HP-MC: Quickly Computing Maximum Clique Sizes of Large Sparse Graphs

Cameron Bradley
Department of Computer Science
Texas State University
San Marcos, TX, USA
clb420@txstate.edu

Martin Burtscher
Department of Computer Science
Texas State University
San Marcos, TX, USA
burtscher@txstate.edu

***Abstract*—Computing the exact size of the maximum clique is an important NP-hard graph problem with applications in biology, chemistry, evolutionary history, and social network analysis. State-of-the-art implementations incorporate a multitude of techniques such as graph coloring and degeneracy ordering to minimize the search space. We present a novel algorithm named HP-MC that, like prior approaches, is based on the branch and bound search. However, our OpenMP parallelized implementation incorporates a number of new optimizations, including using a pivot vertex, employing a hybrid strategy, pruning vertices based on induced connected components, skipping one vertex per component, iteratively coloring and pruning vertices, and utilizing work stealing. Our evaluation shows that HP-MC outperforms the leading maximum clique codes on large sparse graphs of various topologies by up to 10x on average.**

***Index Terms*—Maximum clique size, pivot vertex, parallelism, large sparse graphs, branch and bound search**

I. Introduction

Given an undirected graph $G = (V, E)$, the Maximum Clique Problem (MCP) is to identify the largest subset of vertices that form a complete subgraph, meaning each vertex must have an edge to every other vertex in the subgraph. MCP is a fundamental problem in graph analytics. For example, finding large cliques can be used to solve the compatibility problem in phylogeny [8], to find signals for terrorist recruitment [2], and to serve as building blocks for molecular docking [15]. In practice, the maximum clique can be computed reasonably efficiently on many inputs. However, MCP is NP-hard [3] and, in the worst case, a brute-force search must consider every subset of vertices to identify the largest clique, making the running time exponential since a graph of $|V|$ vertices has $2^{|V|}$ possible subsets. This combinatorial explosion exposes the problem's computational intractability.

The traditional approach to finding the maximum clique relies on enumerating *maximal* cliques. A maximal clique is a subset of vertices that form a complete induced subgraph that cannot be extended by adding any adjacent vertex. Since every maximum clique is necessarily maximal, enumerating all maximal cliques and selecting the largest is a solution to MCP. Classic methods use backtracking and branch and bound search, where one starts with an empty set, known as the partial clique, and adds vertices recursively while maintaining completeness.

Enumerating maximal cliques is inherently exponential. To accelerate this search, reductions to a given set of vertices can take place before any branches are explored. State-of-the-art implementations incorporate several such bounding strategies to limit the depth explored in a branch and bound approach. A common method is to update the lower and upper bounds of clique sizes across the graph. Once both bounds are equal, the search can terminate, and the size is guaranteed to be the size of the maximum clique in the graph.

Currently, some of the best techniques for computing upper bounds use vertex degrees, core sizes, and graph coloring [20]. Lower bounds are typically computed using a heuristic algorithm to approximate the maximum clique size [5]. These and other strategies are described in more detail in Section II. In practice, the resulting bounds are often very close to the actual solution. However, they do not provide any guarantees. Moreover, the topology of an input can arbitrarily widen the gap between the estimated and the true maximum clique size [23]. As a consequence, an exponential branch and bound search is always required if the lower and upper bound are not equal. The cost of the exact clique computation increases exponentially with the width of this gap. Problematic inputs tend to share some combination of structural properties such as heavy-tailed degree distributions [12], creating very dense neighborhoods with near-clique structures.

In this paper, we present the HP-MC algorithm for determining the exact size of the maximum clique. Our approach is specifically designed for large sparse inputs such as social networks. At a high level, it follows the standard branch and bound paradigm used in many state-of-the-art exact solvers. HP-MC starts with a linear-time heuristic to construct a large clique (for obtaining an initial lower bound) and computes a combination of core and color-based upper bounds per vertex to prune the graph. However, it differs in several key aspects that significantly improve performance on sparse inputs. In particular, our OpenMP parallelized HP-MC implementation introduces the following novel optimizations that are not present in other leading maximum-clique codes.

(1) Instead of only using a single coloring pass in each step, HP-MC employs iterative coloring. Whenever coloring results in some vertices being pruned from the candidate set, it recolors the smaller subgraph, which often allows to prune

even more vertices. (2) HP-MC adopts the idea of using a pivot vertex from the related domain of maximal clique enumeration. After exhausting the pivot vertex branch, all direct neighbors of this vertex can be excluded from branching. (3) To make pivoting effective for maximum clique searches, HP-MC incorporates a hybrid strategy. It only uses the pivoting strategy on candidate sets that are large and sparse. For denser and smaller candidate sets, it switches to a different solver that does not employ pivoting. (4) Removing the pivot vertex and its neighbors from a candidate set often results in multiple connected components. HP-MC colors each component and prunes those that contain too few unique colors. (5) It skips one vertex in each connected component. This reduces the number of branches explored while maintaining optimality. Note that the skipped vertex is not pruned from future candidate sets. (6) HP-MC utilizes work stealing to improve load balance.

This paper makes the following main contributions.

- It presents HP-MC, a new maximum clique size finding approach for large sparse graphs.
- It introduces novel pruning, optimization, and parallelization strategies to reduce the maximum search depth, speed up the computation, and maintain parallelism.
- It demonstrates that our parallel CPU implementation can be orders of magnitude faster than the fastest prior codes.

Our OpenMP implementation is freely available in open source at https://github.com/burtscher/HP-MC.

The rest of the paper is organized as follows. Section II provides background information. Section III summarizes related work. Section IV details the implementation of our approach. Section V discusses our experimental methodology. Section VI presents and analyzes the performance results. Section VII concludes the paper with a summary.

II. Background

The goal of HP-MC is to quickly compute the size of the maximum clique of a given input graph. However, there are multiple sub-classes of MCP. For the purpose of this paper, we focus on unweighted, undirected, simple graphs. Most state-of-the-art techniques in this sub-class share the following common approach. They begin with heuristic algorithms to compute an initial upper and lower bound. The heuristics exploit information such as vertex degrees, core sizes, and graph coloring to determine an upper bound. The lower bound is generally obtained with some variation of a greedy clique construction. As larger cliques are found during the exact search, the lower bound is updated. Inside the exact search, the bounds are used to prune branches. For example, if the size of a partial clique plus the number of all remaining candidate vertices is not larger than the current lower bound, this search branch can be pruned as it cannot yield a new larger clique.

A. Lower Bounds

Beyond identifying a new large clique to serve as a lower bound, state-of-the-art implementations employ a greedy computation to find large cliques fast. Often, this approach involves repeatedly picking a vertex with the highest degree that is adjacent to a current partial clique.

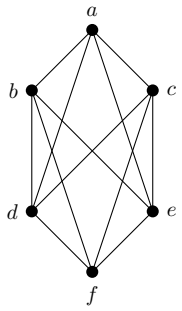

Fig. 1. Example graph with core-bound of 4

B. Degree Bounds

Before computing any explicit lower or upper bounds, the least expensive global constraint on the maximum clique is based on the largest degree vertex. The maximum clique size of any graph $\omega(G) \leq \max_{v \in V}(d(v)+1)$. This is because, in any clique of size $k+1$, every vertex must have at least degree k since they all connect to k other vertices in the clique.

