Title

Maximal Clique Enumeration with Data-Parallel Primitives

ISBN

9781538606179

Authors

Lessley, Brenton

Perciano, Talita

Mathai, Manish

et al.

Publication Date

2017-10-01

DOI

10.1109/ldav.2017.8231847

Peer reviewed

Maximal Clique Enumeration with Data-Parallel Primitives

Brenton Lessley*
University of Oregon

Talita Perciano†
Lawrence Berkeley Nat'l Lab

Manish Mathai‡
University of Oregon

Hank Childs§
University of Oregon

E. Wes Bethel¶
Lawrence Berkeley Nat'l Lab

Abstract

The enumeration of all maximal cliques in an undirected graph is a fundamental problem arising in several research areas. We consider maximal clique enumeration on shared-memory, multi-core architectures and introduce an approach consisting entirely of data-parallel operations, in an effort to achieve efficient and portable performance across different architectures. We study the performance of the algorithm via experiments varying over benchmark graphs and architectures. Overall, we observe that our algorithm achieves up to a 33-time speedup and 9-time speedup over state-of-the-art distributed and serial algorithms, respectively, for graphs with higher ratios of maximal cliques to total cliques. Further, we attain additional speedups on a GPU architecture, demonstrating the portable performance of our data-parallel design.

1 INTRODUCTION

Over the past decade, supercomputers have gone from commodity clusters built out of nodes containing a single CPU core to nodes containing small numbers of CPU cores to nodes containing many cores, whether self-hosted (i.e., Xeon Phi) or from accelerators (typically GPUs). For software developers, this has led to a significant change. One decade ago, software projects could target only distributed-memory parallelism, and, on a single node, could use a single-threaded approach, typically with `C` or `Fortran`. Now, software projects must consider shared-memory parallelism in addition to distributed-memory parallelism. Further, because the architectures on leading edge supercomputers vary, software projects have an additional difficulty, namely how to support multiple architectures simultaneously. One approach to this problem is to maintain separate implementations for separate architectures. That is, to have a CUDA implementation for NVIDIA GPUs and a TBB implementation for Intel architectures. That said, this approach has drawbacks. For one, the software development time increases, as modules need to be developed once for each architecture. Another problem caused by this approach is a lack of future-proofing; as supercomputers adopt new processor architectures (e.g., FPGA), the code for each module must be re-written, or possibly even re-thought.

Data-parallel primitives (DPPs) [6] is an approach for developing a single code base that can run over multiple architectures in a portably performant way. DPPs are customizable building blocks, meaning the algorithm design consists in composing these building blocks to solve a problem. To be a DPP, an algorithm needs to execute in $O(logN)$ time on an array of size N, provided there are N or more cores to work with. Well-known patterns, such as *map*, *reduce*, *gather*, *scatter*, and *scan*, meet this property, and are some of the most commonly used DPPs. While programming with DPPs requires re-thinking algorithms, the payoff comes in reduced code size, reduced development time, portable performance, and the ability to port to new architectures with reduced effort.

With this paper, we explore the maximal clique enumeration problem in the context of DPPs. Our motivator to pursue this work was an image processing problem that required maximal cliques and also aimed to support multiple architectures. In fact, very recent algorithms for the analysis of experimental image data take advantage of graphical models with maximal clique analysis and high performance computing techniques as in [37, 38]. That said, we have found that, for certain conditions, our approach is competitive with leading maximal clique implementations. We focus our comparisons on state-of-the-art maximal clique solvers. In our experiments, we find that our DPP-based algorithm is faster than the leading solutions for some graphs, namely those with higher ratios of maximal cliques to total cliques. Overall, the contribution of this paper is an important demonstration that the DPPs approach can work well for graphs, as well as a specific algorithm for maximal clique enumeration.

2 BACKGROUND AND RELATED WORK

2.1 Maximal Clique Enumeration

A *graph* G consists of a set of *vertices* V, some pairs of which are joined to form a set of *edges* E. A subset of vertices $C \subseteq V$ is a *clique*, or *complete subgraph*, if each vertex in C is connected to every other vertex in C via an edge. C is a *maximal clique* if its vertices are not all contained within any other larger clique in G. The size of a clique can range from zero—if there are no edges in G—to the number of vertices in V, if every vertex is connected to every other vertex (i.e., G is a complete graph). The *maximum clique* is the clique of largest size within G, and is itself maximal, since it cannot be contained within any larger-sized clique. The task of finding all maximal cliques in a graph is known as *maximal clique enumeration* (MCE). Figure 1 illustrates a graph with 6 vertices and 9 undirected edges. An application of MCE on this graph would search through 15 total cliques, of which only 3 are maximal.

The maximum number of maximal cliques possible in G is exponential in size; thus, MCE is considered an NP-Hard problem for general graphs in the worst case [34]. However, for certain sparse graph families that are encountered in practice (e.g., bipartite and planar), G typically contains only a polynomial number of cliques, and numerous algorithms have been introduced to efficiently perform MCE on real-world graphs. A brief survey of prior MCE research, in-

*e-mail:blessley@cs.uoregon.edu
†e-mail:tperciano@lbl.gov
‡e-mail:mmathai@cs.uoregon.edu
§e-mail:hank@cs.uoregon.edu
¶e-mail:ewbethel@lbl.gov

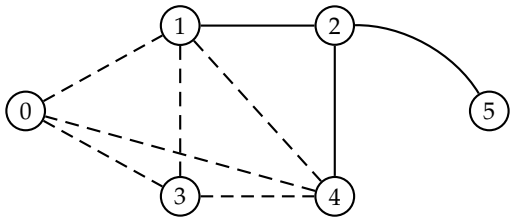


Figure 1: Undirected graph with 6 vertices and 9 edges. This graph consists of 15 total cliques, 3 of which are maximal cliques. The maximal cliques are 2-5, 1-2-4, and 0-1-3-4, the latter of which is the largest-sized clique in the graph. This maximum clique is denoted with dotted edges. Refer to subsection 2.1 for details.

cluding the algorithms we compare against in our study, is provided later in this section.

2.2 Related Work

2.2.1 Visualization and Data Parallel Primitives

While we are considering the DPP approach for a graph algorithm, there have been several similar studies for scientific visualization. In each case, they have studied a specific visualization algorithm: Maynard et al. for thresholding [32], Larsen et al. for ray-tracing [24] and unstructured volume rendering [23], Schroots and Ma for cell-projected volume rendering [40], Lessley et al. for external facelist calculation [27], Lo et al. for isosurface generation [29], Widanagamaachchi et al. and Harrison et al. for connected component finding [44, 15], Carr et al. for contour tree computation [8], and Li et al. for wavelet compression [28]. Moreover, several DPP-based algorithms have been introduced for the construction of spatial search structures in the visualization domain (e.g., ray tracing), particularly for real-time use on graphics hardware. These include *k*-d trees, uniform grids, two-level grids, bounding volume hierarchies (BVH), and octrees [29, 47, 19, 18, 17, 22, 25].

