External Facelist Calculation with Data-Parallel Primitives

Brenton Lessley[*]
Department of Computer and Information Science
University of Oregon

Abstract

External facelist calculation on three-dimensional unstructured meshes is used in scientific visualization libraries to efficiently render the results of operations such as clipping, interval volumes, and material boundaries. With this study, we consider the external facelist algorithm on many-core architectures. We introduce four different approaches: three based on hashing and one based on sorting. Each of the algorithms consists entirely of data-parallel primitive operations, in an effort to achieve portable performance across different architectures. We study the performance of the algorithms via experiments varying over algorithm, data set, hardware, and other factors. Overall, we observe that a hashing-based implementation achieves better runtime performance for the majority of configurations, while also achieving the most-stable performance on highly unstructured data sets.

1 Introduction

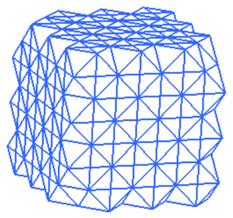
This work considers External Facelist Calculation (EFC) in the paradigm of Data-Parallel Primitives (DPP). We motivate each topic independently, and then motivate the purpose for joint consideration and the corresponding research challenges.

1.1 External Facelist Calculation

Scientific visualization algorithms vary regarding the topology of their input and output meshes. When working with three-dimensional volumes as input, algorithms such as isosurfacing and slicing produce outputs (typically triangles and quadrilaterals) that can be rendered via traditional surface rendering techniques, e.g., rasterization via OpenGL. Algorithms such as volume rendering operate directly on three-dimensional volumes, and use a combination of color and transparency to produce images that represent data both on the exterior of the volume and in the interior of the volume. However, some scientific visualization algorithms take three-dimensional volumes as input and produce three-dimensional volumes as output. While these three-dimensional volume outputs could be rendered with volume rendering or serve as inputs to other algorithms such as isosurfacing, users often want direct renderings of these algorithms' outputs using surface rendering. With this work, we consider this latter case, and consider the approach where geometric primitives are extracted from a volumetric unstructured mesh, in order to use traditional surface rendering techniques.

Given, for example, an unstructured mesh of N tetrahedrons to render (see Figure 1), a naïve solution would be to extract the four faces that bound each tetrahedron, and render the corresponding $4 \times N$ triangles. This naïve solution would be straight-forward to implement and would fit well with existing rendering approaches. However, many of the $4 \times N$ triangles this algorithm would produce are contained within the interior of the volume, and thus not useful. The primary downside to the naïve approach, then, is efficiency. For a data set with N tetrahedrons, only $O(N^{\frac{2}{3}})$ of the faces would actually lie on the exterior, meaning the large majority of the $4 \times N$ faces produced are unwanted, taking up memory to store and slowing down rendering times. If N was one million, then the expected number of external faces would be approximately 10,000, where the naïve algorithm would calculate four million faces, i.e., 400X too many. A second downside to these triangles is that they can create rendering artifacts. If the faces are rendered using transparency, then internal faces become visible, which is typically not the effect the user wants when they opt to use surface rendering for a three dimensional volume.

[*]e-mail: blessley@cs.uoregon.edu


(a) Tetrahedral mesh

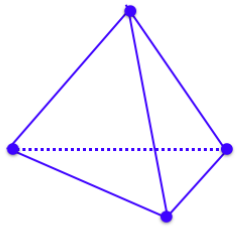

(b) Tetrahedron cell

Figure 1: Example of a mesh of tetrahedron cells. Each tetrahedron cell consists of 4 faces, one or more of which may overlap with the faces of other neighboring cells. The non-overlapping cell faces are denoted *external*.

A better algorithm, then, is to produce only the faces that lie on the exterior of the mesh, so called "External Facelist Calculation," or EFC. EFC is a mainstay in scientific visualization packages, specifically to handle the case of rendering the exteriors of three-dimensional volumes via surface-rendering techniques.

1.2 Data-Parallel Primitives

Many-core architectures are being increasingly included on leading-edge supercomputers, although the specific types of architecture vary. While developers of large-data visualization software packages recognize the need to update their code bases for many-core [4, 9], they view the variation in architecture as problematic, because their packages contain so many algorithms. Restated, if faced with N algorithms and M architectures, they do not want $N \times M$ implementations. Rather, they prefer an approach where they can deal with many-core architectures abstractly, and thus implement their algorithms only one time each. Of course, they still want their instantiation of each algorithm on a given architecture to perform as efficiently as possible.

Data-parallel primitives [7], or DPP, is a paradigm for achieving portable performance across many-core architectures. In this paradigm, programmers compose operators known to perform efficiently on many-core architectures. However, translating an algorithm into data-parallel primitives is a non-trivial task, and often requires "re-thinking" an algorithm rather than "porting" it.

1.3 Combination and Challenges

VTK-m [22] is a new visualization software package, designed for efficient use of many-core architectures via DPP. In terms of motivation for the combination of DPP and EFC, the VTK-m project team made a list of the top algorithms needed to be functional [21], and EFC was one of fourteen algorithms placed in the "first tier" of immediate needs, along with slicing, isosurfacing, thresholding, streamlines, and others. Further, this ranking placed EFC ahead of well-known algorithms such as glyphing, tetrahedralization, vertex normal calculations, coordinate transformations, clipping, and feature edges.

The challenges with EFC and DPP are two-fold. One, serial EFC is traditionally done with hashing, which is non-trivial to implement with DPP. As a result, we needed to construct new, hashing-inspired algorithms that sidestep the problems with traditional hashing with DPP. And, two, DPP has been shown to be efficient with more traditional scientific visualization algorithms that iterate over cells or pixels, but EFC is essentially a search problem, and so it is unclear if DPP will perform well. On this front, we demonstrate that DPP does indeed perform well and again does provide good performance on this class of scientific visualization problem.

This paper's contribution, then, is to illuminate the best techniques to execute EFC with DPP. We introduce three algorithmic variants inspired by hashing, and also an algorithm based on sorting; to our knowledge, these are the first ever shared-memory algorithms for EFC. We then conduct a performance study, where we measure execution time for our algorithms on multiple data sets and architectures. Our findings show that hashing is still the best approach for EFC on parallel architectures.