After a lower bound is greedily computed, the degree-based upper bound can be applied to each vertex locally. For a vertex $v \in V$, the largest possible clique containing v is $d(v)+1$. As a result, all vertices with degrees smaller or equal to the lower bound $lb-1$ can be pruned. Despite these vertices being the easiest to prune, this approach often discards a significant portion of the graph, especially in power-law graphs.

C. Effective Degree / Core Bounds

For a vertex v with a degree of $d(v)$ to be a part of a clique size $d(v)+1$, every adjacent vertex u must also be a part of the clique and have a degree of $d(u) \geq d(v)$. This introduces the concept of core numbers (or effective degrees) for a given vertex. Each vertex in the graph is given a value k, meaning there are at least k vertices adjacent to vertex v with degrees of k or larger. For the purpose of finding the maximum clique, v effectively has a degree of k. This is a much tighter bound in comparison to the degree of each vertex. Regardless, it is still an upper bound and can be arbitrarily distant from the true maximum clique size. For example, in Figure 1, each vertex is given a k value of 4, despite the largest clique size being 3.

D. Coloring Bounds

A valid coloring of a graph by definition results in no adjacent vertices having the same color. The solution to the Graph Coloring Problem is the smallest number of colors needed to color a given graph, which is the chromatic number. This algorithm is NP-hard, similar to computing the maximum clique. For this reason, state-of-the-art approaches only use a heuristic greedy coloring of the graph to guide their upper bounds. Each vertex v is assigned a color number c. For any valid coloring, an upper bound for the size of the maximum clique each vertex can be in is the number of unique colors

adjacent to the vertex plus one [10]. For example, if vertex v has 10 neighbors that, together, use 7 unique colors, the upper bound $ub(v)$ is 8 because the 7 uniquely colored neighbors plus v can form a clique that is no larger than 8 vertices. This approach is based on the fact that all vertices in a clique must have unique colors.

Using graph coloring for determining upper bounds is the most widely used approach in the state of the art as it works well in practice. However, using a coloring heuristic often overestimates the number of colors required. Moreover, even the chromatic number can be arbitrarily larger than the true size of the maximum clique. For example, the existence of odd cycles can increase the number of colors required beyond the maximum clique size. We see this in Mycielski graphs [4], which are triangle-free but can be constructed to have an arbitrarily large chromatic number.

E. Branch and Bound

Ultimately, an exact search using a branch and bound algorithm is employed to bridge any remaining gap between the lower and upper bound. This exact search was originally developed by Land [11] and later tailored for the maximum clique problem by Bron and Kerbosch [3]. The algorithm begins with an initial empty partial clique and incrementally selects a vertex to add from the candidate set. The candidate set is the set of vertices that are adjacent to every vertex in the partial clique. Initially, the candidate set includes all vertices. At each level of the search, called a *branch*, a vertex from the candidate set is selected, added to the partial clique, and the candidate set is updated to again only hold the vertices that are adjacent to all vertices in the partial clique. At this point, new *bounds* can be recomputed. When the upper bound for a vertex does not exceed the size of the currently largest known clique, i.e., the global lower bound, that branch can be pruned, meaning it is not explored. Nevertheless, the algorithm is still exponential. In the worst case, it explores $O(2^n)$ decisions. Additional steps can be taken to reduce the number of branches explored overall. This includes restricting the selected vertices to branch off of to a pivot vertex and all non-neighbors of the pivot. The following paragraphs describe theorems regarding pruning rules to limit the number of required selected branches at each step of the branch and bound search tree.

Theorem 1. *During a branch-and-bound search, let P be the clique built so far, and let C be a set of candidate vertices adjacent to all members of P. Let u be any chosen vertex where $u \in C$. Every maximum clique that extends P must contain either u or at least one vertex of C not adjacent to u.*

Proof: Within a given candidate set of vertices C and partial clique P, we can assume that there is some extension of vertices from C that would build a maximal clique M if $|C| > 0$. By selecting any vertex as the pivot vertex u, there are two exhaustive cases. The first case is if $u \in M$, M will be found when branching on u. The second case is if $u \notin M$, u must not be adjacent to at least one vertex $\in M$. This is because vertex u is adjacent to all members of P, and if u were also adjacent to every member $\in M$, it would be a member of M. Since every vertex $v \in M$ is also $\in C$, branching on all vertices not adjacent to u will still lead to M being found.

Theorem 2. *Let C be a candidate set of vertices, and let each vertex $v \in C$ be assigned a color "$color(v)$" by a greedy coloring. Let C be sorted by color in descending order. Let $|P|$ be the current partial clique size and m be the current lower bound for the maximum clique. Assume that, when branching on a vertex $v_i \in C$, the next candidate set is restricted to $C' = C \cap N(v_i) \cap \{v_j \mid j > i\}$. If $|P| + color(v) \leq m$, then no maximum clique can be found by branching on v_i.*

Proof: Due to the candidate set C being sorted by color, the color assigned to each v_i acts as a bound for the size of any clique that can be formed by C'. Therefore, vertices with $color(v_i) + |P| > m$ are the only vertices required to be branched on at any step of the branch and bound search.

III. Related Work

This section describes the related work in chronological order. Exact MCP algorithms fall broadly into two generations: 1) backtracking algorithms for small, often dense benchmark graphs with bit-parallel and coloring-bounded methods for performance and 2) the current generation of algorithms engineered for large sparse graphs. In addition to this CPU-focused trajectory, there are GPU-based approaches for clique-related problems, though they target maximal clique enumeration due to their limited effectiveness on exact maximum clique searches. As a result, this section describes the publicly available PMC [14], MC-BRB [5], and Lazy-MC [22] exact MCP codes, which represent the current state of the art for large sparse graphs. Other approaches such as MCS [21], MaxCliquePara [6], and CliSAT [16] are also described. They represent the state of the art for small dense inputs, which are not the target of our work.

Tomita [21] improved on the branch and bound algorithm MCR [19] by strengthening vertex ordering and coloring-based pruning. This reduces the amount of search needed in practice on dense graphs. Before an approximate coloring is computed, the candidate set is sorted in descending order of degrees. The approximate coloring is applied in the same order, meaning the highest-degree vertices receive the lowest colors. This ordering helps because vertices with color numbers lower than the minimum required to build a clique that is larger than the current lower bound do not need to be branched on. Hence, this technique minimizes the size of the search space.

Rossi [14] introduced PMC, a parallel maximum clique algorithm that utilizes core-based bounds to prune branches. Their approach begins with a greedy clique construction from each vertex. If the core number of a vertex is smaller than the current lower bound, it is skipped. For all neighbors of a vertex, those that also have a core number larger than the lower bound are sorted in descending order. Then, each such neighbor is added as long as it continues to create a clique. Vertices with core numbers less than the current lower bound are explicitly removed from the graph. After computing the

core numbers of the vertices in an induced subgraph, a greedy coloring is used to attempt to find a tight upper bound for each vertex that still has a higher upper bound. The vertex with the largest color is then always picked as a branch. As soon as no vertex with a sufficiently large color is available, the candidate set is pruned.

