EE/CSCI 451: Parallel and Distributed Computation

Lecture #16

10/13/2020

Viktor Prasanna

prasanna@usc.edu

ceng.usc.edu/~prasanna

University of Southern California

Course Project

- Project timeline
  - Week 7-8: Identify team members and project topic, discuss with instructor or TA
  - End of Week 9: Project proposal due (Oct. 16)
  - Week 12-13: Presentation
  - End of Week 13: Project report due (Nov. 13)

- Presentation order would be randomly assigned

Announcement

- PHW5 due on 10/22 (Thursday)
- HW7 due on 10/16 (Friday)
- HW8 will be out on 10/17 and due on 10/22 (1 day before midterm2)
- Final exam date: 2-4 PM Thursday, November 19

Announcement: Midterm 2

- Time: Oct. 23 (Friday) 3:30-5:30 PM
- A sample exam is posted on Piazza!
- Covers material from Sept. 22 to Oct. 16
  - Program mapping questions (covered in Week 6) will be on midterm 2!
- Logistics: same as Midterm 1

Outline

Last class

- GPU and Data Parallel (CUDA) programming model
  - SIMT
  - Streaming multiprocessor
  - Warp
  - CUDA Core

Today

- Parallel algorithm design (Chapter 3.1)
  - Task and dependency
  - Critical path
  - Task dependency graph
  - Mapping

Parallel Algorithm Design

Parallel Algorithm = Concurrency + Coordination

→ Communication

→ Synchronization

Design issues to be addressed:

- Synchronizing the various activities (processors)
- Identifying tasks that can be performed concurrently
- Mapping — Work → processors
- Scheduling — When to execute
- Data partitioning and distribution — Data → storage (layout)
- Data access management

Metrics and Constraints

- Memory footprint
- # of processors used
- Cost

- Latency, throughput
- Speed-up
- Scalability
- Efficiency
- Energy efficiency

Approaches

Problem eg. Mat. Mult.

Design parallel algorithm

Serial solution eg. 3 level loop

Computation model
Programming model

Parallel code

Compile

Run time

Manual parallelization using directives (ex. OpenMP)

Compile

Run time

Automatic Parallelization

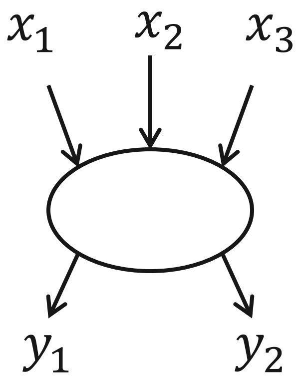
Parallel platform eg. Multi-core, cluster

Tasks and Dependencies

Computation = decompose into tasks

Task = Set of instructions (program segment)

Inputs & outputs



Begin execution once **all** inputs are available

Task size?

= weight of the node
(e.g., # of instructions executed)

Fine grain

Coarse grain

Note: tasks need not be of the same size

Task Dependency Graph

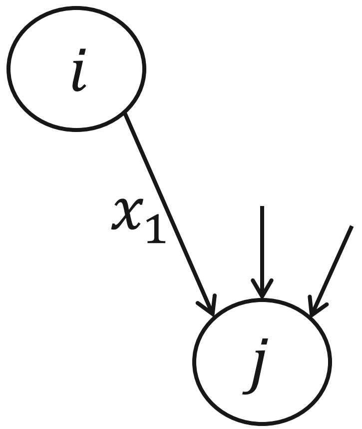
Directed Acyclic Graph

Task_j cannot start until Task_i completes

Data x_1: output of Task_i

Input to Task_j

Weight of a node: task size



Task dependency graph need not be connected

Graph is acyclic

Example (1)

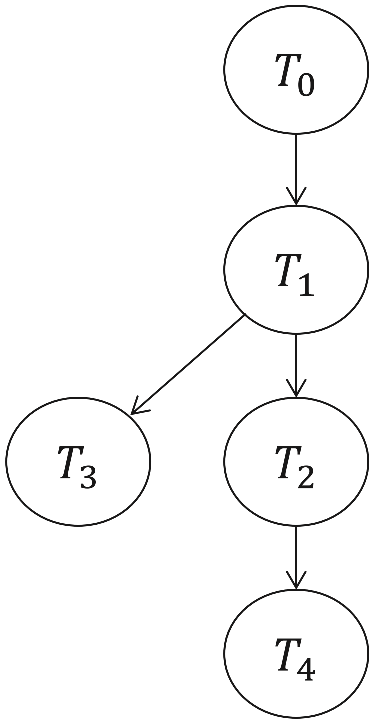
Code

A[2] = A[0] + 1

B[0] = A[2] + 1

Instructions		**Tasks**
Load	R0 ← A[0]	T_0
Add	R1 ← R0 + 1	T_1
Add	R2 ← R1 + 1	T_2
Store	A[2] ← R1	T_3
Store	B[0] ← R2	T_4

