Eurographics Symposium on Parallel Graphics and Visualization (2024)
D. Pugmire, G. Reina, S. Rizzi (Editors)

An Accelerated Clip Algorithm for Unstructured Meshes: A Batch-Driven Approach

Spiros Tsalikis [1,2], Will Schroeder [1], Daniel Szafir [2] and Kenneth Moreland [3]

[1]Scientific Computing, Kitware, Inc., USA
[2]Department of Computer Science, University of North Carolina at Chapel Hill, USA
[3]Oak Ridge National Laboratory, USA

Abstract

The clip technique is a popular method for visualizing complex structures and phenomena within 3D unstructured meshes. Meshes can be clipped by specifying a scalar isovalue to produce an output unstructured mesh with its external surface as the isovalue. Similar to isocontouring, the clipping process relies on scalar data associated with the mesh points, including scalar data generated by implicit functions such as planes, boxes, and spheres, which facilitates the visualization of results interior to the grid. In this paper, we introduce a novel batch-driven parallel algorithm based on a sequential clip algorithm designed for high-quality results in partial volume extraction. Our algorithm comprises five passes, each progressively processing data to generate the resulting clipped unstructured mesh. The novelty lies in the use of fixed-size batches of points and cells, which enable rapid workload trimming and parallel processing, leading to a significantly improved memory footprint and run-time performance compared to the original version. On a 32-core CPU, the proposed batch-driven parallel algorithm demonstrates a run-time speed-up of up to 32.6x and a memory footprint reduction of up to 4.37x compared to the existing sequential algorithm. The software is currently available under an open-source license in the VTK visualization system.

CCS Concepts
• ***Computing methodologies*** $\rightarrow$ ***Shared memory algorithms;*** • ***Theory of computation*** $\rightarrow$ ***Computational geometry;***

1. Introduction

Current computing trends are driven by the increasing size of data and the evolution of parallel computing systems. However, many popular algorithms were originally designed for sequential computing models, which may reduce their effectiveness in modern applications. For example, Marching cubes [LC87, NY06], a well-known isocontouring algorithm, widely employed in visualization software like VTK [SML06], ParaView [AGL05], and VisIt [CBW*12], inherits sequential design patterns that adversely affect performance. These patterns include building data structures incrementally for inserting output points and cells, incremental memory allocations, and duplication of computational operations to produce intermediate results. These deficiencies prompted the recent development of Flying Edges [SMG15], a performant and scalable algorithm that addresses the computational shortcomings of Marching Cubes.

In this work, our goal is to accelerate the sequential clip algorithm introduced by Meredith et al. [MC10], which shares similar deficiencies with those found in Marching Cubes. This algorithm is designed with the primary goal of reconstructing material interfaces from volume fractions for visualization and analysis purposes. It is capable of processing both discrete and continuous data; however, this paper focuses solely on its continuous version due to its simplicity. Furthermore, it exhibits desirable quality across multiple accuracy metrics, generating smooth surfaces with minimal defects. Thanks to these noteworthy features, this clip algorithm is widely recognized as an important visualization technique for visually exploring complex unstructured meshes.

Contributions: In this paper, inspired by the emergence of new parallel computing models, we introduce a high-performance batch-driven parallel clip algorithm optimized for unstructured meshes. Following the example set by Flying Edges, our proposed algorithm represents a redesign of Meredith's sequential clip algorithm, tailored to exploit modern multi-core hardware. Our parallel multi-pass algorithm has the following novel contributions:

1. Eliminates bottleneck functions, such as coincident point merging, and reduces incremental memory allocations.
2. Minimizes memory accesses by reducing memory footprint, optimizing cache-friendly access, and avoiding unnecessary accesses.
3. Utilizes fixed-size batches of points and cells, enabling rapid workload trimming and parallel processing.
4. Significantly surpasses both Meredith's sequential algorithm and VTK-m's [MSU*16] parallel algorithm in terms of performance and memory footprint. Additionally, it demonstrates enchanced parallel efficiency compared to VTK-m's algorithm.

DOI: 10.2312/pgv.20241130

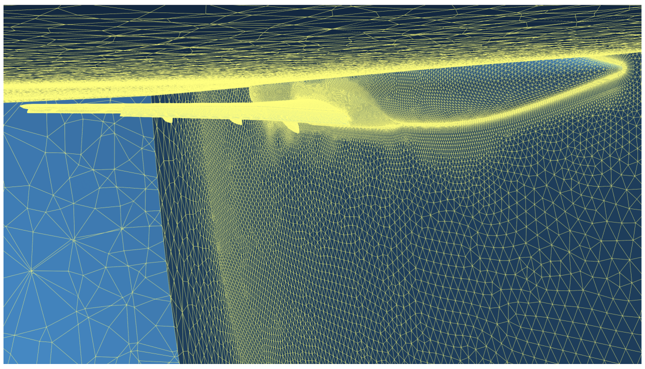


Figure 1: *An interior view of the external surface of the Japan Aerospace Exploration Agency (JAXA) Standard Model (JSM), from the 3rd AIAA CFD High-Lift Prediction Workshop [RSS19], that was clipped in half using a plane.*

In the following sections, we cover related work, define the problem, discuss Meredith's sequential clip algorithm, introduce our batch-driven parallel clip algorithm, evaluate performance and discuss the results, and conclude with future work.

2. Related Work

Prior research has proposed several approaches for implementing a clip algorithm. All of these approaches share the common goal of removing portions of a mesh that are unimportant or interfere with visual inspection or data processing. As Figure 1 illustrates, this may involve removing specific regions of the mesh in space or defining values of a field that should be removed.

Clipping algorithms are typically implemented by processing the cells of a mesh. For each input cell of a mesh, a clip algorithm computes a case index based on the scalar values of its defining points, akin to Marching Cubes, to access a case table. A case table indicates whether a cell produces nothing, the cell itself, or one or more new clipped cells. The output cells reference three types of points: input points, edge point intersections defined by two input points, and points located at the centroid of input cells.

The VTK library's [SML96] first clip algorithm, which is still available today within the `vtkClipDataSet` class, has a single pass. During this pass, the algorithm fully extracts each input cell, computes the case index to access a case table, and generates output cells and points, incrementally. Notably, a point locator is employed to identify duplicate points before they are incrementally added. This algorithm exhibits three noteworthy deficiencies:

- *Full Extraction of Each Cell*: The algorithm fully extracts each cell, including not only the point IDs but also its actual points, leading to increased memory accesses. These points are necessary for unique insertion into the point locator.
- *Use of Point Locator*: The algorithm relies on a point locator, introducing additional computational overhead.
- *Incrementally Built Data Structures*: The algorithm relies on incrementally building data structures by inserting cells and points while ensuring unique point insertion, thereby impeding independent processing. This incremental approach significantly affects the algorithm's scalability and prevents parallelization. Additionally, this leads to incremental memory allocations, contributing to potential inefficiencies in memory usage.

