Loris Marchal — HDR defense

Memory and data aware scheduling

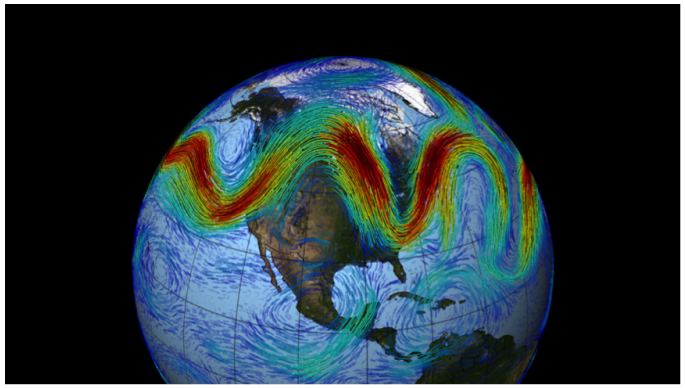
committee:

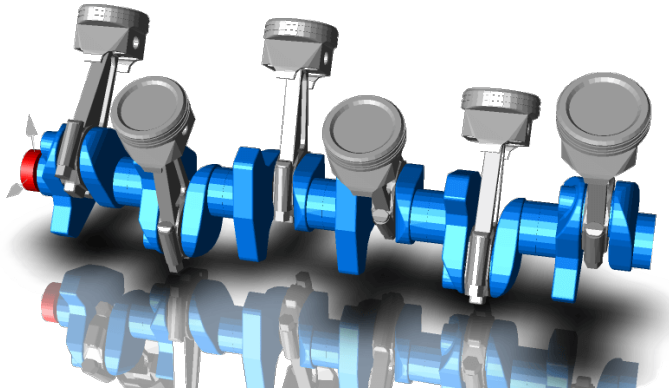
Ümit Çatalyürek (reviewer)	*Georgia Tech.*
Pierre Manneback	*Polytech-Mons*
Alix Munier Kordon	*Univ. Paris 6*
Cynthia Phillips (reviewer)	*Sandia Nat. Lab.*
Yves Robert	*ENS Lyon*
Denis Trystram (reviewer)	*Grenoble INP*

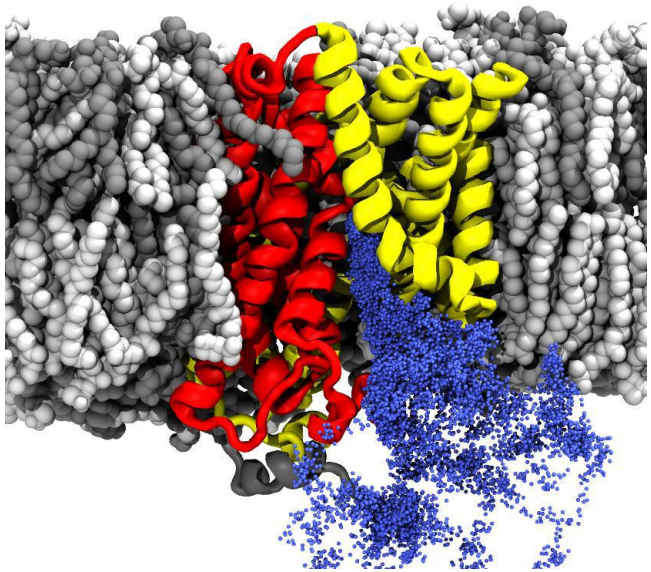
Position and supervision

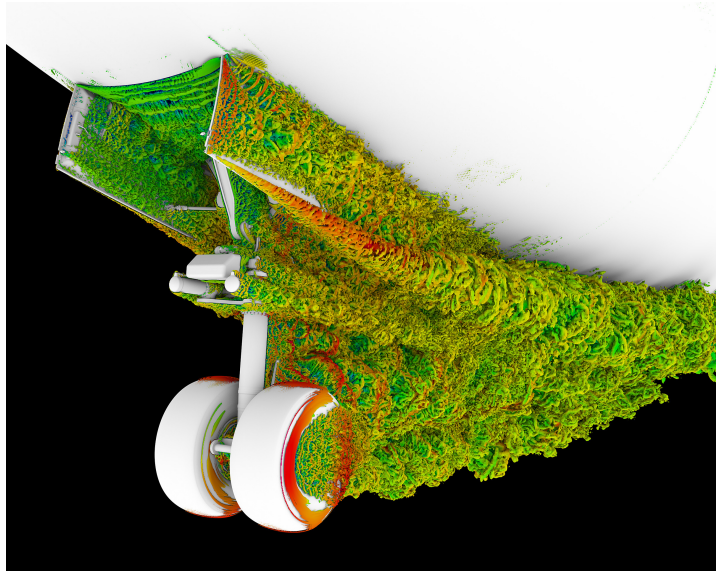
- CNRS researcher since 2007
- 4 PhD students:
  - Mathias Jacquelin: 2008 – 2011 (with Y. Robert)
    (research scientist at Lawrence Berkeley Nat. Lab., USA)
  - Julien Herrmann: 2012 – 2015 (with Y. Robert)
    (postdoc at Georgia Tech., USA)
  - Bertrand Simon: 2015 – 2018 (with F. Vivien)
    (defense in July)
  - Changjiang Gou: 2016 – . . . (with A. Benoit)

Motivation and context – scientific computing





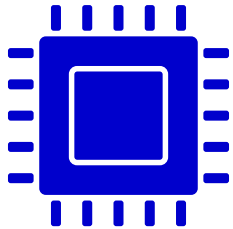




- Simulation of larger systems with better accuracy
- Need for better performance on larger data

Increasing complexity of computing platforms

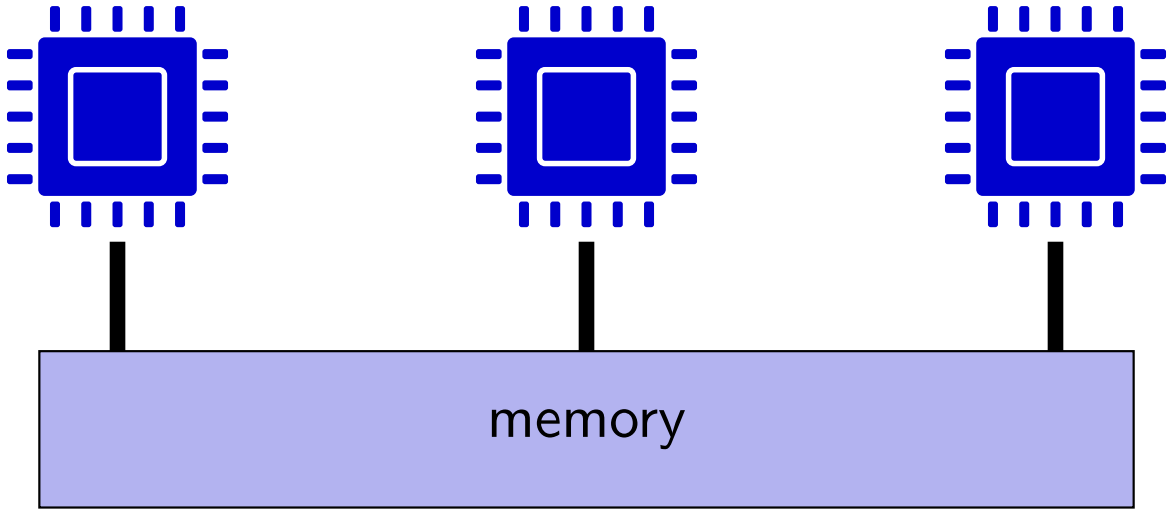
Evolution of computing platforms:



- Single processor

Increasing complexity of computing platforms

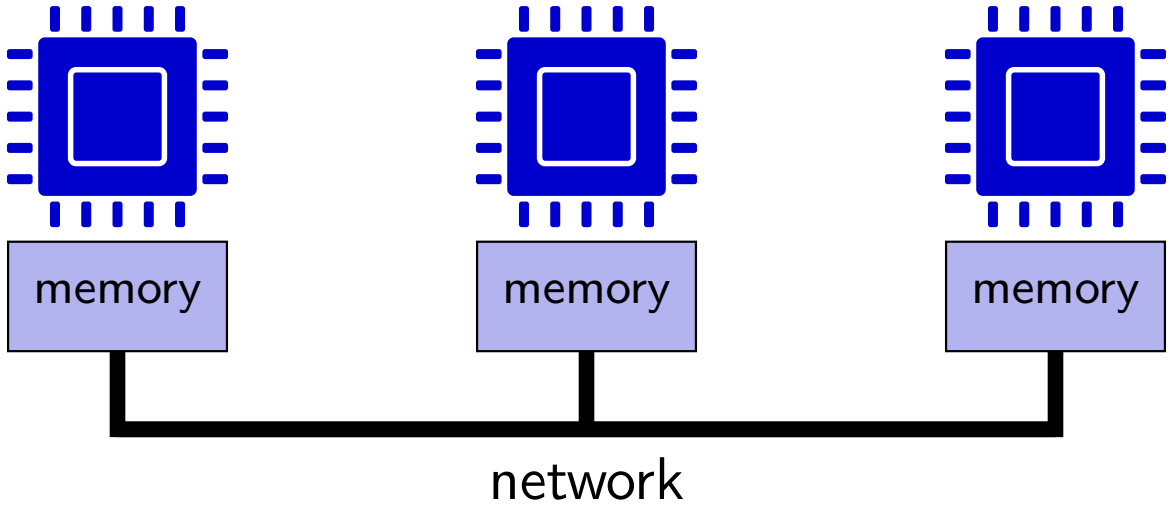
Evolution of computing platforms:



- Single processor
- n processors with shared memory

Increasing complexity of computing platforms

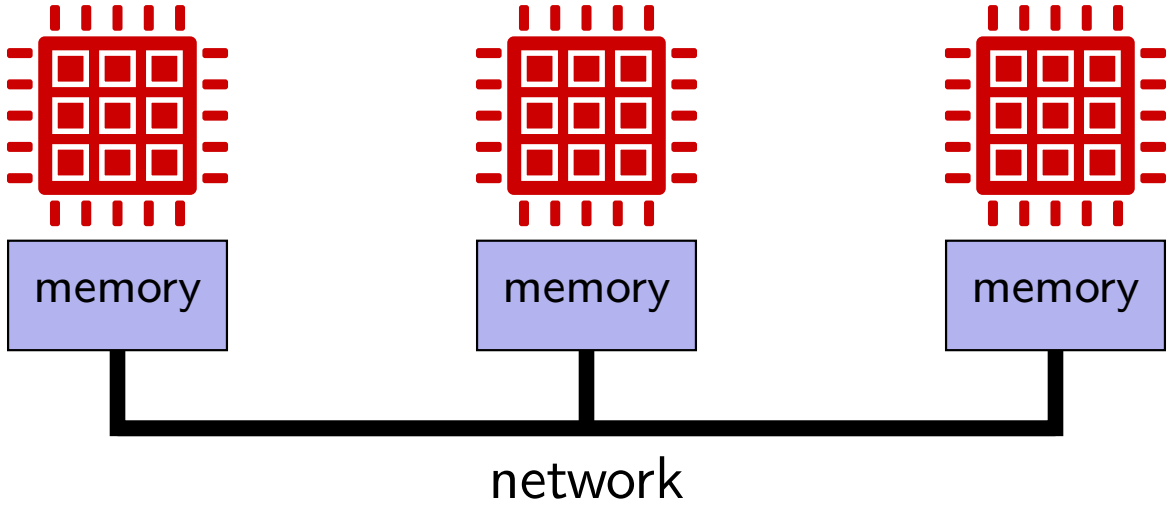
Evolution of computing platforms:



- Single processor
- n processors with shared memory
- n processors with communication delays

Increasing complexity of computing platforms

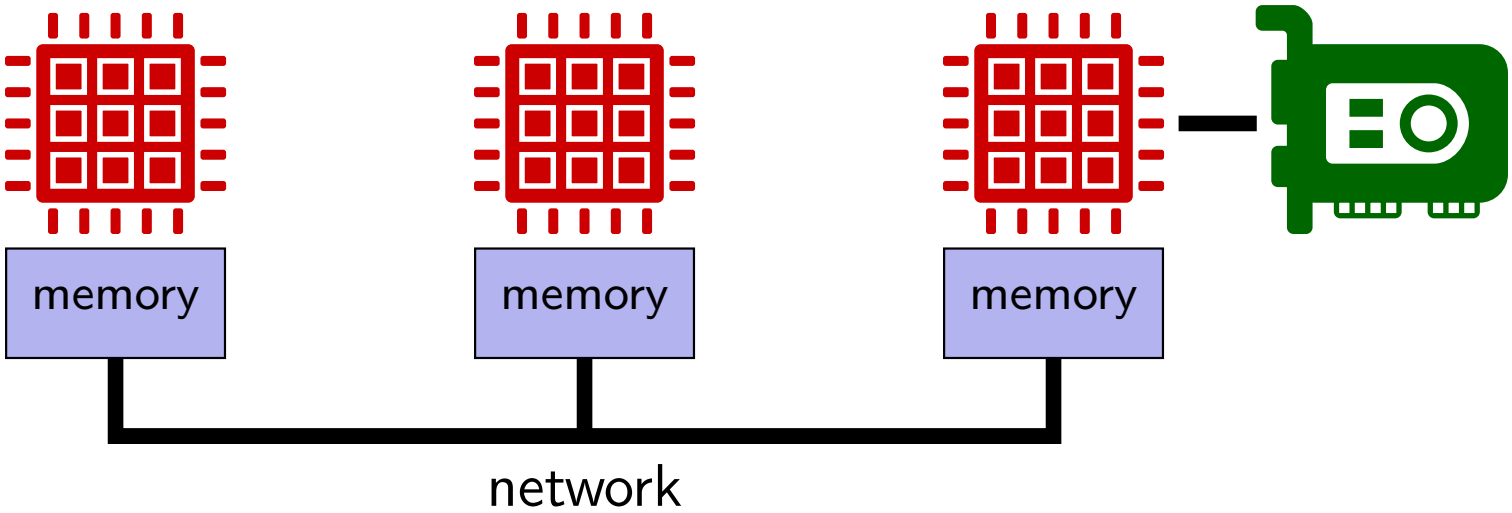
Evolution of computing platforms:



- Single processor
- n processors with shared memory
- n processors with communication delays
- n multi-core processors with memory hierachies

Increasing complexity of computing platforms

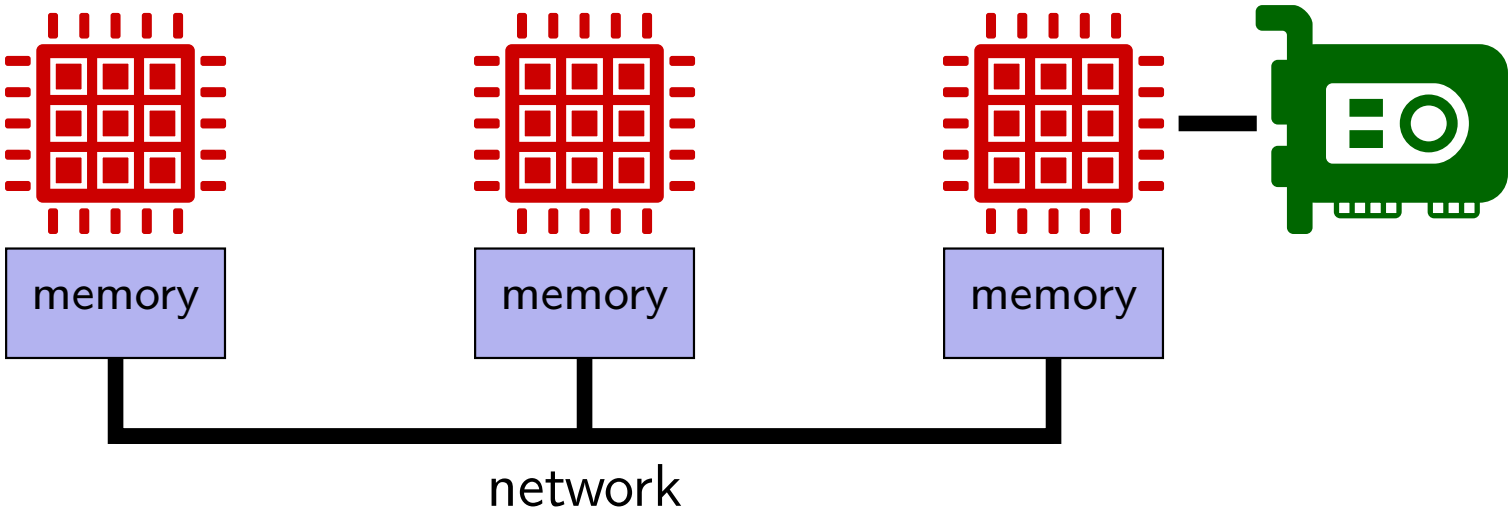
Evolution of computing platforms:



- Single processor
- n processors with shared memory
- n processors with communication delays
- n multi-core processors with memory hierachies
- n multi-core processors and k accelerators (GPUs)

Increasing complexity of computing platforms

Evolution of computing platforms:



- Single processor
- n processors with shared memory
- n processors with communication delays
- n multi-core processors with memory hierachies
- n multi-core processors and k accelerators (GPUs)

My focus: optimize application mapping and task scheduling for memory constraints and data movement

Contributions

- Part I. Task graph scheduling with limited memory
  - Chapter 2. Memory-aware dataflow model
  - Chapter 3. Peak Memory and I/O Volume on Trees
  - Chapter 4. Peak memory of series-parallel task graphs
  - Chapter 5. Hybrid scheduling with bounded memory
  - Chapter 6. Memory-aware parallel tree processing
- Part II. Minimizing data movement for matrix computations
  - Chapter 7. Matrix product for memory hierarchy
  - Chapter 8. Data redistribution for parallel computing
  - Chapter 9. Dynamic scheduling for matrix computations

Contributions

- Part I. Task graph scheduling with limited memory
  - Chapter 2. Memory-aware dataflow model
  - Chapter 3. Peak Memory and I/O Volume on Trees
  - Chapter 4. Peak memory of series-parallel task graphs
  - Chapter 5. Hybrid scheduling with bounded memory
  - Chapter 6. Memory-aware parallel tree processing
- Part II. Minimizing data movement for matrix computations
  - Chapter 7. Matrix product for memory hierarchy
  - Chapter 8. Data redistribution for parallel computing
  - Chapter 9. Dynamic scheduling for matrix computations

Outline of this talk

Introduction

1. Scheduling tree-shaped task graphs with bounded memory

2. Data redistribution for parallel computing

Research perspectives

Outline

Introduction

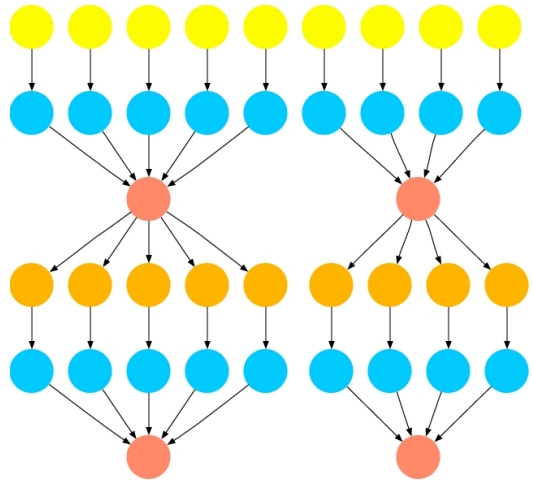
1. Scheduling tree-shaped task graphs with bounded memory

2. Data redistribution for parallel computing

Research perspectives

Modeling scientific applications as task graphs

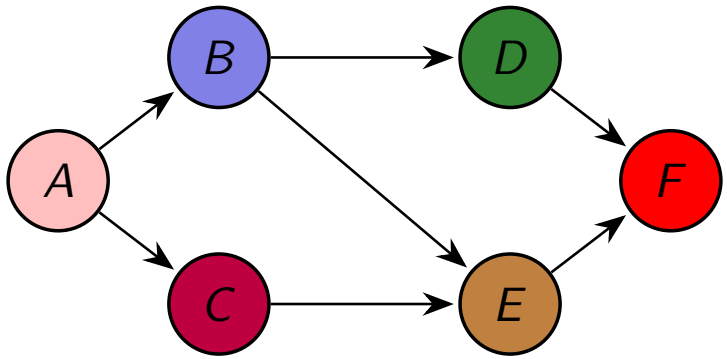
- Scientific applications divided into rather independent modules (tasks)
- Tasks linked through data dependencies
- Directed Acyclic Graph of tasks



- Abundant literature about (theoretical) task graph scheduling
- Popularized by runtime schedulers (ParSec, StarPU, XKaapi, OpenMP 4)
  - Express dependencies between tasks
  - Write code for each task on (possibly several) processing units
  - Choose task mapping at runtime

