Question on topic of “no standard parallel model”: Sequential computers were quite different originally, before one machine (IBM 701) gained widespread use. Won’t the widespread use of Intel (or AMD) CMPs have that same effect for parallelism?

Part III: Concepts

Goal: Understand basic concepts and trade-offs of parallelism

Discuss …

- Do we think that the multicore processor will become the idealized parallel machine in the same way the 701 defined the RAM model?

Finishing the Discussion on CTA

- The CTA is supposed to guide us in finding good computations to run on parallel machines
- Using it should
  - Aid in producing programs exploiting locality
  - Insure the program distributes work ‘well’
  - Other features, to be discussed later
- Consider sorting and HW2 …

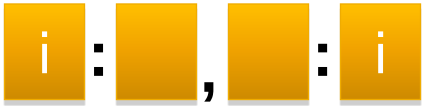
Odd/Even Sort…A Good || Solution?

- The idea: Create a lot of independent parallel work – compare adjacent pairs and exchange if out of order – repeating until ordered. Lots of parallelism; w.c. A[0]==max
- Specifically, … for i Odd
  - First 'half step', compare A[i]:A[i+1], exch if OoO
  - Second 'half step', compare A[i-1]:A[i], exch if OoO
  - If a step has no exchanges, stop

Thinking Realistically …

- General criticisms of this idea –
  - Dependences between threads at ½ step size
  - Parallel work in a 1/2 step is very modest: one compare and (possibly) one exchange
  - Though there is *n*-way parallelism, much is probably wasted
  - *n* == P/2 is unlikely
- Considering the CTA –
  - λ >> amount of work at each ½ step

To Revise the Idea

- Clearly, increasing the work at each step is smart
  - Allocate *n*/*P* items per processor
  - Extend the comparison … 

    to

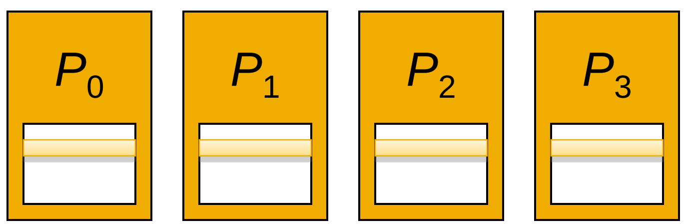
  - A value from the neighbor could propagate along

O/E - E/O Sort

- Overall logic and analysis

One Step:

get end neighbor value:	λ
O/E half step:	(n/P)c
get end neighbor value:	λ
E/O half step:	(n/P)c
And-reduce over done?:	λ log P



What is the worst case number of steps?

An Easy Argument: What crosses the midpoint?

Considering HW2

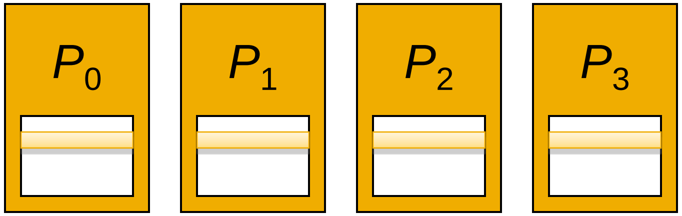
- Task: Recognize the well-formedness of ((xxx))
- An easy sequential solution …

```
open = 0;                          // keep count of opens
for (i=0; i<n; i++) {              // proceeding L to R
  if (A[i] == '(' ) open++;        // found one
  if (A[i] == ')' ) {              // here's a match
      open--;
      if (open < 0) break;         // oops, mismatch
  }
}
```

Does this look totally sequential??

