Manycore Clique Enumeration with Fast Set Intersections

Jovan Blanuša[†‡], Radu Stoica[†], Paolo Ienne[‡], and Kubilay Atasu[†]
[†]IBM Research Europe, Zurich [‡]Ecole Polytechnique Fédérale de Lausanne (EPFL)
jov@zurich.ibm.com, rst@zurich.ibm.com, paolo.ienne@epfl.ch, kat@zurich.ibm.com

ABSTRACT

Listing all maximal cliques of a given graph has important applications in the analysis of social and biological networks. Parallelisation of maximal clique enumeration (MCE) algorithms on modern manycore processors is challenging due to the task-level parallelism that unfolds dynamically. Moreover, the execution time of such algorithms is known to be dominated by intersections between dynamically-created vertex sets. In this paper, we prove that the use of hash-join-based set-intersection algorithms within MCE leads to Pareto-optimal implementations in terms of runtime and memory space compared to those based on merge joins. Building on this theoretical result, we develop a scalable parallel implementation of MCE that exploits both data parallelism, by using SIMD-accelerated hash-join-based set intersections, and task parallelism, by using a shared-memory parallel processing framework that supports dynamic load balancing. Overall, our implementation is an order of magnitude faster than a state-of-the-art manycore MCE algorithm. We also show that a careful scheduling of the execution of the tasks leads to a two orders of magnitude reduction of the peak dynamic memory usage. In practice, we can execute MCE on graphs with tens of millions of vertices and up to two billion edges in just a few minutes on a single CPU.

PVLDB Reference Format:
Jovan Blanuša, Radu Stoica, Paolo Ienne, Kubilay Atasu. Manycore Clique Enumeration with Fast Set Intersections. *PVLDB*, 13(11): 2676-2690, 2020.
DOI: https://doi.org/10.14778/3407790.3407853

1. INTRODUCTION

Subgraph patterns in graph datasets, such as communities, clusters, and motifs, are fundamental concepts used in a wide range of graph mining applications [2]. However, extracting such patterns often requires executing costly recursive algorithms that lead to exploration of an exponentially-large search space and long execution times. In this paper, we focus on maximal clique enumeration (MCE), the graph mining problem of identifying all maximal cliques (i.e., maximal complete subgraphs) of a graph. MCE serves as a building block of many graph mining applications, such as detection of communities in social networks [41], prediction of protein functions in protein interaction networks [62, 63], and prediction of how epidemics spread [20]. In addition, MCE is similar, in terms of its computational patterns—e.g., vertex-set intersections [30] and task parallelism that unfolds dynamically, to other important graph mining algorithms, such as subgraph isomorphism [19, 31], frequent pattern discovery [25, 60], and maximum clique finding [47].

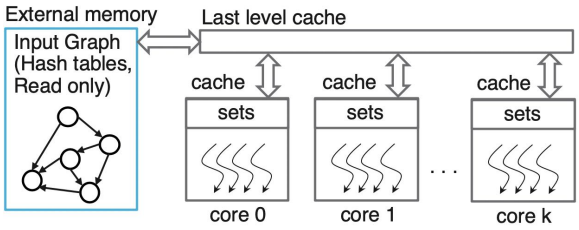


Figure 1: Illustration of our manycore setup: each core can execute several concurrent threads and has a private cache in which the sets it creates can reside. The input graph is stored in external memory in the form of hash tables.

We focus on the Bron-Kerbosch (BK) algorithm with degeneracy ordering [15, 26, 55], which is one of the most efficient and widely-used algorithms for MCE [16, 18]. It is a recursive algorithm that explores an exponentially-large solution space. In order to enable solutions to real-life problems, it is necessary to develop scalable implementations of this algorithm. A large amount of task-parallelism is available in the BK algorithm because different regions of the search space can be traversed independently. However, parallelisation of the algorithm on modern computer architectures, such as manycore CPUs (see Figure 1), is not straightforward. Because a recursion tree is constructed dynamically and its shape is not known in advance, distributing the work evenly across processing resources poses challenges.

