ParSy: Inspection and Transformation of Sparse Matrix Computations for Parallelism

Kazem Cheshmi
Department of Computer Science
University of Toronto
Toronto, Canada
kazem@cs.toronto.edu

Shoaib Kamil
Adobe Research
New York, USA
kamil@adobe.com

Michelle Mills Strout
Department of Computer Science
University of Arizona
Tucson, USA
mstrout@cs.arizona.edu

Maryam Mehri Dehnavi
Department of Computer Science
University of Toronto
Toronto, Canada
mmehride@cs.toronto.edu

***Abstract*—ParSy is a framework that generates parallel code for sparse matrix computations. It uses a novel inspection strategy along with code transformations to generate parallel code for shared memory processors that is optimized for locality and load balance. Code generated by existing automatic parallelism approaches for sparse algorithms can suffer from load imbalance and excessive synchronization, resulting in performance that does not scale well on multi-core systems. We propose a novel task coarsening strategy that creates well-balanced tasks that can execute in parallel. ParSy-generated code outperforms existing highly-optimized sparse matrix codes such as the Cholesky factorization on multi-core processors with speed-ups of 2.8× and 3.1× over the MKL Pardiso and PaStiX libraries respectively.**

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. Data dependence information required for parallelizing sparse codes is dependent on the matrix structure, so parallel codes may use more synchronization than necessary; in addition, to achieve high parallel efficiency, the work must be evenly distributed among cores, but this distribution also depends on the matrix structure.

A large number of parallel sparse libraries, such as Intel's Math Kernel Library (MKL) [56], Pardiso [56], [47], PaStiX [23], and SuperLU [32], provide manually-optimized parallel implementations of sparse matrix algorithms and are some of the most commonly-used libraries in simulations using sparse matrices. These libraries differ in the kind of numerical methods they support and use numerical-method-specific code at runtime, during a phase called *symbolic factorization*, to determine data dependencies. Based on this dependence information, different libraries implement different forms of parallelism. For example, PaStiX uses static scheduling of a fine-grained task graph based on empirical measurements of expected runtime for each task; in contrast, MKL Pardiso implements a form of dynamic scheduling for its fine-grained task graph.

Previous work has extended compilers to resolve memory access patterns in sparse codes by building runtime *inspectors* to examine the nonzero structure and using *executors* to transform code execution and implement parallelism [54], [45], [59]. Inspectors use runtime information to build directed acyclic graphs (DAGs) that expose data dependence relations. The DAGs are traversed in topological order to create a list of *level sets* that represent iterations that can execute in parallel; this is known as *wavefront parallelism*. Synchronization between level sets ensures the execution respects data dependencies. However, synchronization between levels in wavefront parallelism can lead to high overheads since the number of levels increases with the DAG critical path. For sparse kernels such as Cholesky with non-uniform workloads, wavefront methods can additionally lead to load imbalance. Frameworks such as Sympiler [9] have demonstrated the value of creating specializations of sparse matrix methods for exploiting specific matrix structure. However, this approach has only been demonstrated for single-threaded implementations.

In this work, we present an inspection strategy for parallelism on multi-core architectures for sparse matrix kernels. Our proposed inspector applies a novel Load-Balanced Level Coarsening (LBC) algorithm on the data dependence graph to create well-balanced coarsened level sets, which we call the hierarchical level set (H-Level set), mitigating load imbalance and excessive synchronization present in wavefront parallelism. This inspector is implemented in a framework called ParSy, which uses information from the matrix sparsity and the numerical method to obtain data dependencies. The inspector in ParSy can be used for sparse linear algebra libraries, inspector-executor compiler methods, or from within sparsity-specific code generators such as Sympiler.

Our primary focus is complex sparse matrix algorithms where loop-carried data dependencies make efficient parallelization challenging, such as sparse triangular solve, as well as matrix methods that introduce fill-ins (nonzeros) during computation, such as Cholesky factorization. The main contributions of this work include:

- A new LBC strategy that inspects sparse kernel data dependence graphs for parallelism while maintaining an efficient trade-off between locality, load balance, and parallelism by coarsening level sets from wavefront parallelism.
- A novel proportional cost model included in LBC that creates well-balanced partitions for sparse kernels with irregular computations such as sparse Cholesky.
- Implementations of the new parallel inspection strategies and

code transformations for sparse triangular solve and Cholesky factorization, in a framework called ParSy. For evaluation, the proposed implementations are built within the open-source Sympiler infrastructure, but with all Sympiler optimizations disabled. The performance of ParSy is evaluated against MKL Pardiso and PaStiX, and shows that the partitioning strategy in ParSy outperforms the state-of-the-art by 1.4× on average and up to 3.1×.

II. ParSy Overview

ParSy consists of the H-Level inspector and code transformations to generate parallel code for sparse matrix methods. Example input code to ParSy is shown in Listing 1, where the user provides the numerical method, matrix sparsity pattern, and additional information about the desired level of parallelism. ParSy builds a DAG representing data dependencies in the sparse kernel for the given sparsity pattern. Then, the H-Level inspector uses a Load-Balanced Level Coarsening algorithm to create a schedule from the DAG of the kernel. To parallelize the original code and take advantage of the schedule, the numerical method code must be transformed. This section describes the H-Level inspector and discusses code transformations to support the parallel schedule, using sparse Cholesky factorization as an example.

Listing 1: ParSy input code

```
int main() {
 Sparse A(type(float,64),"Matrix.mtx");
 Cholesky chol(A);
 chol.generate_c("chol",k); }
```

Input : DAG G, k, thresh, win, agg
Output : H-LevelSet
1 [vertexCost,edgeCost] = computeCost(G)
2 [H-LevelSet]=LBC(G,vertexCost,edgeCost, k, thresh, win, agg)
3 return H-LevelSet

Algorithm 1: ParSy's H-Level inspector.

A. H-Level Inspector

The goal of ParSy's inspector is to statically partition the DAG of a specific numerical method applied to a specific sparse matrix while creating an efficient load balance with low synchronization cost and high locality. Wavefront parallelism approaches [31], [38], typically used in code transformation frameworks to generate parallel sparse codes, can create load imbalance and excessive synchronizations since sparse kernels like Cholesky have imbalanced workloads for column-based and column-block-based implementations. ParSy's H-Level inspector resolves this issue by creating partitions with coarser tasks while ensuring good balance between execution threads.

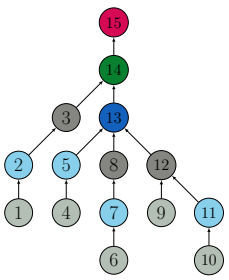
Algorithm 1 shows the basic outline of ParSy's inspector. Line 2 shows the LBC phase, where the DAG along with the number of processor cores (k in Algorithm 1), the computational efficiency of a single core (*thresh*), and two tuning parameters *win* and *agg* related to balancing and coarsening of the levels, are the inputs. The LBC algorithm uses a kernel-specific cost model for vertices and edges, which is used for load balance. With this information the DAG is partitioned into l-partitions that partition the DAG into coarsened levels, and into k or fewer w-partitions each executed on a single core within each l-partition (see Section III for more details).

Example. Cholesky factorization is commonly used in direct linear solvers and is used to precondition iterative solvers. The algorithm factors a Hermitian positive definite matrix A into LL^T, where matrix L is a sparse lower triangular matrix. We use the left-looking Cholesky variant. To compute the factors for a column j in L the algorithm visits all columns i that contain a nonzero in row j of L with $i < j$ and then applies the contributions of columns i to column j [11]. Dependencies between each column-computing iteration are represented by a DAG called the elimination tree (etree) [34], [42]. In an etree each node represents a column and each directed edge denotes that the destination depends on the source. To improve the performance of sparse Cholesky by using dense BLAS operations, columns with similar nonzero patterns are merged to form a block or supernode of columns. Dependencies between column blocks are represented using a modified version of the etree called the assembly tree, where nodes represent column blocks. For Cholesky factorization, using the etree does not create coarse enough nodes to parallelize and thus in most available software [15], [23], [48], [3] the assembly tree is used as the baseline dependency DAG for Cholesky. Figure 1a is an example assembly tree that we will use to demonstrate how ParSy creates an H-Level set.

