Concurrent or Not Concurrent?

Kenneth M. Anderson
University of Colorado, Boulder

CSCI 5828 — Lecture 6 — 01/28/2010

Lecture Goals

- Review material in Chapter 1 and 2 of the Breshears textbook
  - Threading Methodologies
  - Parallel Algorithms (Intro)
  - Shared-Memory vs. Distributed Memory Programming
  - Design Models for Concurrent Algorithms
    - Task Decomposition
    - Example

Threading Methodology

- Breshears presents a threading methodology
  - First produce a tested single-threaded program
    - Use reqs./design/implement/test/tune/maintenance steps
  - Then to create a concurrent system from the former, do
    - Analysis: Find computations that are independent of each other
      - AND take up a large amount of serial execution time (80/20 rule)
    - Design and Implement: straightforward
    - Test for Correctness: Verify that concurrent code produces correct output
    - Tune for performance: once correct, find ways to speed up

Note: does not recommend going straight to concurrency!

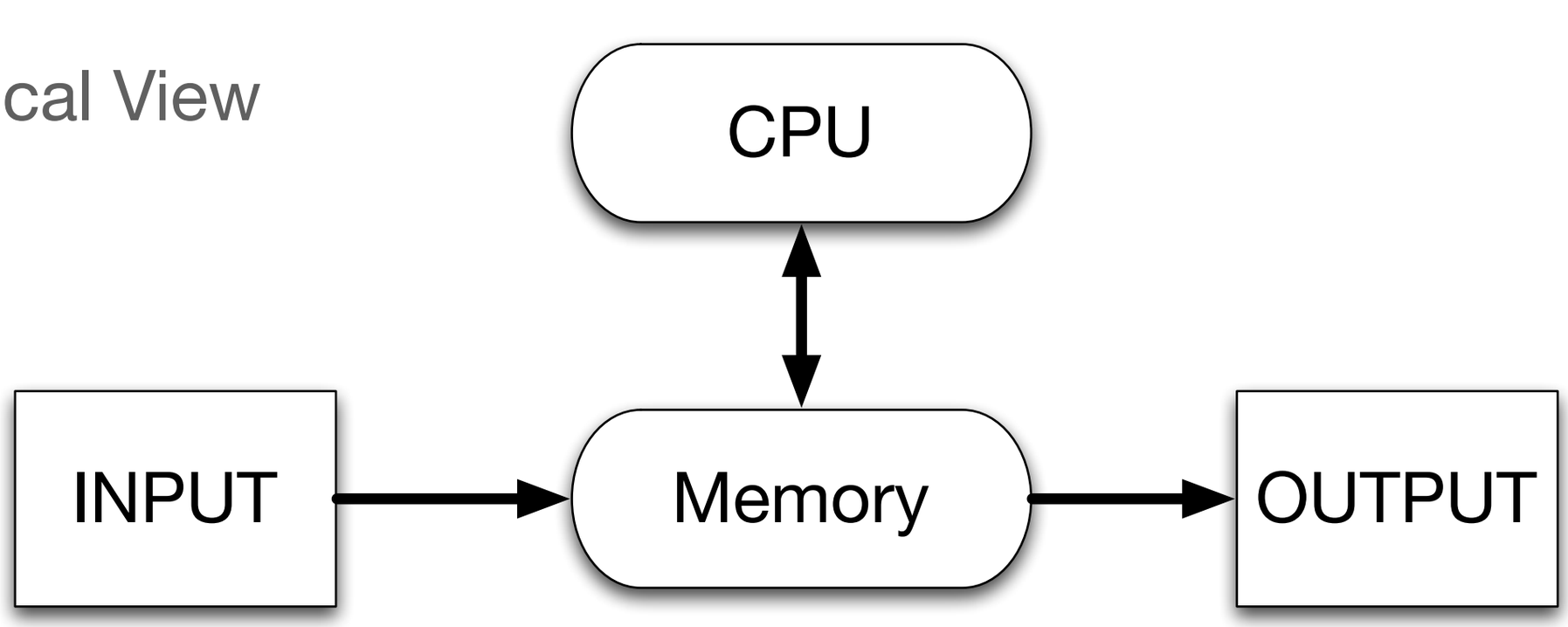
Performing Tuning

- Tuning threaded code typically involves
  - identifying sources of contention on locks (synchronization)
  - identifying work imbalances across threads
  - reducing overhead
- Testing and Tuning
  - Whenever you tune a threaded program, you must test it again for correctness
- Going back further: if you are unable to tune system performance, you may have to re-design and re-implement

