Scalable Maximum Clique Computation Using MapReduce

Jingen Xiang, Cong Guo, Ashraf Aboulnaga

Cheriton School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada
{jxiang, c8guo, ashraf}@uwaterloo.ca

***Abstract*— We present a scalable and fault-tolerant solution for the maximum clique problem based on the MapReduce framework. The key contribution that enables us to effectively use MapReduce is a recursive partitioning method that partitions the graph into several subgraphs of similar size. After partitioning, the maximum cliques of the different partitions can be computed independently, and the computation is sped up using a branch and bound method. Our experiments show that our approach leads to good scalability, which is unachievable by other partitioning methods since they result in partitions of different sizes and hence lead to load imbalance. Our method is more scalable than an MPI algorithm, and is simpler and more fault tolerant.**

I. Introduction

Systems like MapReduce [6] and Pregel [23], which enable scalable computation on massive computing clusters (i.e., computing clouds), are now an essential part of the infrastructure for data analytics. These systems provide a simple and powerful programming model, automatic scalability and fault tolerance, and are usually part of a rich ecosystem of supporting tools and technologies (e.g., the Hadoop ecosystem includes HDFS, HBase, Hive, Pig, Mahout, and other tools). The popularity and power of these systems make it interesting to study whether we can use them to implement various parallel algorithms.

In this paper, we study the use of MapReduce to solve the *maximum clique* problem [21], [26]. A clique is a complete graph, that is, a graph where any two vertices are connected by an edge. The maximum clique within a graph G is the largest size subgraph of G that is a clique. We focus on the maximum clique problem because it is a computationally hard problem with practical applications. We present a novel graph partitioning algorithm that we call *Balanced Multi-depth Color-based partitioning (BMC partitioning)*. BMC partitioning enables scalable computation of maximum clique on MapReduce using a branch and bound technique.

We motivate our work by presenting some applications of the maximum clique problem. A key motivating application for us comes from the area of coding theory, and in particular quantum error correcting codes. A code $\mathcal{C}$ of length n is a subset of all n-bit strings. The Hamming distance $d(\mathbf{x}, \mathbf{y})$ between two n-bit strings $\mathbf{x}$ and $\mathbf{y}$ is the number of positions in which $\mathbf{x}$ and $\mathbf{y}$ differ. The (minimum) distance d of a code $\mathcal{C}$ is the smallest distance between distinct codewords, which is proportional to the number of errors that the code can correct. For a given n and d, finding the largest possible code cardinality K (i.e., the number of codewords in the code) is then a maximum clique problem [18], [22]. The corresponding graph, called the coding graph, contains 2^n vertices where each vertex corresponds to an n-bit string. The edges are given by the pairs of strings $(\mathbf{x}, \mathbf{y})$ with Hamming distance greater than or equal to d. The vertices of the maximum clique of this graph represent the codewords, and the number of vertices of the maximum clique is the cardinality K. The case for quantum error correcting codes is more complex, but for a large class of codes, the problem can be reduced to a maximum clique problem similar to the classical case by modifying the definition of distance d using a framework known as *codeword stabilized (CWS) quantum codes* [4], [5]. In our experiments, we show orders of magnitude improvement in the time required to compute the maximum clique for a real coding graph from a quantum error correcting code.

Another application of the maximum clique problem is the study of Internet topology. The Internet consists of several independent *autonomous systems (ASes)*. An AS is a set of routers within a single administrative domain (e.g., an ISP, a company, or a university). An AS can be related to another AS in one of three ways: *provider-customer*, where one AS provides transit service to the other AS (e.g., an ISP providing service to its customer, who may be a smaller ISP), *peering*, where the two ASes exchange traffic to their customers for free, and *sibling*, where the two ASes exchange traffic freely without any limitation. Knowing how ASes are related to each other is useful for many network research questions such as predicting communication latency, selecting peering partners, and defending against attacks known as BGP prefix hijacking. However, AS relationship information is not public, so it must be inferred from information about the routing paths taken by different packets. Each network packet (BGP packet to be more precise) that arrives at a destination contains information about its *AS path*, the set of ASes that it traversed to get from its source to its destination. An assumption we can make about valid AS paths is that they follow the *valley-free model*. This model states that after traversing a provider-to-customer or peer-to-peer link, an AS path cannot traverse a customer-to-provider or peer-to-peer link [11]. According to the valley-free model, there is at most one peer-to-peer link in an AS path. If we are given a set of AS paths that are assumed to follow the valley-free model, we can find the maximum set of peer-to-peer links by solving the following maximum clique problem. Construct a graph with a vertex for every AS-to-AS link that appears in any AS path. If two AS-to-AS links never appear

in the same AS path, add an edge between their corresponding vertices. Since a valid AS path contains at most one peer-to-peer link, we can assume that the maximum clique in this graph is the maximum set of peer-to-peer links [9].

In biochemistry, maximum clique is used to find the largest possible pharmacophore (a group of atoms responsible for an interaction) for a pair of 3D molecules. Each of the two molecules is represented by a graph with atoms as vertices. A correspondence graph G is constructed for the two graphs of the molecules G_1 and G_2, such that G has a vertex for each pair of vertices with one from G_1 and one from G_2. Two correspondence graph vertices $\{G_1(X), G_2(M)\}$ and $\{G_1(Y), G_2(N)\}$ are adjacent in G if there is an edge from $G_1(X)$ to $G_1(Y)$ in G_1 and an edge from $G_2(M)$ to $G_2(N)$ in G_2. Matching 3D molecular structures is equivalent to the maximum clique problem in G [12].

Having motivated the importance of the maximum clique problem, we now present an overview of our solution. The main idea of our solution is to recursively partition the graph into smaller possibly overlapping subgraphs so that each compute node in the MapReduce cluster can independently compute the maximum clique for its partition. When two vertices connected by an edge are placed in two different partitions, one of the vertices has to be replicated in the other's partition so that the edge is preserved (hence the overlapping partitions). Graph partitioning has been studied in previous work, but partitioning for the maximum clique problem is challenging for two reasons: (1) Since any edge can be part of the maximum clique, there is no notion of a workload that dictates "important" edges that should not be split across partitions. Some partitioning methods require such a workload-driven measure of edge importance [35]. (2) Maximum clique is of interest on *dense graphs*, where a significant fraction of the edges that can be present in a graph are actually present. Most general purpose graph partitioning algorithms do not work well for dense graphs, since they end up replicating a large fraction of the vertices (as demonstrated in Section VIII).

We develop a graph partitioning algorithm that addresses these two challenges and is targeted towards computing maximum cliques. Our partitioning algorithm removes one vertex at a time from the graph and puts the subgraph consisting of this vertex plus all its neighbors in one partition. By repeatedly partitioning the graph, we end up with multiple subgraphs on which the maximum clique can be computed independently. The maximum clique of the original graph is the largest of the maximum cliques computed on the partitions.

Partitioning alone does not result in a scalable maximum clique algorithm, since we still need to compute the maximum clique for all partitions. To avoid this brute-force effort, we use a *branch and bound* approach to avoid computing the maximum clique of a partition if it can be shown that its maximum clique will be smaller than the largest clique found so far. For this pruning to be effective, we need to find large cliques as early as possible. At the same time, we need the partitions to be of similar size for load balancing. Our Balanced Multi-depth Color-based (BMC) partitioning achieves these objectives. It uses graph coloring to compute an upper bound on the clique size and to heuristically order the partitions so that pruning can be effective. It partitions the graph recursively to multiple depths to achieve load balance.

After the graph is partitioned, the maximum cliques for the different partitions can be computed independently on different nodes of a computing cluster, with some pruning based on the size of the largest clique found so far. In this paper, we implement this parallel branch and bound computation using MapReduce, and in particular Hadoop. It is important to note that this approach to computing maximum cliques can be implemented using other cluster-based computing platforms, such as Pregel or even traditional Message Passing Interface (MPI) [15]. We use Hadoop because of its simple programming model, scalability, and fault tolerance, characteristics that are not available with, say, MPI. Moreover, Hadoop is popular and widely available, so a Hadoop-based maximum clique implementation would be easy for users to adopt and deploy on private clouds or public clouds such as Amazon EC2 [10]. Furthermore, such an implementation would work seamlessly with other tools and technologies in the Hadoop ecosystem.