Finally, our experiments make use of the VTK-m framework [36], which is the same framework used in several of these scientific visualization studies. VTK-m is effectively the unification of three predecessor visualization libraries—DAX [35], EAVL [33], and PISTON [29]—each of which were constructed on DPP with an aim to achieve portable performance across multiple many-core architectures.

2.2.2 Maximal Clique Enumeration

Several studies have introduced algorithms for MCE. These algorithms can be categorized along two dimensions: traversal order of clique enumeration and whether it is serial or parallel.

Serial depth-first MCE uses a backtracking search technique to recursively *expand* partial cliques with candidate vertices until maximal cliques are discovered. This process represents a search forest in which the set of vertices along a path from a root to a child constitutes a clique, and a path from a root to a leaf vertex forms a maximal clique. Upon discovering a maximal clique, the algorithm backtracks to the previous partial clique and branches into a recursive expand operation with another candidate vertex. This approach limits the size of the search space by only exploring search paths that will lead to a maximal clique.

The works in [5, 7] introduce two of the earliest serial backtracking-based algorithms for MCE; the implementation of the algorithm in [7] attained more prominence due to its simplicity and effective performance for most practical graphs. The algorithms proposed in [16, 20, 43, 26, 10, 31] build upon [7] and devise similar depth-first, tree-based search algorithms. Tomita et al. [42] optimize the clique expansion (*pivoting*) strategy of [7] to prune unnecessary subtrees of the search forest, make fewer recursive calls, and demonstrate very fast execution times in practice, as compared to [7, 43, 10, 31]. Eppstein et al. [13, 14] develop a variant of [7] that uses a degeneracy ordering of candidate vertices to order the sequence of recursive calls made at the top-most level of recursion. Then, during the inner levels of recursion, the improved pivoting strategy described in [42] is used to recurse on candidate vertices. [14] also introduces two variants of their algorithm, and propose a memory-efficient version of [42] using adjacency lists. Experimental results indicate that [14] is highly competitive with the memory-optimized [42] on large sparse graphs, and within a small constant factor on other graphs.

Distributed-memory, depth-first MCE research has also been conducted. Du et al. [12] present an approach that assigns each parallel process a disjoint subgraph of vertices and then conducts serial depth-first MCE on a subgraph; the union of outputs from each process represents the complete set of maximal cliques. Schmidt et al. [39] introduce a parallel variant of [7] that improves the process load balancing of [12] via a dynamic work-stealing scheme. In this approach, the search tree is explored in parallel among compute nodes, with unexplored search subtrees dynamically reassigned to underutilized nodes. Lu et al. [30] and Wu et al. [45] both introduce distributed parallel algorithms that first enumerate maximal, duplicate, and non-maximal cliques, then perform a post-processing phase to remove all the duplicate and non-maximal cliques. Dasari et al. [11] expand the work of [14] to a distributed, MapReduce environment, and study the performance impact of various vertex-ordering strategies, using a memory-efficient partial bit adjacency matrix to represent vertex connectivity within a partitioned subgraph. Svendsen et al. [41] present a distributed MCE algorithm that uses an enhanced load balancing scheme based on a carefully chosen ordering of vertices. In experiments with large graphs, this algorithm significantly outperformed the algorithm of [45].

Serial breadth-first MCE iteratively expands all *k*-cliques into $(k+1)$ cliques, enumerating maximal cliques in increasing order of size. The number of iterations is typically equal to the size of the largest maximal clique. Kose et al. [21] and Zhang et al. [46] introduce algorithms based on this approach. However, due to the large memory requirements of these algorithms, depth-first-based algorithms have attained more prevalence in recent MCE studies [39, 41].

Shared-memory breadth-first MCE on a single node has not been actively researched to the best of our knowledge. In this study, we introduce a breadth-first approach that is designed in terms of data-parallel primitives. These primitives enable MCE to be conducted in a massively-parallel fashion on shared-memory architectures, including GPU accelerators, which are designed to perform this data-parallel computation. We compare the performance of our algorithm against that of Tomita et al.[42], Eppstein et al.[14] and Svendsen et al.[41]. These studies provide suitable benchmark comparisons because they each introduce the leading MCE implementations in their respective categories: Tomita et al. and Eppstein et al. for serial depth-first MCE and Svendsen et al. for distributed-memory, depth-first MCE.

3 DATA-PARALLEL PRIMITIVES

The new algorithm presented in this study is described in terms of data-parallel primitives, or DPPs. These primitives provide high-level abstractions and permit new algorithms to be platform-portable across many environments. The following primitives are used in our algorithm implementation:

- *Map*: Applies an operation on all elements of the input array, storing the result in an output array of the same size, at the same index;
- *Reduce*: Applies a summary binary operation (e.g., summation or maximum) on all elements of an input array, yielding a single output value. *ReduceByKey* is a variation which performs Reduce on the input array, segmenting it based on a key or unique data value in the input array, yielding an output value for each key;
- *Exclusive Scan*: Calculates partial aggregates, or a prefix sum, for all values in an input array and stores them in an output array of the same size;
- *Scatter*: Writes each value of an input data array into an index in an output array, as specified in the array of indices;
- *Compact*: Applies a unary predicate (e.g., if an input element is greater than zero) on all values in an input array, filtering out all the values which do not satisfy the predicate. Only the remaining elements are copied into an output array of an equal or smaller size;
- *Unique*: Ignores duplicate values which are adjacent to each other, copying only unique values from the input array to the output array of the same or lesser size; and
- *Unzip*: Transforms an input array of pairs into two arrays of the same size, one with all the first components and the other with all the second components.

Most DPPs can be extended with a developer supplied functor, enabling custom operations. For example, if a developer wants to extract all perfect squares from an input array, they can write a unary predicate functor that checks if the fractional parts of the square root of each value is zero. The *Compact* DPP then execute the functor over the entire input array in parallel.

4 ALGORITHM

This section presents our new DPP-based MCE algorithm, which consists of an initialization procedure followed by the main computational algorithm. The goal of the initialization procedure is to represent the graph data in a compact format that fits within shared memory. The main computational algorithm enumerates all of the maximal cliques within this graph. The implementation of this algorithm is available online [4], for reference and reproducibility.

4.1 Initialization

In this phase, we construct a compact graph data structure that consists of the following four component vectors:

- I: List of vertex Ids. The contents of the list are the lower-value vertex Ids of each edge;
- C: List containing the number of edges per vertex in I;
- E: Segmented list in which each segment corresponds to a vertex v in I, and each vertex Id within a segment corresponds to an edge of v. The length of a segment is equal to the number of edges incident to its vertex;
- V: List of indices into the edge list, E, for each vertex in I. Each index specifies the start of the vertex's segment of edges.