A subsequent clip algorithm by Meredith [MC10], initially incorporated into VisIt [CBW*12] under the `vtkVisItClipper` class and later integrated into the VTK library within the `vtkTableBasedClipDataSet` class, comprises four main passes. First, it extracts the point IDs of each input cell, computes its case index to access a case table, and stores its associated output cells into intermediate type-specific connection arrays with temporary point IDs. In the second pass, the number of edge points and centroid points is known, and the number of kept points is determined by constructing a point map. Third, it extracts the output points. Finally, it traverses each intermediate type-specific connection array, renumbers the point IDs of each cell based on the number of kept points, edge points, and centroid points, and writes them into the output cells. Although this algorithm eliminates the use of a point locator and does not fully extract each cell, it still involves incremental memory allocations due to incremental data structure building. Moreover, it has the following additional deficiencies:

- *Increased Memory Footprint*: The algorithm produces intermediate type-specific connection arrays, nearly doubling the memory requirements for output cells.
- *Computational Overhead*: The algorithm requires duplicating the writing of point IDs for each output cell, introducing computational overhead.

Moreland et al. [MGMM13] analyzed widely employed algorithms in the Scientific Visualization domain, categorizing the clip algorithm within the algorithmic class named *Build Connected Topology*. This class poses a significant challenge for parallelization in Scientific Visualization due to its technical complexity, notably stemming from a lack of prior knowledge regarding the output size. Moreland's analysis sparked the collaborative creation of the VTK-m library [MSU*16]. VTK-m provides a collection of algorithms originally found in the VTK library, specifically designed to support various many-core processors, such as CPUs and GPUs.

VTK-m's clip algorithm, implemented within the `vtkmClip` class, is parallel by design, leveraging simple and reusable parallel primitives as presented by Moreland et al. [MMP*21]. These primitives significantly reduce developer costs. The algorithm achieves independent processing through multiple passes. Instead of generating intermediate type-specific connection arrays with temporary point IDs, for each cell several accumulators are used to compute cell statistics. These statistics are then transformed into offsets using a parallel prefix-sum operation, ensuring that the output can be written once with the correct point IDs. Although this algorithm effectively addresses the previous limitations of both sequential algorithms and is inherently parallel, even for GPUs, it exhibits two new important deficiencies:

- *Increased Memory Footprint*: The algorithm's storage of cell statistics significantly increases the memory footprint, as these statistics are stored for each cell.
- *Computational Overhead*: The prefix sum computation, converting the accumulators to offsets, is expensive since it is applied to all cells instead of batches of cells.

3. Problem Definition

Let M be an unstructured mesh defined as a set of cells C of size N_C, connecting points in set P of size N_P. Each cell c_i is defined by its type t_i and point connections $con_{c_i,0}, con_{c_i,1}, \ldots$. As depicted in Figure 2, C is constructed using a "Types" array T of size N_C, containing the type of each cell, and a "Connections" array CON of size N_{CON}, containing the point indices to which each cell connects. To find the connections for a given cell, a third "Offsets" array OF is constructed. Each of_i contains the index in CON for the connections of cell ci, and its points P_{c_i} have size $NP_{c_i} = of_{i+1} - of_i$. The size of OF is $N_{OF} = N_C + 1$ with its last value $of_{N_C} = N_{CON}$.

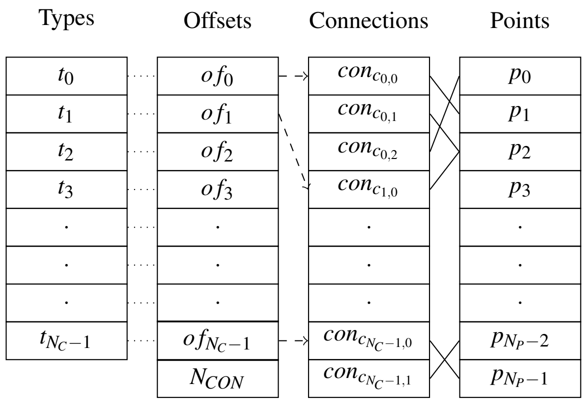


Figure 2: *The data structures used to represent an unstructured mesh.*

The input of a clip algorithm is an unstructured mesh M with scalars S associated with P. S can either be a scalar field, such as density, or generated from P using an implicit function $IF(x)$, e.g., a plane, a box, or a sphere. The points P_{c_i} of cell c_i have associated scalar values S_{c_i} of size NP_{c_i}.

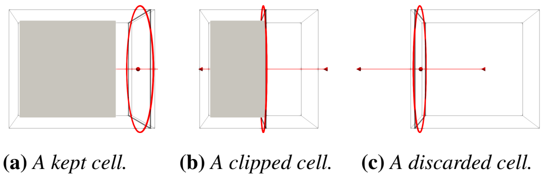


Figure 3: *The three high-level cell cases of a clip algorithm, using a plane implicit function.*

Given an isovalue v, a clip algorithm evaluates each cell c_i using its scalars S_{c_i}. As shown in Figure 3, this evaluation determines if a cell c_i is kept, clipped, or discarded. c_i is kept if all of S_{c_i} are above v. c_i is discarded if none of S_{c_i} are above v. c_i is clipped if it is neither kept nor discarded, i.e., some scalars are above v and some are not.

When a cell is kept, its output points mirror its input points. However, when a cell is clipped, new points must be constructed to represent the new geometry. Two types of points may be generated (see Figure 4). Edge points are constructed on an edge of the input mesh and are defined by the two input points of the edge and an interpolation weight. These points, along with points passed directly from the input, are generally shared among multiple output cells. One centroid point, which is defined by input or edge points, may be constructed within the interior of an input cell. Centroid points are exclusive to output cells generated by a specific input cell. They are required, in certain cases, for triangulating the following 3D cell types: voxels, hexahedrons, wedges, and pyramids.

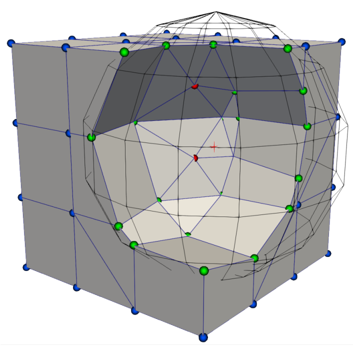


Figure 4: *Example of the point types that clip algorithms can generate: edge points (green) and centroid points (red). Output cells are also constructed from points passed from the input (blue).*

The output of a clip algorithm is an unstructured mesh M', defined with cells C' and points P' using the same representation demonstrated in Figure 2. P' incorporates three sets of points: kept points KP which are a subset of P, edge points E, and centroid points CEN of sizes N_{KP}, N_E, and N_{CEN}, respectively.

3.1. Algorithmic Constraints, Terminology and Data Structures

The clip algorithms discussed in this paper are designed to process an input mesh M consisting exclusively of first-order cells. These cells adhere to the condition $NP_{c_i} \leq 8$ and belong to one of the following types: vertex, line, triangle, pixel, quadrilateral, tetrahedron, voxel, hexahedron, wedge, and pyramid.

