
COMP 322: Principles of Parallel Programming

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



Course Information

- Meeting time: TTh 10:50am - 12:05pm
- Meeting place: Hertzstein Hall 119
- Instructor: Vivek Sarkar (vsarkar@rice.edu)
- Teaching Assistant: Sanjay Chatterjee (cs20@rice.edu)
- Software Assistant: Vincent Cave (vincent.cave@rice.edu)
- Web site: <http://www.cs.rice.edu/~vsarkar/comp322>
- Prerequisites: COMP 314 or equivalent
- Text: Lin & Snyder, Principles of Parallel Programming
- Course Requirements:
 - Midterm Exam 20%
 - Final Exam 20%
 - Homeworks 10% (short written assignments --- total of 2)
 - Projects 50% (larger programming assignments --- total of 2)

Scope of Course

- Foundations of parallel algorithms
- Foundations of parallel programming
 - Task creation and termination
 - Mutual exclusion and isolation
 - Collective and point-to-point synchronization
 - Data parallelism
 - Task and data distribution
- Habanero-Java (HJ) language, developed in the Habanero Multicore Software Research project at Rice
- Abstract executable performance model for HJ programs
- Hands-on programming projects
 - Abstract metrics
 - Real parallel systems (8-core Intel, 64-node Sun Niagara, Nvidia GPGPU's with 100+ cores)
- Beyond HJ: introduction to parallel programming in the real world e.g., Java Concurrency, .Net Task Parallel Library & PLINQ, Intel nodeing Building Blocks, CUDA

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>
- CS 194 course on “Parallel Programming for Multicore” taught by Prof. Kathy Yelick, UC Berkeley, Fall 2007
—<http://www.cs.berkeley.edu/~yelick/cs194f07/>
- Cilk lectures by Profs. Charles Leiserson and Bradley Kuszmaul, MIT, July 2006
—<http://supertech.csail.mit.edu/cilk/>
- Course text: Lin & Snyder, Principles of Parallel Programming

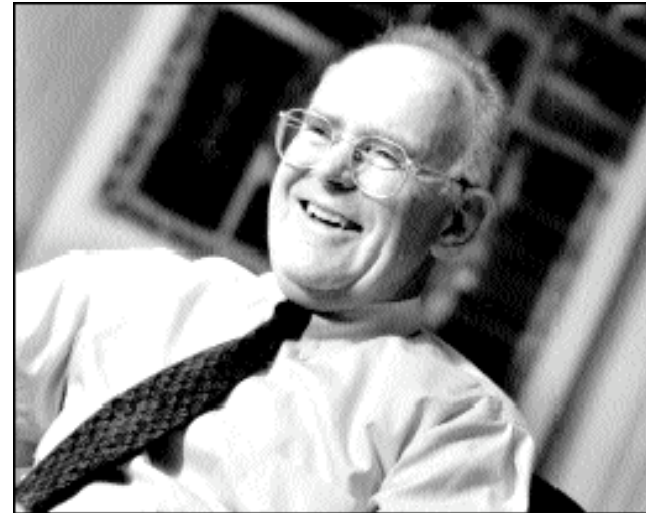
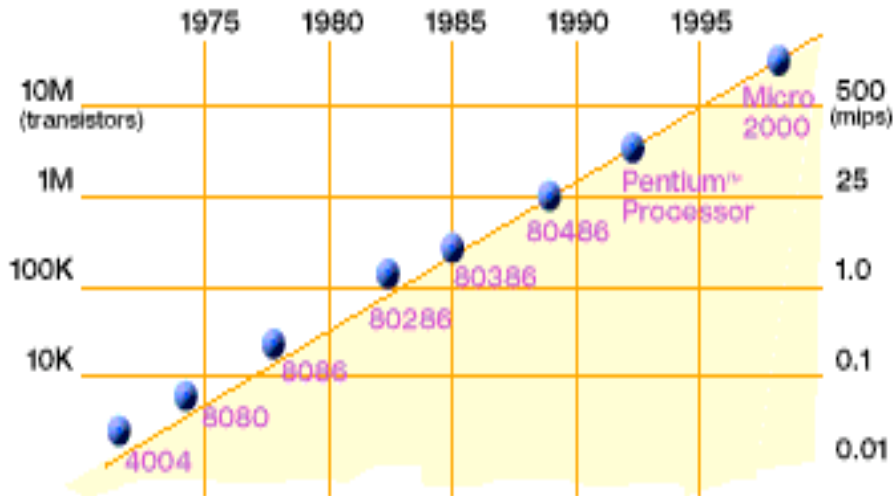
What is Parallel Computing?

- **Parallel computing:** using multiple processors in parallel to solve problems more quickly than with a single processor, or with less energy
- Examples of parallel machines (see pages 4-5 of textbook)
 - A computer **Cluster** that contains multiple PCs with local memories combined together with a high speed network
 - A **Symmetric Multi-Processor (SMP)** that contains multiple processor chips connected to a single shared memory system
 - A **Chip Multi-Processor (CMP)** contains multiple processors (called **cores**) on a single chip, also called **Multi-Core Computers**
- The main motivation for parallel execution historically came from the desire for improved performance
 - **Computation is the third pillar of scientific endeavor, in addition to Theory and Experimentation**
- But parallel execution has also now become a ubiquitous necessity due to power constraints, as we will see ...

Why Parallel Computing Now?