Parallel Algorithms (Intro)

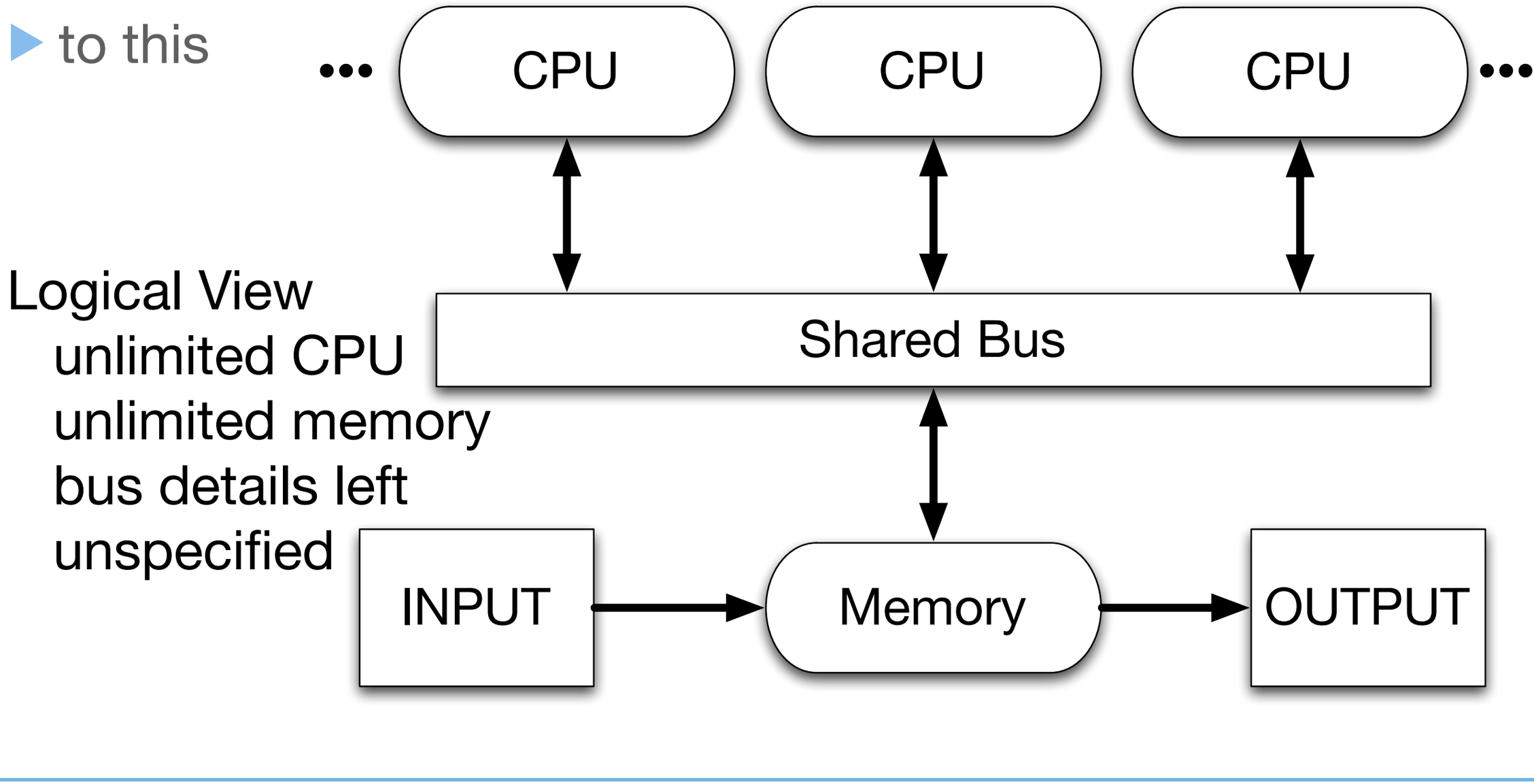
- In looking at the development of parallel algorithms, the standard Von Neumann architecture is modified, from this

