COMP 322: Fundamentals of Parallel Programming

Lecture 3: Computation Graphs and Abstract Performance Metrics

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

Acknowledgments for Today's Lecture

- Cilk lectures, http://supertech.csail.mit.edu/cilk/
- COMP 322 Lecture 3 handout

Computation Graphs for HJ Programs

- A Computation Graph (CG) is an abstract data structure that captures the dynamic execution of an HJ program
- The nodes in the CG are *steps* in the program's execution
  - —A step is a sequential subcomputation of a task that contains no continuation points
  - —When a worker starts executing a step, it can execute the entire step without interruption
  - —Steps need not be maximal i.e., it is acceptable to split a step into smaller steps if so desired

Example HJ Program Decomposed into Non-Maximal Steps (v1 … v23)

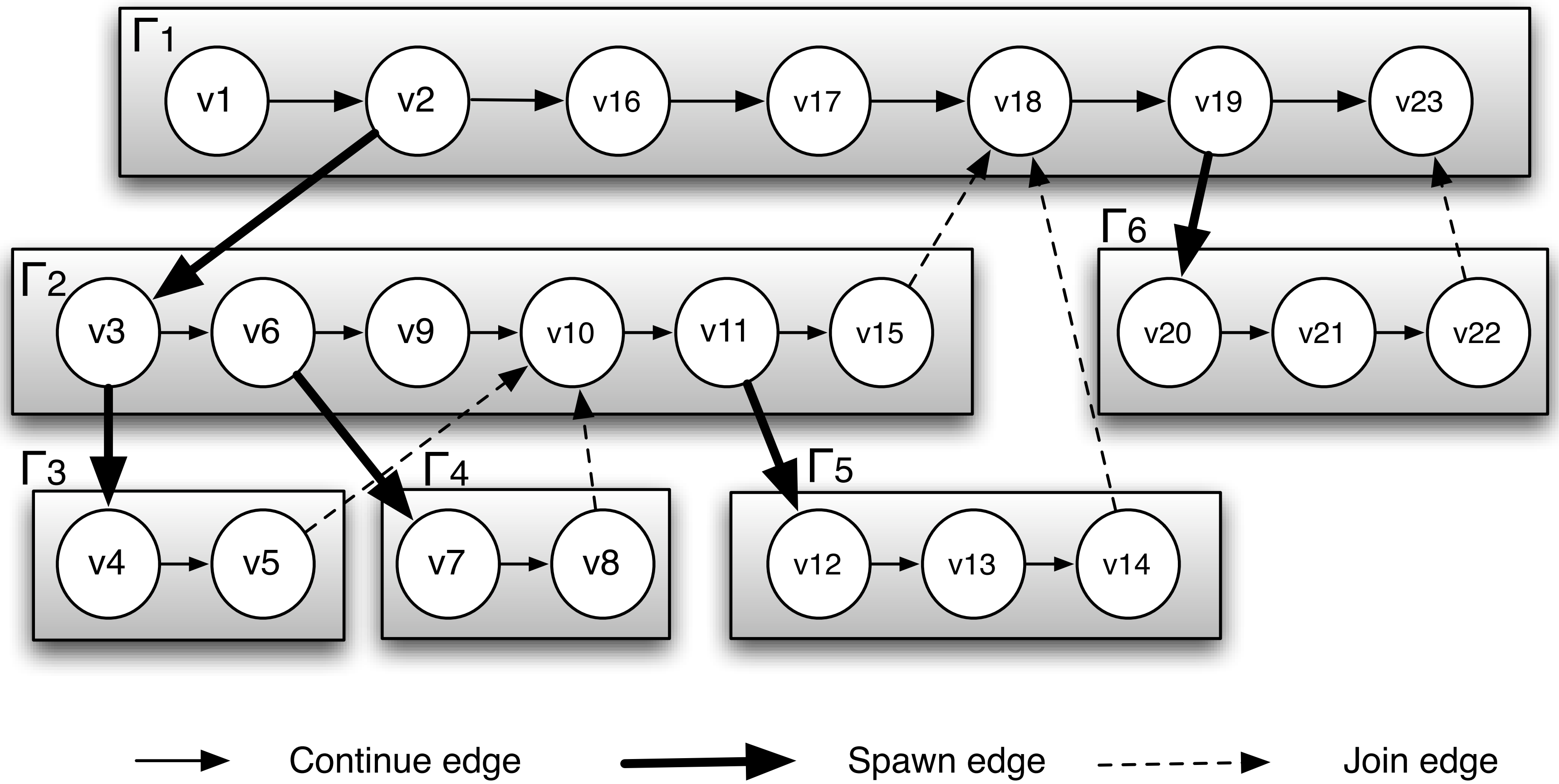
```
// Task T1
v1; v2;
finish {
  async {
    // Task T2
    v3;
    finish {
      async { v4; v5; } // Task T3
      v6;
      async { v7; v8; } // Task T4