Depolli [6] improved on the MaxCliqueSeq algorithm with a parallel implementation. Their MaxCliquePara code first greedily colors the vertices of the graph. This algorithm follows a similar understanding that color-based ordering is only needed above the threshold of the current lower-bound. MaxCliquePara also takes advantage of bit-strings to represent adjacency matrices instead of arrays. This allows for faster intersection functions. This does, however, incur a cost as with a bit-string, the order of vertices in the matrix is predefined. Search trees are then split to multiple cores, allowing for large cliques found with different vertices to contribute to pruning of other threads, thus improve performance over MaxCliqueSeq.

Chang [5] presented MC-BRB, which also starts by greedily constructing a large clique as a lower bound. However, it utilizes multiple approaches. This is done so the hightest lower bound is kept for better pruning in an exact search. The first technique is a degree-based greedy approach, sorting and selecting vertices with the highest degrees until no other vertex can be added to the clique. The second greedy construction involves the degeneracy of each vertex. The degeneracy order of the vertices is an order in which each vertex v has the smallest degree in an induced subgraph of its neighbors called an ego-network. For an upper bound, MC-BRB colors the graph and each induced subgraph. During the exact branch and bound search, when an induced subgraph of a candidate set is sufficiently dense, MC-BRB switches to a *k*-clique finding solver that utilizes multiple reductions. These reductions involve the one-, two-, and three-neighbor missing reduction rules. Due to dense neighborhoods in difficult inputs, it is common to find vertices with degrees close to the size of a candidate set. Based on the number of vertices not adjacent to these high-degree vertices, some can be pruned without exhausting their branches while still guaranteeing correctness.

Segundo [16] introduced CliSAT, which combines branch and bound searches with SAT-based bounding based on two techniques. The first combines graph coloring procedures with partial maximum satisfiability problems. Its goal is to enlarge the pruned set and reduce the number of required branches to exhaust. The second is a filtering phase based on constraint programming and domain propagation, which removes vertices from branching subtrees entirely.

Vandierendonck [22] developed the most recent maximum clique finding code named Lazy-MC. This approach switches from solving the maximum clique problem to solving the *k*-Vertex Cover (*k*-VC) problem of the complement graph when an induced subgraph is sufficiently dense. The *k*-VC solver utilizes reduction rules to reduce the problem further, such as vertices inside the complement with degrees that are too high being forced into all minimum vertex covers and thus out of any maximum clique. To find the value of k for the *k*-Vertex Cover, Lazy-MC uses a binary search over a range of plausible k values.

We compare our HP-MC code to several of the above implementations in the result section. We omit codes such as MCS [21], MaxCliquePara [6], CliSAT [16] since they target small dense inputs and either exceed memory limits or incur prohibitive runtimes on the large sparse graphs used in our evaluation. While differing in scope, HP-MC adopts and adapts key optimizations from prior work, such as coloring each candidate set for an approximate upper bound. It also uses an adjacency matrix with either a word or a bit representation when appropriate for faster intersection functions. However, no prior work incorporates a hybrid implementation of the pivot vertex heuristic and degeneracy ordering, employs work-stealing, includes iterative coloring, or uses connected components to prune additional vertices.

IV. HP-MC Approach

This section describes our HP-MC's algorithm and its parallel implementation in detail. It computes the largest maximal clique a vertex v is a member of by performing a recursive branch and bound search within v's adjacency list. Doing so for all vertices in the graph to find the largest clique would not be practical due to the exponentially large search space. HP-MC addresses this issue by minimizing the number of branches required to be searched per branch-and-bound step and parallelizes the strategies described in this section.

Algorithm 1 outlines these strategies. First, we compute an initial upper bound by coloring the graph using a largest-degree first (LDF) heuristic [1]. The upper bound on the clique size of each vertex is then set to the number of distinct colors used by the neighbors of the vertex plus 1, as described in Section II. Next, the vertices are sorted by degree in descending order. This ensures high-degree vertices are processed early. These vertices are the most likely to be members of large cliques and tend to yield larger lower bounds.

We then apply a parallel greedy clique construction to each vertex with a sufficiently high number of distinct adjacent colors, or upper bound. The greedy clique computation involves constructing the induced subgraph of a vertex v's adjacency list and computing the degrees of each vertex in the subgraph. This induced subgraph is captured in an adjacency matrix owned by a single thread. Vertices in the subgraph are sorted by degree, and the largest-degree vertex is added to the partial clique. Then, the largest-degree vertex that is adjacent to all vertices in the partial clique is iteratively included until no vertices within the subgraph can be added to the partial clique anymore, making it maximal. The size of this clique not only serves as the lower bound of this vertex, but the largest lower bound also serves as the global lower bound for the maximum clique. We skip the greedy computation if the upper bound of a vertex is smaller than the current global lower bound.

If the global lower bound is equal to the highest upper bound in the graph, the computation is done, and the maximum clique size has been found. If not, there is a list of vertices whose

upper bound is still higher than the global lower bound. These vertices are sorted by their upper bound in ascending order.

After this step, we compute, for each vertex in this list, the exact largest maximal clique of which it is a member using a variation of the branch and bound search. These variations are listed in Algorithms 2, 3, and 4. In each variation, a thread selects a vertex and starts its branch and bound search. We optimize this search further with the use of a greedy graph coloring in each step. In Algorithms 2 and 3, we select a pivot vertex and prune entire components outside of the pivot's neighborhood. These steps are described next.

Algorithm 1 General HP-MC Algorithm

Require: graph G, upper-bound ub, lower-bound lb, current maximum clique size m

LDFcolor(G) // ub(v) is determined by number of distinct colors
Relabel(G)
for each vertex $v \in V$ **do**
  if $ub(v) > m$ **then**
    $lb(v) \leftarrow$ GreedyClique(v)
    if $lb(v) > m$ **then**
      $m \leftarrow lb(v)$
$V' \leftarrow$ sort($v \in V$ in descending order of $ub(v) > m$)
for each vertex $v \in V'[1 \dots k]$ in sorted order **do**
  if $ub(v) > m$ **then**
    $ub(v) \leftarrow$ GreedyColorPrune($N(v)$)
    if $lb(v) < ub(v)$ **then**
      $lb(v),\ ub(v) \leftarrow$ csrBRB(v)
    if $lb(v) > m$ **then**
      $m \leftarrow lb(v)$
for each vertex in $v \in V'[k \dots |V'|]$ in sorted order **do**
  $lb(v)\ ,\ ub(v) \leftarrow$ BitExpand($N(v)$)
  if $lb(v) > m$ **then**
    $m \leftarrow lb(v)$
return m

The most frequent operation within the branch and bound search is computing the intersection of the current candidate set with the neighborhood of a chosen vertex. Because this intersection function dominates the runtime, HP-MC attempts to use an appropriate graph representation. While the induced subgraphs are large and sparse, HP-MC uses a CSR graph representation in Algorithm 2. However, dense local induced neighborhoods prompt HP-MC to switch and capture candidate sets in an adjacency matrix in Algorithm 3. HP-MC computes a mapping from each candidate set to the adjacency matrix. This mapping is ideal for dense local neighborhoods due to not requiring a new adjacency matrix at each step of the branch and bound, which would be expensive to build. In addition to the mapping, HP-MC stores a list of each vertex degree with respect to a candidate set. HP-MC uses this list to instantly add easy vertices to the partial clique before branching. If a vertex is entirely adjacent to a candidate set, it is immediately added as all maximal cliques past this point must include it.

