Memory-Aware DAG Scheduling

Grégoire Pichon, Bora Uçar & Frédéric Vivien

(Original slides by Loris Marchal)

CNRS, INRIA, Université Lyon 1 & ENS Lyon

CR15: January 2023

https://gpichon.gitlabpages.inria.fr/m2if-numerical_algorithms/

Outline

Introduction

- Directed Acyclic Graphs: express task dependences
  - nodes: computational tasks
  - edges: dependences
    (data = output of a task = input of another task)
- Formalism proposed long ago in scheduling
- Back into fashion thanks to task based runtimes

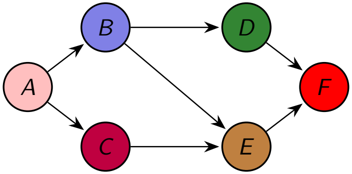
Introduction

- Directed Acyclic Graphs: express task dependences
    - nodes: computational tasks
    - edges: dependences
      (data = output of a task = input of another task)
- Formalism proposed long ago in scheduling
- Back into fashion thanks to task based runtimes

- Decompose an application (scientific computations) into tasks
- Data produced/used by tasks create dependences
- Task mapping and scheduling done at runtime
- Numerous projects:
    - StarPU (Inria Bordeaux) – several codes for each task to execute on any computing resource (CPU, GPU, *PU)
    - DAGUE, ParSEC (ICL, Tennessee) – task graph expressed in symbolic compact form, dedicated to linear algebra
    - StartSs (Barcelona), Xkaapi (Grenoble), and others...
    - Now included in OpenMP API

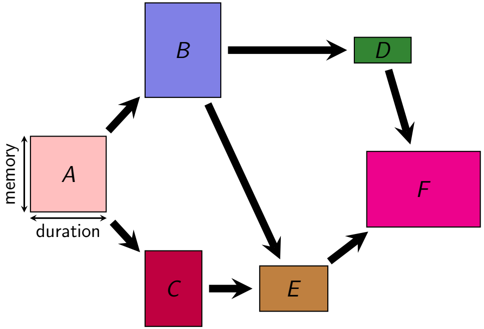
Task graph scheduling and memory

- Consider a simple task graph



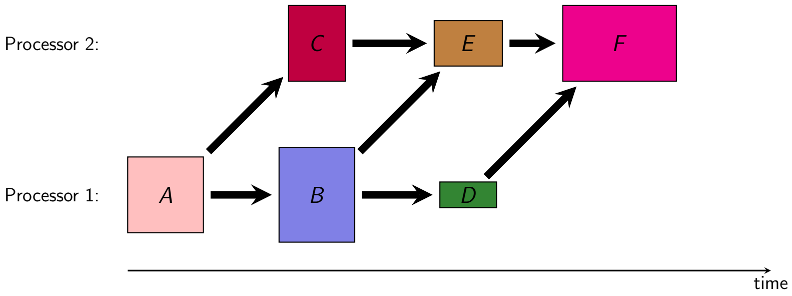
Task graph scheduling and memory

- Consider a simple task graph
- Tasks have durations and memory demands



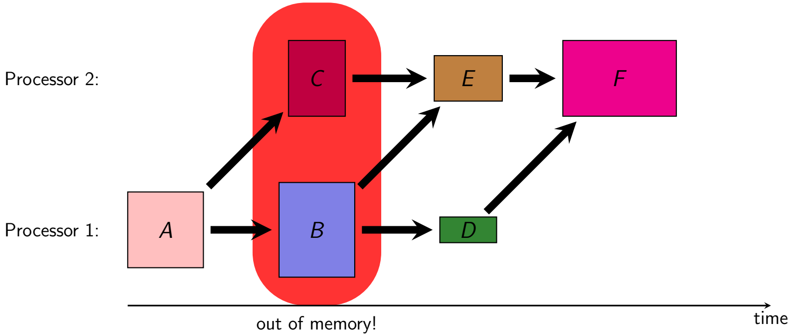
Task graph scheduling and memory

- Consider a simple task graph
- Tasks have durations and memory demands



Task graph scheduling and memory

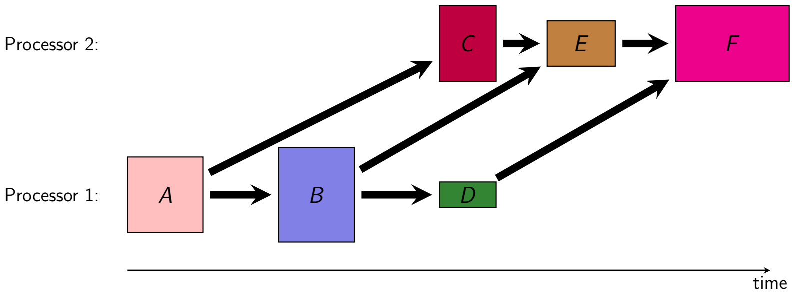
- Consider a simple task graph
- Tasks have durations and memory demands



- Peak memory: maximum memory usage

Task graph scheduling and memory

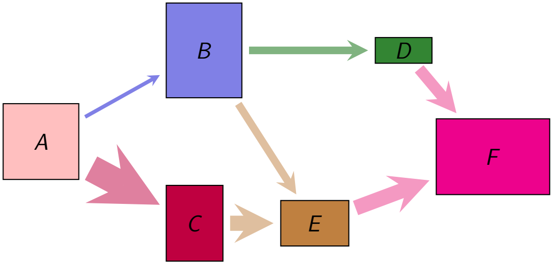
- Consider a simple task graph
- Tasks have durations and memory demands



- Peak memory: maximum memory usage
- Trade-off between peak memory and performance (time to solution)

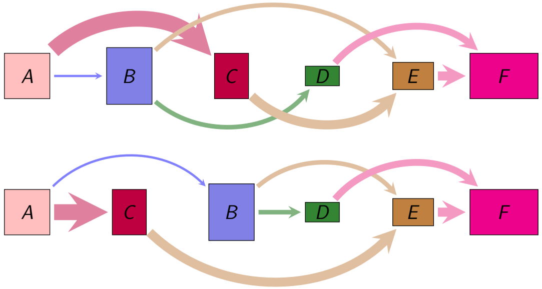
Going back to sequential processing

- Temporary data require memory
- Scheduling influences the peak memory



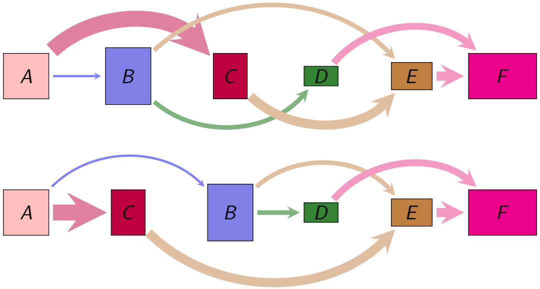
Going back to sequential processing

- Temporary data require memory
- Scheduling influences the peak memory



Going back to sequential processing

- Temporary data require memory
- Scheduling influences the peak memory



When minimum memory demand > available memory:

- Store some temporary data on a larger, slower storage (disk)
- Out-of-core computing, with Input/Output operations (I/O)
- Decide both scheduling and eviction scheme

Research problems

Several interesting questions:

- For sequential processing:
    - Minimum memory needed to process a graph
    - In case of memory shortage, minimum I/Os required

Research problems

Several interesting questions:

- For sequential processing:
  - Minimum memory needed to process a graph
  - In case of memory shortage, minimum I/Os required
- In case of parallel processing:
  - Tradeoffs between memory and time (makespan)
  - Makespan minimization under bounded memory

Research problems

Several interesting questions:

- For sequential processing:
  - Minimum memory needed to process a graph
  - In case of memory shortage, minimum I/Os required
- In case of parallel processing:
  - Tradeoffs between memory and time (makespan)
  - Makespan minimization under bounded memory

Most (all?) of these problems: NP-hard on general graphs ☹

Research problems

Several interesting questions:

- For sequential processing:
  - Minimum memory needed to process a graph
  - In case of memory shortage, minimum I/Os required
- In case of parallel processing:
  - Tradeoffs between memory and time (makespan)
  - Makespan minimization under bounded memory

Most (all?) of these problems: NP-hard on general graphs ☹

Sometimes restrict to simpler graphs:

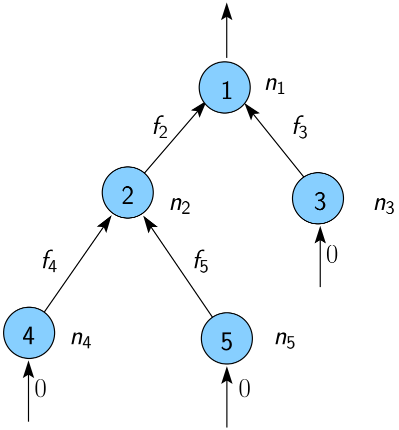
1. Trees (single output, multiple inputs for each task)
   Arise in sparse linear algebra (sparse direct solvers), with large data to handle: memory is a problem
2. Series-Parallel graphs
   Natural generalization of trees, close to actual structure of regular codes

Outline

1. Minimize Memory for Trees
2. Minimize Memory for Series-Parallel Graphs
3. Minimize I/Os for Trees under Bounded Memory
4. Complexity and Space-Time Tradeoffs for Parallel Tree Processing
5. Parallel Processing of DAGs with Limited Memory
    - Model and maximum parallel memory
    - Maximum parallel memory/maximal topological cut
    - Efficient scheduling with bounded memory
    - Heuristics and simulations

Outline

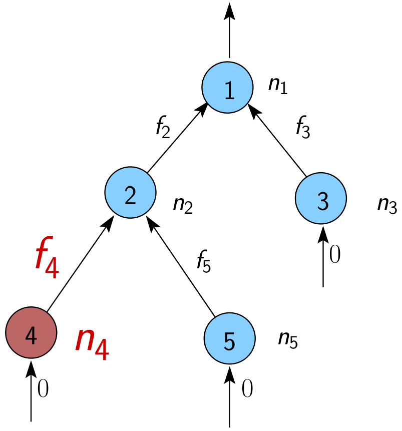
Notations: Tree-Shaped Task Graphs



- In-tree of n nodes
- Output data of size f_i
- Execution data of size n_i
- Input data of leaf nodes have null size

- Memory for node i: $MemReq(i) = \left(\sum_{j \in Children(i)} f_j\right) + n_i + f_i$

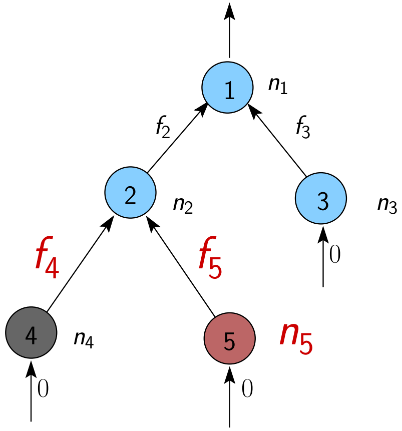
Notations: Tree-Shaped Task Graphs



- In-tree of n nodes
- Output data of size f_i
- Execution data of size n_i
- Input data of leaf nodes have null size