2 Background and Related Work

2.1 Visualization and Data Parallel Primitives

Many community visualization packages, such as VTK, OpenDX [3], and AVS [24], have demonstrated the benefit of having interoperable modules connected via a data-flow paradigm. However, these packages were developed more than two decades ago, and the majority of their functionality is implemented with single-thread programming. Starting around the year 2000, packages such as VisIt [8], ParaView [5], EnSight [10], and FieldView [14] added parallelization via distributed-memory concepts. With this model, the program managed parallel processing of data, but the basis for applying algorithms mostly derived from the existing (serial) visualization packages. This approach allowed these community packages to remain effective on supercomputers until present day. However, recent supercomputing trends increasingly include many-core architectures, such as GPUs and Intel Xeon Phis. In response to this trend, multiple efforts began in the 2010 time frame to create visualization systems that would work efficiently on many-core architectures, and further would provide portable performance over multiple architectures (e.g., both NVIDIA GPUs and Xeon Phi). These efforts, DAX [20], EAVL [17, 19], and PISTON [15] all independently arrived at the same strategy, namely using data-parallel primitives as the basic construct as a way to achieve efficiency in on many-core architectures, and as a way to future-proof their development against upcoming architectures. Two of the efforts, DAX and PISTON, made heavy use of NVIDIA's Thrust [6], which subscribes to the data-parallel primitive approach. Ultimately, these three efforts united into a single effort, now called VTK-m.

A major research question for data-parallel primitives is whether the approach can achieve high efficiency on varying platforms, so-called portable performance.. Some of this evidence was reported in the initial papers by the DAX, EAVL, and PISTON teams, with additional evidence coming afterwards. Specifically, Maynard et al. demonstrated good performance with thresholding [16] and Larsen et al. demonstrated good performance with both raytracing [13] and volume rendering [12]. These latter papers are likely the most closely related work, in that they recast existing visualization algorithms into the data-parallel primitives paradigm and consider their performance. This work is differentiated from those in that (1) EFC has not been previously considered and, moreover, (2) EFC represents a class of scientific visualization algorithm (search-based algorithms rather than iterating over loops of cells or pixels) that has not been previously considered.

2.2 External Facelist Calculation

EFC comes up surprisingly often in scientific visualization. For example, many engineering applications, such as bridge and building design, use the external faces of their model as their default visualization, often to look at displacements. Further, clipping and interval volumes are also commonly used with external facelist calculation. In these algorithms, a filter removes a portion of the volume (based on either spatial selection or data selection); if no further operations are performed then EFC is needed to view the clipped region. As a final example, some material interface reconstructions approaches, like that by Meredith et al. [18], take three-dimensional volumes and create multiple three-dimensional volumes, each one corresponding to a pure material. In this case, when users remove one or more materials, EFC is needed to few the material boundaries.

To our knowledge, no previous research papers have been devoted to EFC. However, implementations are posted on the internet, for example with VTK's `vtkUnstructuredGridGeometryFilter` [1] and VisIt's `avtFacelistFilter` [2]. The basic idea behind the filter is to count how many times a face is encountered. If it is encountered twice, then it is internal, since the face is incident to two different cells, and so it is thus between them (and internal). If a face is encountered a single time, then it is external, since the face bounds one cell, and there is no neighboring cell to provide a second abutment.

In both implementations readily available on the internet, the "face count" is calculated through hashing. That is, in the first phase, every face is hashed (with the hash index derived from the point indices that define the face) into a large hash table. Then, in the second phase, the hash table is traversed. If a face was hashed into a hash table index two times, then it is internal and discarded. But if it was hashed into a hash table index only a single time, then it is external, and the face is added to the output.

3 Data Parallel Primitives

The parallel algorithms presented in this section are based entirely on "Data Parallel Primitives," or DPP, which are architecture-agnostic operations that can be written once in a high-level language and compiled on different environments. The following DPP form the basis of our algorithm implementations:

- Gather: Given an input array of elements, Gather reads values into an output array according to an array of indices.
- Map: Applies an operation on an input array to produce an output array of the same size.
- Reduce: Applies a "combiner" operator (e.g., summation or average) to an input array to produce a single output value. A variation includes performing a Reduce for each key, or unique data value, in the input array.
- Scan: Performs a series of partial reductions, or a prefix-sum, on an input array to produce an output array of the same size.
- Scatter: Given an input array of data and an array of indices, Scatter writes each element of the data array into a location in an output array, as specified in the array of indices.
- Stream Compact: Removes all elements from an input array that satisfy a unary predicate condition (e.g., if an input element equals zero), and places the remaining elements into an ouput array of an equal or smaller size.

Additionally, our parallel algorithms consist of several "functors," which are functions that a DPP applies to an input array, yielding an array of output values. For example, a custom functor can be written to multiply an input value by a constant factor, with a Map DPP executing this functor over the input array, in parallel.

4 Algorithms

This section presents four DPP-based EFC algorithms. All four algorithms share a common initialization procedure, which is described in Section 4.1. The first three algorithms, which are all hashing-based, are described in Section 4.2. The final algorithm, which is sorting-based, is described in Section 4.3.

4.1 Initialization

For each algorithm, the first step is to generate a list of point indices for each of its faces. That is, for a tetrahedral mesh of N cells, we generate an $12 \times N$ array, where the first three elements are the indices of the first face of the first cell, the next three elements are the indices of the second face of the first cell, and so on. While generating this array is conceptually simple, doing so with DPP takes multiple invocations of scans, gathers, and maps, using various functors. For the remainder of this paper, this initialization procedure will be referred to as *GetFacePoints* and its output array as *facePoints*. Both pseudocode and an algorithm description are presented for the routine in the Appendix section. The algorithm details are deferred to the appendix since they are relatively straightforward and distract from the overall message of this study.

4.2 Hashing Approach

Collisions are a key aspect of hashing. Typically, these collisions are dealt with via chaining (i.e., employing linked lists to store multiple entries at a single index) or open addressing (i.e., when an index is occupied, then storing the data at the next open address). While these strategies are straight-forward to implement in a serial setting, they do not directly translate to a parallel setting. For example, in a GPU setting where each thread is executing the same program, the variable number of operations resulting from chaining or open addressing can lead to divergence (while non-collided threads wait for a collided thread to finish), and thus a performance bottleneck. Additionally, if multiple threads map to the same hash entry at the same time, then the behavior may be erratic, unless atomics are employed.

To address the problem of collisions in a parallel setting, we employ a modified hashing scheme that uses multiple iterations. In this scheme, no care is taken to detect collisions. Instead, every face is written directly to the hash table, possibly overwriting previously-hashed faces. The final hash table will then contain the winners of this "last one in" approach. Our next step, however, is to check, for each face, whether it was actually placed in the hash table. If so, the face is included for calculations during that iteration. If not, then the face is saved for future iterations. All faces are eventually processed, with the number of iterations equal to the maximum number of faces hashed to a single index.

