HDagg: Hybrid Aggregation of Loop-carried Dependence Iterations in Sparse Matrix Computations

Behrooz Zarebavani*, Kazem Cheshmi*, Bangtian Liu*, Michelle Mills Strout†, and Maryam Mehri Dehnavi*
*Department of Computer Science, University of Toronto, †Department of Computer Science, University of Arizona
Email:{behrooz, kazem, bangtian}@cs.toronto.edu, mstrout@cs.arizona.edu, mmehride@cs.toronto.edu

***Abstract*—This paper proposes a novel aggregation algorithm, called Hybrid DAG Aggregation (HDagg), that groups iterations of sparse matrix computations with loop carried dependence to improve their parallel execution on multicore processors. Prior approaches to optimize sparse matrix computations fail to provide an efficient balance between locality, load balance, and synchronization and are primarily optimized for codes with a tree-structure data dependence. HDagg is optimized for sparse matrix computations that their data dependence graphs (DAGs) do not have a tree structure, such as incomplete matrix factorization algorithms. It uses a hybrid approach to aggregate vertices and wavefronts in the DAG of a sparse computation to create well-balanced parallel workloads with good locality.**

Across three sparse kernels, triangular solver, incomplete Cholesky, and incomplete LU, HDagg outperforms existing sparse libraries such as MKL with an average speedup of $3.56\times$ and is faster than state-of-the-art inspector-executor approaches that optimize sparse computations, i.e. DAGP, LBC, wavefront parallelism techniques, and SpMP by an average speedup of $3.87\times$, $3.41\times$, $1.95\times$, and $1.43\times$ respectively.

***Index Terms*—Parallelism, Sparse Matrix Computations, Loop-carried Dependence**

I. Introduction

Sparse matrix computations are an important class of algorithms frequently used in scientific simulations. The performance of these simulations relies heavily on the parallel implementations of sparse matrix computations used to solve systems of linear equations. Independent iterations should be evenly distributed amongst cores to achieve high parallel efficiency in sparse matrix codes while minimizing synchronization overhead. Data locality should also be improved by exploiting data reuse between iterations of the sparse computation. Finding an efficient balance between locality, load balance, and synchronization is often challenging in sparse codes due to the existing irregularities in the data structures and the indirect memory access patterns and dependencies in the computations. This problem is exacerbated when the data dependencies in the sparse computation do not have a tree structure.

Data dependencies between iterations of a loop with partial parallelism are described with a data-flow directed acyclic graph (DAG). Then a scheduling algorithm is applied to construct an efficient order to execute iterations of the sparse kernel in parallel. DAG partitioning techniques such as DAGP [1] partition the vertices in the DAG while minimizing edge cuts between partitions to improve locality. Independent partitions are scheduled to execute in parallel. However, the partitioned graph of DAGP has restricted average parallelism, resulting in insufficient parallel workloads for all cores, which leads to a load-imbalanced schedule.

Wavefront parallelism techniques [2], [3] traverse the DAG in topological order to create a list of wavefronts. Each wavefront represent iterations which are scheduled to execute in parallel. A global synchronization, i.e. barrier, is used after each wavefront to satisfy data-dependence relations. However, this can lead to high overheads since the number of wavefronts increases with the DAG's critical path. Wavefront parallelism techniques can also lead to load imbalance because workloads in sparse kernels are often non-uniform. If dependent iterations in the sparse kernel are executed on the same core, the data reuse between their computations can turn into a locality. However, wavefront parallelism techniques often do not take advantage of this property because of the global synchronization between wavefronts.

The SpMP method proposed in [4] improves load balance in wavefront parallelism approaches by grouping vertices inside a wavefront to create balanced workloads for all cores. In SpMP, the grouped vertices in each core are executed wavefront after wavefront. The execution of groups from different wavefronts is overlapped using point-to-point synchronization between the groups to further improve load balance. SpMP suffers from poor data locality as vertices are executed based on the wavefront order, similar to the wavefront parallelism techniques.

Wavefront-coarsening approaches [5], [6] are used to merge vertices across wavefronts to create well-balanced coarsened wavefronts, mitigating load imbalance and excessive synchronization present in wavefront parallelism techniques. Load-Balanced level Coarsening (LBC) [7], implemented inside the ParSy inspector-executor framework [8], is an example of wavefront coarsening. Its algorithm is optimized for the class of sparse computations that their data dependence graphs are trees such as the elimination tree [9] in Cholesky factorization. LBC creates an initial coarse wavefront to create enough well-balanced workloads for all cores. To find an initial cut, LBC relies on the fact that each vertex in a tree has one outgoing edge, and hence its algorithm is optimized for DAGs that are trees. Also, because of wavefront coarsening in LBC,

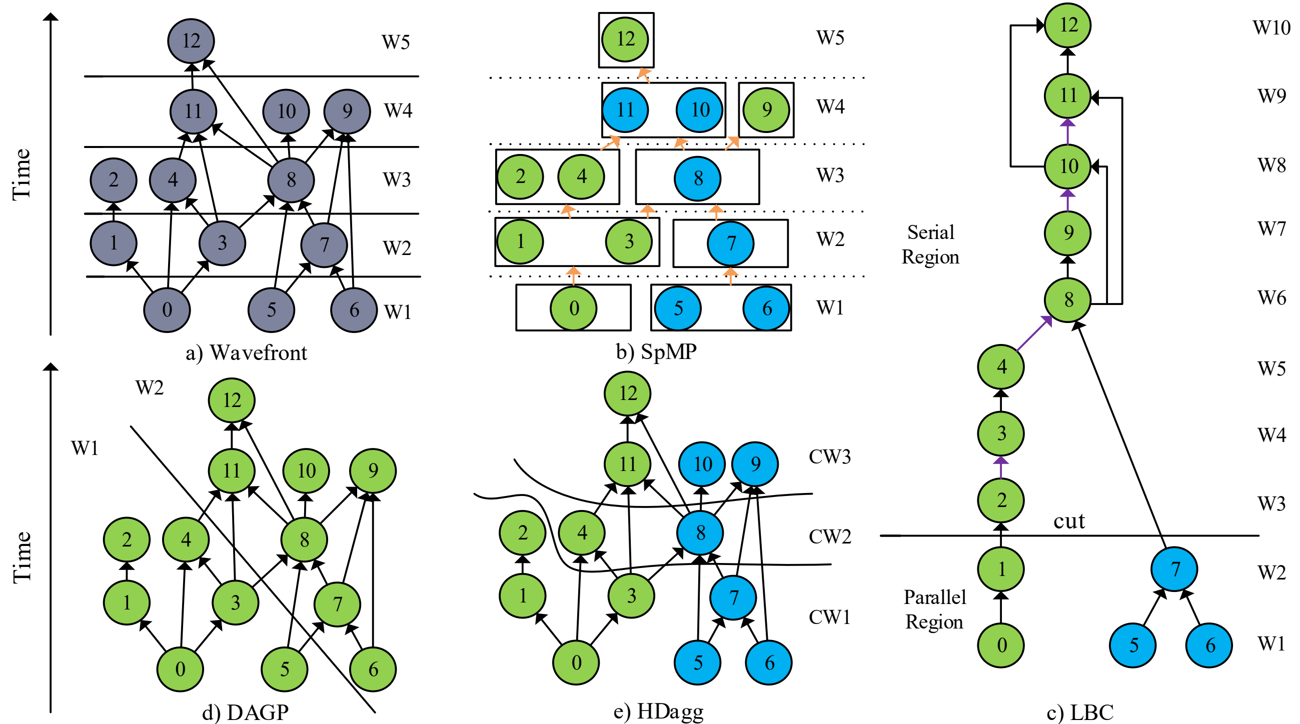

Fig. 1. Figures (a)–(e) show the schedules generated by the five algorithms when applied to the DAG shown in Figure 1(a). Blue and green vertices execute in order in cores 0 and 1. Figure 1(a) shows the schedule of wavefront parallelism techniques with five wavefronts running sequentially. The vertices inside a wavefront execute in parallel, in any order, and are separated from other wavefronts with a global barrier, i.e. the horizontal solid lines. The schedule in Figure 1(b) is created by SpMP where it groups vertices within a wavefront and maps groups to cores. Vertices across wavefronts can be overlapped due to point-to-point synchronization, i.e. orange arrows. Figure 1(c) is the LBC schedule where the DAG is converted to a tree by adding extra edges. Some extra edges are shown with purple. LBC coarsens wavefronts up to the level annotated with the word *cut*. Before cut, enough workloads exist for all cores, vertices after cut execute sequentially. The partitioning created by DAGP is shown in Figure 1(d). The objective of this partitioning is to minimize edge cuts. Both partitions execute on the same core due to the existing dependency between the partitions.

iterations with shared data might be mapped to different coarsened wavefronts and thus do not efficiently use data locality opportunities.

