Parallel Task Graphs

Jinzhen Wang
jwang96@charlotte.edu

Department of Computer Science
UNC Charlotte

Learning Outcomes

Lecture

At the end of this session you will know how to:

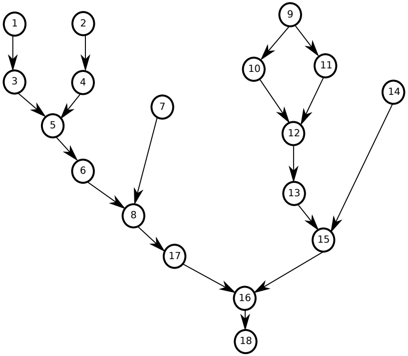
- Give two representations of parallel codes
- Compute metrics on parallel task graphs
- Interpret metrics of parallel task graphs in term of parallel execution

The Parallel Task Graph representation (PTG)

DAG representation

- Represents tasks as vertices.
- Represents x before y using a $x \to y$ directed edge.
- The graph is always without cycles.

Example

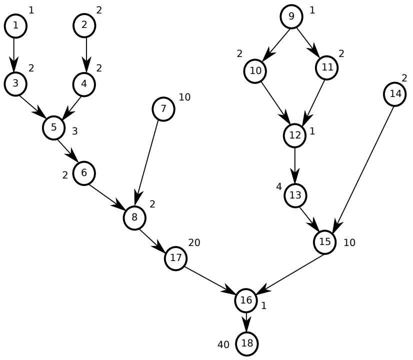


The Parallel Task Graph representation (PTG)

DAG representation

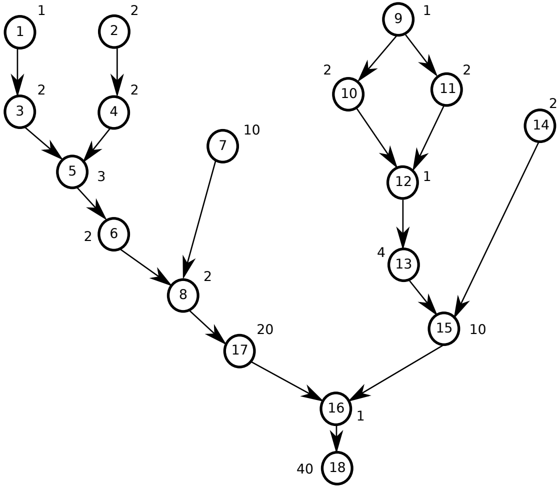
- Represents tasks as vertices.
- Represents x before y using a $x \rightarrow y$ directed edge.
- The graph is always without cycles.
- Processing time required associated with vertices, often denoted p_i

Example



A lemon pie recipe

1. break 2 eggs and split the white and yoke
2. cut 125g of butter in cubes
3. mix yoke and 70g of sugar+5cl of water
4. mix 250g of flour with butter
5. mix (3) and (4) and make a ball
6. spread (5)
7. heat oven to 180C
8. put crust (6) in pie pan
9. wash 4 lemons
10. peel two lemons from (9) and finely cut them
11. press two lemons from (9)
12. mix lemons(11), peel(10), 160g of sugar, 1 sp of flour
13. cook slowly (12)
14. whip 3 eggs
15. mix (14) and (13) and cook fast whipping
16. empty (15) in (17)
17. cook (8) for 20 minutes
18. wait until (16) cools



The conflict graph representation

Conflict graph

- Used to represent a set of tasks that can be executed in any order but that use a common resource.
- Undirected graph with edges that connect tasks with a conflict.

Example

Class	Instructor	Lab
I	A	1
II	B	1
III	A	2
IV	C	2
V	C	1

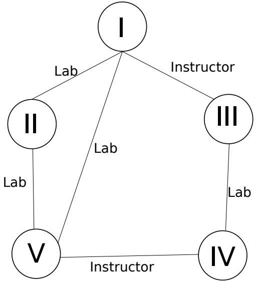
The conflict graph representation

Conflict graph

- Used to represent a set of tasks that can be executed in any order but that use a common resource.
- Undirected graph with edges that connect tasks with a conflict.

Example

Class	Instructor	Lab
I	A	1
II	B	1
III	A	2
IV	C	2
V	C	1



Conflict Graph vs. Dependency Graph

conflict graphs and **dependency graphs** are used in parallel computing to represent relationships between tasks or operations, but they serve different purposes and emphasize different aspects of these relationships.

Dependency Graph

- Purpose - shows the **order** where tasks or operations must be executed. It is used to visualize and analyze the dependencies between tasks, where one task must be completed before another can start.
- Nodes: Each node represents a task or operation.
- Edges: Directed edges ($A \rightarrow B$) represent dependencies, meaning task A must be completed before task B can start. The graph is usually DAG.
- Use Case: Crucial in **scheduling** tasks, optimizing execution order, and understanding parallelization limits. They help determine the **critical path**, which dictates the minimum time to complete all tasks.
- Example: In the robotcoin problem, a dependency graph would show how the computation of each cell in F depends on the cells to the left and above it.

