COMP 322: Principles of Parallel Programming

Vivek Sarkar
Department of Computer Science
Rice University
vsarkar@rice.edu

Acknowledgments for Today's Lecture

- Keynote talk on "Parallel Thinking" by Prof. Guy Blelloch, CMU, PPoPP conference, February 2009
  - http://ppopp09.rice.edu/PPoPP09-Blelloch.pdf
- Cilk lectures by Profs. Charles Leiserson and Bradley Kuszmaul, MIT, July 2006
  - http://supertech.csail.mit.edu/cilk/
- Course text: "Principles of Parallel Programming", Calvin Lin & Lawrence Snyder
- "Introduction to Parallel Computing", 2nd Edition, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Addison-Wesley, 2003
- COMP 422 lectures, Spring 2008
  - http://www.cs.rice.edu/~vsarkar/comp422

Summary of Last Lecture

- Introduction to Parallel Computing
  - Parallel machines: CMP, SMP, Cluster
  - Power vs. Frequency trade-offs
- Algorithmic Complexity Measures
  - Computation graph model
    - Node = sequential unit of computation
    - Edge = dependence (precedence constraint)
  - T_P = execution time on P processors
  - T_1 = *work*
  - T_∞ = *span (critical path length)*
  - LOWER BOUNDS
    - $T_P \geq T_1/P$
    - $T_P \geq T_\infty$
    - ➔ $T_P \geq \max(T_1/P, T_\infty)$

Units of Measure in Parallel Computing

- Units for Parallel and High Performance Computing (HPC) are:
  - Flop: floating point operation
  - Flop/s or Flops: floating point operations per second
  - Bytes: size of data (a double precision floating point number is 8)
- Typical sizes are millions, billions, trillions…

Mega	Mflop/s = 10^6 flop/sec	Mbyte = 2^{20} = 1048576 ~ 10^6 bytes
Giga	Gflop/s = 10^9 flop/sec	Gbyte = 2^{30} ~ 10^9 bytes
Tera	Tflop/s = 10^{12} flop/sec	Tbyte = 2^{40} ~ 10^{12} bytes
Peta	Pflop/s = 10^{15} flop/sec	Pbyte = 2^{50} ~ 10^{15} bytes
Exa	Eflop/s = 10^{18} flop/sec	Ebyte = 2^{60} ~ 10^{18} bytes
Zetta	Zflop/s = 10^{21} flop/sec	Zbyte = 2^{70} ~ 10^{21} bytes
Yotta	Yflop/s = 10^{24} flop/sec	Ybyte = 2^{80} ~ 10^{24} bytes

- See www.top500.org for current list of fastest machines

Historical Concurrency in Top 500 Systems

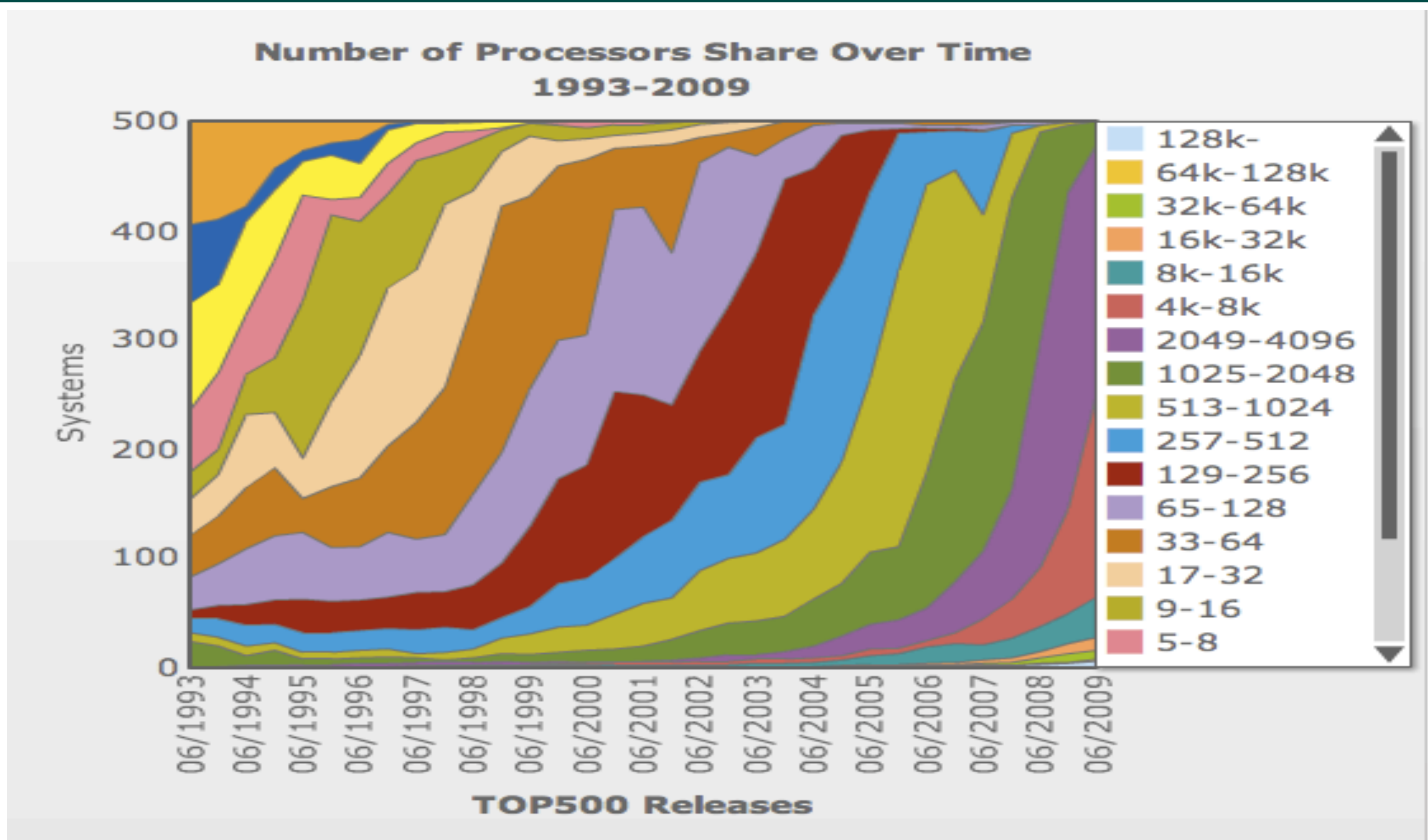
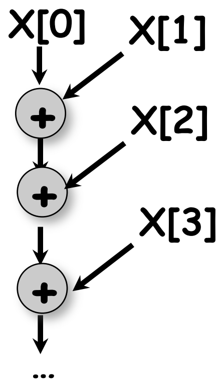