- Memory for node i: $MemReq(i) = \left(\sum_{j \in Children(i)} f_j\right) + n_i + f_i$

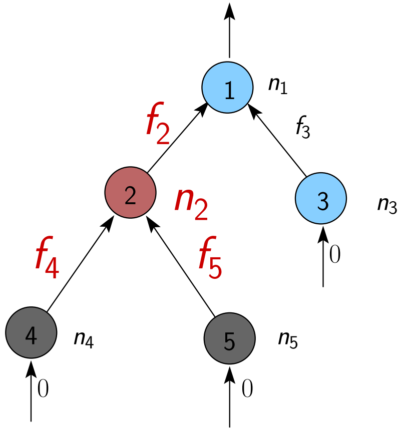
Notations: Tree-Shaped Task Graphs



- In-tree of n nodes
- Output data of size f_i
- Execution data of size n_i
- Input data of leaf nodes have null size

- Memory for node i: $MemReq(i) = \left(\sum_{j \in Children(i)} f_j \right) + n_i + f_i$

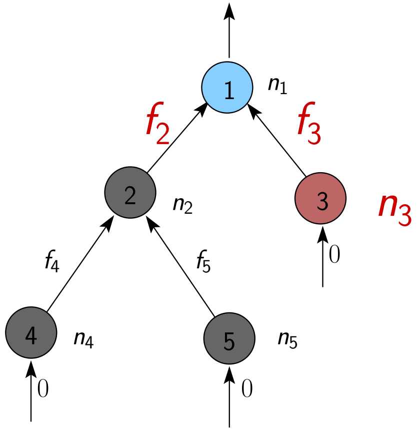
Notations: Tree-Shaped Task Graphs



- In-tree of n nodes
- Output data of size f_i
- Execution data of size n_i
- Input data of leaf nodes have null size

- Memory for node i: $MemReq(i) = \left(\sum_{j \in Children(i)} f_j \right) + n_i + f_i$

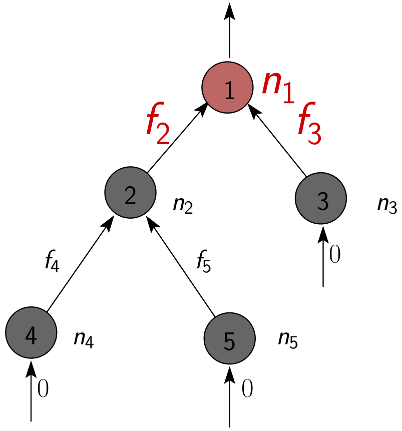
Notations: Tree-Shaped Task Graphs



- In-tree of n nodes
- Output data of size f_i
- Execution data of size n_i
- Input data of leaf nodes have null size

- Memory for node i: $MemReq(i) = \left(\sum_{j \in Children(i)} f_j\right) + n_i + f_i$

Notations: Tree-Shaped Task Graphs

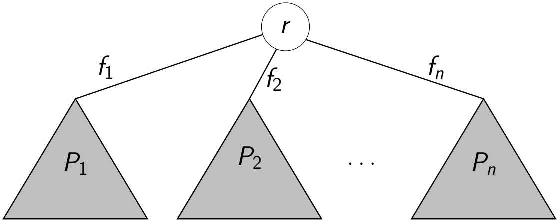


- In-tree of n nodes
- Output data of size f_i
- Execution data of size n_i
- Input data of leaf nodes have null size

- Memory for node i: $MemReq(i) = \left(\sum_{j \in Children(i)} f_j\right) + n_i + f_i$

Liu's Best Post-Order Traversal for Trees

Post-Order: entirely process one subtree after the other (DFS)

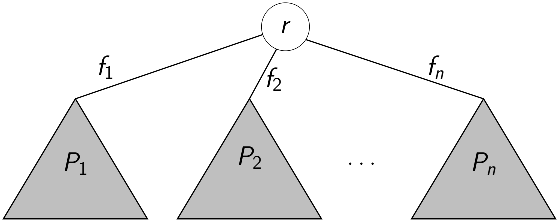


- For each subtree T_i: peak memory P_i, residual memory f_i
- For a given processing order $1, \dots, n$, the peak memory is:

$$\max\{P_1, \qquad\qquad\qquad\qquad\qquad\qquad \}$$

Liu's Best Post-Order Traversal for Trees

Post-Order: entirely process one subtree after the other (DFS)

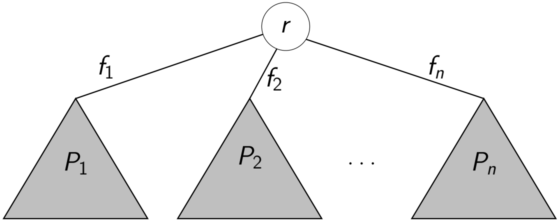


- For each subtree T_i: peak memory P_i, residual memory f_i
- For a given processing order $1, \dots, n$, the peak memory is:

$$\max\{P_1,\ f_1 + P_2, \qquad \}$$

Liu's Best Post-Order Traversal for Trees

Post-Order: entirely process one subtree after the other (DFS)

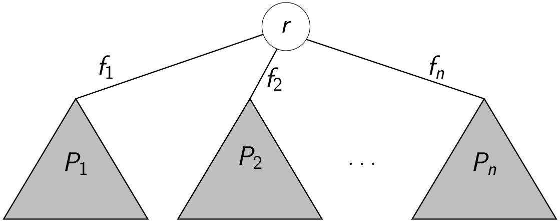


- For each subtree T_i: peak memory P_i, residual memory f_i
- For a given processing order $1, \dots, n$, the peak memory is:

$$\max\{P_1,\ f_1 + P_2,\ f_1 + f_2 + P_3, \qquad \}$$

Liu's Best Post-Order Traversal for Trees

Post-Order: entirely process one subtree after the other (DFS)

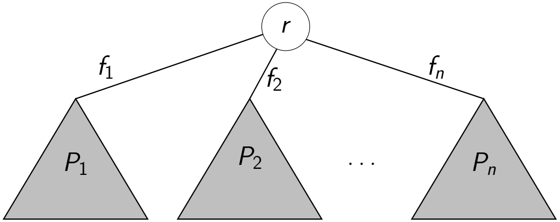


- For each subtree T_i: peak memory P_i, residual memory f_i
- For a given processing order $1, \dots, n$, the peak memory is:

$$\max\{P_1,\ f_1 + P_2,\ f_1 + f_2 + P_3,\ \dots, \sum_{i<n} f_i + P_n, \qquad\qquad\qquad \}$$

Liu's Best Post-Order Traversal for Trees

Post-Order: entirely process one subtree after the other (DFS)

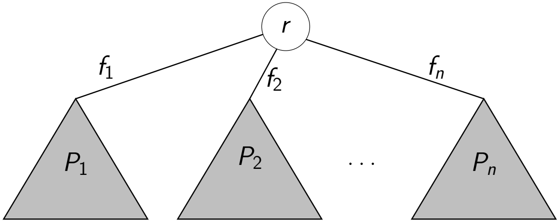


- For each subtree T_i: peak memory P_i, residual memory f_i
- For a given processing order $1, \dots, n$, the peak memory is:

$$\max\{P_1,\ f_1 + P_2,\ f_1 + f_2 + P_3,\ \dots, \sum_{i<n} f_i + P_n,\ \sum f_i + n_r + f_r\}$$

Liu's Best Post-Order Traversal for Trees

Post-Order: entirely process one subtree after the other (DFS)



- For each subtree T_i: peak memory P_i, residual memory f_i
- For a given processing order $1, \dots, n$, the peak memory is:

$$\max\{P_1,\ f_1 + P_2,\ f_1 + f_2 + P_3,\ \dots, \sum_{i<n} f_i + P_n,\ \sum f_i + n_r + f_r\}$$

- Optimal order: non-increasing $P_i - f_i$

Proof for best Post-Order

Theorem (Best Post-Order)

The best Post-Order traversal is obtained by processing subtrees in non-increasing order $P_i - f_i$.

Proof for best Post-Order

Theorem (Best Post-Order)

The best Post-Order traversal is obtained by processing subtrees in non-increasing order $P_i - f_i$.

Proof:

- Consider an optimal traversal which does not respect the order:
  - subtree j is processed right before subtree k
  - $P_k - f_k \geq P_j - f_j$

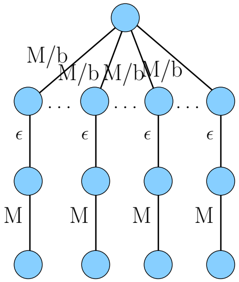
	peak when j, then k	peak when k, then j
during first subtree	$mem\_before + P_j$	$mem\_before + P_k$
during second subtree	$mem\_before + f_j + P_k$	$mem\_before + f_k + P_j$

- $f_k + P_j \leq f_j + P_k$
- Transform the schedule step by step without increasing the memory.

Post-Order is not optimal

Post-Order traversals are arbitrarily bad in the general case

There is no constant *k* such that the best post-order traversal is a *k*-approximation.

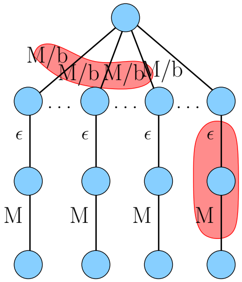


- Minimum post-order peak memory:

Post-Order is not optimal

Post-Order traversals are arbitrarily bad in the general case

There is no constant k such that the best post-order traversal is a k-approximation.

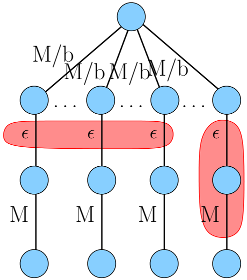


- Minimum post-order peak memory: $M_{\text{min}} = M + \epsilon + \ (b-1)M/b$

Post-Order is not optimal

Post-Order traversals are arbitrarily bad in the general case

There is no constant *k* such that the best post-order traversal is a *k*-approximation.

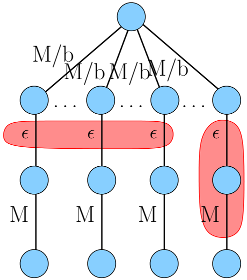


- Minimum post-order peak memory:
  $M_{\text{min}} = M + \epsilon + \ (b-1)M/b$
- Minimum peak memory:

Post-Order is not optimal

Post-Order traversals are arbitrarily bad in the general case

There is no constant k such that the best post-order traversal is a k-approximation.

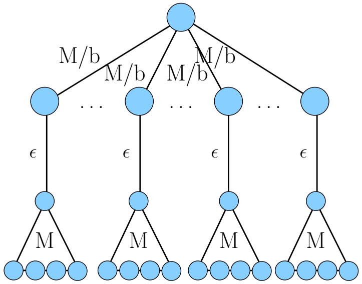


- Minimum post-order peak memory: $M_{\text{min}} = M + \epsilon + \ (b-1)M/b$
- Minimum peak memory: $M_{\text{min}} = M + \epsilon + \ (b-1)\epsilon$

Post-Order is not optimal

Post-Order traversals are arbitrarily bad in the general case

There is no constant k such that the best post-order traversal is a k-approximation.

