COMP 422, Lecture 15: Mid-term Review

Vivek Sarkar

Department of Computer Science
Rice University

vsarkar@rice.edu

Lecture 2: Bandwidth and Latency

- **Memory bandwidth is determined by the bandwidth of the memory bus as well as the memory units.**
- **Memory bandwidth can be improved by increasing the size of memory blocks.**
- **The underlying system takes *l* time units (where *l* is the latency of the system) to deliver *b* units of data (where *b* is the block size).**

Multithreading for Latency Hiding

A thread is a single stream of control in the flow of a program.

We illustrate threads with a simple example:

```
for (i = 0; i < n; i++)
     c[i] = dot_product(get_row(a, i), b);
```

Each dot-product is independent of the other, and therefore represents a concurrent unit of execution. We can safely rewrite the above code segment as:

```
for (i = 0; i < n; i++)
    c[i] = create_thread(dot_product,get_row(a, i), b);
```

Multithreading for Latency Hiding: Example

- In the code, the first instance of this function accesses a pair of vector elements and waits for them.
- In the meantime, the second instance of this function can access two other vector elements in the next cycle, and so on.
- After *l* units of time, where *l* is the latency of the memory system, the first function instance gets the requested data from memory and can perform the required computation.
- In the next cycle, the data items for the next function instance arrive, and so on. In this way, in every clock cycle, we can perform a computation.

Prefetching for Latency Hiding

- Misses on loads cause programs to stall.
- Why not advance the loads so that by the time the data is actually needed, it is already there!
- The only drawback is that you might need more space to store advanced loads.
- However, if the advanced loads are overwritten, we are no worse than before!

Lecture 2 Review Question

- Consider two cases for a processor with a 1 GHz clock, 1ns access to cache, 100 ns access to DRAM, and a workload with 90% and 25% hit rates for cache sizes of 32KB and 1KB respectively:

  Case 1: 1 thread executes the workload with a 32KB cache

  Case 2: 32 threads execute the workload with 1KB cache/thread

— What memory bandwidth is required in the two cases?

- Case 1: (1 - 0.9) words/cycle = 0.1*4B/1ns = 400MB/s
- Case 2: (1 - 0.25) words/cycle = 0.75*4B/1ns = 3GB/s

The bandwidth ratio for Cases 1 and 2 is directly proportional to the miss rate ratio, 0.75 / 0.1

See Example 2.9 on page 23 of textbook

Lecture 3: Metrics for Interconnection Networks

- ***Diameter:*** **The distance between the farthest two nodes in the network. Metric for worst-case latency.**
- ***Bisection Width:*** **The minimum number of wires you must cut to divide the network into two equal parts. Metric for worst-case bandwidth.**
- ***Arc Connectivity:*** **The minimum number of wires you must cut to partition the network into (not necessarily equal) parts. Metric for fault tolerance. Arc Connectivity is always <= Bisection Bandwidth.**
- ***Cost:*** **The number of links or switches (whichever is asymptotically higher) are important contributors to cost. However, a number of other factors, such as the ability to layout the network, the length of wires, etc., also factor in to the cost.**

Bisection Width: Example

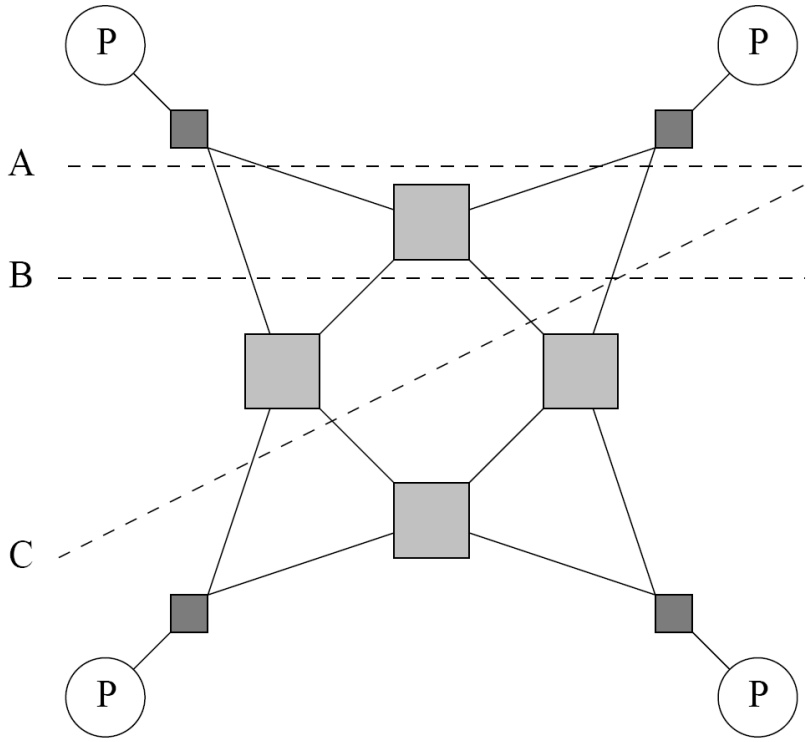


Figure 2.20 Bisection width of a dynamic network is computed by examining various equi-partitions of the processing nodes and selecting the minimum number of edges crossing the partition. In this case, each partition yields an edge cut of four. Therefore, the bisection width of this graph is four.

Evaluating Dynamic Interconnection Networks

Table 2.2 A summary of the characteristics of various dynamic network topologies connecting p processing nodes.