Figure Credit: www.top500.org, June 2009

Example 2: Prefix Sum (sequential version, pg 13)

- Problem: compute partial sums, $Y[i] = \Sigma_{j \le i} X[i]$
- Sequential algorithm
  - `Y[0] = X[0];  for ( i=1 ; i< n ; i++ ) Y[i] = X[i] + Y[i-1];`
- Computation graph



  - Work = O(n), Span = O(n), Parallelism = O(1)
- How can we design an algorithm (computation graph) with more parallelism?

One Approach to solving the Parallel Prefix Sum Problem

1. for each interior node N bottom-up

   Compute N.sum for N's sub-tree

2. root.inherited := 0

3. for each interior node N top.down

   N.left.inherited := N.inherited

   N.right.inherited := N.inherited + N.left.sum

4. for each leaf node L

   L.sum = L.inherited + L.value

- Work = O(n), Span = O(log n), Parallelism = O(n / (log n))

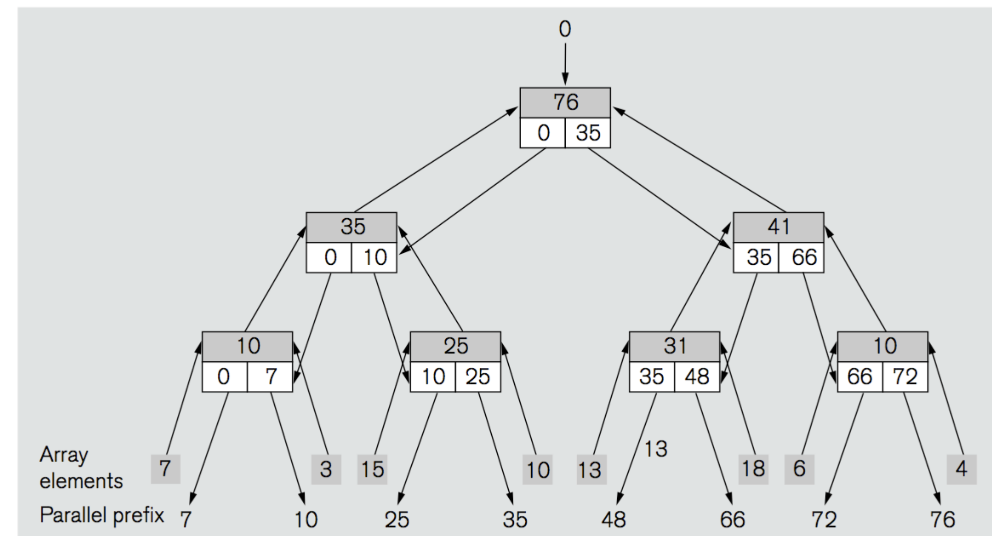


Figure 1.4
Computing the prefix sum. The gray node values, computed going up the tree, are from the pair-wise sum algorithm; the white values, the prefixes, are computed going down the tree by a simple rule: send the value from the parent to the left child; add the sum from the left child (that came up) to the value from the parent and send it to the right child.

