Parallel Topological Sorting

Design of High Performance Computing, Fall 2015

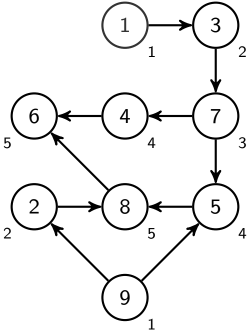
Kevin Wallimann Johannes Baum Matthias Untergassmair

ETH Zürich

December 14, 2015

Overview

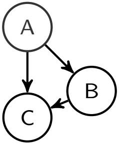
- DAG defines partial order
- Topological sorting defines one total order on a DAG
- Parallel algorithm: finds one topological sorting of a given DAG



Difference to BFS

- BFS visits every node
- Topological sorting algorithm needs to visit every edge

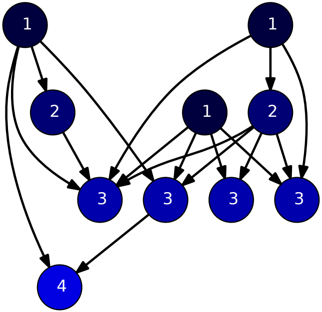
Example:



Consider order A,C,B $\rightarrow$ valid in BFS, invalid in topological sorting

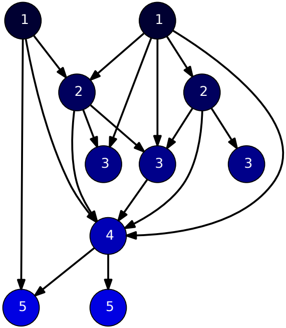
Random graph

- Parameter: Average node degree



Software dependency graph

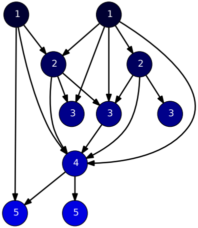
Nodes	Edges	Degree (Median)
100'000	266'680	2
10'000	27'416	2



Musco, V. et al. (2014) "A Generative Model of Software Dependency Graphs to Better Understand Software Evolution."

Parallel algorithm (shared memory)

1. As a preparing step, initialize a counter for every child with the number of its parents.
2. Distribute parent nodes over threads and process them in parallel.
3. For every parent node, get a list of all child nodes and append the parent itself to solution ($\rightarrow$ **lock**).
4. For every child of the parent, decrement its counter ($\rightarrow$ **lock**). Once the counter is zero, we can move on
5. When all parents are processed ($\rightarrow$ **barrier**), distribute new nodes and repeat



Local lists approach

Idea: perfect load balancing by redistributing nodes at every step

- Parent nodes stored in a global list.
- Distribution of parent nodes: scatter the list among the threads. Each thread has now its local list.
- Add new nodes to the end of local list.
- When all parents were processed, gather all local lists into the global list.
- Repeat until there are no parents left in the global list.

Boolean array approach

Idea: minimizing memory access by using lookup table

- Array of length N. 1 if node i is a parent node, 0 otherwise.
- Distribution of parent nodes: Parallel for-loop through the array.
- Mark new parents by setting a 1 in a second array.
- When all parents were processed, swap arrays.
- Loop through the array until there are no new parents.

Optimistic counter check

- Decrement shared counter
- Return true if counter is zero

```
inline bool counterCheck() {
        bool lastone;
        #pragma omp critical
        {
          --parcount_;
          lastone = (parcount_ == 0);
        }
        return lastone;
}
```

Optimistic counter check

- Decrement shared counter
- Return true if counter is zero

```
inline bool counterCheck() {
        #pragma omp atomic
        --parcount_;
        return (parcount_ == 0);
}
```

- Multiple threads could return true, although only one thread should do so.