In terms of hashing specifics, our hash function uses a face's three point indices as input, and produces an unsigned integer as output. This integer value, modulo the size of the hash table, is the hash index for the face. Hash function is important, as good choices of hash function minimize collisions, and poor choices create more collisions. We experimented with multiple hash functions and used the best performing, FNV-1a, for our study.

In this study, we introduce three different hashing-inspired algorithms, which we refer to as General, Compact, and Rehash. Each algorithm is composed entirely of DPP. The pseudocode for all three is listed in Algorithm 1, and the following subsections augment this pseudocode with descriptions of individual variables and operations.

4.2.1 Hashing—General

The algorithm first computes a hash value for each face, using a *ComputeFaceHash* functor, as denoted on Line 17. The set of hash values, *faceHashes*, is the output of *ComputeFaceHash*.

Given *faceHashes*, an iterative process then identifies which of the collisions are associated with either internal or external faces. A description of this process is as follows:

1. Scatter faces into the hash table (Line 19): For each "active" face, f_i, with $isActive[i] = 1$, write the face Id i to location $k = faceHashes[i]$ in $hashTable$. Multiple, distinct faces may hash to the same location k, resulting in collisions; the final value written in $hashTable[k]$ is the Id of the "current" hashed face at index k.
2. Gather current hashed faces (Lines 20 and 21): For each face, f_i, read the last-written face Id from $hashTable[faceHashes[i]]$ and store the Id in $currentHashedIds[i]$. Then, for each index j, $0 \leq j \leq F$, of $currentHashedIds$, extract the three point indices of the current hashed face with Id $k = currentHashedIds[j]$; the point indices are gathered from location k of the $facePoints$ array.
3. Check for internal faces via *CheckForMatches* (Line 22): An active face, f_i, with $isActive[i] = 1$, is an internal face if it satisfies the following two conditions:
   (a) f_i has the same point indices, $facePoints[i]$, as that of the current hashed face, $f_j = currentHashedFaces[i]$.
   (b) The face Id, i, is not equal to the current hashed face Id, $currentHashedIds[i]$.

   If f_i is internal, then f_j must be as well. Hence, $isActive[i]$ is set to 0, and both f_j and f_j are denoted internal by setting $isExternalFace[i] = 0$ and $isExternalFace[j] = 0$. If both the point indices and Ids match, then f_i is the current hashed face at location $faceHashes[i]$ in $hashTable$, and $isActive[i]$ is set to 0; note that f_i maintains its status as an external face until another active face satisfies the internal face criteria with f_i.
4. Update the number of active faces (Line 24): The number of active faces, A, is computed via an Inclusive Scan operation on the *isActive* array. This scan performs a prefix-sum of the binary values within the array; hence, the last array element represents the sum of all faces with $isActive = 1$.

The foregoing process continues until $A = 0$; i.e., when all faces have been set to inactive. Then, to extract the external faces, a *StreamCompact* operation (Line 39) generates an *externalFaces* array that contains the point indices of each face, f_i, satisfying $isExternalFace[i] = 1$.

4.2.2 Hashing—Compact

The Compact hashing algorithm follows the same iterative procedure as the General hashing algorithm, but instead "compacts" the $faceHashes$ and $faceIndices$ arrays each iteration to remove elements with corresponding values of 0 in *isActive*. The Compact algorithm then deploys the following variation to the General algorithm. First, for each active face with $isActive[i] = 1$, its corresponding elements from the $faceHashes$ and $faceIndices$ arrays are removed (Lines 26 and 27). Then, *isActive* is compacted by removing all values of 1 (Line 28). The new size of this array, $A = |isActive|$, becomes the number of active faces for the next iteration (Line 29). As with the General approach, this iterative procedure continues until $A = 0$.

4.2.3 Hashing—Rehash

The Rehash algorithm specializes the General algorithm by recomputing $faceHashes$ every iteration (Line 34), instead of once,

Algorithm 1: Pseudocode for the EFC hashing algorithms. N is the total number of tetrahedral cells, F is the total number of (non-unique) cell faces, E is the number of external faces, and A is the number of active cell faces. The constant c is a multiplier for the hash table size.

```
1  /*Input from GetFacePoints*/
2  Array: Vec<int,3> facePoints[$F$]
3  /*Output*/
4  Array: int outShapes[$E$], outNumIndices[$E$], outConn[3$E$]
5  /*Local Objects*/
6  Array: int faceHashes[$F$], faceIndices[$F$], hashTable[$cF$],
   isActive[$F$], isExternalFace[$F$]
7  Array: Vec<int,3> externalFaces[$E$]
8  ArrayPerm: Vec<int,3> currentHashedFaces[$F$]
9  ArrayPerm: int currentHashedIds[$F$]
10 facePoints←GetFacePoints
11 $F = |\text{facePoints}|$
12 $A \leftarrow F$
13 //Parallel array allocations
14 hashTable← $\vec{\mathbf{0}}$
15 isActive← $\vec{\mathbf{1}}$
16 isExternalFace← $\vec{\mathbf{1}}$
17 faceHashes←ComputeFaceHash(facePoints)
18 while $A > 0$ do
19     hashTable←Scatter(faceHashes, faceIndices, isActive,
       hashTable);
20     currentHashedIds←Gather(faceHashes, hashTable);
21     currentHashedFaces←Gather(currentHashedIds,
       facePoints);
22     (isActive,
       isExternalFace)←CheckForMatches(currentHashedFaces,
       facePoints, currentHashedIds, isActive, isExternalFace);
23     if General then
24         $A$ ←ScanInclusive(isActive);
25     else if Compact then
26         faceHashes←StreamCompact(faceHashes, isActive);
27         faceIndices←StreamCompact(faceIndices, isActive);
28         isActive←StreamCompact(isActive, isActive);
29         $A \leftarrow |\text{isActive}|$;
30     else if Rehash then
31         hashTable←Shrink(hashTable);
32         faceIndices←StreamCompact(faceIndices, isActive);
33         facePoints←StreamCompact(facePoints, isActive);
34         faceHashes←ComputeFaceHash(facePoints);
35         isActive←StreamCompact(isActive, isActive);
36         $A \leftarrow |\text{isActive}|$;
37 end
38 externalFaces←StreamCompact(facePoints, isExternalFace)
39 //Serial loop to create triangle face connectivity
40 return (outShapes, outNumIndices, outConn)
```

prior to the hashing loop. Additionally, Rehash shrinks the size of *hashTable* to a length of $c \times A$ at the end of each iteration (Line 31), where c is a constant size factor. This smaller table size affects the output of the *ComputeFaceHash* routine in the succeeding iteration because the hash value is a function of the table size. Due to the change in hash table size and multiple invocations of *ComputeFaceHash*, new collisions will likely arise each iteration. Thus, Rehash may exhibit a stable number of iterations, regardless of the hash table size.