Task Dependency Graph



Maximum Degree of Concurrency (1)

Given a task dependency graph,
maximum number of tasks that can be executed concurrently

Note: maximum degree of concurrency depends on scheduling strategy

Maximum Degree of Concurrency (2)

Example: **level by level ordering (topological sort)**

Task dependency graph

Order tasks level by level

- Any level i task has dependency with some task in level $i-1$ (and possibly with other lower levels) but no dependency with level i

- All tasks in any level i are independent

Execute level i tasks (in parallel), and then level $i+1$ tasks

Maximum Degree of Concurrency (3)

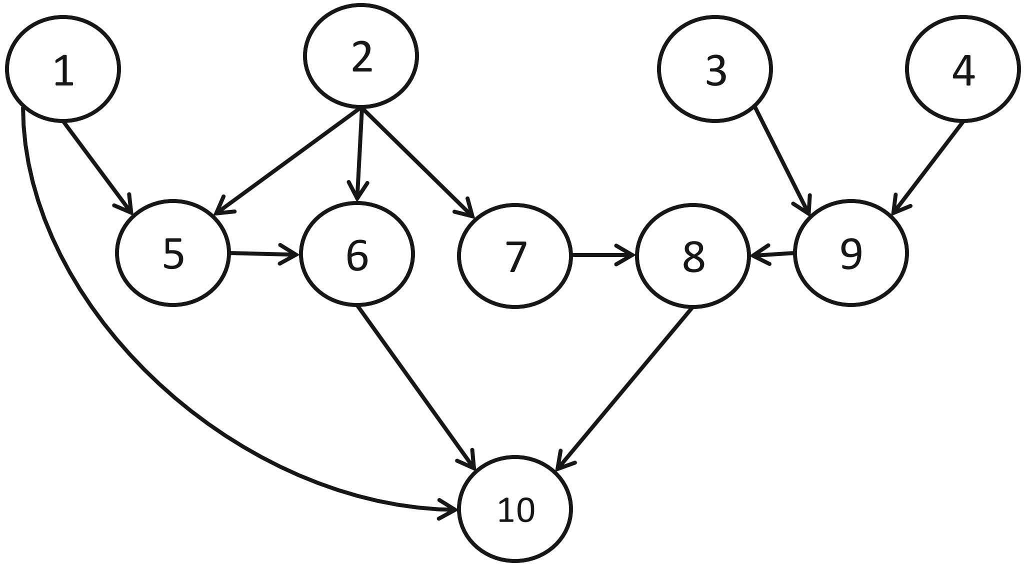
Example: **level by level ordering (cont.)**

Find the number of tasks in each level

Take maximum over all levels = maximum degree of concurrency
(if scheduled using level by level ordering)

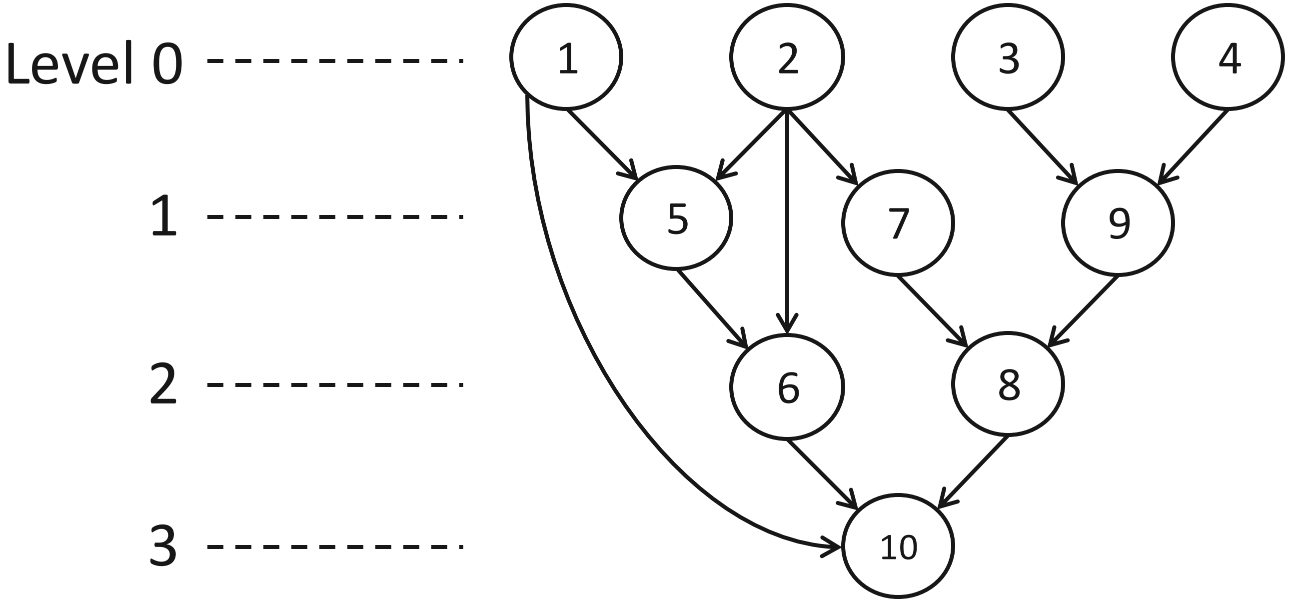
Maximum Degree of Concurrency (4)