      v9;
    } // finish
    v10; v11;
```

```
// 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 Edges

- CG edges represent ordering constraints
- There are three kinds of CG edges of interest in an HJ program with finish &async operations
  1. *Continue* edges define sequencing of steps within a task
  2. *Spawn* edges connect parent tasks to child async tasks
  3. *Join* edges connect async tasks to their Immediately Enclosing Finish (IEF) operations

Computation Graph for previous HJ Example



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

Dependences in a Computation Graph

- Given edge (A,B) in a *CG*, node B can only start execution after node A has completed
- We say that *node Y depends on node X* if there is a path of directed edges from X to Y in the *CG*
  - Also referred to as a "dependence from node X to node Y" or a "dependence from node Y on node X"
- Nodes X and Y can *potentially execute in parallel* if there is no dependence from X to Y or from Y to X
- Dependence is a *transitive* relation
  - if B depends on A and C depends on B, then C must depend on A
- 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)

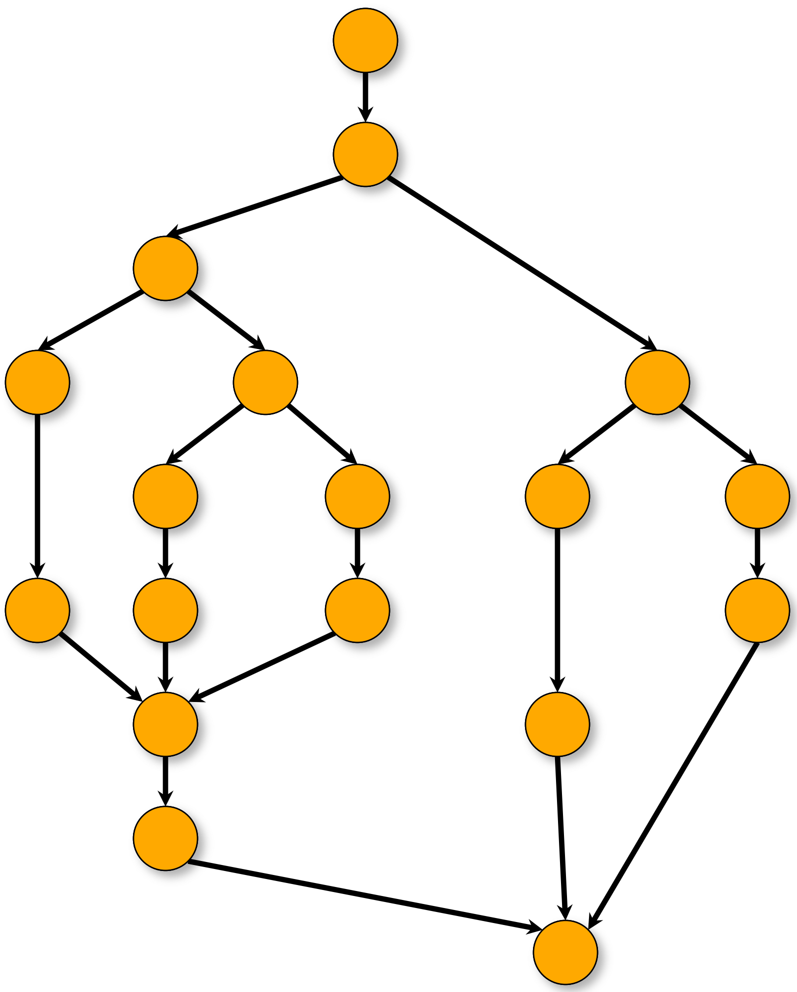
Complexity Measures for Computation Graphs

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 amount of work to be performed in G
- CPL(G) = length of a longest path in CG G, when adding up the 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*)

Example

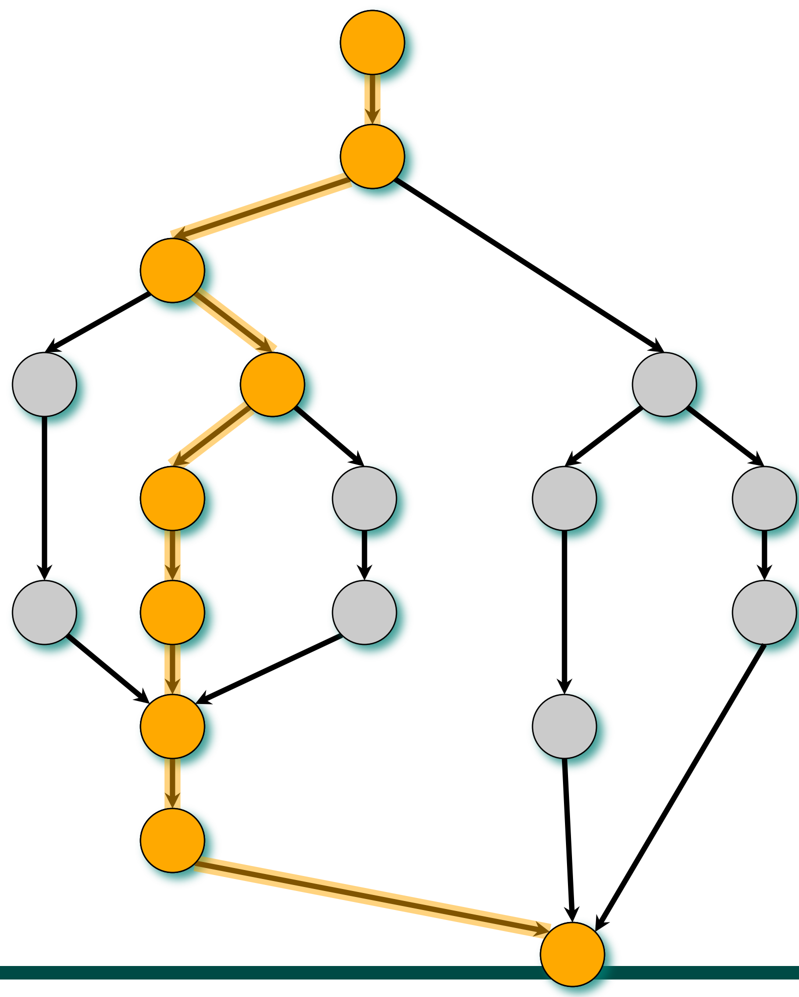
- Assume time(N) = 1 for all nodes in this graph



WORK(G) = 18

Example (contd)

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



CPL(G) = 9

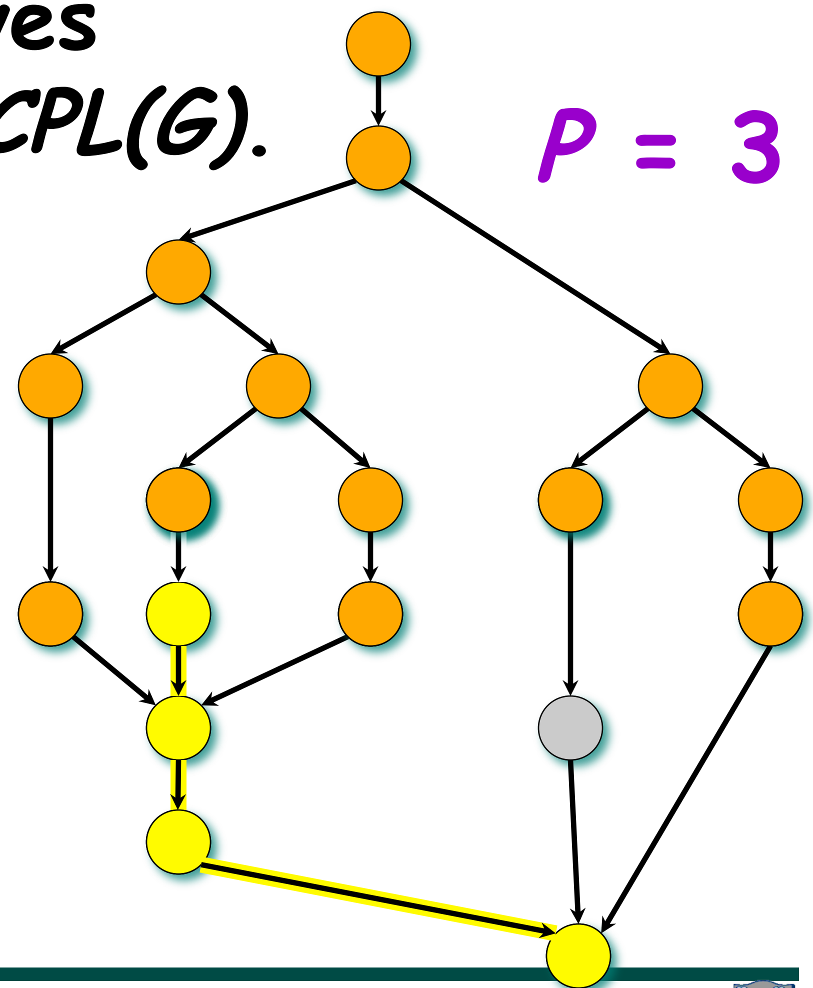
Lower Bounds on Execution Time

- t_P = execution time of computation graph on P processors
- Observations
  - t_1 = WORK(G)
  - t_∞ = CPL(G)
- Lower bounds
  - Capacity bound: $t_P \geq WORK(G)/P$
  - Critical path bound: $t_P \geq CPL(G)$
- Putting it together
  - $t_P \geq \max(WORK(G)/P, CPL(G))$

Greedy-Scheduling Theorem (Upper Bound)

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