Prior work by Han et al. [30] showed that set intersections dominate the execution time of the BK algorithm. Set intersections are special cases of join operations because sets store only keys and no payloads. When implementing set intersections, one can rely on the two main classes of join algorithms: merge joins and hash joins. Merge joins require both sets to be sorted while hash joins require at least one of the sets to be hashed. In this paper, we analyze how intersections based on merge joins and hash joins affect the overall time and space complexity of the BK algorithm.

Proceedings of the VLDB Endowment, Vol. 13, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3407790.3407853

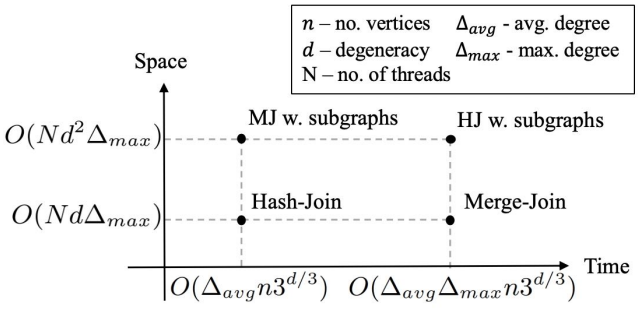


Figure 2: Effect of different set-intersection methods on the time and space complexity of the BK algorithm.

Intersections in the BK algorithm are performed between the adjacency lists of the input graph and some dynamically-created sets. Because the adjacency lists are static, they can be hashed or sorted in advance. As a result, the complexity of set intersections based on hash joins does not depend on the size of the adjacency lists, which is not the case for set intersections based on merge joins. Considering that the adjacency lists are asymptotically larger than the dynamically created sets, using hash-join-based intersections leads to faster BK implementations. We show this result to be valid both in theory and in practice. Yet, performance of merge-join-based approaches can be improved using the modified BK algorithm of Eppstein et al. [26]. We show that creating subgraphs at each recursive call, as proposed by Eppstein et al. [26], shrinks the adjacency lists used in the intersections, leading to faster merge-join-based solutions, but at the cost of an increased space complexity. Our theoretical analysis results given in Figure 2 show that merge-join-based solutions require either more space or more time compared to hash-join-based solution. The results given in Figure 2 motivate us to focus on BK implementations that use hash-join-based set intersections.

Our manycore implementation exploits both data- and task-level parallelism of the BK algorithm. We scale the algorithm across multiple hardware threads using a framework that utilizes dynamic load-balancing and we exploit data-level parallelism using SIMD-accelerated hash-join-based set intersections. In our manycore setup, the input graph resides in external memory and the adjacency list of each vertex is stored as a read-only hash table (Figure 1). Hardware threads perform intersections between adjacency lists and local sets, dynamically creating other sets as results. Our intelligent recursion tree exploration approach guarantees that the dynamic memory usage increases only linearly with the number of threads, independently of the number of graph vertices. Thus, it is possible to fit the dynamically-created data structures in the cache space of the CPUs. Lastly, we minimise the task and memory management overheads, which can also constitute a significant part of the execution time. As a result, we are able to run the BK algorithm on a graph with more than 60 million vertices and 1.8 billion edges on a single manycore CPU in only a few minutes.

The remainder of the text is organised as follows. Section 2 introduces the BK algorithm. Section 3 offers a broad complexity analysis of the improved BK algorithm of Eppstein et al. [26], mainly focusing on the impact of the set-intersection algorithms. Section 4 describes a practical SIMD-accelerated hash-join-based set-intersection implementation. Section 5 describes our scalable manycore implementation of the BK algorithm. Finally, Section 6 shows our experimental results which culminate in an order of magnitude performance improvement with respect to a state-of-the-art manycore MCE implementation.

