Data Aware Algorithms – Part 3
Memory-Aware DAG scheduling

Loris Marchal

October 18, 2023

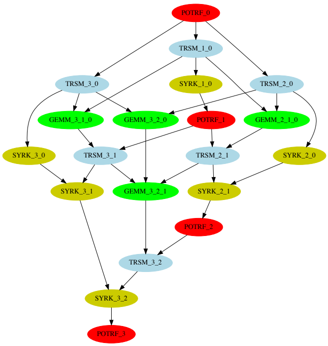
Memory-Aware DAG Scheduling

Memory-Aware DAG Scheduling

Taming HPC platforms with runtime systems

- Write your application as function calls (*tasks*),
- Specify data input/output (*dependencies*)
- Provide function codes for specific cores/GPUs
- Let the system do the scheduling at runtime!

```
Cholesky_decomposition(A):
for(k=0; k<N; k++)
   A[k][k]=POTRF(A[k][k])
   for(m=k+1; m<N; m++)
      A[m][k]=TRSM(A[k][k], A[m][k])
   for(n=k+1; n<N; n++)
      A[n][n]=SYRK(A[n][k], A[n][n])
      for(m=n+1; m<N; m++)
         A[m][n]+=GEMM(A[m][k],A[n][k])
```

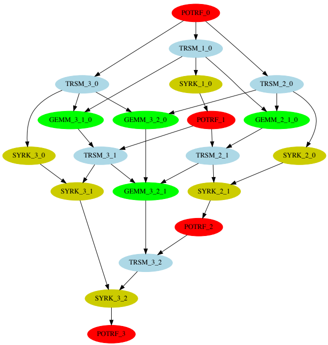


Graph of tasks: Directed Acyclic Graph (DAG)

- Tasks linked with data dependency
- Wide literature on DAG scheduling
- What about memory and data movements (I/Os) ?

Taming HPC platforms with runtime systems

- Write your application as function calls (*tasks*),
- Specify data input/output (*dependencies*)
- Provide function codes for specific cores/GPUs
- Let the system do the scheduling at runtime!



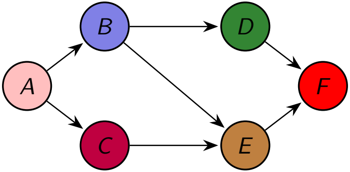

```
Cholesky_decomposition(A):
for(k=0; k<N; k++)
   A[k][k]=POTRF(A[k][k])
   for(m=k+1; m<N; m++)
      A[m][k]=TRSM(A[k][k], A[m][k])
   for(n=k+1; n<N; n++)
      A[n][n]=SYRK(A[n][k], A[n][n])
      for(m=n+1; m<N; m++)
         A[m][n]+=GEMM(A[m][k],A[n][k])
```

Graph of tasks: Directed Acyclic Graph (DAG)

- Tasks linked with data dependency
- Wide literature on DAG scheduling
- What about memory and data movements (I/Os) ?

Task graph scheduling and memory

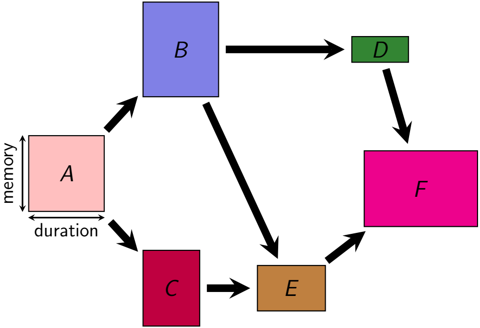
- Consider a simple task graph
- Tasks have durations and memory demands



- Peak memory: maximum memory usage
- Trade-off between peak memory and makespan

Task graph scheduling and memory

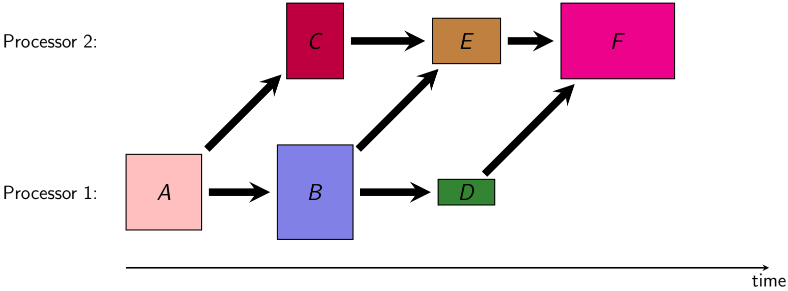
- Consider a simple task graph
- Tasks have durations and memory demands



- Peak memory: maximum memory usage
- Trade-off between peak memory and makespan

Task graph scheduling and memory

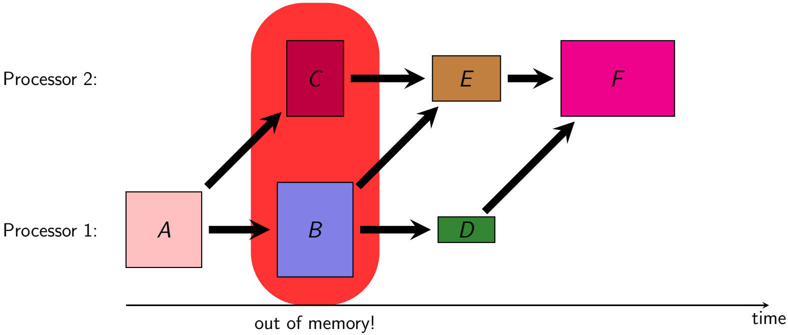
- Consider a simple task graph
- Tasks have durations and memory demands



