Part 3: Memory-Aware DAG Scheduling

CR05: Data Aware Algorithms

October 12 & 15, 2020
Summary of the course

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

- Part 2: External Memory and Cache Oblivious 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
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
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)
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)
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 > 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: 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
Outline

Minimize Memory for Trees

Minimize Memory for Series-Parallel Graphs

Minimize I/Os for Trees under Bounded Memory
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:

$$\max\{P_1, f_1 + P_2, f_1 + f_2 + P_3, \ldots, f_1 + \ldots + f_n + P_n\}$$
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:

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

$$\max\{P_1, \ f_1 + P_2, \ f_1 + f_2 + P_3, \ \ldots, \ \sum_{i<n} f_i + P_n, \ \sum_{i<n} f_i + n_r + f_r\}$$
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 $P_i - f_i$
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>$\text{mem}_\text{before} + P_j$</td>
<td>$\text{mem}_\text{before} + P_k$</td>
</tr>
<tr>
<td>during second subtree</td>
<td>$\text{mem}_\text{before} + f_j + P_k$</td>
<td>$\text{mem}_\text{before} + f_k + P_j$</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.

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

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:
$$M_{\text{min}} = M + \epsilon + (b - 1)M/b + \epsilon$$

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

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.

![Diagram of a tree with labels $M/b$, $\epsilon$, and $M$]

- Minimum post-order peak memory:
  \[ M_{\text{min}} = M + \epsilon + 2(b - 1)\frac{M}{b} \]

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

<table>
<thead>
<tr>
<th></th>
<th>actual assembly trees</th>
<th>random trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non optimal traversals</td>
<td>4.2%</td>
<td>61%</td>
</tr>
<tr>
<td>Maximum increase compared to optimal</td>
<td>18%</td>
<td>22%</td>
</tr>
<tr>
<td>Average increased compared to optimal</td>
<td><strong>1%</strong></td>
<td>12%</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 $H_2$
  - . . .
  - 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
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

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

Theorem
There exists a schedule with minimal memory which synchronises at $e_{\text{min}}^1, \ldots, e_{\text{min}}^B$.

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: $e_{1}^{\text{min}}, \ldots, e_{B}^{\text{min}}$

**Theorem**

There exists a schedule with minimal memory which synchronises at $e_{1}^{\text{min}}, \ldots, e_{B}^{\text{min}}$. 
First Step: Parallel-Chain Graphs

Select edges with minimal weight on each branch: $e^\text{min}_1, \ldots, e^\text{min}_B$

**Theorem**

There exists a schedule with minimal memory which synchronises at $e^\text{min}_1, \ldots, e^\text{min}_B$.

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
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>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>
<tr>
<td>$\Rightarrow$ Memory needed when processing $u$ not larger in $\sigma_2$</td>
<td></td>
</tr>
</tbody>
</table>

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

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

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

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

- Same analysis if $u \in T$
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>$e_j^\text{min}$ if $v \in L$ otherwise</th>
</tr>
</thead>
</table>

⇒ Memory needed when processing $u$ not larger in $\sigma_2$
- Same analysis if $u \in T$
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?
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?
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?
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
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 $S/2$?

\[ n_i = 0 \text{ for all tasks} \]
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: 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, $\max(\text{input, output})$ 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, \( \max(\text{input}, \text{output}) \) 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)

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

![Diagram of a tree with nodes and I/O calculations]

Memory: 3 / 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)
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)