CS 181E: Fundamentals of Parallel Programming

Instructor: Vivek Sarkar
Co-Instructor: Ran Libeskind-Hadas

http://www.cs.hmc.edu/courses/2012/fall/cs181e/

CS 181E Course Information: Fall 2012

- **"Fundamentals of Parallel Programming"**
- **Lectures:** MW, 4:15pm -- 5:30pm, Parsons 1285
- **Instructor:** Vivek Sarkar (vsarkar@rice.edu)
- **Co-Instructor:** Ran Libeskind-Hadas (hadas@cs.hmc.edu)
- **Grutors:** Matt Prince, Mary Rachel Stimson
- **Habanero Java Support (Rice University):**
  - Vincent Cave (vincent.cave@rice.edu)
  - Shams Imam (shams@rice.edu)
- **Syllabus:** http://www.cs.hmc.edu/courses/2012/fall/cs181e/
  - Bookmark the TWiki page, and start reading lecture handout for Module 1

Outline of Today's Lecture

- <u>Introduction</u>
- Async-Finish Parallel Programming
- Computation Graphs
- Abstract Performance Metrics
- Parallel Array Sum

Acknowledgments for Today's Lecture

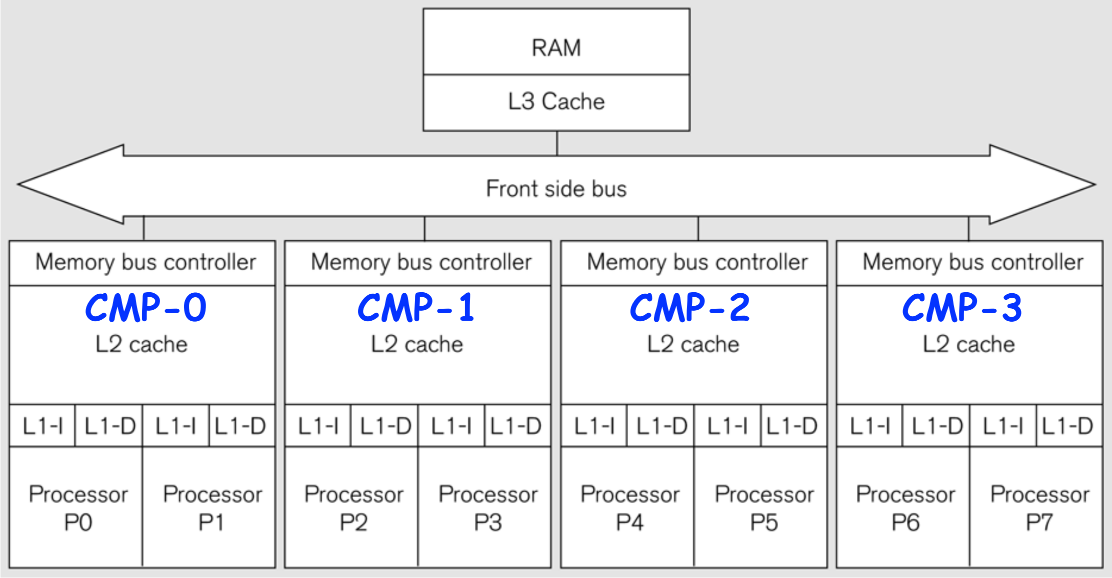
- CS 194 course on "Parallel Programming for Multicore" taught by Prof. Kathy Yelick, UC Berkeley, Fall 2007
  - http://www.cs.berkeley.edu/~yelick/cs194f07/
- "Principles of Parallel Programming", Calvin Lin & Lawrence Snyder, Addison-Wesley 2009
- Cilk lectures, http://supertech.csail.mit.edu/cilk/
- PrimeSieve.java example
  - http://introcs.cs.princeton.edu/java/14array/PrimeSieve.java.html
- CS 181E Module 1 handout

Scope of Course

- Fundamentals of parallel programming
    - Primitive constructs for task creation & termination, collective & point-to-point synchronization, task and data distribution, and data parallelism
    - Abstract models of parallel computations and computation graphs
    - Parallel algorithms & data structures including lists, trees, graphs, matrices
    - Common parallel programming patterns
- Habanero-Java (HJ) language, developed in the Habanero Multicore Software Research project at Rice
- Written assignments
- Programming assignments
    - Abstract metrics
    - Real parallel systems (lab machines + departmental servers)

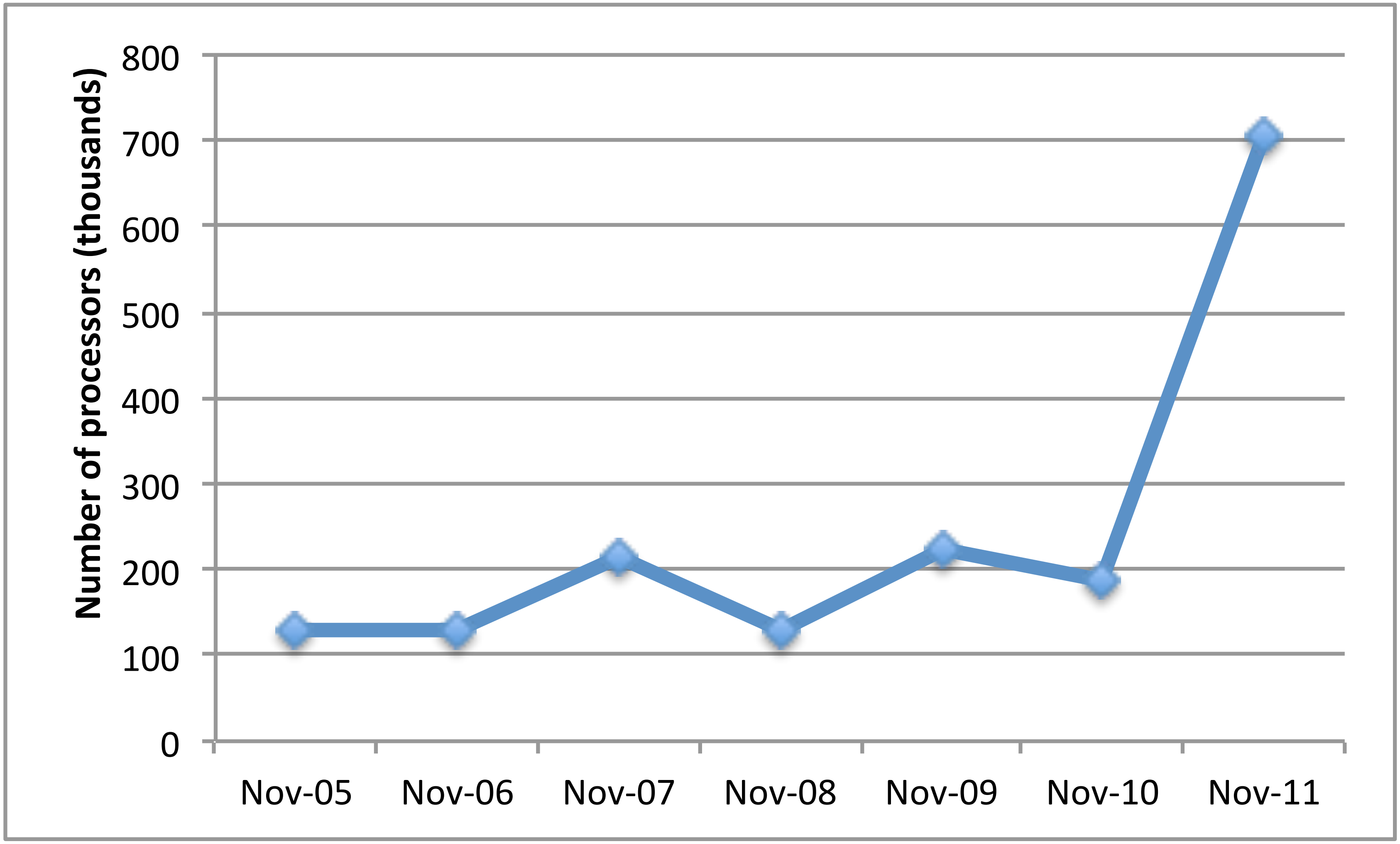
What is Parallel Computing?