- Peak memory: maximum memory usage
- Trade-off between peak memory and makespan

Task graph scheduling and memory

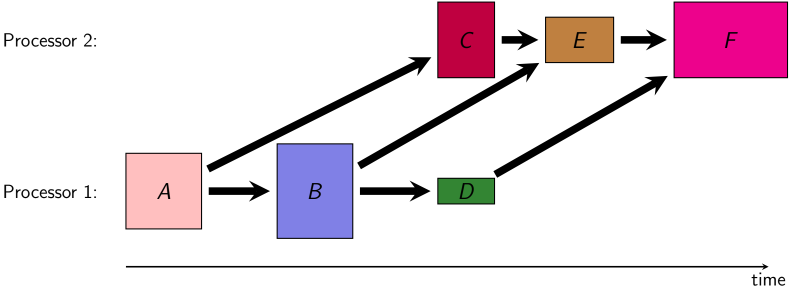
- Consider a simple task graph
- Tasks have durations and memory demands



- Peak memory: maximum memory usage
- Trade-off between peak memory and makespan

Task graph scheduling and memory

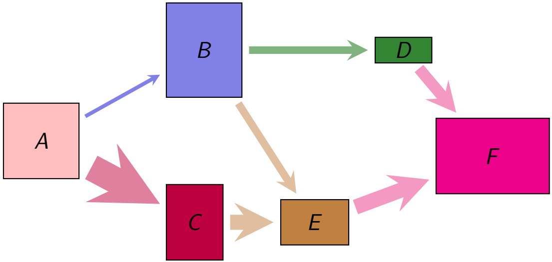
- Consider a simple task graph
- Tasks have durations and memory demands



- Peak memory: maximum memory usage
- Trade-off between peak memory and makespan

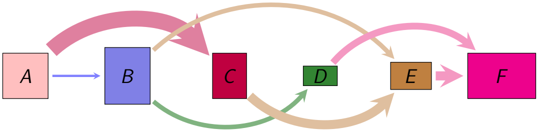
Going back to sequential processing

- Temporary data require memory
- Scheduling influences the peak memory



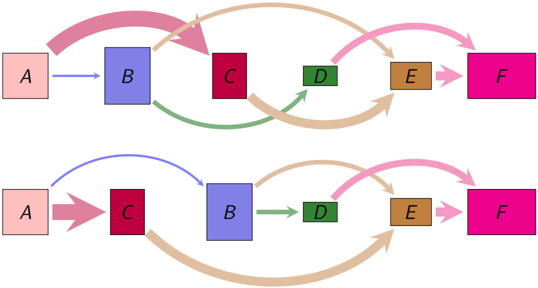
Going back to sequential processing

- Temporary data require memory
- Scheduling influences the peak memory



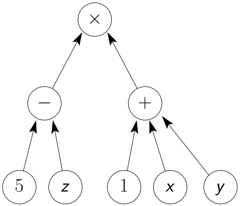
Going back to sequential processing

- Temporary data require memory
- Scheduling influences the peak memory



Pebble game for register allocation (reminder)

- From the 70s: limit usage of scarce registers
- Model expressions as Directed Acyclic Graphs



$(5 - z) \times (1 + x + y)$

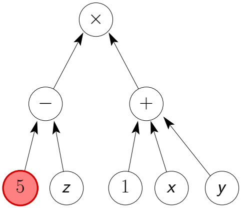
Rules of the game:

- A pebble may be placed on a source node at any time (LOAD)
- If all predecessors of v are pebbled, a pebble may be placed on v. (COMPUTE)
- A pebble may be removed from a vertex at any time. (EVICT)
- Goal: computation all vertices, use minimal number of pebbles

Results: Optimal algorithms for trees — NP-hard on general DAGs

Pebble game for register allocation (reminder)

- From the 70s: limit usage of scarce registers
- Model expressions as Directed Acyclic Graphs



$(5 - z) \times (1 + x + y)$

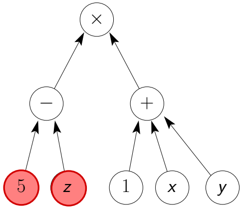
Rules of the game:

- A pebble may be placed on a source node at any time (LOAD)
- If all predecessors of v are pebbled, a pebble may be placed on v. (COMPUTE)
- A pebble may be removed from a vertex at any time. (EVICT)
- Goal: computation all vertices, use minimal number of pebbles

Results: Optimal algorithms for trees — NP-hard on general DAGs

Pebble game for register allocation (reminder)

- From the 70s: limit usage of scarce registers
- Model expressions as Directed Acyclic Graphs



$(5 - z) \times (1 + x + y)$

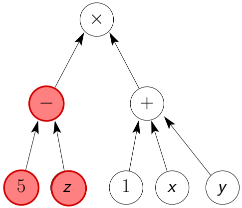
Rules of the game:

- A pebble may be placed on a source node at any time (LOAD)
- If all predecessors of v are pebbled, a pebble may be placed on v. (COMPUTE)
- A pebble may be removed from a vertex at any time. (EVICT)
- Goal: computation all vertices, use minimal number of pebbles

Results: Optimal algorithms for trees — NP-hard on general DAGs

Pebble game for register allocation (reminder)

- From the 70s: limit usage of scarce registers
- Model expressions as Directed Acyclic Graphs



$(5 - z) \times (1 + x + y)$

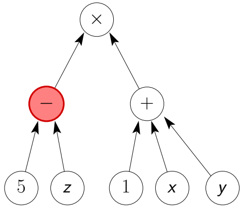
Rules of the game:

- A pebble may be placed on a source node at any time (LOAD)
- If all predecessors of v are pebbled, a pebble may be placed on v. (COMPUTE)
- A pebble may be removed from a vertex at any time. (EVICT)
- Goal: computation all vertices, use minimal number of pebbles

Results: Optimal algorithms for trees — NP-hard on general DAGs

Pebble game for register allocation (reminder)

- From the 70s: limit usage of scarce registers
- Model expressions as Directed Acyclic Graphs