- Researchers have been using parallel computing for decades:
 - Mostly used in computational science and engineering
 - Problems too large to solve on one computer; use 100s or 1000s
- There have been higher level courses in parallel computing (COMP 422, COMP 522) at Rice for several years
- Many companies in the 80s/90s “bet” on parallel computing and failed
 - Sequential computers got faster too quickly for there to be a large market for specialized parallel computers
- Why is Rice adding a 300-level undergraduate course on parallel programming now?
 - Because the entire computing industry has bet on parallelism
 - There is a desperate need for all computer scientists and practitioners to be aware of parallelism

Technology Trends: Microprocessor Capacity



2X transistors/Chip Every 1.5 years

Called "Moore's Law"

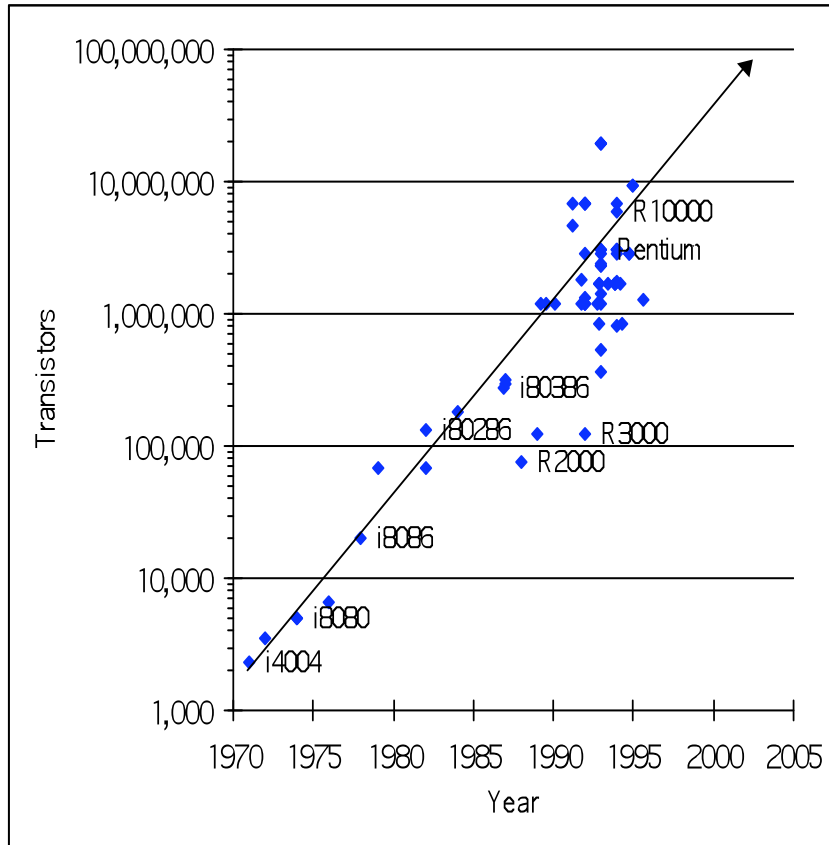
Microprocessors have become smaller, denser, and more powerful.

Gordon Moore (co-founder of Intel) predicted in 1965 that the transistor density of semiconductor chips would double roughly every 18 months.

