Is Acyclic Directed Graph Partitioning Effective for Locality-Aware Scheduling?

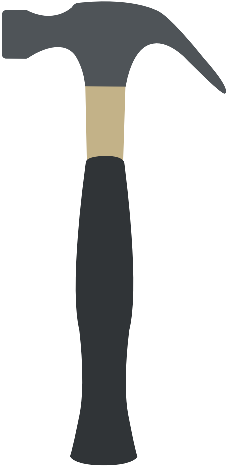
M. Yusuf Özkaya[1], Anne Benoit[1,2], **Ümit V. Çatalyürek**[1]

[1]School of Computational Science and Engineering,
Georgia Institute of Technology, GA, USA

[2]LIP, ENS Lyon, France

14th Scheduling for Large Scale Systems Workshop
June 26-28, 2019 – Bordeaux, France

Introduction



Outline

Outline

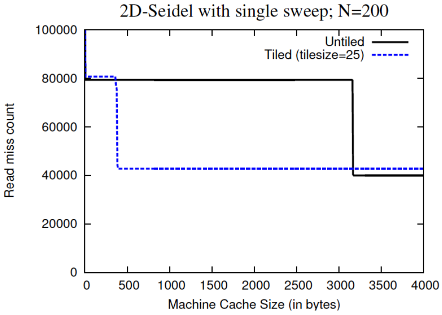
Computational vs Data Move Complexity

```
for (i=1; i<N-1; i++)
  for (j=1;j<N-1; j++)
    A[i][j] = A[i][j-1] + A[i-1][j];
```

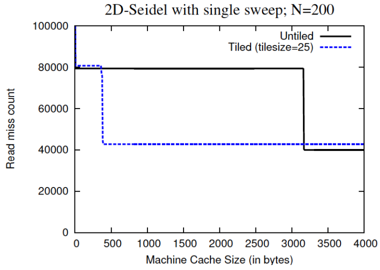
Untiled version

```
for(it = 1; it<N−1; it +=B)
 for(jt = 1; jt<N−1; jt +=B)
  for(i = it; i < min(it+B, N−1); i++)
   for(j = jt; j < min(jt+B, N−1); j++)
    A[i][j] = A[i−1][j] + A[i][j−1];
```

Tiled Version

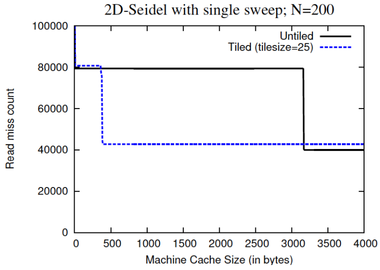


Computational vs Data Move Complexity



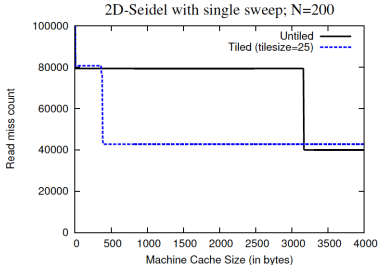
- Both have Comp. Complexity $(N-1)^2$ OPs.
  - Data movement cost different for two versions
  - Also depends on cache size

Computational vs Data Move Complexity



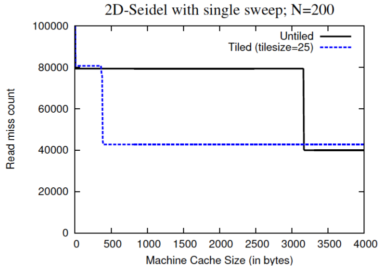
- Both have Comp. Complexity $(N-1)^2$ OPs.
  - Data movement cost different for two versions
  - Also depends on cache size
- Question: Can we achieve lower cache misses than this tiled version? How can we know when much further improvement is not possible?

Computational vs Data Move Complexity



- Both have Comp. Complexity $(N-1)^2$ OPs.
  - Data movement cost different for two versions
  - Also depends on cache size
- Question: Can we achieve lower cache misses than this tiled version? How can we know when much further improvement is not possible?
- Question: What is the lowest achievable data movement cost among all possible equivalent versions of a #computation?

