Supplement material for Adaptive Algebraic Reuse of Reordering in Cholesky Factorizations with Dynamic Sparsity Patterns

BEHROOZ ZAREBAVANI, University of Toronto, Canada
DANNY M. KAUFMAN, Adobe Research, U.S.A
DAVID I.W. LEVIN, University of Toronto, Canada
MARYAM MEHRI DEHNAVI, University of Toronto, Canada

CCS Concepts: • **Dynamic sparse computation**; • **Inspector-Executor Framework**; • **Symbolic Analysis and Numerical Computation**; • **Sparse Cholesky Solver**; • **Matrix Re-ordering**; • **Physics-Based Simulation**; • **Sparse Matrix Computation**;

ACM Reference Format:
Behrooz Zarebavani, Danny M. Kaufman, David I.W. Levin, and Maryam Mehri Dehnavi. 2025. Supplement material for Adaptive Algebraic Reuse of Reordering in Cholesky Factorizations with Dynamic Sparsity Patterns. *ACM Trans. Graph.* 44, 4 (August 2025), 11 pages. https://doi.org/10.1145/3731179

Contents

1 HGD OVERHEAD DISCUSSION

Here we discuss two overhead involved in using HGD data-structure. (1) The decomposition process of HGD and (2) the usage overhead of HGD's binary tree. In particular, we explain why the computation of the binary tree creation is overlapped with fill-reducing ordering computation itself, and then we discussed both the size of the tree which affect the overhead of traversing the tree for different tasks in Parth and also the process of coarsening sub-graphs, where in Parth, has no overhead.

Authors' addresses: Behrooz Zarebavani, University of Toronto, Toronto, Canada, behrooz.zarebavani@gmail.com; Danny M. Kaufman, Adobe Research, Seatle, U.S.A, kaufman@adobe.com; David I.W. Levin, University of Toronto, Toronto, Canada, diwlevin@cs.toronto.edu; Maryam Mehri Dehnavi, University of Toronto, Toronto, Canada, mmehride@cs.toronto.edu.

0730-0301/2025/8-ART $15.00
https://doi.org/10.1145/3731179

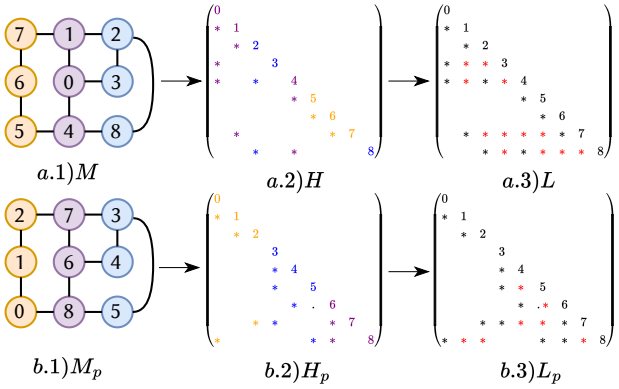
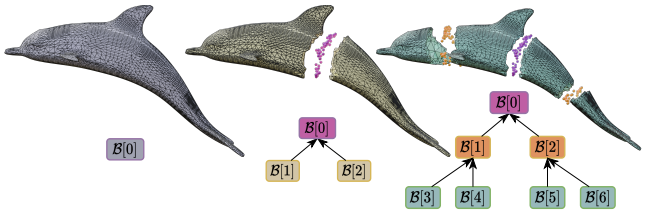


Fig. 1. **Partial fill-reducing ordering using separator set.** Here, the computation used for finding separator set, coloured in purple, is used in the fill-reducing process. This process involves renumbering the left sub-mesh from $\{5, 6, 7\}$ to $\{0, 1, 2\}$, the right sub-mesh from $\{2, 3, 8\}$ to $\{3, 4, 5\}$, and the separator set from $\{1, 0, 4\}$ to $\{7, 6, 8\}$. Consequently, A_p associated with $\mathcal{G}_p$ becomes the permuted version of A. This reordering effectively reduces the fill-ins by separating the computations of $\mathcal{A}_L$ and $\mathcal{A}_R$. As a result, the resulting factor L_p has five fewer fill-ins compared to the factor L of the unordered Hessian H.

Sub-graph Decomposition Overhead: The HGD process leverages the nested-dissection approach [Khaira et al. 1992] to create the sub-graphs. Parth reuses information related to the separator sets and their corresponding left and right sub-graphs for computing the ordering vector which result in lower-overhead. This approach hides the HGD cost within the fill-reducing ordering computation.

To further illustrate how the information in $\mathcal{B}$ contributes to fill-reducing ordering computation, refer to Figure 1. In this figure, the separator set, coloured in purple, is used in reducing fill-ins. The procedure effectively renumbers the nodes in the G such that the computation of the separator set occurs at the end of the factor computation. When comparing Figures 1(a.1) and 1(b.1), it becomes apparent that the graph G undergoes renumbering, positioning the separator set numbers after the orange and blue coloured left and right sub-meshes, respectively. A comparison of the matrix A in Figure 1(a.2) with A_p in Figure 1(b.2) shows an identical count of non-zero entries; however, their sparsity patterns differ due to the renumbering. Factoring both A and A_p shows that the factor L_p possesses fewer fill-ins than L, thus demonstrating fill-reduction achieved by the permutation.



a) max_level = 0 b) max_level = 1 c) max_level = 2

Fig. 2. **HGD evaluation on "Dolphin" mesh for a Laplace-Beltrami operator.** Here, the graph G is identical to the mesh, as each node in the graph corresponds to a DOF in the mesh, and each edge in the graph arises from the connection of DOFs within an element. The figure shows the HGD evaluation for max_level = $\{0, 1, 2\}$. Furthermore, observe how merging fine-grained sub-meshes $\mathcal{B}[1]$, $\mathcal{B}[3]$, and $\mathcal{B}[4]$ in Figure 2(c) results in the coarser sub-graph $\mathcal{B}[1]$ in Figure 2(b).

Binary Tree Usage Overhead: In practice, Parth achieves reasonable reuse by using a small $\mathcal{B}$, regardless of the mesh resolution. This results in low overhead when traversing the binary tree. To understand why a small $\mathcal{B}$ provides effective reuse, we examine the relationship between the size of $\mathcal{B}$ and Parth's reuse capability.

To elaborate, let us consider how much reuse Parth can provide when max_level = 2. Regardless of how Parth confines the changes, these changes occur either within fine-grain or coarse-grain sub-graphs. When max_level = 2, each leaf node of $\mathcal{B}$ represents roughly 25% of the total graph (considering small separator sets). Consequently, Parth can confine changes to either a fine-grain sub-graph of size 25%, achieving 75% reuse on the unchanged sub-graph, or to coarser sub-graphs of sizes 50% or 100%, which are the only coarse sizes generated by $\mathcal{B}$. This results in discretized reuse levels of 75%, 50%, or 0% (where changes cannot be confined locally).

Increasing 'max_level' to 3 produces fine-grain sub-graphs comprising 12.5% of the total graph, resulting in maximum reuse levels of 87.5%, 75%, 62.5%, down to 0%. Thus, 'max_level' controls the granularity of reuse that Parth can provide. Setting 'max_level' to 7 creates a binary tree ($\mathcal{B}$) with 128 leaf nodes, approximating sub-graphs smaller than 1% of the total graph size. This allows for reuse granularity below 1%, enabling Parth's reuse capability to exceed 99% regardless of mesh resolution, sufficient in practice. Additionally, a full binary tree with 'max_level = 7' contains 255 nodes, and traversing such a small tree incurs minimal overhead in practice.

Finally, note that coarsening sub-graphs using this binary tree model does not involve explicitly coarsening the nodes in $\mathcal{B}$. Instead, Parth treats a sub-tree in $\mathcal{B}$ as a coarsened sub-graph. Specifically, a separator and its corresponding left and right sub-graphs are treated as a single coarse sub-graph. This can be seen in Figure 2, which illustrates how the coarse-grain sub-graphs in Figure 2(b) (colored yellow) are formed by viewing the fine-grain sub-graphs and their common separators in Figure 2(c) (colored blue) as single coarse sub-graph.

