Proving Correctness

Kenneth M. Anderson
University of Colorado, Boulder

CSCI 5828 — Lecture 9 — 02/09/2010

© University of Colorado, 2010

Lecture Goals

- Finish review of material in Chapter 2 of Breshears
  - Design Models for Concurrent Algorithms
    - Data Decomposition
      - Example
    - What's Not Parallel
- Begin review of material in Chapter 3 of Breshears
  - Verification of Parallel Algorithms
  - Performance Metrics

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
    - DISCUSSED IN LECTURE 6
  - Data Decomposition
    - program has large data structures where individual data elements can largely be calculated independently
    - data decomposition implies task decomposition in these cases

Data Decomposition

- A common opportunity for making a single-threaded application concurrent is looking for updates to one or more elements of a large data structure
  - if the update computations are independent of each other
    - you can
      - divide up the data structure
      - assign the updates to tasks running on separate threads
- The trick is deciding how to split up the data structure
  - and how independent the updates actually are

Data Structures

- Data structures amenable to decomposition?
  - Arrays, Arrays and Arrays
    - Split along one or more dimensions
  - Lists
    - if the list has index pointers
  - Trees
    - as long as the tree doesn't need to be balanced once it is built

Creating Tasks...

- Regardless of data structure
  - the decomposition into chunks will guide task creation
    - If updates are truly independent, then no locking required
    - If updates share information, then synchronization will be needed
  - load balancing is a concern
    - if the data structure is not regular, then some threads will have more work to do than others
      - if so, adopt a dynamic scheduling approach to divide up the work

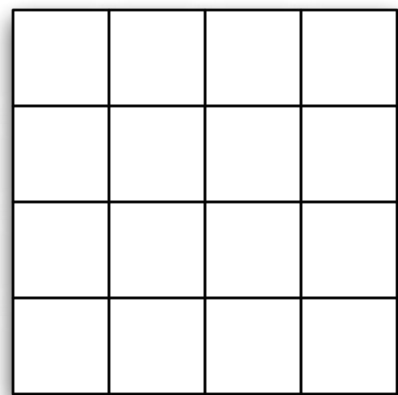
Three Questions of DD

- Three Questions of Data Decomposition
  - How should you divide the data into chunks?
  - How do you ensure that a task has the information it needs?
  - How are data chunks assigned to threads?

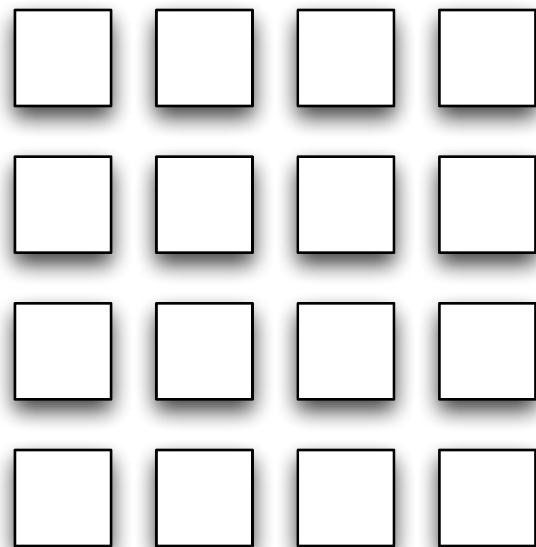
Doing Division

- The key concerns in dividing up the data structure are
  - make sure you have at least one "data chunk" per thread (more is better)
  - make sure there is enough work per chunk to trivialize the overhead incurred by concurrent solutions
- If computations are not independent, you must also be aware of
  - sharing data between tasks
  - synchronizing tasks that might be updating interdependent chunks