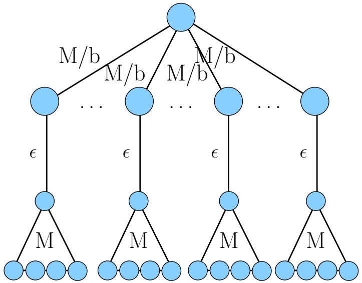


- Minimum post-order peak memory: $M_{\text{min}} = M + \epsilon + \ (b-1)M/b + ?$
- Minimum peak memory: $M_{\text{min}} = M + \epsilon + \ (b-1)\epsilon + ?$

Post-Order is not optimal

Post-Order traversals are arbitrarily bad in the general case

There is no constant k such that the best post-order traversal is a k-approximation.



- Minimum post-order peak memory: $M_{\min} = M + \epsilon + 2(b-1)M/b$
- Minimum peak memory: $M_{\min} = M + \epsilon + 2(b-1)\epsilon$

	actual assembly trees	random trees
Non optimal traversals	4.2%	61%
Maximum increase compared to optimal	18%	22%
Average increased compared to optimal	**1%**	12%

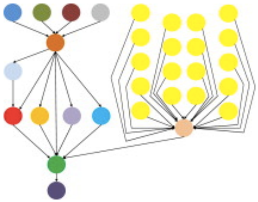
Liu's optimal traversal – sketch

- Recursive algorithm: at each step, merge the optimal ordering of each subtree (sequence)
- Sequence: divided into segments:
  - H_1: maximum over the whole sequence (hill)
  - V_1: minimum after H_1 (valley)
  - H_2: maximum after H_1
  - V_2: minimum after H_2
  - ...
  - The valleys V_i's are the boundaries of the segments
- Combine the sequences by non-increasing $H - V$
- Complex proof based on a partial order on the cost-sequences: $(H_1, V_1, H_2, V_2, \ldots, H_r, V_r) \prec (H'_1, V'_1, H'_2, V'_2, \ldots, H'_{r'}, V'_{r'})$ if for each $1 \leq i \leq r$, there exists $1 \leq j \leq r'$ with $H_i \leq H'_j$ and $V_i \leq V'_j$.

Outline

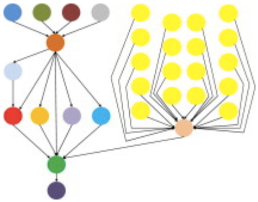
Series-Parallel Graphs: Motivation

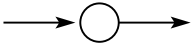
- Not all scientific workflows are trees
- But most workflows exhibit some regularity
- Large class of workflows: Series-Parallel graphs



Series-Parallel Graphs: Motivation

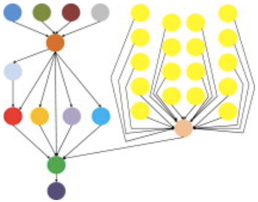
- Not all scientific workflows are trees
- But most workflows exhibit some regularity
- Large class of workflows: Series-Parallel graphs

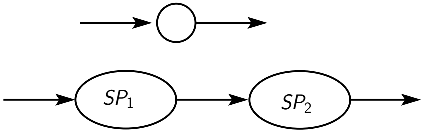




Series-Parallel Graphs: Motivation

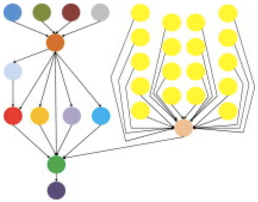
- Not all scientific workflows are trees
- But most workflows exhibit some regularity
- Large class of workflows: Series-Parallel graphs

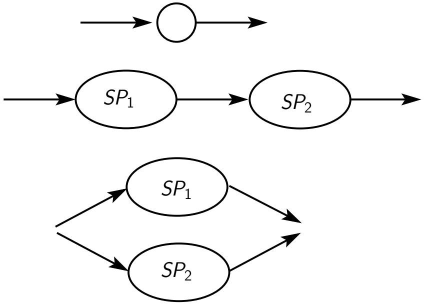




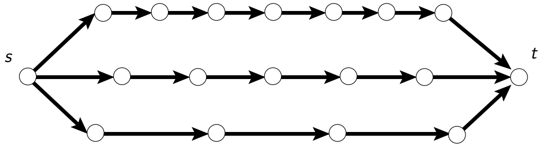
Series-Parallel Graphs: Motivation

- Not all scientific workflows are trees
- But most workflows exhibit some regularity
- Large class of workflows: Series-Parallel graphs

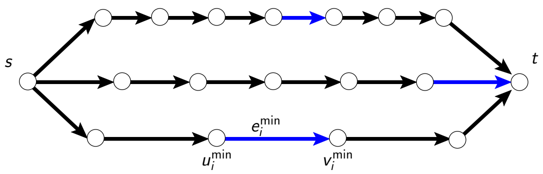




First Step: Parallel-Chain Graphs

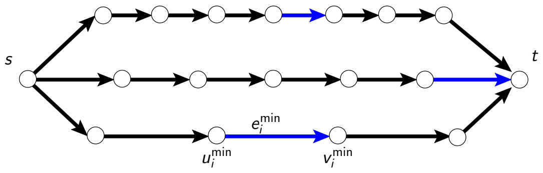


First Step: Parallel-Chain Graphs



Select edges with minimal weight on each branch: $e_1^{\min}, \ldots, e_B^{\min}$

First Step: Parallel-Chain Graphs

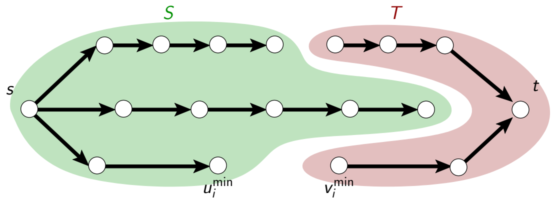


Select edges with minimal weight on each branch: $e_1^{\min}, \dots, e_B^{\min}$

Theorem

There exists a schedule with minimal memory which synchronises at $e_1^{\min}, \dots, e_B^{\min}$.

First Step: Parallel-Chain Graphs



Select edges with minimal weight on each branch: $e_1^{\min}, \dots, e_B^{\min}$

Theorem

There exists a schedule with minimal memory which synchronises at $e_1^{\min}, \dots, e_B^{\min}$.

Sketch of an optimal algorithm:

1. Apply optimal algorithm for out-trees on the left part
2. Apply optimal algorithm for in-trees on the right part

Synchronization on minimal cut – proof

- Consider optimal schedule σ_1
- Transform it into σ_2:
  1. Schedule all nodes from S (following σ_1)
  2. Then, schedule all nodes from T
- New schedule respect precedence constraints (processing order not changed within each branch)
- After scheduling all vertices from S, all $e_i^{\min}$ in memory
- Consider the memory when processing $u \in S$ from branch i:

	in σ_1	in σ_2
edge from branch $j \neq i$	some edge (v, w)	$\begin{cases} (v,w) & \text{if } v \in S \\ e_j^{\min} & \text{otherwise} \end{cases} \Rightarrow$

Memory needed when processing u not larger in σ_2

- Same analysis if $u \in T$

Synchronization on minimal cut – proof

- Consider optimal schedule σ_1
- Transform it into σ_2:
  1. Schedule all nodes from S (following σ_1)
  2. Then, schedule all nodes from T
- New schedule respect precedence constraints (processing order not changed within each branch)
- After scheduling all vertices from S, all $e_i^{\min}$ in memory
- Consider the memory when processing $u \in S$ from branch i:

	in σ_1	in σ_2	
edge from branch $j \neq i$	some edge (v, w)	$\begin{cases} (v,w) & \text{if } v \in S \\ e_j^{\min} & \text{otherwise} \end{cases}$	$\Rightarrow$

Memory needed when processing u not larger in σ_2

- Same analysis if $u \in T$

Synchronization on minimal cut – proof

- Consider optimal schedule σ_1
- Transform it into σ_2:
  1. Schedule all nodes from S (following σ_1)
  2. Then, schedule all nodes from T
- New schedule respect precedence constraints (processing order not changed within each branch)
- After scheduling all vertices from S, all $e_i^{\min}$ in memory
- Consider the memory when processing $u \in S$ from branch i:

	in σ_1	in σ_2
edge from branch $j \neq i$	some edge (v, w)	$\begin{cases} (v, w) & \text{if } v \in S \\ e_j^{\min} & \text{otherwise} \end{cases}$

$\Rightarrow$

Memory needed when processing u not larger in σ_2

- Same analysis if $u \in T$

Synchronization on minimal cut – proof

- Consider optimal schedule σ_1
- Transform it into σ_2:
  1. Schedule all nodes from S (following σ_1)
  2. Then, schedule all nodes from T
- New schedule respect precedence constraints (processing order not changed within each branch)
- After scheduling all vertices from S, all $e_i^{\min}$ in memory
- Consider the memory when processing $u \in S$ from branch i:

	in σ_1	in σ_2
edge from branch $j \neq i$	some edge (v, w)	$\begin{cases} (v, w) & \text{if } v \in S \\ e_j^{\min} & \text{otherwise} \end{cases}$ $\Rightarrow$

Memory needed when processing u not larger in σ_2

- Same analysis if $u \in T$

Synchronization on minimal cut – proof

- Consider optimal schedule σ_1
- Transform it into σ_2:
  1. Schedule all nodes from S (following σ_1)
  2. Then, schedule all nodes from T
- New schedule respect precedence constraints (processing order not changed within each branch)
- After scheduling all vertices from S, all $e_i^{\min}$ in memory
- Consider the memory when processing $u \in S$ from branch i:

	in σ_1	in σ_2
edge from branch $j \neq i$	some edge (v, w)	$\begin{cases} (v, w) & \text{if } v \in S \\ e_j^{\min} & \text{otherwise} \end{cases}$

$\Rightarrow$ Memory needed when processing u not larger in σ_2

- Same analysis if $u \in T$

Synchronization on minimal cut – proof

- Consider optimal schedule σ_1
- Transform it into σ_2:
  1. Schedule all nodes from S (following σ_1)
  2. Then, schedule all nodes from T
- New schedule respect precedence constraints (processing order not changed within each branch)
- After scheduling all vertices from S, all $e_i^{\min}$ in memory
- Consider the memory when processing $u \in S$ from branch i:

	in σ_1	in σ_2	
edge from branch $j \neq i$	some edge (v, w)	$\begin{cases} (v, w) & \text{if } v \in S \\ e_j^{\min} & \text{otherwise} \end{cases}$	$\Rightarrow$