Computational vs Data Move Complexity



- Both have Comp. Complexity $(N-1)^2$ OPs.
  - Data movement cost different for two versions
  - Also depends on cache size
- Question: Can we achieve lower cache misses than this tiled version? How can we know when much further improvement is not possible?
- Question: What is the lowest achievable data movement cost among all possible equivalent versions of a #computation?
- Current performance tools and methodologies do not address this

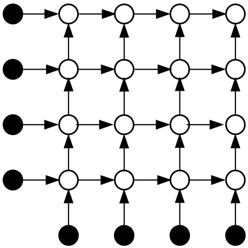
Modeling Data Move Complexity: DAG

```
for (i=1; i<N-1; i++)
  for (j=1;j<N-1; j++)
    A[i][j] = A[i][j-1] + A[i-1][j];
```

Untiled version

```
for(it = 1; it<N−1; it +=B)
  for(jt = 1; jt<N−1; jt +=B)
    for(i = it; i < min(it+B, N−1); i++)
      for(j = jt; j < min(jt+B, N−1); j++)
        A[i][j] = A[i−1][j] + A[i][j−1];
```

Tiled Version



DAG for N=6

- DAG abstraction: Vertex = operation, edges = data dep.

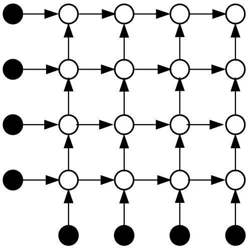
Modeling Data Move Complexity: DAG

```
for (i=1; i<N-1; i++)
  for (j=1;j<N-1; j++)
    A[i][j] = A[i][j-1] + A[i-1][j];
```

Untiled version

```
for(it = 1; it<N−1; it +=B)
 for(jt = 1; jt<N−1; jt +=B)
  for(i = it; i < min(it+B, N−1); i++)
   for(j = jt; j < min(jt+B, N−1); j++)
    A[i][j] = A[i−1][j] + A[i][j−1];
```

Tiled Version



DAG for N=6

- DAG abstraction: Vertex = operation, edges = data dep.
- 2-level memory hierarchy with S fast mem locs. & infinite slow mem. locs.
  - To compute a vertex, predecessor must hold values in fast mem.
  - Limited fast memory ⇒ computed values may need to be temporarily stored in slow memory and reloaded

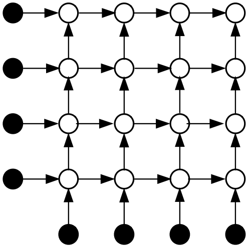
Modeling Data Move Complexity: DAG

```
for (i=1; i<N-1; i++)
  for (j=1;j<N-1; j++)
    A[i][j] = A[i][j-1] + A[i-1][j];
```

Untiled version

```
for(it = 1; it<N−1; it +=B)
  for(jt = 1; jt<N−1; jt +=B)
    for(i = it; i < min(it+B, N−1); i++)
      for(j = jt; j < min(jt+B, N−1); j++)
        A[i][j] = A[i−1][j] + A[i][j−1];
```

Tiled Version



DAG for N=6

- DAG abstraction: Vertex = operation, edges = data dep.
- 2-level memory hierarchy with S fast mem locs. & infinite slow mem. locs.
  - To compute a vertex, predecessor must hold values in fast mem.
  - Limited fast memory $\Rightarrow$ computed values may need to be temporarily stored in slow memory and reloaded
- Data movement complexity of DAG: Minimal #loads+#stores among all possible valid schedules.