Network	Diameter	Bisection Width	Arc Connectivity	Cost (No. of links)
Crossbar	1	p	1	p^2
Omega Network	$\log p$	$p/2$	2	$p/2$
Dynamic Tree	$2 \log p$	1	2	$p - 1$

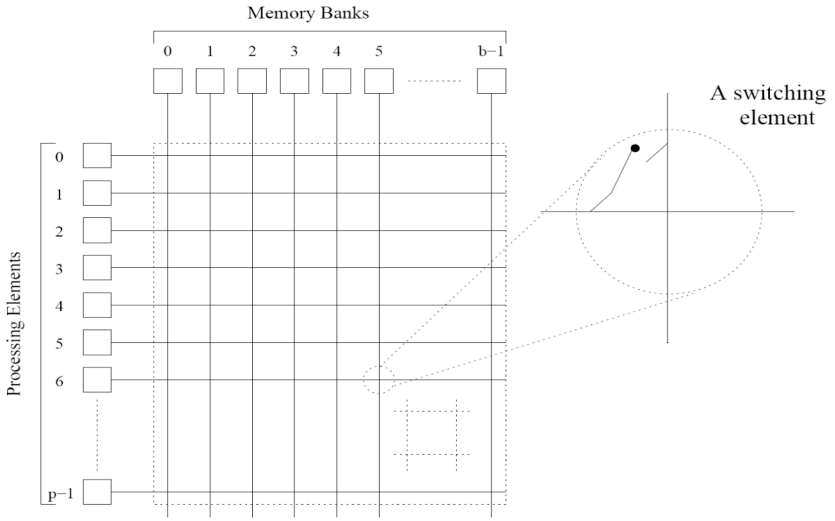
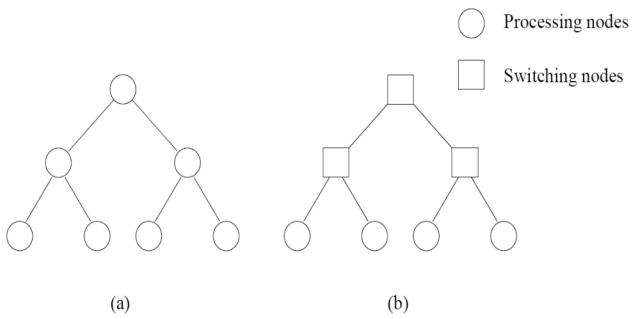


Figure 2.8 A completely non-blocking crossbar network connecting p processors to b memory banks.



Complete binary tree networks: (a) a static tree network; and (b) a dynamic tree

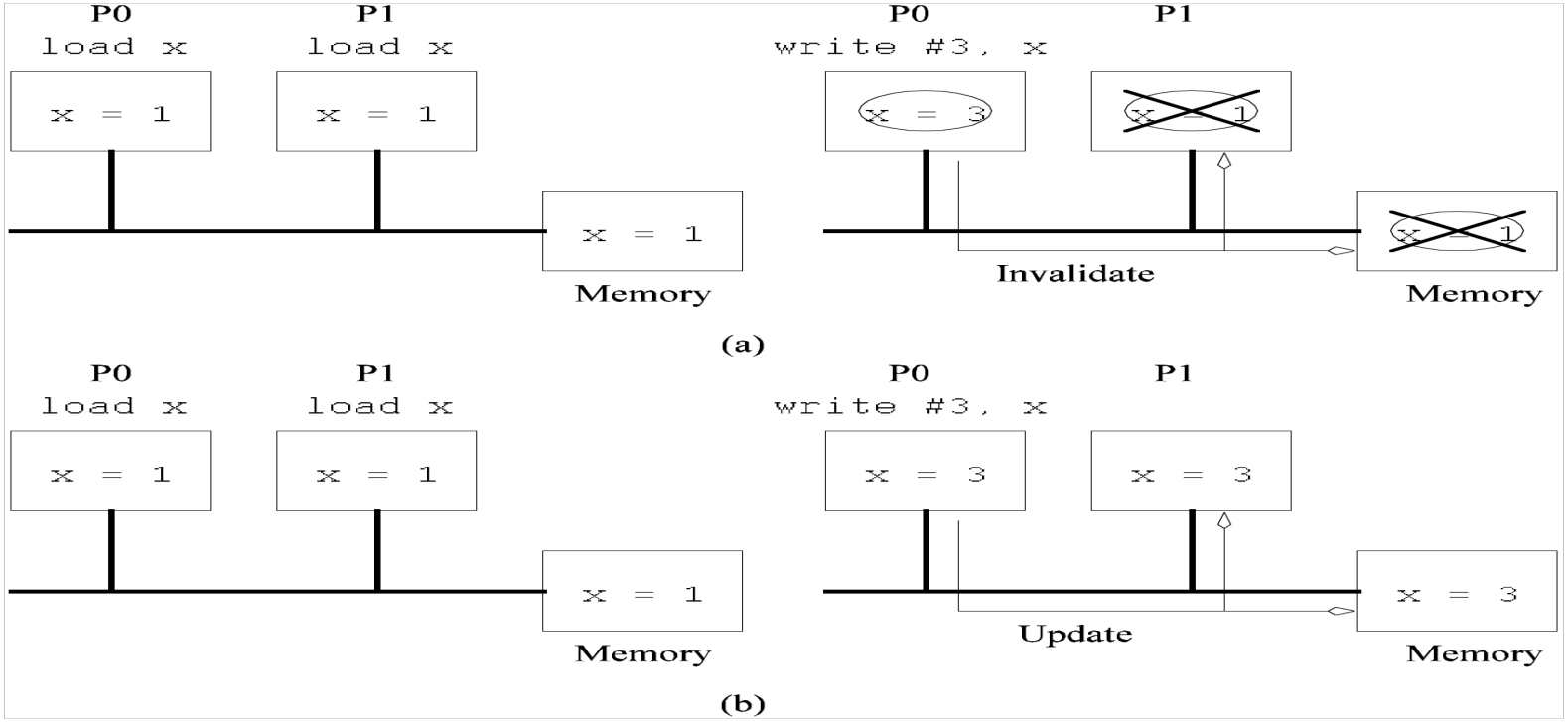
Evaluating Static Interconnection Networks

Table 2.1 A summary of the characteristics of various static network topologies connecting p nodes.

Network	Diameter	Bisection Width	Arc Connectivity	Cost (No. of links)
Completely-connected	1	$p^2/4$	$p-1$	$p(p-1)/2$
Star	2	1	1	$p-1$
Complete binary tree	$2\log((p+1)/2)$	1	1	$p-1$
Linear array	$p-1$	1	1	$p-1$
2-D mesh, no wraparound	$2(\sqrt{p}-1)$	$\sqrt{p}$	2	$2(p-\sqrt{p})$
2-D wraparound mesh	$2\lfloor\sqrt{p}/2\rfloor$	$2\sqrt{p}$	4	$2p$
Hypercube	$\log p$	$p/2$	$\log p$	$(p\log p)/2$
Wraparound k-ary d-cube	$d\lfloor k/2\rfloor$	$2k^{d-1}$	$2d$	dp

Lecture 3: Cache Coherence in Multiprocessor Systems

When the value of a cache line changes, all its copies must either be invalidated or updated.