Memory needed when processing u not larger in σ_2

- Same analysis if $u \in T$

Synchronization on minimal cut – proof

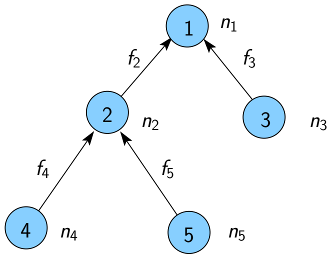
- Consider optimal schedule σ_1
- Transform it into σ_2:
  1. Schedule all nodes from S (following σ_1)
  2. Then, schedule all nodes from T
- New schedule respect precedence constraints (processing order not changed within each branch)
- After scheduling all vertices from S, all e_i^{min} in memory
- Consider the memory when processing $u \in S$ from branch i:

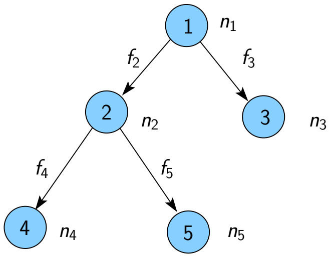
	in σ_1	in σ_2	
edge from branch $j \neq i$	some edge (v, w)	$\begin{cases} (v, w) & \text{if } v \in S \\ e_j^{\text{min}} & \text{otherwise} \end{cases}$	$\Rightarrow$

Memory needed when processing u not larger in σ_2

- Same analysis if $u \in T$

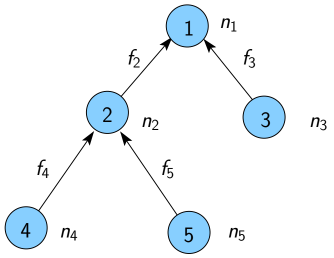
From in-trees to out-trees

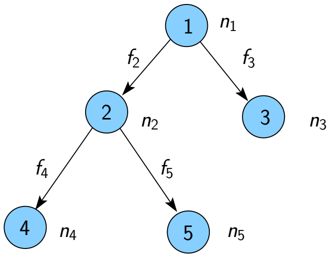




- Given a schedule σ_1 with memory M for the left in-tree, derive a schedule σ_2 for the right out-tree

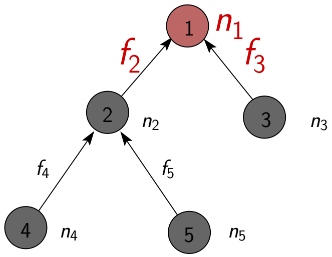
From in-trees to out-trees

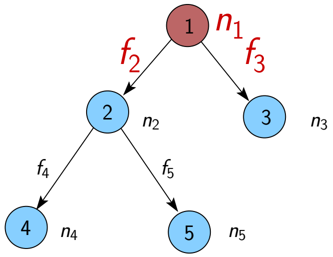




- Given a schedule σ_1 with memory M for the left in-tree, derive a schedule σ_2 for the right out-tree: obtained by reversing all edges

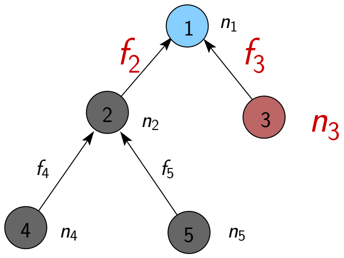
From in-trees to out-trees

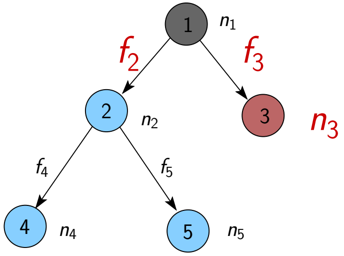




- Given a schedule σ_1 with memory M for the left in-tree, derive a schedule σ_2 for the right out-tree: obtained by reversing all edges

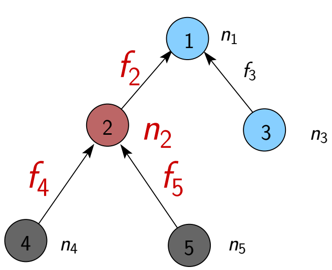
From in-trees to out-trees

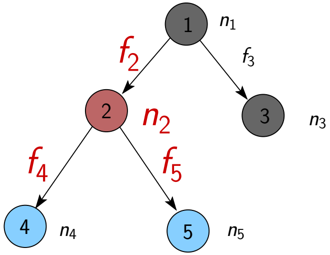




- Given a schedule σ_1 with memory M for the left in-tree, derive a schedule σ_2 for the right out-tree: obtained by reversing all edges

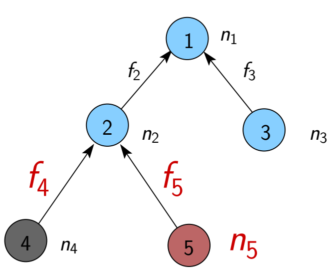
From in-trees to out-trees

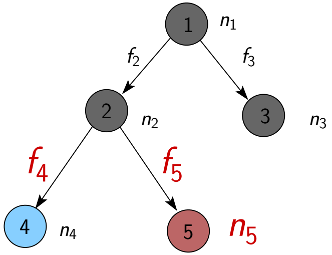




- Given a schedule σ_1 with memory M for the left in-tree, derive a schedule σ_2 for the right out-tree: obtained by reversing all edges

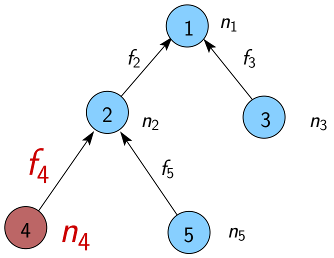
From in-trees to out-trees

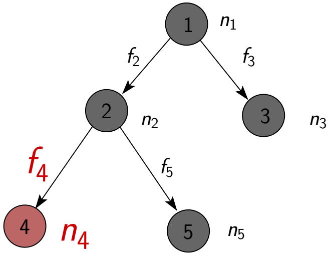




- Given a schedule σ_1 with memory M for the left in-tree, derive a schedule σ_2 for the right out-tree: obtained by reversing all edges

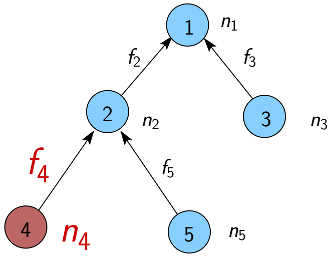
From in-trees to out-trees

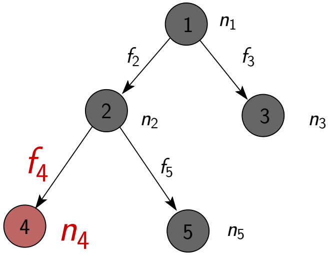




- Given a schedule σ_1 with memory M for the left in-tree, derive a schedule σ_2 for the right out-tree: obtained by reversing all edges

From in-trees to out-trees





- Given a schedule σ_1 with memory M for the left in-tree, derive a schedule σ_2 for the right out-tree: obtained by reversing all edges
- Choose $\sigma_2 = \text{reverse}(\sigma_1)$

General Series-Parallel Graphs

Principle:

- Follow the recursive definition of the SP-graph
- Compute both optimal schedule and minimal cut
- Replace subgraphs by chains of nodes (based on opt. sched.)

For sequential composition:

- Select minimal cut
- Concatenate schedules

For parallel composition (as for Parallel-Chains):

- Merge cuts
- On the left part, use algo. for out-trees for merging schedules
- On the right, use algo. for in-trees for merging schedules

Simple algorithm vs. very complex proof of optimality

General Series-Parallel Graphs

Principle:

- Follow the recursive definition of the SP-graph
- Compute both optimal schedule and minimal cut
- Replace subgraphs by chains of nodes (based on opt. sched.)

For sequential composition:

- Select minimal cut
- Concatenate schedules

For parallel composition (as for Parallel-Chains):

- Merge cuts
- On the left part, use algo. for out-trees for merging schedules
- On the right, use algo. for in-trees for merging schedules

Simple algorithm vs. very complex proof of optimality

General Series-Parallel Graphs

Principle:

- Follow the recursive definition of the SP-graph
- Compute both optimal schedule and minimal cut
- Replace subgraphs by chains of nodes (based on opt. sched.)

For sequential composition:

- Select minimal cut
- Concatenate schedules

For parallel composition (as for Parallel-Chains):

- Merge cuts
- On the left part, use algo. for out-trees for merging schedules
- On the right, use algo. for in-trees for merging schedules

Simple algorithm vs. very complex proof of optimality

Outline

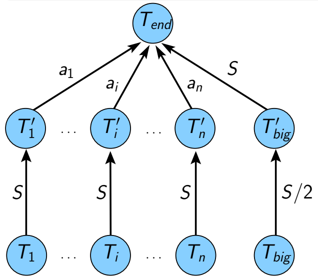
Minimizing I/Os for Trees

Problem:

- Available memory M too small to compute the whole tree
- Some data needs to be written to disk, and read back later
- Objective: minimize the amount of I/Os (total volume)

Theorem

When data must either be kept in memory or fully evicted to disk, deciding which data to write to disk is NP-complete.



$n_i = 0$ for all tasks

Reduction from Partition:

- Integers $a_1, \ldots a_n$, $S = \sum_i a_i$
- Split in two subsets of sum $S/2$

Memory $M = 2S$
Is it possible to schedule the tree with a volume of I/O at most $S/2$?

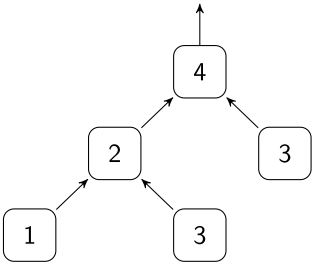
Minimizing I/O for Trees – with Paging

With paging:

- Partial data may be written to disk
- I/O cost metric: volume of data written to disk

Simpler model of memory/computation:

- memory weight only on edges output of $i = w_i$
- When processing a node, max(input, output) is needed
- Can easily emulate previous model (on the board)



Memory: 0 / 5

Disk: 0

I/Os: 0

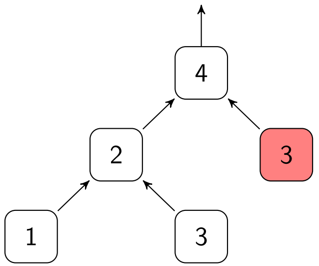
Minimizing I/O for Trees – with Paging

With paging:

- Partial data may be written to disk
- I/O cost metric: volume of data written to disk

Simpler model of memory/computation:

- memory weight only on edges output of $i = w_i$
- When processing a node, max(input, output) is needed
- Can easily emulate previous model (on the board)



Memory: 3 / 5

Disk: 0

I/Os: 0

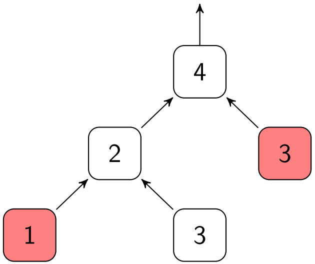
Minimizing I/O for Trees – with Paging

With paging:

- Partial data may be written to disk
- I/O cost metric: volume of data written to disk

Simpler model of memory/computation:

- memory weight only on edges output of $i = w_i$
- When processing a node, max(input, output) is needed
- Can easily emulate previous model (on the board)