2 SYNCHRONIZER: *NodeChangeSynchronizer*

By removing or adding a set of nodes from a graph G, a new graph G_{new} is created. By adding and removing nodes, the indices of the nodes in G_{new} change. For example, Figure 3 illustrates how removing a node and adding a new one can alter the labelling of the node in a graph. The objective of this heuristic is to synchronize $\mathcal{B}$ with the G_{new} to represent the latest node structure. Furthermore, it provides the necessary information for the *Assembler* module to update the permutation vector accordingly.

The Synchronizer uses the map array to detect the added and removed nodes in the graph. To formally define the map, *map* is a function $map : X \rightarrow Y$, where X represents all the node indices in G_{new}, denoted as $X \in G_{new}$, and similarly, $Y \in G \cup \{-1\}$. Any node that is in G but not in the range of *map* (for example, node 0 is not in the range of *map* in Figure 3(a)) is considered a deleted node. Additionally, any node c for which $map[c] = -1$ is an added node. It is natural to assume that if $map[c_1] \neq -1 \wedge map[c_2] \neq -1$, then

Algorithm 1 *NodeChangeSynchronizer*

Global: $\mathcal{B}$
Input: *map*
if *map* is empty **then**
  return
end if
/* Step 1: Deleting the removed nodes from $\mathcal{B}$ */
$N_D = deletedNodes(map, |G|)$
for *node* in S_D **do**
  $i = getSubGraphID(node)$
  $remove(node, \mathcal{B}[i].nodes)$
  $update(node, \mathcal{B}[i].\mathcal{P}_l)$
end for
/* Step 2: Update the index of nodes in $\mathcal{B}$ */
$UpdateNodeIndex(\mathcal{B}[i], map)$
/* Step 3: Assign a sub-graph to each added node */
$N_A = addedNodes(map)$
$ChangeHappened = True$
$FullQueue = N_A$ **and** $EmptyQueue$ = Empty
while $ChangeHappened$ **do**
  $ChangeHappened = False$
  while $\neg FullQueue$.isEmpty() **do**
    $node = FullQueue$.front()
    $FullQueue$.pop()
    $S_{\mathcal{B}} = getAllSubGraphs(node)$
    if $S_{\mathcal{B}}$.isEmpty() **then**
      $EmptyQueue$.push($node$)
    else
      $i = LCA(S_{\mathcal{B}})$
      $AssignNodeToSubGraph(i, node)$
      $ChangeHappened = True$
    end if
  end while
  if $ChangeHappened$ **then**
    $Swap(FullQueue, EmptyQueue)$
  end if
end while
if $\neg EmptyQueue$.isEmpty() **then**
  $AssignNodeToEmptySubGraphORLast(i, node)$
end if

$map[c_1] \neq map[c_2]$, which implies that no two nodes in G_{new} can be mapped to a single node in G. Based on our experience, this map is a common structure in remesher libraries, and we successfully extract this information from the subroutines used in remeshers such as those in [Bischoff et al. 2002; Botsch and Kobbelt 2004; Pfaff et al. 2014; Schmidt et al. 2023]. The loop subdivision of IGL [Jacobson et al. 2013] also provides this information as output. Algorithm 1 describes the synchronization process, which is performed in 3 steps as follows:

Step 1: Deleting the removed nodes from $\mathcal{B}$: In the first step of this heuristic, Parth deletes the removed nodes from $\mathcal{B}$. This process also involves updating the local $\mathcal{P}_l$ accordingly (Lines 4-8 in Algorithm 1). For example, in Figure 3(a), the removed nodes $N_D = \{0\}$ are identified using the map since they are not present in the range of the map. Then, in Figure 3(b), this node is deleted from $\mathcal{B}[0]$.

Step 2: Updating the indices: By adding and deleting DOFs from a mesh and renaming the DOFs based on that process, the corresponding naming of the nodes in the graph also changes. As a result, it is crucial to update the indices of the nodes to maintain consistency. Using the *map*, Parth updates the indices (Line 9). For example, in Figure 3(b), since the new name for node 8 in G is 0, the indices of nodes in the sub-graph represented by $\mathcal{B}[6]$ are updated from $\{8\}$ to $\{0\}$. In practice, Steps 1 and 2 are applied simultaneously for computational reuse.

Step 3: Assign a sub-graph to each added node: After updating the indices and deleting the nodes, the Synchronizer assigns the added nodes to $\mathcal{B}$ (Lines 11-34). Parth uses a heuristic that assigns the nodes to each sub-graph based on their neighbours. For each added node, he first computes all the sub-graphs that are adjacent to it. If there is only a single sub-graph, then that node is assigned to that sub-graph. If it is adjacent to more than one sub-graph, the lowest common ancestor in $\mathcal{B}$ is selected as the sub-graph to which it will be assigned. As an example, see Figure 3(c). By doing so, we maintain the separator relation across each sub-graph. Finally, this process repeats in a greedy manner to assign all the nodes to sub-graphs. If a completely separated graph is added, Parth simply assigns that separated graph to a leaf in $\mathcal{B}$.

3 SYNCHRONIZER: *DirtySubGraphDetection*

The objective of the algorithm 2 is to first categorize the edges into three groups. (I) The edges that show a change in connectivity within a sub-graph. (II) The edges that connect a separator set to its left and right sub-graphs. (III) Finally, the edges violate a separator condition by connecting two sub-graphs that are otherwise completely separated from each other. Lines (3–5) detect the first group by checking whether the two ends of the edge are within the same sub-graph, i.e., $a = b$. Line 7 distinguishes between groups (II) and (III). If this condition is true, it means that the edge is between a separator and its corresponding left and right sub-graphs. Otherwise, it violates a separator. To find this separator, we simply need to find the Lowest Common Ancestor (LCA) of the two sub-graphs, a and b, by traversing the $\mathcal{B}$ (line 8).

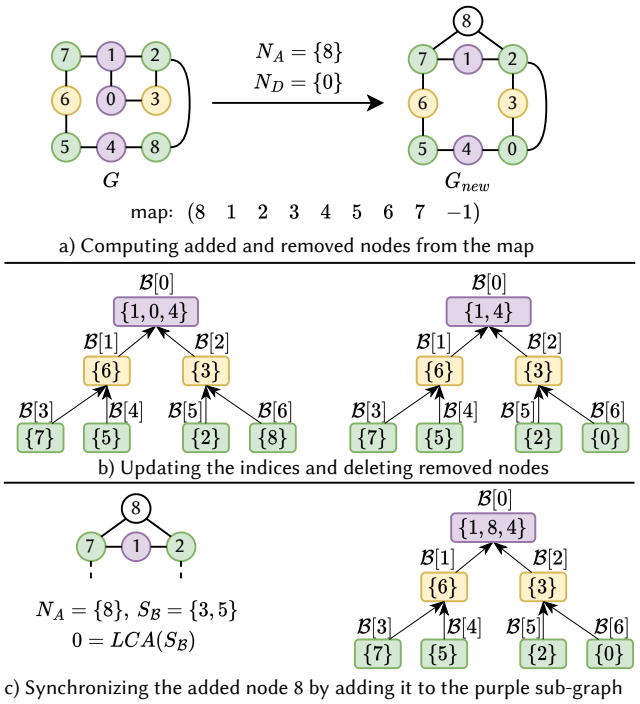


Fig. 3. **Example of the Node Synchronizer.** In this example, node 0 is deleted from G and node 8 is added to G_{new}. Note that since node 8 in G_{new} is newly added, $map[8] = -1$. Also, note that node 8 in G and node 0 in G_{new} still represent the same node. In (b), we can see that $\mathcal{B}$ is oblivious to the naming of the nodes, and the indices can simply be updated as $\mathcal{B}$ is created based on the structure of the graph. Finally, in (c), we can see that the heuristic synchronizes the added node 8 into $\mathcal{B}$ by computing the lowest common ancestor between sub-graphs that it is connected to. This maintains the separator relation between sub-graphs.