A large class of sparse matrix computations operate on DAGs with non-tree structure. For example, incomplete matrix factorization methods such as the sparse incomplete Cholesky zero (SpIC0) and the sparse incomplete LU zero (SpILU0) [10] preconditioners have non-tree DAGs. Other sparse kernels, such as the sparse triangular solver (SpTRSV), often operate on non-tree DAGs when used inside a preconditioned iterative solver. For sparse kernels with non-tree DAGs, LBC uses chordalization to convert the DAG to a tree. This process introduces additional edges to the DAG and hence degrades parallelism.

We propose an algorithm called Hybrid DAG Aggregation (HDagg) and implement it in an inspector-executor framework. The inspector statically partitions the DAG of the sparse matrix computation using the HDagg algorithm. HDagg groups iterations in sparse codes with non-tree DAGs to create an efficient final schedule for execution. To improve data locality in the final schedule, it first finds groups of densely connected vertices inside the DAG and merges the vertices inside each group. HDagg then uses a novel wavefront coarsening strategy, called load-balance persevering (LBP), that efficiently coarsens non-tree DAGs. A novel bin-packing approach is also used in LBP to find a balance between locality and synchronization.

II. Motivating Example

Figures 1(a)–(e) compare the schedule created by HDagg with that of wavefront parallelism techniques, SpMP, LBC, and DAGP. All methods are applied to the same DAG and on a processor with two cores. Each of these scheduling algorithms provides a different order of execution for the outermost loop of the sparse kernel. The order of execution affects locality, load balance, and the number of synchronizations. In the following, we first use an example sparse kernel to demonstrate how a DAG is created from the sparse code. We then compare the schedules created from the aforementioned scheduling methods.

The algorithms in Figure 1 operate on the DAG of a sparse kernel. The structure of the DAG depends on the type of sparse kernel and the sparsity pattern of the input matrix. Each vertex in the DAG represents an iteration of the outermost loop of a kernel. The data dependencies between the outermost-loop iterations are represented with directed edges between vertices. As an example, we show in Listing 1 the code for the

sparse triangular solve kernel in the Compressed Sparse Row (CSR) storage format, where n, Ap, Ai, and Ax represent the number of rows, row pointer, column index, and non-zeros respectively. The outermost loop (Line 1) iterates over each row i of the matrix A, and the inner loop (Line 3) iterates over column indices in row i. The outermost loop carries dependence because $x[k]$, written in iteration k, can be read in a proceeding iteration c. For example, if a non-zero exists in row 3 and column 0 in A, then iterations 0 and 3 will be dependent. Hence a directed edge from vertex 0 to vertex 3 would exist in the DAG of this computation.

```
1 for(int i = 0; i < n; i++){
2     x[i] = b[i];
3     for(int j = Ap[i]; j < Ap[i + 1] - 1; j++){
4         x[i] -= Ax[j] * x[Ai[j]];
5     }
6     x[i] /= Ax[Ap[i + 1] - 1];
7 }
```

Listing 1. The SpTRSV code for the CSR storage format.

The wavefront parallelism technique is shown in Figure 1(a) where five global barriers are used to ensure correctness. Each iteration performs a different number of operations depending on the number of non-zeros per row of the matrix. Thus, running iterations in parallel makes wavefront parallelism load imbalanced. The SpMP schedule shown in Figure 1(b) solves the load imbalance issue in wavefront parallelism techniques by using point to point synchronization, which allows for the execution of vertices across different wavefronts to be overlapped. For example, vertices 2 and 4 can be executed immediately after 1 and 3 since their dependence relations are satisfied, and thus unlike a global barrier, they do not need to wait for vertex 7. Wavefront parallelism techniques and SpMP typically do not take advantage of the data reuse opportunities between iterations. For instance, the available data reuse between 1 and 2 might not turn into locality because vertex 3 is executed immediately after vertex 1.

To improve locality in wavefront parallelism techniques, the LBC and DAGP methods group iterations by inspecting the DAG as shown in Figure 1(c) and Figure 1(d), respectively. DAGP partitions vertices to minimize edge cuts and to improve data reuse between iterations of a partition, e.g. enabling data reuse between iterations 0, 1, and 2 in the first partition, colored in green. However, because of the existing dependencies between the two partitions, they cannot execute in parallel resulting in load imbalance. LBC analyzes the DAG using a wavefront coarsening approach to find enough parallel workloads for all cores. To find a coarsened wavefront, LBC first chordalizes the DAG by adding more edges and then converts it to a tree as shown in Figure 1(c). It then inspects the wavefronts, starting from the last wavefront, and selects a wavefront with more than two vertices, e.g. wavefront 2 in Figure 1(c) and creates two parallel workloads for the two cores. However, because of the added edges, the final coarsened wavefront only has one component and has a large computation load, leading to load imbalance. Applying LBC directly to the DAG of Figure 1(a) results in the coarsened wavefronts 1–4 with only one connected component which also creates load imbalance.

As shown in Figure 1(e), the HDagg strategy creates three global barriers, which is lower than that of Wavefront parallelism techniques. Unlike LBC and DAGP, the created partitions from HDagg provide balanced parallelism within each coarsened wavefront. For example, in Figure 1(e), the three coarsened wavefronts have enough workloads for the two cores. Also, to ensure that wavefront coarsening does not reduce locality, HDagg groups vertices 1 and 2 prior to wavefront coarsening to ensure they will execute on the same thread. The schedule created by HDagg provides an efficient parallel implementation for SpTRSV, SpIC0, and SpILU0 and is faster than DAGP, LBC, wavefront parallelism techniques, and SpMP with an average speedup of $3.87\times$, $3.41\times$, $1.95\times$, and $1.43\times$ respectively.

```
1 #include "HDagg.h"
2 int main() {
3  Sparse A("path/to/mat.mtx");
4  Kernel ILU0;
5  /*********** Inspector ***********/
6  Graph G = ILU0.DAG(A);
7  Cost C = ILU0.cost(A);
8  Schedule S = HDagg(G, C, num_cores(), epsilon());

9  /*********** Executor ***********/
10 Factor f = ilu0_omp(A, S);
11 return 0;
12 }
```

Listing 2. The inspector-executor driver code of HDagg for SpILU0.

III. Framework Overview

We implement HDagg in an inspector-executor framework. The inspector creates an efficient order of execution for the sparse kernel using the HDagg algorithm. The kernel code in the executor is transformed using the created partitions from the inspector and is executed in parallel using OpenMP. Both the inspector and the executor are implemented in `C++`.

The inspector and executor are used as a library as shown in the SpILU0 driver code in Listing 2. The driver code first loads the input matrix in Line 3. It then creates the DAG for the sparse computation and computes the cost of each iteration using a kernel-specific embedded library in Lines 6 and 7 respectively. To compute the DAG of the three supported kernels, i.e. SpTRSV, SpIC0, and SpILU0, we use the input matrix [8], [11]. We do not create the DAG explicitly for efficiency and instead reuse the input matrix as the DAG. The computed DAG and the cost in Line 8 are used by the algorithm to build the schedule. `num_cores` in Line 8 returns the number of physical cores in the target processor. A predefined load balance threshold is returned by calling `epsilon` in Line 8. The parallel kernel code in Line 10 goes over the created schedule and runs the iterations of the sparse kernel in parallel.