$(5 - z) \times (1 + x + y)$

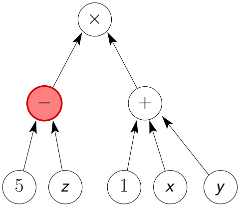
Rules of the game:

- A pebble may be placed on a source node at any time (LOAD)
- If all predecessors of v are pebbled, a pebble may be placed on v. (COMPUTE)
- A pebble may be removed from a vertex at any time. (EVICT)
- Goal: computation all vertices, use minimal number of pebbles

Results: Optimal algorithms for trees — NP-hard on general DAGs

Pebble game for register allocation (reminder)

- From the 70s: limit usage of scarce registers
- Model expressions as Directed Acyclic Graphs



$(5 - z) \times (1 + x + y)$

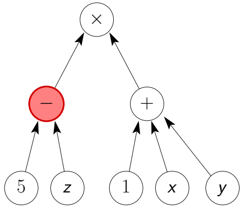
Rules of the game:

- A pebble may be placed on a source node at any time (LOAD)
- If all predecessors of v are pebbled, a pebble may be placed on v. (COMPUTE)
- A pebble may be removed from a vertex at any time. (EVICT)
- Goal: computation all vertices, use minimal number of pebbles

Results: Optimal algorithms for trees — NP-hard on general DAGs

Pebble game for register allocation (reminder)

- From the 70s: limit usage of scarce registers
- Model expressions as Directed Acyclic Graphs



$(5 - z) \times (1 + x + y)$

Rules of the game:

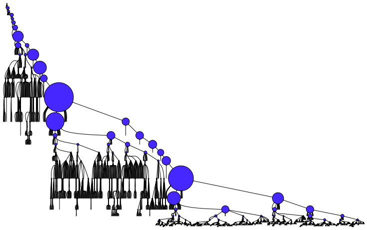
- A pebble may be placed on a source node at any time (LOAD)
- If all predecessors of v are pebbled, a pebble may be placed on v. (COMPUTE)
- A pebble may be removed from a vertex at any time. (EVICT)
- Goal: computation all vertices, use minimal number of pebbles

Results: Optimal algorithms for trees — NP-hard on general DAGs

Memory-Aware DAG Scheduling

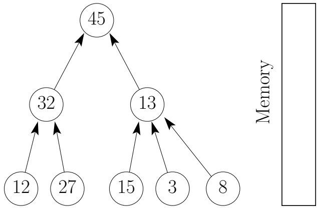
Generalized Pebble Game

- Sparse matrix factorization
- Task graph: tree (with dependencies towards the root)
- Large temporary data



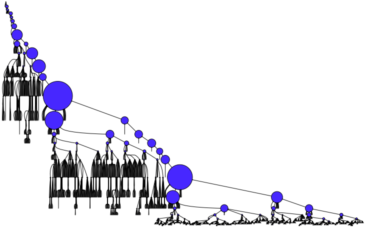
Generalized pebble game [Liu 1986]:

- Node have heterogeneous weights (memory demand)
- Compute task = replace inputs by outputs in memory
- output memory $\neq \sum$ input memory



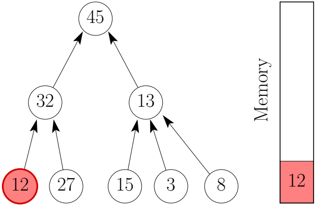
Generalized Pebble Game

- Sparse matrix factorization
- Task graph: tree (with dependencies towards the root)
- Large temporary data



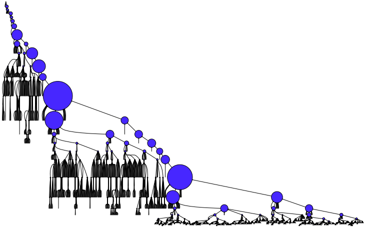
Generalized pebble game [Liu 1986]:

- Node have heterogeneous weights (memory demand)
- Compute task = replace inputs by outputs in memory
- output memory $\neq \sum$ input memory



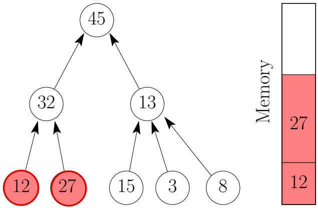
Generalized Pebble Game

- Sparse matrix factorization
- Task graph: tree (with dependencies towards the root)
- Large temporary data



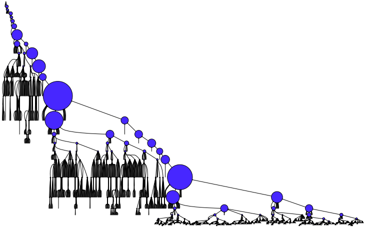
Generalized pebble game [Liu 1986]:

- Node have heterogeneous weights (memory demand)
- Compute task = replace inputs by outputs in memory
- output memory $\neq \sum$ input memory



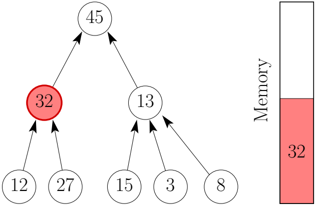
Generalized Pebble Game

- Sparse matrix factorization
- Task graph: tree (with dependencies towards the root)
- Large temporary data



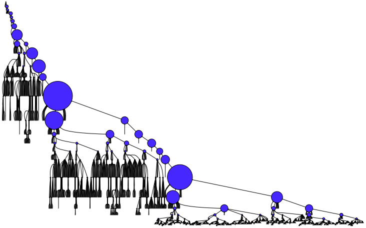
Generalized pebble game [Liu 1986]:

- Node have heterogeneous weights (memory demand)
- Compute task = replace inputs by outputs in memory
- output memory $\neq \sum$ input memory