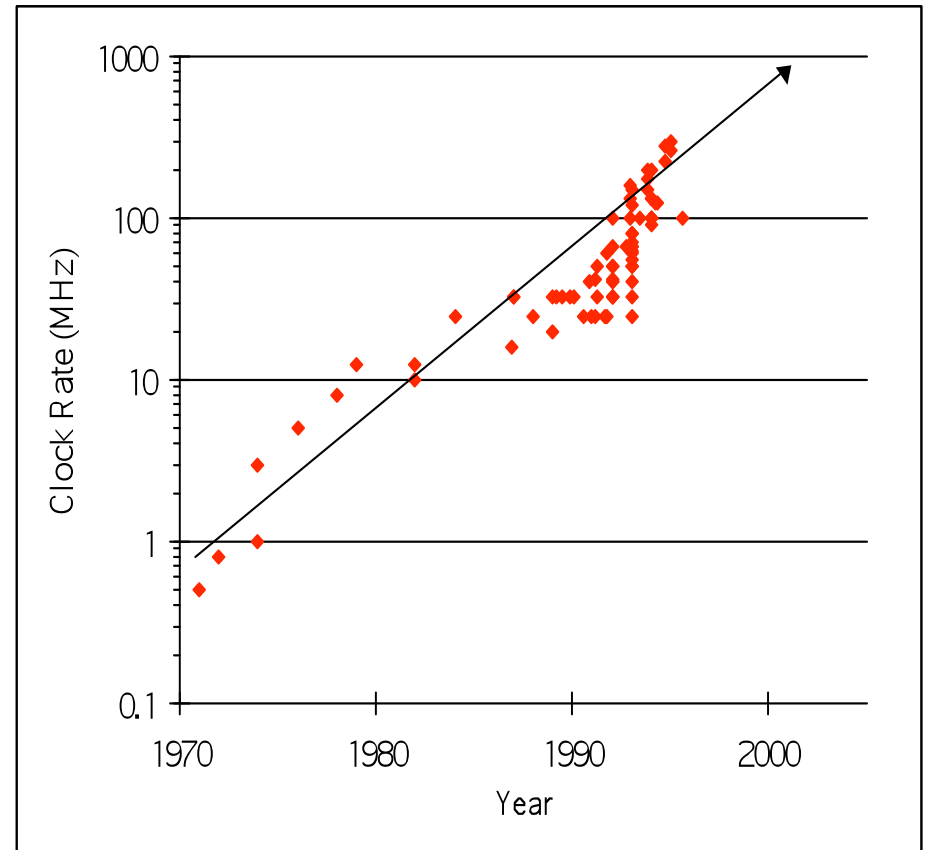
Slide source: Jack Dongarra

Microprocessor Transistors and Clock Rate

Growth in transistors per chip



Increase in clock rate

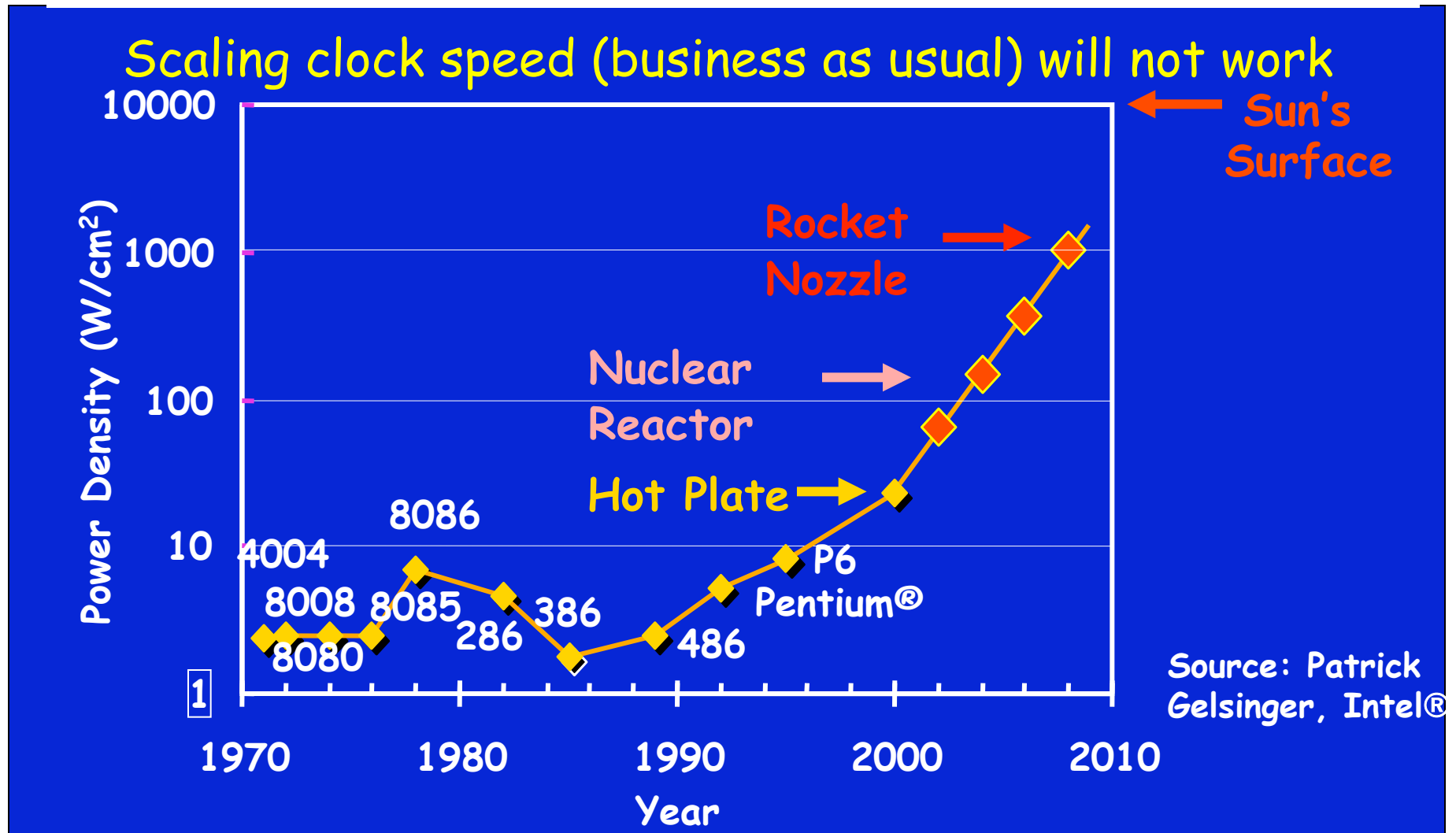


Old view: why bother with parallel programming for performance? Just wait a year or two...

Limit #1: Power density

Can soon put more transistors on a chip than can afford to turn on.

-- Patterson '07



Parallelism Saves Power

- Exploit explicit parallelism for reducing power

$$\text{Power} = (C * V^2 * F)/4 \qquad \text{Performance} = (\text{Cores} * F)*1$$

Capacitance Voltage Frequency

- **Using additional cores**

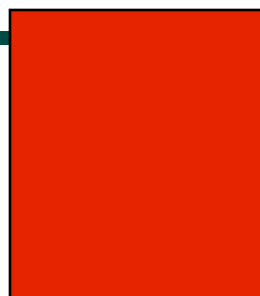
- Increase density (= more transistors = more capacitance)
- Can increase cores (2x) and performance (2x)
- Or increase cores (2x), but decrease frequency & voltage (1/2): same performance at $\frac{1}{4}$ the power

- **Additional benefits**

- Small/simple cores \rightarrow more predictable performance

Limit #2: Speed of Light (Fundamental)