IV. Hybrid DAG Aggregation Algorithm

The HDagg algorithm statically partitions the DAG of a sparse matrix computation using a hybrid approach during

Algorithm 1 The HDagg Algorithm.

```
Input: G, C, p, ϵ
Output: S
    /*Step1: Aggregating Densely Connected Vertices*/
 1: G′ = TransitiveReduction(G)
 2: T.append(G′.Sink())
 3: for (i = 0; i < T.size(); i = i + 1) do
 4:   H = T_i
 5:   for (j = 0; j < H.size(); j = j + 1) do
 6:     v = H_j
 7:     A = parents(G′, v)
 8:     if {v} ∪ A is tree then
 9:       H.append(A)
10:     else
11:       for (c ∈ A) do
12:         if isNotVisited(c) then
13:           T.append([c]); visit(c);
14:         end if
15:       end for
16:     end if
17:   end for
18:   T_i = H
19: end for
20: G′′ = Coarsened_DAG(G′, T)
    /*Step2: Load-balance Preserving Wavefront Coarsening*/
21: W, l = Wavefront(G′′)
22: cut = 0
23: S_curr = S_prev = BinPack(CC(W[0 : 1]), C, p)
24: for (i = 1; i ≤ l; i = i + 1) do
25:   S_curr = BinPack(CC(W[cut : i]), C, p)
26:   if PGP(S_curr) > ϵ then
27:     if cut == i − 1 then      ▷ Single Unbalanced Wave
28:       S.append(S_curr)
29:     else
30:       S.append(S_prev)
31:     end if
32:     cut = i
33:   end if
34:   S_prev = S_curr
35: end for
36: if PGP(S) < ϵ then
37:   S.DisableBinPack()
38: end if
```

inspection. Its objective is to create a partitioning of a general DAG (DAGs that do not necessarily have a tree structure) that when executed provides a good load balance and low synchronization cost while improving locality. HDagg first groups vertices that form densely connected regions inside the DAG to ensure they execute on the same thread to improve locality. As a result, a grouped DAG is created where each vertex corresponds to a group of vertices. Then, to reduce synchronization costs and to improve load balance and locality, HDagg coarsens wavefronts of the grouped DAG. The proposed wavefront coarsening uses a novel cost model called the potential gain proxy (PGP) that provides enough parallel and balanced workloads for all cores when applied to a general DAG.

A. Input and Outputs

HDagg takes the dependency graph G, the cost function C, the number of cores p, and the load balance threshold ϵ as inputs and builds the schedule S. G is defined as $G = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ is the set of vertices and $\mathcal{E}$ is the set of directed edges between the vertices. Integer numbers are used to represent vertices in the DAG. For example, $i \in \mathcal{V}$ means that the vertex with id i belongs to the set $\mathcal{V}$. Directed edges are shown with $i \rightarrow j$ where i is the parent of j, i.e. there exists an outgoing edge from i to j. Each vertex i is annotated with a cost shown with C_i. The number of non-zero elements touched is used as the cost function C [7]. The computed schedule from HDagg is shown with S. The created schedule is composed of a set of disjoint partitions called coarsened wavefronts. Each coarsened wavefront is composed of one or more disjoint partitions called width-partitions. The coarsened wavefronts execute sequentially and width-partitions of a coarsened wavefront run in parallel. As an example, schedule S for the DAG in Figure 2(a) is shown in Figure 2(d). This example schedule has three coarsened wavefronts, the first coarsened wavefront has two width-partitions $\{0, 1, 2, 3\}$ and $\{5, 6, 7\}$.

B. Aggregating Densely Connected Vertices

The objective of the first step of HDagg is to aggregate vertices that benefit from executing sequentially and on the same core because of being highly dependent with many edges connecting them. The computations associated with these vertices share a lot of data, i.e. they are densely connected vertices. Densely connected components in a DAG are often converted to subtrees if their transitive edges are removed. A transitive edge is an edge that, if removed, does not violate any dependence in the computation [4]. Leveraging this feature, the first step of HDagg removes transitive edges and detects subtrees from the reduced DAG to efficiently find densely connected vertices in the input.

HDagg removes transitive edges from the input DAG G and builds a reduced graph G' using a two-hop transitive edge reduction approximation method [4]. This method removes an edge $i \rightarrow f$, if there is a vertex j with incoming edge $i \rightarrow j$ and outgoing edge $j \rightarrow f$. After building the reduced graph G', HDagg uses a modified breadth-first search (BFS) traversal method to find subtrees that represent densely connected vertices. The algorithm finds a subtree by defining a sink vertex and then groups vertices with the sink vertex if they form a connected subtree. A sink vertex is defined as a vertex that does not have any outgoing edges to another vertex in its subtree. Each tree only has one sink vertex, HDagg uses this property to find subtrees in the reduced DAG.

Densely connected vertices are detected in Lines 1–20 in Algorithm 1. The transitive reduction of the input DAG G is computed in Line 1 using *TransitiveReduction* and is stored

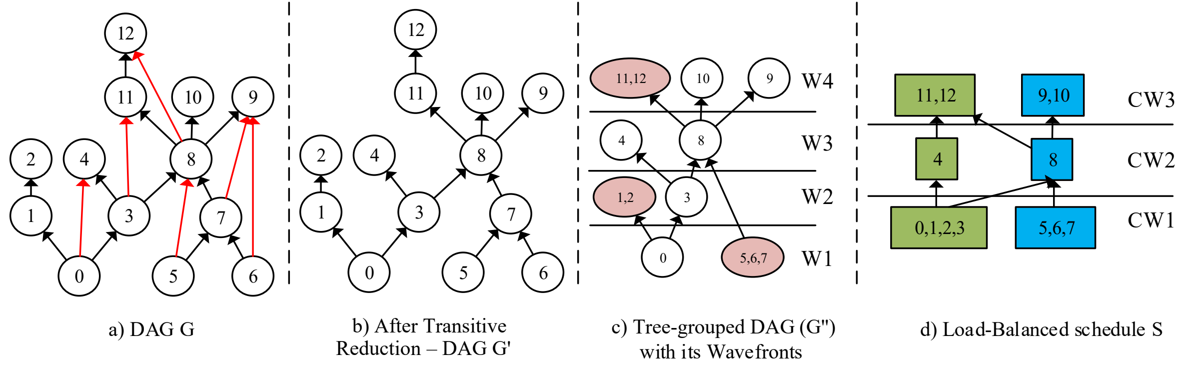

Fig. 2. Steps in HDagg to build the schedule in Figure 1(e) from the DAG in Figure 1(a). The first step of the algorithm removes transitive edges (shown with red edges in Figure 2(a)) and constructs DAG G' in Figure 2(b). Then vertices that form a subtree are merged to improve locality as shown in Figure 2(c). Pink vertices in DAG G'' in Figure 2(c) represent a merged subtree. The second step of HDagg coarsens wavefronts of G'' to improve locality while preserving load balance as shown in Figure 2(d).

as G'. Line 2 of Algorithm 1 uses the list of sink vertices in the reduced graph G' as the initial partitioning for $\mathcal{T}$. Then, in Lines 3–19, the algorithm iterates over every entry in $H \in \mathcal{T}$ to determine if H can be expanded as a subtree. For this purpose, HDagg finds the parent vertices of every vertex $v \in H$ using *parents* in Line 7. If v and all of its parents A create a tree, i.e. each vertex in A only has one outgoing edge, the algorithm expands the subtree in H with vertices in A in Line 9. Otherwise, each vertex in A is appended to $\mathcal{T}$ as sink vertices in upcoming iterations.

Example. Figure 2(a) shows graph G with its transitive edges colored in red. The output of the transitive reduction is graph G' shown in Figure 2(b) which has 6 less edges compared to DAG G. The original G only has one subtree, i.e. vertices 1 and 2, while the reduced DAG G' has two additional subtrees formed by the set of $\{11, 12\}$ and $\{5, 6, 7, 8\}$. Sink vertices of G' are $\{\{2\}, \{4\}, \{9\}, \{10\}, \{12\}\}$; used to pre-initialize the partitioning $\mathcal{T}$. A BFS from vertex 12 results in groupings 12 and 11. Vertex 8 has multiple outgoing edges and thus is added as a sink vertex to $\mathcal{T}$ and will not be grouped with vertices 11 and 12. When the traversal starts from vertex 8, merging vertex 8 with its parents does not form a subtree. However, grouping vertex 7 with its parents $\{5, 6\}$ creates in a subtree, thus, these vertices are grouped.