Proceeding As Usual

- Allocate a contiguous sequence of symbols to a processor



- Each processor gets an ill-formed subsequence
  - (x)) (((x) x x x (x))
- Begin by resolving locally
  - ~~(x)~~) (~~((x) x x x (x))~~

  Leaving unresolved closes and opens

The Global Steps

- The unresolved values from each subproblem produce a similar problem, except optimized

) (becomes 1 1 and) ((((becomes 1 4
- Adjacent pairs *combine* their unresolved counts to a new pair describing the larger sequence:

  1 1 and 1 4 become 1 4

  2 4 and 1 2 become 2 5
- Resolved to the root: 0 0 is balanced

Summarizing the Solution

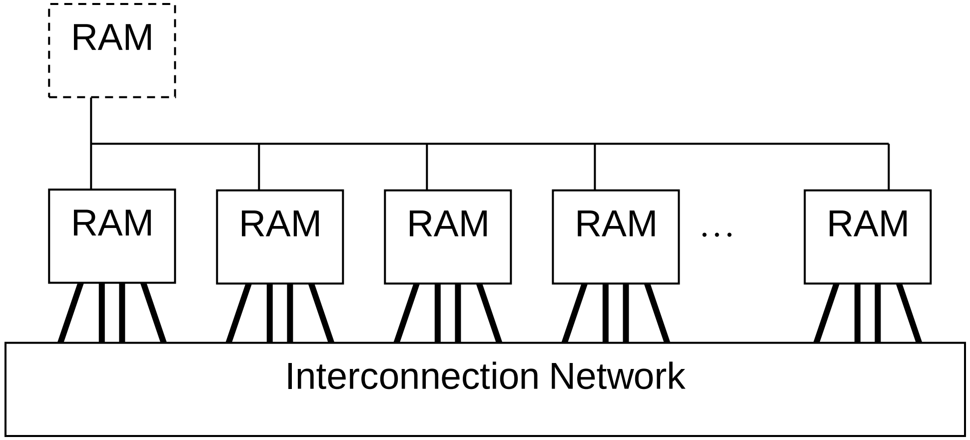
- Allocate contiguous subsequences of size *n*/*P* to each processor, starting with P_0
- Sequentially, locally resolve, creating c o *cn*/*P*
- Combine pairs to produce new c o descriptors by inducing a tree on PE indices: [0-1][2-3] … for level 1, [0-3][4-7] … for level 2, etc.
- Log levels of the tree to produce a final descriptor: c o $c\lambda\log_2 P$
- Only a result of o o means balanced

Where Was The Focus?

- First step: Allocated work to processors, generally by dividing it evenly
- Next step: Found local, independent work to perform
- Next step: Focused on combining subproblems into a tree network
- Made correctness and termination conditions explicit

Completing the CTA Discussion

- Controller
  - Not strictly needed
  - Often available
- How well does the CTA match other parallel architectures?
  - CMPs & SMPs
  - Clusters
  - Blue Gene



Precision of the CTA

- The CTA is a 'machine model' – an abstraction
- How can it be wrong?
  - Architecture has more features – shared memory
  - CTA predicts a certain behavior and features in the architecture make the program much faster
  - If it mispredicts … it's in trouble
- Isn't it a mistake for the CTA to ignore all the great stuff architects put in a processor

The CTA focuses on the parts that matter

Using the CTA

- Why should we believe it's right?
  - In his thesis (1993) Calvin Lin did a careful study of using the CTA as a programming model against the models used by others (whatever they were)
    - CTA consistently pointed programmers to better solutions
    - The CTA's effectiveness was independent of architecture
    - The apparent value of the model is emphasizing locality – always a benefit in computing
- The greatest value of the CTA would be if it is the basis for parallel programming languages

Threads

- A thread consists of program code, a program counter, call stack, and a small amount of thread-specific data
  - Threads share access to memory (and the file system) with other threads
  - Threads communicate through the shared memory
  - Though it may seem odd, apply the CTA model to thread programming -- emphasize locality, expect sharing to cost plenty

Threads are familiar, but don't use std model

Processes

- A process is a thread in its own private address space
  - Processes do not communicate through shared memory, but need another mechanism like message passing
  - Key issue: How is the problem divided among the processes, which includes data and work
  - Processes (logically subsume) threads

