Part 3: Memory-Aware DAG Scheduling

CR05: Data Aware Algorithms

October 12 & 15, 2020
Outline

Minimize Memory for Trees

Minimize Memory for Series-Parallel Graphs

Minimize I/Os for Trees under Bounded Memory

Complexity and Space-Time Tradeoffs for Parallel Tree Processing

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
Summary of the course

- Part 1: Pebble Games
  models of computations with limited memory

- Part 2: External Memory and Cache Oblivous Algorithm
  2-level memory system, some parallelism (work stealing)

- Part 3: Streaming Algorithms
  Deal with big data, distributed computing

- Part 4: DAG scheduling (today)
  structured computations with limited memory

- Part 5: Communication Avoiding Algorithms
  regular computations (lin. algebra) in distributed setting
Introduction

- Directed Acyclic Graphs: express task dependencies
  - nodes: computational tasks
  - edges: dependencies
    (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 created dependancies
- 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
Introduction

- Directed Acyclic Graphs: express task dependencies
  - nodes: computational tasks
  - edges: dependencies
    (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 created dependencies
- 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
- Tasks have durations and memory demands

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

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
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
Going back to sequential processing

- Temporary data require memory
- Scheduling influences the peak memory

When minimum memory demand $>\text{available memory}$:

- 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

▶ 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 on 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
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 on 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
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 on 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
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: \textbf{NP-hard} on general graphs 😞

Sometimes restrict on 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

Minimize Memory for Trees

Minimize Memory for Series-Parallel Graphs

Minimize I/Os for Trees under Bounded Memory

Complexity and Space-Time Tradeoffs for Parallel Tree Processing

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

Minimize Memory for Trees

Minimize Memory for Series-Parallel Graphs

Minimize I/Os for Trees under Bounded Memory

Complexity and Space-Time Tradeoffs for Parallel Tree Processing

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
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 $i$: $\text{MemReq}(i) = \left( \sum_{j \in \text{Children}(i)} f_j \right) + n_i + f_i$
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 $i$: $\text{MemReq}(i) = \left( \sum_{j \in \text{Children}(i)} f_j \right) + n_i + f_i$
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 $i$: $\text{MemReq}(i) = \left( \sum_{j \in \text{Children}(i)} f_j \right) + n_i + f_i$
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 $i$: $\text{MemReq}(i) = \left( \sum_{j \in \text{Children}(i)} f_j \right) + n_i + f_i$
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 $i$: $\text{MemReq}(i) = \left( \sum_{j \in \text{Children}(i)} f_j \right) + n_i + f_i$
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 $i$: $\text{MemReq}(i) = \left( \sum_{j \in \text{Children}(i)} f_j \right) + n_i + f_i$
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 $1, \ldots, n$, the peak memory is:

$$\max\{P_1, \ldots, P_n\}$$
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 $1, \ldots, n$, the peak memory is:

$$\max\{P_1, f_1 + P_2, \ldots, f_n + P_n\}$$
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 $1, \ldots, n$, the peak memory is:

$$\max\{ P_1, f_1 + P_2, f_1 + f_2 + P_3, \ldots \}$$
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 $1, \ldots, n$, the peak memory is:

$$\max\{ P_1, f_1 + P_2, f_1 + f_2 + P_3, \ldots, \sum_{i<n} f_i + P_n, \}$$
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, \ldots, n, the peak memory is:

\[
\max\{P_1, f_1 + P_2, f_1 + f_2 + P_3, \ldots, \sum_{i<n} f_i + P_n, \sum_{i<n} f_i + n_r + f_r\}
\]
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 $1, \ldots, n$, the peak memory is:

\[
\max\{P_1, f_1 + P_2, f_1 + f_2 + P_3, \ldots, \sum_{i<n} f_i + P_n, \sum_{i<n} f_i + n_r + f_r\}
\]

- 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 $k$
  - $P_k - f_k \geq P_j - f_j$