Cache coherence in multiprocessor systems:
(a) Invalidate protocol; (b) Update protocol

Cache Coherence: Update and Invalidate Protocols

- If two processors make interleaved test and updates to a shared variable, then an *update* protocol is better
- If one processor performs multiple writes on a shared variable before it is accessed by another processor, then an *invalidate* protocol is better
- Most current machines use invalidate protocols.
- Both protocols suffer from *false sharing* overheads (two words that are not actually shared, but happen to lie on the same cache line).

Lecture 3 Review Question

- **Consider the following parallel computation**
  - **CPU 0: Update even-numbered rows**

```
for ( j = 0 ; j < N ; j += 2 )
    for ( k = 0 ; k < N ; k++ )
        A[j,k] = f(j,k);
```

  - **CPU 1: Update odd-numbered rows**

```
for ( j = 1 ; j < N ; j += 2 )
    for ( k = 0 ; k < N ; k++ )
        A[j,k] = g(j,k);
```

- **Assume that a cache line size of 32B, an invalidate protocol, an element size of 8B for A, and that A's first element is aligned on a cache line boundary. What is the maximum number of invalidate messages that will be sent between CPU 0 and 1 as a function of N?**

Recap of Lecture 4

- Tasks, Dependence Graphs, Scheduling Theory
  - —Task = indivisible sequential unit of computation
  - —Task Dependency Graph
  - —T_P = execution time on P processors
    - T_1 = sequential time = total work
    - T_∞ = critical path length
  - — Lower & upper bounds: $\max(T_1/P, T_\infty) \leq T_P \leq T_1/P + T_\infty$
- Decomposition Techniques
  - —recursive decomposition
    - data decomposition
    - exploratory decomposition
    - speculative decomposition

Lecture 4: Critical Path Length

- A directed path in the task dependency graph represents a sequence of tasks that must be processed one after the other.
- The longest such path determines the shortest time in which the program can be executed in parallel.
- The length of the longest path in a task dependency graph is called the *critical path length*.
- The ratio of the total amount of work to the critical path length is the *average degree of concurrency*.

Examples of Critical Path Length

Consider the task dependency graphs of the two database query decompositions:

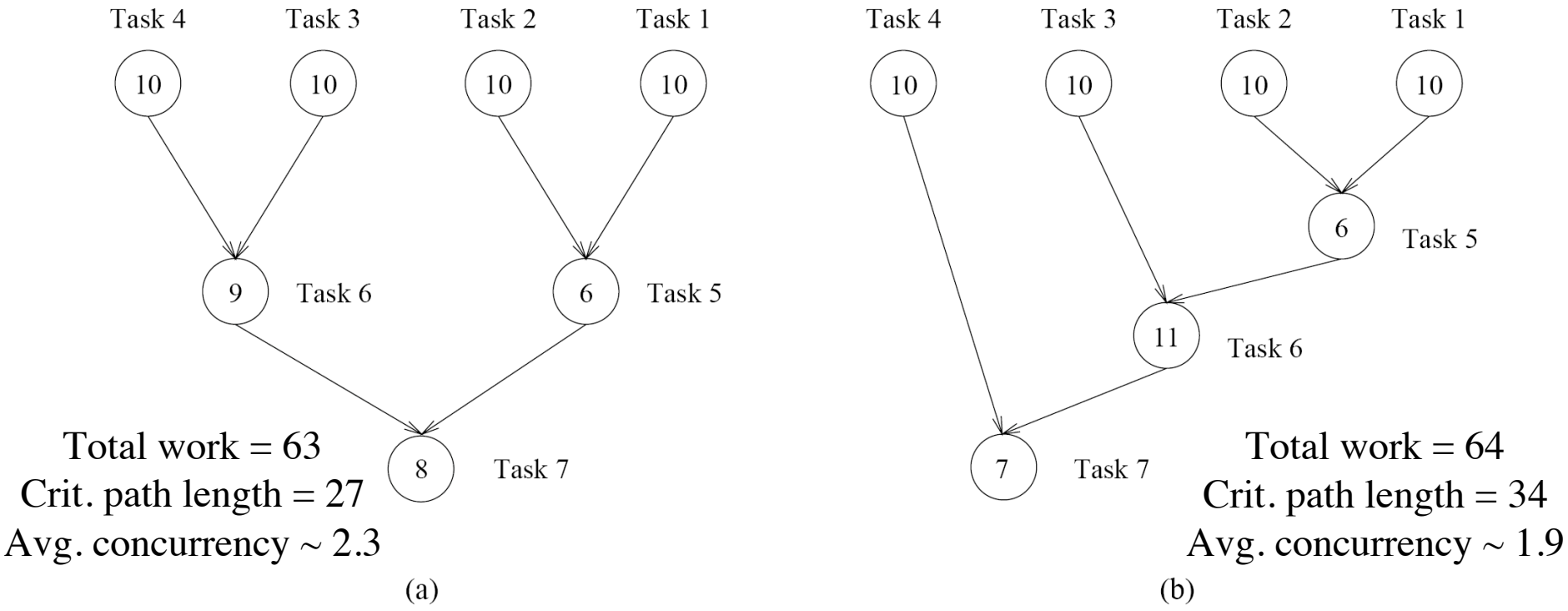


Figure 3.5 Abstractions of the task graphs of Figures 3.2 and 3.3, respectively.

Developing ★Socrates

- For the competition, ★Socrates was to run on a 512-processor Connection Machine Model CM5 supercomputer at the University of Illinois.
- The developers had easy access to a similar 32-processor CM5 at MIT.
- One of the developers proposed a change to the program that produced a speedup of over 20% on the MIT machine.
- After a back-of-the-envelope calculation, the proposed "improvement" was rejected!