Compare Threads & Processes

- Both have code, PC, call stack, local data
  - Threads -- One address space
  - Processes -- Separate address spaces
- Weight and Agility
  - Threads: lighter weight, faster to setup, tear down, more dynamic
  - Processes: heavier weight, setup and tear down more time consuming, communication is slower

Mostly we use 'thread' & 'process' interchangeably

Terminology

- Terms used to refer to a unit of parallel computation include: thread, process, processor, …
  - Technically, thread and process are SW, processor (including SMT) is HW
  - Usually, it doesn’t matter
  - I will (try to) use “thread/process” for logical parallelism, and “processor” when I mean physical parallelism

Parallelism vs Performance

- Naïvely, many people think that applying *P* processors to a *T* time computation will result in *T*/*P* time performance
- Generally wrong
  - For a few problems (Monte Carlo) it is possible to apply more processors directly to the solution
  - For most problems, using *P* processors requires a paradigm shift
- Assume "P processors => T/P time" to be the best case possible

Better Intuition

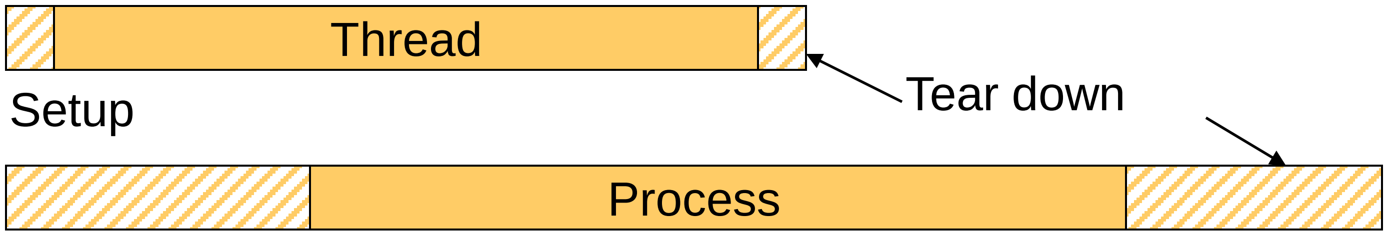
- (Because of the presumed paradigm shift) the sequential and parallel solutions differ so we do not expect a simple performance relationship between the two
  - More or fewer instructions must be executed
- Examples of other differences
  - The hardware is different
  - Parallel solution has difficult-to-quantify costs such as communication time, wait time, etc. that the serial solution does not have

More Instructions Needed

- To implement parallel computations requires overhead that sequential computations do not need
  - All costs associated with communication are overhead: locks, cache flushes, coherency, message passing protocols, etc.
  - All costs associated with thread/process setup
  - Lost optimizations -- many compiler optimizations not available in parallel setting
    - Instruction reordering

Performance Loss: Overhead

- Threads and processes incur overhead



- Obviously, the cost of creating a thread or process must be recovered through parallel performance:

$$(t_1 + o_{su} + o_{td} + \mathrm{cost}(t_2))/2 < t_2$$

t_p = p proc execution time
o_{su} = setup, o_{td} = tear down
cost(t_2) = all other || costs

More Instructions (Continued)

- Redundant execution can avoid communication -- a parallel optimization

New random number needed for loop iteration:

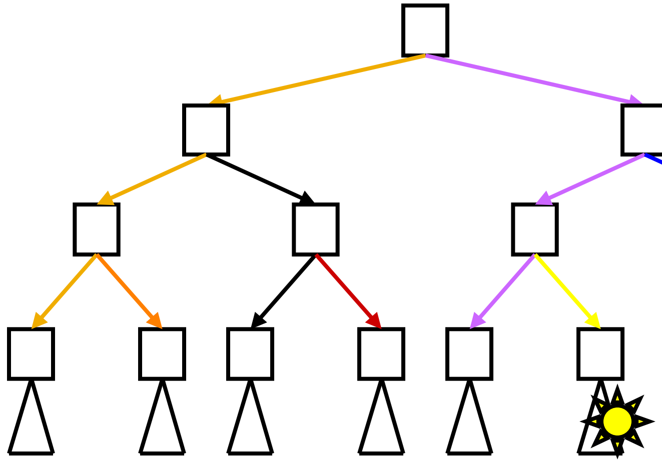
(a) Generate one copy, have all threads ref it … requires communication

