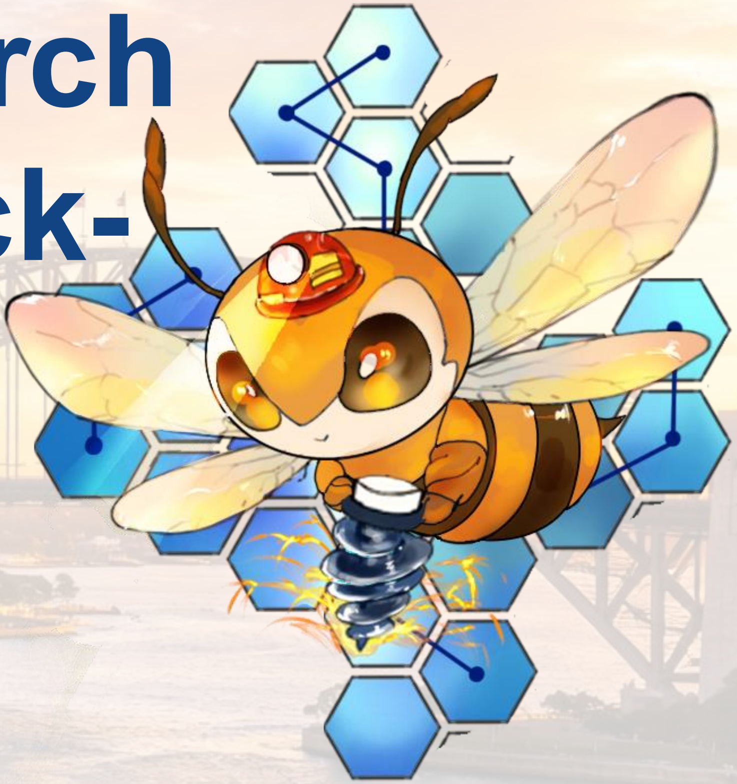
PPoPP 2026

DiggerBees: Depth First Search Leveraging Hierarchical Block-Level Stealing on GPUs



Yuyao Niu[1, 2] Yuechen Lu[3] Weifeng Liu[3] Marc Casas[1, 2]

[1] Barcelona Supercomputing Center
[2] Universitat Politècnica de Catalunya
[3] SSSLab, China University of Petroleum-Beijing

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Departament d'Arquitectura de Computadors

中国石油大学
CHINA UNIVERSITY OF PETROLEUM

SSSLab
超级科学软件实验室
Super Scientific Software Laboratory

Sydney, Australia • February 2, 2026

https://doi.org/10.5281/zenodo.17709254

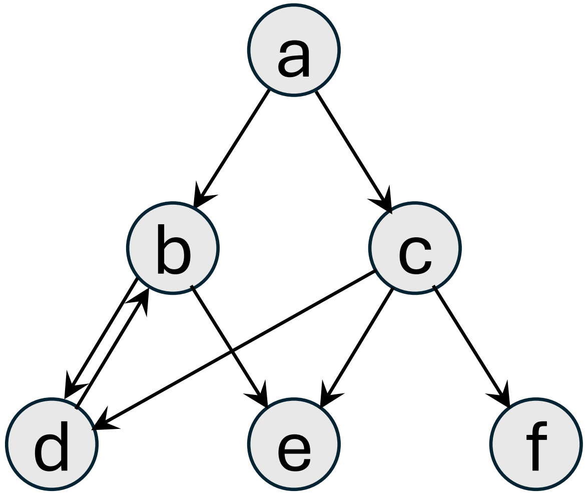
OUTLINE

1. Introduction and Motivations
2. DiggerBees Implementation
3. Performance Evaluation
4. Conclusion

OUTLINE

Introduction

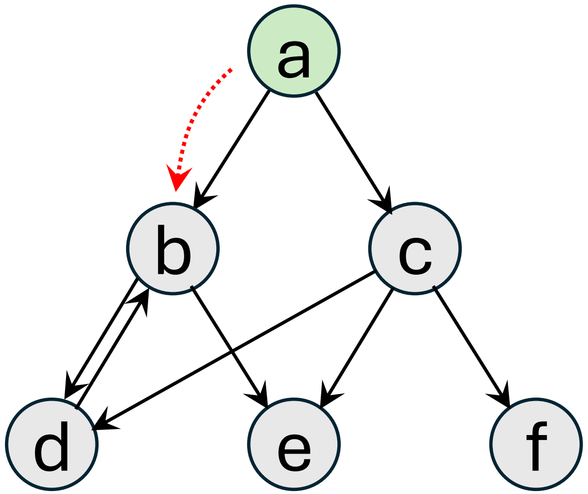
- **Depth First Search (DFS)** traverses a graph by exploring vertices along one branch as deeply as possible before backtracking, generating a valid DFS tree.
- **Serial DFS** produces the unique lexicographically ordered DFS tree.



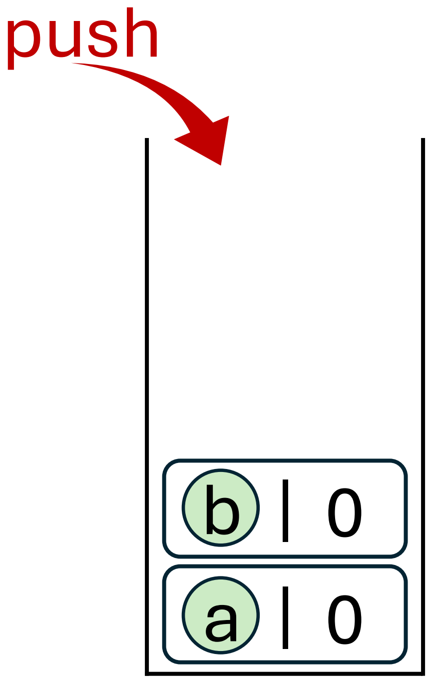
Input graph

Introduction

- **Depth First Search (DFS)** traverses a graph by exploring vertices along one branch as deeply as possible before backtracking, generating a valid DFS tree.
- **Serial DFS** produces the unique lexicographically ordered DFS tree.



Input graph

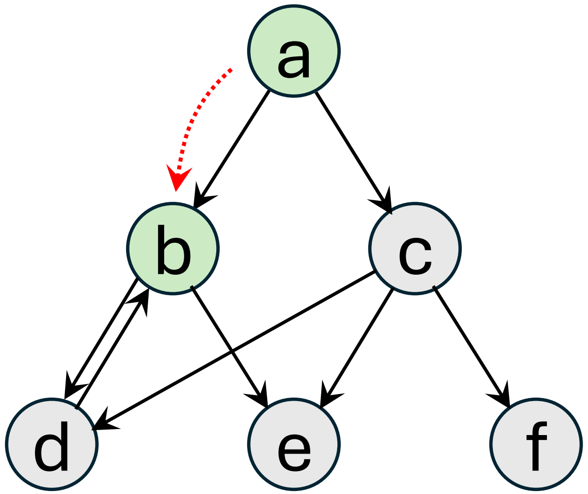


Stack

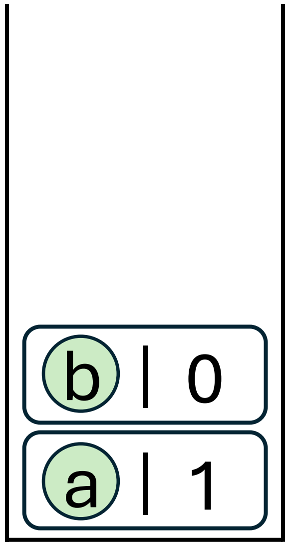
vertex | *next_idx*

Introduction

- **Depth First Search (DFS)** traverses a graph by exploring vertices along one branch as deeply as possible before backtracking, generating a valid DFS tree.
- **Serial DFS** produces the unique lexicographically ordered DFS tree.



Input graph

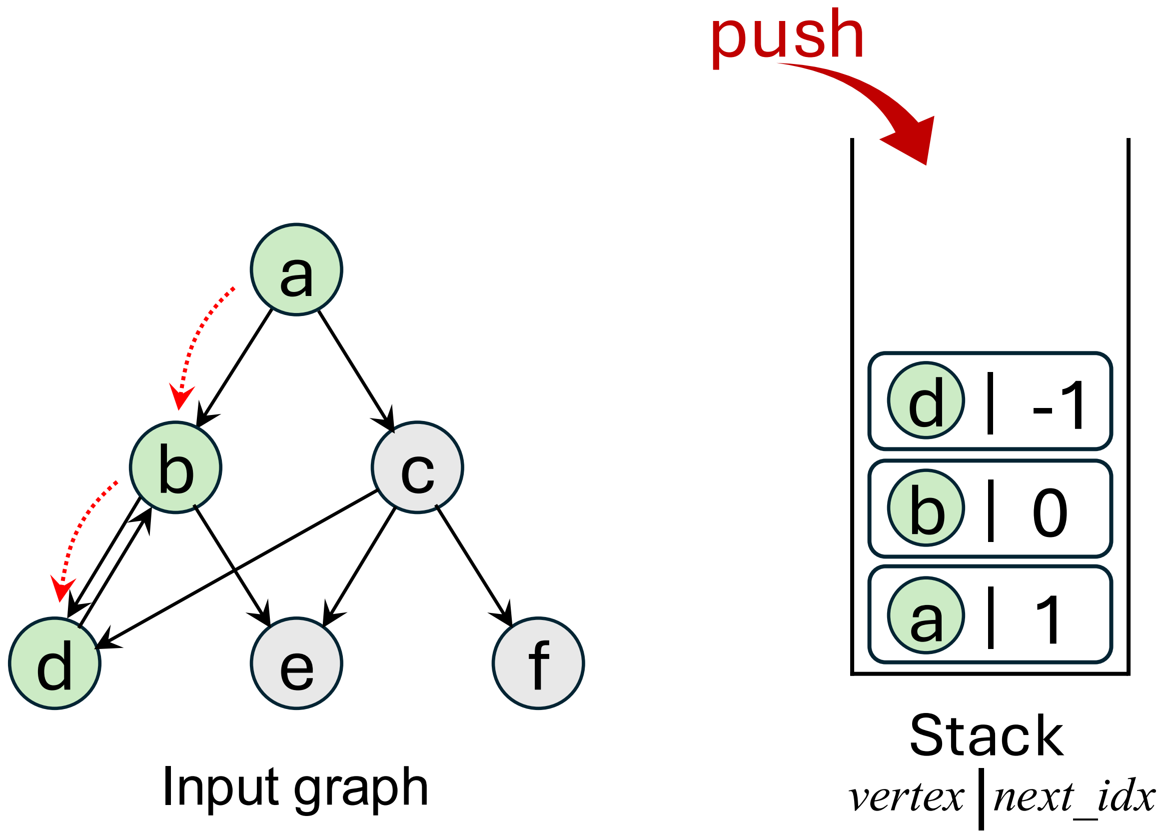


Stack

vertex | *next_idx*

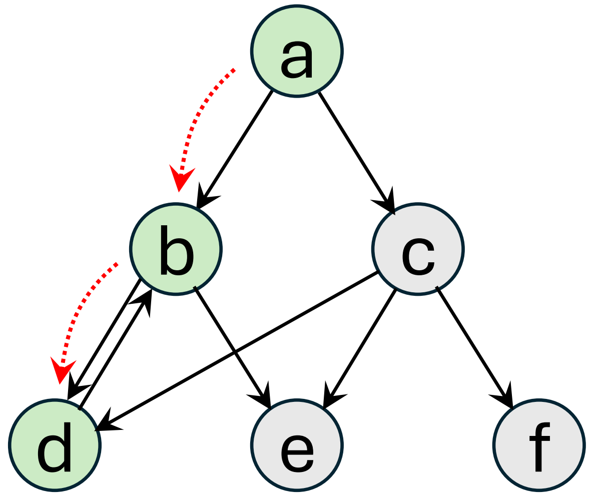
Introduction

- **Depth First Search (DFS)** traverses a graph by exploring vertices along one branch as deeply as possible before backtracking, generating a valid DFS tree.
- **Serial DFS** produces the unique lexicographically ordered DFS tree.

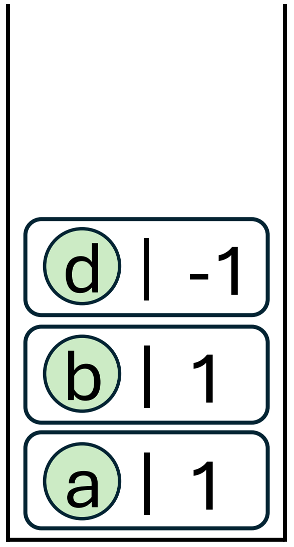


Introduction

- **Depth First Search (DFS)** traverses a graph by exploring vertices along one branch as deeply as possible before backtracking, generating a valid DFS tree.
- **Serial DFS** produces the unique lexicographically ordered DFS tree.



Input graph

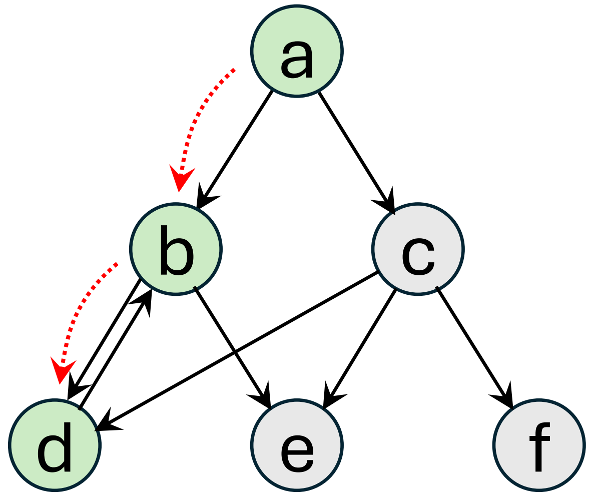


Stack

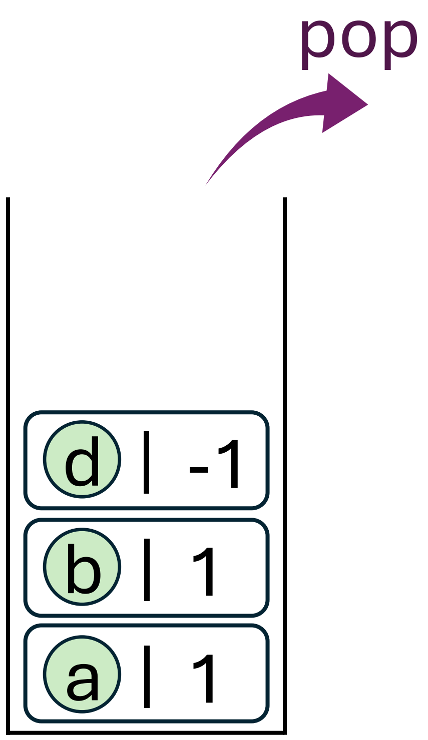
vertex | *next_idx*

Introduction

- **Depth First Search (DFS)** traverses a graph by exploring vertices along one branch as deeply as possible before backtracking, generating a valid DFS tree.
- **Serial DFS** produces the unique lexicographically ordered DFS tree.



Input graph

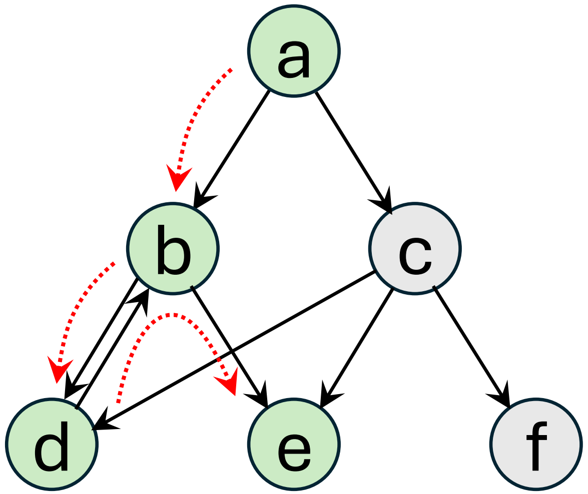


Stack

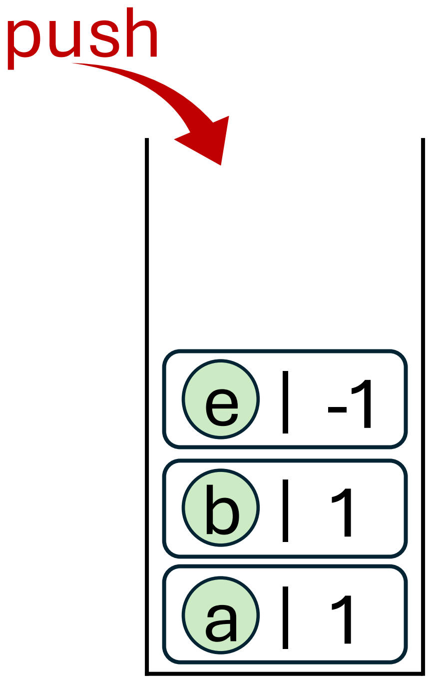
vertex | *next_idx*

Introduction

- **Depth First Search (DFS)** traverses a graph by exploring vertices along one branch as deeply as possible before backtracking, generating a valid DFS tree.
- **Serial DFS** produces the unique lexicographically ordered DFS tree.



Input graph

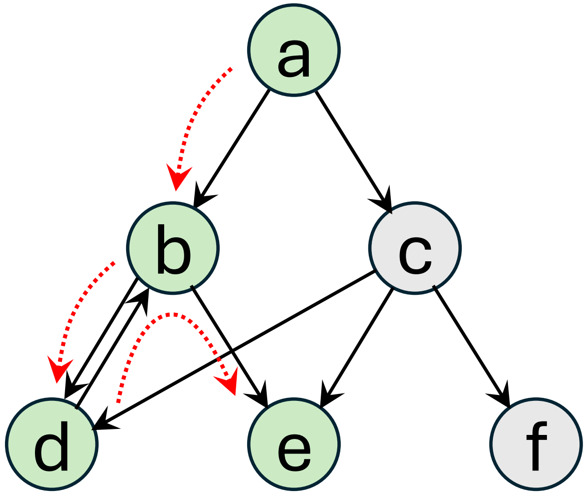


Stack

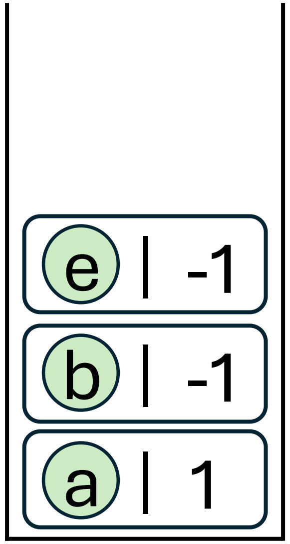
vertex | *next_idx*

Introduction

- **Depth First Search (DFS)** traverses a graph by exploring vertices along one branch as deeply as possible before backtracking, generating a valid DFS tree.
- **Serial DFS** produces the unique lexicographically ordered DFS tree.