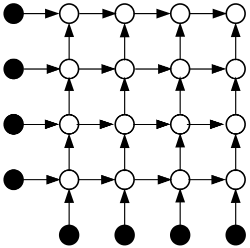
Modeling Data Move Complexity: DAG

```
for (i=1; i<N-1; i++)
  for (j=1;j<N-1; j++)
    A[i][j] = A[i][j-1] + A[i-1][j];
```

Untiled version

```
for(it = 1; it<N−1; it +=B)
  for(jt = 1; jt<N−1; jt +=B)
    for(i = it; i < min(it+B, N−1); i++)
      for(j = jt; j < min(jt+B, N−1); j++)
        A[i][j] = A[i−1][j] + A[i][j−1];
```

Tiled Version



DAG for N=6

Develop upper bounds on min-cost

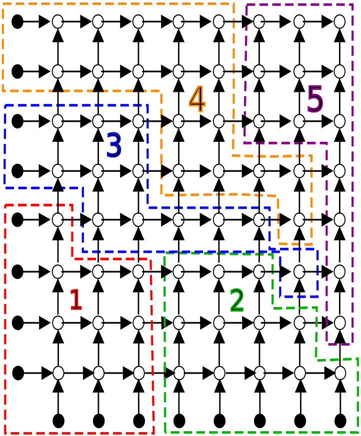
Minimum possible data movement cost?

No known effective solution to problem

Develop lower bounds on min-cost

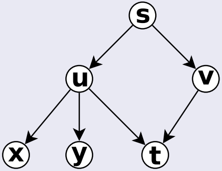
Data Movement Upper Bounds

- Perform acyclic partitioning of the DAG
- Assign each node in a single acyclic part
- Acyclic partitioning of a DAG $\approx$ Tiling the iteration space
- Each part is acyclic
  - Can be executed atomically
  - No cyclic data dependence among parts
- Topologically sorted order of the acyclic parts $\Rightarrow$ a valid execution order
- **Hammer = Acyclic DAG Partitioner.**

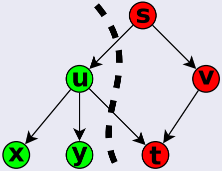


Outline

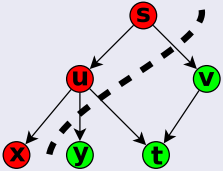
Acycling DAG Partitioning



(a) A toy graph



(b) A partition ignoring the directions; it is cyclic.



(c) An acyclic partitioning.

A Multilevel Acyclic DAG Partitioning

- Recursive bisection.
- Multilevel: coarsening, initial partitioning, refinement: all acyclic.

[SISC'19]: Herrmann, Özkaya, Uçar, Kaya, Ç, "Multilevel Algorithms for Acyclic Partitioning of Directed Acyclic Graphs", SIAM Journal on Scientific Computing,to appear.

Objectives and Constraints

Objectives

- Minimize the edge cut between components
- Minimize the total volume of communication between components (edge cut counting edges coming from a same node only once)
- There should exist a traversal of the graph such that **alive** data fit into the cache at any moment

Constraints

- Upper bound on the weights of the part
- Upper bound on the weight of each part plus the sum of weights of the boundary vertices that are sources of the part's incoming edges
- There should exist a traversal of the graph such that **alive** data fit into the cache at any moment

Outline

Problem

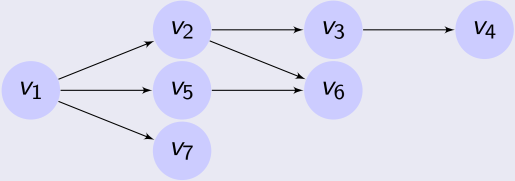
Model

- Directed acyclic task graph: $G = (V, E)$
  w_i: is vertex weight – $c_{i,j}$: communication cost
- For $v_i \in V$,
  - predecessors: $pred_i = \{v_j \mid (v_j, v_i) \in E\}$
  - succesors: $succ_i = \{v_j \mid (v_i, v_j) \in E\}$
  - cannot start until all predecessors have completed,
  - size of (scratch) memory: w_i
  - produces a data of size out_i that will be communicated to all of its successors, i.e., $c_{i,j} = out_i$.

- Fast memory is C, and slow memory is large enough.
- In order to compute task $v_i \in V$, the processor must access $in_i + w_i + out_i$ fast memory locations.
- Because of the limited fast memory, some computed values may need to be temporarily stored in slow memory and reloaded later.

An example

For simplicity in the presentation: $w_i = 0$ and $out_i = 1$. Hence, total input size of task v_i is $in_i = |pred_i|$.