(b) Communicate seed once, then each thread generates its own random number … removes communication and gets parallelism, but by increasing instruction load

A common (and recommended) programming trick

Fewer Instructions

- Searches illustrate the possibility of parallelism requiring fewer instructions



- Independently searching subtrees means an item is likely to be found faster than sequential

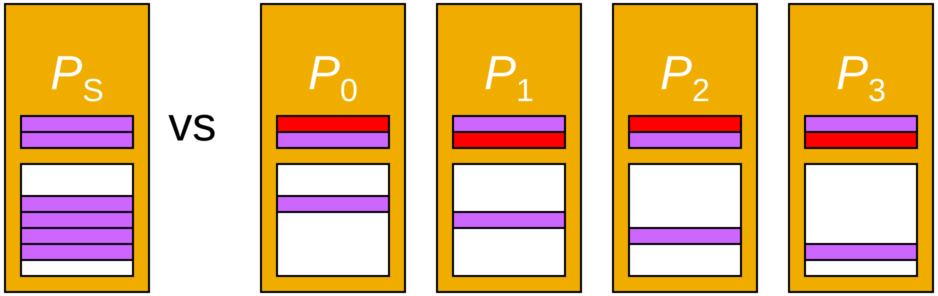
One vs Many

- Sequential hardware ≠ parallel hardware
  - There is more parallel hardware, e.g. memory
  - There is more cache on parallel machines
  - Sequential computer $\overset{?}{\neq}$ 1 processor of || computer, because of coherence hw, power, etc.
    - Important in multicore context
  - Parallel channels to disk, possibly

These differences *tend* to favor || machine

Superlinear Speed up

- Additional cache is an advantage of ||ism



- The effect is to make execution time < *T*/*P* because data (& program) memory references are faster
- Cache-effects help mitigate other || costs

Other Parallel Costs

- Wait: All computations must wait at points, but serial computation waits are well known
- Parallel waiting …
  - For serialization to assure correctness
  - Congestion in communication facilities
    - Bus contention; network congestion; etc.
  - Stalls: data not available/recipient busy
- These costs are generally time-dependent, implying that they are highly variable

Bottom Line …

- Applying *P* processors to a problem with a time *T* (serial) solution can be either …
  better or worse …
- It's up to programmers to exploit the advantages and avoid the disadvantages

Break

Amdahl's Law

- If $1/S$ of a computation is inherently sequential, then the maximum performance improvement is limited to a factor of S

$$T_P = 1/S \times T_S + (1-1/S) \times T_S / P$$

T_S=sequential time
T_P=parallel time
P =no. processors

- Amdahl's Law, like the Law of Supply and Demand, is a fact

Gene Amdahl -- IBM Mainframe Architect

Interpreting Amdahl's Law

- Consider the equation

$$T_P = 1/S \times T_S + (1-1/S) \times T_S / P$$

- With no charge for || costs, let $P \to \infty$ then $T_P \to 1/S \times T_S$

The best parallelism can do to is to eliminate the parallelizable work; the sequential work remains

- Amdahl's Law applies to problem *instances*

Parallelism seemingly has little potential

More On Amdahl's Law

- Amdahl's Law assumes a fixed problem instance: Fixed *n*, fixed input, perfect speedup
  - The algorithm can change to become more ||
  - Problem instances grow implying proportion of work that is sequential may be smaller %
  - … Many, many realities including parallelism in 'sequential' execution imply analysis is simplistic
- *Amdahl is a fact; it's not a show-stopper*

Digress: Inherently Sequential

- As an artifact of *P*-completeness theory, we have the idea of *Inherently Sequential* -- computations not appreciably improved by parallelism

> Circuit Value Problem:
> Given a circuit α over Boolean inputs, values $b_1, \ldots, b_n$ and designated output value y, is the circuit true for y?

