Parallel Topological Sorting

Design of High Performance Computing, Fall 2015

Kevin Wallimann Johannes Baum Matthias Untergassmair

ETH Zürich

November 2, 2015

Problem Description

- Directed acyclic graph (DAG) describes a partial order
- A topological ordering is a total order on a DAG
- Any DAG has at least one topological ordering

"Efficient" Parallel and Distributed Topological Sort Algorithms

Ma et al. 1997

- Runtime: $\mathcal{O}(\log^2 N)$
- Reduces to matrix-matrix multiplication problem

Problem:

$\mathcal{O}(N^3)$ execution units required

"Efficient" Parallel and Distributed Topological Sort Algorithms

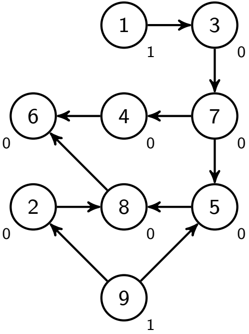
M. C. Er 1983

- Runtime: $\mathcal{O}(N)$
- More precise: longest distance between source node and sink
- Propagation of node values from all source nodes to all sink nodes

Topological ordering - algorithm

Input Directed acyclic graph (DAG) with N nodes

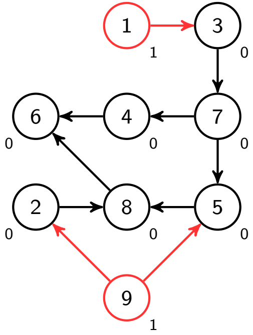
Output Topological Sortings of DAG



Topological ordering - algorithm

Input Directed acyclic graph (DAG) with N nodes

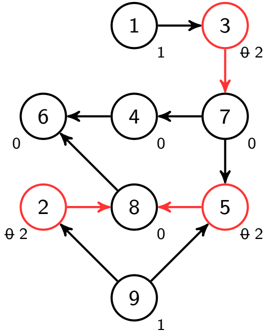
Output Topological Sortings of DAG



Topological ordering - algorithm

Input Directed acyclic graph (DAG) with N nodes

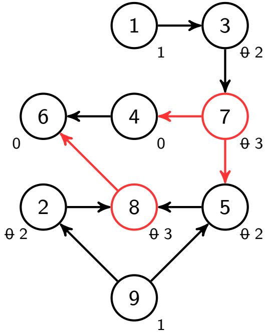
Output Topological Sortings of DAG



Topological ordering - algorithm

Input Directed acyclic graph (DAG) with N nodes

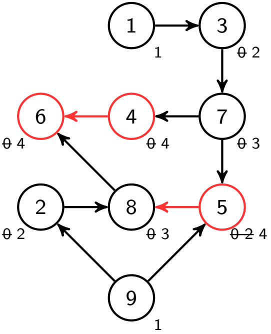
Output Topological Sortings of DAG



Topological ordering - algorithm

Input Directed acyclic graph (DAG) with N nodes

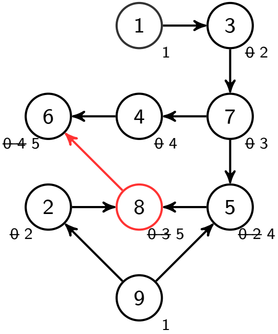
Output Topological Sortings of DAG



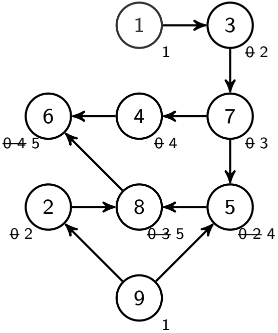
Topological ordering - algorithm

Input Directed acyclic graph (DAG) with N nodes

Output Topological Sortings of DAG

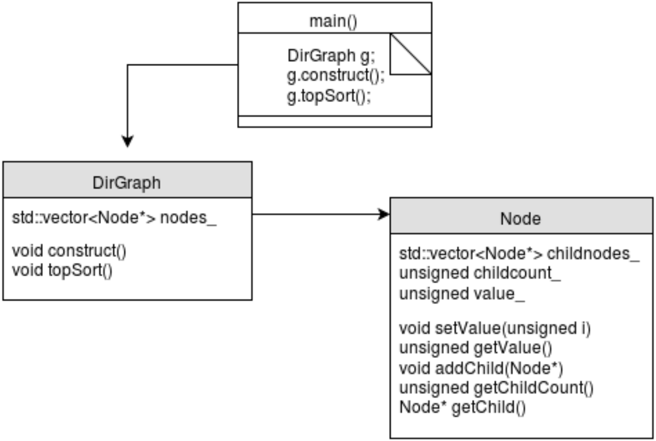


Topological ordering - algorithm



Name	1	9	2	3	7	4	5	8	6
Value	1	1	2	2	3	4	4	5	5

UML Diagram



main()