Example



Maximum Degree of Concurrency (5)

Example (cont.)



Maximum degree of concurrency = 4

Critical Path (1)

Dependency graph

Start nodes (indeg = 0)

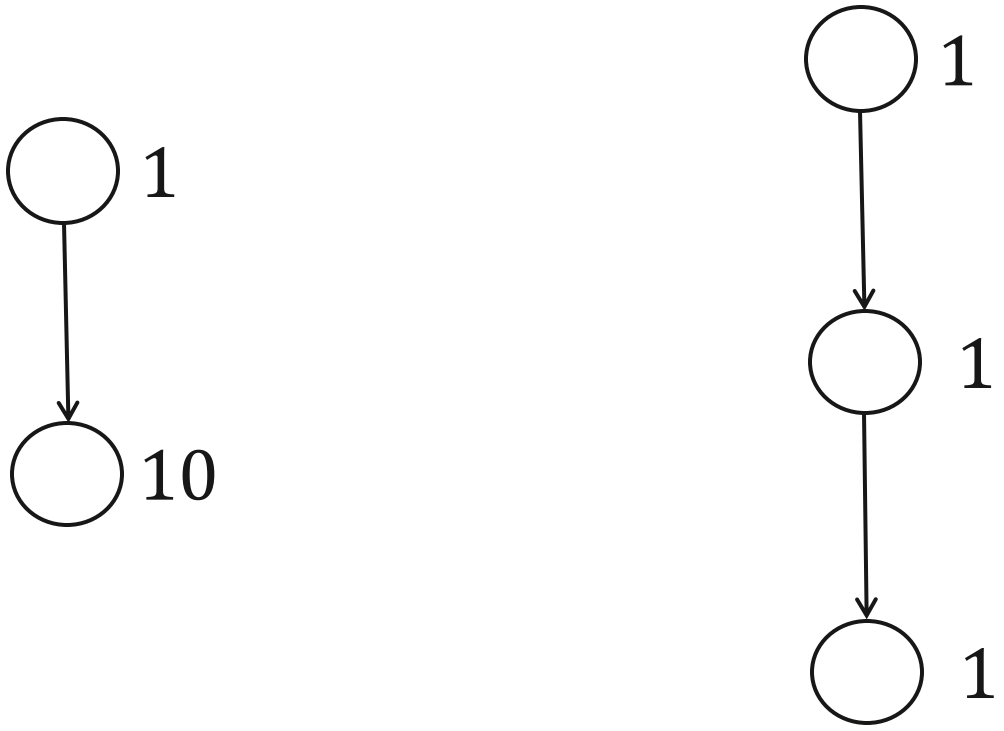
Finish nodes (outdeg = 0)

Critical path = A longest path from a start node to a finish node (# of edges)

Critical path length = Sum of the task weights of the nodes along the critical path

Critical Path (2)

Note: Critical path length considers critical path only (long paths)