★Socrates Speedup Paradox

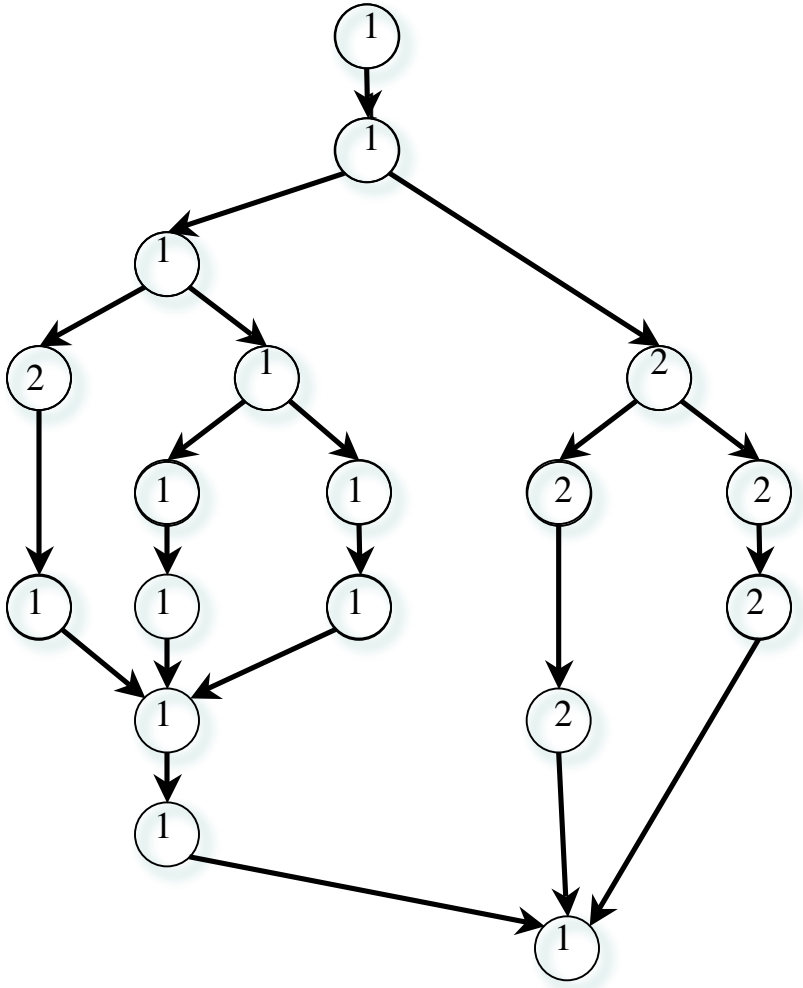
Original program	*Proposed program*
$T_{32} = 65$ seconds	$T'_{32} = 40$ seconds

$$T_P \approx T_1/P + T_\infty$$

$T_1 = 2048$ seconds	$T'_1 = 1024$ seconds
$T_\infty = 1$ second	$T'_\infty = 8$ seconds
$T_{32} = 2048/32 + 1$ $= 65$ seconds	$T'_{32} = 1024/32 + 8$ $= 40$ seconds
$T_{512} = 2048/512 + 1$ $= 5$ seconds	$T'_{512} = 1024/512 + 8$ $= 10$ seconds

Lecture 4 Review Question

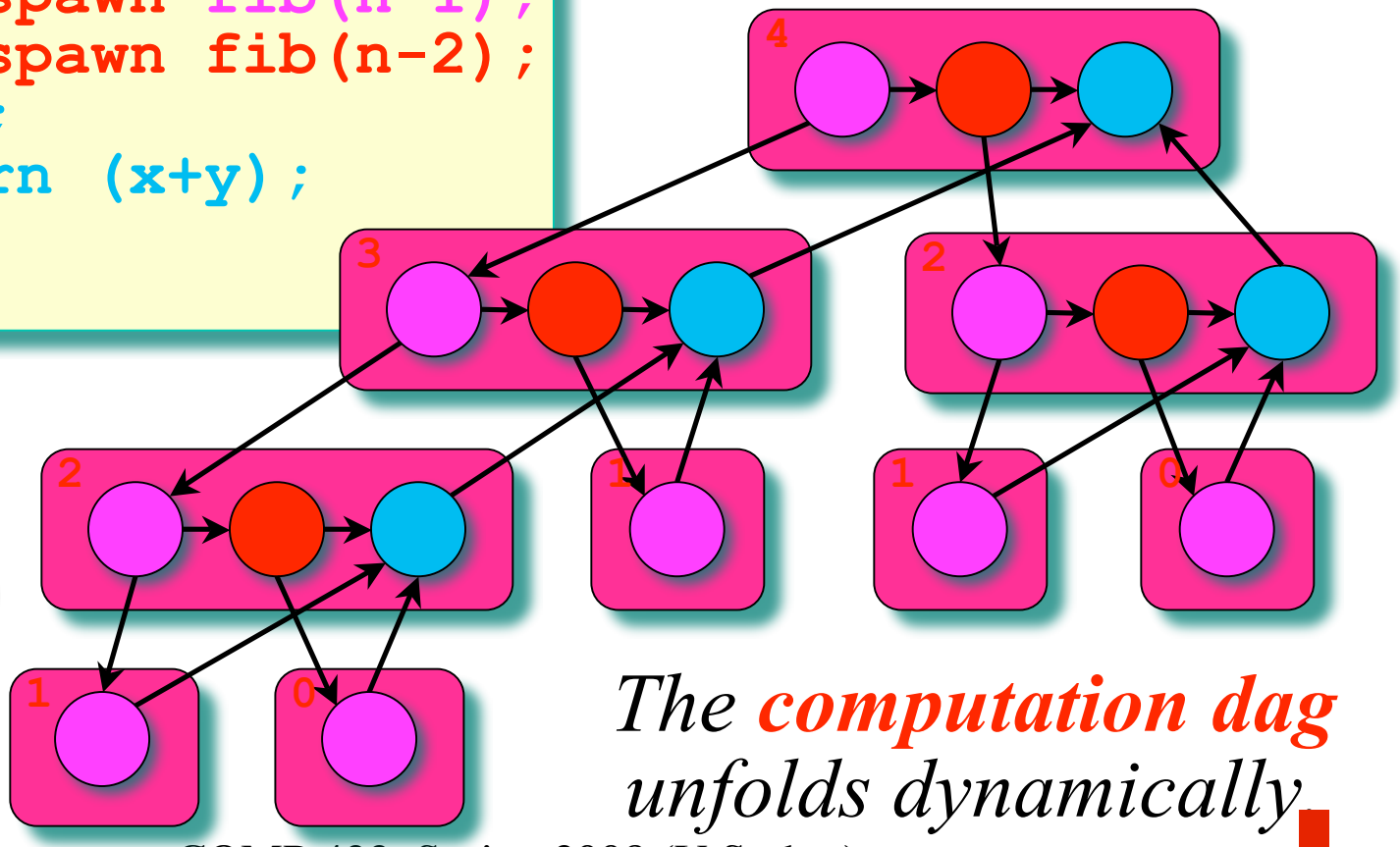
- Give lower and upper bounds on the parallel execution time of this task graph on
  - 2 processors, and
  - 100 processors