- Parallel computing: using multiple processors in parallel to solve problems more quickly than with a single processor and/or with less energy
- Examples of a parallel computer
  - An 8-core Symmetric Multi-Processor (SMP) consisting of four dual-core Chip Multi-Processors (CMPs)



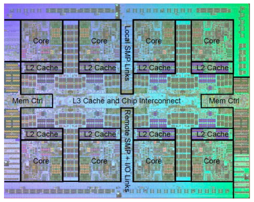
Source: Figure 1.5 of Lin & Snyder book, Addison-Wesley, 2009

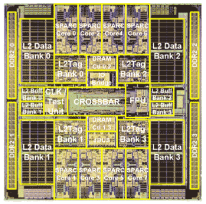
Number of processors in the world's fastest computers during 2005-2011

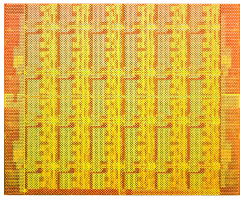


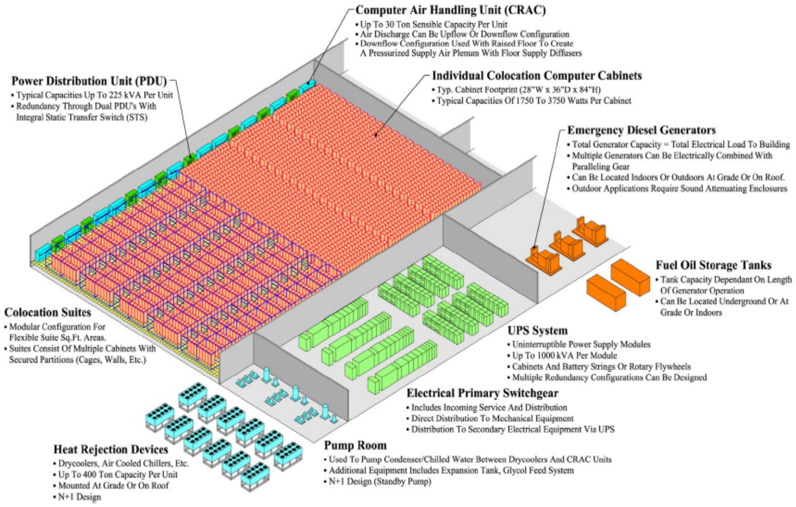
Source: http://www.top500.org

All Computers are Parallel Computers --- Why?









Computer Air Handling Unit (CRAC)
- Up To 30 Ton Sensible Capacity Per Unit
- Air Discharge Can Be Upflow Or Downflow Configuration
- Downflow Configuration Used With Raised Floor To Create A Pressurized Supply Air Plenum With Floor Supply Diffusers

Power Distribution Unit (PDU)
- Typical Capacities Up To 225 kVA Per Unit
- Redundancy Through Dual PDU's With Integral Static Transfer Switch (STS)

Individual Colocation Computer Cabinets
- Typ. Cabinet Footprint (28"W x 36"D x 84"H)
- Typical Capacities Of 1750 To 3750 Watts Per Cabinet

Emergency Diesel Generators
- Total Generator Capacity = Total Electrical Load To Building
- Multiple Generators Can Be Electrically Combined With Paralleling Gear
- Can Be Located Indoors Or Outdoors At Grade Or On Roof.
- Outdoor Applications Require Sound Attenuating Enclosures

Fuel Oil Storage Tanks
- Tank Capacity Dependant On Length Of Generator Operation
- Can Be Located Underground Or At Grade Or Indoors

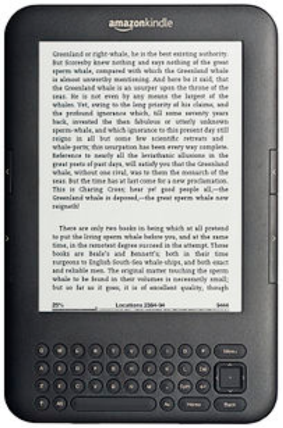
Colocation Suites
- Modular Configuration For Flexible Suite Sq.Ft. Areas.
- Suites Consist Of Multiple Cabinets With Secured Partitions (Cages, Walls, Etc.)

UPS System
- Uninterruptible Power Supply Modules
- Up To 1000 kVA Per Module
- Cabinets And Battery Strings Or Rotary Flywheels
- Multiple Redundancy Configurations Can Be Designed

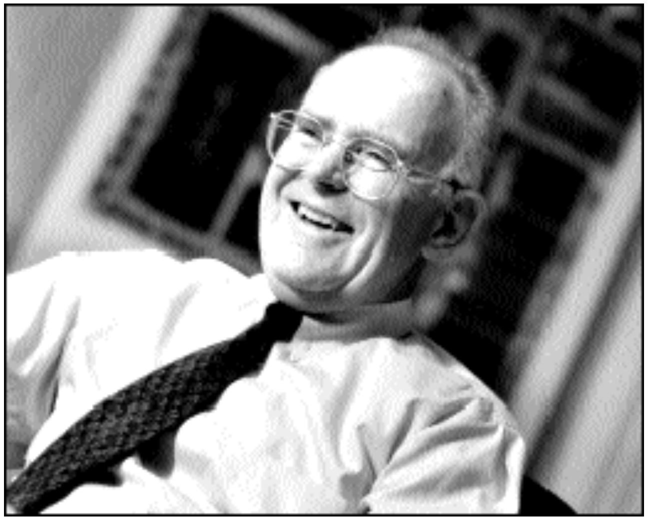
Electrical Primary Switchgear
- Includes Incoming Service And Distribution
- Direct Distribution To Mechanical Equipment
- Distribution To Secondary Electrical Equipment Via UPS