The remaining parallel components of Rehash mirror those of the Compact hashing algorithm.

4.3 Sorting Approach

The idea behind this approach is to use sorting to identify duplicate faces. First, faces are placed in an array and sorted. Then, the array can then be searched for duplicates in consecutive entries. Faces that repeat in consecutive entries are internal, and the rest are external. The sorting operation requires a way of comparing two faces (i.e., a "less-than" test); we order the vertices within a face, and then compare the vertices with the lowest index, proceeding to the next indices in cases of ties.

The pseudocode for the DPP Sorting algorithm is in Algorithm 2; the remainder of this paragraph augments this description with additional details on individual variables and operations. The *facePoints* array is sorted in ascending order so that a reduce-by-key operation can be performed to determine the unique faces and their frequency counts within the array. All faces with counts greater than 1 are considered internal and removed, in parallel, from the array via a *StreamCompact* functor. The final, compacted array consists of only the external faces, which are returned as output.

Algorithm 2: Pseudocode for the Sorting approach of external facelist calculation. N is the total number of tetrahedral cells, M is the total number of (non-unique) cell faces, and E is the number of external faces.

```
1  /*Input from GetFacePoints*/
2  Array: Vec<int,3> facePoints[$M$]
3  Int: $M$
4  /*Output*/
5  Array: int outShapes[$E$], outNumIndices[$E$], outConn[$3 * E$]
6  /*Local Objects*/
7  Array: Vec<int,3> uniqueFaces[$E \leq L \leq M$],
   externalFaces[$E$]
8  Array: int uniqueFaceCounts[$E \leq L \leq M$]
9  ArrayConstant: int ones[$M$]
10 (facePoints, $M$)←GetFacePoints
11 facePoints←Sort(facePoints)
12 uniqueFaces, uniqueFaceCounts←ReduceByKey(facePoints,
   ones)
13 externalFaces←StreamCompact(uniqueFaces,
   uniqueFaceCounts)
14 //Serial loop to create triangle face connectivity
15 return (outShapes, outNumIndices, outConn)
```

5 Experiment Overview

This section describes the details of our experiment.

5.1 Factors

This study varied the following four factors:

- Data set: Since data set may affect algorithm performance, we varied them over both size and data layout.

- Hardware architecture: Aligned with our motivation of developing portable performance across different architectures, we test our implementation for two architectures: CPU and GPU. For the CPU, we also consider the effect of concurrency on runtime performance by varying the number of hardware cores.
- Algorithm implementation: We assess the variation in performance over our four different parallel algorithms for EFC.
- Hash table size: For the hashing-based algorithms, we varied the size of the hash table, and observe its effect on performance.

5.2 Software Implementation

All four of the EFC algorithms are implemented in the VTK-m toolkit. With VTK-m, a developer chooses data parallel primitives to employ, and then customizes those primitives with kernels of C++-compliant code. This code is then used to create architecture-specific code for architectures of interest, for example CUDA code for NVIDIA GPUs and Threading Building Blocks (TBB) code for Intel CPUs. Data parallel operations are templated by a *device adapter tag* that specifies both the type of device on which the operation will be dispatched and the method of parallel processing. Since these device modes only need to be enabled pre-compilation, each EFC algorithm is written once and used (and compiled) for, in our case, both TBB and CUDA execution. This flexibility demonstrates the advantage of using DPP as part of VTK-m, as a single code base can be deployed to multiple hardware architectures.

In terms of configuration, both the TBB and CUDA configurations of VTK-m were compiled with the gcc compiler, and the VTK-m index integer (vtkm::Id) size was set to 32 bits.

5.3 Configuration

In this study, we vary the four factors over a sequence of five phases, resulting in 59 total test configurations. The number of options, per factor is as follows:

- Data set (6 options)
- Hardware architecture (7 options)
- Algorithm (4 options)
- Hash table size (5 options)

These configurations are discussed in the following subsections.

5.3.1 Data Sets

We applied our test cases to six data sets, four of which were derived from two primary data sets. Figure 2 contains renderings for these two data sets.

- Enzo-10M: A cosmology data set from the Enzo [23] simulation code. The data set was originally on a 128^3 rectilinear grid, but was mapped to a 10.2M tetrahedral grid. The data set contains approximately 20M unique faces, of which 194K are external.
- Enzo-80M: An 83.9M tetrahedron version of Enzo-10 M, with approximately 166M unique faces, of which 780.3K are external.
- Nek-50M: An unstructured mesh that contains 50M tetrahedrons from a thermal hydraulics simulation Nek5000 [11]. The data set contains approximately 100M unique faces, of which 550K are external.
- Re-Enzo-10M, Re-Enzo-80M, Re-Nek-50M: Versions of our previous three data sets where the point lists were randomized. Especially for the Enzo data sets, the regular layout of the data leads to cache coherency — by randomizing the point list, each tetrahedron touches more memory. To be clear, each individual tetrahedron in the mesh occupies the same spatial location as its non-randomized predecessor, but the four points that define the tetrahedron no longer occupy consecutive or nearby points in the point list.

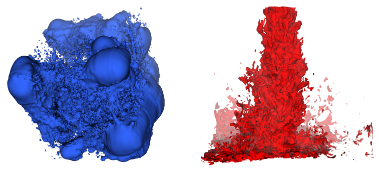


Figure 2: Visualizations of two of the data sets used in this study. The Enzo-10M data set is on the left and Nek-50M is on the right.

Finally, while we reference the data sources, we note the only important aspect for EFC performance is the mesh and mesh connectivity.

5.3.2 Hardware architecture

We ran our tests on the following two architectures:

- CPU: A 16-core machine running 2 nodes. Each node has a 2.60 GHz Intel Xeon(R) E5-2650 v2 CPU with 8 cores and 16 threads. For each CPU, the base frequency is 2.6 GHz, memory bandwidth is 59.7 GB/s, and memory 64 GB. In our experiment we also varied the number of cores: 1, 2, 4, 8, 12 and 16. Each concurrency employed the Intel TBB multithreading library for many-core parallelism.
- GPU: An NVIDIA Tesla K40 Accelerator with 2880 processor cores, 12 GB memory, and 288 GB/sec memory bandwidth. Each core has a base frequency of 745 MHz, while the GDDR5 memory runs at a base frequency of 3 GHz. All GPU experiments use NVIDIA CUDA V6.5.