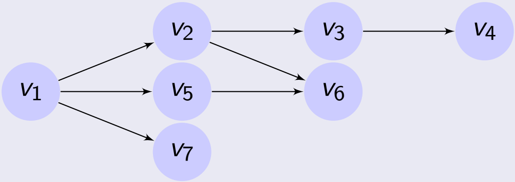


Sample execution order

vertex	v_1	v_2	v_3	v_4
data size	1			

An example

For simplicity in the presentation: $w_i = 0$ and $out_i = 1$. Hence, total input size of task v_i is $in_i = |pred_i|$.

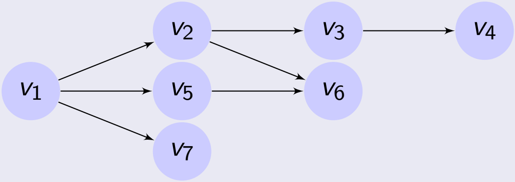


Sample execution order

vertex	v_1	v_2	v_3	v_4
data size	1	2		

An example

For simplicity in the presentation: $w_i = 0$ and $out_i = 1$. Hence, total input size of task v_i is $in_i = |pred_i|$.

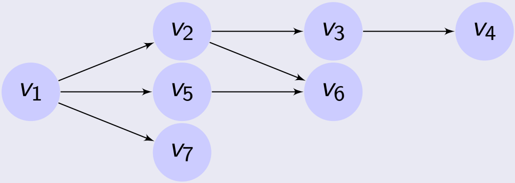


Sample execution order

vertex	v_1	v_2	v_3	v_4
data size	1	2	3	

An example

For simplicity in the presentation: $w_i = 0$ and $out_i = 1$. Hence, total input size of task v_i is $in_i = |pred_i|$.

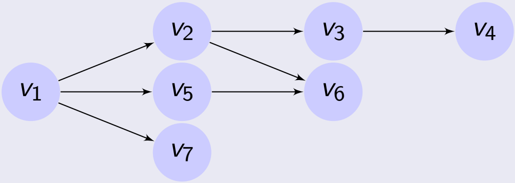


Sample execution order

vertex	v_1	v_2	v_3	v_4
data size	1	2	3	4

An example

For simplicity in the presentation: $w_i = 0$ and $out_i = 1$. Hence, total input size of task v_i is $in_i = |pred_i|$.



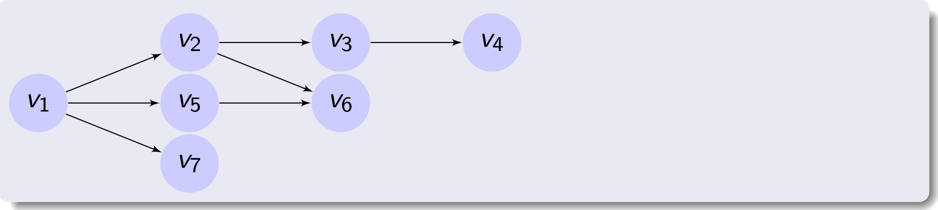
Sample execution order

vertex	v_1	v_2	v_3	v_4
data size	1	2	3	4

if $C = 3$, one will need to evict a data from the cache, hence resulting in a **cache miss**.

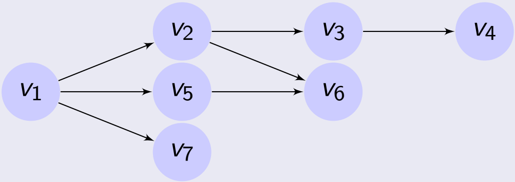
An example: livesize

livesize: *live set size* is defined as the minimum cache size required for the execution so that there are no cache misses.



An example: livesize

livesize: *live set size* is defined as the minimum cache size required for the execution so that there are no cache misses.