To illustrate the power of our solution, we consider the DIMCAS Maximum Clique Benchmark [7]. Maximum clique was part of the Second DIMACS Implementation Challenge [8], and the DIMCAS Maximum Clique Benchmark consists of a set of graphs for which finding the maximum clique is difficult. Despite being well studied, the maximum clique for some of these graphs was still not known. We ran our algorithm on one such graph with 4000 vertices (C4000.5) on 128 Amazon EC2 high CPU medium instances, and completely solved the maximum clique problem for this graph for the first time in 39 hours. We estimate that a serial solution on a single CPU would have taken more than 1 year. (More details about this experiment in Section VIII.)

Note that while this is the largest graph that we used in our experiments, it is still only a few tens of megabytes in size. The complexity of maximum clique stems not from the size of the data but rather from the CPU time required to compute the maximum clique on individual partitions. The key factor determining complexity is the density of the graph not its size. Thus, our work represents an interesting study in using MapReduce for CPU-intensive parallel computation, in contrast to the data intensive computation for which MapReduce is typically used.

The rest of this paper is organized as follows: in Section II, we present preliminary information about the maximum clique problem. Section III presents related work, and Section IV discusses single-node maximum clique algorithms. In Section V, we present different graph partitioning techniques and introduce our proposed BMC partitioning. Section VI describes our approach to running time estimation, which is required for BMC partitioning. The MapReduce implementation of our algorithm is discussed in Section VII. Section VIII presents an experimental evaluation and Section IX concludes.

II. Preliminaries and notation

In this paper, we focus on undirected unweighted graphs. An undirected graph G is represented by an ordered pair (V, E), where V is a finite set of vertices and E is a set of two-element subsets of vertices called edges. For a graph G, we write $V(G)$ and $E(G)$ for the vertex set and the edge set, respectively. The number of vertices of a graph, i.e., the cardinality of $V(G)$, denoted by $|V(G)|$ or $|G|$ for simplicity, is called the *order* of the graph. The number of edges of a graph, denoted by $|E(G)|$, is called the *size* of the graph.

Two vertices $a, b \in V(G)$ are called adjacent if $(a, b) \in E(G)$. A graph G is called *complete* if all its vertices are pairwise adjacent. For a set of vertices $S \subseteq V(G)$, $S(G) = (S, E(G) \cap (S \times S))$ is the subgraph induced by S. A clique C is a subset of V such that the induced subgraph $C(G)$ is complete. A k-clique of G is a clique in G of order k. A clique is called *maximal* if it cannot be extended by including one more adjacent vertex. In other words, a maximal clique does not exist exclusively within the vertex set of a larger clique. For a given graph G, a maximum clique, denoted as $MaxClique(G)$, is a clique of the largest possible order in G. The order of a maximum clique in G is denoted by $\omega(G)$.

We now turn our attention to graph partitioning. The neighborhood of vertex v in G, denoted by $N(G, v)$, is a subgraph of G consisting of the adjacent vertices to vertex v. That is, $V(N(G, v)) = \{w|(v, w) \in E(G)\}$ and $E(N(G, v)) = \{(v, w)|v, w \in V(N(G, v)),\ (v, w) \in E(G)\}$. The degree of a vertex v in G, denoted by $D(G, v)$, is the number of adjacent vertices of v in G, i.e., $D(G, v) = |N(G, v)|$, while the extension degree of a vertex v in G, denoted by $Ext\_D(G, v)$, is the sum of the degree of its adjacent vertices in G, i.e., $Ext\_D(G, v) = \sum_{w \in N(G,v)} D(G, w)$.

Consider a graph G which is a subgraph of G'. For any $v \in V(G') - V(G)$, the union $G \cup \{v\}$ is a subgraph of G' induced by adding a vertex v to $V(G)$, i.e. $V(G \cup \{v\}) = V(G) \cup \{v\}$ and $\forall (v, w) \in E(G)$, $(v, w) \in E(G \cup \{v\})$.

For two subgraphs of G, G_1 and G_2, where $V(G_1) \cap V(G_2) = \{\}$, the *inter-graph* of G_1 and G_2 is a subgraph of G, denoted by $I(G, G_1, G_2)$. The vertex set of $I(G, G_1, G_2)$, $V(I)$, is the set of all vertices $v \in V(G_1)$ and $w \in V(G_2)$ such that $(v, w) \in G$. The edge set of $I(G, G_1, G_2)$, $E(I)$, is given by $E(I) = \{(v, w)|v, w \in V(I), (v, w) \in E(G)\}$.

Our partitioning algorithm relies on graph coloring. A coloring of a graph G is a list, denoted by C_G, containing for each vertex a number denoting the color of that vertex. The color of a vertex v is denoted by $C_G[v]$.

Our partitioning also relies on graph density. The density of a graph G is defined as $\rho(G) = \frac{2|E(G)|}{|G|(|G|-1)}$. For any graph G, $0 \leq \rho(G) \leq 1$, and for a complete graph $\rho(G) = 1$. We summarize our notation in Table I.

Symbol	Definition
$G = (V, E)$	Graph with vertices V and edges E
$V(G)$	All vertices of G
$E(G)$	All edges of G
$\|G\|$ or $\|V(G)\|$	Number of vertices in G
$\|E(G)\|$	Number of edges in G
$G - \{v\}$	Subgraph of G obtained by removing vertex v
$G \cup \{v\}$	Adding the vertex $v \in G' - G$ to $G \subseteq G'$
$MaxClique(G)$	Maximum clique of G
$\omega(G)$	Number of vertices in the maximum clique of G, $\|MaxClique(G)\|$
$N(G, v)$	Neighborhood of vertex v in G, $V = \{w\|(v, w) \in E(G)\}$ $E = \{(v, w)\|v, w \in V,\ (v, w) \in E(G)\}$
$D(G, v)$	Degree of v in G, $\|N(G, v)\|$
$Ext\_D(G, v)$	Extension degree of v in G, $\sum_{w \in N(G,v)} D(G, w)$
$I(G, G_1, G_2)$	*Inter-graph* induced by subgraphs G_1, G_2 of G
C_G	Coloring of G (each vertex has a color number)
$C_G[v]$	Color of vertex v
ρ, or $\rho(G)$	Density of G, $\frac{2\|E(G)\|}{\|G\|(\|G\|-1)}$

TABLE I
Notation used in this paper.

III. Related Work

Ever since the term "clique" was introduced [21], the maximum clique problem has been investigated by many researchers. The k-clique decision problem asks whether there exists a k-clique in a given graph G, and is known to be NP-complete [13], [29]. Therefore, the maximum clique problem is NP-hard. If one can solve the maximum clique problem, one can also solve the k-clique decision problem by comparing the number of vertices in the maximum clique to k.

A common and effective method to find the maximum clique is to use branch and bound search. A classical algorithm proposed by Carraghan and Pardalos [3] is a straightforward heuristic and pruning algorithm. At every stage of the algorithm, there is a global control parameter ω, which is the largest known clique at this stage. For each subgraph G', the algorithm finds the set of vertices $\{w\}$ which are not in G' but are connected to all vertices in G'. In addition, any two vertices in $\{w\}$ are connected to each other. Denote by m the number of vertices in $\{w\}$. If $m + |G'| \leq \omega$, the subgraph G' is pruned. This algorithm is shown to be much more effective than a brute-force exhaustive search and is easy to parallelize.

Another maximum clique algorithm is proposed by Östergård [24]. That paper introduces an additional pruning strategy based on the following observation. Given a graph $G = (V, E)$ containing vertices $\{v_1, v_2, \ldots, v_{|V|}\}$, we can sort the vertices in some order and let subgraph $G_i = \{v_i, v_{i+1}, \ldots, v_{|V|}\}$. Observe that $\omega(G_i) = \omega(G_{i+1}) + 1$ if vertex v_i is included in the maximum clique of G_i, and $\omega(G_i) = \omega(G_{i+1})$ if vertex v_i is not included in the maximum clique of G_i. The algorithm uses a search strategy similar to [3], and adds a pruning rule based on the above observation.

An effective algorithm named MCR is proposed by Tomita and Kameda [30]. This algorithm uses graph coloring to obtain an upper bound on the size of the maximum clique, which achieves better branch pruning than previous algorithms. We use the MCR algorithm as our single-node maximum clique algorithm, and discuss it in more detail in the next section.