5.3.3 Algorithm implementation

The four EFC algorithms are evaluated in this study:

- General: The general hashing algorithm presented in Section 4.2.1.
- Compact: The compact hashing algorithm presented in Section 4.2.2.
- Rehash: The rehash algorithm presented in Section 4.2.3.
- Sorting: The sorting algorithm presented in Section 4.3.

5.3.4 Hashing Table Size

For the hashing-based algorithms, we assess the runtime performance as the hash table size changes. The table size is set at various multiples of the total number of faces in the data set. In this study, we considered five options: 0.5X, 1X, 2X, 4X, and 8X.

The presence of the 0.5X option underscores the difference between regular hashing and our hashing variant. With regular hashing and a chaining approach, the size of the hash table must be at least as large as the number of elements to hash, and preferably much larger. With our variant, the table size can be decreased, with the only penalty being that there will be more iterations, as the maximum number of faces hashed to a single index will (on average) increase proportionally. In this way, the memory allocated to hashing can be reduced, at the cost of increased execution time.

5.4 Methodology

Our study contains five phases, with each phase examining the impact of different parameters on performance.

5.4.1 Phase 1: Base Case

In this phase, we explore the basic performance characteristics for our four algorithms: General, Compact, Rehash, and Sorting. We perform our tests using the Enzo-10M data set, 16 CPU cores, and a hashing table factor of two (for hash-based algorithms).

Configuration: (CPU, 16 cores, Enzo-10M, hash table factor 2 for hashing-based algorithms) $\times$ 4 algorithms.

5.4.2 Phase 2: Hash Table Size

In this phase, we examine the impact of the hash table size on performance. We consider 5 different sizes, varying the proportion over 0.5X, 1X, 2X, 4X, and 8X. We again run the experiment using the Enzo-10M data set and 16 CPU cores.

Configuration: (CPU, 16 cores, Enzo-10M) $\times$ 5 different hash table proportions $\times$ 3 hashing-based algorithms.

5.4.3 Phase 3: GPU

In this phase, we investigate the runtime performance on a GPU, comparing the Sorting algorithm against each of the three hashing-based algorithms.

Configuration: (GPU, Enzo-10M) $\times$ 4 algorithms.

5.4.4 Phase 4: Data Sets

In this phase, we assess the impact of data set size and memory locality on the runtime performance of the Sorting and Rehash algorithms, using the data sets of Section 5.3.1. **Configuration:** (CPU, 16 cores, GPU) $\times$ 6 data sets $\times$ 2 algorithms.

5.4.5 Phase 5: Concurrency

In this phase, we investigate the CPU runtime performance by changing the number of hardware cores. We perform our test on both the Enzo-10M and Re-Enzo-10M data sets.

Configuration: (CPU) $\times$ 6 different concurrency levels $\times$ 2 data sets $\times$ 2 algorithms.

6 Results

In this section, we present and analyze the results of our suite of EFC experiments for the different phases outlined in Section 5.4.

6.1 Phase 1: Base Case

Our base case assesses the performance for the four algorithms — General, Compact, Rehash, and Sorting—with the following factors fixed:

- Enzo-10M data set
- 16 CPU cores
- Hash table size of $2 * F$, where F is the total number of non-unique faces in the data set

For each algorithm, we present the total execution time (in seconds), along with the sub-times for the primary parallel operations and routines. Additionally, we present the overhead time for memory allocations and deallocations. The results of the hashing- and sorting-based experiments are presented in Tables 1 through 4.

Table 1: Comparison of overall execution time (sec) for the CPU execution time (sec) for the EFC algorithms. The Main Computation quantity measurement does not include initialization, and instead measures the time for either sorting or hashing. Total time includes the time for both Main Computation and initialization.

Time	Sorting	General	Compact	Rehash
Main Computation	0.5	1.6	0.6	0.6
Total Time	0.9	2.0	0.9	0.9

As seen in Tables 1 and 2, Sorting completed the experiment in 0.9 seconds, with the Sort and Reduction operations—the main computation—accounting for 56% of the total time. Among the three hashing algorithms, both Compact and Rehash performed comparably with Sorting, while General demonstrated a slower runtime of 2.0 seconds.

Table 2: Individual phase times for the Sorting algorithm

Phase	CPU Time
GetFacePoints	0.2
Sort	0.3
Reduction	0.2
StreamCompact	0.02
Overhead	0.2
Total time	0.9

The three hashing algorithms differ in how much work they do each iteration. With General, all faces are traversed each iteration, although some faces are immediately discarded since they have been previously evaluated. With Compact and Rehash, the number of faces traversed goes steadily down each iteration. Tables 3 and 4 show the effect of this behavior. Table 3 shows that General is much slower, with its CheckForMatches phase (see Section 4.2.1), accounting for 65% of the total time. Further, Table 4 shows that the time per iteration for General is staying constant, while it gets much faster for the other two. Finally, we can see how many total iterations are needed from Table 4; for Enzo-10M with a hash table size of 2X, eight iterations are needed for Rehash. That said, the strategy of reducing the number of faces considered in each iteration makes most of these iterations inconsequential in terms of total run time.

Table 3: Execution time (sec) for the three hashing-based algorithms, broken down by phase

Phase	General	Compact	Rehash
GetFacePoints	0.2	0.2	0.2
Scatter	0.1	0.1	0.1
CheckForMatches	1.3	0.3	0.3
StreamCompact	0.02	0.15	0.1
ScanInclusive	0.1	-	-
ComputeFaceHash	0.03	0.02	0.05
Overhead	0.3	0.2	0.1
Total time	2.0	0.9	0.9

Table 4: Hashing algorithm execution time

Loop	General	Compact	Rehash
0	0.4	0.5	0.5
1	0.2	0.06	0.06
2	0.2	6.2e-03	4.3e-03
3	0.2	6.9e-04	6.6e-04
4	0.2	1.8e-04	2.2e-04
5	0.2	3.3e-05	4.1e-05
6	0.2	2.7e-05	4.2e-05
7	-	-	2.9e-05

6.2 Phase 2: Hashing Factor

In this phase, we study the effect of the hash table size on the performance of the three hashing algorithms. For each multiplier c, the hash table size is computed as $c * F$, where F is the total number of non-unique faces ($F \approx 40$ million for the Enzo-10M data set).