1 Tflop/s, 1
Tbyte
sequential
machine



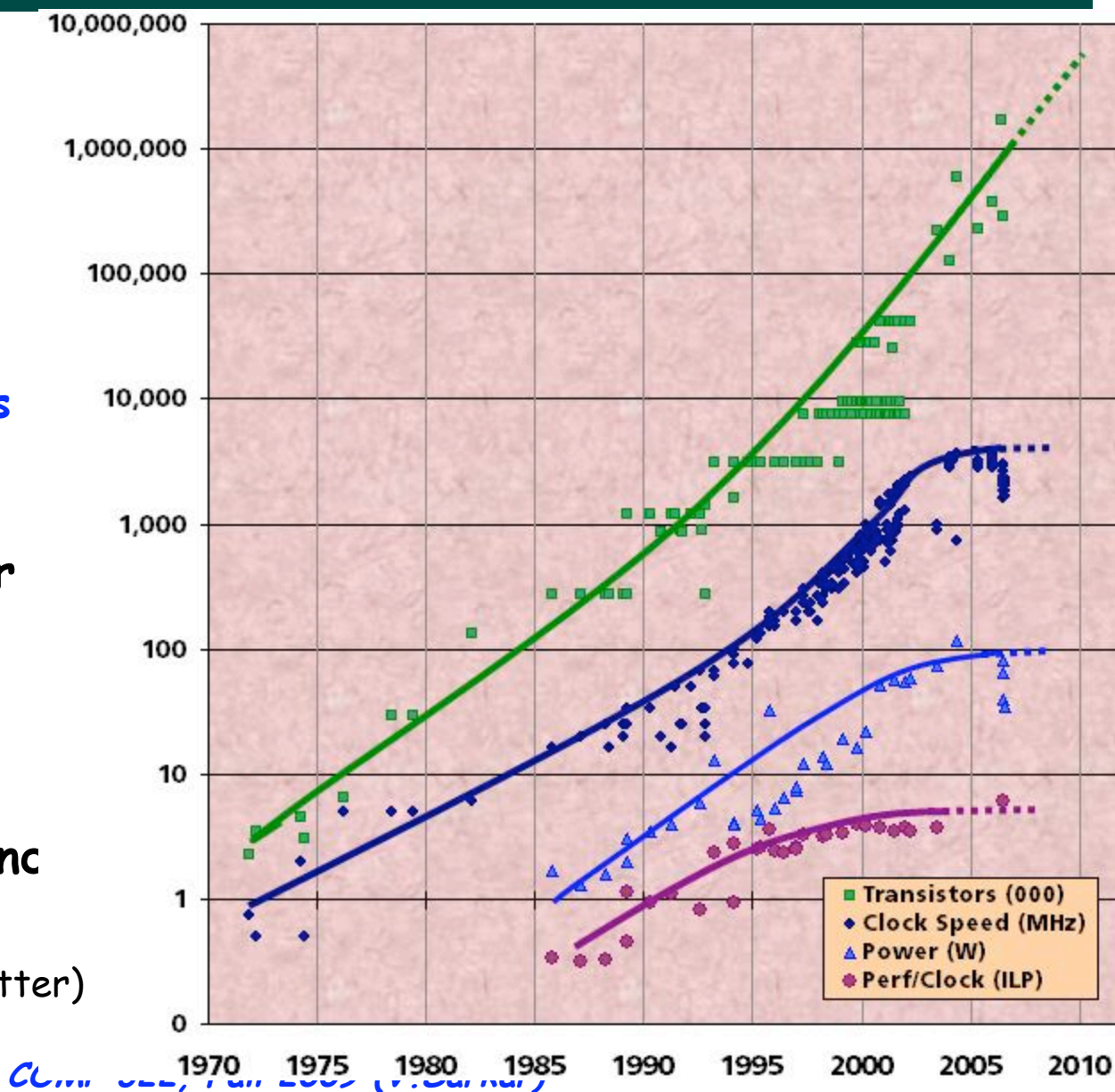
$r = 0.3$
mm

- Consider the 1 Tflop/s sequential machine:
 - Data must travel some distance, r , to get from memory to CPU.
 - To get 1 data element per cycle, this means 10^{12} times per second at the speed of light, $c = 3 \times 10^8$ m/s. Thus $r < c/10^{12} = 0.3$ mm.
- Now put 1 Tbyte of storage in a 0.3 mm \times 0.3 mm area:
 - Each bit occupies about 1 square Angstrom, or the size of a small atom.
- No choice but parallelism

Revolution is Happening Now

- Chip density is continuing increase $\sim 2x$ every 2 years
 - Clock speed is not
 - Number of processor cores may double instead
- There is little or no hidden parallelism (ILP) to be found
- Parallelism must be exposed to and managed by

Source: Intel, Microsoft (Sutter) and Stanford (Olukotun, Hammond)



Multicore in Products

- “We are dedicating all of our future product development to multicore designs. ... This is a sea change in computing”

Paul Otellini, President, Intel (2005)

- All microprocessor companies switch to MP (2X CPUs / 2 yrs)

Manufacturer/Year	AMD/'05	Intel/'06	IBM/'04	Sun/'07
Processors/chip	2	2	2	8
nodes/Processor	1	2	2	16
nodes/chip	2	4	4	128