C. Load-balance Preserving Wavefront Coarsening

When applying wavefront coarsening to trees, the number of independent workloads is known from the number of vertices in the wavefronts. However, non-tree DAGs have irregular outgoing edges, and thus it is challenging to ensure load balance with wavefront coarsening. The second step of HDagg coarsens wavefronts using a load-balance preserving (LBP) strategy that uses a novel cost metric, i.e. potential gain proxy (PGP), to measure workloads. Using a bin packing strategy, LBP further explores the trade-off between locality and load balance.

Because vertices of a general DAG have different number of outgoing edges, wavefront coarsening can significantly affect the number of connected components and thus load balance. HDagg merges wavefronts up until the wavefront that LBP can ensure load balance using a bin packing strategy and with respect to the PGP metric.

LBP first coarsens the wavefronts of the grouped DAG resulting from the first step and finds connected components (CCs) in each coarsened wavefront using a variant of the Shiloach-Vishkin [12] algorithm. It then packs vertices of a coarsened wavefront into a maximum of p bins using a bin packing strategy to make approximately equal bins with respect to the PGP metric. For low overhead packing, HDagg uses a first-fit strategy [13] where a connected component is assigned to the first bin that is not balanced. Along with packing, vertices are ordered inside bins with the smallest ID first to improve spatial locality.

In Lines 21–35, HDagg starts from the first wavefront and then merges consecutive wavefronts until a wavefront cut. A cut occurs if continuing to merge with the next wavefront results in load imbalance. In Line 21, the algorithm computes wavefronts of the grouped DAG G'' and in Lines 22–23, it initializes variables with the first two wavefronts. Then as shown in Line 25, HDagg finds connected components for the selected range of wavefronts ($W[cut : i]$) and packs them into a maximum of p bins (*BinPack*). The algorithm uses PGP to determine if the partitioning is balanced with respect to the load balance threshold ϵ (Line 26). The balanced partitions (S_{prev}) are put into the final schedule as shown in Line 30.

The created coarsened wavefronts are not always balanced because some matrices do not have enough workloads due to their sparsity pattern. Also, for some matrices with many wavefronts, the accumulated load imbalance of coarsened wavefronts becomes noticeable. Thus, the HDagg algorithm in Lines 36–38 computes the accumulation of imbalance cost across all coarsened wavefronts using the PGP metric. If the accumulated cost is higher than the load balance threshold, it creates fine-grain tasks by disabling bin packing. Fine-grain tasks allow the scheduler to create a balance execution at runtime. When the final schedule is balanced, bin packing is

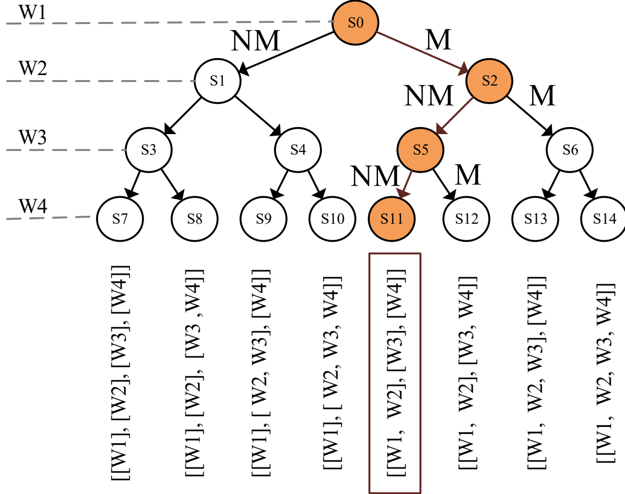

Fig. 3. Using a binary tree, this figure shows the decision space of wavefront coarsening choices that LBP explores for four wavefronts of the DAG shown in Figure 2(a). The highlighted path (in orange) from the root to the leaf vertex $S11$ shows how LBP coarsens wavefronts in the DAG. Each vertex in the path corresponds to a wavefront. An incoming edge to a vertex shows if the wavefront is merged with previous coarsened wavefronts. For instance, $S5$ shows $W3$ will not be merged with merged $W1$ and $W2$ ($[W1, W2]$) and $S6$ shows that $W3$ is not merged with $[W1, W2]$.

enabled which in return improves locality.

Example. Figure 2(d) shows the created schedule for two cores using the second step of HDagg. The input DAG to the second step along with its initial wavefront W is shown in Figure 2(c). DAG G'' has $l = 4$ wavefronts which are stored in W. The second step of HDagg starts from the first wavefront and merges wavefronts as shown in Figure 3. Each circle in Figure 3 represents a decision related to a wavefront, and each edge shows a choice (merged or not merged) that HDagg makes during the second step. The algorithm first coarsens $W1$ and $W2$. Then it detects two connected components, $\{\{1, 2\}, 0, 3\}$ and $\{5, 6, 7\}$, from these coarsened wavefronts. Assuming $p = 2$, the bin pack algorithm assigns the first connected component to the first bin, i.e. the first core, and the second one to the second bin. By merging wavefronts $W3$ with the coarsened wavefront $[W1, W2]$ shown as $CW1$ in figure 2(d), one connected component consist of $\{1, 2, 3, \dots 8\}$ vertices will be formed, which would result in one connected component that is not enough for all cores. Thus, the algorithm does not merge $CW1$ with $W3$. The final coarsened-wavefronts are $[[W1, W2], [W3], [W4]]$. Waves $W1$ and $W2$ are merged but $W3$ and $W4$ are not merged. In Figure 2(d), $CW1$, $CW2$, and $CW3$ correspond to $[W1, W2]$, $[W3]$, and $[W4]$ respectively, where the bin packing strategy is applied to coarsened wavefronts.

D. Potential Gain Proxy

The Potential Gain Proxy (PGP) is a metric to measure the load balance of aggregated iterations statically. It shows the reduction in runtime if all cores have a balanced workload.

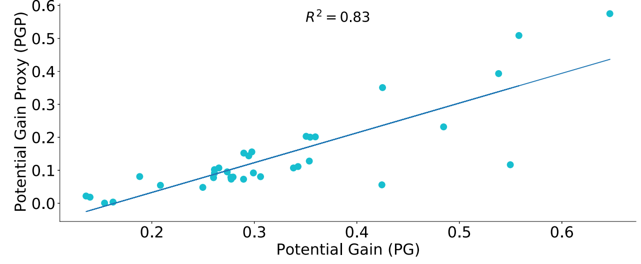

Fig. 4. The correlation between the potential gain proxy (PGP) metric and the measured potential gain (PG) for different matrices for SpTRSV.

Consider the set $\mathcal{B} = \{B_1, B_2, \dots B_p\}$ where B_i is the amount of workloads assigned to each core. Each B_i is computed as $B_i = \sum_{j \in I_i} C_j$ where I_i is the list of vertices assigned to core i and C_j is the cost assigned to vertex $j \in I_i$. PGP is computed via:

$$PGP = 1 - \frac{\bar{\mathcal{B}}}{\max_{1 \leq i \leq p}(B_i)} \quad (1)$$

where $\bar{\mathcal{B}}$, and $\max_{1 \leq i \leq p}(B_i)$ are the average and maximum values over the set $\mathcal{B}$ respectively.

When the schedule is balanced, i.e. all cores have the same amount of workload, PGP becomes zero. In the worse case scenario, all workloads are assigned to a single core leading to PGP of $1 - 1/p$. Any value in between shows the percentage that the runtime can be improved by making the schedule balanced. For instance, assuming $p = 2$ and that all tasks are assigned to one core, PGP becomes $1 - 1/2 = 50\%$. This means that runtime is reduced (improved) by 50% from improving load balance. To show the accuracy of PGP, we measure the potential gain (PG) using CPU compute time obtained from using PAPI [14] and Vtune. We also compute PGP using the cost function C. Figure 4 shows the correlation between PGP and the measured PG for our dataset and for SpTRSV. As shown, there is a linear correlation between PGP and the measured potential gain with the coefficient of determination or R^2= 0.83, this indicates PGP is a good approximation of PG.

