Lawrence Berkeley National Laboratory

Recent Work

Title
Maximal Clique Enumeration with Data Parallel Primitives

Permalink
https://escholarship.org/uc/item/4rj928sn

Authors
Lessley, B
Perciano, T
Mathai, M
et al.

Publication Date
2017-10-01

Peer reviewed
Maximal Clique Enumeration with Data-Parallel Primitives

Brenton Lessley
University of Oregon

Talita Perciano
Lawrence Berkeley Nat’l Lab

Manish Mathai
Lawrence Berkeley Nat’l Lab

Hank Childs
University of Oregon

E. Wes Bethel
University of Oregon

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

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 customiz-
including 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 [22], Larsen et al. for ray-tracing [23] and unstructured volume rendering [24], Schroots and Ma for cell-projected volume rendering [25], Lessley et al. for external facelist calculation [26], Lo et al. for isosurface generation [27], Widenagamaachchi et al. and Harrison et al. for connected component finding [28, 29], Carr et al. for contour tree computation [30], and Li et al. for wavelet compression [31]. 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, 30, 32, 33].

Finally, our experiments make use of the VTK-m framework [34], 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 [36], and PISTON [37]—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 expansion 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 [2] attained more prominence due to its simplicity and effective performance for most practical graphs. The algorithms proposed in [16, 21, 38, 42, 11, 33] build upon [2] and devise similar depth-first, tree-based search algorithms. Tomita et al. [34] optimize the clique expansion (pivoting) strategy of [2] to prune unnecessary sub-trees of the search forest, make fewer recursive calls, and demonstrate very fast execution times in practice, as compared to [2, 37, 10, 31]. Eppstein et al. [15, 37] develop a variant of [2] 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 [32] is used to recurse on candidate vertices. [13] also introduces two variants of their algorithm, and propose a memory-efficient version of [12] using adjacency lists. Experimental results indicate that [13] is highly competitive with the memory-optimized [32] 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. [22] 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 [2] 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 sub-trees dynamically reassigned to underutilized nodes. Lu et al. [40] and Wu et al. [41] 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. [42] expand the work of [13] 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. [43] 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 [35].

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. [44] and Zhang et al. [45] 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 [46, 47].

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. [13] and Svendsen et al. [43]. 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 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 executes 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 [8], 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.

\[
\begin{align*}
0 & 1 & 1 & 2 & 2 & 3 \\
1 & 3 & 4 & 2 & 3 & 4 & 4 & 5 & 4
\end{align*}
\]

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

\[
\begin{align*}
0 & 0 & 0 & 1 & 1 & 1 & 2 & 2 & 3 \\
1 & 3 & 4 & 2 & 3 & 4 & 4 & 5 & 4
\end{align*}
\]
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, \( N_{\text{edges}} \) is the number of output edges, and \( N_{\text{verts}} \) is the number of output vertices. See Section 4.3 for details.

```plaintext
/*Input*/
Array: int edgesIn[M]
/*Output*/
Array: int C[Nverts], E[Nedges], I[Nverts], V[Nverts]
/*Local Objects*/
Array: int edgesOrdered[M], edgesSorted[M], edgesUndirected[Nedges], A[Nedges]
Int: Nedges, Nverts

edgesOrdered ← Reorder(edgesIn)
edgesSorted ← Sort(edgesOrdered)
edgesUndirected ← Unique(edgesSorted)
A, E ← Unzip(edgesUndirected)
Nedges ← |E|
C, I ← ReduceByKey(A, I)
Nverts ← |I|
V ← ExclusiveScan(C)
//Continue with Algorithm 2 after returning.
return (C, E, I, V)
```

- Operation counts the number of adjacent edges from \( A \) that are associated with each unique vertex in \( E \). The resulting output arrays represent the lists \( I \) and \( C \) in our \( v \)-graph structure:
  
  \[
  I : [0 \ 1 \ 2 \ 3] \\
  C : [3 \ 3 \ 2 \ 1]
  \]

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] \rightarrow \text{ExclusiveScan} \rightarrow 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.

