Improving Locality-Aware Scheduling with Acyclic Directed Graph Partitioning

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

PPAM 2019
September 8-11, 2019 – Bialystok, Poland

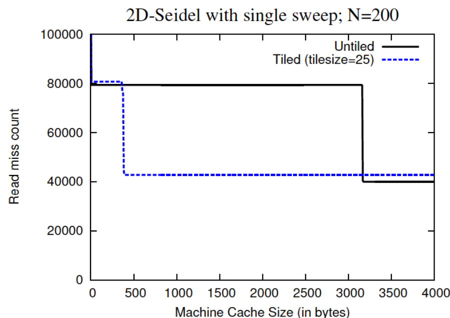
Complexity of computations vs data movements

```
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];
```

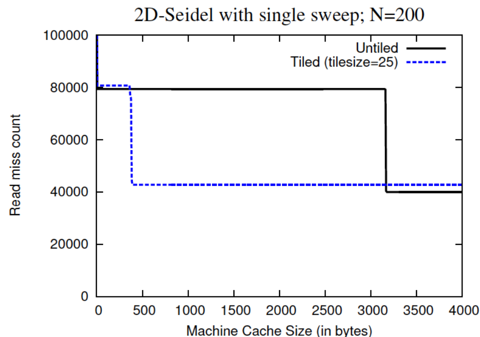
Untiled version

```
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



Complexity of computations vs data movements



- 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

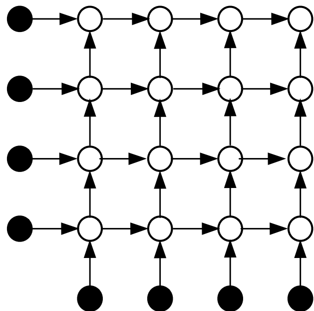
Modeling data move complexity: DAG

```
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.
- 2-level memory hierarchy with C **fast mem. locations** and infinite slow mem. locations
  - To compute a vertex, predecessor must hold values in fast memory
  - Limited fast memory ⇒ computed values may need to be temporarily stored in slow memory and reloaded
- Data movement complexity of DAG:
  Min. #loads+#stores among all possible valid schedules

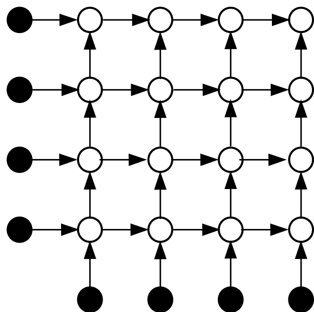
Modeling data move complexity: DAG

```
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

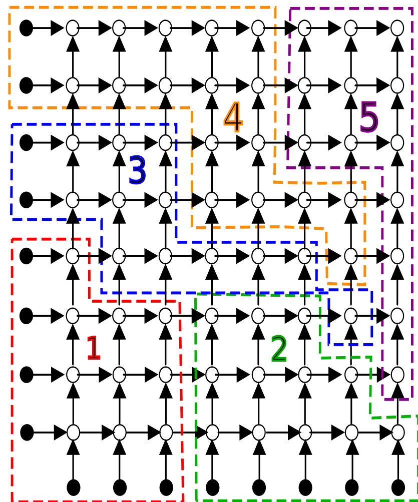
Minimum possible data movement cost?

