Eurographics Symposium on Parallel Graphics and Visualization (2016)
W. Bethel, E. Gobbetti (Editors)

External Facelist Calculation with Data-Parallel Primitives

Brenton Lessley[1], Roba Binyahib[1], Robert Maynard[2], and Hank Childs[1,3]

[1]University of Oregon, Eugene, [2]Kitware, Inc., [3]Lawrence Berkeley National Laboratory

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 design and introduce two novel approaches, one based on sorting and one based on hashing. Both of these algorithms consist 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 data set, hardware, and other factors. Overall, we observe that the hashing-based implementation achieves better runtime performance for the majority of configurations, while also achieving the most-stable performance on highly unstructured data sets.

Categories and Subject Descriptors (according to ACM CCS): D.1.3 [Programming Techniques]: Concurrent Programming—Parallel programming I.3.3 [Computer Graphics]: Picture/Image Generation—Display algorithms

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 connected tetrahedrons to render, 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 ef-

fect the user wants when they opt to use surface rendering on a three dimensional volume.

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 [A*11, CGS*13], 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 [Ble90], 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

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 within DPP. And, although DPP has been shown to be efficient with more traditional scientific visualization algorithms that iterate over cells or pixels, 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 design and introduce two novel algorithmic variants inspired by hashing and by sorting. We then conduct a performance study, where we measure execution time for our algorithms on multiple data sets and architectures. Our findings show that our hashing-inspired variant is the best approach for EFC on parallel architectures using DPP.

2. Background and Related Work

2.1. Visualization and Data Parallel Primitives

Many community visualization packages, such as VTK [SML98], OpenDX [AT95], and AVS [UJK*89], 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 [C*12], ParaView [AGM*12], EnSight [Com16], and FieldView [Leg90] 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). The high-level approach for these efforts was informed by significant research on efficient visualization algorithms on GPUs [A*13], which illuminated the best patterns for implementing visualization systems in many-core environments. But the new efforts — DAX [MAGM11], EAVL [MAPS12, MSPA12], and PISTON [LSA12] — aimed to achieve efficiency on any many-core architecture, and to be future-proofed against upcoming architectures. All of the three efforts arrived independently at the same strategy, namely using data-parallel primitives as their basic construct. Two of the efforts, DAX and PISTON, made heavy use of NVIDIA’s Thrust [BH11], which subscribes to the data-parallel primitive approach. Ultimately, these three efforts united into a single effort, now called VTK-m [MSU*16].

Despite the adoption of data-parallel primitives so far, a major research question for the approach is whether it 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 [MMA*13] and Larsen et al. demonstrated good performance with both ray-tracing [LMNC15] and volume rendering [LLN*15]. Schroots and Ma extended these two studies by examining the tradeoffs of performance and portability for ray casting and cell projection volume renderers [SM15]. 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. Our 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 that has not been previously considered: search-based algorithms rather than iterating over loops of cells or pixels.

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. [MC10], 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 view the material boundaries.

While not an active area of research, implementations of EFC can be found on the internet, for example with VTK's `vtkUnstructuredGridGeometryFilter` [vtk16a] and VisIt's `avtFacelistFilter` [avt16]. The basic idea behind these filters 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 between them. If a face is encountered a single time, then it is external, since 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.

Niessner et al. [NZIS13] employ a DPP-based hashing approach on the GPU to perform scalable, real-time 3D scene reconstruction of live captures from a depth camera. This work uses a speed-efficient hash table data structure to insert, retrieve, and delete voxel blocks, each of which store sensor data for a uniform subregion of the perceived world. While the hashing routines are data-parallel, they, however, depend on atomic operations to avoid race conditions and collisions that can arise when inserting hash entries in parallel. Our proposed DPP-based hashing algorithms for EFC do not depend on this restriction and resolve collisions via an iterative, non-blocking process. Moreover, the work of [NZIS13] does not apply data-parallel hashing to the EFC task, which is the focus of our work.

3. Data Parallel Primitives

The new parallel algorithms presented in this study 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 primitives are used in 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 output array of an equal or smaller size.

Additionally, while DPPs capture high-level operations, they are augmented with functors, which describe the specific operation to perform. For example, if a developer wants to convert an array of temperatures from Celsius to Fahrenheit, then they can write a functor that does the conversion for a single value, followed by the use of a Map DPP to execute this functor over the entire input array in parallel.