$$t_P \leq WORK(G)/P + CPL(G).$$



Proof sketch.

 # complete steps ≤ *WORK(G)/P*, since each complete step performs *P* work.

 # incomplete steps ≤ *CPL(G)*, since each incomplete step reduces the span of the unexecuted dag by 1.

■

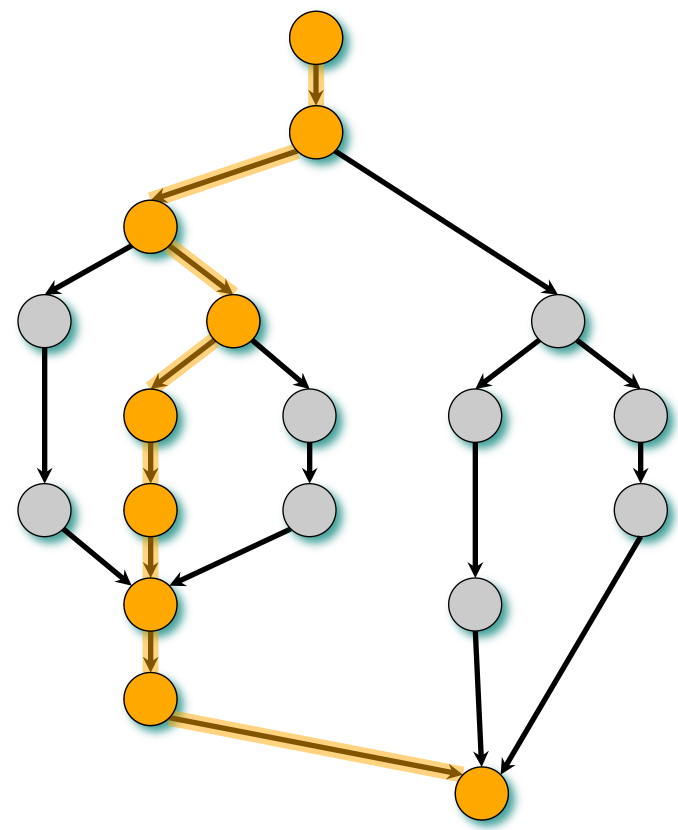
Parallelism ("Ideal Speedup")

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, WORK(G)/CPL(G) = T_1/T_∞

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



HJ Abstract Performance Metrics

- Serial code sequence
  - Dynamic sequence of instructions with no parallel operations
- Calls to perf.addLocalOps()
  - *Programmer* inserts calls of the form, perf.addLocalOps(N), inside a step to indicate execution of N application-specific abstract operations e.g., floating-point ops, stencil ops, data structure ops, etc.
  - Multiple calls add to the execution time of the step
- -perf=true runtime option
  - If an HJ program is executed with this option, abstract metrics are printed at end of program execution with WORK(G), CPL(G), *Ideal Speedup = WORK(G)/ CPL(G)*