Lecture 5: Dynamic Multithreading in Cilk

```
cilk int fib (int n) {
  if (n<2) return (n);
  else {
    int x,y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return (x+y);
  }
}
```

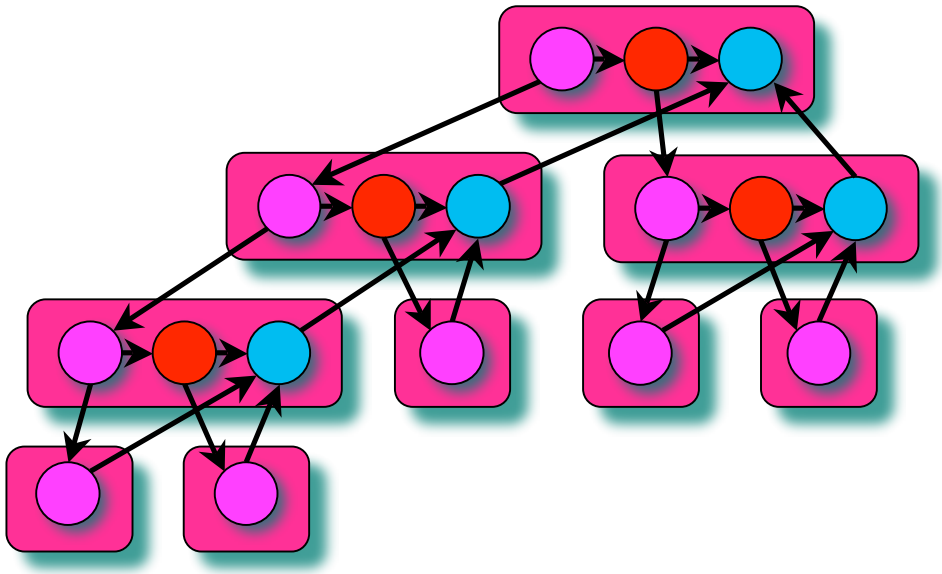
Example: `fib(4)`



"Processor oblivious"

The computation dag unfolds dynamically.

Example: `fib(4)`



Assume for simplicity that each Cilk thread in `fib()` *takes unit time to execute.*

Work: $T_1 = 17$

Span: $T_\infty = 8$

Parallelism: $T_1 / T_\infty = 2.125$

Using many more than 2 processors makes little sense.

Performance of Work-Stealing

Theorem: Cilk's work-stealing scheduler achieves an expected running time of

$$T_P \leq T_1/P + O(T_\infty)$$

on P processors.

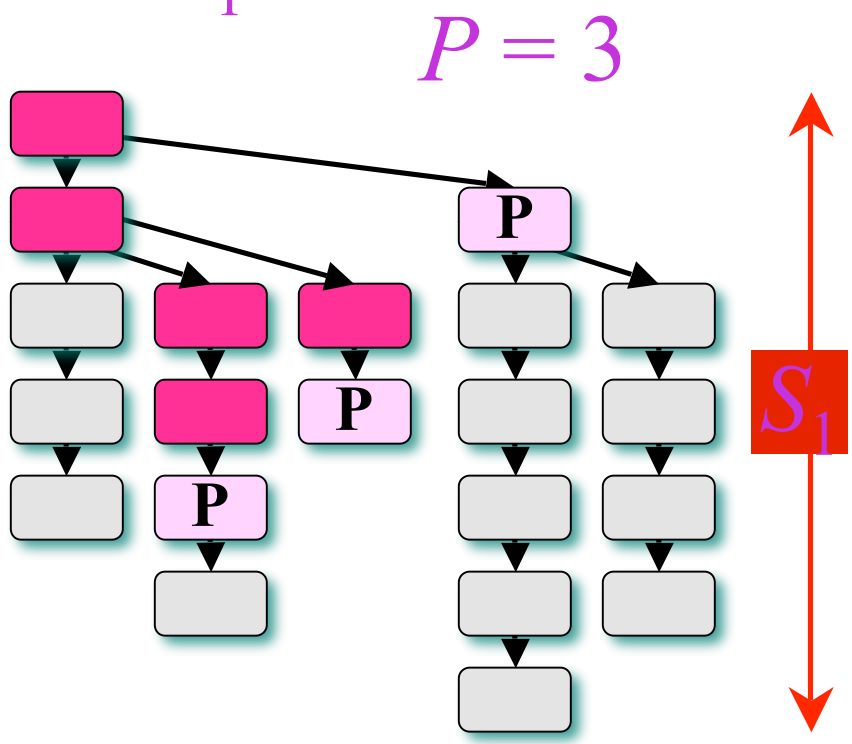
Pseudoproof. A processor is either ***working*** or ***stealing***. The total time all processors spend working is T_1. Each steal has a $1/P$ chance of reducing the span by 1. Thus, the expected cost of all steals is $O(P\,T_\infty)$. Since there are P processors, the expected time is