Wang and Cheng [33] recently suggested using the size of the maximum k-truss of a graph as an upper bound on the size

of the maximum clique, so that some subgraphs can be more effectively pruned. The k-truss of a graph G is the largest subgraph of G in which every edge is contained in at least $(k-2)$ triangles within the subgraph. Finding the k-truss of a graph can be solved in polynomial time.

If a graph could be represented as a combination of a random graph of order n and a large clique of order larger than $\sqrt{n}$ (a "planted clique"), the maximum clique can be found with high probability using spectral methods [2]. In this paper, we encounter maximum cliques whose order is smaller than $\sqrt{n}$, so spectral methods cannot be used even if the planted clique assumption holds.

There have been previous attempts to solve the maximum clique problem on computing clusters. A parallel maximum clique algorithm based on MPI has been proposed in [25]. This algorithm uses a master-worker architecture where the master uses graph partitioning to assign a separate task to each worker. Once a worker has finished the task, the master will assign another subgraph to it. The partitioning strategy used by this algorithm is a simple version of our one-depth partitioning strategy. This strategy balances the load among workers in many cases, but it is not sufficient when the density of the graph is large. For dense graphs, the first few subgraphs generated by the partitioning are likely to be larger than others and will therefore take much longer to finish. The time taken for one of these large subgraphs can be larger than the total time of the smaller subgraphs. We show in our experiments that this limits the scalability of the approach in [25], even if we use an improved single-node maximum clique algorithm. In addition, MPI is a complex programming model, does not provide fault tolerance, and is not as scalable as MapReduce.

MapReduce solutions for the maximum clique problem have been proposed before. Lin et al. [20] proposed a MapReduce algorithm for maximum clique, but their partitioning strategy is naive random partitioning, and they do not employ good pruning. At best, their algorithm would perform similar to our one-depth partitioning strategy, which is much less scalable than our multi-depth partitioning strategy. In practice, it would be difficult for their algorithm to match our one-depth performance since they employ a naive random partitioning strategy with no partition optimization at all and also no pruning optimization. Wu et al. [34] also proposed a MapReduce algorithm for the clique problem, but they focus on enumerating all maximal cliques instead of the maximum clique problem. Moreover, their algorithm shows poor scalability.

It may be possible to use a system dedicated to graph processing for solving the maximum clique problem. One such system is Pregel [23], and its open source implementation based on Hadoop called Giraph [14]. Pregel implements a computational model called *bulk synchronous parallel computation* [32], which is targeted towards iterative computation on graphs. Our partitioning algorithm results in subgraphs that are completely independent, so iteration on the graph vertices is not required. What is needed is independently computing maximum clique on the individual partitions, for which MapReduce is suitable.

IV. Maximum Clique Algorithm on a Single Node

Our approach partitions a graph into subgraphs, and can use any single-node algorithm to compute the maximum clique on one partition. We experimented with different single-node algorithms and found that the MCR algorithm [30] gives the best performance. Therefore, we use it as our single-node algorithm. Furthermore, the graph coloring that is used by MCR to order vertices is the basis for our color-based partitioning. Hence, we describe the MCR algorithm in some detail in this section.

The MCR algorithm is a branch and bound algorithm. Given a graph $G = (V, E)$, if $\omega(G)$ is the number of vertices in the maximum clique of G, then $\omega(G)$ can be obtained by independently computing the maximum cliques of different independent subgraphs as follows:

$$
\begin{aligned}
&\omega(N(G, v_1)) + 1,\\
&\omega(N(G - \{v_1\}, v_2)) + 1,\\
&\omega(N(G - \{v_1, v_2\}, v_3)) + 1,\\
&\ldots,\\
&\omega(N(G - \{v_1, v_2, \ldots, v_k\}, v_{k+1})) + 1,\\
&\ldots,\\
&\omega(N(G - \{v_1, v_2, \ldots, v_{|V|-2}\}, v_{|V|-1})) + 1
\end{aligned}
$$

To compute the maximum clique of $N(G, v_1)$, we can perform the same process on the subgraph $N(G, v_1)$. Thus, the maximum clique can be found in a recursive way. Without a pruning strategy, this algorithm is equivalent to brute-force exhaustive search. Therefore, it is important to prune the search based on bounds on the size of the maximum clique.

The MCR algorithm proposes a pruning strategy based on coloring the vertices of the graph. In this method, each vertex in the graph is given a color, and any two adjacent vertices cannot have the same color. Since no adjacent vertices can have the same color, a clique will always consist of vertices of different colors. Thus, the total number of colors used to color the graph is an upper bound on the size of the maximum clique. Using fewer colors results in a tighter upper bound, but the coloring procedure becomes more complex. At one extreme, we can use a different color for each vertex, in which case the coloring procedure is very simple but has no pruning power. At the other extreme, we can use the minimum possible number of colors, which increases the pruning power of coloring but makes the coloring procedure very complex.

A coloring procedure is given in [30] that strikes a balance between coloring complexity and pruning power. The coloring procedure starts by sorting the vertices of the graph in descending order according to their degree. The procedure for doing this is as follows: (1) Select a vertex v with the minimum degree from the graph G and append it to R (which is initially an empty list). If there are $k > 1$ vertices with the same degree (denote this set of vertices as $G' = \{v_1, v_2, \ldots, v_k\}$), select the vertex with the minimum extension degree in G'. (2) Remove v from G. When removing v, we also remove all edges incident on v, which changes the degrees of the

Algorithm 1 Simple coloring.

```
1: function Color(G)
2: G_1, G_2, ..., G_n = {}
3: i = 1
4: while i < |G| do
5:     j = 1
6:     while ∃v_k ∈ G_j, s.t., v_i ∈ N(G, v_k) do
7:         j = j + 1
8:     end while
9:     C_G(v_i) = j, append v_i to G_j
10:    i = i + 1
11: end while
12: Sort V(G) based on C_G in descending order
13: return C_G
```

Algorithm 2 MCR for single-node maximum clique.

```
1: function MaxClique(G)
2: while |G| > 0 do
3:     /* C_G is the color lis tof G */
4:     v = {p|C_G[p] = max(C_G)}
5:     if |Q| + C_G[v] > |Q_max| then
6:         Q = Q ∪ v
7:         G_1 = N(G, v)
8:         if |G_1| > 0 then
9:             MaxClique(G_1)
10:        else if |Q| > |Q_max| then
11:            Q_max = Q
12:        end if
13:        Q = Q − {v}
14:    end if
15:    G = G − {v}
16: end while
17: return Q_max
```

other vertices in G and hence may change the vertex with the minimum degree chosen in the next step. (3) Repeat Step (1) until there are no vertices in G. (4) Reverse the list R.

After sorting, the procedure shown in Algorithm 1 is applied. Each color is represented by a number, with 1 being the first color used. The last step in Algorithm 1 sorts the vertices by color in descending order. Sorting the vertices by degree before coloring and then sorting the colors in descending order is shown to improve the effectiveness of coloring. We return to this coloring algorithm in Section V-F when we discuss color-based partitioning.

The MCR algorithm processes vertices in descending order of color as shown in Algorithm 2. The variables Q_{max} and Q are global variables recording the maximum clique and the current maximal clique, respectively. The pruning based on the maximum color is in line 5 of the algorithm.

V. Partitioning Strategies

In order to compute the maximum clique in parallel on a computing cluster, we need to partition the graph into several smaller subgraphs and compute the maximum clique for these subgraphs independently on the compute nodes. Graph partitioning has been studied before (e.g., [1], [17], [19]). In this section, we present some possible partitioning strategies from the literature, and our proposed BMC partitioning. Our experiments show that BMC partitioning outperforms all other partitioning strategies presented in this section.

A. Bisection Partitioning

A straightforward partitioning strategy is bisection partitioning, which partitions the original graph G into two graphs G_A and G_B that have a similar number of vertices. In order to obtain the maximum clique of G, we need to compute the maximum cliques of G_A, G_B, and the inter-graph $I(G, G_A, G_B)$ (recall the definition of the inter-graph from Section II). Each of the three subgraphs G_A, G_B, and $I(G, G_A, G_B)$ can be further partitioned into three more subgraphs, and so on.

In order to reduce the number of vertices of the inter-graph, we need to minimize the number of edges connecting G_A and G_B. This converts the partitioning problem into the bisection partitioning problem, which is a well-known NP-hard problem. Some approximate bisection partitioning algorithms exist, such as the classical Kernighan and Lin algorithm [19] as well as more recent algorithms [27], [28]. However, in our experiments we found that bisection partitioning (using the Kernighan and Lin algorithm) is not suitable for the maximum clique problem because it results in large inter-graphs $I(G, G_A, G_B)$. In general, we found that the largest inter-graphs typically have the same size as the original graph, so we gain no reduction in complexity through bisection partitioning.