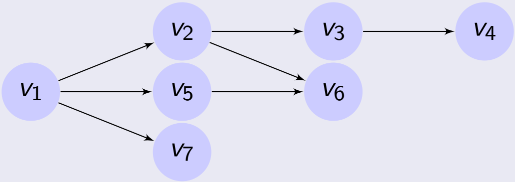


Traversals

- traversal $v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4 \rightarrow v_5 \rightarrow v_6 \rightarrow v_7$, liveset $= 4$.

An example: livesize

livesize: *live set size* is defined as the minimum cache size required for the execution so that there are no cache misses.

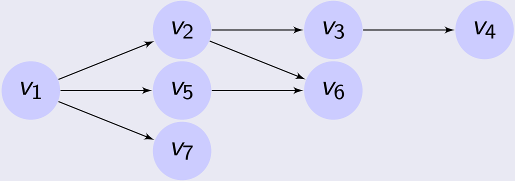


Traversals

- traversal $v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4 \rightarrow v_5 \rightarrow v_6 \rightarrow v_7$, liveset $= 4$.
- For another traversal, $v_1 \rightarrow v_7 \rightarrow v_2 \rightarrow v_5 \rightarrow v_6 \rightarrow v_3 \rightarrow v_4$, livesize $= 3$.

An example: livesize

livesize: *live set size* is defined as the minimum cache size required for the execution so that there are no cache misses.



Traversals

- traversal $v_1 \to v_2 \to v_3 \to v_4 \to v_5 \to v_6 \to v_7$, liveset $= 4$.
- For another traversal, $v_1 \to v_7 \to v_2 \to v_5 \to v_6 \to v_3 \to v_4$, livesize $= 3$.
  This is the minimum cache size to execute this DAG, since task v_6 requires 3 cache locations to be executed.

Outline

Parts, Cuts, Traversals

Parts

Consider an acyclic k-way partition $P = \{V_1, \ldots, V_k\}$ of the DAG $G = (V, E)$:

- the set of vertices V is divided into k disjoint subsets, or *parts*
- There is a path between V_i and V_j ($V_i \rightsquigarrow V_j$) if and only if there is a path in G between a vertex $v_i \in V_i$ and a vertex $v_j \in V_j$.

Cuts

- **cut edge**: if its endpoints are in different parts.
- Let $E_{cut}(P)$ be the set of cut edges for this partition.
- The **edge cut** of a partition:

$$\texttt{EdgeCut}(P) = \sum_{(v_i, v_j) \in E_{cut}(P)} c_{i,j}.$$

Traversals, Livesize

Traversal

Let $V_i \subseteq V$ be a *part* of the DAG $(1 \leq i \leq k)$.
$\tau(V_i)$ **: a traversal** of the part V_i is an ordered list of the vertices that respect precedence constraints within the part:
if there is an edge $(v, v') \in E$, then v must appear before v' in the traversal.

Livesize

Given a part V_i and a traversal of this part $\tau(V_i)$
$L(\tau(V_i))$**: livesize of the traversal** is the maximum memory usage required to execute the whole part.

We define $L(\tau(V_i))$ as the livesize computed such that inputs and outputs (of part V_i) are evicted from the cache if they are no longer required inside the part.

Cache Eviction, Optimization Problem

Cache Eviction

- During execution, if the livesize is greater than the cache size C some data must be transferred from the cache back into slow memory.
- The data that will be evicted may affect the number of cache misses.
- Given a traversal, the optimal strategy consists in evicting the data whose next use will occur farthest in the future during execution [Belady IBM SysJ'66].

MinCacheMiss

- Given a DAG G, a cache of size C, find a topological order of G that minimizes the number of cache misses when using the OPT strategy.
- Finding the optimal traversal to minimize the livesize is an NP-complete problem [Sethi STOC'73], even though it is polynomial on trees [Jacquelin et al. IPDPS'11].

DAG-Assisted Locality-Aware Scheduling

Instead of looking for a global traversal of the whole graph, we propose to partition the DAG in an acyclic way.