Input graph

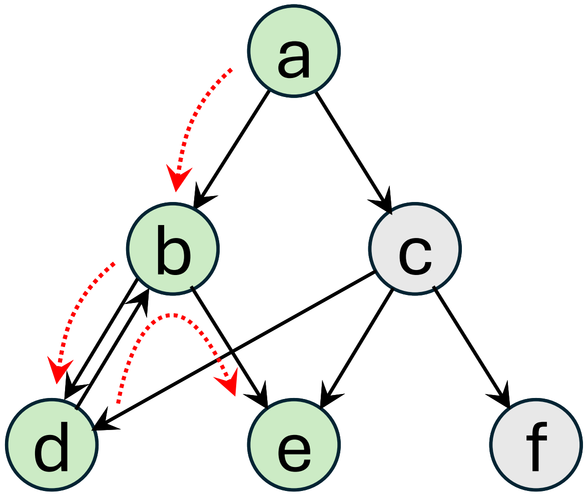


Stack

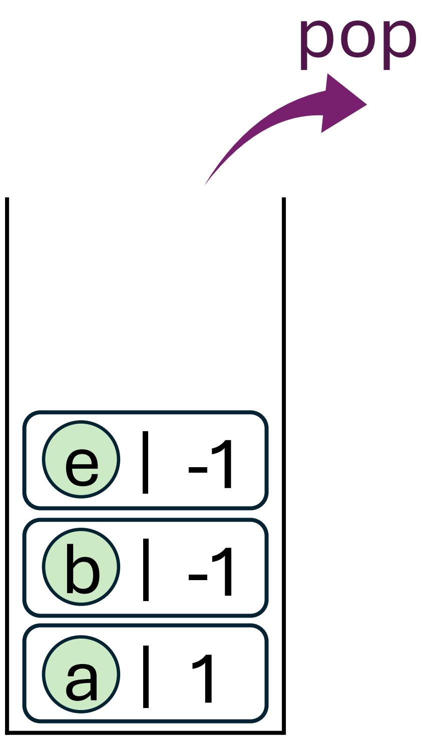
vertex | *next_idx*

Introduction

- **Depth First Search (DFS)** traverses a graph by exploring vertices along one branch as deeply as possible before backtracking, generating a valid DFS tree.
- **Serial DFS** produces the unique lexicographically ordered DFS tree.



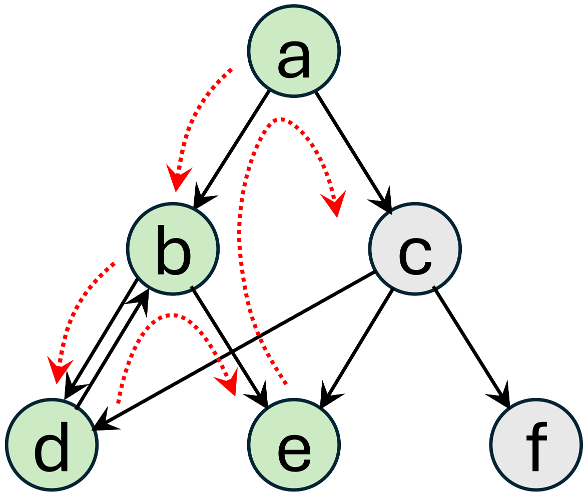
Input graph



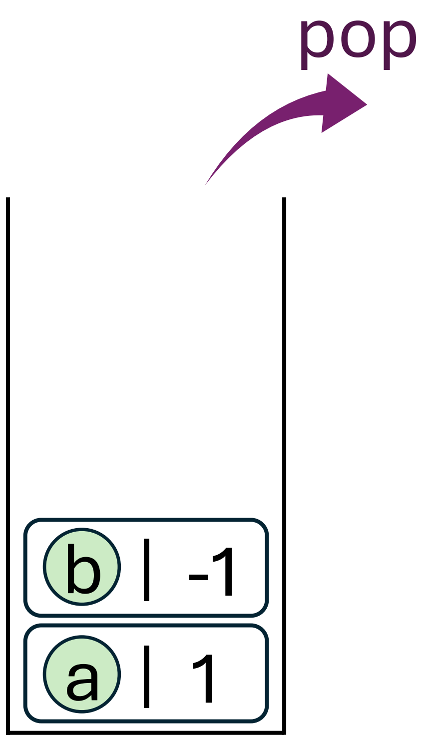
Stack
vertex | *next_idx*

Introduction

- **Depth First Search (DFS)** traverses a graph by exploring vertices along one branch as deeply as possible before backtracking, generating a valid DFS tree.
- **Serial DFS** produces the unique lexicographically ordered DFS tree.



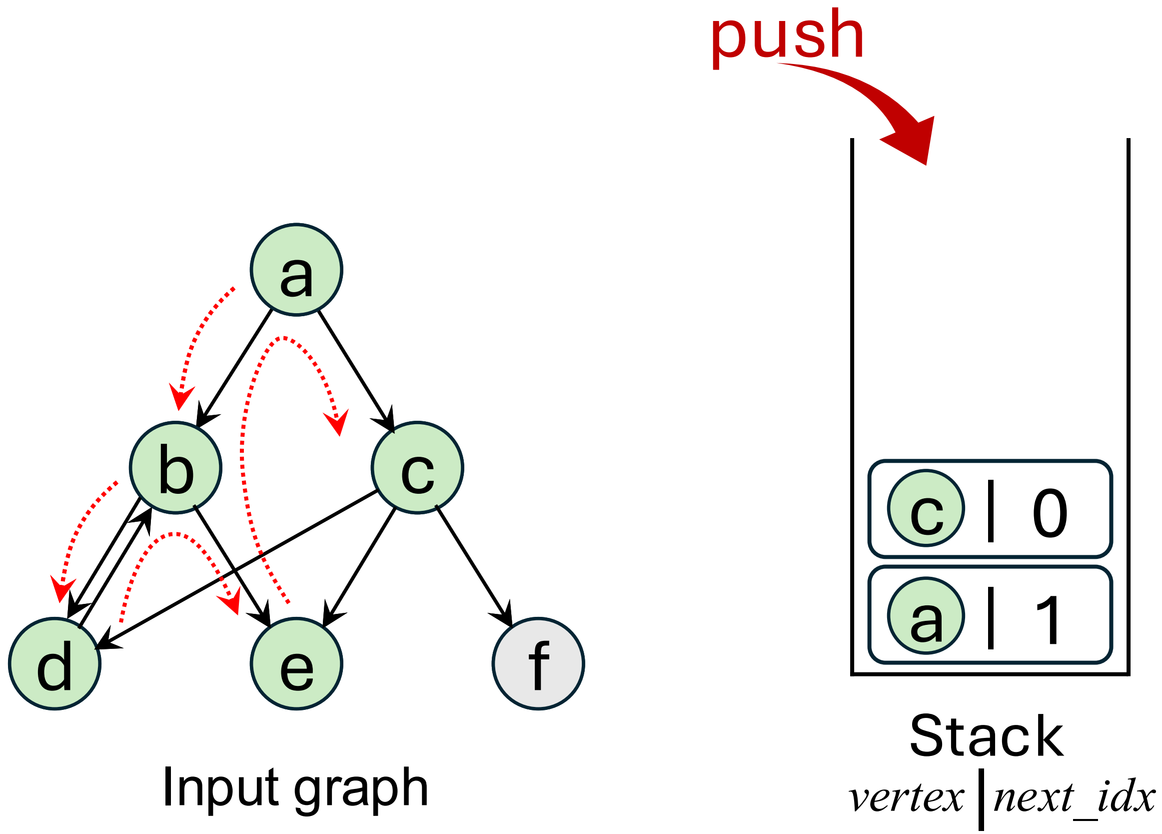
Input graph



Stack
vertex | *next_idx*

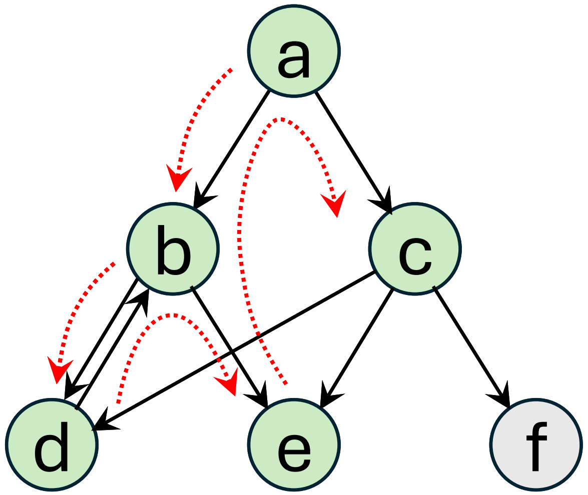
Introduction

- **Depth First Search (DFS)** traverses a graph by exploring vertices along one branch as deeply as possible before backtracking, generating a valid DFS tree.
- **Serial DFS** produces the unique lexicographically ordered DFS tree.

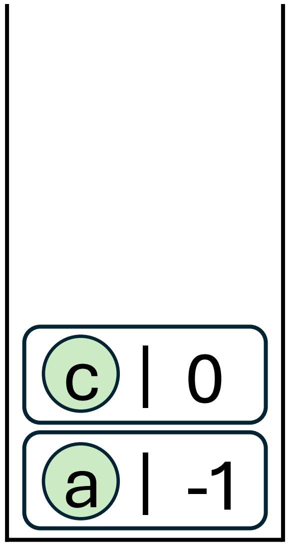


Introduction

- **Depth First Search (DFS)** traverses a graph by exploring vertices along one branch as deeply as possible before backtracking, generating a valid DFS tree.
- **Serial DFS** produces the unique lexicographically ordered DFS tree.



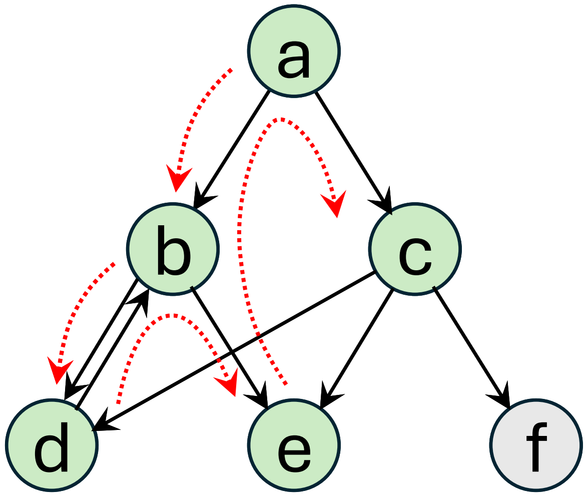
Input graph



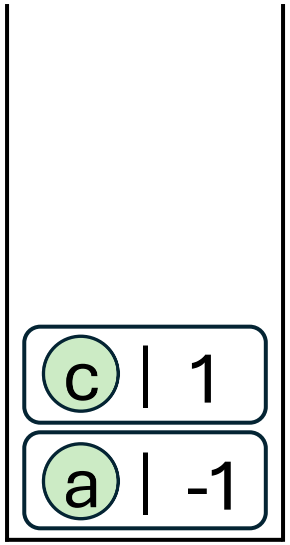
Stack
vertex | *next_idx*

Introduction

- **Depth First Search (DFS)** traverses a graph by exploring vertices along one branch as deeply as possible before backtracking, generating a valid DFS tree.
- **Serial DFS** produces the unique lexicographically ordered DFS tree.



Input graph

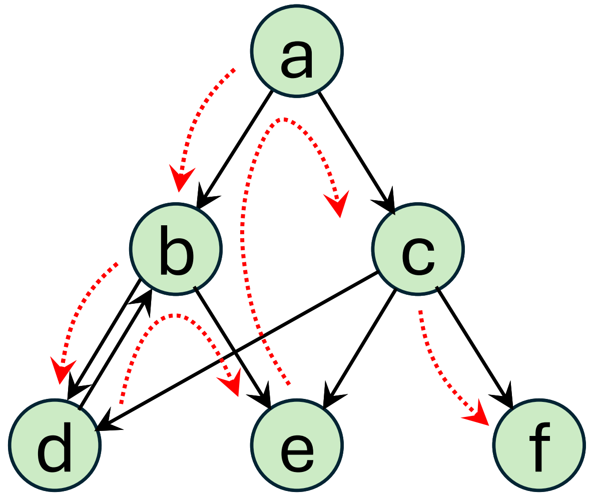


Stack

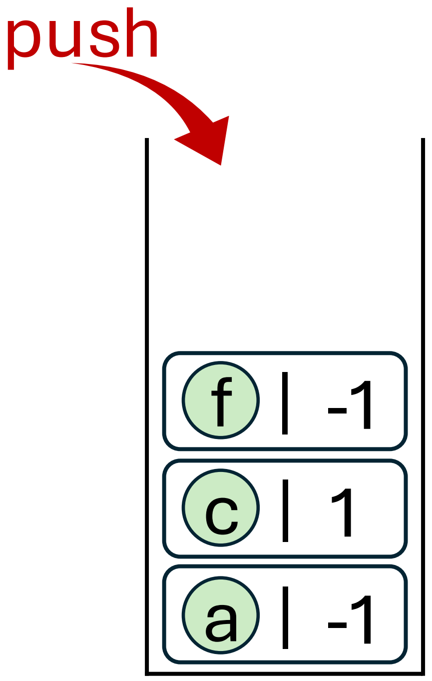
vertex | *next_idx*

Introduction

- **Depth First Search (DFS)** traverses a graph by exploring vertices along one branch as deeply as possible before backtracking, generating a valid DFS tree.
- **Serial DFS** produces the unique lexicographically ordered DFS tree.



Input graph

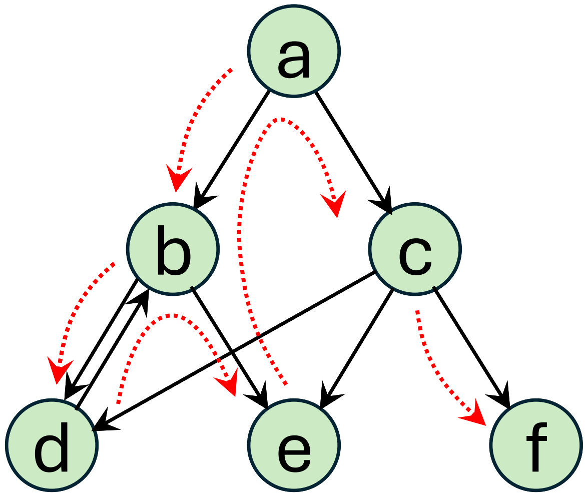


Stack

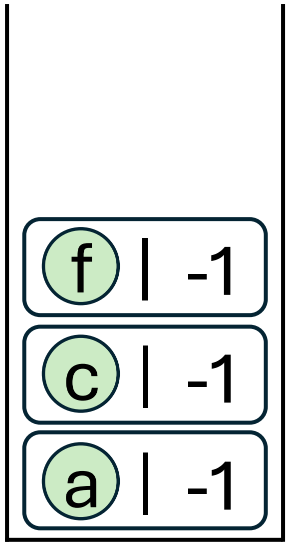
vertex | *next_idx*

Introduction

- **Depth First Search (DFS)** traverses a graph by exploring vertices along one branch as deeply as possible before backtracking, generating a valid DFS tree.
- **Serial DFS** produces the unique lexicographically ordered DFS tree.



Input graph

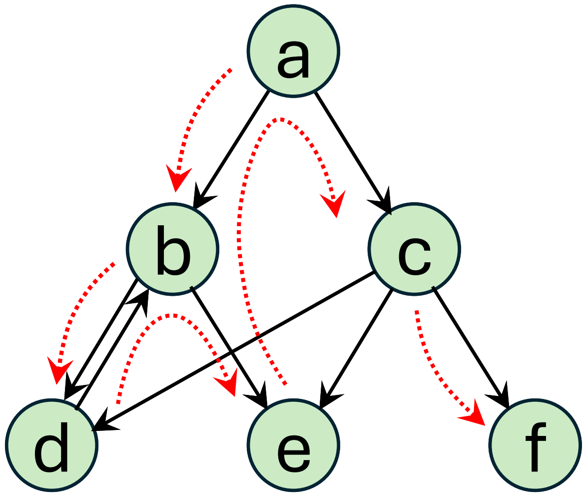


Stack

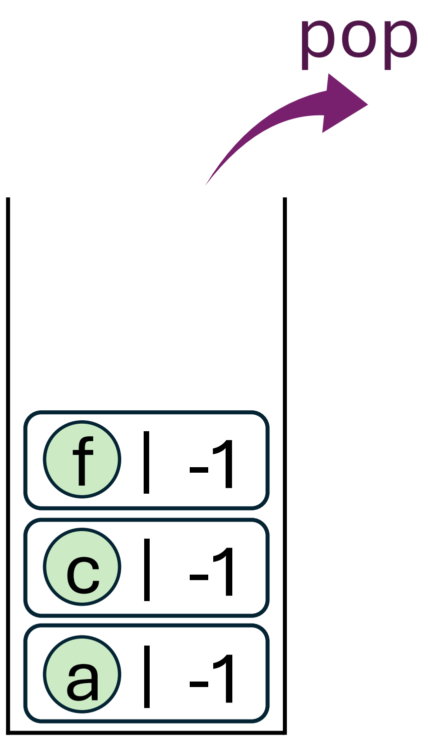
vertex | *next_idx*

Introduction

- **Depth First Search (DFS)** traverses a graph by exploring vertices along one branch as deeply as possible before backtracking, generating a valid DFS tree.
- **Serial DFS** produces the unique lexicographically ordered DFS tree.



Input graph

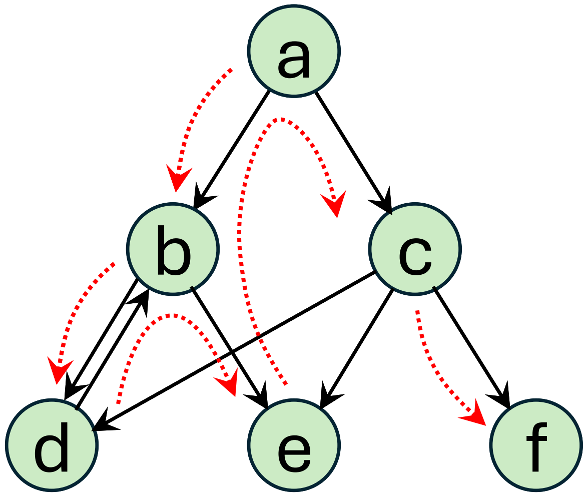


Stack

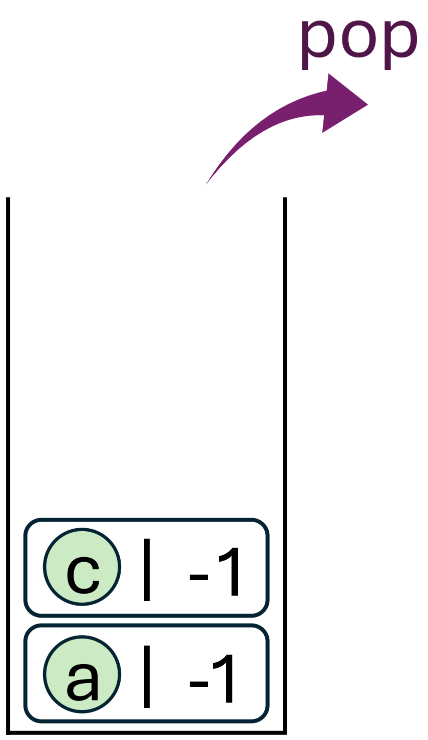
vertex | *next_idx*

Introduction

- **Depth First Search (DFS)** traverses a graph by exploring vertices along one branch as deeply as possible before backtracking, generating a valid DFS tree.
- **Serial DFS** produces the unique lexicographically ordered DFS tree.