4. Algorithms

This section presents our two new DPP-based EFC algorithms, which are described in Sections 4.2 and 4.3. Both algorithms share a common initialization procedure, which is described in Section 4.1.

The implementations of both algorithms are available online [vtk16b], for reference and reproducibility.

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. A detailed description of this process is contained in a technical report [Les15]. For the remainder of this paper, this initialization procedure will be referred to as *GetFacePoints* and its output array as *facePoints*.

4.2. Hashing-Based Algorithm

4.2.1. Algorithm Overview

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 straightforward 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 atomic operations are employed.

To address the problem of collisions in a parallel setting, we employ a modified hashing scheme that uses multiple iterations. In our scheme, no care is taken to detect collisions, making atomic operations unnecessary. 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. However, our next step 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. The hash function is important, as good function choices help minimize collisions, while poor choices create more collisions and, thus, more iterations. We experimented with multiple hash functions and used the best performing, FNV-1a, for our study.

4.2.2. Algorithm Details

The pseudocode for Hashing is listed in Algorithm 1, and the following subsections complement this pseudocode with descriptions.

The algorithm begins by computing a hash value for each

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],
5               outConn[3E]
6  /*Local Objects*/
7  Array: int faceHashes[F], faceIndices[F],
8               hashTable[cF], isActive[F],
9               isExternalFace[F]
10 Array: Vec<int,3> externalFaces[E]
11 ArrayPerm: Vec<int,3> currentHashedFaces[F],
12                       activePoints[F]
13 ArrayPerm: int currentHashedIds[F]
14 facePoints←GetFacePoints
15 F = |facePoints|
16 A ← F
17 //Parallel array allocations
18 hashTable← 0⃗
19 activePoints← <0,...,F−1>
20 isActive← 1⃗
21 isExternalFace← 1⃗
22 faceHashes←ComputeFaceHash(facePoints)
23 while A > 0 do
24     hashTable←Scatter(faceHashes, faceIndices,
25                       isActive, hashTable);
26     currentHashedIds←Gather(faceHashes, hashTable);
27     currentHashedFaces←Gather(currentHashedIds,
28                               facePoints);
29     (isActive, isExternalFace)←
30         CheckForMatches(currentHashedFaces,
31                       facePoints, currentHashedIds,
32                       isActive, isExternalFace);
33     hashTable←Shrink(hashTable);
34     faceIndices←StreamCompact(faceIndices,
35                               isActive);
36     activePoints←Gather(faceIndices, facePoints);
37     faceHashes←ComputeFaceHash(activePoints);
38     isActive←StreamCompact(isActive, isActive);
39     A ← |isActive|;
40 end
41 externalFaces←StreamCompact(facePoints,
42                              isExternalFace)
43 //Serial loop to create triangle face connectivity
44 return (outShapes, outNumIndices, outConn)
```

face (line 22). After this, the algorithm applies an iterative process to identify duplicate faces (lines 23-40). Unlike serial hashing-based EFC, our algorithm is iterative, which allows us to account for the collisions that can arise without the use of atomics. Within an iteration, some faces will not be successfully placed into the hash table, because a collision with another face will displace it. These "lost" faces are identified, and considered again in subsequent iterations. The algorithm terminates when every face has been considered, meaning that it was successfully placed into the hash table and then classified as external or internal. We refer to the set of faces that still need to be considered as "active faces," and the algorithm begins with every face as an active face. The specifics of an iteration are as follows:

1. Place all active faces into the hash table, via a scatter DPP (line 24). The destination of the scatter is the hash indices, and so it is this scatter process that results in collisions. The results of this operation are non-deterministic: faces with the same hash index displace (overwrite) each other, meaning only a subset of the active faces actually remain in the hash table at the end of the scatter.
2. Retrieve the point indices for each of the faces in the hash table, via a pair of gather DPPs (lines 26–28). Each of these faces is denoted as a "current" hashed face and consists of 3 point indices.
3. Detect which of the current hashed faces is an internal face, via a map DPP (lines 29–32). Each active face is hashed to a location in the hash table and compared to the current face residing at that location. Given the possibility of hashing collisions, multiple faces may be compared to the current hashed face at a given location. If an active face and current face have different face Ids but the same point indices, then the faces overlap and are considered internal; however, if the indices differ, then a hashing collision has occurred at this hash table location. If the faces share the same Ids and point indices, then the active face resides in the hash table and is considered external unless another active face (different Id) satisfies the internal face criteria with the current face. After being designated as an internal or external face, an active face becomes "inactive".
4. Rehash the remaining set of active faces into the hash table, which is shrunk each iteration to be proportional in size to the number of active faces; these operations are performed via stream compact, gather, and map DPPs (lines 33–37). This change in hash table size will affect the output of the hash function and the face hash values.
5. Update the list and number of active faces via a stream compact DPP (lines 38–39).

The foregoing process continues until every face becomes inactive and has been considered as either an internal or external face. If the maximum number of distinct faces (different point indices and face Ids) that hash to a given hash table location is K, then at most K iterations will be performed. Since an active face that "collides" with a current face does not immediately become inactive, it must wait to become the current face (reside in the hash table) before becoming inactive and considered as an internal or external face. If $K-1$ other faces hash to the same location, then it can take up to K iterations for this face to become a current face.

4.2.3. Variants on Hashing-Based Algorithms

In this study, we also considered two alternative data-parallel, hashing-based approaches to EFC. Both designs only compute the face hash values once before the hashing loop begins, reusing these values throughout each iteration via the same gather DPP operations of Hashing (lines 26–28). This strategy differs from the per-iteration $ComputeFaceHash$ operation of Hashing (line 37), which populates the decreasing-sized $faceHashes$ (decreased size via a stream compact DPP) array with new hash values. One of the alternative approaches compacts the $faceHashes$, $faceIndices$, and $isActive$ arrays in the same fashion as Hashing, whereas the other approach always kept the arrays at a fixed size F, which is the initial number of faces. This latter approach therefore avoids compaction operations and updates the same blocks of memory each iteration.

During our experiments, we found that both of our variants performed at best comparably to our main hashing-based algorithm from Section 4.2.2. Hence, we elected not to consider them in the remainder of this paper.

4.3. Sorting-Based Algorithm

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 Sorting algorithm is presented in Algorithm 2. In terms of details, the array of face points is parallely sorted in ascending order (line 13) so that a reduce-by-key DPP (lines 14–15) 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 stream compact DPP (lines 16–17). The final, compacted array consists of only the external faces, which are returned as output.

5. Experiment Overview

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.

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.