$$(T_1 + O(P\,T_\infty))/P = T_1/P + O(T_\infty) \;.\; \blacksquare$$

Space Bounds

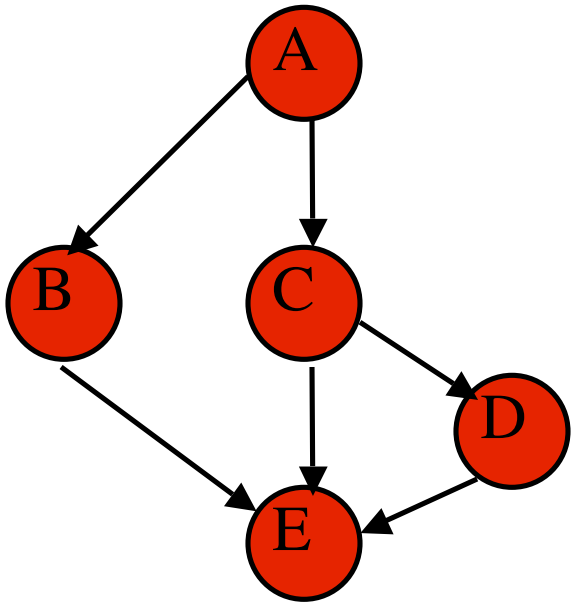
Theorem. Let S_1 be the stack space required by a serial execution of a Cilk program. Then, the space required by a P-processor execution is at most $S_P <= PS_1$.

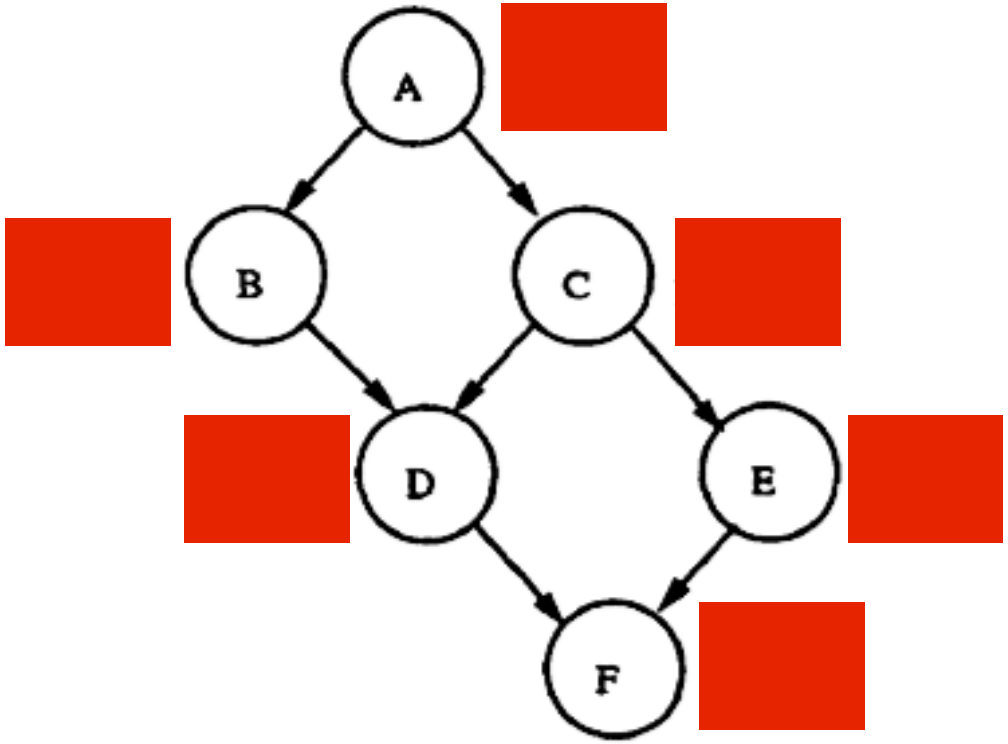
Proof (by induction). The work-stealing algorithm maintains the ***busy-leaves property***: *every extant procedure frame with no extant descendents has a processor working on it.* ■



Lecture 5 Review Question

- **Which of the following graphs can be realized as a computation dag using Cilk's async and spawn constructs only?**





Lecture 6: Semantics of Cilk Inlets

```
int max, ix = -1;
inlet void update ( int val, int index ) {
  if (idx == -1 || val > max) {
    ix = index; max = val;
  }
}
:
for (i=0; i<1000000; i++) {
  update ( spawn foo(i), i );
}
sync; /* ix now indexes the largest foo(i) */
```

- The **inlet** keyword defines a **void** internal function to be an inlet.
- In the current implementation of Cilk, the inlet definition may not contain a **spawn**, and only the first argument of the inlet may be spawned at the call site.

Semantics of Inlets

```
int max, ix = -1;
inlet void update ( int val, int index ) {
  if (idx == -1 || val > max) {
    ix = index; max = val;
  }
}
:
for (i=0; i<1000000; i++) {
  update ( spawn foo(i), i );
}
sync; /* ix now indexes the largest foo(i) */
```

1. The non-**spawn** args to **update()** are evaluated.
2. The Cilk procedure **foo(i)** is spawned.
3. Control passes to the next statement.
4. When **foo(i)** returns, **update()** is invoked.

Semantics of Inlets

```
int max, ix = -1;
inlet void update ( int val, int index ) {
  if (idx == -1 || val > max) {
    ix = index; max = val;
  }
}
:
for (i=0; i<1000000; i++) {
  update ( spawn foo(i), i );
}
sync; /* ix now indexes the largest foo(i) */
```