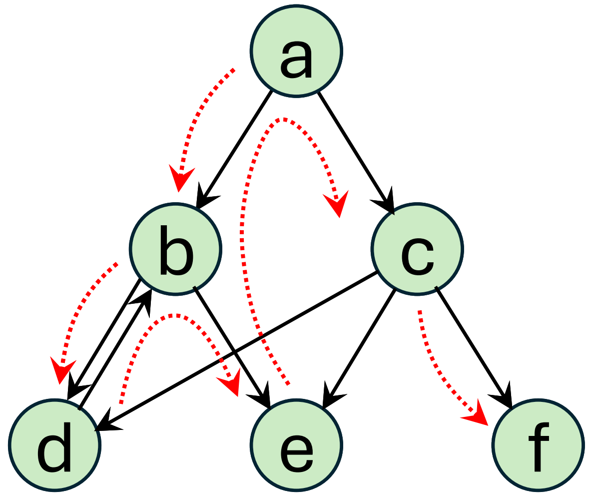
Input graph



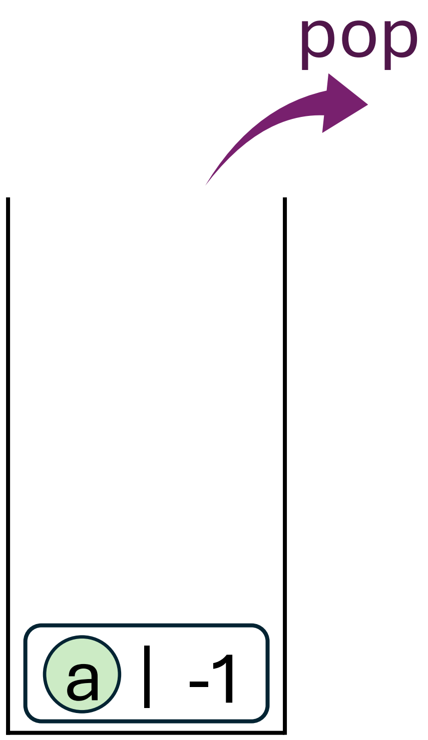
Stack
vertex | *next_idx*

Introduction

- **Depth First Search (DFS)** traverses a graph by exploring vertices along one branch as deeply as possible before backtracking, generating a valid DFS tree.
- **Serial DFS** produces the unique lexicographically ordered DFS tree.



Input graph

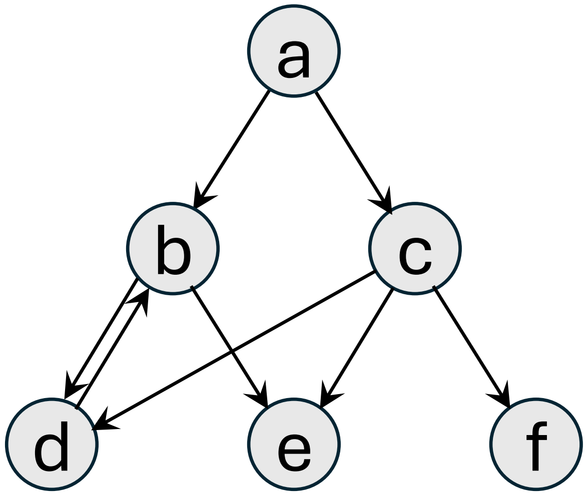


Stack
vertex | *next_idx*

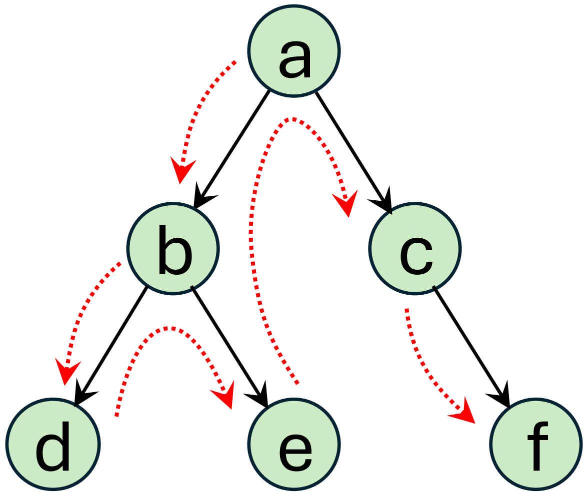
Introduction

- **Depth First Search (DFS)** traverses a graph by exploring vertices along one branch as deeply as possible before backtracking, generating a valid DFS tree.
- **Serial DFS** produces the unique lexicographically ordered DFS tree.

strong dependencies
hard to parallelize



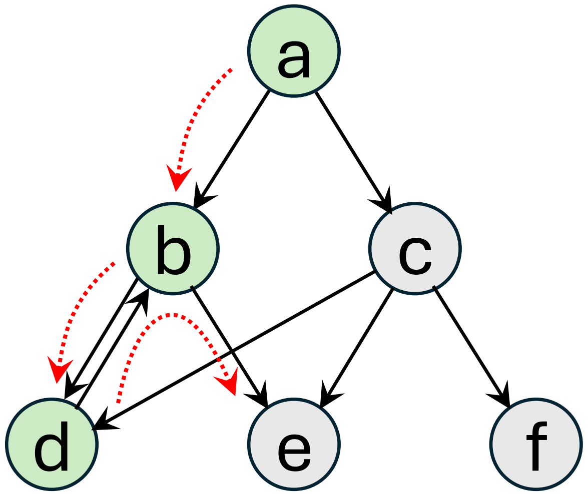
Input graph



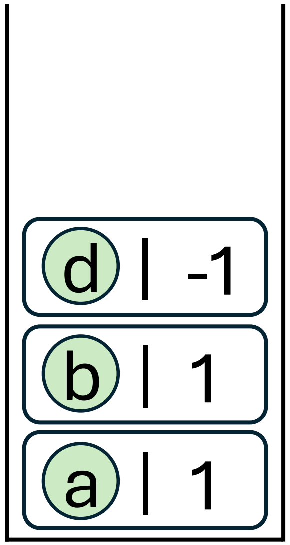
Lex-ordered DFS tree

Introduction

- **Depth First Search (DFS)** traverses a graph by exploring vertices along one branch as deeply as possible before backtracking, generating a valid DFS tree.
- **Serial DFS** produces the unique lexicographically ordered DFS tree.
- **Parallel DFS** relaxes the constraints and constructs a DFS tree without enforcing lexicographic order.



Input graph

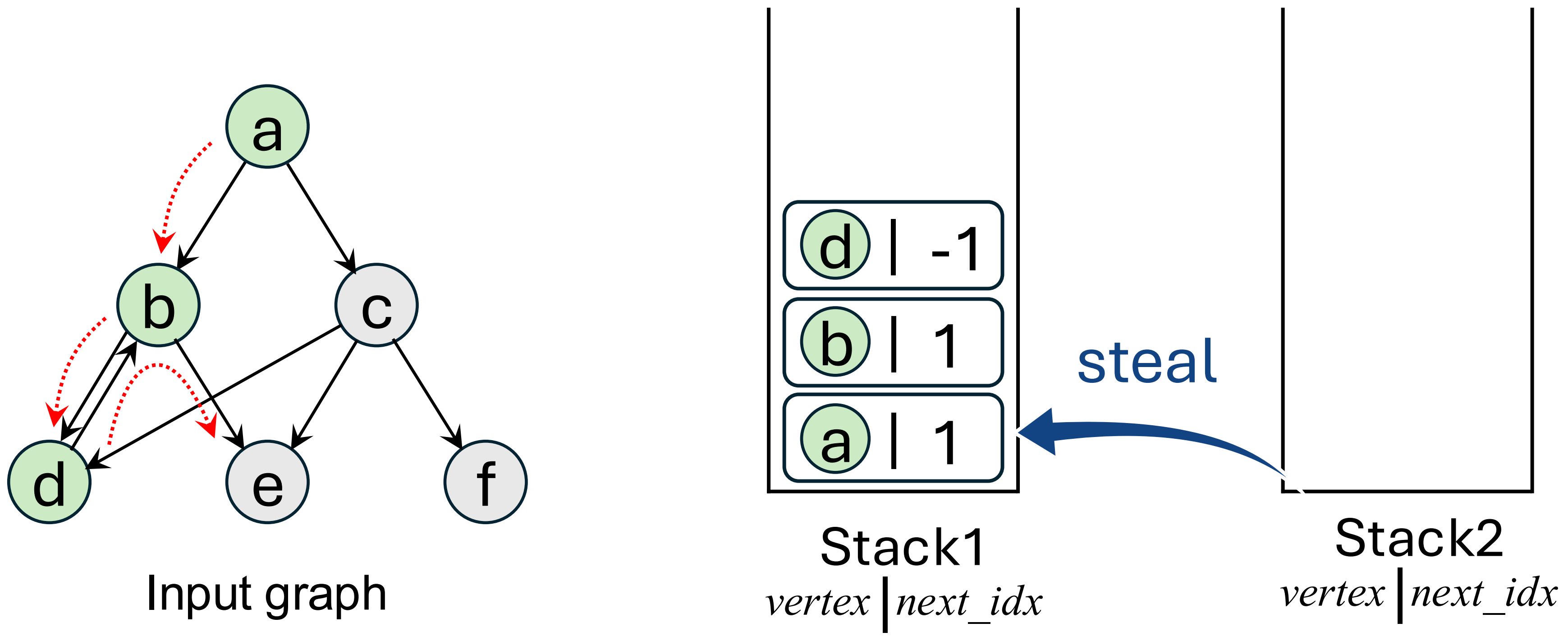


Stack1

vertex | *next_idx*

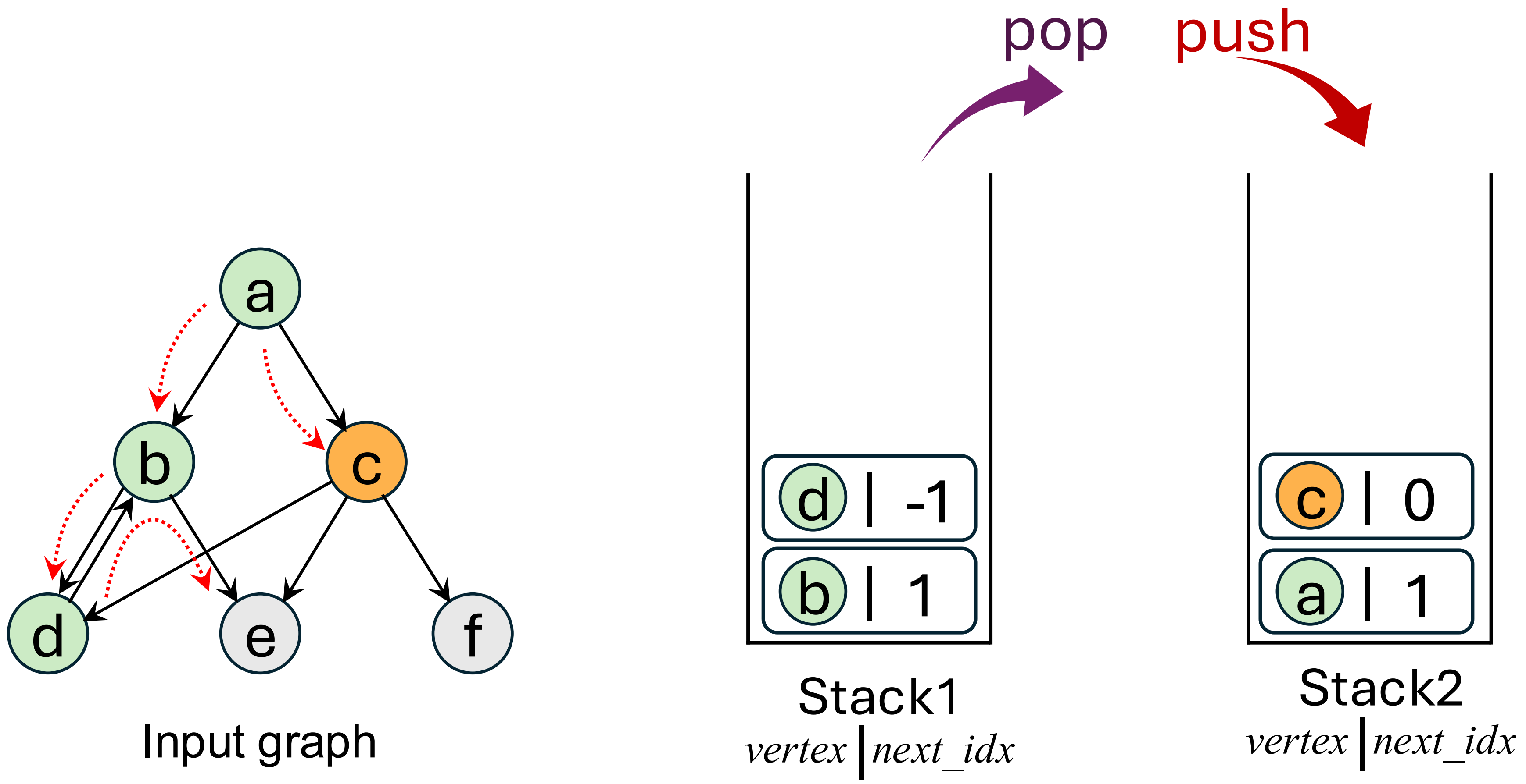
Introduction

- **Depth First Search (DFS)** traverses a graph by exploring vertices along one branch as deeply as possible before backtracking, generating a valid DFS tree.
- **Serial DFS** produces the unique lexicographically ordered DFS tree.
- **Parallel DFS** relaxes the constraints and constructs a DFS tree without enforcing lexicographic order.



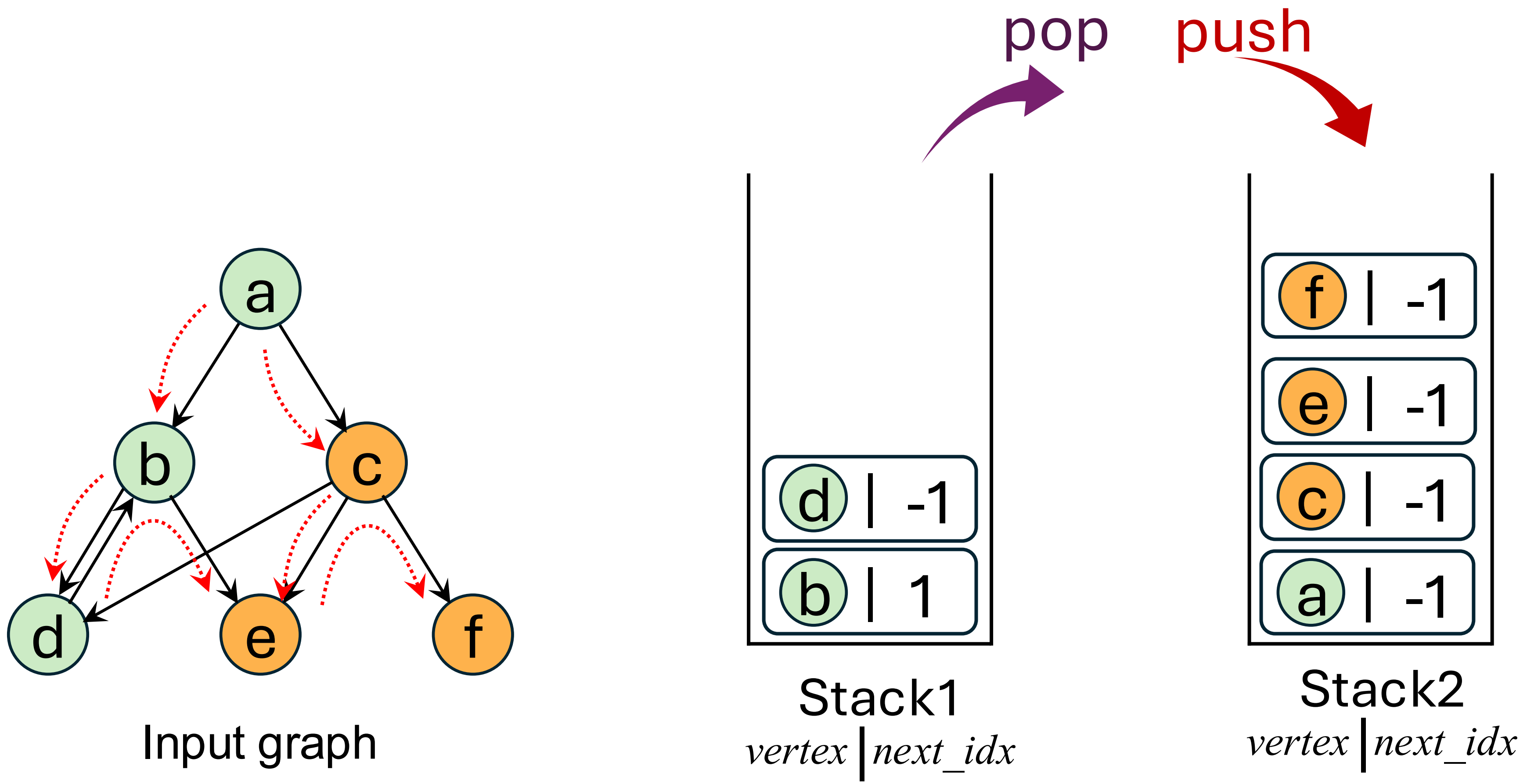
Introduction

- **Depth First Search (DFS)** traverses a graph by exploring vertices along one branch as deeply as possible before backtracking, generating a valid DFS tree.
- **Serial DFS** produces the unique lexicographically ordered DFS tree.
- **Parallel DFS** relaxes the constraints and constructs a DFS tree without enforcing lexicographic order.