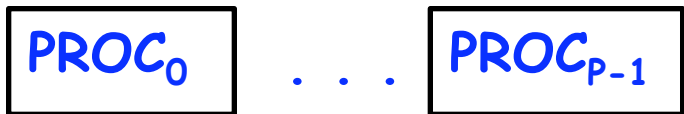
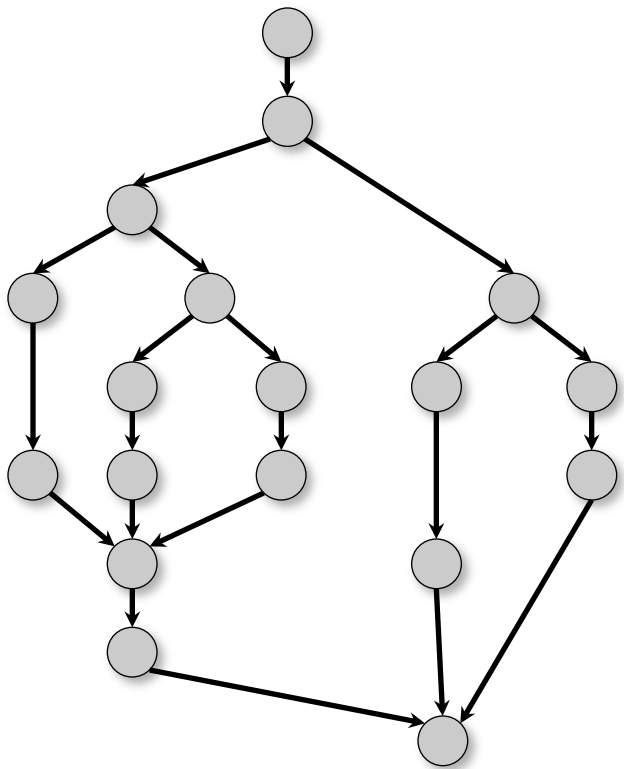
- The STI Cell processor (PS3) has 8 cores
- The latest NVidia Graphics Processing Unit (GPU) has 128 cores
- Intel has demonstrated an 80-core research chip

Implications

- These arguments are no long theoretical
- All major processor vendors are producing multicore chips
 - Every machine will soon be a parallel machine
 - All programmers will be parallel programmers???
- Some may eventually be hidden in libraries, compilers, and high level languages
 - But a lot of work is needed to get there
- Big open questions:
 - What will be the killer apps for multicore machines?
 - How should the chips be designed?
 - How will they be programmed?

Algorithmic Complexity Measures

$T_P =$ execution time on P processors

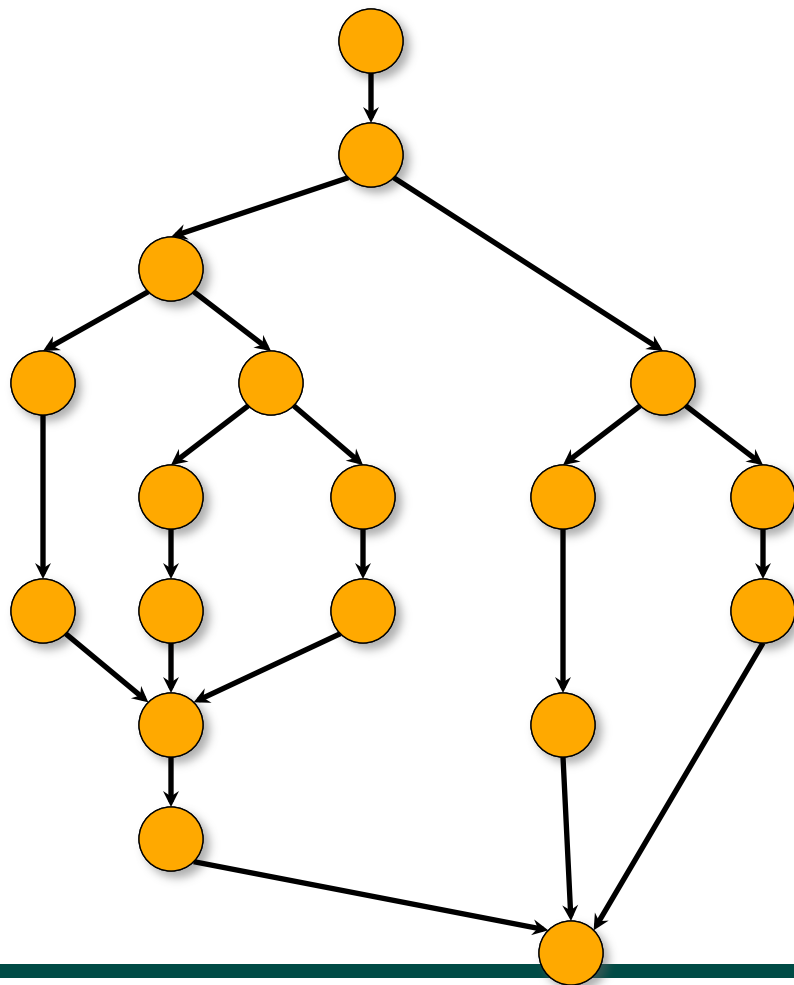


- Computation graph abstraction:*
- Node = arbitrary sequential computation
 - Edge = dependence (successor node can only execute after predecessor node has completed)
 - Directed acyclic graph (dag)

- Processor abstraction:*
- P identical processors
 - Each processor executes one node at a time