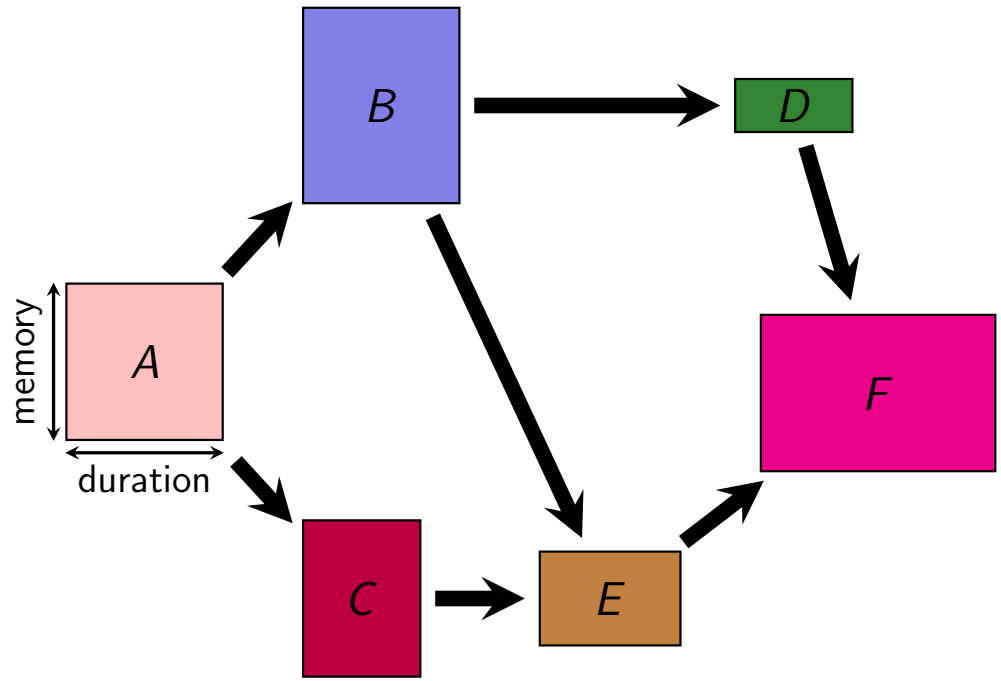
Task graph scheduling and memory

- Consider a simple task graph



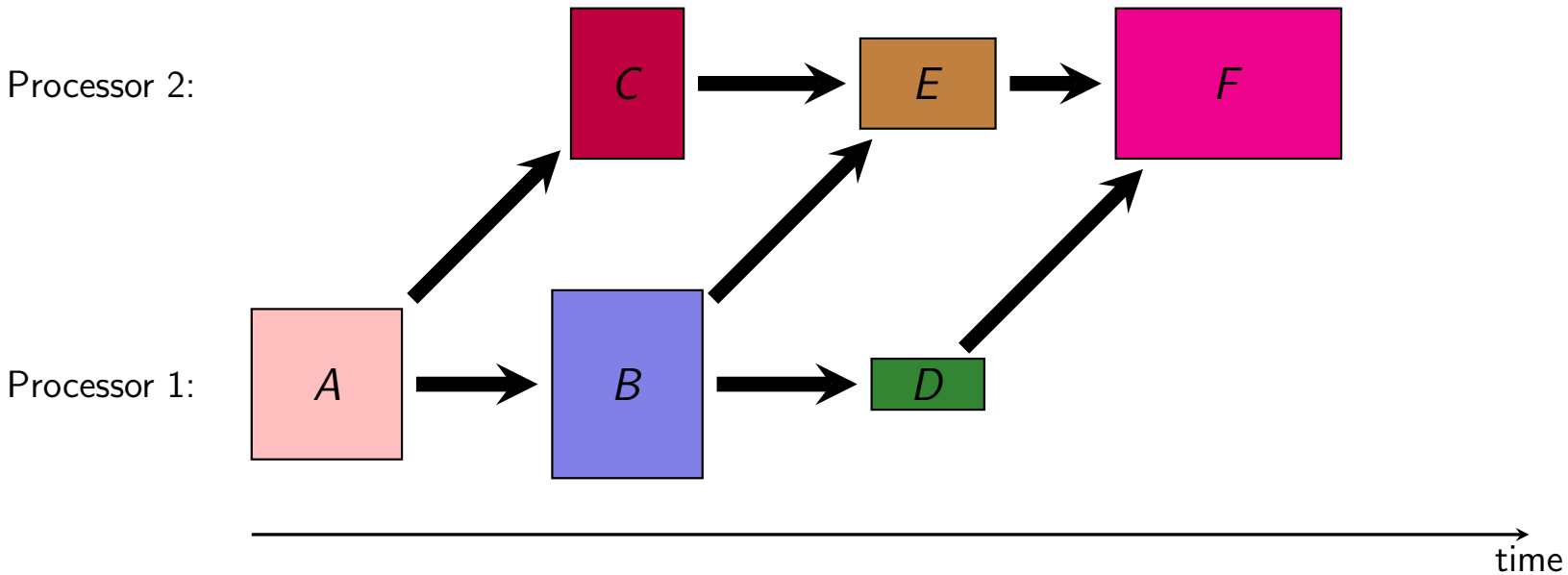
Task graph scheduling and memory

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



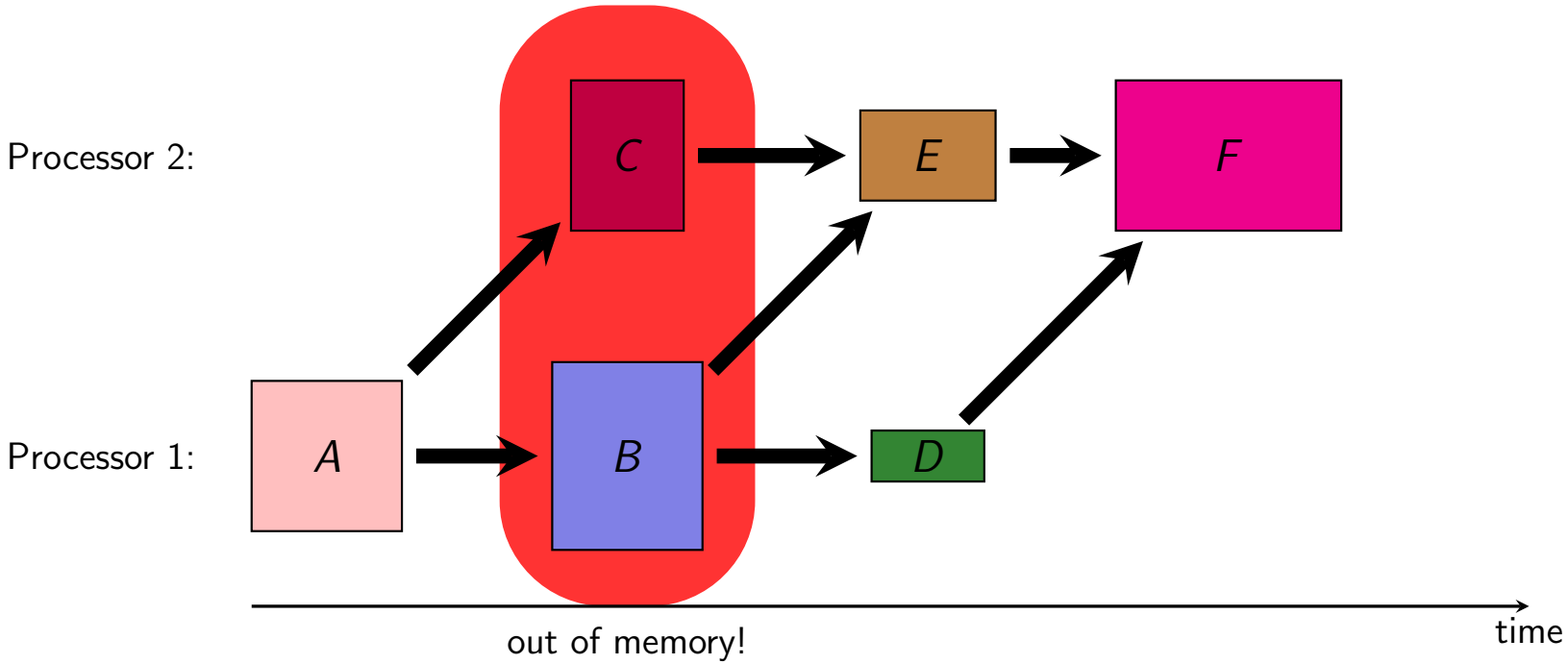
Task graph scheduling and memory

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



Task graph scheduling and memory

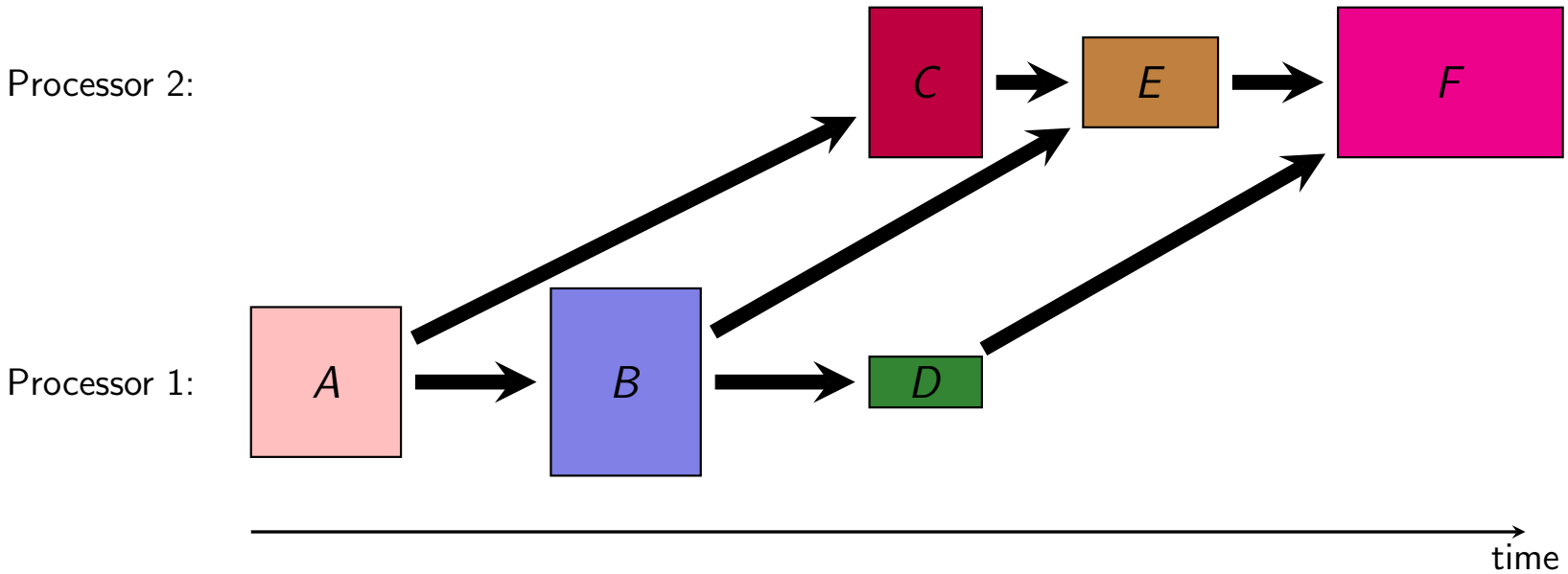
- Consider a simple task graph
- Tasks have durations and memory demands



- Peak memory: maximum memory usage

Task graph scheduling and memory

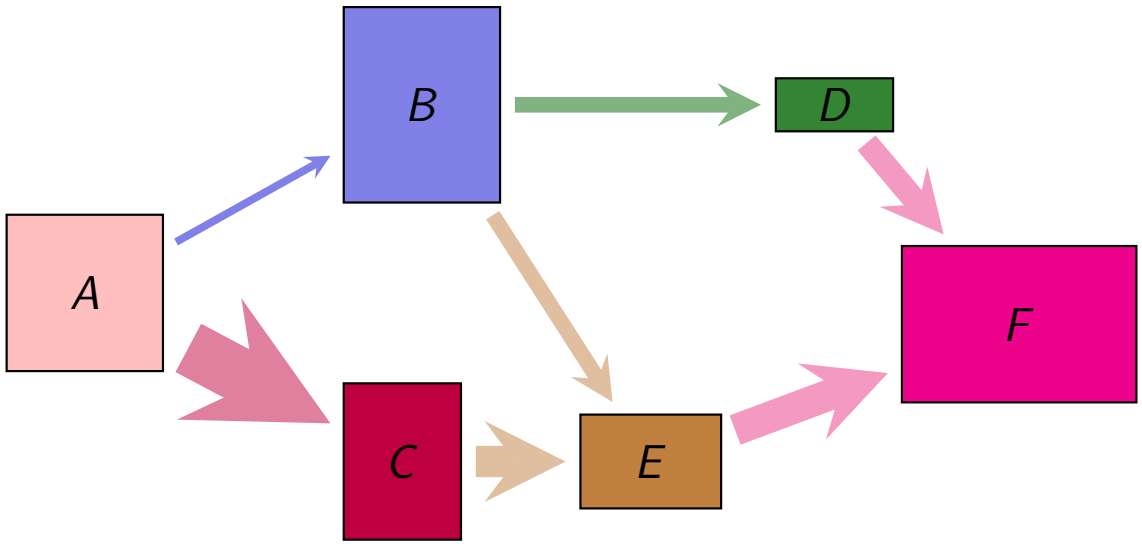
- Consider a simple task graph
- Tasks have durations and memory demands



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

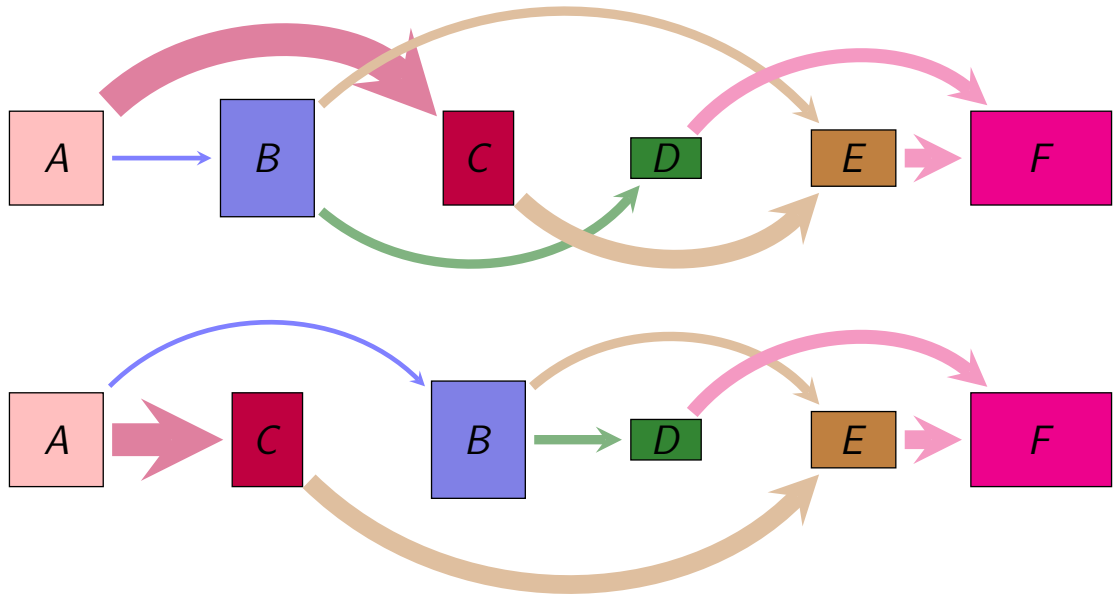
Going back to sequential processing

- Temporary data require memory
- Scheduling influences the peak memory



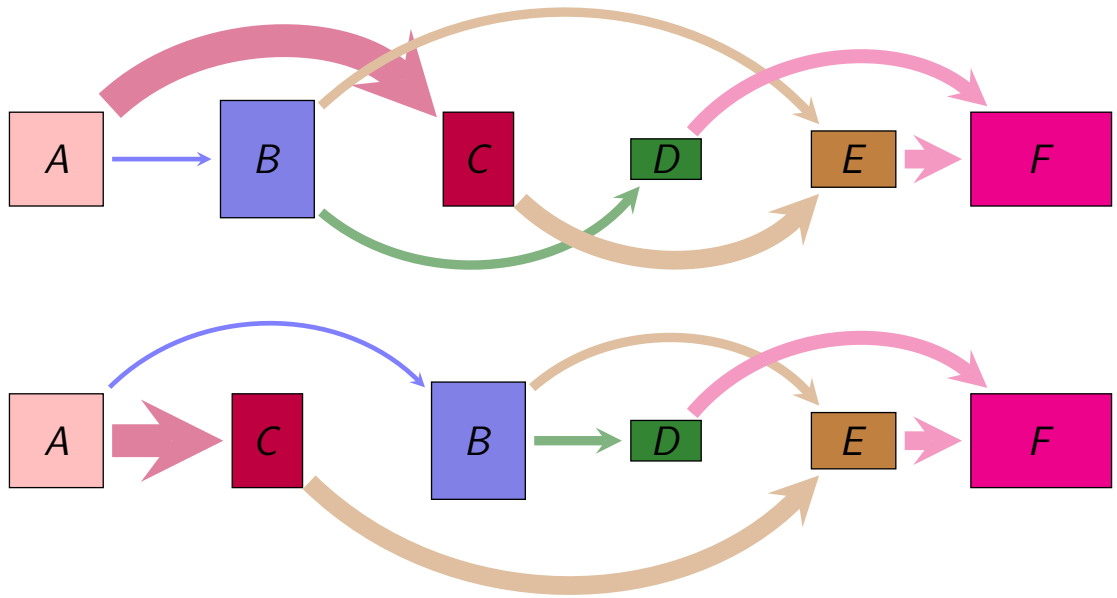
Going back to sequential processing

- Temporary data require memory
- Scheduling influences the peak memory



Going back to sequential processing

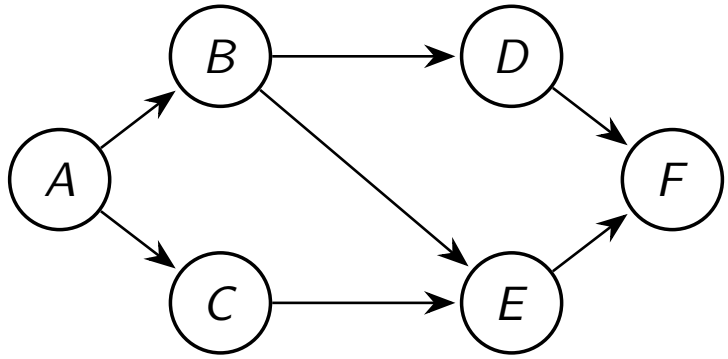
- Temporary data require memory
- Scheduling influences the peak memory



When minimum memory demand $>$ available memory:

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

(Black) Pebble game (1970s)



Rules of the game (possible moves):

1. Put a pebble on a source vertex
2. Remove a pebble from a vertex
3. Put a pebble on a vertex if all its predecessors are pebbled

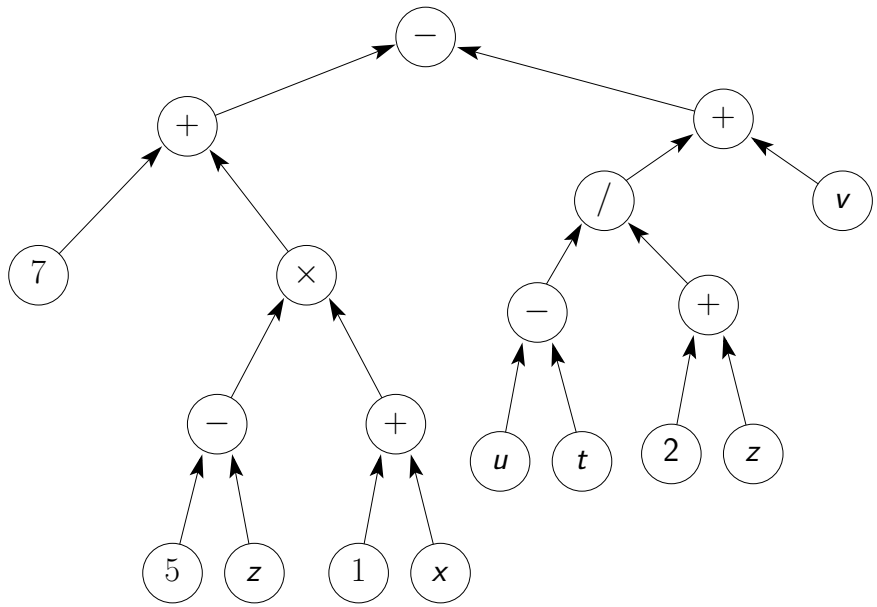
Objectives:

- Pebble all output vertices
- Minimize the number of pebbles used

Register allocation & pebble game

How to efficiently compute the following arithmetic expression with the minimum number of registers?

$$7 + (1 + x)(5 - z) - ((u - t)/(2 + z)) + v$$



Pebble movements corresponds to register operations:

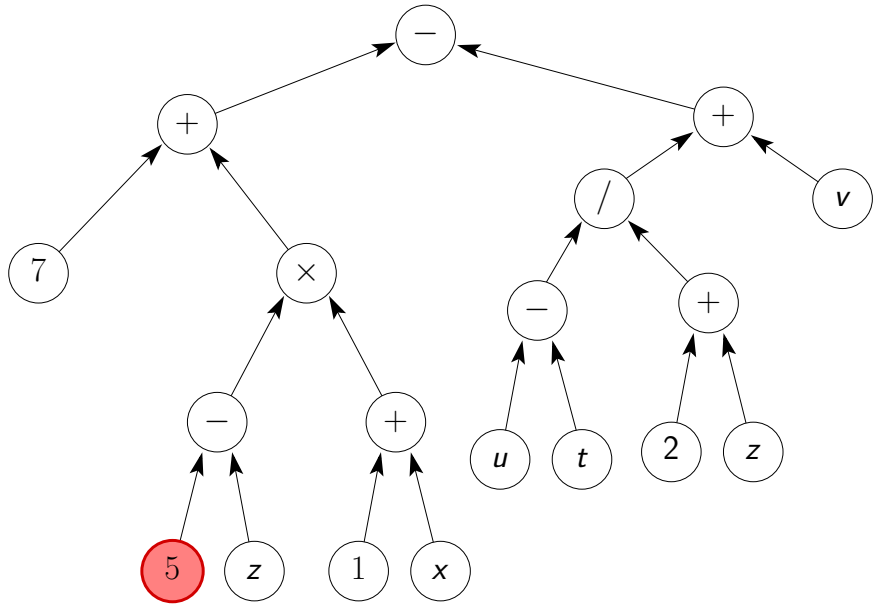
1. Pebbling a source vertex: load an input into register
2. Removing a pebble: discard value in register
3. Pebbling a vertex: computing a value in a new register

Objective: use a minimal number of registers

Register allocation & pebble game

How to efficiently compute the following arithmetic expression with the minimum number of registers?

$$7 + (1 + x)(5 - z) - ((u - t)/(2 + z)) + v$$



Pebble movements corresponds to register operations:

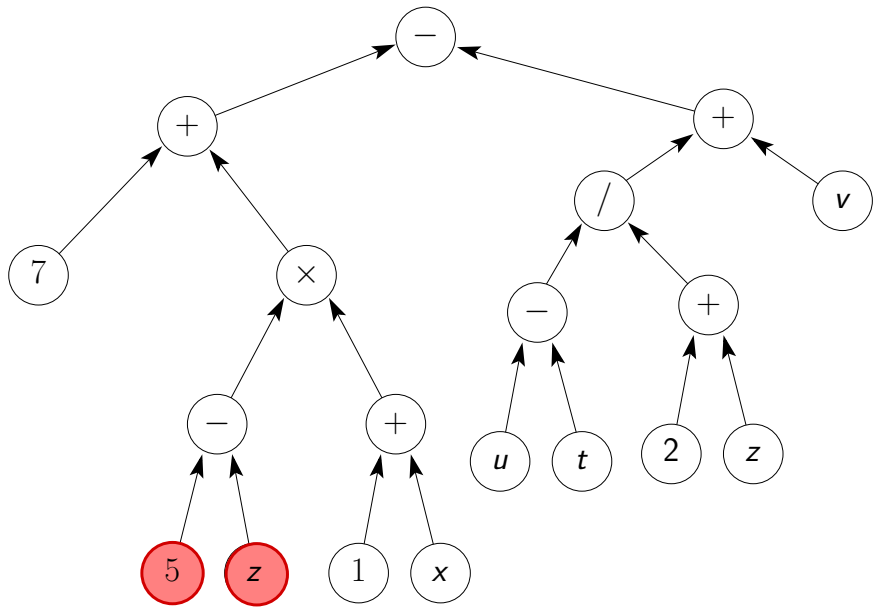
1. Pebbling a source vertex: load an input into register
2. Removing a pebble: discard value in register
3. Pebbling a vertex: computing a value in a new register

Objective: use a minimal number of registers

Register allocation & pebble game

How to efficiently compute the following arithmetic expression with the minimum number of registers?

$$7 + (1 + x)(5 - z) - ((u - t)/(2 + z)) + v$$



Pebble movements corresponds to register operations:

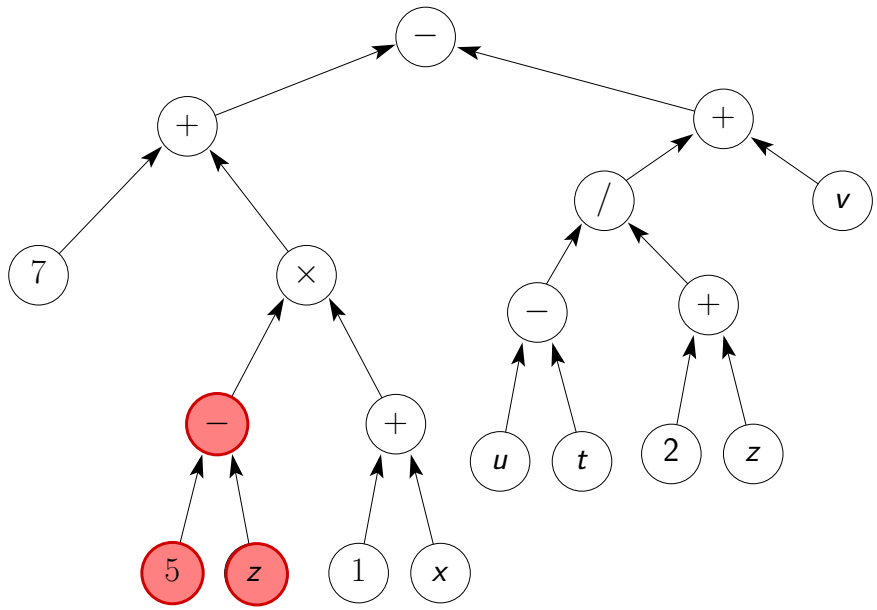
1. Pebbling a source vertex: load an input into register
2. Removing a pebble: discard value in register
3. Pebbling a vertex: computing a value in a new register

Objective: use a minimal number of registers

Register allocation & pebble game

How to efficiently compute the following arithmetic expression with the minimum number of registers?

$$7 + (1 + x)(5 - z) - ((u - t)/(2 + z)) + v$$



Pebble movements corresponds to register operations:

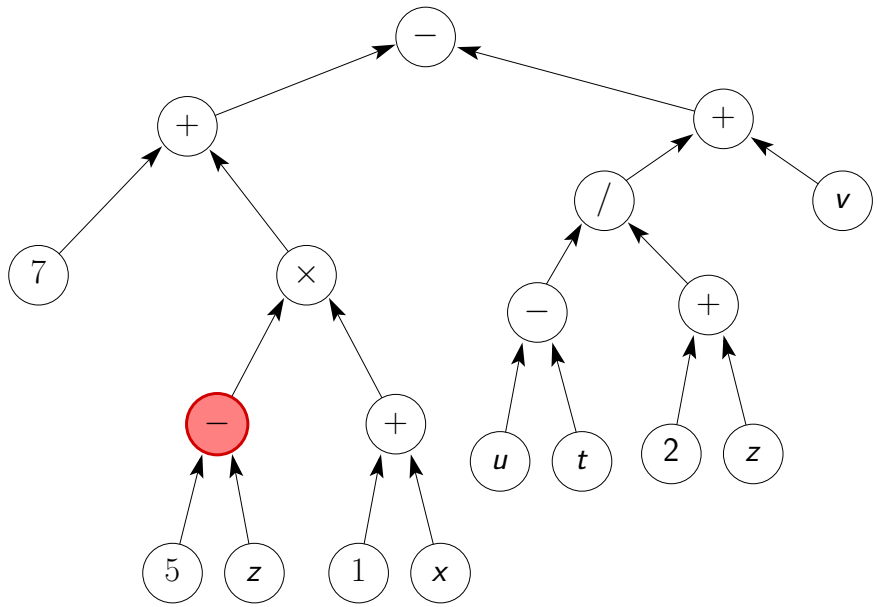
1. Pebbling a source vertex: load an input into register
2. Removing a pebble: discard value in register
3. Pebbling a vertex: computing a value in a new register

Objective: use a minimal number of registers

Register allocation & pebble game

How to efficiently compute the following arithmetic expression with the minimum number of registers?

$$7 + (1 + x)(5 - z) - ((u - t)/(2 + z)) + v$$



Pebble movements corresponds to register operations:

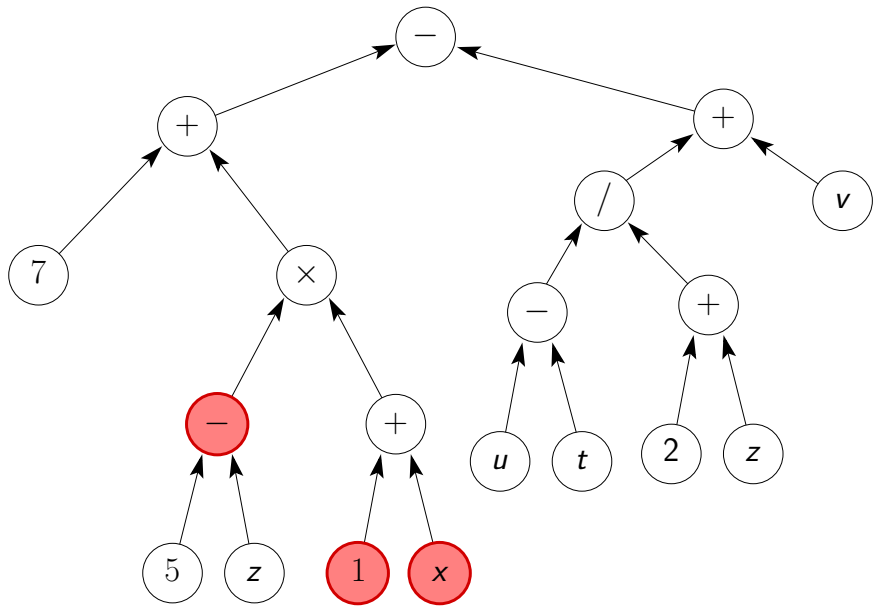
1. Pebbling a source vertex: load an input into register
2. Removing a pebble: discard value in register
3. Pebbling a vertex: computing a value in a new register

Objective: use a minimal number of registers

Register allocation & pebble game

How to efficiently compute the following arithmetic expression with the minimum number of registers?

$$7 + (1 + x)(5 - z) - ((u - t)/(2 + z)) + v$$



Pebble movements corresponds to register operations:

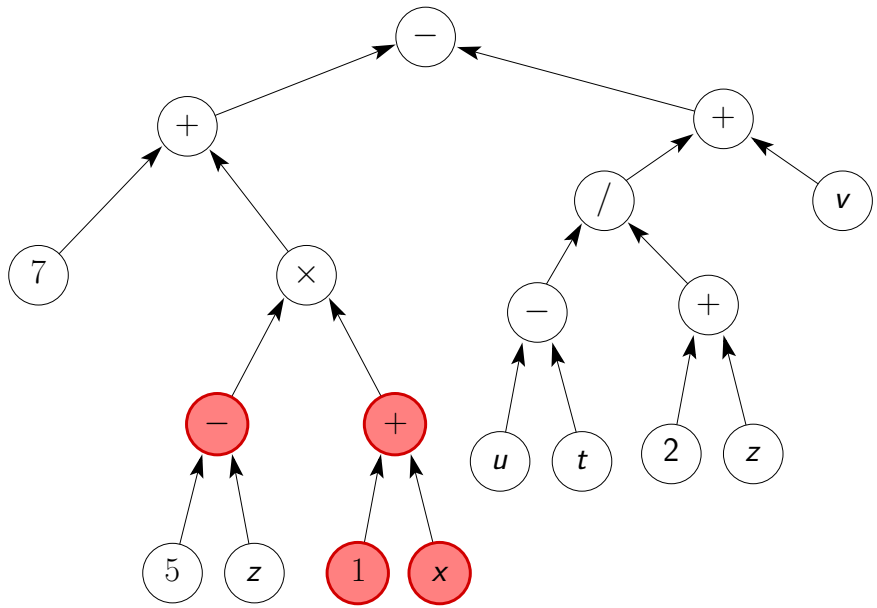
1. Pebbling a source vertex: load an input into register
2. Removing a pebble: discard value in register
3. Pebbling a vertex: computing a value in a new register

Objective: use a minimal number of registers

Register allocation & pebble game

How to efficiently compute the following arithmetic expression with the minimum number of registers?

$$7 + (1 + x)(5 - z) - ((u - t)/(2 + z)) + v$$



Pebble movements corresponds to register operations:

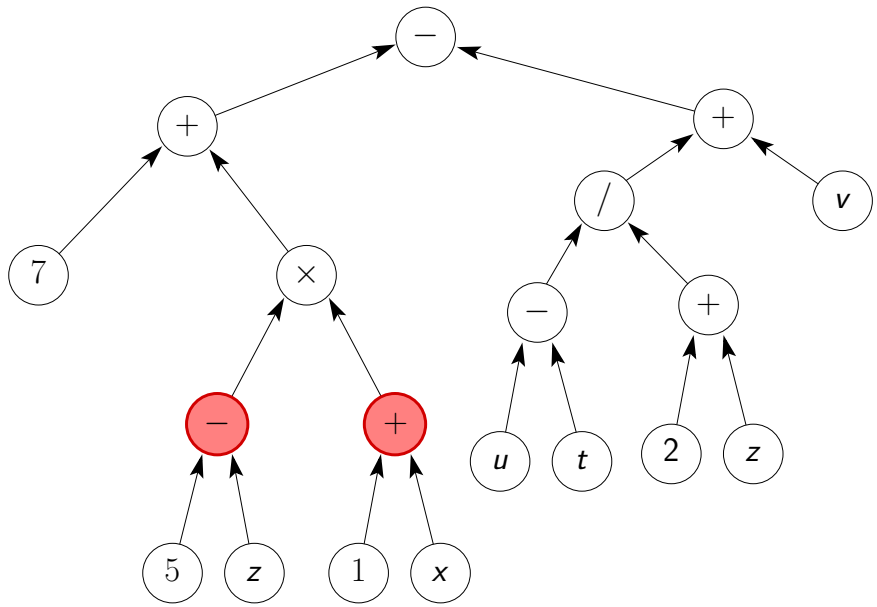
1. Pebbling a source vertex: load an input into register
2. Removing a pebble: discard value in register
3. Pebbling a vertex: computing a value in a new register

Objective: use a minimal number of registers

Register allocation & pebble game

How to efficiently compute the following arithmetic expression with the minimum number of registers?

$$7 + (1 + x)(5 - z) - ((u - t)/(2 + z)) + v$$



Pebble movements corresponds to register operations:

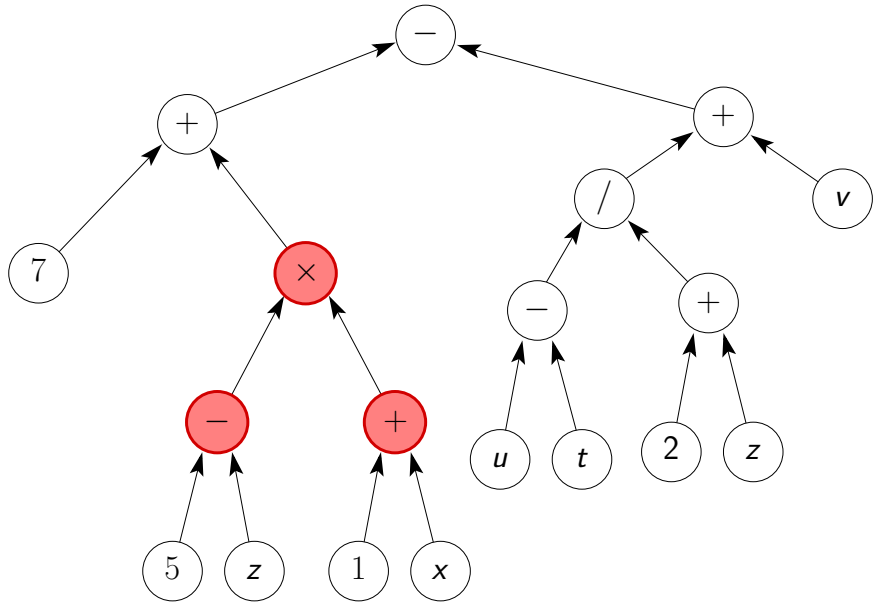
1. Pebbling a source vertex: load an input into register
2. Removing a pebble: discard value in register
3. Pebbling a vertex: computing a value in a new register

Objective: use a minimal number of registers

Register allocation & pebble game

How to efficiently compute the following arithmetic expression with the minimum number of registers?

$$7 + (1 + x)(5 - z) - ((u - t)/(2 + z)) + v$$



Pebble movements corresponds to register operations:

1. Pebbling a source vertex: load an input into register
2. Removing a pebble: discard value in register
3. Pebbling a vertex: computing a value in a new register

Objective: use a minimal number of registers

Register allocation & pebble game

How to efficiently compute the following arithmetic expression with the minimum number of registers?

$$7 + (1 + x)(5 - z) - ((u - t)/(2 + z)) + v$$

Complexity results

Problem on trees:

- Polynomial algorithm [Sethi & Ullman, 1970]

General problem on DAGs (common subexpressions):

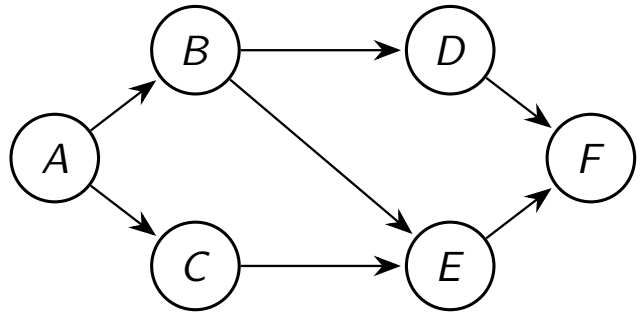
- P-Space complete [Gilbert, Lengauer & Tarjan, 1980]
- Without re-computation: NP-complete [Sethi, 1973]

Pebble movements corresponds to register operations:

1. Pebbling a source vertex: load an input into register
2. Removing a pebble: discard value in register
3. Pebbling a vertex: computing a value in a new register

Objective: use a minimal number of registers

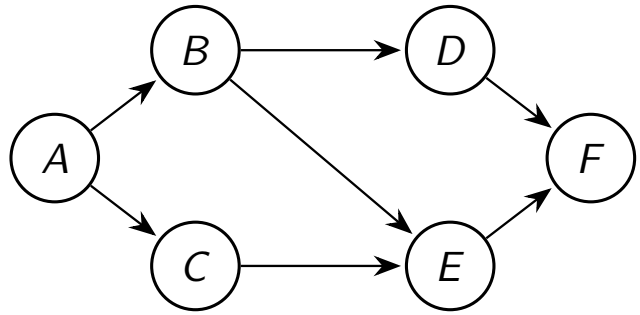
Red-Blue pebble game (Hong & Kung 1991)



Rules of the game (possible moves):

1. Put a red pebble on a source vertex
2. Remove a red pebble from a vertex
3. Put a red pebble on a vertex if all predecessors red-pebbled

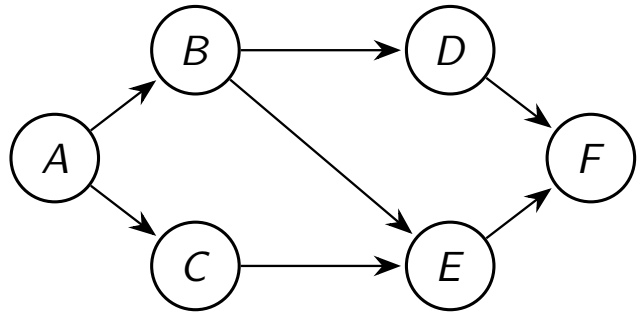
Red-Blue pebble game (Hong & Kung 1991)



Rules of the game (possible moves):

1. Put a red pebble on a source vertex
2. Remove a red pebble from a vertex
3. Put a red pebble on a vertex if all predecessors red-pebbled
4. Put a red pebble on a blue-pebbled vertex
5. Put a blue pebble on a red-pebbled vertex
6. Remove a blue pebble from a vertex

Red-Blue pebble game (Hong & Kung 1991)

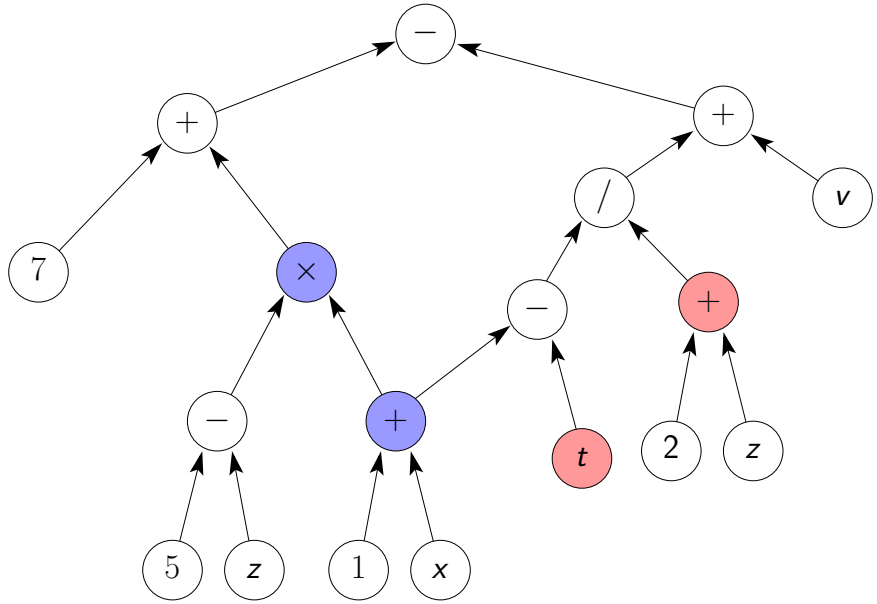


Rules of the game (possible moves):