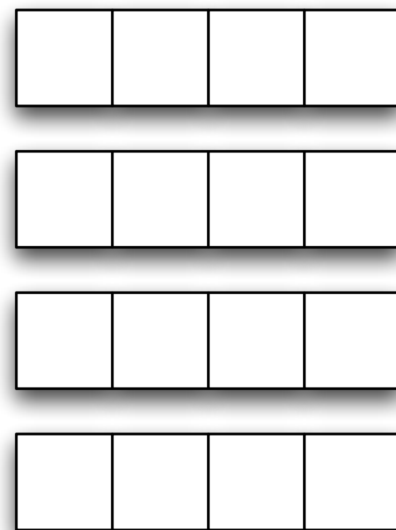
Dividing Arrays



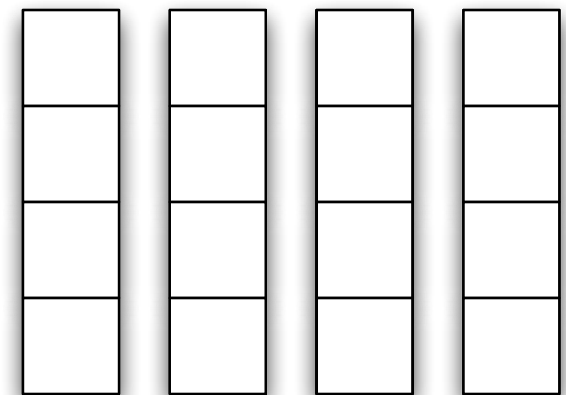
Original



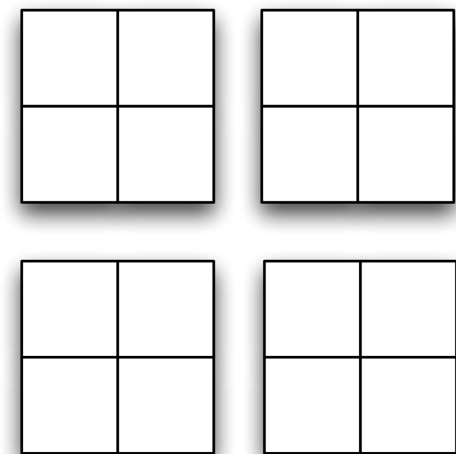
By Element



By Row

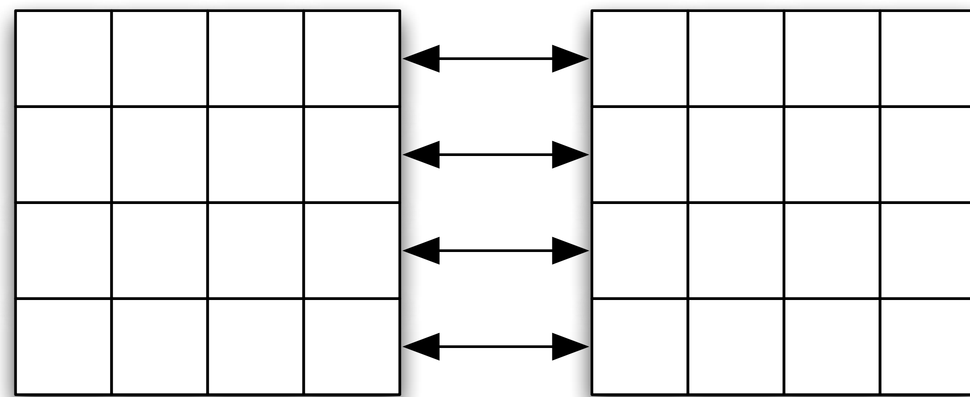


By Column

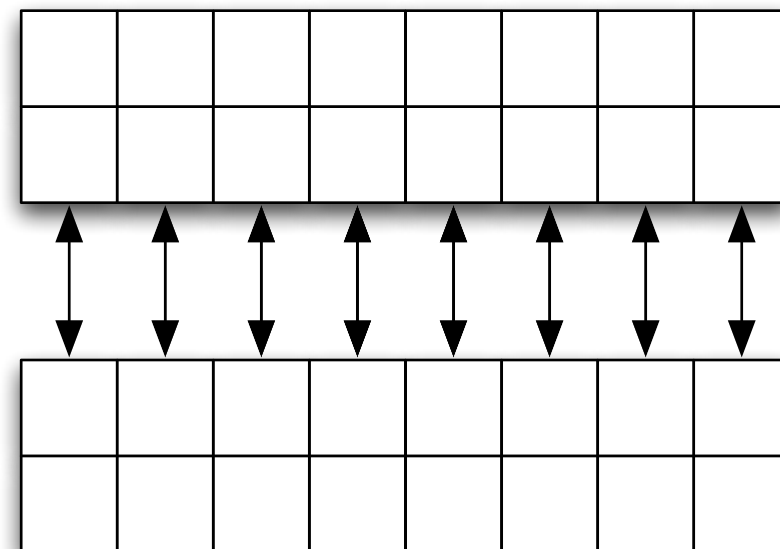


By Block

Shared Borders



4 shared borders



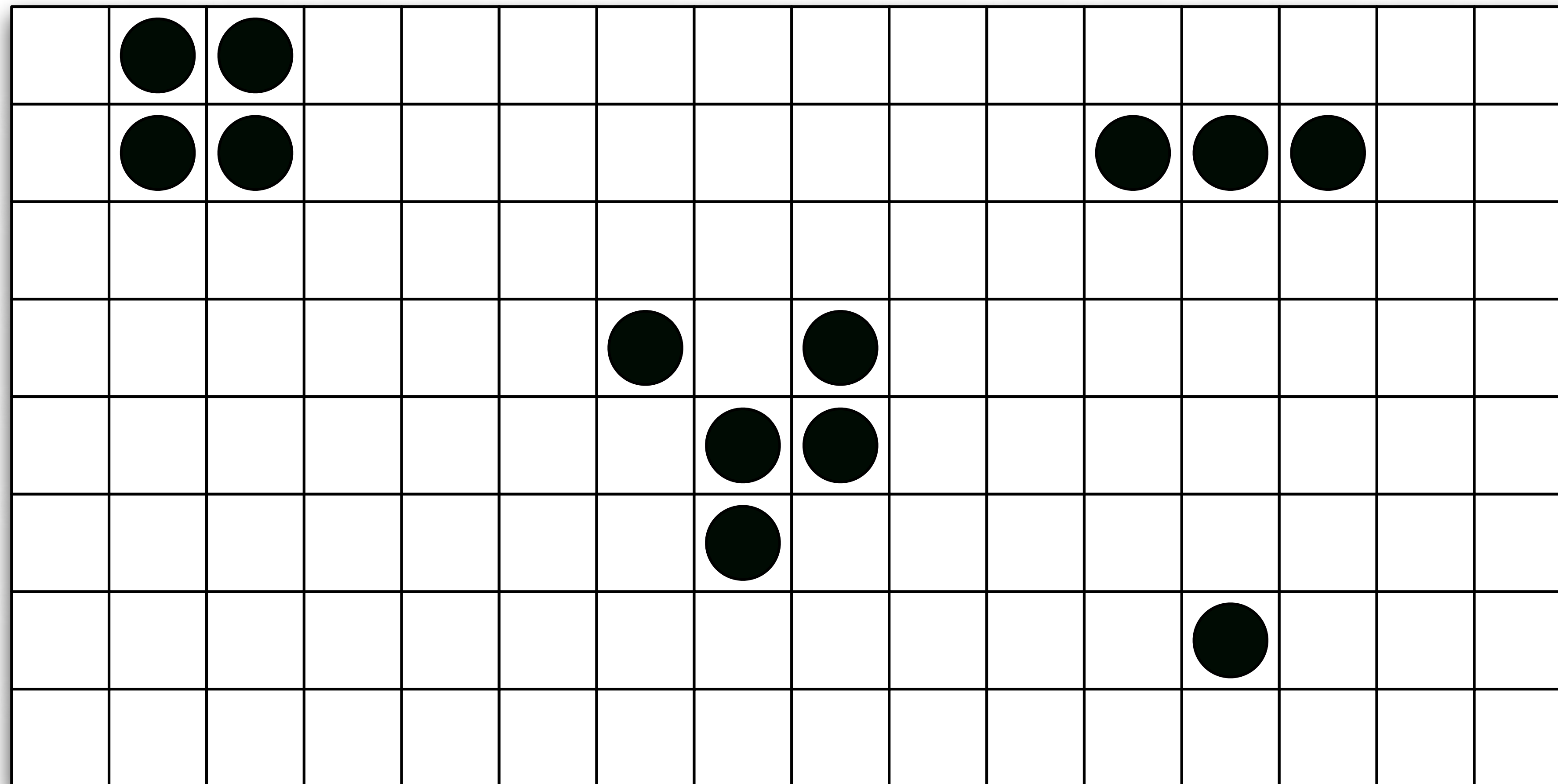
8 shared borders

When determining the granularity of tasks that have to deal with dependencies be cognizant of shared borders between data chunks; it directly impacts the work you have to do and memory use

Sharing Information

- One technique to deal with dependencies is
  - to copy data from other chunks
  - have a task access the copied data instead
- The issue here is "when do I copy the information"?
  - If another task is going to be updating the information in a shared data chunk, you must use synchronization to ensure the copy is made after the other task is done with its update
- 2nd technique is to assign dependent chunks to the same thread; may help to reduce the need for synchronization
  - consider dividing up a large array into groups of columns

Example: Game of Life



How might we design a threaded update for this game?