Algorithmic Complexity Measures

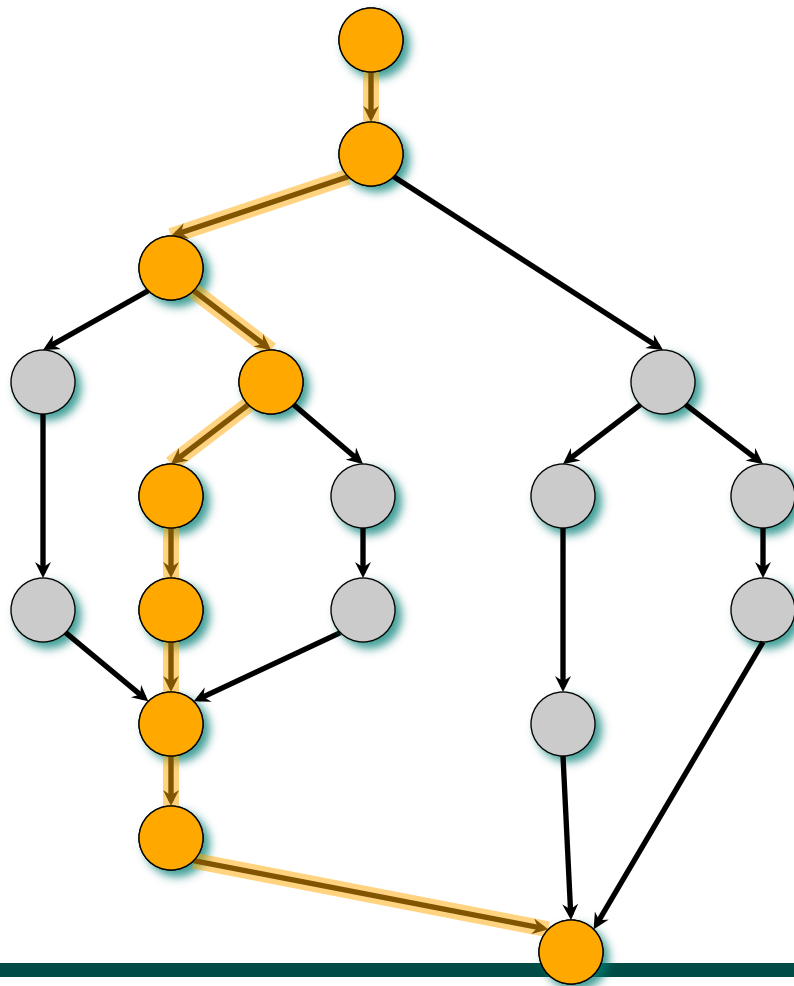
T_P = execution time on P processors



T_1 = work

Algorithmic Complexity Measures

T_P = execution time on P processors

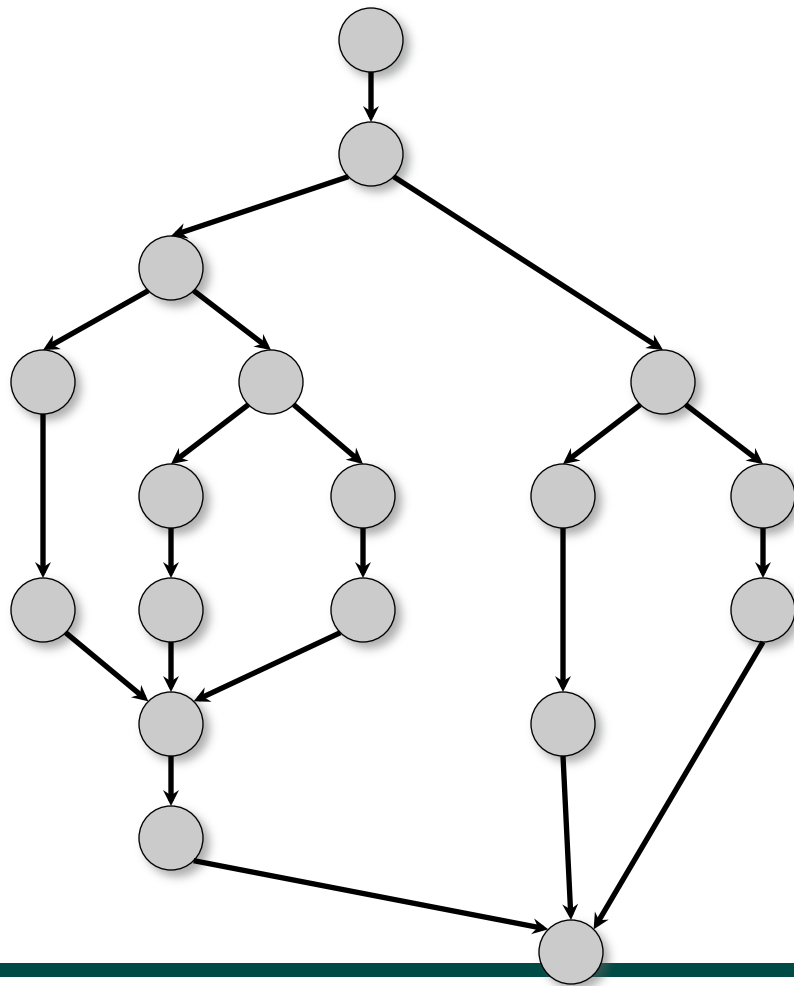


$$T_1 = \text{work}$$
$$T_\infty = \text{span}^*$$

* Also called *critical-path length* or *computational depth*.

Algorithmic Complexity Measures

T_p = execution time on P processors



T_1 = work

T_∞ = span

LOWER BOUNDS

- $T_p \geq T_1/P$
- $T_p \geq T_\infty$

Speedup

Definition: $T_1/T_p = \text{speedup}$ on P processors.