Logical View



Parallel Algorithms (Intro)

- to this



Logical View
- unlimited CPU
- unlimited memory
- bus details left unspecified

Memory Access

- What is specified, is the types of memory access allowed by the threads running on the CPUs
  - Concurrent Read / Concurrent Write ; no restrictions
  - Concurrent Read / Exclusive Write
  - Exclusive Read / Concurrent Write
  - Exclusive Read / Exclusive Write
- It is up to algorithm to enforce the chosen model including how to resolve two or more writes to the same location
  - random or sum of values, for example

Number of CPUs

- Typically specified in terms of N where N is tied to the input in some way
- As with single threaded algorithms, once you have the specification for a parallel algorithm you must
  - figure out how to map it to your specific machine / language

Shared vs. Distributed

- When building large, concurrent systems you will run into issues of whether to use a distributed vs. shared memory model
  - Due to issues related to the shared bus, a limit of 32 processors was hit in the early 90s for parallel computers making use of shared memory algorithms
    - To address this and to scale concurrency to more processors, distributed memory configurations were utilized
      - Key difference; now you can accrue overhead by work that copies values between threads on one machine to threads on another machine

Shared vs. Distributed

- Common Features between Shared and Distributed
  - Redundant Work
    - To avoid sharing overhead, sometimes tasks perform the same calculations; as opposed to having one task compute that value once and share with all other threads
      - In this case, each task is performing redundant work
  - Shared Data
    - Tasks may sometimes share data; depending on the semantics of the application that sharing has to be protected with synchronization objects which adds overhead; or overhead is incurred when shared data is copied between machines

Shared vs. Distributed

- Common Features between Shared and Distributed
  - Static vs. Dynamic Allocation of Work
    - Some programs will have a clear mapping of tasks to threads
      - The mapping can be assigned statically (within the source code) and the same thread will perform the same type of work each time the program is run
      - In situations, where tasks cannot be predicted ahead of time or there are simply way more tasks than threads, a dynamic allocation strategy is employed
        - simple manifestation: jobs placed on synchronous queue; threads block on the queue waiting for jobs to arrive

Shared Only

- Characteristics of Shared Memory Parallelism
  - Local declarations and thread-local storage
    - Not all variables are shared among threads
    - Indeed, you want to minimize the number of shared vars
      - We saw an example of thread-local storage last week in Ruby
  - Memory Effects
    - Threads can interact with cache in detrimental ways
      - Sharing can reduce the amount of cache available to each thread or two threads hitting the same cache line for different values can trigger poor performance of the cache

Shared Only

- Characteristics of Shared Memory Parallelism
  - Communication in Memory
    - To share data between threads, one thread writes to a variable and the other thread reads to it; without careful design or synchronization objects, reads and writes can occur in unpredictable orders leading to incorrect output
    - Demo
      - Example makes use of Ruby's distributed programming framework, Rinda, an implementation of Linda tuple spaces
      - Consists of three pieces: a Rinda server, a service and a client with ten threads
      - Note: in this example, the tuple space becomes the shared mem.

Shared Only

- Characteristics of Shared Memory Parallelism
  - Mutual Exclusion
    - In shared memory situations, if you have multiple readers / writers accessing the same value, you will need to ensure that only one of those threads can update the value at a time
      - and that no thread can read the value while the update occurs)
  - We saw this in the previous example
    - Our use of ring_server.read vs ring_server.take amounted to the non-use and use (respectively) of mutual exclusion semantics
      - In that example, the correct result (100) was only achieved when mutual exclusion was achieved