Algorithm 2 csrBRB

Require: Graph CSR g, candidate set C, step $step$, current maximum clique size m, partial clique P

for $v \in C$ **do**
  if $deg(v) = C.size() - 1$ **then**
    $P \leftarrow v$
    $C \leftarrow C \setminus v$
$pruned\_v \leftarrow$ true
while $pruned\_v$ **do**
  $pruned\_v \leftarrow$ GreedyColorPrune(C)
$density \leftarrow 2 \times C.edges/(C.size() \times C.size() - 1)$
$pivot \leftarrow \max_{v \in C} \deg(v)$
$component\_list \leftarrow$ CC(C)
$C\_ub \leftarrow color_{ub}(pivot)$
$cc\_skip \leftarrow \varnothing$ // set of speculated skipped vertices
for $cc \in component\_list$ **do**
  if $cc(color) + C\_ub \leq m$ **then**
    Prune cc
  else
    $cc\_skip \leftarrow \max_{v \in cc}\ deg(v)$
for $v \in C \notin cc\_skip$ **do**
  if $density \geq 0.8$ **then**
    matrixBRB(v)
  else
    csrBRB(v)

Algorithm 3 matrixBRB

Require: Adjacency Matrix adj, candidate set C, step $step$, current maximum clique size m, partial clique P

for $v \in C$ **do**
  if $deg(v) = C.size() - 1$ **then**
    $P \leftarrow v$
    $C \leftarrow C \setminus v$
$pruned\_v \leftarrow$ true
while $pruned\_v$ **do**
  $pruned\_v \leftarrow$ GreedyColorPrune(C)
$pivot \leftarrow \max_{v \in C} \deg(v)$
$component\_list \leftarrow$ CC(C)
$C\_ub \leftarrow ub(pivot)$
$cc\_skip \leftarrow \varnothing$ // set of speculated skipped vertices
for $cc \in component\_list$ **do**
  if $cc(color) + C\_ub \leq m$ **then**
    Prune cc
  else
    $cc\_skip \leftarrow \max_{v \in cc}\ deg(v)$
for $v \in C \notin cc\_skip$ **do**
  matrixBRB(v)

A. Coloring

At the beginning of each branch and bound step in all variations, the candidate set is colored using a greedy coloring algorithm. The upper bound for each vertex is then computed in Algorithms 2 and 3 by counting the number of distinct colors a vertex in the candidate set is adjacent to. If the number of distinct colors in combination with the size of the current partial clique is smaller or equal to the current global lower bound, that vertex is pruned from the candidate set. If a vertex has been pruned, the candidate set is recolored, and the vertices' upper bounds are recomputed. We do this iteratively until no more vertices can be pruned as it yields tighter upper bounds for the largest clique found within a candidate set. In large sparse graphs, this is often preferable to coloring a single time as the additional pruning reduces the number of branches by a significant factor.

B. Pivot Vertices and Connected Components

During a step of the branch and bound search in Algorithms 2 and 3, a pivot vertex u can be selected from the induced subgraph of the candidate set. All maximal cliques either contain u or not. Therefore, to find maximal cliques, and thus the maximum clique, a branch and bound search need only branch on the pivot vertex and non-neighbors (but not the pivot's neighbors), reducing the number of recursive calls. The selection of the pivot vertex is done by selecting the vertex with the highest degree within the candidate set to maximize the number of vertices that can be skipped. This heuristic is typically used for enumerating maximal cliques; however, HP-MC optimizes the heuristic for the maximum clique with an important additional step. By considering the candidate set after removing the pivot vertex and all neighbors, the resulting induced subgraph is often quite sparse. This sparsity leads to multiple small connected components. Based on the well-known coloring bound, a clique can contain at most one vertex per color, since same-colored vertices are never adjacent. Hence, the number of colors is an upper bound on the clique size (as used in PMC, MC-BRB, and Lazy-MC). This bound holds for any subgraph. So, for a set of disjoint subgraphs, the sum of their number of colors forms an upper bound for the union of those subgraphs. This allows entire connected components to be pruned.

Let u be the pivot vertex and C be a connected component of the disjoint subgraph induced by candidate vertices not adjacent to u. If $colors(N(u)) + colors(C) + |P| < LB$, where $|P|$ is the size of the current partial clique and LB the lower bound, then C can safely be pruned.

Additionally, despite the connected components being comprised of all non-neighbors of the pivot vertex, exhausting branches of all vertices within them is not required. We show this by deriving rules based on the one-neighbor-missing (ONM) reduction rule presented in MC-BRB [5].

If a vertex v is adjacent to all but one vertex v' of the candidate set, no larger clique can be found by branching on v' than on v.

Let S be the induced subgraph of candidate vertices not adjacent to the pivot vertex u. For a clique not containing u to exceed the current lower bound, at least two vertices of S must belong to it (per the ONM rule). Since any clique members within S are pairwise adjacent, they must belong to the same CC of S. Hence, it suffices to branch on u and, for each connected component (CC), to branch on all but one vertex. We can safely skip an arbitrary vertex per CC because, even if we skip one of the vertices belonging to the maximum clique, we will eventually branch from the other of the at least two vertices.

At a minimum, there is one vertex in every connected component that we do not have to branch from to find the maximum clique. HP-MC takes advantage of this and skips the first vertex appearing in the candidate set's order of each connected component, which, due to static resorting of the graph, are the highest degree vertices. It is important to note that these vertices are still included in subsequent candidate sets and may be considered in later branches.

This is the main method used in HP-MC for reducing the number of branches required to find the maximum clique, which is dependent on the effectiveness of finding upper bounds with the recoloring at the beginning of each branch and bound step. The smaller the number of colors within each connected component, and the smaller the upper bound found with the pivot vertex, the more effective our pruning is. This is also true for the number of skipped vertices with the lower bound found using the pivot vertex.

C. Parallelization

Our goal with HP-MC is to maximize parallelism and load balance across the entire computation. MCP implementations tend to exhibit highly unbalanced branching behavior, where different branches of the search tree vary greatly in size and depth. Static work distribution therefore leads to poor load balancing, with threads working on a small portion of vertices for most of the runtime. Parallelization of our initial global lower bound computation is trivial as the neighborhood of each vertex can be evaluated independently across different starting vertices. This requires no communication or synchronization between threads until a final reduction step to determine the best bound. The heuristic runs in $O(n^2)$ time and completes very quickly in practice. Particularly on difficult instances, where the branch and bound search dominates the runtime, this cost is negligible. To address these issues on the branch and bound search, we adopt a work-stealing-based approach, allowing our code to dynamically distribute work across threads. Each recursive expansion of the search tree is expressed as an independent OpenMP task. When the size of a candidate set exceeds a predefined threshold, new tasks are spawned for each branch of the recursion. For smaller subproblems, the thread completes that branch on its own to limit task creation overhead. The use of tasks enables dynamic load balancing, as idle threads can pick up tasks from other non-idle threads by observing their state at a particular level of their recursive branch. Due to the nondeterministic nature of OpenMP task

scheduling, threads may execute tasks in any order. The underlying matrices (that store mappings to candidate sets) and adjacency information of vertices are overwritten when threads backtrack. As a result, without additional synchronization or the expensive storing of all state whenever a task is created, it is impossible to guarantee that a previously constructed state remains unchanged. To address this issue, for a particular depth of the search tree, all child tasks must finish before a thread may backtrack beyond this point.