If $T_1/T_p = \Theta(P)$, we have *linear speedup*;
 $= P$, we have *perfect linear speedup*;
 $> P$, we have *superlinear speedup*,

Superlinear speedup is not possible in this model because of the lower bound $T_p \geq T_1/P$, but superlinear speedup can be possible in practice (as we will see later in the course)

Parallelism (“Ideal Speedup”)

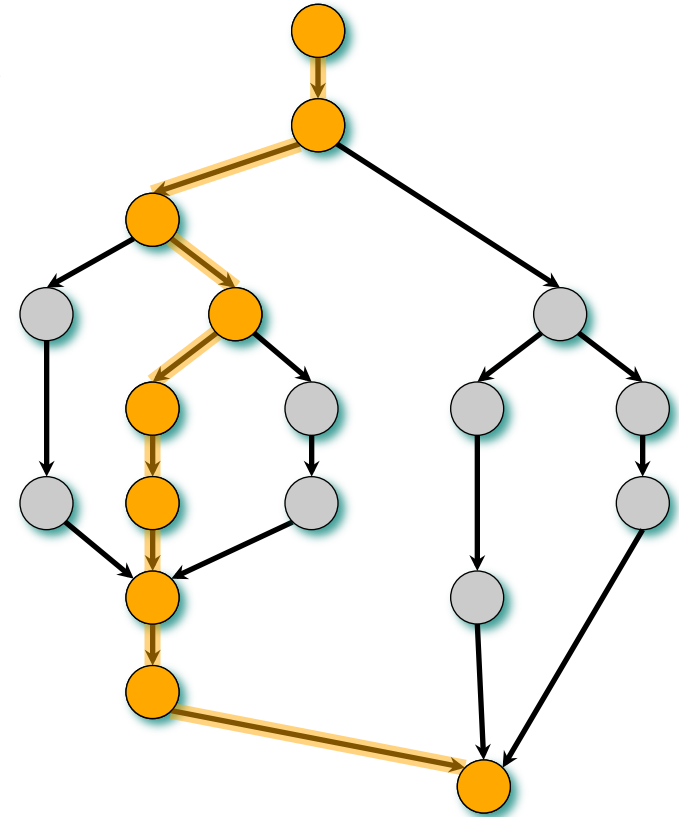
T_p depends on the schedule of computation graph nodes on the processors

→ Two different schedules can yield different values of T_p for the same P

For convenience, define *parallelism* (or ideal speedup) as the ratio T_1/T_∞

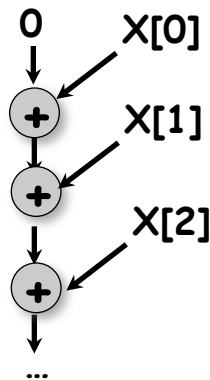
Parallelism is independent of P , and only depends on the computation graph

Also define *parallel slackness* as the ratio, $(T_1/T_\infty)/P$



Example 1: Array Sum (sequential version)

- Problem: compute the sum of the elements $X[0] \dots X[n-1]$ of array X
- Sequential algorithm
 - `sum = 0; for (i=0 ; i < n ; i++) sum += X[i];`
- Computation graph



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

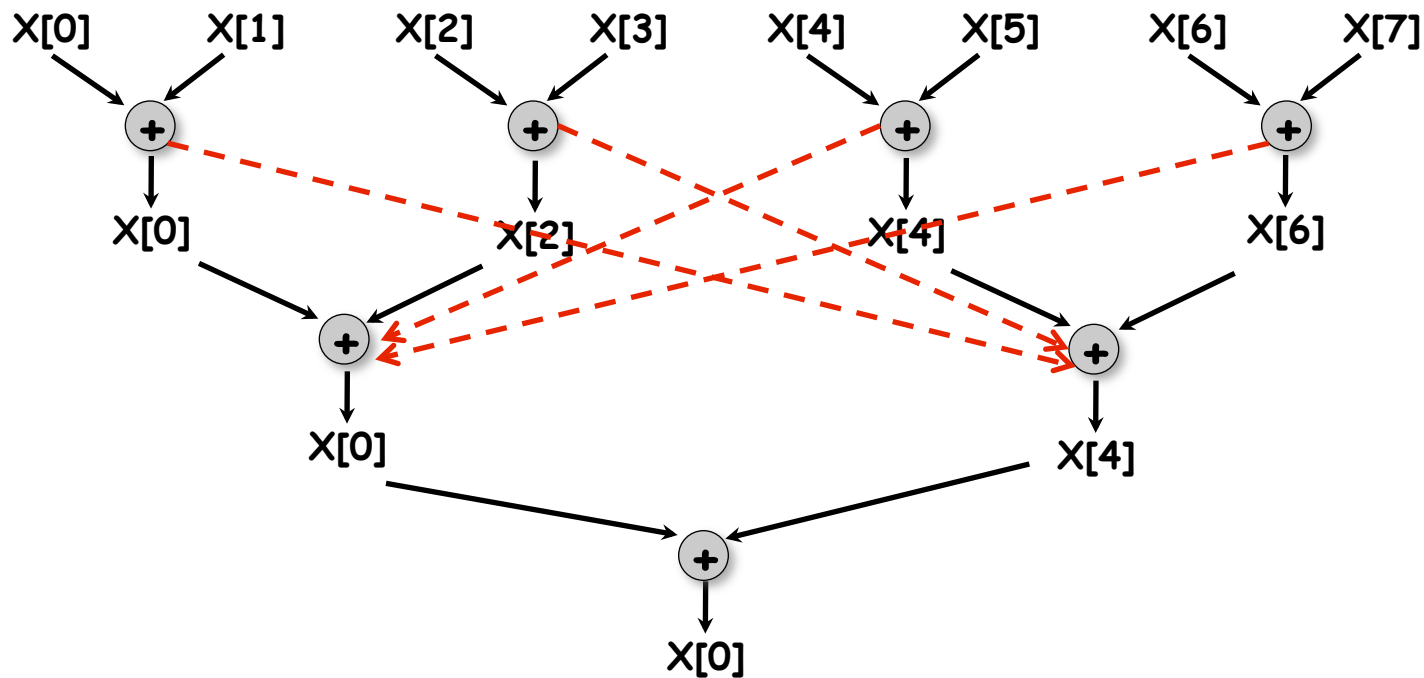
Example 1: Array Sum (parallel iterative version)