Algorithm 2 *DirtySubGraphDetection*

Global: $\mathcal{B}$
Input: $E_{\mathcal{B}}$
Output: D_C, D_F

```
    /*Detect dirty sub-graph in B*/
1:  for < B[a], B[b] > in E_B do
    /*Within sub-graph change*/
2:    if a == b then
3:      D_F.insert(a)
4:      Continue
5:    end if
    /*Across sub-graph changes*/
6:    s_min ← min(a, b) and s_max ← max(a, b)
    /*Filter changes between separators and their ancestors*/
7:    if !s_max.isDescendent(s_min) then
8:      D_C.insert(LCA(a, b))
9:    end if
10: end for
```

Algorithm 3 *MarkAndDecomposeSubGraphs*

Global: $\mathcal{B}$
Input: D_C, D_F
Output: $C_{\mathcal{B}}$
/*Mark the fine-grain sub-graphs for fill-reducing ordering*/
1: **for** a in D_F **do**
2: $C_{\mathcal{B}}[a] = False$
3: **end for**
/*Re-decompose dirty sub-graphs in $\mathcal{B}$*/
4: **for** a in D_C **do**
5: $nodes \leftarrow getCoarseSubGraphNodes(\mathcal{B}[a])$
6: $G_{sub} \leftarrow getSubGraph(G, nodes)$
7: $l \leftarrow ComputeCurrentLevel(a)$
8: $HGD(G_{sub}, l, i$,max_level)
/*Mark new fine-grain sub-graphs for fill-reducing ordering*/
9: $C_{\mathcal{B}}[a] \leftarrow False$
10: $setFalseAllDescendents(\mathcal{B}[a], C_{\mathcal{B}})$
11: **end for**

4 SYNCHRONIZER: *MarkAndDecomposeSubGraphs*

Algorithm 3 uses the sets D_C and D_F to assign values to the cache $C_{\mathcal{B}}$. This array is then used as an indicator of sub-graphs that need an updated local permutation vector $\mathcal{P}_l$ (see Section 4.3). The entries of $C_{\mathcal{B}}$ are computed in two steps. First, fine-grain sub-graphs indicated by D_F are marked as not cached in $C_{\mathcal{B}}$. This means that the fill-reducing ordering information for these sub-graphs is not valid and needs to be recomputed by *Assembler*. To handle changes in coarse-grain sub-graphs, the HGD algorithm is used to re-decompose them, creating a set of valid separators. Lines 5-8 first extract the coarse-grain sub-graph, then apply the HGD algorithm for re-decomposition. Note that the input to HGD is defined so that the sub-tree in $\mathcal{B}$ representing the coarse-grain sub-graph is replaced with a new sub-tree containing the same number of sub-graphs (same sub-tree structure). After computing a valid set of sub-graphs, each of the sub-graphs is marked as not cached in $C_{\mathcal{B}}$ (line 10) so that *Assembler* can recompute the fill-reducing ordering for each of them.

5 SYNCHRONIZER: AGGRESSIVE REUSE

One of the problems with the Parth pipeline is the strict requirement that even for a single violation between two sub-graphs, with the root of $\mathcal{B}$ as their lowest common ancestor, the reuse is zero. Although this effect sometimes occurs in our IPC benchmark, it does not result in significant performance loss. However, since we provide whisker plots of speedups to show the wide range of possible outcomes with various meshes and configurations of local changes, we also need to address this problem.

Based on the IPC benchmark, we observe that zero reuse can happen when two objects collide. In these scenarios, the root in $\mathcal{B}$ is an empty set when the objects are not colliding (see Frame 0 in Figure 4). At the moment of collision, reuse is zero because the empty root is no longer a separator. The point of contact then becomes the separator (Frame 7). If the contact area includes a small number of DOFs, fluctuations in the contact area result in zero

Algorithm 4 *AggressiveReuse*

Global: $\mathcal{B}$
Input: $E_G, l_{LCA}, |G|$
Output: D_F
/*Step 1: Count the node occurrence*/
1: $counter = Vector(|G|, 0)$
2: **for** <a,b> in E_G **do**
3: $counter[a]++$
4: $counter[b]++$
5: **end for**
/*Step 2: Relocate*/
6: **for** <a,b> in E_G **do**
7: **if** $a! = b$ and $level(LCA(a,b)) < l_{LCA}$ **then**
8: $D_F.append(LCA(a,b)$
9: **if** counter[a] > counter[b] **then**
10: $Move(a, LCA(a,b))$
11: **else**
12: $Move(b, LCA(a,b))$
13: **end if**
14: **end if**
15: **end for**

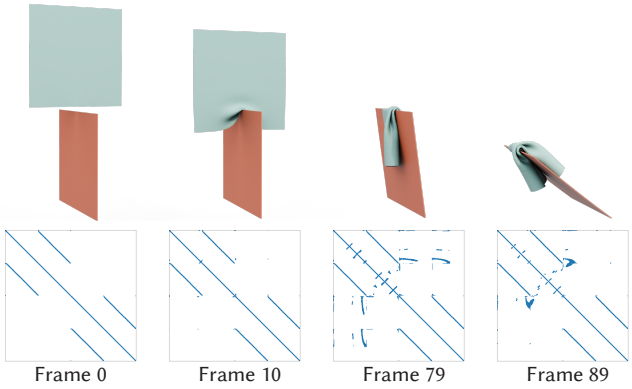


Fig. 4. **4 samples from the "Mat on Board" simulation in the IPC [Li et al. 2020] benchmark.** The top row shows the simulation scene, while the bottom row displays the first system of linear equations required to be solved for these frames. As observed, between frames 0 and 10, when contact occurs, the graph can still be divided into two separate pieces with a small separator, making the contact points a potential separator. Additionally, note that the extent of changes across frames varies depending on the simulation scene. For instance, the difference between frames 0 and 10 is much smaller than the difference between frames 79 and 89.

reuse. However, when the contact area increases, this problem is no longer significant, as the separator is no longer solely the point of contact. This can also occur in the remeshing benchmark, for example, when a DOF corresponding to a separator node in the graph collapses into its left or right sub-meshes. This results in all the edges connected to that separator violating the separator sub-graph condition (see Figure 5 for an example). To alleviate this problem, Parth uses Algorithm 4 to provide reuse even in these scenarios.

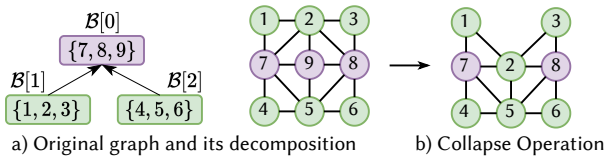


Fig. 5. **Collapse Example.** Remeshers used in SurfaceMap [Schmidt et al. 2023], IGL decimate function [Jacobson et al. 2013] and Botsch and Kobbelt [2004] use operations such as flip, split and collapse. Here, an example of collapse is shown where node 2 from the left sub-graph is collapsed into node 9 which is a separator, resulting in a violation of separator set properties as now there is a connection between $\mathcal{B}[1]$ and $\mathcal{B}[2]$. Without an aggressive reuse heuristic, Parth needs to re-decompose the whole graph which results in zero reuse.

Algorithm 4 performs the reallocation in two steps. First, it counts the number of occurrences of each node in E_G, which are the problematic edges (Step 1, Lines 1-5). In Step 2, similar to how the Synchronizer module detects and resolves changes in the graph, it finds the edges that have an LCA less than a user-defined level (l_{LCA}). In other words, the user can define when not to re-decompose a coarse-grained sub-graph using this variable. By detecting the separator set that has a problem with a specific edge <a,b>using the LCA function, the node with higher occurrence is moved to that specific separator sub-graph. Note that this heuristic is a greedy approach to reduce the number of nodes reallocated to the separator sub-graph, as we do not want to significantly increase the separator sizes. Also, note that we omit implementation details here. For example, Parth checks whether a node has already been reallocated or not. However, these details can be found in the open-source code.