Algorithm 1: Bron-Kerbosch w. pivoting

```
1  Function BKPivot (Sets R, P, X, graph G)
2      if P = ∅ then
3          if X = ∅ then Report R as a maximal clique;
4          return;
5      pivot = getPivot(P, X, G);
6      foreach vertex v : P/N_G(pivot) do
7          P' = P ∩ N_G(v); X' = X ∩ N_G(v);
8          BKPivot(R + {v}, P', X',G);
9          P = P − {v}; X = X + {v};
10 Function getPivot (Set P, Set X, graph G)
11     foreach vertex v : P ∪ X do
12         t_v = |P ∩ N_G(v)|;
13     return arg max_v(t_v);
```

2. BRON-KERBOSCH ALGORITHM

One of the most successful algorithms for listing all maximal cliques of a graph is the Bron-Kerbosch (BK) algorithm [15]. It is a recursive algorithm that operates on three sets of vertices during each recursive call: set R stores the vertices that form the currently largest clique; set P the candidate vertices that may form a clique with the ones from R; and set X the vertices that have already been considered and, therefore, cannot take part in new cliques. The vertices of P and X have to be adjacent to the vertices of R at each step, which is ensured by set intersection. At each recursive call, a vertex v from the set of possible vertices P is added to R, such that R still forms a clique. We then recursively determine whether the extended set R is part of a larger clique or not. After all the cliques that contain vertex v have been enumerated, the vertex is moved to the set X. If sets P and X are both empty, R is reported as a maximal clique. At the beginning, sets R and X are empty and set P contains all the vertices of the input graph.

To reduce the number of unnecessary recursive calls, Bron and Kerbosch introduced a pivoting strategy. When expanding R, instead of considering all vertices of set P, a pivot vertex from P is chosen, and vertices neighboring the pivot vertex $N_G(pivot)$ are not considered for expansion. This approach was improved by Tomita et al. [55] by choosing the pivot vertex from $P \cup X$ in a way that maximizes the number of vertices from $N_G(pivot)$ that are excluded from expansion (i.e., $|P \cap N_G(pivot)|$ is maximized). The BK algorithm with the pivoting strategy is shown in Algorithm 1.

Algorithm 2: Bron-Kerbosch w. degeneracy order

```
1 Function BKDegeneracy (Graph G(V, E)) is
2     Order vertices in G using degeneracy ordering;
3     foreach vertex v_i : V do
4         P = N_G(v_i) ∩ {v_{i+1}, v_{i+2}, ..., v_{n−1}};
5         X = N_G(v_i) ∩ {v_0, v_1, ..., v_{i−1}};
6         BKPivot({v_i}, P, X, G);
```

Table 1: Worst-case time complexity when using different vertex ordering strategies. SFG stands for scale-free graphs.

Vertex ordering	Time complexity
Arbitrary [55]	$O(3^{n/3})$
Degree [61]	$O(hn3^{h/3})$
Degeneracy [26]	$O(dn3^{d/3})$
Arbitrary (this work)	$O(n3^{\Delta_{max}/3})$
Arbitrary - SFG (this work)	$O(3^{\Delta_{max}/3})$
Degeneracy (this work)	$O(\Delta_{avg}n3^{d/3})$

Eppstein et al. [26] improved the original BK algorithm using the degeneracy ordering. The degeneracy represents the smallest value d, such that each nonempty subgraph of a graph has a vertex with at most d edges. If a graph has degeneracy d, its largest clique can have at most $d+1$ vertices, and the vertices of the graph can be ordered in such a way that each vertex has d or less neighboring vertices that come later in the ordering. The main idea used in the improved algorithm is to compute a degeneracy ordering of the vertices before invoking the original BK algorithm with pivoting, as shown in Algorithm 2. For each vertex v_i, sets P and X represent its neighbors that come later and before in the ordering, respectively. The vertex itself is assigned to set R and, for the sets created as described, the original algorithm with pivoting is invoked. Every set P is going to have at most d vertices due to degeneracy ordering, so the recursion tree depth is bounded by d. This property can be used to reduce the exponential worst-case time complexity.