- Probably not much of a limitation

Two kinds of performance

- Latency -- time required before a requested value is available
  - Latency, measured in seconds; called *transmit time* or *execution time* or just *time*
- Throughput -- amount of work completed in a given amount of time
  - Throughput, measured in "work"/sec, where "work" can be bits, instructions, jobs, etc.; also called *bandwidth* in communication

Both terms apply to computing and communications

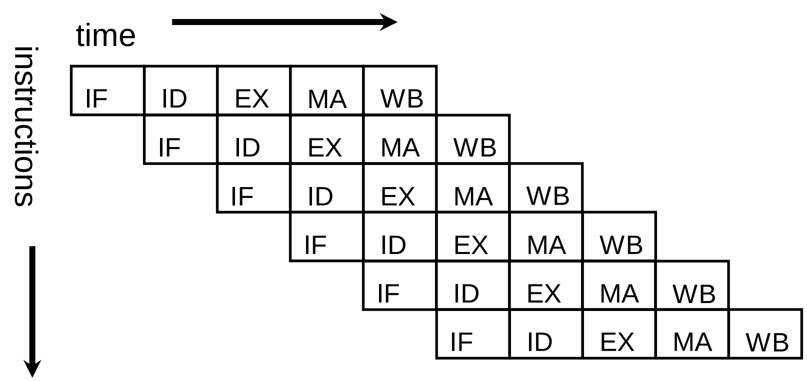
Latency

- Reducing latency (execution time) is a principal goal of parallelism
- There is upper limit on reducing latency
  - Speed of light, esp. for bit transmissions
  - In networks, switching time (node latency)
  - (Clock rate) x (issue width), for instructions
  - Diminishing returns (overhead) for problem instances

Hitting the upper limit is rarely a worry

Throughput

- Throughput improvements are often easier to achieve by adding hardware
  - More wires improve bits/second
  - Use processors to run separate jobs
  - Pipelining is a powerful technique to execute more (serial) operations in unit time



Better throughput often hyped as if better latency

Latency Hiding

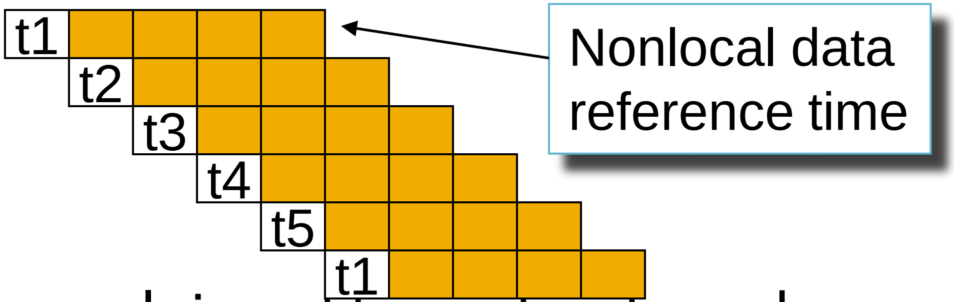
- Reduce wait times by switching to work on different operation (multithreading)
  - Old idea, dating back to Multics
  - In parallel computing it's called *latency hiding*
- Idea most often used to lower impact of λ cost
  - Have many threads ready to go …
  - Execute a thread until it makes nonlocal ref
  - Switch to next thread
  - When nonlocal ref is filled, add to ready list

See discussion from Part II

Latency Hiding (Continued)

- Latency hiding requires …
  - Consistently large supply of threads ~ λ/e

  where e = average # cycles between nonlocal refs
  - Enough network throughput to have many requests in the air at once