Wavefront parallelism techniques [54], [45] first create a topologically-ordered level set, shown in Figure 1a and then execute nodes within each level in parallel. However, this often leads to higher-than-necessary overhead, because it requires synchronization between each level. Furthermore, the work per node varies depending on the non-zero structure, often resulting in poor load balance in each level. Our Load-Balanced Level Coarsening (LBC) algorithm, described in detail in Section III, partitions the assembly tree with the objective of facilitating efficient parallel execution while producing a good balance between load and locality. Our partitioning works in two stages; the first partitions the DAG by level to create topologically-ordered l-partitions. In the second phase, the disjoint sub-DAGs inside each level are divided into k or fewer equally-balanced w-partitions, where k is the number of cores. The H-Level set improves locality compared to the wavefront approach and also reduces inter-level synchronizations from six to two for this example. Furthermore, the LBC algorithm balances the workload of each partition by packing enough independent sub-DAG into each w-partition. This packing approach is important in sparse Cholesky factorization where the workload for each column block differs from other blocks. Finally, each w-partition does not communicate with any other w-partition in the same level, since each consists of disjoint sub-DAGs.

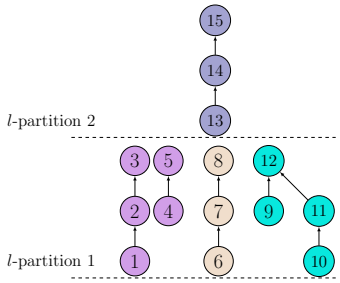
B. Parallel Code Transformation

To utilize the H-Level set to efficiently execute the schedule, the original code must be transformed for parallelism. Figure 2 shows how the H-Level transformation modifies Cholesky



Level set $= \{ \{1,4,6,9,10\}; \{2,5,7,11\}; \{3,8,12\}; \{13\}; \{14\}; \{15\}; \}$

(a) Assembly Tree $G(V, E)$



H-Level Set $= \{\{\{1,2,3,4,5\},\{6,7,8\},\{10,11,9,12\}\};\{\{13,14,15\}\};\}$

(b) H-Level set of G

Fig. 1: (a) An example DAG, in this case an assembly tree where nodes represent column blocks and edges show the dependencies between columns during factorization. Wavefront methods create a level set, represented by node coloring; nodes with the same color can be executed in parallel. (b) The H-Level set created by LBC from G in (a).

factorization[1]. As shown, the outermost loop in line 2 of Figure 2a is transformed to lines 1–5 in Figure 2b. Since in the left-looking Cholesky algorithm the nodes do not write to other nodes, the loop body does not change because no critical region is required. The OpenMP pragma enables parallelism over sub-DAGs, executing dependent nodes within the same thread, which increases locality. For the example DAG in Figure 1a, the outer loop in the code of Figure 2b executes only one iteration, resulting in a single synchronization, compared to the six synchronizations required by wavefront parallelism.

The available parallelism in a sparse algorithm is not uniform and typically different approaches for parallelism must be used to efficiently exploit the underlying parallel architecture. For example, l-partition 1 in the partitioned DAG in Figure 1b benefits from tree parallelism; however, the nodes in l-partition 2, which contains the sync node (the node with no outgoing edges), have no tree parallelism but such nodes can be repartitioned to increase data parallelism within their corresponding dense computations [23]. The last iteration, which corresponds to the last partition of the H-Level set, is peeled and optimized differently. For such nodes, ParSy enables using the parallel BLAS for each operation in the node; however, ParSy can be extended to support more advanced specialization techniques such as repartitioning.

C. Implementation

We have implemented ParSy in the open-source Sympiler [9] framework. Even though ParSy can be implemented at run-time similar to library-based approaches, we build on top of Sympiler to ease implementation and for potential future benefits of integrating ParSy with sparsity-specific code generation from Sympiler. Because of using Sympiler, the inspectors in ParSy are executed at compile time and their information is used to automatically transform the code. The following provides a short overview of the Sympiler framework and illustrates how ParSy is implemented using Sympiler.

Overview of Sympiler. Sympiler is a domain-specific compiler that generates specialized code for sparse matrix methods on single-core architectures. Given the numerical method and input matrix stored in compressed sparse column (CSC) format, Sympiler uses a symbolic inspector to generate inspection sets to guide code transformations. The numerical solver is internally represented using a domain-specific abstract syntax tree (AST) and is annotated with potential transformations. The lowered code is also annotated with hints for low-level transformations which are used in the transformation phase for sparsity-specific code specialization. In the transformation phase, the inspection sets are used to lower the annotated code to apply inspector-guided and low-level transformations and output transformed C source code.

To implement ParSy, the inputs to Sympiler are extended to provide information that the H-Level inspector requires. The H-Level inspector and H-Level transformation are implemented as additional stages in the inspection and transformation phases of Sympiler respectively. The inspector creates the data dependence graph based the input numerical method and the sparsity pattern. ParSy uses the created data dependence graph and creates a coarsened level set that later will be used as an input to the generated code. Sympiler's low-level transformations are disabled in the current version of ParSy, so we do not specialize code for a specific sparsity pattern; we intend to explore this feature in future releases of ParSy. This paper considers solely the impact of the H-Level inspector.

III. Load-Balanced Level Coarsening (LBC)

ParSy utilizes the Load-Balanced Level Coarsening (LBC) algorithm to partition the DAG that describes the dependencies of the computation. LBC statically creates a set of partitions that minimize load imbalance and communication while attempting to maximize available parallelism and locality. In this section,

[1]For space, we provide the general form of the transformation in Appendix A.

```
1  H-Level:
2  for (int i=0; i<blockNo; ++i){
3    b1 = block2col[i]; b2 = block2col[i+1];
4    f = A(:,b1:b2);
5    // Update phase
6    for(block r=0 to i-1 L(i,r)!=0){
7    f-=GEMM(L(b1:n,r),transpose(L(i,r)));}
8    // Diagonal operation
9    L(b1:b2,b1:b2)=POTRF(f(b1:b2));
10   // Off-diagonal operations
11   for(off-diagonal elements in f){
12   L(b2+1:n,b1:b1) =
13   TRSM(f(b1+1:n,b1:b2),L(b1:b2,b1:b2)); }  }
```

(a) Serial blocked left-looking Cholesky

```
1  for(every l-partition i){
2  #pragma omp parallel for private(f){
3   for(every w-partition j){
4    for(every v ∈ HLevelSet[i][j]){
5      int i = v;
6      b1 = block2col[i];b2 = block2col[i+1];
7      f = A(:,b1:b2);
8      for(block r=0 to i-1 L(i,r)!=0){
9      f-=GEMM(L(b1:n,r),transpose(L(i,r)));}
10     L(b1:b2,b1:b2)=POTRF(f(b1:b2));
11     for(off-diagonal elements in f){
12     L(b2+1:n,b1:b1) =
13     TRSM(f(b1+1:n,b1:b2),L(b1:b2,b1:b2)); } }}}}
14 //Specilized code for the last l-partition.
15 Cholesky_Specialized(HLevelSet[n − 1][0]);
```

(b) ParSy's generated code

Fig. 2: The application of the H-Level transformation on blocked left-looking Cholesky factorization. (b) shows the transformed version of the code in (a) with the H-Level transformation. The gray lines remain unchanged.

we describe the partitioning produced by LBC, its associated constraints, and the algorithm that produces this partitioning. Finally, we show the proportional cost model used by LBC to estimate load costs for each partition.

A. Problem Definition

The goal of Load-Balanced Level Coarsening is to find a set of l-partitions, and within each l-partition, to find a set of disjoint w-partitions with as balanced cost as possible. For improved performance, these partitions adhere to additional constraints to reduce synchronization between threads and maintain load balance. Additionally, there are objective functions for minimizing communication between threads and the number of synchronizations between levels. To describe the partitioning and constraints, we use the follwoing notation.

Definitions: $G(V, E)$ denotes the input DAG with vertex set V and edge set E, along with a nonnegative integer weight $d(v)$ for each vertex $v \in V$ and nonnegative integer weight $c(e)$ for each edge $e \in E$. The *level* of a node $level(v)$ is the length of the longest path between the node v and a *source node*, which is a node with no incoming edge. The level of the sync node is *critical path* P; in the case of multiple sync nodes, P is the maximal level among all sync nodes.

Definition 1: Given DAG G and an integer number of partitions $n > 1$, the LBC algorithm produces n l-partitions of V with sets of nodes $(V_{l_1}, ..., V_{l_n})$ such that $V_{l_1} \cup ... \cup V_{l_n} = V$ and $V_{l_i} \cap V_{l_j} = \emptyset$. Each l-partition $l_i = [lb_i..ub_i]$ is represented by a lower bound and upper bound on the level, and contains all nodes with levels between the two bounds. In addition, $\cup_{i=1}^{n} l_i = [1..P]$.

Definition 2: Given the number of threads $k > 1$, for each set of nodes V_{l_i}, the LBC algorithm produces $m_i \leq k$ w-partitions $(V_{l_i,w_1}, ..., V_{l_i,w_m})$ such that $V_{l_i,w_1} \cup ... \cup V_{l_i,w_{m_i}} = V_{l_i}$ and $\forall i, j, p, q$, where $i \neq j$ and $p \neq q$, $V_{l_i,w_p} \cap V_{l_j,w_q} = \emptyset$.