6 PARSY AND EIGEN EVALUATION

Figure 6 shows that the numerical performance of MKL is now faster than Parsy and Eigen for three simulations from the IPC benchmark and for applying the Laplace-Beltrami operator on our triangle-mesh dataset. Note that Parsy is the combination of Sympiler [Cheshmi et al. 2017] and Parsy [Cheshmi et al. 2018], and Eigen is the default simplicial LLT. Additionally, for supernodal computation, Eigen acts as a wrapper around the Cholesky solvers—CHOLMOD, Apple Accelerate, and MKL—and its performance is consistent across them.

Based on this analysis, we can see that Parsy is generally slower than MKL, and Eigen is much slower than MKL for tetrahedral meshes. For triangle meshes, which are sparser, simplicial LLT can be performant, especially for small meshes. However, since our focus is on high-resolution meshes, including Eigen does not make sense. Note that if one wants to use Parsy with Eigen, one must permute the input matrix and use the natural ordering, as Eigen does not accept user-defined ordering.

7 IPC: NUMERICAL PERFORMANCE ANALYSIS

In Section 5.5 we demonstrate that across our benchmark, Parth generates state-of-the-art quality fill-reduction for its factors with significant runtime speedups. A final, and critical, measure of quality for all fill-reduction methods is to sanity check the resulting numerical *quality* of the symbolic analyses which, for each method,

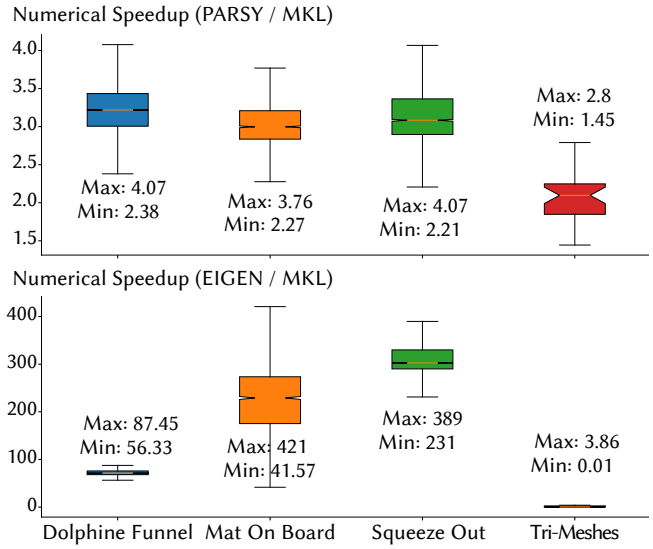


Fig. 6. **MKL Vs. Parsy and Eigen Evaluation:** This whisker plot shows the distribution of MKL numerical speedup over Parsy and Eigen. As it is shown, due to MKL's constant development on Intel, MKL numerical performance is now significantly faster than Parsy which is the combination of the LBC algorithm in [Cheshmi et al. 2018] and code generation in [Cheshmi et al. 2017]. Furthermore, MKL is much faster than simplical LLT for high-resolution meshes.

Table 1. **Effect of Parth on the numerical stability of Cholesky solvers.** The table displays the total number of iterations required for the combined Newton-based solver called in end-to-end simulations. It compares results from the high-performance solver alone with those obtained when Parth is integrated. For instance, in the "MKL, Parth" column, the first number represents the total iterations using only MKL, while the second number reflects iterations when Parth is integrated with MKL. Note that MKL does not converge for "Arma Roller" without Parth. Additionally, due to a compatibility issue with IPC, Accelerate is not integrated into the "Simulator" approach.

Example	MKL Default, Parth	CHOLMOD Default, Parth
(1) Dolphin Funnel	36154, 35748	34625, 35406
(2) Ball mesh roller	58438, 58212	61224, 57925
(3) Mat On Board	2797, 2788	2811, 2817
(4) Rods Twist	8287, 8210	8345, 8267
(5) Squeeze out	12157,12216	12305, 12322
(6) Arma Roller	-, 25950	24199, 24655

is determined primarily from rounding errors accrued by varying approaches to permutation and parallelization [Anand 1980]. In turn, as discussed above in Section 5.2, these variations lead to changes in the descent directions computed per Newton iteration and so, downstream, the total number of linear solves necessary to complete a simulation. In Table 1, we confirm that Parth-integration preserves

the high-quality accuracy, per solve, required for efficient and effective Newton solves. Here, utilizing the saved initial conditions per time-step in our benchmark (see Section 5.2).

We (re-)solve each Newton time-step problem, across each simulation sequence, via the IPC code's Newton solver, using both default MKL and CHOLMOD solves and our new, Parth-integrated versions. Here we see that Parth-integrated and comparable default solvers converge to the same tolerances, with comparable numbers of iterations for both, demonstrating only minor variations of both slightly larger and smaller iteration counts (With a maximum difference of 5.34% in favour of Parth and 0.14% against Parth). Here, in one notable exception, we observe that MKL is unable to solve a number of iterations in the "Arma Roller" sequence to sufficient accuracy, in turn resulting in non-converging Newton solves and so an incomplete simulation sequence for this portion of the benchmark. This rare but significant failure is not entirely surprising as prior IPC implementations [Li et al. 2023] have avoided MKL Pardiso for this reason. Interestingly, in contrast, we note that Parth-integrated MKL is able to complete more Newton time-step solves for the "Arma Roller" sequence. However, at this time we do not know if this change is due to differences in the quality between a few permutation vectors generated by Parth vs MKL's custom METIS routines or other code variations in the MKL settings that occur when we pass it custom fill-reducing orderings.

Figure 7 demonstrates the runtime change of numerical phase. The analysis suggests that Parth-integrated Cholesky solvers should provide symbolic speedup while leaving the already significantly optimized performance of numerical computation steps (including the Cholesky factorization and sparse triangular forward/backward solves) unharmed. We see this confirmed for all three solvers, where we confirm integration of Parth brings the speedup without additional overhead elsewhere in the already optimized numerical computations of the Cholesky solver packages.

Finally, note that the MKL documentation states that using a user-provided permutation vector disables forward/backward solve parallelism in the MKL numeric phase. However, in practice, we did not find this to be a significant limitation for two reasons. First, it is well known that the runtime of sparse Cholesky factorization is the primary bottleneck in the numeric phase. Second, our analysis shows almost no difference in solve runtime when Parth is used for our IPC benchmark. Figure 8 illustrates the performance difference between MKL using its own permutation vector and MKL using the permutation vector provided by Parth.

