CSE502: Foundations of Parallel Programming

Lecture 12: Midterm Review

Vivek Kumar

Computer Science and Engineering

IIIT Delhi

vivekk@iiitd.ac.in

Multicores Saves Power

- Nowadays (post Dennard Scaling), power is proportional to $(\text{Frequency})^3$
- Baseline example: single 1GHz core with power P
  - Option A: Increase clock frequency to 2GHz
    - Power = 8P
  - Option B: Use 2 cores at 1 GHz each
    - Power = 2P
- Option B delivers same performance as Option A with 4x less power … provided software can be decomposed to run in parallel !!

Source:
https://wiki.rice.edu/confluence/download/attachments/4435861/comp322-s16-lec1-slides.pdf?version=1&modificationDate=1452732285045&api=v2

Floating Point Operations per Second (FLOPS)

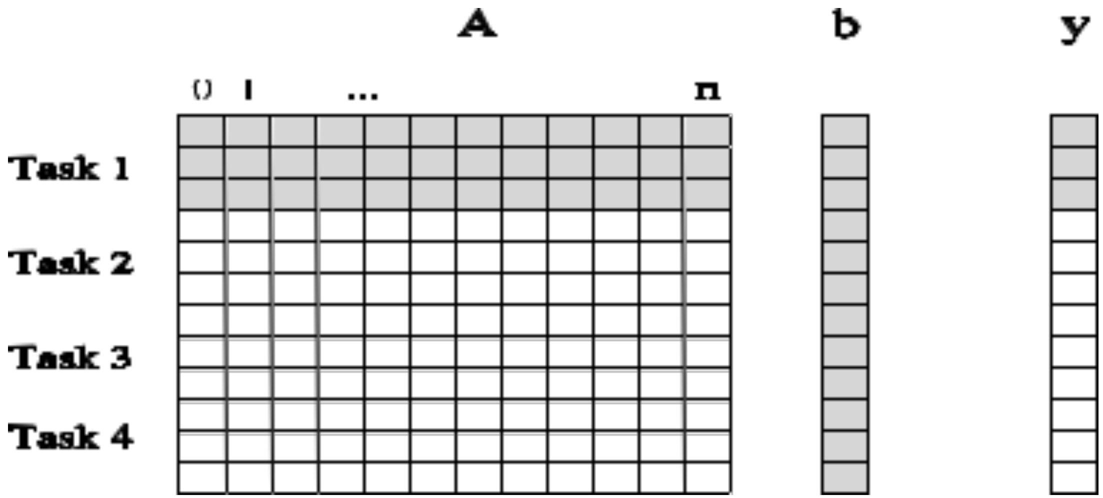
- Measure of computer performance in scientific computing
- FLOPS = (Total Cores) x (Clock) x (FLOPS per cycle)

Concurrency v/s Parallelism

- Concurrency
  - Refers to tasks that appear to be running simultaneously, but which may, in fact, actually be running serially
  - “**Dealing**” with lots of things together
- Parallelism
  - Refers to concurrent tasks that actually run at the same time
  - Always implies multiple processors
  - Parallel tasks always run concurrently, but not all concurrent tasks are parallel
  - “**Doing**” lots of things at once

Task Decomposition for Parallel Programming

- Granularity = task size
  - depends on the number of tasks
- Fine-grain = large number of tasks
- Coarse-grain = small number of tasks
- Granularity examples for dense matrix-vector multiply
  - fine-grain: each task represents an individual element in y
  - coarser-grain: each task computes 3 elements in y



Task Decomposition Techniques

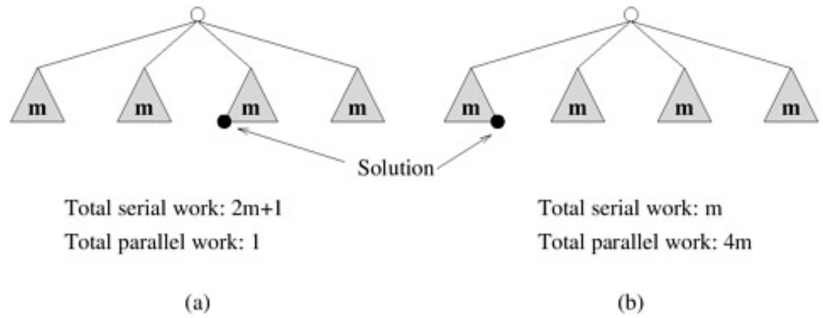
How should one decompose a task into various subtasks?

