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

Yusuf M. Özkaya, Anne Benoit, Ümit V. Çatalyürek

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

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

[1] CSE, Georgia Institute of Technology, Atlanta, GA, USA
[2] LIP, ENS Lyon, France
{myozka, umit}@gatech.edu, anne.benoit@ens-lyon.fr

Abstract We investigate efficient execution of computations, modeled as Directed Acyclic Graphs (DAGs), on a single processor with a two-level memory hierarchy, where there is a limited fast memory and a larger slower memory. Our goal is to minimize execution time by minimizing redundant data movement between fast and slow memory. We utilize a DAG partitioner that finds localized, acyclic parts of the whole computation that can fit into fast memory, and minimizes the edge cut among the parts. We propose a new scheduler that executes each part one-by-one, obeying the dependency among parts, aiming at reducing redundant data movement needed by cut-edges. Extensive experimental evaluation shows that the proposed DAG-based scheduler significantly reduces redundant data movement.

1 Introduction

In today's computers, the cost of data movement through the memory hierarchy is dominant relative to the cost of arithmetic operations, and it is expected that the gap will continue to increase [6,12]. Hence, a significant portion of research efforts focuses on optimizing the data locality [1,5].

Sparse computations are used by many scientific applications but they are notorious for their poor performance due to their irregular memory accesses and hence poor data locality in the cache. One of widely used such kernels is Sparse Matrix-Vector Multiplication (SpMV), which is the main computation kernel for various applications (e.g., spectral clustering [9], dimensionality reduction [13], PageRank algorithm [10], etc.). Similarly, Sparse Matrix-Matrix Multiplication (SpMM), which is one of the main operations in many Linear Algebraic problems, suffers from similar cache under-utilization caused by unoptimized locality decisions. In this work, we aim to optimize repeated execution of such kernels, whose dependency structure can be expressed as a DAG.

We investigate efficient execution of sparse and irregular computations, modeled as Directed Acyclic Graphs (DAGs) on a single processor, with a two-level memory hierarchy. There are limited number of fast memory locations, and an unlimited number of slow memory locations. In order to compute a task, the processor must load all of the input data that the task needs, and it will also

need some scratch memory for the task and its output. Because of the limited fast memory, some computed values may need to be temporarily stored in slow memory and reloaded later.

Utilizing graph and hypergraph partitioning for effective use of cache has been studied by others before (e.g., [1,14]). However, to the best of our knowledge, our work is the first one that utilizes a multilevel acyclic DAG partitioning. Acyclic partitioning allows us to develop a new scheduler, which will load each part only once, and execute parts non-preemptively. Since the parts are computed to fit into fast memory, and they are acyclic, once all "incoming" edges of the part are loaded, all the tasks in that part can be executed non-preemptively, and without any need to bring additional data. Earlier studies that use undirected partitioning, cannot guarantee such execution model.

The rest of the paper is organized as follows. In Section 2, we present the computational model. Section 3 describes the proposed DAG-partitioning based scheduling algorithms. Experimental evaluation of the proposed methods is displayed in Section 4. We end with a brief conclusion and future work in Section 5.

2 Model

We consider a directed acyclic graph (DAG) $G = (V, E)$, where the vertices in set $V = \{v_1, \ldots, v_n\}$ represent computational tasks, and the dependency among them, hence the communication of data between tasks, is captured by graph edges in set E. Given $v_i \in V$, $pred_i = \{v_j \mid (v_j, v_i) \in E\}$ is the set of predecessors of task v_i in the graph, and $succ_i = \{v_j \mid (v_i, v_j) \in E\}$ is the set of successors of task v_i.

Vertices and edges have weights representing the size of the (scratch) memory required for task $v_i \in V$, w_i, and size of the data that is communicated to its successors. Here, we assume v_i produces a data of size out_i that will be communicated to all of its successors, hence weight of each edge $(v_i, v_j) \in E$ will be out_i. An *entry* task $v_i \in V$ is a task with no predecessors (i.e., $pred_i = \emptyset$), and such a task is a virtual task generating the initial data, hence $w_i = 0$. All other tasks (with $pred_i \neq \emptyset$) cannot start until all predecessors have completed, and the data from each predecessor has been generated. For simplicity in the presentation, we will use $w_i = 0$ and $out_i = 1$. Hence, total input size of task v_i is $in_i = |pred_i|$, and $in_i = 0$ if task v_i is an entry task.

