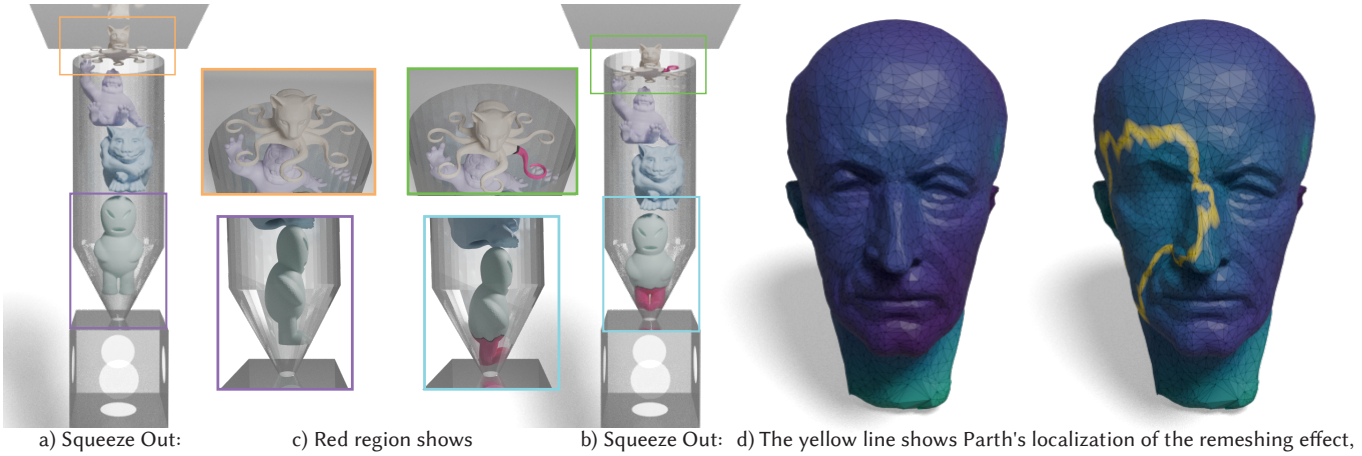
Adaptive Algebraic Reuse of Reordering in Cholesky Factorization with Dynamic Sparsity Pattern

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



a) Squeeze Out: Frame 6 | c) Red region shows Parth's localized region. | b) Squeeze Out: Frame 20 | d) The yellow line shows Parth's localization of the remeshing effect, while the coloring represents the Laplacian solution.

Fig. 1. **Parth's confinement of changes in sparsity patterns mapped onto the simulation mesh.** (a) and (b) show frames 6 and 20 of the Squeeze Out simulation from the IPC benchmark [Li et al. 2020]. Due to the activation of the barrier method, the sparsity pattern of the Hessian changes, necessitating the re-computation of symbolic analysis. Here, we can see that Parth adaptively localizes the changes in the Hessian. This confinement is visible on the mesh, highlighted in red in Figure (c). Note that in the top part of Figure (c), while there is no physical contact, the effect of the barrier function can add or remove non-zero entries as the tip of the tail gets closer to another part of the tail. This confinement results in a 7.1× speedup in symbolic analysis, which leads to a 2.2× speedup per Cholesky solve. In (d), we evaluate a computational pipeline where we first compute a Laplacian operator [Jacobson et al. 2013]. Then, a random patch comprising 2% of the faces in the triangular mesh is chosen and remeshed. After that, we recompute the Laplacian on the mesh. Here, the topological changes due to remeshing alter both the size of the Laplacian matrix and its non-zero entries. Parth reuses 92% of ordering computation in this case resulting in an 8.2× speedup in symbolic analysis and a 5.4× speedup in the Cholesky solve.

Cholesky linear solvers are a critical bottleneck in a wide range of challenging applications within computer graphics and scientific computing. These applications include, but are not limited to, elastodynamic barrier methods such as Incremental Potential Contact (IPC), and geometric operations such as remeshing, and morphology. In these contexts, the sparsity patterns of the linear systems frequently change across successive calls to the Cholesky solver, necessitating repeated symbolic analyses that dominate the overall solver runtime.

To address this bottleneck, we evaluate our method on over 150,000 linear systems generated from diverse nonlinear problems with dynamic sparsity changes in Incremental Potential Contact (IPC) and Patch remeshing on a wide range of triangular meshes with various sizes. Our analysis using three leading sparse Cholesky libraries—Intel MKL Pardiso, SuiteSparse CHOLMOD, and Apple Accelerate—reveals that the primary performance constraint lies in the symbolic re-ordering phase of the solver. Recognizing this, we introduce Parth, an innovative re-ordering method designed to update ordering vectors only where local connectivity changes occur adaptively. Parth employs a novel hierarchical graph decomposition algorithm to break down the dual graph of the input matrix into fine-grained subgraphs, facilitating the selective reuse of fill-reducing orderings when sparsity patterns exhibit temporal coherence.

Our extensive evaluation demonstrates that Parth achieves up to a 255× and 13x speedup in fill-reducing ordering for our IPC and remeshing benchmark and a 6.85x and 10.7x acceleration in symbolic analysis. These enhancements translate to up to 2.95x and 5.89x reduction in overall solver runtime. Additionally, Parth's integration requires only three lines of code, resulting in significant computational savings without the requirement of change in the computational stack. By providing a comprehensive analysis of Parth's

Authors' addresses: Behrooz Zarebavani, University of Toronto, Canada; Danny Kaufman, Adobe Research, U.S.A; David I.W. Levin, University of Toronto, Canada; Maryam Mehri Dehnavi, University of Toronto, Canada.

XXXX-XXXX/2024/11-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

performance across various solvers and applications, we enable practitioners to make informed decisions about their linear solver choice tailored to their specific computational workflows. We hope that Parth facilitates faster and more scalable solutions in dynamic and complex computational environments.

CCS Concepts: • **Physics-Based Simulation**; • **Inspector-Executor Framework**; • **Numerical Methods**; • **Sparse Matrix Computation**;

ACM Reference Format:
Behrooz Zarebavani, Danny Kaufman, David I.W. Levin, and Maryam Mehri Dehnavi. 2024. Adaptive Algebraic Reuse of Reordering in Cholesky Factorization with Dynamic Sparsity Pattern. 1, 1 (November 2024), 24 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Linear solvers lie at the heart of many applications in graphics and scientific computing. Due to the structure induced by mesh-based computations, many common operations such as solving partial differential equations or minimizing a variational energy give rise to sparse systems of linear equations – for the solution of which, it is either necessary or convenient to rely on Cholesky solvers. As a result, their accelerations are well-studied. However, the runtime of graphic applications often remain dominated by the cost of Cholesky solves, necessitating their further acceleration.

Efficient sparse Cholesky solvers are typically designed with static sparsity patterns in mind. The solution procedure itself is divided into two steps: "symbolic analysis," where the sparsity pattern of the matrix representing the system of linear equations is analyzed, and "numerical computation," which uses the symbolic analysis information to efficiently compute the solution. Often, a single symbolic analysis requires more computational resources than a single numerical computation (see Section 5). In cases of fixed sparsity, repeated solves can be accelerated by caching and reusing the results of a single symbolic computation. However, in applications with changing sparsity patterns, this optimization is unavailable.

In this work, we focus on optimizing the performance of Cholesky solvers in applications with temporally-coherent, local changes in sparsity pattern, such as those observed in contact simulation with elasticity [Li et al. 2020] and geometric operations with remeshing including mapping [Schmidt et al. 2023] and morphology [Sellán et al. 2020] in their computational pipeline. In such applications, the overhead of repeated symbolic analyses becomes the bottleneck of linear solve costs. For example, in our evaluation of Incremental Potential Contact (IPC) [Li et al. 2020], we see that symbolic analysis accounts for up to 78% of the total runtime of the Cholesky solver. Also, in our evaluation of patch remeshing pipeline (see Section 18, we observe that 82% of the total runtime of Cholesky solver is spend on symbolic analysis. In this work, we leverage temporal coherence in sparsity patterns, common in graphics workflows, to adaptively reuse prior symbolic analysis to accelerate sparse Cholesky solves.

Our main contribution is Parth, an adaptive and general algorithm that enables reuse in symbolic analysis across repeated Choleksy solves when changes in the sparsity pattern are localized and temporally consistent. Parth has two key objectives: (I) adaptive reuse focused on changing linear-system sparsity structure, and (II) general-purpose, practical integration across high-performance Cholesky solver packages and libraries. The first provides generality and portability across diverse applications, while the second enables speedups with the state-of-the-art reliable and performant Cholesky solvers.

We evaluate well over 150 thousand linear solves from challenging physics simulations and geometric processing problems with dynamic sparsity patterns. We first identify symbolic analysis as a primary bottleneck in these applications. Within symbolic analysis, we then pinpoint fill-reducing ordering as the bottleneck of the process. Symbolic analysis provides a permutation that minimizes fill-ins during Cholesky solve computation [Davis et al. 2016]. Consequently, this paper specifically focuses on a set of algorithms for adaptive reuse of fill-reducing ordering computations.

Integrating Parth into high-performance Cholesky solvers—Apple Accelerate [Inc. 2023], Intel MKL [Schenk et al. 2001], and CHOLMOD [Chen et al. 2008]—and evaluating them on our two benchmarks, IPC and Remeshing, we observe up to a 255× and 13× speedup in fill-reducing ordering performance, respectively. These improvements result in a 6.85× and 10.7× speedup in symbolic analysis performance. Consequently, these enhancements translate into up to a 2.95× and 5.89× speedup in the per-solve Cholesky computation, respectively (refer to Section 5 for more details). Additionally, we demonstrate that by adding the three lines of code required to integrate Parth into the Cholesky solve computational pipeline of our IPC benchmark (see Section 5.2), our most challenging simulation ("Arma Roller") achieves seven hours less computational time with only a 1.5× speedup in the total Cholesky solve runtime, without any side effects on numerical performance. Finally, recognizing that different graphical computational pipelines may exhibit varying behaviours in gradual reuse, we provide a per-solve analysis of Parth's effects. This analysis demonstrates different reuse scenarios, including local changes in single and multiple locations, as well as varying sizes of changes in both the number of rows/columns and the non-zero entries of the matrix. These insights allow practitioners to understand how Parth performs in their specific applications.

Our technical contributions enabling this reuse are: (I) A novel hierarchical graph decomposition algorithm that decomposes the graph dual of the input matrix—representing the sparsity pattern of the system of linear equations into multiple fine-grain sub-graphs. (II) To leverage the hierarchical characteristics of the decomposition, we propose a new algorithm that confines changes to fine- or coarse-grain sub-graphs, enabling reuse in static parts of the graph. (III) Finally, we provide an extensive evaluation of high-performance Cholesky solves, which, to the best of our knowledge, is not currently available, allowing practitioners to make informed choices for their linear solver baselines and computational pipelines.

In summary, we present Parth, a method that reuses symbolic information in the presence of dynamic sparsity patterns with temporal coherence across calls to the Cholesky solver. We evaluate Parth's performance across a wide range of different solve scenarios. We focus on three highest-performing Cholesky solvers—CHOLMOD [Chen et al. 2008], the recently developed Apple Accelerate sparse kernels [Inc. 2023], and Intel MKL [Schenk et al. 2001] [1] We hope that

[1]We choose these by also comparing them with alternate available solvers, Sympiler [Cheshmi et al. 2017], Parsy [Cheshmi et al. 2018a], and Eigen [Liu et al. 2021]. See Appendix G.

our extensive analysis not only provides insight into how Parth improves performance for these tools across the board but also into the specifics of each of these state-of-the-art Cholesky solvers' performance. Together these will help practitioners to make informed application-specific decisions for which framework to use based on our presented comprehensive quantitative comparison.

2 RELATED WORK

Enhancing the performance of Cholesky factorizations and Sparse Triangular Solves (SpTRSV), remain a significant focus in computational mathematics and computer graphics due to their critical roles in simulation and optimization problems.

Exploiting Dense Computation: Early optimizations center on exploiting dense computations within sparse factorizations.Liu [1990] utilize the elimination tree to create supernode—groups of consecutive rows/columns sharing the same sparsity pattern. These supernodes are then factorized using dense BLAS[Dongarra et al. 1990] kernels, enhancing computational efficiency by leveraging optimized dense linear algebra routines. Subsequent methods, such as CHOLMOD [Chen et al. 2008], relax strict supernode constraints, allowing for better trade-offs by utilizing dense computation more effectively. These come at the expense of more redundant computation by increasing the size of the supernodes even when some rows/columns do not have a matching pattern.

Parallelism and Scheduling Algorithms: While initial focus was primarily on finding dense computations within the chaos of sparse computations, later work focuses on further enabling parallelism across the computation of these dense blocks. This led to the introduction of advanced scheduling algorithms that balance load across parallel units, optimize memory usage through data reuse, and reduce synchronization overhead. MKL PARDISO [Schenk et al. 2001] improves parallelism by efficiently distributing tasks across multiple cores and handling thread-level parallelism. Load-Balancing Coarsening (LBC)[Cheshmi et al. 2018b] enhances memory usage by improving data locality and minimizing cache misses during computation on CPUs, as the parallelism of Cholesky factorization is limited by the sparsity pattern, and memory reuse can provide some compensation for the lack of parallelism. Additionally, the CHOLMOD GPU scheduler[Rennich et al. 2016] targets GPU hardware to achieve significant speedups in numerical computation, achieving up to a 2x speedup compared to CPU implementations.

Code Generation: Recent approaches focus on providing optimized code by inspecting the sparsity pattern during symbolic analysis to improve numerical computation efficiency. Sympiler [Cheshmi et al. 2017] analyzes sparsity patterns to generate specialized, pattern-specific code that optimizes memory access and computational efficiency. Similarly, Cheshmi et al. [2023] automates the generation of high-performance code for sparse applications by fusing kernels used in linear solvers. However, this analysis comes with a high overhead of inspection, which further increases symbolic analysis overhead and must be redone if the sparsity pattern changes.

Reuse of Numerical Factorization: Efforts to reuse numerical factorization in linear solve computations aim to reduce the costs associated with recomputing factors for systems with numerical temporal coherence. In Davis and Hager [2005], a method for reusing factors via low-rank updates is introduced. Li et al. [2021] extend this work to support low-rank updates in the presence of dynamic sparsity patterns. However, as they mention in their paper, this method does not perform well when multiple local changes occur in the sparsity pattern, and they can not reuse symbolic analysis in such cases. Another body of work on reusing factorization computation is proposed in [Herholz and Alexa 2018; Herholz and Sorkine-Hornung 2020], where a prior factor is reused across calls to the linear solver by only recomputing the affected supernodes. However, these methods often introduce approximation errors, limiting their applicability in contexts where the precision of Cholesky solvers is essential. Furthermore, they can not be applied to applications with dynamic sparsity patterns, as the elimination tree structure relies on changes across calls to Cholesky solvers, leading to the reconstruction of supernodes. NASOQ [Cheshmi et al. 2020] provides a constraint-based QP solver that reuses factorization computation across calls to a constraint-based QP solver, as adding and removing these constraints requires a factorization. To handle changes in the sparsity pattern due to changes in constraints, it performs a full symbolic analysis with all possible constraints added and uses a subset of the symbolic analysis when the constraints are added or removed from the KKT matrix. However, this approach assumes prior knowledge of all potential constraints and can not handle new sparsity patterns not known in advance.

In contrast to existing methods, our work focuses on reusing symbolic analysis instead of numerical factorization to handle the overhead caused by dynamic sparsity patterns in Cholesky factorization. Additionally, for these applications, we assume the local occurrence of multiple changes in different parts of the sparse matrix without knowing where they might happen in advance or how many changes are occurring in those local regions. By developing algorithms that identify and reuse unchanged portions of the symbolic analysis across iterations, we significantly reduce the overhead associated with the symbolic phase. This approach can handle multiple local changes effectively while providing high-quality fill-reducing ordering.