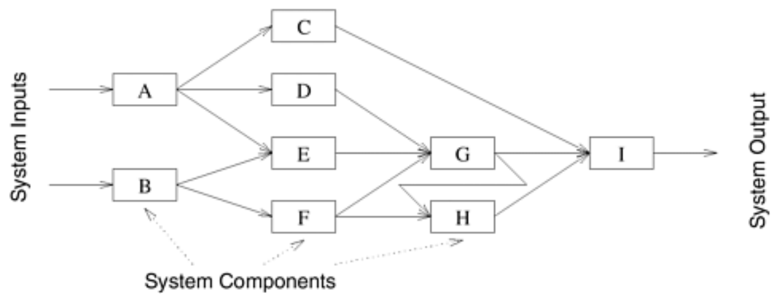
- No single universal recipe
- In practice, a variety of techniques are used including
  - Recursive decomposition
  - Data decomposition

Task Decomposition Techniques

How should one decompose a task into various subtasks?

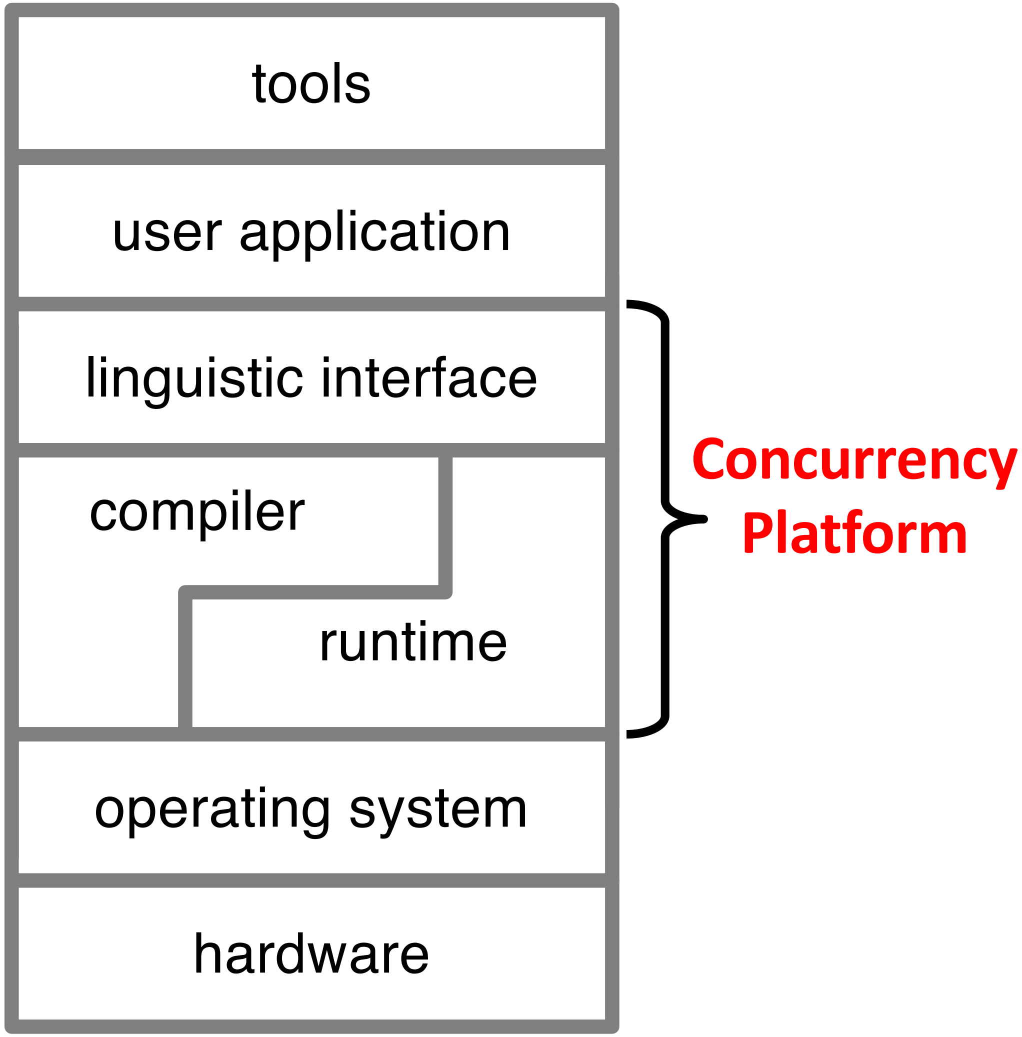
- No single universal recipe
- In practice, a variety of techniques are used including
  - Recursive decomposition
  - Data decomposition
  - **Exploratory decomposition**
  - **Speculative decomposition**