We will assume the size of fast memory is C, and slow memory is large enough to hold whole data and computation. 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.

Consider the example in Figure 1. Task v_1 is the entry task and therefore just requires one cache location to generate its output data ($out_1 = 1$). The data corresponding to out_1 remains in cache while there is no need to evict it, while in_1 and w_1 will no longer be needed and can be evicted (but for an entry task, these are equal to 0). Consider that v_2 is executed next: it will need two cache

locations ($in_2 = out_2 = 1$). Then v_3 is executed. We want to keep out_1 in cache, if possible, since it will be reused later by v_5 and v_7. Hence, 3 memory locations are required. Similarly, when executing v_4, 4 memory locations are required (out_2 should also remain in cache). However, if the cache is too small, say $C = 3$, one will need to evict a data from the cache, hence resulting in a *cache miss*.

Given a traversal of the DAG, the *livesize* (live set size) is defined as the minimum cache size required for the execution so that there are no cache misses. In the example, with the traversal $v_1 \to v_2 \to v_3 \to v_4 \to v_5 \to v_6 \to v_7$, it would be 4. For another traversal however, the livesize may be smaller, and hence the order of traversal may greatly influence the number of cache misses. Consider the execution $v_1 \to v_7 \to v_2 \to v_5 \to v_6 \to v_3 \to v_4$: the livesize never exceeds 3 in this case, and this is the minimum cache size to execute this DAG, since task v_6 requires 3 cache locations to be executed.

Parts, cuts, traversals, and livesize. We generalize and formalize the definition of the livesize introduced above. 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 $v_j \in V_j$. The acyclic condition means that given any two parts V_i and V_j, we cannot have $V_i \rightsquigarrow V_j$ and $V_j \rightsquigarrow V_i$. In the example of Figure 1, an example of acyclic partition is $V_1 = \{v_1, v_2, v_3, v_4\}$ and $V_2 = \{v_5, v_6, v_7\}$.

Given a partition P of the DAG, an edge is called a *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 is defined as the sum of the costs of the cut edges, and can be formalized as follows: $\texttt{EdgeCut}(P) = \sum_{(v_i,v_j)\in E_{cut}(P)} out_i$, hence it is equal to $|E_{cut}(P)|$ with unit weights.

Let $V_i \subseteq V$ be a *part* of the DAG ($1 \leq i \leq k$). A *traversal* of the part V_i, denoted by $\tau(V_i)$, 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. In the example, there is only a single possible traversal for V_1: $\tau(V_1) = [v_1, v_2, v_3, v_4]$, while there are three possibilities for V_2: $[v_5, v_6, v_7]$, $[v_5, v_7, v_6]$, and $[v_7, v_5, v_6]$.

Given a part V_i and a traversal of this part $\tau(V_i)$, we can now define the livesize of the traversal as 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. Algorithm 1 details how the livesets are computed.

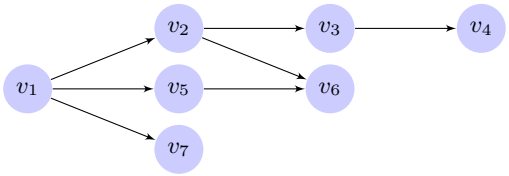

Figure 1: DAG example.

Cache eviction algorithm. If the livesize is greater than the cache size C, during execution, 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. This strategy is called optimal replacement algorithm (OPT, clairvoyant replacement algorithm, Belady's optimal page replacement algorithm) [2].

Hence, if a data needs to be evicted, the OPT algorithm looks at the upcoming schedule and orders the data in the cache by their next use times in ascending order. If a data is not used by any other task, it is assigned infinite value. The last data in this ordered list (the data not used or with farthest upcoming use) is then evicted from the cache. After scheduling a task, the next upcoming use of data generated by its immediate predecessors are updated.

Optimization problems. The goal is to minimize the number of loads and stores among all possible valid schedules. Formally, the MinCacheMiss problem is the following: 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.