Rules: 1. cell alive → only one living neighbor → cell dead
2. cell alive → >4 living neighbors → cell dead
3. cell alive → 2-3 living neighbors → cell alive
4. cell dead → 3 living neighbors → cell alive

All computations performed “at once” for each cycle

Design Factor Scorecard

- Key factors when designing concurrent applications
  - Efficiency
    - amount of overhead, thread organization
  - Simplicity
    - amount of overhead, percentage of serial version present
  - Portability
    - across machines, across threading models
  - Scalability
    - ability to finish faster with more cores/threads

What’s Not Parallel?

- Breshears starts this section with a riff on Fred Brooks
  - “The bearing of a child takes nine months, no matter how many women are assigned!”
- It a task takes nine months to produce 1/9th of a result
  - (“a baseball team” in Breshears example)
- then nine tasks can create the entire result in 9 months
  - a single task would take 6.75 years to produce the entire result!
- The point? Not everything can be parallelized!

Algorithms with State

- Any algorithm that depends on information from a previous execution of the algorithm
  - HTTP gains so much of its power by being “stateless”
    - HTTP Cookies which add state to HTTP weakens the protocol
- When confronted with this situation
  - add locks
    - forces all concurrent executions of this algorithm to be serial
      - may be acceptable if algorithm is small part of larger computation
  - make reentrant (no global variables; only thread-local storage)

Recurrences

- `for (i = 1; i < N; i++) a[i] = a[i-1] + b[i]`
- If the calculation of a loop depends on a value previously calculated by the loop…
  - a[0] would need to be initialized prior to the above loop
- … then, you can’t make it concurrent
- Hopefully this type of loop will be a small part in a larger computation that can be made concurrent
  - otherwise, you are out of luck
- Note: this is a special case of “loop-carried dependence”

Induction Variables

- Variables that are **incremented** on each trip through a loop
  - i = 0; for (k=1; k < N; k++) { … i += k; … ; a[i] = … ; }
- The only way to get around this is to discover a way to calculate the exact value of the induction variable based on the current value of the loop variable
- The book gives an example in which the induction variable just happens to be the sum of the integers from 1 to k which can be calculated via (k*k + k)/2
  - it won’t always be this easy, however!

Reduction

- Reductions take a collection of data and reduce it to a single scalar value through some combining operation
  - To parallelize, the operation must be associative and commutative
    - if so, you can perform the operation on subsets and then combine the results of the initial tasks in one additional task
- Otherwise, you're out of luck and must look to see if the reduction is part of a larger computation that can be made independent

Verification of Parallel Algorithms

- Now that we have discussed design strategies for creating concurrent applications, how do we verify that they are correct?
  - Beyond executing them on test cases
- What we are looking for are techniques that can be used at design time
  - Breshears starts with a method drawn from a book by M. Ben-Ari called Principles of Concurrent and Distributed Programming, Second Edition (Addison-Wesley)

Ben-Ari's technique

- 1. programs are the execution of atomic statements
  - atomic statements cannot be interrupted by the OS
- 2. concurrent programs are interleavings of atomic statements from two or more threads
  - operating systems schedule such threads non-deterministically
- 3. All statements will be included in an interleaving (fairness)
- 4. Thus, to prove or verify a desirable property, ***we must show that the desirable property holds for all interleavings of atomic statements from two or more threads***

Background

- Breshears now works through a set of examples known as the “critical section problem”
  - The examples eventually reveal an approach known as Dekker’s Algorithm
- But the important point of this section is what you have to train your brain to do in order to analyze code like this
  - Think in terms of multiple threads that can arbitrarily interleave their statements; identify all **relevant paths** through the code; ensure all desirable properties are maintained and all undesirable properties are avoided