Static v/s Dynamic mapping??

Concurrency Platforms



A concurrency platform should provide:

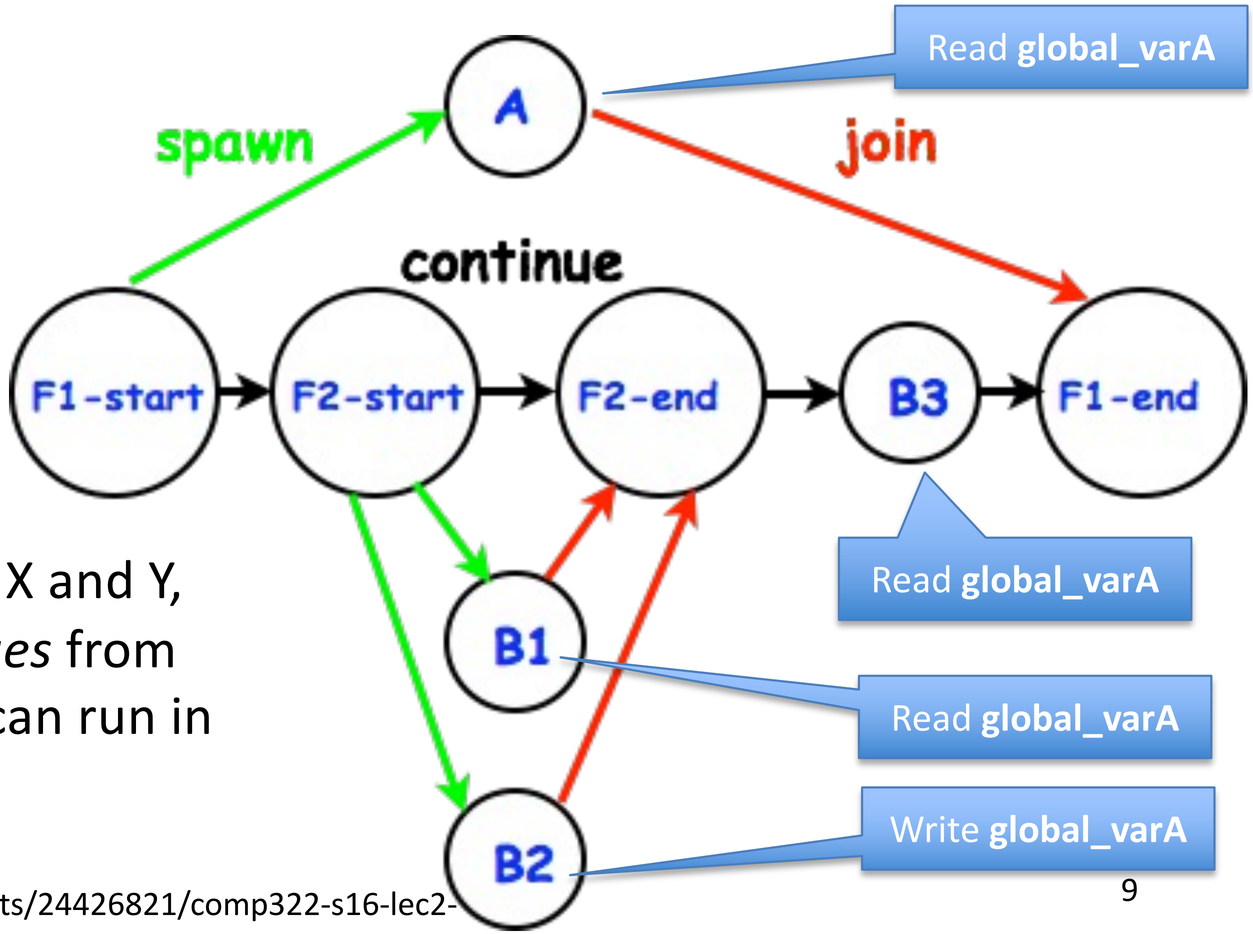
- an interface for specifying the ***logical parallelism*** of the computation;
- a runtime layer to automate scheduling and synchronization; and
- guarantees of performance and resource utilization competitive with hand-tuned code.