- Latency hiding has been claimed to make shared memory feasible in the presence of large λ

There are difficulties

Latency Hiding (Continued)

- Challenges to supporting shared memory
  - Threads must be numerous, and the shorter the interval between nonlocal refs, the more
    - Running out of threads stalls the processor
  - Context switching to next thread has overhead
    - Many hardware contexts -- or --
    - Waste time storing and reloading context
  - Tension between latency hiding & caching
    - Shared data must still be protected somehow
  - Other technical issues

Performance Loss: Contention

- Contention -- the action of one processor interferes with another processor's actions -- is an elusive quantity
  - Lock contention: One processor's lock stops other processors from referencing; they must wait
  - Bus contention: Bus wires are in use by one processor's memory reference
  - Network contention: Wires are in use by one packet, blocking other packets
  - Bank contention: Multiple processors try to access different locations on one memory chip simultaneously

Contention is very time dependent, that is, variable

Performance Loss: Load Imbalance

- Load imbalance, work not evenly assigned to the processors, underutilizes parallelism
  - The assignment of *work*, not data, is key
  - Static assignments, being rigid, are more prone to imbalance
  - Because dynamic assignment carries overhead, the quantum of work must be large enough to amortize the overhead
  - With flexible allocations, load balance can be solved late in the design programming cycle

The Best Parallel Programs …

- Performance is maximized if processors execute continuously on local data without interacting with other processors
  - To unify the ways in which processors could interact, we adopt the concept of dependence
  - A *dependence* is an ordering relationship between two computations
    - Dependences are usually induced by read/write
    - Dependences that cross process boundaries induce a need to synchronize the threads

Dependences are well-studied in compilers

Dependences

- Dependences are orderings that must be maintained to guarantee correctness
  - Flow-dependence: read after write — True
  - Anti-dependence: write after read — False
  - Output-dependence: write after write — False
- True dependences affect correctness
- False dependences arise from memory reuse

Example of Dependences

- Both true and false dependences

```
1.    sum = a + 1;
2.    first_term = sum * scale1;
3.    sum = b + 1;
4.    second_term = sum * scale2;
```

Example of Dependences

- Both true and false dependences

```
1.    sum = a + 1;
2.    first_term = sum * scale1;
3.    sum = b + 1;
4.    second_term = sum * scale2;
```

- Flow-dependence read after write; must be preserved for correctness
- Anti-dependence write after read; can be eliminated with additional memory

Removing Anti-dependence

- Change variable names

```
1.    sum = a + 1;
2.    first_term = sum * scale1;
3.    sum = b + 1;
4.    second_term = sum * scale2;
```

```
1.    first_sum = a + 1;
2.    first_term = first_sum * scale1;
3.    second_sum = b + 1;
4.    second_term = second_sum * scale2;
```

Granularity

- Granularity is used in many contexts…here *granularity* is the amount of work between cross-processor dependences
  - Important because interactions usually cost
  - Generally, larger grain is better

    \+ fewer interactions, more local work

    \- can lead to load imbalance
  - Batching is an effective way to increase grain

Locality

- The CTA motivates us to maximize locality
  - Caching is the traditional way to exploit locality … but it doesn't translate directly to ||ism
  - Redesigning algorithms for parallel execution often means repartitioning to increase locality
  - Locality often requires redundant storage and redundant computation, but in limited quantities they help