```
/*Input from GetFacePoints*/
Array: Vec<int,3> facePoints[M]
Int: M
/*Output*/
Array: int outShapes[E], outNumIndices[E],
           outConn[3*E]
/*Local Objects*/
Array: Vec<int,3> uniqueFaces[E <= L <= M],
                  externalFaces[E]
Array: int uniqueFaceCounts[E <= L <= M]
ArrayConstant: int ones[M]
(facePoints, M)<-GetFacePoints
facePoints<-Sort(facePoints)
(uniqueFaces, uniqueFaceCounts)<-
    ReduceByKey(facePoints, ones)
externalFaces<-StreamCompact(uniqueFaces,
                             uniqueFaceCounts)
//Serial loop to create triangle face connectivity
return (outShapes, outNumIndices, outConn)
```

- Hardware architecture: In order to evaluate portable performance, 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 for two different EFC data-parallel algorithms.
- 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

Both 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 functors 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. In our experiments, both the TBB and CUDA configurations of VTK-m are 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 420 total test configurations. The number of options per factor is as follows:

- Data set (6 options)
- Hardware architecture (7 options)
- Algorithm (2 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 1 contains renderings for these two data sets.

- Enzo-10M: A cosmology data set from the Enzo [OBB*04] 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 Nek5000 thermal hydraulics simulation [FLK08]. 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. Specifically, 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.

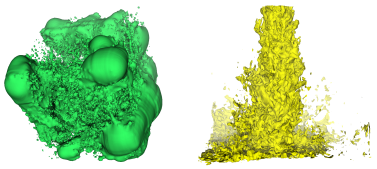

Figure 1: Visualizations of two of the data sets used in this study. The Enzo-10M data set is on the left and Nek-50M is on the right.

Finally, while we reference the data sources, we note the only important aspect for evaluating 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 multi-threading 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

Both of our DPP-based EFC algorithms are evaluated:

- Hashing: The algorithm presented in Section 4.2.
- Sorting: The 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 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.

6. Results

Our study contains five phases. The first phase examines one case in depth ("base case") and the next four phases each examine the impact of different parameters on performance: hash table size, architecture, data sets, and concurrency. In this section, we present and analyze the results of these different phases.

6.1. Phase 1: Base Case

Our base case assesses the performance of the Sorting and Hashing algorithms with the following configuration of factors:
Configuration: (CPU, 16 cores, Enzo-10M, hash table factor 2 for Hashing) $\times$ 2 algorithms.

For each algorithm, we measured the total execution time, along with the sub-times for the primary parallel operations and routines. Additionally, we measured the overhead time for memory allocations and deallocations. The results of the Sorting and Hashing CPU-based experiments are presented in Tables 1 through 3.

Table 1: Comparison of overall execution time (sec) for the CPU execution time (sec) for the Sorting and Hashing 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	Hashing
Main Computation	0.5	0.6
Total Time	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. From Tables 1 and 3, Hashing performed comparably with Sorting, with the main hashing operations contributing 67% of the 0.9-second total runtime.

Table 2: Individual CPU phase times (sec) 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

Table 3: Individual CPU phase times (sec) for the Hashing algorithm

Phase	CPU Time
GetFacePoints	0.2
Scatter	0.1
CheckForMatches	0.3
StreamCompact	0.1
ComputeFaceHash	0.05
Overhead	0.1
Total time	0.9

6.2. Phase 2: Hash Table Size

In this phase, we study the effect of the hash table size on the performance of the Hashing algorithm, using the following set of factors:
Configuration: (CPU, 16 cores, Enzo-10M) $\times$ 5 different hash table proportions.
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 4 show that Hashing is only modestly affected by the hash table multiplier. This primarily occurs because the hash values are recomputed each iteration, leading to a slower decrease in collisions, which results in a more-stable number of hashing iterations. Moreover, in memory-poor environments, lower multipliers can be used with only a modest slowdown in execution time.

Table 4: Performance of Hashing (sec) as a function of hash table size multiplier.

Multiplier	0.5X	1X	2X	4X	8X
Total Time	1.0	0.9	0.9	0.8	0.8

6.3. Phase 3: Architecture

In this phase, we assess the performance of the algorithms on the GPU architecture with the Enzo-10M data set.
Configuration: (GPU, Enzo-10M) $\times$ 2 algorithms.

Table 5: GPU execution time (sec) for the two algorithms, along with the main computation time for the sort-and-reduction and hashing loop phases.

Time	Sorting	Hashing
Main Computation	0.5	0.2
Total Time	0.7	0.4