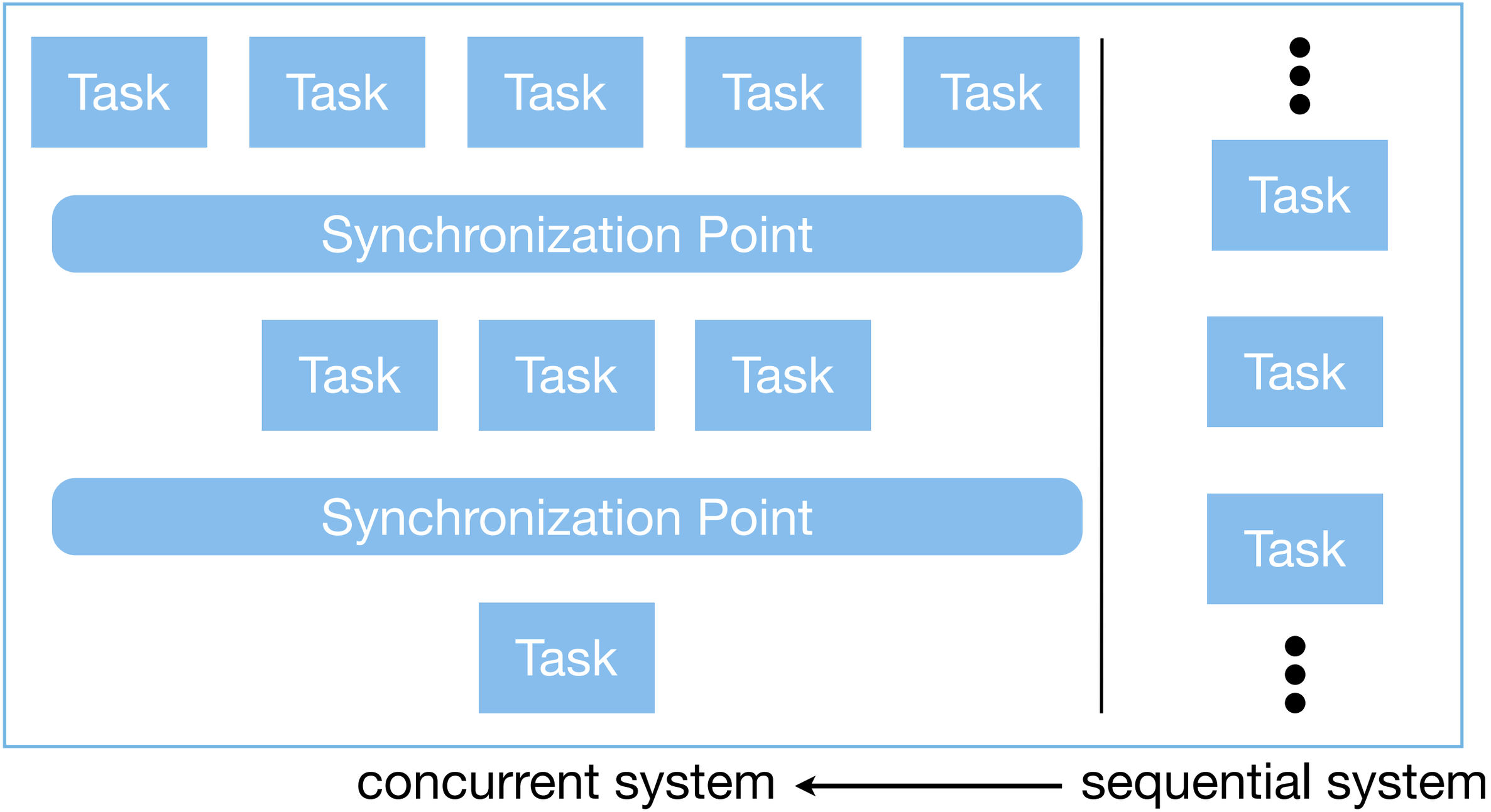
Shared Only

- Characteristics of Shared Memory Parallelism
  - Producer/Consumer
    - A common approach to work allocation
    - Have small set of producers generating tasks
    - Have larger set of consumers taking tasks and executing them
    - In between have a shared queue that uses synchronization to prevent the queue from being corrupted
  - Reader/Writer Locks
    - A variant of mutual exclusion that allows threads to declare whether they only read or only write a value; enforces sync. such that lots of reads can occur simultaneous but only everyone (both readers and writers) are blocked when a write occurs