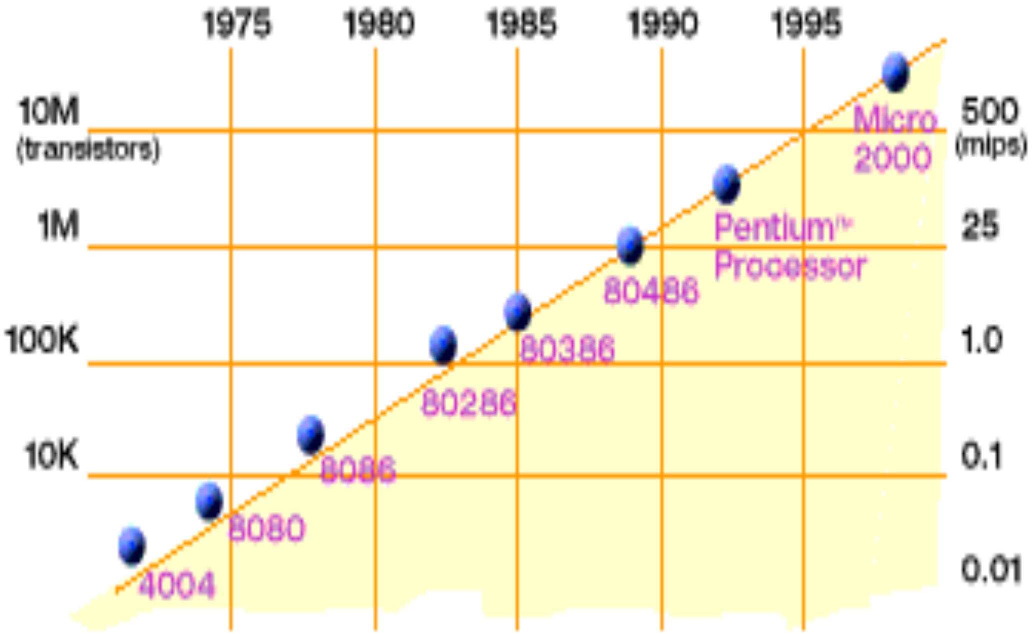
Heat Rejection Devices
- Drycoolers, Air Cooled Chillers, Etc.
- Up To 400 Ton Capacity Per Unit
- Mounted At Grade Or On Roof
- N+1 Design

Pump Room
- Used To Pump Condenser/Chilled Water Between Drycoolers And CRAC Units
- Additional Equipment Includes Expansion Tank, Glycol Feed System
- N+1 Design (Standby Pump)



Moore's Law





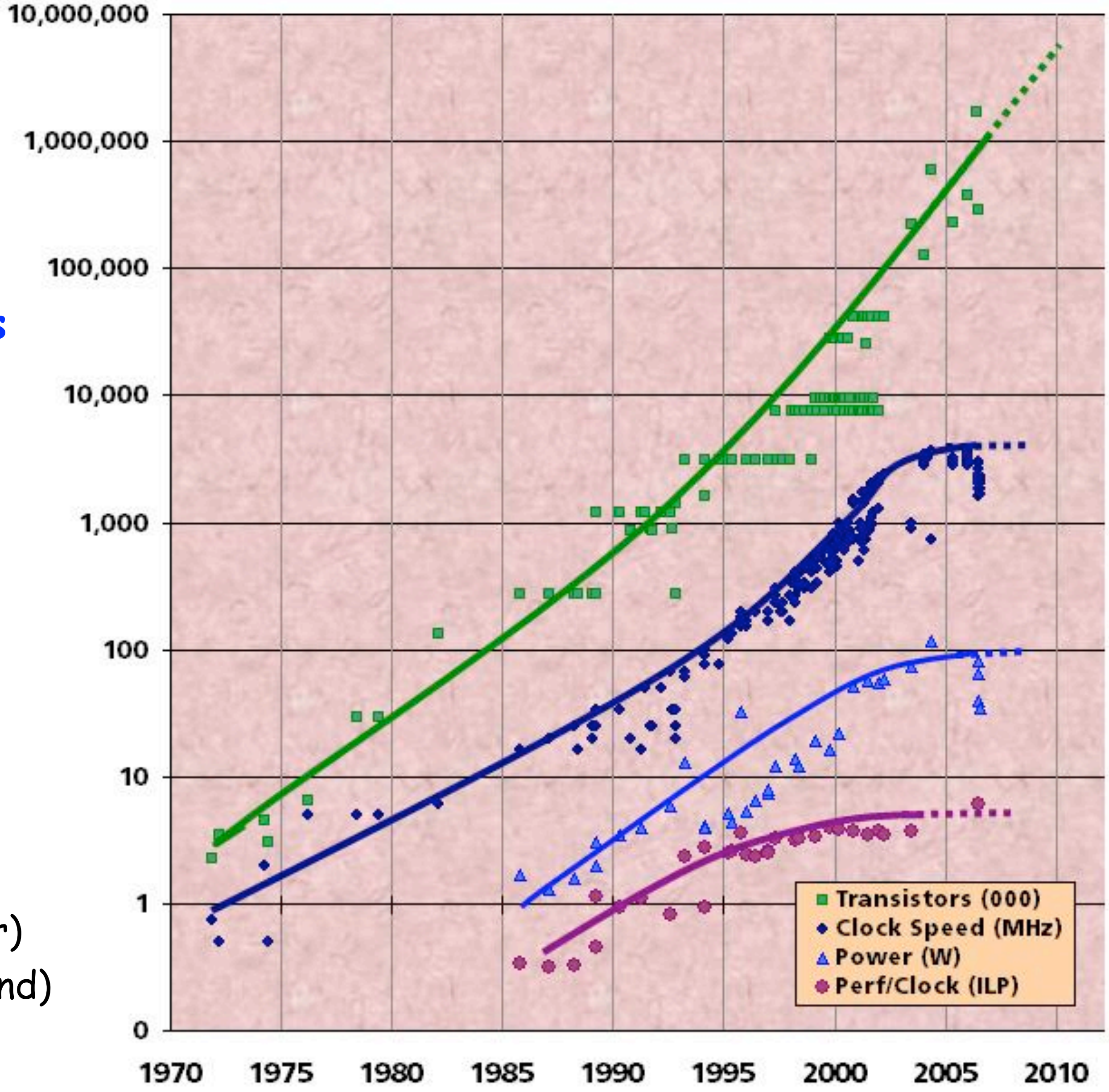
Gordon Moore (co-founder of Intel) predicted in 1965 that the transistor density of semiconductor chips would double roughly every 1-2 years

Resulted in CPU clock speed doubling roughly every 18 months, but not any longer

Slide source: Jack Dongarra

Current Technology Trends

- **Chip density is continuing to increase ~2x every 2 years**
  - Clock speed is not
  - Number of processors is doubling instead
- **Parallelism must be managed by software**



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

Parallelism Saves Power (Simplified Analysis)

Power = (Capacitance) * (Voltage)^2 * (Frequency)

➔ Power α (Frequency)^3

<u>Baseline example</u>: single 1GHz core with power P

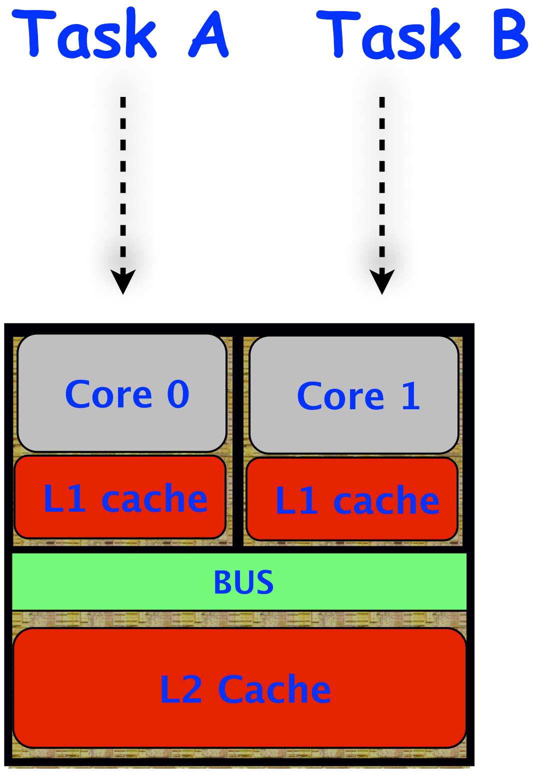
<u>Option A</u>: Increase clock frequency to 2GHz ➔ Power = 8P

<u>Option B</u>: Use 2 cores at 1 GHz each ➔ Power = 2P

- Option B delivers same performance as Option A with 4x less power … provided software can be decomposed to run in parallel!