3 BACKGROUND

Solving the fill-reducing ordering problem is NP-hard [Yannakakis 1981]. However, commercial Cholesky linear solvers apply well-tested heuristic-based methods such as METIS [Karypis and Kumar 1997] and AMD [Amestoy et al. 2004] that provide high-quality fill-reducing orderings. These algorithms incorporate randomness in their routines, resulting in multiple fill-reducing orderings for a single input, highlighting the existence of multiple possible high-quality orderings for a given linear system. Parth introduces a method that finds a high-quality fill-reducing ordering by reusing computations from previous calls to the ordering module. As an example, in Figure 2, two systems of linear equations with gradual changes can produce multiple high-quality permutation vectors, where one of these ordering vectors has high similarity to the previous call indicating the existence of a solution while reusing the computation from the previous call.

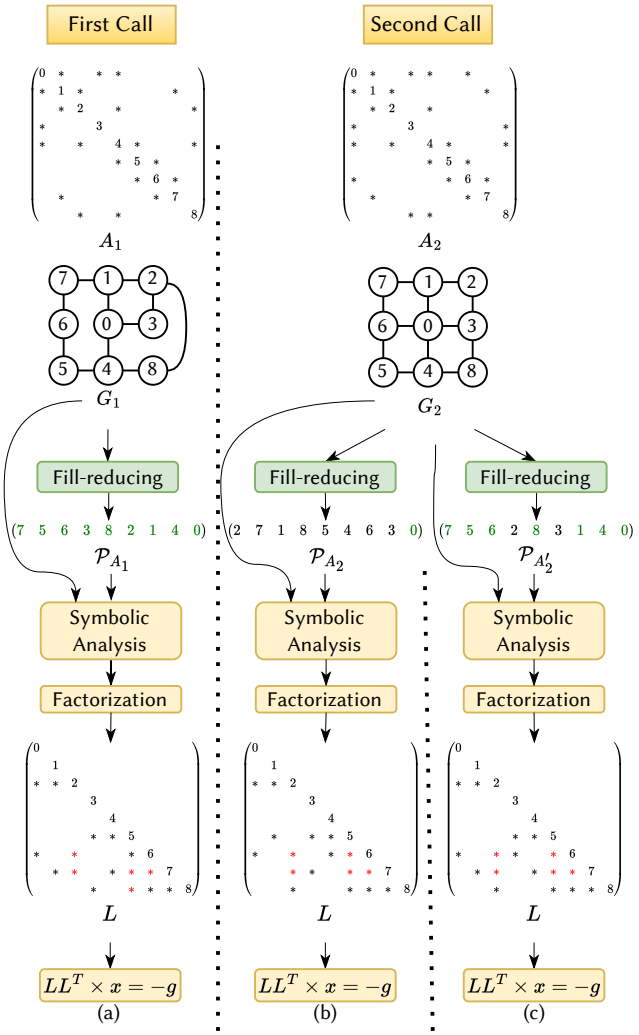


Fig. 2. **Cholesky solver computational pipeline for two calls.** The difference between columns 0,2, and 3 between the first and second calls requires the computation of a fill-reducing vector in both calls. As shown in sub-figures (b) and (c) which process the same matrix, one can see that two permutation vectors produce identical fill-in counts. However, Figure 2(c) displays an ordering vector, $\mathcal{P}_{A_2'}$, that is more similar to $\mathcal{P}_{A_1}$, as shown by the values coloured green, especially when contrasted with Figure 2(b). This similarity shows the potential for computational reuse of fill-reducing orderings while providing a high-quality permutation vector.

Parth achieves this by operating on the graph dual G of the system of linear equations A, rather than directly analyzing application-specific properties. In particular, for a system of linear equations $Ax = b$, where A is symmetric, the matrix can be viewed as an adjacency matrix. In this view, each row/column corresponds to a node in graph G, and non-zero entries define the edges between nodes. This approach makes Parth general, as A could represent, for example, a discrete Laplacian operator or the Hessian of an energy function constructed for a Newton iteration solve. Additionally, fill-reducing ordering algorithms also use the graph dual of the input matrix A, aligning Parth's computational pipeline with that of a general high-performance Cholesky solver. An example of graph duals G_1 and G_2 for A_1 and A_2 matrices is shown in Figure 2. Note that the graph dual G does not consider diagonal entries of A.

While the input Parth considers is general, the heuristics applied by Parth are based on an abstract connection between the graph and an underlying application's degrees of freedom (DOF) and evaluation stencils. For instance, many geometric processing applications utilize cotangent Laplace-Beltrami operators where the nodes in the graph can represent DOF, and the edges correspond to the stencils formed by elements in the mesh. For Hessian formed by the Incremental Potential Contact (IPC) [Li et al. 2020] method, each set of three consecutive rows in Hessian A $\{3i, 3i+1, 3i+2\}$ represent the properties of the x, y and z directions of a single DOF. This abstract connection - linking the mesh to the graph dual of A - provides us with a geometric connection that is considered in Parth algorithms. Consequently, while our approach may apply to other scenarios, as we aim to be as general as possible, this work focuses on mesh-based computations where the relationship between the graph and the underlying application, as described, is key.

The following section explains Parth's modules, which efficiently produce high-quality fill-reducing ordering vectors by reusing computation, requiring only the addition of three lines of code to add it to pre-existing Cholesky linear solver pipelines.

4 PARTH FRAMEWORK

Figure 3 illustrates Parth's integration into high-performance Cholesky solver libraries for solving the two calls as shown in Figure 2. As shown, Parth replaces the fill-reducing module of high-performance Cholesky solvers and provides its own permutation vector to these tools. In the following, we explain the internal modules of Parth, followed by its input and output to present the computational pipeline that enables the reuse of fill-reducing ordering computation.

Parth's Modules: Figure 3 shows the three underlying modules of Parth. The **Hierarchical Graph Decomposition** (HGD) algorithm decomposes the dual graph G of the system of linear equations A into multiple smaller sub-graphs (fine-grain) that can be coarsened to form larger sub-graphs (coarse-grain). Furthermore, the decomposition enables the creation and combination of local permutation vectors per sub-graph. A **Synchronizer** module detects and integrates sparsity pattern changes into the created sub-graphs, thereby localizing the effects of these changes by confining them to specific coarse- or fine-grain sub-graphs. The **Assembler** unit then reuses the local permutation vectors in unchanged sub-graphs and updates those that are changed, finally assembling all this information into a single permutation vector, $\mathcal{P}_A$.

Input: As shown in Figure 3, Parth's input is matrix A. The graph dual, as shown in Figure 2, simply uses A as an adjacency matrix (without its diagonal entries). If the user specifies that, for example, every three consecutive rows in the form of $\{3i, 3i+1, 3i+2\}$ represent the properties of a single node and are fully coupled, Parth can coarsen the graph and use that for analysis, as these terms are fully coupled, and this compression does not affect the quality of

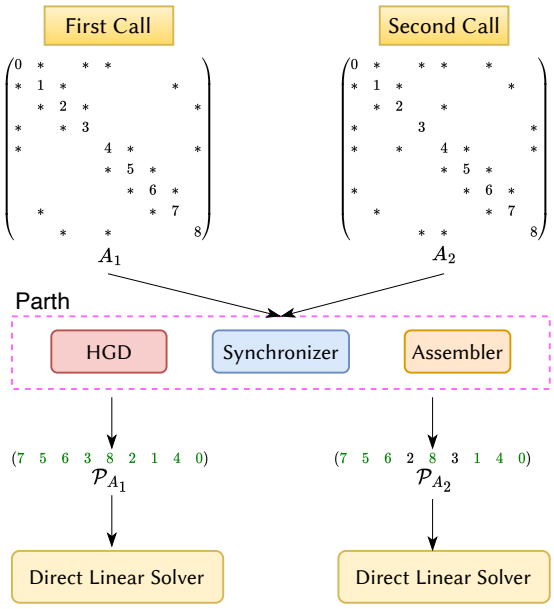


Fig. 3. **Parth's integration into high-performance Cholesky solvers for two calls with dynamic sparsity patterns.** The sparsity pattern shown here is the same as in Figure 2. As illustrated, in the second call, Parth provides fill-reducing vectors with small changes by reusing computations from the first call and feeding them to the Cholesky linear solvers through their provided API. Note that current Cholesky linear solvers have no mechanism for reusing computations unless the sparsity pattern remains constant across calls. A detailed explanation of the HGD, Synchronizer, and Assembler modules, and how they interact, is provided in the rest of this section.

fill-reducing ordering. This process is explained in more detail in Section 4.3.

Output: Parth's output is a permutation vector applicable to matrix A. The permutation vector generated in the first call shares seven identical entries with the permutation vector in the second call. This is due to Parth's mechanism, which reuses the permutation vector information from the first call. By updating only the necessary local permutation vectors and integrating the updated information into $\mathcal{P}_{A_1}$, the permutation vector $\mathcal{P}_{A_2}$ has far more similarity to previous vector, $\mathcal{P}_{A_1}$ than a conventional fill-reducing algorithm. By reusing the information, Parth enhances the performance of fill-reducing ordering computation by up to 255x speedup per frame in our IPC benchmark.

This performance can be easily provided to practitioners, as many commercial Cholesky solvers such as MKL, CHOLMOD, or Accelerate accept user-provided fill-reducing orderings. When such an interface exists, the fast fill-reducing orderings computed by Parth can be directly supplied, increasing the performance of these off-the-shelf tools without any modification to the underlying source code.

Algorithm 1 Hierarchical Graph Decomposition (*HGD*)

Global: $\mathcal{B}$
Input: G_{sub}, l, i, max_level
1: **if** $l \neq max\_level$ **then**
2: $g_l, g_r, g_s \leftarrow computeMinSeparator(G_{sub})$
3: $\mathcal{B}[i].nodes \leftarrow g_s.nodes$
/*The left sub-graph in HGD procedure*/
4: $HGD(g_l, l+1, 2\times i+1, max\_level)$
/*The right sub-graph in HGD procedure*/
5: $HGD(g_r, l+1, 2\times i+2, max\_level)$
6: **else**
7: $\mathcal{B}[i].nodes \leftarrow G_{sub}.nodes$
8: **end if**

4.1 Parth: Hierarchical Graph Decomposition

The Hierarchical Graph Decomposition (HGD) algorithm is designed to encapsulate changes within sub-graphs. Changes in the sparsity pattern due to, for example, contact during a simulation or remeshing during geometry processing are reflected in changes in the graph dual G. The Hierarchical Graph Decomposition module (HGD) attempts to encapsulate those changes within local sub-graphs to facilitate local updates to the fill-reducing ordering.

To accomplish this, HGD's decomposition uses a separator set computation which is also used in nested dissection algorithms. As a variant of this algorithm is employed in METIS, a well-known fill-reducing ordering algorithm, Parth can reuse this computation as part of the fill-reducing ordering process. Moreover, HGD encodes the decomposed graph information into a binary tree. This representation enables Parth to provide both fine and coarse-grained subgraphs simultaneously, allowing for encompassing changes in arbitrary places with negligible overhead when compared to the computation of fill-reducing ordering itself. The following details the HGD process.

As described in Algorithm 1, Parth uses the HGD algorithm to iteratively construct a binary tree data structure, $\mathcal{B}$, stored as an array. The total number of nodes in the binary tree is computed using the maximum level in the binary tree. A level is defined as the distance between the node and the root. Thus, HGD uses the max_level variable as the termination condition (line 1) for the recursion. Each call to the HGD algorithm results in a new binary tree node, represented as $\mathcal{B}[i]$. Each binary tree node corresponds to a sub-graph within the graph-dual of A. These subsets are determined by the recursive use of the function $computeMinSeparator$ (line 2). Note that this function divides the graph into three distinct sub-graph: a separator set, g_s, and two other sub-graphs, g_l and g_r. The separator set acts as a minimal sub-graph that, when removed, dissects the remaining graph into two nearly equal parts, ensuring a balance in size between g_l and g_r.

To provide intuition on how these sets are connected to the underlying mesh, as explained in Section 3, for example, for the discrete Laplacian operator, the graph is identical to the mesh; as a result, we can directly show these sets on the mesh as shown in Figure 4, where 4 twisted rods are divided into two approximately equal size

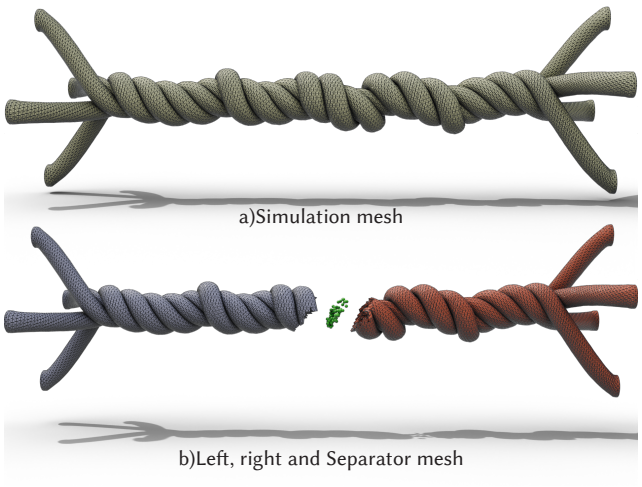


Fig. 4. **Example of a separator set.** In this"Rods Twist" simulation mesh, used in our IPC benchmark, by considering every three nodes representing x,y,z of a DOF as one node, we can create a mapping between graph and the mesh. As a result, the left sub-graph (purple), right sub-graph (red), and the separator set (green) can be identified in the simulation mesh as shown. Note that the separator consists of a small number of nodes when compared to the left and right sub-meshes.

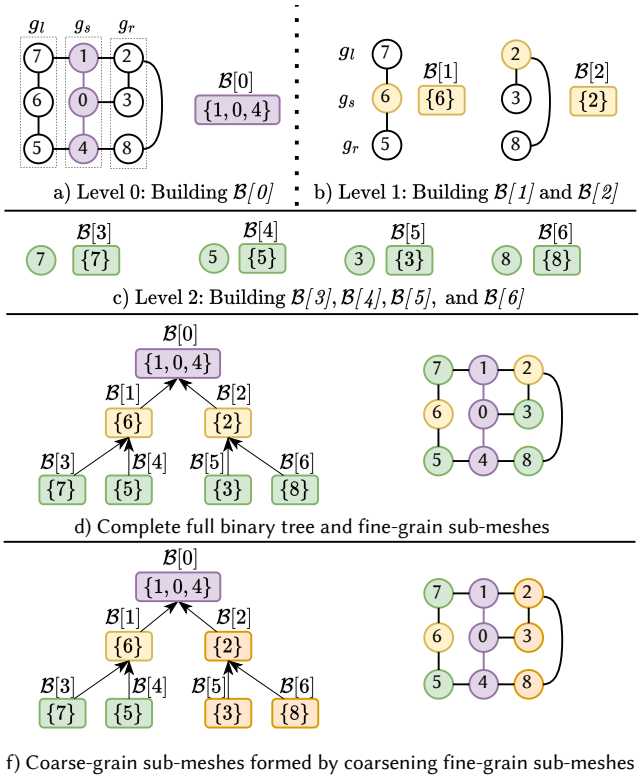


Fig. 5. **Example of HGD evaluation on "First Call" of Figure 3 simulation mesh.** In this figure, the HGD algorithm operates across three levels as depicted in Figures 5(a-c). In Figure 5(a), the root node of the binary tree, $\mathcal{B}[0]$, is created, forming level 0 of the binary tree. The algorithm assigns the set $\{0, 1, 4\}$ to g_s, the set $\{5, 6, 7\}$ to g_l, and the set $\{2, 3, 8\}$ to g_r. Figures 5(b) and (c) illustrate the recursive expansion of the binary tree, generating all nodes at levels 1 and 2. Figure 5(d) displays the resulting fine-grain sub-meshes. Ultimately, Figure 5(f) reveals the coarse-grain sub-mesh, which is derived from merging the separator $\mathcal{B}[2]$ with its ancestor nodes $\mathcal{B}[5]$ and $\mathcal{B}[6]$.