<table>
<thead>
<tr>
<th></th>
<th>peak when $j$, then $k$</th>
<th>peak when $k$, then $j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>during first subtree</td>
<td>$\text{mem_before} + P_j$</td>
<td>$\text{mem_before} + P_k$</td>
</tr>
<tr>
<td>during second subtree</td>
<td>$\text{mem_before} + f_j + P_k$</td>
<td>$\text{mem_before} + f_k + P_j$</td>
</tr>
</tbody>
</table>

- $f_k + P_j \leq f_j + P_k$
- 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:
$$M_{\text{min}} = M + \epsilon + (b - 1)M/b$$

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.

\[ M_{\text{min}} = M + \epsilon + (b-1)\frac{M}{b} \]

\[ M_{\text{min}} = M + \epsilon + (b-1)\epsilon \]
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:
$$M_{\text{min}} = M + \epsilon + 2(b - 1)\frac{M}{b}$$

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

<table>
<thead>
<tr>
<th>actual assembly trees</th>
<th>random trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non optimal traversals</td>
<td>4.2%</td>
</tr>
<tr>
<td>Maximum increase compared to optimal</td>
<td>18%</td>
</tr>
<tr>
<td>Average increased compared to optimal</td>
<td>1%</td>
</tr>
</tbody>
</table>
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 \( H_2 \)
  - \( \ldots \)
  - 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

Minimize Memory for Trees

Minimize Memory for Series-Parallel Graphs

Minimize I/Os for Trees under Bounded Memory

Complexity and Space-Time Tradeoffs for Parallel Tree Processing

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
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
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

\[ SP_1 \rightarrow SP_2 \]

\[ SP_1 \leftrightarrow SP_2 \]
First Step: Parallel-Chain Graphs

Select edges with minimal weight on each branch: \( e_{\min 1}, \ldots, e_{\min B} \).

Theorem
There exists a schedule with minimal memory which synchronizes at \( e_{\min 1}, \ldots, e_{\min B} \).

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
First Step: Parallel-Chain Graphs

Select edges with minimal weight on each branch: $e_1^{\text{min}}, \ldots, e_B^{\text{min}}$
First Step: Parallel-Chain Graphs

Select edges with minimal weight on each branch: $e_{1 \min}, \ldots, e_{B \min}$

Theorem

There exists a schedule with minimal memory which synchronises at $e_{1 \min}, \ldots, e_{B \min}$. 
Select edges with minimal weight on each branch: $e_1^\text{min}, \ldots, e_B^\text{min}$

**Theorem**

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

**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
Consider optimal schedule $\sigma_1$

Transform it into $\sigma_2$:
1. Schedule all nodes from $S$ (following $\sigma_1$)
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 $e_i^{\text{min}}$ in memory

Consider the memory when processing $u \in L$ from branch $i$:

<table>
<thead>
<tr>
<th>edge from branch $j \neq i$</th>
<th>some edge $(v, w)$</th>
<th>in $\sigma_1$</th>
<th>in $\sigma_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(v, w)$ if $v \in L$</td>
<td>$e_j^{\text{min}}$ otherwise</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

⇒ Memory needed when processing $u$ not larger in $\sigma_2$

Same analysis if $u \in T$
Synchronization on minimal cut – proof

- Consider optimal schedule $\sigma_1$
- Transform it into $\sigma_2$:
  1. Schedule all nodes from $S$ (following $\sigma_1$)
  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 $e_{min}$ in memory
- Consider the memory when processing $u \in L$ from branch $i$:

<table>
<thead>
<tr>
<th>in $\sigma_1$</th>
<th>in $\sigma_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>edge from branch $j \neq i$</td>
<td>some edge $(v, w)$</td>
</tr>
</tbody>
</table>

  - If $v \in L$:
    - $e_{min}$
  - Otherwise:
    - $e_{min}$

  ⇒ Memory needed when processing $u$ not larger in $\sigma_2$
- Same analysis if $u \in T$
Synchronization on minimal cut – proof

- Consider optimal schedule $\sigma_1$
- Transform it into $\sigma_2$:
  1. Schedule all nodes from $S$ (following $\sigma_1$)
  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 $e_i^{\text{min}}$ in memory
- Consider the memory when processing $u \in L$ from branch $i$:

<table>
<thead>
<tr>
<th>edge from branch $j \neq i$</th>
<th>some edge $(v, w)$</th>
<th>in $\sigma_1$</th>
<th>in $\sigma_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>(v, w) if v \in L</td>
<td>$e_j^{\text{min}}$ otherwise</td>
</tr>
</tbody>
</table>

⇒ Memory needed when processing $u$ not larger in $\sigma_2$
- Same analysis if $u \in T$
Synchronization on minimal cut – proof

- Consider optimal schedule $\sigma_1$
- Transform it into $\sigma_2$:
  1. Schedule all nodes from $S$ (following $\sigma_1$)
  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 $e_i^{\text{min}}$ in memory
- Consider the memory when processing $u \in L$ from branch $i$:

<table>
<thead>
<tr>
<th>edge from branch $j \neq i$</th>
<th>some edge $(v, w)$</th>
<th>in $\sigma_1$</th>
<th>in $\sigma_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$(v, w)$</td>
<td>$e_j^{\text{min}}$ if $v \in L$ otherwise</td>
</tr>
</tbody>
</table>

⇒ Memory needed when processing $u$ not larger in $\sigma_2$
- Same analysis if $u \in T$
Synchronization on minimal cut – proof

- Consider optimal schedule $\sigma_1$
- Transform it into $\sigma_2$:
  1. Schedule all nodes from $S$ (following $\sigma_1$)
  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 $u \in L$ from branch $i$:

<table>
<thead>
<tr>
<th>edge from branch $j \neq i$</th>
<th>in $\sigma_1$</th>
<th>in $\sigma_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>some edge $(v, w)$</td>
<td></td>
<td>$\begin{cases} (v, w) &amp; \text{if } v \in L \ e_j^{min} &amp; \text{otherwise} \end{cases}$</td>
</tr>
</tbody>
</table>

$\Rightarrow$ Memory needed when processing $u$ not larger in $\sigma_2$
- Same analysis if $u \in T$
Consider optimal schedule $\sigma_1$

Transform it into $\sigma_2$:
1. Schedule all nodes from $S$ (following $\sigma_1$)
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 $e_{i}^{\text{min}}$ in memory

Consider the memory when processing $u \in L$ from branch $i$:

<table>
<thead>
<tr>
<th>Edge from branch $j \neq i$</th>
<th>Some edge $(v, w)$</th>
<th>Memory needed when processing $u$ not larger in $\sigma_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>in $\sigma_1$</td>
<td>$(v, w)$ if $v \in L$</td>
<td>$e_{j}^{\text{min}}$ otherwise</td>
</tr>
<tr>
<td>in $\sigma_2$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

⇒ Same analysis if $u \in T$
Consider optimal schedule $\sigma_1$

Transform it into $\sigma_2$:
1. Schedule all nodes from $S$ (following $\sigma_1$)
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 $e_i^{\text{min}}$ in memory

Consider the memory when processing $u \in L$ from branch $i$:

<table>
<thead>
<tr>
<th>in $\sigma_1$</th>
<th>in $\sigma_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>edge from branch $j \neq i$</td>
<td>some edge $(v, w)$</td>
</tr>
</tbody>
</table>

\[
\begin{cases}
(v, w) & \text{if } v \in L \\
e_j^{\text{min}} & \text{otherwise}
\end{cases}
\]

⇒ Memory needed when processing $u$ not larger in $\sigma_2$

Same analysis if $u \in T$
Given a schedule $\sigma_1$ with memory $M$ for the left in-tree, derive a schedule $\sigma_2$ for the right out-tree, obtained by reversing all edges?
Given a schedule $\sigma_1$ with memory $M$ for the left in-tree, derive a schedule $\sigma_2$ for the right out-tree, obtained by reversing all edges?
Given a schedule $\sigma_1$ with memory $M$ for the left in-tree, derive a schedule $\sigma_2$ for the right out-tree, obtained by reversing all edges?
From in-trees to out-trees