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 ≈ 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
  ⇒ a valid execution order
- **Rely on Acyclic DAG Partitioner**



Outline

1 Model

2 Scheduling strategies and experiments

3 Conclusion

Problem

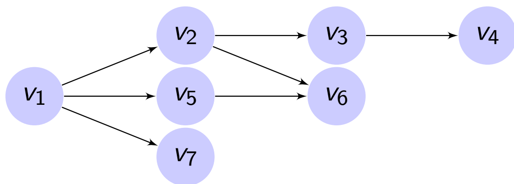
Model

- Directed acyclic task graph: $G = (V, E)$
- For $v_i \in V$,
  - predecessors: $pred_i = \{v_j \mid (v_j, v_i) \in E\}$; cannot start until all predecessors have completed
  - successors: $succ_i = \{v_j \mid (v_i, v_j) \in E\}$
  - size of (scratch) memory: w_i
  - produces a data of size out_i that will be communicated to all of its successors
  - total size of input: $in_i = |pred_i|$ if $out_j = 1$ for all tasks

- Fast memory of size C, and slow memory large enough
- Compute $v_i \in V$: must access $in_i + w_i + out_i$ fast memory locations
- Limited fast memory $\rightarrow$ some computed values may need to be temporarily stored in slow memory and reloaded later $\rightarrow$ cache misses

An example

For simplicity in the presentation: $w_i = 0$ and $out_i = 1$

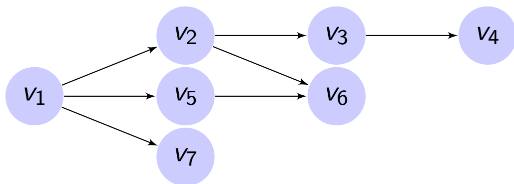


Sample execution order

vertex	v_1	v_2	v_3	v_4
data size	1			

An example

For simplicity in the presentation: $w_i = 0$ and $out_i = 1$

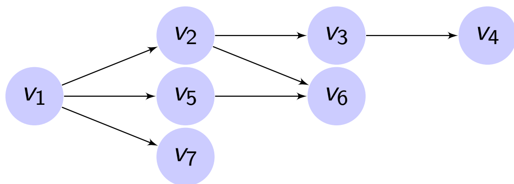


Sample execution order

vertex	v_1	v_2	v_3	v_4
data size	1	2		

An example

For simplicity in the presentation: $w_i = 0$ and $out_i = 1$

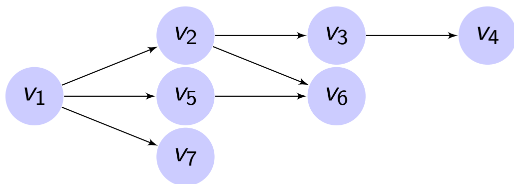


Sample execution order

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 $out_i = 1$

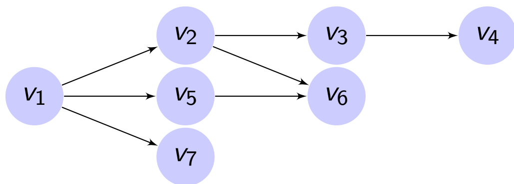


Sample execution order

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 $out_i = 1$



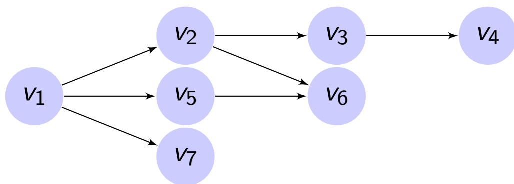
Sample execution order

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

For simplicity in the presentation: $w_i = 0$ and $out_i = 1$



Sample execution order

vertex	v_1	v_2	v_3	v_4
data size	1	2	3	4

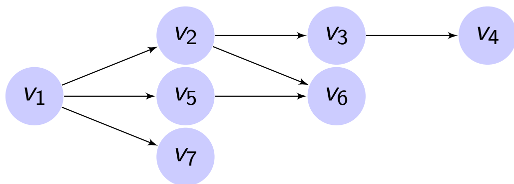
If $C = 3$, one will need to evict a data from the cache, hence resulting in a **cache miss**

Livesize and traversals

- **Livesize** (*live set size*): minimum cache size so that there are no cache misses
- **Traversal** $v_1 \to v_2 \to v_3 \to v_4 \to v_5 \to v_6 \to v_7$, livesize $= 4$

An example

For simplicity in the presentation: $w_i = 0$ and $out_i = 1$



Sample execution order

vertex	v_1	v_2	v_3	v_4
data size	1	2	3	4

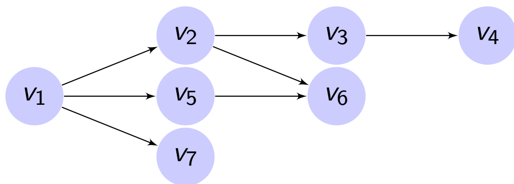
If $C = 3$, one will need to evict a data from the cache, hence resulting in a **cache miss**

Livesize and traversals

- **Livesize** (*live set size*): minimum cache size so that there are no cache misses
- **Traversal** $v_1 \to v_2 \to v_3 \to v_4 \to v_5 \to v_6 \to v_7$, livesize $= 4$
- For another traversal, $v_1 \to v_7 \to v_2 \to v_5 \to v_6 \to v_3 \to v_4$, livesize $= 3$

An example

For simplicity in the presentation: $w_i = 0$ and $out_i = 1$



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**

Livesize and traversals

- **Livesize** (*live set size*): minimum cache size so that there are no cache misses
- **Traversal** $v_1 \to v_2 \to v_3 \to v_4 \to v_5 \to v_6 \to v_7$, livesize $= 4$
- For another traversal, $v_1 \to v_7 \to v_2 \to v_5 \to v_6 \to v_3 \to v_4$, livesize $= 3$