Critical Section Problem

- A critical section is a portion of code from a concurrent system where shared variables are accessed by a thread and updated
  - To avoid "race conditions" mutual exclusion is required; only a single thread can be in a critical section at once
- The challenge here is to develop an algorithm that achieves this property WITHOUT using synchronization objects provided by your threading model

Critical Section Problem

- Two Properties
  - The code enforces mutual exclusion of the critical region; plus if there are multiple threads waiting to enter the region and it becomes available, only one thread is allowed to enter
  - A thread not in the critical region cannot prevent another thread from entering the region
- Our experimental set-up
  - Two threads, two critical regions represented by methods and two methods to represent work done outside the critical region

Attempt One

- The lock step approach
  - Each thread has a “spin loop”
    - while (cond) {}; // I typically write: while (cond) {Thread.yield();}
  - otherwise the scheduler will sit in the loop for a LONG time
  - A global variable keeps track of whose “turn” it is
  - A loop wanting to enter the critical region waits for its turn
  - After leaving the critical region, it sets the global variable to indicate that it’s done and it’s the other threads turn
- Let’s look at the code

Analysis (I)

- The code starts indicating that it is thread zero’s turn
  - Thread 1 will always block in its spin loop; thread zero will charge ahead and enter the critical region
  - When it is done, it sets the turn to thread 1, releasing thread 1
  - It can then go on with other work, and return to the top of the loop; it might get back to its spin loop before thread 1 has done anything (remember the two threads execute independent of one another: arbitrary interleaving can occur)
    - It will then block however; mutual exclusion (property 1) has been achieved

Analysis (II)

- Now, imagine the following situation
  - Thread 0 is in critical region, exits, sets var to 1, and enters OtherStuffZero, a long running task
  - Thread 1 enters critical region, exits, sets var to 0, does other stuff, spins around and now wants to enter the critical region
- Can it do it?
  - No! Thread zero has to finish OtherStuffZero, spin around, enter the critical region and set the var to 1 again!
  - This algorithm thus violates our second desirable property

Attempt Two

- Two variables, one for each thread, to indicate when their respective threads are in the critical regions
  - If thread zero wants to enter its critical region
    - it spins while thread one is in its critical region
    - then it updates its variable
    - enters its critical region
    - and updates it again when it is done
  - the same is true for thread one, except it spins on thread zero
- Let's look at the code

Analysis (I)

- The problem from attempt one is gone
  - There is no “turn taking” variable preventing thread one from entering its critical region while thread zero is doing stuff outside its critical region
- Now, let’s analyze the code to see if it demonstrates mutual exclusion

Analysis (II)

- T0 tests T1INSIDE
- T1 tests T0INSIDE
- T0 finds FALSE
- T1 finds FALSE
- T0 sets T0INSIDE
- T1 sets T1INSIDE
- T0 enters critical region
- T1 enters critical region

- BOOM!
- There is nothing in this code to prevent the threads from both checking to see if the other is in the CR at the SAME time
- Let’s run the code

Attempt Three

- Less selfish threads: “more genteel threads”
  - Two variables express an “intent to enter the critical region”
  - Threads first set their variable, then spin while the other thread has the same intent, then enter the critical region and then set their intent to zero
  - Similar to second attempt, but the order in which flags are set and checked are different
    - in attempt_two, we check then set
    - in attempt_three, we set and then check
- Potential Problem: no distinction between “intent” and “in”

Analysis

- Unstable situation
  - If you run the code, it doesn't take long for it to lock up
- While a cursory walkthrough will show that both properties can be maintained
  - mutual exclusion is maintained if the problem is avoided
  - a thread can enter its CR as many times as it wants if the other thread is off doing other things
- BUT: we have a similar problem as attempt 2
  - both can set their "intent" at the same time and then spin forever waiting for the other to "finish" → DEADLOCK

Sidebar: DEADLOCK

- To have the potential for deadlocks, you need multiple threads and shared resources and the following conditions
  - mutual exclusion: resources are not shared at the same time
  - hold and wait: a thread waits to acquire a resource
  - no preemption: thread cannot be forced to release a resource
  - circular wait: A is waiting for B who is waiting for A
- If you have these conditions, then deadlock can occur
  - as we saw in attempt three
  - here the resources were the “intent” variables