1. Put a red pebble on a source vertex
2. Remove a red pebble from a vertex
3. Put a red pebble on a vertex if all predecessors red-pebbled
4. Put a red pebble on a blue-pebbled vertex
5. Put a blue pebble on a red-pebbled vertex
6. Remove a blue pebble from a vertex
7. Never use more than M red pebbles

Objective: pebble graph with minimum number of rules 4/5

Red-Blue pebble game and I/O complexity



Analogy with out-of-core processing:

- red pebbles: memory slots
- blue pebbles: secondary storage (disk)
- red $\longrightarrow$ blue: write to disk, evict from memory
- blue $\longrightarrow$ red: read from disk, load in memory
- M: number of available memory slots

Objective: minimum number of I/O operations

Red/Blue pebble game – Results

Idea of Hong & Kung:

- Partition graph into sets with at most M reads and writes
- Number of sets needed $\Rightarrow$ lower-bound on I/Os

Lower-bound on I/Os:

- Product of 2 n^2-matrices: $\Theta(n^3/\sqrt{M})$
- Other regular graphs (FFT)

Later extended to other matrix operations:

- Lower bounds
- Communication-avoiding algorithms

Summary

Three problems:

- Memory minimization
  Black pebble game
- I/O minimization for out-of-core processing
  Red-Blue pebble game
- Memory/Time tradeoff for parallel processing

Summary

Three problems:

- Memory minimization
  Black pebble game

- I/O minimization for out-of-core processing
  Red-Blue pebble game

- Memory/Time tradeoff for parallel processing

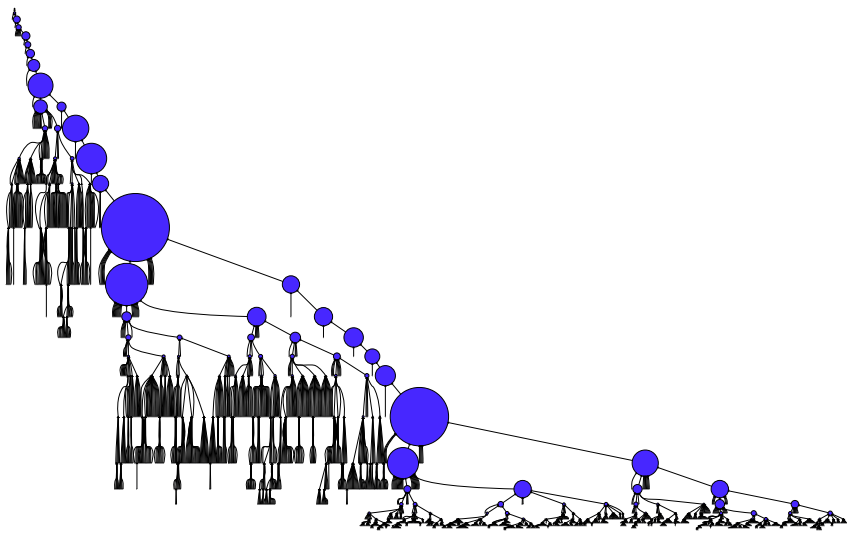
Shift of focus:

- Pebble games limited to unit-size data
- Target coarse-grain tasks, with heterogeneous data sizes

Tree-shaped task graphs

- Multifrontal sparse matrix factorization

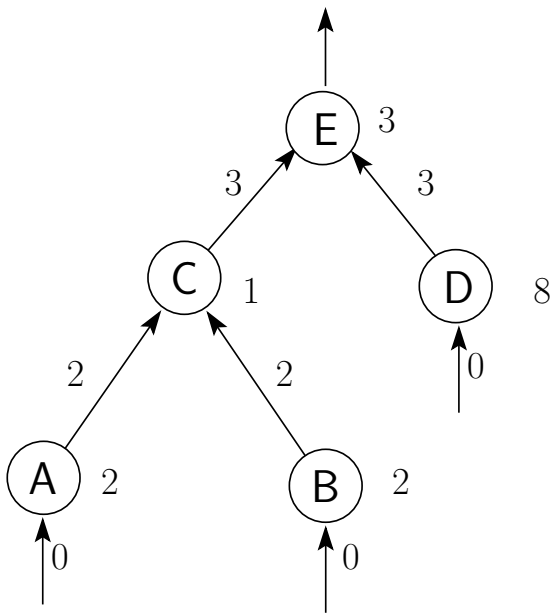
- To cope with complex/heterogeneous platforms:
  - Express factorization as a task graph
  - Scheduled using specialized runtime
- Assembly/Elimination tree: task graph is an in-tree



Problem:

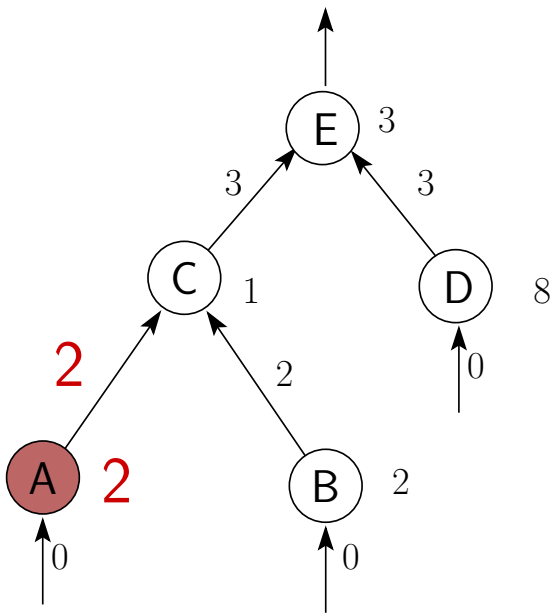
- Large temporary data
- Memory becomes a bottleneck
- Schedule trees with limited memory

Tree traversal influences peak memory



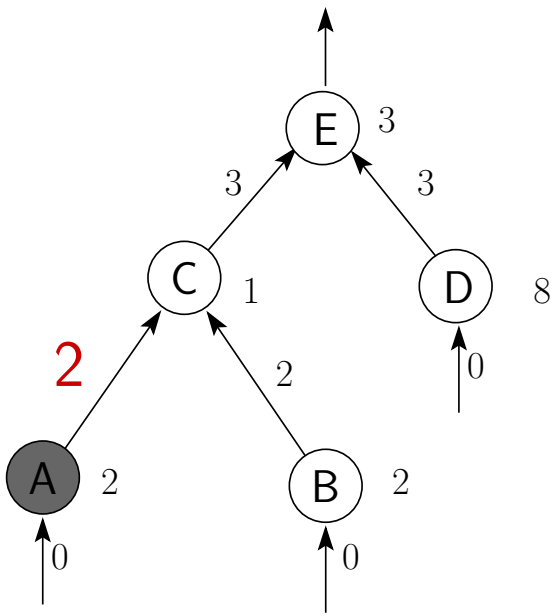
- Nodes: tasks
  - Node weight: temporary data (m_i)
- Edges: dependencies (data)
  - Edge weight: data size ($d_{i,j}$)
- Process a node:
  load inputs + output + temporary data

Tree traversal influences peak memory



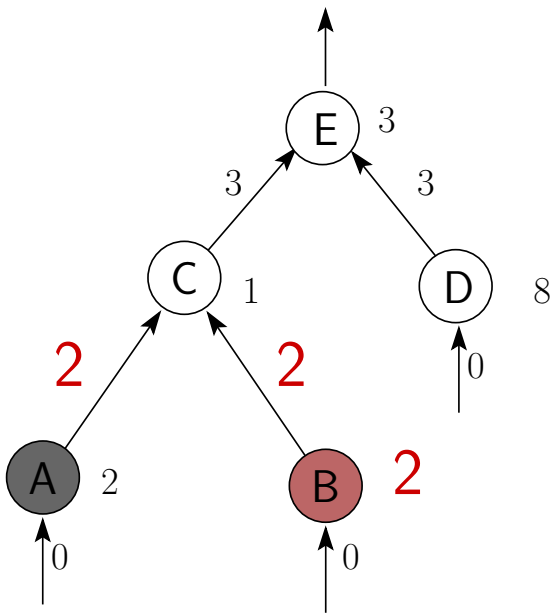
- Nodes: tasks
  - Node weight: temporary data (m_i)
- Edges: dependencies (data)
  - Edge weight: data size ($d_{i,j}$)
- Process a node:
  load inputs + output + temporary data
- Memory ($\rightarrow$): 4

Tree traversal influences peak memory



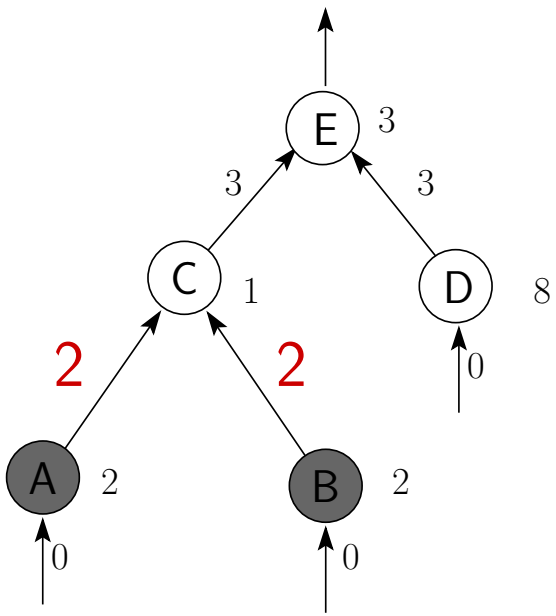
- Nodes: tasks
  - Node weight: temporary data (m_i)
- Edges: dependencies (data)
  - Edge weight: data size ($d_{i,j}$)
- Process a node:
  load inputs + output + temporary data
- Memory ($\rightarrow$): 4, 2

Tree traversal influences peak memory



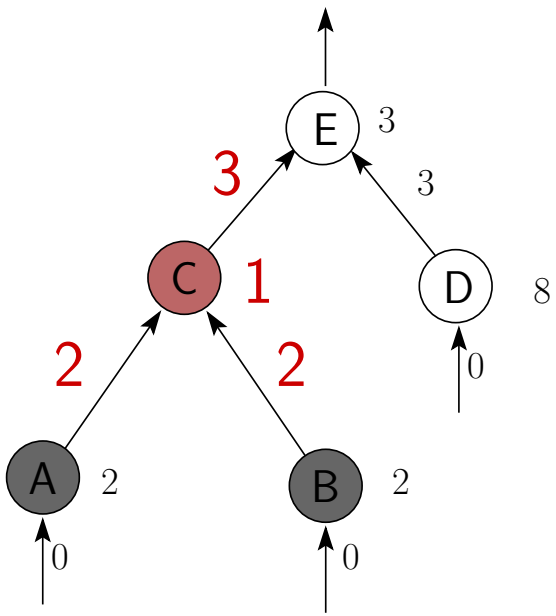
- Nodes: tasks
  - Node weight: temporary data (m_i)
- Edges: dependencies (data)
  - Edge weight: data size ($d_{i,j}$)
- Process a node:
  load inputs + output + temporary data
- Memory ($\rightarrow$): 4, 2, 6

Tree traversal influences peak memory



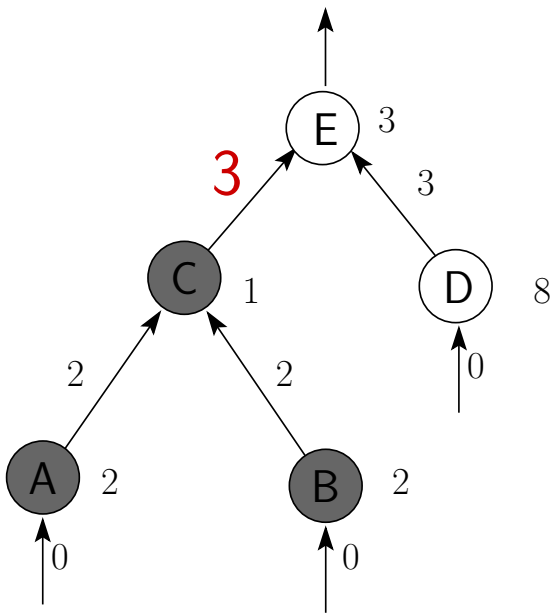
- Nodes: tasks
  - Node weight: temporary data (m_i)
- Edges: dependencies (data)
  - Edge weight: data size ($d_{i,j}$)
- Process a node:
  load inputs + output + temporary data
- Memory ($\rightarrow$): 4, 2, 6, 4

Tree traversal influences peak memory



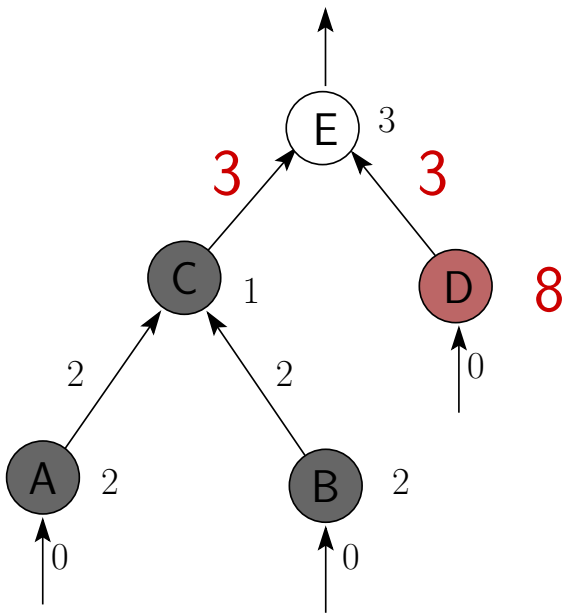
- Nodes: tasks
  - Node weight: temporary data (m_i)
- Edges: dependencies (data)
  - Edge weight: data size ($d_{i,j}$)
- Process a node:
  load inputs + output + temporary data
- Memory ($\rightarrow$): 4, 2, 6, 4, 8

Tree traversal influences peak memory



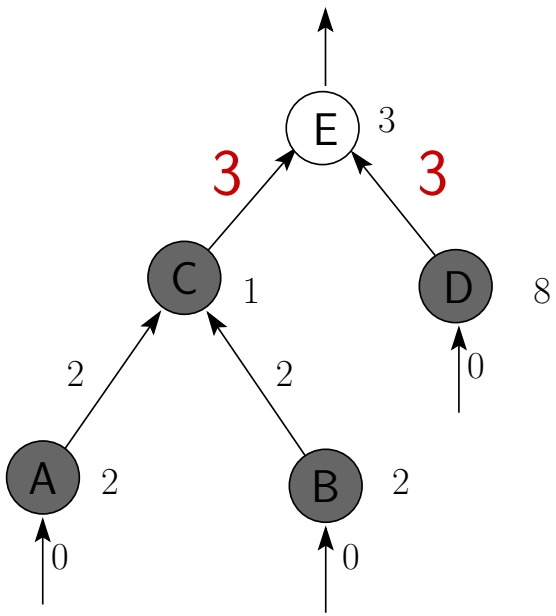
- Nodes: tasks
  - Node weight: temporary data (m_i)
- Edges: dependencies (data)
  - Edge weight: data size ($d_{i,j}$)
- Process a node:
  load inputs + output + temporary data
- Memory ($\rightarrow$): 4, 2, 6, 4, 8, 3

Tree traversal influences peak memory



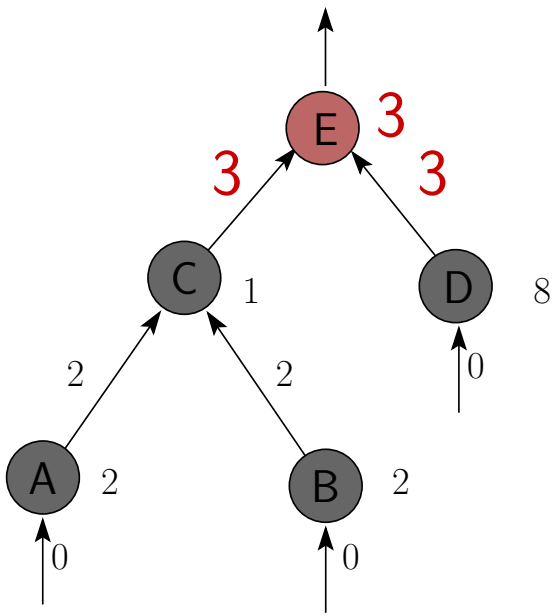
- Nodes: tasks
  - Node weight: temporary data (m_i)
- Edges: dependencies (data)
  - Edge weight: data size ($d_{i,j}$)
- Process a node:
  load inputs + output + temporary data
- Memory ($\rightarrow$): 4, 2, 6, 4, 8, 3, 14

Tree traversal influences peak memory



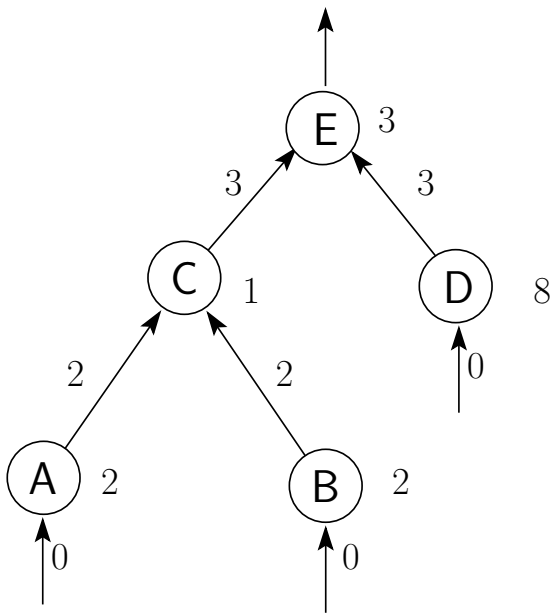
- Nodes: tasks
  - Node weight: temporary data (m_i)
- Edges: dependencies (data)
  - Edge weight: data size ($d_{i,j}$)
- Process a node:
  load inputs + output + temporary data
- Memory ($\rightarrow$): 4, 2, 6, 4, 8, 3, 14, 6

Tree traversal influences peak memory



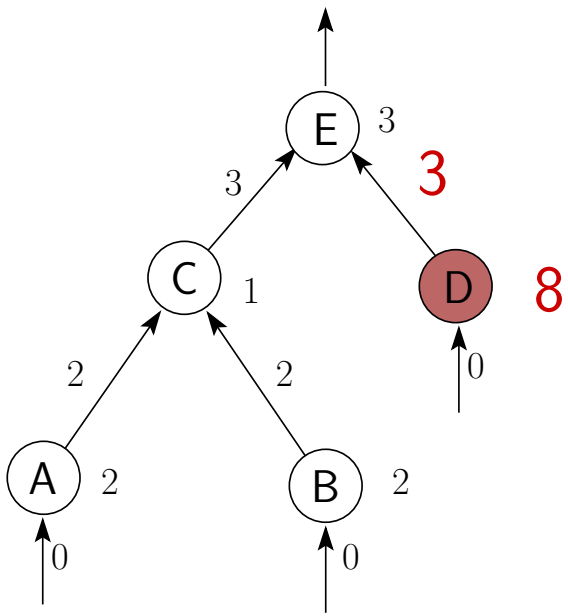
- Nodes: tasks
  - Node weight: temporary data (m_i)
- Edges: dependencies (data)
  - Edge weight: data size ($d_{i,j}$)
- Process a node:
  load inputs + output + temporary data
- Memory ($\rightarrow$): 4, 2, 6, 4, 8, 3, 14, 6, 9

Tree traversal influences peak memory



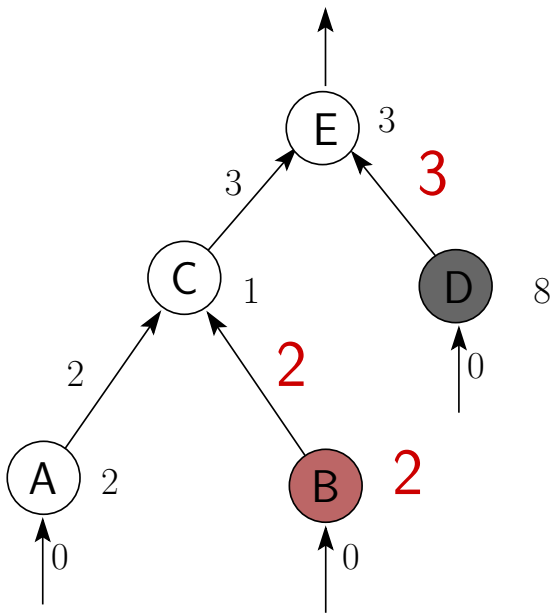
- Nodes: tasks
  - Node weight: temporary data (m_i)
- Edges: dependencies (data)
  - Edge weight: data size ($d_{i,j}$)
- Process a node:
  load inputs + output + temporary data
- Memory ($\rightarrow$): 4, 2, 6, 4, 8, 3, 14, 6, 9