Given a schedule \( \sigma_1 \) with memory \( M \) for the left in-tree, derive a schedule \( \sigma_2 \) for the right out-tree, obtained by reversing all edges?
Given a schedule $\sigma_1$ with memory $M$ for the left in-tree, derive a schedule $\sigma_2$ for the right out-tree, obtained by reversing all edges?
From in-trees to out-trees

Given a schedule $\sigma_1$ with memory $M$ for the left in-tree, derive a schedule $\sigma_2$ for the right out-tree, obtained by reversing all edges?
Given a schedule $\sigma_1$ with memory $M$ for the left in-tree, derive a schedule $\sigma_2$ 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
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

Minimize Memory for Trees

Minimize Memory for Series-Parallel Graphs

Minimize I/Os for Trees under Bounded Memory

Complexity and Space-Time Tradeoffs for Parallel Tree Processing

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
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.

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, \( \text{max}(\text{input, output}) \) 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, $\max(\text{input}, \text{output})$ 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, $\max(\text{input, output})$ 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, $\max(\text{input, output})$ 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, $\max(\text{input, output})$ 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, \( \max(\text{input}, \text{output}) \) is needed
▶ Can easily emulate previous model (on the board)

Memory: 4 / 5
Disk: 0
I/Os: 2
Description of a solution

**Traversal**

- **Schedule** $\sigma$: $\sigma(i) = t$ if task $i$ is the $t$-th executed
- **I/O function** $\tau$: 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: $\forall i \in V$

$$\left( \sum_{(k,i) \in E} (w_k - \tau(k)) \right) + \max \left( \sum w_i, \sum_{(i,j) \in E} w_j \right) \leq M$$
Description of a solution

Traversal

- **Schedule** \( \sigma \): \( \sigma(i) = t \) if task \( i \) is the \( t \)-th executed
- **I/O function** \( \tau \): 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, \( \forall i \in V \):

\[
\left( \sum_{{(k,p) \in E} \atop {\sigma(k) < \sigma(i) < \sigma(p)}} (w_k - \tau(k)) \right) + \max \left( w_i, \sum_{{(j,i) \in E}} w_j \right) \leq M
\]
Description of a solution

Traversal

- Schedule $\sigma$: $\sigma(i) = t$ if task $i$ is the $t$-th executed
- I/O function $\tau$: 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, $\forall i \in V$:

$$\left( \sum_{(k,p) \in E} (w_k - \tau(k)) \right) + \max \left( w_i, \sum_{(j,i) \in E} w_j \right) \leq M$$
Description of a solution

Traversal

▶ Schedule \( \sigma \): \( \sigma(i) = t \) if task \( i \) is the \( t \)-th executed

▶ I/O function \( \tau \): 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, \( \forall i \in V \):

\[
\left( \sum_{(k,p) \in E, \sigma(k) < \sigma(i) < \sigma(p)} (w_k - \tau(k)) \right) + \max \left( w_i, \sum_{(j,i) \in E} w_j \right) \leq M
\]
Description of a solution

Traversal

- **Schedule** $\sigma$: $\sigma(i) = t$ if task $i$ is the $t$-th executed
- **I/O function** $\tau$: 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, $\forall i \in V$:

$$\left( \sum_{(k,p) \in E} (w_k - \tau(k)) \right) + \max \left( w_i, \sum_{(j,i) \in E} w_j \right) \leq M$$
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 softwares (e.g., MUMPS, QR-MUMPS)
Preliminary results

Let \((\sigma, \tau)\) be an optimal traversal for \textsc{MinIO} of a given instance

**Lemma (Schedule is enough).**

Given \(\sigma\): the \textit{Furthest In the Future} I/O policy minimizes I/Os.

**Lemma (I/O function is enough).**

Given \(\tau\): a valid traversal \((\sigma', \tau)\) can be computed in polynomial time.

**Proof.**

Expand each node following:

\[
\begin{align*}
&w_i \\
&\implies \quad w_i \\
&\quad \quad w_i \\
&\quad \quad w_i - \tau(i) \\
&\quad \quad w_i \\
\end{align*}
\]

Then minimize the memory peak.
Preliminary results

Let \((\sigma, \tau)\) be an optimal traversal for \textsc{MinIO} of a given instance

**Lemma (Schedule is enough).**

Given \(\sigma\): the Furthest In the Future I/O policy minimizes I/Os.