Conflict Graph

- Purpose: represents **mutual exclusivity** in task execution, typically in the context of shared resources or parallel execution.
- Each node represents a task or operation.
- A (undirected) edge between two nodes (A - B) indicates that tasks A and B cannot be executed in parallel due to a conflict.
- Use case: Useful in scenarios like resource allocation, where tasks must be scheduled in such a way that conflicts are avoided. They are often used to optimize task assignments in parallel systems.
- Example: In the robotcoin problem, a conflict graph might represent scenarios where two operations cannot be performed at the same time because they access the same data in a conflicting manner.

Conflict Graph vs. Dependency Graph

Comparison in Context

- Focus:
  - The dependency graph focuses on the sequential dependencies required to compute each cell in the DP table.
  - The conflict graph focuses on which cells (tasks) cannot be updated simultaneously due to conflicts, such as resource contention.
- Types of Edges:
  - Dependency graphs have **directed edges** that show the direction of dependency.
  - Conflict graphs have **undirected edges** that indicate mutual exclusion.
- Execution Implications:
  - The dependency graph helps determine the overall **execution order** and critical path.
  - The conflict graph helps identify **parallelization constraints** by showing which operations cannot be run concurrently.

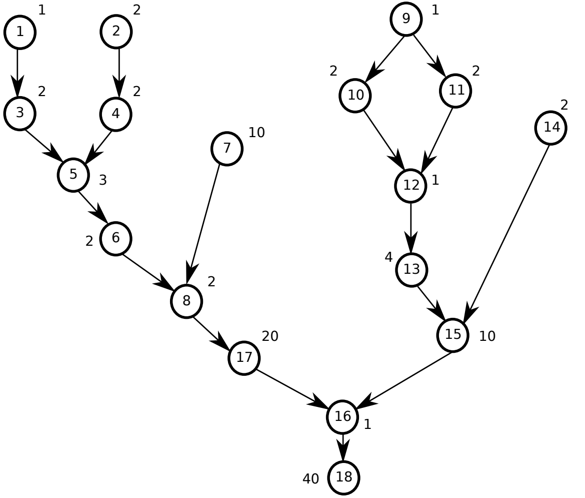
Practical application

Internal representation of compilers. (https://ars.els-cdn.com/content/image/3-s2.0-B9780120884780000128-f12-02-9780120884780.jpg?_ source: cooper torczon)
Direct representation of SQL queries (https://docs.oracle.com/cd/B10500_01/server.920/a96533/scratchpad1.gif Source: oracle)
Workflows of metagenomics analysis (https://www.researchgate.net/profile/Ulf-Leser/publication/257799855/figure/fig5/AS:297330902355989@1447900619823/A-generic-Galaxy-workflow-for-performing-a-metagenomic-analysis-on-NGS-data-A png source: wandelt et al. 2012) leveraging the Galaxy Framework
Project management https://pmatechnologies.com/wp-content/uploads/2019/09/Picture2.png Source: PMA Technologies

Metrics: Work

Work

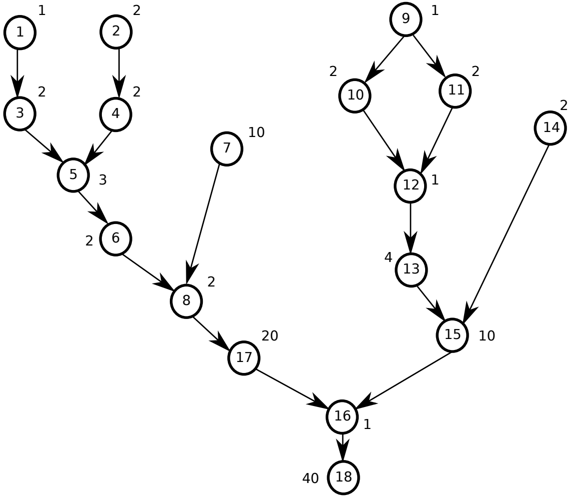
- Total amount of work that is to perform on the application.
- Simply the sum of all processing times.
- Often denoted $\sum p_i$



Metrics: Work

Work

- Total amount of work that is to perform on the application.
- Simply the sum of all processing times.
- Often denoted $\sum p_i$