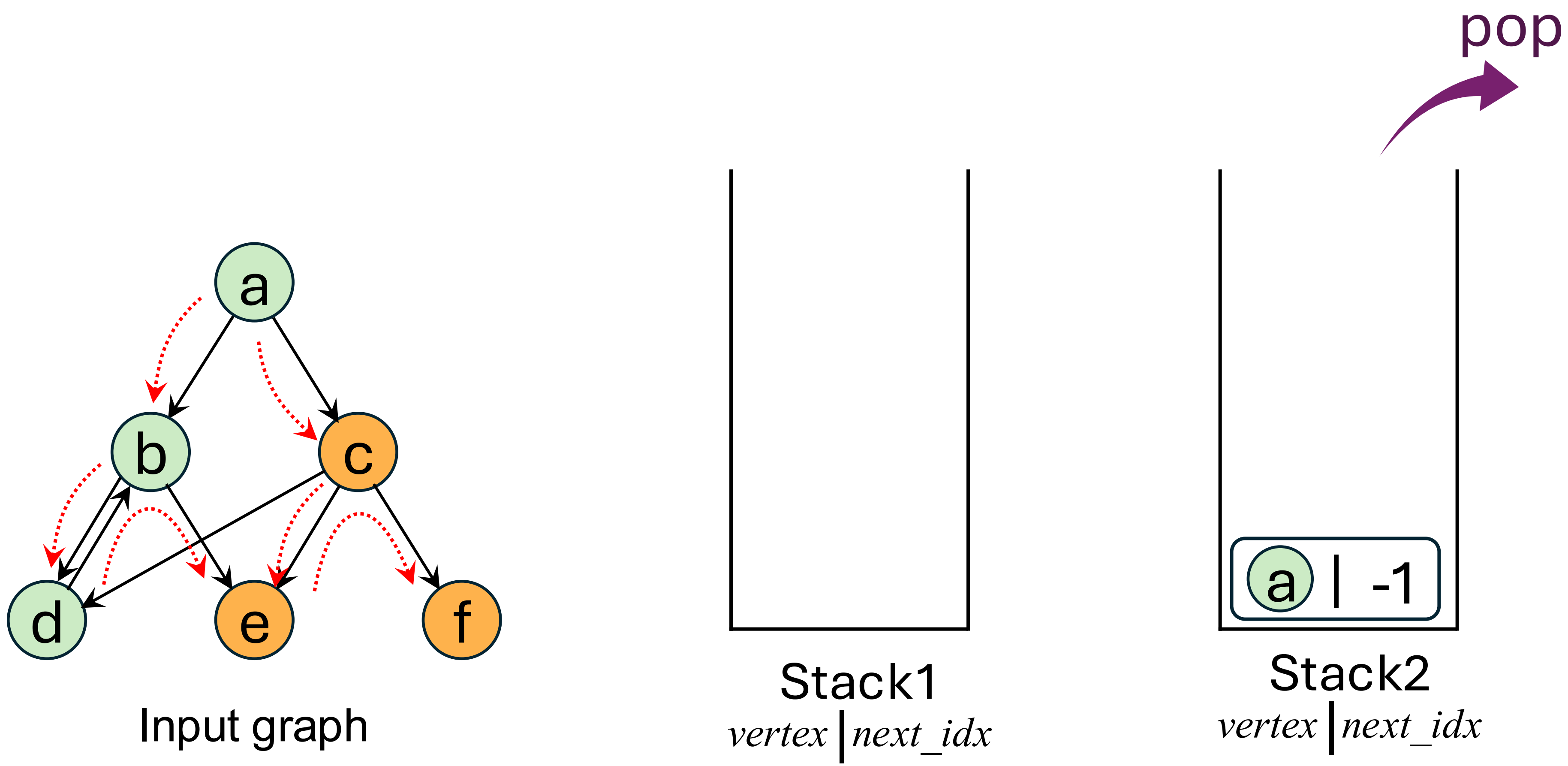
Introduction

- **Depth First Search (DFS)** traverses a graph by exploring vertices along one branch as deeply as possible before backtracking, generating a valid DFS tree.
- **Serial DFS** produces the unique lexicographically ordered DFS tree.
- **Parallel DFS** relaxes the constraints and constructs a DFS tree without enforcing lexicographic order.



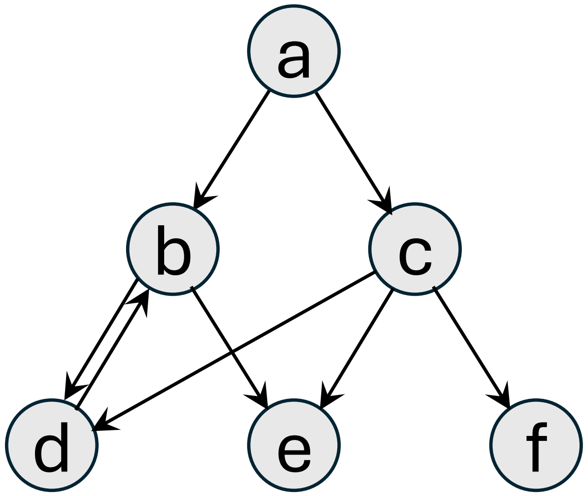
Introduction

- **Depth First Search (DFS)** traverses a graph by exploring vertices along one branch as deeply as possible before backtracking, generating a valid DFS tree.
- **Serial DFS** produces the unique lexicographically ordered DFS tree.
- **Parallel DFS** relaxes the constraints and constructs a DFS tree without enforcing lexicographic order.

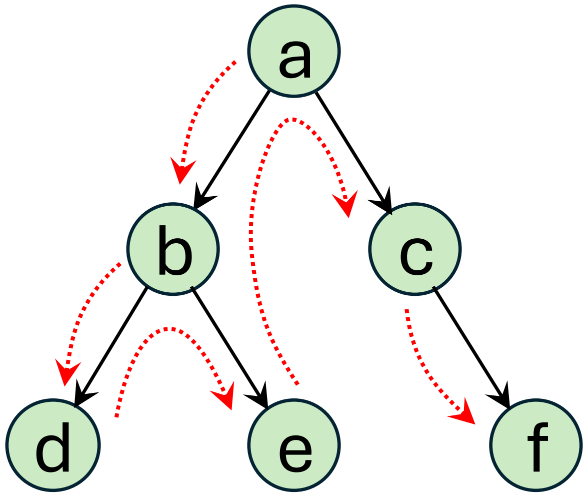


Introduction

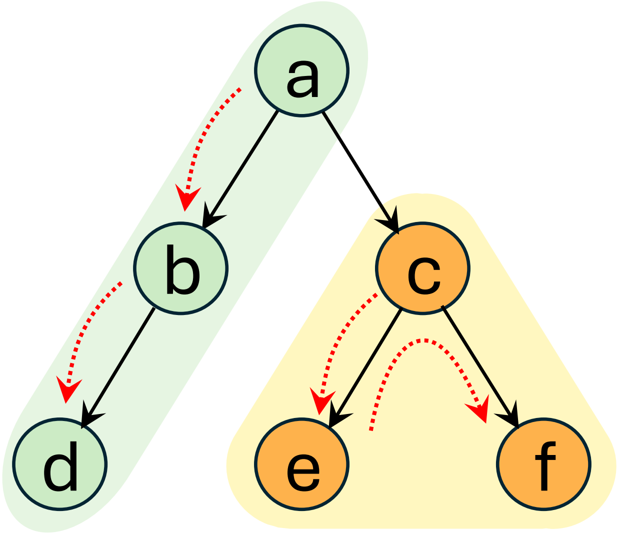
- **Depth First Search (DFS)** traverses a graph by exploring vertices along one branch as deeply as possible before backtracking, generating a valid DFS tree.
- **Serial DFS** produces the unique lexicographically ordered DFS tree.
- **Parallel DFS** relaxes the constraints and constructs a DFS tree without enforcing lexicographic order.



Input graph



Lex-ordered DFS tree



Non-Lex DFS tree

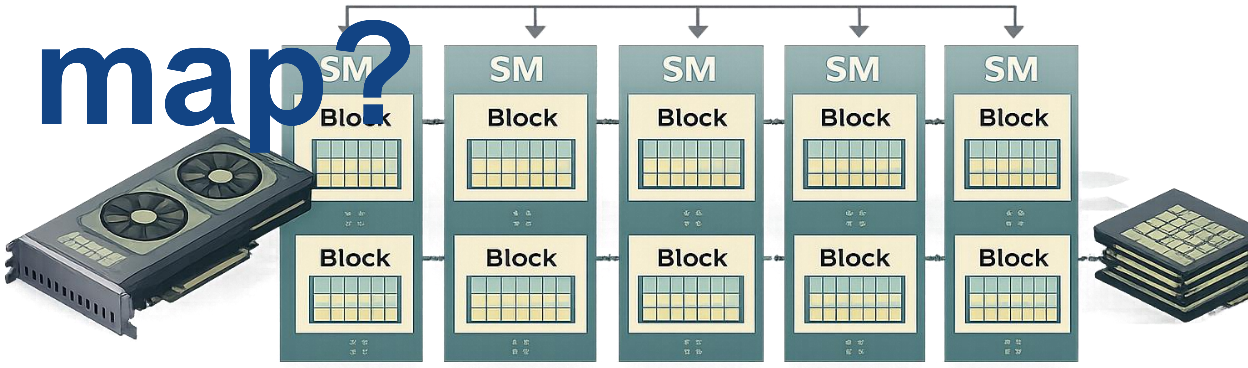
Introduction

- **Depth First Search (DFS)** traverses a graph by exploring vertices along one branch as deeply as possible before backtracking, generating a valid DFS tree.
- **Serial DFS** produces the unique lexicographically ordered DFS tree.
- **Parallel DFS** relaxes the constraints and constructs a DFS tree without enforcing lexicographic order.

Algorithm 2 A pseudocode of the parallel DFS

```
S_i ← Local stack of processor P_i
while not terminated do
    while S_i ≠ ∅ do
        Execute DFS on S_i
    end while
    Steal work from other processors
    Termination Check
end while
```

How to map?



[1] V. Nageshwara Rao and Vipin Kumar. 1987. Parallel depth first search. part i. implementation. International Journal of Parallel Programming 16, 6 (1987), 479–499.
[2] Vipin Kumar and V. Nageshwara Rao. 1987. Parallel depth first search. part ii. analysis. International Journal of Parallel Programming 16, 6 (1987), 501–519.
[3] Guojing Cong, Sreedhar Kodali, Sriram Krishnamoorthy, Doug Lea, Vijay Saraswat, and Tong Wen. 2008. Solving Large, Irregular Graph Problems Using Adaptive Work-Stealing. In ICPP '08. 536–545.
[4] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. 2015. A work efficient algorithm for parallel unordered depth-first search. In SC '15. 1–12.
[5] Prasoon Mishra and V. Krishna Nandivada. 2024. COWS for High Performance: Cost Aware Work Stealing for Irregular Parallel Loop. ACM Trans. Archit. Code Optim., Article 12 (2024), 26 pages.

Motivations

Issue #1: Shared memory *vs* DFS depth

GPU on-chip memory is **limited** (shared memory per SM is small).

DFS may require **very deep stacks** (depth proportional to longest path)

⇒ Cannot keep the whole stack on-chip

Need a segmented stack (on-chip + off-chip segments)

Algorithm 2 A pseudocode of the parallel DFS

$S_i \leftarrow$ Local stack of processor P_i

Execute DFS on S_i
end while
Steal work from other processors
Termination Check
end while

Graphs	#vertices	longest path
rgg_24	16.7 M	2622
road_usa	57.7 M	6262
delaunay	16.7 M	1651
euro_osm	50.9 M	17346

Motivations

Issue #2: Divergence vs Synchronization

Thread-private stacks: all threads follow different execution paths, causing warp divergence.

Shared stack in a block: require costly atomic operations and synchronization.

Hard to get efficient intra-block execution.

Algorithm 2 A pseudocode of the parallel DFS

```
S_i ← Local stack of processor P_i
while not terminated do
    while S_i ≠ ∅ do
        Execute DFS on S_i
    end while
    Steal work from other processors
    Termination Check
end while
```

Motivations

Issue #3: Scalability vs Global Coordination Cost

Execution must extend from single- to multi-block so that more SMs and blocks become active.

It requires costly communication, and irregular DFS workloads complicate balanced distribution.

Hard to achieve scalable inter-block execution while ensuring load balance.

Algorithm 2 A pseudocode of the parallel DFS

```
Sᵢ ← Local stack of processor Pᵢ
while not terminated do
    while Sᵢ ≠ ∅ do
        Execute DFS on Sᵢ
    end while
    Steal work from other processors
    Termination Check
end while
```

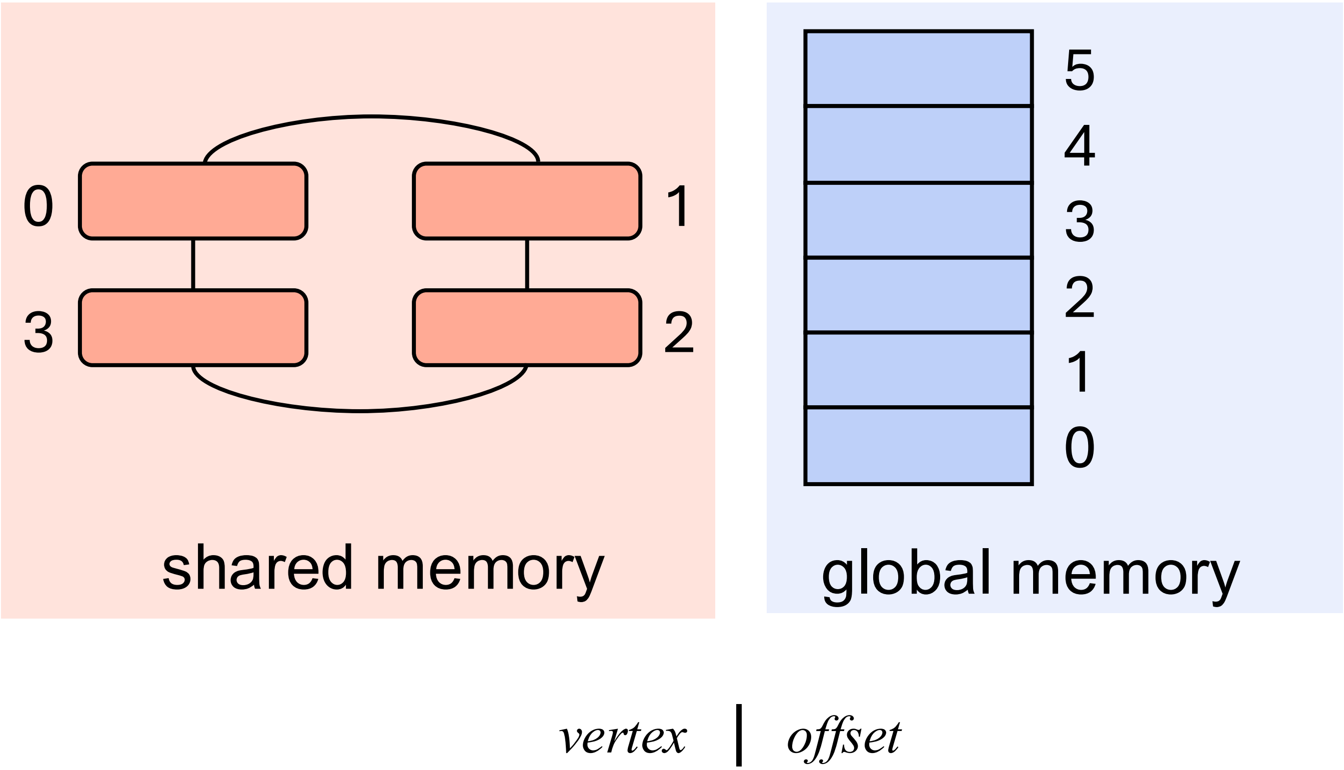
OUTLINE

1. Introduction and Motivations
2. **DiggerBees Implementation**
3. Performance Evaluation
4. Conclusion

Two-Level Stack Data Structure

HotRing: a circular buffer in shared memory serving as the fast-access portion of the stack.

ColdSeg: a contiguous region in global memory serving as the large-capacity portion of the stack.

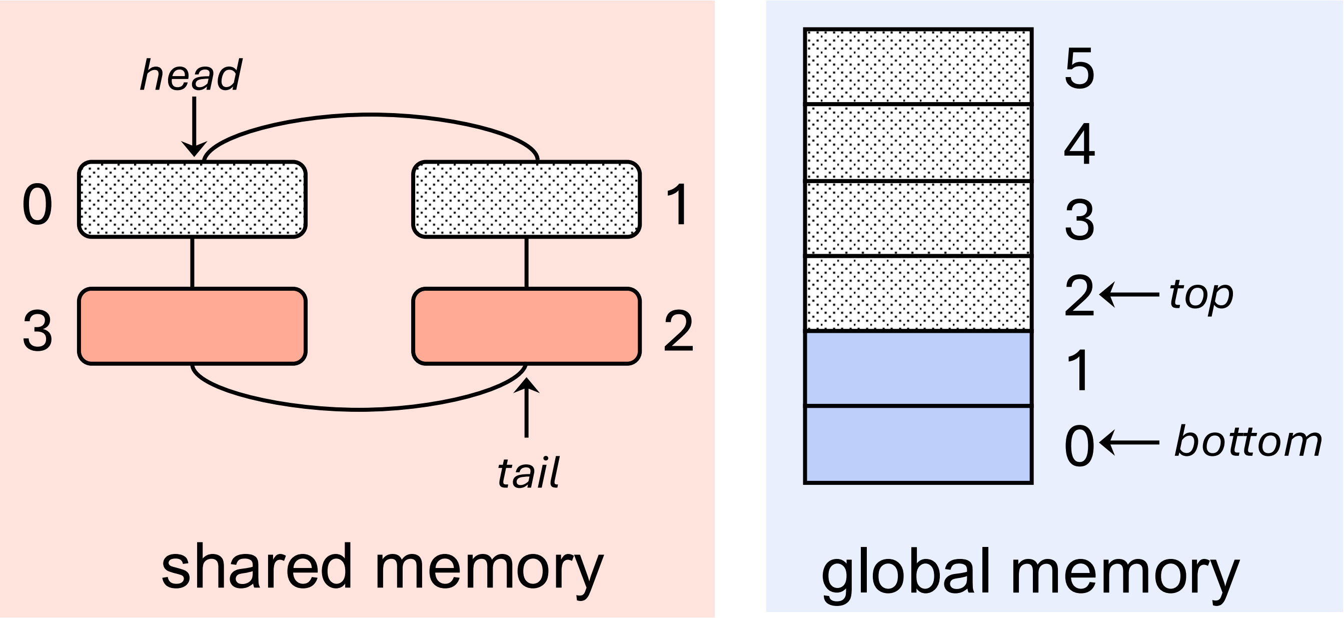


DiggerBees Implementation

Two-Level Stack Data Structure

HotRing: a circular buffer in shared memory serving as the fast-access portion of the stack.

ColdSeg: a contiguous region in global memory serving as the large-capacity portion of the stack.