From Table 5, we observe that Hashing achieves a faster run time than Sorting. Moreover, Hashing devotes only half of its 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 6 for GPU sub-times of the Sorting algorithm. Table 7 shows that the Scatter and CheckForMatches parallel routines account for at least half of the work for Hashing. This contrasts slightly from the equivalent CPU findings of Phase 1, in which the algorithm spent a larger percentage of the time on the StreamCompact and ScanInclusive operations. The results of Table 7 indicate that the GPU significantly reduced the runtime of these parallel operations.

Table 6: Individual GPU phase 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 7: Individual GPU phase times (sec) for the Hashing algorithm

Phase	GPU Time
GetFacePoints	0.1
Scatter	0.1
CheckForMatches	0.1
StreamCompact	4.1e-02
ComputeFaceHash	8.0e-03
Overhead	0.1
Total time	0.4

6.4. Phase 4: Data Sets

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 (Sorting and Hashing).
Configuration: (CPU, 16 cores, GPU) $\times$ 6 data sets $\times$ 2 algorithms.

Table 8 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 sets' meshes, as evident from the increase in runtime between the pairs of regular and restructured data sets. Further corroborating this observation, Table 8 also shows that Sorting realizes a nearly 1.5–time speedup in total runtime when presented with the restructured version of a data set on the GPU architecture. Contrarily, Hashing maintains stable execution times regardless of the cell locality in data sets.

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

Table 8: CPU and GPU execution times, in seconds, for different data set/algorithm pairs

Data set	CPU		GPU	
	Sorting	Hashing	Sorting	Hashing
Enzo-10M	0.9	0.9	0.7	0.4
Nek-50M	4.3	4.3	3.3	2.1
Enzo-80M	7.4	7.3	7.4	7.3
Re-Enzo-10M	1.2	0.9	1.0	0.4
Re-Nek-50M	5.5	4.5	5.3	2.2
Re-Enzo-80M	9.2	7.7	10.1	6.5

6.5. Phase 5: Concurrency

In this phase, we investigate the CPU runtime performance of both Sorting and Hashing using different numbers of hardware cores with the base case Enzo-10M data set and

its corresponding Re-Enzo-10M data set.
Configuration: (CPU) $\times$ 6 different concurrency levels $\times$ 2 data sets $\times$ 2 algorithms.
Tables 9 and 10 show that, although Sorting performs better than Hashing on configurations of 8 cores or fewer, Hashing 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 Hashing. This observation demonstrates clear parallelism; however, the speedup is sub-linear.

Table 9: Impact of the number of CPU cores on the execution time (sec) for Sorting and Hashing using Enzo-10M and Re-Enzo-10M

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

A review of Hashing 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.

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

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

7. Comparing to Existing Serial Implementations

In Section 6.5, Hashing 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 Hashing and motivates a comparison with the serial EFC implementations of community visualization packages. This section compares the runtime of serial Hashing (1-core) with that of the VTK `vtkUnstructuredGridGeometryFilter` and VisIt `avtFacelistFilter`, both of which are serial, single-threaded algorithms for EFC.

In Table 11, we observe that the VisIt algorithm outperforms both the VTK and Hashing algorithms on all of the data sets from Section 6.4, while Hashing performs comparably with the VTK implementation. The overall weak performance of Hashing 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, Hashing is neither optimized nor designed for 1-core execution; specifically, it introduces extra instructions to resolve hash collisions that are unnecessary in this setting. Contrarily, both VisIt and VTK are optimized specifically for single-core, non-parallel environments, leading to better runtimes than Hashing on the majority of the datasets. However, in a parallel setting, both Sorting and Hashing achieve better runtime performance than the serial algorithms

Table 11: 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	Hashing
Enzo-10M	6.2	1.4	10.8
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 two novel algorithms for external facelist calculation. They are thought to be the first shared-memory parallel algorithms and provide portable performance via the data-parallel primitive (DPP) paradigm. Our experiments reveal that the hashing-based variant outperforms the sorting-based variant on GPU architectures and with large, complex data sets. Additionally, the DPP approach leads to improved runtime performance as concurrency increases.

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.

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Award Number 14-017566. It was also supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. Hank Childs is grateful for support from

the DOE Early Career Award, Contract No. DE-FG02-13ER26150, Program Manager Lucy Nowell.

References