Memory: 4 / 5

Disk: 0

I/Os: 0

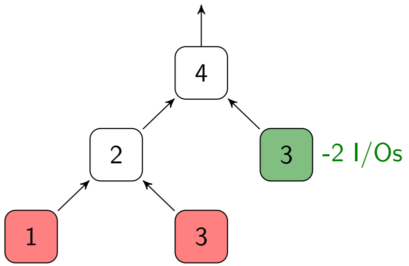
Minimizing I/O for Trees – with Paging

With paging:

- Partial data may be written to disk
- I/O cost metric: volume of data written to disk

Simpler model of memory/computation:

- memory weight only on edges output of $i = w_i$
- When processing a node, max(input, output) is needed
- Can easily emulate previous model (on the board)



Memory: 5 / 5

Disk: 2

I/Os: 2

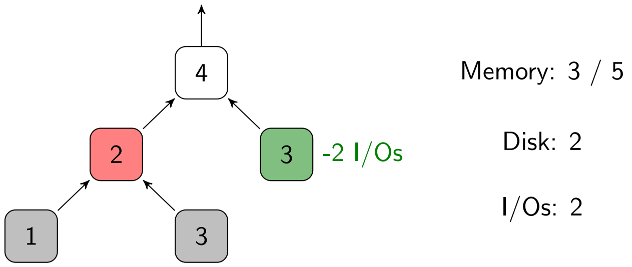
Minimizing I/O for Trees – with Paging

With paging:

- Partial data may be written to disk
- I/O cost metric: volume of data written to disk

Simpler model of memory/computation:

- memory weight only on edges output of $i = w_i$
- When processing a node, max(input, output) is needed
- Can easily emulate previous model (on the board)



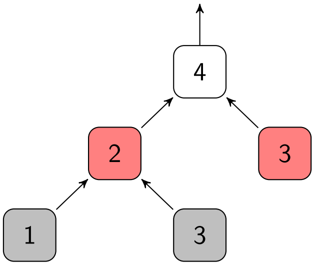
Minimizing I/O for Trees – with Paging

With paging:

- Partial data may be written to disk
- I/O cost metric: volume of data written to disk

Simpler model of memory/computation:

- memory weight only on edges output of $i = w_i$
- When processing a node, max(input, output) is needed
- Can easily emulate previous model (on the board)



Memory: 5 / 5

Disk: 0

I/Os: 2

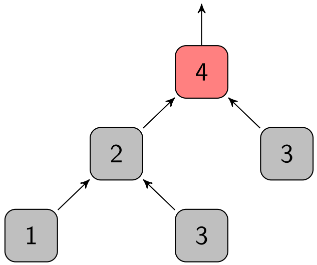
Minimizing I/O for Trees – with Paging

With paging:

- Partial data may be written to disk
- I/O cost metric: volume of data written to disk

Simpler model of memory/computation:

- memory weight only on edges output of $i = w_i$
- When processing a node, max(input, output) is needed
- Can easily emulate previous model (on the board)



Memory: 4 / 5

Disk: 0

I/Os: 2

Description of a solution

Traversal

- Schedule σ: $\sigma(i) = t$ if task i is the t-th executed task
- I/O function τ: output data of task i has $\tau(i)$ slots written to disk
- W.l.o.g. data written to disk ASAP and read ALAP

Validity of a traversal

- Schedule respects precedences
- I/Os consistent: $\tau(i) \le w_i$
- The main memory (size M) is never exceeded, $\forall i \in V$:

$$\left(\sum_{\substack{(k,p)\in E \\ \sigma(k)<\sigma(i)<\sigma(p)}} (w_k - \tau(k)) \right) + \max\left(w_i, \sum_{(j,i)\in E} w_j \right) \le M$$

Description of a solution

Traversal

- Schedule σ: $\sigma(i) = t$ if task i is the t-th executed task
- I/O function τ: output data of task i has $\tau(i)$ slots written to disk
- W.l.o.g. data written to disk ASAP and read ALAP

Validity of a traversal

- Schedule respects precedences
- I/Os consistent: $\tau(i) \leq w_i$
- The main memory (size M) is never exceeded, $\forall i \in V$:

$$\left(\sum_{\substack{(k,p)\in E \\ \sigma(k)<\sigma(i)<\sigma(p)}} (w_k - \tau(k)) \right) + \max\left(w_i, \sum_{(j,i)\in E} w_j \right) \leq M$$

Description of a solution

Traversal

- Schedule σ: $\sigma(i) = t$ if task i is the t-th executed task
- I/O function τ: output data of task i has $\tau(i)$ slots written to disk
- W.l.o.g. data written to disk ASAP and read ALAP

Validity of a traversal

- Schedule respects precedences
- I/Os consistent: $\tau(i) \leq w_i$
- The main memory (size M) is never exceeded, $\forall i \in V$:

$$\left(\sum_{\substack{(k,p)\in E \\ \sigma(k)<\sigma(i)<\sigma(p)}} (w_k - \tau(k)) \right) + \max\left(w_i, \sum_{(j,i)\in E} w_j \right) \leq M$$

Description of a solution

Traversal

- Schedule σ: $\sigma(i) = t$ if task i is the t-th executed task
- I/O function τ: output data of task i has $\tau(i)$ slots written to disk
- W.l.o.g. data written to disk ASAP and read ALAP

Validity of a traversal

- Schedule respects precedences
- I/Os consistent: $\tau(i) \leq w_i$
- The main memory (size M) is never exceeded, $\forall i \in V$:

$$\left(\sum_{\substack{(k,p)\in E \\ \sigma(k)<\sigma(i)<\sigma(p)}} (w_k - \tau(k)) \right) \quad + \quad \max\left(w_i, \sum_{(j,i)\in E} w_j \right) \quad \leq \quad M$$

Description of a solution

Traversal

- Schedule σ: $\sigma(i) = t$ if task i is the t-th executed task
- I/O function τ: output data of task i has $\tau(i)$ slots written to disk
- W.l.o.g. data written to disk ASAP and read ALAP

Validity of a traversal

- Schedule respects precedences
- I/Os consistent: $\tau(i) \leq w_i$
- The main memory (size M) is never exceeded, $\forall i \in V$:

$$\left(\sum_{\substack{(k,p)\in E \\ \sigma(k)<\sigma(i)<\sigma(p)}} (w_k - \tau(k)) \right) + \max\left(w_i, \sum_{(j,i)\in E} w_j \right) \leq M$$

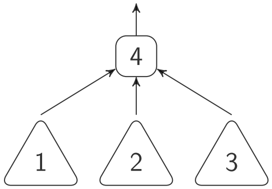
Objective

The MinIO problem

Given a tree G and a memory limit M, find a valid traversal that minimizes the total amount of I/Os (that is, $\sum \tau(i)$).

An interesting subclass: postorder traversals

- Fully process a subtree before starting a new one
- Completely characterized by the execution order of subtrees
- Widely used in sparse matrix software (e.g., MUMPS, QR-MUMPS)



Preliminary results

Let (σ, τ) be an optimal traversal for MinIO of a given instance

Lemma (Schedule is enough)

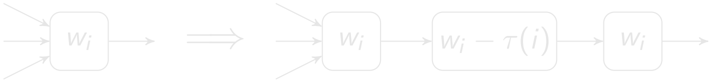
Given σ: the Furthest In the Future I/O policy minimizes I/Os.

Lemma (I/O function is enough)

Given τ: a valid traversal (σ', τ) can be computed in polynomial time.

Proof.

Expand each node following:



Then minimize the memory peak.

Preliminary results

Let (σ, τ) be an optimal traversal for MinIO of a given instance

Lemma (Schedule is enough)

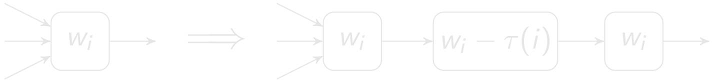
Given σ: the Furthest In the Future I/O policy minimizes I/Os.

Lemma (I/O function is enough)

Given τ: a valid traversal (σ', τ) can be computed in polynomial time.

Proof.

Expand each node following:



Then minimize the memory peak.

Preliminary results

Let (σ, τ) be an optimal traversal for MinIO of a given instance

Lemma (Schedule is enough)

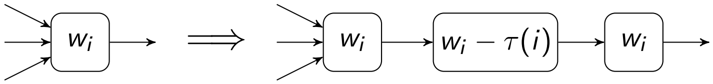
Given σ: the Furthest In the Future I/O policy minimizes I/Os.

Lemma (I/O function is enough)

Given τ: a valid traversal (σ', τ) can be computed in polynomial time.

Proof.

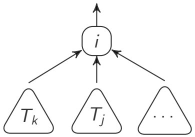
Expand each node following:



Then minimize the memory peak.

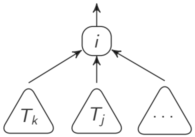
Postorder algorithms [Liu 1986, Agullo et al. 2010]

- When executing T_i : order of execution of children of i



Postorder algorithms [Liu 1986, Agullo et al. 2010]

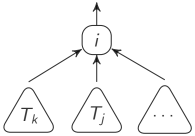
- When executing T_i : order of execution of children of i
- First compute the storage requirement of subtree T_i:



Postorder algorithms [Liu 1986, Agullo et al. 2010]

- When executing T_i : order of execution of children of i
- First compute the storage requirement of subtree T_i:

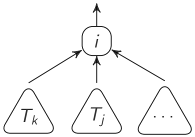
$$S_i = \max\left(w_i \ , \max_{j \in Chil(i)} \left(S_j + \sum_{\substack{k \in Chil(i) \\ \sigma(k) < \sigma(j)}} w_k \right)\right)$$



Postorder algorithms [Liu 1986, Agullo et al. 2010]

- When executing T_i : order of execution of children of i
- First compute the storage requirement of subtree T_i:

$$S_i = \max\left(w_i \;,\; \max_{j \in Chil(i)}\left(S_j + \sum_{\substack{k \in Chil(i)\\ \sigma(k)<\sigma(j)}} w_k\right)\right)$$

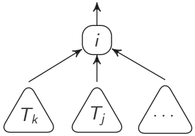


- Memory really used: $A_i = \min(S_i, M)$

Postorder algorithms [Liu 1986, Agullo et al. 2010]

- When executing T_i : order of execution of children of i
- First compute the storage requirement of subtree T_i:

$$S_i = \max\left(w_i \ , \max_{j\in Chil(i)}\left(S_j + \sum_{\substack{k\in Chil(i)\\ \sigma(k)<\sigma(j)}} w_k\right)\right)$$



- Memory really used: $A_i = \min(S_i, M)$
- For a given order σ, the volume of I/O is given by:

$$V_i = \max\left(0, \max_{j\in Chil(i)}\left(A_j + \sum_{\substack{k\in Chil(i)\\ \sigma(k)<\sigma(j)}} w_k\right) - M\right) + \sum_{j\in Chil(i)} V_j$$

Best Postorder for Minimizing I/Os

For a given order σ, the volume of I/O is given by:

$$V_i = \max\left(0, \max_{j \in Chil(i)} \left(A_j + \sum_{\substack{k \in Chil(i) \\ \sigma(k) < \sigma(j)}} w_k\right) - M\right) + \sum_{j \in Chil(i)} V_j$$

Theorem

Given a set of values (x_i, y_i), the minimum of $\max(x_i + \sum_{j<i} y_j)$ is obtained by sorting the sequence by non-increasing $x_i - y_i$.