Attempt Four

- To address deadlock, the spin loops are changed such that
  - a thread revokes its “intent”
  - sleeps
  - declares its “intent”
  - and then checks the condition of the spin loop again
- This breaks the potential for deadlock since now each thread is voluntarily revoking its locked resource
- Lets look at the code

Analysis

- Both properties are now maintained BUT there is still a problem
  - Now there is a potential for a thread to starve
  - Starvation occurs when a thread is inadvertently blocked from making progress
    - It is not being blocked explicitly but some other thread is “winning” when the scheduler makes a decision concerning which instruction to run next
- In this program, it is possible for one of the threads to continually enter its CR, preventing the other thread from accessing its CR for long periods of time

Attempt Five (Dekker's Algorithm)

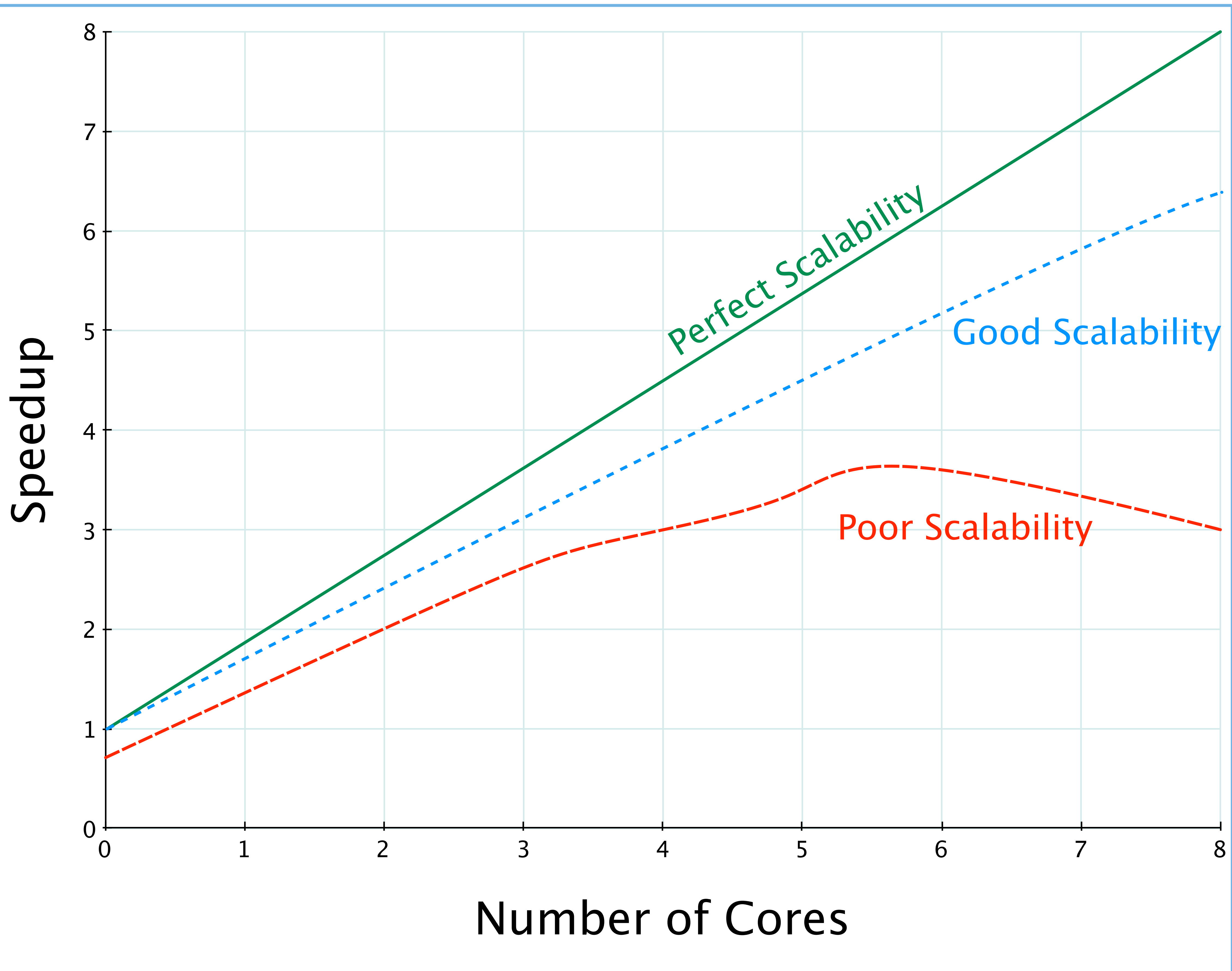
- To address starvation, we add one final modification
  - a variable that keeps track of “favored” status
  - Similar to attempt_1 but now all favored status says is
    - if both threads want to enter the CR, the favored thread wins
    - favored status switches back and forth to prevent starvation
- Lets look at the code
  - We will keep attempt_four the way it is, just add the favored status flag and the code to manage it; now the program will run forever (if you and the universe let it)