However, finding the optimal traversal to minimize the livesize is already an NP-complete problem [11], even though it is polynomial on trees [8]. Therefore, MinCacheMiss is NP-complete (consider a problem instance where the whole DAG could be executed without any cache miss if the livesize was minimum).

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.

Algorithm 1: Computing the livesize of a part

Data: Directed graph $G = (V, E)$, part $V_\ell \subseteq V$, traversal $\tau(V_\ell)$
Result: Livesize $L(\tau(V_\ell))$

$L(\tau(V_\ell)) \leftarrow 0$; current $\leftarrow 0$; $V'_\ell \leftarrow V_\ell$;
foreach v_i **in** $\tau(V_\ell)$ **order do**
  $V'_\ell \leftarrow V'_\ell \setminus \{v_i\}$;
  current $\leftarrow$ current $+ w_i + out_i$; /* Add current task to liveset */
  $L(\tau(V_\ell)) \leftarrow \max\{L(\tau(V_\ell)),$ current$\}$; /* Update L if needed */
  if $succ_i = \emptyset$ *or* $\forall v_j \in succ_i$, $v_j \notin V_\ell$ **then**
    current $\leftarrow$ current $- out_i$; /* No need to keep output in cache */
  for $v_j \in pred_i$ **do**
    if $V'_\ell \cap succ_j = \emptyset$ /* No successors of v_j after v_i in $\tau(V_\ell)$ */
    then
      current $\leftarrow$ current $- out_j$; /* No need to keep v_j in cache */
  current $\leftarrow$ current $- w_i$; /* Task is done, no need scratch space */

3 DAG-partitioning Assisted Locality-Aware Scheduling

We propose novel DAG-partitioning-based cache optimization heuristics that can be applied on top of classical ordering approaches to improve cache locality, using a modified version of a recent directed acyclic graph partitioner [7].

Three classical approaches are considered for the traversal of the whole DAG, and they return a total order on the tasks. The traversal must respect precedence constraints, and hence a task $v_i \in V$ is said to be *ready* when all its predecessors have already been executed: all predecessors must appear in the traversal before v_i.

- *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.

When applied to the whole DAG, these three traversal algorithms are baseline algorithms, and they will serve as a basis for comparison in terms of cache miss. Also, these traversals can be applied to a part of the DAG in the DAG-partitioned approach, and also they can be extended to parts themselves: part V_i is ready if, for all $v_i \in V_i$, if $(v_j, v_i) \in E$ and $v_j \notin V_i$, then v_j has already been scheduled. Furthermore, the part id is the minimum of ids of the nodes in the part. Then, considering parts as macro tasks, the same traversals can be used.

The modified acyclic DAG partitioner takes a DAG with vertex and edge weights (in this work, unit weights assumed), and a maximum livesize L_m for each part as input. Its output is a partition of the vertices of G into K nonempty pairwise disjoint and collectively exhaustive parts satisfying three conditions: (i) the livesize of the parts are smaller than L_m; (ii) the edge cut is minimized; (iii) the partition is acyclic; in other words, the inter-part edges between the vertices from different parts should preserve an acyclic dependency structure among the parts. The partitioner does not take the number of parts as input, but takes a maximum livesize. That is, the partitioner continues recursive bissection until the livesize of the part at hand is less than L_m, meaning that it can be loaded and completely run in the cache of size L_m without any extra load operations.

Hence, the DAGP approach (with DAG partitioner) uses multilevel recursive bisection approach, matching (coarsening) methods, and refinement methods that are specialized to create acyclic partitions. The refinement methods are modified versions of Fiduccia-Mattheyses' move-based refinement algorithm. The partitioning algorithm starts by computing the livesize of the graph. If computed livesize is less than L_m, then it stops and leaves everything in the same part. Otherwise, the graph is partitioned into two subgraphs with the edge cut minimization goal. The same procedure is repeated for the two subgraphs until each part has a livesize smaller than L_m. The partitioner also duplicates the

boundary nodes of predecessor parts to successor parts, so as to account for the input data (caching) requirement of the tasks from predecessor parts.

Given K parts $V_1, \ldots, V_K$ forming a partition of the DAG, we consider three variants of DAGP (DAGP-NAT, DAGP-DFS, and DAGP-BFS), building upon the traversals described above. Due to page limits, we always use the same algorithm for parts and for tasks within parts, but any combination would be possible (e.g., order parts with *nat*, and then tasks within parts with *dfs*).