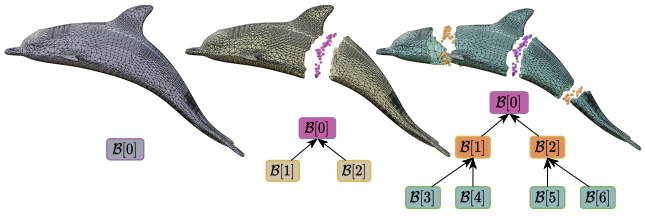
sub-meshes using the separator. Although finding the optimal minimum separator set is NP-hard, tools such as METIS [Karypis and Kumar 1997] or Scotch [Pellegrini 2009] offer high-quality heuristic solutions. As METIS is widely used in high-performance Cholesky solvers, we use METIS for our evaluation *computeMinSeparator*. However, integrating Scotch is also straightforward in the Parth framework.

To provide intuition on how these sets are connected to the underlying mesh, as explained in Section 3, for example, for the discrete Laplacian operator, the graph is identical to the mesh; as a result, we can directly show these sets on the mesh as shown in Figure 4, where 4 twisted rods are divided into two approximately equal size sub-meshes using the separator. Although finding the optimal minimum separator set is NP-hard, tools such as METIS [Karypis and Kumar 1997] or Scotch [Pellegrini 2009] offer high-quality heuristic solutions. As METIS is widely used in high-performance Cholesky solvers, we use METIS for our evaluation *computeMinSeparator*. However, integrating Scotch is also straightforward in the Parth framework.

As an example, Figure 5 shows the HGD evaluation on the G_1 from"First Call" in Figure 3. Figure 5(a-c) presents the HGD process at three levels. Initially, the root node of the binary tree is created, shown as $\mathcal{B}[0]$. The decomposition then recursively advances to levels 1 and 2, resulting in the creation of 2 and 4 additional nodes within $\mathcal{B}$. Each node in $\mathcal{B}$ represents the smallest sub-graph in our decomposition. The final full binary tree is represented in Figure 5(d). Note that each leaf of this binary tree represents a separated, approximately equal-sized sub-graph, and intermediate nodes in the binary tree are the separators which also form part of the sub-graphs. This methodology effectively allows for coarsening sub-graphs by merging each sibling with their corresponding separator. Note that the coarsening can continue until all the sub-graphs are coarsened into the root, forming the whole G. This shows the hierarchical nature of this decomposition.

Note that traversing this tree and applying decomposition and coarsening are common tasks for Parth. As a result, they should have a low overhead. In the following, we explain why the overhead of these two tasks is small in practice.

Sub-graph Decomposition Overhead: The HGD process leverages the nested-dissection approach [Khaira et al. 1992] to create the sub-graphs. The information computed for finding these sub-graphs, such as the separator sets and their corresponding descendants in $\mathcal{B}$, can be reused as part of the fill-reducing ordering algorithm, effectively hiding the HGD cost within the fill-reducing ordering computation, as shown in Section 5. For an example of how this



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

Fig. 6. **HGD evaluation on "Dolphin" mesh for a discrete Laplacian operator.** In here the graph G is identical to the mesh. The figure shows the HGD evaluation for max_level={0,1,2}. Note that how merging fine-grain sub-graphs $\mathcal{B}[1]$, $\mathcal{B}[3]$, $\mathcal{B}[4]$ in Figure 6(c) can result in coarse sub-graphs $\mathcal{B}[1]$ in Figure 6(b).

information is reused for fill-reducing ordering computation, see Appendix A.

Binary Tree Usage Overhead: In practice, we use a small $\mathcal{B}$, regardless of the mesh resolution, which results in low overhead when traversing it. To elaborate, first note that the *max_level* parameter defines the size of the binary tree. For example, $max\_level = 1$ indicates a binary tree of 3 nodes, including a root and two left and right nodes. For all of our test cases, we use $max\_level = 7$. This tree depth allows for a reuse granularity of 1%. Recall that applying a nested dissection algorithm divides a graph into two almost equally sized sub-graphs and a small separator. Without considering the separator size, each of these sub-graphs has approximately 50% of the size of the input graph. Applying nested dissection to these two sub-graphs creates four new sub-graphs with two new separators. As a result, each of these sub-graphs has 25% of the size of the graph. This process is equivalent to creating a $\mathcal{B}$ with $max\_level = 2$. Choosing $max\_level = 7$ results in 128 sub-graphs and 127 separators. Assuming small sizes for the separators, each of the sub-graphs represents less than 1% of the input graph. Since the granularity of reuse is defined by these fine-grain sub-graphs (the reuse capability is discretized by the fine-grain sub-graphs), our reuse can exceed 99%, which is sufficient in practice.

Finally, note that coarsening the sub-graphs using this binary tree model does not actually involve coarsening the nodes in $\mathcal{B}$. Parth's approach only considers a sub-tree in $\mathcal{B}$ as a coarsened sub-graph, meaning a separator with its left and right sub-graphs is treated as a single coarse sub-graph. For instance, in Figure 5(f), the coarse sub-graph 2, 3, 8 is formed by coarsening $\mathcal{B}[3]$ and $\mathcal{B}[6]$ with their shared separator, $\mathcal{B}[2]$. This process does not involve creating entirely new sub-graphs, as it represents coarser ones using their mutual separator from all the fine-grain sub-graphs. A real-world example is shown in Figure 5, where it illustrates how the coarse-grain sub-graphs seen in Figure 5(b) (coloured yellow) are formed by coarsening fine-grain sub-graphs with their common separators in Figure 5(c) (coloured blue). This illustration demonstrates how $\mathcal{B}$ maintains both fine- and coarse-grained perspectives concurrently.

4.2 Parth: Synchronizer

The Synchronizer module in Parth synchronizes the information between G and $\mathcal{B}$ across calls to the linear solver by identifying

Algorithm 2 Parth: Synchronizer

Global: $\mathcal{B}$
Input: G, G_{new}, *map max_level, Aggressive*
Output: $C_{\mathcal{B}}$

1: $setTrue(C_{\mathcal{B}})$
2: $\mathcal{B} \leftarrow NodeChangeSynchronizer(map)$
3: $E_G \leftarrow ComputeChanges(G, G_{new}, map)$
/* Fine- and coarse-grain dirty sub-mesh*/
4: $D_F \leftarrow \{\}, D_C \leftarrow \{\}$
5: $E_{\mathcal{B}} \leftarrow MapTo\mathcal{B}(E_G)$
/*Detect dirty sub-graphs in $\mathcal{B}$*/
6: $D_F, D_C \leftarrow DirtySubGraphDetection(E_{\mathcal{B}})$
/*Filtering redundant work*/
7: FilterRedundantSubGraphs(D_F, D_S)
/*Mark the fine-grain sub-graphs for fill-reducing ordering*/
8: $C_{\mathcal{B}} \leftarrow MarkAndDecomposeSubGraphs(D_C, D_F)$

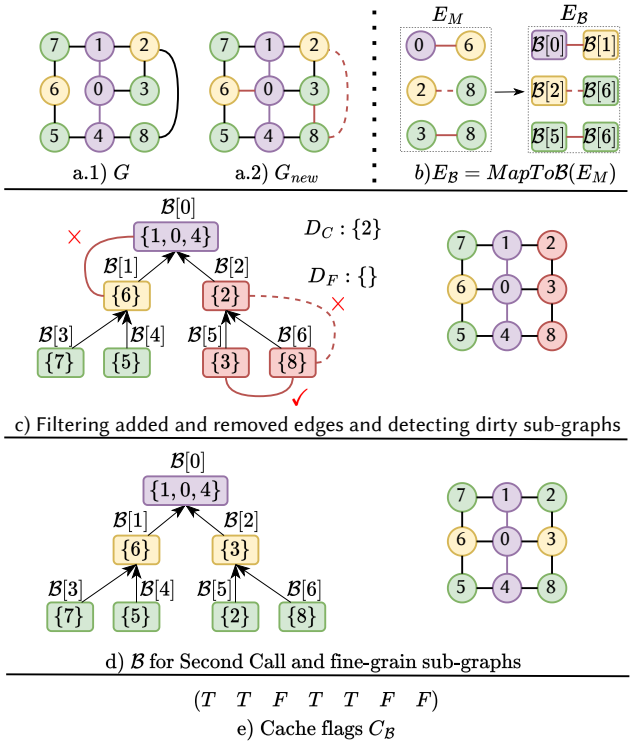


Fig. 7. **Example of the Synchronizer procedure.** In Figure 7(a,b), changes are detected and converted into changes across their corresponding fine-grain sub-meshes. In Figure 7(c), the change that violates the separator condition (between $\mathcal{B}[5]$ and $\mathcal{B}[6]$) is considered, and other changes are disregarded. The lowest common descendant of $\mathcal{B}[5]$ and $\mathcal{B}[6]$ is $\mathcal{B}[2]$, where node 2 in G_{new} no longer acts as a separator. The coarse-grain sub-graph encompassing the change is constructed by merging the sub-graphs in $\mathcal{B}[2]$, $\mathcal{B}[5]$, and $\mathcal{B}[6]$. In Figure 7(d), the coarse-grain sub-graph represented by $\mathcal{B}[2]$ and its ancestors are re-decomposed. As a result, a new separator is chosen (node 3 in G_{new}). Figure 7(e) displays the $C_{\mathcal{B}}$ array where 'T' (True) and 'F' (False) indicate which sub-meshes are intact and which have changed, respectively.

changes in G and encapsulating them within the set of sub-graphs in $\mathcal{B}$. If necessary, it re-decomposes the sub-graphs affected by these changes, as their nodes' connectivity may now differ. This allows Parth to preserve fill-reducing ordering information in sub-graphs that remain unchanged, enabling the reuse of this information. The Synchronizer achieves this through the five-step procedure outlined in Algorithm 2, explained as follows:

Step 1: Synchronizing the Added or Removed DOFs: The Synchronizer algorithm begins by analyzing changes in the number of nodes in the graph G (line 2). If the nodes differ, it then examines the map input. The map array simply maps the index of nodes in the current call to the index of the same nodes in the previous call. For a detailed explanation of how Parth uses the map array to synchronize the added or removed nodes into $\mathcal{B}$, see Appendix B. In our experience, remeshers either create the map array explicitly in their underlying process or straightforwardly allow for its creation, as it is required for generating the faces and vertices metadata for (re)defining a mesh. Therefore, we hope this requirement is not too restrictive for other applications that could benefit from Parth.

Step 2: Detecting Added or Removed Edges: The Synchronizer now identifies added or removed edges from the graph by comparing the current graph (G_{new}) with the previous one (G) (line 3). Note that for an added DOF, all the edges are considered new, and a removed DOF's edges are not considered for computing the changed edges set E_G. After this step, edges in E_G are mapped to changes in $\mathcal{B}$, indicated by $E_{\mathcal{B}}$. To clarify, Figure 7(a) illustrates the detection of changes between the graph duals shown in Figure 3. The difference between the two sub-graphs includes the addition of edges <0,6>and <3,8>and the deletion of edge <2,8>. Additionally, these modifications are mapped to the connectivity changes between fine-grain sub-meshes represented by <$\mathcal{B}[0]$, $\mathcal{B}[1]$>, <$\mathcal{B}[2]$, $\mathcal{B}[6]$>, and <$\mathcal{B}[5]$, $\mathcal{B}[6]$>, as visualized in Figure 7(b).

Step 3: Detecting Dirty Sub-graphs: The Synchronizer module utilizes $E_{\mathcal{B}}$ to identify sub-graphs affected by these changes. These affected sub-graphs, called "dirty sub-graphs," no longer possess valid information due to the changes. There are two categories of dirty sub-graphs: D_F, which are fine-grain dirty sub-graphs, and D_C, which are coarse-grain dirty sub-graphs (line 4). The distinction is made because fine-grain sub-graphs do not require re-decomposition; only their fill-reducing ordering information needs to be updated. In contrast, coarse-grain sub-graphs may experience a violation of the separator's characteristics due to connectivity changes. That is, two sub-graphs that were supposed to be completely separated by a separator set are now connected via some nodes in the graph. Consequently, the Synchronizer must apply the HGD algorithm to re-decompose the coarse-grain regions and update the $\mathcal{B}$ information as needed. For a detailed explanation of line 6 in the algorithm, see Appendix C.

To clarify the effect of the mentioned procedure, Figure 7(c) demonstrates the process of computing D_F and D_C using $E_{\mathcal{B}}$. The two changed edges <$\mathcal{B}[0]$, $\mathcal{B}[1]$>and <$\mathcal{B}[2]$, $\mathcal{B}[6]$>are disregarded because they do not disrupt the connection between the sub-graphs within the binary tree $\mathcal{B}$. For example, the change involving <$\mathcal{B}[0]$, $\mathcal{B}[1]$>is dismissed as the link between a separator and its adjacent left sub-graph does not violate the validity of the separator, given that $\mathcal{B}[0]$ separates $\mathcal{B}[1]$ from sub-graphs $\mathcal{B}[2]$, $\mathcal{B}[5]$, and $\mathcal{B}[6]$.

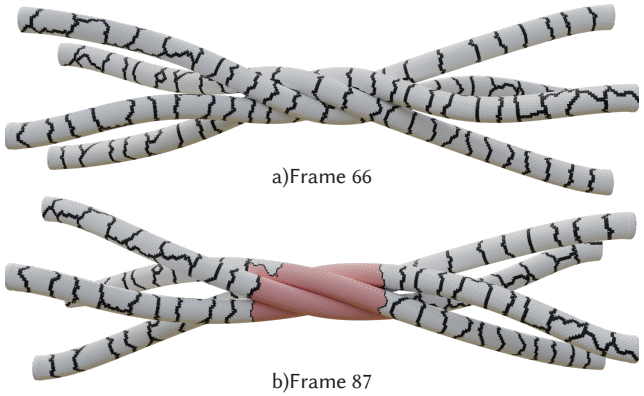


Fig. 8. **Synchronizer confinement of contacts in the "Rods Twist" simulation.** For this figure, multiple frames are skipped, and Parth is applied to the dual graph of the Hessian for the first Newton-solve iteration of each of the 66th and 87th frames. Since four rods are twisting, multiple contacts occur on each of the rods. To map the sub-graphs to the sub-meshes, we used the same procedure as described in Figure 4. Using `max_level=7`, the black lines are the separators, and the white regions are sub-meshes represented by the leaves of $\mathcal{B}$. Note how the Synchronizer successfully coarsens multiple small sub-meshes, shown in red, and confines 1612 changes in the simulation mesh even when the simulation's step size is 21 frames.

Conversely, the change involving <$\mathcal{B}[5]$, $\mathcal{B}[6]$>violates the separator's role of $\mathcal{B}[2]$, since the sub-graph $\mathcal{B}[5]$ now connects to sub-graph $\mathcal{B}[6]$, resulting from the new connections between node 3 and node 8 in the graph. Consequently, the coarse sub-graph, annotated as $\mathcal{B}[2]$ and comprising simulation graph nodes $\{2, 3, 8\}$, is added to D_C as it encapsulates the change <3,8>.

Step 4: Filtering Sub-graphs: During the formation of D_F and D_C, each change is assessed independently from the others. As a result, some coarse-grained sub-graphs can encompass other fine- and coarse-grained sub-graphs. These smaller sub-graphs can subsequently be removed from D_F and D_C, as they will be re-decomposed when the larger, encompassing sub-graph is re-decomposed. After this stage, the coarse sub-graphs requiring re-decomposition are fully identified. For a real-world example, refer to Figure 8, where the contacts are restricted to a set of coarse sub-graphs (subsequently sub-meshes in the simulation mesh of IPC), colored in red, at the end of Step 4 (line 6 of Algorithm 2).

Step 5: Re-decomposing Sub-graphs: Finally, after the creation and filtering of D_F and D_C, the decomposition information in $\mathcal{B}$ must be updated accordingly. For fine-grain sub-graphs in D_F, we only need to mark them in the $C_{\mathcal{B}}$ array so that the Assembler module (Section 4.3) updates the fill-reducing ordering information of these sub-graphs, as only the connectivity between the nodes has changed. For coarse-grain sub-graphs in D_C, the Synchronizer first re-decomposes the sub-graphs using the HGD algorithm (Section 4.1). The newly formed fine-grain sub-graphs are then marked in $C_{\mathcal{B}}$ for the computation of fill-reducing ordering. For a detailed explanation of the algorithm, see Appendix D.