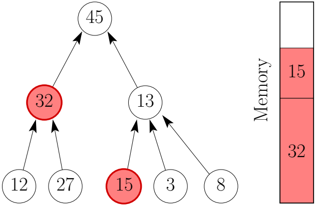
Generalized Pebble Game

- Sparse matrix factorization
- Task graph: tree (with dependencies towards the root)
- Large temporary data



Generalized pebble game [Liu 1986]:

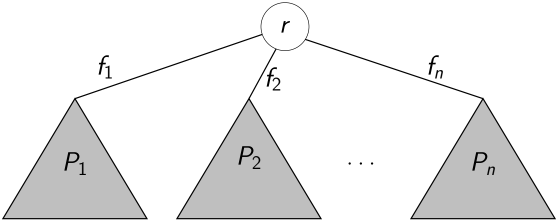
- Node have heterogeneous weights (memory demand)
- Compute task = replace inputs by outputs in memory
- output memory $\neq \sum$ input memory



Outline

Liu's Best Post-Order Traversal for Trees

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



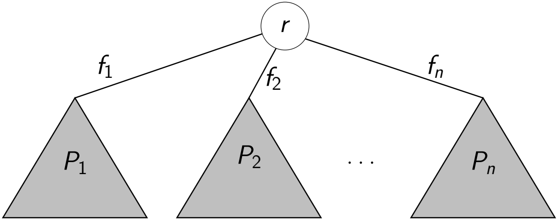
- For each subtree T_i: peak memory P_i, residual memory f_i
- For a given processing order $1, \ldots, n$, the peak memory is:

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

- Optimal order:

Liu's Best Post-Order Traversal for Trees

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



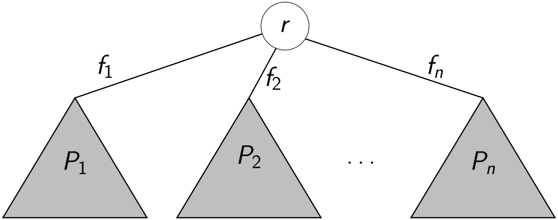
- For each subtree T_i: peak memory P_i, residual memory f_i
- For a given processing order $1, \ldots, n$, the peak memory is:

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

- Optimal order:

Liu's Best Post-Order Traversal for Trees

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



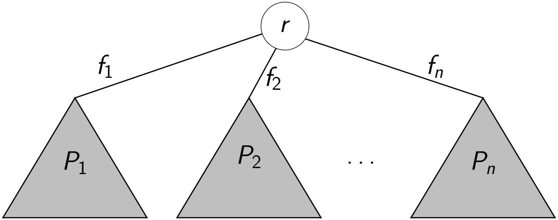
- For each subtree T_i: peak memory P_i, residual memory f_i
- For a given processing order $1, \ldots, n$, the peak memory is:

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

- Optimal order:

Liu's Best Post-Order Traversal for Trees

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



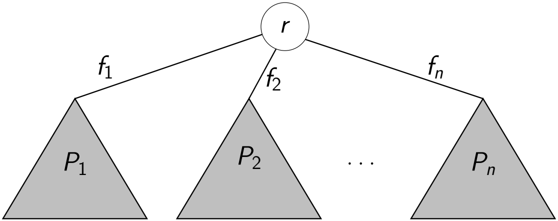
- For each subtree T_i: peak memory P_i, residual memory f_i
- For a given processing order $1, \ldots, n$, the peak memory is:

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

- Optimal order:

Liu's Best Post-Order Traversal for Trees

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



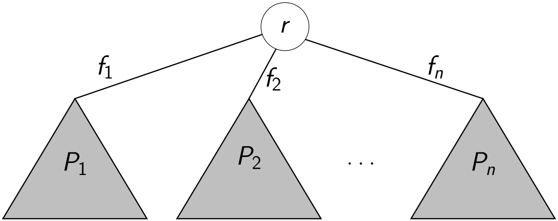
- For each subtree T_i: peak memory P_i, residual memory f_i
- For a given processing order $1, \dots, n$, the peak memory is:

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

- Optimal order:

Liu's Best Post-Order Traversal for Trees

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



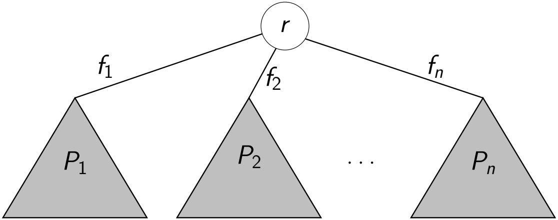
- For each subtree T_i: peak memory P_i, residual memory f_i
- For a given processing order $1, \ldots, n$, the peak memory is:

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

- Optimal order: ?

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 obtain 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.

Proof for best post-order

Theorem (Best Post-Order).

The best post-order traversal is obtain 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
  - $P_k - f_k \geq P_j - f_j$

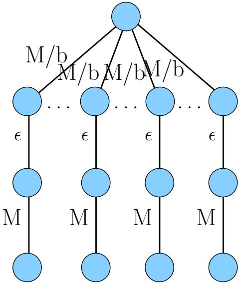
	peak when j, then k	peak when k, then j
during first subtree	$mem\_before + P_j$	$mem\_before + P_k$
during second subtree	$mem\_before + f_j + P_k$	$mem\_before + f_k + P_j$

- $f_k + P_j \leq f_j + P_k$
- 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.



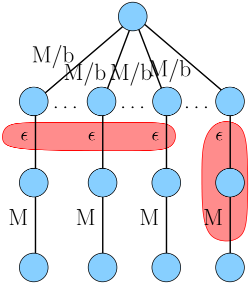
- Minimum peak memory: $M_{\text{min}} = M + \; + (b-1)\epsilon$
- Minimum post-order peak memory: $M_{\text{min}} = M \; + \; (b-1)M/b$

	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%

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.