The results in Table 5 show that the execution time of General is impacted strongly by the hash table size, while both Compact and

Rehash are more size-independent. As hash table size increases, General realizes fewer hashing collisions, which results in fewer hashing iterations. The smaller number of iterations leads to the 0.7-second speedup in total runtime from the base multiplier of 2 to the multiplier of 8. Contrastingly, Compact and Rehash do not observe such a speedup. Rehash is likely not affected by the hash table multiplier because it recomputes hash values each iteration, leading to a slower decrease in collisions. While Compact demonstrates a reduction in the hashing iterations as the multiplier increases, the saved execution time of these iterations is not significant enough to yield a noticeable speedup. Summarizing, in memory-rich environments, higher multipliers can be used to gain modest speedups, while in memory-poor environments, lower multipliers can be used with only modest slowdowns.

Table 5: Performance of hashing algorithms as a function of hash table size multiplier.

Multiplier	General	Compact	Rehash
0.5	2.6	1.1	1.0
1	2.3	1.0	0.9
2	2.0	0.9	0.9
4	1.5	0.9	0.8
8	1.3	0.9	0.8

While increasing the hash table multiplier can reduce the total execution of General, additional CPU memory is required. This tradeoff should be taken into consideration when running larger data sets or using hardware architectures with lower memory capabilities. In our General experiments, running the Enzo-80M data set with a hashing multiplier of 8 exceeded the memory capacity of both the CPU and GPU architectures.

6.3 Phase 3: GPU

In this phase, we assess the performance of the algorithms on the GPU architecture with the Enzo-10M data set.

Table 6: GPU execution time (sec) for all 4 algorithms, along with the main computation time for the sort-and-reduction and hashing loop phases.

Time	Sorting	General	Compact	Rehash
Main Computation	0.5	0.3	0.2	0.2
Total Time	0.7	0.5	0.4	0.4

From Table 6, we observe that all three of the hashing-based algorithms achieve faster run times than Sorting. Both Compact and Rehash devote only half of their total execution time on *main computation*, which, for hashing, is the cumulative time spent in the hashing while-loop. Contrarily, Sorting spends more than 70% of its total runtime on the CUDA Thrust Sort operation, which, along with the Reduction operation, comprises the main computation; see Table 7 for GPU sub-times of the Sorting algorithm. Table 8 shows that the Scatter and CheckForMatches parallel routines account for at least half of the work for all three hashing algorithms. This contrasts slightly from the equivalent CPU findings of Phase 2, in which the algorithms spend a larger percentage of the time on the StreamCompact and ScanInclusive operations. The results of Table 8 indicate that the GPU significantly reduced the runtime of these parallel operations.

Based on the CPU and GPU experimental findings, the Rehash algorithm provides equal or better performance than the other two algorithms on the base case data set. So, for Phases 4 and 5, we opted to evaluate only Rehash of the hashing algorithms.

Table 7: GPU execution times (sec) for the Sorting algorithm

Phase	GPU Time
GetFacePoints	0.1
Sort	0.5
Reduction	4.0e-02
StreamCompact	4.5e-03
Overhead	0.1
Total time	0.7

Table 8: GPU execution time (sec) for the three hashing-based algorithms, broken down into primary parallel operations

Phase	General	Compact	Rehash
GetFacePoints	0.1	0.1	0.1
Scatter	0.1	0.1	0.1
CheckForMatches	0.2	0.1	0.1
StreamCompact	5.9e-03	3.7e-02	4.1e-02
ScanInclusive	1.4e-02	-	-
ComputeFaceHash	3.3e-03	4.6e-03	8.0e-03
Overhead	0.1	0.1	0.1
Total time	0.5	0.4	0.4

6.4 Phase 4: Data Set

This phase explores the effects of data set, by looking at six different data sets which vary over data size and memory locality. The study also varies architecture (CPU and GPU) and algorithm (Rehash and Sorting).

Table 9 displays the execution times on the CPU architecture using 16 cores. These results show that Sorting is affected by the locality of the cells within the data set meshes, as evident from the increase in runtime between the pairs of regular and restructured data sets. Table 10 further corroborates this observation by showing that Sorting nearly doubles its total runtime when presented with the restructured version of a data set on the GPU architecture. Contrarily, Rehash maintains stable execution times regardless of the cell locality in data sets.

With respect to execution time on both the CPU and GPU, Rehash consistently achieves comparable performance to Sorting for the regular data sets and significantly better performance for the restructured data sets. These findings indicate Rehash is superior for large data sets and for data sets with poor memory locality.

Table 9: CPU execution time in seconds for different data set/algorithm pairs

Data set	Sorting	Rehash
Enzo-10M	0.9	0.9
Nek-50M	4.3	4.3
Enzo-80M	7.4	7.3
Re-Enzo-10M	1.2	0.9
Re-Nek-50M	5.5	4.5
Re-Enzo-80M	9.2	7.7

Table 10: GPU execution time in seconds for different data set/algorithm pairs

Data set	Sorting	Rehash
Enzo-10M	0.7	0.4
Nek-50M	3.3	2.1
Enzo-80M	5.7	5.6
Re-Enzo-10M	1.0	0.4
Re-Nek-50M	5.3	2.2
Re-Enzo-80M	10.1	6.5

6.5 Phase 5: Concurrency

In this phase we investigate the CPU runtime performance of both Sorting and Rehash using different numbers of hardware cores with the base case Enzo-10M data set and its corresponding Re-Enzo-10M data set.

Tables 11 and 12 reveal that, although Sorting performs better than Rehash on configurations of 8 cores or fewer, Rehash provides stable performance regardless of memory locality; this confirms our findings from the previous phases. Additionally, the results indicate that with 1 CPU core, there is nearly a 10-time increase in runtime over the 16-core experiment, for both Sorting and Rehash. This observation demonstrates clear parallelism; however, the speedup is sub-linear.