B. k-way Partitioning

Another possible partitioning strategy is k-way partitioning, which partitions the graph G into k subgraphs of similar size, denoted as $\{G_1, G_2, G_3, \ldots, G_k\}$, instead of two subgraphs as in bisection partitioning. A good partitioning also minimizes the number of edges running between separated subgraphs. There are many methods for k-way partitioning, such as spectral partitioning [16] and multi-level graph partitioning [17].

In order to compute the maximum clique of G, we need to separately compute the maximum clique of graphs $G_1, G_2, G_3, \ldots, G_k$, and the maximum clique of the inter-graphs that contain edges running between different subgraphs and the vertices connected by those edges. The inter-graphs can be obtained in various ways. The straightforward way is to compute the inter-graphs using multiple steps. The i-th step produces $k/2^i$ inter-graphs, each of which is computed from 2^i subgraphs. That is, in the first step, $I_{1,1} = I(G, G_1, G_2)$, …, $I_{1,k/2} = I(G, G_{k-1}, G_k)$, and in the i-th step, the m-th inter-graph is $I_{i,m} = I(G, \cup_{n=1}^{2^{i-1}} G_{(m-1)2^{i-1}+n}, \cup_{n=1}^{2^{i-1}} G_{m2^{i-1}+n})$. The last inter-graph is $I_{log(k),1} = I(G, G_1 \cup G_2 \cup \ldots \cup G_{k/2}, G_{k/2+1} \cup G_{k/2+2} \cup \ldots \cup G_k)$, which is very similar to the inter-graph obtained from bisection partitioning. As with bisection partitioning, such an inter-graph generally has the same size as the original graph G.

To avoid this problem, we use another method. Here we consider the inter-graphs one by one. That is, the i-th inter-graph is $I(G, G_1 \cup G_2 \cup \ldots \cup G_{i-1}, G_i)$ $(i = 2,\ 3, \ldots, n)$. Using this method, we can get smaller inter-graphs for graphs with low densities, but we still face the problem of large inter-graphs for graphs with high densities.

C. PICS Partitioning

A recently proposed partitioning strategy that is fundamentally different from bisection partitioning or k-way partitioning is PICS partitioning [1]. Instead of partitioning the graph into subgraphs with a similar number of vertices, PICS partitioning tries to find clusters in a given graph. If we consider these clusters as subgraphs, we can use PICS to partition graph G into several subgraphs (*subgroups* to use the PICS terminology) $G_1, G_2, \ldots, G_k$. In contrast to bisection partitioning and k-way partitioning, it is not necessary that these subgraphs have a similar number of vertices.

In order to compute the maximum clique of the original graph G, we need to separately compute the maximum cliques of those subgraphs and the inter-graphs, as described above. The inter-graphs can be obtained using the same method as in Section V-B. If we want to partition the subgraphs into smaller subgraphs, we can apply the PICS algorithm to those subgraphs. However, we find that this algorithm cannot guarantee to partition some subgraphs into two or more smaller subgraphs. Additionally, the algorithm suffers the same problem of large inter-graphs and is not suitable for the maximum clique problem.

D. One-Depth Partitioning

Motivated by the shortcomings of existing partitioning techniques, we present a simple partitioning that we call *one-depth partitioning*, which is similar to the partitioning used by the MCR algorithm in Section IV. Given a graph $G = (V, E)$, the subgraphs are obtained as follows:

$$\begin{aligned}
&G_1 = v_1 \cup N(G, v_1),\\
&G_2 = v_2 \cup N(G - \{v_1\}, v_2),\\
&G_3 = v_3 \cup N(G - \{v_1, v_2\}, v_3),\\
&\ldots,\\
&G_k = v_k \cup N(G - \{v_1, v_2, \ldots, v_{k-1}\}, v_k),\\
&\ldots,\\
&G_{|V|-1} = v_{|V|-1} \cup (G - \{v_1, v_2, \ldots, v_{|V|-2}\}, v_{|V|-1})
\end{aligned} \tag{1}$$

This method has been used previously in the parallel computation of maximum cliques based on MPI [25] and MapReduce [20]. However, this method does not work for dense graphs because it results in poor load balancing between the partitions, since some partitions are much larger than others.

E. Multi-Depth Partitioning

Due to the load balancing problem of one-depth partitioning, we propose a novel partitioning method suitable for parallel maximum clique computation using MapReduce. First, we partition the original graph into several smaller subgraphs as in Equation (1). Next, we estimate the running time of the maximum clique algorithm on each partition. Estimating the running time is based on a simple experiment-driven performance model described in Section VI. If the estimated running time on a partition is above a bound value, we further partition that subgraph into smaller subgraphs. The bound value in this paper is chosen through experiments to be 60 seconds (see Section VIII-B). We repeat this partitioning step until the estimated running time on each of the partitions is below the bound value.

If the estimated running time of any subgraph in G_1, G_2, $\ldots$, $G_{|V|-1}$ is larger than the bound value (let us assume it is G_k), the subgraph G_k will be partitioned again as follows:

$$\begin{aligned}
&G_{k,1} = \{v_k, w_1\} \cup N(G_k - \{v_k\}, w_1),\\
&G_{k,2} = \{v_k, w_2\} \cup N(G_k - \{v_k, w_1\}, w_2),\\
&G_{k,3} = \{v_k, w_3\} \cup N(G_k - \{v_k, w_1, w_2\}, w_3),\\
&\ldots,\\
&G_{k,l} = \{v_k, w_l\} \cup N(G_k - \{v_k, w_1, \ldots, w_{l-1}\}, w_l),\\
&\ldots,\\
&G_{k,m} = \{v_k, w_m\} \cup N(G_k - \{v_k, w_1, \ldots, w_{m-1}\}, w_m),
\end{aligned} \tag{2}$$

where m is a cutoff value such that the estimated running time of the remaining subgraph $G_k - \{v_k, w_1, w_2, \ldots, w_{m-1}\}$ is no larger than the bound value, while the estimated running time of $G_k - \{v_k, w_1, w_2, \ldots, w_{m-2}\}$ is larger than the bound value.

Again, if there is a subgraph in $G_{k,1}, G_{k,2}, \ldots, G_{k,m}$ such that its estimated running time is larger than the bound value (let us assume it is $G_{k,l}$), then $G_{k,l}$ needs to be partitioned again. The n-th partitioned subgraph from $G_{k,l}$ is defined as follows: $G_{k,l,n} = \{v_k, w_l, u_n\} \cup N(G_{k,l} - \{v_k, w_l, u_1, u_2, \ldots, u_{n-1}\}, u_n)$. At the end of the partitioning, the estimated running time of any subgraph is less than the bound value.

In our algorithm, the partitioning path is recorded. This path indicates how the subgraph is obtained from the original graph. For example, the subgraph $G_{k,l,n}$ is obtained by the partitioning algorithm via $v_k \rightarrow w_l \rightarrow u_n$. Therefore, the maximum clique of $G_{k,l,n}$ can be computed with the call MaxClique$(N(G_{k,l} - \{u_1, u_2, \ldots, u_{n-1}\}, u_n)) \cup \{v_k, w_l, u_n\}$ instead of MaxClique$(\{v_k, w_l, u_n\} \cup N(G_{k,l} - \{u_1, u_2, \ldots, u_{n-1}\}, u_n))$.