Definition 3: Within a partition, the number of *connected components* is the number of disjoint sub-DAGs in the partition, which is shown by $comp(V_{l_i,w_p})$ for a partition V_{l_i,w_p}.

In summary, the partitioning produced by LBC creates l-partitions, and within each l-partition i, it creates up to k disjoint w-partitions. Each node in the DAG belongs to one l-partition and one w-partition. Note that some l-partitions, those with connected vertices, will only contain one w-partition (see V_{l_2} in Figure 1). Some of the values for that example are as follows: $n = 2$, $V_{l_1} = \{\{1, 2, 3, 4, 5\}, \{6, 7, 8\}, \{10, 11, 9, 12\}\}$, $V_{l_1,w_2} = \{6, 7, 8\}$, and $V_{l_2} = \{\{13, 14, 15\}\}$. The number of w-partitions for V_{l_1} is $m_1 = 3$, and $m_2 = 1$ for V_{l_2}. The number of connected components in l-partition V_{l_i} is shown with $comp(V_{l_i})$. For example, $comp(V_{l_1,w_1})$ is 2, $comp(V_{l_1,w_2})$ is 1, etc.

Constraints: The *space-partition constraint* ensures that threads executing iterations in different w-partitions need not synchronize amongst each other. The name of this constraint comes from affine partitioning [33], where the goal of the constraint is the same; however, the constraint definition is different here since the input is a DAG. If $E(V_{l_i,w_p}, V_{l_i,w_q})$ is the set of cut edges between two partitions V_{l_i,w_p} and V_{l_i,w_q}, the space-partition constraint is:

$$\forall 1 \leq i \leq n \wedge (1 \leq p, q \leq m_i), \quad E(V_{l_i,w_p}, V_{l_i,w_q}) = \emptyset \quad (1)$$

The w-partitions within each V_{l_j} must have no edges in common, which is the constraint expressed in Equation (1).

The *load balance constraint* ensures that the w-partitions within V_{l_i} are balanced up to a threshold. Assuming $\epsilon \in \mathbb{R}$ with $\epsilon \geq 0$ is a given input threshold for determining the maximum imbalance, the load balance constraint is:

idea: each w-partition's load cannot exceed the average load by (1+epsilon)

$$\forall i, 1 \leq i \leq n \wedge comp(V_{l_i}) > 1 \wedge \forall 1 \leq p \in m_i,$$
$$cost(V_{l_i,w_p}) \leq (1 + \epsilon)\lceil cost(V_{l_i})/m_i \rceil \quad (2)$$

where $cost(V_{l_i,w_p}) = \sum_{v \in V_{l_i,w_p}} d(v)$ and $cost(V_{l_i}) = \sum_{p \in 1..m_i} cost(V_{l_i,w_p})$. As shown in Equation 2, the load balance constraint does not apply to an l-partition with only a single w-partition, because creating load balance for one

component is not feasible. The constraint ensures that the cost of executing an l-partition V_{l_i} is uniformly distributed to w-partitions V_{l_i,w_p} such that the maximum difference is less than 2ϵ.

Objective: The objective function for LBC is to reduce the critical path of the partitioned DAG, also known as quotient graph Q_G, as well as the communication cost between the partitions. Q_G is the DAG induced by the partitioning V_{h_j,w_i}, where each vertex in Q_G is a partition and edges E_q exist only if an edge exists such that the two endpoints are in separate partitions. The critical path minimization objective is to minimize P_{Q_G}. The communication cost objective is to minimize $\sum_{e \in E_q} c(e)$, where c is the cost associated with each edge of Q_G. Since no edges exist between w-partitions, this objective minimizes the edge costs between l-partitions.

B. LBC Algorithm

As shown in Algorithm 1, the inputs to LBC are a DAG annotated with a cost model for both vertices and edges, the number of requested w-partitions, an architecture-related threshold, and two tuning parameters *win* and *agg*. An example cost model used in LBC is illustrated in Section III-C. Since optimizing for both l-partitions and w-partitions is complex, our algorithm uses heuristics for speed and simplicity. A major simplification is to separate the two kinds of partitioning so that the algorithm, shown in Algorithm 2, proceeds in three stages: (1) l-partitioning, (2) w-partitioning and, optionally, (3) reordering.

l-partitioning: This step finds l as defined in Section III-A. The algorithm begins by finding the first partition, which contains the source nodes of the DAG; note that the upper and lower bounds for each partition represent the range of levels (the distance from the source nodes) for the vertices in the partition. In line 7, the algorithm finds the largest level (closest to the sync node of the DAG) containing enough disjoint sub-DAGs to result in approximately k w-partitions. Then, in lines 9–16 the algorithm searches through adjacent candidates up to win levels away for where to cut the partition, by finding the one that results in the most load-balanced w-partitions (see Section III-C). Once the first l-partition is set, the loop in line 17 groups the remaining levels into l-partitions with agg levels per partition. Tuning parameters win and agg show the search window for a load-balanced cut and coarseness of the remaining levels respectively. Finally, the algorithm builds the last partition, containing the sync node, in line 20.

w-partitioning: In this step, each l-partition, which is a collection of sub-DAGs with different costs, is divided into w-partitions such that the cost of each partition is balanced. To find the sub-DAGs, we do a sequence of depth-first searches from all source nodes in the l-partition. The sub-DAGs that intersect are merged. Then in our algorithm, we use a variant of the first-fit decreasing bin packing [25], [10] approach to find w-partitions with near-equal overall cost. Lines 21–26 in Algorithm 2 produce w-partitions of size k if there are enough components; otherwise, the number of bins is set to $comp(G_g)/2$. Once the balanced components are found, we use a modified breadth-first search (BFS) to store the nodes of w-partitions in a precedence order. The modified BFS algorithm starts from the source nodes of a w-partition and places the nodes in a queue. Every node that is removed from the queue is placed in the final H-Level set and then the incoming degree of its adjacent nodes is decremented. The algorithm ends when the queue is empty.

Because our w-partitioning algorithm merges sub-DAGs that intersect, it is possible that fewer than k components are found due to the intersection. However, we have not encountered this case in practice, and in such cases it is possible to modify the algorithm to perform w-partitioning for multiple candidate l-partitionings to find one where the most subcomponents exist.

Reordering: Optionally, the w-partitions in each l-partition can be reordered to further enhance locality. This phase reorders the computation within each w-partition to optimize the communication cost objective. The goal is to ensure that a w-partition V_{l_j,w_i} in l-partition j that synchronizes with w-partition V_{l_{j+1},w_k} can be moved so that both w-partitions are assigned to the same thread; as a result, the data will remain local to the thread. In lines 27–32 of Algorithm 2, the LBC algorithm checks adjacent l-partitions and ensures that w-partitions with the highest communication cost are aligned vertically. During execution, w-partitions with the same ID will be assigned to the same processor, ensuring that inter-thread communication between l-partitions is minimal.

C. Cost Model & Windowing Heuristic

Statically scheduling the DAG for parallelism requires estimating the cost of each node in the DAG accurately, to ensure a high degree of parallelism and good load balance. The LBC algorithm implements two heuristics for two different parts of the algorithm that make this possible to do efficiently: a simple cost model that does not require machine-specific empirical performance measurements, and a heuristic for searching only among a small number of possible partitionings.

Existing approaches for static scheduling of sparse factorization algorithms such as that used in PaStiX [23] rely on accurate cost estimates for each BLAS operation to find load balanced partitions; PaStiX uses empirically-measured runtimes for each BLAS kernel. In contrast, the H-Level inspector uses a simple *proportional cost model* to find an efficient partitioning of the DAG. Motivated by the fact that sparse matrix computations are generally memory bandwidth-bound, this model uses the number of *participating nonzeros* in each node of the DAG as a proxy for the cost of execution.