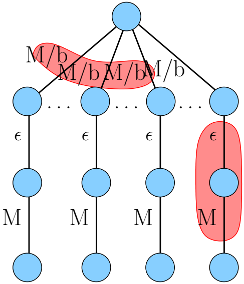
- Minimum peak memory: $M_{\text{min}} = M + \; + (b-1)\epsilon$
- Minimum post-order peak memory: $M_{\text{min}} = M \; + \; (b-1)M/b$

	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%

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.



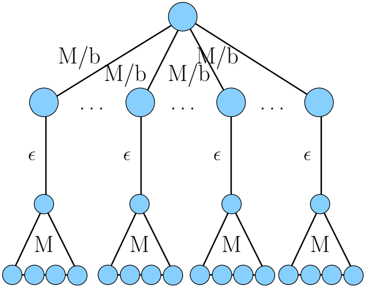
- Minimum peak memory: $M_{\min} = M + \ + (b-1)\epsilon$
- Minimum post-order peak memory: $M_{\min} = M \ + \ (b-1)M/b$

	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%

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.



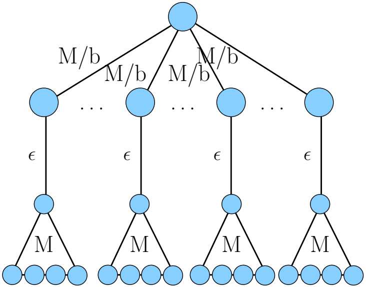
- Minimum peak memory: $M_{\min} = M + \; + 2(b-1)\epsilon$
- Minimum post-order peak memory: $M_{\min} = M \; + 2(b-1)M/b$

	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%

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

	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%

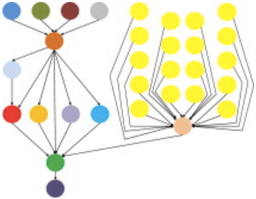
Liu's optimal traversal – sketch

- 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_is 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: $(H_1, V_1, H_2, V_2, \dots, H_r, V_r) \prec (H'_1, V'_1, H'_2, V'_2, \dots, H'_{r'}, V'_{r'})$ 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

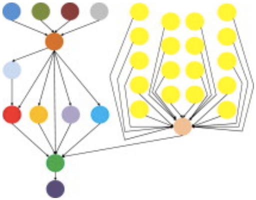
Series-Parallel Graphs: Motivation

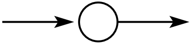
- Not all scientific workflows are trees
- But most workflows exhibit some regularity
- Large class of workflows: Series-Parallel graphs



Series-Parallel Graphs: Motivation

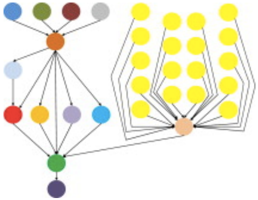
- Not all scientific workflows are trees
- But most workflows exhibit some regularity
- Large class of workflows: Series-Parallel graphs

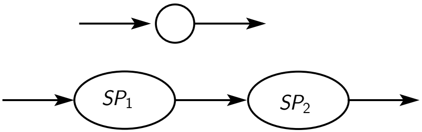




Series-Parallel Graphs: Motivation

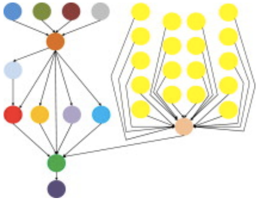
- Not all scientific workflows are trees
- But most workflows exhibit some regularity
- Large class of workflows: Series-Parallel graphs

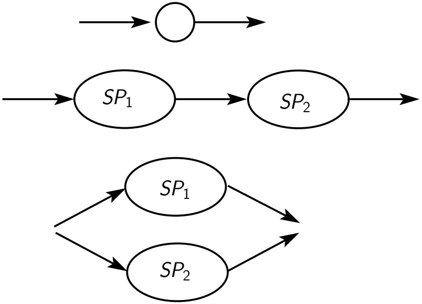




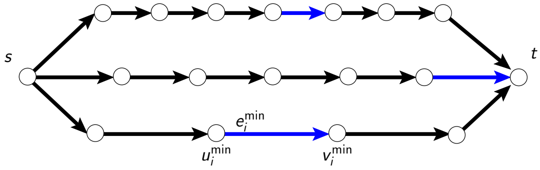
Series-Parallel Graphs: Motivation

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





First Step: Parallel-Chain Graphs



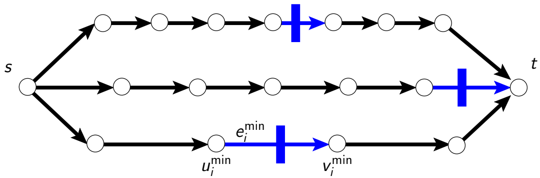
Edge using the minimum amount of memory, on each chain: $e_1, \ldots, e_n$.

Lemma

There exists an schedule with minimal memory stopping on edges $e_1, \ldots, e_n$.

1. Split the graph on minimal cut $e_1, \ldots, e_n$
2. Apply Liu's algorithm on resulting trees

First Step: Parallel-Chain Graphs



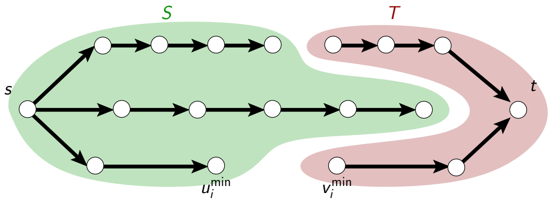
Edge using the minimum amount of memory, on each chain: $e_1, \ldots, e_n$.

Lemma

There exists an schedule with minimal memory stopping on edges $e_1, \ldots, e_n$.