Here $\sum p_i = 107$

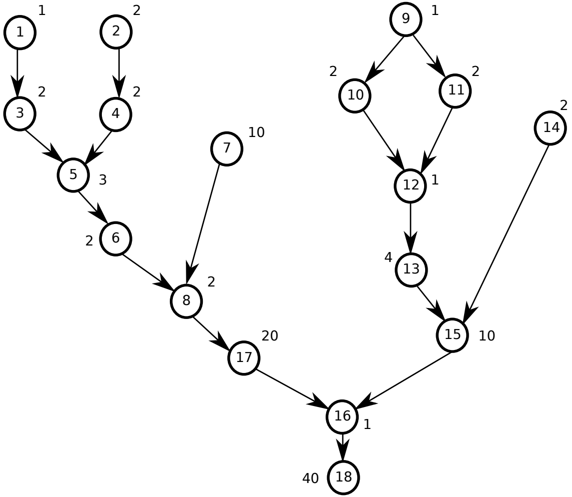
Metrics: Work

Work

- Total amount of work that is to perform on the application.
- Simply the sum of all processing times.
- Often denoted $\sum p_i$

Usage

- On m processors, the application can not be processed faster than $\frac{\sum p_i}{m}$.
- $\frac{\sum p_i}{m}$ is a **lower bound** of the **makespan**.
- $C_{max} \geq \frac{\sum p_i}{m}$

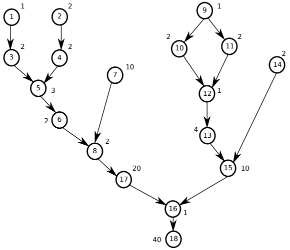


Here $\sum p_i = 107$

Metrics: Width

Width

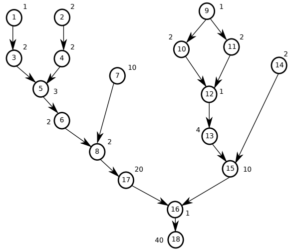
- Maximum number of tasks that do not have direct dependencies, or transitive dependencies.
- Maximum number of independent tasks.
- Sometimes called the longest antichain.



Metrics: Width

Width

- Maximum number of tasks that do not have direct dependencies, or transitive dependencies.
- Maximum number of independent tasks.
- Sometimes called the longest antichain.

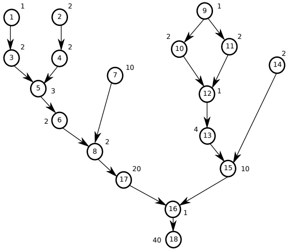


Here the width is 6.

Metrics: Width

Width

- Maximum number of tasks that do not have direct dependencies, or transitive dependencies.
- Maximum number of independent tasks.
- Sometimes called the longest antichain.



Here the width is 6.

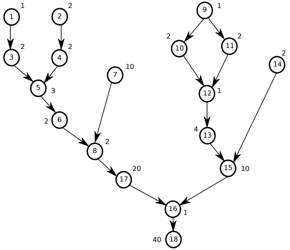
Usage

- Maximum number of useful processors.
- $\forall m > Width, S(m) = S(Width)$

Metrics: Width

Width

- Maximum number of tasks that do not have direct dependencies, or transitive dependencies.
- Maximum number of independent tasks.
- Sometimes called the longest antichain.



Here the width is 6.
How to find ?

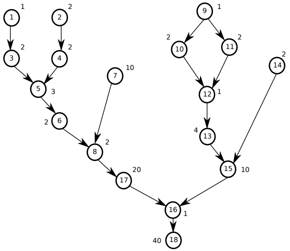
Usage

- Maximum number of useful processors.
- $\forall m > Width, S(m) = S(Width)$

Metrics: Width

Width

- Maximum number of tasks that do not have direct dependencies, or transitive dependencies.
- Maximum number of independent tasks.
- Sometimes called the longest antichain.



Usage

- Maximum number of useful processors.
- $\forall m > Width, S(m) = S(Width)$

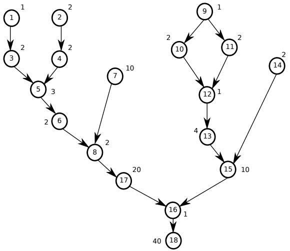
Dilworth's algorithm

- Build a bipartite graph from dependencies
- Compute a matching
- Extract longest antichain from matching

Metrics: Critical Path

Critical Path

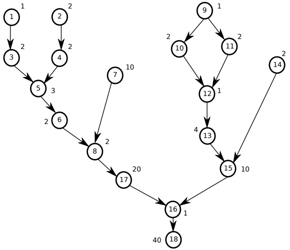
- Longest chain of dependency (in term of processing time)
- The length of the chain is often denoted CP, or T_∞.



Metrics: Critical Path

Critical Path

- Longest chain of dependency (in term of processing time)
- The length of the chain is often denoted CP, or T_∞.



Here $7 \rightarrow 8 \rightarrow 17 \rightarrow 16 \rightarrow 18$.