DiggerBees Implementation

Two-Level Stack Data Structure

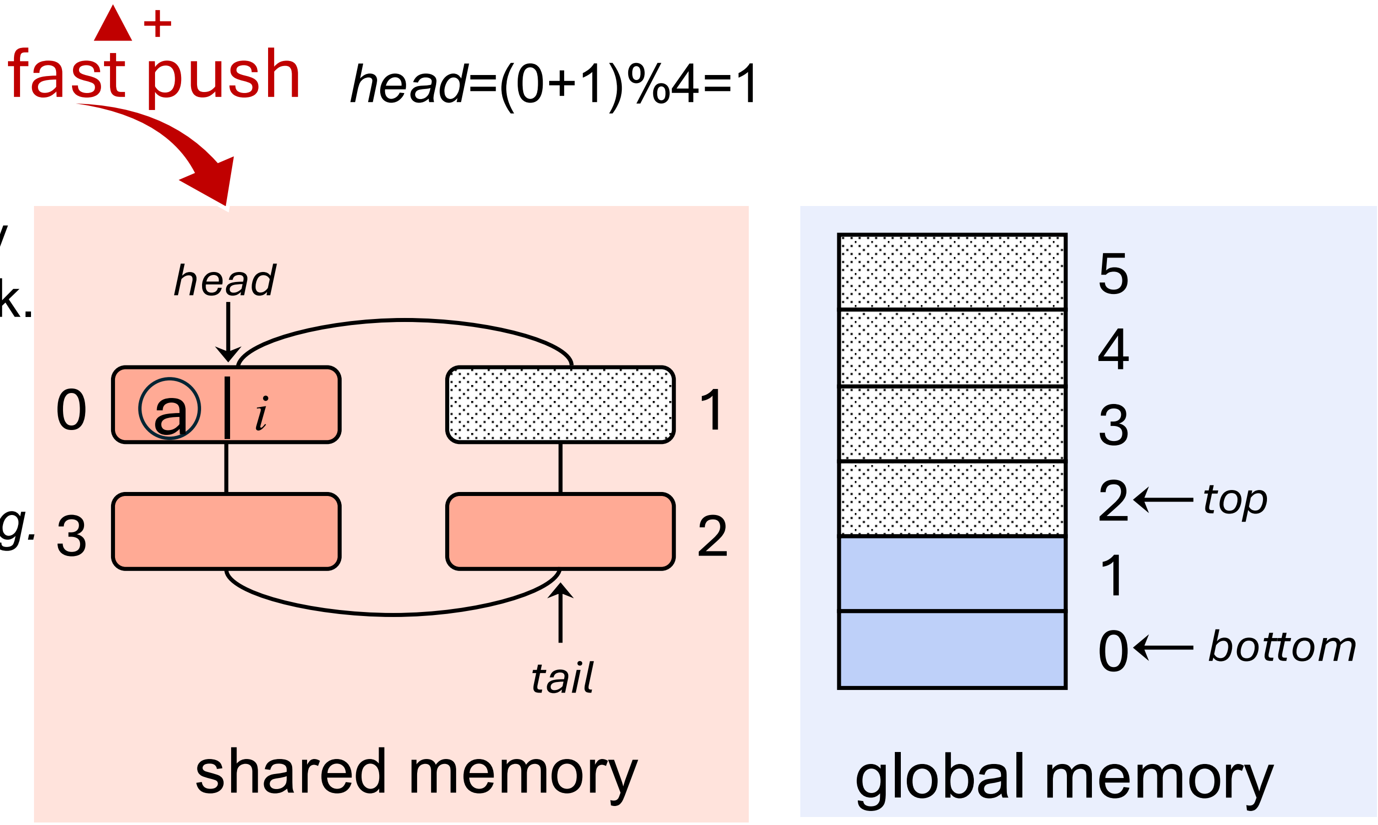
HotRing: a circular buffer in shared memory serving as the fast-access portion of the stack.

ColdSeg: a contiguous region in global memory serving as the large-capacity portion of the stack.

Four core operations:

- Fast push: insert a new entry into the *HotRing*.

  head = (*head* + 1) % *hot_size*



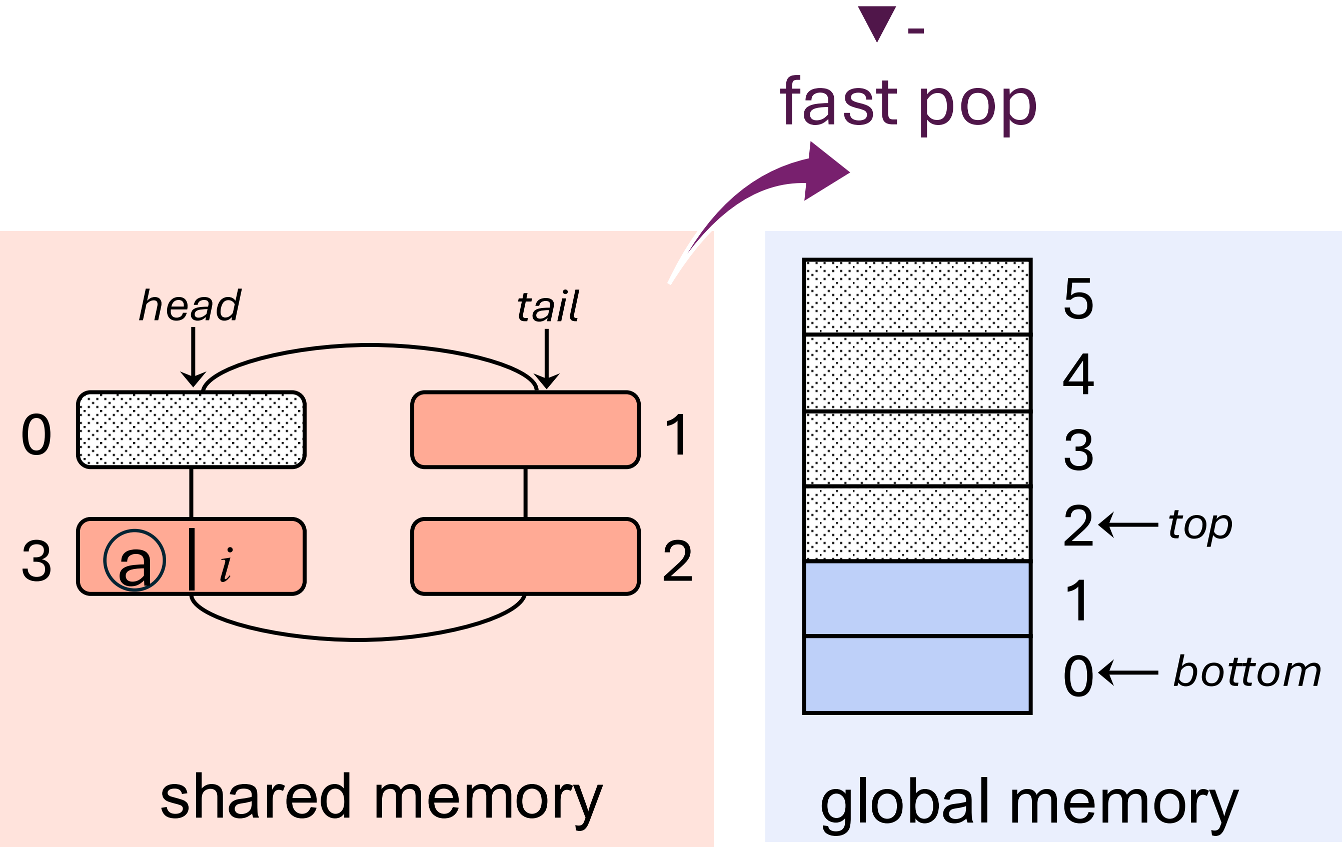
Implementation / Two-level Stack Data Structure

HotRing: a circular buffer in shared memory serving as the fast-access portion of the stack.

ColdSeg: a contiguous region in global memory serving as the large-capacity portion of the stack.

Four core operations:

- Fast push: insert a new entry into the *HotRing*.
  head = (*head* + 1) % *hot_size*
- Fast pop: retrieve the top entry in the HotRing.
  head = (*head* – 1 + *hot_size*) % *hot_size*



DiggerBees Implementation

Two-Level Stack Data Structure

HotRing: a circular buffer in shared memory serving as the fast-access portion of the stack.

ColdSeg: a contiguous region in global memory serving as the large-capacity portion of the stack.

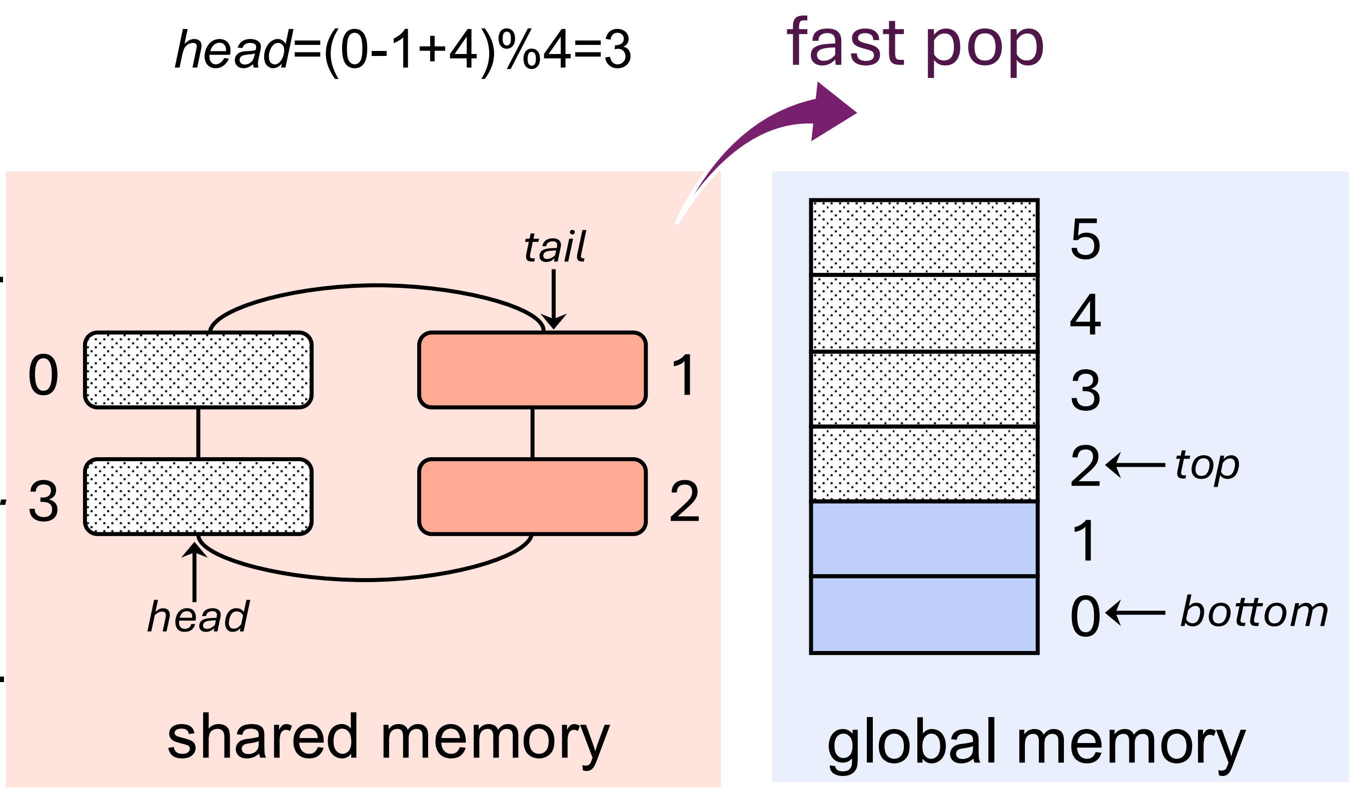
Four core operations:

- Fast push: insert a new entry into the *HotRing*.

  head = (*head* + *1*) % *hot_size*

- Fast pop: retrieve the top entry in the HotRing.

  head = (*head* – *1* + *hot_size*) % *hot_size*



DiggerBees Implementation

Two-Level Stack Data Structure

HotRing: a circular buffer in shared memory serving as the fast-access portion of the stack.

ColdSeg: a contiguous region in global memory serving as the large-capacity portion of the stack.

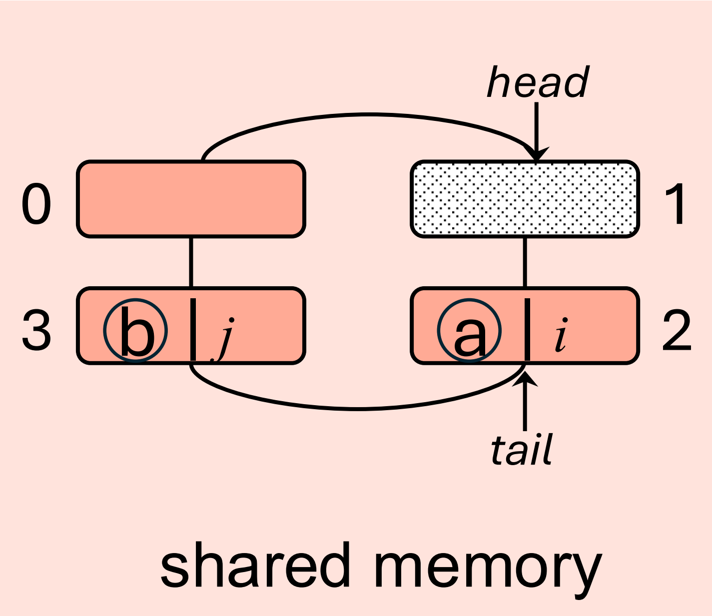
Four core operations:

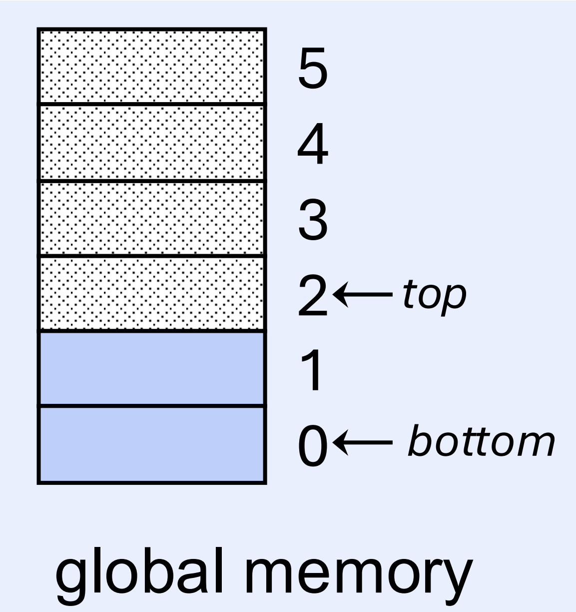
- Fast push: insert a new entry into the *HotRing*.

  head = (*head* + 1) % *hot_size*

- Fast pop: retrieve the top entry in the HotRing.

  head = (*head* – 1 + *hot_size*) % *hot_size*





- Flush: when the HotRing is full, a batch of the oldest entries is moved to the ColdSeg.

  tail = (*tail* + *batch*) % *hot_size*, top = *top* + *batch*

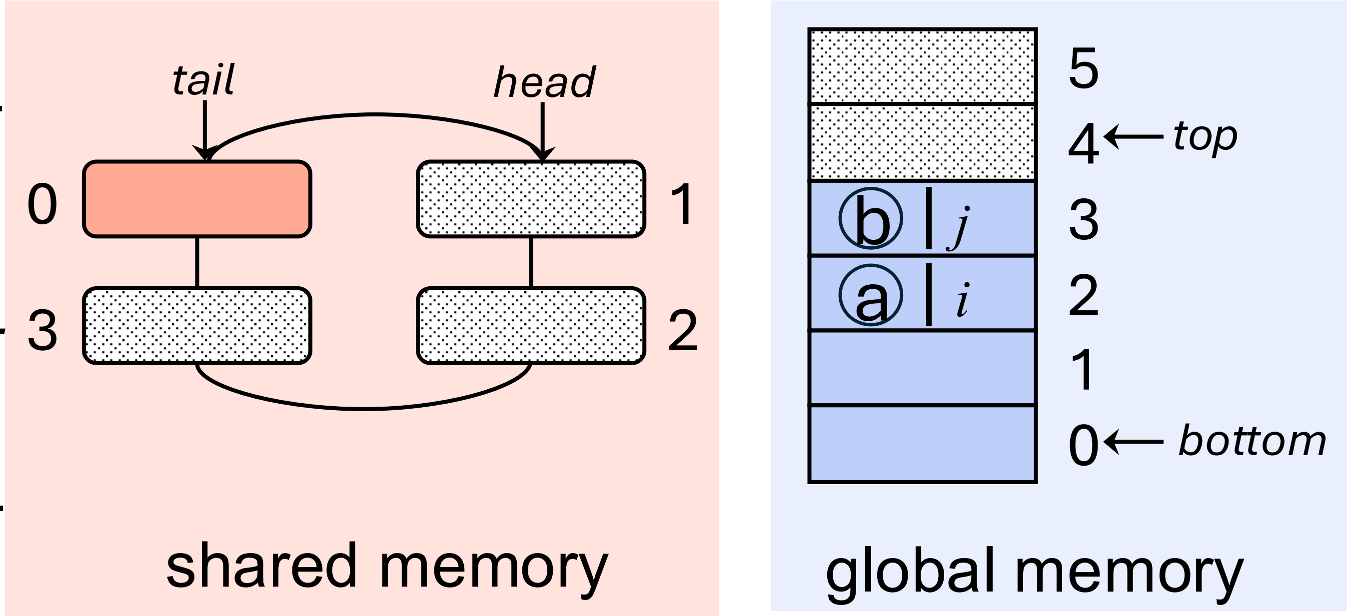
Two-Level Stack Data Structure

HotRing: a circular buffer in shared memory serving as the fast-access portion of the stack.

ColdSeg: a contiguous region in global memory serving as the large-capacity portion of the stack.

tail=(2+2)%4=0

top=2+2=4



Four core operations:

- Fast push: insert a new entry into the *HotRing*.
  head = (*head* + 1) % *hot_size*
- Fast pop: retrieve the top entry in the HotRing.
  head = (*head* – 1 + *hot_size*) % *hot_size*
- Flush: when the HotRing is full, a batch of the oldest entries is moved to the ColdSeg.
  tail = (*tail* + *batch*) % *hot_size*, top = *top* + *batch*

DiggerBees Implementation

Two-Level Stack Data Structure

HotRing: a circular buffer in shared memory serving as the fast-access portion of the stack.

ColdSeg: a contiguous region in global memory serving as the large-capacity portion of the stack.