As an illustration, on the right side of Figure 7(c), the Synchronizer initially extracts the coarse sub-graph, colored in red (nodes

2, 3, 8). Given that the size of $\mathcal{B}$ is constant, the Synchronizer is only required to substitute the invalid sub-graphs with valid ones. To achieve this, the process first identifies the sub-tree within $\mathcal{B}$ that contains all the invalid sub-graphs (illustrated in red on the left side of Figure 7(c)). Next, it invokes the HGD algorithm on the sub-graph colored in red (Figure 7(c)-right) to produce a valid decomposition, which is then visualized as a new sub-tree. This new sub-tree is used to replace the invalidated one, resulting in a completely valid binary tree $\mathcal{B}$, as depicted in Figure 7(d). It is important to note that the sub-graph $\mathcal{B}[2]$ has been updated from the set 2 to 3, which now effectively isolates sub-graph $\mathcal{B}[5] = 2$ from sub-graph $\mathcal{B}[6] = 8$.

After updating the hierarchical graph decomposition, the new sub-graphs need the recomputation of fill-reducing ordering as the node structures they are representing have now changed. These dirty fine-grain sub-graphs are flagged within the $C_{\mathcal{B}}$ array. For instance, Figure 7(e) indicates that the sub-graphs $\mathcal{B}[2]$, $\mathcal{B}[5]$, and $\mathcal{B}[6]$ are new and thus require updated fill-reducing information. As a result, both the $C_{\mathcal{B}}$ array and the binary tree $\mathcal{B}$ will be fed to the *Assembler* module for new fill-reducing ordering information.

As a final note, one of the drawbacks of this procedure is that a single edge connecting two sub-graphs with a common separator $\mathcal{B}[0]$ will result in the reuse of zero, as the coarse-grain sub-graph will encompass the whole graph. To alleviate this, we created a heuristic, explained in the Appendix D. The heuristic moves one of the nodes that form the problematic edge to the corresponding separator. That is, the sub-graph forming that separator will expand if this happens, allowing Parth to achieve high reuse even in these scenarios.

4.3 Parth: Assembler

The *Assembler* module generates the permutation vector $\mathcal{P}_A$ by reusing computation across calls to Parth. Initially, the *Assembler* computes the required permutation vectors for each sub-graph represented by $\mathcal{B}$. It then assembles these local permutation vectors to form $\mathcal{P}_A$ which applies to the whole matrix A. Due to changes in sparsity pattern, if *Synchronizer* module marks some of the sub-graph "dirty" (refer to Section 4.2), the *Assembler* computes a new local permutation vector for these "dirty" sub-graphs. The values linked to these modified sub-graphs are then updated in $\mathcal{P}_A$, while the rest remain unchanged. This approach results in the computational reuse for all unchanged sub-graphs.

The *Assembler* procedure has three steps, as outlined in Algorithm 3. In Step 1, the procedure determines the placement of the local permutation vectors $\mathcal{P}_l$ within $\mathcal{P}_A$. Step 2 involves the calculation of the necessary $\mathcal{P}_l$ vectors which, using the previously computed positions from Step 1, are inserted into the $\mathcal{P}_G$. Finally, in Step 3, the graph permutation vector $\mathcal{P}_G$ is converted into $\mathcal{P}_A$ based on DIM value. All three steps are capable of reusing pre-computed data based on the array $C_{\mathcal{B}}$, which indicates the affected sub-meshes. Details of these three steps are further explained as follows:

Step 1: Placement Computation: To use separators and their corresponding left and right sub-graphs as fill-reducing ordering information, we follow the nested-dissection approach. In this approach, the permutation vector is arranged in the computation so that the separator computation is placed after the left and right sub-graph computations (see (Appendix A). Applying this order recursively is equal to placing each local permutation vector $\mathcal{P}_l$ related to each $\mathcal{B}$ node based on the post-order traversal of the tree. Consequently, the *Assembler* module creates the entire graph permutation vector $\mathcal{P}_G$. The *Assembler* executes this procedure in Lines 1-11 of Algorithm 3.

Algorithm 3 Parth: Assembler

```
Global: B, Order
Input: C_B, DIM
Output: P_A
1:  /*Initialization of C_B*/
2:  if C_B.empty() then
3:      setFalse(C_B)
4:      Order ← PostOrdering(B)
5:  end if
6:  /*Computing offset of C_B*/
7:  Offset ← 0
8:  for i in Order do
9:      B[i].Offset ← Offset
10:     Offset ← Offset + |B[i].nodes|
11: end for
12: /*Assembling fill-reducing ordering*/
13: for i in Order do
14:     if !C_B[i] then
15:         G_i ← getSubGraph(G, B[i].nodes)
16:         B[i].P_l ← FillReducingOrdering(G_i)
17:         start ← B[i].offset
18:         end ← start + |B[i].nodes|
19:         P_G[start : end] ← B[i].P_l
20:     end if
21: end for
    /*Convert mesh permutation into Hessian permutation*/
22: for (j = 0; j < |M.nodes|; j = j + 1) do
23:     for (d = 0; d < DIM; d = d + 1) do
24:         P_A[j * DIM + d] = P_G[j] * DIM + d
25:     end for
26: end for
```

Looking at Figure 9, the post-order is computed once in Figure 9(a), as the binary tree remains unchanged throughout the simulation. Subsequently, starting with $\mathcal{B}[3]$, the position of each $\mathcal{P}_l$ vector is identified, as illustrated in Figure 9(b.3), based on the "offset" variable. For example, since sub-graphs $\mathcal{B}[3]$ and $\mathcal{B}[4]$ contain only a single node each, the aggregated node count resulting from visiting these sub-graph positions the $\mathcal{P}_l$ associated with sub-graph $\mathcal{B}[1]$ at the starting position $offset = 2$ in $\mathcal{P}_G$. This procedure recurs in "Second Call" when the graph decomposition of three sub-graphs changes, leading to Figure 9(c.3). Since computing the offset is not computationally intensive, we omit the reuse procedure of this step to simplify the explanation of offset computation.

Step 2: $\mathcal{P}_l$ Computation and Mapping to $\mathcal{P}_G$: Once the placement of each $\mathcal{P}_l$ in $\mathcal{P}_G$ is determined, the *Assembler* uses the array $C_{\mathcal{B}}$ to compute and update $\mathcal{P}_l$s for each sub-graph specified in $\mathcal{B}$ (lines 14-23). For instance, in Figure 9 (b.2 and c.2), the local permutation vectors $\mathcal{P}_l$ are shown for each sub-graph. In Figure 9(c.2)

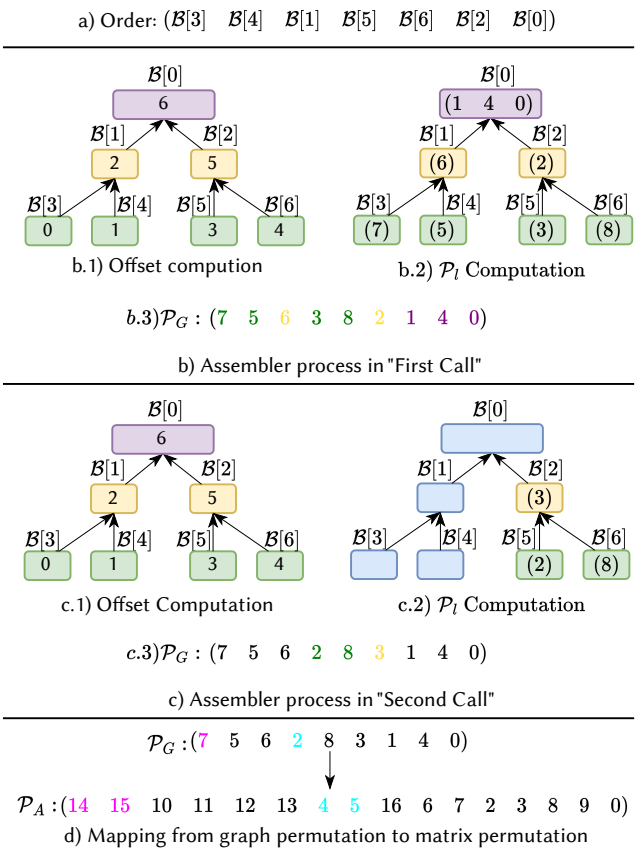


Fig. 9. **Example of the Assembler procedure.** Considering the two calls as shown in Figure 2, blue nodes in the $\mathcal{B}$ indicates computations that are skipped in "Second Call" due to reuse from "First Call". This computation is performed in high-performance Cholesky solvers that do not use Parth. Figure 9(a) displays the post-order traversal of $\mathcal{B}$, used in computing the position of each local permutation vector $\mathcal{P}_l$. During "First Call", the offsets, $\mathcal{P}_l$s, and the graph-based permutation vector $\mathcal{P}_G$ are computed. This initialization is presented in Figure 9(b.1), (b.2), and (b.3). As detailed in Figure 7, due to alterations in sub-graphs $\mathcal{B}[2]$, $\mathcal{B}[5]$, and $\mathcal{B}[6]$, the Assembler proceeds to update the offsets and $\mathcal{P}_l$ for these specific sub-graphs as shown in Figure 9(c.1) and (c.2), and subsequently updates a portion of $\mathcal{P}_G$ depicted in Figure 9(c.3). Finally, Figure 9(d) illustrates the mapping of $\mathcal{P}_G$ to the P_A for a 2D simulation.

that shows the local permutation computation for "Second Call", only three sub-graphs require new $\mathcal{P}_l$s, which allows Parth to reuse the $\mathcal{P}_l$ for the other four sub-graphs (coloured blue). By utilizing the "offset" variable calculated in Step 1 and the local permutation vectors, the Assembler constructs $\mathcal{P}_G$ by inserting the $\mathcal{P}_l$s into $\mathcal{P}_G$ (lines 15-23). Note that while Parth uses a separator set and nested-dissection approach, for $\mathcal{P}_l$ computation, any fill-reducing algorithm such as AMD [Amestoy et al. 2004], Scotch [Pellegrini 2009], and Morton Code ordering will work. Currently, Parth supports METIS and AMD for computing $\mathcal{P}_l$, but inserting new ones is straightforward.

Table 1. **Summary of benchmark problem statistics.** We build a linear solve benchmark by precomputing and storing, in consecutive order, the sequential linear systems and geometries generated by the iterations of Newton solves for time-stepping six of the most challenging deformable-body benchmark problems from Li et al. [Li et al. 2020]. Here we summarize statistics for each simulation sequence with ID numbers for each (used throughout) on the left, H_{rank} giving system dimension, and H_{nnz} and L_{nnz} respectively giving the average number of non-zeros per Hessian H and corresponding L-factors (via METIS's symbolic analysis). The #F represents a number of frames in each simulation while *Changes* gives the percent of iterations per sequence where sparsity patterns change from a prior iteration.

Example	H_{rank}	H_{nnz}, L_{nnz}	#F	Changes
(1) Dolphin Funnel	24K	500K, 11M	800	99.9%
(2) Ball Mesh Roller	23K	464K, 11M	1000	96%
(3) Mat On Board	120K	2.2M, 60M	200	98.9%
(4) Rods Twist	16K	3M, 54M	1600	98.73%
(5) Squeeze out	135K	2.8M, 72M	1500	96%
(6) Arma Roller	201K	4.8M, 344M	400	99.9%

Step 3: Mapping $\mathcal{P}_G$ into $\mathcal{P}_A$: For 1D simulation ($DIM = 1$), $\mathcal{P}_G$ is equal to $\mathcal{P}_A$. However, for different DIM, since Parth can compress the input G, or obtain the compressed form of G by merging DIM consecutive rows and columns in the form of $\{i * DIM + 1, \ldots, i * DIM + DIM - 1\}$, the output P_G is DIM times smaller than P_A. Parth employs a straightforward mapping, as illustrated in lines 24-28, which results in forming $\mathcal{P}_A$. Note that this does not reduce quality, as these DIM consecutive rows form a clique in G. Figure 9(d) displays an example of mapping the graph simulation into $\mathcal{P}_A$ when $DIM = 2$. Note that this mapping also utilizes reuse capability. However, to simplify the explanation, we've excluded the details of that implementation as these computations are fast compared to the $\mathcal{P}_l$ computation.

5 EVALUATION

We focus on benchmarking Parth across a range of sparsity pattern types, and variations in changes to these sparsity patterns. Specifically we evaluate Parth on both triangle and tetrahedral mesh geometries to introduce sparsity pattern variations. We evaluate solves from Incremental Potential Contact (IPC) [Li et al. 2020] simulations, which, due to contact, provide dynamically changing sparsity pattern with fixed numbers of rows and columns. Here we use the "IPC" keyword to indicate evaluation of these benchmark examples.

To evaluate applications with additionally dynamic numbers of rows/columns, we apply the popular Botsch remesher [Botsch and Kobbelt 2004] to provide local changes to the triangular meshes. We then re-solve with a discrete Laplacian [Jacobson et al. 2024] and observe reuse performance in these scenarios. This provides insight into how Parth performs in geometric processing tools when consecutive computations of a Cholesky linear solver on the whole geometry are required. Here we use the "Remeshing" keyword to indicate evaluation of these benchmark examples.

Table 2. **Breakdown of all Cholseky solves costs per library across all linear solves in our benchmark.** Here we summarize the timing (wall-clock seconds) and percent end-to-end Cholesky solver runtime per simulation breakdown per simulation sequence in our benchmark for all three state-of-the-art Cholesky solvers.

Tool	Step	Dolphin Funnel	Ball Mesh Roller	Mat On Board	Rods Twist	Squeeze out	Arma Roller
MKL	Symbolic time(s)	3261 (77.8%)	2623 (70.7%)	1551 (76.1%)	8635 (75.5%)	9098 (78.1%)	35357 (53.5%)
	Numeric time(s)	932 (22.2%)	1085 (29.3%)	485 (23.9%)	2800 (24.5%)	2543 (21.9%)	30703 (46.5%)
Accelerate	Symbolic time(s)	1773 (71.52%)	1465 (62.97%)	985 (70.50%)	5363 (67.54%)	5540 (71.28%)	25331 (32.81%)
	Numeric time(s)	706 (28.47%)	861 (37.03%)	412 (29.49%)	2578 (32.46%)	2231 (28.71%)	51871 (67.19%)
CHOLMOD	Symbolic time(s)	2857 (40.5%)	2353 (34.3%)	1600 (49.03%)	8552 (48.4%)	9012 (48.7%)	35021 (37.5%)
	Numeric time(s)	4200 (59.5%)	4511 (65.7%)	1663 (50.97%)	9098 (51.6%)	9483 (51.3%)	58282 (62.5%)

For each application, we first analyze the runtime bottleneck by determining how much time is spent on the numerical and symbolic phases of solves to illustrate the potential benefits of accelerating each part based on Amdahl's Law [Amdahl 1967]. Furthermore, we demonstrate Parth's performance benefits when integrated into the solvers and how it reduces the symbolic analysis runtime. Finally, we demonstrate how Parth's performance impacts downstream numerical performance, highlighting the quality of Parth's fill-reducing ordering. In the following section, we summarize the hardware and software setup for our evaluations.

5.1 Evaluation Setup

Parth is implemented with C++. The separator computation uses METIS. For the local permutation vector, we allow the use of both AMD [Amestoy et al. 2004] and METIS. However, all of these libraries can be easily replaced as Parth's implementation does not rely on the underlying implementation of these ordering algorithms. For evaluation, we integrate Parth with, to our knowledge, the three most popular, highest-performing, robust Cholesky-based sparse linear-solver libraries: Intel MKL (MKL Pardiso LLT) [Schenk et al. 2001], SuiteSparse (CHOLMOD) [Chen et al. 2008], and Apple Accelerate (Accelerate LLT) [Inc. 2023]. We do not include Eigen and Parsy [Cheshmi et al. 2018a] linear solvers, as they do not perform as well as the three high-performance libraries. Please see Appendix 27 for our evaluation leading to these choices. We evaluate and compare the timing and accuracy between the Parth-augmented versions and the original versions of each of these linear solvers on Intel (20-core Xeon(R) Gold, 6248 CPU 2.5GHz, 28MB LLC cache, 202GB RAM) and Apple (12-core M2 Pro chip, 16GB RAM) platforms. For the Intel platform, we use MKL version 2023.4-912 and CHOLMOD version 7.6.0 with Ubuntu 22.04. For the Apple platform, we use the latest shipping Accelerate framework compiled with Xcode. We will release our code for the Parth module with this paper.