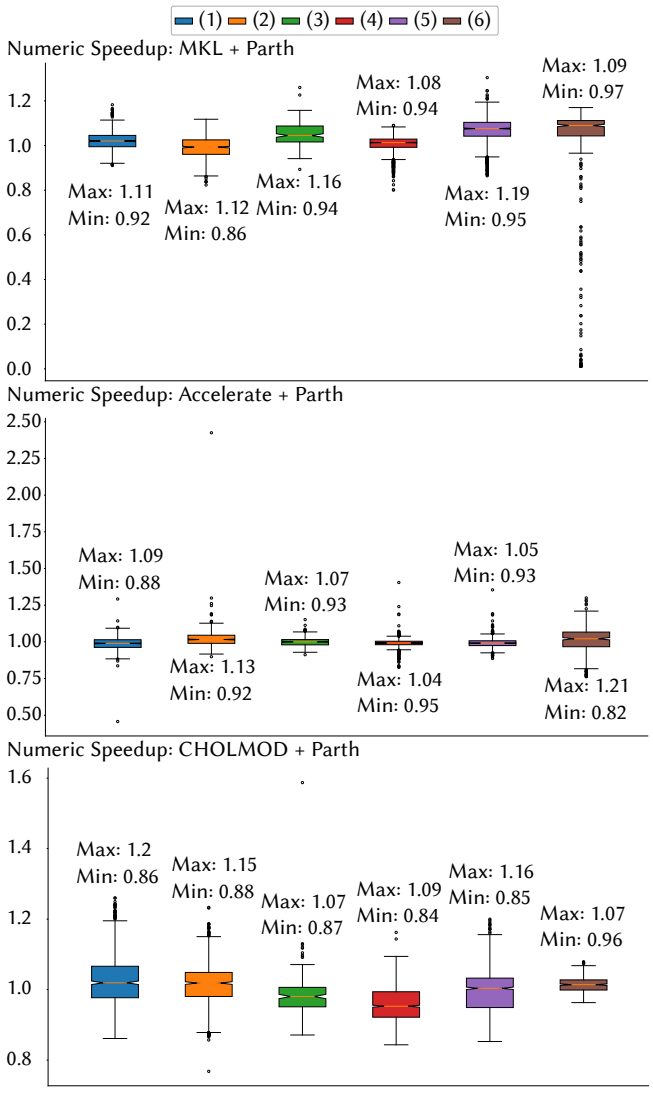


Fig. 7. **IPC: Parth preserves the well-optimized performance of the numerical phases in each base solver across the benchmark.** The three whisker plots indicate the performance impact of Parth on the numeric step of MKL, Accelerate and CHOLMOD with a maximum of 21% performance improvement and 18% performance decrease due to using Parth. Notice that the median of the changes is well within 5% difference showing that the numerical step of these tools delivers comparable performance when using Parth. Note that the slowdown of MKL in "Arma Roller" is due to the numerical instability of MKL in that simulation.

8 REMESHING: NUMERICAL PERFORMANCE ANALYSIS

Figure 9 shows Parth's initialization performance is on par with state-of-the-art fill-reducing ordering algorithms. For 1% changes, Parth outperforms Accelerate and MKL in median speedup. As expected, performance decreases with larger patches due to the global nature of fill-reducing ordering, but even at 20% changes, the median speedup remains around 0.95x. Given the significant symbolic-stage gains, this minor trade-off is acceptable, as shown in Section 5.9.

9 LAGGING ANALYSIS

Lagging reuses fill-reducing order vectors by assuming that the current ordering remains sufficiently good. Because, to the best of our knowledge, no lagging approach exists for matrices with changing dimensions, we focus on cases where only non-zero entries change in the input system of linear equations.

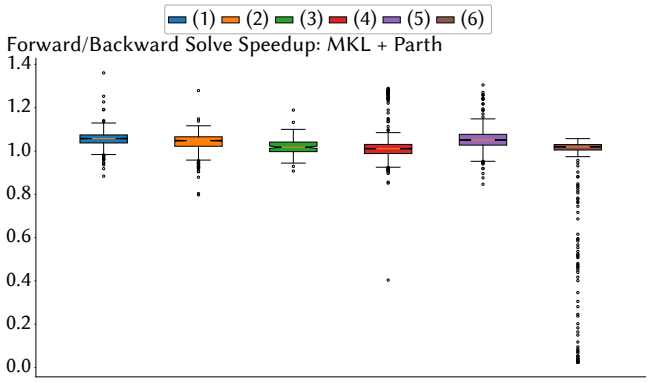


Fig. 8. **IPC: Parth-integrated MKL forward/backward solve speedup.** Here we can see that while MKL disables parallelism for forward/backward solve, the performance change is minimal. This shows that the parallelism effect for these sparsity patterns is not significant. Note that the slowdown of MKL in "Arma Roller" is due to the numerical instability of MKL in that simulation.

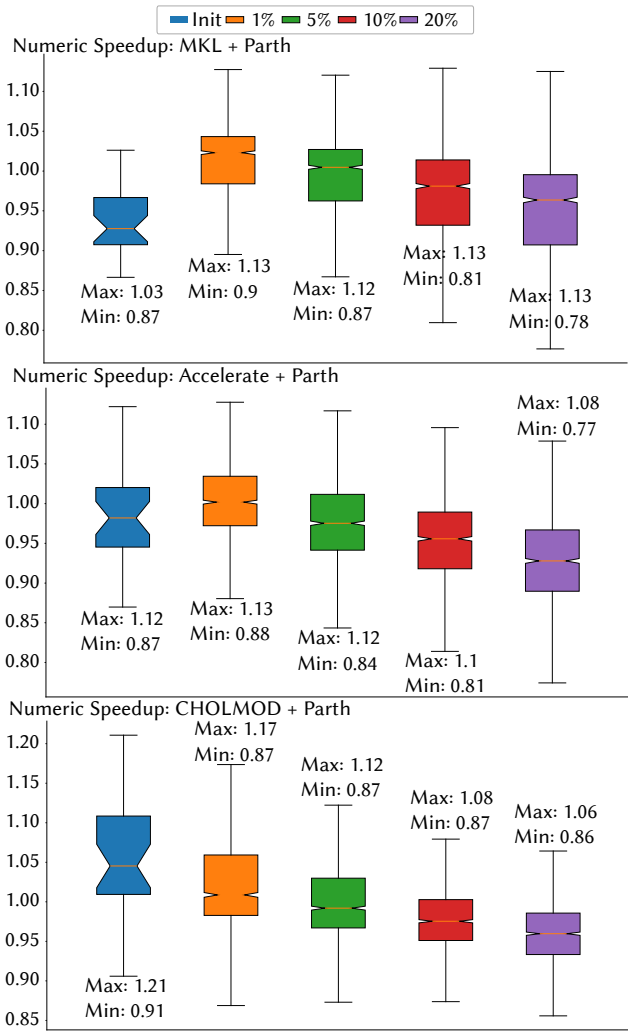


Fig. 9. **Remeshing: Numerical performance of using Parth.** The figure shows the numerical performance of Parth-integrated solvers under remeshing for various patch sizes. Parth demonstrates comparable initialization performance, with median speedups close to 1 for MKL, Accelerate, and CHOLMOD. Notably, for numerical computation, performance at 1% and 5% patch sizes exceeds that of the initialization step for Accelerate and MKL. As expected, performance decreases at 20% patch sizes, but even in this case, the median speedup remains close to 1 for MKL, Accelerate, and CHOLMOD.

We analyze two lagging approaches: (I) lagging every y number of frames, and (II) lagging when an x% of non-zeros in matrix change. Determining both x and y requires heuristics based on domain-specific knowledge. Next, we analyze these approaches and explain why they limit general applicability.

Analysis of Lagging by Frames: Figure 10 shows Cholesky performance in our IPC benchmark when lagging the permutation computation over different frame intervals for two simulations: "Squeeze Out" and "Roller Ball." The baseline (no lag) uses Parth's permutation. For "Squeeze Out," performance falls short of baseline when lagging every {5, 10, 100} frames which means that the search space should be expanded for smaller lagging. For "Roller Ball," lagging 5 or 10 frames can yield good performance, especially as motion stabilizes toward the ends. However, optimal intervals change over time and depend on the simulation scene, which requires domain-specific knowledge known in advance. This adaptive requirement is a major challenge for frame-based lagging. Furthermore, note that lagging cannot eliminate the need for symbolic analysis—only the ordering step may be skipped.

Limitations of Percentage-Based Lagging: Lagging based on a percentage of changes faces two problems. First, the range of changes differs between simulations (e.g., 0.4% for "Squeeze Out" vs. 0.08% for "Roller Ball"), creating scaling issues. Second, the relation between change percentage and performance is not straightforward, making it difficult to set a universal threshold.