Definition: The participating nonzeros for a node N_i in the DAG is the total number of nonzeros touched in order to complete the computation of N_i. For example, for Cholesky factorization, the participating nonzeros for a node are the nonzeros in the column block represented by N_i, plus the nonzeros touched when eliminating the block. This can be computed exactly during symbolic factorization or can be approximated with the sum of the column counts for every column such that the rows corresponding to N_i have a nonzero, which can be derived in near-linear time in the size of the

Algorithm : Load-Balanced Level Coarsening
Input : G, d, c k, $thresh$, win, agg
Output : V_{l_j,w_i} , l_j

```
/* For small DAG, use a single partition                    */
if G ≤ thresh then
    V_{l_0} = G
    l_0.lb = 0 l_0.ub = G.P
    return {V,l}
end
/* l-partitioning, starting from source nodes               */
l_0.lb = 0
/* Find closest level to the sync node with
   enough sub-DAG                                           */
l_initCut = max({l|comp(G_{0:l}) ≥ k})
ϵ=∞
/* Explore cuts to find good load balance                   */
for i=l_initCut; i> l_initCut-win; i-=1 do
    CurCost(:) = BinPack(G_{0:i},d,k)
    maximalDiff = max(CurCost) - min(CurCost)
    if maximalDiff < ϵ then
        ϵ = maximalDiff
        h_0.ub = i
    end
end
/* Group rest of levels into l-partitions                   */
for i=l_0.ub; i > G.P − agg; i+=agg do
    l.append([i, i + agg])
end
/* Final partition includes the sync node                   */
l.append([l_n.ub,G.P])
/* w-partitioning                                           */
for g ∈ l do
    if comp(G_g) > 1 then
        parts = comp(G_g) > k ? k : comp(G_g)/2
        V_g= BinPack(G_g,d,parts)
    end
end
/* Reorder w-partitions                                     */
for i=n; i> 0; i-=1 do
    for j=0; j< m_i; j+=1 do
        Q = {∃q ∈ child(V_{l_i,w_j})|c(e_{qV_{l_i,w_j}}) is max }
        swap(V_{l_{i+1},w_Q},V_{l_{i+1},w_j})
    end
end
return {V,l}
```

Algorithm 2: The LBC DAG partitioning algorithm. $G_{a:b}$ is the DAG induced by including only vertices v where $a \leq level(v) \leq b$ and the incident edges. The lower and upper bounds for each l-partition are values for the node levels.

matrix [11]. We use a similar metric for computing edge cost, which is the number of nonzeros that must be communicated.

The proportional cost model need not be as exact as the kinds of cost models used in PaStiX, due to the much coarser granularity of scheduling in ParSy. However, any model used for static scheduling, even for coarse-grained tasks, must be accurate enough to use as a proxy for performance. This simple cost is sufficient to capture the real behavior of our static partitioning scheme. Figure 3 shows the actual maximal difference in time versus the estimated maximal difference in cost for an example matrix based on participating nonzeros for l-partitions constructed at different levels, with the left

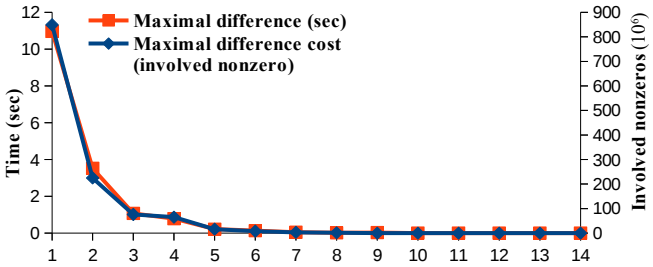


Fig. 3: The maximal difference in time matches the maximal difference in cost, i.e. participating nonzeros. This is shown for matrix *Flan_1565* as an example, but all matrices used in our experiments exhibit similar behavior.

side being cuts closest to the the sync node. The cost closely matches the observed difference in time measured using cycle counters. Unlike other static partitioning schemes, the cost model used by ParSy is simple and requires no empirical measurement, while effectively estimating performance for candidate partitions using the H-Level inspector.

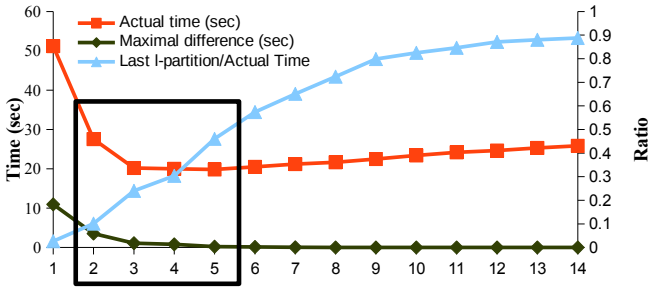
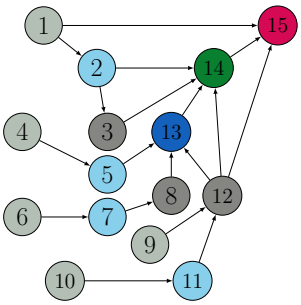


Fig. 4: The effect of l-partitioning on the performance and load balancing of Cholesky for matrix *Flan_1565* starting from the sync node, (shown with 1) to close to the source nodes (shown with 14). The dark rectangle shows the search window from the initial point which is point 2. The dark green line shows the maximal difference. The blue line shows the percentage of actual time spent on the closest-to-sync l-partition, which uses node-level parallelism. The red line shows the actual total runtime using each edge cut.

Given this cost metric, the second heuristic tries to find the partitioning with minimal load imbalance without searching through a large number of candidates. This *windowed search* heuristic examines a small number of candidates in the neighborhood of the first l-partitioning containing enough sub-DAGs for parallel execution. For the implementation in this paper, the window size (that is, the number of additional candidates to search over) is 3. Figure 4 shows the effect of the local search. The first l-partition with enough sub-DAGs is at point 2, but the windowing heuristic chooses a cut at point 5, which has the best load balance among candidates. As illustrated by the blue line in Figure 4, choosing cuts closer to the source nodes results in less work that can be done in parallel, since the l-partitions closer to the sync node cannot usually be divided into enough w-partitions to achieve the best



(a) DAG of dependencies

```
x=b; // copy RHS to x
HLevel:
for ( int i=0; i<blockNo; ++i){
 b1 = block2col[i];
 b2 = block2col[i+1];
 //Diagonal
 x(b1:b2)=TRSM(L(b1:b2,b1:b2),x(b1:b2));
 // Off-diagonal
 tempX=GEMV(L(b2:blockNo,b1:b2),x(b1:b2));
 for(row index j in column i,k=0){
Atomic:
   x(Li(j)) -= tempX(k++); }
 }
```

(b) Serial blocked code

```
x=b;
for(every l-partition i){
#pragma omp parallel for private(tempX){
 for(every w-partition j){
  for(every v ∈ HLevelSet[i][j]){
   i = v;
   b1 = block2col[i];
   b2 = block2col[i+1];
   x(b1:b2)=TRSM(L(b1:b2,b1:b2),x(b1:b2));
   tempX=GEMV(L(b2:blockNo,b1:b2),x(b1:b2));
   for(row index j in column i,k=0){
#pragma omp atomic
     x(Li(j)) -= tempX(k++) ;}
   }}}}
```

(c) Transformed with H-level

Fig. 5: H-Level transformation for sparse triangular solve. (a) An example DAG representing the dependencies for sparse triangular solve. (b) A blocked forward substitution algorithm with compressed column format that is annotated with `HLevel` and `Atomic`. (c) shows the code after H-Level transformation. Gray lines in the code are not affected by the transformation.

parallel performance.

IV. Other Sparse Matrix Methods

The data dependence graphs and H-level inspection strategy in ParSy can be used for a large class of sparse matrix computations. For example, for kernels such as LU, QR, and orthogonal factorization methods [34], which introduce fill-in during computation, the input DAG to ParSy is the assembly tree that captures the dependencies in the computation, including dependencies that come from fill-ins. For kernels with no fill-in such as ILU(0), IChol(0), and triangular solve, the input is the matrix DAG. This section describes how ParSy works for sparse triangular solve, where data dependence is represented with a DAG and the computations are more regular than Cholesky.

Triangular Solve. This kernel solves the linear equation $Lx = b$ for x where L is a lower triangular matrix and b is the right-hand side (RHS) vector. Figure 5 shows two different implementations of sparse lower triangular solve for a matrix in column storage format and dense RHS. A serial implementation of the algorithm is shown in Figure 5b. Figure 5a shows the DAG of dependencies for the column-blocked version of matrix L. ParSy's H-Level inspector uses the DAG of L and builds the H-Level set which is an input for the code in Figure 5c. The H-Level set corresponding to the DAG shown in Figure 5a is shown in Figure 1b. Since the iterations in the sparse triangular solve kernel are more regular compared to the Cholesky algorithm [5] the benefits of creating an H-Level set using ParSy are mainly in reducing synchronizations in the code and increasing locality from level coarsening.

V. Experimental Results

We compare the performance of ParSy-generated code with PaStiX [23] and with MKL Pardiso [48], which are both specialized libraries for matrix factorization. PaStiX uses the same left-looking supernodal algorithm as ParSy and also uses a static scheduling heuristic. MKL Pardiso uses the left-right looking supernodal variant of Cholesky and uses hybrid static/dynamic scheduling. MKL also provides optimized implementations for sparse triangular solve in compressed row, compressed column, and blocked compressed row formats. Thus, Cholesky factorization results are compared with both PaStiX and MKL Pardiso while results for triangular solve are compared to MKL's best performing implementation amongst the three data structures. For triangular solve, we use the factorized lower-triangular matrix L that is the result of running Cholesky on each test matrix. Appendix B has additional triangular solve experiments on matrices with non-chordal DAGs. We also parallelize each sparse kernel with the level set used in wavefront techniques [54] and call this implementation *level set*. The performance of the level set implementation is used as a baseline.