Cells are clipped in much the same way as Marching Cubes [LC87] slices cells. A case index array CI of size N_C is constructed by categorizing each cell's incident points as either above or below v, and then combining them to form a unique *case index* ci_i for the tessellation. Since only first-order cells with $NP_{c_i} \leq 8$ are supported, the case index ci_i of c_i can be represented by 8 bits to capture all 2^8 possible cases, where each bit of ci_i is set as $ci_{i,j} = s_{c_{i,j}} \geq v$.

The case index ci_i of a cell c_i is used to access a clip case table, akin to the Marching Cubes case tables, which facilitate efficient access to information such as the number and types of output shapes, the number of points for each shape, and the types of points within each shape. In this context, shapes encompass both the resulting output cells and optionally a centroid point used to define them.

The computation of kept points KP requires the creation of a point map M_P of size N_P. If $m_{p_i} \geq 0$, then p_i is part of KP and m_{p_i} is its point ID in P', else if $m_{p_i} = -1$, then p_i is discarded. The computation of E uses an edge locator EL that identifies and combines shared edges in adjacent cells. It provides a unique index for an edge's pair of point IDs and stores the linear interpolation weight used to compute coordinates and other fields at this location. The computation of CEN requires the storing of the point IDs defining a centroid point in an input cell.

4. Existing Algorithm: Sequential Clip

As described in Section 2, Meredith's clip algorithm comprises four passes. Algorithm 1 provides a high-level description of its passes.

Algorithm 1 Sequential Clip

```
Input: Unstructured mesh $M$ with points $P$, scalars $S$, cells $C$, and isovalue $v$.
Output: Clipped unstructured mesh $M'$ with points $P'$ and cells $C'$.
1: Pass 1: Evaluate cells and extract output cells into intermediate
   type-specific connection arrays with temporary point IDs.
   Determine number of output cells, edge points, and centroid points.
2: Pass 2: Define a point map and determine the number of kept points.
3: Pass 3: Extract kept points, edge points, and centroid points.
4: Pass 4: Extract the output cells by renumbering the point IDs
   of the intermediate type-specific connection arrays.
```

4.1. Pass 1: Evaluate Cells

Initially, the number of kept points N_{KP}, edge points N_E and centroid points N_{CEN}, is not known. To characterize intermediate output cells' temporary point IDs, three ID segments are defined: $0 \leq ID < N_P$ (input point), $ID \geq N_P$ (edge point), and $ID < 0$ (centroid point).

In the first pass, the cells are traversed. For each cell c_i, its case index ci_i is computed using S_{c_i} to access a case table. Then, the number of output shapes is extracted, and the shapes are traversed.

For each output shape, its points are traversed, and their IDs are defined as follows: If it is an input point, its temporary ID remains the same as its input point. If it is an edge point, it is inserted into the edge locator EL; its unique edge ID is retrieved, and the temporary edge point's ID is set as $ID \leftarrow N_P + ID$. If it is a centroid point, the temporary centroid point's ID is set as $ID \leftarrow -1 - N_{CEN}$, where N_{CEN} represents the current size of CEN.

After defining the point IDs, if the output shape is the special centroid, the centroid is defined using the IDs and incrementally inserted into CEN. For other shape types, the cell type is extracted, and the IDs are incrementally inserted into an intermediate type-specific connection array. Finally, upon completion, $N_{C'}$ (including $N_{CON'}$ and $N_{OF'}$), N_E, and N_{CEN} become known.

4.2. Pass 2: Define Point Map

To compute KP, a point map M_P of size N_P is initialized to -1, and a $counter \leftarrow 0$ is created. Next, the output point IDs of each cell in the intermediate type-specific connection arrays are traversed. If $0 \leq ID < N_P$, and $mp_{P_{ID}} = -1$, $mp_{P_{ID}} \leftarrow counter$ is set, and the counter is incremented. At the end of this loop, $N_{KP} \leftarrow counter$ and KP is derived from the non-negative entries in M_P.

4.3. Pass 3: Extract Output Points

In the third pass, the output points P' are allocated, and the following types of points are inserted: the kept points KP, the edge points E that are created using their two point IDs and their linear interpolation weight, and the centroid points CEN.

4.4. Pass 4: Extract Output Cells

In the fourth pass, the point IDs of each cell in the intermediate type-specific connection arrays are renumbered and written into the output cells C'. This renumbering is possible since N_{KP}, N_E and N_{CEN} are now known. First, the output cells C' of size $N_{C'}$ are allocated. To populate C', each intermediate type-specific connections array is traversed. For each cell, its type is inserted in T', and its point IDs are renumbered as follows and written in CON' by defining OF': If $0 \leq ID < N_P$ (input point), then $ID \leftarrow mp_{P_{ID}}$. If $ID \geq N_P$ (edge point), then $ID \leftarrow N_{KP} + ID - N_P$. If $ID < 0$ (centroid point), then $ID \leftarrow N_{KP} + N_E - ID - 1$.

5. Proposed Algorithm: Batch-Driven Parallel Clip

The proposed batch-driven parallel clip algorithm comprises five passes aimed at enhancing parallel execution and eliminating all deficiencies mentioned in Section 2. Algorithm 2 provides a high-level description of its passes. Subsequent subsections offer comprehensive details on batch definition, detailed descriptions of each pass, and implementation details regarding workload trimming, memory footprint and access, load balancing, and parallel execution.

Algorithm 2 Batch-Driven Parallel Clip

```
Input: Unstructured mesh $M$ with points $P$, scalars $S$, cells $C$, and isovalue $v$.
Output: Clipped unstructured mesh $M'$ with points $P'$ and cells $C'$.
1: Pass 1: Evaluate batches of points and associated scalars
   against the isovalue, defining a point map, and determining the
   number of kept points.
2: Pass 2: Evaluate batches of cells, collect edge points, and
   determine the number of centroid points, number of output
   cells, output connections' size, case indices, and batch-related
   information enabling the writing of cells in Pass 4.
3: Pass 3: Create the edge locator.
4: Pass 4: Extract output cells and define the centroid points.
5: Pass 5: Extract kept points, edge points, and centroid points.
```

5.1. Batch Definition

Assume a set A, such as points or cells, of size N_A. A can be partitioned into batches B_A with each batch having a fixed size of S_B, except the last batch containing the remainder. Each batch b_{A_k} can have extra variables, such as accumulators, storing statistics related to the algorithm's processing primitive and used for generating output.

Batches can be efficiently removed or trimmed if they meet specific criteria, such as having one of their accumulator variables equal to 0, thereby enabling workload trimming. The accumulator variables of a batch, which is part of a collection of batches, can be leveraged to calculate global summations. Furthermore, these accumulator variables can be efficiently transformed in-place to offsets using a parallel prefix-sum.