To examine how non-zero entries in A affect fill-ins in factor L, we applied the Laplace-Beltrami operator to the "*bunny*" mesh from [Stein 2024] as part of our Remeshing benchmark. We solved a Laplacian problem with CHOLMOD using a Parth-generated fill-reducing order for max_level = 3. Then, we added 1% extra non-zeros (set to zero) to A, altering its sparsity without affecting numerical stability. By reapplying the solver with the same permutation, we measured non-zeros in L to show that both the amount and location of changes matter. We consider three scenarios for these 1% changes: in the first scenario, all changes occur within a single leaf node of Parth; in the second, they span two leaf nodes that share a common parent; and in the third, the changes affect two leaf nodes whose least common ancestor is the root of the HGD tree.

Figure 11 shows the ratio of non-zeros in L—the sum of fill-ins and non-zeros in A—relative to Parth base case scenarios where we used METIS as the inner ordering algorithm. The results demonstrate that where changes occur drastically affects factor size and performance. We use Parth here to highlight where non-zero entries

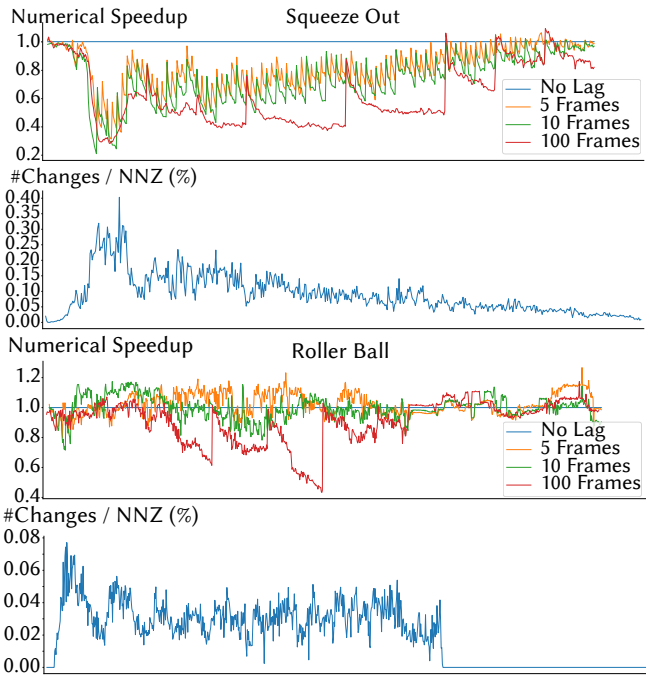


Fig. 10. **Lagging effect on Cholesky numerical performance.** Top: two plots show the numerical speedup, compared to no lagging approach, as well as the percentage of changes for the Squeeze Out simulation. Bottom: the two plots show the numerical speedup and percentage of changes for the Roller Ball simulation. We can see that lagging each x number of frames for Squeeze Out does not perform well, while for Roller Ball it can provide decent results. However, even for Roller Ball, to maximize performance the lagging should be dynamic and change as the simulation progresses. Furthermore, the magnitude of changes is significantly different between Squeeze Out and Roller Ball, showing that domain-specific knowledge is necessary even for different simulations within the same IPC framework.

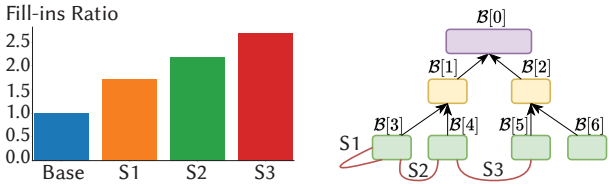


Fig. 11. **Effect of non-zero entries on factor fill-ins.** Left: Bar plot showing the number of non-zero entries in the factor, scaled based on the "Base" case. S1 to S3 represent scenarios 1 to 3, respectively. The number of non-zero entries in the factor for the "Base" case and scenarios S1–S3 are 108,156; 184,986; 236,128; and 290,486, respectively. The red lines on $\mathcal{B}$ on the right show the connections created between sub-graphs. For example, in scenario 1 (S1), all the changes are applied within the $\mathcal{B}[3]$ subclass. Note that a fixed number of changes, when applied in different places, can result in significant changes (2.68x in this experiment) in the factor size, even for a 1% change in the sparsity pattern.

can cause excessive fill-ins if not properly considered. In general, this is related to the elimination tree theory and how Parth relates to the elimination tree. Since it is out of the scope of this paper, please see [Davis et al. 2016; Khaira et al. 1992] for more detail.

As a final observation from this experiment, we want to focus on Parth runtime for fixing these different scenarios. Based on Section 4's description, we can see that for fixing Scenario 1, Parth only focuses on fine-grain sub-graph $\mathcal{B}[3]$, which is a small part of the whole graph. For fixing Scenarios 2 and 3, it needs to consider coarser regions, which also require more computation for providing high-quality fill-ins. Essentially, Parth's runtime is proportional to the importance of the changes, which shows its adaptability. Also, Parth supports a lagging approach where it can ignore changes if they happen above specific levels. We did not evaluate this, as it requires the user to define where they want the effects to be ignored. However, the Parth code base has this capability for practitioners.

Finally, let us consider how much computation is required by Parth for these scenarios. For Scenario 1, Parth targets only a small fine-grain sub-graph $\mathcal{B}[3]$; for Scenarios 2 and 3, it must process larger, coarser regions, increasing computation. Thus, Parth's runtime scales with the importance of changes, demonstrating adaptability. Note that Parth has the capability to ignore changes (lag the update) when the common ancestor is above a certain level. However, we did not analyze this capability in the paper as Parth already produce satisfactory results.

10 PARTH COMPRESSION ANALYSIS

In this section, we first demonstrate the performance benefits of Parth's compression and compare them with those of METIS to highlight their differences. We then provide a runtime analysis of Parth's speedup—with compression enabled—relative to METIS with compression, illustrating the practical performance implications of using Parth over METIS. Note that the scenario in which neither Parth nor METIS uses compression is presented in our "Remeshing benchmark" (Sections 5.7, 5.8, and 5.9). This analysis show the contributions of Parth's compression in isolation as well as Parth's reuse capability for our IPC benchmark where this compression opportunity exists for both Parth and METIS (though, to the best of our knowledge, Accelerate and CHOLMOD do not allow for enabling this compression in their built-in METIS).

Figure 12 shows the performance gain achieved solely by using Parth's compression. To generate this plot, we first measure the runtime of METIS after compressing the graph using Parth's method, feeding the compressed graph into METIS, and then expanding the permutation vector afterward (see Section 4.2 for more details). We then measure the runtime of METIS without any compression. As expected, the variation in speedups is small, since this is a pure computation without reuse. All average speedups exceed 4.6x, with a maximum of 5.27x and a minimum of 4.5x. We exclude outlier speedups in this plot for consistency with the main paper. These outliers—computed based on 1.5x the interquartile range above the third quartile or below the first quartile—occur infrequently.

Next, to show the difference between Parth's compression and METIS's compression, we measure the runtime of METIS using

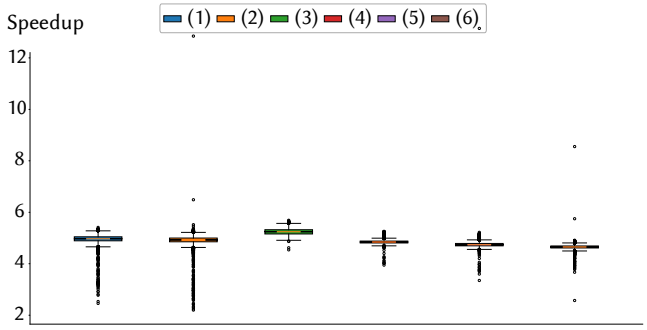


Fig. 12. **IPC: Effect of Parth Compression.** Here we show the speedup that METIS can gain by integrating Parth compression into its pipeline. The average speedup for 6 simulations are 4.96x, 4.92x, 5.24x, 4.85x, 4.75x, 4.65x.