Source: http://classes.engineering.wustl.edu/cse539/web/lectures/lec01_intro.pdf

Async and Finish Statements for Task Creation and Termination & Data Races

```
finish {            // F1
  async A;
  finish {          // F2
    async B1;
    async B2;
  }                 // F2
  B3;
}                   // F1
```

Computation Graph

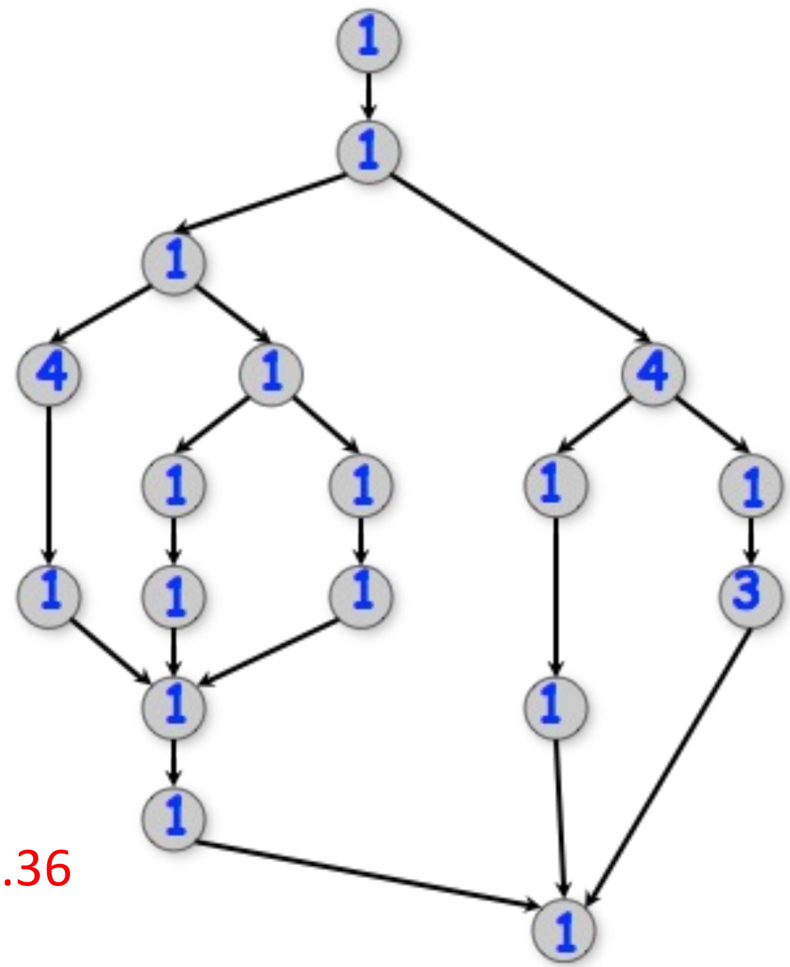


<u>Key idea</u>: If two statements, X and Y, have *no path of directed edges* from one to the other, then they can run in parallel with each other.