What is Parallel Programming?

- Specification of operations that can be executed in parallel
- A parallel program is decomposed into sequential subcomputations called <u>tasks</u>
- Parallel programming constructs define task creation, termination, and interaction

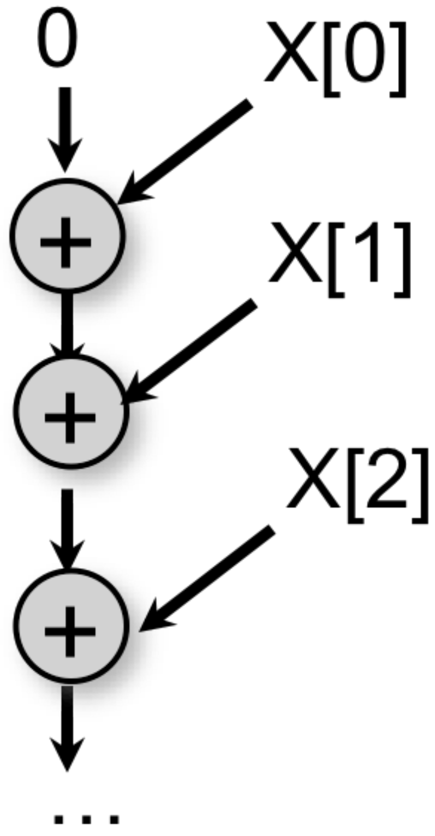


Schematic of a dual-core Processor

Example of a Sequential Program: Computing the sum of array elements

```
int sum = 0;
for (int i=0 ; i < X.length ; i++)
    sum += X[i];
```

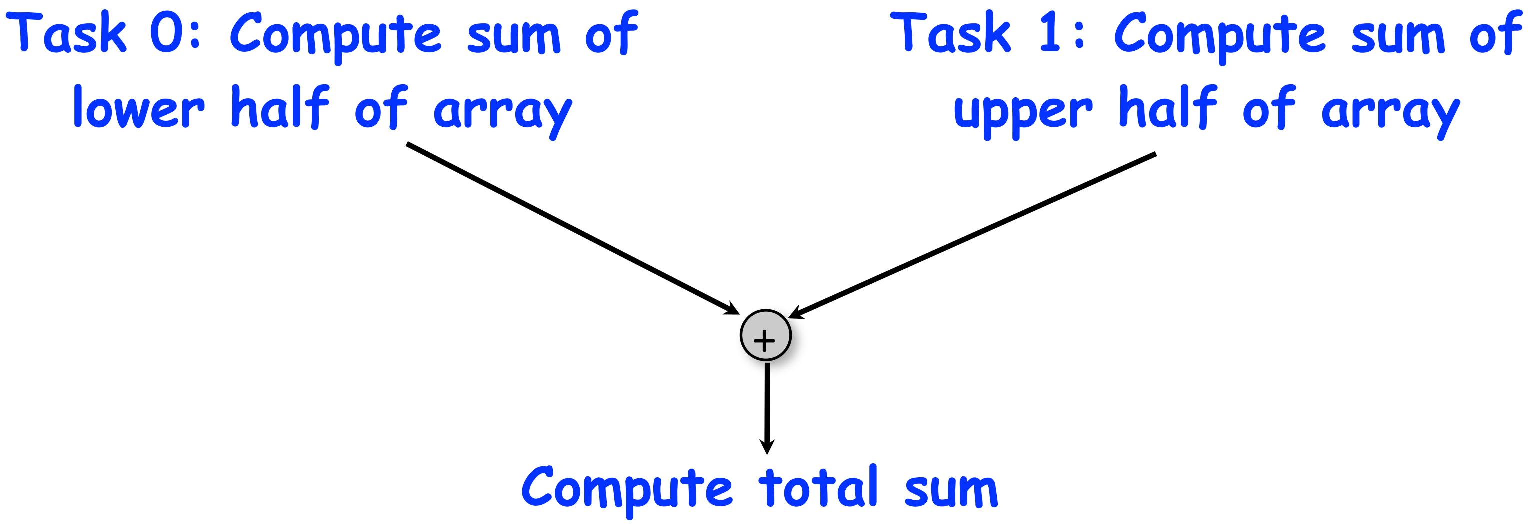
Computation Graph



Observations:

- The decision to sum up the elements from left to right was arbitrary
- The computation graph shows that all operations must be executed sequentially

Parallelization Strategy for two cores (Two-way Parallel Array Sum)



Task 0: Compute sum of lower half of array

Task 1: Compute sum of upper half of array

Compute total sum

Basic idea:

- Decompose problem into two tasks for partial sums
- Combine results to obtain final answer
- Parallel divide-and-conquer pattern

Outline of Today's Lecture

- Introduction
- <u>Async-Finish Parallel Programming</u>
- Computation Graphs
- Abstract Performance Metrics
- Parallel Array Sum

Async and Finish Statements for Task Creation and Termination

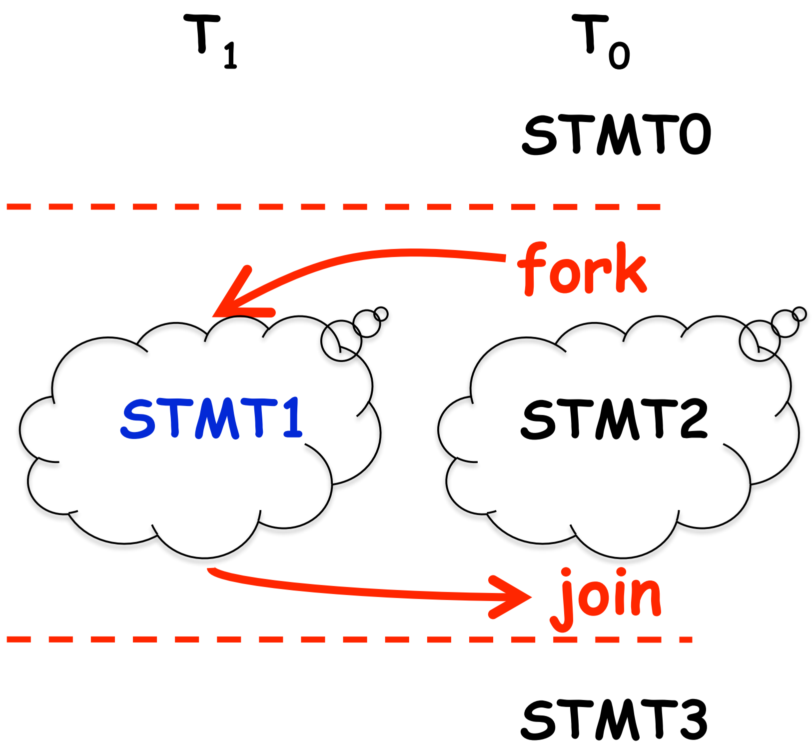
async S

- Creates a new child task that executes statement S

finish S

- Execute S, but wait until *all* asyncs in S's scope have terminated.

```
// T0(Parent task)
STMT0;
finish {     //Begin finish
  async {
    STMT1; //T1(Child task)
  }
  STMT2;     //Continue in T0
              //Wait for T1
}            //End finish
STMT3;       //Continue in T0
```



Two-way Parallel Array Sum using async & finish constructs

```
// Start of Task T0 (main program)
sum1 = 0; sum2 = 0; // sum1 & sum2 are static fields
async { // Task T1 computes sum of upper half of array
  for(int i=X.length/2; i < X.length; i++)
    sum2 += X[i];
}
// T0 computes sum of lower half of array
for(int i=0; i < X.length/2; i++) sum1 += X[i];
// Task T0 waits for Task T1 (join)
return sum1 + sum2;
```

Where does finish go?
Time for worksheet #1!