$CP = 73$

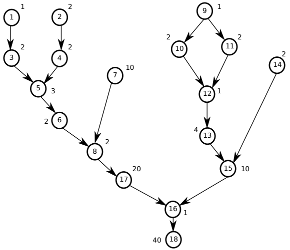
Metrics: Critical Path

Critical Path

- Longest chain of dependency (in term of processing time)
- The length of the chain is often denoted CP, or T_∞.

Usage

- Whichever way the algorithm unfolds, the critical path will have to be done.
- The length of the critical path is a lower bound to the makespan
- $C_{max} \geq CP$



Here $7 \to 8 \to 17 \to 16 \to 18$.
$CP = 73$

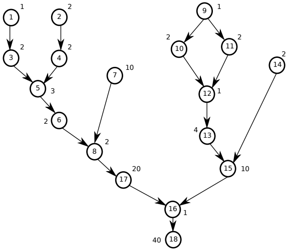
Metrics: Critical Path

Critical Path

- Longest chain of dependency (in term of processing time)
- The length of the chain is often denoted CP, or T_∞.

Usage

- Whichever way the algorithm unfolds, the critical path will have to be done.
- The length of the critical path is a lower bound to the makespan
- $C_{max} \geq CP$



Here $7 \to 8 \to 17 \to 16 \to 18$.
$CP = 73$

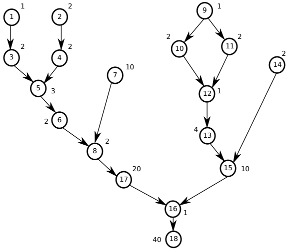
Metrics: Critical Path

Critical Path

- Longest chain of dependency (in term of processing time)
- The length of the chain is often denoted CP, or T_∞.

Usage

- Whichever way the algorithm unfolds, the critical path will have to be done.
- The length of the critical path is a lower bound to the makespan
- $C_{max} \geq CP$



Here $7 \rightarrow 8 \rightarrow 17 \rightarrow 16 \rightarrow 18$.
$CP = 73$

How to find it?

Recursively, from the roots down

Algorithm to compute the critical path

An algorithm

Let's assume we have a Directed Acyclic Graph (DAG) representing tasks where:

- Each node v represent a task.
- Each directed edge $u \to v$ indicates that task v depends on task u.
- Each node v has an associated duration (processing time).

1. Topological Sort (DFS/Kahn's algo, checkout here for more info):
   gives a linear ordering of the tasks where for every directed edge $u \to v$, task u comes before task v.
2. Initialize Distances:
   create a *distance* array, where $distance[v]$ stores the longest path from the start node to v.
3. Calculate the distance for each node, following the order given by topological sort.
   $distance[v] = max(distance[v], distance[u] + d(u))$;
4. Determine the Critical Path:
   maximum value in the *distance* array.

External

Textbook:

- Chapter 2 to 5.1 of Oliver Sinnen. Task Scheduling for Parallel Systems. John Wiley & Sons, Inc. 2007. Access it through the library: https://librarylink.uncc.edu/login?url=https://onlinelibrary.wiley.com/doi/book/10.1002/0470121173

Cilk on graphs metrics:

- The Cilkview Scalability Analyzer, SPAA 2010.http://web.mit.edu/willtor/www/res/cilkview-spaa-10.pdf. a paper describing parallel application as a DAG and metrics.

Width:

- Dilworth's algorithm https://en.wikipedia.org/wiki/Dilworth%27s_theorem

Conflict graph and coloring:

- Conflict graphs: http://math.cmu.edu/~bkell/21110-2010s/conflict-graphs.html
- A. H Gebremedhin, F. Manne, Alex Pothen. What Color Is Your Jacobian? Graph Coloring for Computing Derivatives. Siam Review 2005.
- M. Deveci, E. Boman, K. Devine, and S. Rajamanickam. Parallel Graph Coloring for Manycore Architectures. IPDPS 2016.

Scheduling:

- A taxonomy of scheduling problems: Srishti Srivastava and Ioana Banicescu. Scheduling in Parallel and Distributed Computing Systems. Chapter 11 of Prasad, Gupta, Rosenberg, Sussman, and Weems. Topics in Parallel and Distributed Computing: Enhancing the Undergraduate Curriculum: Performance, Concurrency, and Programming on Modern Platforms, Springer International Publishing, 2018. https://grid.cs.gsu.edu/~tcpp/curriculum/?q=system/files/Ch11_4.pdf
- Scheduling is NP-Hard: M. Garey and D. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman. 1979.
- LS for independent tasks: R. Graham. Bounds for certain multiprocessing anomalies. Bell System Technical Journal. 1966
- LPT and LS with precedence: R. Graham. Bounds on Multiprocessing Timing Anomalies. SIAM Journal on Applied Mathematics. 1969.
- Chapter 1 and 7 of. H. Casanova, A. Legrand, Y. Robert. Parallel Algorithms, CRC Press. 2008