Source: https://wiki.rice.edu/confluence/download/attachments/24426821/comp322-s16-lec2-slides-v1.pdf?version=1&modificationDate=1483206145211&api=v2

Ideal Parallelism

- Define ideal parallelism of Computation G Graph as the ratio, WORK(G)/CPL(G)
- Ideal Parallelism only depends on the computation graph, and is the speedup that you can obtain with an unbounded number of processors



Example:

WORK(G) = 26

CPL(G) = 11

Ideal Parallelism = WORK(G)/CPL(G) = 26/11 ~ 2.36

Source:
https://wiki.rice.edu/confluence/download/attachments/24426821/comp322-s16-lec2-slides-v1.pdf?version=1&modificationDate=1483206145211&api=v2

Greedy Schedule

- A greedy schedule is one that never forces a processor to be idle when one or more nodes are ready for execution
- A node is **ready** for execution if all its predecessors have been **executed**
- Observations
  - T_1 = WORK(G), for all greedy schedules
  - T_∞ = CPL(G), for all greedy schedules
- where T_P = execution time of a schedule for computation graph G on P processors

Source: https://wiki.rice.edu/confluence/download/attachments/24426821/comp322-s16-lec3-slides-v1.pdf?version=1&modificationDate=1483206145596&api=v2

Bounds on Execution Time of Greedy Schedules

- Let T_P = execution time of a schedule for computation graph G on P processors
  - Can be different for different schedules
- Lower bounds for all greedy schedules
  - Capacity bound: $T_P \geq WORK(G)/P$
  - Critical path bound: $T_P \geq CPL(G)$
  - Putting them together
    - $T_P \geq \max(WORK(G)/P, CPL(G))$
- Upper bounds for all greedy schedules
  - Theorem [Graham '66]. Any greedy scheduler achieves
    - $T_P \leq WORK(G)/P + CPL(G)$

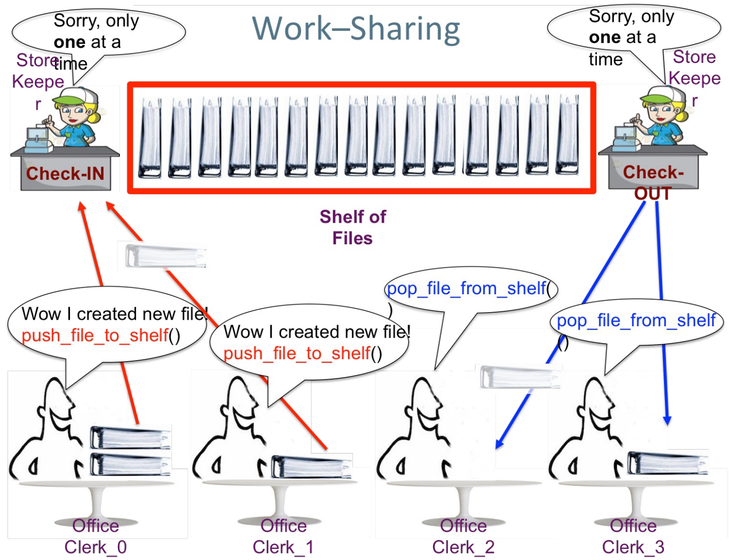
Source: https://wiki.rice.edu/confluence/download/attachments/24426821/comp322-s16-lec3-slides-v1.pdf?version=1&modificationDate=1483206145596&api=v2

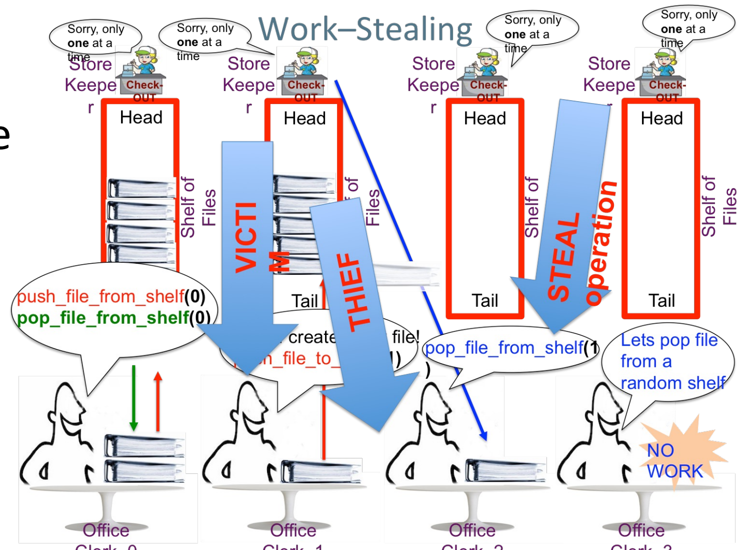
Greedy Scheduling using Thread Pool

- Task scheduling paradigms
  - Work-sharing scheduling
  - Work-stealing scheduling