Some Properties of Async & Finish constructs

1. Scope of async/finish can be any arbitrary statement
   - async/finish constructs can be arbitrarily nested e.g.,
   - `finish { async S1; finish { async S2; S3; } S4; } S5;`
2. A method may return before all its async's have terminated
   - Enclose method body in a finish if you don't want this to happen
   - main() method is enclosed in an implicit finish e.g.,
   - `main(){ foo();}  void foo() {async S1; S2; return;}`
3. Each dynamic async task will have a unique Immediately Enclosing Finish (IEF) at runtime
4. Async/finish constructs cannot "deadlock"
   - Cannot have a situation where both task A waits for task B to finish, and task B waits for task A to finish
5. Async tasks can read/write shared data via objects and arrays
   - Local variables have special restrictions (next slide)

Local Variables

Three rules for accessing local variables across tasks in HJ:

1) *An async may read the value of any final outer local var*

```
final int i1 = 1; async { ... = i1; /* i1=1 */ }
```

2) *An async may read the value of any non-final outer local var (copied on entry to async like method parameters)*

```
int i2 = 2; // i2=2 is  copied on entry to the async
async { ... = i2; /* i2=2*/}
i2 = 3; // This assignment is not seen by the above async
```

3) *An async is not permitted to modify an outer local var*

```
int[] A; async { A = ...; /*ERROR*/  A[i] = ...; /*OK*/ }
```

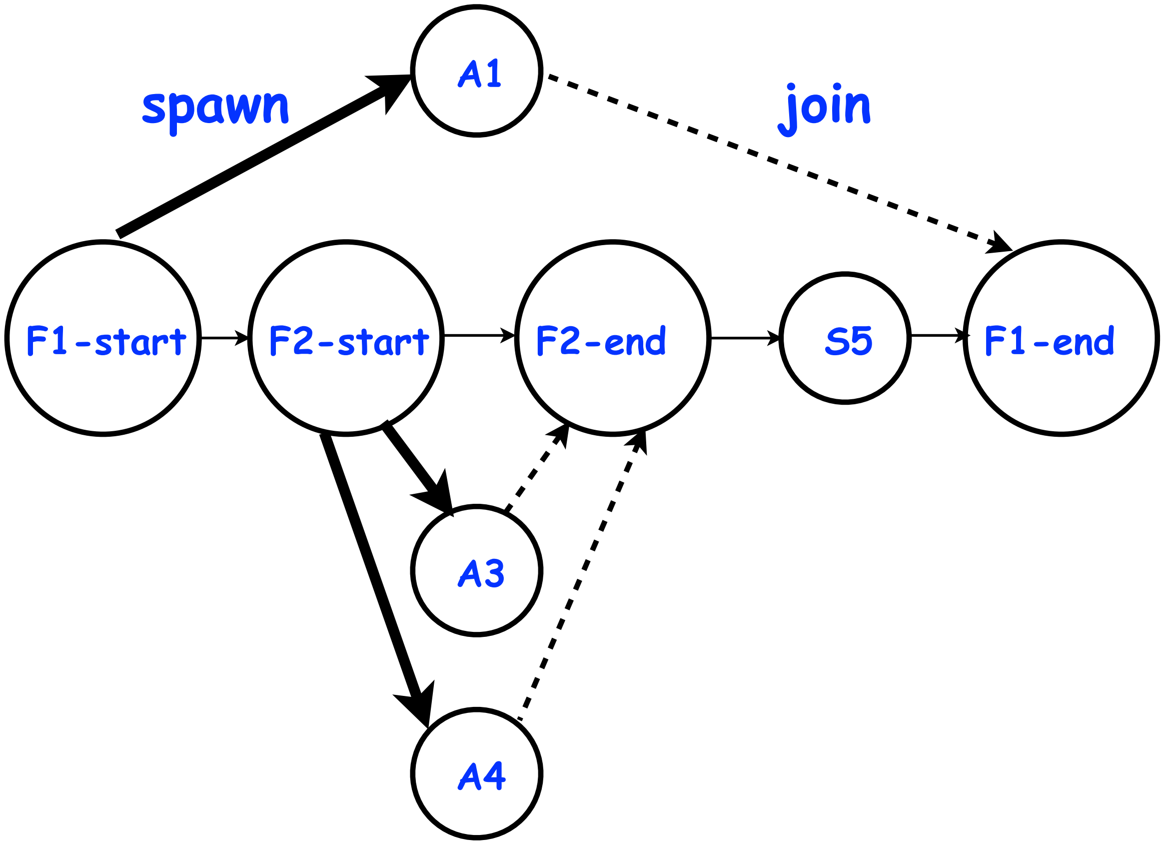
Outline of Today's Lecture

- Introduction
- Async-Finish Parallel Programming
- <u>Computation Graphs</u>
- Abstract Performance Metrics
- Parallel Array Sum

Which statements can potentially be executed in parallel with each other?

```
finish { // F1
  async A1;
    finish { // F2
      async A3;
      async A4;
    } // F2
  S5;
} // F1
```