The configurability of the batch size S_B, which is evaluated in Section 6.2, significantly affects both run-time performance and memory footprint, showcasing noticeable enhancements or deterioration depending on the dataset size.

5.2. Pass 1: Define Point Map

The batches of points B_P have a size of N_{B_P}. Each batch b_{P_k} has an accumulator variable for the number of kept points. Initially, the batches of points are traversed in parallel. For each scalar value s_i of a point p_i in batch b_{P_k}, if $s_i \geq v$, then $m_{p_i} \leftarrow 1$, else $m_{p_i} \leftarrow -1$. If $m_{p_i} = 1$, then the accumulator is incremented. Upon completion, the accumulator contains the number of kept points in the batch.

Subsequently, the batches whose accumulator of kept points is zero are trimmed. For the remaining batches, a parallel prefix sum of the accumulated kept points count is computed. The summed values contain the offset in the output points array for the kept points, and the total summation is N_{KP}.

Finally, the remaining batches of points are traversed in parallel again. Before traversing each batch b_{P_k}, a $counter \leftarrow$ offset is created. For each p_i in b_{P_k}, if $m_{p_i} = 1$, then $m_{p_i} \leftarrow counter$ and the counter is incremented. Eventually, an M_P equivalent to that generated in the sequential algorithm is defined, and KP is determined.

5.3. Pass 2: Evaluate Cells

The batches of cells B_C have a size of N_{B_C}. Each batch b_{C_k} contains three accumulator variables: output connections' size, number of output cells, and number of centroid points. Initially, the batches of cells are traversed in parallel. For each cell c_i in each batch b_{C_k}, its case index ci_i is computed using S_{c_i} to access a case table. Then, the number of output shapes is extracted, and the shapes are traversed.

For each output shape, its points are traversed. Any edge point is inserted into thread-local edge points. Afterwards, if the output shape is the special centroid, the number of centroid points in b_{C_k} is incremented. For other shape types, the number of output cells and their output connections' size in b_{C_k} are incremented.

Once the batch processing is completed, the batches whose number of output cells accumulator is zero are trimmed. Subsequently, the global summation of all accumulators, including $N_{CON'}$, $N_{C'}$, and N_{CEN}, is computed, and the accumulators of each batch are converted to offsets. Finally, the thread-local edge points are merged.

5.4. Pass 3: Create Edge Locator

The merged edge points undergo parallel sorting, prioritizing the first point's ID and then the second point's ID. This sorting facilitates the identification of duplicate edge points through sequential traversal of the sorted edges, resulting in the formation of the unique edge points E of size N_E. The EL is created by defining beginning and ending offsets for each first point ID defining a static hash map. Given the two point IDs defining an edge, EL enables the swift retrieval of a unique edge point ID.

5.5. Pass 4: Extract Output Cells

First, output arrays for CON', OF', and T' are allocated using the value of $N_{CON'}$ and $N_{C'}$ computed in Pass 2. Likewise, an array CEN to hold centroid points is allocated using N_{CEN}. Then, the trimmed batches of cells are traversed in parallel. For each cell c_i in b_{C_k}, its case index ci_i is used to access a case table. Then, the number of output shapes is extracted, and the shapes are traversed.

For each output shape, its points are traversed, and their IDs are defined as follows: if it is an input point, its ID is set as $ID \leftarrow m_{p_{ID}}$; if it is an edge point, its unique ID is retrieved using the edge locator EL and set as $ID \leftarrow N_{KP} + ID$; and if it is a centroid point, its ID is determined using the offset of the number of centroids' accumulator in b_{C_k} and set as $ID \leftarrow N_{KP} + N_E + ID$.

After defining the point IDs, if the output shape is the special centroid, the centroid is defined using the point IDs and is inserted into CEN. For other shape types, the cell type is extracted, and point IDs, offset, and cell type are written to CON', OF', and T', respectively, at the correct memory position. This is facilitated by the offsets of the output connections' size and the number of output cells accumulators in b_{C_k}, computed at the end of Pass 2. All offsets are then appropriately incremented. Finally, the last offset value is assigned as $of'_{N_{C'}} = N_{CON'}$.

5.6. Pass 5: Extract Output Points

As described in Section 3, there are three types of points: kept points, edge points, and centroid points. With the completion of Pass 4, all necessary information is available to define the output points P'. Firstly, the output points P' of size $N_{P'} = N_{KP} + N_E + N_{CEN}$ are allocated. Afterwards, the trimmed batches of input points are traversed in parallel. For each point p_i in b_{P_k}, if $m_{p_i} > 0$, it is written to P' at position m_{p_i}. Subsequently, the edges E are traversed in parallel. For each edge e_i with two point IDs and a linear interpolation weight, the edge point is constructed and written to P' at position $N_{KP} + i$. Finally, the centroid points CEN are traversed in parallel. For each centroid cen_i, the centroid point is constructed and written to P' at position $N_{KP} + N_E + i$.

5.7. Workload Trimming

As most cells in a clip operation are either fully kept or entirely discarded, utilizing trimming techniques whenever feasible can significantly reduce the workload. To that end, the following two optimizations are implemented:

- *Trim Batches*: In Passes 1, 2, 4, and 5, the algorithm iterates over batches of points or cells. One motivation for this approach is to enable trimming point batches with no kept points in Pass 1 and cell batches that yield no output in Pass 2. As a result, in the second part of Pass 1, in Pass 4, and in Pass 5, fewer batches of points and cells are traversed, respectively, whenever possible.
- *Store Case Index*: In Pass 2, we save the 8-bit case index of each cell. Although this requires additional memory, in Pass 4 it enables the quick determination of whether a cell is kept, clipped or discarded without recomputing it by re-accessing the scalars, and therefore, enabling quick cell trimming.

5.8. Memory Footprint and Access

The clip algorithm is not inherently computationally expensive; therefore, its performance and scalability are constrained by memory bandwidth. It is crucial to minimize memory accesses by reducing the memory footprint and ensuring cache-friendly access or avoiding memory access whenever possible. To achieve this goal, the following five optimizations are implemented:

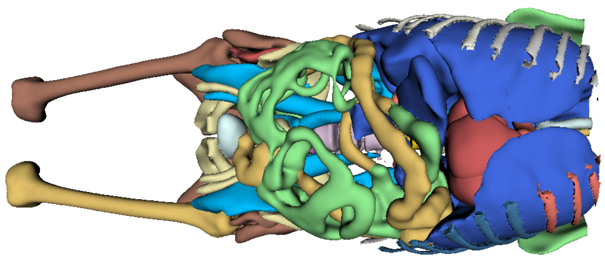
- *Optimize Point Map Computation*: Instead of constructing the point map by traversing the point IDs of output cells to determine if they use an input point, as done in Meredith's and VTK-m's clip algorithms, our proposed algorithm traverses input points. It determines whether they are kept based on their scalar value, as described in Section 5.2. Traversing the cell connectivity is less cache-friendly because it writes values in the point map in a non-sequential order. Furthermore, points shared by adjacent cells will access the same point map entries multiple times.
- *Use 32-bit IDs*: If $N_C \leq 2,147,483,647$, the output cells and point map can be constructed using 32-bit IDs, instead of 64-bit IDs, leading to a halved memory footprint for both.
- *Optimize Case Tables*: The case tables used by the sequential clip algorithm, which consist of several arrays per cell type, are packed into one array that can be accessed using an index lookup table. This optimization ensures cache-friendly memory access, especially for input meshes containing more than one cell type.
- *Avoid Accessing Case Tables*: Using the case index, the determination of whether a cell is kept or discarded can be performed without accessing the case tables. Even with the optimized case tables, this improvement proves to be a significant fast-path, since the majority of the input cells are either kept or discarded, and only a small portion is clipped.
- *Minimize Accumulators' Memory Footprint*: In contrast to VTK-m's clip algorithm, the batch-driven parallel algorithm allows batches to store accumulators for a sequence of cells, not just one. The higher the batch size, the greater the memory footprint reduction benefit.

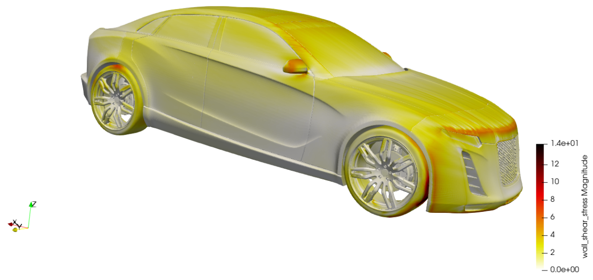
5.9. Load Balancing and Parallel Execution

Load balancing is challenging when the size of the output cannot be determined *a priori* or when the amount of work differs for ostensibly similar tasks. For all parallel traversals, VTK uses `vtkSMPTools` to perform parallel execution. `vtkSMPTools` has four backends: 1) `Sequential`, 2) `std::thread` without load balancing, 3) `OpenMP` using *schedule(runtime)* for load balancing, and 4) `TBB` [Rob11] using *parallel_for*, which manages a thread pool to process tasks and performs work-stealing when necessary. Based on performance analysis of various algorithms, TBB has been chosen as the default backend for `vtkSMPTools` due to its load balancing capabilities. Nonetheless, the backend is configurable at runtime using `vtkSMPTools::SetBackend("BackendName")`.

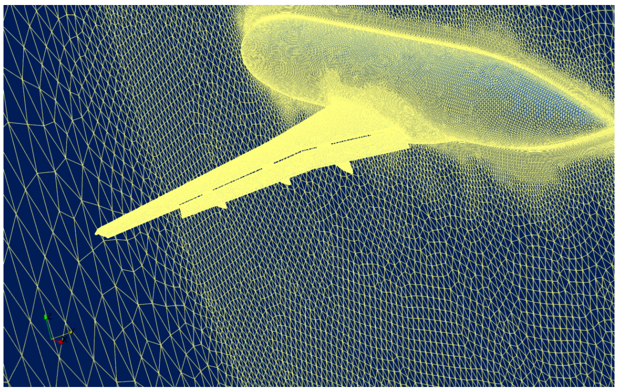
Alternative approaches to enhance load balancing in parallel isocontouring involve domain decomposition using data structures such as octrees, sphere trees, and contour trees. However, the cost of creating such data structures incurs overhead that is not justifiable for the clip algorithm, as the output size can be orders of magnitude larger compared to isocontouring. In contrast, the batch-driven approach, which includes workload trimming, boosts thread utilization by eliminating batches without work, thereby improving parallel scalability.

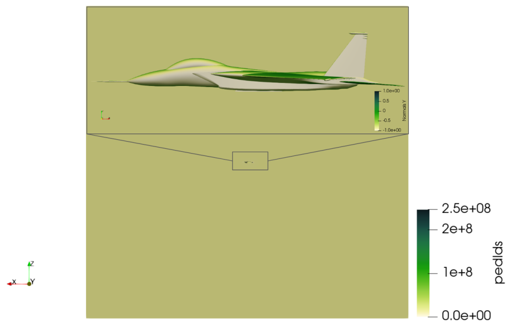
The proposed batch-driven parallel clip algorithm has been seamlessly incorporated into VTK and ParaView, both of which possess MPI awareness. Assuming data have already been distributed across MPI nodes, and since no data dependencies exist between nodes, this algorithm can be effortlessly executed at large scales.


(a) *A Human Torso generated by parallel Surface Nets [STHF24] using as input an AI-generated segmentation from Total Segmentator [WBM*23].*



(b) *The CX-1 Sedan provided by Altair Engineering, Inc.*


(c) *The Japan Aerospace Exploration Agency (JAXA) Standard Model (JSM) from the 3rd AIAA CFD High-Lift Prediction Workshop [RSS19].*



(d) *The fluid volume around an F-15 aircraft, provided by Helden Aerospace, Inc.*

Figure 5: *The datasets used for the performance evaluation.*

6. Performance Evaluation

In this performance evaluation of clip algorithms, we assess the performance of Meredith's sequential algorithm (**clip-seq**) [MC10], VTK-m's parallel algorithm (**clip-m-par**) [MSU*16], and our batch-driven parallel algorithm (**clip-par**). VTK's first sequential algorithm [SML96] is omitted from this evaluation due to its significant slowness. While all algorithms are MPI-aware thanks to VTK's infrastructure, we opted not to use MPI because an algorithm would simply be executed independently on each node. Thus, analyzing beyond the desktop is unnecessary for the scope of this paper.

The performance evaluation is performed on a computing node that has an Intel Xeon Gold 6226R processor with 16 physical and 32 logical cores (including 16 hyper-threads), and 64GB of DDR4 memory. All algorithms are executed within a Docker container, utilizing the GCC compiler 13.2.0, CMake 3.27.0, and TBB 2021.9 for a standardized evaluation environment (see Appendix A).

6.1. Methods

We evaluated the clip algorithms' performance using four datasets (see Table 1 and Figure 5) with varied sizes and cell distributions. For each clip algorithm execution, a dataset is clipped using a plane implicit function with a normal vector $(1,0,0)$. Its origin is centered on the dataset's bounds along the y and z axes, while the percentage variable Per controls its position along the x-axis, enabling the creation of output meshes with different sizes. For Per values equal to 20%, 50%, and 80%, we aim to clip most, half, and a small fraction of the dataset, respectively. The run-time and memory footprint are averaged over ten algorithm executions, excluding dataset reading.