4 Experimental Evaluation

Experiments were conducted on a computer equipped with dual 2.1 GHz Xeon E5-2683 processors and 512 GB memory. We have performed an extensive evaluation of the proposed heuristics on DAG instances obtained from the matrices available in the SuiteSparse Matrix Collection (formerly known as the University of Florida Sparse Matrix Collection) [3]. From this collection, we picked 15 matrices satisfying the following properties: listed as binary, square, and has at least 100000 rows and at most 2^{26} nonzeros. All edges have unit costs, and all vertices have unit weights. For each such matrix, we took the strict upper triangular part as the associated DAG instance whenever this part has more nonzeros than the lower triangular part; otherwise, we took the lower triangular part. The graphs and their characteristics are listed in Table 1. We execute each graph with the natural, DFS, and BFS traversal, and report the corresponding livesize (computed with Algorithm 1).

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.

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

Table 1: Graph instances from [3] and their livesizes.

We start by comparing the average performance of the three baseline traversal algorithms on varying cache sizes. Figure 2 shows the geometric mean of cache miss counts, normalized by the number of nodes, with cache size ranging from 512 to 10240 words. In smaller cache sizes, the natural ordering of the nodes provides the best results on average. As the cache size increases, DFS traversal surpasses the others and becomes the best option starting at 3072.

Effect of Cache Size on Reported Relative Cache Miss. Figure 3 shows the improvement of our algorithms over their respective baselines, averaged over 50 runs. The left figure shows the relative cache miss on a cache $C = 512$ and the right one on $C = 10240$, with $L_m = C$ (i.e., the partitioner stops when the livesize of each part fits in cache). A relative cache miss of 1 means that we get the same number of cache misses as without partitioning; the proposed solution is better than the baseline for a value lower than 1 (0 means that we reduced the cache misses to 0), and it is worsened in a few cases (i.e., values greater than 1).

One important takeaway from this figure is that as the cache increases, the input graph's livesize may become less than the cache size (e.g., `fe_ocean`, `luxembourg-osm`, `rgg-n-2-17-s0`, and `usroads` graphs have smaller livesize than the cache size), meaning that there is no cache miss even without partitioning. Thus, the partitioning phase does not need to divide the graph at all and just returns the initial graph.

Finally, the variance from one run to another is relatively low, demonstrating the stability of the algorithm, hence we perform the average over only 10 runs in the remaining experiments.

Effect of L_m and C on Cache Miss Improvement and Edge Cut. Figure 4 shows the normalized cache misses when the execution strategy of the graph is *nat*, *dfs*, and *bfs* traversal respectively. We compare the number of cache misses when traversing the input graph with the three partitioning based-heuristics. We use five different values for the L_m parameter of the algorithms: $L_m = \{2C, C, 0.5C, 0.25C, 0.125C\}$ (i.e., from twice the size of cache to down to one eighth of the cache size), and compare the results. Each bar in the chart shows the respective relative cache miss of partitioning-assisted ordering compared to the baseline. Throughout all the bars in all the charts, we can see

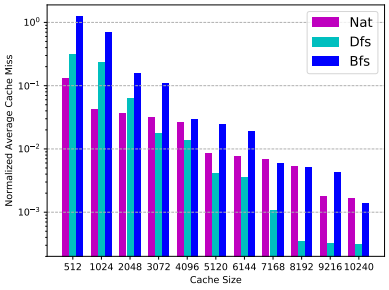

Figure 2: Geometric mean of cache misses using *nat*, *dfs*, and *bfs* traversals.

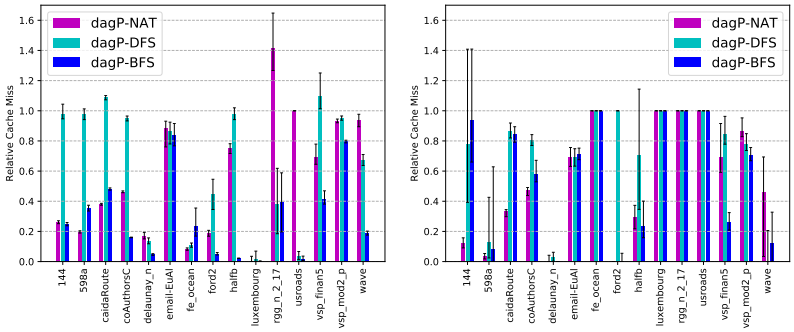

Figure 3: Relative cache misses (geomean of average of 50 runs) for each graph separately (left cache size 512; right cache size 10240).