**Lemma (I/O function is enough).**

Given \(\tau\): a valid traversal \((\sigma', \tau)\) can be computed in polynomial time.

**Proof.**

Expand each node following:

\[
\begin{align*}
  w_i & \quad \Rightarrow \quad w_i - \tau(i) \\
  w_i & \rightarrow \quad w_i
\end{align*}
\]

Then minimize the memory peak.
Postorder algorithms [Liu 1986, Agullo et al. 2010]

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

![Diagram of a tree with node $i$ and children $T_k$, $T_j$, ...]
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 $T_i$: 

\[
A_i = \min(S_i, M)
\]

For a given order $\sigma$, the volume of I/O is given by:

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

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

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

- 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 $T_i$:

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

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

$$V_i = \max \left( 0, \max_{j \in \text{Chil}(i)} \left( A_j + \sum_{k \in \text{Chil}(i), \sigma(k) < \sigma(j)} w_k \right) - M \right) + \sum_{j \in \text{Chil}(i)} V_j$$
Best Postorder for Minimizing I/Os

For a given order $\sigma$, the volume of I/O is given by:

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

**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 decreasing $x_i - y_i$.

**Corollary**

*The postorder traversal that minimizes I/Os sorts the subtrees by decreasing $A_j - w_j$.***
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 why optimal traversal does not
- Even in 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

Minimize Memory for Trees

Minimize Memory for Series-Parallel Graphs

Minimize I/Os for Trees under Bounded Memory

Complexity and Space-Time Tradeoffs for Parallel Tree Processing

Parallel Processing of DAGs with Limited Memory
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 ($P|\text{trees}|C_{\text{max}}$)
- Polynomial when unit-weight tasks ($P|p_i = 1, \text{trees}|C_{\text{max}}$)
- 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 $\sum a_i = mB$,
- 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 $\alpha$-approximation for makespan minimization and a $\beta$-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 $\alpha$-approximation for makespan minimization and a $\beta$-approximation for memory peak minimization when scheduling tree-shaped task graphs.

**Lemma**

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

\[
M \times C_{\text{max}} \geq 2(n - 1)
\]
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 $\alpha$-approximation for makespan minimization and a $\beta$-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 $\alpha$-approximation for makespan minimization and a $\beta$-approximation for memory peak minimization when scheduling tree-shaped task graphs.

**Lemma**

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

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

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 $C_{\text{max}}$,

$$M \times C_{\text{max}} \geq \sum_i \text{mem\_needed\_for\_task}_i \times p_i$$
Space-Time Tradeoff – Proof

- With $m^2$ processors: $C_{\text{max}}^* = 3$
- With 1 processor, sequentialize the $a_i$ subtrees: $M^* = 2m$
- By contradiction, approximating both objectives: $C_{\text{max}} \leq 3\alpha$ and $M \leq 2m\beta$
- But $M \times C_{\text{max}} \geq 2(n - 1) = 2m^2 + 2m$
- $2m^2 + 2m \leq 6m\alpha\beta$
- Contradiction for a sufficiently large value of $m$
Complexity – Summary

For task trees:

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

Minimize Memory for Trees

Minimize Memory for Series-Parallel Graphs

Minimize I/Os for Trees under Bounded Memory

Complexity and Space-Time Tradeoffs for Parallel Tree Processing

Parallel Processing of DAGs with Limited Memory
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

Minimize Memory for Trees

Minimize Memory for Series-Parallel Graphs

Minimize I/Os for Trees under Bounded Memory

Complexity and Space-Time Tradeoffs for Parallel Tree Processing

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
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
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

Minimize Memory for Trees

Minimize Memory for Series-Parallel Graphs

Minimize I/Os for Trees under Bounded Memory

Complexity and Space-Time Tradeoffs for Parallel Tree Processing

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
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 LP

Consider one classical LP formulation for finding a minimum cut:

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

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

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

\[p_s = 1, \quad p_t = 0\]
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
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)
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