E. Computational Complexity

For the theoretical complexity of inspector overhead, we use an approximation of transitive reduction from [4] with the time complexity of $O(|\mathcal{E}| \times E[D] + |\mathcal{V}| \times Var[D])$ where $|\mathcal{V}|$ and $|\mathcal{E}|$ are the number of vertices and the number of edges in the DAG respectively. D is defined as the average number of non-zeros per row of each matrix and $E[D]$ and $Var[D]$ are the expected value and the variance of D [4] respectively. Aggregating densely connected vertices and the modified BFS algorithm in the first step have a time complexity of $O(|\mathcal{E}| + |\mathcal{V}|)$. Finding connected components after merging two consecutive wavefronts is the most computationally expensive stage in the second step. The connected component algorithm in HDagg [12] has the computational

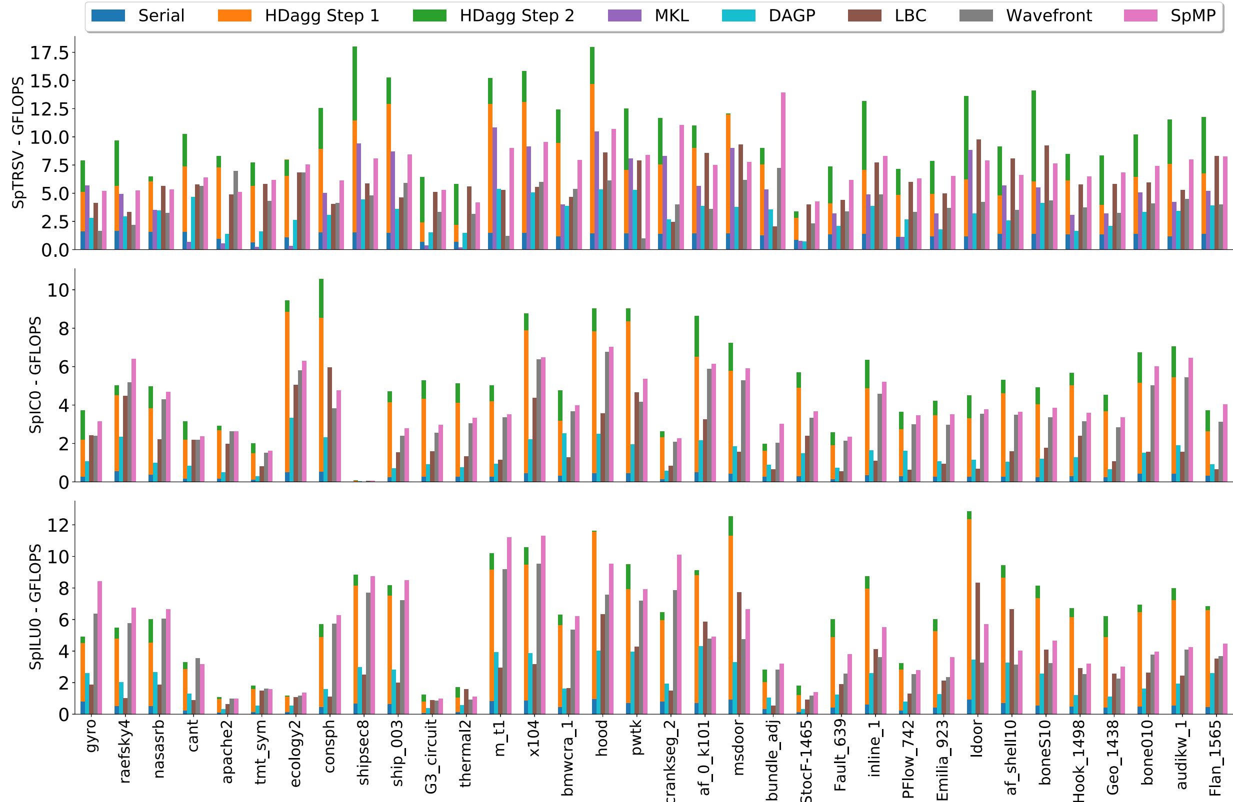

Fig. 5. The performance of SpTRSV (top), SpIC0 (middle), and SpILU0 (bottom) in GFlops. Different steps in HDagg are shown in the stacked bar, Step 1 is the effect of aggregating densely connected vertices, Step 2 shows the effect of wavefront coarsening combined with the bin packing scheme.

complexity of $O(|\mathcal{E}| \times log(|\mathcal{V}|)$ on a single core. Since the connected component algorithm is called at most the number of wavefronts, l, its complexity is $O(l \times |\mathcal{E}| \times log(|\mathcal{V}|))$. With enough parallel cores the complexity of this step converges to $O(l \times log(|\mathcal{V}|))$ [12].

V. Experimental Results

We compare the performance of HDagg with MKL [15], DAGP [1], LBC [7], wavefront parallelism techniques [2] (which we refer to as *Wavefront*), and SpMP [4] for sparse matrix computations. Three sparse kernels, namely, SpTRSV, SpIC0, and SpILU0, are used for comparison. Since MKL's implementations of SpILU0 and SpIC0 are not parallel, we only compare to MKL for SpTRSV. HDagg's code is available from *https://github.com/sympiler/aggregation* and the HDagg evaluation framework is available from *https://github.com/BehroozZare/HDagg-benchmark*.

We use 34 symmetric positive definite (SPD) matrices from the Suite Sparse matrix collection [16]; SPD matrices are selected so that SpIC0 becomes numerically stable. To demonstrate the scalability of HDagg we choose matrices of different sizes from 51×10^3 number of non-zeros to 59×10^6 number of non-zeros. The matrix sparsity patterns are diverse not to favor one algorithm. For example, *(i)* some have many chains in their DAGs, helping DAGP; *(ii)* some matrices have large average parallelism, where average parallelism is obtained by dividing the number of vertices by the number of wavefronts, helping Wavefront/SpMP; *(iii)* and some need a small number of edges to become chordal, helping LBC.

The algorithms/libraries are executed on an AMD and an Intel processor. All analysis is conducted on Intel. The *Intel processor* is an Intel(R) Xeon(R) Gold 6248 CPU with a 2.50GHz clock rate and 28MB LLC cache with 20 cores. The *AMD processor* is AMD EPYC 7742 with a 2.25GHz clock rate and 256MB LLC cache with 64 cores. DAGP, LBC, and SpMP libraries are open source and their latest publicly available version is used, and the MKL version is 2020.4.304. The default settings or best performing configurations of each library are used for comparison. We use the MKL inspector and set the `expected_call` parameter to 1000 to ensure that MKL applies all the optimizations and achieves its maximum performance when executing in parallel. For a fair comparison with DAGP, different partition numbers (k) are investigated, and the result for partitioning with the best speedup, i.e. $k = 1000$, is reported. All source codes are compiled with GCC v.8.3.0 using the `-O3` optimization flag with "close" thread-binding. The median of 10 executions for each experiment is reported. For all algorithms, the matrices are first reordered with Metis [17] and then passed as input.

TABLE I
THE AVERAGE SPEEDUP OF HDAGG COMPARED TO OTHER ALGORITHMS FOR THREE SPARSE KERNELS ON INTEL AND AMD ARCHITECTURES

	Kernel	DAGP	LBC	Wavefront	SpMP	MKL
Intel	SpTRSV	3.8×	5.3×	2.9×	1.56×	3.56×
	SpIC0	4.12×	2.67×	1.52×	1.34×	–
	SpILU0	3.41×	2.82×	1.62×	1.47×	–
AMD	SpTRSV	8.34×	4.67×	13.53×	1.13×	3.36×
	SpIC0	10.01×	10.62×	1.31×	1.12×	–
	SpILU0	7.13×	6.96×	1.27×	1.06×	–

A. *Executor Evaluation*

Table I shows the average speedup of HDagg over other algorithms on Intel and AMD processors. Across three kernels, SpTRSV, SpIC0, and SpILU0, HDagg outperforms DAGP, LBC, Wavefront, and SpMP with an average speedup of $3.87\times$, $3.41\times$, $1.95\times$, and $1.43\times$ on Intel processor and $8.41\times$, $7.01\times$, $2.83\times$, and $1.10\times$ on AMD processor respectively. Hereafter, we will use results on Intel to analyze the performance of HDagg.