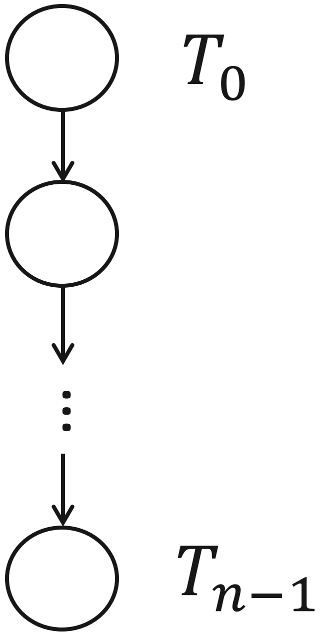


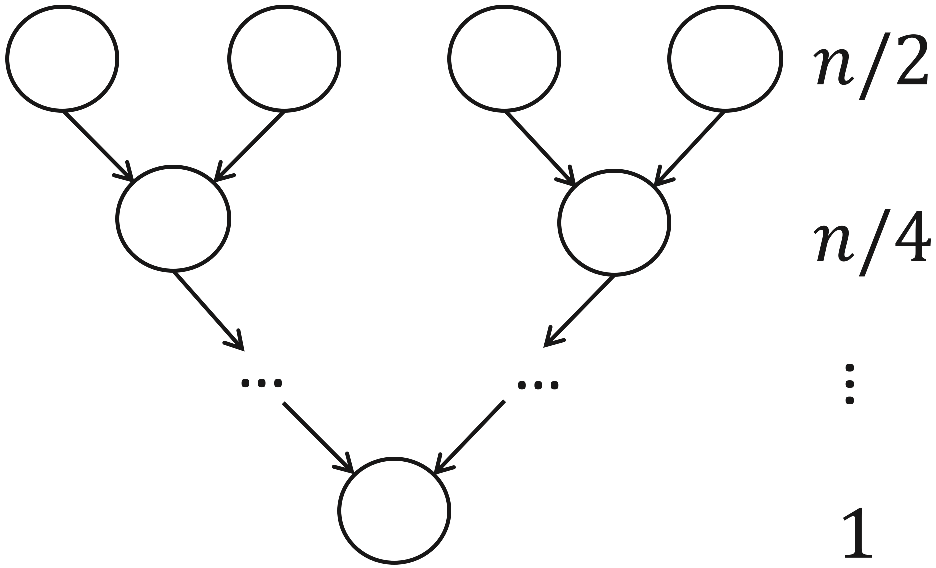
Critical Path (3)

For a given number of tasks,

Longer critical path $\Rightarrow$ Longer execution time
(may also mean less concurrency)

Critical Path (4)





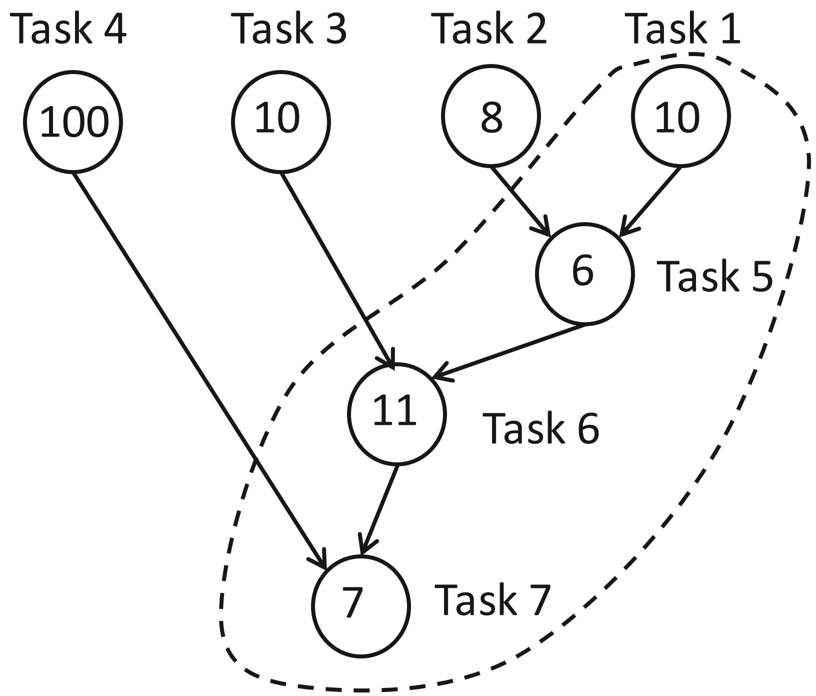
Total no. of tasks $= n$

Critical path length $= n$

Total no. of tasks $= n - 1$

Critical path length $= \log_2 n$

Critical Path (5)



Critical path length = 10+6+11+7 = 34

Task dependency graph to Parallel Program (1)

Given a task dependency graph

Assume weight of each node = 1

$c \implies$ in degree of each node = constant

Maximum degree of concurrency = c

Critical path length = l

Then, the task dependency graph can be executed on a PRAM ??

# of processors?

Time?