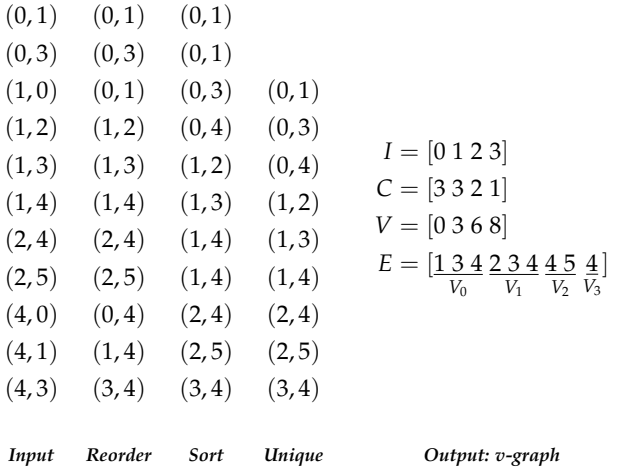


Figure 2: Initialization process to obtain a v-graph representation of the undirected graph from Figure 1. Starting with an unordered set of (possibly) directed edges, we first reorder the two vertices in each edge to ascending Id order. Second, all edges are sorted in ascending order. Third, all duplicate edges are removed, leaving unique undirected edges. These edges are then further processed to construct the output v-graph. Refer to subsection 4.1 for details.

This data structure is known as a *v-graph* [6] and it is constructed using only data-parallel operations. The compressed form of the *v-graph* in turn enables efficient data-parallel operations for our MCE algorithms.

We construct the *v-graph* as follows. Refer to algorithm 1 for pseudocode of these steps and Figure 2 for an illustration of the input and output.

1. **Reorder**: Accept either an undirected or directed graph file as input; if the graph is directed, then it will be converted into an undirected form. We re-order an edge (b, a) to (a, b) if $b > a$. This maintains the ascending vertex order that is needed in our algorithms;
2. **Sort**: Invoke a data-parallel *Sort* primitive to arrange all edge pairs in ascending order (line 9 of algorithm 1). The input edges in Figure 2 provide an example of this sorted order;
3. **Unique**: Call the *Unique* data-parallel primitive to remove all duplicate edges (line 10 of algorithm 1). This step is necessary for directed graphs, which may contain bi-directional edges (a, b) and (b, a);
4. **Unzip**: Use the *Unzip* data-parallel primitive to separate the edge pairs (a_i, e_i) into two arrays, A and E, such that all of the first-index vertices, a_i, are in A and all of the second-index vertices, e_i, are in E (line 11 of algorithm 1). For example, using the edges from Figure 2, we can create the following A and E arrays:

$$\begin{bmatrix} 0&0&0&1&1&1&2&2&3 \\ 1&3&4&2&3&4&4&5&4 \end{bmatrix} \xrightarrow{Unzip} \begin{matrix} A: [0\ 0\ 0\ 1\ 1\ 1\ 2\ 2\ 3] \\ E: [1\ 3\ 4\ 2\ 3\ 4\ 4\ 5\ 4] \end{matrix}$$

   The array E represents the edge list in our *v-graph* structure;
5. **Reduce**: Use the *ReduceByKey* data-parallel primitive to compute the edge count for each vertex (line 13 of algorithm 1). Using the arrays A and E from step 3, this

Algorithm 1: Pseudocode for the construction of the v-graph data structure, which consists of vertex Ids (I), segmented edges (E), per-vertex indices into the edge list (V), and per-vertex edge counts (C). M is the number of input edges, N_{edges} is the number of output edges, and N_{verts} is the number of output vertices. See Section 4.1 for details.

```
1  /*Input*/
2  Array: int edgesIn[M]
3  /*Output*/
4  Array: int C[N_verts], E[N_edges], I[N_verts], V[N_verts]
5  /*Local Objects*/
6  Array: int edgesOrdered[M], edgesSorted[M],
     edgesUndirected[N_edges], A[N_edges]
7  Int: N_edges, N_verts
8  edgesOrdered←Reorder(edgesIn)
9  edgesSorted←Sort(edgesOrdered)
10 edgesUndirected←Unique(edgesSorted)
11 A, E←Unzip(edgesUndirected)
12 N_edges ← |E|
13 C, I←ReduceByKey(A, 1⃗)
14 N_verts ← |I|
15 V←ExclusiveScan(C)
16 //Continue with Algorithm 2 after returning.
17 return (C, E, I, V)
```

operation counts the number of adjacent edges from E that are associated with each unique vertex in A. The resulting output arrays represent the lists I and C in our *v-graph* structure:

$$I: \quad [0\ 1\ 2\ 3]$$

$$C: \quad [3\ 3\ 2\ 1]$$

6. **Scan**: Run the *ExclusiveScan* data-parallel operation on the edge counts array, C, to obtain indices into the edge list, E, for each entry in I (line 15 of algorithm 1). This list of indices represents the list V in our *v-graph* (see Figure 2). In our running example, vertex 0 has 3 edges and vertex 1 has 3 edges, representing index segments 0-2 and 3-5 in E, respectively. Thus, vertex 0 and vertex 1 will have index values of 0 and 3, respectively:

$$C: [3\ 3\ 2\ 1] \xrightarrow{ExclusiveScan} V: [0\ 3\ 6\ 8]$$

4.2 Hashing-Based Algorithm

We now describe our hashing-based algorithm to perform maximal clique enumeration, which comprises the main computational work. This algorithm takes the *v-graph* from the initialization phase as input. In the following subsections we provide an overview of the algorithm, along with a more detailed, step-by-step account of the primary data-parallel operations.

4.2.1 Algorithm Overview

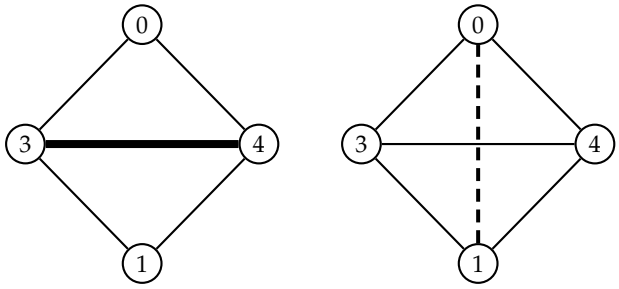
We perform MCE via a bottom-up scheme that uses multiple iterations, each consisting of a sequence of data-parallel operations. During the first iteration, all 2-cliques(edges) are expanded into zero or more 3-cliques and then tested for maximality. During the second iteration, all of these new 3-cliques are expanded into zero or more 4-cliques and then tested for maximality, so on and so forth until there are no newly-expanded cliques. The number of iterations is equal to the size of the *maximum* clique, which itself is *maximal* and cannot be expanded into a larger clique. Figure 3 presents the progression of clique expansion for the example graph in Figure 1.

2-cliques: 0-1 | 0-3 | 0-4 | 1-2 | 1-3 | 1-4 | 2-4 | **2-5** | 3-4

3-cliques: 0-1-3 | 0-1-4 | 0-3-4 | **1-2-4** | 1-3-4

4-cliques: **0-1-3-4**

Figure 3: Clique expansion process for the example undirected graph of Figure 1. In the first iteration, only 2-cliques (edge pairs) are considered. Then, these cliques are expanded into larger 3-cliques. The 4-clique in the final iteration cannot be expanded further since it is maximal; this clique also cannot be expanded further because it is the maximum-sized clique. All maximal cliques are denoted in boxes with bold font. See subsubsection 4.2.1 for details.