Figure 5 shows, per matrix, the performance of HDagg versus the algorithms mentioned above on the Intel processor. For the SpTRSV and SpIC0 kernels, HDagg provides better performance in over 94% of the matrices, while for SpILU0, we beat other algorithms in 73% of the matrices. SpILU0 is more challenging to optimize, and HDagg does not provide strong speedups on some matrices for this kernel. Hence, we will use SpILU0 for analysis and comparison for the rest of this section.

To analyze the performance of HDagg, DAGP, LBC, Wavefront, and SpMP, we use the three *performance metrics* of locality, load balance, and synchronization (Figure 6). The average memory access latency [18] is used as a metric to measure locality. To measure load balance, we compute the potential gain as described in Section IV-D; the number of cycles per core is used as the cost function (C) in Equation 1. The number of point-to-point synchronization for each algorithm is measured to evaluate synchronization overhead. PAPI's performance counters [14] are used to measure architecture information needed in computations related to the locality and load balance metrics. For algorithms that use global barriers, we calculate its equivalent number of point-to-point synchronization by multiplying the number of global barriers by $p \times log(p)$ [4] where p is the number of cores.

Figure 6 compares the performance metrics in HDagg to DAGP, LBC, Wavefront, and SpMP for SpILU0 on the Intel processor. Table II summarizes Figure 6 with each row showing the average improvement of a performance metric in HDagg compared to aforementioned algorithms. As demonstrated, compared to DAGP and LBC, HDagg improves locality and load balance. However, compared to SpMP and Wavefront, HDagg does not improve load balance but provides better locality and less synchronization. In the following, we compare the performance of each algorithm with HDagg using the performance metrics shown in Figure 6 and Table II.

TABLE II
PERFORMANCE METRICS IN HDAGG VS. OTHER ALGORITHMS FOR SPILU0 ON THE INTEL PROCESSOR.

Performance Metrics	DAGP	LBC	Wavefront	SpMP
Locality Improvement	2.66×	2.33×	1.52×	1.44×
Load Balance Improvement	2.60×	2.27×	0.56×	0.34×
Synchronization Reduction	5.07×	0.07×	7.95×	1.49×

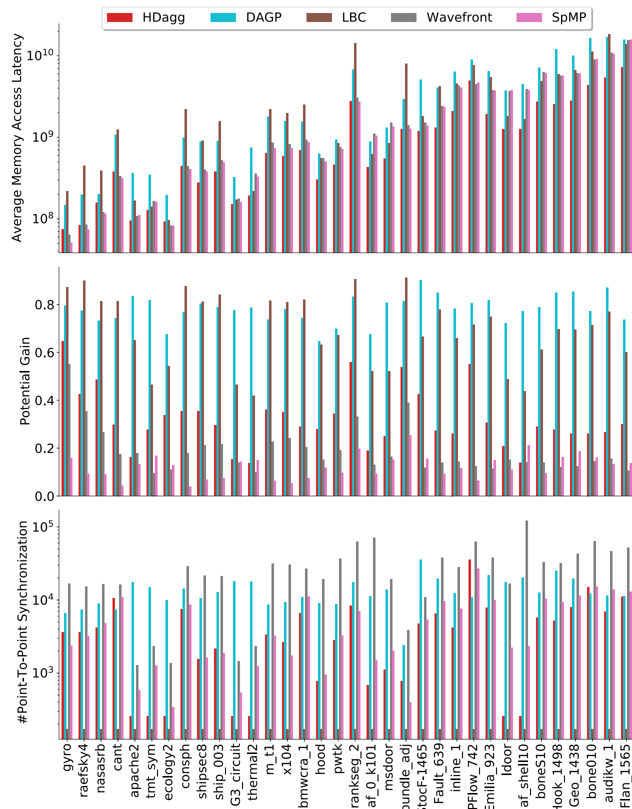

Fig. 6. Performance metrics of different algorithms for SpILU0 on the Intel processor. The top plot shows locality measured by average memory access latency (in log scale). The middle plot demonstrates load balance using the potential gain metric, and the bottom plot shows the number of point-to-point synchronizations used that indicate synchronization overhead (in log scale).

HDagg compared to DAGP. As shown in Figure 6, for all matrices in our benchmark, HDagg outperforms DAGP on the performance metrics leading to the performances reported in Figure 5. Per Table II, HDagg improves locality and load balance over DAGP on average $2.66\times$ and $2.60\times$ respectively and reduces synchronization $5.07\times$ over DAGP. We use Figure 7 to analyze the load balance trend shown in Figure 6 middle. A wavefront is imbalanced if the number of independent workloads in the wavefront is less than the number of cores p. Figure 7 shows the ratio of imbalanced wavefronts over the total number of wavefronts per matrix/algorithm (we call this the *load imbalance ratio*, the lower, the better). As demonstrated, DAGP has the highest load imbalance ratio compared to other algorithms leading to an imbalanced load

TABLE III

MATRIX CATEGORIZATION BASED ON THE NUMBER OF NON-ZEROS (nnz) AND AVERAGE PARALLELISM. ALL COLUMNS EXCEPT FAST MATRICES PERCENTAGE ARE THE REPORT AVERAGE AMONGST MATRICES FOR THAT CLASS. THE FAST MATRICES PERCENTAGE AND SPEEDUP COLUMNS ARE MEASURED WITH RESPECT TO THE BEST OF SPMP AND WAVEFRONT.

Category	Average nnz per Wavefront	Locality Improvement	Load Balance Improvement	Fast Matrices Percentage	Speedup
$nnz > 10^7$	61747	1.90	0.47	93%	1.75
$nnz < 10^7$, $Average\ Parallelism > 400$	47280	1.37	0.43	100%	1.26
$nnz < 10^7$, $Average\ Parallelism < 400$	7787	0.92	0.22	63%	0.90

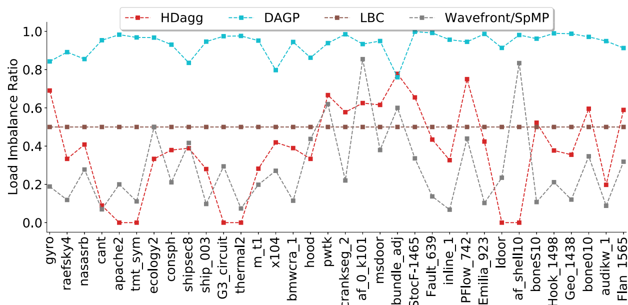

Fig. 7. The ratio of imbalanced wavefronts over the total number of wavefronts per matrix/algorithm, i.e. *load imbalance ratio.* Lower is better.

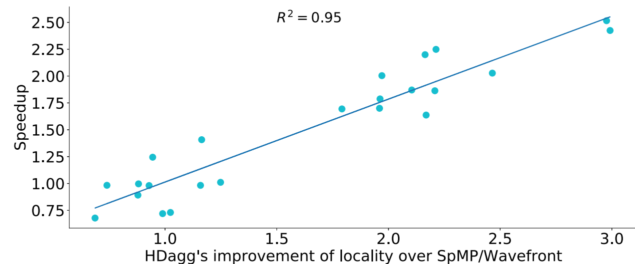

Fig. 8. The correlation between HDagg's speedup over SpMP/Wavefront, shown as Speedup, and HDagg's improvement of locality over SpMP/Wavefront.

execution. HDagg also improves locality and synchronization over DAGP. Hence we outperform DAGP on all matrices.