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.

```
/*Input from Algorithm 1*/
v-graph: int C[|V|], E[|E|], I[|V|], V[|V|]
/*Output*/
Array: int[|maximal| * |maximalVertices|]: maxCliques
/*Local Objects*/
int: Ncliques, Nnew Cliques, Nchains, Nmaximal, NmaxVertices, iter
Array: int[Ncliques]: cliqueStarts, cliqueSizes, cliqueIds, sortedCliqueIds, newCliqueCounts, scanNew, writeLocations
Array: float[Ncliques]: hashes, sortedHashes
Array: int[Nnew Cliques]: newCliqueStarts
Array: int[Ncliques] × (iter + 1): new Cliques
Array: int[Ncliques] × iter: cliques, isMaximal, hashTable
Array: int[Nnew Cliques] × (iter – 1): repCliqueIds, repCliqueStarts, localIndices, vertToOmit
Array: int[Nchains]: uniqueHashes, chainStarts, chainSizes, scanChainSizes
iter ← 2
claques ← Get2-Cliques(E)
Ncliques ← Nedges
```

fer to Algorithm 2 for pseudocode of these phases and operations. In the following description, the 3-cliques from Algorithm 1 are used as a running example. We start from iteration 3 with the linear array

\( \text{cliques} = [0-1-4 \ 0-3-4 \ 1-2-4 \ 1-3-4 \ 0-1-3] \)

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 vertices \(0\) and \(1\) because we just need to search \(0\) and \(1\).

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

\[
\text{hashTable} = \begin{bmatrix}
0-3-4 & 1-3-4 & 0-1-3 & 1-2-4 \\
\end{bmatrix}
\]

\[
\text{tableIndex} = \begin{bmatrix}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
\end{bmatrix}
\]

\[
\text{cliqueIds} = \begin{bmatrix}
0 & 1 & 2 & 3 & 4 \\
\end{bmatrix}
\]

\[
\text{chainIds} = \begin{bmatrix}
0 & 0 & 1 & 1 \\
\end{bmatrix}
\]

\[
\text{cliqueStarts} = \begin{bmatrix}
0 & 1 & 2 & 3 & 4 \\
\end{bmatrix}
\]

\[
\text{chainStarts} = \begin{bmatrix}
0 & 0 & 6 & 6 & 12 \\
\end{bmatrix}
\]

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 \( h(1-3) = h(1-4) = 1 \), creating a collision in Chain1. 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 and Theorem 7.2). 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 3 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, numNewClique (line 30 of algorithm 2). The second step works as follows. This sum is used to allocate a new array, cliques, of size numNewClique \((k + 1)\) 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 newClique 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 numClique, 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 numNewClique \(\times (k - 1)\), with \((k - 1)\) local indices per new \((k + 1)\)-clique: \([z_0 \ldots z_0 \ldots z_{\text{numClique}-1} \ldots z_{\text{numClique}-1}]\). Then, we parallelize over this index array via a Map opera-

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 [32], Eppstein [13], and Svendsen [31]. 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 [8], 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 function 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 [8], 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.
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. 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 (Maxsize), number of maximal cliques (Cliques\text{max}), number of total cliques (Cliques\text{all}), and ratio of maximal cliques to total cliques (Maxratio). 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 1. For each graph in Table 1, 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. [13], 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 [13] 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.

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, Maxratio. 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. [33], which we refer to as Svendsen. We use the set of 12 test graphs from Table 1.