Each of the above solver libraries offer a range of options and default settings for fill-reducing ordering. These choices significantly impact solution quality and overall solve speeds. CHOLMOD's default configuration is to first apply AMD [Amestoy et al. 2004]. If a low-quality AMD ordering (based on measures of non-zeros and operation count) is generated, METIS [Karypis and Kumar 1997] is then applied, and the better ordering of the two is adopted [Chen et al. 2008]. Across our benchmarks (see below), we observe that CHOLMOD almost always invokes secondary analysis with METIS and 95.02% of the time and METIS then accepted as the final ordering. To improve CHOLMOD's overall performance, we set it to directly apply METIS ordering in all subsequent experiments, eliminating this overhead from CHOLMOD's speeds. However, we emphasize that Parth also offers the option to use AMD when it is prefered for specific applications. MKL's LLT, by default, uses its own custom-optimized implementation of METIS, which we retain throughout our evaluation. Finally, Accelerate provides both AMD and METIS re-ordering options, with AMD the default. As with CHOLMOD, and as documented by Accelerate [Inc. 2023], we observe a significant degradation in solution quality and speed when using AMD orderings on large-scale meshes compared to METIS in Accelerate's LLT. Thus, we likewise apply its METIS reordering for all benchmarks. To ensure optimal performance, we further augment all three libraries' solvers with additional logic to *reuse* rather than recompute their symbolic analysis when the sparsity pattern remains unchanged across successive linear solves in our benchmark. This means that reported speedups are only with respect to sparsity pattern changes.

5.2 IPC: Benchmark

To analyze timings, bottlenecks and relative performance of these high-performance linear solvers in a consistent and fair side-by-side setting for IPC, we build a benchmark by precomputing and storing in consecutive order 143.5K sequential linear systems (Hessians, A, and gradients, g) which are generated by 5.5K Newton solves of challenging IPC volumetric FEM time-step problems [Li et al. 2020]. We compute these systems by time-stepping six of the most challenging (over 96% of all iterations in these simulations have changing sparsity due to contacts) deformable-body benchmark problems from Li et al. [2020]. Hessians in these systems range from 500K to 4.8M non-zero entries (with well over an order of magnitude increase in non-zeros for L-factors using state-of-the-art reordering with METIS [Karypis and Kumar 1997]) depending on number of active contact stencils, model resolution and geometry; see Table 1, and Li et al. [2020] for simulation model statistics. We additionally store the initial conditions of each time step so that each Newton problem can also be analyzed consistently and independently across varying linear solvers. We use the IPC library [IPC 2020] to both

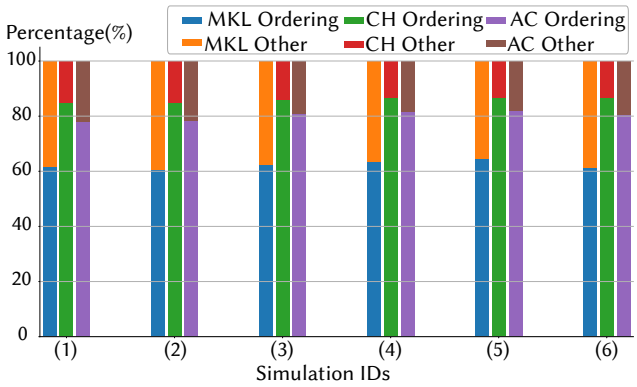


Fig. 10. **Fill-reducing ordering is the bottleneck for all symbolic analyses.** Here we summarize a bottleneck analysis for the symbolic steps of all three Cholesky solvers across all six simulation sequences in our benchmark. For each simulation and corresponding tool, both the fill-reducing ordering time and all additional symbolic analysis time are recorded. Here, "CH", and "AC" respectively denote CHOLMOD and Accelerate LLT runtimes. "Other" categories summarize all other symbolic analysis costs per Cholesky solver with tasks that vary depending on the Cholesky solver method.

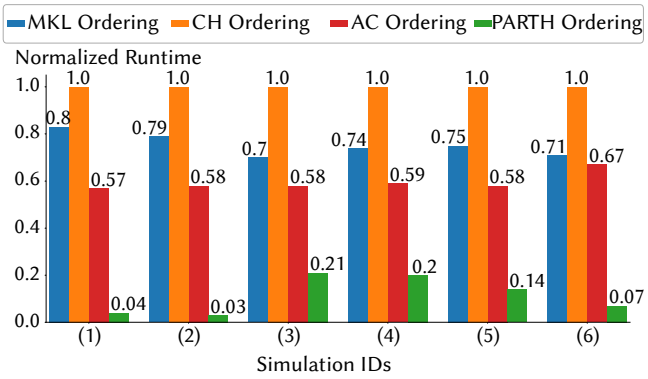


Fig. 11. **Parth Speedup.** the normalized runtime of Parth fill-reducing ordering compared to high-performance Cholesky solvers, namely, MKL, CHOLMOD (CH) and Accelerate (AC). Note that lower is better. The figure is normalized based on the slowest ordering algorithm. Across all 6 simulations, Parth fill-reducing ordering is faster than the fastest tool by 8.04x, 11.66x, 2.29x, 2.75x, 3.70x, and 9.79x speedup from simulation (1) to (6) respectively.

generate this benchmark and to perform Newton solves for some of the analyses in the following sections.

Specifically, here we focus on providing insight into Parth's performance on volumetric, and triangular mesh (Mat On Board) simulation. Furthermore, this consecutive linear problem benchmark is critical for our analysis across high-accuracy linear solvers (within nonlinear-solve inner-loops) as convergence behaviour in Newton-type methods for stiff problems, as in the time-stepped elastodynamics simulations we consider here, are sensitive to minor changes in computed descent directions. These variations are generated by solvers due to rounding and parallelization and, in turn, produce differences in the numbers of linear system iterations per Newton solve (and so the entire simulation runs) when using different solvers. Note that, as we demonstrate in Section 5.6, these variations do not generate iteration counts in favour of any of the Cholesky solvers. This benchmark then enables fair side-by-side evaluation of linear-solver methods within nonlinear solvers.

5.3 IPC: Bottleneck Analysis

Table 2 summarizes the breakdown of total runtime costs, per direct-solver library, for the linear solves of each simulation sequence in the benchmark. Here we see that for the two significantly faster solvers, MKL and Accelerate, symbolic analysis is clearly the primary bottleneck. For CHOLMOD, the story is a bit more nuanced. While the runtimes of CHOLMOD's symbolic computation phase are closely in line with MKL, its significant slowdown in comparison to MKL is in its numeric computation phase which ranges from two to four times slower than MKL. Here symbolic analysis, of course, remains a significant cost (generally of the same magnitude as numerical costs) and future optimizations of its numerical phase, should be expected to bring its numerical costs down similarly to those of MKL's.

This bottleneck, along with the significant prior research and engineering focus in the last years on heavily optimizing numerical computation (see Section 2), again reiterates our research focus here on improving symbolic analyses. Concretely, as an example consider that MKL spends, on average, 71.8% of its solve time here in symbolic analysis. Best (an impossible zero-cost computation) improvements in numerical computation e.g., the "Dolphin Funnel" sequence, would then be limited to a 1.28x speedup of linear solve costs.

If we then begin to look at the symbolic phase with finer granularity, we see that symbolic computation consists of multiple steps that vary with Cholesky solver implementation. However, *all* sparse Cholesky solvers require and employ high-quality fill-reducing ordering methods in their symbolic phase. In turn, as we summarize in Figure 10, across all three Cholesky solvers, the fill-reducing ordering is by far the largest bottleneck in each solver's symbolic analysis steps. Here we see that across benchmark problems, fill-reducing ordering takes an average of 62.32%, 62.77%, and 86.06% of the total symbolic analysis time for MKL, Accelerate, and CHOLMOD respectively. At the same time, we observe that even the most opaque Cholesky solver libraries, which do not provide access to their symbolic analysis otherwise, offer APIs that can be used to replace default fill-reducing ordering implementations with customized methods. Thus, if we can show a significant boost in symbolic fill-reducing ordering, this offers the combined advantages of addressing the symbolic analysis bottleneck while staying modular to take advantage of the current, highly optimized, machine-specific numerical phases in existing high-performance solvers. In the following sections, we will demonstrate that Parth can indeed be cleanly integrated in a plug-and-play fashion into all three solver solutions (and certainly others) for a significant overall performance improvement in linear solve times.

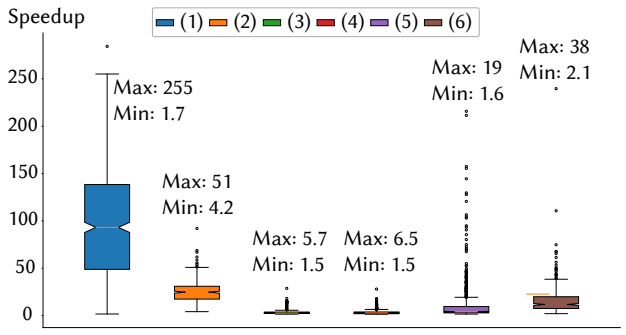


Fig. 12. **Parth speedup compared with best-available timing per solve.** The speedup achieved by Parth's fill-reducing ordering is compared to that of the best competitor among the three tools: Apple Accelerate, MKL, and CHOLMOD. Across all simulations, Parth's fill-reducing ordering consistently outperforms the state-of-the-art tool, resulting in speedups ranging from 1.5X to 255X (with outlying samples demonstrating further speedup in all sequences).

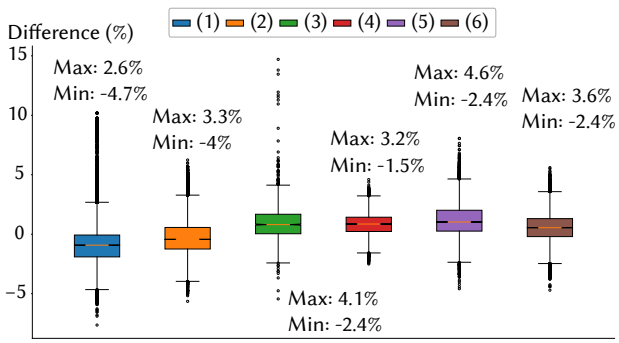


Fig. 13. **Fill-in reduction comparison across benchmark.** Here we we first identify the "optimal" fill-reducing ordering per solve, by choosing the sparsest factor generated across MKL, Accelerate, and CHOLMOD, for each. We then denote this smallest non-zero count among these three tools per linear system as t_{best} and Parth's comparable non-zero count for its factor of the system as t_p. We then plot here and in Figure 14 the difference $(t_p - t_{best})/t_{best}$ quantifying the percent deviation between the non-zeros generated by the best high-performance Cholesky solvers and those by Parth per iteration. Here the distribution of this measure across all simulation sequences shows that the median remains near zero, with minimum and maximum values generally (excluding outliers) falling well within ±5% of the best otherwise obtained solution.

5.4 IPC: Parth Speedup

We first consider here Parth's speedup in comparison to the fill-reduction timings of all three Cholesky solver libraries across our full benchmark as Parth only accelerates this part of the linear solver pipeline. In Figure 11, we summarize ordering runtimes normalized against the slowest solver. Here we see that Parth outperforms all three Cholesky solvers across *all* problem sequences in the benchmark, with speedups ranging from 2.8X to well over an order of magnitude.

We next push this further and consider an "optimal" competing symbolic analysis which is enabled, without overhead, to pick the *fastest* ordering among MKL, CHOLMOD and Accelerate (which can otherwise vary per solve for best speeds), for each Newton solve sequence in the benchmark. In Figure 12 we analyze Parth's speedup against this hypothetical best-speed analysis per Newton-solve sequence. Here we see that, even at the granularity of individual solves, Parth always remains significantly faster than the next best tool with speedups per solve ranging from 1.5x to 255X, across a wide range of solve sequences with both large and small sparsity changes. Note that this is partly due to the Parth compression of graph dual G which can be coarsened due to $DIM = 3$. However, in our *Remeshing* benchmark, we will see Parth overhead when $DIM = 1$ and Parth does not compress the graph.

5.5 IPC: Parth Ordering Quality

In the above analysis we demonstrate that Parth efficiently computes permutation vectors with significant speedups in comparison to state-of-the-art symbolic methods across diverse sparsity patterns, Hessian sizes, and changing regions of sparsity updates. This demonstrates Parth's efficient reuse of computation across iterations within Newton solves, per time-step, and across sequential Newton solves, in time-stepped simulation sequences. With *timing* settled we next analyze here the *fill-reduction quality* of the orderings computed by Parth, and see that Parth delivers high-quality fill-reduction, comparable to the best sparsity generated among all three compared Cholesky solvers.

In Figure 13 we compare Parth fill-reduction, with (as in the last section) an imagined, "optimal" competing symbolic analysis method across our benchmark. Specifically, we compare, per linear solve in the benchmark, against the *best fill-reducing* ordering among MKL, CHOLMOD and Accelerate that generates the sparsest factor. Across all simulation sequences in the benchmark Parth's permutation quality remains consistent with the best sparse factors computed by the Cholesky solvers' fill-reduction—largely remaining well within a ±5% range, with medians close to zero.

Looking even closer at individual iterations, in Figure 14 we plot the relative sparsity difference in non-zeros between the best-of solution and Parth across the first 40K successive linear solves in the high-contact and compression "Ball Mesh Roller" simulation sequence. Here we see that the total relative sparsity difference comparably remains in a range of ±6%.

In summary, we find that across our benchmark, permutation vectors generated by Parth sometimes decrease, and sometimes slightly increase, sparsity (albeit both marginally) over the best provided by top competitor libraries. Recalling that heuristics applied in fill-reducing ordering, e.g. as in METIS [Karypis and Kumar 1997], lead to similar-scale minor variations in output when executed multiple times on the same problem, we see that Parth then provides comparable quality fill reduction at significant speed-up.

5.6 IPC: Parth Numerical Effect

So far we have demonstrated that across our benchmark Parth generates state-of-the-art quality fill-reduction for its factors with

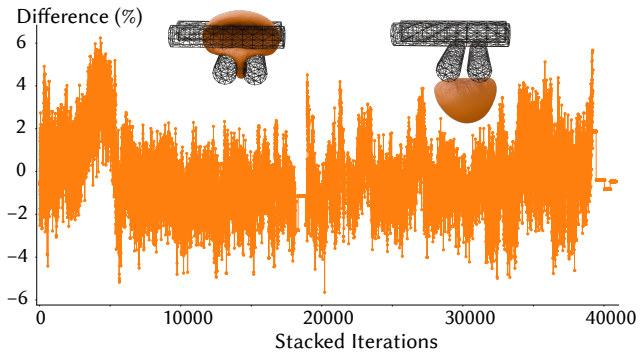


Fig. 14. **Detailed fill-in reduction analysis per solve.** The difference in the number of non-zeros (NNZ) in the factor generated by Parth is compared with that of the best results from MKL, Accelerate, and CHOLMOD. This comparison spans 40,000 stacked iterations from the Newton solves of time steps, with highly varying contact configurations and so large and rapid changes in sparsity patterns. The range of the difference is in ±6%, demonstrating Parth's generated sparse factors are comparable to the best available fill-reducing ordering while delivering large speedups in timings.