What's the point again?

- Not to demonstrate Dekker's Algorithm!
  - It only works for two threads anyway; not very useful
- The important goal of this exercise was to demonstrate the
  - types of properties that are desirable for concurrent apps
  - types of problems to avoid for concurrent apps
    - race conditions, deadlock, starvation
  - types of analysis that you must use to achieve these goals
- You must be able to simulate the execution of multiple threads in your head, identifying problematic interleavings

Performance Metrics

- How do you measure the performance of your concurrent application?
  - elapsed time
- How do you know if it's any good?
  - compare its elapsed time with the elapsed time of the fastest single-threaded program that does the same thing (using the same input, of course)
- How do you report it?
  - Speedup: How many times faster is the concurrent app?


Speedup
Number of Cores
Perfect Scalability
Good Scalability
Poor Scalability
0
1
2
3
4
5
6
7
8

Beware Superlinear…

- If you see superlinear speedup, be suspicious
  - for example, you get a 10x speedup with two cores
- Check List
  - Double check the timings of both programs
  - Make sure you are computing correct results
  - Is your test set realistic?
    - Your “chunked” test data set may be small enough to fit into local cache, giving a false speedup that goes down, one a realistic data set is provided

Predicting Performance

- Amdahl's Law
  - Can predict upper bound of speedup for a given project
- Inputs
  - percentage of serial execution time that can be made parallel
    - e.g., you know 10 functions can be made to run in parallel and you know they take 60% of the single threaded execution time
  - number of cores