Computation Graph



Computation Graphs for HJ Programs

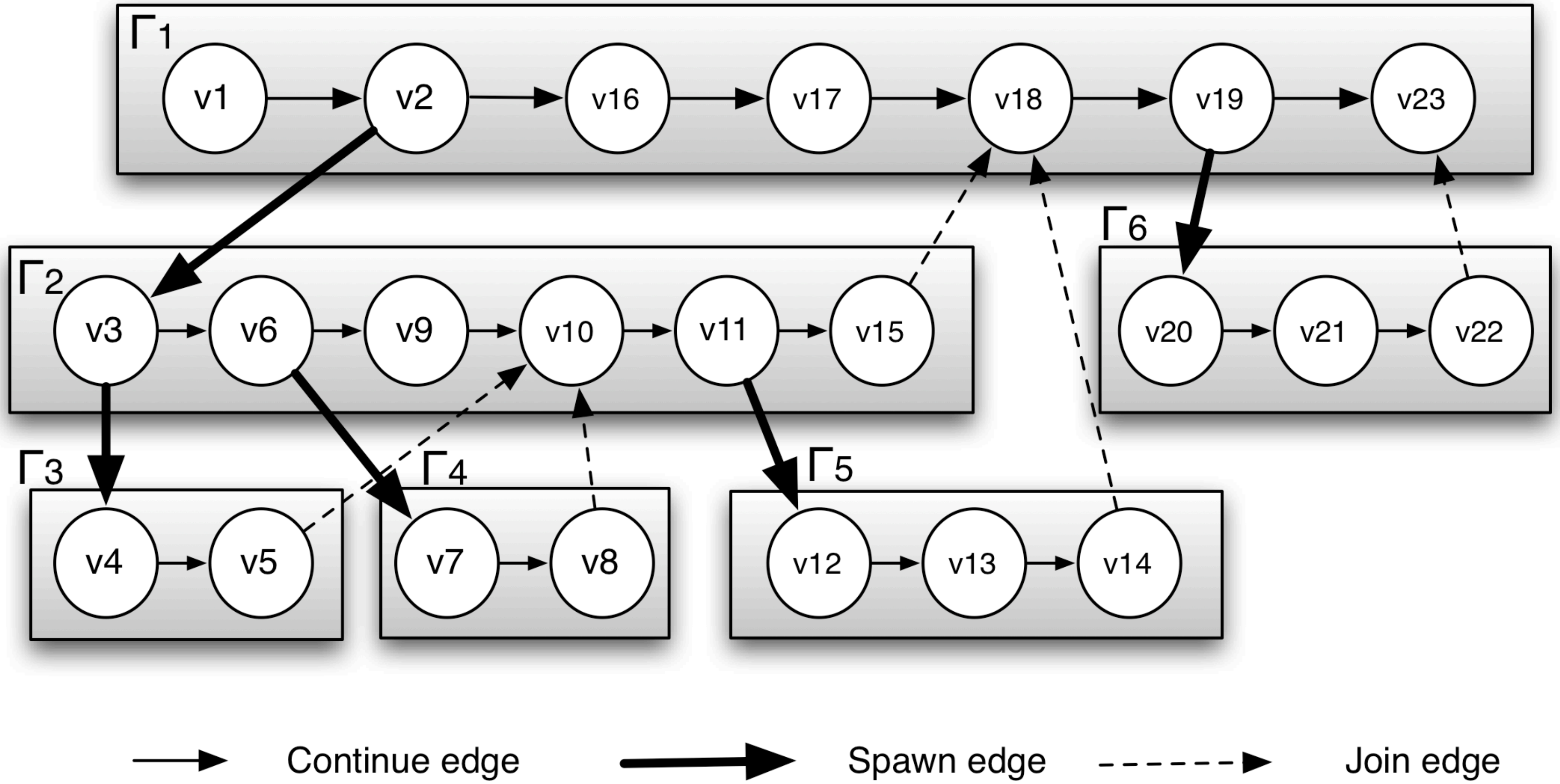
- A Computation Graph (CG) captures the dynamic execution of an HJ program, for a specific input
- CG nodes are "steps" in the program's execution
  - A step is a sequential subcomputation without any async, begin-finish and end-finish operations
- CG edges represent ordering constraints
  - "Continue" edges define sequencing of steps within a task
  - "Spawn" edges connect parent tasks to child async tasks
  - "Join" edges connect the end of each async task to its IEF's end-finish operations
- All computation graphs must be acyclic
  - It is not possible for a node to depend on itself
- Computation graphs are examples of "directed acyclic graphs" (dags)

Example HJ Program with statements v1 … v23

```
// Task T1
v1; v2;
finish {
  async {
    // Task T2
    v3;
    finish {
      async { v4; v5; } // Task T3
      v6;
      async { v7; v8; } // Task T4
      v9;
    } // finish
    v10; v11;
```

```
// Task T2 (contd)
    async { v12; v13;
            v14; } // Task T5
    v15;
  } // end of task T2
  v16; v17; // back in Task T1
} // finish
v18; v19;
finish {
  async {
    // Task T6
    v20; v21; v22; }
}
v23;
```

Computation Graph for previous HJ Example



Example: Step v16 can potentially execute in parallel with steps v3 … v15

Outline of Today's Lecture

- Introduction
- Async-Finish Parallel Programming
- Computation Graphs
- Abstract Performance Metrics
- Parallel Array Sum

Complexity Measures for Computation Graphs

Define

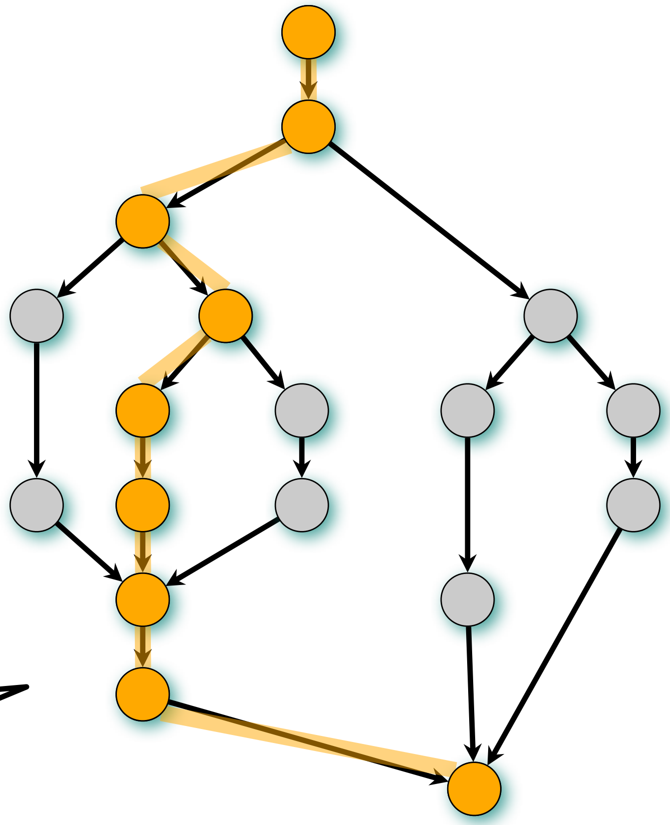
- TIME(N) = execution time of node N
- WORK(G) = sum of TIME(N), for all nodes N in CG G
  - WORK(G) is the total work to be performed in G
- CPL(G) = length of a longest path in CG G, when adding up execution times of all nodes in the path
  - Such paths are called critical paths
  - CPL(G) is the length of these paths (critical path length)

Ideal Speedup

Define ideal speedup of Computation G Graph as the ratio, WORK(G)/CPL(G)

Ideal Speedup is independent of the number of processors that the program executes on, and only depends on the computation graph

What is the ideal speedup of this graph?
Time for worksheet #2!



Lower Bounds on Execution Time

- Let T_P = execution time of computation graph on P processors
  - Assume an idealized machine where node N takes TIME(N) regardless of which processor it executes on, and that there is no overhead for creating parallel tasks
- Observations
  - $T_1 = WORK(G)$
  - $T_\infty = CPL(G)$
- Lower bounds
  - Capacity bound: $T_P \geq WORK(G)/P$
  - Critical path bound: $T_P \geq CPL(G)$
- Putting them together
  - $T_P \geq \max(WORK(G)/P, CPL(G))$