Tree traversal influences peak memory



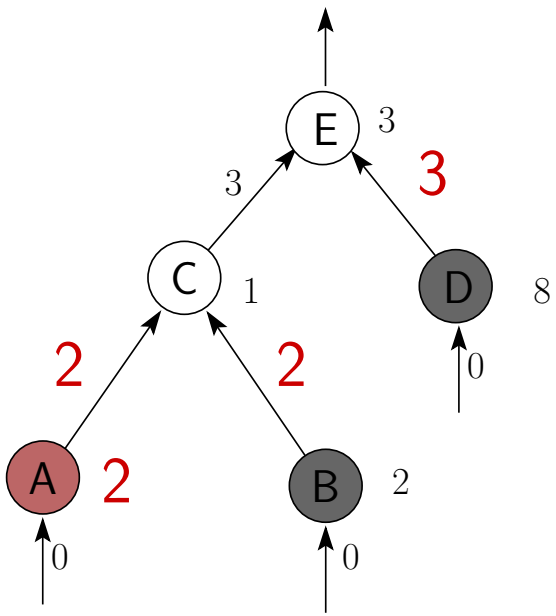
- Nodes: tasks
  - Node weight: temporary data (m_i)
- Edges: dependencies (data)
  - Edge weight: data size ($d_{i,j}$)
- Process a node:
  load inputs + output + temporary data
- Memory ($\rightarrow$): 4, 2, 6, 4, 8, 3, 14, 6, 9
- Memory ($\leftarrow$): 11

Tree traversal influences peak memory



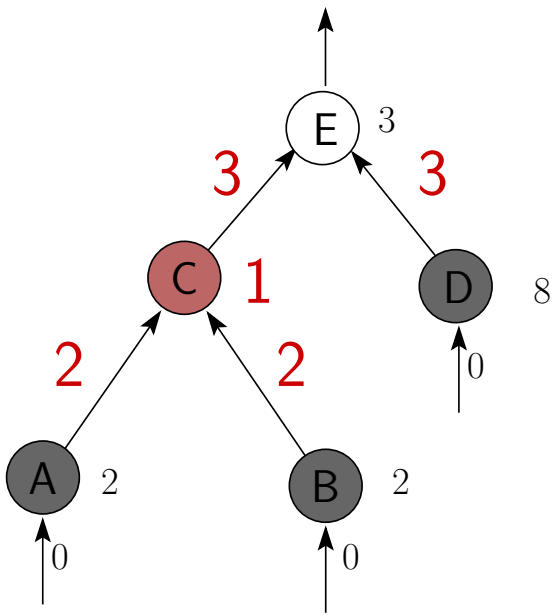
- Nodes: tasks
  - Node weight: temporary data (m_i)
- Edges: dependencies (data)
  - Edge weight: data size ($d_{i,j}$)
- Process a node:
  load inputs + output + temporary data
- Memory ($\rightarrow$): 4, 2, 6, 4, 8, 3, 14, 6, 9
- Memory ($\leftarrow$): 11, 7

Tree traversal influences peak memory



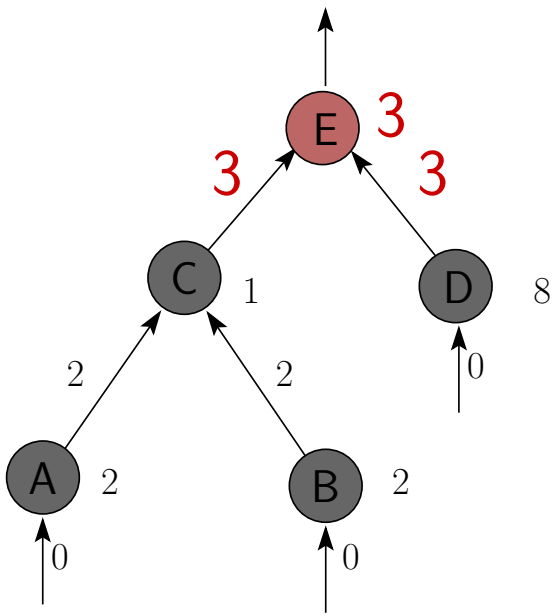
- Nodes: tasks
  - Node weight: temporary data (m_i)
- Edges: dependencies (data)
  - Edge weight: data size ($d_{i,j}$)
- Process a node:
  load inputs + output + temporary data
- Memory ($\rightarrow$): 4, 2, 6, 4, 8, 3, 14, 6, 9
- Memory ($\leftarrow$): 11, 7, 9

Tree traversal influences peak memory



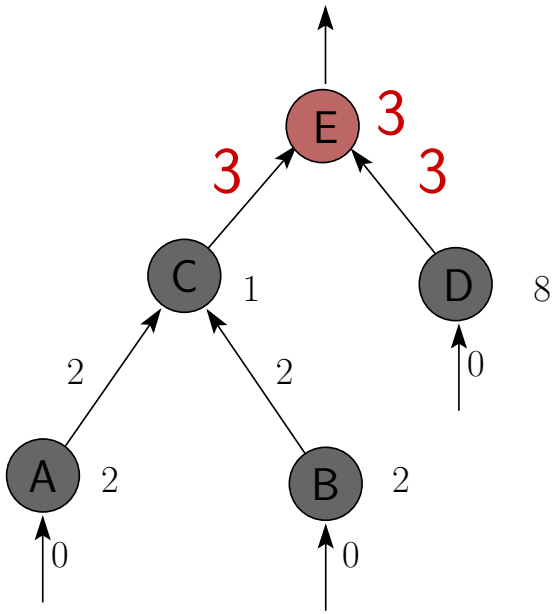
- Nodes: tasks
  - Node weight: temporary data (m_i)
- Edges: dependencies (data)
  - Edge weight: data size ($d_{i,j}$)
- Process a node:
  load inputs + output + temporary data
- Memory ($\rightarrow$): 4, 2, 6, 4, 8, 3, 14, 6, 9
- Memory ($\leftarrow$): 11, 7, 9, 11

Tree traversal influences peak memory



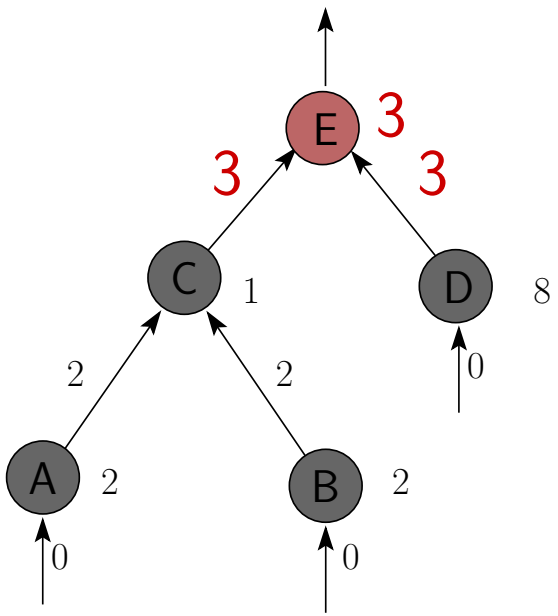
- Nodes: tasks
  - Node weight: temporary data (m_i)
- Edges: dependencies (data)
  - Edge weight: data size ($d_{i,j}$)
- Process a node:
  load inputs + output + temporary data
- Memory ($\rightarrow$): 4, 2, 6, 4, 8, 3, 14, 6, 9
- Memory ($\leftarrow$): 11, 7, 9, 11, 9

Tree traversal influences peak memory



- Nodes: tasks
  - Node weight: temporary data (m_i)
- Edges: dependencies (data)
  - Edge weight: data size ($d_{i,j}$)
- Process a node:
  load inputs + output + temporary data
- Memory ($\rightarrow$): 4, 2, 6, 4, 8, 3, 14, 6, 9
- Memory ($\leftarrow$): 11, 7, 9, 11, 9

Tree traversal influences peak memory



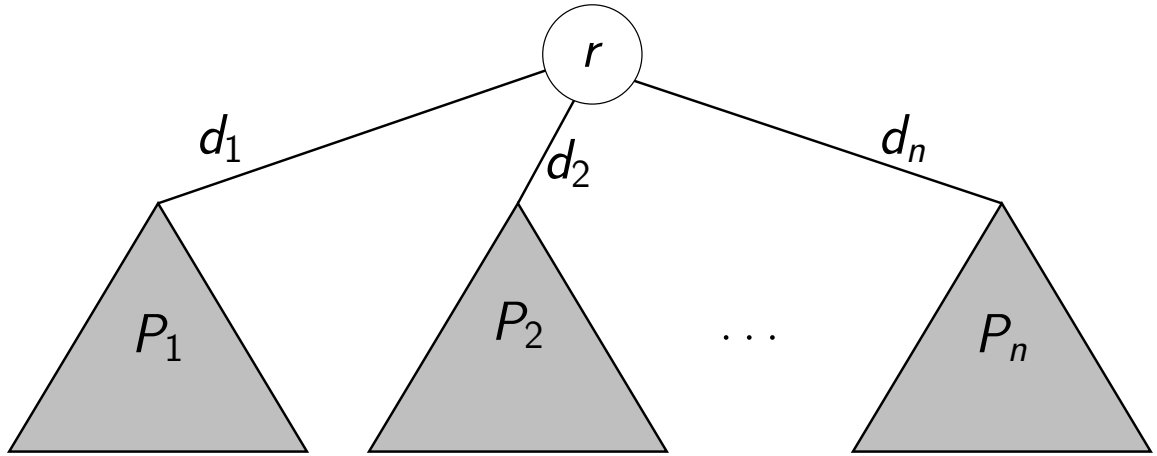
- Nodes: tasks
  - Node weight: temporary data (m_i)
- Edges: dependencies (data)
  - Edge weight: data size ($d_{i,j}$)
- Process a node:
  load inputs + output + temporary data
- Memory ($\rightarrow$): 4, 2, 6, 4, 8, 3, 14, 6, 9
- Memory ($\leftarrow$): 11, 7, 9, 11, 9

Focus on two problems:

- How to minimize the memory requirement of a tree?
  - Best post-order traversal
  - Optimal traversal
- Given an amount of available memory, how to efficiently process a tree?
  - Parallel processing
  - Goal: minimize processing time

Best post-order traversal for trees [Liu 86]

Post-Order: totally process a subtree before starting another one



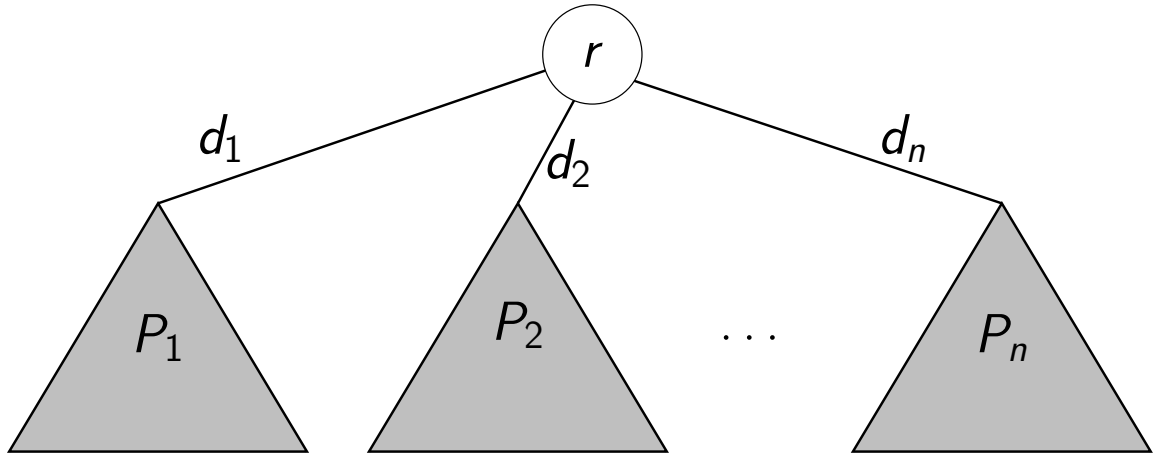
- For each subtree: peak memory P_i, residual memory d_i
- Given a processing order $1, \ldots, n$, the peak memory:

$$\max \left\{ P_1, \right.$$

$$\left. \right\}$$

Best post-order traversal for trees [Liu 86]

Post-Order: totally process a subtree before starting another one



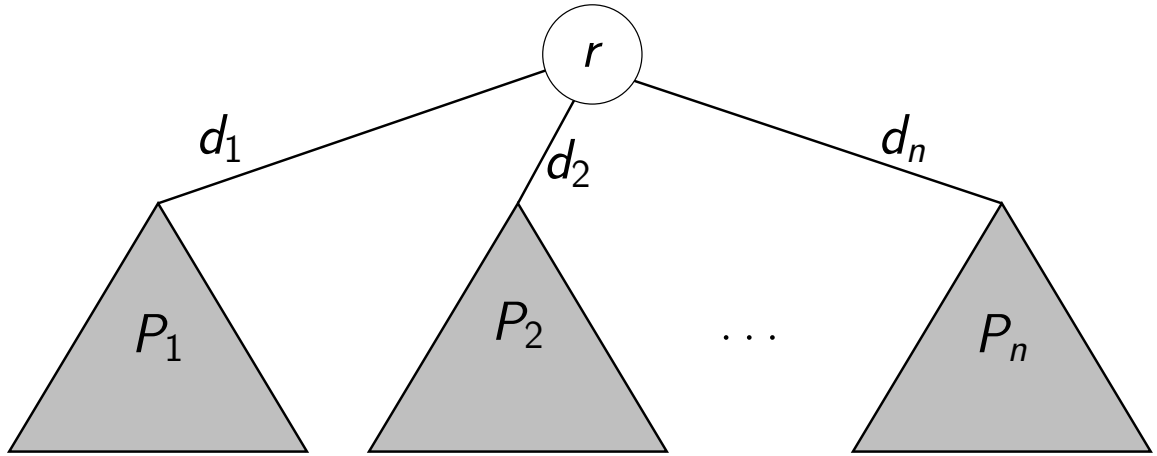
- For each subtree: peak memory P_i, residual memory d_i
- Given a processing order $1, \ldots, n$, the peak memory:

$$\max \left\{ P_1,\ d_1 + P_2, \right.$$

$$\left. \right\}$$

Best post-order traversal for trees [Liu 86]

Post-Order: totally process a subtree before starting another one

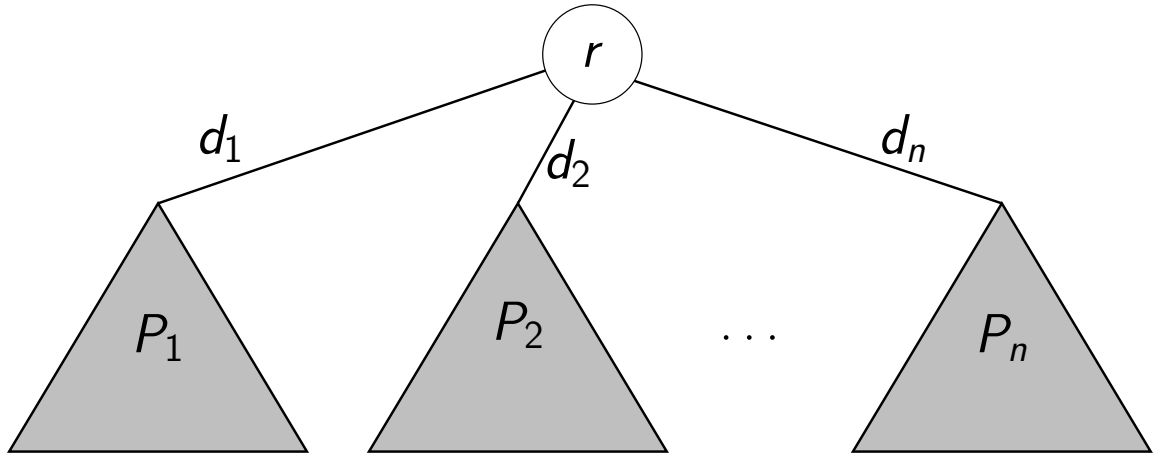


- For each subtree: peak memory P_i, residual memory d_i
- Given a processing order $1, \ldots, n$, the peak memory:

$$\max \left\{ P_1,\ d_1 + P_2,\ d_1 + d_2 + P_3, \right.$$

Best post-order traversal for trees [Liu 86]

Post-Order: totally process a subtree before starting another one

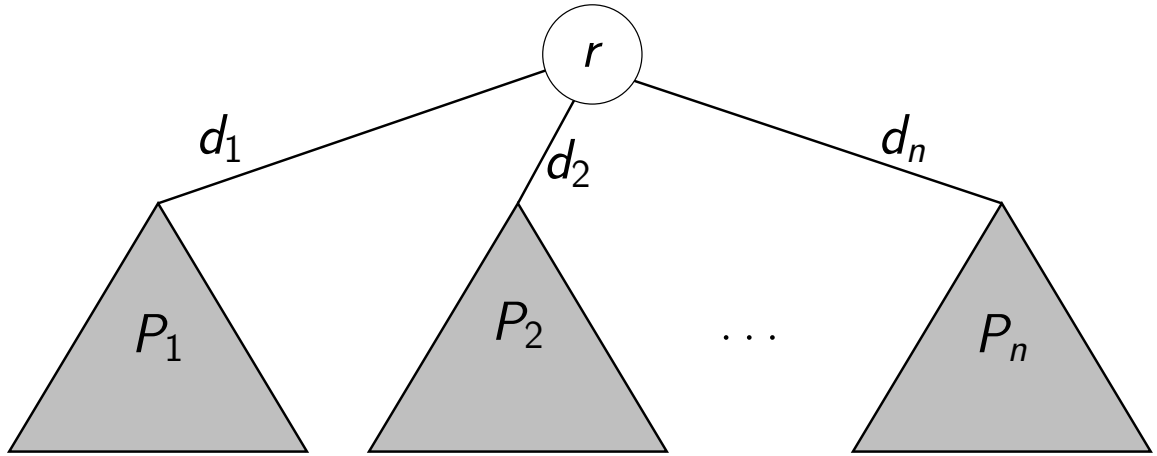


- For each subtree: peak memory P_i, residual memory d_i
- Given a processing order $1, \ldots, n$, the peak memory:

$$\max \left\{ P_1,\ d_1 + P_2,\ d_1 + d_2 + P_3,\ \ldots, \sum_{i<n} d_i + P_n, \right\}$$

Best post-order traversal for trees [Liu 86]

Post-Order: totally process a subtree before starting another one

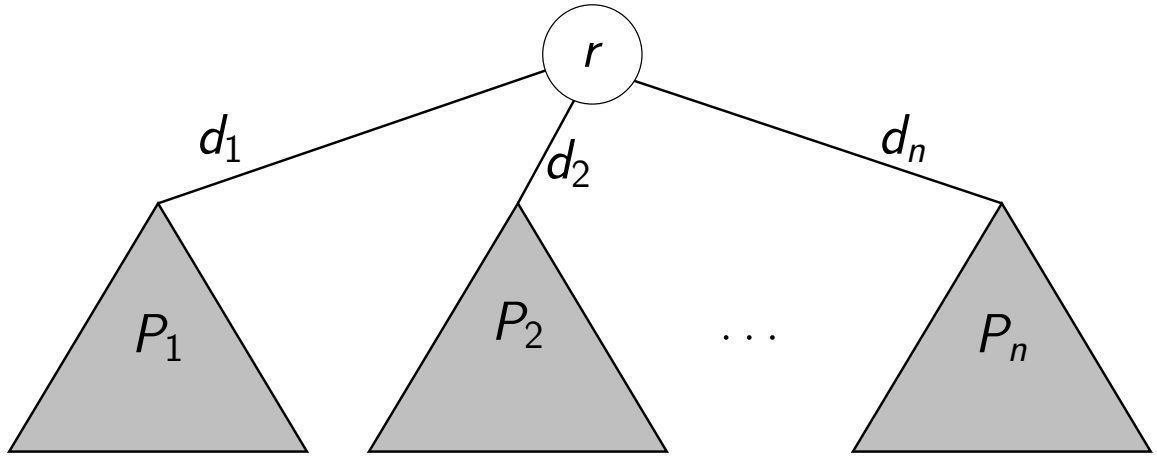


- For each subtree: peak memory P_i, residual memory d_i
- Given a processing order $1, \ldots, n$, the peak memory:

$$\max \left\{ P_1,\ d_1 + P_2,\ d_1 + d_2 + P_3,\ \ldots, \sum_{i<n} d_i + P_n,\ \sum_{i \leq n} d_i + m_i + d_r \right\}$$

Best post-order traversal for trees [Liu 86]