D. Dense Solver

When the induced subgraph of a candidate set found in the branch and bound search is sufficiently dense, HP-MC switches to Algorithm 3. However, as mentioned, the new order in which vertices are branched on as starting vertices is in ascending order of local upper bounds. Generally, this means the difficulty, and the number of branches, increases exponentially as HP-MC traverses this list. The final group of vertices are, thus, the most difficult as their neighborhoods result in the most dense induced subgraphs. As a result, instead of switching when the neighborhood becomes dense, once the list of remaining vertices can reasonably fit within an adjacency matrix, HP-MC sorts and relabels the remaining vertices based on their degeneracy ordering. This adjacency matrix as used in Algorithm 4 is a bitset representation of the remaining vertices. This graph representation allows HP-MC to utilize fast bit-wise operations in its intersection function.

Algorithm 4 BitExpand

Require: Bitset Adjacency Matrix adj, candidate set C, step $step$, current maximum clique size m, partial clique P

1: **if** $|P| > m$ **then**
2: $m \leftarrow |P|$
3: **if** $C.size() + P.size() \leq m$ **then**
4: **return** m
5: $Order \leftarrow$ BitGreedyColor(C)
6: $deg = dv_{Order.size()}$
7: **if** $deg = Order.size() - 1$ **then**
8: $P \leftarrow v_{Order.size()}$
9: $C \leftarrow C \setminus v_{Order.size()}$
10: expand(C)
11: **else**
12: **for** $v \in C$ **do**
13: **if** $color(v) + P.size() \leq m$ **then**
14: **return** m
15: $P \leftarrow v$
16: $new\_C \leftarrow N(v) \cap C$
17: $m \leftarrow$ BitExpand(new_C)
18: $P \leftarrow P \setminus v$
19: **return** m

As shown in Algorithm 4, coloring is no longer placed in its own loop for pruning vertices. This is due to the assumption that, for sufficiently dense candidate sets, it is rare for vertices to be pruned based on distinct colors of adjacent vertices alone. Instead, the coloring of a candidate set within the dense solver is meant to sort the candidate set into a new order, in which high-color vertices appear first. This function also computes the degree of each vertex. The vertex at the end of this order would have the highest degeneracy and is the most connected, and if its degree is the entire candidate set, HP-MC immediately branches on this vertex.

Within this dense solver, HP-MC no longer uses the pivot vertex heuristic. This is because one of the strongest benefits of reducing candidate sets by pruning entire connected components is outweighed by the benefit of stronger pruning in these dense neighborhoods. As a result, HP-MC's dense solver utilizes a coloring-based selection strategy. As shown in Algorithm 4, we restrict the search to vertices whose color exceeds a given threshold at each step, based on the current partial clique size.

V. Evaluation Methodology

We compare the performance of our parallel CPU implementation of HP-MC with the parallel CPU implementations of PMC [14], MC-BRB [5], and Lazy-MC [22]. These are the fastest publicly available implementations that focus on large sparse inputs. There are more approaches in the literature such as MCQ [20], which is an older version of MCS [21] and BBMC [17]. However, these approaches have either been shown to be slower or are not publicly available. We do not compare to approaches such as CliSAT [16] and MaxCliquePara [6] as they are specifically designed for small dense inputs, which are not the target of our work. All approaches find the exact maximum clique size of the graph and display the final size as output. All codes arrive at the same final size, which is listed in the column labeled $w(G)$ in Table I.

We use the 21 input graphs listed in Table I. These inputs were chosen for their various type, size, average degree, maximum degree, and maximum clique size $w(G)$. They stem from the Galois framework [9], Stanford Network Analysis Platform [13], SuiteSparse Matrix Collection [18], and the Center for Discrete Mathematics and Theoretical Computer Science at the University of Rome [7].

We evaluated all codes on two systems, labeled System A and System B. System A is a shared-memory system with an AMD Ryzen Threadripper 3970X 32-core processor. System B is a shared-memory system with an Intel Xeon Gold 6226R 32-core processor. In both systems, we compiled the codes using g++ version 13.3.0, which supports OpenMP 4.5. Each code is run 9 times on each input, and the median runtime is reported. This is done to account for variation in performance on a single code. The runtimes are limited to an hour per input. Performance is shown in terms of throughput, which is the number of vertices in the graph divided by the runtime. Throughput is a higher-is-better metric. Lazy-MC was configured with the environment variable GRAPTOR_PARALLEL=parlay and executed using 64 Parlay threads. PMC was compiled with the flags -O3 -fPIC -fopenmp and executed using 64 OpenMP threads. MC-BRB was compiled with -O3 -flto -std=c++11. HP-MC was compiled with the flags -O3 -fopenmp and executed using 64 OpenMP threads.

TABLE I
INFORMATION ABOUT THE SPARSE INPUT GRAPHS

type	name	vertices	edges	d_min	d_max	w(G)
product co-purchases	amazon0601	403,394	4,886,816	1	2,752	11
Internet topology	as-skitter	1,696,415	22,190,596	1	35,455	67
publication citations	citationCiteseer	268,495	2,313,294	1	1,318	13
patent citations	cit-Patents	3,774,768	33,037,894	1	793	11
publication citations	coPapersDBLP	540,486	30,491,458	1	3,299	337
triangulation	delaunay_n24	16,777,216	100,663,202	3	26	4
road map	europe_osm	50,912,018	108,109,320	1	13	4
web links	in-2004	1,382,908	27,182,946	0	21,869	489
Internet topology	internet	124,651	387,240	1	151	7
Kronecker	kron_g500-logn16	65,536	4,912,142	0	17,997	136
Kronecker	kron_g500-logn17	131,072	10,227,970	0	29,935	157
Kronecker	kron_g500-logn18	262,144	21,165,372	0	49,162	179
Kronecker	kron_g500-logn19	524,288	43,561,574	0	80,674	207
random	r4-2e23.sym	8,388,608	67,108,846	2	26	3
random	rgg_n_2_24_s0	16,777,216	265,114,400	0	40	21
RMAT	rmat20.sym	1,048,576	16,224,434	0	2,027	18
RMAT	rmat22.sym	4,194,304	65,660,814	0	3,687	20
journal community	soc-LiveJournal1	4,847,571	85,702,474	0	20,333	321
web links	uk-2002	18,520,486	523,574,516	0	194,955	944
road map	USA-road-d.CTR	14,081,816	33,866,826	1	8	4
road map	USA-road-d.USA	23,947,347	57,708,624	1	9	4

VI. RESULTS

In this section, we compare the performance of HP-MC with the state of the art across different graph topologies. Then, we separately evaluate the performance impact of various features in HP-MC on selected inputs.