- Task v_6 requires 3 cache locations $\to 3 =$ minimum cache size to execute this DAG

Cache eviction and optimization problem

Cache eviction

- During execution, if **livesize** $> C$, data transferred from cache back into slow memory
- The data that will be evicted may affect the number of cache misses
- Given a traversal, the optimal strategy (OPT) 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 **traversal of G** (topological order) 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]

Outline

1. Model
2. Scheduling strategies and experiments
3. Conclusion

DAG-partitioning assisted locality-aware scheduling

A novel approach

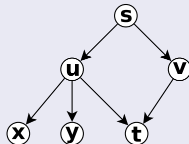
- Solution to MinCacheMiss = *traversal* of the graph
- Instead of looking for a global traversal of the whole graph, we propose to partition the DAG in an acyclic way: V divided in k disjoint subsets, or **parts**
- Key: have all parts executable **without cache misses**, hence the only cache misses can be incurred by data on the cut between parts
- Hence: minimize **edge cut** of the partition (**cut edge**: endpoints in different parts)

Livesize

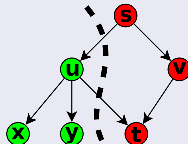
- Livesize for the traversal of a part: memory required to execute whole part, assuming inputs and outputs of the part are evicted if no longer required inside the part
- Partition such that, for each part, the livesize fits in cache

Acycling DAG partitioner

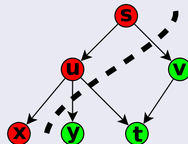
- Minimize number of **cache misses**: rely on acyclic DAG partitioner
- Input: maximum livesize of a part L_m



(a) A toy graph



(b) A partitioning ignoring the directions; it is cyclic



(c) An acyclic partitioning

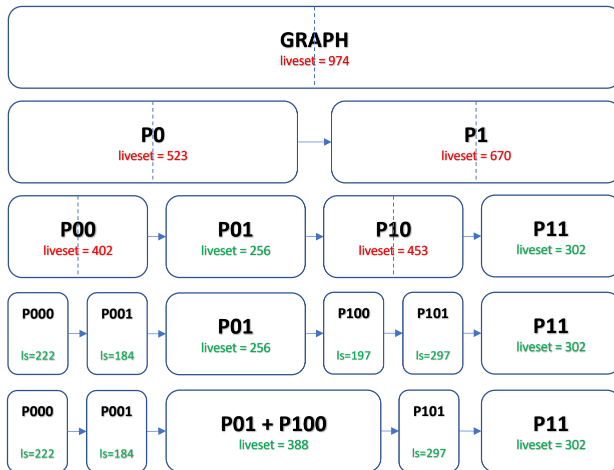
Multilevel acyclic DAG partitioning

- Recursive bisection until livesize of part $\leq L_m$
- Multilevel: coarsening, initial partitioning, refinement – all acyclic

[SISC'19]: Herrmann, Özkaya, Uçar, Kaya, Çatalyürek, "Multilevel Algorithms for Acyclic Partitioning of Directed Acyclic Graphs", SIAM Journal on Scientific Computing, 41(4):A2117-A2145, 2019.

Recursive bisection with target liveset size

Target liveset size $L_m = 400$



Traversals

- Return **total order on tasks**
- Must respect precedence constraints

Three classical approaches

- *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 $v_1 \to v_2 \to \cdots \to v_n$, 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 ready tasks
- *BFS traversal ordering (bfs)* follows a breadth-first traversal strategy among ready tasks

Traversals

- Return **total order on tasks**
- Must respect precedence constraints

Three classical approaches

- *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 ready tasks
- *BFS traversal ordering (bfs)* follows a breadth-first traversal strategy among ready tasks

- May be applied on whole DAG or on a part
- Can be extended to schedule parts (each part is a macro-task)
- We use same algorithm for parts and tasks within parts

  $\rightarrow$ Three novel strategies DAGP-NAT, DAGP-DFS, and DAGP-BFS

Graph instances

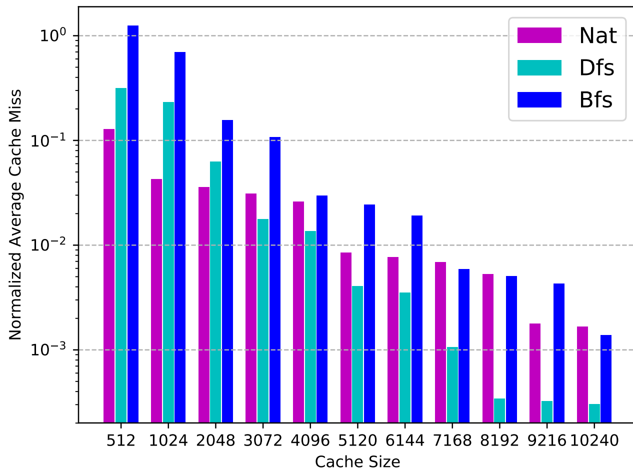
Instances from the SuiteSparse Matrix Collection (formerly know as UFL):

Graph	$\lvert V\rvert$	$\lvert E\rvert$	$max_{in.deg}$	$max_{out.deg}$	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.