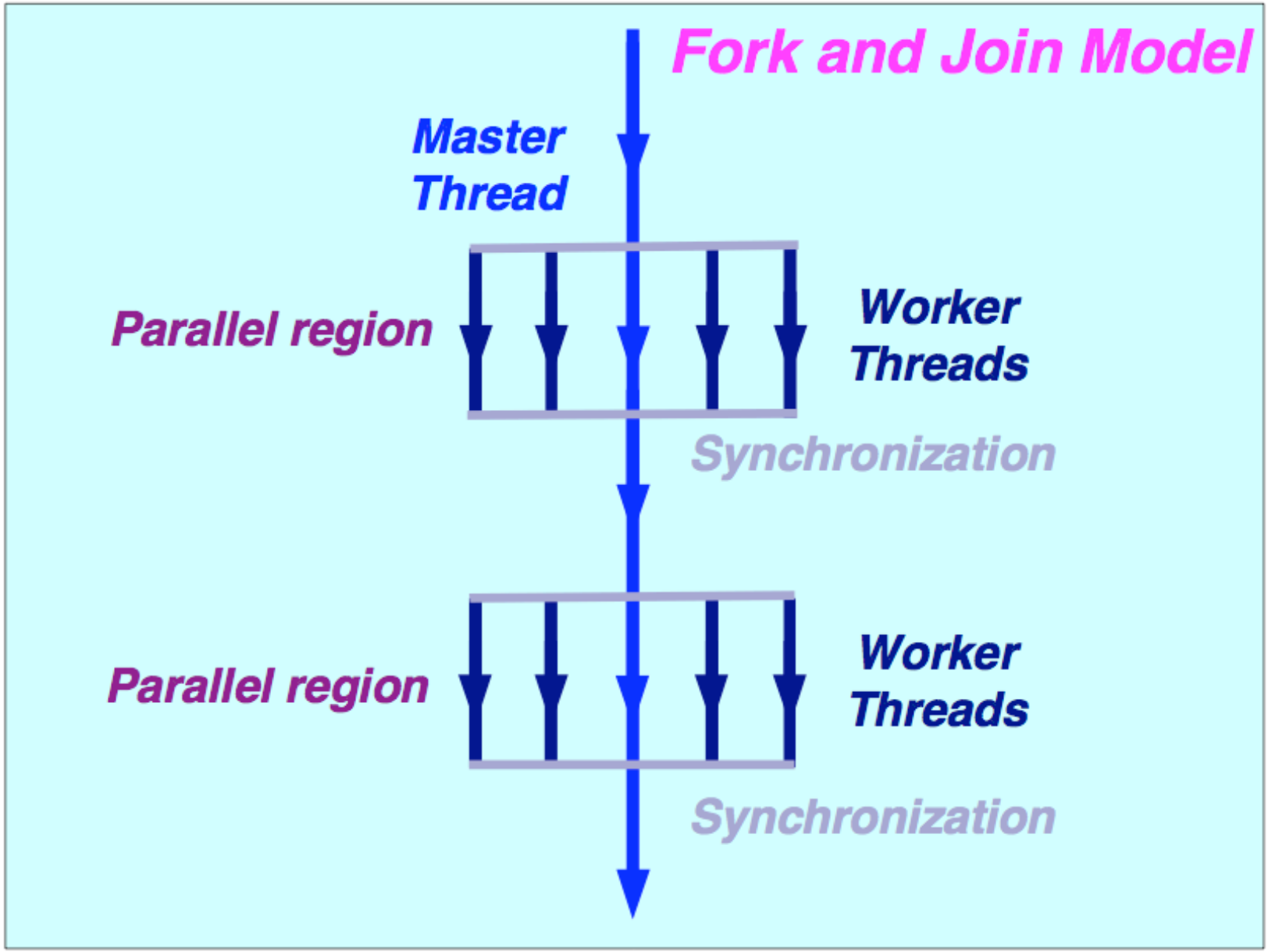
Cilk implicitly guarantees ***atomicity*** among the threads/tasks (including inlets) belonging to the same procedure instance, and thus no locking is necessary to avoid data races.

Implicit Inlets

```
cilk int wfib(int n) {
  if (n == 0) {
    return 0;
  } else {
    int i, x = 1;
    for (i=0; i<=n-2; i++) {
      x += spawn wfib(i);
    }
    sync;
    return x;
  }
}
```

For assignment operators, the Cilk compiler automatically generates an ***implicit inlet*** to perform the update.

Lecture 7: The OpenMP Execution Model



Parallel Region

```
#pragma omp parallel [clause[[,] clause] ...]
{
    "this is executed in parallel"

} (implied barrier)
```

A parallel region is a block of code executed by multiple threads simultaneously, and supports the following clauses:

if	*(scalar expression)*	
private	*(list)*	
shared	*(list)*	
default	*(none|shared)*	*(C/C++)*
default	*(none|shared|private)*	*(Fortran)*
reduction	*(operator: list)*	
copyin	*(list)*	
firstprivate	*(list)*	
num_threads	*(scalar_int_expr)*	

Work-sharing constructs in a Parallel Region

```
#pragma omp for
{
    ....
}
```

```
#pragma omp sections
{
        ....
}
```

```
#pragma omp single
{
        ....
}
```

- The work is distributed over the threads
- Must be enclosed in a parallel region
- Must be encountered by all threads in the team, or none at all
- No implied barrier on entry; implied barrier on exit (unless nowait is specified)
- A work-sharing construct does not launch any new threads
- Shorthand syntax supported for parallel region with single work-sharing construct e.g.,

```
#pragma omp parallel
#pragma omp for
   for (...)
```

```
#pragma omp parallel for
for (....)
```

Reduction Clause in OpenMP

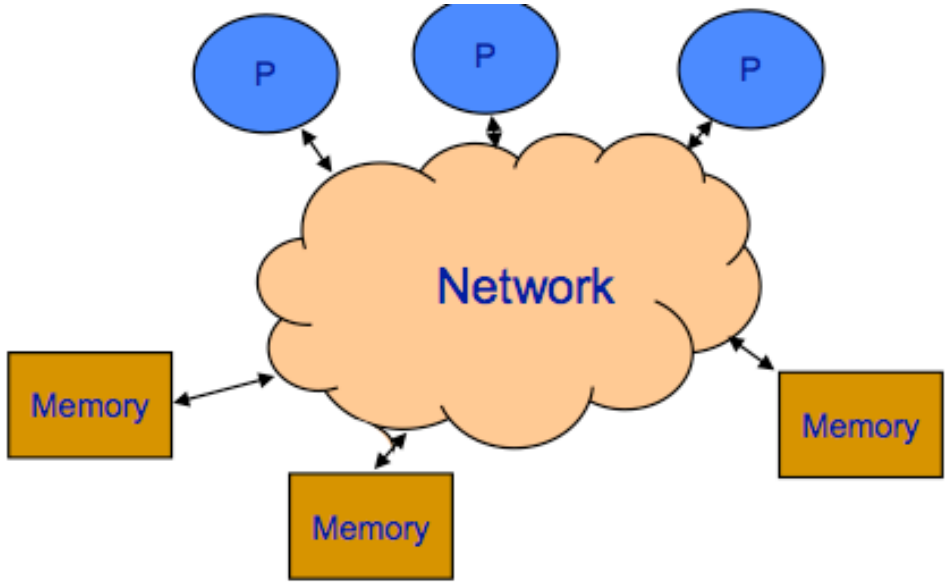
- The `reduction` clause specifies how multiple local copies of a variable at different threads are combined into a single copy at the master when threads exit.
- The usage of the `reduction` clause is `reduction (operator: variable list)`.
- The variables in the list are implicitly specified as being private to threads.
- The `operator` can be one of `+, *, -, &, |, ^, &&,` and `||`.

```
#pragma omp parallel reduction(+: sum) num_threads(8) {
/* compute local sums here */
}
/*sum here contains sum of all local instances of sums */
```