Table 1: *Datasets' input size and output size for {20%, 50%, 80%} Per values. M stands for millions and K for thousands.*

Dataset			Torso	CX-1	JSM	F-15
Input		N_P	1.4 M	8.2 M	50.4 M	116.4 M
		N_C	2.9 M	16.2 M	120.0 M	246.1 M
Per	20%	$N_{P'}$	0.2 M	1.2 M	0.9 K	3.1 K
		$N_{C'}$	0.4 M	2.4 M	2.0 K	9.0 K
Per	50%	$N_{P'}$	0.7 M	3.1 M	35.3 M	91.0 M
		$N_{C'}$	1.5 M	6.2 M	83.0 M	192.0 M
Per	80%	$N_{P'}$	1.2 M	5.3 M	50.4 M	116.4 M
		$N_{C'}$	2.5 M	10.4 M	120.0 M	246.0 M

6.2. Batch Trimming and Size

Figure 6 clearly demonstrates the significant impact of trimming on the run-time of the clip-par algorithm across different batch sizes. Notably, the effects of batch trimming are most clearly demonstrated at $Per = 20\%$, highlighting its substantial efficacy. While the median run-time differences with and without trimming are smaller at $Per = 50\%$ and $Per = 80\%$, since they generate more output cells, the reduction in run-time variance is noteworthy. Both aforementioned observations support that batch trimming enhances clip-par's algorithm performance. Therefore, batch trimming is consistently enabled in all subsequent executions.

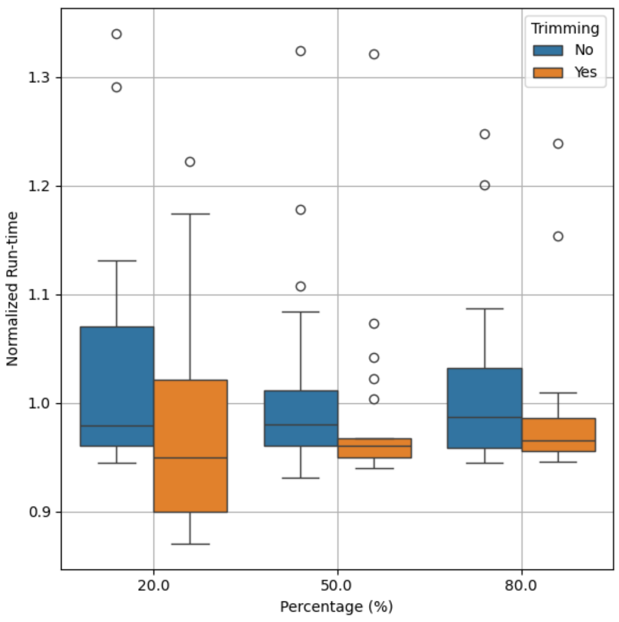


Figure 6: *Clip-par algorithm's normalized run-time distributions of {20%, 50%, 80%} Per values with and without batch trimming for varying batches, evaluated using the F-15 dataset, and employing 32 threads. Each run-time is divided by the average run-time with and without trimming for every Per value.*

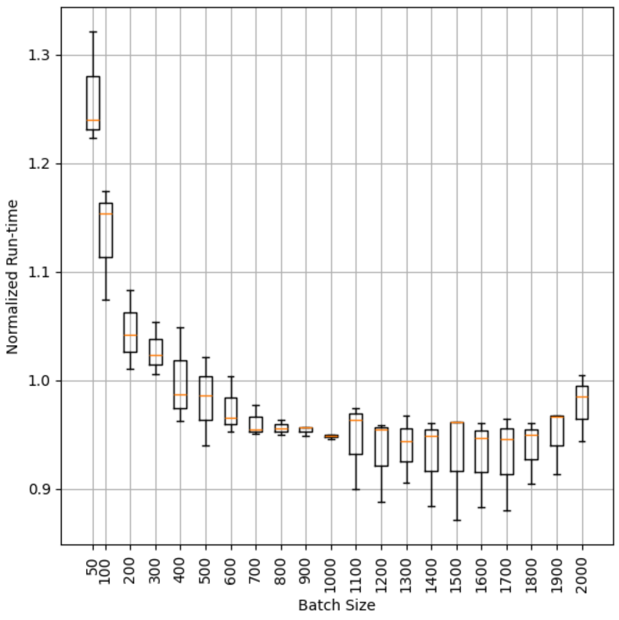


Figure 7: *Clip-par algorithm's normalized run-time distributions of varying batch sizes, with trimming, constructed using the normalized run-times of {20%, 50%, 80%} Per values, evaluated using the F-15 dataset, and employing 32 threads. Each run-time is divided by the average run-time with and without trimming for every Per value.*

Table 2: *The run-time of clip-seq, clip-m-par, and clip-par algorithms evaluated using the datasets shown in Figure 5 with {20%, 50%, 80%} Per values, and employing 1, 16 and 32 threads (T).*

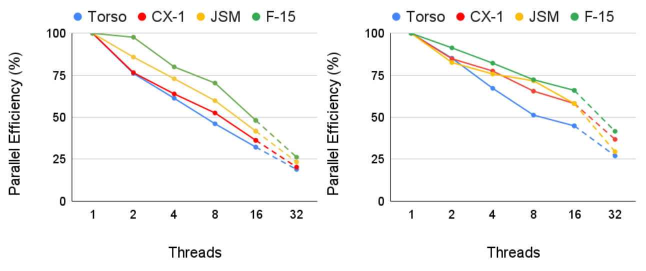
			Torso		CX-1		JSM		F-15	
Algorithm	T	Per	Time (ms)	Speed-up	Time (ms)	Speed-up	Time (ms)	Speed-up	Time (ms)	Speed-up
clip-seq	1	20%	142	1.00	980	1.00	5,357	1.00	11,285	1.00
clip-m-par	1	20%	352	0.40	2,238	0.44	14,720	0.36	30,402	0.37
clip-m-par	16	20%	82	1.73	432	2.27	2,688	1.99	5,497	2.05
clip-m-par	32	20%	71	2.00	414	2.37	2,369	2.26	5,010	2.25
clip-par	1	20%	100	1.42	542	1.81	3,233	1.66	6,813	1.66
clip-par	16	20%	15	9.47	66	14.85	389	13.77	687	16.43
clip-par	32	20%	**11**	**12.91**	**54**	**18.15**	**385**	**13.91**	**652**	**17.31**
clip-seq	1	50%	246	1.00	1,547	1.00	18,347	1.00	44,770	1.00
clip-m-par	1	50%	495	0.50	3,116	0.50	30,621	0.60	81,135	0.55
clip-m-par	16	50%	109	2.26	560	2.76	5,206	3.52	11,151	4.01
clip-m-par	32	50%	93	2.65	513	3.02	4,641	3.95	10,081	4.44
clip-par	1	50%	132	1.86	805	1.92	7,397	2.48	19,068	2.35
clip-par	16	50%	21	11.71	92	16.82	888	20.66	1,854	24.15
clip-par	32	50%	**15**	**16.40**	**80**	**19.34**	**820**	**22.37**	**1,497**	**29.91**
clip-seq	1	80%	363	1.00	2,169	1.00	25,940	1.00	55,225	1.00
clip-m-par	1	80%	637	0.57	3,876	0.56	40,260	0.64	93,661	0.59
clip-m-par	16	80%	124	2.93	670	3.24	6,044	4.29	12,180	4.53
clip-m-par	32	80%	106	3.42	600	3.62	5,412	4.79	11,210	4.93
clip-par	1	80%	172	2.11	1,116	1.94	8,307	3.12	22,508	2.45
clip-par	16	80%	24	15.13	120	18.08	894	29.02	2,133	25.89
clip-par	32	80%	**20**	**18.15**	**95**	**22.83**	**884**	**29.34**	**1,694**	**32.60**