[A*11] AHERN S., ET AL.: Scientific Discovery at the Exascale: Report for the DOE ASCR Workshop on Exascale Data Management, Analysis, and Visualization, July 2011. 2

[A*13] AMENT M., ET AL.: GPU-Accelerated Visualization. In *High Performance Visualization: Enabling Extreme-Scale Scientific Insight*, Bethel W. E., Childs H., Hansen C., (Eds.). CRC Press, Boca Raton, FL, USA, 2013, pp. 223–259. 2

[AGM*12] AYACHIT U., GEVECI B., MORELAND K., PATCHETT J., AHRENS J.: The ParaView Visualization Application. In *High Performance Visualization—Enabling Extreme-Scale Scientific Insight*. Oct. 2012, pp. 383–400. 2

[AT95] ABRAM G., TREINISH L. A.: *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. 2

[avt16] Apr. 2016. https://github.com/visit-vis/VisIt/blob/master/avt/Filters/avtFacelistFilter.C. 3

[BH11] BELL N., HOBEROCK J.: Thrust: A productivity-oriented library for CUDA. In *GPU Computing Gems*, Hwu W.-M., (Ed.). Elsevier/Morgan Kaufmann, 2011, pp. 359–371. 2

[Ble90] BLELLOCH G. E.: *Vector models for data-parallel computing*, vol. 356. MIT press Cambridge, 1990. 2

[C*12] CHILDS H., ET AL.: VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data. In *High Performance Visualization—Enabling Extreme-Scale Scientific Insight*. Oct. 2012, pp. 357–372. 2

[CGS*13] CHILDS H., GEVECI B., SCHROEDER W., MEREDITH J., MORELAND K., SEWELL C., KUHLEN T., BETHEL E. W.: Research Challenges for Visualization Software. *IEEE Computer 46*, 5 (May 2013), 34–42. 2

[Com16] COMPUTATIONAL ENGINEERING INTERNATIONAL, INC.: *EnSight website*, Apr 2016. URL: http://www.ceisoftware.com/. 2

[FLK08] FISCHER P. F., LOTTES J. W., KERKEMEIER S. G.: nek5000 Web page, 2008. http://nek5000.mcs.anl.gov. 6

[Leg90] LEGENSKY S. M.: Interactive Investigation of Fluid Mechanics Data Sets. In *VIS '90: Proceedings of the 1st conference on Visualization '90* (1990), IEEE Computer Society Press, pp. 435–439. 2

[Les15] LESSLEY B.: *Directed Research Project: External Facelist Calculation with Data-Parallel Primitives*. Technical Report , University of Oregon, Eugene, OR, USA, Nov. 2015. https://www.cs.uoregon.edu/Reports/DRP-201511-Lessley.pdf. 4

[LLN*15] LARSEN M., LABASAN S., NAVRÁTIL P., MEREDITH J., CHILDS H.: Volume Rendering Via Data-Parallel Primitives. In *Proceedings of EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV)* (Cagliari, Italy, May 2015), pp. 53–62. 2

[LMNC15] LARSEN M., MEREDITH J., NAVRÁTIL P., CHILDS H.: Ray-Tracing Within a Data Parallel Framework. In *Proceedings of the IEEE Pacific Visualization Symposium* (Hangzhou, China, Apr. 2015), pp. 279–286. 2

[LSA12] LO L.-T., SEWELL C., AHRENS J.: PISTON: A portable cross-platform framework for data-parallel visualization operators. Eurographics Symposium on Parallel Graphics and Visualization, pp. 11–20. 2

[MAGM11] MORELAND K., AYACHIT U., GEVECI B., MA K.-L.: 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* (October 2011), pp. 97–104. 2

[MAPS12] MEREDITH J. S., AHERN S., PUGMIRE D., SISNEROS R.: EAVL: the extreme-scale analysis and visualization library. In *Eurographics Symposium on Parallel Graphics and Visualization, EGPGV* (May 2012), The Eurographics Association, pp. 21–30. 2

[MC10] MEREDITH J. S., CHILDS H.: Visualization and Analysis-Oriented Reconstruction of Material Interfaces. *Computer Graphics Forum (CGF) 29*, 3 (June 2010), 1241–1250. 3

[MMA*13] MAYNARD R., MORELAND K., ATYACHIT U., GEVECI B., MA K.-L.: Optimizing threshold for extreme scale analysis. In *IS&T/SPIE Electronic Imaging* (2013), International Society for Optics and Photonics, pp. 86540Y–86540Y. 2

[MSPA12] MEREDITH J. S., SISNEROS R., PUGMIRE D., AHERN S.: 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* (2012), ACM, pp. 11–19. 2

[MSU*16] MORELAND K., SEWELL C., USHER W., TA LO L., 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 (CG&A)* (May/June 2016). To appear. 2

[NZIS13] NIESSNER M., ZOLLHÖFER M., IZADI S., STAMMINGER M.: Real-time 3d reconstruction at scale using voxel hashing. *ACM Transactions on Graphics (TOG)* (2013). 3

[OBB*04] O'SHEA B. W., BRYAN G., BORDNER J., NORMAN M. L., ABEL T., HARKNESS R., KRITSUK A.: Introducing Enzo, an AMR Cosmology Application. *ArXiv Astrophysics e-prints* (Mar. 2004). arXiv:astro-ph/0403044. 6

[SM15] SCHROOTS H. A., MA K.-L.: Volume Rendering with Data Parallel Visualization Frameworks for Emerging High Performance Computing Architectures. In *SIGGRAPH Asia 2015 Visualization in High Performance Computing* (2015), SA '15, ACM, pp. 3:1–3:4. 2

[SML98] SCHROEDER W., MARTIN K. M., LORENSEN W. E.: *The Visualization Toolkit (2nd Ed.): An Object-oriented Approach to 3D Graphics*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998. 2

[UJK*89] UPSON C., JR. T. F., KAMINS D., LAIDLAW D. H., SCHLEGEL D., VROOM J., GURWITZ R., VAN DAM A.: The application visualization system: A computational environment for scientific visualization. *Computer Graphics and Applications 9*, 4 (July 1989), 30–42. 2

[vtk16a] Apr. 2016. http://www.vtk.org/doc/nightly/html/classvtkUnstructuredGridGeometryFilter.html. 3

[vtk16b] Apr. 2016. https://gitlab.kitware.com/vtk/vtk-m/. 3