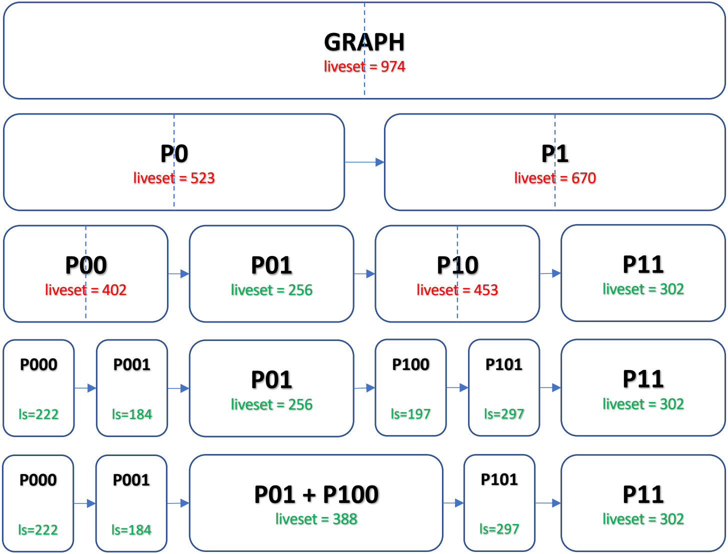
The key is, then, to have all the parts executable without cache misses, hence the only cache misses can be incurred by data on the cut between parts.

Therefore, we aim at minimizing the edge cut of the partition.

Traversals Used

- *Natural Ordering (Nat)* treats the node id's as the priority of the node, where the lower id has a higher priority, hence the traversal is $v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_n$, except if node id's do not follow precedence constraints (schedule ready task of highest priority first).
- *DFS Traversal Ordering (Dfs)* follows a depth-first traversal strategy among the ready tasks.
- *BFS Traversal Ordering (Bfs)* follows a breadth-first traversal strategy among the ready tasks.

Recursive bisection with target Liveset Size



Outline

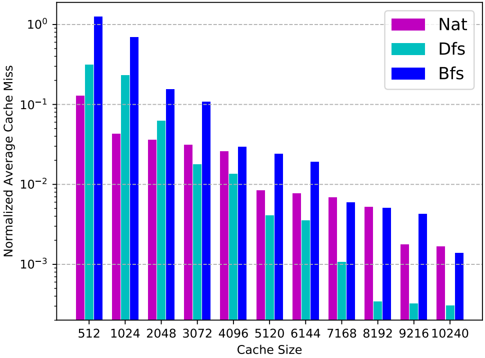
Graph Instances

Instances from the SuiteSparse Matrix Collection (formerly know as UFL):

Graph	$\|V\|$	$\|E\|$	$\max_{in.deg}$	$\max_{out.deg}$	L_{Nat}	L_{Dfs}	L_{Bfs}
`144`	144,649	1,074,393	21	22	74,689	31,293	29,333
`598a`	110,971	741,934	18	22	81,801	41,304	26,250
`caidaRouterLev.`	192,244	609,066	321	1040	56,197	34,007	35,935
`coAuthorsCites.`	227,320	814,134	95	1367	34,587	26,308	27,415
`delaunay-n17`	131,072	393,176	12	14	32,752	39,839	52,882
`email-EuAll`	265,214	305,539	7,630	478	196,072	177,720	205,826
`fe-ocean`	143,437	409,593	4	4	8,322	7,099	3,716
`ford2`	100,196	222,246	29	27	26,153	4,468	25,001
`halfb`	224,617	6,081,602	89	119	66,973	25,371	38,743
`luxembourg-osm`	114,599	119,666	4	5	4,686	2,768	6,544
`rgg-n-2-17-s0`	131,072	728,753	18	19	759	1,484	1,544
`usroads`	129,164	165,435	4	5	297	8,024	9,789
`vsp-finan512.`	139,752	552,020	119	666	25,830	24,714	38,647
`vsp-mod2-pgp2.`	101,364	389,368	949	1726	41,191	36,902	36,672
`wave`	156,317	1,059,331	41	38	13,988	22,546	19,875

Note that when reporting the cache miss counts, we do not include **compulsory (cold, first reference) misses**, the misses that occur at the first reference to a memory block, as these misses cannot be avoided.