Example 3: QuickSort (sequential version)

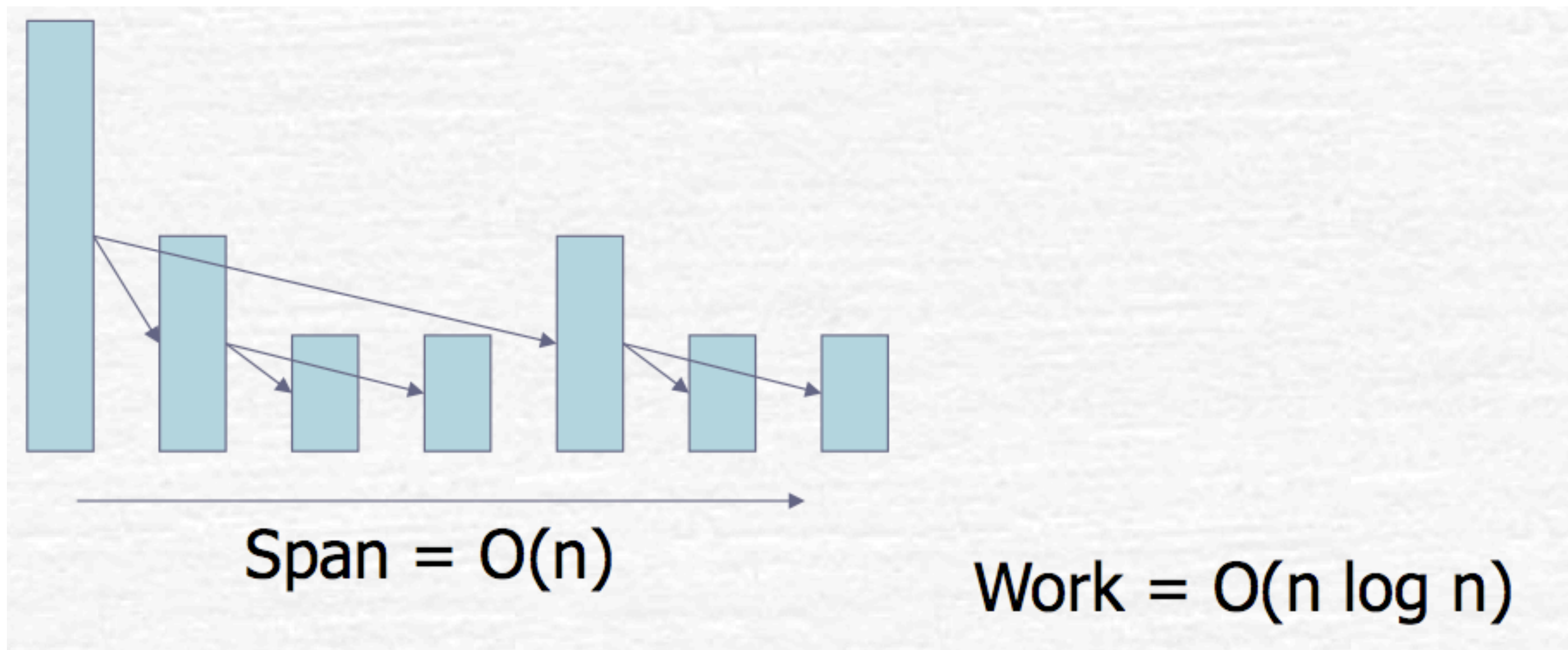
- Work = O(n log n), Span = O(n log n), Parallelism = O(1) :

```
public void quickSort(int[] a, int left, int right) {
    int i = left-1;  int j = right;
    if (right <= left) return;
    while (true) {
        while (a[++i] < a[right]);
        while (a[right]<a[--j]) if (j==left) break;
        if (i >= j) break;
        swap(a,i,j);
    }
    swap(a, i, right);
    quickSort(a, left, i - 1); quickSort(a, i+1, right);
}
```

Example 3: Parallelizing QuickSort

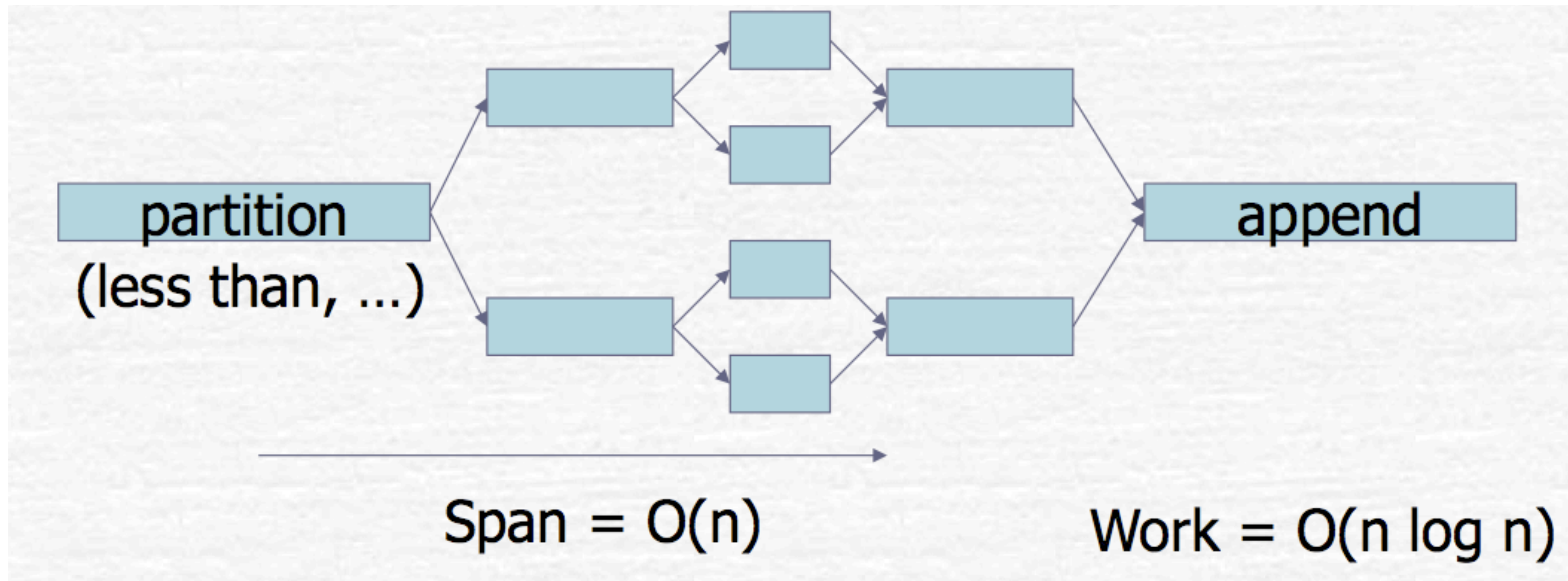
```
procedure QUICKSORT(S) {
   if S contains at most one element then return S
   else {
       choose an element a randomly from S;
       // Opportunity 1: Parallel Partition
       let S1, S2 and S3 be the sequences of elements in S less
       than, equal to, and greater than a, respectively;
       // Opportunity 2: Parallel Calls
       return (QUICKSORT(S1) followed by S2 followed by
               QUICKSORT(S3))
    } // else
} // procedure
```