Task dependency graph to Parallel Program (2)

Idea

Note: TDG $\leftrightarrow$ DAG

DAG $\longrightarrow$ Organize into levels 0, 1, …, l (level by level ordering)

$(l + 1)$ levels

Execute level by level, 0 to l

Total number of processors needed $\leq c$

DAG To Parallel Program (3)

Correct parallel program $-$ all dependencies are satisfied

Parallel time $T_p = O(l)$

Using $p = c$ processors

Tasks, Processes, and Mapping (1)

Parallel algorithm → Tasks + interactions

Mapping problem ⟶ ↓ Map

Processes + interactions

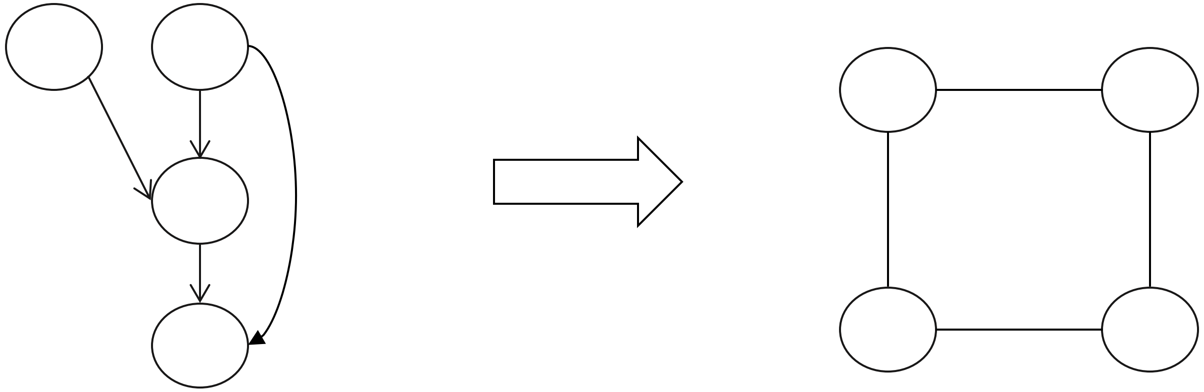
Further optimizations to explicit hardware features (eg. memory hierarchy, …) ⟶ ↓ ⇡ Abstract

Physical hardware (Processors + interconnection)

Tasks, Processes, and Mapping (2)