Post-Order: totally process a subtree before starting another one



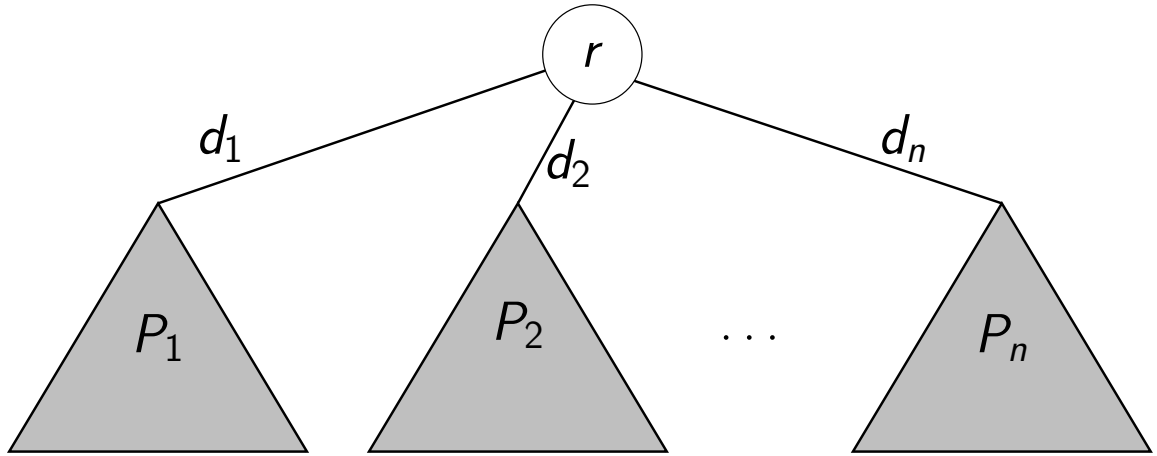
- For each subtree: peak memory P_i, residual memory d_i
- Given a processing order $1, \ldots, n$, the peak memory:

$$\max \left\{ P_1,\ d_1 + P_2,\ d_1 + d_2 + P_3,\ \ldots, \sum_{i<n} d_i + P_n,\ \sum_{i \leq n} d_i + m_i + d_r \right\}$$

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

Best post-order traversal for trees [Liu 86]

Post-Order: totally process a subtree before starting another one



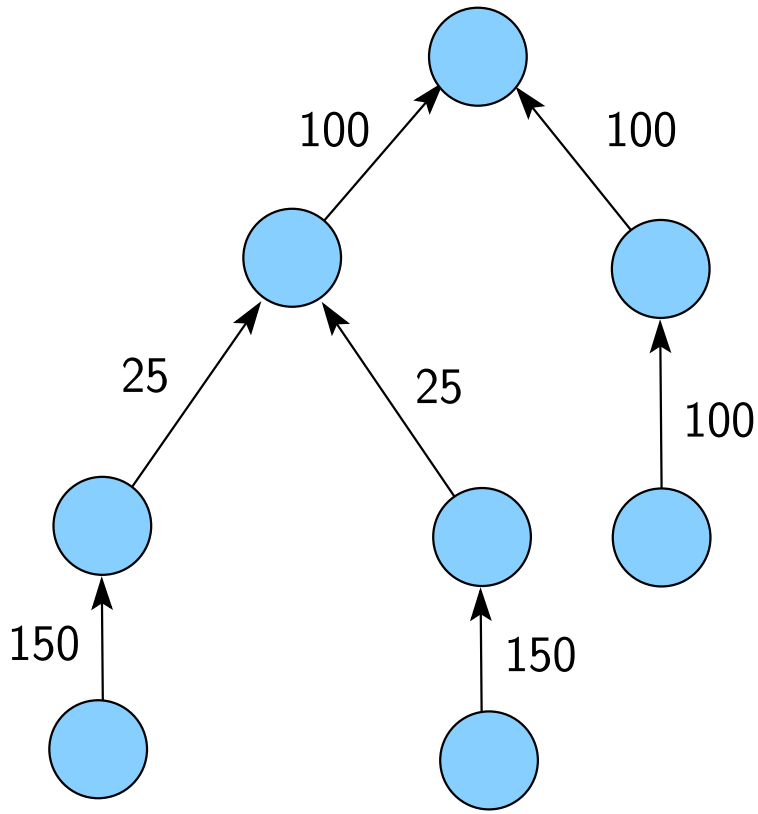
- For each subtree: peak memory P_i, residual memory d_i
- Given a processing order $1, \ldots, n$, the peak memory:

$$\max \left\{ P_1,\ d_1 + P_2,\ d_1 + d_2 + P_3,\ \ldots, \sum_{i<n} d_i + P_n,\ \sum_{i \leq n} d_i + m_i + d_r \right\}$$

- Optimal order: non-increasing $P_i - d_i$
- Best post-order traversal is optimal for unit-weight trees

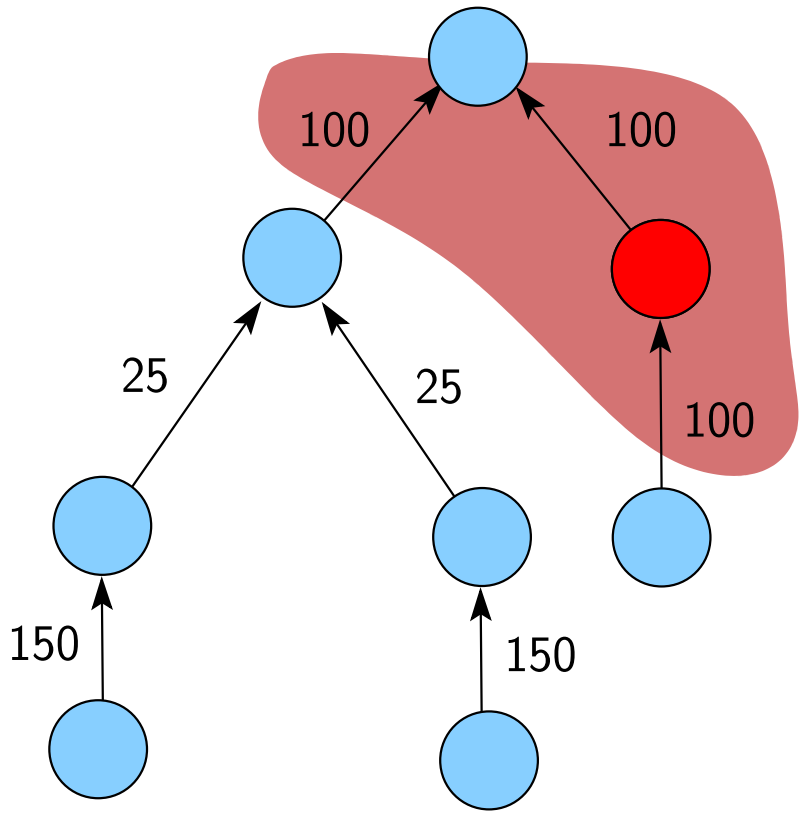
Post-Order vs. optimal traversals

- In some cases, interesting to stop within a subtree (if there exists a cut with small weight)



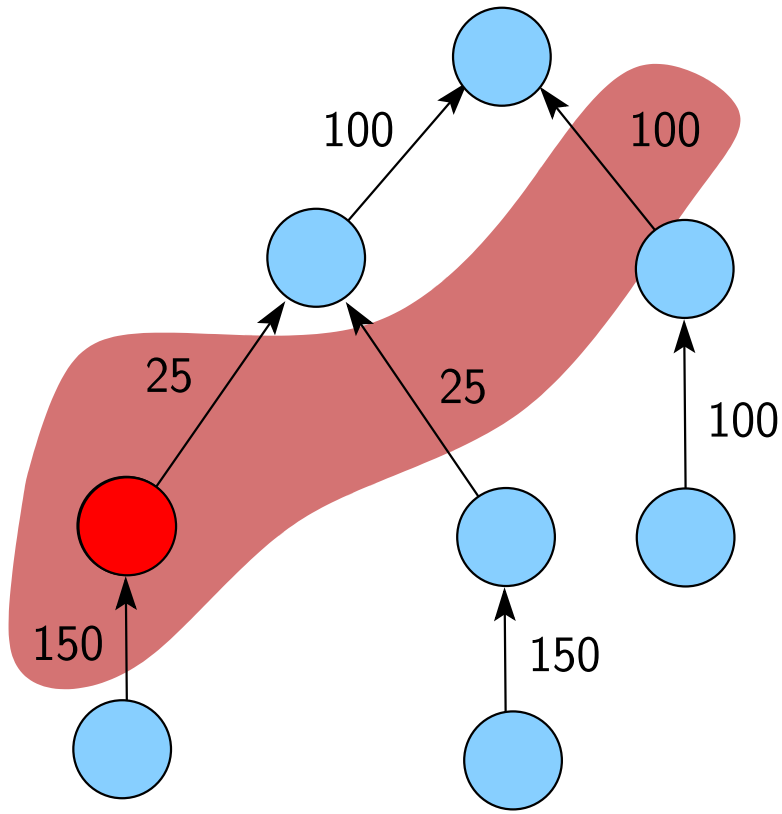
Post-Order vs. optimal traversals

- In some cases, interesting to stop within a subtree (if there exists a cut with small weight)



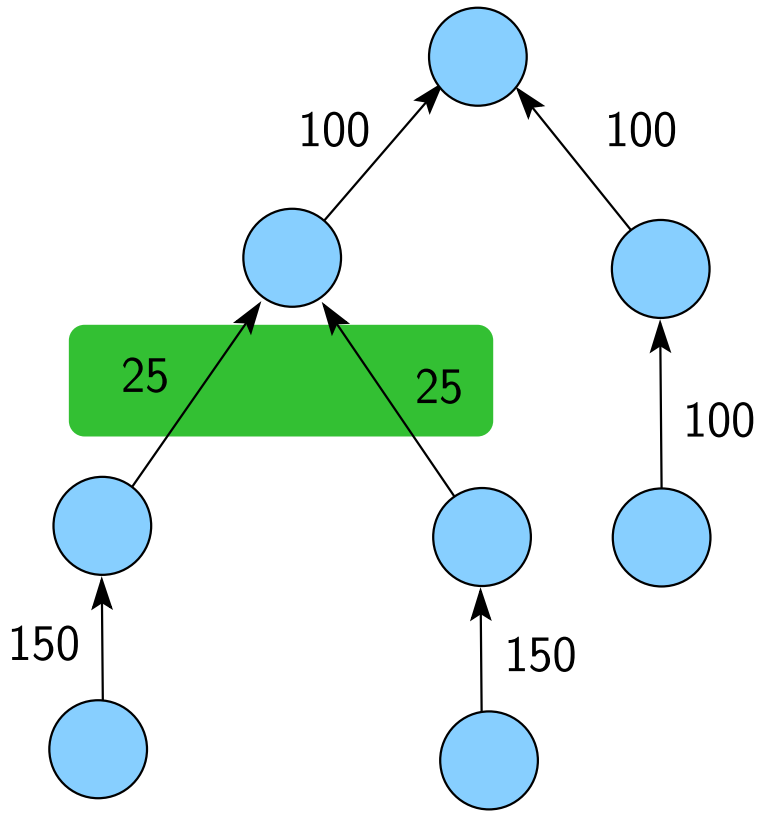
Post-Order vs. optimal traversals

- In some cases, interesting to stop within a subtree (if there exists a cut with small weight)



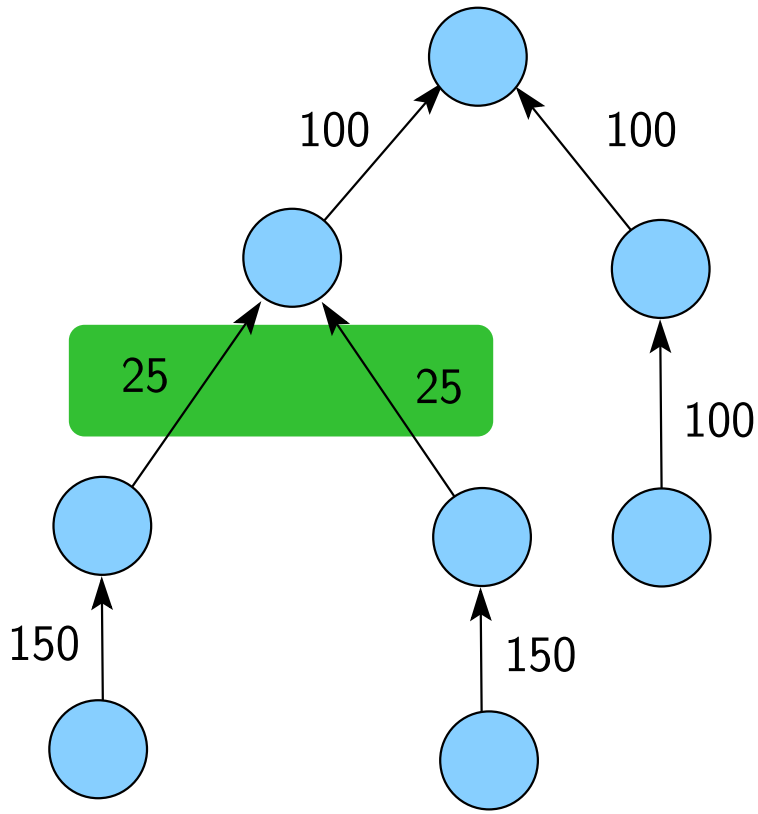
Post-Order vs. optimal traversals

- In some cases, interesting to stop within a subtree (if there exists a cut with small weight)



Post-Order vs. optimal traversals

- In some cases, interesting to stop within a subtree (if there exists a cut with small weight)
- For any K, possible to build a tree such that post-order uses K times as much memory as the optimal traversal



Post-Order vs. optimal traversals

- In some cases, interesting to stop within a subtree (if there exists a cut with small weight)
- For any K, possible to build a tree such that post-order uses K times as much memory as the optimal traversal

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

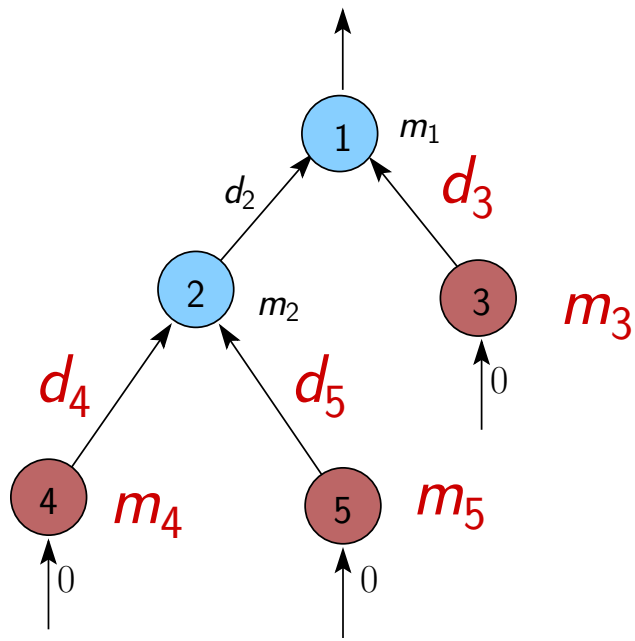
Optimal algorithms:

- First algorithm proposed by [Liu 87]
  Complex mutli-way merge, $O(n^2)$
- New algorithm
  Recursive exploration of the tree, $O(n^2)$, faster in practice

[M. Jacquelin, L. Marchal, Y. Robert & B. Uçar, IPDPS 2011]

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: bi-objective problem
  - Peak memory
  - Makespan (total processing time)



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: $m_i = 0, d_i = 1$
  (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.

[L. Eyraud-Dubois, L. Marchal, O. Sinnen, F. Vivien, TOPC 2015]

Space-Time tradeoff

No 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.

Theorem 2

For any $\alpha(p)$-approximation for makespan and $\beta(p)$-approximation for memory peak with $p \geq 2$ processors,

$$\alpha(p)\beta(p) \geq \frac{2p}{\lceil \log(p) \rceil + 2}.$$

How to cope with limited memory?

- When processing a tree on a given machine: bounded memory
- Objective: Minimize processing time under this constraint
- NB: bounded memory $\geq$ memory for sequential processing

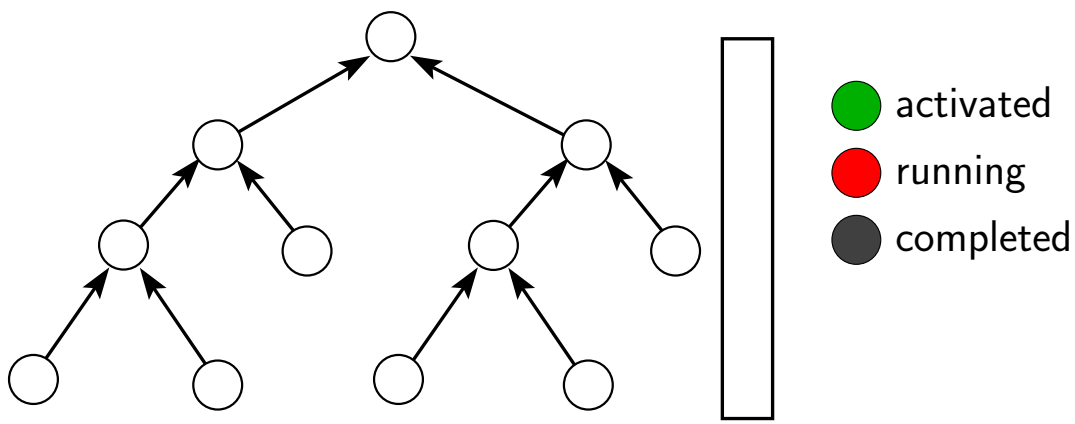
- Intuition:
  - When data sizes $\ll$ memory bound:
    process many tasks in parallel
  - When approaching memory bound, limit parallelism
- Rely on a (memory-friendly) sequential traversal

Existing (system) approach:

- Book memory as in sequential processing

Conservative approach: task activation

- From [Agullo, Buttari, Guermouche & Lopez 2013]
- Choose a sequential task order (e.g. best post-order)
- While memory available, activate tasks in this order:
  book memory for their output + tmp. data
- Process only activated tasks (with given scheduling priority)

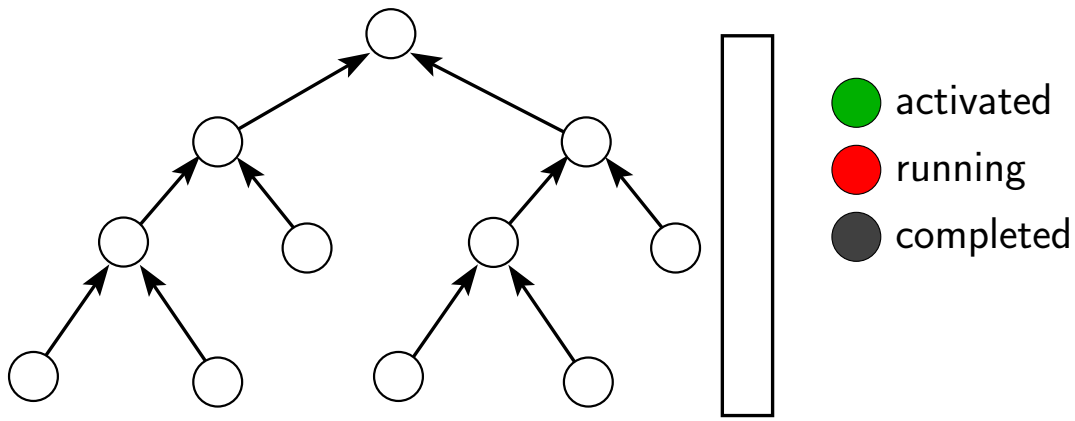


Conservative approach: task activation

- From [Agullo, Buttari, Guermouche & Lopez 2013]
- Choose a sequential task order (e.g. best post-order)
- While memory available, activate tasks in this order:
  book memory for their output + tmp. data
- Process only activated tasks (with given scheduling priority)

When a tasks completes:

- Free inputs
- Activate as many new tasks as possible
- Then, start scheduling activated tasks

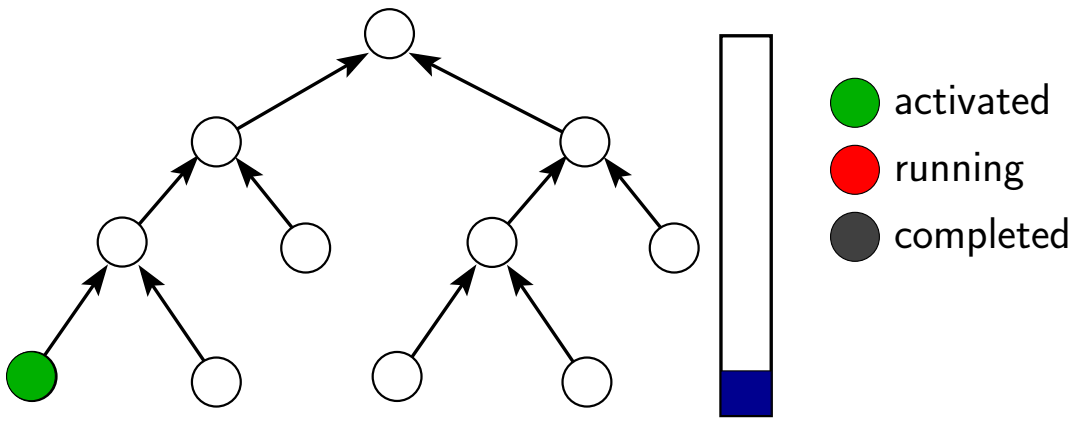


Conservative approach: task activation

- From [Agullo, Buttari, Guermouche & Lopez 2013]
- Choose a sequential task order (e.g. best post-order)
- While memory available, activate tasks in this order:
  book memory for their output + tmp. data
- Process only activated tasks (with given scheduling priority)

When a tasks completes:

- Free inputs
- Activate as many new tasks as possible
- Then, start scheduling activated tasks