Approach 1: Parallel partition, sequential calls



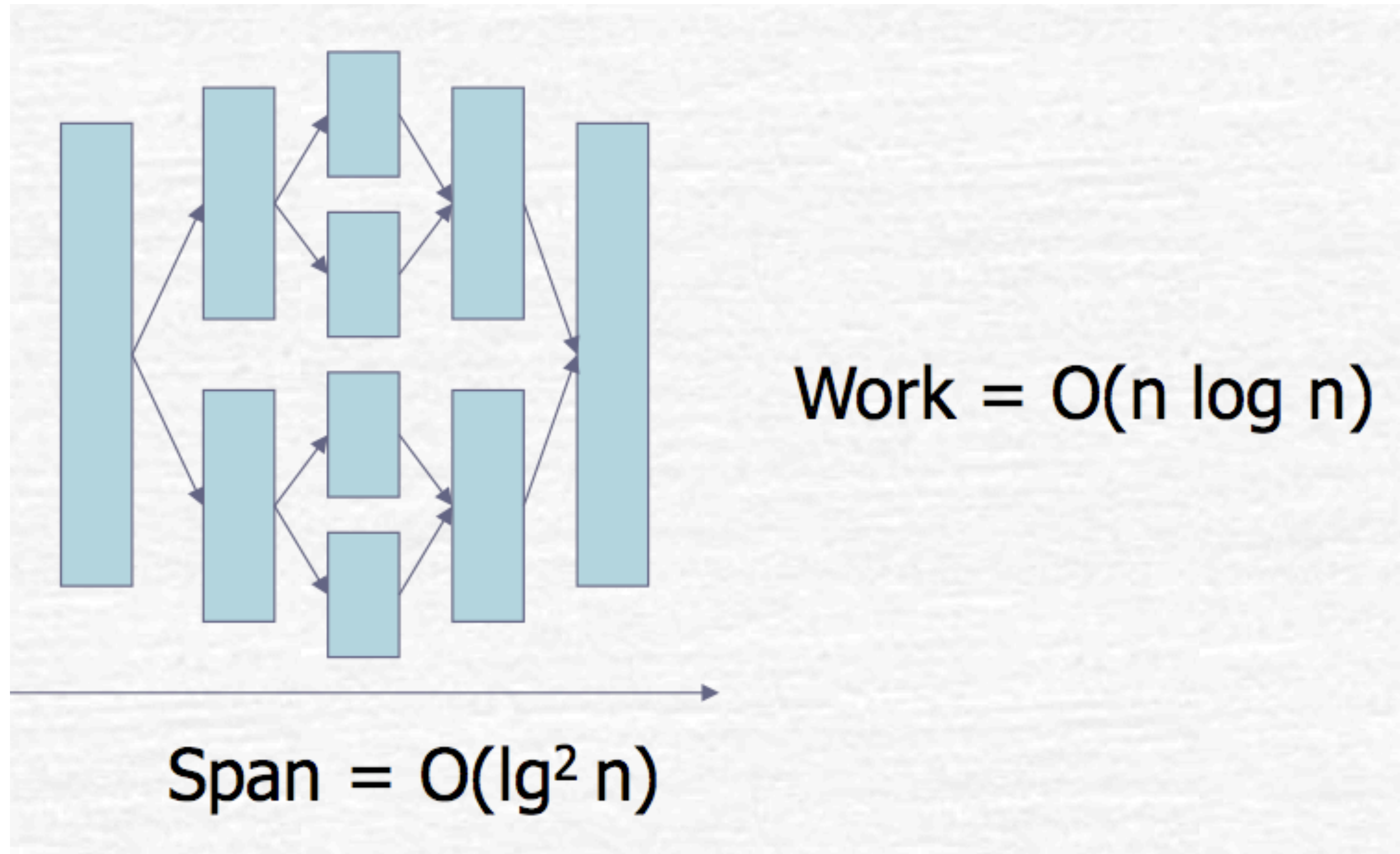
Parallelism = O(log n)