Measuring Performance

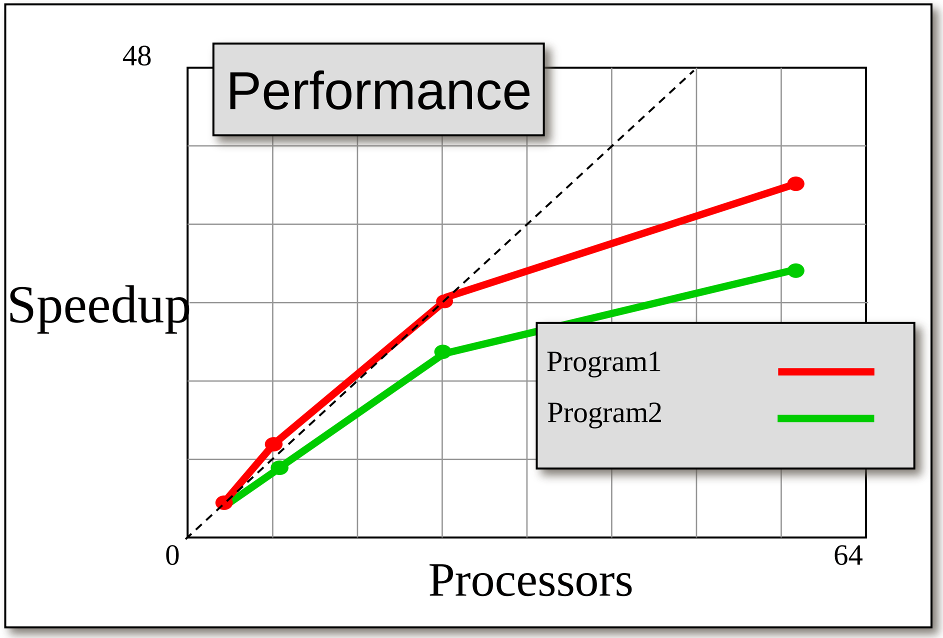
- Execution time … what’s time?
  - ‘Wall clock’ time
  - Processor execution time
  - System time
- Paging and caching can affect time
  - Cold start vs warm start
- Conflicts w/ other users/system components
- Measure kernel or whole program

FLOPS

- Floating Point Operations Per Second is a common measurement for scientific pgms
  - Even scientific computations use many ints
  - Results can often be influenced by small, low-level tweaks having little generality: mult/add
  - Translates poorly across machines because it is hardware dependent
  - Limited application … but it won't go away!

Speedup and Efficiency

- Speedup is the factor of improvement for *P* processors: T_S/T_P



Efficiency = Speedup/*P*

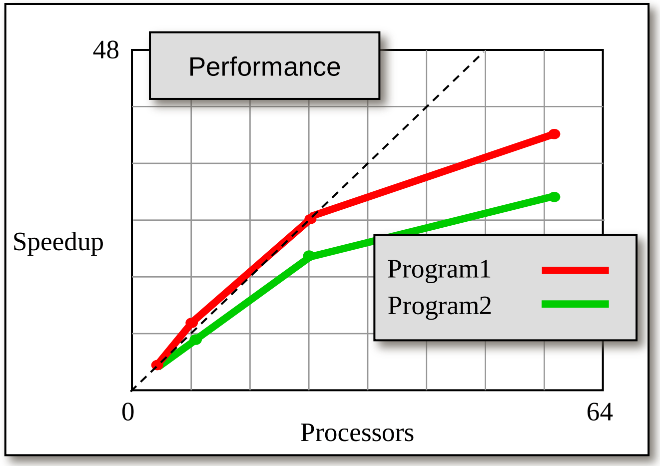
Issues with Speedup, Efficiency

- Speedup is best applied when hardware is constant, or for family within a generation
  - Need to have computation, communication in same ratio
  - Great sensitivity to the T_S value
    - T_S should be time of best sequential program on 1 processor of the ||-machine
    - $T_{P=1} \neq T_S$ Measures *relative speedup*