Optimistic counter check

```
inline bool counterCheck() {
        #pragma omp atomic
        --parcount_;
        return (parcount_ == 0);
}
```

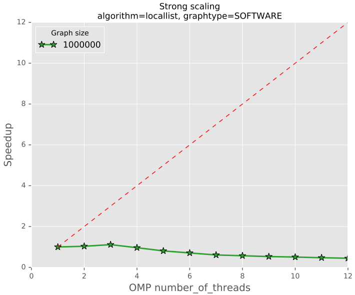
- List based approach: Child is inserted multiple times to solution $\Rightarrow$ Wrong.
- Array approach: Multiple threads write 1 to the array $\Rightarrow$ Ok, doesn't matter.

Euler

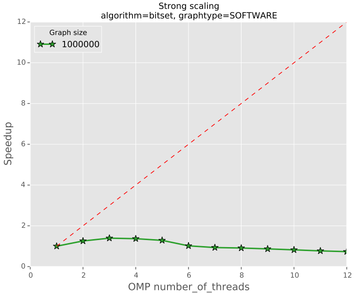
- Intel Xeon E5 on Euler cluster
- 2 processors per node
- 12 cores
- 30 MB shared last-level cache
- Software graph with around 1M nodes should fit into cache

Strong scaling software graph

Local lists



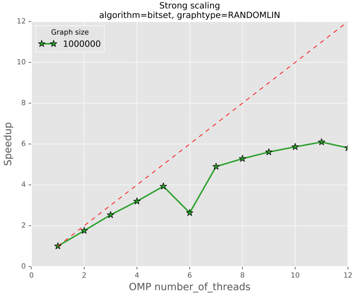
Bool array



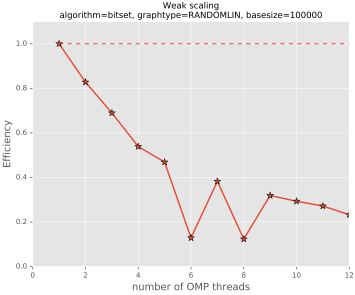
Absolute runtimes on 1 core	serial	bool array	local list
	0.45 s	0.58 s	0.48 s

Bool array & random graph (node degree 30)

Strong scaling



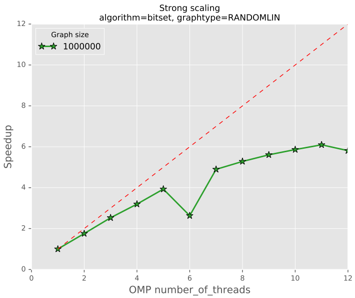
Weak scaling



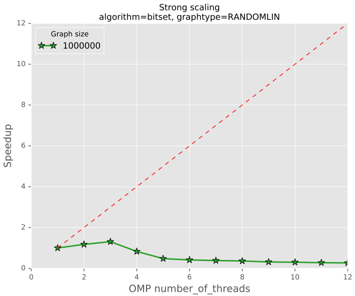
Absolute runtimes on 1 core	serial	bool array
	2.76 s	3.51 s

Optimistic counter check

Optimistic



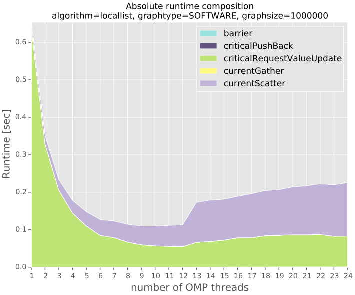
Conventional



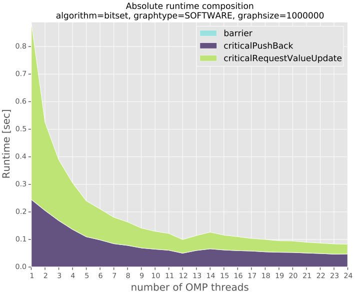
Absolute runtimes on 1 core	optimistic	conventional
	3.51 s	3.52 s

Absolute runtime composition

Local lists



Bool array



Remaining issues

- Pinpoint the reasons for bad scaling
- Work stealing could help to improve local list approach