For the comparison, we use the set of symmetric positive definite matrices listed in Table I. The matrices are from [13] and belong to different domains with real number values in double precision. The testbed architectures are listed in Table II.

TABLE I: Test matrices, sorted in order of decreasing parallelism. *nnz* is the number of nonzeros in the factor L.

ID	Name	Rank (10^3)	nnz (10^6)	Parallelism (METIS)	Parallelism (SCOTCH)
1	G3_circuit	1585	127.3	16284	12154
2	ecology2	1000	54.3	11444	7454
3	thermal2	1228	71.9	10618	7087
4	apache2	715.2	164.7	10216	4427
5	StocF_1465	1465.1	1245	7755	6003
6	Hook_1498	1498	1783.8	7651	6032
7	tmt_sym	726.8	41.9	6371	4233
8	PFlow_742	742.8	598	5390	4796
9	af_shell10	1508	394.3	4900	3752
10	parabolic_fem	525.9	35	4712	3488
11	Flan_1565	1564.8	1715.9	3725	3271
12	audikw_1	943.7	1473.1	2438	2203
13	bone010	986.8	1210.1	2332	2020
14	thermomech_dM	204.3	9.7	2310	1480
15	Emilia_923	923.1	1992	2277	1927
16	Fault_639	638.8	1275.4	1595	1493
17	bmwcra_1	148.8	79.4	497	402
18	nd24k	72	435.9	48	48
19	nd12k	36	161.9	29	28

TABLE II: Testbed architectures.

Family	Haswell-E	Haswell-EP	Skylake
Processor	Core™i7-5820K	Xeon™E5-2680v3	Xeon™Platinum 8160
Cores	6 @ 3.30 GHz	12 @ 2.5 GHz	24 @ 2.1 GHz
L3 cache	15MB	30MB	33MB

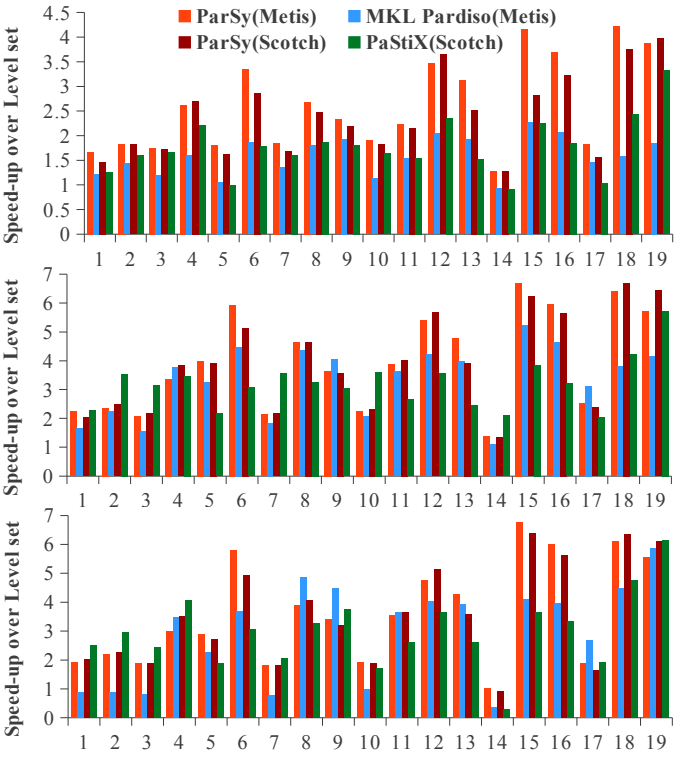


Fig. 6: ParSy's (numeric) performance for Cholesky compared to MKL Pardiso (numeric) and Pastix (numeric) on Haswell-E (top), Haswell-EP (middle), and Skylake (bottom). All times are normalized over the level set numeric time.

All ParSy-generated code is compiled with GCC v.5.4.0 using the `-O3` option. We report the median of 5 executions for each experiment. The PaStiX and MKL Pardiso libraries are installed and executed using the recommended default configuration. For Cholesky, the default ordering method for PaStiX is Scotch [40] and for MKL Pardiso is Metis [28]. We use Metis ordering in ParSy for comparison to MKL Pardiso, and use Scotch ordering when comparing to PaStiX; this removes the effect of ordering and allows for a fair comparison. For triangular solve, we do not show the effect of reodering since reordering would possibly change the pattern of the matrix to something other than a triangular pattern. Unless otherwise stated, we include only numeric factorization time and do not include time for symbolic factorization.

Cholesky Performance. Figure 6 shows the performance of ParSy-generated code compared to MKL Pardiso, PaStiX, and the level set implementation. The ParSy-generated code is faster than MKL Pardiso by up to 2.7×, 1.7×, and 2.8× and is faster than PaStiX by up to 1.7×, 1.8×, and 3.1× on Haswell-E, Haswell-EP, and Skylake respectively.

One of the main objectives of ParSy's inspector is to improve locality in sparse codes. Figure 7 shows the relationship between the performance of ParSy and MKL Pardiso to their memory accesses on the Haswell-E[2]. The average memory access latency [22] is a measure for locality and is obtained by gathering the TLB, L1 cache, and last level cache (LLC) accesses and misses using the *perf* profiler. The Haswell-E specification parameters are obtained from [22]. Figure 7 demonstrates a correlation between the performance of the ParSy-generated code and the average memory access cost. The coefficient of determination or R^2 is 0.65, showing good correlation between speed-up and memory access latency. For matrices where ParSy provides better speedups, locality has been improved more. Data in Figure 7 shows the original measurements for the 5 runs and not the medians.

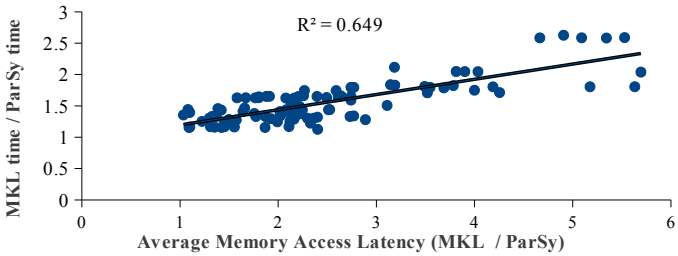


Fig. 7: Relation between speed up and locality on Haswell-E. Average memory access latency is the average cost of accessing memory in ParSy's code and MKL Pardiso. The relation between speed-up and the memory access ratio is approximated with a line. The coefficient of determination or R^2 of the fitted line is 0.65.

Figure 8 compares the ratio of wait time to CPU time in ParSy and MKL Pardiso on Hasewell-E, measured using Intel's VTune Amplifier. Wait time[3] is the time that a software thread is stalled due to APIs that block or cause synchronization. CPU time[4] is the time that the CPU takes to execute numerical factorization. Because it uses dynamic scheduling, MKL Pardiso is more load balanced and thus has a nearly zero wait time for all matrices, averaging 99% CPU utilization. ParSy, however, prioritizes locality over load balance. ParSy improves locality as shown in Figure 7 and also utilizes the CPU cores fairly efficiently with an average of 95% CPU utilization (a ratio of 0.05) as shown in Figure 8. Compared to MKL Pardiso, ParSy provides a better trade-off between locality and load balance which leads to the better performance results for ParSy shown in Figure 6.

To analyze the performance of ParSy we provide the average parallelism metric, shown with *Parallelism* in Table I, which is related to the sparsity of the matrix. Parallelism is obtained by dividing the number of nodes in the DAG by the critical path of the DAG and is an approximate indicator of available parallelism. The analysis based on parallelism is provided for

[2]We did not have root permission to profile on other architectures.
[3]https://software.intel.com/en-us/vtune-amplifier-help-wait-time
[4]https://software.intel.com/en-us/vtune-amplifier-help-cpu-time

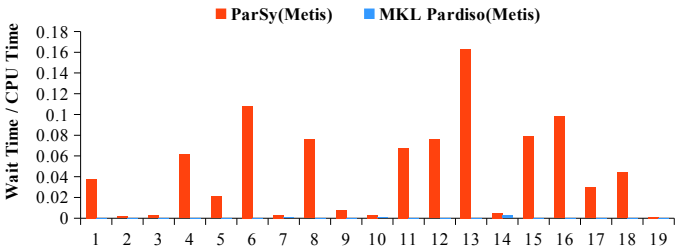


Fig. 8: The ratio of wait time to the total execution time of numerical factorization for Cholesky in ParSy and MKL Pardiso on Haswell-E.

both Metis and Scotch ordering methods. The performance of ParSy is shown with two different orderings. Figure 6 shows how the ParSy-generated code improves the performance of matrices with different sparsity patterns on the testbed processors. The Skylake processor has a larger number of cores compared to the other architectures; thus, we expect matrices with more parallelism to perform better with ParSy on this architecture; matrices 1, 2, and 3 which achieve high speed-ups in ParSy compared to MKL Pardiso have the most parallelism while matrices 17 and 19 with the least parallelism do not perform as well as the other matrices.