Figure 7 clearly demonstrates the significant impact of choosing the appropriate batch size on the run-time of the clip-par algorithm for different *Per* values with trimming enabled. Notably, very small batch sizes lead to a substantial performance overhead. This stems from the fact that the size of the accumulators increases, and therefore, more memory is accessed to compute the prefix-sums and global summations of the accumulators. When analyzing the ideal batch size, it is evident that the run-times follow a quadratic curve. While several batches with sizes greater than 1000 lead to similar median run-times, the variance of their distribution is larger. Notably, a batch size of 1000 seems to be the optimal point on this curve, demonstrating the least variability across different *Per* values. Consequently, the optimal batch size of 1000 is chosen and consistently used in all subsequent executions.

6.3. Run-time Performance

Table 2 showcases the run-time performance of clip-seq, clip-m-par, and clip-par algorithms. Notably, clip-par stands out with significant speed-ups over clip-seq of up to 3.12x on just 1 thread and up to 32.60x on 32 threads. This enhancement is credited to optimizations detailed in Sections 5.7 and 5.8 and is underscored by clip-seq's limitations in Section 2. Conversely, clip-m-par shows a noticeable slowdown compared to clip-seq with a single thread, with suboptimal speed-ups from 0.36x to 0.64x. Despite this, thanks to its parallel capabilities, clip-m-par achieves a noteworthy speed-up of up to 4.93x. The relative slowness of clip-m-par compared to clip-seq with a single thread is attributed to the usage of accumulators per cell. Overall, clip-par is always faster than clip-m-par by up to 7.68x on the same number of threads and faster than clip-seq on a single thread by up to 3.12x.

Overall, as illustrated in Figure 8, it is evident that clip-par demonstrates enhanced parallel efficiency compared to clip-m-par. With 16 and 32 threads, clip-par achieves efficiencies of 66% and 42%, respectively. The inability to reach 100% efficiency is attributed to load balancing challenges, as discussed in Section 5.9, and memory bandwidth constraints, as noted in Section 5.8. The drop in parallel efficiency from 66% to 42% is expected due to the additional



(a) *Clip-m-par's parallel efficiency.* **(b)** *Clip-par's parallel efficiency.*

Figure 8: *The parallel efficiency of the clip-m-par and clip-par algorithms evaluated using the datasets shown in Figure 5 with 80% Per value, and employing 1, 2, 4, 8, 16 and 32 threads. The dashed lines represent the usage of 16 additional hyper-threads.*

16 threads being hyper-threads. Also, a discernible trend emerges: parallel efficiency declines as the number of threads increases. However, it is important to highlight that parallel efficiency improves with larger datasets.

Table 3 provides a breakdown of the run-time for each pass of the clip-par algorithm. Notable, since *Per* directly influences the amount of output to extract, and Passes 4 and 5 extract the output, it's understandable that they exhibit high sensitivity to *Per*. Moreover, Pass 3 emerges as the least resource-intensive pass, because $N_E << N_P, N_C$. Lastly, the implicit function computation and Pass 1 demonstrate comparatively lower parallel efficiency.

Table 3: *The run-time of each pass (P) of the clip-par algorithm, including the scalar point data computation with the plane implicit function ($IF(x)$), evaluated using the F-15 dataset with {20%, 50%, 80%} Per values, and employing 1 and 32 threads (T).*

		Time (ms)					
T	Per	IF(x)	P1	P2	P3	P4	P5
1	20%	553	276	6,178	37	1	1
32	20%	140	38	382	41	1	1
1	50%	549	361	6,354	76	9,655	1,881
32	50%	117	52	400	50	597	171
1	80%	621	383	6,104	35	12,158	2,280
32	80%	116	57	358	42	667	205

6.4. Memory Footprint

Table 4 demonstrates the substantial reduction in memory footprint achieved by the clip-par algorithm, which is up to 4.37 times less than that of clip-seq, thanks to optimizations explained in Section 5.8. In contrast, the memory footprint of clip-m-par exhibits a distinct pattern, increasing by up to one order of magnitude for low *Per* values and by 1.73x for high *Per* values compared to clip-seq. In summary, clip-par consistently maintains the smallest memory footprint under most conditions and under the largest memory conditions.

Table 4: *The memory footprint of clip-seq, clip-m-par, and clip-par algorithms evaluated using the datasets shown in Figure 5 with {20%, 50%, 80%} Per values. The memory footprint is the same regardless of how many threads are being utilized.*

		Memory (megabytes)			
Algorithm	Per	Torso	CX-1	JSM	F-15
clip-seq	20%	92	581	**588**	**1,349**
clip-m-par	20%	388	4,946	13,743	28,156
clip-par	20%	**29**	**230**	695	1,664
clip-seq	50%	196	1,058	13,375	33,294
clip-m-par	50%	447	1,925	15,991	35,914
clip-par	50%	**53**	**417**	**3,295**	**8,640**
clip-seq	80%	302	1,592	19,008	40,370
clip-m-par	80%	523	1,969	19,715	41,788
clip-par	80%	**80**	**624**	**4,348**	**10,573**

7. Conclusions and Future Work

We have designed a batch-driven parallel clip algorithm optimized for clipping unstructured meshes with a continuous point-associated set of scalar data that exhibits high performance and low memory footprint. The inspiration for developing this algorithm comes from the advent of new parallel computing models and the realization that many algorithms need to be redesigned to take full advantage of modern multi-core hardware. Our parallelized clip algorithm operates over fixed-size batches of points and cells. Batches enable rapid workload trimming and parallel processing, leading to a significantly improved run-time performance and memory footprint compared to Meredith's sequential algorithm and VTK-m's parallel algorithm.

In the future, when batch processing capabilities are introduced into VTK-m, we aim to incorporate our proposed batch-driven clipping algorithm, thereby harnessing GPU utilization within the framework. Additionally, our proposed batching technique, along with several concepts from the Flying Edges algorithm, could be potentially used to design an efficient clipping algorithm tailored for structured meshes, such as medical images. Finally, we plan to enhance the parallel Surface Nets discrete isocontouring algorithm [STHF24] by integrating batch processing. We anticipate that this enhancement will substantially improve the processing of segmented medical images that are sparse.

We have adhered to the principles of reproducible science by integrating our batch-driven parallel algorithm into the Open-Source VTK library [SML06]. The code has been included in a later version of the `vtkTableBasedClipDataSet` class. For details regarding the versions of VTK used, please refer to Appendix A.