$L_m \leq C$ gives better results. This is expected since the partitioning phase is trying to decrease the livesize, which in turn, decreases the upper bound of cache misses, since the whole part can be executed without cache misses if $L_m \leq C$. Otherwise, for each part, we might still have cache misses.

There are two reasons why the partitioning result is not an exact cache miss count but an upper bound. First, the edge cut counts the edges more than once when a node u from part V_i is predecessor for multiple nodes in V_j. In cache, however, after node u is loaded for one of its successors in V_j, ideally, it is not removed before its other successors are also scheduled ($L_m \leq C$ guarantees all the nodes in part V_j can be computed without needing to remove node u from cache). The second reason is that right after a part is completely scheduled, the partitioner does not take into account the fact that the last tasks (nodes) scheduled from the previous part are still in the cache.

On average, continuing the partitioning after $L_m = C$ usually improves the performance. However, the improvement as L_m decreases and the increase in the complexity/runtime of the partitioning is a tradeoff decision.

Comparing the plots of the natural ordering, dfs traversal, and bfs traversal, one can argue that the improvement over dfs ordering is not as high as natural, or bfs ordering. This can be explained by relating to the Figure 2, which shows the average cache miss counts for the baseline algorithms. It is obvious that,

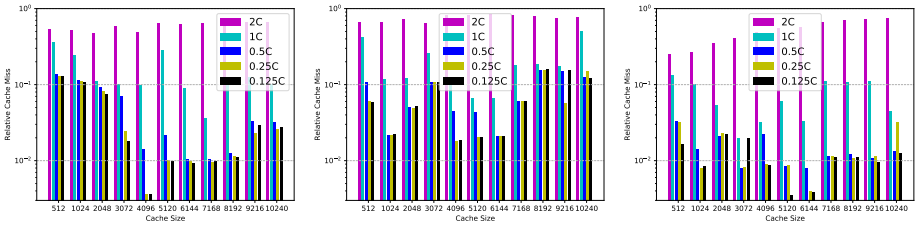

Figure 4: Relative cache misses of DAGP-* with the given partition livesize for *nat* (left), *dfs* (middle), and *bfs* (right) traversals.

on average, dfs traversal gives much better results compared to the other two. Thus, DAGP-DFS variation has less room to improve over this baseline.

Another general trend is that as the cache size increases, the performance improvements slightly decrease. This can also be explained by the phenomenon shown in Figure 3. That is, as the cache size increases, some of the graphs do not have any cache misses, therefore, the partitioning, cannot improve over zero cache misses (relative cache miss is equal to 1), decreasing the overall performance improvement of the heuristics.

When we look at the values in the figure, we see that average relative cache miss for $L_m = 0.5 \times C$, is in the range `10x` to `100x` better than the baseline. In addition, relative cache miss of DAGP-BFS goes lower than $3 \cdot 10^{-3} = 0.003$ for $C = 6144$ and $L_m = 0.25C$, which is nearly `400x` better than the baseline.

Overall Comparison of Heuristics. Figure 5 (left) shows the performance profile for the three baseline algorithms and heuristics applied versions. A performance profile shows ratio of the instances in which an algorithm obtains a cache miss count on an instance that is no larger than θ times the best cache miss count found by any algorithm for that instance [4].

Here, we compare the heuristics using $L_m = 0.5 \times C$, since it constantly provides better results than $L_m = C$ and increase in the partitioning overhead is minimal. We can see that DAGP-DFS gives the best ordering approximately 75% of the time; and 90% of the time, it gives cache misses no worse than 1.5 times the best heuristic. We can also see that all three proposed heuristics perform better over their respective baselines. Also, all three heuristics perform better than any of the baselines.