Table 11: Impact of the number of CPU cores on the execution time (sec) for Sorting and Rehash using Enzo-10M

Method	1	2	4	8	12	16
Sorting	8.0	4.3	2.3	1.7	1.1	0.9
Rehash	10.8	5.6	3.9	1.9	1.1	0.9

Table 12: Impact of the number of CPU cores on the execution time (sec) for Sorting and Rehash using Re-Enzo-10M

Method	1	2	4	8	12	16
Sorting	9.6	5.1	2.9	1.9	1.3	1.1
Rehash	11.2	5.8	3.1	1.9	1.1	0.9

A review of Rehash over both Enzo-10M datasets indicates that only the *GetFacePoints*, *ComputeFaceHash*, and *CheckForMatches* data-parallel operations achieve near-linear speedup from 1 core to 16 cores. The remaining operations (e.g., *Scatter* and *StreamCompact*) achieve sub-linear speedup, contributing to the overall sub-linear speedup. For a majority of the individual operations, the smallest runtime speedup from a doubling of the hardware cores occurs in the switch from 8 to 16 cores. These findings suggest that, on up to 8 cores (a single CPU node), scalable parallelism is achieved, whereas from 8 to 16 cores (two CPU nodes with shared memory) parallelism does not scale optimally, possibly due to hardware and multi-threading limitations.

7 Serial Results

In Section 6.5, Rehash demonstrated a nearly 10-time increase in runtime over the base 16-core configuration, when executed on 1 CPU core. This single-core experiment simulates a serial execution of Rehash and motivates a comparison with the serial EFC implementations of community visualization packages. This section compares the runtime of serial Rehash (1-core) with that of the VTK `vtkUnstructuredGridGeometryFilter` and VisIt `avtFacelistFilter`, both of which are serial, single-threaded algorithms for EFC.

In Table 13, we observe that the VisIt algorithm outperforms both the VTK and Rehash algorithms on all of the data sets from Section 6.4, while Rehash performs comparably with the VTK implementation. The overall weak performance of Rehash is to be expected, since the DPP-based implementation is optimized for use in parallel environments. When compiled in VTK-m serial mode, the DPP functions are resolved into backend, sequential loop operations that iterate through large arrays without the benefit of multi-threading; hence, Rehash is neither optimized nor designed for 1-core execution. Contrarily, both VisIt and VTK are optimized specifically for single-core, non-parallel environments, leading to better runtimes than Rehash on the majority of the datasets.

Table 9 confirms that both Sorting and Rehash achieve better runtime performance than the serial algorithms on the base case 16-core CPU architecture. This finding validates the use of the DPP-based algorithms when presented with a multi-core CPU system.

Table 13: 1-core (serial) CPU execution time in seconds for different data set/algorithm pairs. VTK and VisIt are visualization toolkits that each provide serial, hashing-based EFC algorithms.

Data set	VTK	VisIt	Rehash
Enzo-10M	6.2	1.4	8.3
Nek-50M	33.1	5.2	60.9
Enzo-80M	51.7	9.1	102.6
Re-Enzo-10M	9.9	2.1	8.2
Re-Nek-50M	59.1	10.3	41.5
Re-Enzo-80M	84.4	17.7	109.1

8 Conclusions and Future Work

Our study has contributed new EFC algorithms within the DPP paradigm. Further, our experiments had the following findings:

- From Phase 1, we found that the General hashing algorithm is slower than the other algorithms, and that the overhead for removing previously considered faces greatly pays off in subsequent iterations. We also learned that, for smaller data sets, a Sorting-based approach is competitive with the better hashing algorithms.
- From Phase 2, we found that smaller hash table sizes slow down performance, but only modestly. This finding makes the hashing algorithms increasingly viable, since it allows them to operate with reduced memory.
- From Phase 3, we found that the hashing algorithms demonstrated an improved runtime on the GPU compared to sorting. While this finding is interesting, it is not consistent with the DPP mantra of "portable performance," as portable performance would imply that algorithms have the same relative speedup from architecture to architecture.
- From Phase 4, we found that Rehash outperforms Sorting on large and complex data sets. This finding is likely to be expected, since sorting has to do additional work when data is disorganized, but hashing is unaffected.
- From Phase 5, we see that the DPP approach does lead to improved performance as concurrency increases, but the speedup is not linear.

In terms of future work, we would like to expand this study to include more architectures, more data types, and further understand scalability limitations on multi-core CPUs.

References

[1] Sept. 2015. http://www.vtk.org/doc/nightly/html/ classvtkUnstructuredGridGeometryFilter.html.

[2] Sept. 2015. https://github.com/visit-vis/VisIt/blob/master/avt/Filters/avtFacelistFilter.C.

[3] G. Abram and L. A. Treinish. An extended data-flow architecture for data analysis and visualization. Research report RC 20001 (88338), IBM T. J. Watson Research Center, Yorktown Heights, NY, USA, Feb. 1995.

[4] S. Ahern et al. Scientific Discovery at the Exascale: Report for the DOE ASCR Workshop on Exascale Data Management, Analysis, and Visualization, July 2011.

[5] U. Ayachit, B. Geveci, K. Moreland, J. Patchett, and J. Ahrens. The ParaView Visualization Application. In *High Performance Visualization—Enabling Extreme-Scale Scientific Insight*, pages 383–400. Oct. 2012.

[6] N. Bell and J. Hoberock. Thrust: A productivity-oriented library for CUDA. In W.-M. Hwu, editor, *GPU Computing Gems*, pages 359–371. Elsevier/Morgan Kaufmann, 2011.

[7] G. E. Blelloch. *Vector models for data-parallel computing*, volume 356. MIT press Cambridge, 1990.

[8] H. Childs et al. VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data. In *High Performance Visualization—Enabling Extreme-Scale Scientific Insight*, pages 357–372. Oct. 2012.

[9] H. Childs, B. Geveci, W. Schroeder, J. Meredith, K. Moreland, C. Sewell, T. Kuhlen, and E. W. Bethel. Research Challenges for Visualization Software. *IEEE Computer*, 46(5):34–42, May 2013.

[10] Computational Engineering International, Inc. *EnSight website*, Sep 2015.

[11] P. F. Fischer, J. W. Lottes, and S. G. Kerkemeier. nek5000 Web page, 2008. http://nek5000.mcs.anl.gov.

[12] M. Larsen, S. Labasan, P. Navrátil, J. Meredith, and H. Childs. Volume Rendering Via Data-Parallel Primitives. In *Proceedings of EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV)*, pages 53–62, Cagliari, Italy, May 2015.

[13] M. Larsen, J. Meredith, P. Navrátil, and H. Childs. Ray-Tracing Within a Data Parallel Framework. In *Proceedings of the IEEE Pacific Visualization Symposium*, pages 279–286, Hangzhou, China, Apr. 2015.

[14] S. M. Legensky. Interactive Investigation of Fluid Mechanics Data Sets. In *VIS '90: Proceedings of the 1st conference on Visualization '90*, pages 435–439. IEEE Computer Society Press, 1990.