Mapping problem

- Assign each task to a process (processor)
  - Where to execute



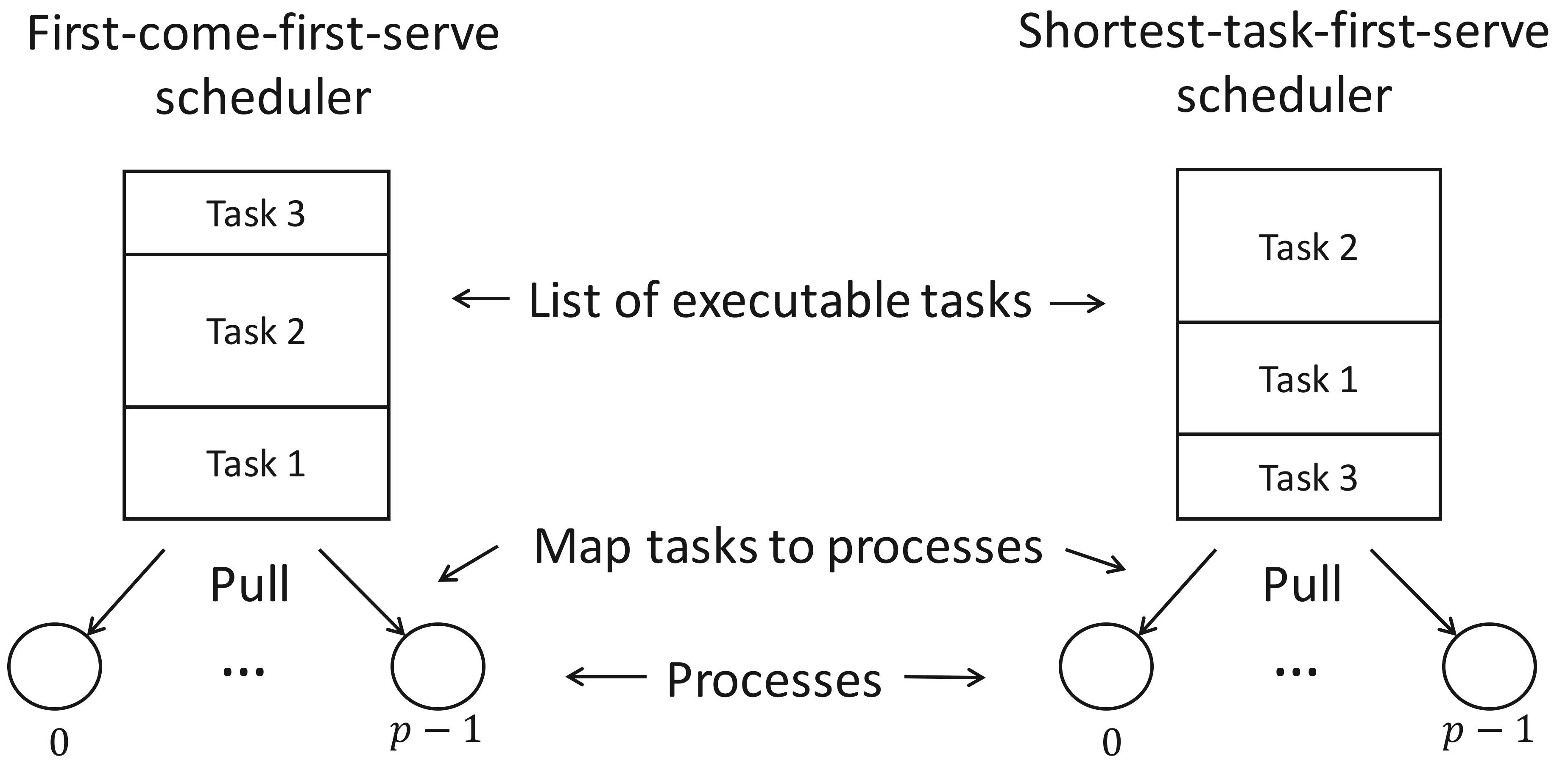
Task dependency graph

Tasks, Processes, and Mapping (3)

Scheduling problem

- Determine the execution order of each task
    - When to execute a task (after satisfying the dependencies)

Tasks, Processes, and Mapping (4)



Task, Processes, and Mapping (5)

Cluster tasks into processes

(minimize interaction among processes)

Map processes onto processers

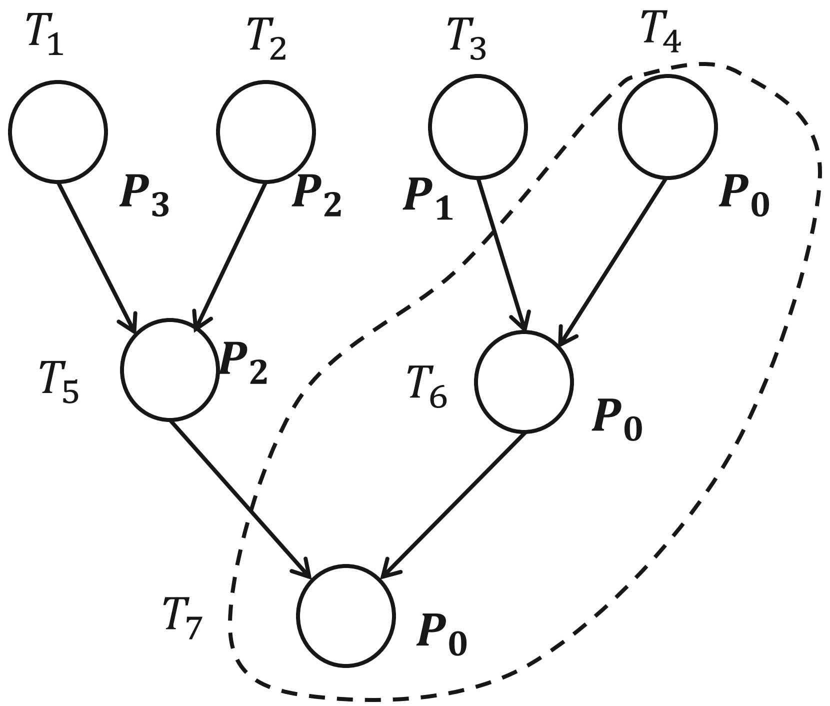
(physical resources)