OpenMP Programming: Example

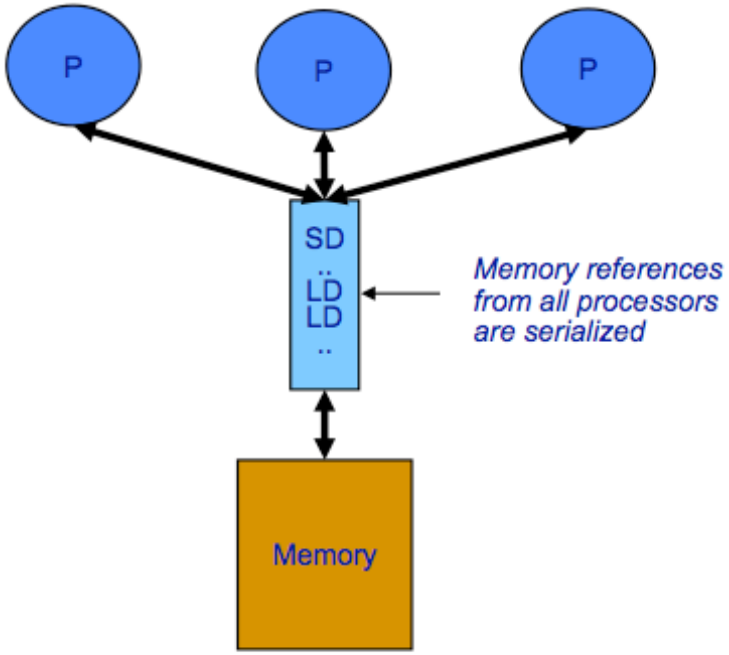
```
/* ************************************************
An OpenMP version of a threaded program to compute PI.
************************************************ */
#pragma omp parallel default(private) shared (npoints) \
    reduction(+: sum) num_threads(8)
{
    num_threads = omp_get_num_threads();
    sample_points_per_thread = npoints / num_threads;
    sum = 0;
    for (i = 0; i < sample_points_per_thread; i++) {
       rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
       rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
       if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
           (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
           sum ++;
    }
}
```

Lecture 8: Memory Consistency Models



- A memory consistency model →
  - A set of rules governing how the memory systems will process memory operations from multiple processors
  - Contract between the programmer and system
  - Determines what optimizations can be performed for correct programs

Sequential Consistency



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

Weak Ordering

- Weak ordering:
  - Divide memory operations into data operations and synchronization operations
  - Synchronization operations act like a fence:
    - All data operations before synch in program order must complete before synch is executed
    - All data operations after synch in program order must wait for synch to complete
    - Synchs are performed in program order
  - Hardware implementation of fence: processor has counter that is incremented when data op is issued, and decremented when data op is completed

OpenMP ordering vs. weak ordering

- **OpenMP re-ordering restrictions are analogous to weak ordering with "flush" identified as a "synch" op.**
- **But, it's weaker than weak ordering.**
  - **—Synchronization operations on disjoint variables are not ordered with respect to each other in OpenMP**

Relaxed memory model enables use of NUMA machines e.g., clusters and accelerators

Flush Is the Key OpenMP Operation

#pragma omp flush [(list)]

- **Prevents re-ordering of memory accesses across flush**
- **Allows for overlapping computation with communication**
- **A flush** construct with a list applies the flush operation to the items in the list,and does not return until the operation is complete for all specified list items.
  - —If a pointer is present in the list, the pointer itself is flushed, not the object to which the pointer refers
- A **flush** construct without a list, executed on a given thread, operates as if the whole thread-visible data state of the program, as defined by the base language, is flushed.

Re-ordering Example

```
a = ...;   //(1)
b = ...;   //(2)
c = ...;   //(3)

#pragma omp flush(c)    //(4)
#pragma omp flush(a,b)  //(5)

. . . a . . . b . . .; //(6)
. . . c . . .;         //(7)
```

(1) and (2) may not be moved after (5).

(6) may not be moved before (5).

(4) and (5) may be interchanged at will.

Moving data between threads

- To move the value of a shared var from thread a to thread b, do the following in exactly this order:
  - —Write var on thread a
  - —Flush var on thread a
  - —Flush var on thread b
  - —Read var on thread b

Implicit flushes

- In barriers
- At entry to and exit from
  - Parallel, parallel worksharing, critical, ordered regions
- At exit from worksharing regions (unless `nowait` is specified)
- In `omp_set_lock`, `omp_set_nest_lock`, `omp_set_nest_lock`, `omp_unset_nest_lock`
- In `omp_test_lock`, `omp_test_nest_lock`, if lock is acquired
- At entry to and exit from `atomic` - flush-set is the address of the variable atomically updated

Importance of variable list in flush

Incorrect example:

```
a = b = 0
```

thread 1

```
b = 1
flush(b)
flush(a)
if (a == 0) then
   critical section
end if
```

thread 2

```
a = 1
flush(a)
flush(b)
if (b == 0) then
   critical section
end if
```

Correct example:

```
a = b = 0
```

thread 1

```
b = 1
flush(a,b)
if (a == 0) then
   critical section
end if
```

thread 2

```
a = 1
flush(a,b)
if (b == 0) then
   critical section
end if
```

In-class Midterm Exam on 2/28/08

- Duration: 1 hour
- Weightage: 20%
- Three written questions to cover the following areas:
  - —Performance models for parallel algorithms and machines
    - Sections 2.2, 2.3, 2.4, 2.5, 3.1, 3.2
  - —Cilk
    - Cilk reference manual
  - —OpenMP
    - Section 7.10, OpenMP 2.5 specification