Relative speedup is often important but it must be labeled as such

Scaled v. Fixed Speedup

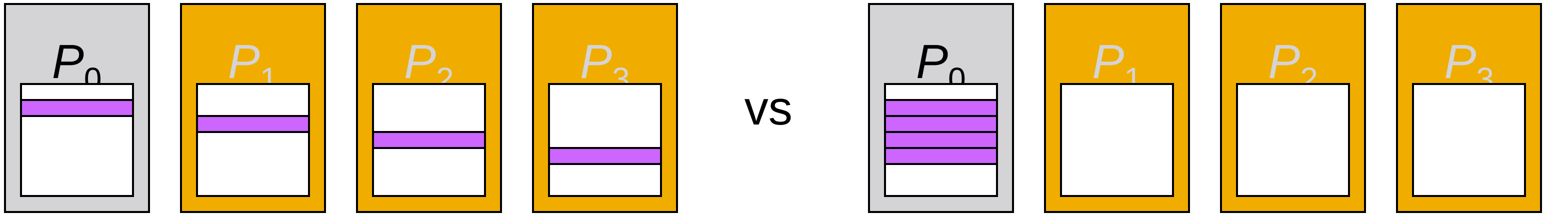
- As *P* increases, the amount of work per processor diminishes, often below the amt needed to amortize costs
- Speedup curves bend down
- Scaled speedup keeps the work per processor constant, allowing other effects to be seen
- Both are important



If not stated, speedup is fixed speedup

"Cooking" The Speedup Numbers

- The sequential computation should not be charged for **any** || costs … consider



- If referencing memory in other processors takes time (λ) and data is distributed, then one processor solving the problem results in greater *t* compared to true sequential

This complicates methodology for large problems

What If Problem Doesn't Fit?

- Cases arise when sequential doesn't fit in 1 processor of parallel machine
- Best solution is relative speed-up
  - Measure $T_{\pi=\mathit{smallest\ possible}}$
  - Measure T_P, compute T_π/T_P as having P/π potential improvement

We Will Return …

- Many issues regarding parallelism have been introduced, but they require further discussion … we will return to them when they are relevant

Summary of Key Points

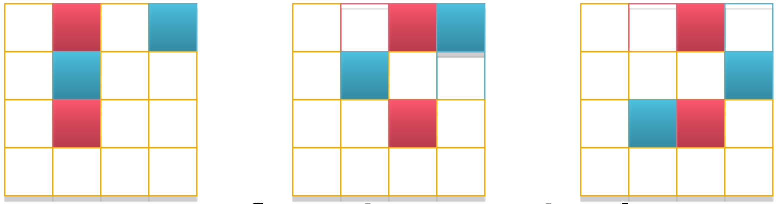
- Amdahl's Law is a fact but it doesn't impede us much
- Inherently sequential problems (probably) exist, but they don't impede us either
- Latency hiding could hide the impact of λ with sufficiently many threads and much (interconnection) bandwidth
- Impediments to parallel speedup are numerous: overhead, contention, inherently sequential code, waiting time, etc.

Review Key Points (continued)

- Concerns while parallel programming are also numerous: locality, granularity, dependences (both true and false), load balance, etc.
- Happily: Parallel and sequential computers are different: More hardware means more fast memory (cache, RAM), implying the possibility of superlinear speedup
- Measuring improvement is complicated

For Next Time

- Consider the Red/Blue Simulation: A 2D torus array, that is with wrap around, is randomly filled with some red & blue cells; unoccupied is white. In 1st half step, reds move right into unoccupied cell; in 2nd half step, blues move down into unoccupied cell; both happening (legally) is OK; terminate if occupancy of any 10x10 tile is outside [0.45, 0.55]; tile 0 is A[0..9,0..9]; 1 is A[0..9,10..19]; …



- Write a parallel program for the Red/Blue problem for a multicore or SMP machine using Pthreads (intro Ch 6); apply CTA-type analysis, trying to increase locality

Two Part Problem

- This program will have a 2 part turn-in
  - Part 1: Turn in a brief description (for a human) saying how your solution will go, and why you have chosen to do it that way. Rationale is key: "I will allocate the array as follows … because … ." Due Sunday (17 APR) by 5:00 PM
  - Part 2: Turn in a program with measured performance, that is, speedup, on a small parallel machine (CMP, SMP). Due Tuesday (20 APR) by class; a "flexibility week" is allowed.