Performance of the three baseline traversal algorithms

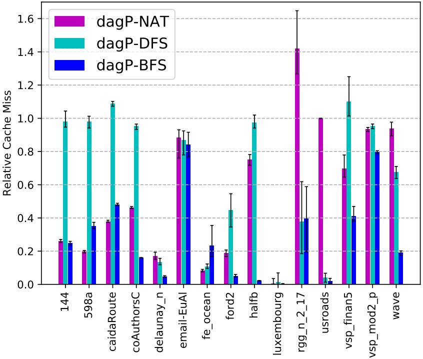


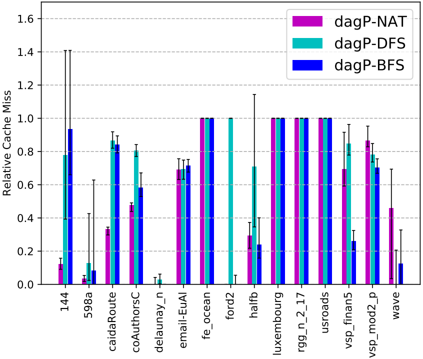
In smaller cache sizes, *Nat* is best.

As the cache size increases, after 3072, *Dfs traversal is best.*

Relative Cache Miss

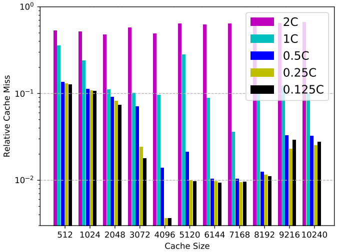
Relative cache misses (geomean of average of 50 runs) for each graph separately (left cache size 512; right cache size 10240).

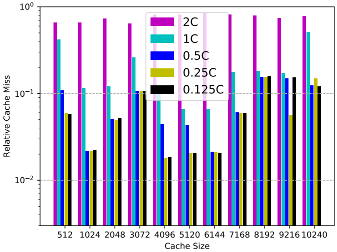


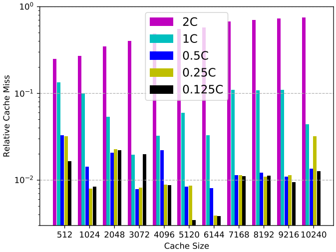


Effect of L_m and C on Cache Miss Improvement

Relative cache misses of DAGP-* with the given partition livesize for *Nat* (left), *Dfs* (right), and *Bfs* (bottom) traversals.

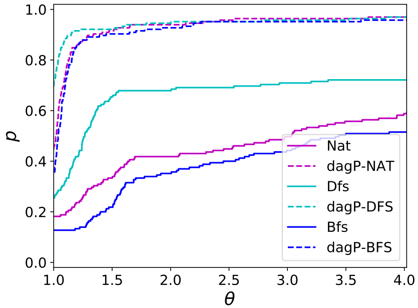


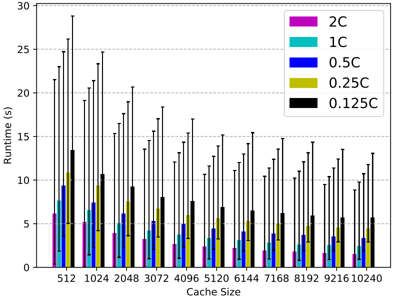




Performance Profiles and Runtime

(Left) Performance profile comparing baselines and heuristics with $L_m = 0.5 \times C$.
(Right) Average runtime of all graphs for dagP-dfs partitioning.





Outline

Conclusion and Future work

Conclusion

- A DAG-partitioning assisted approach for improving data locality.
- Experimental evaluation shows significant reduction in the number of cache misses.

Future Work

- Study the effect of a customized DAG-partitioner specifically for cache optimization purposes
- Design traversal algorithms to optimize cache misses.
- Use a better fitting **directed hypergraph** representation for the model.

Thanks

Thanks

To P. Sadayappan for sharing his motivation slides.

More information

contact : umit@gatech.edu

visit: `http://cc.gatech.edu/~umit` or `http://tda.gatech.edu`