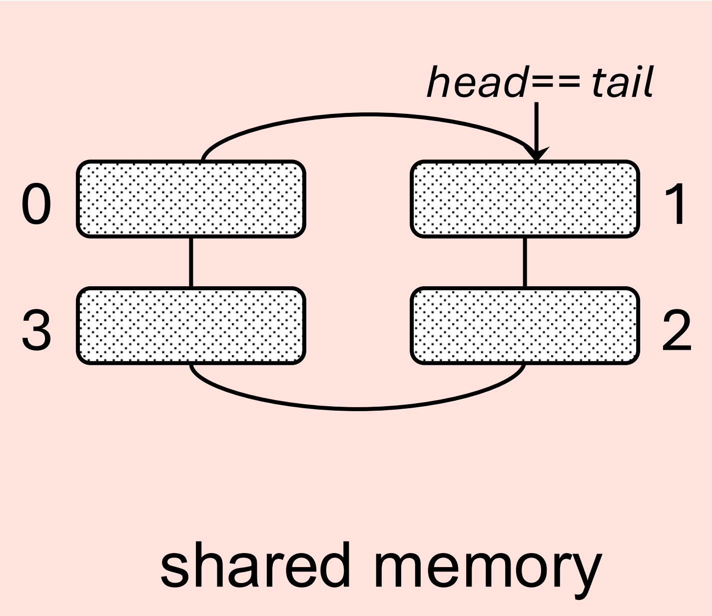
Four core operations:

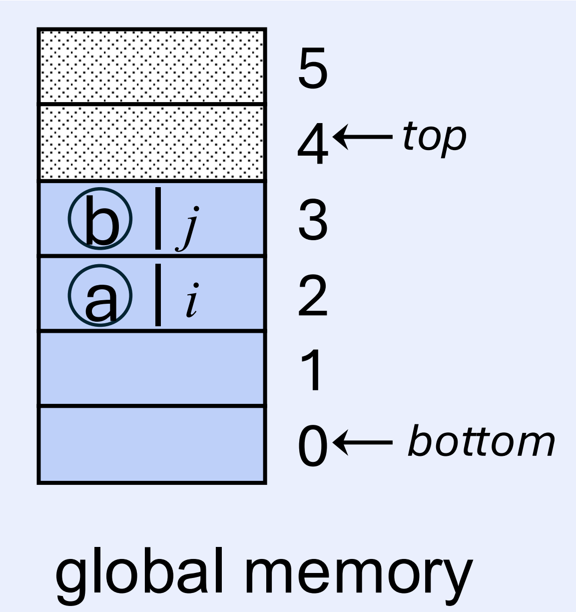
- Fast push: insert a new entry into the *HotRing*.

  head = (*head* + *1*) % *hot_size*

- Fast pop: retrieve the top entry in the HotRing.

  head = (*head* – *1* + *hot_size*) % *hot_size*





- Flush: when the HotRing is full, a batch of the oldest entries is moved to the ColdSeg.

  tail = (*tail* + *batch*) % *hot_size,* top = *top* + *batch*

- Refill: When the HotRing is empty, a batch is refilled from the ColdSeg.

  head = (*head* + *batch*) % *hot_size,* top = *top* - *batch*

DiggerBees Implementation

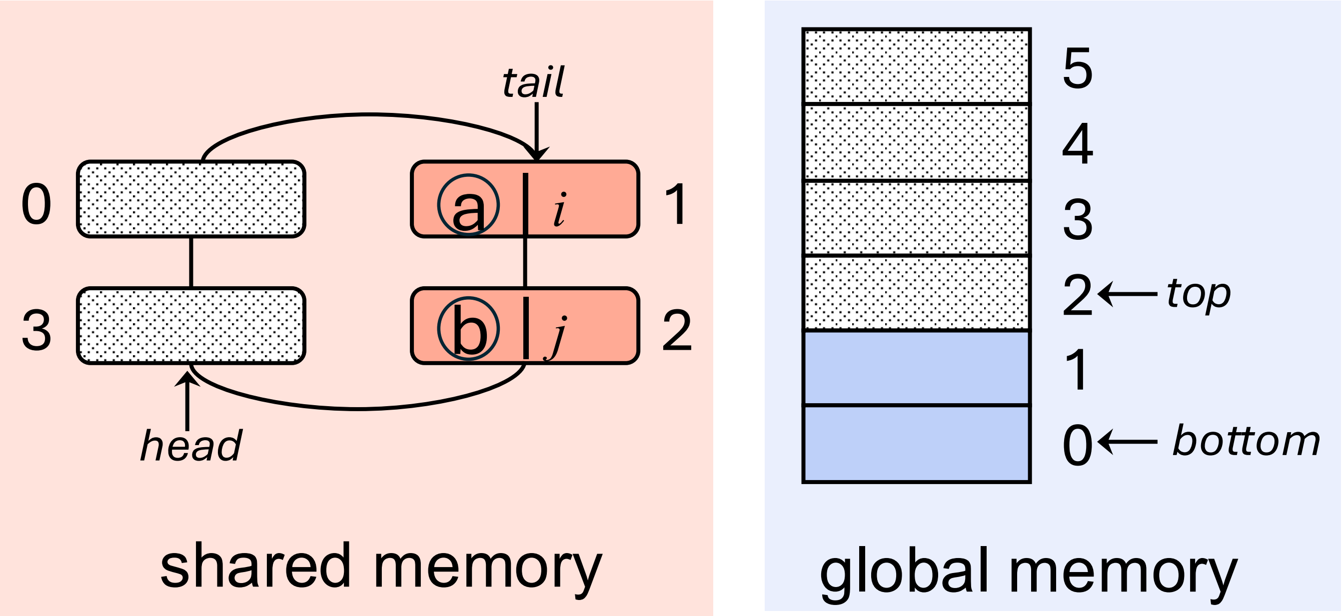
Two-Level Stack Data Structure

HotRing: a circular buffer in shared memory serving as the fast-access portion of the stack.

ColdSeg: a contiguous region in global memory serving as the large-capacity portion of the stack.

head=(1+2)%4=3

top=4-2=2



Four core operations:

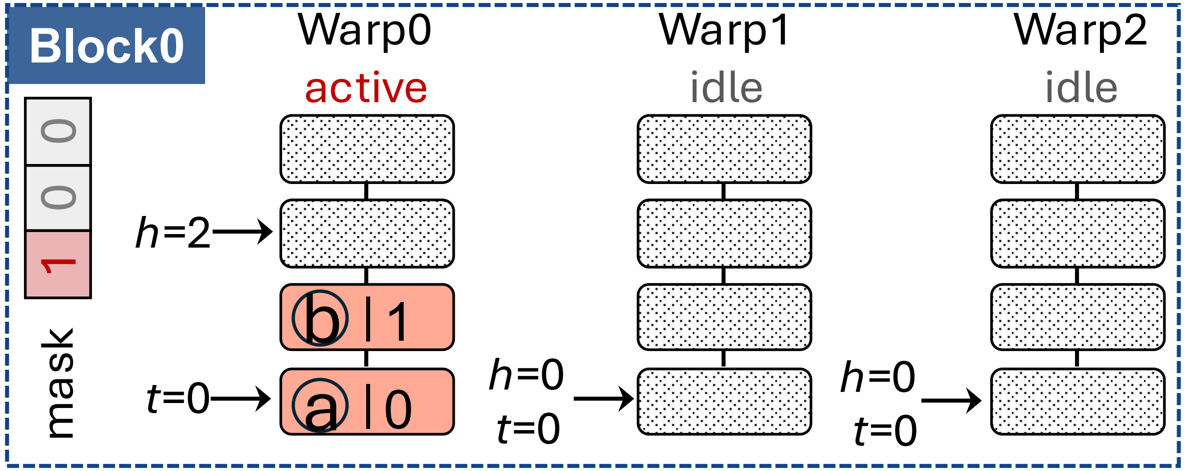
- Fast push: insert a new entry into the *HotRing*.
  head = (*head* + *1*) % *hot_size*
- Fast pop: retrieve the top entry in the HotRing.
  head = (*head – 1* + *hot_size*) % *hot_size*
- Flush: when the HotRing is full, a batch of the oldest entries is moved to the ColdSeg.
  tail = (*tail* + *batch*) % *hot_size,* top = *top* + *batch*
- Refill: When the HotRing is empty, a batch is refilled from the ColdSeg.
  head = (*head* + *batch*) % *hot_size,* top = *top - batch*

DiggerBees Implementation

Intra-Block Work Stealing

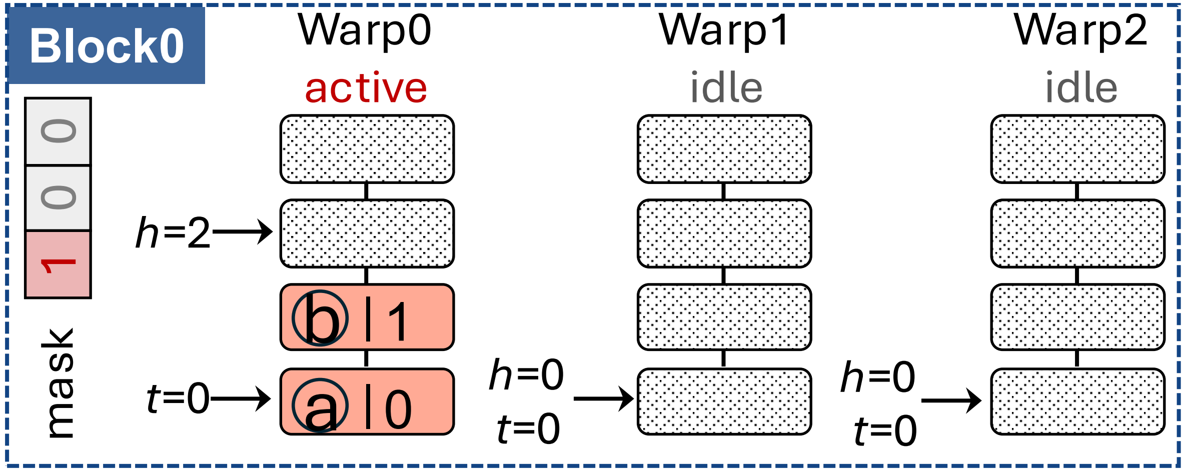
- **Warp-Level Workload: one warp = one DFS worker**

  All 32 threads within a warp follow the same DFS path (no warp divergence)



DiggerBees Implementation

Intra-Block Work Stealing



- **Warp-Level Workload: one warp = one DFS worker**

  All 32 threads within a warp follow the same DFS path (no warp divergence)

- **Idle warp steals locally: three-step mechanism**

  Step1: *victim selection:* An idle warp steals from the deepest HotRing above *hot_cutoff*.

DiggerBees Implementation

Intra-Block Work Stealing

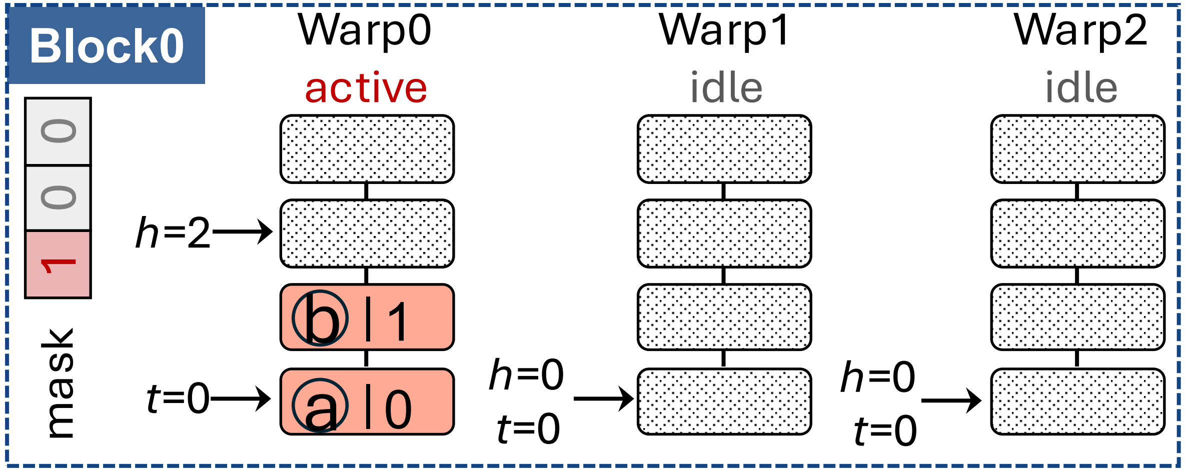
- **Warp-Level Workload: one warp = one DFS worker**

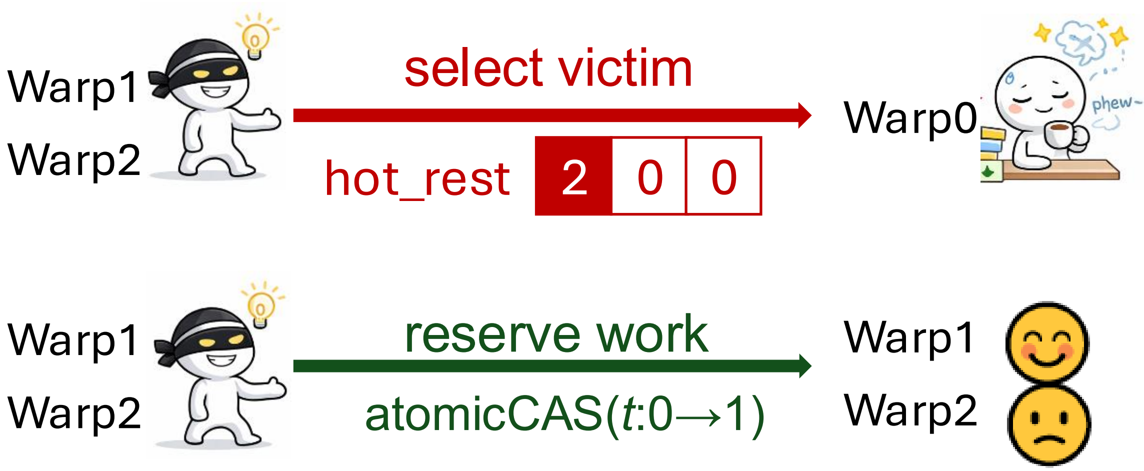
  All 32 threads within a warp follow the same DFS path (no warp divergence)

- **Idle warp steals locally: three-step mechanism**

  Step1: *victim selection:* An idle warp steals from the deepest HotRing above *hot_cutoff*.

  Step2: work reservation*:* The thief claims a batch from the victim's HotRing tail.
  If success: steals *hot_cutoff*/2 entries and updates tail.
  If fail: retry.





DiggerBees Implementation

Intra-Block Work Stealing

- **Warp-Level Workload: one warp = one DFS worker**

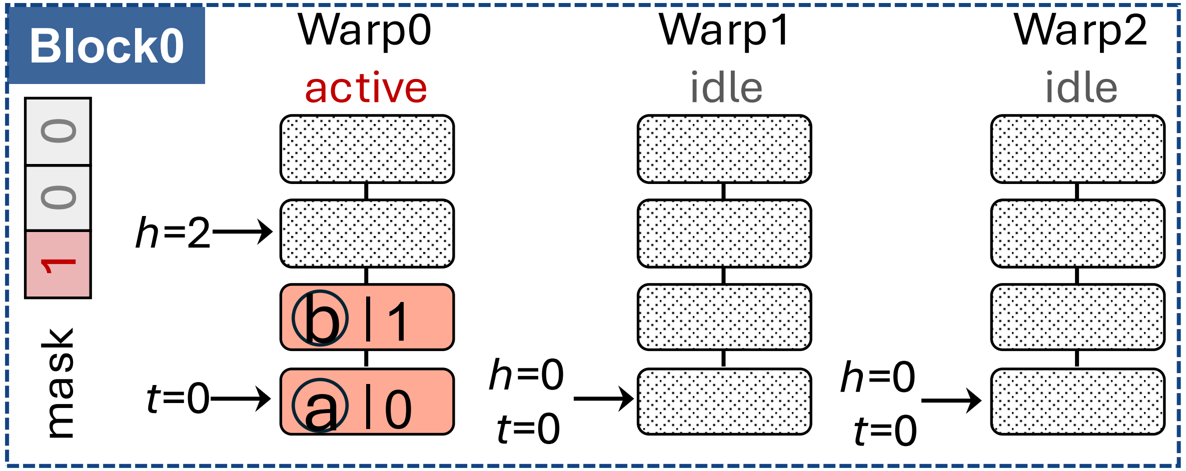
  All 32 threads within a warp follow the same DFS path (no warp divergence)

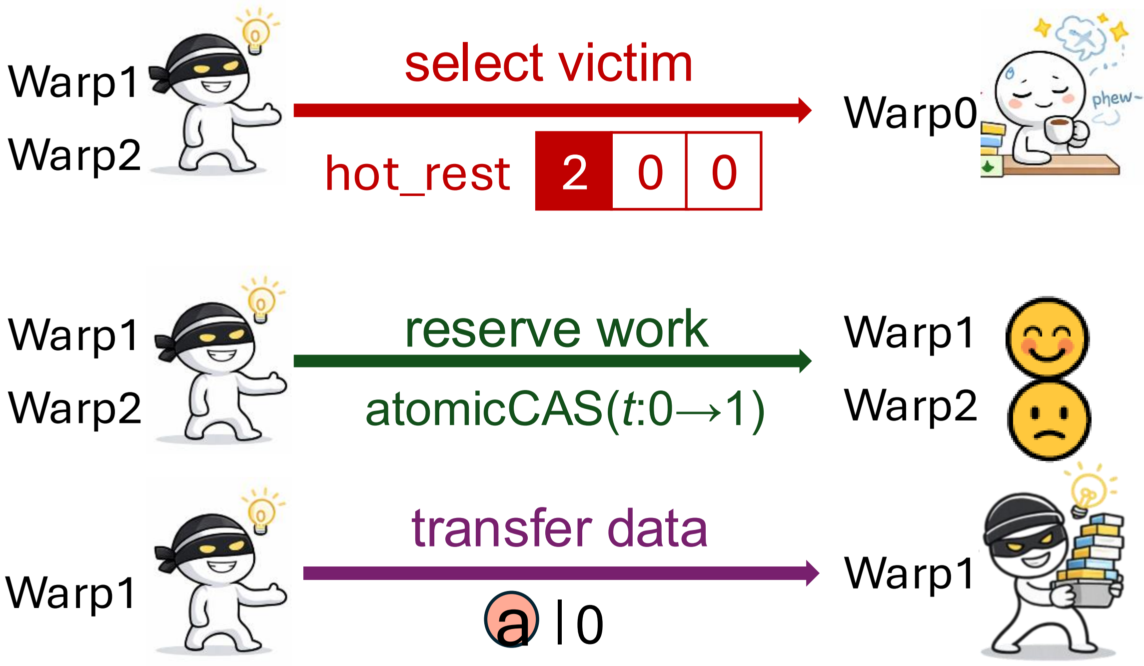
- **Idle warp steals locally: three-step mechanism**

  Step1: *victim selection:* An idle warp steals from the deepest HotRing above *hot_cutoff*.

  Step2: work reservation*:* The thief claims a batch from the victim's HotRing tail.
  If success: steals *hot_cutoff*/2 entries and updates tail.
  If fail: retry.