TABLE II
MEASURED RUNTIMES IN SECONDS ON SYSTEM A

	PMC	MC-BRB	Lazy-MC	HP-MC
amazon0601	0.062	0.053	0.031	0.001
as-skitter	0.525	0.200	0.159	0.101
citationCiteseer	0.062	0.040	0.028	0.011
cit-Patents	1.928	1.493	0.201	0.035
coPapersDBLP	0.138	0.052	0.099	0.014
delaunay_n24	9.926	4.210	1.356	0.188
europe_osm	3.240	4.719	1.076	0.025
in-2004	0.157	0.066	0.114	0.061
internet	0.027	0.006	0.010	0.013
kron_g500-logn16	0.741	2.012	0.565	0.891
kron_g500-logn17	2.609	6.021	3.665	1.421
kron_g500-logn18	44.043	42.128	31.463	5.901
kron_g500-logn19	3340.880	805.171	302.588	275.596
r4-2e23.sym	9.303	15.987	0.847	0.245
rgg_n_2_24_s0	9.565	1.522	0.774	0.005
rmat20.sym	1.162	0.578	0.128	0.271
rmat22.sym	7.837	3.951	0.468	1.080
soc-LiveJournal1	2.577	1.313	0.444	0.061
uk-2002	6.203	1.625	0.814	0.081
USA-road-d.CTR	3.629	0.896	0.463	0.018
USA-road-d.USA	5.913	1.218	0.710	0.031

A. Performance Comparison

Figure 2 shows throughput in millions of vertices per seconds of our parallel CPU code alongside that of PMC, MC-BRB, and Lazy-MC. Note that the y-axis uses a logarithmic scale. The x-axis lists the input graphs and the geometric mean. The measured runtimes are also listed in Tables II and III.

On average, HP-MC is nearly an order of magnitude faster than the other three leading implementations on these tested

TABLE III
MEASURED RUNTIMES IN SECONDS ON SYSTEM B

	pmc	MC-BRB	Lazy-MC	HP-MC
amazon0601	0.084	0.056	0.039	0.001
as-skitter	0.679	0.235	0.205	0.204
citationCiteseer	0.088	0.048	0.030	0.008
cit-Patents	3.022	2.031	0.234	0.042
coPapersDBLP	0.175	0.065	0.122	0.014
delaunay_n24	13.006	5.303	2.933	0.181
europe_osm	5.106	5.661	1.441	0.014
in-2004	0.195	0.076	0.205	0.072
internet	0.035	0.013	0.015	0.007
kron_g500-logn16	0.983	2.364	0.719	1.369
kron_g500-logn17	3.155	7.265	4.986	2.369
kron_g500-logn18	76.276	52.205	43.136	7.880
kron_g500-logn19	3554.370	1006.298	334.650	354.755
r4-2e23.sym	13.133	20.052	1.446	0.264
rgg_n_2_24_s0	16.635	1.656	1.231	0.007
rmat20.sym	1.607	0.822	0.143	0.466
rmat22.sym	11.564	5.278	0.602	1.776
soc-LiveJournal1	4.234	1.687	0.501	0.115
uk-2002	8.693	1.801	1.090	0.105
USA-road-d.CTR	5.176	1.075	0.616	0.007
USA-road-d.USA	7.526	1.357	1.023	0.012

inputs. On System A, based on the geometric-mean throughput, HP-MC is 19.25× faster than PMC, 10.43× faster than MC-BRB, and 4.56× faster than Lazy-MC. HP-MC performs the best on rgg_n_2_24_s0, with speedups of 1771.21×, 281.84×, and 143.39× compared to PMC, MC-BRB, and Lazy-MC, respectively. The slowest relative runtimes are on the internet input with a 0.51× speedup compared to MC-BRB, the rmat22.sym input with a 0.43× speedup compared to Lazy-MC, and the kron_g500-logn16 input with a 0.83× speedup compared to PMC. HP-MC is not the best performing implementation on 4 of the 21 tested inputs, namely internet, kron_g500-logn16, rmat20.sym, and rmat22.sym. These graphs are not unique in their size, minimum, or maximum degrees compared to other inputs tested. This is because

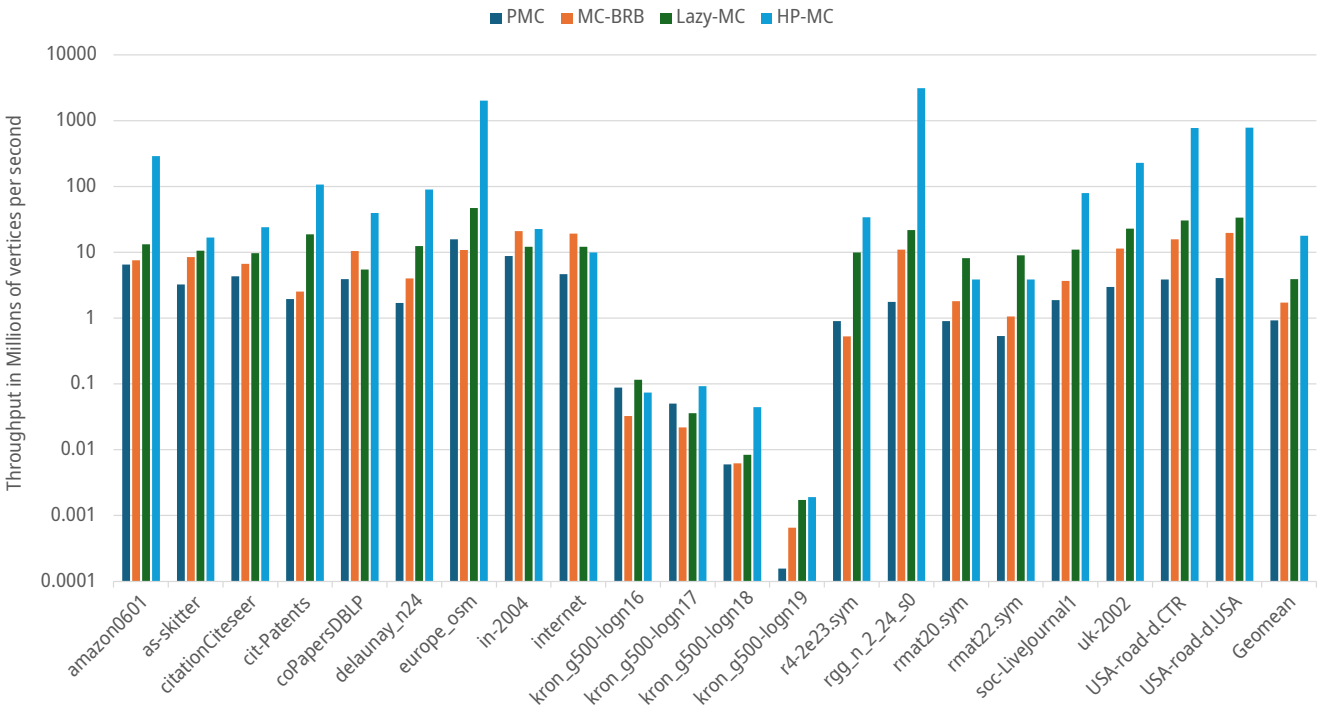

Fig. 2. Throughput of codes on 21 sparse inputs on System A

performance is dominated by search-space shape rather than global metrics of each graph. Local topology, such as density of small candidate sets, or induced subgraphs, and bound effectiveness of graph coloring have a much larger impact on the branch and bound search. Inputs with a small clique-core gap are also impacted heavily with the tightness of the initial lower bound estimation. This is why inputs such as rgg_n_2_24_s0 have some of the largest speedups, ranging from 143.39× compared to Lazy-MC and 1771.21× compared to PMC. Our initial lower bound computation is very fast relative to the other codes, finds the maximum clique, and is equal to the upper bound found with graph coloring. Inputs such as kron_g500-logn18 do not benefit from the same tightness in lower bounds, but the search space collapses due to repeated coloring of each candidate set, resulting in speedups ranging from a minimum of 5.33× compared to Lazy-MC to a maximum of 7.46× compared to PMC. However, on instances such as kron_g500-logn16, these optimizations introduce considerable overhead, leading to better performance with codes that do not repeatedly color the candidate set such as PMC.