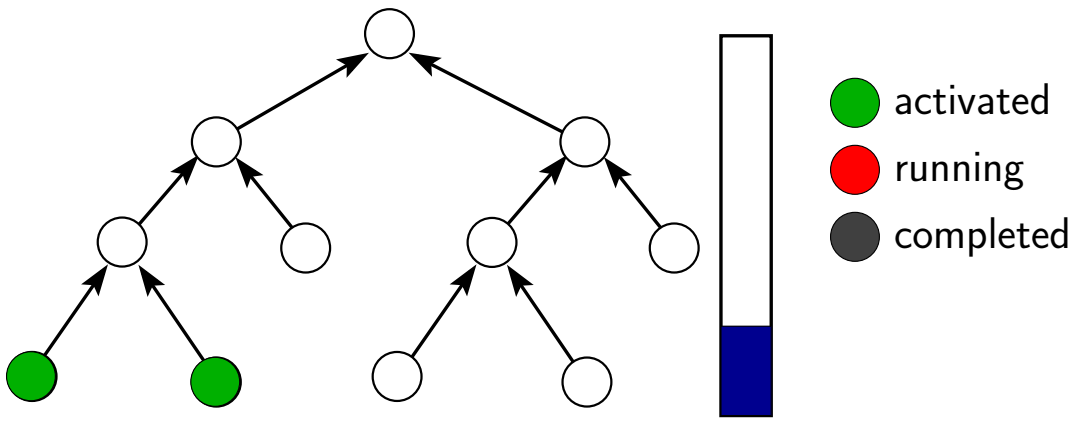


Conservative approach: task activation

- From [Agullo, Buttari, Guermouche & Lopez 2013]
- Choose a sequential task order (e.g. best post-order)
- While memory available, activate tasks in this order:
  book memory for their output + tmp. data
- Process only activated tasks (with given scheduling priority)

When a tasks completes:

- Free inputs
- Activate as many new tasks as possible
- Then, start scheduling activated tasks

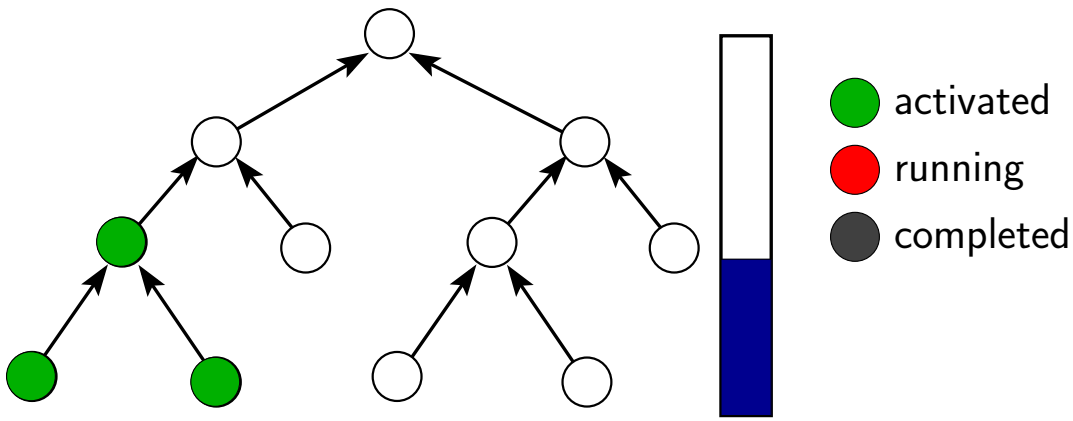


Conservative approach: task activation

- From [Agullo, Buttari, Guermouche & Lopez 2013]
- Choose a sequential task order (e.g. best post-order)
- While memory available, activate tasks in this order:
  book memory for their output + tmp. data
- Process only activated tasks (with given scheduling priority)

When a tasks completes:

- Free inputs
- Activate as many new tasks as possible
- Then, start scheduling activated tasks

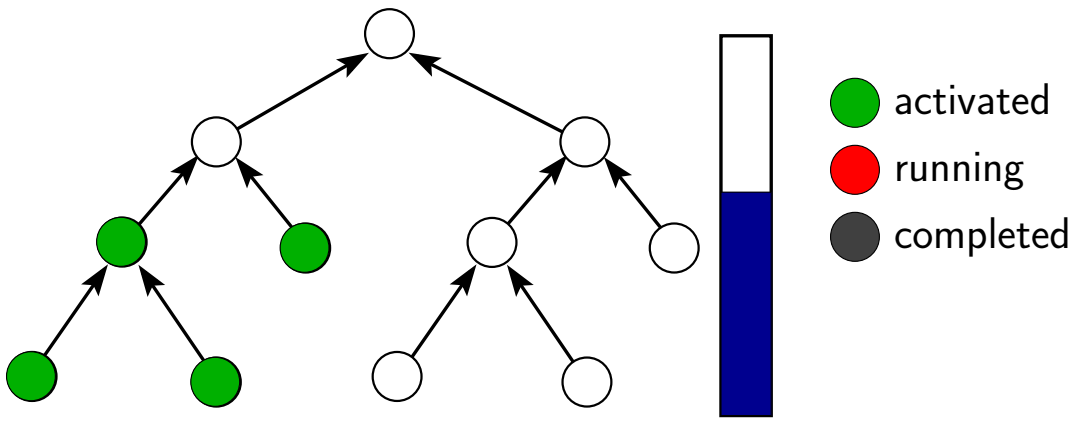


Conservative approach: task activation

- From [Agullo, Buttari, Guermouche & Lopez 2013]
- Choose a sequential task order (e.g. best post-order)
- While memory available, activate tasks in this order:
  book memory for their output + tmp. data
- Process only activated tasks (with given scheduling priority)

When a tasks completes:

- Free inputs
- Activate as many new tasks as possible
- Then, start scheduling activated tasks

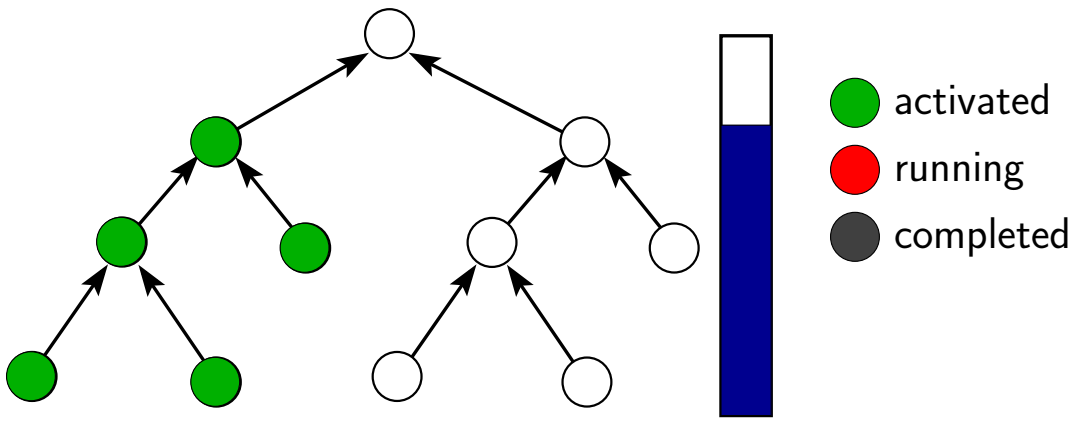


Conservative approach: task activation

- From [Agullo, Buttari, Guermouche & Lopez 2013]
- Choose a sequential task order (e.g. best post-order)
- While memory available, activate tasks in this order:
  book memory for their output + tmp. data
- Process only activated tasks (with given scheduling priority)

When a tasks completes:

- Free inputs
- Activate as many new tasks as possible
- Then, start scheduling activated tasks

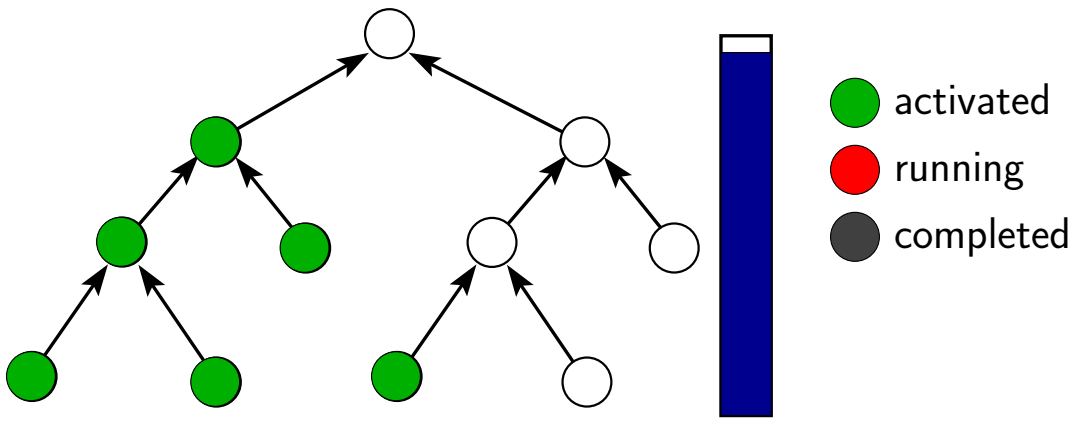


Conservative approach: task activation

- From [Agullo, Buttari, Guermouche & Lopez 2013]
- Choose a sequential task order (e.g. best post-order)
- While memory available, activate tasks in this order:
  book memory for their output + tmp. data
- Process only activated tasks (with given scheduling priority)

When a tasks completes:

- Free inputs
- Activate as many new tasks as possible
- Then, start scheduling activated tasks

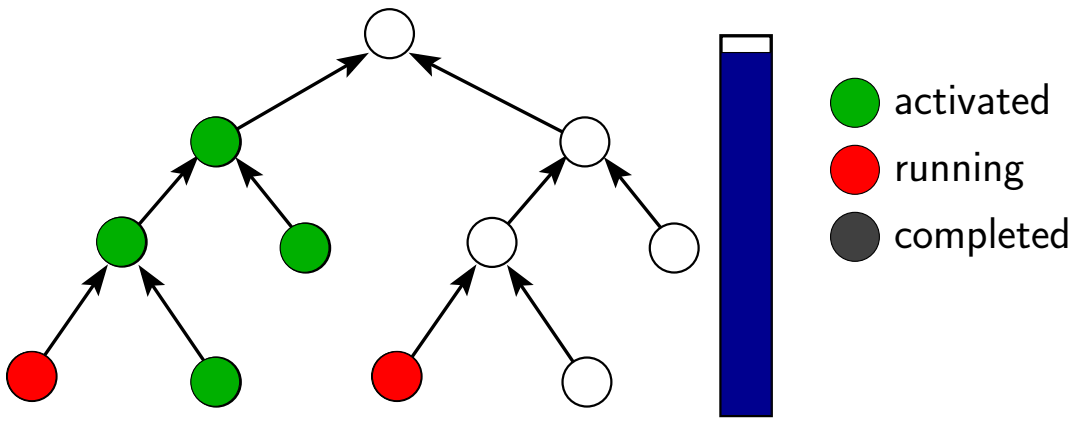


Conservative approach: task activation

- From [Agullo, Buttari, Guermouche & Lopez 2013]
- Choose a sequential task order (e.g. best post-order)
- While memory available, activate tasks in this order:
  book memory for their output + tmp. data
- Process only activated tasks (with given scheduling priority)

When a tasks completes:

- Free inputs
- Activate as many new tasks as possible
- Then, start scheduling activated tasks

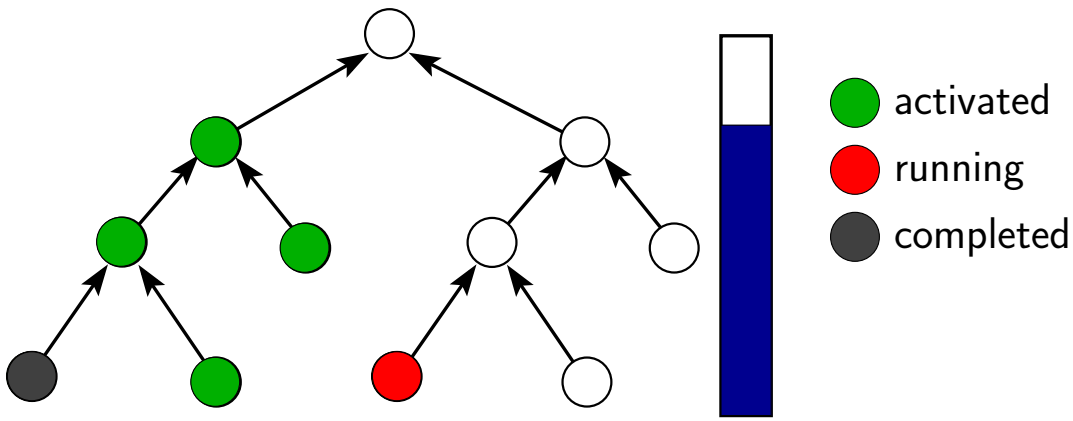


Conservative approach: task activation

- From [Agullo, Buttari, Guermouche & Lopez 2013]
- Choose a sequential task order (e.g. best post-order)
- While memory available, activate tasks in this order:
  book memory for their output + tmp. data
- Process only activated tasks (with given scheduling priority)

When a tasks completes:

- Free inputs
- Activate as many new tasks as possible
- Then, start scheduling activated tasks

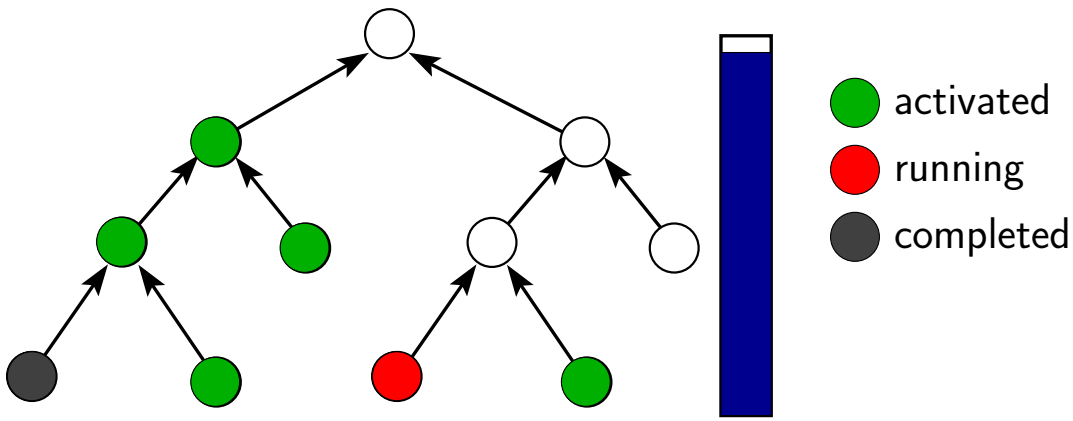


Conservative approach: task activation

- From [Agullo, Buttari, Guermouche & Lopez 2013]
- Choose a sequential task order (e.g. best post-order)
- While memory available, activate tasks in this order:
  book memory for their output + tmp. data
- Process only activated tasks (with given scheduling priority)

When a tasks completes:

- Free inputs
- Activate as many new tasks as possible
- Then, start scheduling activated tasks

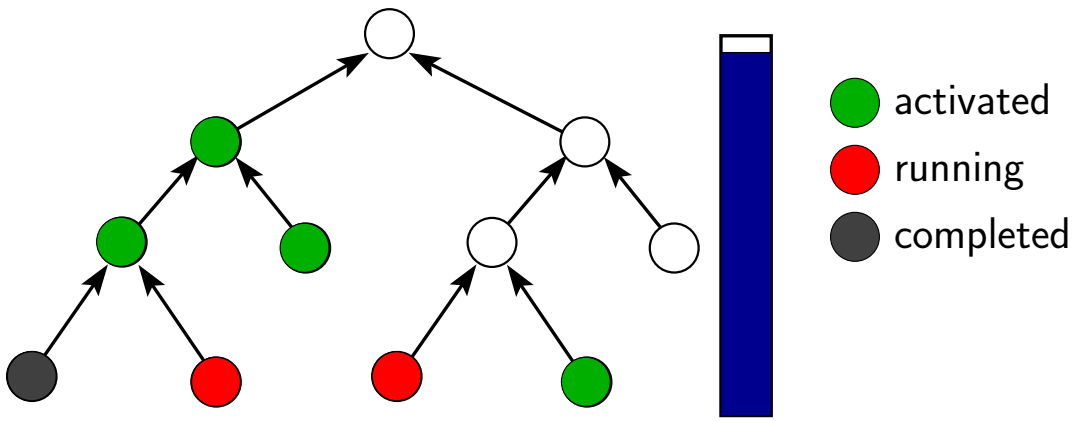


Conservative approach: task activation

- From [Agullo, Buttari, Guermouche & Lopez 2013]
- Choose a sequential task order (e.g. best post-order)
- While memory available, activate tasks in this order:
  book memory for their output + tmp. data
- Process only activated tasks (with given scheduling priority)

When a tasks completes:

- Free inputs
- Activate as many new tasks as possible
- Then, start scheduling activated tasks

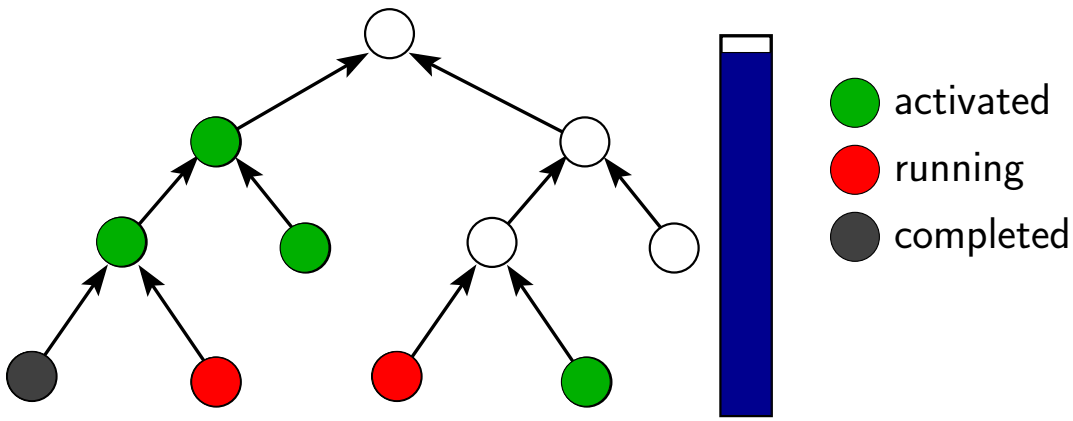


Conservative approach: task activation

- From [Agullo, Buttari, Guermouche & Lopez 2013]
- Choose a sequential task order (e.g. best post-order)
- While memory available, activate tasks in this order:
  book memory for their output + tmp. data
- Process only activated tasks (with given scheduling priority)

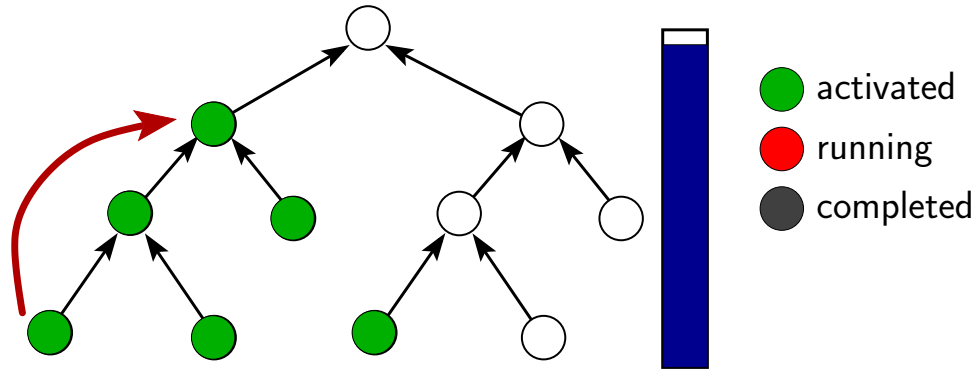
When a tasks completes:

- Free inputs
- Activate as many new tasks as possible
- Then, start scheduling activated tasks



- ☺ Can cope with very small memory bound
- ☹ No memory reuse

Refined activation: predict memory reuse



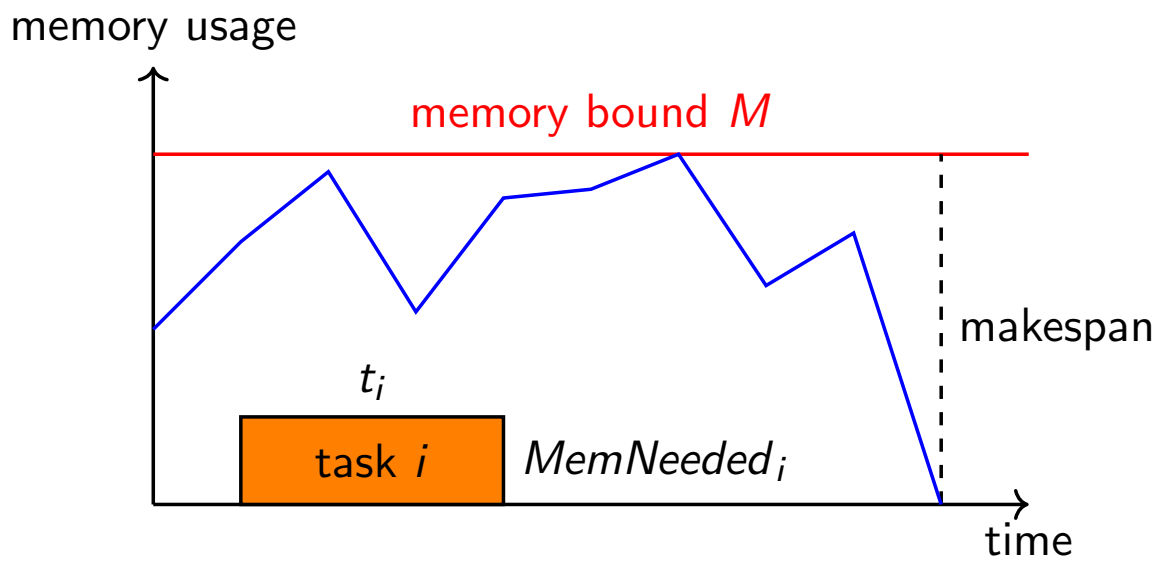
- Follow the same activation approach
- When activating a node:
  - Check how much memory is already booked by its subtree
  - Book only what is missing (if needed)
- When completing a node:
  - Distribute the booked memory to all activated ancestors
  - Then, release the remaining memory (if any)
- Proof of termination
  - Based on a sequential schedule using less than the memory bound
  - Process the whole tree without going out of memory