  Step3: *local transfer:* After a successful claim, the thief copies the batch from the victim's HotRing into its own, updates *head,* and resumes DFS.





DiggerBees Implementation

Intra-Block Work Stealing

- **Warp-Level Workload: one warp = one DFS worker**

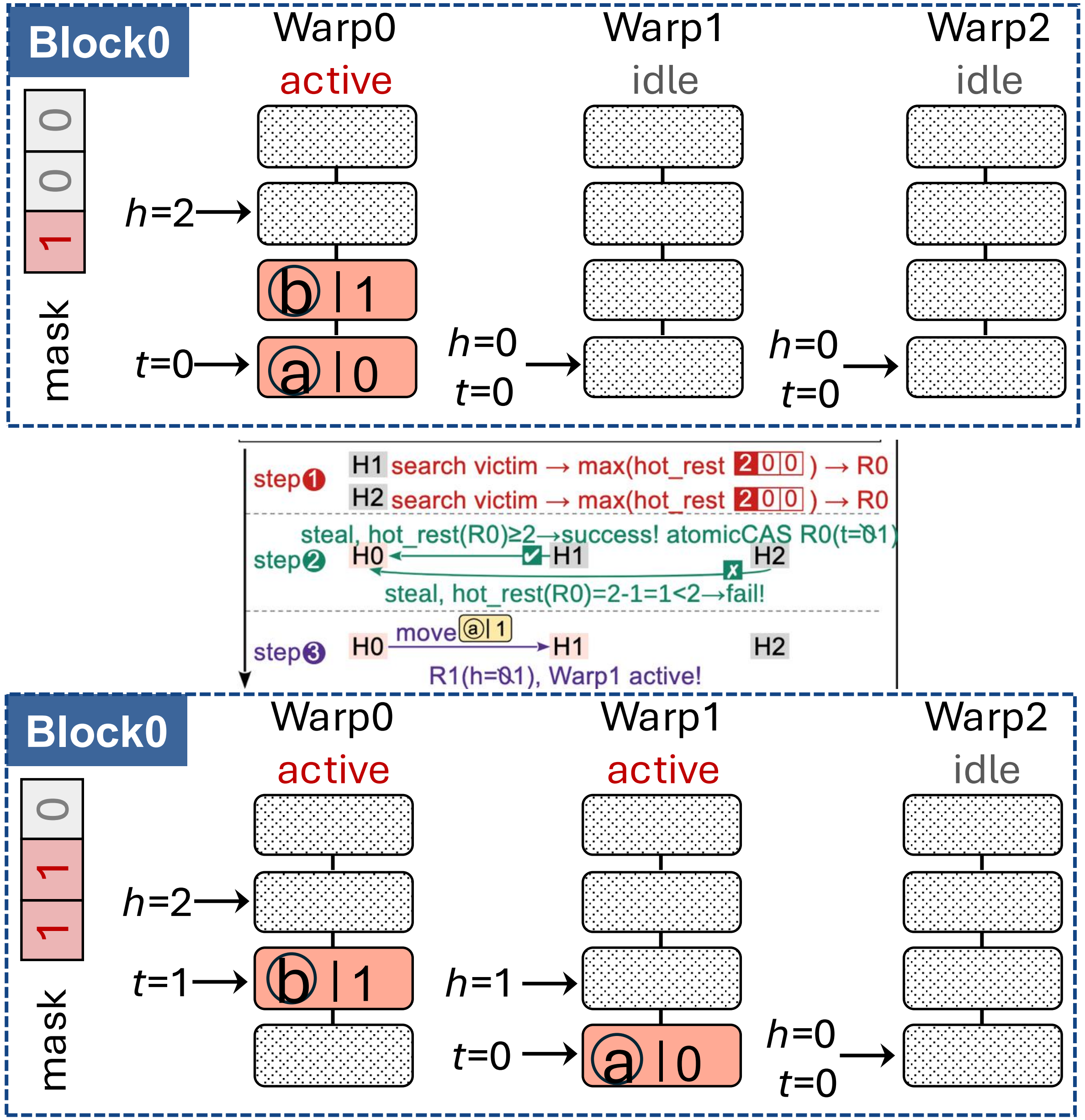
  All 32 threads within a warp follow the same DFS path (no warp divergence)

- **Idle warp steals locally: three-step mechanism**

  Step1: *victim selection:* An idle warp steals from the deepest HotRing above *hot_cutoff*.

  Step2: work reservation*:* The thief claims a batch from the victim's HotRing tail.
  If success: steals *hot_cutoff*/2 entries and updates tail.
  If fail: retry.

  Step3: *local transfer:* After a successful claim, the thief copies the batch from the victim's HotRing into its own, updates *head,* and resumes DFS.



DiggerBees Implementation

Inter-Block Work Stealing

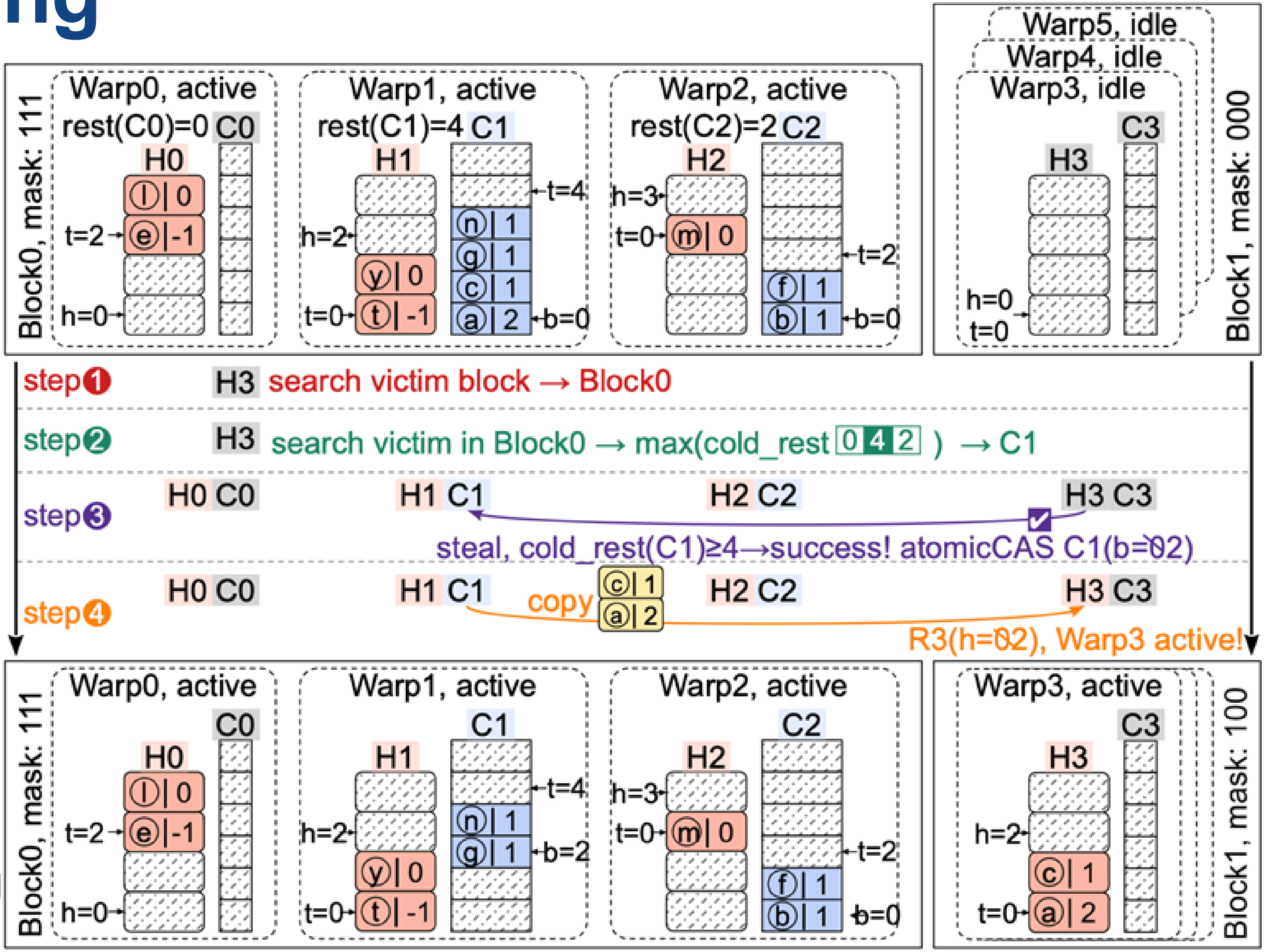
- **When triggered:** a block becomes idle (all warps run out of local work).
- **What to steal:** a batch from the victim warp's **ColdSeg**.
- **How it works:** four-step mechanism

Step1: *victim blcok selection:* power-of-two choices.

Step2: victim warp selection*:* with cold_rest = *top-bottom* and ≥ *cold_cutoff*

Step3: *work reservation:* reserve batch by atomicCAS(*bottom*).

Step4: *remote transfer:* ColdSeg→HotRing



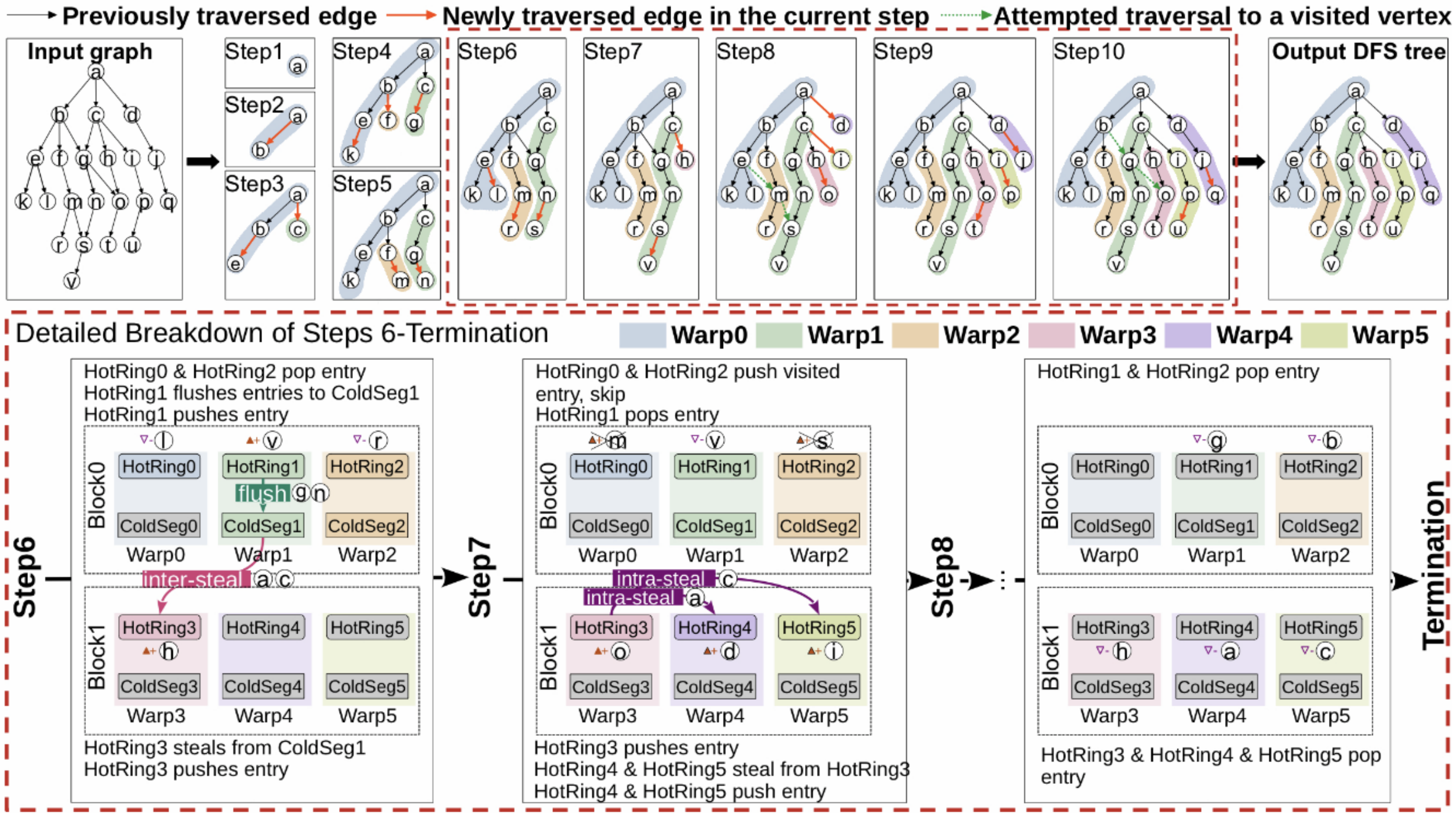
DiggerBees Implementation

An Execution Example

The effectiveness of load balancing:

In Block0:
Warp0: 5 vertices
Warp1: 5 vertices
Warp2: 3 vertices

In Block1:
Warp3: 3 vertices,
Warp4: 3 vertices,
Warp5: 3 vertices



An example of the complete execution flow of DiggerBees, where different colored regions indicate the subtrees explored by different warps.

PPoPP 2026

OUTLINE

Performance Evaluation

Experimental Setup

Method	visited	DFS Tree	Lex-Order	Level
CKL-PDFS	✓	N/A	N/A	N/A
ACR-PDFS	✓	N/A	N/A	N/A
NVG-DFS	✓	✓	**Ordered**	N/A
Gunrock/BerryBees	✓	N/A	N/A	✓
DiggerBees (this work)	✓	✓	**Unordered**	N/A

Platforms: One CPU with a 64-core Intel Xeon Max 687 (9462) processor and two NVIDIA GPUs: A100 (Ampere architecture) and H100 (Hopper architecture).

Tested methods: Two CPU DFS implementations (CKL-PDFS[1] and ACR-PDFS[2]), one GPU DFS implementation (NVG-DFS[3]), and two GPU BFS implementations (Gunrock[4] and BerryBees[5]).

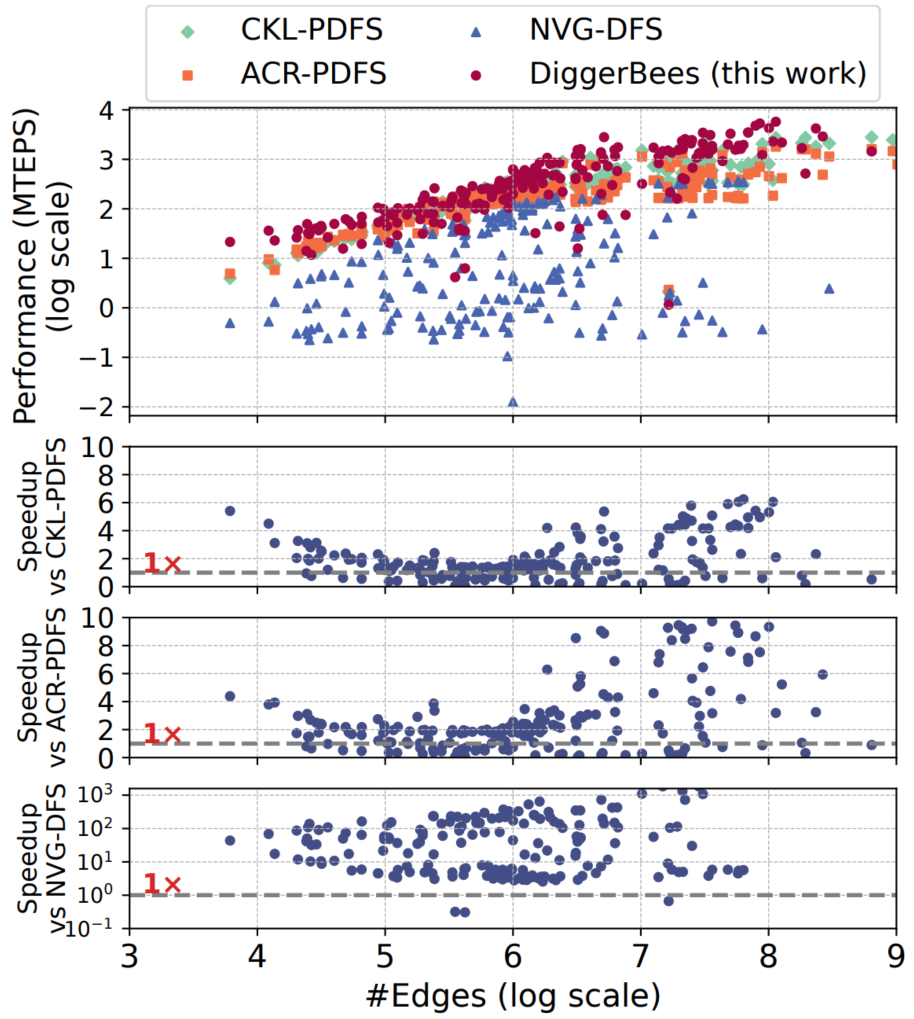
Dataset: all 234 graphs from three widely used graph collections, DIMACS10, SNAP[6], and LAW[7] available in the SuiteSparse Matrix Collection[8].

Group	Count	Description
DIMACS10	151	Benchmark graphs from the 10th DIMACS Implementation Challenge, covering clustering, numerical simulation, and road networks.
SNAP	68	Real-world networks from the Stanford Network Analysis Platform, including social, citation, and web graphs.
LAW	15	Large-scale web graphs from the Laboratory for Web Algorithmics, based on real web crawls and compressed via WebGraph.

[1] Guojing Cong et al.. 2008. Solving Large, Irregular Graph Problems Using Adaptive Work-Stealing. In ICPP ’08. 536–545.
[2] Umut A. Acar et al.. 2015. A workefficient algorithm for parallel unordered depth-first search. In SC ’15. 1–12.
[3] Maxim Naumov et al.. 2017. Parallel Depth-First Search for Directed Acyclic Graphs. In IA3’17.
[4] Yangzihao Wang et al.. 2016. Gunrock: a high-performance graph processing library on the GPU. In PPoPP ’16. 1–12.
[5] Yuyao Niu et al.. 2025. BerryBees: Breadth First Search by Bit-Tensor-Cores. In PPoPP ’25. 339–354.
[6] Marinka Zitnik et al.. 2018. BioSNAP Datasets: Stanford Biomedical Network Dataset Collection.
[7] Paolo Boldi et al.. 2011. Layered Label Propagation: A Multiresolution Coordinate-Free Ordering for Compressing Social Networks. In WWW ’11. 587–596.
[8] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. ACM Trans. Math. Softw. 38, 1 (2011).

Performance Evaluation

Comparison with Existing DFS Approaches



Performance comparison of DiggerBees with three state-of-the-art DFS methods on the H100 GPU.