A fill-in reducing ordering method such as Metis or Scotch determines the number of nonzeros in the factor L and also affects the structure of the assembly tree. For fair comparison with the libraries and also to show the effect of ordering on ParSy, the performance of ParSy with Metis and Scotch ordering is shown in Figure 6. As shown, ParSy is faster than the library using the same ordering; also, ParSy performs well with both orderings. Library approaches are optimized for a speific ordering and do not perform well when the ordering is different from their default. For example, PaStiX with Metis ordering is on average 2.2× slower than PaStiX with Scotch ordering and MKL Pardiso with Scotch is on average 7.9× slower than MKL Pardiso with Metis.

Triangular Solve Performance. Figure 9 compares the performance of triangular solve in ParSy to MKL and wavefront parallelism. The average speed-up of ParSy-generated code compared to the level set implementation is 1.2×, 1.3×, 1.0× on Haswell-E, Haswell-EP, and Skylake respectively. The speed-up for triangular solve is relatively smaller than speed-ups for Cholesky. This may be due to two reasons: (1) the triangular solve is more regular, and thus the level set implementation does not create much load imbalance; (2) the kernel has less data reuse compared to Cholesky which reduces the effects of optimizing for locality. However, ParSy is faster than the highly-tuned MKL library on average by 2.6×, 4.7×, and 2.8× on Haswell-E, Haswell-EP, and Skylake respectively, showing the efficiency of LBC versus widely-used libraries.

Inspection Overhead. The H-Level inspection is performed at compile time in ParSy and the generated code only manipulates numerical values. The accumulated time for ParSy includes compile-time inspection, code generation time, and numeric factorization time. As demonstrated in Figure 10 ,

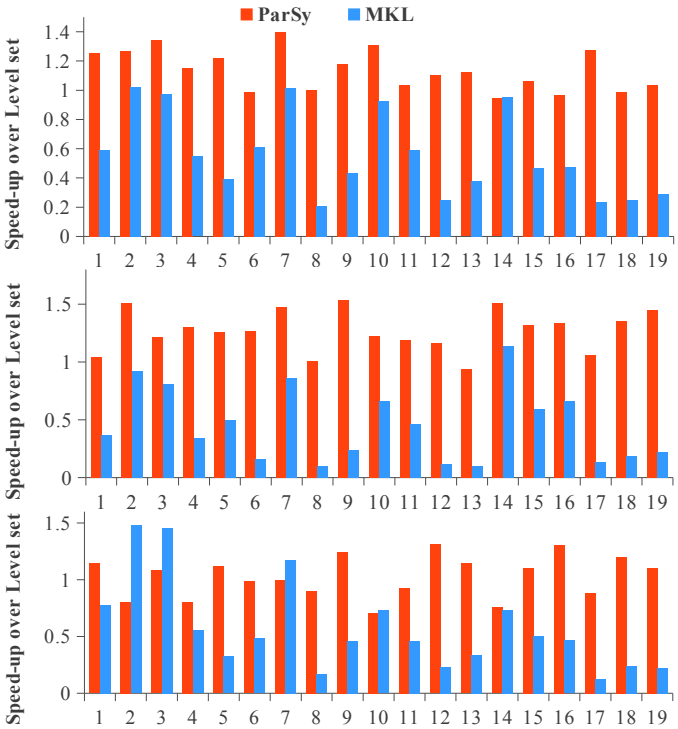


Fig. 9: The performance of ParSy (numeric) for triangular solve compared to MKL (numeric) on Haswell-E (top), Haswell-EP (middle), and Skylake (bottom) processors. All times are normalized over the level set numeric time.

the accumulated time of ParSy is 1.3× and 1.0× faster than MKL Pardiso and PaStiX respectively, on average across all architectures. Figure 11 shows the accumulated time of ParSy-generated code for triangular solve is in average 4.0× and 3.4× faster than the MKL accumulated time on Haswell-E and Skylake respectively. The accumulated times for Haswell-EP follows a similar pattern to Haswell-E.

Scalability Analysis. The average speed-up for ParSy is 4×, 6.6×, and 6.8× compared to ParSy serial code on Haswell-E, Haswell-EP, and Skylake respectively. For MKL Pardiso and PaStiX the average speed-ups compared to their own serial codes are 3.9×, 7.8×, and 8.4× for MKL Pardiso and 4.3×, 7.4×, 7.5× for PaStiX for Haswell-E, Haswell-EP, and Skylake. These numbers demonstrate good scaling in all three implementations. However, the performance of ParSy is 1.4× faster than the two libraries across all architectures.

VI. Related Work

Wavefront parallelism [54], [45], [59], [51], [38], [20] is one of the most common approaches inspector-executor frameworks use to parallelize sparse matrix methods. These either employ manually-written inspectors and executors [51], [38], [20], [39], [35] or automate parts of the process by simplifying the inspector [54], [45], [59], [19]. These approaches use inspectors to obtain dependence information that is only known at runtime. The H-Level sets created in ParSy are typically coarser than level sets in wavefront parallelism, reducing the

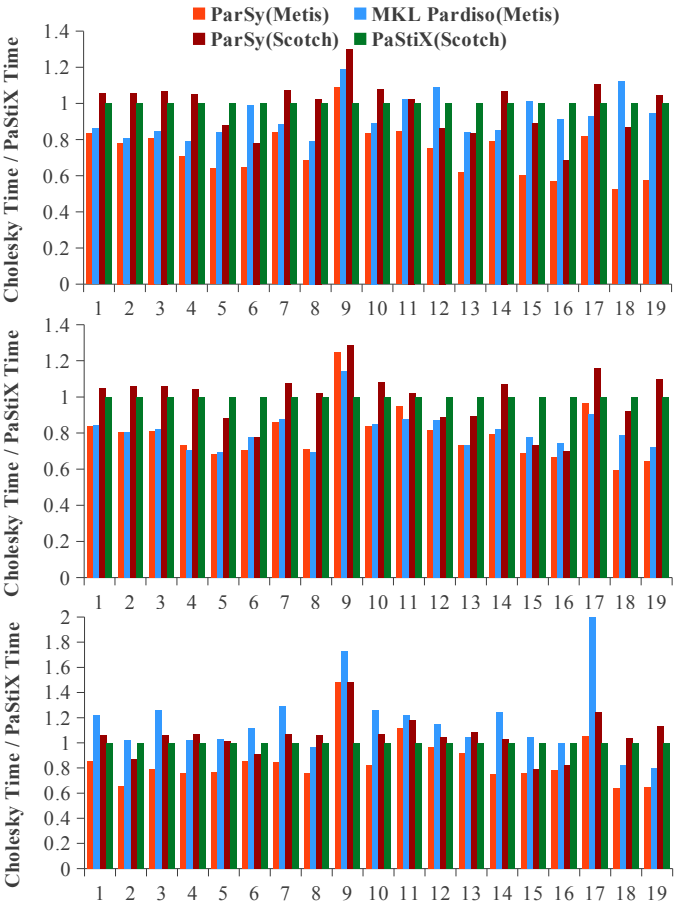


Fig. 10: Symbolic + numeric time for ParSy-generated code, MKL Pardiso, and PaStiX for Cholesky on Haswell-E (top), Haswell-EP (middle), and Skylake (bottom). All times are normalized to PaStiX's accumulated symbolic + numeric time.

number of costly synchronizations. ParSy also improves load balance in irregular sparse codes such as Cholesky compared to wavefront approaches. The closest approach to ours that finds an efficient trade-off between locality and load balance can be found in [5], which extends the Pluto framework [7] with an automatic parallelization approach for transforming input affine sequential codes. However, this is limited to structured and dense kernels.