1. Split the graph on minimal cut $e_1, \ldots, e_n$
2. Apply Liu's algorithm on resulting trees

First Step: Parallel-Chain Graphs



Edge using the minimum amount of memory, on each chain: $e_1, \ldots, e_n$.

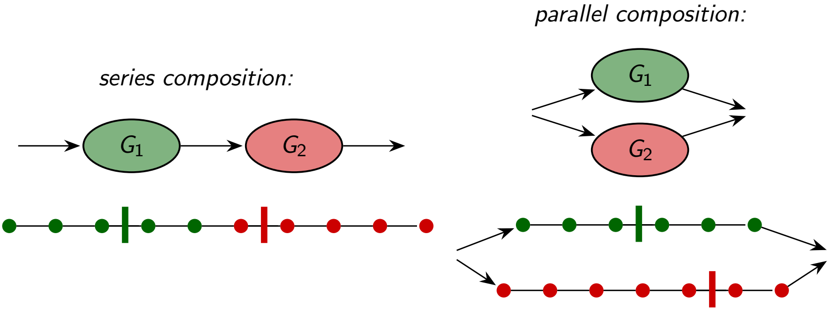
Lemma

There exists an schedule with minimal memory stopping on edges $e_1, \ldots, e_n$.

1. Split the graph on minimal cut $e_1, \ldots, e_n$
2. Apply Liu's algorithm on resulting trees

Algorithm for General Series-Parallel Graphs

- Follow recursive definition of the graph
- Simultaneously compute minimal cut and optimal schedule
- Replace subgraph by linear chain corresponding to the schedule



Heuristic method for general graphs

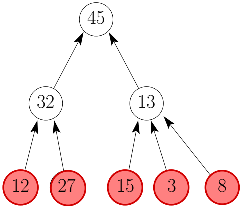
- Transform graph into SP-graph by adding synchronisation points
- Compute optimal schedule on obtained SP-graph

Memory-Aware DAG Scheduling

Memory-Aware DAG Scheduling

Model for Parallel Tree Processing

- p identical 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

Theorem

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

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.

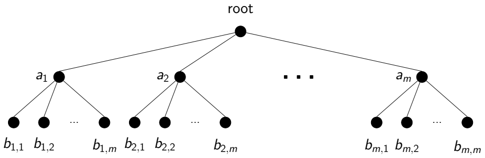
Lemma

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.

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

Complexity – Summary

For task trees:

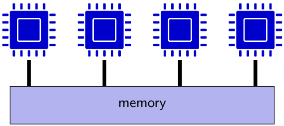
- Optimizing both makespan memory is NP-Complete
  $\Rightarrow$ Same for minimizing makespan under memory budget

- No scheduling algorithm can be a constant factor approximation on both memory and makespan

Memory-Aware DAG Scheduling

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

Part 3: Memory-Aware DAG Scheduling

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

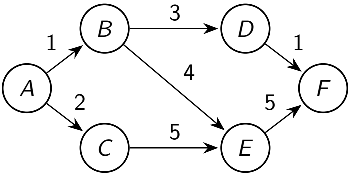
Task graphs with:

- *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

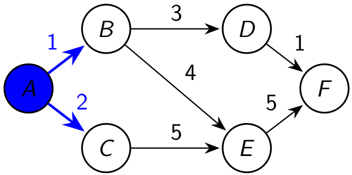
Task graphs with:

- *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

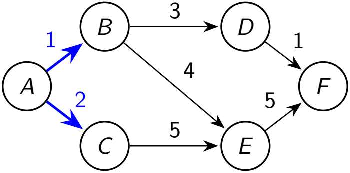
Task graphs with:

- *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

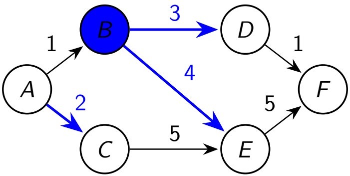
Task graphs with:

- *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

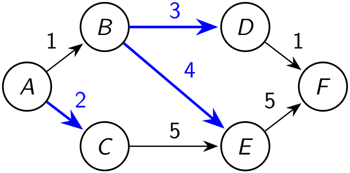
Task graphs with:

- *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



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

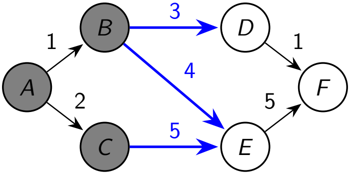
At the end of a task: outputs stay in memory

Part 3: Memory-Aware DAG Scheduling

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 this 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:

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

- 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 – using LP

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

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

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

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

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

- 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 – using LP

- 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 – using LP

- 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

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 → maximum topological cut

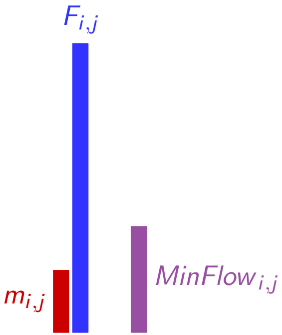
Complexity: same as maximum flow, e.g., $O(|V|^2|E|)$

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



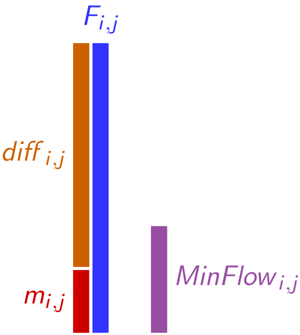
Complexity: same as maximum flow, e.g., $O(|V|^2|E|)$

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



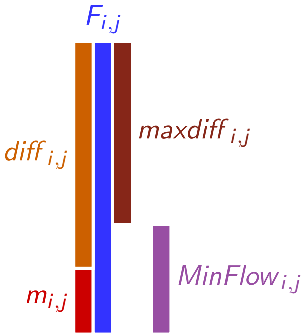
Complexity: same as maximum flow, e.g., $O(|V|^2|E|)$