(a) 3-cliques: 0-3-4, 1-3-4 (b) 4-clique: 0-1-3-4

Figure 4: Example of clique expansion. As shown in (a), the set of four vertices, 0-1-3-4, is composed of two 3-cliques, 0-3-4 and 1-3-4. Both of these cliques share a common 2-clique, 3-4, which is highlighted in bold. If $(0, 1)$ is an edge in the graph (dotted line), as shown in (b), then 0-1-3-4 is a 4-clique. Refer to subsubsection 4.2.2 for details.

During this process, we assess whether a given k-clique is a subset of one or more larger $(k+1)$-cliques. If so, then the k-clique is marked as non-maximal and the new $(k+1)$-cliques are stored for the next iteration; otherwise, the k-clique is marked as maximal and discarded from further computation.

In order to determine whether a k-clique is contained within a larger clique, we use a hashing scheme that searches through a hash table of cliques for another k-clique with the same hash value. These matching cliques share common vertices and can be merged into a larger clique if certain criteria are met. Thus, hashing is an important element to our algorithm. Figure 4 illustrates the clique merging process between two different k-cliques.

4.2.2 Algorithm Details

Within each iteration, our MCE algorithm consists of three phases: dynamic hash table construction, clique expansion, and clique maximality testing. These phases are conducted in a sequence and invoke only data-parallel operations; re-

Algorithm 2: Pseudocode for the hashing-based maximal clique enumeration algorithm. See Section 4.2 for a description of this algorithm.

1 /*Input from Algorithm 1*/
2 **v-graph:** int C[N_{verts}], E[N_{edges}], I[N_{verts}], V[N_{verts}]
3 /*Output*/
4 **Array:** int[$N_{maximal} \times N_{maximalVerts}$]: maxCliques
5 /*Local Objects*/
6 **int:** $N_{cliques}$, $N_{newCliques}$, N_{chains}, $N_{maximal}$, $N_{maximalVerts}$, $iter$
7 **Array:** int[$N_{cliques}$]: cliqueStarts, cliqueSizes, cliqueIds, sortedCliqueIds, newCliqueCounts, scanNew, writeLocations
8 **Array:** float[$N_{cliques}$]: hashes, sortedHashes
9 **Array:** int[$N_{newCliques}$]: newCliqueStarts
10 **Array:** int[$N_{cliques} \times (iter+1)$]: newCliques
11 **Array:** int[$N_{cliques} \times iter$]: cliques, isMaximal, hashTable
12 **Array:** int[$N_{newCliques} \times (iter-1)$]: repCliqueIds, repCliqueStarts, localIndices, vertToOmit
13 **Array:** int[N_{chains}]: uniqueHashes, chainStarts, chainSizes, scanChainSizes
14 $iter \leftarrow 2$
15 cliques←Get2-Cliques(E)
16 $N_{cliques} \leftarrow N_{edges}$
17 **while** $N_{cliques} > 0$ **do**
18 cliqueStarts← $[iter \times i], 0 \leq i < N_{cliques}$;
19 cliqueSizes← $[iter_i], 0 \leq i < N_{cliques}$;
20 cliqueIds← $[i], 0 \leq i < N_{cliques}$;
21 hashes←ComputeHash(cliques, cliqueStarts, cliqueSizes);
22 sortedHashes, sortedIds ←SortByKey(hashes, cliqueIds);
23 hashTable←Permute(sortedIds, cliques);
24 uniqueHashes, chainSizes ←ReduceByKey(sortedHashes, $\vec{1}$);
25 $N_{chains} \leftarrow$ |uniqueHashes|;
26 scanChainSizes←ScanExclusive(chainSizes);
27 chainStarts← [scanChainSizes[i] $\times$ $iter$];
28 isMaximal← $\vec{1}$;
29 newCliqueCounts, isMaximal ←FindCliques(v-graph, $iter$, cliqueStarts, chainStarts, chainSizes, hashTable, isMaximal);
30 $N_{newCliques}$, scanNew←ScanExclusive(newCliqueCounts);
31 writeLocations←Multiply(scanNew, $iter+1$);
32 newCliques← $\vec{0}$;
33 newCliques←GetCliques(v-graph, $iter$, writeLocations, chainStarts, chainSizes, hashTable, newCliques);
34 repCliqueIds← $[i_0 \ldots i_{iter-2}], 0 \leq i < N_{newCliques}$;
35 newCliqueStarts← $[iter \times i], 0 \leq i < N_{newCliques}$;
36 repCliqueStarts←Gather(repCliqueIds, newCliqueStarts);
37 localIndices, vertToOmit←Modulus($iter-1$, repCliqueIds);
38 isMaximal←TestForMaximal(repCliqueIds, repCliqueStarts, $iter-1$, localIndices, vertToOmit, chainStarts, chainSizes, hashTable, isMaximal, newCliques);
39 maxCliques = maxCliques + Compact(hashTable, isMaximal, IsIntValue(1));
40 $N_{cliques} \leftarrow N_{newCliques}$;
41 $iter \leftarrow iter+1$;
42 **end**
43 **return** (maxCliques)

fer to algorithm 2 for pseudocode of these phases and operations. In the following description, the 3-cliques from Figure 3 are used as a running example. We start from iteration 3 with the linear array

$$cliques = [\text{0-1-4 0-3-4 1-2-4 1-3-4 0-1-3}].$$

of length $(k = 3) \times (numCliques = 5) = 15$.

Dynamic Hash Table Construction: As our algorithm uses hashing as an integral component, we discuss the operations that are used to construct a hash table into which the cliques are hashed and queried.

First, each clique is hashed to an integer (line 21 of algorithm 2). This is done using the FNV-1a hash function [3], h, and taking the result modulo the number of cliques. Further, only the clique's last $k-1$ vertex indices are hashed. Only the last $(k-1)$ vertices are hashed because we just need to search (via a hash table) for matching $(k-1)$-cliques to form a new $(k+1)$-clique. For example, cliques 0-3-4 and 1-3-4 both hash their last two vertices to a common index, i.e., h(3-4), and can combine to form 0-1-3-4, since leading vertices 0 and 1 are connected (see Figure 4).

Next, we allocate an array, *hashTable*, of the same size as *cliques*, into which the cliques will be rearranged (permuted) in order of hash value. In our example, there are 5 cliques, each with an Id in $cliqueIds = [0\ 1\ 2\ 3\ 4]$. After applying the hash operation, these cliques have the hash values, $hashes = [1\ 0\ 4\ 0\ 1]$. Sorting *cliqueIds* in order of hash value (*hashes*), we obtain $sortedIds = [1\ 3\ 0\ 4\ 2]$ and $sortedHashes = [0\ 0\ 1\ 1\ 4]$ (line 22 of algorithm 2). A *ReduceByKey* operation (line 24 of algorithm 2) computes a count for each unique hash value in *hashes*: $unique = [0\ 1\ 4]$ and $count = [2\ 2\ 1]$. A series of *Scan*, *Map*, and *Scatter* primitives are then employed on *cliqueIds* and *counts* to construct offset index arrays (of length *numCliques*) into *hashTable*, denoted as *cliqueStarts* and *chainStarts*, respectively (lines 26 and 27 algorithm 2). Permuting *cliques* by *sortedIds* (line 23 of algorithm 2), we obtain