Approach 2: Sequential partition, parallel calls



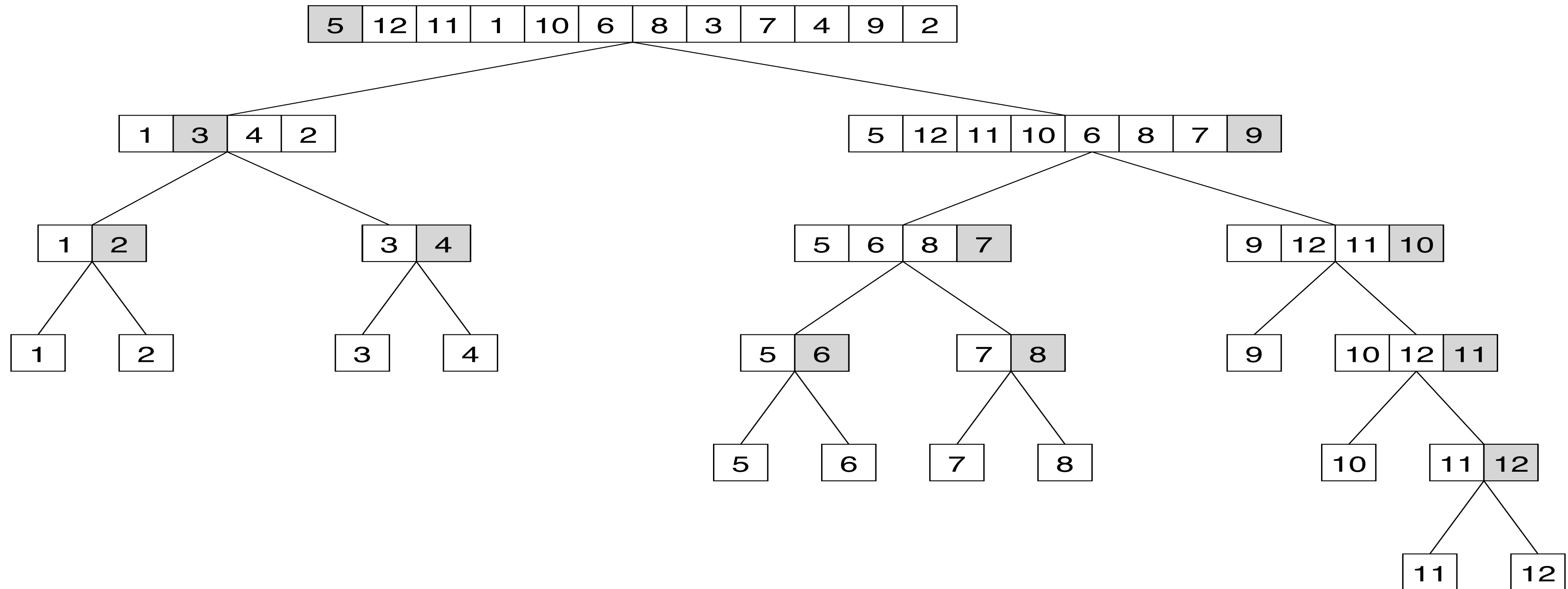
Parallelism = O(log n)

Approach 3: parallel partition, parallel calls



Parallelism = $O(n / \log n)$

Example Execution of Parallel Quicksort



Upper Bound for Greedy Scheduling

BASIC IDEA: Do as much as possible on every step.

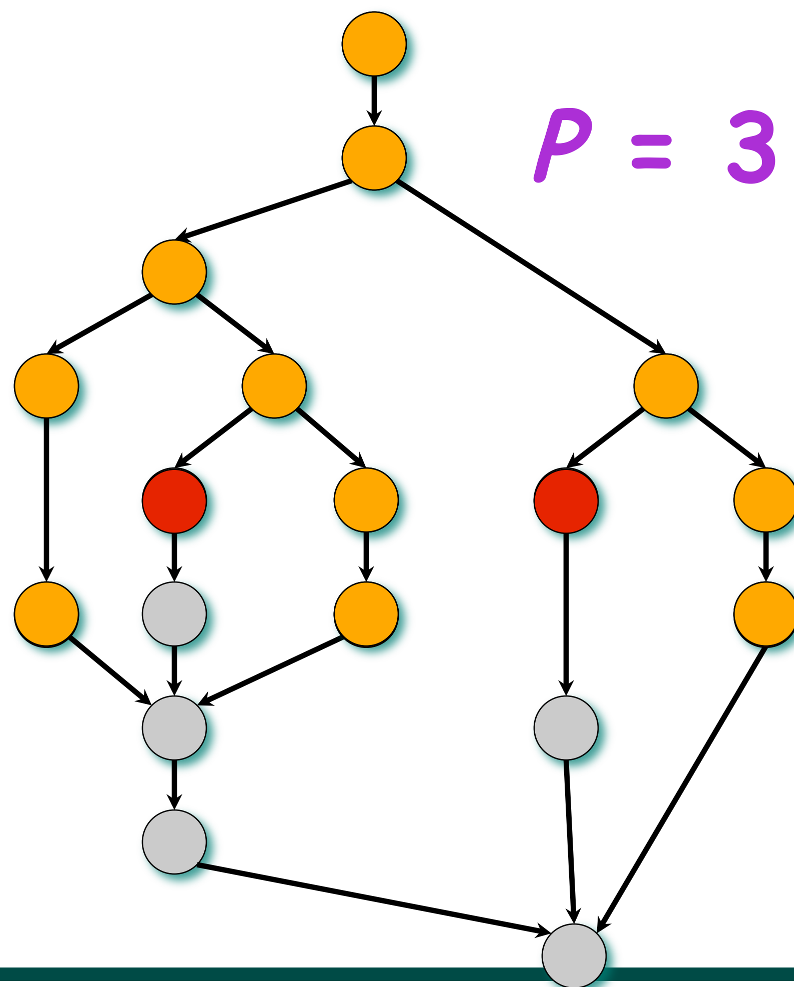
Definition: A node is *ready* if all its predecessors have *executed*.

