

# PARALLEL TOPOLOGICAL SORTING

DESIGN OF HIGH PERFORMANCE COMPUTING, FALL 2015

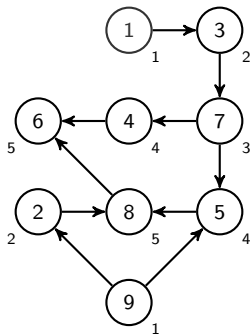
Kevin Wallimann    Johannes Baum    Matthias Untergassmair

ETH Zürich

December 14, 2015

# OVERVIEW

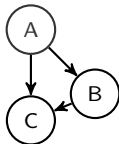
- DAG defines partial order
- Topological sorting defines one total order on a DAG
- Parallel algorithm: finds one topological sorting of a given DAG



## DIFFERENCE TO BFS

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

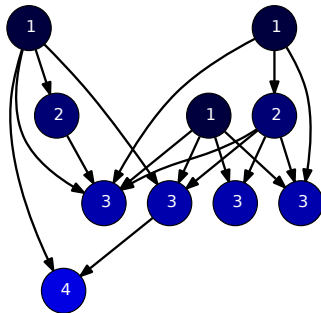
Example:



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

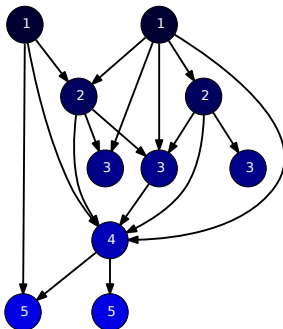
# RANDOM GRAPH

- Parameter:  
Average node  
degree



# SOFTWARE DEPENDENCY GRAPH

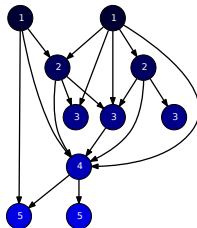
Nodes	Edges	Degree (Median)
100'000	266'680	2
10'000	27'416	2



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);  
}
```

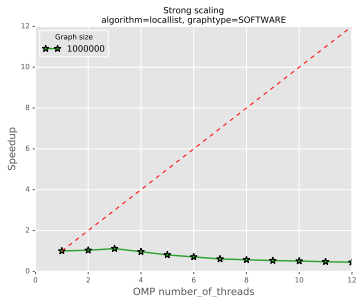
- List based approach: Child is inserted multiple times to solution  $\Rightarrow$  Wrong.
- Array approach: Multiple threads write 1 to the array  $\Rightarrow$  Ok, doesn't matter.

# 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



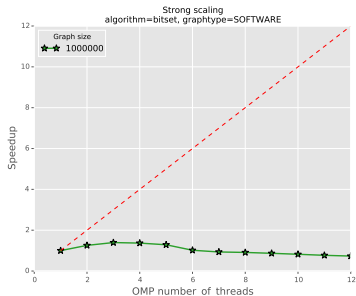
**Absolute runtimes on 1 core**

serial  
 0.45 s

bool array  
 0.58 s

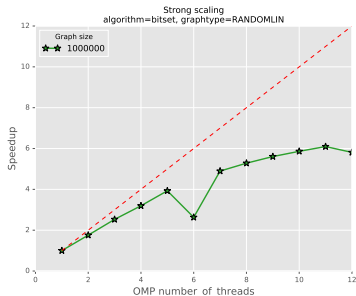
local list  
 0.48 s

## Bool array



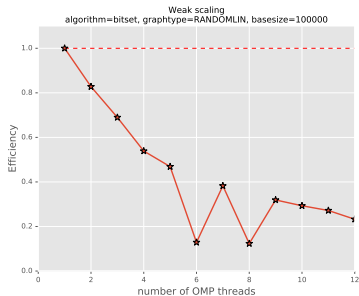
# BOOL ARRAY & RANDOM GRAPH (NODE DEGREE 30)

## Strong scaling



**Absolute runtimes on 1 core**

## Weak scaling

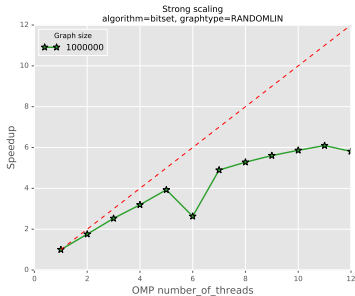


serial  
2.76 s

bool array  
3.51 s

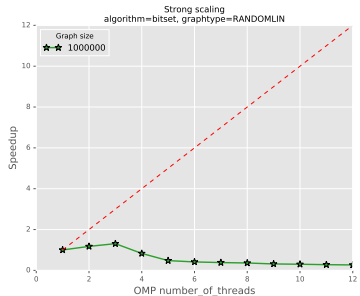
# OPTIMISTIC COUNTER CHECK

## Optimistic



Absolute runtimes on 1 core

## Conventional

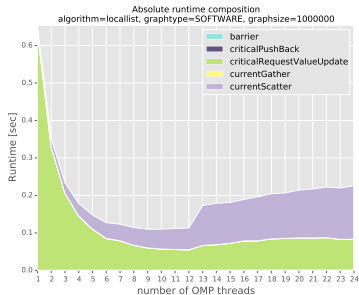


optimistic  
3.51 s

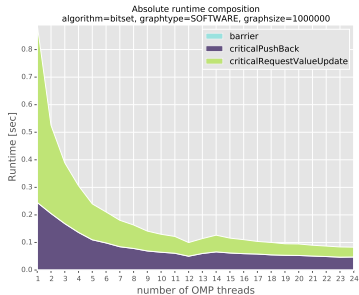
conventional  
3.52 s

# ABSOLUTE RUNTIME COMPOSITION

## Local lists



## Bool array





## REMAINING ISSUES

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

## REFERENCES



M. C. Er.

A Parallel Computation Approach to Topological Sorting  
The Computer Journal, Vol 28, 1983, Wiley Heyden Ltd



V. Musco, M. Monperrus, P. Preux

A Generative Model of Software Dependency Graphs to Better Understand  
Software Evolution  
arXiv 1410.7921