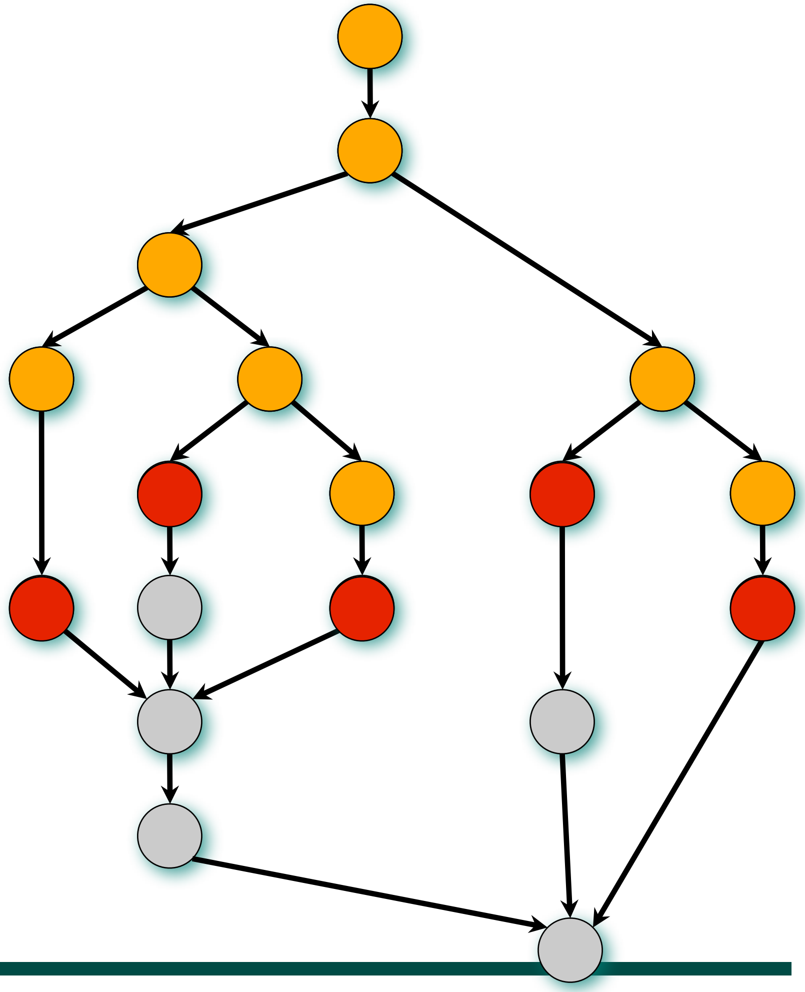
Upper Bound for Greedy Scheduling

Theorem [Graham '66]. Any "greedy scheduler" achieves

$$T_P \leq WORK(G)/P + CPL(G)$$

- A greedy scheduler is one that never forces a processor to be idle when one or more nodes are ready for execution

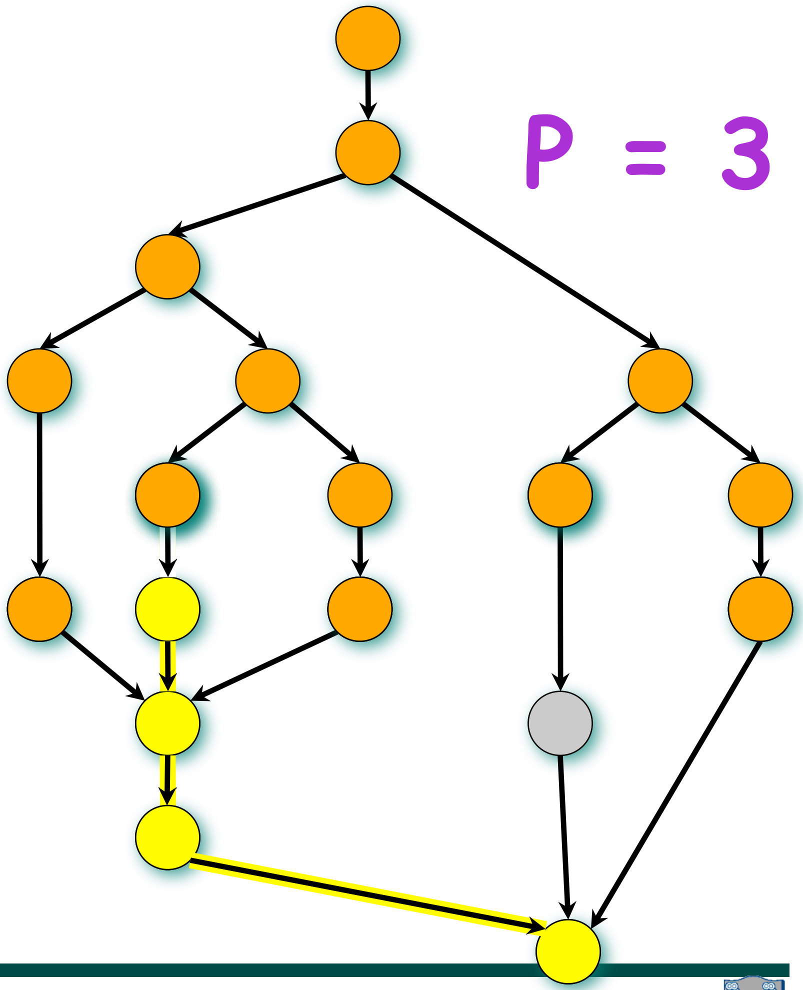
- A node is ready for execution if all its predecessors have been executed



Upper Bound on Execution Time: Greedy-Scheduling Theorem

Theorem [Graham '66]. Any greedy scheduler achieves

$$T_P \leq WORK(G)/P + CPL(G)$$



P = 3

Proof sketch:

- Define a time step to be complete if $\geq$ P nodes are ready at that time, or incomplete otherwise

\# complete time steps $\leq$ WORK(G)/P, since each complete step performs P work.

\# incomplete time steps $\leq$ CPL(G), since each incomplete step reduces the span of the unexecuted dag by 1.

Optimality of Greedy Schedulers

Combine lower and upper bounds to get

$\max(\mathrm{WORK}(G)/P, \mathrm{CPL}(G)) \leq T_P \leq \mathrm{WORK}(G)/P + \mathrm{CPL}(G)$

Corollary 1: Any greedy scheduler achieves execution time T_P that is within a factor of 2 of the optimal time (since max(a,b) and (a+b) are within a factor of 2 of each other, for any $a \geq 0, b \geq 0$).

Corollary 2: Lower and upper bounds approach the same value whenever

- There's lots of parallelism, WORK(G)/CPL(G) >> P
- Or there's little parallelism, WORK(G)/CPL(G) << P

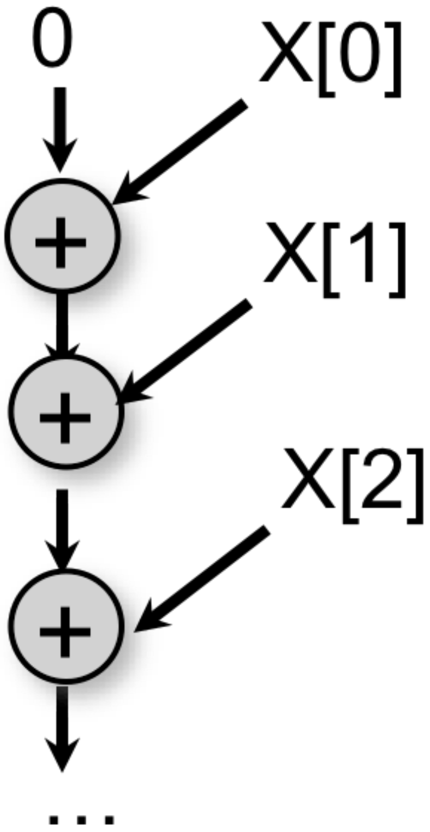
Outline of Today's Lecture

- Introduction
- Async-Finish Parallel Programming
- Computation Graphs
- Abstract Performance Metrics
- <u>Parallel Array Sum</u>