Greedy Scheduling

Definition: A node is *ready* if all its predecessors have *executed*.

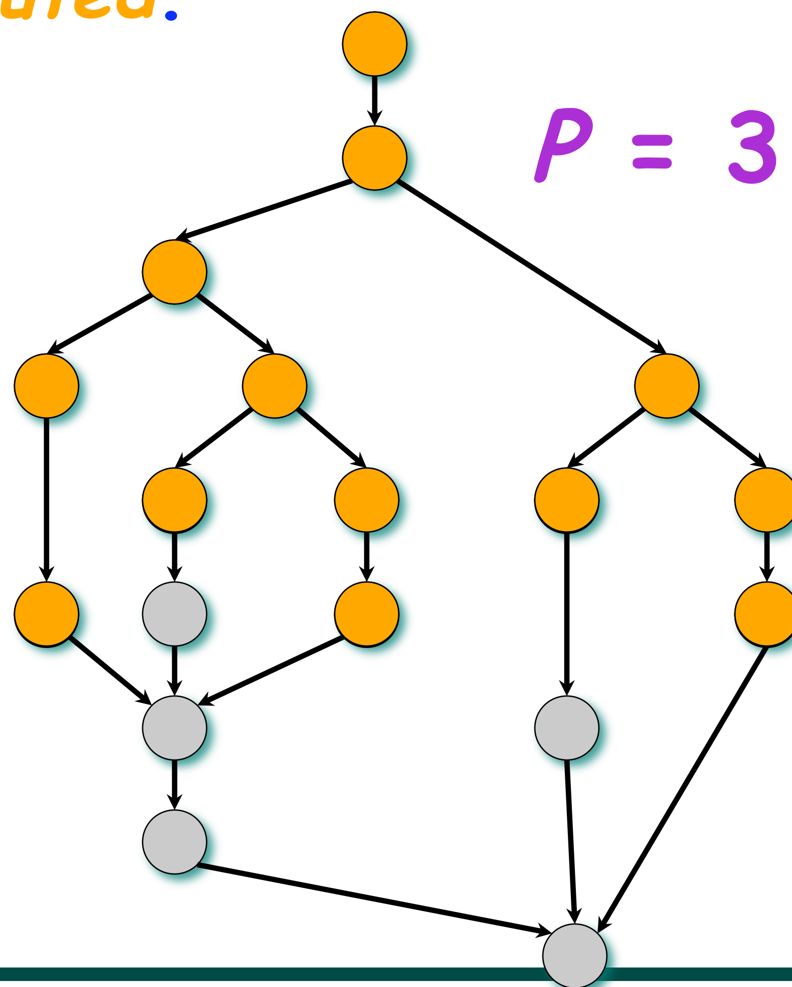
Complete step

- ≥ P nodes ready.
- Run any P.



Greedy Scheduling

Definition: A node is *ready* if all its predecessors have *executed*.



Complete step

- $\geq P$ nodes ready.
- Run any P.