Table 3. **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

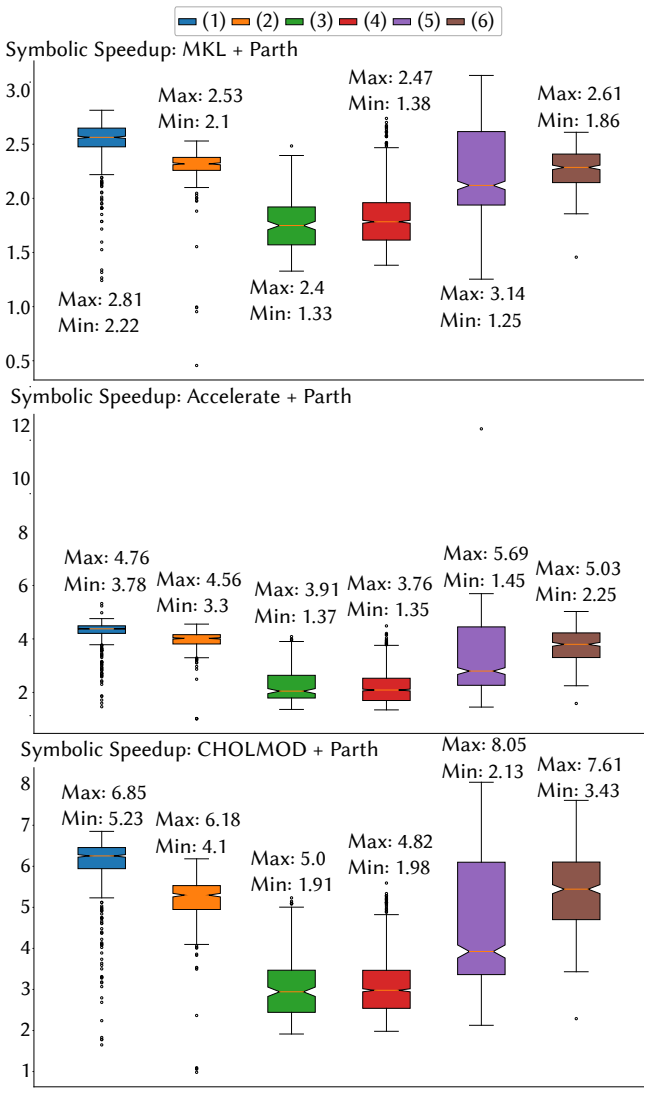


Fig. 15. **IPC: Parth symbolic performance impact across benchmark.** The three whisker plots indicate the performance impact of Parth on the symbolic step of MKL, Accelerate and CHOLMOD. In all cases, except one iteration in "Ball Mesh Roller", Parth improved the symbolic analysis performance by up to 8.05x speedup.

significant runtime speed-ups. 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, 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 3, 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

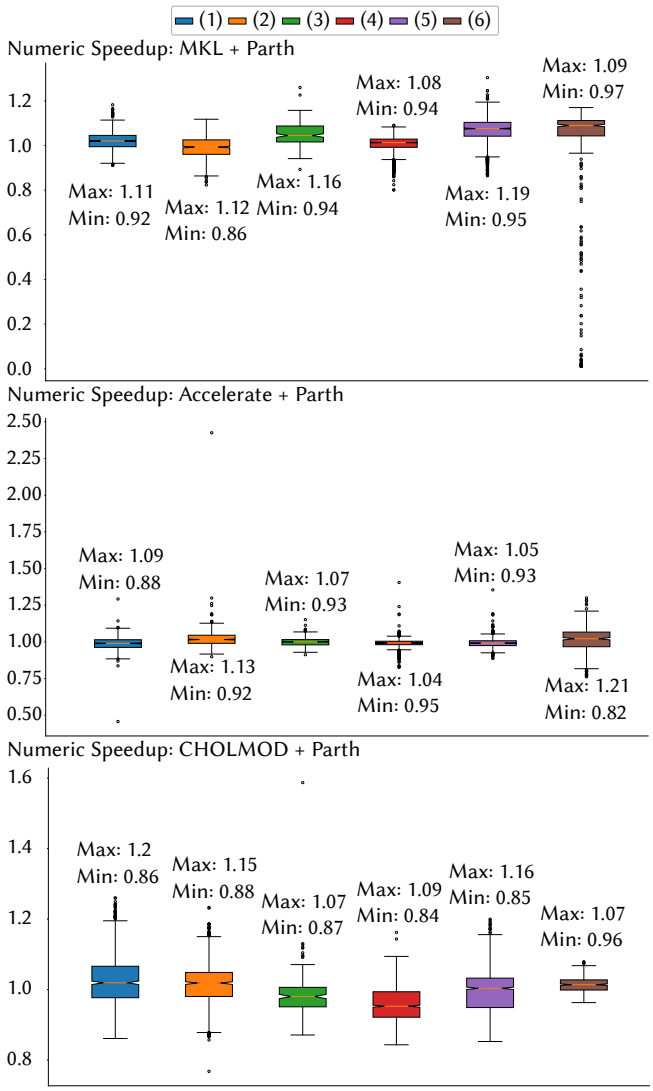


Fig. 16. **IPC: Parth preserves the well-optimized performance of the numerical phases in each base solver across the benchmark.** The three whisker plots indicate the performance impact of Parth on the numeric step of MKL, Accelerate and CHOLMOD with a maximum of 21% performance improvement and 18% 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.

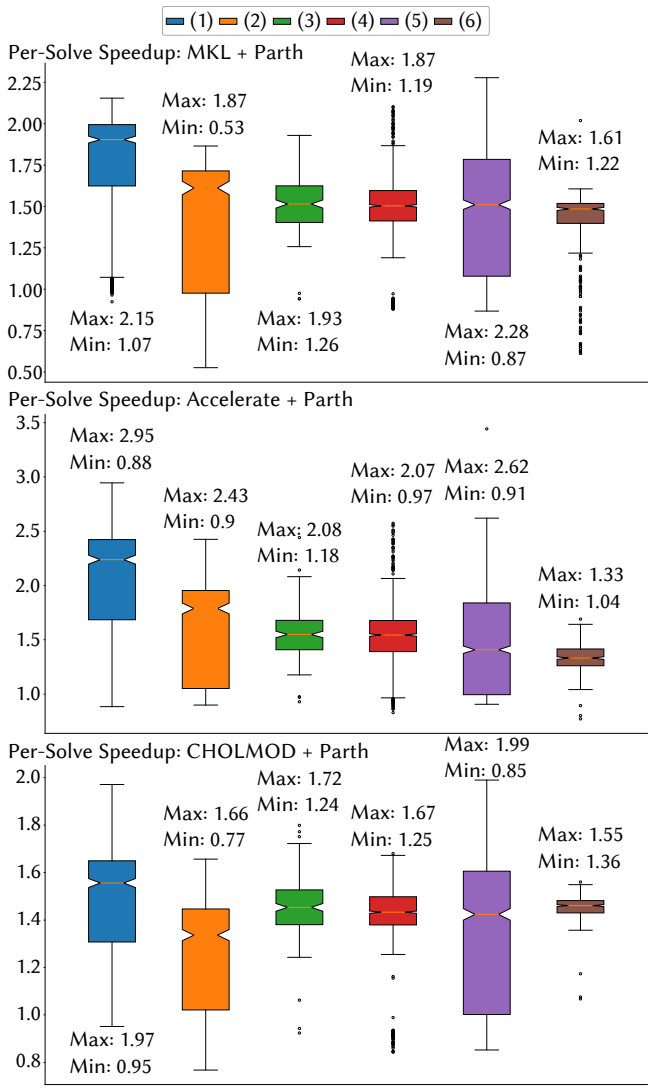


Fig. 17. **IPC: Parth's Per-Solve (Symbolic + Numeric) Performance Impact Across Benchmark.** As previously stated, MKL encountered numerical problems in the "Arma Roller"(6) simulation. Moreover, our analysis reveals that the minimum of 0.53x speedup recorded in "Roller Ball" (2) with MKL is due to a high number of repetitive iterations where the sparsity pattern remained unchanged. This implies a repeated use of a single permutation vector with slightly lower quality due to the absence of contact. Excluding these iterations, Parth achieves a 1.53 end-to-end speedup for this simulation.

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.

5.7 IPC: Per Sparse Linear Solve Performance

Finally, we consider the performance impact of our three Parth-integrated Cholesky solvers on the symbolic and total costs of linear solves. Following the breakdown in Table 2 we focus on the symbolic analysis phase per solver, In Figure 15, across the benchmark, we see maximum speedups of 8.05x, 5.69x, and 3.14x for CHOLMOD, Accelerate and MKL respectively, with corresponding median speedups

of 4.9x, 3.5x, and 2.2x, and minimum speedups of 1.9x, 1.4x, and 1.2x. As covered in Sections 5.5 and 5.6, with this speedup, Parth still maintains state-of-the-art fill-in and comparable numerical quality to CHOLMOD and Accelerate (recalling that we observe MKL has occasional but catastrophic failures in accuracy). This suggests that Parth-integrated solves 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 in Figure 16, where we confirm integration of Parth brings the above speedup without additional overhead elsewhere in the already optimized numerical computations of the Cholesky solver packages.

In Figure 17 we see that by just direct integration of the Parth module into Cholesky solver packages, we obtain speedups on full Cholesky solve costs of up to 2.95X, 2.28X and 1.97X (1.5X, 1.54X, and 1.43X median) for Accelerate, MKL and CHOLMOD respectively. In line with our earlier analysis, we also confirm that Parth consistently extracts the greatest performance speedups as it is integrated with successively more optimized and more performant Cholesky solvers taking advantage of hardware – here in Accelerate and then following with MKL. In Table 4 we correspondingly report the *total* Cholesky-solver runtime costs across all linear solves for each simulation sequence in the benchmark, and again observe consistent breakdowns.

There are two important takeaways from Table **??**. First, the Cholesky solve baselines currently used by practitioners are not the fastest possible, leading to confusion and suboptimal choices when integrating fast direct and approximation solvers. For example, IPC involves multiple steps, and to achieve fast end-to-end acceleration, all these steps must be optimized [Huang et al. 2024]. However, selecting the best linear solver requires balancing accuracy and speedup. For instance, Huang et al. [2024] presents a sophisticated computational pipeline for IPC that leads to impressive end-to-end speedup, but their linear solve baseline is not well optimized. In their paper, they provide a detailed evaluation of their linear solvers, including speedups and failure cases. The reported 4x speedup over CHOLMOD for their PCG linear solve comes with the drawback of potential failures in some benchmarks (as reported in [Huang et al. 2024]). However, by looking at Table4, we observe that switching from CHOLMOD to Apple Accelerate on an M2 processor, combined with integrating Parth, can deliver up to 6x speedup over CHOLMOD, with an average speedup of 4.07x. This means a practitioner could opt to use the CCD strategy from [Huang et al. 2024] along with a Parth-integrated Apple Accelerate, benefiting from both the performance improvements in CCD and the numerical stability of a Cholesky solve. This highlights one of the key contributions of our work: providing a crucial comparison between different Cholesky solvers.

The second important takeaway is the runtime savings achieved by integrating Parth. For example, consider using CHOLMOD for the "Arma Roller" simulation due to its numerical stability on an Intel processor. While the solve speedup is 1.45x, the 45% reduction in runtime translates to saving 9.75 hours. Given the minimal effort required to integrate Parth (3 lines of code) and the consistent numerical stability, many practitioners can easily gain these performance benefits without needing to refactor the entire computational pipeline, which is often not a straightforward task.

5.8 Remeshing: Benchmark (NEW)

In this section we next evaluate how Parth performs in remeshing applications where sparsity patterns change due to localized updates in mesh geometry and topology which result in both adding and deleting of the nodes in G and added and removed edges. Here we test with a remeshing pipeline in which we select remeshing patch regions on triangle meshes, remesh with Botsch and Kobbelt [2004] to alter patch structure, and then apply global discrete Laplacian operator [Jacobson et al. 2024]. For comprehensive analysis, remeshing operations are chosen to cover patches with a range of sizes comprising 1%, 5%, 10%, 20% of the faces. Each patch is created around a randomly selected face ID. Here, per each mesh, the same face ID is used for comparison between different linear solvers. As a result, comparisons across linear solvers for a specific mesh and patch is consistent with identical computation. For each patch size, fifty such samples are chosen, covering different areas of a surface mesh, forming 200 sparse linear solves per mesh. After applying each patch remeshing, we reset the mesh and select another patch to measure Parth's reuse capability for each patch size. This testing is applied across all the meshes in the Stein [2024] repository with the exclusion of the overly simple Cube mesh.

In addition to testing an application with changing matrix size, this benchmark additionally allows evaluation of Parth's performance on triangular meshes, and on linear solves with a problem dimension of $DIM = 1$ (recalling for IPC, it is 3). As a result, in this evaluation, Parth directly employs the graph dual of the input matrix instead of the coarsened version where the graph nodes associated with a single DOF are combined. As a result, this benchmark does not require application of Parth's compression.

5.9 Remeshing: Bottleneck Analysis

Figure 18 provides an analysis of the bottleneck of the symbolic stage. Here, we divide the meshes into separate groups based on their number of DOFs to show the effect of mesh sizes on the symbolic analysis stage. As we increase mesh size, and consequently increase the size of the graph dual of the Laplacian operator, the fill-reducing ordering performance overhead becomes more prominent. For example, for small meshes, fill-reducing ordering is 57% of the symbolic analyzing, and for Large meshes, this overhead increases to 82% for the MKL solver. As a result, for large-scale problems, optimizing symbolic analysis is clearly important for the Cholesky solve pipeline. Note that these results are consistent with our analysis of the IPC benchmark.

5.10 Remeshing: Total Linear Solve Performance

Here we discuss only the symbolic analysis and total per-solve linear solve performance of the remeshing benchmark. However, consistent with our IPC benchmark analysis, we also provide our numerical performance analysis of the remeshing benchmark in Appendix F. For this set of analyses, we divide the performance data into five categories. The first category is the initialization step,

Table 4. **IPC: Breakdown of Cholesky solve costs per library across all Parth-integrated Cholesky solves in our benchmark.** Here we summarize the timing (wall-clock seconds) and peresent total linear solve runtime for default and Path-integrated versions per simulation sequence in our IPC benchmark for all three state-of-the-art Cholesky solvers corresponding to original costs in Table 2. In here, the solve speedup from left (Dolphin Funnel) to right (Arma Roller) for 3 Cholesky solvers are as follow: MKL speedup={1.79, 1.68, 1.49, 1.56, .1.71, 1.37} and Accelerate speedup={2.07, 1.89, 1.56, 1.78, 1.33} and CHOLMOD speedup={1.5, 1.41, 1.43, 1.45, 1.55, 1.45}. Note that here, 45% performance benefits for CHOLMOD in "Arma Roller" is equal to saving 9.75 hours which acheived by simple integration of Parth without numerical side effect.

Parth + Tool	Step	Dolphin Funnel	Ball Mesh Roller	Mat On Board	Rods Twist	Squeeze out	Arma Roller
MKL	Symbolic time(s)	3261 →1424	2623 →1134	1551 →914	5363 →2481	9098 →4425	35357 →15714
	Numeric time(s)	932 →923	1085 →1075	485 →457	2800 →2751	2543 →2397	30703 →32335
Accelerate	Symbolic time(s)	1773 →490	1465 →369	985 →483	5363 →2481	5540 →2119	25331 →6913
	Numeric time(s)	706 →709	861 →861	412 →412	2578 →2602	2231 →2251	51871 →51098
CHOLMOD	Symbolic time(s)	2857 →597	2353 →453	1600 →558	8552 →2764	9012 →2422	35021 →6696
	Numeric time(s)	4200 →4106	4511 →4432	1663 →1727	9098 →9385	9483 →9499	58282 →57492

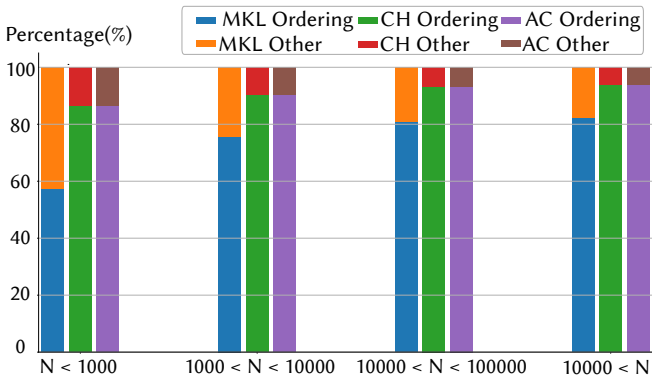


Fig. 18. **Fill-reducing ordering is the bottleneck for all symbolic analyses.** Here we summarize a bottleneck analysis for the symbolic steps of all three Cholesky solvers across all 20 Meshes ranging from 642 DOFs to approximately 1.6m DOFs in our benchmark. For each simulation and corresponding tool, both the fill-reducing ordering time and all additional symbolic analysis time are recorded. Here, "CH", and "AC" respectively denote CHOLMOD and Accelerate LLT runtimes. "Other" categories summarize all other symbolic analysis costs per Cholesky solver with tasks that vary depending on the Cholesky solver method.

while the next four categories reflect performance with respect to patch size. This categorization allows us to offer further insight into Parth's initialization cost without the compression phase, making the overhead of building the HGD data structure more visible. Furthermore, as shown in the teaser, a 2% selection of the surface mesh is not considered small. By increasing these patch sizes to 20%, we aim to provide insight into more challenging situations where the changes are more drastic, demonstrating Parth's performance in terms of both reuse capability and fill-reducing quality.