[15] L.-t. Lo, C. Sewell, and J. Ahrens. PISTON: A portable cross-platform framework for data-parallel visualization operators. pages 11–20. Eurographics Symposium on Parallel Graphics and Visualization, 2012.

[16] R. Maynard, K. Moreland, U. Atyachit, B. Geveci, and K.-L. Ma. Optimizing threshold for extreme scale analysis. In *IS&T/SPIE Electronic Imaging*, pages 86540Y–86540Y. International Society for Optics and Photonics, 2013.

[17] J. S. Meredith, S. Ahern, D. Pugmire, and R. Sisneros. Eavl: the extreme-scale analysis and visualization library. 2012.

[18] J. S. Meredith and H. Childs. Visualization and Analysis-Oriented Reconstruction of Material Interfaces. *Computer Graphics Forum (CGF)*, 29(3):1241–1250, June 2010.

[19] J. S. Meredith, R. Sisneros, D. Pugmire, and S. Ahern. A distributed data-parallel framework for analysis and visualization algorithm development. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, pages 11–19. ACM, 2012.

[20] K. Moreland, U. Ayachit, B. Geveci, and K.-L. Ma. Dax Toolkit: A Proposed Framework for Data Analysis and Visualization at Extreme Scale. In *Proceedings of the IEEE Symposium on Large-Scale Data Analysis and Visualization*, pages 97–104, October 2011.

[21] K. Moreland, H. Childs, et al. Algorithms needed for vtk-m, Nov. 2014. http://www.xvis.org/index.php/Missing_algorithms.

[22] K. Moreland, R. Maynard, et al. Vtk-m: Visualization for many-core, Sept. 2015. http://m.vtk.org.

[23] B. W. O'Shea, G. Bryan, J. Bordner, M. L. Norman, T. Abel, R. Harkness, and A. Kritsuk. Introducing Enzo, an AMR Cosmology Application. *ArXiv Astrophysics e-prints*, Mar. 2004.

[24] C. Upson, T. F. Jr., D. Kamins, D. H. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam. The application visualization system: A computational environment for scientific visualization. *Computer Graphics and Applications*, 9(4):30–42, July 1989.

9 Appendix

The following appendix section provides an algorithm description and corresponding pseudocode (see Algorithm 3) for the *GetFacePoints* initialization procedure.

Given cell shape, index, and connectivity arrays as input to GetFacePoints, the first parallel operation is to compute the number of faces per cell via *NumFacesPerCell* functor. In the current implementation, all tetrahedral cells have four faces. Subsequently, the prefix sum (inclusive scan) of these face counts is performed to obtain both the total number of (non-unique) faces, M, and the offsets for a forthcoming face-to-cell Id lookup array. For tetrahedral cells, *facesPerCell* would be $<4,\ldots,4>$ with a prefix sum output, *numFacesPrefixSum*, of $<4,8,\ldots,4N>$.

Next, an *UpperBounds* function is employed to generate the *face2CellId* lookup array in parallel. For each value i of a *countingArray*, $<1,2,\ldots,N>$, *UpperBounds* finds the uppermost location in *numFacesPrefixSum* where i could be inserted without disrupting the sorted order. Hence, a lookup array $<0,0,0,0,1,1,1,1,\ldots,N-1>$ is produced of length M. Furthermore, in preparation for the upcoming connectivity array construction, a *localFaceIds* lookup array is created to assign a local Id to each unique face of a cell. We obtain $<0,1,2,3,\ldots,0,1,2,3>$ by computing $(countingArray[i] - face2CellId[i]) \bmod 4$, for all $0 \leq i \leq N-1$, via a functor.

The next phase of the algorithm involves the extraction of point coordinates (vertex Ids) for each of the M faces. First, two gather operations are performed to obtain the shape types, $pt1 = <TET,\ldots,TET>$, and number of indices, $pt2 = <4,\ldots,4>$, for each of the cell indices in *face2CellId*. Second, a face connectivity array, *faceConn*, of length $4M$ is composed that repeats the four vertex Ids of a cell for each of it's four faces. For example, if cell 1 has vertices $<4,7,6,3>$ and cell 2 has vertices $<4,6,3,2>$, then *faceConn* would be $<4,7,6,3,4,7,6,3,\ldots,4,6,3,2,4,6,3,2,\ldots>$.

Finally, $pt1$, $pt2$, and $faceConn$ are combined to form a new explicit cell set, which is passed into a *GetFacePoint* functor along with *localFaceIds*. This functor assigns one of the four possible combinations of three vertex Ids to a given face, using the face's local Id value as the combination index; for instance, in cell 1 above, vertices $<4,7,6>$ will be assigned to the first face of the cell (local ID 0), vertices $<4,7,3>$ to the second face (local ID 1), etc. The resulting array of face point Ids, $facePoints$, is used as input to the subsequent sorting or hashing operations.

Algorithm 3: Pseudocode for the extraction of all face point Ids. The resulting *facePoints* array and other output arrays are subsequently used by both the Sorting- and Hashing-based External Faces algorithms. N is the total number of tetrahedral cells, M is the total number of (non-unique) cell faces, and E is the number of external faces.

```
/*Input*/
Array: int shapes[$N$], numIndices[$N$], conn[$4*N$]
/*Output*/
Array: Vec<int,3> facePoints[$M$]
Int: $M$
/*Local Objects*/
Array: int facesPerCell[$N$], numFacesPrefixSum[$N$],
    face2CellId[$M$], localFaceIds[$M$]
ArrayPermutation: int pt1[$M$], pt2[$M$], faceConn[$4*M$]
ArrayImplicit: int connIndices[$4*M$]
ArrayCounting: int countingArray[$M$]
CellSetExplicit: permutedCellSet

facesPerCell← NumFacesPerCell(shapes)
$M$, numFacesPrefixSum←ScanInclusive(facesPerCell)
face2CellId←UpperBounds(numFacesPrefixSum,
    countingArray)
localFaceIds←SubtractAndModulus(countingArray,
    face2CellId)
pt1←Gather(face2CellId, shapes)
pt2←Gather(face2CellId, numIndices)
connIndices←GetConnIndex($4*M$)
faceConn←Gather(connIndices, conn)
permutedCellSet←(pt1, pt2, faceConn)
facePoints←GetFacePoint(localFaceIds, permutedCellSet)
//Continue with Algorithm 2 or Algorithm 3 after returning.
return (facePoints, $M$)
```