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 $G^{\text{diff}}$ with edge weights $F_{i,j} - m_{i,j}$
3. Compute a maximum flow $\text{maxdiff}$ in $G^{\text{diff}}$

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 $G^{\text{diff}}$ with edge weights $F_{i,j} - m_{i,j}$
3. Compute a maximum flow $\text{maxdiff}$ in $G^{\text{diff}}$
4. $F - \text{maxdiff}$ 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

⇔

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
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 $i \rightarrow j$:
  \[
  M(G, k) = \begin{cases} 
  m_{i,j} & \text{if edge is black or } k \geq 0 \\
  -\infty & \text{otherwise}
  \end{cases}
  \]
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 $k = 1, \ldots, p$

- Single edge $i \rightarrow j$:
  
  $$M(G, k) = \begin{cases} 
  m_{i,j} & \text{if edge is black or } k \geq 0 \\
  -\infty & \text{otherwise}
  \end{cases}$$

- 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 $i \rightarrow j$:
  $M(G, k) = \begin{cases} 
  m_{i,j} & \text{if edge is black or } k \geq 0 \\
  -\infty & \text{otherwise}
\end{cases}$

▶ 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: $O(|E|p)$. 
Special Case of SP Graphs – Approximation

Definition (Dual Approximation).
For a given guess $\lambda$, algo. that answers “1” if $M(G, p) \leq \lambda$ and “0” if $M(G, p) > \lambda/2$.

Idea:
- Consider only edges whose weight is $> \lambda/2p$
- Apply SP algorithms for without bound on $p$
- Return 1 iff $M(G, \infty) \geq \lambda/2$

Using binary search: 2-approximation algorithm
Summary 2

Predict the maximal memory of any dynamic scheduling
⇔
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 \(O(|E|p)\)
▶ Max. Top. cut with \(p\) procs: 2-approximation in \(O(|E|)\)
Outline

Minimize Memory for Trees

Minimize Memory for Series-Parallel Graphs

Minimize I/Os for Trees under Bounded Memory

Complexity and Space-Time Tradeoffs for Parallel Tree Processing

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
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 dependencies to reduce maximum memory*
- Minimize the obtained *critical path*

![Graph with nodes A, B, C, D, E, F, and edges labeled 1, 2, 3, 4, 5, with $M = 10$.]
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 dependencies* to reduce maximum memory
- Minimize the obtained *critical path*

![Graph Diagram]

$M = 10$
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, $\text{MaxMem}(G') \leq M$ and $\text{CritPath}(G')$ is minimized.

Theorem.
PartialSerialization is NP-hard in the strong sense.

NB: stays NP-hard if we are given a sequential schedule $\sigma$ 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

Minimize Memory for Trees

Minimize Memory for Series-Parallel Graphs

Minimize I/Os for Trees under Bounded Memory

Complexity and Space-Time Tradeoffs for Parallel Tree Processing

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
**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)

- **x**: memory ($0 = \text{DFS}, 1 = \text{MaxTopCut}$)
  - median ratio $\text{MaxTopCut} / \text{DFS memory} \approx 1.3$

- **y**: CP / original CP → lower is better

- **MinLevels** performs best
Simulations: sparse random graphs (25, 50, 100 nodes)

Different heuristics were tested:
- MinLevels
- RespectOrder
- MaxMinSize
- MaxSize

- **x:** memory (0 = DFS, 1 = MaxTopCut)
  median ratio MaxTopCut / DFS memory ≈ 2

- **y:** CP / original CP → lower is better

MinLevels performs best, but might fail
Simulations – Pegasus workflows (LIGO 100 nodes)

- Median ratio $\text{MaxTopCut} / \text{DFS} \approx 20$
- MinLevels performs best, RespectOrder always succeeds
Simulations – Pegasus workflows (LIGO 100 nodes)

- Median ratio MaxTopCut / DFS ≈ 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 ⇔ 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
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 ⇔ 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 \((\alpha, \beta)\)-approx.
▶ Dynamic scheduling with memory bound:
   ▶ Compute the worst memory: polynomial (linear for SP-graphs)
   ▶ Limit memory: NP-complete, heuristic solutions