Figure 19 shows Parth's effect on the symbolic analysis. As expected, since $DIM = 1$ for this computational pipeline, Parth reduces performance in the initialization step. However, since the HGD computation is also reused in the fill-reducing ordering computation, the average performance speed is 70% of the baseline fill-reducing routine. After the initialization step, Parth consistently provides speedup. For example, in CHOLMOD with patch sizes of 1%, Parth achieves an order-of-magnitude speedup. It is important to note that as the size of the patches on the mesh increases, Parth's performance decreases due to less temporal coherence and fewer opportunities for reuse, i.e., the changes become less gradual. In patch sizes of 20%, for instance, we observe up to a 3.36x speedup, which is smaller than the order-of-magnitude speedup achieved with patch sizes of 1% of the surface faces in symbolic analysis.

The performance benefits in symbolic analysis lead to an overall improvement in total Cholesky solve time, as shown in Figure **??**. As expected, this performance boost mirrors the trends seen in symbolic analysis, resulting in up to a 5.89x, 3.55x, and 2.82x speedup compared to Accelerate, CHOLMOD, and MKL, respectively.

6 LIMITATIONS AND FUTURE WORK(NEW)

Currently, Parth is designed to provide performance benefits by reusing symbolic analysis computations. However, when the computational bottleneck is not symbolic analysis, and specifically not fill-reducing ordering, Parth's improvement is not applicable. We plan to address this limitation by adding comparable and complementary adaptive numerical acceleration techniques to Cholesky solves.

Another important limitation for future investigation is how long Parth can preserve high-quality fill-reducing ordering performance, as finding high-quality fill-reducing ordering generally requires global information. To assess Parth's limitations, we apply remeshing to 1,000 patches, each comprising 1% of mesh faces, on the triangular meshes evaluated in our remeshing benchmark. Each of these 1,000 patches is selected around face IDs that were not chosen in the previous patch, and the selection is made from a uniform distribution. This randomness allows for the evaluation of different scenarios while modifying the triangle meshes. Here, we evaluate how many applications of these 1% patch remeshings can be applied before the performance of numerical computation drops below 85% of the baseline. This, in turn, indicates how many times Parth can generally provide high-quality reuse without recomputing an entire fill-reducing ordering. Note that for this test, the aggressive reuse algorithm (Appendix E) is activated to ensure that Parth does not

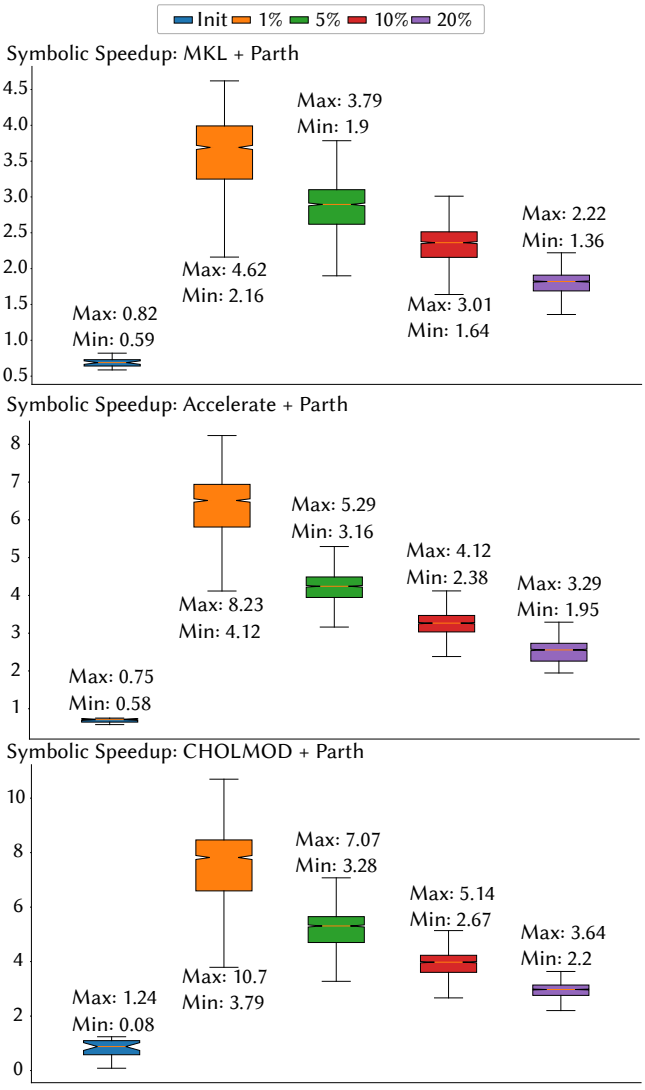


Fig. 19. **Symbolic performance impact across Patch benchmark.** Here the performance impact of Parth on symbolic analysis of the Patch pipeline is shown. As it is shown, the initialization step is slowed down to 0.7x performance on average, due to Parth overhead. However, after that, due to the reuse a significant boost in performance is observed. Note that as expected, the reuse performance benefits are reduced due to the more aggressive changes in the mesh structure resulting from using the remesher.

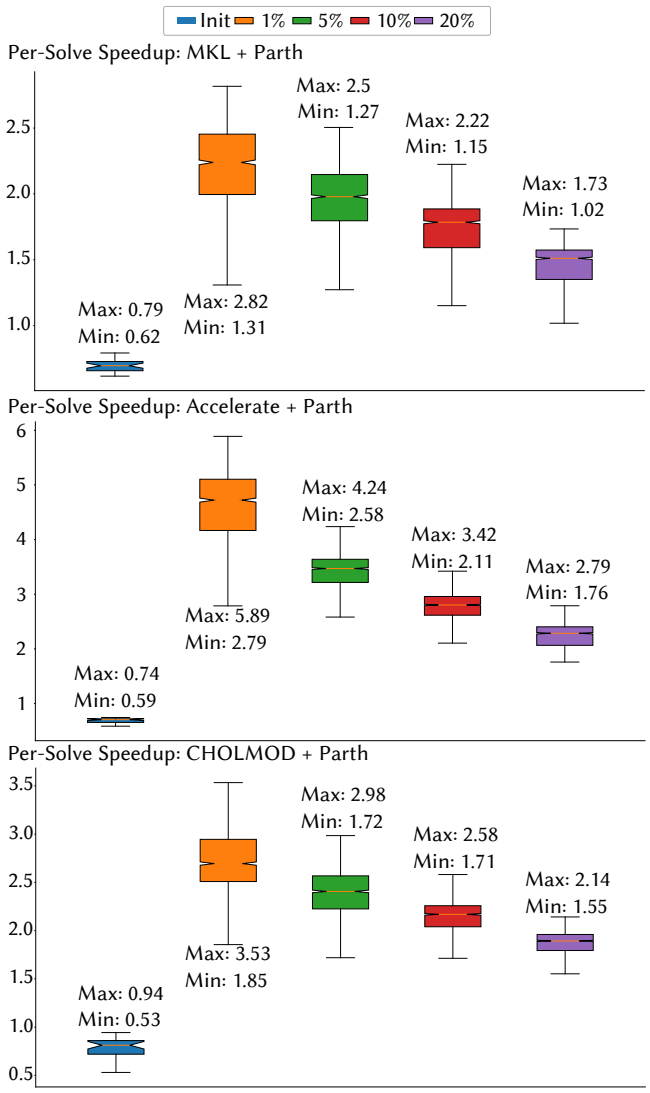


Fig. 20. **Per linear solve speedup when using remesher.** Here, after each re-meshing, the per-solve speedup is shown. As we expected, when the numerical time comprises the smaller ratio of a solve runtime, Parth speedup is more significant. For example, since the Accelerate framework has a very fast numerical performance due to its fast processor, Parth shows up to 5.89x speedup with a median of 4.6x for a Patch size of 1% of the total faces.

recompute the full fill-reducing ordering when a patch on the root separator changes.

In Figure 21, we see the results of this test, with the x-axis showing mesh IDs from [Stein 2024], sorted by the number of DOF, and the y-axis indicating the number of reuses. As shown, for some meshes, Parth can maintain high-quality fill-reducing ordering even after 500 applied patches. On average, performance drops below 85% after 70 remeshings of distinct patches, which approximately result in 70% changes to the mesh. However, this generally depends on the shape of the mesh and the degree of deformation caused by the remesher. On average, after approximately 70% changes to the triangle meshes, Parth needs to be reset to obtain new fill-reducing ordering information, as the underlying geometry has changed significantly.

7 CONCLUSION AND DISCUSSION

In summary, our evaluation confirms that the Parth module provides successful, simple and direct integration into all three state-of-the-art Cholesky solver libraries. Parth generates for each, across widely

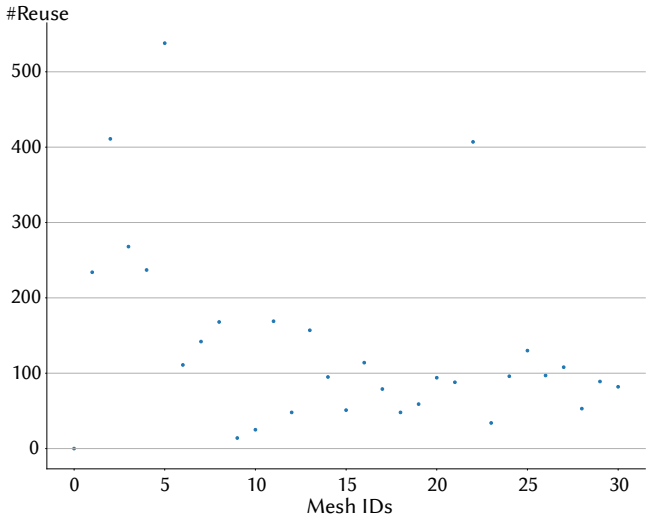


Fig. 21. **Local Usefulness of Fill-Reducing Ordering** We apply a sequence of 1,000 remesh operations, each affecting 1% of the faces. After each remeshing, we apply a discrete Laplacian smoother and measure the numerical runtime. The Parth-integrated CHOLMOD factorization runtime is then compared with the default CHOLMOD numerical performance. Assuming t_p as the Parth-integrated runtime and t as the default, we measure $(t_p - t)/t$ to quantify the differences. We count the number of remeshing operations until this metric exceeds 15%, meaning that the numerical performance has degraded to within 15% of the default performance. This number is reported for each mesh, indicating how many times Parth can be applied without a full re-computation of fill-reducing ordering.

ranging and challenging linear systems in our benchmark, comparable, high-quality fill-in reduction – critical for efficient sparse linear solves. At the same time Parth delivers an up to 255x speedup for fill-reducing ordering computation, while preserving the numerical stability and performance in the resulting downstream Cholesky factorization and subsequent solve processes of all three solver libraries, resulting in an up to 6x speedup in overall direct solve times – Outperforming the *best* speedups obtained by recent architecture-customized Cholesky solver updates [Inc. 2023]. As direct solvers are a core and sometimes unavoidable bottleneck in applications with dynamically changing sparsity, this represents a significant improvement obtained by a simple module addition, at just the API level per-solver. We expect this enhancement will likewise, as proofed out here across solvers, be equally applicable to other and future-developed improved Cholesky solver packages.

We have presented a new comprehensive benchmark for fair side-by-side evaluation of Cholesky solvers across sequential, highly challenging practical linear solve systems, with dynamically changing and coherent sparsity transitions. We have then presented a new and extensive evaluation of state-of-the-art Cholesky solvers on this benchmark. Both of these first two contributions have directly led to the insights enabling the new Parth module's perfromance boost and we expect that this benchmarking can and will lead to more comparable improvements in this challenging and important area critical across domains in computer graphics ranging from physics simulation to deformation processing and adaptive remeshing to name just a few.

At the same time we have extensively evaluated our new Parth module to demonstrate its advantages in challenging applications with rapid sparsity change – contacting elastodynamics simulation and remeshing. Looking ahead, we observe that Parth is a quite general solution and so we expect its flexibility and efficiency should be a widely applicable solution for improving the fill-reducing ordering runtime for high-performance direct solvers in diverse applications with changing sparsity patterns. Parth's flexibility is likewise evidenced in that it requires no-per application tuned parameters, and its underlying algorithm does not rely on the type of algorithm applied in creating and solving the sequential linear systems treated and so should be generally applicable. Likewise, although we focus entirely here on its performance benefits for direct solvers, we look ahead to interesting investigations in it potential application to preconditioning methods, including both linear and nonlinear methods [Li et al. 2019]. Finally we observe that with our long-term goal of easy plug-and-play generality we have likely left many domain specific improvements possible for even further speedups.

REFERENCES

2020. IPC. https://github.com/ipc/ipc-sim GitHub repository.

Gene M Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*. 483–485.

Patrick R Amestoy, Timothy A Davis, and Iain S Duff. 2004. Algorithm 837: AMD, an approximate minimum degree ordering algorithm. *ACM Transactions on Mathematical Software (TOMS)* 30, 3 (2004), 381–388.

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.

Yanqing Chen, Timothy A Davis, William W Hager, and Sivasankaran Rajamanickam. 2008. Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate. *ACM Transactions on Mathematical Software (TOMS)* 35, 3 (2008), 1–14.

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. 2018a. 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.

Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. 2018b. 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.

Kazem Cheshmi, Danny M Kaufman, Shoaib Kamil, and Maryam Mehri Dehnavi. 2020. NASOQ: numerically accurate sparsity-oriented QP solver. *ACM Transactions on Graphics (TOG)* 39, 4 (2020), 96–1.

Kazem Cheshmi, Michelle Strout, and Maryam Mehri Dehnavi. 2023. Runtime composition of iterations for fusing loop-carried sparse dependence. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

Timothy A Davis and William W Hager. 2005. Row modifications of a sparse Cholesky factorization. *SIAM J. Matrix Anal. Appl.* 26, 3 (2005), 621–639.

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.

Jack J Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S Duff. 1990. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)* 16, 1 (1990), 1–17.