Work-Sharing v/s Work-Stealing

- Work-sharing
  - Busy worker re-distributes the task eagerly
  - Easy implementation through global task pool
  - Access to the global pool needs to be synchronized: scalability bottleneck
- Work-stealing
  - Busy worker pays little overhead to enable stealing
    - A lock is required for pop and steal only in case single task remaining on deque
  - Distributed task pools
    - Idle worker steals the tasks from busy workers
  - Better scalability





Types of Work-Stealing

```
1. finish {
2.     async S1;
3.     //continuation of S1
4.     async S2;
5.     //continuation of S2
6.     S3;
7. }
```

Work-first

With single worker, program execution using work-first policy is similar to serial execution

```
start_finish();
push_task_to_runtime(Line_3);
S1;
if(Line_3_stolen) return;
push_task_to_runtime(Line_5);
S2;
if(Line_5_stolen) return;
S3;
end_finish();
```

Help-first

```
start_finish();
push_task_to_runtime(S1);
push_task_to_runtime(S2);
S3;
end_finish();
```


Source: Work-first and help-first scheduling policies for async-finish task parallelism, Guo et. al., IPDPS 2009

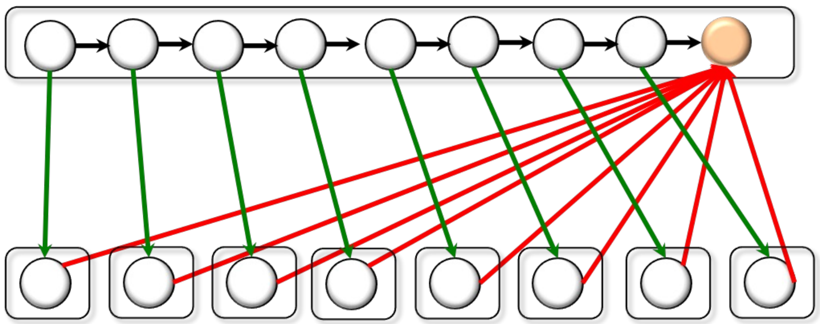
FORASYNC_MODE_FLAT

```
void foo() {
  loop_domain_t loop = {0, 8, 1, 1};
  finish([&]() {
    forasync1D (&loop, [=](int i) {
      S(i); // can execute in parallel for all i
    }, MODE);
  });
}
```

MODE= FORASYNC_MODE_FLAT

Work = O(n)

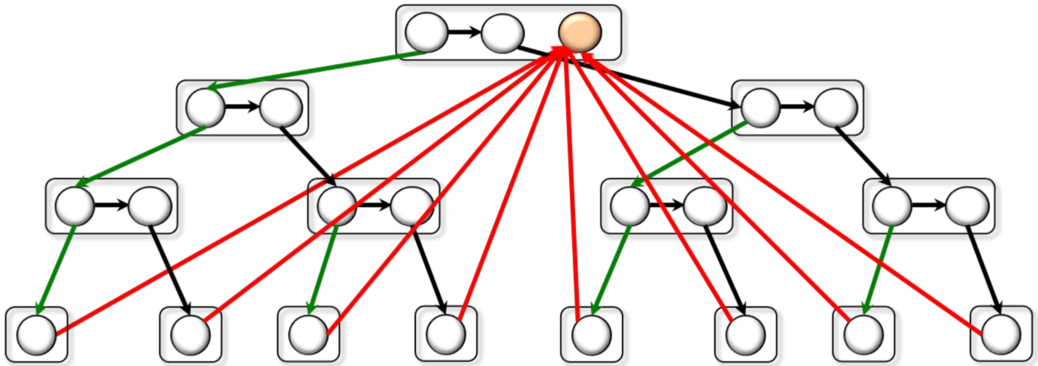
CPL = O(n)