HDagg compared to LBC. HDagg outperforms LBC for all matrices in our dataset per Figure 5. As shown in Figure 6, HDagg improves load balance and locality over LBC for all matrices with an average of 2.27× and 2.33× respectively (from Table II). LBC always creates two wavefronts where one of the wavefronts has fewer than p (the number of cores) workloads, resulting in a load imbalance ratio of 50% for all matrices. For most matrices, HDagg has a load imbalance ratio of less than 50% and hence a better load balance than LBC. As shown in Table II, the number of synchronizations in LBC is on average 0.07× lower than HDagg. However, synchronization is not a bottleneck in HDagg's execution. The synchronization overhead in HDagg contributes to on average 0.0011% of the execution time, which is negligible. Hence, with only improving load balance and locality HDagg is able to outperform LBC on all matrices.

HDagg compared to SpMP and Wavefront. HDagg is faster than SpMP/Wavefront in 88% of matrices (Figure 5) with an average speedup of 1.47×. It improves locality in most of the matrices compared to SpMP/Wavefront. However, as shown in Figure 6, SpMP/Wavefront overall provides a better load balance compared to HDagg (as also demonstrated in Figure 7 with a lower load imbalance ratio for SpMP/Wavefront for most matrices). Synchronization overhead is improved in HDagg for most matrices. However, because the time spent on synchronization is negligible in HDagg, the locality and load balance metrics are the major contributing factors to HDagg's speedup or slowdown.

Table III categorizes the matrices in our dataset, based on matrix size (non-zeros in the matrix) and average parallelism. Average parallelism is an indicator for load balance, obtained by dividing the number of vertices by the number of wavefronts. For each category, we show the average number of non-zeros per wavefront as a measure for potential locality improvement. With more average number of non-zeros per wavefront, the SpILU0 kernel accesses more of the already computed factors while executing iterations of a wavefront, which provides more opportunities to improve locality. As shown in Table III for the first two categories (rows 1 and 2), HDagg outperforms SpMP/Wavefront on more than 90% of the matrices. For the matrices in these two categories, the average number of non-zeros per wavefront, an indicator of data reuse between iterations across wavefronts, is relatively large. Figure 8 shows that HDagg's speedup over SpMP/Wavefront is correlated to HDagg's improvement of locality over SpMP/Wavefront with a coefficient of determination (R^2) of 95%, for the matrices in the first two categories in Table I. This correlation indicates that HDagg's two-step aggregation method successfully utilizes reuse opportunities between iterations across wavefronts and improves locality.

To summarize, for the matrices in the first two categories in Table III, HDagg provides better performance than SpMP/Wavefront because locality is important for these matrices, and HDagg improves locality noticeably more than SpMP/Wavefront. The matrices in the third category have a low number of non-zero per wavefront. Thus, HDagg can not always improve locality. Also, the average parallelism is low in these matrices, and algorithms such as SpMP and Wavefront with better load balance strategies will outperform HDagg in some matrices in this category, i.e. in 37% of the matrices.

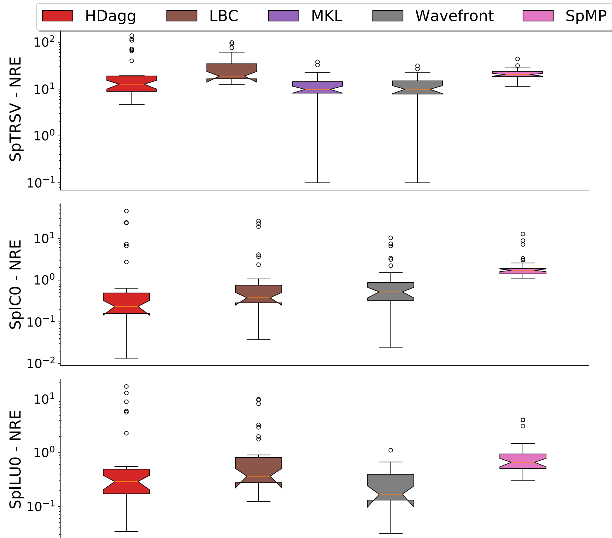

Fig. 9. The number of required kernel executions (NRE) to amortize the cost of inspection for SpTRSV (top), SpIC0 (middle), and SpILU0 (bottom).

B. Inspection Overhead

Figure 9 shows the Number of Required kernel Executions (NRE) to amortize the cost of inspection. NRE is calculated from

$$NRE = \frac{inspector_time}{(sequential_time - parallel_time)} \quad (2)$$

Since DAGP needs an average of 5305 executions to amortize its inspection time, we do not report its NRE in graphs. LBC, Wavefront, SpMP, and HDagg, on average, require 24, 9.4, 21, and 16 kernel executions, respectively, to amortize their inspector overhead for SpTRSV. These overheads are quickly amortized in iterative solvers where a kernel is executed tens of thousands of times [19], [20]. For SpIC0, and SpILU0, NRE of HDagg is on average 0.38 and 0.41, respectively. An NRE of lower than one means that total inspection and executor time (i.e. single kernel execution) is smaller than the sequential time.

VI. Related Work

Sparse matrix computations have been optimized in numerous libraries each targeting a specific class of sparse kernels and a specific computing platform. For example, the sparse LU and Cholesky factorization methods have been accelerated in work such as MKL [15], SuperLU [21], and PaStix [22] on shared memory and distributed memory architectures [23], [24]. The sparse triangular solve has been optimized in [25]–[28]. Kernels with parallel loops such as the sparse matrix-vector multiplication have also been optimized in implementations such as [29]–[32]. Libraries are often not portable and have to be hand-optimized at a high cost, also their optimizations are often specialized for a specific application. For example, KLU is optimized for circuit simulation problems [33].

Domain-specific compilers leverage information from the domain to determine a sequence of optimizations that the compiler can apply to the sparse code. For example work such as [34]–[36] optimize stencil computations, Taichi [37] and TACO [38] optimize tensor algebra, and signal processing applications are optimized in [21]. Recent work has extended compilers to build inspectors that execute at runtime to analyze the indirect memory access patterns in sparse kernels [2], [39], [40]. Amongst these approaches, wavefront parallelism techniques [2]–[4], [40]–[43] have gained notable popularity. These approaches either use a manually written inspector [3], [42], [43] or use automation to simplify parts of the inspector [40], [44], [45]. The inspectors either schedule the wavefronts to execute one after another [2] or works such as LBC use load-balanced coarsening approaches [4] to create parallel workloads for the sparse computation. DAG partitioning approaches such as DAGP [1] also inspect the dependencies in the sparse computation and create a schedule for the sparse computation that can execute in parallel.

VII. Conclusion

This paper proposes HDagg, a novel iteration aggregation approach that creates an efficient partitioning of the data dependence graphs from sparse matrix computations that do not have a tree-structure DAG. The created partitions from HDagg lead to a load-balanced parallel execution of the sparse code and improve data locality. HDagg is composed of a vertex aggregation phase and a novel wavefront coarsening step that uses a load balancing metric called PGP. It is implemented as an inspector-executor framework and outperforms state-of-the-art iteration schedulers of sparse kernels such as SpTRSV up to 13 fold.

VIII. Acknowledgement

This work was supported in part by NSERC Discovery Grants (RGPIN-06516, DGECR00303), the Canada Research Chairs program, and U.S. NSF awards NSF CCF-1814888, NSF CCF1657175, NSF CCF-1563732; used the Extreme Science and Engineering Discovery Environment (XSEDE) [46] which is supported by NSF grant number ACI-1548562; and was enabled in part by Compute Canada and Scinet.

References

[1] J. Herrmann, M. Y. Ozkaya, B. Uçar, K. Kaya, and U. V. Catalyurk, "Multilevel algorithms for acyclic partitioning of directed acyclic graphs," *SIAM Journal on Scientific Computing*, vol. 41, no. 4, pp. A2117–A2145, 2019.

[2] A. Venkat, M. S. Mohammadi, J. Park, H. Rong, R. Barik, M. M. Strout, and M. Hall, "Automating wavefront parallelization for sparse matrix computations," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 480–491.

[3] J. Anantpur and R. Govindarajan, "Runtime dependence computation and execution of loops on heterogeneous systems," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2013, pp. 1–10.