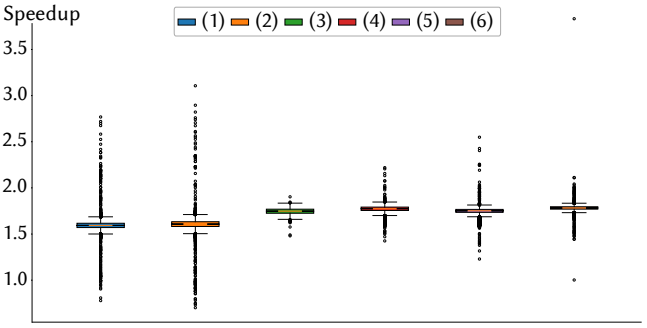


Fig. 13. **IPC: Parth compression vs METIS compression.** Here we show the speedup that METIS can gain by integrating Parth compression into its pipeline versus when METIS is used with its built-in compression. The average speedup for 6 simulations are 1.59x, 1.60x, 1.74x, 1.77x, 1.74x, 1.78x.

Parth's compression integrated into it and compare it to the runtime of METIS using its own built-in compression by setting the METIS_OPTION_COMPRESS flag. Figure 13 shows that Parth's compression outperforms METIS's built-in compression. This is because METIS relies on heuristics such as heavy-edge matching for compression, while Parth uses a straightforward and lightweight approach. Although both methods produce the same compressed graph, the simplicity of Parth's compression leads to a lower overall runtime when integrated into METIS. As in the previous analysis, the variation in speedups is small, with a maximum of 1.85x and a minimum of 1.50x.

To show the reuse benefits of Parth, we compare Parth and METIS when both use their own compression methods. In this case, the additional speedup observed—compared to Figure 13—is attributed to Parth's reuse capability. Figure 14 presents the performance of Parth in these scenarios. Looking at the raw numbers, Parth can achieve speedups as high as 500x. However, for consistency with the main paper text, we exclude outliers; the maximum speedup in this test is 242x, which aligns closely with our previous results, as expected. Based on the reported average and median speedups in Figure 14, we observe significant performance improvements due to reuse in simulations (1), (2), (5), and (6), where the runtime is an

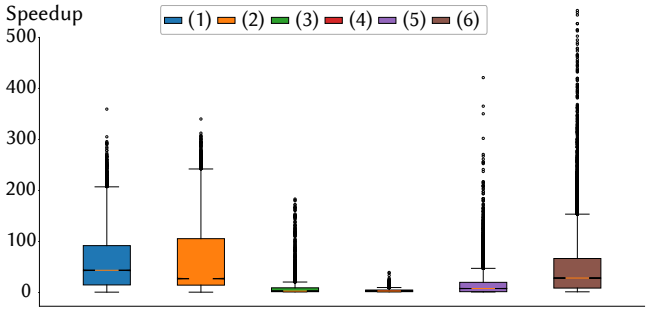


Fig. 14. **IPC: Parth vs METIS with both compression activated.** Here we show the speedup of Parth with its own compression and METIS with its own compression. The average speedup is: 58x, 59x, 11x, 3.67x, 16x, and 47x. The median speedup is: 43x, 27x, 2.7x, 2.8x, 7.77x, 28.32x. Note that the median is lower as it is not going to consider the outlier values.

order of magnitude faster. For example, "Dolphin funnel" shows a 43x median speedup; even assuming a 2x contribution from Parth's faster compression compared to METIS, the remaining 20x speedup can be attributed to reuse.

We next compare this figure with Figure 11 in the main paper. Figure 13 reports Parth's speedup when both Parth and METIS have compression activated, using the open-source METIS on Intel architecture. In contrast, Figure 11 shows Parth's speedup compared to METIS without compression, using the built-in METIS of Cholesky solvers across a mix of Apple M2 Pro and Intel architectures. It also compares against the best METIS runtime across three solvers. Despite METIS having compression enabled in Figure 13, we observe higher speedups than those reported in Figure 11. Due to the many differences in the setup of these two analyses, it is difficult to pinpoint the exact cause of this discrepancy. However, we share our thoughts on this observation below, as we do not have access to the internal implementations of the built-in ordering algorithms in MKL and Apple's Cholesky solvers.

Runtime Measurement. The way we measure the runtime of the built-in ordering algorithms in these solvers is as follows. First, we measure the total runtime of the symbolic analysis phase of the Cholesky solvers. Then, we provide our own permutation vector and measure the symbolic analysis time again. The difference between the symbolic analysis runtime with the built-in ordering and the one without it gives us an approximation of the built-in ordering's runtime. This measurement can result in different scales of speedups; however, the trend of speedups across the six simulations remains consistent between the two figures.

Comparison with Best Performance. Figure 11 compares Parth's performance against the best performance of the built-in METIS implementations in commercial Cholesky solvers. That is, the baseline in this case is more efficient than the open-source METIS. For example, we observed that the built-in METIS algorithms in Cholesky solvers, such as those in MKL, perform faster than the open-source METIS used for the comparison in Figure 13. Specifically, MKL's METIS achieves up to 30% better performance than the open-source version.

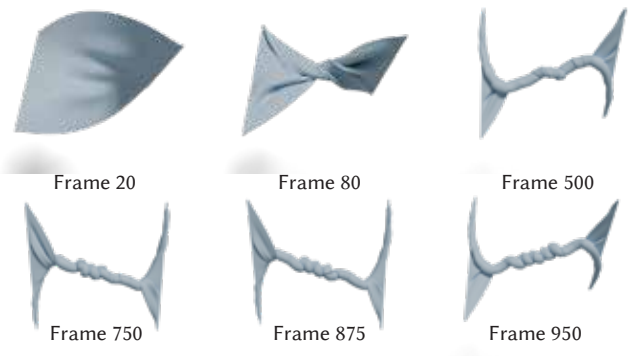


Fig. 15. **4 samples from the "Mat Twist" simulation in the IPC [Li et al. 2020] benchmark.** This simulation involves many self-collision due to its nature, allowing for measuring Parth limits on reuse. Note that each time that a knot created (3 knots for frame 750 and 5 knot for frame 950) there is a fast movement in the middle.

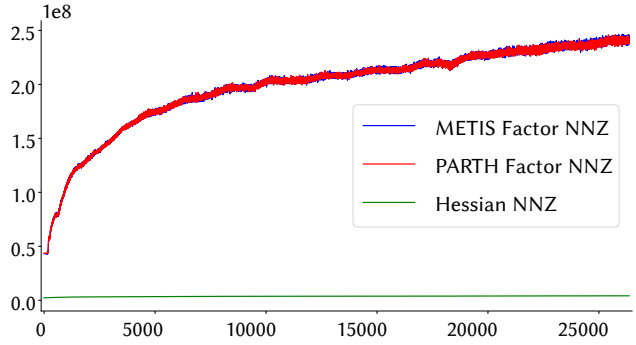


Fig. 16. **Hessian and factor size for "Mat Twist" simulation.** Across all solves performed over 1000 frames of the "Mat Twist" simulation, we plot the factor size (in terms of non-zero entries) for both Parth and METIS. To highlight the difference between the factor and the original matrix, we also show the number of non-zero entries in the Hessian. For example, in the first solve, the Hessian contains approximately 2.2 million non-zero entries, while the factor size is 43 million. By the final solve, the Hessian grows to 4 million non-zeros, and the factor reaches 241 million.

Different Architectures. Note that there is also a significant difference between Apple M2 Pro hardware and Intel Xeon hardware. Since we are not using parallelism in this analysis, the single-core performance of each architecture can lead to differences in runtime for both Parth and METIS. This is another important factor contributing to the observed discrepancies.

METIS Randomness. As mentioned in the Background section, heuristics used by METIS introduce randomness. As a result, applying the same ordering algorithm twice can yield different orderings. Specifically, this may produce different separator sets, which in turn affect the reuse patterns in Parth and lead to variations in the observed speedups.