MODE= FORASYNC_MODE_RECURSIVE

Work = O(n)

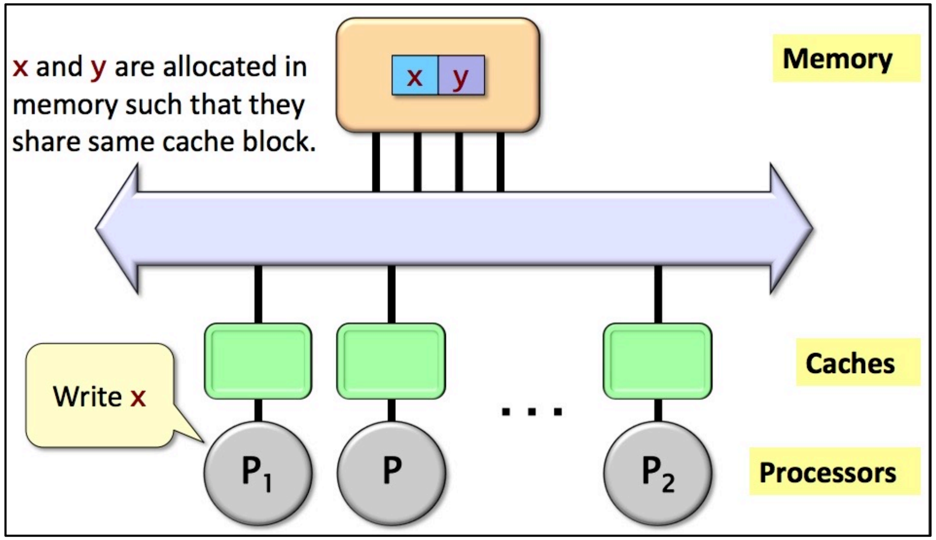
CPL = O(log n)

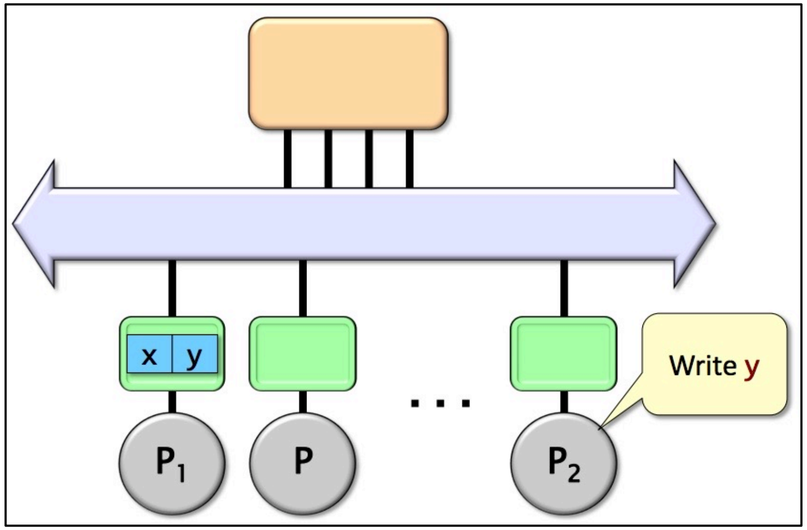


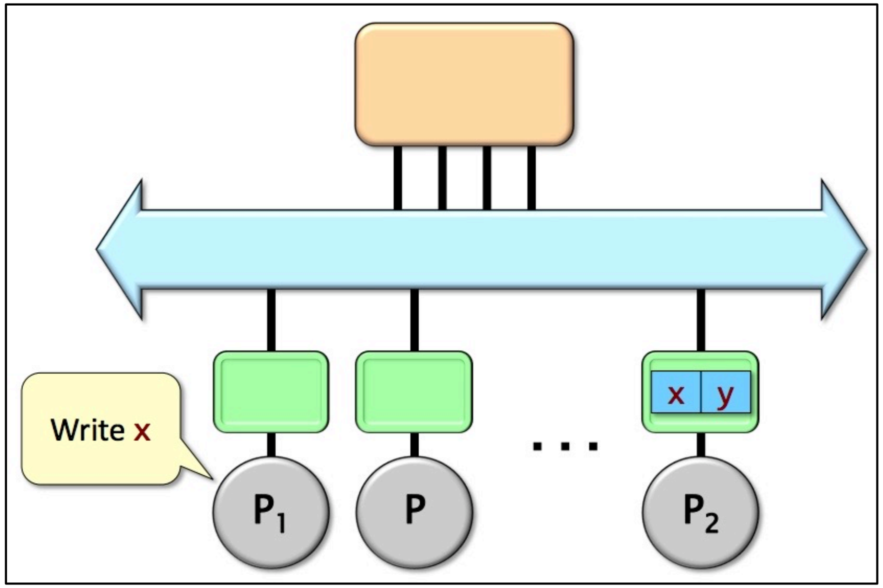
Source: http://classes.engineering.wustl.edu/cse539/web/lectures/lec02_parallelism.pdf