F. Sorting Vertices Based on Color

As described in Section IV, sorting the vertices based on graph coloring can significantly reduce the running time of solving the maximum clique problem on a single node. Our experiments based on random graphs also show that, in most cases, sorting the vertices based on color before partitioning leads to fewer partitions than other sorting methods such as sorting the vertices based on order. After partitioning, the running time to find the maximum clique in each subgraph of the final partitioning does not vary much, regardless of how the vertices were sorted. And since sorting based on color leads

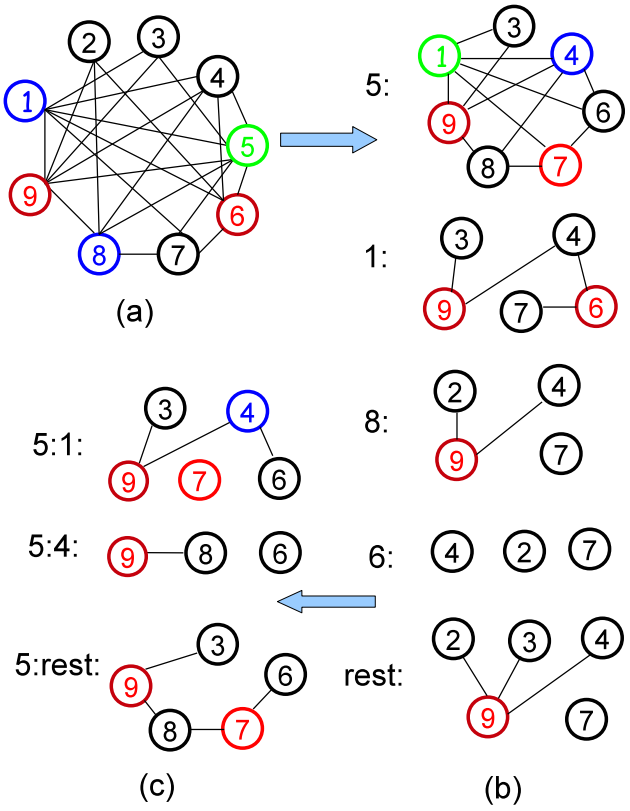

Fig. 1. BMC partitioning. The numbers for the colors are: $black = 1$, $red = 2$, $blue = 3$, and $green = 4$.

to fewer partitions, it reduces the running time of the overall maximum clique computation.

Therefore, we heuristically propose to sort the vertices based on color before each partitioning step in the multi-depth partitioning is performed. The coloring method is the same as the one used in the MCR algorithm [30], described in Section IV. The vertex with the largest color is first selected, then the one with the largest color in the remaining graph is selected, and so on.

In our implementation, we sort based on color in both one-depth partitioning and multi-depth partitioning. However, our recommended partitioning method is multi-depth partitioning with sorting based on color. We call this partitioning method *Balanced Multi-depth Color-based (BMC)* partitioning.

Figure 1 presents an example to illustrate BMC partitioning. In this example, we assume that the running time of a graph with 6 vertices is greater than the bound value, i.e, the bound value corresponds to 5 vertices. First, the vertices are sorted based on degree in descending order, resulting in the sorted list {5, 1, 9, 4, 6, 8, 7, 2, 3}. Next, we use Algorithm 1 to color these vertices. The colored vertices are shown in Figure 1 (a). The sorted vertices based on color in descending order are {5, 1, 8, 6, 9, 3, 2, 7, 4}. Therefore, vertex 5 is first selected for the partitioning (Figure 1 (b)) since it has the maximum color = 4. Then vertex 1 is selected because it has the highest color number after vertex 3 is removed, then vertex 8 is selected after vertices {5, 1} are removed, and so on until the remaining graph has only 5 vertices and the estimated running time is less than the bound value. But we find that the subgraph from the neighborhood of vertex 5 has 7 vertices, so it needs to be partitioned again. Before partitioning, its vertices are again sorted based on the degree in this subgraph resulting in the list {1, 4, 9, 6, 7, 8, 3}. Then this list is colored and the sorted list of vertices based on color is {1, 4, 7, 9, 6, 8, 3}. Thus, vertex 1 is first selected since it has the maximum color number and then the next selected vertex is vertex 4. After that, we find that both the subgraphs partitioned via vertex 4 and the remaining subgraph have no more than 5 vertices (Figure 1 (c)). The BMC partitioning is then done.

VI. Running Time Estimation

When we partition the graph into smaller subgraphs for better load balancing, the running time of each subgraph should be no greater than a bound value. Therefore, we need to model the relationship of the running time versus the number of vertices and the density of the graph. In this section, we present an experiment-driven model to estimate running time. The model can easily be calibrated for different computing clusters with different hardware and software configurations.

The exact relationship between running time and vertices and density is difficult to obtain. Therefore, we only aim to get an approximate relationship. First, we know that the worst case running time for a given graph G is $T(G) = O(a^{|G|})$, where a is a constant. Thus, an accurate model for running time would require an exponential function. For our model, we assume that $T(G) = f(|G|, \rho(G))^{|G|g(\rho(G))}$, where $f(|G|, \rho(G))$ and $g(\rho(G))$ are polynomial functions of the number of vertices $|G|$ and the density $\rho(G)$. We conducted experiments in which we observed that the function $g(\rho(G))$ can be modeled accurately as a quadratic function of $\rho(G)$. The Taylor series expansion of $f(|G|, \rho(G))$ gives us the following:

$$T(G) = (\sum_{i=0}^{k} |G|^i \sum_{j=0}^{k} a_{ij} \rho(G)^j)^{|G|(\rho(G)^2 + b\rho(G) + c)} \quad (3)$$

where k is set to control the degrees of freedom in the model, and a_{ij}, b, and c are parameters determined by fitting training data obtained through experiments.

We run the model training experiments on a given machine A and obtain the model parameters for that machine. After that, we can estimate the running time on another machine B by multiplying a normalization factor $f = T_B(G_0))/T(G_0)$, where G_0 is a calibration graph, $T(G_0)$ is the estimated running time for graph G_0 on machine A based on Equation (3), and $T_B(G_0)$ is the actual running time for graph G_0 on machine B. Thus, for a given input graph G, the estimated running time on machine B is given by:

$$T_B(G) = T(G) \times \frac{T_B(G_0)}{T(G_0)} \quad (4)$$

We have found that this equation is not overly sensitive to the choice of calibration graph, G_0. In our experiments, we choose G_0 as a random graph with $|G_0| = 500$ and $\rho(G_0) = 0.5$.

VII. Implementation and Optimizations

In this section, we show the implementation of our algorithm in MapReduce, in particular Hadoop, and optimizations of this implementation. We reiterate that other cluster computing frameworks such as MPI or Pregel can be used for this implementation, but they do not give us the flexibility, automatic scalability and fault tolerance, and wide adoption that we have with MapReduce.

A. Implementation

In our implementation, we use two MapReduce jobs: the first job performs the partitioning and the second one computes the maximum clique of the subgraphs. There is only one input to the partitioning MapReduce job, namely the original graph. In order to parallelize the computation as soon as possible, we do only one-depth partitioning in the map phase of this job, and distribute the subgraphs to the reducers to continue the partitioning process. The *(key, value)* pair from this map function contains $(v, N(G, v))$. Each reducer task will partition the subgraph into smaller subgraphs until the estimated running time of each smaller subgraph is less than the bound value. The output of the reducers is a set of $(key, value)$ pairs. Each $value$ is a partitioned subgraph, e.g., G', and the key is the set of vertices $\{v_{d1}, v_{d2}, \ldots, v_{dm}\}$ that records how G' was partitioned from the original graph G. The pseudocode of the MapReduce implementation of BMC partitioning is shown in Algorithm 3. The global parameter *BOUND* is used to determine when partitioning is terminated. The function *Running_Time(G)* estimates the running time required to solve the maximum clique problem for graph G.

In the second MapReduce job, the maximum clique of each subgraph is computed using Algorithm 2. The map function reads strings containing the subgraphs and their expanding vertices from HDFS files, which are created by the reducers of the first MapReduce job. The map function then splits these strings into $(key, value)$ pairs, where the $value$ is the subgraph G' and the key is the set of vertices that records how subgraph G' was partitioned from G. In the reduce function, we use Algorithm 2 to calculate the maximum clique for each subgraph. It should be noted that this algorithm can easily be replaced by other maximum clique algorithms such as MCS [31] and Cliquer [24], if desired, without compromising the scalability of the solution. The pseudocode of the second MapReduce job is shown in Algorithm 4.