Design Models

- Two primary design models for concurrent algorithms
  - Task Decomposition
    - identify tasks (computations) that can occur in any order
    - assign such tasks to threads and run concurrently
  - Data Decomposition
    - program has large data structures where individual data elements can largely be calculated independently
    - data decomposition implies task decomposition in these cases

Task Decomposition



Task Decomposition

- As shown on the previous slide
  - to convert a sequential system into a concurrent system
  - you need to identify sequential pieces of work that can run independent of one another
    - this can be hard to do, as tasks will have dependencies
      - a makes use of info generated by b
      - a and b both read/modify a value created by c
      - etc.
- Ultimate Goal: sequential consistency: concurrent program must produce the same answer as the sequential version

Task Decomposition

- Basic Framework
  - main thread
    - defines/prepares tasks
    - spawns threads
    - assigns tasks to threads
    - wait for threads to complete: threads.each { |t| t.join }
    - repeat until done
- Lots of variations
  - for instance, don’t kill threads, keep them around so you don’t have to spawn new ones the second time around

Task Decomposition

- Three Key Elements
  - What are the tasks and how are they defined?
  - What are the dependencies between tasks and how can they be satisfied?
  - How are tasks assigned to threads?

Finding Tasks…

- There is no magic formula
  - "You need to be able to mentally simulate the execution of two parallel streams on suspected parts of the application to determine whether those suspected parts are independent of each other (or might have manageable dependencies)."
    - One exception:
      - if you find a loop, see if you can execute it backwards and achieve the same result
        - if so, you might be able to create one task per loop iteration
- But there are heuristics to guide your work…

Finding Tasks…

- Use profiler to identify “hotspots” in the sequential program
  - if a program spends 80% of its execution time in one section of the program, and you can parallelize that section, you might see big performance gain: “biggest bang for the buck”
- Any decomposition must meet two criteria
  - There should be at least as many tasks as threads (cores)
  - The amount of computation within each task (granularity) must be large enough to offset the overhead needed to manage tasks and threads
    - Goal: More tasks than threads with high granularity

Dependencies

- Order Dependencies
  - Task A must be completed before Task B
    - Schedule A and B on the same thread, with A coming first
    - If that's not possible, then introduce synchronization object
- Data Dependencies
  - Two tasks are writing to the same variable or the potential exists for one or more tasks to read a variable while it is being updated by other tasks
  - Data dependencies can be harder to eliminate

Dependencies

- Addressing data dependencies
  - Sometimes a dependence can be broken by having a dependent task calculate the required value itself (redundant work) using local variables or thread-specific storage
  - If these techniques do not work, then you will have to use synchronization objects (aka locks) to ensure that a result is generated correctly
    - biggest problem here is that adding locks cannot be done without impacting performance and reasoning about locks in a system can be VERY hard to get right

Assigning Tasks

- Static Scheduling vs. Dynamic Scheduling
  - Discussed previously… (slide 11)

Food for Thought

```
1  static long num_rects = 100000;
2
3  void main() {
4    int i;
5    double mid, height, width, sum = 0.0;
6    double area;
7
8    width = 1.0/(double)num_rects;
9    for (i = 0; i < num_rects; i++) {
10     mid = (i + 0.5) * width;
11     height = 4.0/(1.0 + mid*mid);
12     sum += height;
13   }
14
15   area = width * sum;
16   printf("Computed pi = %f\n", area);
17 }
18
```

How to decompose?

- What should we do to perform task decomposition on the example on the previous slide?
- Demo
  - Ruby solution
    - proves that ruby 1.9.x is providing concurrency
  - Java solution
    - proves that java 1.6.x is providing parallelism

Wrapping Up

- Concepts
  - Threading Methodologies
  - Parallel Algorithms (Intro)
  - Shared-Memory vs. Distributed Memory Programming
  - Design Models for Concurrent Algorithms
    - Task Decomposition
- For next time
  - Data Decomposition and What's Not Parallel

Coming Up Next

- Lecture 7: Project Planning
  - Chapter 3 of Pilone & Miles
- Lecture 8: Proving Correctness and Measuring Performance
  - Remainder of Chapter 2 Topics
  - Chapter 3 of Breshears