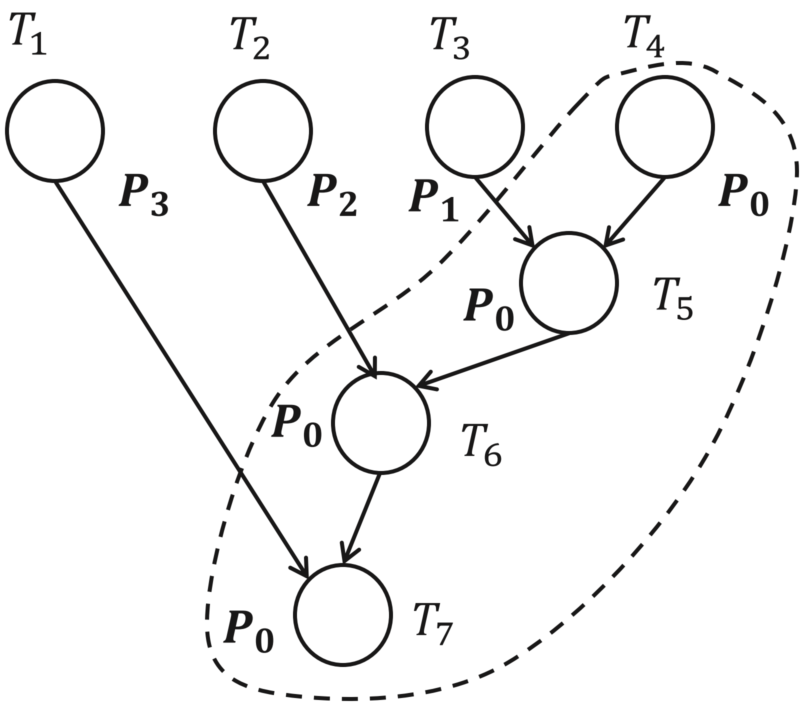
Process = Code + Data

Execute to produce output

Task, Processes, and Mapping (6)

Mapping tasks to processes

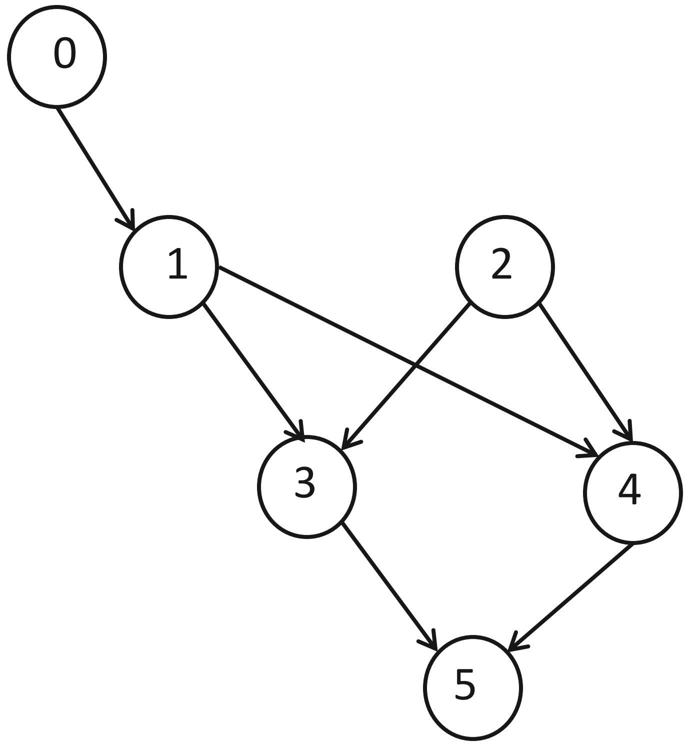




Use of Task Dependency Graph

- Program analysis
- Compiler optimizations (mapping, code generation)
- Automatic parallelization

Example



Pthreads program?

Level by Level Ordering (1)