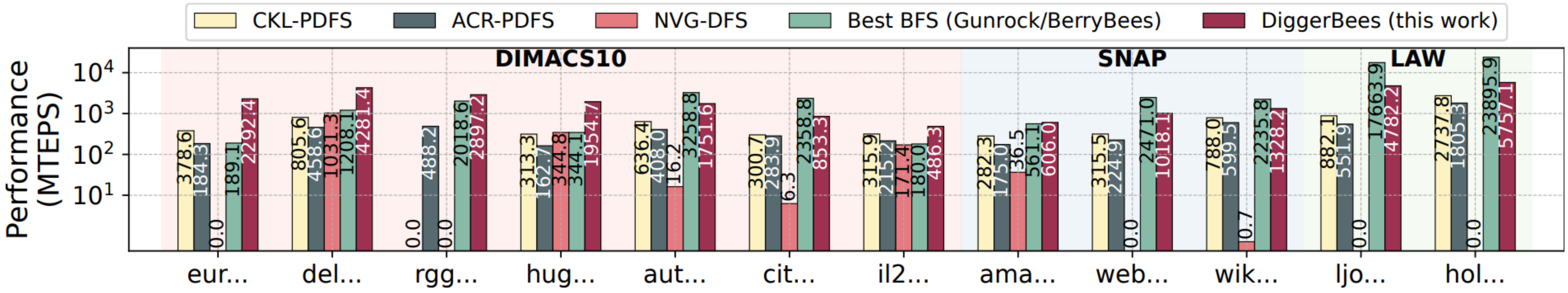
vs. CKL-PDFS (CPU): achieves an average speedup (geometric mean) of **1.37×**, with the best case **6.24×** on **hugebubbles**.

vs. ACR-PDFS (CPU): achieves an average speedup of **1.83×**, with the best case **12.44×** on **euro_osm**.

vs. NVG-DFS (GPU): delivers an average speedup of **30.18×**, reaching **1841.68×** on **higgs-twitter** and **1075.21×** on **soc-Pokec**.

Performance Evaluation

Comparison with Existing BFS Approaches



Performance comparison of four DFS methods and the best BFS baseline (the better-performing result between Gunrock and BerryBees) across 12 representative graphs from three groups on the H100 GPU.

Group	Graph	$\|V\|$	$\|E\|$	Graph	$\|V\|$	$\|E\|$
DIMACS10	euro_osm	50.9M	108.1M	delaunay	16.8M	100.7M
	rgg	16.8M	265.1M	hugebubble	21.2M	63.6M
	auto	0.4M	6.6M	citation	0.3M	2.3M
	il2010	0.5M	2.2M			
SNAP	amazon	0.3M	1.2M	google	0.9M	5.1M
	wiki	1.8M	28.6M			
LAW	ljournal	5.4M	79.0M	hollywood	1.1M	113.9M

Detailed information of 12 representative graphs.

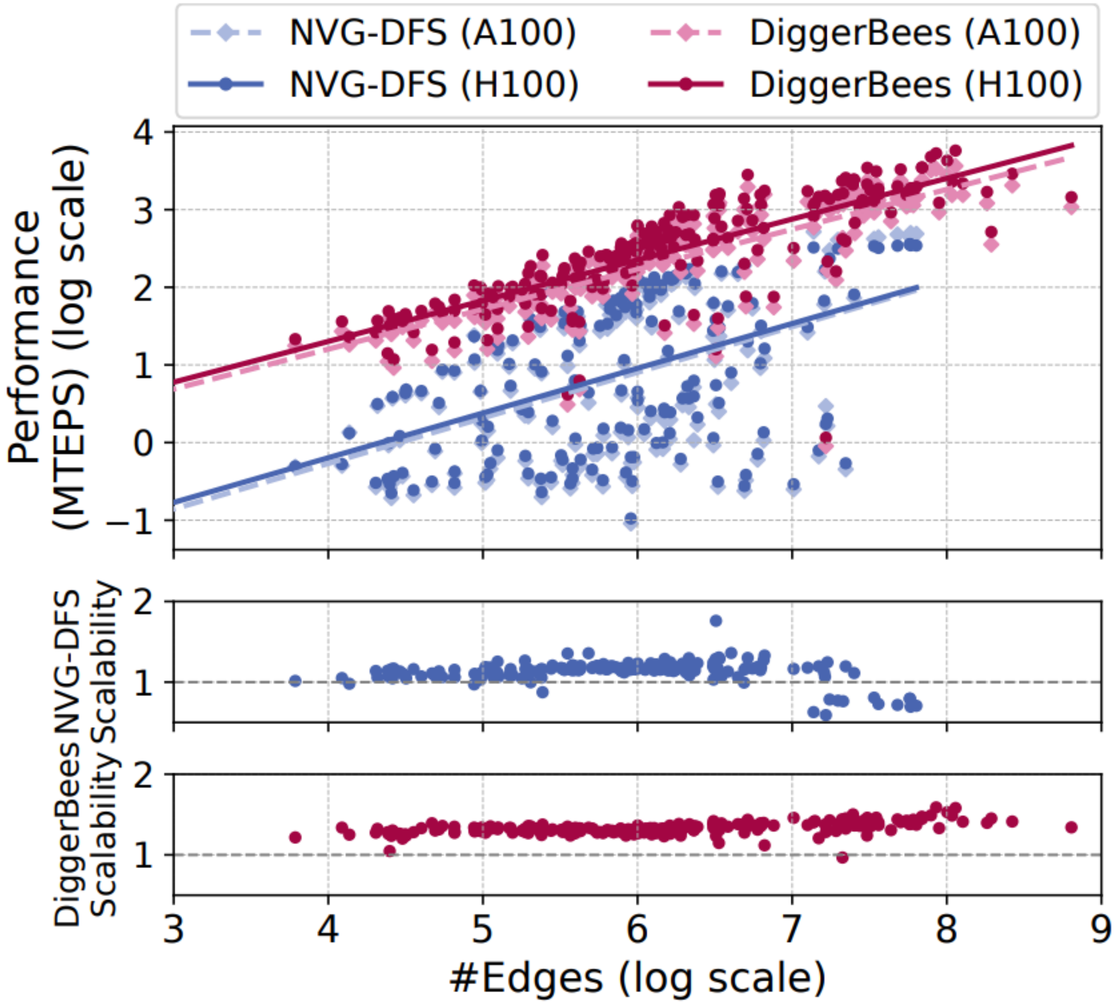
DiggerBees outperforms **Gunrock/BerryBees** on several graphs, e.g., **euro_osm: 12.12× faster**, where BFS requires **17,346 levels**.

Why: Long, narrow traversals hurt BFS; **block-level work stealing** keeps DFS efficient.

Limit: On small-diameter graphs (e.g., **ljournal**, **10** BFS levels), BFS wins; DiggerBees is **3.70×** slower.

Performance Evaluation

Scalability Comparison with GPU DFS



Scalability comparison of DiggerBees and NVGDFS on A100 and H100 GPUs.

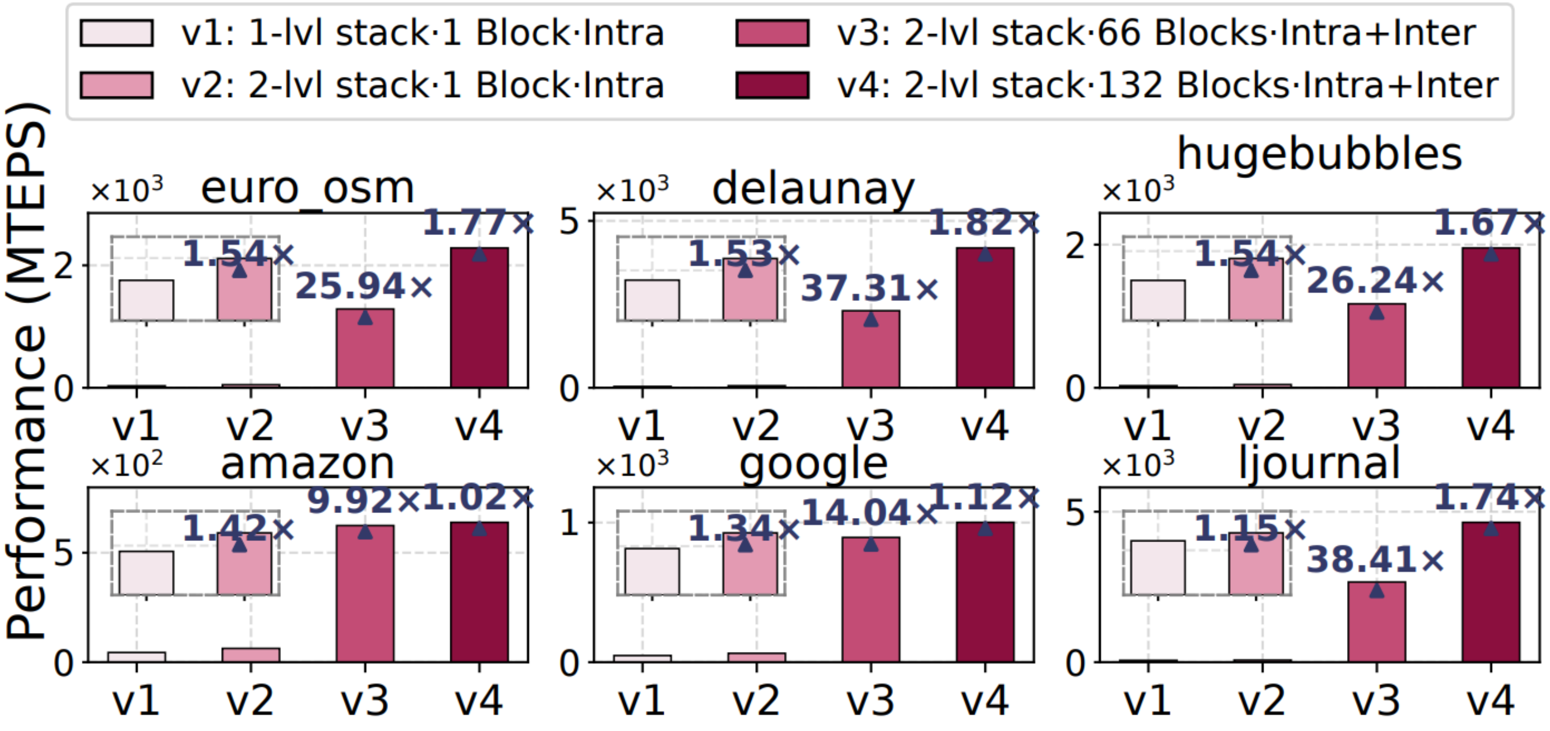
DiggerBees outperforms NVG-DFS on both two GPUS.

Better scaling to H100: average H100-to-A100 speedup is **1.33×** for DiggerBees vs **1.18×** for NVG-DFS.

Why it scales: DiggerBees better utilizes H100's higher compute capacity (**132 SMs vs 108 SMs, +22.2%**), delivering gains that closely track the hardware scaling.

Performance Evaluation

Performance Breakdown



Performance breakdown of four versions of DiggerBees across six representative graphs on the H100 GPU.

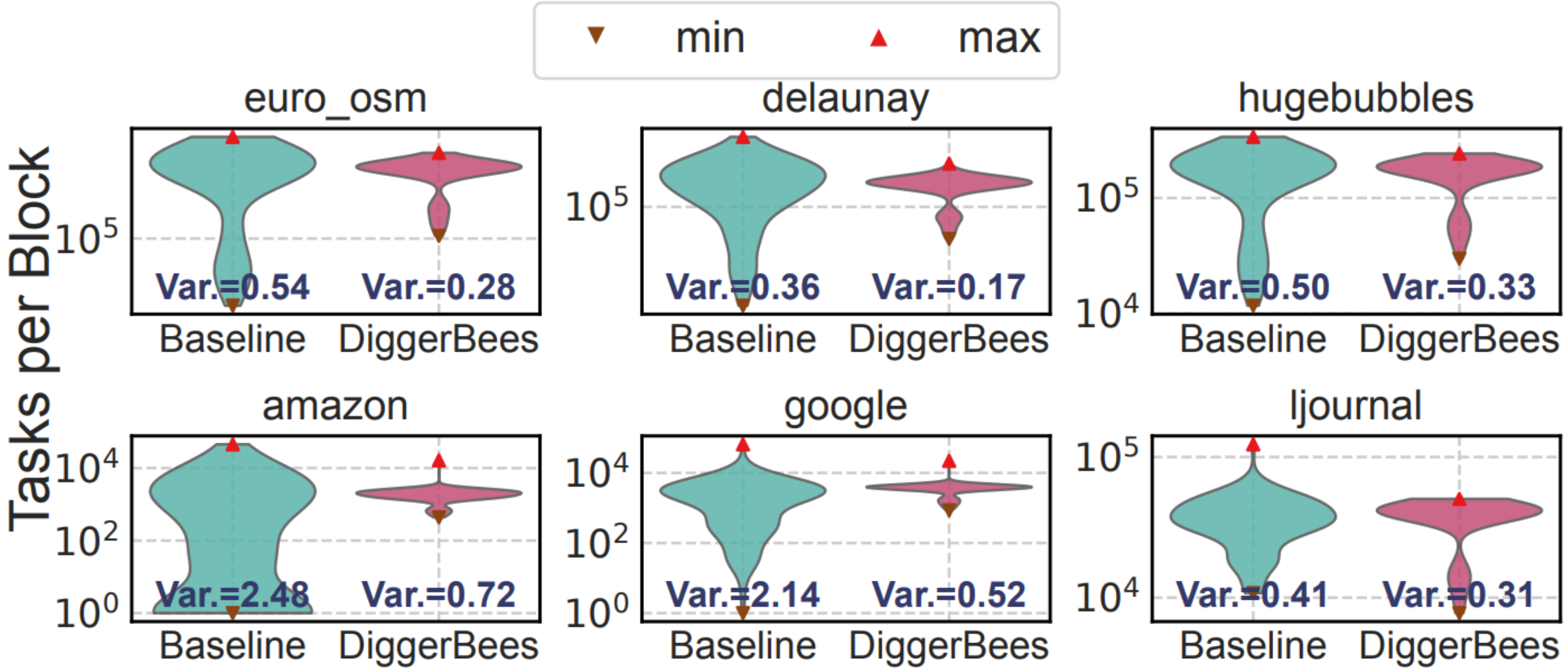
- **v1 → v2 (2-level stack)**: leveraging the memory hierarchy for low-latency stack access yields **~45%** higher throughput on average.

- **v2 → v3 (inter-block work stealing)**: enabling multi-block collaboration brings **dramatic gains** on deep-path graphs, e.g., **25.94×** on *euro_osm* and **37.31×** on *delaunay* (work stealing is key to scaling across SMs).

- **v3 → v4 (scale to all SMs)**: increasing blocks to match SM count provides an additional **67–82%** improvement on most graphs, while small graphs see limited gains **(2–12%)** due to already sufficient parallelism.

Performance Evaluation

Block-Level Load Balance Analysis



Block-level workload distribution for six representative graphs, comparing Baseline (left) and DiggerBees (right).

DiggerBees balances work: consistently reduces variance by **>2×**, e.g., *amazon* drops to **0.72** (**3.44×** lower variance).

Why: load-aware two-choice victim selection + hierarchical work stealing improves **block-level balance**, boosting **scalability and performance**.

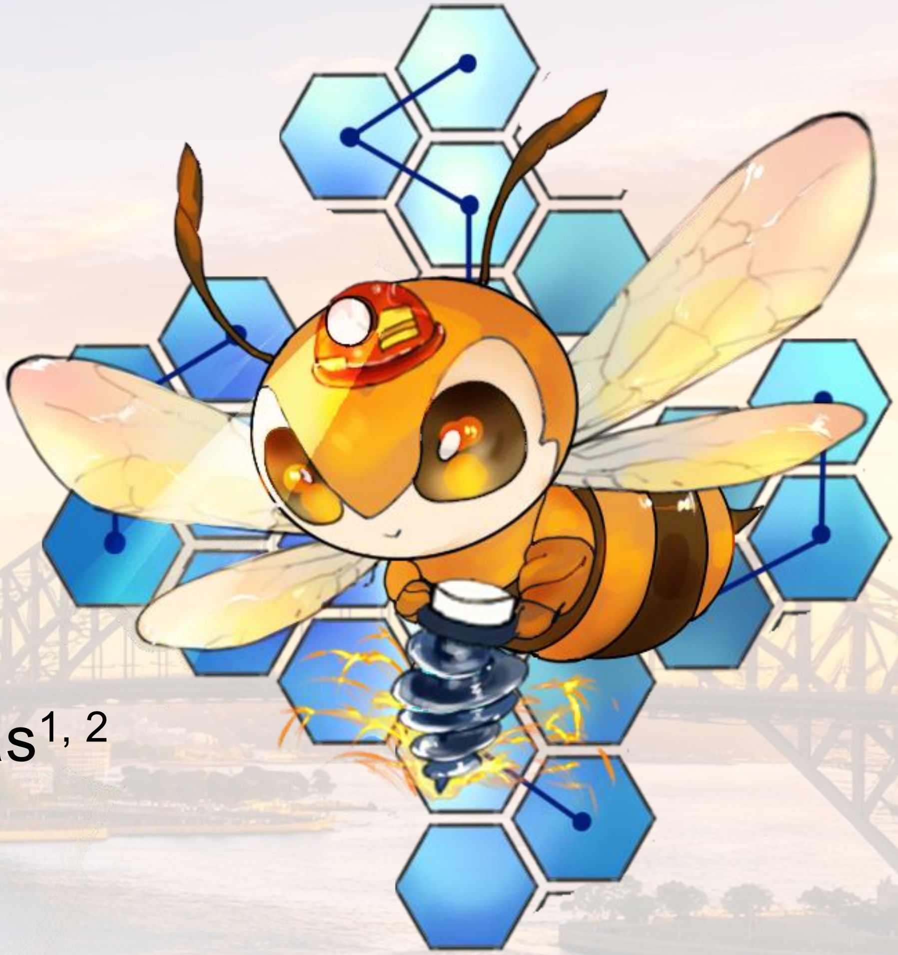
PPoPP 2026

OUTLINE

Conclusion

- We design a two-level stack structure that maps DFS workloads onto the GPU memory hierarchy.
- We develop a hierarchical work-stealing mechanism tailored specifically for DFS traversal on GPUs.
- We achieve significant performance gains over existing approaches on the latest NVIDIA GPUs.

PPoPP 2026



Thanks for Listening!
Any Questions?

Yuyao Niu[1, 2] Yuechen Lu[3] Weifeng Liu[3] Marc Casas[1, 2]

[1] Barcelona Supercomputing Center
[2] Universitat Politècnica de Catalunya
[3] SSSLab, China University of Petroleum-Beijing

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Departament d'Arquitectura de Computadors

中国石油大学
CHINA UNIVERSITY OF PETROLEUM

Lab
超级科学软件实验室
Super Scientific Software Laboratory

Sydney, Australia • February 2, 2026

https://doi.org/10.5281/zenodo.17709254