DirGraph g;
g.construct();
g.topSort();

DirGraph

std::vector<Node*> nodes_

void construct()
void topSort()

Node

std::vector<Node*> childnodes_
unsigned childcount_
unsigned value_

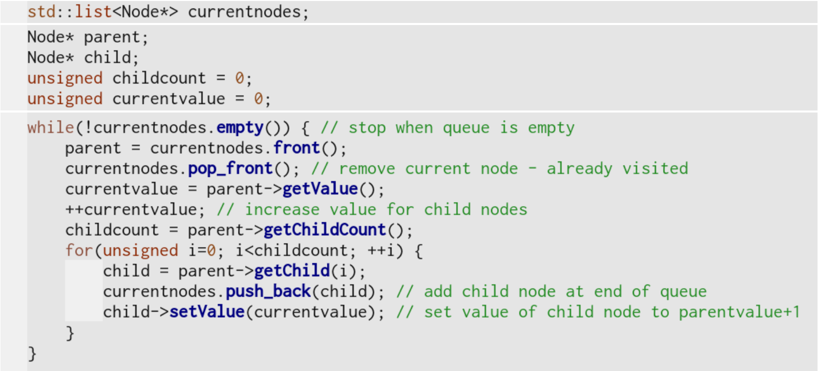
void setValue(unsigned i)
unsigned getValue()
void addChild(Node*)
unsigned getChildCount()
Node* getChild()

Serial Code

```
std::list<Node*> currentnodes;
```

```
Node* parent;
Node* child;
unsigned childcount = 0;
unsigned currentvalue = 0;
```

Serial Code



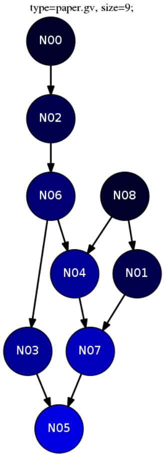
```
std::list<Node*> currentnodes;

Node* parent;
Node* child;
unsigned childcount = 0;
unsigned currentvalue = 0;

while(!currentnodes.empty()) { // stop when queue is empty
    parent = currentnodes.front();
    currentnodes.pop_front(); // remove current node - already visited
    currentvalue = parent->getValue();
    ++currentvalue; // increase value for child nodes
    childcount = parent->getChildCount();
    for(unsigned i=0; i<childcount; ++i) {
        child = parent->getChild(i);
        currentnodes.push_back(child); // add child node at end of queue
        child->setValue(currentvalue); // set value of child node to parentvalue+1
    }
}
```

Parallelization Ideas

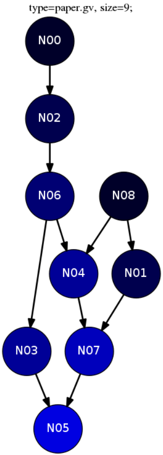
- Start in parallel at source nodes
- Spawn a new thread for every child node
- Synchronization needed when a node has multiple parents
- Performance dependent on topology of graph. Worst case $\mathcal{O}(n)$



Synchronization ideas

- Each thread enumerates the nodes it visits.
- If a node already has a number, take the higher number

- Problem: Multiple threads might "follow" each other.
- Idea: Only last thread arriving at a node may continue.



Hardware / Tools

- Shared memory parallelization
- OpenMP or C++11 threads.
- On Euler: 12-core Intel Xeon E5
- Intel Xeon Phi

Challenges

- Task queue
- Find a way to cope with chain-like graphs.
- Optional goal: Find all possible topological sortings.

Questions

- Up to how many execution units should we parallelize at least?
- From your experience, what are the main time-consuming, maybe unexpected obstacles when running on Intel Xeon Phi?