$$hashTable = [\underbrace{\text{0-3-4 1-3-4}}_{Chain_0}\ \underbrace{\text{0-1-4 0-1-3}}_{Chain_1}\ \underbrace{\text{1-2-4}}_{Chain_2}]$$

$$tableIndex = [\underbrace{0\ 1\ 2}_{Clique_0}\ \underbrace{3\ 4\ 5}_{Clique_1}\ \underbrace{6\ 7\ 8}_{Clique_2}\ \underbrace{9\ 10\ 11}_{Clique_3}\ \underbrace{12\ 13\ 14}_{Clique_4}]$$

$$cliqueIds = [0\ 1\ 2\ 3\ 4]$$

$$chainIds = [0\ 0\ 1\ 1\ 2]$$

$$cliqueStarts = [0\ 3\ 6\ 9\ 12]$$

$$chainStarts = [0\ 0\ 6\ 6\ 12],$$

with three *chains* of contiguous cliques, each sharing the same hash value. Since the cliques within a chain are not necessarily in sorted order, the chain must be *probed* sequentially using a constant number of lookups to find the clique of interest. This probing is also necessary since different cliques may possess the same hash value, resulting

in *collisions* in the chain. For instance, cliques 0-1-3 and 0-1-4 both hash to index $h(1\text{-}3) = h(1\text{-}4) = 1$, creating a collision in $Chain_1$. Thus, the hash function is important, as good function choices help minimize collisions, while poor choices create more collisions and, thus, more sequential lookups.

Clique Expansion: Next, a two-step routine is performed to identify and retrieve all valid $(k+1)$-cliques for each k-clique in *hashTable*. The first step focuses on determining the sizes of output arrays and the second step focuses on allocating and populating these arrays.

In the first step, a *Map* primitive computes and returns the number of $(k+1)$-cliques into which a k-clique can be expanded (line 29 of algorithm 2). The first step works as follows. For a given k-clique, i, at $cliqueStarts[i]$, we locate its chain at $chainStarts[i]$ and iterate through the chain, searching for another k-clique, j, with (a) a larger leading vertex Id and (b) the same ending $(k-1)$ vertices; these two criteria are needed to generate a larger clique and avoid duplicates (see Theorem 7.1 and Theorem 7.3). For each matching clique j in the chain, we perform a binary search over the adjacent edges of i in the v-graph edge list E to determine whether the leading vertices of i and j are connected. If so, then, by Theorem 7.1, cliques i and j can be expanded into a larger $(k+1)$-clique consisting of the two leading vertices and the shared $(k-1)$ vertices, in ascending order. The total number of expanded cliques for i is returned. In our example, this routine returns a counts array, $newCliqueCounts = [1\ 0\ 0\ 0\ 0]$, indicating that only clique 0-3-4 could be expanded into a new 4-clique; Figure 4 illustrates the generation of this 4-clique.

In the second step, an inclusive *Scan* primitive is invoked on *newCliqueCounts* to compute the sum of the clique counts, *numNewCliques* (line 30 of algorithm 2). The second step works as follows. This sum is used to allocate a new array, *cliques*, of size $numNewcliques \cdot (k+1)$ to store all of the $(k+1)$-cliques, along with a new offset index array with increments of $(k+1)$. With these arrays, we invoke a parallel *Map* operation that is identical to the *Map* operation of the first step, except that, upon discovery of a new $(k+1)$-clique, we write the clique out to its location in *cliques* (using the offset array), instead of incrementing a *newCliques* counter (line 32 of algorithm 2). For the running example, the new *cliques* array consists of the single 4-clique, 0-1-3-4.

Clique Maximality Test: Finally, we assess whether each k-clique is maximal or not. Prior to **Clique Expansion**, a bit array, *isMaximal*, of length *numCliques*, is initialized with all 1s. During the first step of **Clique Expansion**, if a clique i merged with one or more cliques j, then they all are encompassed by a larger clique and are not maximal; thus, we set $isMaximal[i] = isMaximal[j] = 0$, for all j. Since each $(k+1)$-clique includes $(k+1)$ distinct k-cliques—two of which are the ones that formed the $(k+1)$-clique—we must ensure that the remaining $k-1$ k-cliques are marked as non-maximal with a value of 0 in *isMaximal*. In our example, the 4-clique 0-1-3-4 is composed of 4 different 3-cliques: 1-3-4, 0-3-4, 0-1-4, and 0-1-3. The first two were already marked as non-maximal, but the remaining two are non-maximal as well, and need to be marked as so in this phase. Our approach for marking these remaining cliques as non-maximal is as follows.

First, we use a custom modulus map operator (line 34 of algorithm 2) to construct, in parallel, an array of length $numNewCliques \times (k-1)$, with $(k-1)$ local indices per new $(k+1)$-clique: $[2_0 \ldots k_0 \ldots 2_{numCliques-1} \ldots k_{numCliques-1}]$. Then, we parallelize over this index array via a *Map* operation (line 35 of algorithm 2) that, given an index $2 \leq i \leq k$ and corresponding clique $0 \leq t \leq numCliques - 1$, determines whether the k-clique formed by omitting vertex $t[i]$ is maximal or not. If the k-clique is discovered in *hashTable* (using the same hashing approach as in **Clique Expansion**), then it is marked as 0 in *isMaximal*. A *Compact* primitive then removes all k-cliques in *hashTable* that have $isMaximal = 0$, leaving only the maximal cliques, which are appended in an auxiliary array (line 36 of algorithm 2).

The algorithm terminates when $numNewCliques = 0$ (line 17 of algorithm 2). The generated *cliques* array of new $(k+1)$-cliques becomes the starting array (line 37 of algorithm 2) for the next iteration (line 38 of algorithm 2), if the termination condition is not met.

5 EXPERIMENTAL OVERVIEW

We assess the performance of our MCE algorithm in two phases, using a collection of benchmark input graphs and both CPU and GPU systems. In the first phase, we run our algorithm–denoted as *Hashing*—on a CPU platform and compare its performance with three state-of-the-art MCE algorithms—*Tomita* [42], *Eppstein* [14], and *Svendsen* [41]. In the second phase, we evaluate portable performance by testing *Hashing* on a GPU platform and comparing the runtime performance with that of the CPU platform, using a common set of benchmark graphs. The following subsections describe our software implementation, hardware platforms, and input graph datasets.

5.1 Software Implementation

Both of our MCE algorithms are implemented using the platform-portable VTK-m toolkit [4], which supports fine-grained concurrency for data analysis and scientific visualization algorithms. 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, i.e., CUDA code for NVIDIA GPUs and Threading Building Blocks (TBB) code for Intel CPUs. Thus, by refactoring an algorithm to be composed of VTK-m data-parallel primitives, it only needs to be written once to work efficiently on multiple platforms. In our experiments, the TBB configuration of VTK-m was compiled using the gcc compiler, the CUDA configuration using the nvcc compiler, and the VTK-m index integer (vtkm::Id) size was set to 64 bits. The implementation of this algorithm is available online [4], for reference and reproducibility.