[4] J. Park, M. Smelyanskiy, N. Sundaram, and P. Dubey, "Sparsifying synchronization for high-performance shared-memory sparse triangular solver," in *International Supercomputing Conference*. Springer, 2014, pp. 124–140.

[5] E. Anderson and Y. Saad, "Solving sparse triangular linear systems on parallel computers," *International Journal of High Speed Computing*, vol. 1, no. 01, pp. 73–95, 1989.

[6] J. H. Saltz, "Aggregation methods for solving sparse triangular systems on multiprocessors," *SIAM journal on scientific and statistical computing*, vol. 11, no. 1, pp. 123–144, 1990.

[7] K. Cheshmi, S. Kamil, M. M. Strout, and M. M. Dehnavi, "Parsy: Inspection and transformation of sparse matrix computations for parallelism," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 779–793.

[8] K. Cheshmi, S. Kamil, M. M. Strout, and M. M. Dehnavi, "Sympiler: transforming sparse matrix codes by decoupling symbolic analysis," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–13.

[9] T. A. Davis, S. Rajamanickam, and W. M. Sid-Lakhdar, "A survey of direct methods for sparse linear systems," *Acta Numerica*, vol. 25, pp. 383–566, 2016.

[10] E. Chow, "A priori sparsity patterns for parallel sparse approximate inverse preconditioners," *SIAM Journal on Scientific Computing*, vol. 21, no. 5, pp. 1804–1822, 2000.

[11] Y. Saad and M. H. Schultz, "Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM Journal on scientific and statistical computing*, vol. 7, no. 3, pp. 856–869, 1986.

[12] Y. Shiloach and U. Vishkin, "An o (log n) parallel connectivity algorithm," Computer Science Department, Technion, Tech. Rep., 1980.

[13] G. Dósa and J. Sgall, "First Fit bin packing: A tight analysis," in *30th International Symposium on Theoretical Aspects of Computer Science (STACS 2013)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), N. Portier and T. Wilke, Eds., vol. 20. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013, pp. 538–549. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2013/3963

[14] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with papi-c," in *Tools for High Performance Computing 2009*. Springer, 2010, pp. 157–173.

[15] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang, "High-performance computing on the intel xeon phi," *Springer*, vol. 5, p. 2, 2014.

[16] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.

[17] G. Karypis and V. Kumar, "Multilevel algorithms for multi-constraint graph partitioning," in *SC'98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. IEEE, 1998, pp. 28–28.

[18] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[19] M. Benzi, J. K. Cullum, and M. Tuma, "Robust approximate inverse preconditioning for the conjugate gradient method," *SIAM Journal on Scientific Computing*, vol. 22, no. 4, pp. 1318–1332, 2000.

[20] K. Cheshmi, D. M. Kaufman, S. Kamil, and M. M. Dehnavi, "Nasoq: numerically accurate sparsity-oriented qp solver," *ACM Transactions on Graphics (TOG)*, vol. 39, no. 4, pp. 96–1, 2020.

[21] X. S. Li, "An overview of superlu: Algorithms, implementation, and user interface," *ACM Transactions on Mathematical Software (TOMS)*, vol. 31, no. 3, pp. 302–325, 2005.

[22] P. Hénon, P. Ramet, and J. Roman, "Pastix: a high-performance parallel direct solver for sparse symmetric positive definite systems," *Parallel Computing*, vol. 28, no. 2, pp. 301–321, 2002.

[23] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster, "A fully asynchronous multifrontal solver using distributed dynamic scheduling," *SIAM Journal on Matrix Analysis and Applications*, vol. 23, no. 1, pp. 15–41, 2001.

[24] J. W. Demmel, J. R. Gilbert, and X. S. Li, "An asynchronous parallel supernodal algorithm for sparse gaussian elimination," *SIAM Journal on Matrix Analysis and Applications*, vol. 20, no. 4, pp. 915–952, 1999.

[25] R. Li and Y. Saad, "Gpu-accelerated preconditioned iterative linear solvers," *The Journal of Supercomputing*, vol. 63, no. 2, pp. 443–466, 2013.

[26] B. Yılmaz, B. Sipahioğrlu, N. Ahmad, and D. Unat, "Adaptive level binning: A new algorithm for solving sparse triangular systems," in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, 2020, pp. 188–198.

[27] X. Wang, W. Liu, W. Xue, and L. Wu, "swsptrsv: A fast sparse triangular solve with sparse level tile layout on sunway architectures," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2018, pp. 338–353.

[28] R. Vuduc, S. Kamil, J. Hsu, R. Nishtala, J. W. Demmel, and K. A. Yelick, "Automatic performance tuning and analysis of sparse triangular solve." ICS, 2002.

[29] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarath, and P. Sadayappan, "Fast sparse matrix-vector multiplication on gpus for graph applications," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 781–792.

[30] S. Kamin, M. J. Garzarán, B. Aktemur, D. Xu, B. Yılmaz, and Z. Chen, "Optimization by runtime specialization for sparse matrix-vector multiplication," in *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, 2014, pp. 93–102.

[31] C. Liu, B. Xie, X. Liu, W. Xue, H. Yang, and X. Liu, "Towards efficient spmv on sunway manycore architectures," in *Proceedings of the 2018 International Conference on Supercomputing*, 2018, pp. 363–373.

[32] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *SC'07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. IEEE, 2007, pp. 1–12.

[33] T. A. Davis and E. Palamadai Natarajan, "Algorithm 907: Klu, a direct sparse solver for circuit simulation problems," *ACM Transactions on Mathematical Software (TOMS)*, vol. 37, no. 3, pp. 1–17, 2010.

[34] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *Acm Sigplan Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[35] L. Polok, V. Ila, M. Solony, P. Smrz, and P. Zemcik, "Incremental block cholesky factorization for nonlinear least squares in robotics." in *Robotics: Science and Systems*, 2013, pp. 328–336.

[36] R. C. Dorf, *Electronics, power electronics, optoelectronics, microwaves, electromagnetics, and radar*. CRC press, 2018.

[37] Y. Hu, T.-M. Li, L. Anderson, J. Ragan-Kelley, and F. Durand, "Taichi: a language for high-performance computation on spatially sparse data structures," *ACM Transactions on Graphics (TOG)*, vol. 38, no. 6, pp. 1–16, 2019.

[38] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017. [Online]. Available: https://doi.org/10.1145/3133901

[39] M. M. Strout, M. Hall, and C. Olschanowsky, "The sparse polyhedral framework: Composing compiler-generated inspector-executor code," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1921–1934, 2018.

[40] X. Zhuang, A. E. Eichenberger, Y. Luo, K. O'Brien, and K. O'Brien, "Exploiting parallelism with dependence-aware scheduling," in *2009 18th International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2009, pp. 193–202.

[41] L. Rauchwerger, N. M. Amato, and D. A. Padua, "Run-time methods for parallelizing partially parallel loops," in *Proceedings of the 9th international conference on Supercomputing*, 1995, pp. 137–146.

[42] M. M. Strout, L. Carter, J. Ferrante, J. Freeman, and B. Kreaseck, "Combining performance aspects of irregular gauss-seidel via sparse tiling," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2002, pp. 90–110.

[43] M. Naumov, "Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the gpu," *NVIDIA Corp., Westford, MA, USA, Tech. Rep. NVR-2011*, vol. 1, 2011.

[44] J. R. Gilbert and R. Schreiber, "Highly parallel sparse cholesky factorization," *SIAM Journal on Scientific and Statistical Computing*, vol. 13, no. 5, pp. 1151–1172, 1992.

[45] M. S. Mohammadi, T. Yuki, K. Cheshmi, E. C. Davis, M. Hall, M. M. Dehnavi, P. Nandy, C. Olschanowsky, A. Venkat, and M. M. Strout, "Sparse computation data dependence simplification for efficient compiler-generated inspectors," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 594–609.

[46] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson *et al.*, "Xsede: accelerating scientific discovery," *Computing in science & engineering*, vol. 16, no. 5, pp. 62–74, 2014.