On System B, HP-MC is 25.21× faster than PMC, 12.20× faster than MC-BRB, and 5.91× faster than Lazy-MC based on the geometric-mean throughput. The overall trends are still the same, other than no longer yielding a slowdown on the internet input and experiencing a slowdown on kron_g500-logn19 compared to Lazy-MC. The largest speedups are 2410.87×, 419.35×, and 178.34× compared to PMC, MC-BRB, and Lazy-MC, respectively.

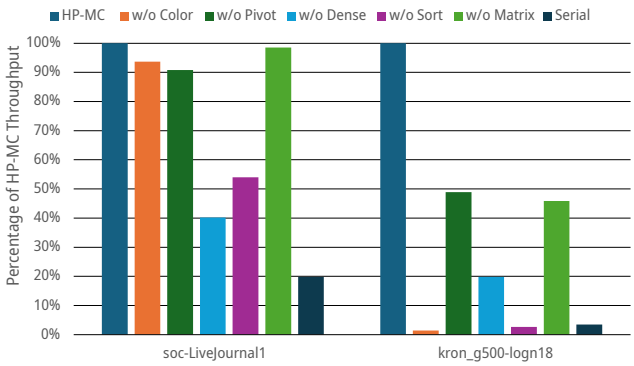

Fig. 3. Percentage of throughput of various HP-MC versions

TABLE IV
RUNTIMES IN SECONDS OF VARIOUS HP-MC VERSIONS

	soc-LiveJournal1	kron_g500-logn18
HP-MC	0.061	5.901
w/o Color	0.065	411.217
w/o Pivot	0.067	12.059
w/o Dense	0.152	29.765
w/o Sort	0.113	222.418
w/o Matrix	0.062	12.860
Serial	0.308	169.513

B. Optimizations

HP-MC's performance is due to a combination of factors. To evaluate the benefit of various factors, Figure 3 shows the percentage of throughput of HP-MC on System A after disabling

individual optimizations. In particular, we disable the iterative recoloring of each candidate set until no more vertices can be pruned (w/o Color), the pivot vertex selection and processing (w/o Pivot), the switching to a binary matrix representation and a dense solver that does not use the pivot heuristic (w/o Dense), sorting and relabeling the graph at the beginning of the computation (w/o Sort), switching to a matrix representation with the pivot heuristic (w/o Matrix), and the parallelization of HP-MC to evaluate its serial performance without work stealing (Serial). Table IV presents the corresponding runtimes.

Since the purpose is to assess the contribution of various optimizations, we only show results for two inputs where all optimizations matter. Most of the other inputs are solved during the greedy clique construction, as the maximum clique size is equal to the upper bound computed with graph coloring or the core computation. Hence, these other inputs do not invoke all of the optimizations.

Based on the geometric-mean throughput, disabling repeated coloring to prune the maximum number of vertices per step slows execution down by 8.33×. Not using any pivot vertices results in HP-MC running 1.49× slower. Disabling the dense solver and, therefore, exhausting all branches using the compressed-sparse row and matrix branch and bound yields 3.57× longer runtimes. Not sorting and not relabeling the graph make HP-MC take 8.33× longer. Disabling the conversion to an adjacency matrix representation yields 1.49× longer runtimes. Forcing the code to run serially incurs the largest geometric-mean slowdown of 12.5×.

Coloring candidate sets has the largest performance implications for difficult inputs. The general trend for other optimizations such as the pivot vertex heuristic and the addition of the dense solver is similar, that is, their impact increases with the difficulty of the input. By "difficult", we mean inputs with larger clique-core gaps such as kron_g500-logn18 that are locally very dense in comparison to other inputs. On kron_g500-logn18, for example, disabling the pivot vertex heuristic results in 48% of the original throughput, and disabling the dense solver results in 20% of the original throughput. Without sorting, HP-MC performs at 3% of its original speed at the worst, and without converting to an adjacency matrix representation, it performs at 46% of its original speed.

These results suggest there is no single optimization that accounts for most of HP-MC's performance. However, the largest contribution comes from HP-MC's novel iterative coloring and parallelization strategies, especially on difficult inputs such as kron_g500-logn18. Without iteratively coloring a candidate set, the pivot vertex heuristic becomes less effective at combinatorially reducing the number of branches.

C. Scalability

Figure 4 shows the geometric mean of HP-MC's throughput over all inputs except one for different thread counts. We excluded kron_g500-logn19 as it exceeds the timeout of one hour for low thread counts. The x-axis lists the number of threads and the y-axis the throughput on a linear scale.

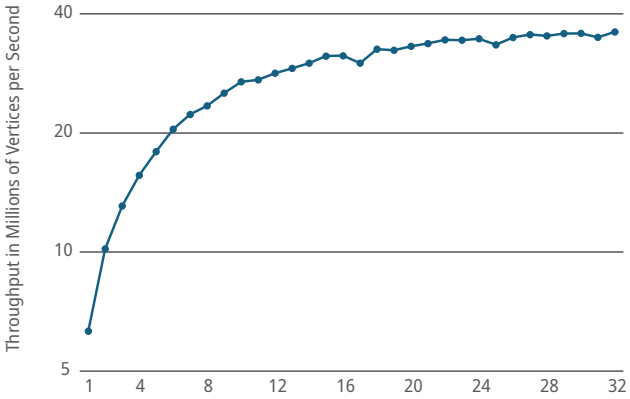

Fig. 4. HP-MC throughput for different thread counts

Performance increases up to 32 threads, that is, the number of physical cores in our system. Hyperthreading does not help on this irregular computation. When considering all inputs, including small graphs where parallelization is less effective, the geometric-mean speedup is 5.71. On the largest graphs, the geometric-mean speedup increases to 17.31. This coincides with what we expect for inputs in which our new optimizations are not critical. As mentioned, some inputs complete quickly after the initial lower-bound estimation, as there are few vertices left with relatively small branch and bound search spaces. In these cases, task generation introduces considerable overhead, reducing the performance gained from parallelization. On more difficult inputs such as kron_g500-logn18, the speedup grows to 22.66. Additionally, we see small dips in performance as the thread count increases. They are the result of nondeterministic behavior related to the OpenMP tasking system.

D. Preprocessing

The runtimes in Tables II and III are search times and do not include the I/O cost of reading the graph from disk into memory nor the subsequent preprocessing step in which the graph is sorted by vertex degree. The reason for this is twofold. First, the disk I/O and reordering are separate steps that execute prior to the search. Second, and more importantly, excluding these times from all algorithms facilitates comparing the actual branch-and-bound search efficiency.