Sequential Array Sum Program

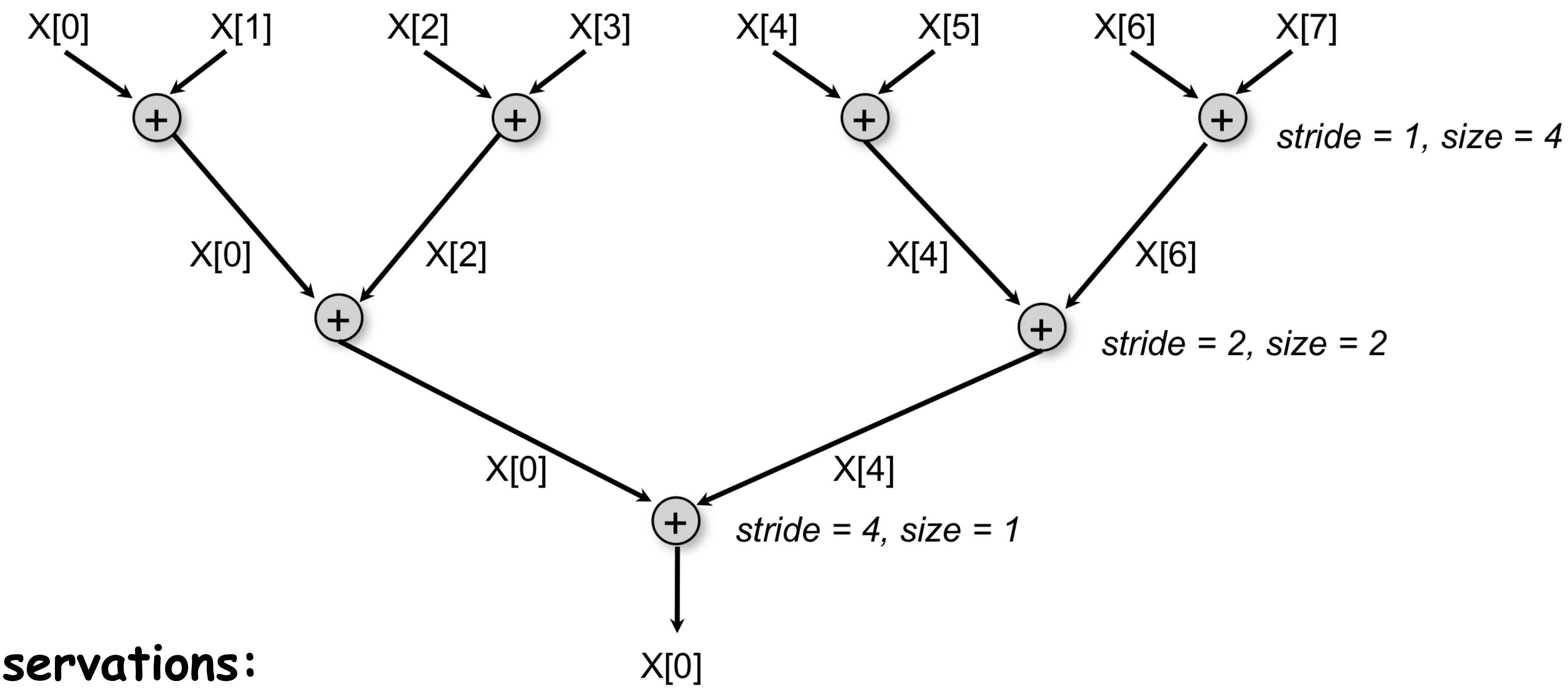
```
int sum = 0;
for (int i=0 ; i < X.length ; i++ )
    sum += X[i];
```

Computation Graph



- The original computation graph is sequential
- We studied a 2-task parallel program for this problem
- How can we expose more parallelism?

Reduction Tree Schema for computing Array Sum in parallel



Observations:

- This algorithm overwrites X (make a copy if X is needed later)
- stride = distance between array subscript inputs for each addition
- size = number of additions that can be executed in parallel in each level (stage)

CS 181E Course Information: Fall 2012

- **"Fundamentals of Parallel Programming"**
- **Lectures:** MW, 4:15pm -- 5:30pm, Parsons 1285
- **Syllabus:** http://www.cs.hmc.edu/courses/2012/fall/cs181e/
  - Bookmark the TWiki page, and start reading lecture handout for Module 1
- **Course Requirements:**

Homeworks (6)	70%
Final Exam	20%
Class Participation	10%

- **HW0 is assigned today and is due on Tuesday, Sep 11th**

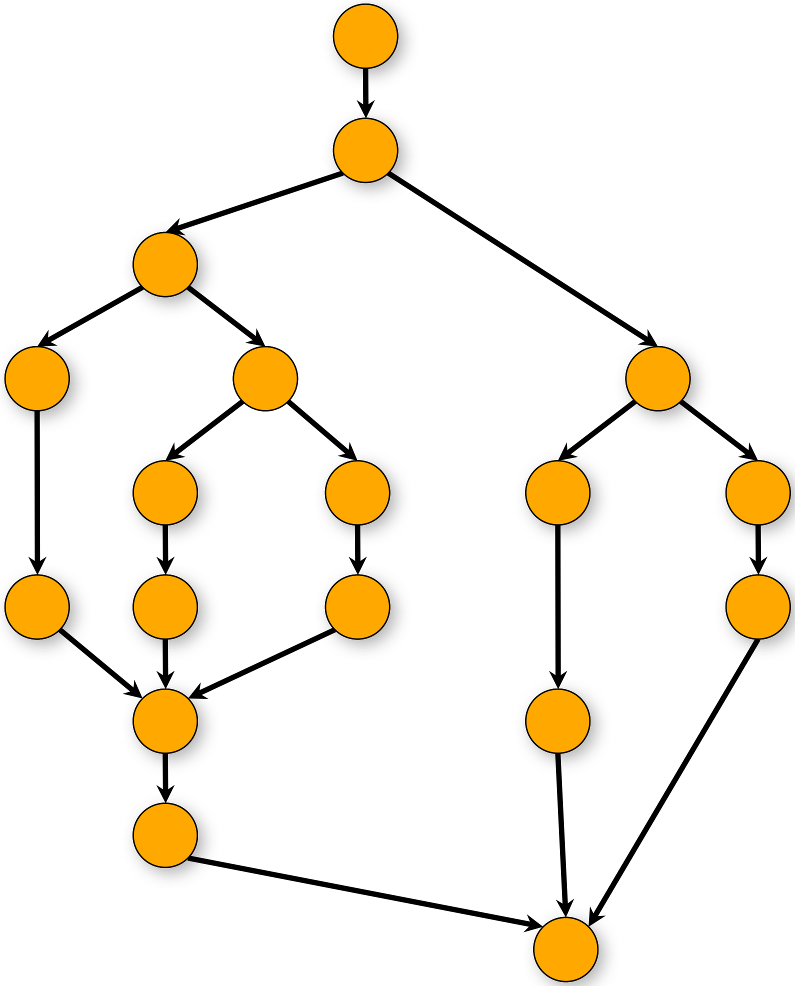
Worksheet #1: Insert finish to get correct Two-way Parallel Array Sum program

Your name: ______________________________

```
// Start of Task T0 (main program)
sum1 = 0; sum2 = 0; // sum1 & sum2 are static fields
async { // Task T1 computes sum of upper half of array
  for(int i=X.length/2; i < X.length; i++)
    sum2 += X[i];
}
// T0 computes sum of lower half of array
for(int i=0; i < X.length/2; i++) sum1 += X[i];
// Task T0 waits for Task T1 (join)
return sum1 + sum2;
```

Worksheet #2: what is the critical path length and ideal speedup of this graph?

- Assume time(N) = 1 for all nodes in this graph



WORK(G) = 18