5.2 Test Platforms

We conducted our experiments on the following two CPU and GPU platforms:

- CPU: A 16-core machine running 2 nodes, each with a 3.2 GHz Intel Xeon(R) E5-2667v3 CPU with 8 cores. This machine contains 256GB DDR4 RAM memory. All the CPU experiments use 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.

Graph	Collection	V	E	Max_{size}	$Cliques_{max}$	$Cliques_{all}$	Max_{ratio}
amazon0601	Stanford	403,394	3,387,388	11	1,023,572	18,043,744	0.06
cit-Patents	Stanford	3,774,768	16,518,947	11	14,787,032	36,180,638	0.41
email-Enron	Stanford	36,692	183,831	20	226,859	107,218,609	< 0.01
loc-Gowalla	Stanford	196,591	950,327	29	1,212,679	1,732,143,035	$\ll 0.01$
soc-wiki-Vote	Stanford	7,115	103,689	17	459,002	41,792,503	0.01
roadNet-CA	Stanford	1,965,206	2,766,607	4	2,537,996	2,887,325	0.88
brock200-2	DIMACS	200	9,876	12	431,586	6,292,399	0.07
hamming6-4	DIMACS	64	704	4	464	1,904	0.24
MANNa9	DIMACS	45	918	16	590,887	160,252,675	< 0.01
p_hat300-1	DIMACS	300	10,933	8	58,176	367,022	0.16
UG100k.003	DIMACS	100,000	14,997,901	4	10,589,956	19,506,096	0.54

Table 1: Statistics for a subset of the test graphs used in this study. Graphs are either from the Stanford Large Network Dataset Collection [2] or the DIMACS Challenge data set [1]. V is the number of graph vertices, E is the number of edges, Max_{size} is the size of the largest clique, $Cliques_{max}$ is the number of maximal cliques, $Cliques_{all}$ is the total number of cliques, and Max_{ratio} is the ratio of $Cliques_{max}$ to $Cliques_{all}$. Refer to subsection 5.3 for details.

5.3 Test Data Sets

We applied our algorithm to a selected set of benchmark and real-world graphs from the DIMACS Challenge [1] and Stanford Large Network Dataset collections [2]. Table 1 lists a subset of these test graphs, along with their statistics pertaining to topology and clique enumeration. For each graph, we specify the number of vertices (V), edges (E), maximum clique size (Max_{size}), number of maximal cliques ($Cliques_{max}$), number of total cliques ($Cliques_{all}$), and ratio of maximal cliques to total cliques (Max_{ratio}). The DIMACS Challenge data set includes a variety of benchmark instances of randomly-generated and topologically-challenging graphs, ranging in size and connectivity. The Stanford Large Network Data Collection contains a broad array of real-world directed and undirected graphs from social networks, web graphs, road networks, and autonomous systems, to name a few.

6 RESULTS

In this section, we present the results of our set of MCE experiments, which consists of two phases: CPU and GPU.

6.1 Phase 1: CPU

This phase assesses the performance of our *Hashing* algorithm on a CPU architecture with the set of graphs listed in Table 2 and Table 3.

For each graph in Table 2, the total runtime (in seconds) of *Hashing* is compared with that of *Tomita* and *Eppstein*, two serial algorithms that have demonstrated state-of-the-art performance for MCE. The set of graphs used for comparison was adopted from the paper of Eppstein et. al [14], which compared the CPU results of three newly-introduced MCE algorithms with that of the *Tomita* algorithm. In this phase, we report the best total runtime among these three algorithms as *Eppstein*. Moreover, we only test on those graphs from [14] that are contained with the DIMACS Challenge and Stanford Large Network Data collections. Among these graphs, 9 were omitted from the comparison because our *Hashing* algorithm exceeded available shared memory on our single-node CPU system (approximately 256GB). Each of these graphs has a very large number of non-maximal cliques relative to maximal cliques. Thus, most of these non-maximal cliques are progressively expanded and passed on to the next iteration of our algorithm, increasing the computational workload and storage requirements. Reducing our memory needs and formalizing the graph properties that lead to a high memory consumption by our algorithm will be investigated in future work.

Graph	Tomita	Eppstein	Hashing
amazon0601	**	3.59	**1.69**
cit-Patents	**	28.56	**3.27**
email-EuAll	**	1.25	2.24
email-Enron	31.96	0.90	17.91
roadNet-CA	**	2.00	**0.27**
roadNet-PA	**	1.09	**0.16**
roadNet-TX	**	1.35	**0.19**
brock200-2	0.55	1.22	0.71
hamming6-4	< 0.01	< 0.01	$<$ **0.01**
johnson8-4-4	0.13	0.24	0.50
johnson16-2-4	5.97	12.17	**5.10**
MANNa9	0.44	0.53	27.74
p_hat300-1	0.07	0.15	**0.07**
soc-wiki-Vote	0.96	1.14	6.14
keller4	5.98	11.53	7.22

Table 2: Total CPU execution times (sec) for our *Hashing* algorithm as compared to the serial *Tomita* and *Eppstein* algorithms, over a set of common test graphs. Results with double asterisk symbols indicate that the graph could not be processed due to memory limitations. Results in bold indicate that *Hashing* achieved the fastest execution time for that particular graph.

From Table 2, we observe that *Hashing* performed comparably or better on more than half—8 out of 15—of the test graphs. Using the graph statistics from Table 1, it is apparent that our algorithm performs best on graphs with a high ratio of maximal cliques to total cliques, Max_{ratio}. This is due to the fact that, upon identification, maximal cliques are discarded from further computation in our algorithm. So, the larger the number of maximal cliques, the smaller the amount of computation and memory accesses that will need to be performed. *Tomita* and *Eppstein* do not perform as well on these types of graphs due to the extra sequential recursive branching and storage of intermediary cliques that is needed to discover a large number of maximal cliques. From Table 2 we see that *Tomita* exceeded the available shared memory of its CPU system (approximately 3GB) for the majority of the graphs on which we possess the faster runtime.

Next, we compare *Hashing* to the CPU-based distributed-memory MCE algorithm of Svendsen et al. [41], which we refer to as *Svendsen*. We use the set of 12 test graphs from [41],

Graph	*Svendsen*	*Hashing*
cit-Patents	109	**3.27**
loc-Gowalla	112	545.25
UG100k.003	353	**5.39**
UG1k.30	129	**11.10**

Table 3: Total CPU execution times (sec) for our *Hashing* algorithm as compared to the distributed-memory *Svendsen* algorithm, over a set of common test graphs. Results in bold indicate that *Hashing* achieved the fastest execution time for that particular graph.

Graph	*Tomita-Eppstein*	*Hashing*-CPU	*Hashing*-GPU
amazon0601	3.59	1.69	**0.86**
email-Enron	0.90	17.91	15.56
email-EuAll	1.25	2.24	1.52
roadNet-CA	2.00	0.27	**0.17**
roadNet-PA	1.09	0.16	**0.11**
roadNet-TX	1.35	0.19	**0.13**
brock200-2	0.55	0.71	**0.45**
p_hat300-1	0.07	0.07	0.09
soc-wiki-Vote	0.96	6.14	4.78

Table 4: Total GPU execution times (sec) for our *Hashing* algorithm over a set of test graphs. For comparison, the best execution time between *Tomita* and *Eppstein* is listed, along with the CPU execution time of *Hashing*. Results in bold indicate that *Hashing*-GPU attained the fastest execution time for that particular graph.