Corollary

The postorder traversal that minimizes I/Os sorts the subtrees by non-increasing $A_j - w_j$.

Minimizing I/Os for Homogeneous Trees

Theorem

Both PostOrderMinMem *and* PostOrderMinIO *minimize I/Os on homogeneous trees (unit sizes).*

Note: PostOrderMinMem does not rely on M so is optimal for any memory size and several memory layers (cache-oblivious)

Minimizing I/Os for Homogeneous Trees

Theorem

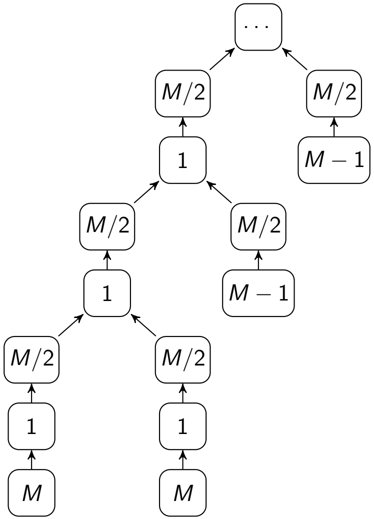
Both PostOrderMinMem *and* PostOrderMinIO *minimize I/Os on homogeneous trees (unit sizes).*

Note: PostOrderMinMem does not rely on M so is optimal for any memory size and several memory layers (cache-oblivious)

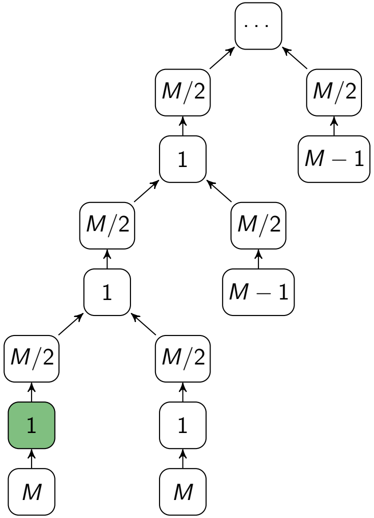
But PostOrderMinIO is not competitive on heterogeneous trees:

- Cases when PostOrderMinIO needs I/O when optimal traversal does not
- Even when the optimal traversal requires I/Os. . .

PostOrderMinIO is not competitive



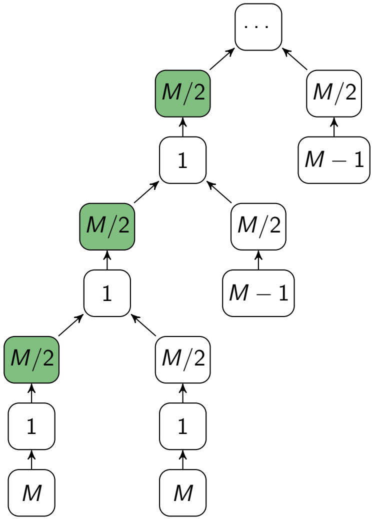
PostOrderMinIO is not competitive



I/O optimal

- Peak memory: $M+1$
- I/Os: 1

PostOrderMinIO is not competitive



I/O optimal

- Peak memory: $M+1$
- I/Os: 1

PostOrderMinIO

- Peak memory: $\frac{3}{2}M$
- I/Os: $\Theta(|V|M)$

Competitive ratio: $\Omega(|V|M)$

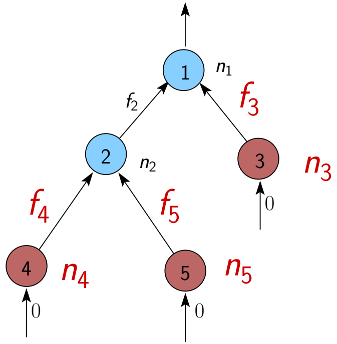
MinIO for Trees – Summary

- PostOrder algorithms optimal for homogeneous trees
- No known competitive algorithms for heterogeneous trees
- Heterogeneous trees: still an open problem!

Outline

Model for Parallel Tree Processing

- p uniform processors
- Shared memory of size M
- Task i has execution times p_i
- Parallel processing of nodes $\Rightarrow$ larger memory
- Trade-off time vs. memory



NP-Completeness in the Pebble Game Model

Background:

- Makespan minimization NP-complete for trees ($P|trees|C_{\max}$)
- Polynomial when unit-weight tasks ($P|p_i = 1, trees|C_{\max}$)
- Pebble game polynomial on trees

Pebble game model:

- Unit execution time: $p_i = 1$
- Unit memory costs: $n_i = 0, f_i = 1$
  (pebble edges, equivalent to pebble game for trees)

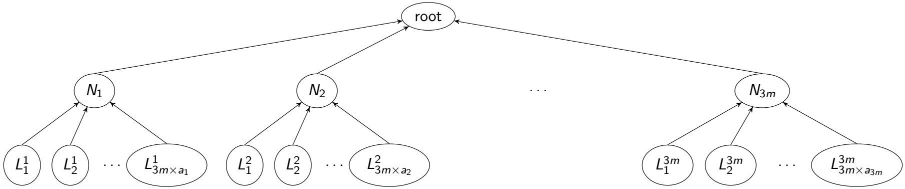
Theorem

Deciding whether a tree can be scheduled using at most B pebbles in at most C steps is NP-complete.

NP-Completeness – Proof

Reduction from 3-Partition:

- $3m$ integers a_i and B with $\sum ai = mB$,
- find m subsets S_k of 3 elements with $\sum_{i \in S_k} a_i = B$



Schedule the tree using:

- $p = 3mB$ processors,
- at most $B = 3m \times B + 3m$ pebbles,
- at most $C = 2m + 1$ steps.

Space-Time Tradeoff

Not possible to get a guarantee on both memory and time simultaneously:

Theorem 1

There is no algorithm that is both an α-approximation for makespan minimization and a β-approximation for memory peak minimization when scheduling tree-shaped task graphs.

Space-Time Tradeoff

Not possible to get a guarantee on both memory and time simultaneously:

Theorem 1

There is no algorithm that is both an α-approximation for makespan minimization and a β-approximation for memory peak minimization when scheduling tree-shaped task graphs.

Lemma

For a schedule with peak memory M and makespan $C_{\max}$,

$$M \times C_{\max} \geq 2(n-1)$$

Space-Time Tradeoff

Not possible to get a guarantee on both memory and time simultaneously:

Theorem 1

There is no algorithm that is both an α-approximation for makespan minimization and a β-approximation for memory peak minimization when scheduling tree-shaped task graphs.

Lemma

For a schedule with peak memory M and makespan $C_{\max}$,

$$M \times C_{\max} \geq 2(n-1)$$

Proof: each edge stays in memory for at least 2 steps.

Space-Time Tradeoff

Not possible to get a guarantee on both memory and time simultaneously:

Theorem 1

There is no algorithm that is both an α-approximation for makespan minimization and a β-approximation for memory peak minimization when scheduling tree-shaped task graphs.

Lemma

For a schedule with peak memory M and makespan $C_{\max}$,

$$M \times C_{\max} \geq 2(n-1)$$

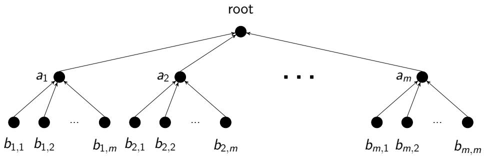
Proof: each edge stays in memory for at least 2 steps.

Corollary: Lower Bound on Space-Time Product

For a schedule with peak memory M and makespan $C_{\max}$,

$$M \times C_{\max} \geq \sum_i mem\_needed\_for\_task_i \times p_i$$

Space-Time Tradeoff – Proof



- With m^2 processors: $C^*_{\max} = 3$
- With 1 processor, sequentialize the a_i subtrees: $M^* = 2m$
- By contradiction, approximating both objectives: $C_{\max} \leq 3\alpha$ and $M \leq 2m\beta$
- But $M \times C_{\max} \geq 2(n-1) = 2m^2 + 2m$
- $2m^2 + 2m \leq 6m\alpha\beta$
- Contradiction for a sufficiently large value of m
- There does not exist any zenith-approximation algorithm

Complexity – Summary

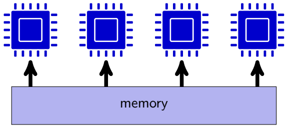
For task trees:

- Optimizing both makespan and memory is NP-Complete
  $\Rightarrow$ Same for minimizing makespan under memory budget
- No scheduling algorithm can be a constant factor approximation on both memory and makespan

Outline

Processing DAGs with Limited Memory

- Schedule general graphs
- On a shared-memory platform



First option: design good static scheduler:

- NP-complete, non-approximable
- Cannot react to unpredicted changes in the platform or inaccuracies in task timings

Second option:

- Limit memory consumption of any dynamic scheduler
  Target: runtime systems
- Without impacting too much parallelism

Outline

Memory model

Task graphs with:

- Vertex weights (w_i): task (estimated) durations
- Edge weights ($m_{i,j}$): data sizes

Memory model

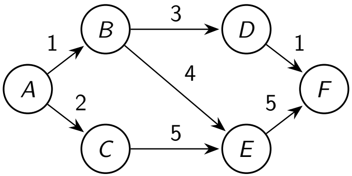
Task graphs with:

- Vertex weights (w_i): task (estimated) durations
- Edge weights ($m_{i,j}$): data sizes

Simple memory model: at the beginning of a task

- Inputs are freed (instantaneously)
- Outputs are allocated

At the end of a task: outputs stay in memory


$$\begin{tikzcd}
 & B \arrow[r, "3"] \arrow[rd, "4"] & D \arrow[rd, "1"] & \\
A \arrow[ru, "1"] \arrow[rd, "2"] & & & F \\
 & C \arrow[r, "5"] & E \arrow[ru, "5"] &
\end{tikzcd}$$

Memory model

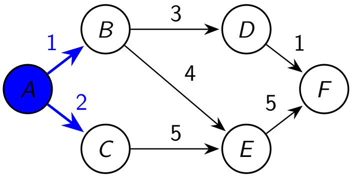
Task graphs with:

- Vertex weights (w_i): task (estimated) durations
- Edge weights ($m_{i,j}$): data sizes

Simple memory model: at the beginning of a task

- Inputs are freed (instantaneously)
- Outputs are allocated

At the end of a task: outputs stay in memory



Memory model

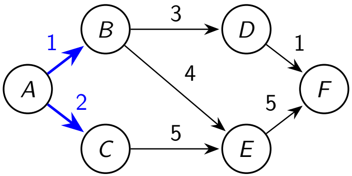
Task graphs with:

- Vertex weights (w_i): task (estimated) durations
- Edge weights ($m_{i,j}$): data sizes

Simple memory model: at the beginning of a task

- Inputs are freed (instantaneously)
- Outputs are allocated

At the end of a task: outputs stay in memory



Memory model

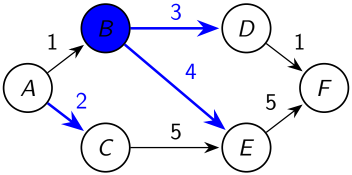
Task graphs with:

- Vertex weights (w_i): task (estimated) durations
- Edge weights ($m_{i,j}$): data sizes

Simple memory model: at the beginning of a task

- Inputs are freed (instantaneously)
- Outputs are allocated

At the end of a task: outputs stay in memory



Memory model

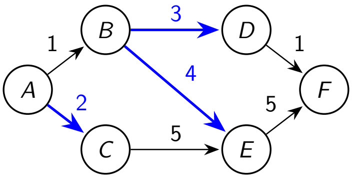
Task graphs with:

- Vertex weights (w_i): task (estimated) durations
- Edge weights ($m_{i,j}$): data sizes

Simple memory model: at the beginning of a task

- Inputs are freed (instantaneously)
- Outputs are allocated

At the end of a task: outputs stay in memory



Memory model

Task graphs with:

- Vertex weights (w_i): task (estimated) durations
- Edge weights ($m_{i,j}$): data sizes

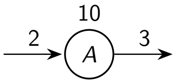
Simple memory model: at the beginning of a task

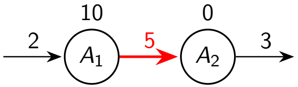
- Inputs are freed (instantaneously)
- Outputs are allocated

At the end of a task: outputs stay in memory

Emulation of other memory behaviours:

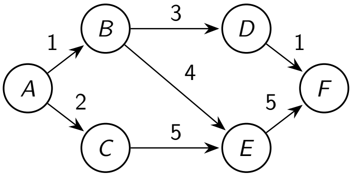
- Inputs + outputs allocated during task: duplicate nodes





Outline

Computing the maximum memory peak

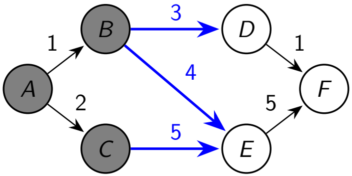


- What is the maximum memory of any parallel execution?

Computing the maximum memory peak

Topological cut: (S, T) with:

- S include the source node, T include the target node
- No edge from T to S
- Weight of the cut $=$ weight of all edges from S to T



Any topological cut corresponds to a possible state when all node in S are completed or being processed.

Two equivalent questions (in our model):

- What is the maximum memory of any parallel execution?
- What is the topological cut with maximum weight?

Computing the maximum topological cut

Literature:

- Lots of studies of various cuts in non-directed graphs ([Diaz 2000] on Graph Layout Problems)
- Minimum cut is polynomial on both directed/non-directed graphs
- Maximum cut NP-complete on both directed/non-directed graphs ([Karp 1972] for non-directed, [Lampis 2011] for directed ones)
- Not much for topological cuts

Theorem

Computing the maximum topological cut of a DAG can be done in polynomial time.

Maximum topological cut – using Linear Programming

- Consider one classical LP formulation for finding a minimum cut:

$$\min \sum_{(i,j)\in E} m_{i,j} d_{i,j}$$

$$\forall (i,j) \in E, \quad d_{i,j} \geq p_i - p_j$$

$$\forall (i,j) \in E, \quad d_{i,j} \geq 0$$

$$p_s = 1, \quad p_t = 0$$

Maximum topological cut – using Linear Programming

- Consider one classical LP formulation for finding a minimum cut:

$$\min \sum_{(i,j)\in E} m_{i,j} d_{i,j}$$
$$\forall (i,j) \in E, \quad d_{i,j} \geq p_i - p_j$$
$$\forall (i,j) \in E, \quad d_{i,j} \geq 0$$
$$p_s = 1, \quad p_t = 0$$

- Integer solution $\Leftrightarrow$ topological cut

Maximum topological cut – using Linear Programming

- Consider one classical LP formulation for finding a minimum cut:

$$\max \sum_{(i,j)\in E} m_{i,j} d_{i,j}$$
$$\forall (i,j) \in E, \quad d_{i,j} = p_i - p_j$$
$$\forall (i,j) \in E, \quad d_{i,j} \geq 0$$
$$p_s = 1, \quad p_t = 0$$

- Integer solution $\Leftrightarrow$ topological cut
- Then change the optimization direction (min $\rightarrow$ max)

Maximum topological cut – using Linear Programming

- Consider one classical LP formulation for finding a minimum cut:

$$\max \sum_{(i,j)\in E} m_{i,j} d_{i,j}$$
$$\forall (i,j) \in E, \quad d_{i,j} = p_i - p_j$$
$$\forall (i,j) \in E, \quad d_{i,j} \geq 0$$
$$p_s = 1, \quad p_t = 0$$

- Integer solution $\Leftrightarrow$ topological cut
- Then change the optimization direction (min $\rightarrow$ max)
- Draw w uniformly in $]0,1[$, define the cut such that $S_w = \{i \mid p_i > w\}, \quad T_w = \{i \mid p_i \leq w\}$
- Expected cost of this cut $= M^*$ (opt. rational solution)
- All cuts with random w have the same cost M^*

Maximum topological cut – direct algorithm

- Dual problem: Min-Flow *(larger than all edge weights)*
- Idea: use an optimal algorithm for Max-Flow

Algorithm sketch

$m_{i,j}$
$MinFlow_{i,j}$

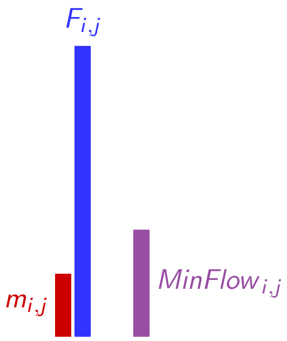
Complexity: same as maximum flow, e.g., $O(|V|^2|E|)$

Maximum topological cut – direct algorithm

- Dual problem: Min-Flow *(larger than all edge weights)*
- Idea: use an optimal algorithm for Max-Flow

Algorithm sketch

1. Build a large flow F on the graph G



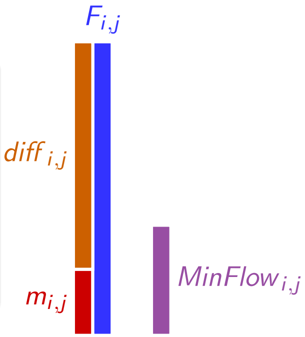
Complexity: same as maximum flow, e.g., $O(|V|^2|E|)$

Maximum topological cut – direct algorithm

- Dual problem: Min-Flow *(larger than all edge weights)*
- Idea: use an optimal algorithm for Max-Flow

Algorithm sketch

1. Build a large flow F on the graph G
2. Consider G^{diff} with edge weights $F_{i,j} - m_{i,j}$



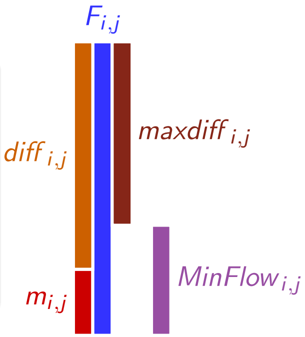
Complexity: same as maximum flow, e.g., $O(|V|^2|E|)$

Maximum topological cut – direct algorithm

- Dual problem: Min-Flow *(larger than all edge weights)*
- Idea: use an optimal algorithm for Max-Flow

Algorithm sketch

1. Build a large flow F on the graph G
2. Consider G^{diff} with edge weights $F_{i,j} - m_{i,j}$
3. Compute a maximum flow $maxdiff$ in G^{diff}



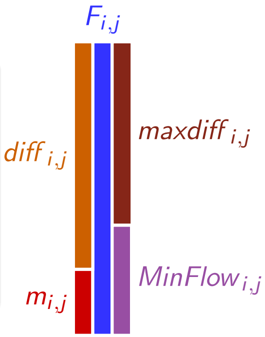
Complexity: same as maximum flow, e.g., $O(|V|^2|E|)$

Maximum topological cut – direct algorithm

- Dual problem: Min-Flow *(larger than all edge weights)*
- Idea: use an optimal algorithm for Max-Flow

Algorithm sketch

1. Build a large flow F on the graph G
2. Consider G^{diff} with edge weights $F_{i,j} - m_{i,j}$
3. Compute a maximum flow $maxdiff$ in G^{diff}
4. $F - maxdiff$ is a minimum flow in G
5. Residual graph $\rightarrow$ maximum topological cut



Complexity: same as maximum flow, e.g., $O(|V|^2|E|)$

Summary 1

Predict the maximal memory of any dynamic scheduling

$\Leftrightarrow$

Compute the maximal topological cut

Two algorithms:

- Linear program + rounding
- Direct algorithm based on MaxFlow/MinCut

Downsides:

- Large running time: $O(|V|^2|E|)$ or solving a LP
- May include edges corresponding to the computing of more than p tasks

Faster Max. Memory Computation for SP Graphs

Recursive algorithm to compute maximum topological cut on SP-graphs:

- Single edge $i \rightarrow j$:
  $M(G) = m_{i,j}$
- Series combination:
  $M(G) = max(M(G_1), M(G_2))$
- Parallel combination:
  $M(G) = M(G_1) + M(G_2)$

Complexity: $O(|E|)$
Proof:

- Consider tree of compositions: (full) binary tree
- $|E|$ leaves
- $|E| - 1$ internal nodes (compositions)

Maximum memory with p processors

Change in the model:

- Black (regular) edges
- Red edges corresponding to computations

Definition

P-MaxTopCut Given a graph with black/red edges and a number p of processor, what is the maximal weight of a topological cut including at most p red edges?

Theorem

P-MaxTopCut is NP-complete

Special Case of SP Graphs – Exact Algorithm

Compute the maximum memory with p red edges $M(G, p)$:

- Adapt previous algorithm:
  Compute $M(G, k)$ for each $k = 1, \ldots, p$

Special Case of SP Graphs – Exact Algorithm

Compute the maximum memory with p red edges $M(G,p)$:

- Adapt previous algorithm:
  Compute $M(G,k)$ for each $k = 1, \ldots, p$
- Single edge $i \to j$:
  $$M(G,k) = \begin{cases} m_{i,j} & \text{if edge is black or } k \geq 1 \\ -\infty & \text{otherwise} \end{cases}$$

Special Case of SP Graphs – Exact Algorithm

Compute the maximum memory with p red edges $M(G, p)$:

- Adapt previous algorithm:
  Compute $M(G, k)$ for each $k = 1, \dots, p$
- Single edge $i \to j$:
  $$M(G, k) = \begin{cases} m_{i,j} & \text{if edge is black or } k \geq 1 \\ -\infty & \text{otherwise} \end{cases}$$
- Series combination:
  $M(G, k) = max(M(G_1, k), M(G_2, k))$

Special Case of SP Graphs – Exact Algorithm

Compute the maximum memory with p red edges $M(G, p)$:

- Adapt previous algorithm:
  Compute $M(G, k)$ for each $k = 1, \ldots, p$
- Single edge $i \to j$:
  $$M(G, k) = \begin{cases} m_{i,j} & \text{if edge is black or } k \geq 1 \\ -\infty & \text{otherwise} \end{cases}$$
- Series combination:
  $M(G, k) = max(M(G_1, k), M(G_2, k))$
- Parallel combination:
  $M(G, k) = max_{j=0 \ldots k} M(G_1, j) + M(G_2, k - j)$

Complexity:

- Simple Dynamic Programming algorithm: $O(|E|p^2)$.
- By restricting the search on each subgraph to $w(G)$ (maximum width), and with tighter analysis: $O(|E|p)$.