New makespan lower bound

Theorem (Memory aware makespan lower bound).

$$C_{\max} \geq \frac{1}{M} \sum_i MemNeeded_i \times t_i.$$

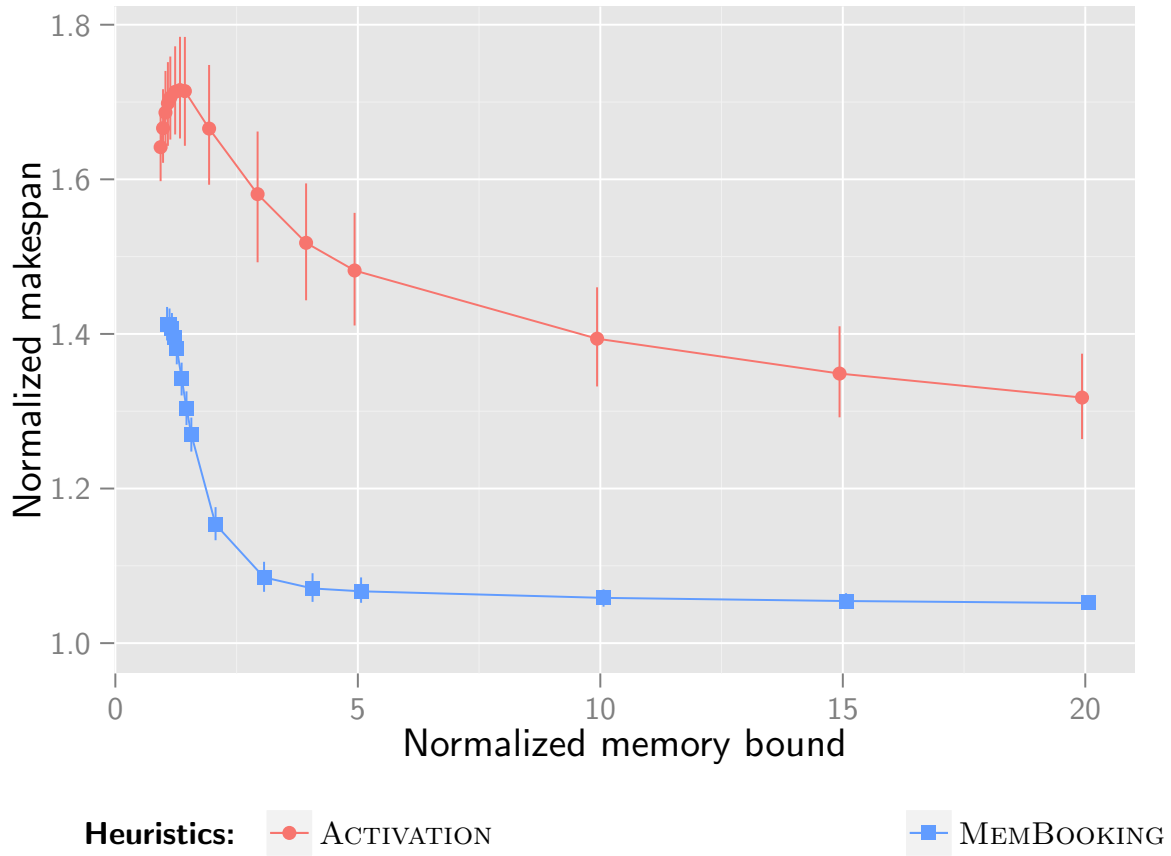
- M: memory bound
- $C_{\max}$: makespan (total processing time)
- $MemNeeded_i$: memory needed to process task i
- t_i: processing time of task i.



Simulation on assembly trees

- Dataset: assembly trees of actual sparse matrices
- Algorithms:
  - ACTIVATION from [Agullo et al, Europar 2013]
  - MEMBOOKING
- Sequential tasks (simple performance model)
- 8 processors (similar results for 2,4,16 and 32)
- Reference memory M_{PO} :
  peak memory of best sequential post-order
- Activation and execution orders: best seq. post-order
- Makespan normalized by $\max(CP, \frac{W}{p}, MemAwareLB)$

Simulations: total processing time



- MemBooking able to activate more nodes, increase parallelism
- Even for scarce memory conditions

[G. Aupy, C. Brasseur, L. Marchal, IPDPS 2017]

Conclusion on memory-aware tree scheduling

Summary:

- Related to pebble games
- Well-known sequential algorithms for trees
- Parallel processing difficult:
  - Complexity and inapproximability
  - Efficient booking heuristics (guaranteed termination)

Other contributions in this area:

- Optimal sequential algorithm for SP-graphs
- Complexity and heuristics for two types of cores (hybrid)
- I/O volume minimization: optimal sequential algorithm for homogeneous trees
- Guaranteed heuristic for memory-bounded parallel scheduling of DAGs

Outline

Introduction

Distributed computing:

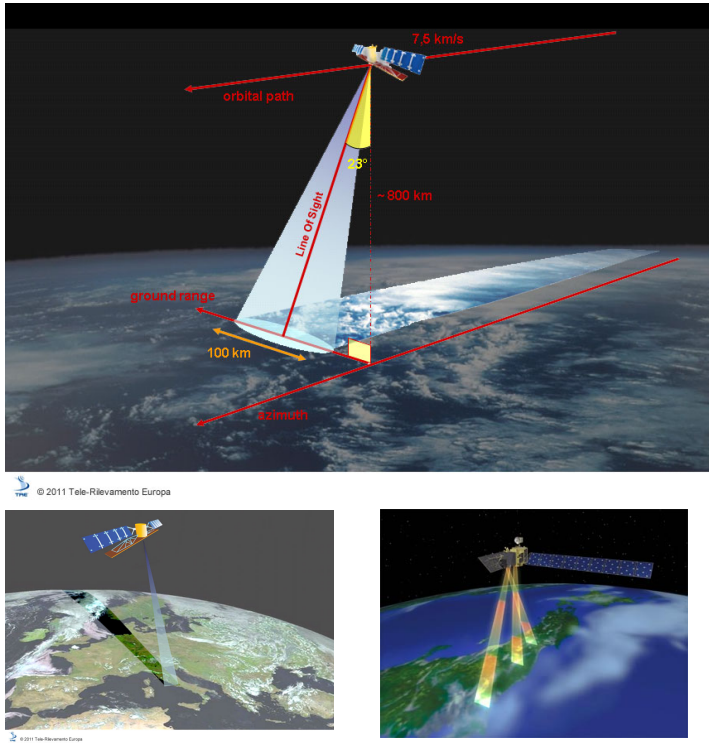
- Processors have their own memory
- Data transfers are needed, but costly (time, energy)
- Computing speed increases faster than network bandwidth
- Need for limiting these communications

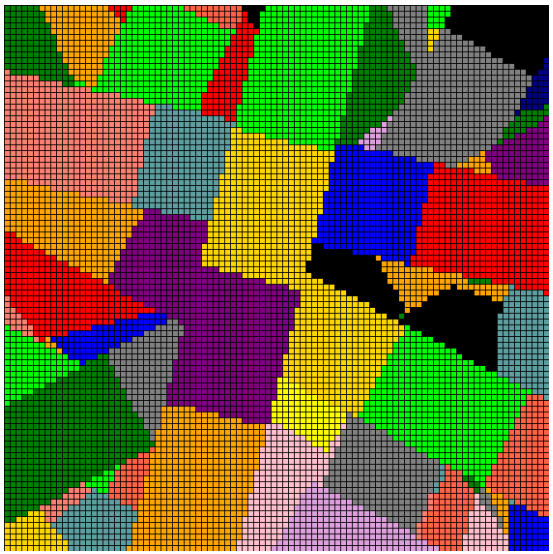
Following study:

- Data is originally (ill) distributed
- Computation to be performed has a preferred data layout
- Should we redistribute the data? How?

Data collection and storage

- Origin of data: sensors (e.g. satellites) that aggregate snapshots
- Data partitioned and distributed before the computation
  - During the collection
  - By a previous computation

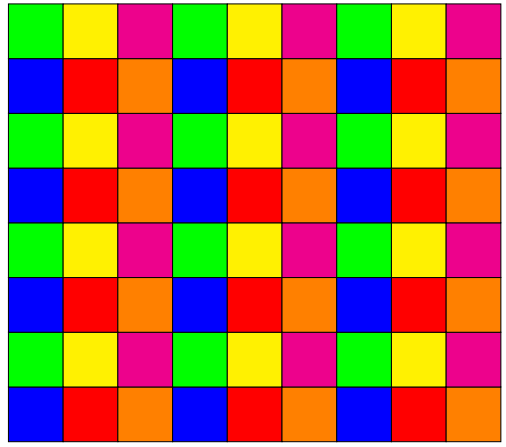




- Computation kernel (e.g. linear algebra kernels) must be applied to data
- Initial data distribution may be inefficient for the computation kernel

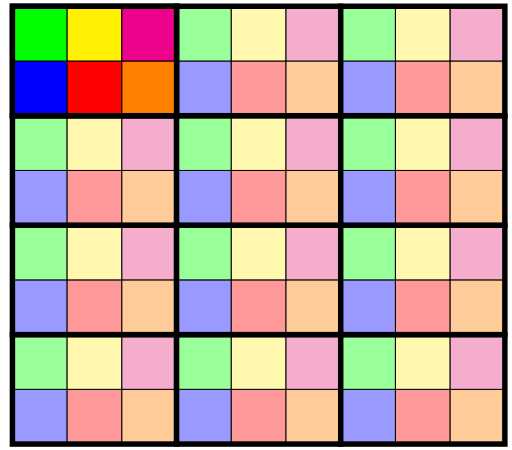
Data distribution and mapping

- A data distribution is usually defined to minimize the completion time of an algorithm
  - Ex: 2D-cyclic
- There is not necessarily a single data distribution that maximizes this efficiency
- Find the one-to-one mapping (subsets of data - processors) for which the cost of the redistribution is minimal



Data distribution and mapping

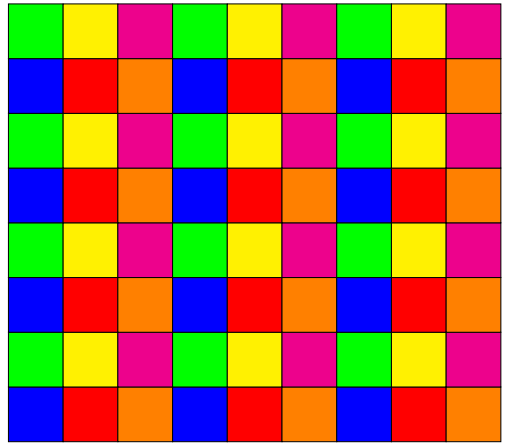
- A data distribution is usually defined to minimize the completion time of an algorithm
  - Ex: 2D-cyclic

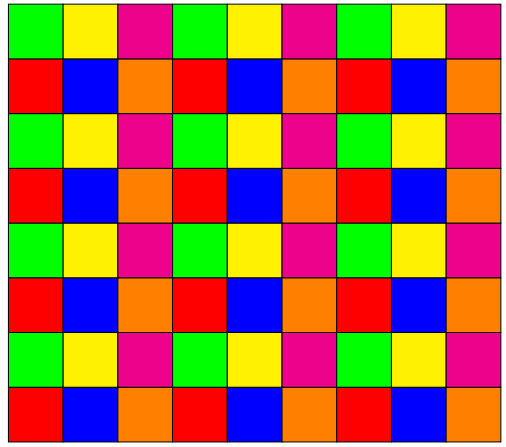


- There is not necessarily a single data distribution that maximizes this efficiency
- Find the one-to-one mapping (subsets of data - processors) for which the cost of the redistribution is minimal

Data distribution and mapping

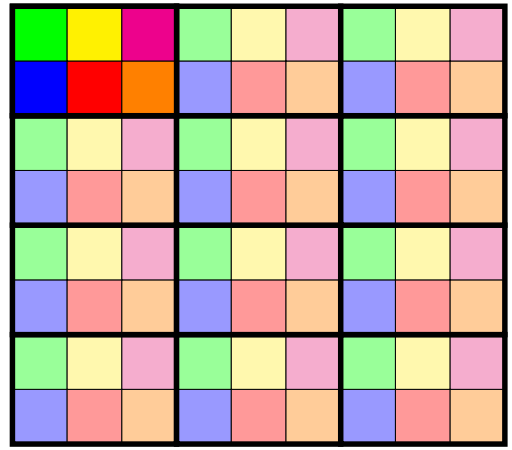
- A data distribution is usually defined to minimize the completion time of an algorithm
  - Ex: 2D-cyclic
- There is not necessarily a single data distribution that maximizes this efficiency
- Find the one-to-one mapping (subsets of data - processors) for which the cost of the redistribution is minimal

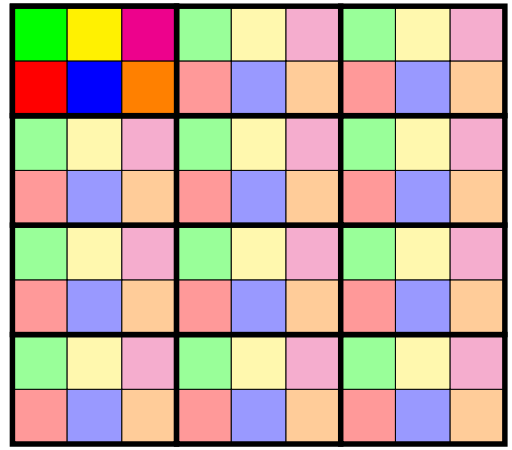




Data distribution and mapping

- A data distribution is usually defined to minimize the completion time of an algorithm
  - Ex: 2D-cyclic
- There is not necessarily a single data distribution that maximizes this efficiency
- Find the one-to-one mapping (subsets of data - processors) for which the cost of the redistribution is minimal





Data distribution / Data partition

- Let P be a finite set of identical processors
- Let A be a finite set of data items
- Data Distribution: $\mathcal{D} : A \mapsto P$
  $\forall a \in A, \mathcal{D}(a) = p \Leftrightarrow$ a hosted on proc p
- Data Partition: $\mathcal{P} : A \mapsto P$
  $\forall a, b \in A, \mathcal{P}(a) = \mathcal{P}(b) \Leftrightarrow$ a and b are hosted by the same processor
- A data distribution $\mathcal{D}$ is compatible with the data partition $\mathcal{P}$ iif there exists a permutation σ such that
  $\forall a \in A, \mathcal{D}(a) = \sigma(\mathcal{P}(a))$

Cost of redistribution

- Hardware symmetry assumption: the efficiency of the computation algorithm is a function of the data partition
- Unitary size assumption: all data items are of the same size
- Evaluation of the redistribution with two metrics:
  - Total volume of communication: the total number of data items sent from one processor to another
  - Number of parallel communication steps: one-port bi-directional model

Best redistribution to given partition

- For many algorithms, we know ideal data partitions that minimize completion time
- There are $|P|!$ data distributions compatible with the ideal partition

Best redistribution problem

Given an initial data distribution $\mathcal{D}_{ini}$, find the target data distribution $\mathcal{D}_{tar}$ compatible with the ideal data partition that minimizes the cost of redistribution.

- Optimal algorithms for each metric
- Based on building bipartite graphs and computing perfect matching

Redistribution followed by computation kernel

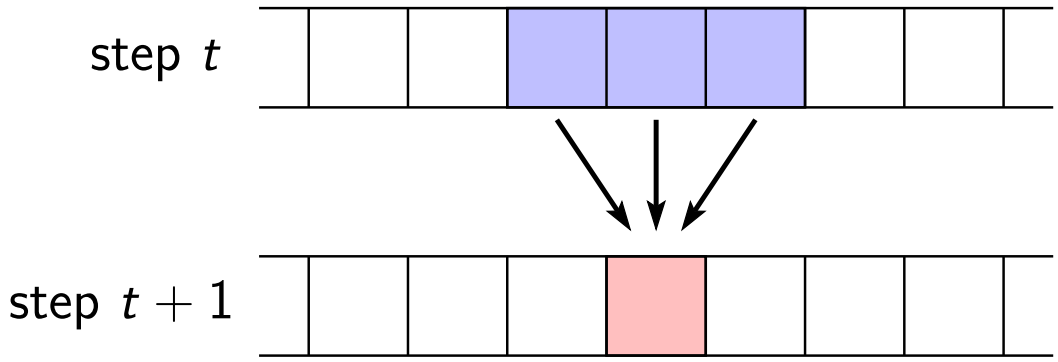
- Non-overlapping phases assumption:

$$T_{\text{tot}} = T_{\text{redist}}(D_{\text{ini}} \rightarrow D_{\text{tar}}) + T_{\text{comp}}(D_{\text{tar}})$$

- Close formula for $T_{\text{redist}}(D_{\text{ini}} \rightarrow D_{\text{tar}})$ depending on the communication model
- No close formula for $T_{\text{comp}}(D_{\text{tar}})$ in the general case

NP-completeness for 1D Stencil

- Consider the simple case of iterative 1D Stencil



- Simple close formula for $T_{\text{comp}}^{\text{stencil}}(D_{\text{tar}})$ for both communication models

Theorem

Finding the optimal distribution $\mathcal{D}_{tar}$ that minimizes

$$T_{\text{tot}} = T_{\text{redist}}(D_{\text{ini}} \to D_{\text{tar}}) + T_{\text{comp}}^{\text{stencil}}(D_{\text{tar}})$$

is NP-complete in the strong sense.

Heuristics for redistribution + computation

Naïve options:

- Do not redistribute (owner-compute)
- Canonical redistribution to target partition:
  - Processor i gets part i

Using previous algorithms:

- Compute best redistribution for each metric
  - Total volume (vol)
  - Redistribution steps (steps)

Experimental set up

- Implementation with the ParSEC runtime
- Initial distribution: random balanced distributions
- Targeted partition $\mathcal{P}_{tar}$: optimal partition for the considered computation kernel (QR: 2D-block cyclic)
- Parsec moves data from initial distribution to the target compute location when needed (computation/communication overlap)
- Target distribution computed according to four heuristics:
  - Owner compute (default heuristics of Parsec)
  - Canonical redistribution to $\mathcal{P}_{tar}$
  - Best redistribution for total volume (vol)
  - Best redistribution for number of steps (steps)

Results on QR factorization

- Improvements in total completion time (redistribution + computation)
- Compared to Owner compute (no redistribution)
- Average on 50 matrices

n	canonical	Vol. algo.	Steps algo.
16	41.9%	39.5%	43.4%
34	64.1%	67.7%	66.4%
52	65.8%	70.5%	71.2%
70	70.8%	72.7%	71.4%
88	70.8%	72.6%	72.4%

Results on skewed distribution (2D-block cyclic + 50% tiles randomly moved)

n	canonical	Vol. algo.	Steps algo.
16	27.0%	28.1%	28.1%
34	20.6%	25.5%	22.1%
52	13.6%	25.8%	26.2%
70	12.7%	14.5%	4.8%
88	12.0%	15.7%	13.4%

Results for ChunkSet (Earth science application)

[J. Herrmann et al., Parallel Computing 2016]

Data redistribution – Conclusion

Summary:

- Algorithms that find the optimal target distribution for different redistribution metrics
- NP-completeness proof for minimizing redistribution time followed by a computation kernel
- Experimental validation on ParSEC for QR factorization kernel

Outline

Perspectives – scheduling problems

Data locality still a very timely research topic

Memory-aware scheduling for distributed memories

- Consider data movement at the same time
- Trade-off between performance and data movement
- Partition trees/graphs for both performance and memory

Memory-aware work-stealing

- Work-stealing: distributed, dynamic scheduler
- Existing lower/upper bounds on data locality
- How to derive memory guarantees?
- Based on which pre-computed information?

Perspectives – runtime schedulers

Collaboration with runtime experts:

- Started during the SOLHAR project
- Need to adapt our algorithms:
  - Lower scheduling complexity
  - Make the algorithms dynamic (graph gradually uncovered)
  - Distributing scheduling decisions
- Possible tools:
  - Hierarchical scheduling
  - Precompute memory information on the graph

$\rightsquigarrow$ New scheduling problems!

Outline