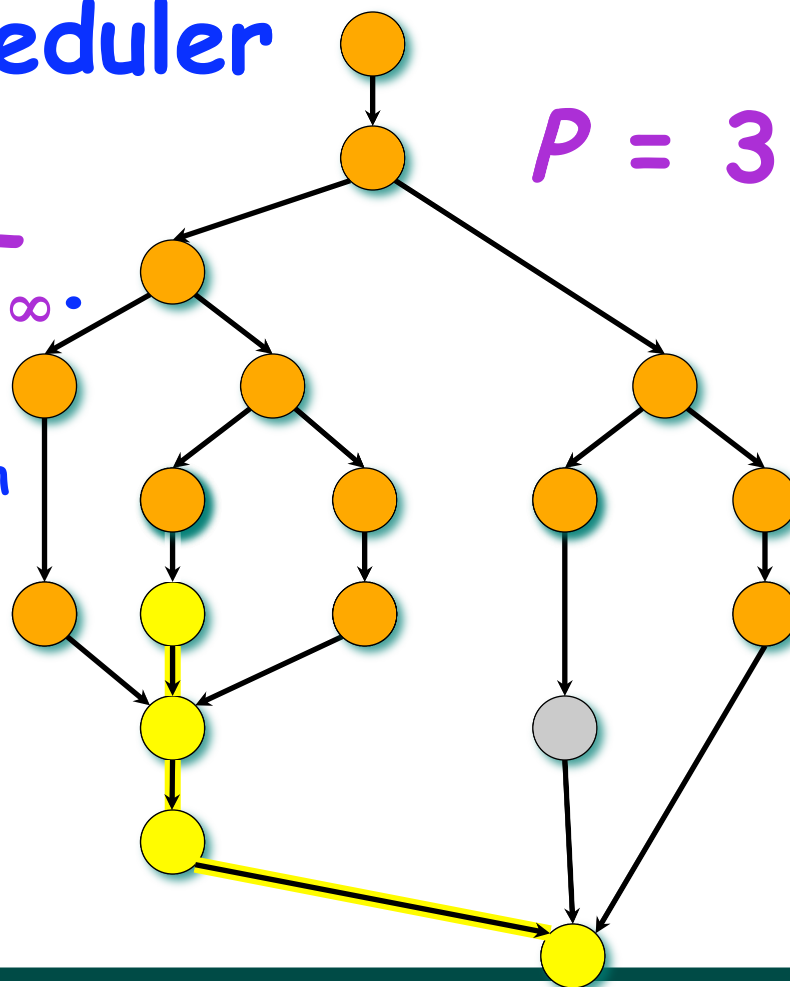
Incomplete step

- $< P$ nodes ready.
- Run all of them.

Greedy-Scheduling Theorem

Theorem [Graham '68 & Brent '75]. Any greedy scheduler achieves

$$T_P \le T_1/P + T_\infty.$$



Proof.

- \# complete steps $\le T_1/P$, since each complete step performs P work.
- \# incomplete steps $\le T_1$, since each incomplete step reduces the span of the unexecuted dag by 1. ■

Optimality of Greedy Schedulers

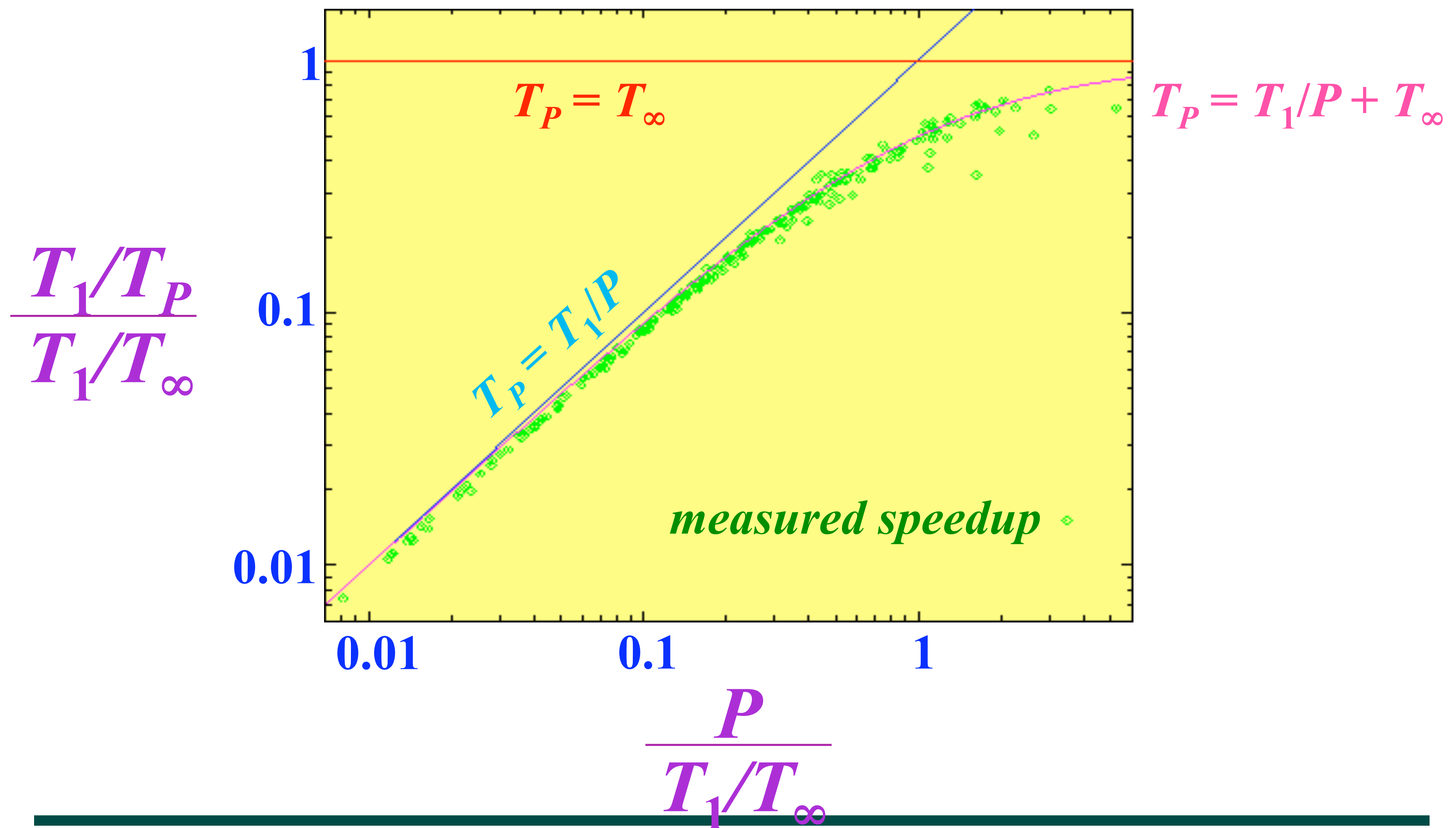
Corollary. Any greedy scheduler achieves a T_P that is within a factor of 2 of optimal, T_P^*.