Figure 5 shows the preprocessing times. The y-axis lists the runtime in seconds. The x-axis lists the inputs sorted in ascending order of the number of edges in the graph. We see a consistent trend in which larger graphs cause longer preprocessing times, sometimes taking longer than the search itself. This is in line with the other tested codes and shows how separate the preprocessing is from the cost of the search. In fact, many of the graphs that take a long time in terms of preprocessing happen to have some of the shortest search times. Hence, the duration of the preprocessing step tells us nothing about the clique-core gap of the graph or how long the search will take, which is why we show these times separately.

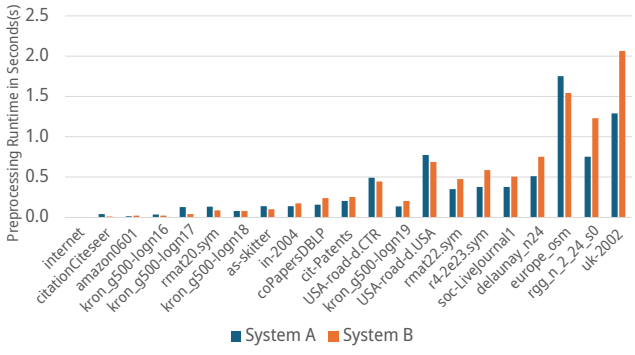

Fig. 5. Preprocessing times for each input on both systems

VII. Conclusion

The maximum clique problem is a classical NP-hard problem with broad applications in multiple domains. Its computational difficulty has led to a wide range of exact algorithms that take advantage of sophisticated pruning and branching strategies. Our approach, called HP-MC, includes several novelties to boost performance. For example, it improves efficiency by using a hybrid strategy that utilizes a pivot vertex heuristic while candidate sets remain large and sparse but switches to a matrix and dense solver when the candidate set shrinks and becomes dense. This is aided by iterative coloring of each candidate set to prune as many vertices as possible. HP-MC also removes entire connected components of vertices that are not adjacent to the pivot vertex and cannot build larger cliques.

Parallel efficiency is normally limited by search-tree imbalance, with a small number of "heavy" branches that often dominate runtime. HP-MC encourages parallelism by utilizing work stealing near the top of the search tree to address high-cost branching vertices and improve load balance.

In addition to overall performance, we evaluated the contribution of individual features of HP-MC by disabling various optimizations. These experiments show that the observed performance gains are not due to a single optimization but rather the combined effect across multiple design choices. Our parallel OpenMP implementation of HP-MC incorporates these state-of-the-art optimizations. The experimental results demonstrate that our approach generally outperforms the leading implementations from the literature. Comparing HP-MC to the state of the art on various sparse graphs from different domains, we find that it is never more than 2.3 times slower, delivers a geometric-mean speedup of 9.7, and can be 3 orders of magnitude faster, reaching speedups of over 1771.2.

Acknowledgment

We thank the reviewers for their valuable feedback. This work has been supported by the U.S. Department of Education under a Graduate Assistance in Areas of National Need (GAANN) award.

References

[1] Ghadeer Alabandi and Martin Burtscher. Improving the speed and quality of parallel graph coloring. *ACM Trans. Parallel Comput.*, 9(3), August 2022.

[2] Nina Berry, Teresa Ko, Tim Moy, Julienne Smrcka, Jessica Turnley, and Ben Wu. Emergent clique formation in terrorist recruitment, 01 2016.

[3] Coen Bron and Joep Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Commun. ACM*, 16(9):575–577, September 1973.

[4] Gerard J. Chang, Lingling Huang, and Xuding Zhu. Circular chromatic numbers of mycielski's graphs. *Discrete Mathematics*, 205(1):23–37, 1999.

[5] Lijun Chang. Efficient maximum clique computation over large sparse graphs. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '19, page 529–538, New York, NY, USA, 2019. Association for Computing Machinery.

[6] Matjaž Depolli, Janez Konc, Kati Rozman, Roman Trobec, and Dušanka Janežič. Exact parallel maximum clique algorithm for general and protein graphs. *Journal of Chemical Information and Modeling*, 53(9):2217–2228, 2013. PMID: 23965016.

[7] DIMACS. Center for discrete mathematics and theoretical computer science at the university of rome (dimacs). https://www.diag.uniroma1.it/challenge9/download.shtml, 2006. Accessed: 2025-04-30.

[8] John D. Eblen, Charles A. Phillips, Gary L. Rogers, and Michael A. Langston. The maximum clique enumeration problem: algorithms, applications, and implementations. *BMC Bioinformatics*, 13:S5 – S5, 2011.

[9] Galois. https://iss.oden.utexas.edu/?p=projects/galois, 2018.

[10] Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972.

[11] Ailsa H. Land and Alison G. Doig. *An Automatic Method for Solving Discrete Programming Problems*, pages 105–132. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[12] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker graphs: An approach to modeling networks. *Journal of Machine Learning Research*, 11(33):985–1042, 2010.

[13] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[14] Ryan A. Rossi, David F. Gleich, Assefaw H. Gebremedhin, and Md. Mostofa Ali Patwary. Parallel maximum clique algorithms with applications to network analysis and storage. *arXiv preprint arXiv:1302.6256*, 2013.

[15] Kati Rozman, An Ghysels, Bogdan Zavalnij, Tanja Kunej, Urban Bren, Dušanka Janežič, and Janez Konc. Enhanced molecular docking: Novel algorithm for identifying highest weight k-cliques in weighted general and protein-ligand graphs. *Journal of Molecular Structure*, 1304:137639, 2024.

[16] Pablo San Segundo, Fabio Furini, David Álvarez, and Panos M. Pardalos. Clisat: A new exact algorithm for hard maximum clique problems. *European Journal of Operational Research*, 307(3):1008–1025, 2023.

[17] Pablo San Segundo, Fernando Matia, Diego Rodriguez-Losada, and Miguel Hernando. An improved bit parallel exact maximum clique algorithm. *Optimization Letters*, 7(3):467–479, 2013.

[18] SuiteSparse Matrix Collection. https://sparse.tamu.edu/, 2019. Accessed: 2025-04-30.

[19] Etsuji Tomita and Toshikatsu Kameda. An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments. *Journal of Global Optimization*, 37(1):95–111, 2007.

[20] Etsuji Tomita and Tomokazu Seki. An efficient branch-and-bound algorithm for finding a maximum clique. In Cristian S. Calude, Michael J. Dinneen, and Vincent Vajnovszki, editors, *Discrete Mathematics and Theoretical Computer Science*, pages 278–289, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[21] Etsuji Tomita, Yoichi Sutani, Takanori Higashi, Shinya Takahashi, and Mitsuo Wakatsuki. A simple and faster branch-and-bound algorithm for finding a maximum clique. In Md. Saidur Rahman and Satoshi Fujita, editors, *WALCOM: Algorithms and Computation*, pages 191–203, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[22] Hans Vandierendonck. Less is more: Faster maximum clique search by work-avoidance. *arXiv preprint arXiv:2509.22245*, 2025.

[23] Jose L. Walteros and Austin Buchanan. Why is maximum clique often easy in practice? *Oper. Res.*, 68(6):1866–1895, November 2020.