---

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

<table>
<thead>
<tr>
<th>Graph</th>
<th>Collection</th>
<th>V</th>
<th>E</th>
<th>Maxsize</th>
<th>Cliques\text{max}</th>
<th>Cliques\text{all}</th>
<th>Maxratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>amazon0601</td>
<td>Stanford</td>
<td>403</td>
<td>3,878</td>
<td>11</td>
<td>1,023,572</td>
<td>18,043,744</td>
<td>0.06</td>
</tr>
<tr>
<td>cit-Patents</td>
<td>Stanford</td>
<td>3,774,768</td>
<td>16,518,947</td>
<td>11</td>
<td>14,787,032</td>
<td>36,180,638</td>
<td>0.41</td>
</tr>
<tr>
<td>email-Enron</td>
<td>Stanford</td>
<td>36,692</td>
<td>183,831</td>
<td>20</td>
<td>226,859</td>
<td>107,218,609</td>
<td>&lt; 0.01</td>
</tr>
<tr>
<td>loc-Gowalla</td>
<td>Stanford</td>
<td>196,591</td>
<td>950,327</td>
<td>29</td>
<td>1,212,679</td>
<td>1,732,143,035</td>
<td>&lt; 0.01</td>
</tr>
<tr>
<td>soc-wiki-Vote</td>
<td>Stanford</td>
<td>7,115</td>
<td>103,689</td>
<td>17</td>
<td>459,002</td>
<td>41,792,503</td>
<td>0.01</td>
</tr>
<tr>
<td>roadNet-CA</td>
<td>Stanford</td>
<td>1,965,206</td>
<td>2,766,607</td>
<td>4</td>
<td>2,537,996</td>
<td>2,887,325</td>
<td>0.88</td>
</tr>
<tr>
<td>brock200-2</td>
<td>DIMACS</td>
<td>200</td>
<td>9,876</td>
<td>12</td>
<td>431,586</td>
<td>6,292,399</td>
<td>0.07</td>
</tr>
<tr>
<td>hammerings-4</td>
<td>DIMACS</td>
<td>64</td>
<td>704</td>
<td>4</td>
<td>464</td>
<td>1,904</td>
<td>0.24</td>
</tr>
<tr>
<td>MANNa9</td>
<td>DIMACS</td>
<td>45</td>
<td>918</td>
<td>16</td>
<td>590,887</td>
<td>160,252,675</td>
<td>&lt; 0.01</td>
</tr>
<tr>
<td>p\text{hat}300-1</td>
<td>DIMACS</td>
<td>300</td>
<td>10,933</td>
<td>8</td>
<td>58,176</td>
<td>367,022</td>
<td>0.16</td>
</tr>
<tr>
<td>UG100k.003</td>
<td>DIMACS</td>
<td>100,000</td>
<td>14,997,901</td>
<td>4</td>
<td>10,589,956</td>
<td>19,506,096</td>
<td>0.54</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Graph</th>
<th>Tomita</th>
<th>Eppstein</th>
<th>Hashing</th>
</tr>
</thead>
<tbody>
<tr>
<td>amazon0601</td>
<td>**</td>
<td>3.59</td>
<td>1.69</td>
</tr>
<tr>
<td>cit-Patents</td>
<td>**</td>
<td>28.56</td>
<td>3.27</td>
</tr>
<tr>
<td>email-EuAll</td>
<td>**</td>
<td>1.25</td>
<td>2.24</td>
</tr>
<tr>
<td>email-Enron</td>
<td>31.96</td>
<td>0.90</td>
<td>17.91</td>
</tr>
<tr>
<td>roadNet-CA</td>
<td>**</td>
<td>2.00</td>
<td>0.27</td>
</tr>
<tr>
<td>roadNet-PA</td>
<td>**</td>
<td>1.09</td>
<td>0.16</td>
</tr>
<tr>
<td>roadNet-TX</td>
<td>**</td>
<td>1.35</td>
<td>0.19</td>
</tr>
<tr>
<td>brock200-2</td>
<td>0.55</td>
<td>1.22</td>
<td>0.71</td>
</tr>
<tr>
<td>hammerings-4</td>
<td>&lt; 0.01</td>
<td>&lt; 0.01</td>
<td>&lt; 0.01</td>
</tr>
<tr>
<td>johnson8-4-4</td>
<td>0.13</td>
<td>0.24</td>
<td>0.50</td>
</tr>
<tr>
<td>johnson16-2-4</td>
<td>5.97</td>
<td>12.17</td>
<td>5.10</td>
</tr>
<tr>
<td>MANNa9</td>
<td>0.44</td>
<td>0.55</td>
<td>27.74</td>
</tr>
<tr>
<td>p\text{hat}300-1</td>
<td>0.07</td>
<td>0.15</td>
<td>0.07</td>
</tr>
<tr>
<td>soc-wiki-Vote</td>
<td>0.96</td>
<td>1.14</td>
<td>6.14</td>
</tr>
<tr>
<td>keller4</td>
<td>5.98</td>
<td>11.53</td>
<td>7.22</td>
</tr>
</tbody>
</table>
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.

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

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.

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

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. [3]. 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 theorems 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 [4]. 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