Domain-specific compilers use domain information to dictate optimizations and transformations the compiler can apply. These compilers cover numerous applications such as stencil computations [44], [52], [24], signal processing [43], tensor algebra [29], matrix assembly in scientific simulations [2], [36], [30], [6], and dense [21], [50] and sparse [12], [46], [9] linear algebra. Amongst the domain-specific compilers for sparse methods Sympiler [9] benefits from specializing the generated code for a specific sparsity structure and numerical method. However, Sympiler does not support parallelism on multi-core. ParSy's goal is to integrate with the Sympiler framework to generate parallel code for sparse matrix methods on multiple processor cores while benefiting from the performance that

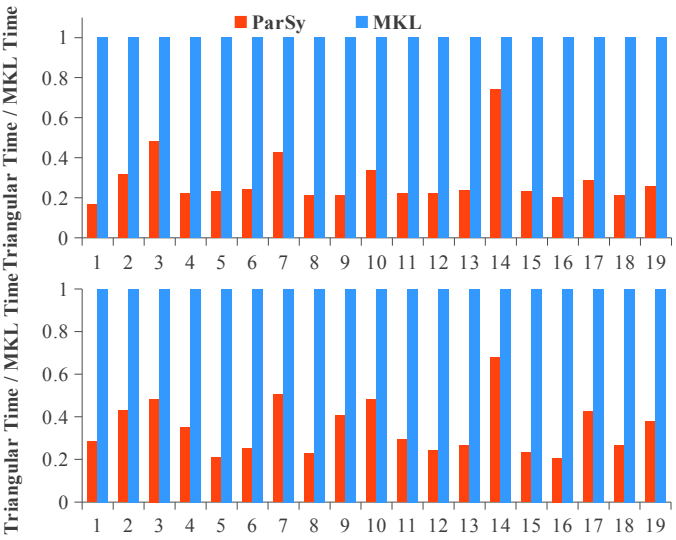


Fig. 11: The symbolic + numeric time for ParSy-generated code and MKL for triangular solve on Haswell-E (top), and Skylake (bottom) processors. All times are normalized to MKL's accumulated symbolic + numeric time.

Sympiler provides with sparsity-specific code specialization.

Numerous hand-optimized parallel sparse libraries exist with efficient sparse matrix kernels. These libraries differ in numerical methods they optimize and the platforms supported. Implementations in [11], [8], [14] provide sequential sparse kernels such as LU and Cholesky while parallel implementations exist in work such as SuperLU [16], MKL Pardiso [48], and PaStiX [23] for shared memory architectures, and in [16], [4] for distributed memory. Several libraries have also optimized specific sparse kernels such as triangular solve [31], [38], [57], [55], [53] and sparse matrix-vector multiply [58], [26], [37]. Sparse kernel variants differ between libraries; for example, PaStiX implements left-looking sparse Cholesky while MKL Pardiso uses a left-right looking approach [47]. ParSy optimizes left-looking Cholesky on shared memory architectures.

Parallel sparse libraries use numerical method-specific code to determine data dependencies and schedule the computation. These libraries typically inspect the symbolic information of the matrix, which is called static/symbolic analysis, and use the information for numerical manipulation with the objective of creating load-balanced tasks that can execute in parallel. Libraries such as PaStiX [23] use static analysis and static scheduling [1] while most other libraries use hybrid static/dynamic [49], [47] scheduling. Typically the DAG is partitioned during inspection with algorithms such as the subtree-to-subcube heuristic [18], [41], [27]. While dynamic scheduling can introduce overheads at runtime, static schedulers using profiling data on a specific architecture limit portability. ParSy uses the matrix structure and numerical method to compute a proportional cost that does not rely on the underlying architecture and enables compile-time scheduling of tasks.

VII. Conclusion

In this paper we demonstrate how Load-Balanced Level Coarsening can improve locality and reduce synchronization in sparse kernels, especially those with non-uniform workloads such as Cholesky. ParSy takes the numerical algorithm and sparsity pattern of the matrix and generates optimized parallel multi-core code. ParSy's inspector uses the LBC algorithm for inspection along with H-Level transformation for generating the code. ParSy-generated code outperforms two state-of-the-art sparse libraries for sparse Cholesky and triangular solve across different multi-core processors.

References

[1] Emmanuel Agullo, Olivier Beaumont, Lionel Eyraud-Dubois, and Suraj Kumar. Are static schedules so bad? a case study on cholesky factorization. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 1021–1030. IEEE, 2016.

[2] Martin S Alnæs, Anders Logg, Kristian B Ølgaard, Marie E Rognes, and Garth N Wells. Unified form language: A domain-specific language for weak formulations of partial differential equations. *ACM Transactions on Mathematical Software (TOMS)*, 40(2):9, 2014.

[3] Patrick R Amestoy, Iain S Duff, and J-Y L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Computer methods in applied mechanics and engineering*, 184(2):501–520, 2000.

[4] Patrick R Amestoy, Iain S Duff, Jean-Yves L'Excellent, and Jacko Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[5] Muthu Manikandan Baskaran, Nagavijayalakshmi Vydyanathan, Uday Kumar Reddy Bondhugula, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors. *PPOPP*, 44(4):219–228, 2009.

[6] Gilbert Louis Bernstein, Chinmayee Shah, Crystal Lemire, Zachary Devito, Matthew Fisher, Philip Levis, and Pat Hanrahan. Ebb: A dsl for physical simulation on cpus and gpus. *ACM Trans. Graph.*, 35(2):21:1–21:12, May 2016.

[7] Uday Bondhugula, A Hartono, J Ramanujam, and P Sadayappan. Pluto: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*. Citeseer, 2008.

[8] Yanqing Chen, Timothy A Davis, William W Hager, and Sivasankaran Rajamanickam. Algorithm 887: Cholmod, supernodal sparse cholesky factorization and update/downdate. *ACM Transactions on Mathematical Software (TOMS)*, 35(3):22, 2008.

[9] Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. Sympiler: transforming sparse matrix codes by decoupling symbolic analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 13. ACM, 2017.

[10] Edward G Coffman, Jr, Michael R Garey, and David S Johnson. An application of bin-packing to multiprocessor scheduling. *SIAM Journal on Computing*, 7(1):1–17, 1978.

[11] Timothy A Davis. *Direct methods for sparse linear systems*, volume 2. Siam, 2006.

[12] Timothy A Davis. Algorithm 930: Factorize: An object-oriented linear system solver for matlab. *ACM Transactions on Mathematical Software (TOMS)*, 39(4):28, 2013.

[13] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[14] Timothy A Davis and Ekanathan Palamadai Natarajan. Algorithm 907: Klu, a direct sparse solver for circuit simulation problems. *ACM Transactions on Mathematical Software (TOMS)*, 37(3):36, 2010.

[15] James W Demmel, Stanley C Eisenstat, John R Gilbert, Xiaoye S Li, and Joseph WH Liu. A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20(3):720–755, 1999.

[16] James W Demmel, John R Gilbert, and Xiaoye S Li. An asynchronous parallel supernodal algorithm for sparse gaussian elimination. *SIAM Journal on Matrix Analysis and Applications*, 20(4):915–952, 1999.

[17] Perry A Emrath, S Chosh, and David A Padua. Event synchronization analysis for debugging parallel programs. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 580–588. ACM, 1989.

[18] Alan George, Joseph WH Liu, and Esmond Ng. Communication results for parallel sparse cholesky factorization on a hypercube. *Parallel Computing*, 10(3):287–298, 1989.

[19] John R Gilbert and Robert Schreiber. Highly parallel sparse cholesky factorization. *SIAM Journal on Scientific and Statistical Computing*, 13(5):1151–1172, 1992.

[20] R Govindarajan and Jayvant Anantpur. 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)*, pages 1–10. IEEE Computer Society, 2013.

[21] John A Gunnels, Fred G Gustavson, Greg M Henry, and Robert A Van De Geijn. Flame: Formal linear algebra methods environment. *ACM Transactions on Mathematical Software (TOMS)*, 27(4):422–455, 2001.

[22] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2017.

[23] Pascal Hénon, Pierre Ramet, and Jean Roman. Pastix: a high-performance parallel direct solver for sparse symmetric positive definite systems. *Parallel Computing*, 28(2):301–321, 2002.

[24] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 311–320, New York, NY, USA, 2012. ACM.

[25] David S Johnson. Fast algorithms for bin packing. *Journal of Computer and System Sciences*, 8(3):272–314, 1974.

[26] Sam Kamin, María Jesús Garzarán, Barış Aktemur, Danqing Xu, Buse Yılmaz, and Zhongbo Chen. Optimization by runtime specialization for sparse matrix-vector multiplication. In *ACM SIGPLAN Notices*, volume 50, pages 93–102. ACM, 2014.

[27] George Karypis and Vipin Kumar. A high performance sparse cholesky factorization algorithm for scalable parallel computers. In *Frontiers of Massively Parallel Computation, 1995. Proceedings. Frontiers' 95., Fifth Symposium on the*, pages 140–147. IEEE, 1995.

[28] George Karypis and Vipin Kumar. A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. *University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, Minneapolis, MN*, 1998.

[29] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):77, 2017.

[30] Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David IW Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M Kaufman, Gurtej Kanwar, Wojciech Matusik, and Saman Amarasinghe. Simit: A language for physical simulation. *ACM Transactions on Graphics (TOG)*, 35(2):20, 2016.

