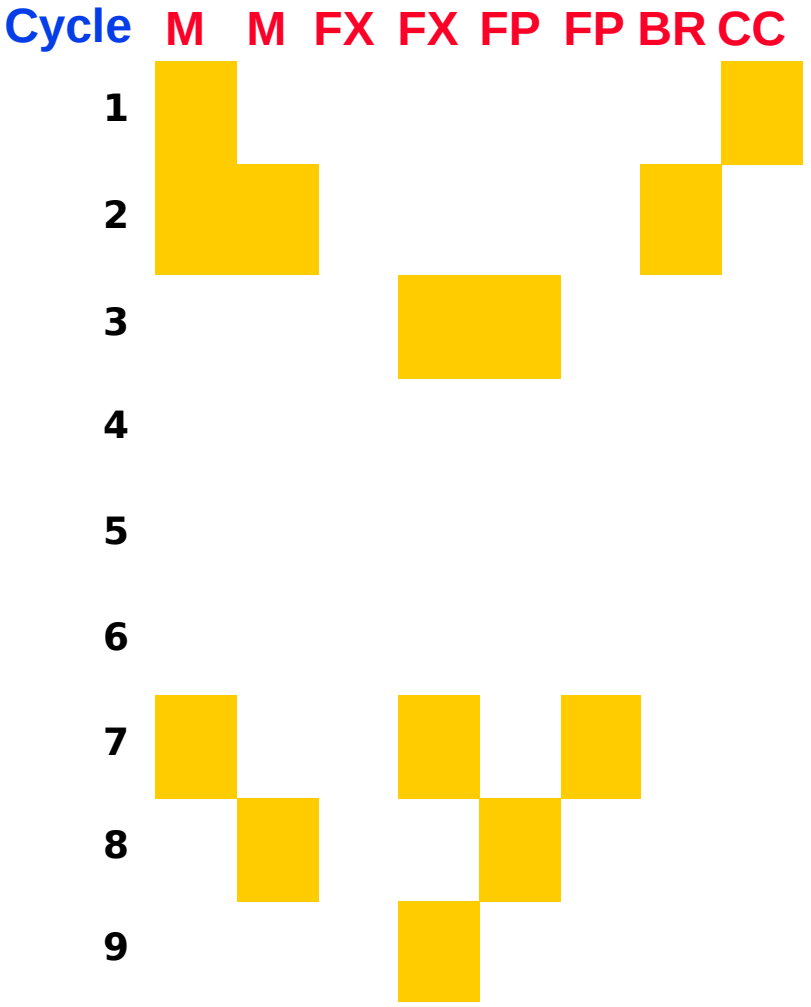


For an 8-way superscalar.

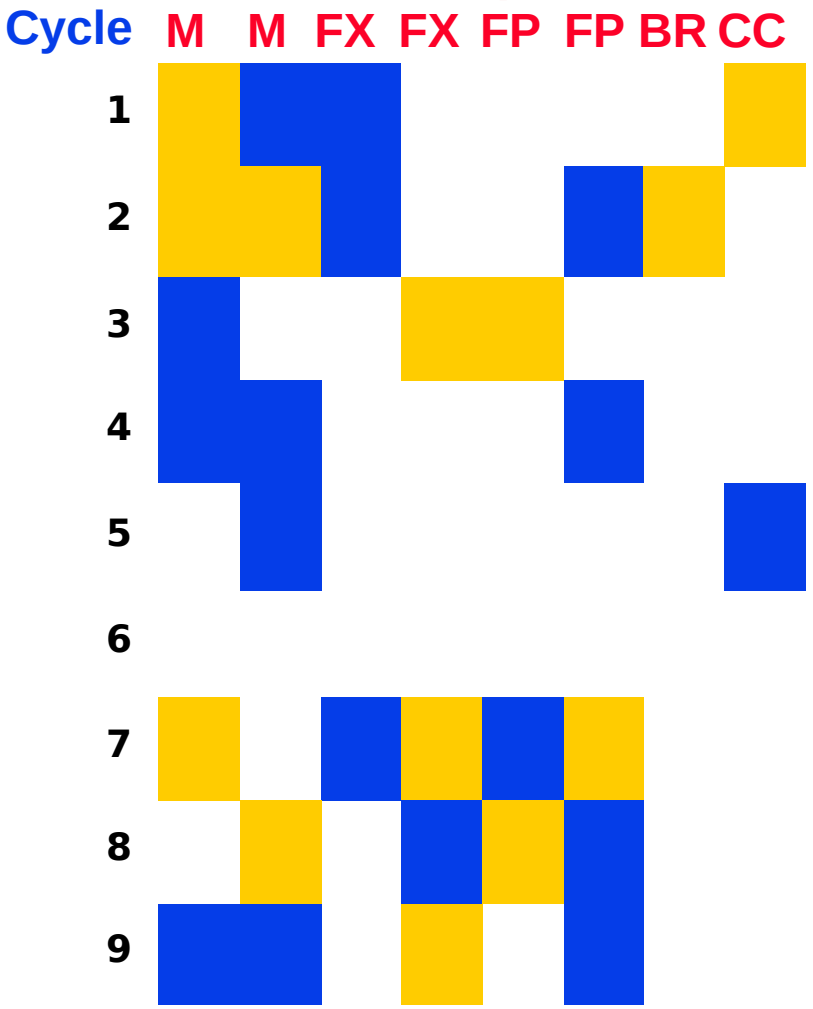
From: Tullsen, Eggers, and Levy, "Simultaneous Multithreading: Maximizing On-chip Parallelism, ISCA 1995. Slide source: John Kubiatowicz

Simultaneous Multi-threading ...

One thread, 8 units



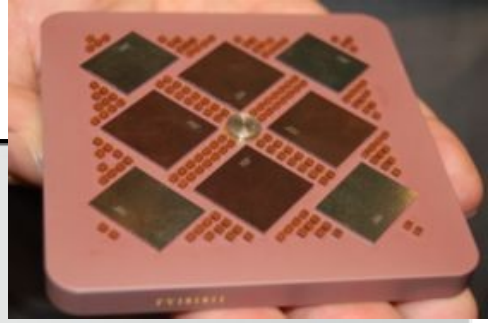
Two threads, 8 units



M = Load/Store, FX = Fixed Point, FP = Floating Point, BR = Branch, CC = Condition Codes

Slide source: John Kubiatowicz

Power 5 dataflow ...



- Why only two threads?
  - **With 4, one of the shared resources (physical registers, cache, memory bandwidth) would be prone to bottleneck**
- Cost:
  - **The Power5 core is about 24% larger than the Power4 core because of the addition of SMT support**