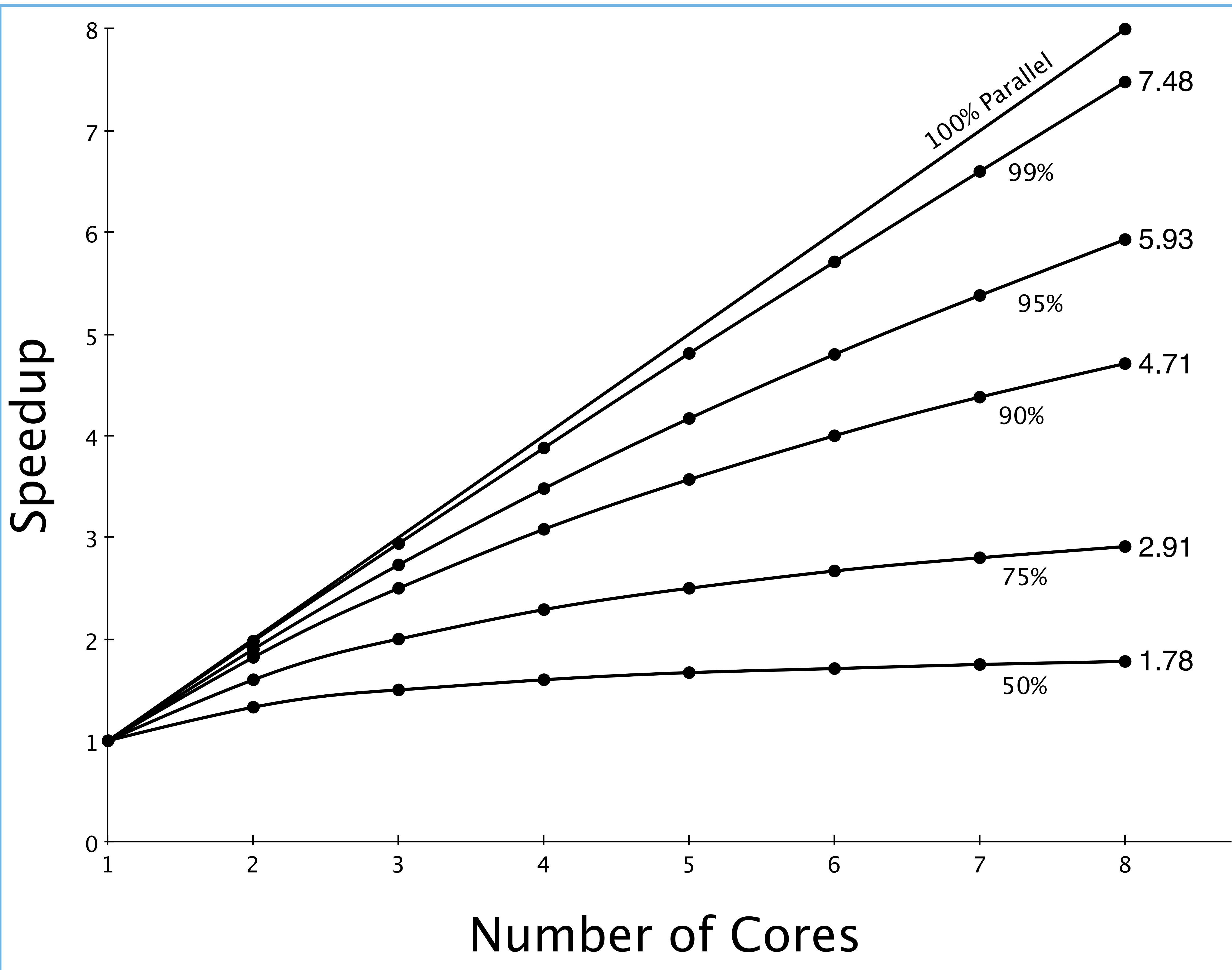
Amdahl's Law

$$Speedup \leq \frac{1}{(1 - pctPar) + \frac{pctPar}{p}}$$

Notes:

(p = 1) ➞ equation becomes 1/1 = 1x speedup
(> p) ➞ smaller impact of second term ➞ larger overall value
(> pctPar) ➞ smaller bottom fraction ➞ larger overall value

(p = 8, pctPar = 0.99) ➞ 1/(0.01) + .12375 = 1/.13375 = 7.48x


Speedup
Number of Cores
100% Parallel
99%
95%
90%
75%
50%
7.48
5.93
4.71
2.91
1.78
0
1
2
3
4
5
6
7
8

Upper Bound

- Amdahl's Law is an upper bound prediction
  - All it does is predict how close you get to the ideal linear speedup based on the amount of code you think you can get to run in parallel
- It ignores real-world concerns like
  - overhead (communication, synchronization, etc.)
- and assumes a fixed data set size for any and all numbers of cores used

Gustafson-Barsis's Law

- A measure that tries to factor in how the size of the data processed by an application can go up when it has more cores being thrown at it

$$Speedup \leq p + (1 - p)\, s$$

- p = num. cores, s = serial execution time of parallel app
- computes how much faster this app is over its single threaded version
- s = 10/1000 and p = 64, gbl = 63.37

Efficiency

- Finally, efficiency is a metric that tells us how well are we utilizing the computational resources of our overall system
  - Efficiency = Speedup / number of cores
- If speedup is 50x on 60 cores, then efficiency is 83.4%
  - over a run of the program, each of the cores is idle 16.6% of the time

Example

- Program with 8 threads
  - 1 thread reads files from disk; puts contents into a queue
  - 7 threads retrieve entries from the queue and process them
- Potential exists for a 7x speedup vs. single threaded app.
  - But performance will likely be less because
    - the thread reading from disk may not be able to fill the queue fast enough for the seven processing threads
    - if so, then those threads will sit idle, blocked until data is present

Wrapping Up

- Concepts
  - Data Decomposition
  - What’s Not Parallel
  - Verification of Parallel Algorithms
  - Performance Metrics

Coming Up Next

- Lecture 10: Eight Simple Rules for Designing Multithreaded Applications
  - Chapter 4 of Breshears
- Lecture 11: Good Enough Design
  - Chapter 5 of Pilone & Miles