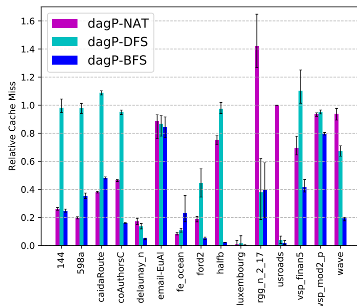
Performance of the three baseline traversal algorithms

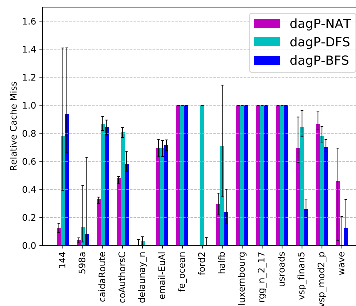
- Geometric mean of cache misses, normalized by number of nodes
- Smaller cache sizes: *nat* is best
- Cache size ≥ 3072: *dfs* is best



Relative cache misses of new algorithms

- Relative cache misses (geomean of average of 50 runs) for each graph separately
- DAG-partitioning assisted algorithm vs baseline with same traversal
- Left cache size 512; right cache size 10240; $L_m = C$

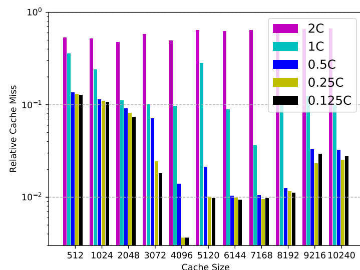


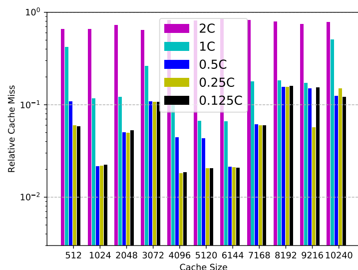


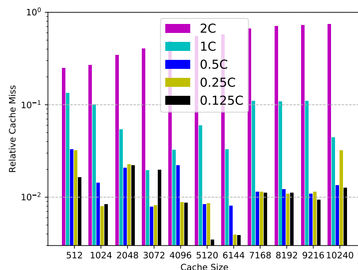
- DAGP-* performs almost always better than $*$, and good stability of algorithms
- With larger caches, may not need to partition

Effect of L_m and C on cache miss improvement

- Relative cache misses of DAGP-* with the given partition livesize
- Traversals *nat* (left), *dfs* (middle), and *bfs* (right)



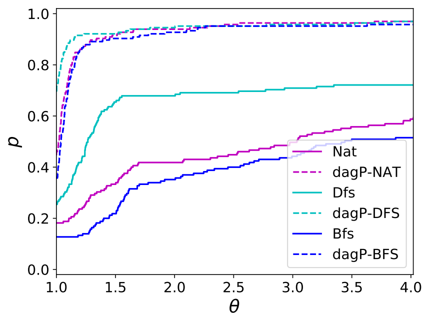


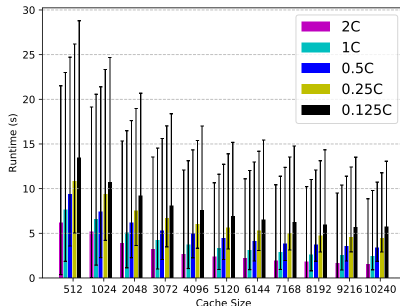


- $L_m \leq C$ is better: part fits in cache
- Further partitioning may help, but increases complexity of partitioning phase
- DAGP-DFS improves less than others... Indeed, baseline is better, less room for improvement!

Overall comparison of heuristics

- Left: Performance profile comparing baselines and heuristics with $L_m = 0.5 \times C$
  - Ratio of instances in which algo obtains cache miss count no larger than θ times the best CMC found by any algo for that instance
  - DAGP-DFS best 75% of the time; DAGP-* all very good
- Right: Average runtime of all graphs for DAGP-DFS partitioning





Outline

1 Model

2 Scheduling strategies and experiments

3 Conclusion

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