10 of which are from the Stanford Large Network Data Collection and 2 of which are from the DIMACS Challenge collection. As can be seen in Table 3, we attain significantly better total runtimes for 3 of the graphs. Each of these graphs have high values of Max_{ratio}, corroborating the findings from the CPU experiment of Table 2. For the remaining 9 graphs, one completed in a very slow runtime (loc-Gowalla) and 8 exceeded available shared memory. We do not report the graphs that failed to finish processing due to insufficient memory; each of these graphs have low values of Max_{ratio}. The loc-Gowalla graph just fits within available device memory, but possesses a low Max_{ratio} (see Table 1), leading to the significantly slower runtime than *Svendsen*.

6.2 Phase 2: GPU

Next, we demonstrate and assess the portable performance of *Hashing* by running it on a GPU architecture, using the graphs from Table 4. Each GPU time is compared to both the *Hashing* CPU time and the best time between the *Tomita* and *Eppstein* algorithms. From Table 4 we observe that, for 8 of the 9 graphs, *Hashing* GPU achieves a speedup over the CPU. Further, for 5 of these 8 graphs, *Hashing* GPU performs better than both *Hashing* CPU and *Tomita*/*Eppstein*. These speedups demonstrate the ability of a GPU architecture to utilize the highly-parallel design of our algorithm, which consists of many fine-grained and compute-heavy data-parallel operations. Moreover, this experiment demonstrates the portable performance of our algorithm, as we achieved improved execution times without having to write custom, optimized GPU functions within our algorithm; the same high-level algorithm was used for both the CPU and GPU experiments.

7 CONCLUSIONS AND FUTURE WORK

We have described a data-parallel primitive (DPP)-based algorithm for maximal cliques that has shown good performance on both CPUs and GPUs. The algorithm performs well on graphs with a large ratio of maximal cliques to total cliques, and outperformed leading implementations on standard data sets. Overall, the contribution of the paper is not only a new algorithm for maximal clique enumeration, but also significant evidence that the DPP approach can work well for graphs.

In terms of future work, the memory requirements for our approach prevented us from considering certain graphs, especially in a GPU setting. We would like to reduce overall memory usage and also explore the usage of host memory when running on the GPU, such as by leveraging the out-of-core MCE techniques presented in Cheng et al. [9]. We also hope to improve the algorithm to work better on denser graphs with low ratios of maximal cliques to total cliques, via pruning strategies for cliques that will eventually be covered by larger non-maximal cliques. Finally, when our algorithm cannot be further improved, we would like to formalize the conditions for which it outperforms the current leading algorithms.

APPENDIX

The following lemmas and theorem relate to properties upon which our hashing and sorting-based algorithms are based.

Lemma 7.1. *For $k \geq 2$, a $(k+1)$-clique is comprised of two k-cliques that both share $(k-1)$ vertices.*

Proof. Please refer to [21]. Figure 4 demonstrates this property by creating a 4-clique, 0-1-3-4, from two 3-cliques, 0-3-4 and 1-3-4, both of which share the 2-clique 3-4. Effectively, once two k-cliques with matching (and trailing) $(k-1)$ vertices are found, we only need to test whether the leading vertices are connected; if so, the two k-cliques can be merged into a new $(k+1)$-clique. □

Lemma 7.2. *An expanded $(k+1)$-clique maintains the ascending vertex order.*

Proof. In our algorithm, a k-clique is only matched with another k-clique that that a higher leading vertex Id. Both of the leading vertices of these two k-cliques have lower Ids and are distinct from the vertices of the matching $(k-1)$-clique. By induction, this $(k-1)$-clique must also be in ascending vertex order. Thus, the expanded $(k+1)$-clique must possess an ascending vertex Id order. □

Theorem 7.3. *During iteration k, there are no duplicate k-cliques.*

Proof. We will prove by induction on the size k.

- Base case of $k = 2$. The starting set of 2-cliques are the edges from the *v-graph* edge list, all of which are unique, since duplicate edges were removed in the initialization routine.
- Induction hypothesis. There are no duplicate k-cliques in iteration k.
- Inductive step. No duplicate $(k+1)$-cliques exist in iteration $k+1$. In the previous iteration k, a $(k+1)$-clique was produced via a merging between two k-cliques that shared a trailing $(k-1)$-clique (see Theorem 7.1). If duplicate copies of this $(k+1)$-clique existed, then duplicate pairs of the k-cliques must have also existed, since

any two k-cliques can only produce one new $(k+1)$-clique in our algorithm (see proof of Theorem 7.2). However, by the induction hypothesis, there are no duplicate k-cliques. Thus, by contradiction, a $(k+1)$-clique cannot have duplicates.

□

References

[1] DIMACS - The Maximum Cliques Problem. `http://iridia.ulb.ac.be/~fmascia/maximum_clique/DIMACS-benchmark`, June 2017.

[2] Stanford Large Network Dataset Collection. `https://snap.stanford.edu/data/`, June 2017.

[3] The FNV Non-Cryptographic Hash Algorithm. `https://tools.ietf.org/html/draft-eastlake-fnv-13`, June 2017.

[4] VTK-m. `https://gitlab.kitware.com/vtk/vtk-m`, June 2017.

[5] E. A. Akkoyunlu. The enumeration of maximal cliques of large graphs. *SIAM Journal on Computing*, 2(1):1–6, 1973.

[6] G. E. Blelloch. *Vector models for data-parallel computing*, vol. 75. MIT press Cambridge, 1990.

[7] C. Bron and J. Kerbosch. Algorithm 457: Finding all cliques of an undirected graph. *Commun. ACM*, 16(9):575–577, Sept. 1973.

[8] H. A. Carr, G. H. Weber, C. M. Sewell, and J. P. Ahrens. Parallel peak pruning for scalable SMP contour tree computation. In *6th IEEE Symposium on Large Data Analysis and Visualization, LDAV 2016, Baltimore, MD, USA, October 23-28, 2016*, pp. 75–84, 2016.

[9] J. Cheng, L. Zhu, Y. Ke, and S. Chu. Fast algorithms for maximal clique enumeration with limited memory. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, pp. 1240–1248. ACM, New York, NY, USA, 2012.

[10] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. Comput.*, 14(1):210–223, Feb. 1985.

[11] N. S. Dasari, D. Ranjan, and Z. Mohammad. Maximal clique enumeration for large graphs on hadoop framework. In *Proceedings of the First Workshop on Parallel Programming for Analytics Applications*, PPAA '14, pp. 21–30. ACM, New York, NY, USA, 2014.

[12] N. Du, B. Wu, L. Xu, B. Wang, and X. Pei. A parallel algorithm for enumerating all maximal cliques in complex network. In *Sixth IEEE International Conference on Data Mining - Workshops (ICDMW'06)*, pp. 320–324, Dec 2006.