[31] Ruipeng Li and Yousef Saad. Gpu-accelerated preconditioned iterative linear solvers. *The Journal of Supercomputing*, 63(2):443–466, 2013.

[32] Xiaoye S Li. An overview of superlu: Algorithms, implementation, and user interface. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):302–325, 2005.

[33] Amy W Lim, Gerald I Cheong, and Monica S Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *Proceedings of the 13th international conference on Supercomputing*, pages 228–237. ACM, 1999.

[34] Joseph W. H. Liu. The role of elimination trees in sparse factorization. *SIAM J. Matrix Anal. Appl.*, 11(1):134–172, January 1990.

[35] Weifeng Liu, Ang Li, Jonathan Hogg, Iain S Duff, and Brian Vinter. A synchronization-free algorithm for parallel sparse triangular solves. In *European Conference on Parallel Processing*, pages 617–630. Springer, 2016.

[36] Fabio Luporini, David A Ham, and Paul HJ Kelly. An algorithm for the optimization of finite element integration loops. *arXiv preprint arXiv:1604.05872*, 2016.

[37] Duane Merrill and Michael Garland. Merge-based parallel sparse matrix-vector multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 58. IEEE Press, 2016.

[38] Maxim 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*, 1, 2011.

[39] Jongsoo Park, Mikhail Smelyanskiy, Narayanan Sundaram, and Pradeep Dubey. Sparsifying synchronization for high-performance shared-memory sparse triangular solver. In *International Supercomputing Conference*, pages 124–140. Springer, 2014.

[40] François Pellegrini and Jean Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *International Conference on High-Performance Computing and Networking*, pages 493–498. Springer, 1996.

[41] Alex Pothen and Chunguang Sun. A mapping algorithm for parallel sparse cholesky factorization. *SIAM Journal on Scientific Computing*, 14(5):1253–1257, 1993.

[42] Alex Pothen and Sivan Toledo. Elimination structures in scientific computing., 2004.

[43] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[44] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

[45] Lawrence Rauchwerger, Nancy M Amato, and David A Padua. Run-time methods for parallelizing partially parallel loops. In *Proceedings of the 9th international conference on Supercomputing*, pages 137–146. ACM, 1995.

[46] Hongbo Rong, Jongsoo Park, Lingxiang Xiang, Todd A Anderson, and Mikhail Smelyanskiy. Sparso: Context-driven optimizations of sparse linear algebra. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, pages 247–259. ACM, 2016.

[47] Olaf Schenk and Klaus Gärtner. Two-level dynamic scheduling in pardiso: Improved scalability on shared memory multiprocessing systems. *Parallel Computing*, 28(2):187–197, 2002.

[48] Olaf Schenk, Klaus Gärtner, Wolfgang Fichtner, and Andreas Stricker. Pardiso: a high-performance serial and parallel sparse linear solver in semiconductor device simulation. *Future Generation Computer Systems*, 18(1):69–78, 2001.

[49] Fengguang Song, Asim YarKhan, and Jack Dongarra. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In *High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on*, pages 1–11. IEEE, 2009.

[50] Daniele G Spampinato and Markus Püschel. A basic linear algebra compiler. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 23. ACM, 2014.

[51] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, Jonathan Freeman, and Barbara Kreaseck. Combining performance aspects of irregular gauss-seidel via sparse tiling. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 90–110. Springer, 2002.

[52] Yuan Tang, Rezaul Alam Chowdhury, Bradley C Kuszmaul, Chi-Keung Luk, and Charles E Leiserson. The pochoir stencil compiler. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 117–128. ACM, 2011.

[53] Ehsan Totoni, Michael T Heath, and Laxmikant V Kale. Structure-adaptive parallel solution of sparse triangular linear systems. *Parallel Computing*, 40(9):454–470, 2014.

[54] Anand Venkat, Mahdi Soltan Mohammadi, Jongsoo Park, Hongbo Rong, Rajkishore Barik, Michelle Mills Strout, and Mary Hall. Automating wavefront parallelization for sparse matrix computations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 41. IEEE Press, 2016.

[55] Richard Vuduc, Shoaib Kamil, Jen Hsu, Rajesh Nishtala, James W Demmel, and Katherine A Yelick. Automatic performance tuning and analysis of sparse triangular solve. ICS, 2002.

[56] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi*, pages 167–188. Springer, 2014.

[57] Xinliang Wang, Wei Xue, Weifeng Liu, and Li 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*, pages 338–353. ACM, 2018.

[58] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix–vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178–194, 2009.

[59] Xiaotong Zhuang, Alexandre E Eichenberger, Yangchun Luo, Kevin O'Brien, and Kathryn O'Brien. Exploiting parallelism with dependence-aware scheduling. In *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*, pages 193–202. IEEE, 2009.

```
1 H-Level:
2 for(I1){
3    .
4     .
5      .
6
7          for(In(I1)){
8 Atomic:
9              c /= a[idx(I1,...,In)]; }}
```

(a) Before

```
1 for(every l-partition i){
2 #pragma omp parallel for private(pVars)
3      for(every w-partition j){
4        for(every v ∈ HLevelSet[i][j]){
5          I1 = v;
6          ...
7            for(In(I1)){
8 #pragma omp atomic
9              c /= a[idx(I1 ,... ,In)];}}}}}
```

(b) After

Level, loop[1].HLevel(HLevelSet,pVars)

Fig. 12: The H-Level transformation. The loop over I_1 in (a) transforms to two nested loops that iterate over the `H-Level set` in (b). Any use of the original loop index I_1 is replaced with its corresponding value from `HLevelSet`.

APPENDIX

A. General From of Code Transformation

Figure 12 shows the general form of the H-level transformation. The loop in line 2 of the code in Figure 12a is changed to lines 1–4 in the code in Figure 12b. After transformation, all operations and indices that use I_1, which is the index of the transformed loop, will be replaced with a proper value from `HLevelSet`. The parallel pragma in line 2 ensures that all w-partitions within an l-partition run in parallel. Note that some algorithms may require atomic pragmas; such cases are detectable using existing analysis techniques [17].

B. Experimental Results for Non-Chordal DAGs

In order to test our algorithm on non-chordal DAGs, we take the matrices in Table I and modify them to include only the non-zeros in the lower triangular part of each matrix; we then run triangular solve on this synthetic lower triangular matrix. Unlike the L factors from matrix factorization, these lower triangular matrices are not chordal. Figure 13 compares the performance of ParSy-generated code against the MKL library for the lower triangular part of matrices in Table I. All matrices are first reordered with the Metis ordering method. ParSy code is faster than MKL on average by 1.6×, 2.3×, and 7.0× for Haswell-E (top), Haswell-EP (middle), and Skylake processors respectively. We observe that the heuristic approach used for finding sufficient w-partitions finds enought independent components for LBC to produce a load balanced partitioning. The number of connected components is on average 1019× the target k number of w-partitions for these matrices with non-chordal DAGs.

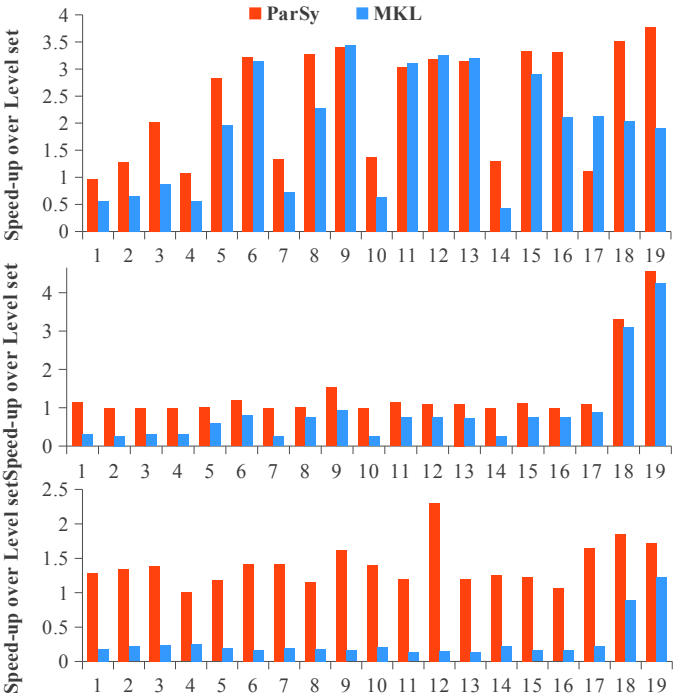


Fig. 13: The performance of ParSy (numeric) for triangular solve compared to MKL (numeric) on Haswell-E (top), Haswell-EP (middle), and Skylake (bottom) processors. All times are normalized over the level set numeric time.