B. Optimizations in Hadoop

In order to implement an efficient branch and bound search, we use a a global parameter MAX to store the number of vertices in the largest clique found so far. This global parameter is used to set the pruning parameter $|Q_{max}|$ in Algorithm 2 so that it can reduce the search space for most subgraphs. The global parameter MAX is communicated via a socket to the Hadoop job tracker. Before computing the maximum clique for a subgraph G' expanded from vertices $\{v_{d1}, v_{d2}, \ldots, v_{dm}\}$, the reduce function of the second MapReduce job first receives MAX from the socket, and then adds $\{v_{d1}, v_{d2}, \ldots, v_{dm}\}$ to

Algorithm 3 BMC partitioning using MapReduce.

```
1: /* First map function (key, values) */
2: function map_one(key, values)
3: G = get_graph(values)
4: while Running_Time(G) > BOUND do
5:     /* C_G is the color list of G*/
6:     i = {j|C_G[v_j] = max(C_G[v_k]|v_k ∈ G)}
7:     Emit (v_i, N(G, v_i))
8:     G = G − {v_i}
9: end while
10: Emit (-1, G)
11: return
12:
13: /* First reduce function (key, values) */
14: function reduce_one(key, values)
15: G = get_graph(values)
16: w = get_vertex(values) /* v_i in line 7 */
17: multi_partition(G, w, BOUND)
18: return
19:
20: function multi_partition(G, w, BOUND)
21: C_G = Color(G)
22: while Running_Time(G) > BOUND do
23:     i = {j|C_G[v_j] = max(C_G[v_k]|v_k ∈ G)}
24:     if Running_Time(N(G, v_i)) > BOUND then
25:         multi_partition(N(G, v_i), w ∪ v_i, BOUND)
26:     end if
27:     Emit (w ∪ v_i, N(G, v_i))
28:     G = G − v_i
29: end while
30: Emit (w, G)
31: return
```

Algorithm 4 Maximum clique on the partitioned subgraphs using MapReduce.

```
1: /*Second map function (key, values) */
2: function map_two(key, values)
3: G = get_graph(values) /* Read graph */
4: /* key is w ∪ v_i in line 29 of algorithm 3 */
5: key = get_vertices(values)
6: Emit (key, G)
7: return
8:
9: /* Second reduce function (key, values) */
10: function reduce_two(key, values)
11: G = get_graph(values)
12: Q = get_vertex(key)
13: |Q_max| = MAX /* MAX is a global parameter */
14: Q = MaxClique(G)
15: if |Q| > MAX then
16:     Update global parameter MAX with |Q|
17: end if
18: return
```

Graph	Order	Size	Density	ω	Comment
G_1	738	192243	0.71	19	Quantum Code
G_2	250	27980	0.9	44	DIMACS C250.9
G_3	2000	999836	0.5	16	DIMACS C2000.5
G_4	4000	4000268	0.5	18	DIMACS C4000.5

TABLE II

FOUR GRAPHS USED FOR THE EXPERIMENTS.

Q and sets $|Q_{max}| = MAX$. The maximum clique of this subgraph is then obtained by running MaxClique(G'). After the algorithm terminates, we again request a current value of MAX over the socket and compare it with the $|Q_{max}|$ found for G'. If $|Q_{max}|$ is greater than MAX, a new maximum clique has been found from this subgraph and we send the value of $|Q_{max}|$ to the job tracker over the socket and update the parameter MAX. Otherwise, we do not update MAX. To obtain fault-tolerance, we write MAX into an HDFS file once it is updated, so that if the socket connection fails, the parameter MAX can be read from this HDFS file.

When deciding the *BOUND* parameter, there is a tradeoff between load balancing and the amount of time taken to move graphs to worker nodes. To obtain good load balancing, we need to ensure that there are no large subgraph that take a long time for maximum clique computation. Therefore, we should partition the graph into small subgraphs, i.e., set the bound value to be small. However, if the bound value is too small, the number of partitioned subgraphs is too large, and it takes a long time to shuffle the subgraphs to worker nodes. It should be noted that the number of partitioned subgraphs increases exponentially with decreasing the bound value. Therefore, an optimal choice of the bound value may improve the performance of our algorithm. We experimentally study the optimal value of the bound in Section VIII-B.

VIII. EXPERIMENTS

A. Experimental Environment

Our experiments are based on four graphs, which are summarized in Table II. The first graph is a real-world coding graph for a CWS quantum error-correcting code, for given length $n = 10$ and $d = 3$. The other three graphs are DIMACS graphs [7], [8], which are synthetically generated graphs meant as benchmarks for maximum clique computation. All the graphs are undirected and unweighted.

We implemented our algorithm on Hadoop 1.0.0, and we perform all our experiments using Amazon's Elastic Compute Cloud (EC2) [10], except the investigation of the running time estimation (Section VIII-B). Since we mainly want to measure the scalability of our algorithm, we use small Amazon EC2 instances to run our experiments. Each small instance has 1.7 GB of memory and 1 virtual core with 1 EC2 compute unit, which has a similar performance as one 2007-era 1.0–1.2 GHz AMD Opteron or Xeon processor.

B. Running Time Estimation and Bound Value

In order to evaluate the accuracy of Equation (3), we first generate many random graphs with the number of vertices $|G|$ from 100 to 1000 in a step of 50 and the density ρ from 0.1 to $0.93 - 0.0003|G|$. Second, we use the MCR algorithm to solve the maximum clique problem for those graphs and record the running time on a PC with one dual core processor (2.9 GHz) and 4 GB memory. Third, we use some graphs with $|G| = 100, 200, \ldots, 1000$ as training data to fit Equation (3), and the remaining graphs to test whether the model can predict the running time.

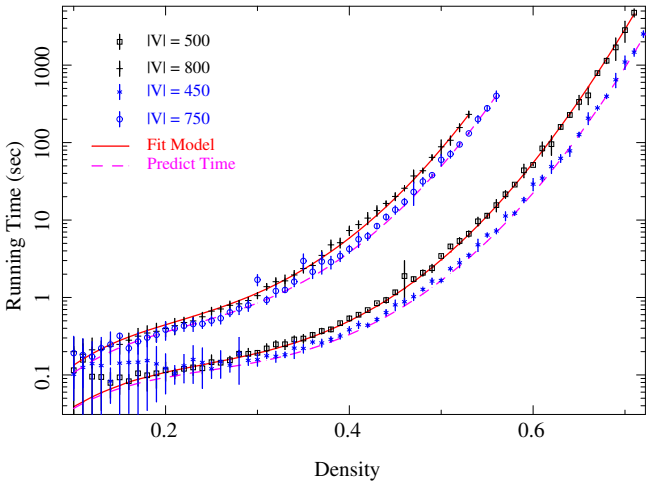

Fig. 2. Running time for random graphs over-plotted by the best-fit model. The black data are training data used to fit the model (red solid lines), while the blue data are test data used to evaluate model predictions (purple dashed lines).

We find that when $k = 4$, our model could fit the data very well ($\chi^2/d.o.f = 550/527$ with 27 parameters). Therefore, we use $k = 4$ in this paper. Some graphs over-plotted by the best-fit model are shown in Figure 2 (note that the y-axis for Figures 2, 4, and 5 is log scale). The predicted running time from the model is also shown in the figure, which shows that the model does indeed give a good estimation of the running time. The figure shows that the model is accurate even up to a running time of around 100 minutes. Since we set the bound value to 60 seconds (i.e., 1 minute), the subgraphs on which the model will be invoked will be well within the region in which it is accurate. Moreover, even if the model estimates are inaccurate, the end-to-end performance of the maximum clique algorithm will not be severely impacted since the algorithm is robust over a wide range of the bound value, as we show next.

To evaluate the robustness of the algorithm when changing the bound value on the estimated running time of the partitioned subgraphs, we first note that if the bound value is too large, some subgraphs will be too large and the load balance will be bad. However, if the bound value is too small, the number of subgraphs will be too large and Hadoop will need more time to shuffle, read, and write the data. Recall that the number of subgraphs increases exponentially with decreasing the bound value.

We use 32 small EC2 instances for this experiment. The total running time versus the bound value for graphs G_1 and G_2 is shown in Figure 3. From this figure, we can see that there is a wide range in which the bound value results in

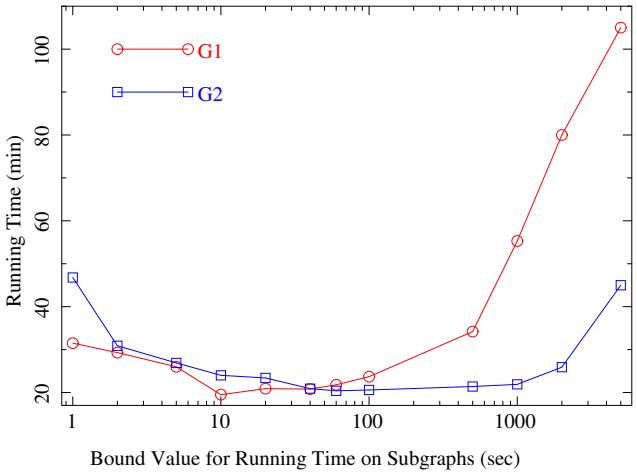

Fig. 3. Running time of G_1 and G_2 versus the bound value on the running time of the partitioned subgraphs on 32 EC2 nodes.