- Parallel algorithm (iterative version, assumes n is a power of 2)

```
for ( step = 1; step < n ; step *= 2 ) {  
    final int size = n / (2*step);  
    final int step_f = step;  
    forall ( point [i] : [0:size-1] ) X[2*i*step_f] += X[(2*i+1)*step_f];  
}  
sum = X[0];
```
- HJ forall construct executes all iterations in parallel
 - forall body can only access outer local variables that are final
- This algorithm overwrites X (make a copy if X is needed later)
- Work = $O(n)$, Span = $O(\log n)$, Parallelism = $O(n / (\log n))$
- NOTE: this and the next parallel algorithm can be used for any associative operation on array elements (need not be commutative) e.g., multiplication of an array of matrices

Example 1: Array Sum (parallel iterative version)

- Computation graph for $n = 8$



---> Extra dependence edges due to forall construct

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

Example 1: Array Sum (parallel recursive version)

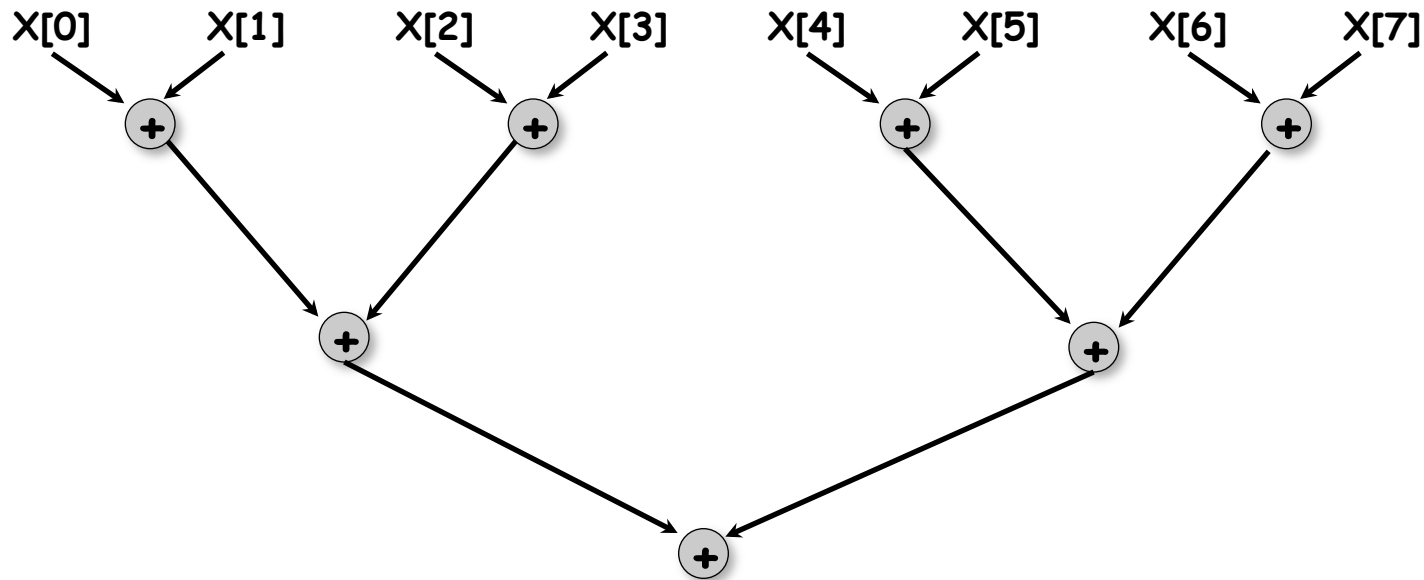
- Parallel algorithm (recursive version, assumes n is a power of 2)

```
sum = computeSum(X, 0, n-1);
int computeSum(final int[] X, final int lo, final int hi) {
    if ( lo > hi ) return 0;
    else if ( lo == hi ) return X[lo];
    else {
        final int mid = (lo+hi)/2;
        final future<int> sum1 =
            async<int> { return computeSum(X, lo, mid); }
        final future<int> sum2 =
            async<int> { return computeSum(X, mid+1, hi); }
        return sum1.force() + sum2.force();
    }
} // computeSum
```

- HJ “async future” executes child expression in parallel with parent
– `force()` causes the parent to wait for the child.
-

Example 1: Array Sum (parallel recursive version)

- Computation graph for $n = 8$



- Work = $O(n)$, Span = $O(\log n)$, Parallelism = $O(n / (\log n))$
- No extra dependences as in forall case

Summary of Today's Lecture

- Introduction to Parallel Computing
- Algorithmic Complexity Measures
- Reading list for next lecture
 - Chapter 1, Introduction
 - Especially, Parallel Prefix Sum on page 13
- Reading list for following lecture (Chapter 2 will be covered later in the semester)
 - Chapter 3, Reasoning about Performance