Corollary. Any greedy scheduler achieves near-perfect linear speedup whenever $T_1/P \gg T_\infty$ or $T_1/P \ll T_\infty$

Case Study: Cilk Chess Programs

- *★Socrates* placed 3rd in the 1994 International Computer Chess Championship running on NCSA's 512-node Connection Machine CM5.
- *★Socrates 2.0* took 2nd place in the 1995 World Computer Chess Championship running on Sandia National Labs' 1824-node Intel Paragon.
- *Cilkchess* placed 1st in the 1996 Dutch Open running on a 12-processor Sun Enterprise 5000. It placed 2nd in 1997 and 1998 running on Boston University's 64-processor SGI Origin 2000.
- *Cilkchess* tied for 3rd in the 1999 WCCC running on NASA's 256-node SGI Origin 2000.

★Socrates Normalized Speedup



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_∞=1 second	T'_∞= 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

Amdahl's Law

Amdahl's Law places a strict limit on the speedup that can be realized by using multiple processors. Two equivalent expressions for Amdahl's Law are given below:

$t_N = (f_p/N + f_s)t_1$ Effect of multiple processors on run time

$S = 1/(f_s + f_p/N)$ Effect of multiple processors on speedup

Where:

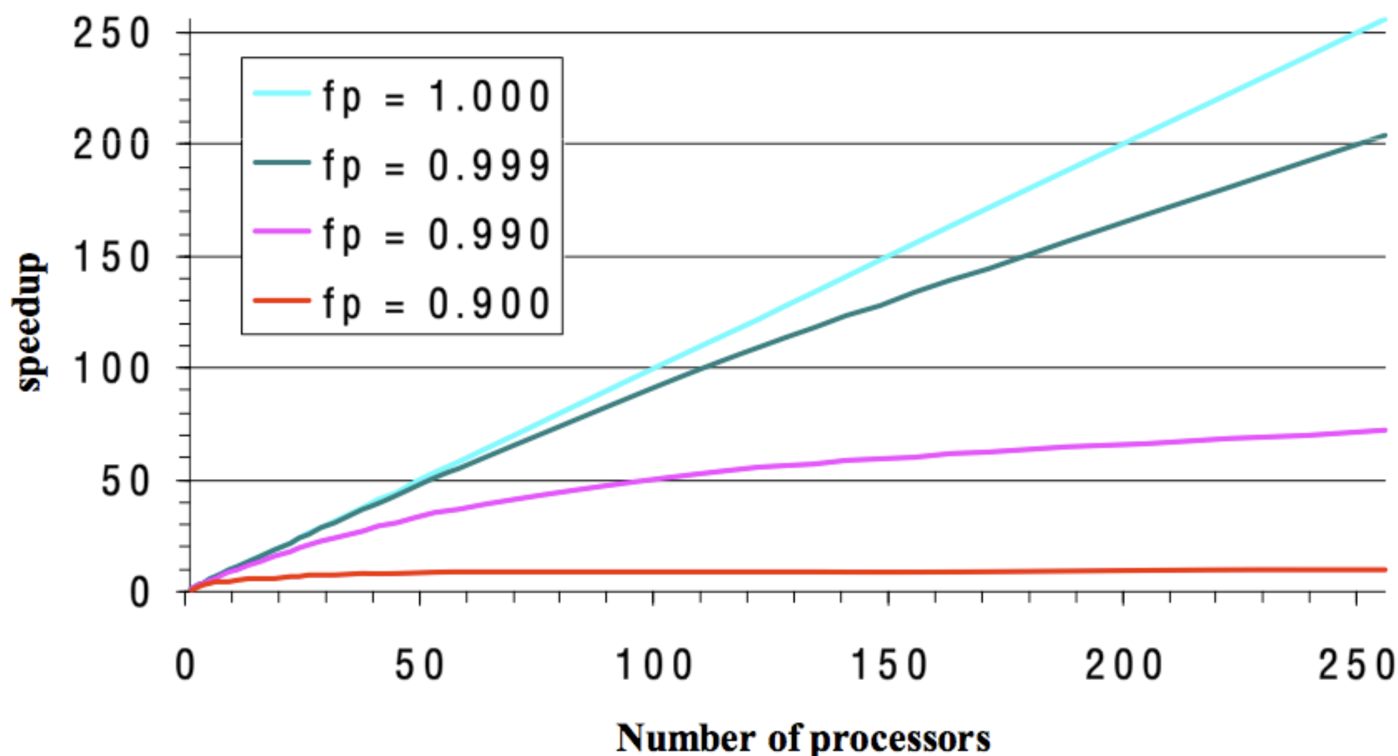
f_s = serial fraction of code

f_p = parallel fraction of code = $1 - f_s$

N = number of processors

Illustration of Amdahl's Law

It takes only a small fraction of serial content in a code to degrade the parallel performance. It is essential to determine the scaling behavior of your code before doing production runs using large numbers of processors



Summary of Today's Lecture

- Analysis of Parallel Algorithms
  - Prefix sum, Quicksort
- Greedy Scheduling and Upper Bound on T_P
- Amdahl's Law
- Reading list for next lecture
  - Chapter 3, Reasoning about Performance