Finally, the runtime averages, including all graphs in the dataset for the given parameter configurations, are depicted in Figure 5 (right). The black error bars show the minimum and maximum run times for that bar. It is clear that for smaller L_m values, the partitioner needs to do more work, but partitioner is fast and does not take more than 10 seconds in most common cases (i.e., $L_m = 0.5 \times C$).

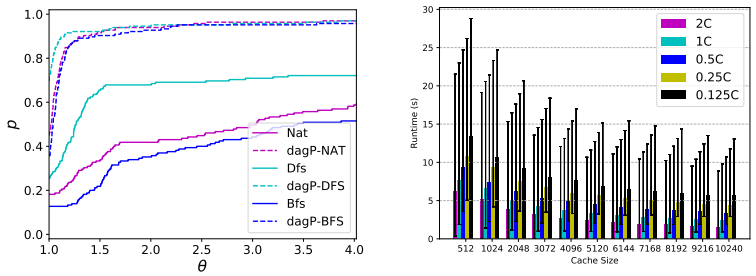

Figure 5: (Left) Performance profile comparing baselines and heuristics with $L_m = 0.5 \times C$. (Right) Average runtime of all graphs for DAGP-DFS partitioning.

5 Conclusion and Future Work

As the cost of data movement through the memory hierarchy dominates the cost of computational operations in today's computer systems, data locality is a significant research focus. Although there have been many approaches to improve data locality of applications, to the best of our knowledge, there is no work that employs a DAG-partitioning assisted approach. Building upon such a partitioner, we design locality-aware scheduling strategies, and evaluate the proposed algorithms extensively on a graph dataset from various areas and applications, demonstrating that we can significantly reduce the number of cache misses. As the next step, it would be interesting to study the effect of a customized DAG-partitioner specifically for cache optimization purposes, and also to design traversal algorithms to optimize cache misses. It would also be interesting to use a better fitting hypergraph representation for the model.

References

1. K. Akbudak, E. Kayaaslan, and C. Aykanat. Hypergraph partitioning based models and methods for exploiting cache locality in sparse matrix-vector multiplication. *SIAM Journal on Scientific Computing*, 35(3):C237–C262, 2013.
2. L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.*, 5(2):78–101, June 1966.
3. T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, 2011.
4. E. D. Dolan and J. J. Moré. Benchmarking optimization software with performance profiles. *Mathematical programming*, 91(2):201–213, 2002.
5. N. Fauzia, V. Elango, M. Ravishankar, J. Ramanujam, F. Rastello, A. Rountev, L.-N. Pouchet, and P. Sadayappan. Beyond reuse distance analysis: Dynamic analysis for characterization of data locality potential. *ACM Trans. Archit. Code Optim.*, 10(4):53:1–53:29, Dec. 2013.
6. S. H. Fuller and L. I. Millett. *The Future of Computing Performance: Game Over or Next Level?* National Academy Press, 2011.
7. J. Herrmann, M. Y. Özkaya, B. Uçar, K. Kaya, and Ü. V. Çatalyürek. Multilevel algorithms for acyclic partitioning of directed acyclic graphs. *SIAM Journal on Scientific Computing (SISC)*, 2019. to appear.
8. M. Jacquelin, L. Marchal, Y. Robert, and B. Uçar. On optimal tree traversals for sparse matrix factorization. In *IPDPS 2011*, pages 556–567, 2011.
9. A. Y. Ng, M. I. Jordan, and Y. Weiss. On spectral clustering: Analysis and an algorithm. In *Advances in neural information processing syst.*, pages 849–856, 2002.
10. L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.
11. R. Sethi. Complete register allocation problems. In *Proceedings of the 5th Annual ACM Symp. on Theory of Computing (STOC'73)*, pages 182–195, 1973.
12. J. Shalf, S. Dosanjh, and J. Morrison. Exascale computing technology challenges. In *Proceedings of the 9th Int. Conf. on High Performance Computing for Computational Science*, VECPAR'10, pages 1–25, 2011.
13. S. Wold, K. Esbensen, and P. Geladi. Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52, 1987.
14. A. N. Yzelman and R. H. Bisseling. Two-dimensional cache-oblivious sparse matrix-vector multiplication. *Parallel Computing*, 37(12):806–819, 2011.