[13] D. Eppstein, M. Löffler, and D. Strash. *Listing All Maximal Cliques in Sparse Graphs in Near-Optimal Time*, pp. 403–414. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[14] D. Eppstein and D. Strash. *Listing All Maximal Cliques in Large Sparse Real-World Graphs*, pp. 364–375. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[15] C. Harrison, H. Childs, and K. P. Gaither. Data-parallel mesh connected components labeling and analysis. In *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization*, EGPGV '11, pp. 131–140. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2011.

[16] H. C. Johnston. Cliques of a graph-variations on the bron-kerbosch algorithm. *International Journal of Computer & Information Sciences*, 5(3):209–238, 1976.

[17] J. Kalojanov, M. Billeter, and P. Slusallek. Two-level grids for ray tracing on gpus. *Computer Graphics Forum*, 30(2):307–314, 2011.

[18] J. Kalojanov and P. Slusallek. A parallel algorithm for construction of uniform grids. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pp. 23–28. ACM, New York, NY, USA, 2009.

[19] T. Karras. Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees. In C. Dachsbacher, J. Munkberg, and J. Pantaleoni, eds., *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics*, pp. 33–37. The Eurographics Association, 2012.

[20] I. Koch. Enumerating all connected maximal common subgraphs in two graphs. *Theoretical Computer Science*, 250(1):1 – 30, 2001.

[21] F. Kose, W. Weckwerth, T. Linke, and O. Fiehn. Visualizing plant metabolomic correlation networks using cliquemetabolite matrices. *Bioinformatics*, 17(12):1198–1208, 2001.

[22] A. Lagae and P. Dutré. Compact, fast and robust grids for ray tracing. In *ACM SIGGRAPH 2008 Talks*, SIGGRAPH '08, pp. 20:1–20:1. ACM, New York, NY, USA, 2008.

[23] 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)*, pp. 53–62. Cagliari, Italy, May 2015.

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

[25] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast bvh construction on gpus. *Computer Graphics Forum*, 28(2):375–384, 2009.

[26] E. L. Lawler, J. K. Lenstra, and A. H. G. R. Kan. Generating all maximal independent sets: Np-hardness and polynomial-time algorithms. *SIAM Journal on Computing*, 9(3):558–565, 1980.

[27] B. Lessley, R. Binyahib, R. Maynard, and H. Childs. External Facelist Calculation with Data-Parallel Primitives. In *Proceedings of EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV)*, pp. 10–20. Groningen, The Netherlands, June 2016.

[28] S. Li, N. Marsaglia, V. Chen, C. Sewell, J. Clyne, and H. Childs. Achieving Portable Performance For Wavelet Compression Using Data Parallel Primitives. In *Proceedings of EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV)*, pp. 73–81. Barcelona, Spain, June 2017.

[29] L.-t. Lo, C. Sewell, and J. P. Ahrens. Piston: A portable cross-platform framework for data-parallel visualization operators. In *EGPGV*, pp. 11–20, 2012.

[30] L. Lu, Y. Gu, and R. Grossman. dmaximalcliques: A distributed algorithm for enumerating all maximal cliques and maximal clique distribution. In *2010 IEEE International Conference on Data Mining Workshops*, pp. 1320–1327, Dec 2010.

[31] K. Makino and T. Uno. New algorithms for enumerating all maximal cliques. pp. 260–272. Springer-Verlag, 2004.

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

[33] J. S. Meredith, S. Ahern, D. Pugmire, and R. Sisneros. EAVL: The Extreme-scale Analysis and Visualization Library. In H. Childs, T. Kuhlen, and F. Marton, eds., *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2012.

[34] J. W. Moon and L. Moser. On cliques in graphs. *Israel Journal of Mathematics*, 3(1):23–28, 1965.

[35] K. Moreland, U. Ayachit, B. Geveci, and K. L. Ma. Dax toolkit: A proposed framework for data analysis and visualization at extreme scale. In *2011 IEEE Symposium on Large Data Analysis and Visualization*, pp. 97–104, Oct 2011.

[36] K. Moreland, C. Sewell, W. Usher, L. Lo, J. Meredith, D. Pugmire, J. Kress, H. Schroots, K.-L. Ma, H. Childs, M. Larsen, C.-M. Chen, R. Maynard, and B. Geveci. VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures. *IEEE Computer Graphics and Applications (CG&A)*, 36(3):48–58, May/June 2016.

[37] T. Perciano, D. Ushizima, H. Krishnan, D. Parkinson, N. Larson, D. Pelt, W. Bethel, F. Zok, and J. Sethian. Insight into 3d micro-ct data: exploring segmentation algorithms through performance metrics. *Journal of Synchrotron Radiation*, 24(5), Sept 2017.

[38] T. Perciano, D. M. Ushizima, E. W. Bethel, Y. D. Mizrahi, D. Parkinson, and J. A. Sethian. Reduced-complexity image segmentation under parallel markov random field formulation

using graph partitioning. In *2016 IEEE International Conference on Image Processing (ICIP)*, pp. 1259–1263, Sept 2016.

[39] M. C. Schmidt, N. F. Samatova, K. Thomas, and B.-H. Park. A scalable, parallel algorithm for maximal clique enumeration. *Journal of Parallel and Distributed Computing*, 69(4):417–428, 2009.

[40] H. A. Schroots and K.-L. Ma. Volume Rendering with Data Parallel Visualization Frameworks for Emerging High Performance Computing Architectures. In *SIGGRAPH Asia 2015 Visualization in High Performance Computing*, SA '15, pp. 3:1–3:4. ACM, 2015.

[41] M. Svendsen, A. P. Mukherjee, and S. Tirthapura. Mining maximal cliques from a large graph using mapreduce: Tackling highly uneven subproblem sizes. *Journal of Parallel and Distributed Computing*, 79:104–114, 2015.

[42] E. Tomita, A. Tanaka, and H. Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theor. Comput. Sci.*, 363(1):28–42, Oct. 2006.

[43] S. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM Journal on Computing*, 6(3):505–517, 1977.

[44] W. Widanagamaachchi, P. T. Bremer, C. Sewell, L. T. Lo, J. Ahrens, and V. Pascuccik. Data-parallel halo finding with variable linking lengths. In *2014 IEEE 4th Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 27–34, Nov 2014.

[45] B. Wu, S. Yang, H. Zhao, and B. Wang. A distributed algorithm to enumerate all maximal cliques in mapreduce. In *Proceedings of the 2009 Fourth International Conference on Frontier of Computer Science and Technology*, FCST '09, pp. 45–51. IEEE Computer Society, Washington, DC, USA, 2009.

[46] Y. Zhang, F. N. Abu-Khzam, N. E. Baldwin, E. J. Chesler, M. A. Langston, and N. F. Samatova. Genome-scale computational approaches to memory-intensive applications in systems biology. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, pp. 12–12. IEEE Computer Society, Washington, DC, USA, 2005.

[47] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time kd-tree construction on graphics hardware. In *ACM SIGGRAPH Asia 2008 Papers*, SIGGRAPH Asia '08, pp. 126:1–126:11. ACM, New York, NY, USA, 2008.