8. Acknowledgements

The implementation of this work was supported by the Data Analysis and Assessment Center (DAAC) of the United States Department of Defense High-Performance Computing Modernization Program.

This work was also supported in part by the U.S. Department of Energy (DOE) RAPIDS SciDAC project under contract number DE-AC05-00OR22725 and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government. Portions of this work is funded through a contract with Sandia National Laboratories. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Special acknowledgments are extended to Altair Engineering, Inc. and Helden Aerospace, Inc. for their support and provision of datasets used for the performance evaluation of the proposed algorithm. Finally, the authors would like to express their appreciation for the insightful feedback from Kevin Garner, Christos Tsolakis, and Cory Quammen during the review process, which greatly improved the rigor and quality of this research work.

References

[AGL05] AHRENS J., GEVECI B., LAW C.: Paraview: An end-user tool for large-data visualization. *Energy* (2005), 717–731. doi:10.1016/B978-012387582-2/50038-1. 1

[CBW*12] CHILDS H., BRUGGER E., WHITLOCK B., MEREDITH J., AHERN S., PUGMIRE D., BIAGAS K., MILLER M. C., HARRISON C., WEBER G. H., KRISHNAN H., FOGAL T., SANDERSON A., GARTH C., BETHEL E. W., CAMP D., RUBEL O., DURANT M., FAVRE J. M., NAVRATIL P.: VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data, October 2012. doi:10.1201/b12985. 1, 2

[LC87] LORENSEN W. E., CLINE H. E.: Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph. 21*, 4 (aug 1987), 163–169. doi:10.1145/37402.37422. 1, 3

[MC10] MEREDITH J. S., CHILDS H.: Visualization and analysis-oriented reconstruction of material interfaces. *Computer Graphics Forum 29*, 3 (2010), 1241–1250. doi:10.1111/j.1467-8659.2009.01671.x. 1, 2, 7

[MGMM13] MORELAND K., GEVECI B., MA K.-L., MAYNARD R.: A classification of scientific visualization algorithms for massive threading. In *Proceedings of the 8th International Workshop on Ultrascale Visualization* (New York, NY, USA, 2013), UltraVis '13, Association for Computing Machinery. doi:10.1145/2535571.2535591. 2

[MMP*21] MORELAND K., MAYNARD R., PUGMIRE D., YENPURE A., VACANTI A., LARSEN M., CHILDS H.: Minimizing development costs for efficient many-core visualization using mcd3. *Parallel Computing 108* (2021), 102834. doi:10.1016/j.parco.2021.102834. 2

[MSU*16] MORELAND K., SEWELL C., USHER W., LO L.-T., MEREDITH J., PUGMIRE D., KRESS J., SCHROOTS H., MA K.-L., CHILDS H., LARSEN M., CHEN C.-M., MAYNARD R., GEVECI B.: Vtk-m: Accelerating the visualization toolkit for massively threaded architectures. *IEEE Computer Graphics and Applications 36*, 3 (2016), 48–58. doi:10.1109/MCG.2016.48. 1, 2, 7

[NY06] NEWMAN T. S., YI H.: A survey of the marching cubes algorithm. *Computers & Graphics 30*, 5 (2006), 854–879. doi:10.1016/j.cag.2006.07.021. 1

[Rob11] ROBISON A. D.: *Intel® Threading Building Blocks (TBB)*. Springer US, Boston, MA, 2011, pp. 955–964. doi:10.1007/978-0-387-09766-4_51. 6

[RSS19] RUMSEY C. L., SLOTNICK J. P., SCLAFANI A. J.: Overview and summary of the third aiaa high lift prediction workshop. *Journal of Aircraft 56*, 2 (2019), 621–644. doi:10.2514/1.C034940. 2, 6

[SMG15] SCHROEDER W., MAYNARD R., GEVECI B.: Flying edges: A high-performance scalable isocontouring algorithm. In *2015 IEEE 5th Symposium on Large Data Analysis and Visualization (LDAV)* (Chicago, IL, USA, 2015), Institute of Electrical and Electronics Engineers, pp. 33–40. doi:10.1109/LDAV.2015.7348069. 1

[SML96] SCHROEDER W., MARTIN K., LORENSEN W.: The design and implementation of an object-oriented toolkit for 3d graphics and visualization. In *Proceedings of Seventh Annual IEEE Visualization '96* (1996), pp. 93–100. doi:10.1109/VISUAL.1996.567752. 2, 7

[SML06] SCHROEDER W., MARTIN K., LORENSEN W.: *The Visualization Toolkit: An Object-Oriented Approach To 3D Graphics*. Kitware, Clifton Park, 01 2006. 1, 9

[STHF24] SCHROEDER W., TSALIKIS S., HALLE M., FRISKEN S.: A high-performance surfacenets discrete isocontouring algorithm, 2024. arXiv:2401.14906. 6, 9

[WBM*23] WASSERTHAL J., BREIT H.-C., MEYER M. T., PRADELLA M., HINCK D., SAUTER A. W., HEYE T., BOLL D. T., CYRIAC J., YANG S., BACH M., SEGEROTH M.: Totalsegmentator: Robust segmentation of 104 anatomic structures in ct images. *Radiology: Artificial Intelligence 5*, 5 (2023), e230024. doi:10.1148/ryai.230024. 6

Appendix A: Reproducibility of Results

To reproduce the results presented in Section 6, access the vtk-clip-evaluation repository. This repository includes the evaluation code along with a link to the used datasets and a Docker image that includes all the built executables used for the performance evaluation. There are six executables: *clip-par-test-various-batchSizes-trimming-no*, *clip-par-test-various-batchSizes-trimming-yes*, *clip-par-pass-time*, *clip-seq*, *clip-m-par*, and *clip-par*. Each executable compiles against a different version of VTK that includes the appropriate version of the `vtkTableBasedClipDataSet` and `vtkmClip` classes. The VTK version that is utilized by each executable can be found at its *CMakeLists.txt* in the form of a Git commit SHA.

All executables have the *input filename*, *percentage*, and *number of iterations* parameters, while the parallel ones also have the *number of threads* parameter. At the end of each execution, the following information is printed: the memory used by the dataset in kilobytes, the average time spent by each algorithm in milliseconds, and the maximum memory used by the executable in kilobytes. The memory footprint of the algorithm is determined by subtracting the memory used by the dataset from the maximum memory used by the executable. The following command shows an output example of the clip-par executable.

```
$ /usr/bin/time -vv\
      ./clip-par ./Torso.vtu 0.5 1 8
Origin: 490.803, 287.599, 648.45
Normal: 1, 0, 0
Number of input cells: 2867390
Number of input points: 1404982
Memory used by dataset in KB: 139372
Number of output cells: 1454024
Number of output points: 720730
Time in ms: 19
Command being timed:\
      "./clip-par ./Torso.vtu 0.5 1 8"
...
Maximum resident set size (kbytes): 197924
...
```