Summary 2

Predict the maximal memory of any dynamic scheduling

$\Leftrightarrow$

Compute the maximal topological cut

Two algorithms:

- Linear program + rounding
- Direct algorithm based on MaxFlow/MinCut

Downsides:

- Large running time ($O(|V|^2|E|)$)
- Taking into account the bound on task being processed makes the problem NP complete

Special case of SP graphs:

- Max. Top. cut computed in $O(|E|)$
- Max. Top. cut with p procs computed in $O(|E|p)$

Outline

Coping with limiting memory

Problem:

- Limited available memory M
- Allow use of dynamic schedulers
- Avoid running out of memory
- Keep high level of parallelism (as much as possible)

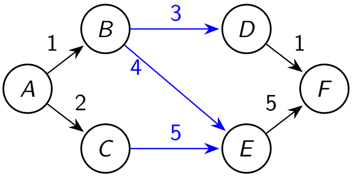
Coping with limiting memory

Problem:

- Limited available memory M
- Allow use of dynamic schedulers
- Avoid running out of memory
- Keep high level of parallelism (as much as possible)

Our solution:

- Add edges to guarantee that any parallel execution stays below M *fictitious dependences to reduce maximum memory*
- Minimize the obtained critical path



$M = 10$

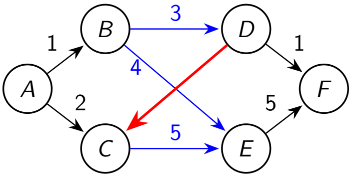
Coping with limiting memory

Problem:

- Limited available memory M
- Allow use of dynamic schedulers
- Avoid running out of memory
- Keep high level of parallelism (as much as possible)

Our solution:

- Add edges to guarantee that any parallel execution stays below M *fictitious dependences to reduce maximum memory*
- Minimize the obtained critical path



$M = 10$

Problem definition and complexity

Definition (PartialSerialization)

Given a DAG $G = (V, E)$ and a bound M, find a set of new edges E' such that $G' = (V, E \cup E')$ is a DAG, $MaxMem(G') \leq M$ and $CritPath(G')$ is minimized.

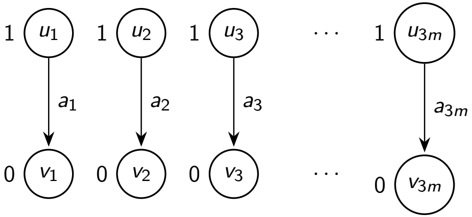
Theorem

PartialSerialization is NP-hard in the stronge sense.

NB: stays NP-hard if we are given a sequential schedule σ of G which uses at most a memory M.

NP-completeness – proof sketch

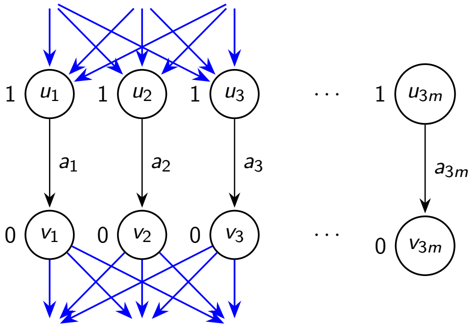
- Reduction from 3-Partition: a_i s.t. $\sum a_i = mB$, solution: m sets of 3 a_i's summing to B



- Set the memory bound to B
- Bound on the critical path: m

NP-completeness – proof sketch

- Reduction from 3-Partition: a_i s.t. $\sum a_i = mB$, solution: m sets of 3 a_i's summing to B



- Set the memory bound to B
- Bound on the critical path: m
- Solution to PartialSerialization $\Leftrightarrow$ group edges by 3 s.t. $\sum a_i = B$

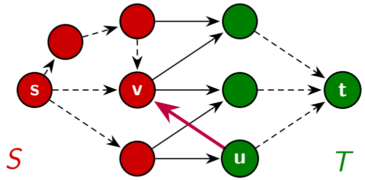
Outline

Heuristic solutions for PartialSerialization

Framework:

(inspired by [Sbîrlea et al. 2014])

1. Compute a max. top. cut (S, T)
2. If weight $\leq M$: succeeds
3. Add edge (u, v) with $u \in T$, $v \in S$ without creating cycles; or fail
4. Goto Step 1

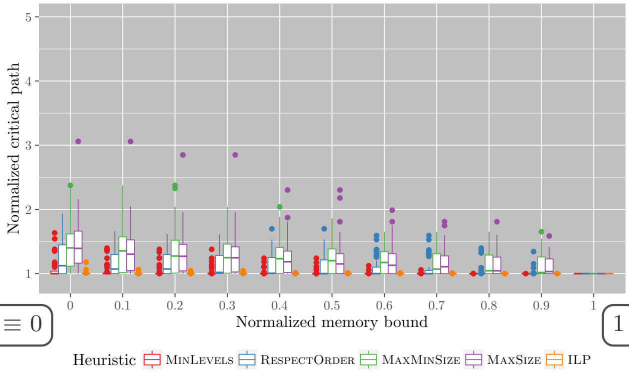


Several heuristic choices for Step 3:

MinLevels	does not create a large critical path
RespectOrder	follows a precomputed memory-efficient schedule, always succeeds
MaxSize	targets nodes dealing with large data
MaxMinSize	variant of MaxSize

Simulations: dense random graphs (25, 50, 100 nodes)

lower is better



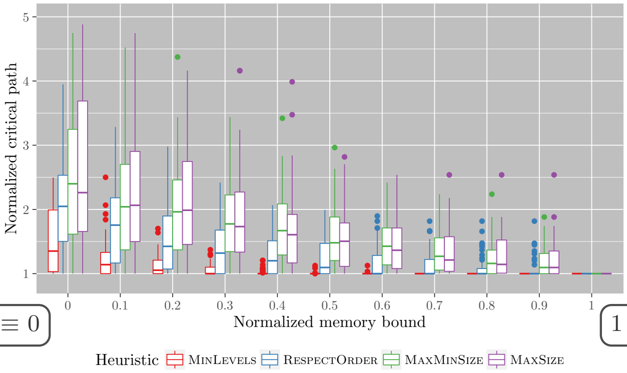
DFS memory $\equiv 0$

$1 \equiv$ MaxTopCut

- *x: memory (0 = DFS, 1 = MaxTopCut)*
  median ratio MaxTopCut / DFS memory ≈ 1.3
- *y: CP / original CP* $\rightarrow$ *lower is better*
- MinLevels performs best

Simulations: sparse random graphs (25, 50, 100 nodes)

lower is better

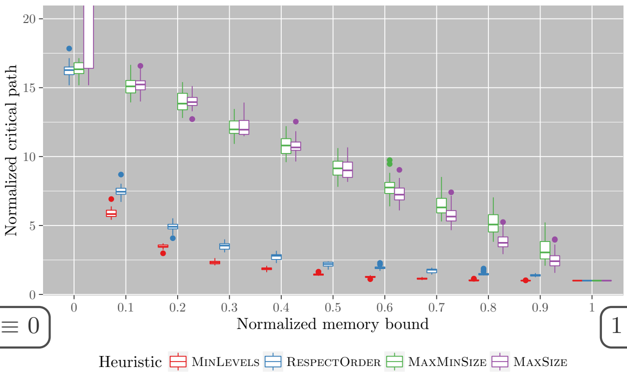


DFS memory $\equiv 0$

$1 \equiv$ MaxTopCut

- *x: memory (0 = DFS, 1 = MaxTopCut)*
  median ratio MaxTopCut / DFS memory $\approx$ 2
- *y: CP / original CP $\rightarrow$ lower is better*
- MinLevels performs best, but might fail

Simulations – Pegasus workflows (LIGO 100 nodes)



lower is better

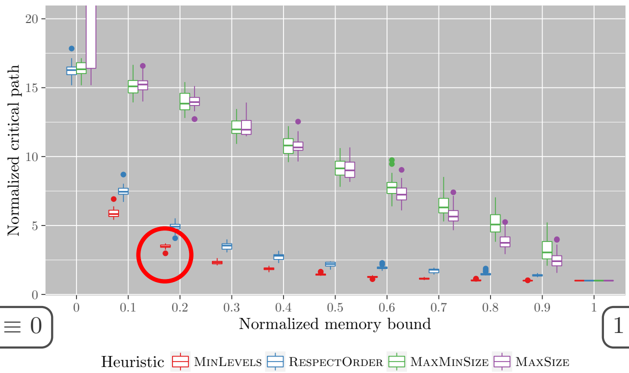
DFS memory $\equiv 0$

$1 \equiv$ MaxTopCut

- *Median ratio MaxTopCut / DFS $\approx$ 20*
- MinLevels performs best, RespectOrder always succeeds

Simulations – Pegasus workflows (LIGO 100 nodes)

lower is better



DFS memory $\equiv 0$

$1 \equiv$ MaxTopCut

- *Median ratio MaxTopCut / DFS $\approx$ 20*
- MinLevels performs best, RespectOrder always succeeds
- Memory divided by 5 for CP multiplied by 3

Summary – Memory-Aware DAG Scheduling

Several models:

1. Memory weights on edges and nodes,
   inputs+outputs+tmp needed to compute tasks
2. Memory weights only on edges
   Processing tasks $\Leftrightarrow$ replace inputs by outputs
3. (Memory increment on nodes)

    - Model 2 emulates 1, Model 3 emulates 1 and 2, ...
    - Choose the right model to solve each problem
    - Same for in-trees vs. out-trees

Summary – Memory-Aware DAG Scheduling

Several models:

1. Memory weights on edges and nodes,
   inputs+outputs+tmp needed to compute tasks
2. Memory weights only on edges
   Processing tasks $\Leftrightarrow$ replace inputs by outputs
3. (Memory increment on nodes)

   - Model 2 emulates 1, Model 3 emulates 1 and 2, ...
   - Choose the right model to solve each problem
   - Same for in-trees vs. out-trees

Results:

- One processor: optimal algorithms for trees (postorder or not)
- Several processors: NP-complete problem, no (α, β)-approx.
- Dynamic scheduling with memory bound:
  - Compute the worst memory: polynomial (linear for SP-graphs)
  - Limit memory: NP-complete, heuristic solutions