All nodes with indeg $= 0$ $\longrightarrow$ Level 0

$i \leftarrow 1$

Repeat

Delete all nodes with indeg $= 0$ and all their outgoing edges

Place all nodes with indeg $= 0$ at level i

$i \leftarrow i + 1$

Until no nodes left

Can do in $O(n + e)$ time

$n =$ # of nodes

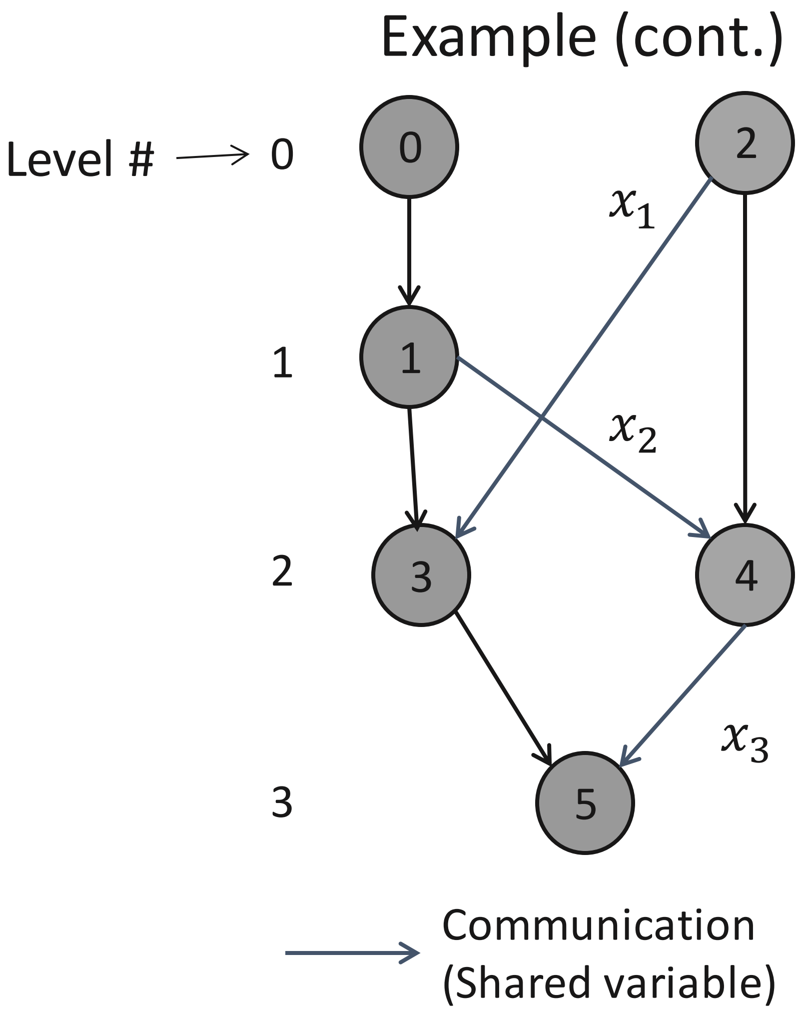
$e =$ # of edges

Level by Level Ordering (2)

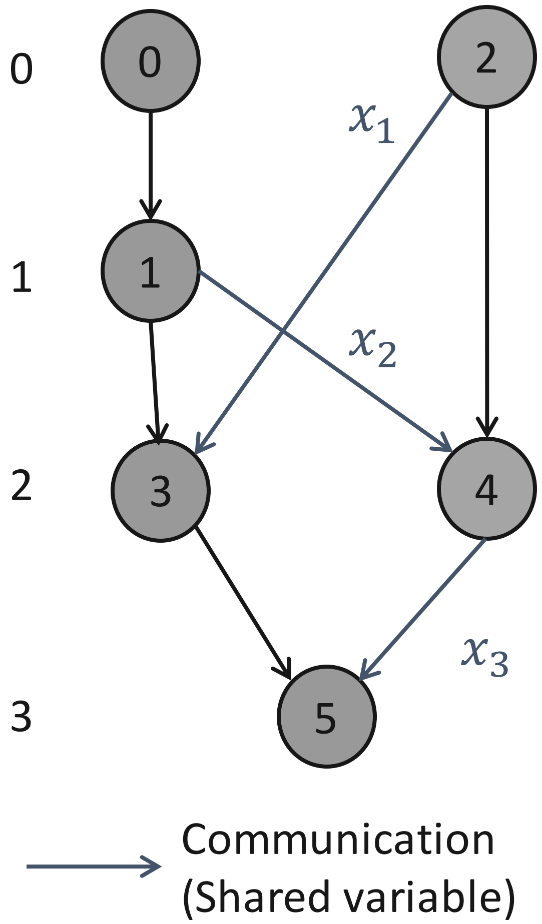
Thread = Path (chain) from an input node
to an output node

Partition TDG into collection of paths

Insert synchronization primitives
(disjoint)



Example (Parallel Program – Shared Memory)



Thread 0	Shared	Thread 1
T_0		T_2
T_1	x_1	
Wait for x_1	x_2	Wait for x_2
T_3	x_3	T_4
Wait for x_3		
T_5		

Producer Consumer Synchronization

Task dependency graph to parallel program

Level by level ordering

Parallel program

Execute level 0 tasks (in parallel)

Barrier

Communicate

Execute level 1 tasks

Barrier

Communicate

…

…

Map tasks in each level

Schedule tasks in each level

Example (Parallel Program – Message Passing)

P_0

T_0

Barrier(1)

T_1

Send x_2 to P_1

Barrier(2)

T_3

Barrier(1)

T_5

P_1

T_2

Barrier(1)

Send x_1 to P_0

Barrier(2)

T_4

Send x_3 to P_0

Barrier(3)

Summary

- Parallel algorithm design
  - Task and dependency
  - Critical path
  - DAG to parallel program