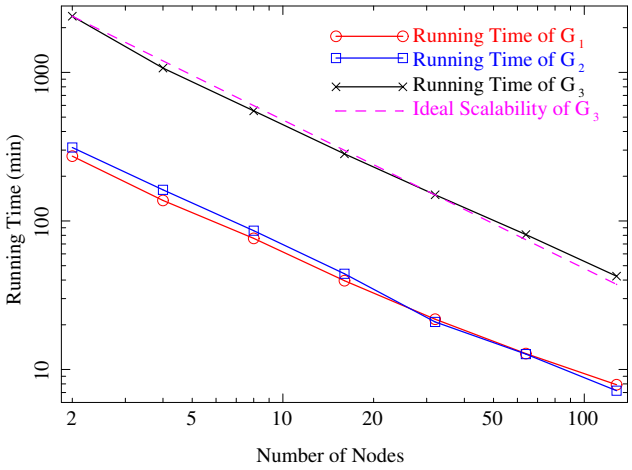

Fig. 4. The scalability of our algorithm. The dashed purple line shows ideal scalability, i.e., $T(n) = T(1)/n$, where $T(n)$ is the running time on n EC2 nodes.

good performance (approximately 5–500 seconds). Therefore, our algorithm is robust to changes or inaccuracy in setting the bound value, and our choice of 60 seconds is justified.

C. Scalability

In this section, we investigate the scalability of our proposed algorithm, namely, BMC partitioning with the optimization of the global parameter MAX. Comparisons between our algorithm and other alternatives, such as different partitioning strategies, removing the global parameter MAX, one-depth partitioning, and comparing to an MPI algorithm, are presented in the following sections.

We run our algorithm for three graphs shown in Table II, G_1, G_2, and G_3. We vary the number of EC2 nodes from 2 to 128. The total running time versus the number of EC2 nodes is shown in Figure 4. In order to illustrate the scalability of our algorithm, we over-plot a dashed line that indicates ideal scalability, i.e, $T(n) = T(1)/n$, where $T(n)$ is the running time on n EC2 nodes. This figure shows that our algorithm provides excellent scalability that is very close to ideal.

The largest graph (G_4) is not used to investigate scalability because its running time is too large, particularly on a small number of nodes such as 2 or 4. However, we use this graph to test the effectiveness of our algorithm for solving the maximum clique problem on such large graphs by running it on 128 high CPU medium Amazon EC2 instances, each of which has two CPU cores equivalent to 5 compute units. The problem is completely solved in 39 hours. The size of the maximum clique was found to be 18, which is the same size as the largest clique previously found in this graph. However, we can now say with certainty that this is indeed the maximum clique, and we can enumerate all cliques that have this maximum size. This is the first time to completely solve the maximum clique problem for this DIMACS instance. We estimate that a solution on a single CPU would have required more than 1 year.

D. Comparison of Partitioning Strategies

In this section, we use graphs G_1 and G_2, plus 4 randomly generated graphs, to investigate whether our BMC partitioning is better than other partitioning strategies.

In this experiment, bisection partitioning is performed using the Kernighan and Lin algorithm [19], as well as k-way partitioning based on spectral factorization [16] with k set to 16. We also evaluate PICS partitioning, which is implementated using the code in [1]. In each experiment, when the number of vertices of a subgraph is 150 or less, the subgraph is not partitioned any further.

In order to investigate the effectiveness of each partitioning strategy, Table III lists the final number of subgraphs resulting from the partitioning and the number of vertices in the largest subgraph. From the table we can see that bisection partitioning and PICS cannot partition the original graph into smaller subgraphs for any graph. The largest subgraph has the same size as the original graph even for a graph with low density (R_4: $|G| = 500$, $\rho(G) = 0.05$). The k-way partitioning algorithm can partition graphs with low density into smaller subgraphs but it is not as effective as our BMC partitioning algorithm. Additionally, k-way partitioning also cannot partition graphs with medium or high density. In contrast, Table III shows that BMC partitioning can effectively partition any graph into smaller subgraphs.

E. Sorting By Color vs. Random Ordering

The sorting order of vertices (based on color or not) may affect the performance of our algorithm. Here, we use graphs G_1 and G_2 to investigate whether BMC partitioning, which does sort based on color, is better than a partitioning strategy based on a random order of vertices. The comparison of the running time of G_1 and G_2 based on these two strategies on 32 EC2 nodes is shown in Table IV. The experiment shows that BMC partitioning with vertices sorted based on color can significantly improve performance.

F. Global Parameter MAX

The running time of graph G_1 and G_2 on 32 EC2 nodes without the global MAX parameter is shown in Table IV.

Graph	Name	G_1		G_2		R_1		R_2		R_3		R_4	
	$\|V\|$	250		738		500		500		500		500	
	Density	0.9		0.73		0.3		0.2		0.1		0.05	
Name	depth	n	v	n	v	n	v	n	v	n	v	n	v
bisection	1	3	250	3	738	3	500	3	500	3	500	3	500
	2	5	250	9	738	9	500	9	500	9	500	9	500
	3	7	250	27	738	19	500	19	500	19	500	19	500
	4	9	250	65	738	33	500	33	500	33	500	33	500
partitioning	5	11	250	131	738	51	500	51	500	51	500	51	500
	6	13	250	233	738	73	500	73	500	73	500	73	500
	1	31	250	31	737	31	500	31	496	31	458	31	444
	2	211	250	421	737	421	500	421	495	361	426	271	350
k-partitioning	3	1051	250	4381	737	4321	500	4351	470	2341	389	781	241
	4	4021	250	34771	737	35191	500	31711	461	9241	341	1170	148
	5	12721	250	224851	737	224160	500	174300	457	24270	304	1168	148
	6	34831	250	1228131	737	1174463	500	764451	450	37949	269	1168	148
	1	3	250	7	737	3	500	3	500	3	500	3	500
PICS	2	5	250	67	737	5	500	7	500	5	500	7	500
	3	7	250	353	737	7	500	13	500	7	500	11	500
	4	9	250	2171	737	9	500	21	500	9	500	15	500
partitioning	5	11	250	13419	737	11	500	31	500	11	500	19	500
	6	13	250	78496	737	13	500	43	500	13	500	23	500
	1	101	210	589	484	351	150	351	150	351	150	351	150
BMC	2	2830	183	98062	350	351	150	351	150	351	150	351	150
	3	18703	160	4693076	258	351	150	351	150	351	150	351	150
	4	20951	150	40869924	184	351	150	351	150	351	150	351	150
partitioning	5	20951	150	42730190	150	351	150	351	150	351	150	351	150
	6	20951	150	42730190	150	351	150	351	150	351	150	351	150

TABLE III

DIFFERENT PARTITIONING STRATEGIES. n IS THE FINAL NUMBER OF SUBGRAPHS, AND v IS THE NUMBER OF VERTICES IN THE LARGEST SUBGRAPH.

Graph	Color Partitioning	Random Partitioning		Without MAX	
	T_0	T_r	$\frac{T_r - T_o}{T_o}$ (%)	T_m	$\frac{T_m - T_o}{T_o}$ (%)
G_1	21.8	51.5	135	198	804
G_2	20.9	30.0	46	108	427

TABLE IV

RUNNING TIME OF G_1 AND G_2 ON 32 EC2 NODES BASED ON COLOR PARTITIONING COMPARED TO THE RUNNING TIME BASED ON RANDOM PARTITIONING. THE RUNNING TIME WITHOUT THE GLOBAL PARAMETER MAX IS ALSO SHOWN. TIMES (T_0, T_r, AND T_m) IN MINUTES.

Without the MAX parameter, $|Q_{max}|$ in Algorithm 2 is set to zero for each subgraph. The table clearly shows that the global parameter MAX significantly improves performance.

G. One-depth Partitioning vs. Multi-depth Partitioning

In order to investigate whether BMC partitioning is better than one-depth partitioning, we implement a Hapdoop MapReduce program to calculate the maximum clique of a graph using one-depth partitioning. The graph is partitioned with one-depth partitioning until the estimated running time of the remaining graph is less than the bound value, i.e., 60 seconds, which is the same as the bound value in multi-depth partitioning. The relationship between the total running time and the number of EC2 nodes for graph G_2 is shown in Figure 5. The figure also shows the result of BMC partitioning. It is clear that one-depth partitioning does not have good scalability for dense graphs because it does not effectively balance load.