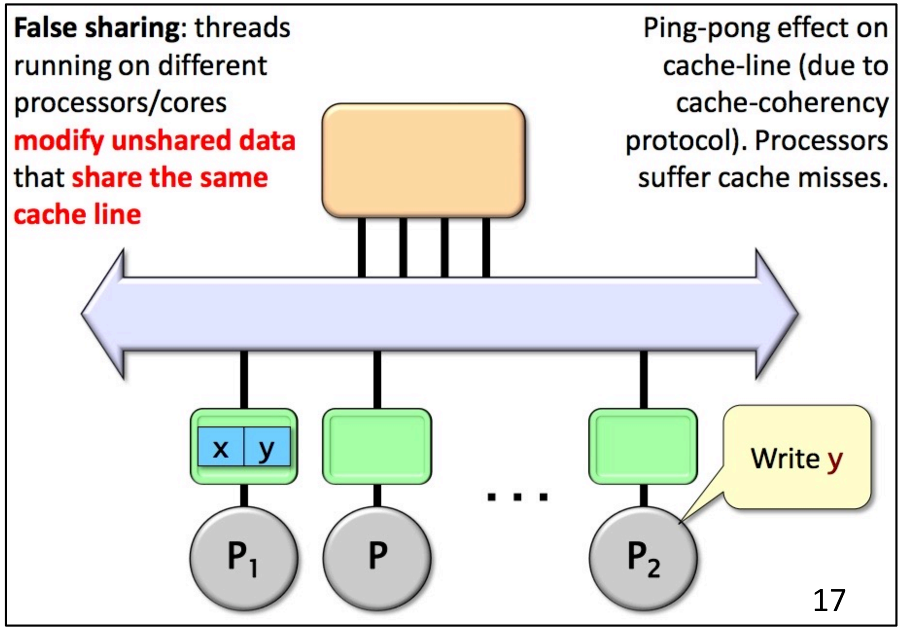
False Sharing due to Cache Coherency

False sharing occurs when threads on different processors modify variables that reside on the same cache line. This invalidates the cache line and forces a memory update to maintain cache coherency









Properties of a Good Locking Algorithm

- Mutual exclusion
- ***Deadlock freedom:*** *system as a whole makes progress.*
  If some thread calls **lock()** and never returns, then other threads must complete **lock()** and **unlock()** calls infinitely often.
- ***Starvation freedom:*** *individual thread makes progress. (This implies deadlock freedom.)*
  If some thread calls **lock()**, it will eventually return.

Acknowledgement: Slides adopted from the companion slides for the book "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit

Object Based Isolation

isolated(obj1, obj2, …, lambda_function)

- In this case, programmer specifies list of objects for which isolation is required
- Mutual exclusion is only guaranteed for instances of isolated constructs that have a common object in their object lists
  - Standard isolated is equivalent to "isolated(*)" by default i.e., isolation across all objects

Pros and Cons of Object Based Isolation

- Pros
  - Increases parallelism relative to critical section approach
  - Simpler approach than “locks”
  - Deadlock-freedom property is still guaranteed
- Cons
  - Programmer needs to worry about getting the object list right

Source: https://wiki.rice.edu/confluence/download/attachments/24426821/comp322-s16-lec20-slides-v1.key.pdf?version=1&modificationDate=1483206145246&api=v2

- Mid semester exam