As a final note, note that throughout the paper, we emphasize on reporting median speedups instead of average as based on our experiment, this value is more expected in practice compare to average. As can be seen due to large outliers with high-speedup (500x), the average runtime can be significantly more than median speedup, even though such speedups are rarely seen. For instance, in Simulation (2), the median speedup is 27x, while the average is 59x. However, this average speedup does not totally reflect practical speedups that a practitioner may due to outliers with high speedup, showing that median report is more reasonable.

11 PARTH ON MAT TWIST SIMULATION

To push the limits of Parth's reuse in IPC simulations, we present a detailed analysis of Parth's reuse across 1,000 frames of the "Mat Twist" simulation in the IPC codebase, where numerous self-collisions occur throughout the simulation. Figure 15 shows 6 sample frames from the 1,000 used in this analysis. As seen in the figure, contacts occur across the entire mat, making it challenging for Parth to localize changes. In the following, we provide a detailed analysis of Parth's ordering quality, reuse, speedup, and edge changes in the dual graph of the Hessian generated during Newton solves in the IPC simulation. We also discuss the limitations and benefits of using Parth for this challenging simulation.

Same as in our Evaluation section, we begin with ordering quality. Figure 16 shows the factor fill-ins resulting from both Parth and METIS permutation vectors, with compression activated in both cases. As expected, Parth provides comparable fill-ins. To offer more insight into the sparsity pattern behavior of this simulation, we also report the number of non-zeros in the Hessian to illustrate the scale of difference between the Hessian and its factor, as well as the additional non-zeros introduced due to the activation of the barrier function in IPC. We observe that the factor size increases by 5x—a significant change, considering the smallest factor has 43 million non-zeros. Furthermore, the Hessian size itself increases from 2.2 million to 4 million non-zeros, which is also substantial and highlights the importance of high-quality fill-reducing ordering.

We now turn to the reuse capability of Parth. The first plot in Figure 17 shows the number of edge changes in the Hessian graph between consecutive solves. To improve clarity, we also average the number of changes across all solves within a single frame.

At the beginning of the simulation, there are no contacts, leading to the full reuse of the permutation vector. However, starting around Frame 80, extensive contacts across the mat introduce over 20,000 changes in the Hessian's non-zero structure. Similar spikes in edge changes are observed around Frames 760, 875, and 950. These frames correspond to moments when the mat twists significantly at its center (see Figure 15). Since these changes are widespread, they reduce Parth's ability to effectively reuse prior orderings.

This change in reuse is directly reflected in the speedup of Parth. Across the 1000 frames, Parth achieves up to a 3.5x speedup per frame for fill-reducing ordering. When reuse drops to near zero, the speedup similarly approaches one. Notably, thanks to Parth's effective compression and minimal overhead in handling changes, the speedup never falls below one—demonstrating consistent performance even in the presence of substantial structural updates.

In summary, this detailed analysis shows that Parth offers reliable performance with minimal overhead, even under demanding

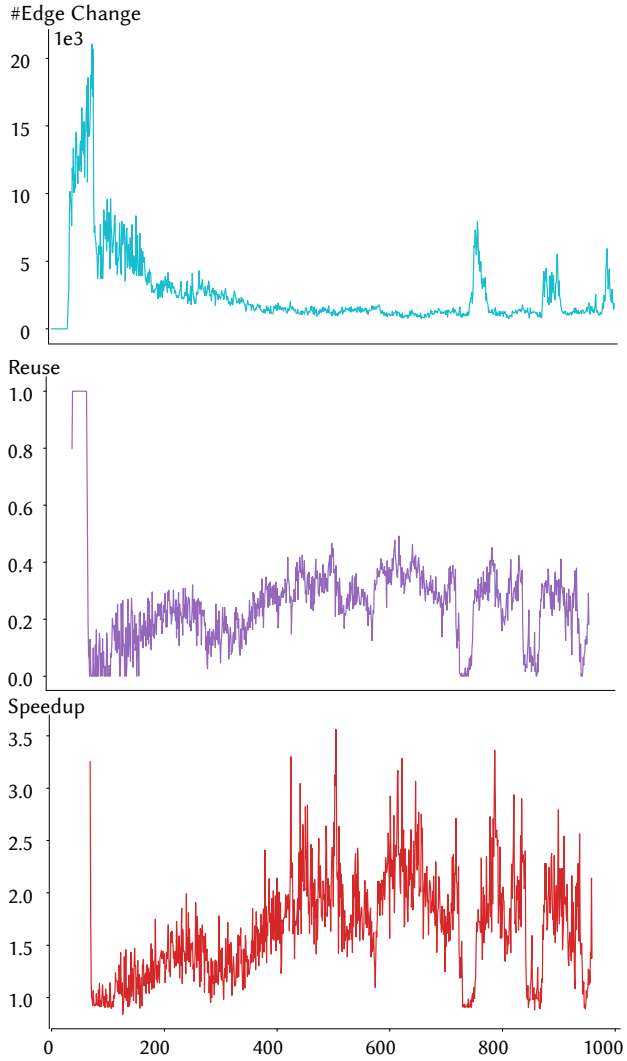


Fig. 17. **Parth performance based on reuse and number of edge changes in the graph dual of the Hessian.**. For the speedup plots, the total ordering computation time per frame is measured for both Parth and METIS, with compression activated in both cases. For the reuse plot, the average reuse across all solves within each frame is reported. The same procedure is followed for the third plot. Note that the number of changed edges is measured relative to the previous solve.

simulation scenarios. Its ability to adaptively reuse ordering information enables efficient, high-performance Cholesky solvers when integrated with commercial direct solvers.

REFERENCES

Indu Mati Anand. 1980. Numerical stability of nested dissection orderings. *Math. Comp.* 35, 152 (1980), 1235–1249.

Botsch Steinberg Bischoff, M Botsch, S Steinberg, S Bischoff, L Kobbelt, and Rwth Aachen. 2002. OpenMesh–a generic and efficient polygon mesh data structure. In *In openSG symposium*, Vol. 18.

Mario Botsch and Leif Kobbelt. 2004. A remeshing approach to multiresolution modeling. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*. 185–192.

Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. 2017. Sympiler: transforming sparse matrix codes by decoupling symbolic analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. 2018. ParSy: inspection and transformation of sparse matrix computations for parallelism. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 779–793.

Timothy A Davis, Sivasankaran Rajamanickam, and Wissam M Sid-Lakhdar. 2016. A survey of direct methods for sparse linear systems. *Acta Numerica* 25 (2016), 383–566.

Alec Jacobson, Daniele Panozzo, C Schüller, Olga Diamanti, Qingnan Zhou, N Pietroni, et al. 2013. libigl: A simple C++ geometry processing library. *Google Scholar* (2013).

Manpreet S Khaira, Gary L Miller, and Thomas J Sheffler. 1992. *Nested Dissection: A survey and comparison of various nested dissection algorithms.* Carnegie-Mellon University. Department of Computer Science.

Minchen Li, Zachary Ferguson, Teseo Schneider, Timothy R Langlois, Denis Zorin, Daniele Panozzo, Chenfanfu Jiang, and Danny M Kaufman. 2020. Incremental potential contact: intersection-and inversion-free, large-deformation dynamics. *ACM Trans. Graph.* 39, 4 (2020), 49.

Minchen Li, Zachary Ferguson, Teseo Schneider, Timothy R Langlois, Denis Zorin, Daniele Panozzo, Chenfanfu Jiang, and Danny M Kaufman. 2023. Private Correspondence with IPC Authors. Personal Communication.

Tobias Pfaff, Rahul Narain, Juan Miguel De Joya, and James F O'Brien. 2014. Adaptive tearing and cracking of thin sheets. *ACM Transactions on Graphics (TOG)* 33, 4 (2014), 1–9.

Patrick Schmidt, Dörte Pieper, and Leif Kobbelt. 2023. Surface maps via adaptive triangulations. In *Computer Graphics Forum*, Vol. 42. Wiley Online Library, 103–117.

Oded Stein. 2024. odedstein-meshes: A Computer Graphics Example Mesh Repository. (2024).