Philipp Herholz and Marc Alexa. 2018. Factor once: reusing cholesky factorizations on sub-meshes. *ACM Transactions on Graphics (TOG)* 37, 6 (2018), 1–9.
Philipp Herholz and Olga Sorkine-Hornung. 2020. Sparse cholesky updates for interactive mesh parameterization. *ACM Transactions on Graphics (TOG)* 39, 6 (2020), 1–14.
Kemeng Huang, Floyd M Chitalu, Huancheng Lin, and Taku Komura. 2024. GIPC: Fast and stable Gauss-Newton optimization of IPC barrier energy. *ACM Transactions on Graphics* 43, 2 (2024), 1–18.
Apple Inc. 2023. Accelerate Framework. Available at https://developer.apple.com/documentation/accelerate.
Alec Jacobson, Daniele Panozzo, et al. 2024. Libigl Tutorial - Laplace Equation. https://libigl.github.io/tutorial/#laplace-equation. Accessed: 2024-10-18.
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).
George Karypis and Vipin Kumar. 1997. METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. (1997).
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.
Jing Li, Tiantian Liu, Ladislav Kavan, and Baoquan Chen. 2021. Interactive cutting and tearing in projective dynamics with progressive cholesky updates. *ACM Transactions on Graphics (TOG)* 40, 6 (2021), 1–12.
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.
Minchen Li, Ming Gao, Timothy Langlois, Chenfanfu Jiang, and Danny M Kaufman. 2019. Decomposed optimization time integrator for large-step elastodynamics. *ACM Transactions on Graphics (TOG)* 38, 4 (2019), 1–10.
Joseph WH Liu. 1990. The role of elimination trees in sparse factorization. *SIAM journal on matrix analysis and applications* 11, 1 (1990), 134–172.
Yang Liu, Pieter Ghysels, Lisa Claus, and Xiaoye Sherry Li. 2021. Sparse approximate multifrontal factorization with butterfly compression for high-frequency wave equations. *SIAM Journal on Scientific Computing* 43, 5 (2021), S367–S391.
François Pellegrini. 2009. Distillating knowledge about Scotch. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.
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.
Steven C Rennich, Darko Stosic, and Timothy A Davis. 2016. Accelerating sparse Cholesky factorization on GPUs. *Parallel Comput.* 59 (2016), 140–150.
Olaf Schenk, Klaus Gärtner, Wolfgang Fichtner, and Andreas Stricker. 2001. PARDISO: a high-performance serial and parallel sparse linear solver in semiconductor device simulation. *Future Generation Computer Systems* 18, 1 (2001), 69–78.
Patrick Schmidt, Dörte Pieper, and Leif Kobbelt. 2023. Surface maps via adaptive triangulations. In *Computer Graphics Forum*, Vol. 42. Wiley Online Library, 103–117.
Silvia Sellán, Jacob Kesten, Ang Yan Sheng, and Alec Jacobson. 2020. Opening and closing surfaces. *ACM Transactions on Graphics (TOG)* 39, 6 (2020), 1–13.
Oded Stein. 2024. odedstein-meshes: A Computer Graphics Example Mesh Repository. (2024).
Mihalis Yannakakis. 1981. Computing the minimum fill-in is NP-complete. *SIAM Journal on Algebraic Discrete Methods* 2, 1 (1981), 77–79.

A HGD COMPUTATION REUSED IN FILL-REDUCING ORDERING

To further illustrate how the information in $\mathcal{B}$ contributes to fill-reducing ordering computation, refer to Figure 22. In this figure, the separator set, colored 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 22(a.1) and 22(b.1), it becomes apparent that the graph G undergoes renumbering, positioning the separator set numbers after the orange and blue colored left and right sub-meshes, respectively. A comparison of the matrix A in Figure 22(a.2) with A_p in Figure 22(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.

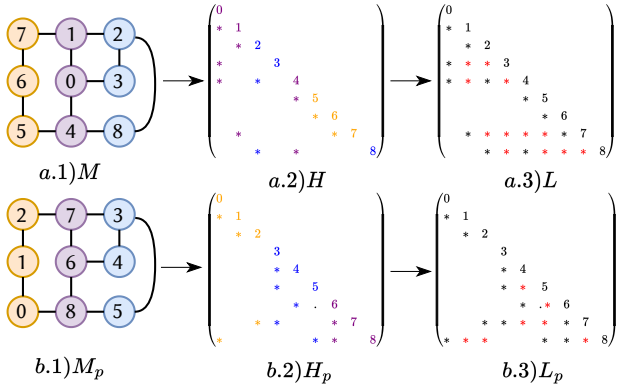


Fig. 22. **Partial fill-reducing ordering using separator set.** In Figure 5, the computation used for forming $\mathcal{B}[0]$ (sub-mesh colored in purple) is used in fill-reducing process. This process involves renumbering the left sub-mesh from $\{5, 6, 7\}$ to $\{0, 1, 2\}$, the right sub-mesh from $\{2, 3, 8\}$ 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.

B NODE CHANGE SYNCHRONIZER

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 23 illustrates how removing a node and adding a new one can alter the labeling 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 23(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 $map[c_1] \neq map[c_2]$, 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 4 describe the synchronization process, which is performed in 3 steps as follows:

Algorithm 4 *NodeChangeSynchronizer*

Global: $\mathcal{B}$
Input: *map*
1: **if** *map* is empty **then**
2: return
3: **end if**
/* Step 1: Deleting the removed nodes from $\mathcal{B}$ */
4: $N_D = deletedNodes(map, |G|)$
5: **for** *node* in S_D **do**
6: $i = getSubGraphID(node)$
7: $remove(node, \mathcal{B}[i].nodes)$
8: $update(node, \mathcal{B}[i].\mathcal{P}_l)$
9: **end for**
/* Step 2: Update the index of nodes in $\mathcal{B}$ */
10: $UpdateNodeIndex(\mathcal{B}[i], map)$
/* Step 3: Assign a sub-graph to each added node */
11: $N_A = addedNodes(map)$
12: $ChangeHappened = True$
13: $FullQueue = N_A$ **and** $EmptyQueue$ = Empty
14: **while** $ChangeHappened$ **do**
15: $ChangeHappened = False$
16: **while** $\neg FullQueue$.isEmpty() **do**
17: $node = FullQueue$.front()
18: $FullQueue$.pop()
19: $S_{\mathcal{B}} = getAllSubGraphs(node)$
20: **if** $S_{\mathcal{B}}$.isEmpty() **then**
21: $EmptyQueue$.push($node$)
22: **else**
23: $i = LCA(S_{\mathcal{B}})$
24: $AssignNodeToSubGraph(i, node)$
25: $ChangeHappened = True$
26: **end if**
27: **end while**
28: **if** $ChangeHappened$ **then**
29: $Swap(FullQueue, EmptyQueue)$
30: **end if**
31: **end while**
32: **if** $\neg EmptyQueue$.isEmpty() **then**
33: $AssignNodeToEmptySubGraphORLast(i, node)$
34: **end if**

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 (see Section 4.3 and Lines 4-8 in Algorithm 4). For example, in Figure 23(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 23(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 23(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

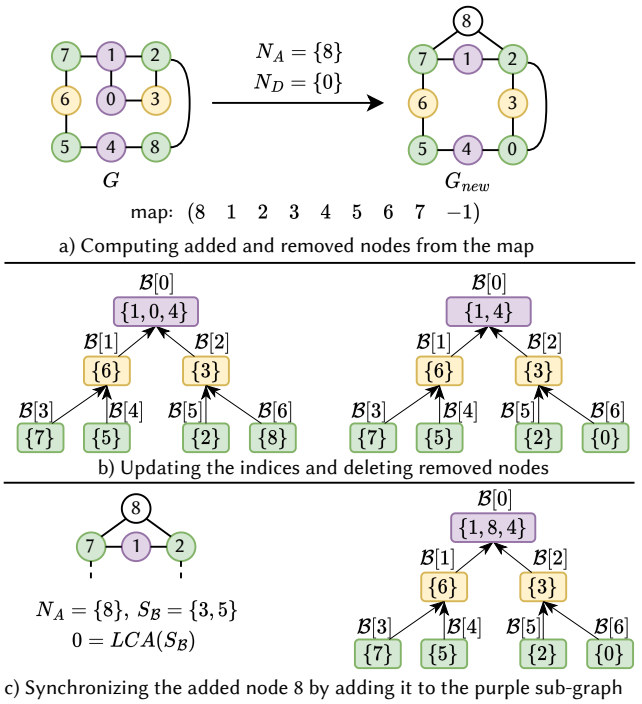


Fig. 23. **Example of the Node Synchronizer.** In this example, node 0 is deleted from G and node 8 is added to G_{new}. Note that since node 8 in G_{new} is newly added, $map[8] = -1$. Also note that node 8 in G and node 0 in G_{new} still represent the same node. In (b), we can see that $\mathcal{B}$ 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 too. This maintain the separator relation between sub-graphs.

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 neighbors. 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 23(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 $\mathcal{B}$.

C SYNCHRONIZER: DIRTY SUB-GRAPH DETECTION

The objective of the algorithm 5 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

Algorithm 5 *DirtySubGraphDetection*

Global: $\mathcal{B}$
Input: $E_{\mathcal{B}}$
Output: D_C, D_F
/*Detect dirty sub-graph in $\mathcal{B}$*/
1: **for** $< \mathcal{B}[a], \mathcal{B}[b] >$ in $E_{\mathcal{B}}$ **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: $s_{min} \leftarrow min(a, b)$ and $s_{max} \leftarrow max(a, b)$
/*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 6 *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**

Algorithm 7 *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**

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).

D SYNCHRONIZER: MARK AND DECOMPOSE SUB-GRAPHS

Algorithm 6 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.

E 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 24). 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 reuse.

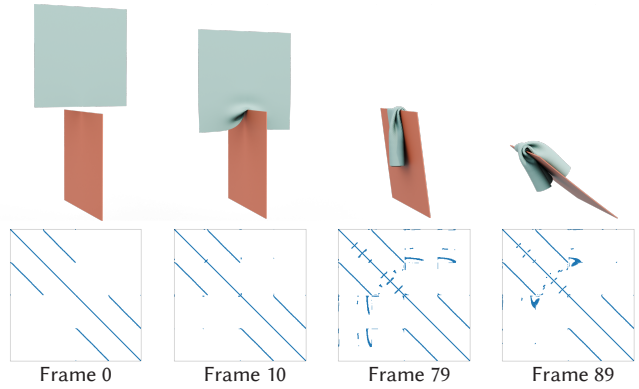


Fig. 24. **4 samples from the "Mat on Board" simulation in the IPC [Li et al. 2020] benchmark.** The top row shows the simulation scene, while the bottom row displays the first system of linear equations required to be solved for these frames. As observed, between frames 0 and 10, when contact occurs, 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.

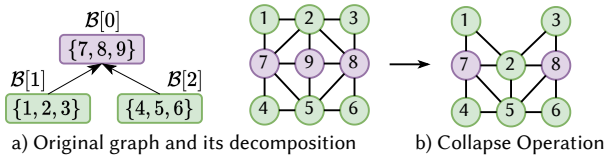


Fig. 25. **Collapse Example.** Remeshers used in SurfaceMap [Schmidt et al. 2023], IGL decimate function [Jacobson et al. 2013] and Botsch and Kobbelt [2004] use operations such as flip, split and collapse. In here, an example of collapse is shown where node 2 from left sub-graph is collapsed into node 9 which is a separator, resulting in violation of separator set properties as now there is a connection between $\mathcal{B}[1]$ and $\mathcal{B}[2]$. Without aggressive reuse heuristic, Parth needsl am no to re-decompose the whole graph which result in zero reuse.

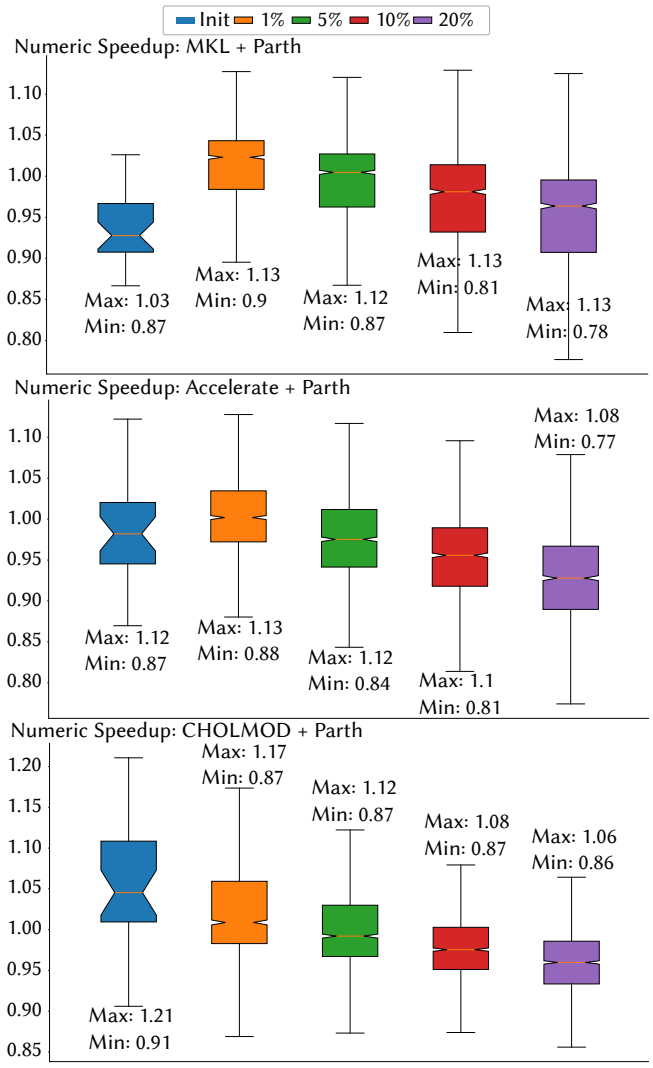


Fig. 26. **Remeshing: Numerical performance of using Parth.** The figure shows comparable numerical performance of Parth-integrated solvers when remesher is applied for different patch sizes. Parth shows comparable performance (in terms of median) for initialization. For MKL, Accelerate and CHOLMOD, we can see that the median performance is close to one. Furthermore, we can see that the performance of numerical computation for 1% and 5% is even better that initialization step for Accelerate and MKL. However, for 20% patch sizes, we can see the performance redcues more that others as expected. However, even in this case, the median of speedup for MKL, Accelerate and CHOLMOD is x,y,z respectively which is close to 1.

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 25 for an example). To alleviate this problem, Parth uses Algorithm 7 to provide reuse even in these scenarios.

Algorithm 7 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.

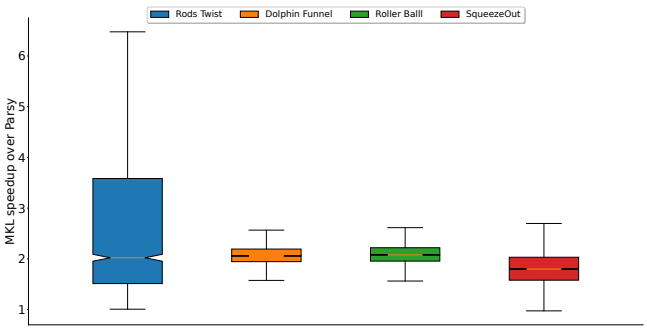


Fig. 27. **MKL Vs. Parsy Evaluation:** This whisker plot shows the distribution of MKL speedup over Parsy for 4 simulation in IPC benchmark. As it is shown, due to MKL constant development on Intel, MKL numerical performance is now signficantly faster than Parsy which is the combination of LBC algorithm in [Cheshmi et al. 2018a] and code generation in [Cheshmi et al. 2017].

F REMESHING: FILL-REDUCING QUALITY ANALYSIS

Looking at Figure 26, or the initialization stage, Parth's performance is comparable to the state-of-the-art fill-reducing ordering algorithms as expected. Comparing the median line of whisker plots, for 1% changes, this performance is even better when Parth is integrated into Accelerate and MKL. However, since fill-reducing ordering is a global operation and Parth integrates those locally, by remeshing the bigger patch, the performance starts to reduce. However, even in 20% changes, we can see that the median of Parth speedup is around 0.95x. Considering the significant performance gained in the symbolic stage, this small draw-back is acceptable as it is shown for per-solve performance in 5.

G PARSY AND EIGEN PERFORMANCE

Figure 27 shows how the numerical performance of MKL is now faster than Parsy by an average of 2.7x, 2.08x, 2.09x, 1.84x on Rode Twist, Dolphin Funnel, Roller Ball and Squeeze Out simulation on IPC [Li et al. 2020] benchmark. This evaluation shows that not only MKL cached up to the Sympiler library which uses both Sympiler [Cheshmi et al. 2017] and Parsy [Cheshmi et al. 2018a] contributions, but it also significantly outperforming them due to its constant development with state-of-the-art hardwares. Due to this reason, we decide to develop Parth so it can be integrated into these solver and provide performance benefits on top of the performance benefits gained by these high-performance linear solver.