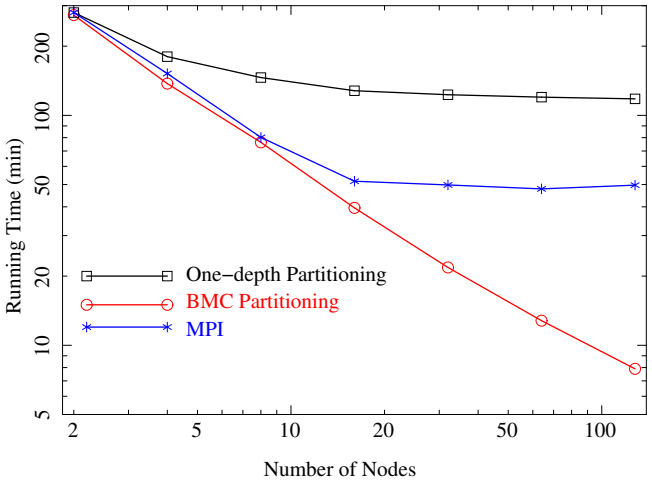

Fig. 5. Running time of G_2 using a varying number of EC2 nodes based on one-depth partitioning and an MPI implementation, compared to BMC partitioning.

H. Comparison with an MPI Algorithm

We developed an MPI-based maximum clique algorithm using the MPICH implementation of MPI. We employ the framework proposed in [25], in which one process acts as the master and collects results from other processes, which act as workers. The master first sends the original graph to all the workers, and then assigns one vertex to each worker node. If the master sends the vertex index "k" to worker i, worker i calculates the maximum clique of the subgraph $N(G - \{v_1, v_2, \ldots, v_{k-1}\}, v_k)$, and sends the result back to

the master. If one worker has finished its work, the master will send another partition to it until the entire job has been finished. The algorithm in [25] calculates the maximum clique of subgraphs using the classical algorithm in [3]. We replace the algorithm in the workers with the MCR algorithm [30] to improve performance.

The MPI algorithm is similar to the one-depth partitioning algorithm in that it also suffers from poor load balancing. In order to investigate the performance and scalability of the MPI algorithm, we use this algorithm to calculate the maximum clique of graph G_2. Figure 5 shows the total running time versus the number of EC2 nodes. The number of nodes in this figure refers to the number of worker nodes, since the master node in the MPI algorithm does not do any computation. Our one-depth algorithm is comparable in performance with the MPI algorithm even though the MPI implementation is in C and our one-depth implementation is in Java. Our multi-depth BMC partitioning significantly outperforms the MPI implementation because BMC has better scalability.

IX. Conclusion

In this paper, we presented a scalable and fault-tolerant maximum clique algorithm based on MapReduce. The main component of our approach is the BMC partitioning algorithm, which partitions a graph into subgraphs in a way that maintains load balance. A branch and bound search based on the partitioned graph is used to compute the maximum clique. To improve the pruning power of the branch and bound search, a global parameter MAX is used to communicate between different maximum clique computations (different reducers in our MapReduce implementation). Our experiments show that our approach outperforms other approaches on real and synthetic graphs, and is robust to tuning parameter settings. The paper demonstrates that judicious use of simple partitioning techniques can lead to good scalability. At a high level, the paper presents an interesting study in the use of MapReduce for CPU intensive analytics computations.

X. Acknowledgements

We thank Dr. Bei Zeng and Dr. Markus Grassl for helpful discussions and for providing us with real-world quantum coding graphs. This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and by an Amazon Web Services Research Grant.

References

[1] L. Akoglu, H. Tong, B. Meeder, and C. Faloutsos. PICS: Parameter-free identification of cohesive subgroups in large attributed graphs. In *Proc. SIAM Data Mining Conf. (SDM)*, 2012.

[2] N. Alon, M. Krivelevich, and B. Sudakov. Finding a large hidden clique in a random graph. In *Symp. on Discrete Algorithms (SODA)*, 1998.

[3] R. Carraghan and P. M. Pardalos. An exact algorithm for the maximum clique problem. *Operations Research Letters*, 9(6), 1990.

[4] I. Chuang, A. Cross, G. Smith, J. Smolin, and B. Zeng. Codeword stabilized quantum codes: Algorithm and structure. *J. Mathematical Physics*, 50(4), 2009.

[5] A. Cross, G. Smith, J. Smolin, and B. Zeng. Codeword stabilized quantum codes. *IEEE Trans. on Information Theory*, 55(1), 2009.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Comm. ACM*, 51(1), 2008.

[7] DIMACS Benchmark Set. http://iridia.ulb.ac.be/∼fmascia/maximum_clique/.

[8] DIMACS Implementation Challenges. http://dimacs.rutgers.edu/Challenges/.

[9] X. Dimitropoulos, D. Krioukov, M. Fomenkov, B. Huffaker, Y. Hyun, kc Claffy, and G. Riley. AS relationships: inference and validation. *SIGCOMM Comp. Comm. Rev.*, 37(1), 2007.

[10] Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/.

[11] L. Gao. On inferring autonomous system relationships in the internet. *IEEE/ACM Trans. Netw.*, 9(6), 2001.

[12] E. Gardiner, P. Artymiuk, and P. Willett. Clique-detection algorithms for matching three-dimensional molecular structures. *J. Molecular Graphics and Modelling*, 15(4), 1997.

[13] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Co., 1979.

[14] Giraph project. http://giraph.apache.org/.

[15] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Comp.*, 22(6), 1996.

[16] L. W. Hagen and A. B. Kahng. New spectral methods for ratio cut partitioning and clustering. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 11(9), 1992.

[17] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Proc. Conf. on Supercomputing (SC)*, 1995.

[18] C. W. Huffman and V. Pless. *Fundamentals of Error-Correcting Codes*. Cambridge University Press, 2003.

[19] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49, 1970.

[20] P. Lin, Z. Wang, and M. Guo. A maximum clique algorithm based on MapReduce. In *Proc. Int. Conf. on Advanced Computer Theory and Engineering*, 2010.

[21] R. D. Luce and A. Perry. A method of matrix analysis of group structure. *Psychometrika*, 14, 1949.

[22] F. MacWilliams and N. Sloane. *The Theory of Error-Correcting Codes*. North-Holland, second edition, 1978.

[23] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2010.

[24] P. R. Östergård. A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics*, 120(1–3), 2002.

[25] P. M. Pardalos, J. Rappe, and M. G. Resende. An exact parallel algorithm for the maximum clique problem. *High Performance Algorithms and Software in Nonlinear Optimization*, 38, 1998.

[26] P. M. Pardalos and J. Xue. The maximum clique problem. *J. Global Optimization*, 4(3), 1994.

[27] M. Sellmann, N. Sensen, and L. Timajev. Multicommodity flow approximation used for exact graph partitioning. In *Proc. Annual European Symp. on Algorithms (ESA)*, 2003.

[28] N. Sensen. Lower bounds and exact algorithms for the graph partitioning problem using multicommodity flows. In *Proc. Annual European Symp. on Algorithms (ESA)*, 2001.

[29] M. Sipser. *Introduction to the Theory of Computation*. Course Technology, third edition, 2012.

[30] E. Tomita and T. Kameda. An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments. *J. Global Optimization*, 37(1), 2007.

[31] E. Tomita, Y. Sutani, T. Higashi, S. Takahashi, and M. Wakatsuki. A simple and faster branch-and-bound algorithm for finding a maximum clique. In *WALCOM: Algorithms and Computation*. Springer, 2010.

[32] L. G. Valiant. A bridging model for parallel computation. *Comm. ACM*, 33(8), 1990.

[33] J. Wang and J. Cheng. Truss decomposition in massive networks. *Proc. VLDB Endowment (PVLDB)*, 5(9), 2012.

[34] B. Wu, S. Yang, H. Zhao, and B. Wang. A distributed algorithm to enumerate all maximal cliques in MapReduce. In *Proc. Int. Conf. on Frontiers of Computer Science and Technology*, 2009.

[35] S. Yang, X. Yan, B. Zong, and A. Khan. Towards effective partition management for large graphs. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2012.