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 → maximum topological cut



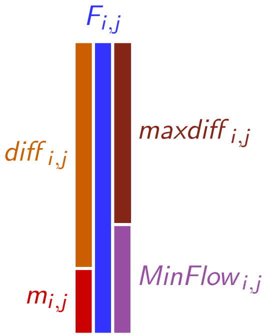
Complexity: same as maximum flow, e.g., $O(|V|^2|E|)$

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

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

Part 3: Memory-Aware DAG Scheduling

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)

Possible solution:

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

$M = 10$

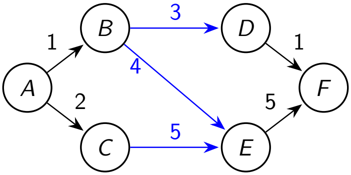
Coping with limiting memory

Problem:

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

Possible solution:

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



$M = 10$

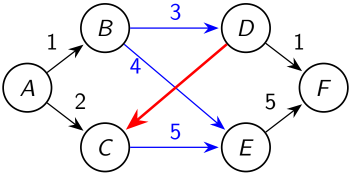
Coping with limiting memory

Problem:

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

Possible solution:

- Add edges to guarantee that any parallel execution stays below M
  fictitious dependencies 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.

Part 3: Memory-Aware DAG Scheduling

Task Graph Scheduling vs. Limited Memory

Minimizing Memory for Task Graphs
- Minimizing Memory for Task Trees
- Minimizing Memory for SP-Graphs

Shared Memory of Parallel Processing
- Complexity and Space-Time Tradeoffs for Trees
- Processing DAGs with Limited Memory
  - Model and maximum parallel memory
  - Maximum parallel memory/maximal topological cut
  - Efficient scheduling with bounded memory
  - Heuristics and simulations

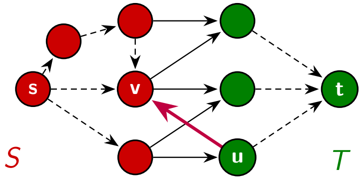
Reducing I/Os for Task Graphs

Heuristic solutions for PartialSerialization

Framework:

(inspired by [Sbîrlea et al. 2014])

1. Compute a max. top. cut (S, T)
2. If weight $\leq M$: exit with success
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

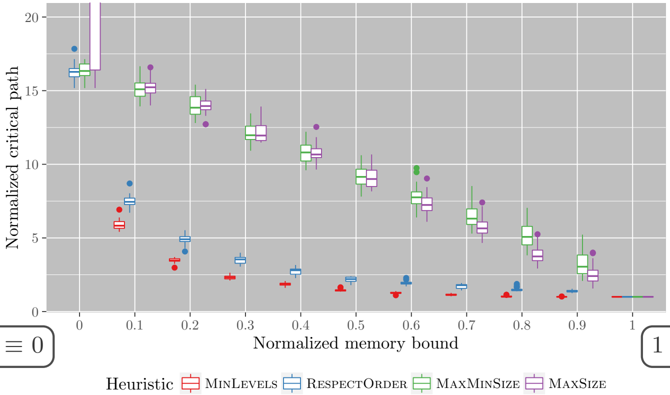
RespectOrder follows a precomputed memory-efficient schedule, always succeeds

MaxSize targets nodes dealing with large data

MaxMinSize variant of MaxSize

Simulations – Pegasus workflows (LIGO 100 nodes)

lower is better



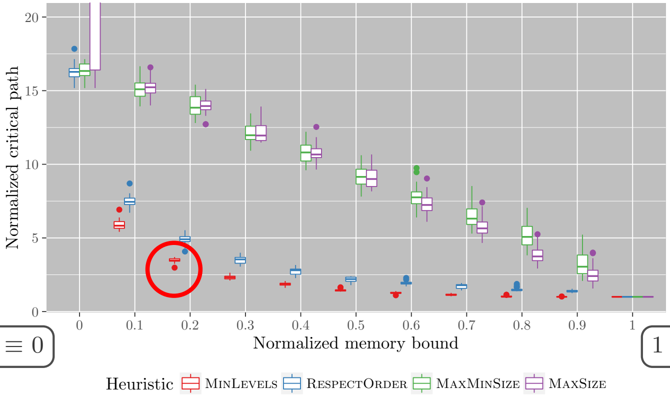
DFS memory $\equiv 0$

$1 \equiv$ MaxTopCut

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

Simulations – Pegasus workflows (LIGO 100 nodes)

lower is better



DFS memory ≡ 0

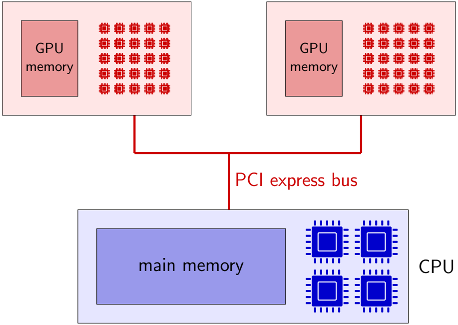
1 ≡ MaxTopCut

- *Median ratio MaxTopCut / DFS ≈ 20*
- MinLevels performs best, RespectOrder always succeeds
- Memory divided by 5 – critical path multiplied by 3

Memory-Aware DAG Scheduling

Platform model

- Memory too scarce to accomodate all (input) data
- Data initially on a large, slow storage



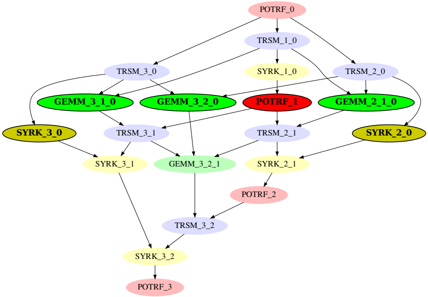
GPUs provide large speed-ups for reduced energy, but:

- *limited memory* within GPU
- connected through bus with *limited bandwidth*

Dynamic view of a task graph

At any time step: consider only available tasks

- Independant tasks
- Sharing some input data

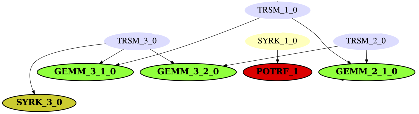


→ bipartite graph between data and tasks

Dynamic view of a task graph

At any time step: consider only available tasks

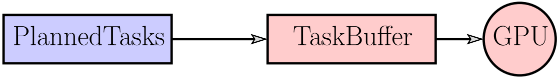
- Independant tasks
- Sharing some input data



→ bipartite graph between data and tasks

Dynamic scheduling of task graphs

- Tasks appear over time (task graph discovered at runtime)
- Two questions:
  - Partition tasks among GPUs
  - Order task on each GPUs
- When task input data not on GPU: load it from main memory (possibly before the execution: prefetching)
- When memory is full: evict data Eviction policy

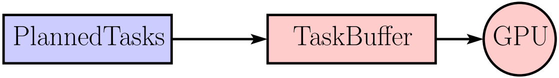


Two sorted sets of tasks per GPU (FIFO):

1. TaskBuffer: tasks definitively allocated on a GPU (data possibly being prefetched)
2. PlannedTasks: good candidate tasks for a GPU

Dynamic scheduling of task graphs

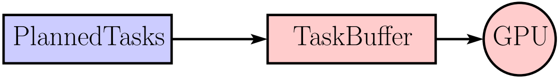
- Tasks appear over time (task graph discovered at runtime)
- Two questions:
  - Partition tasks among GPUs
  - Order task on each GPUs
- When task input data not on GPU: load it from main memory (possibly before the execution: prefetching)
- When memory is full: evict data Eviction policy



Two sorted sets of tasks per GPU (FIFO):

1. TaskBuffer: tasks definitively allocated on a GPU (data possibly being prefetched)
2. PlannedTasks: good candidate tasks for a GPU

DARTS (Data-Aware Reactive Task Scheduling)

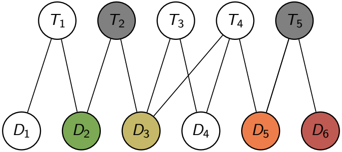


How to fill PlannedTasks_k when needed:

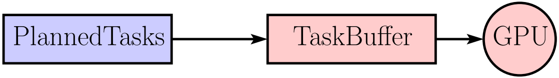
1. Concentrate on data, choose "best" data to load
2. Look for tasks that GPU_k can do with D + its current data
3. Choose data with largest ratio:

$$\frac{\text{computation time of tasks enabled with } D}{\text{time needed to transfer data } D}$$

4. Break ties with task priorities (critical path)
5. Put all "enabled" tasks in PlannedTasks_k



DARTS (Data-Aware Reactive Task Scheduling)

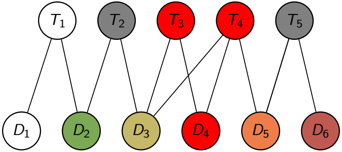


How to fill PlannedTasks$_k$ when needed:

1. Concentrate on data, choose "best" data to load
2. Look for tasks that GPU_k can do with D + its current data
3. Choose data with largest ratio:

$$\frac{\text{computation time of tasks enabled with } D}{\text{time needed to transfer data } D}$$

4. Break ties with task priorities (critical path)
5. Put all "enabled" tasks in PlannedTasks$_k$



Custom eviction policy

Existing cache management policies:

- With no information about future tasks/requests:
  simple policies based on past usage, eg. Last Recently Used (LRU)
- With perfect information on future accesses:
  Belady's rule (1966): evict data with furthest access

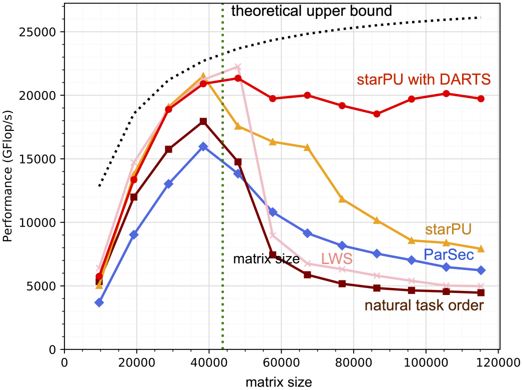
In our system:

- No complete vision of the future ☹
- Window of allocated tasks and planned tasks ☺

Eviction policy for DARTS:

1. Remove data used by fewest tasks in PlannedTasks
2. If needed, apply Belady's rule on TaskBuffer

Performance on memory-limited GPUs



- Cholesky factorization on 2 GPUs
- Green vertical line: matrix uses all available memory

Summary and Perspectives

- DAGs: convenient way to model structured computations, can include memory demand
- Polynomial algorithms to limit memory for simple graphs: trees, SP (sequential scheduling)
- Parallel processing: trade-off memory vs. disk, NP-complete even for trees, but workarounds exist!
- Other models exist:
  - Memory demand for computation
  - Output data shared by several successors
- Other problems:
  - If memory too scarce, store data on disk, minimize I/Os
  - Or delete data and recompute it later ("offloading" in neural network training)

Summary and Perspectives

- DAGs: convenient way to model structured computations, can include memory demand
- Polynomial algorithms to limit memory for simple graphs: trees, SP (sequential scheduling)
- Parallel processing: trade-off memory vs. disk, NP-complete even for trees, but workarounds exist!
- Other models exist:
  - Memory demand for computation
  - Output data shared by several successors
- Other problems:
  - If memory too scarce, store data on disk, minimize I/Os
  - Or delete data and recompute it later (“offloading” in neural network training)