In summary, Tomita et al. [55] assumed an arbitrary ordering of vertices when building the recursion tree and obtained a worst-case complexity of $O(3^{n/3})$, where n is the number of graph nodes. On the other hand, Eppstein et al. [26] proved a worst-case complexity bound of $O(dn3^{d/3})$, where d is the degeneracy of the input graph. Similarly, Xu et al. [61] focused on the degree-ordering of the vertices, and derived a worst-case complexity bound of $O(hn3^{d/3})$, where h is the h-index of the input graph. However, none of these approaches explicitly evaluated the impact of set-intersections on the complexity of the BK algorithm. In addition, despite the well-established complexity bounds for degeneracy- and degree-based orderings of vertices, a bound lower than $O(3^{n/3})$ has not been reported for arbitrary orderings.

3. A BROAD COMPLEXITY ANALYSIS

This section contributes a broad time and space complexity analysis of the BK algorithm, taking into account both vertex ordering strategies and set intersection algorithms. In particular, we show that whereas hash-join-based set-intersection algorithms lead to ideal complexity bounds, merge-join-based set intersections can lead to a Δ_{max} times higher worst-case time complexity, where Δ_{max} is the maximum node degree of the input graph. We also show that the recursive subgraph-creation scheme given by Eppstein et al. [26] enables the ideal worst-case time complexity to be achieved even when using merge-join-based set intersection algorithms, but the cost of doing so is a d-times higher peak space complexity. These results are summarized in Figure 2.

The order in which the vertices are processed when building the recursion tree has a significant impact on the exponential factors of the worst-case complexity. These results are presented in Table 1. Note that in the case of the arbitrary ordering we contribute improved complexity bounds for general and scale free graphs. In the case of the degeneracy ordering, our result are slightly different from those of Eppstein et al. [26] because we introduce the additional term Δ_{avg}, which is the average node degree of the input graph.

Examples of some real-world graph datasets with their parameters affecting the complexity are given in Table 2. These graphs come from the *Network Data Repository* [46] and *SNAP* [39]. In all of these cases $\Delta_{avg} < d < h < \Delta_{max}$, which means that the algorithms using the degeneracy ordering lead to significantly lower theoretical complexity bounds than the ones with arbitrary and degree ordering. Therefore, we focus on the degeneracy ordering in this work.

3.1 Effect of set-intersection algorithms

Because set intersections are heavily used in the Bron-Kerbosch algorithm, it is important to determine the effect of specific set-intersection algorithms on the overall complexity. In this work, we focus on the two most commonly used methods: merge-join- and hash-join-based algorithms.

In this section, we call $p = |P|$ and $x = |X|$ the sizes of the sets P and X in one recursive call of *BKPivot* subroutine of Algorithm 1. Lets assume that a graph has n vertices, m edges, and degeneracy d. The maximum degree of a vertex is Δ_{max}, and the average degree of the vertices is Δ_{avg}. The following properties hold for p, x, and d [52]:

$$p \leq d, \quad p + x \leq \Delta_{max}, \quad \frac{\Delta_{avg}}{2} \leq d < \Delta_{max}.$$

The relationship between the intersection time and the execution time of the *BKPivot* function is given as follows:

LEMMA 1. *The complexity of the BKPivot function without its child recursive calls is*

$$O\left((p+x)\, I\left(p, \Delta\right) + p\left(I\left(p, \Delta\right) + I\left(x, \Delta\right)\right)\right), \qquad (1)$$

where Δ is size of the largest adjacency list accessed, and $I(a, b)$ is the time to intersect a set with a elements and an adjacency list with b elements.

PROOF. To determine the pivot vertex, we perform $p + x$ intersections between the set P and the adjacency list of a vertex from $P \cup X$ (see *getPivot* function), which takes $O\left((p+x)I(p, \Delta)\right)$ time. Then, the *BKPivot* function intersects sets P and X up to p times with the adjacency list of vertices in $P/N(u)$, which takes $O\left(p(I(p, \Delta) + I(x, \Delta))\right)$ time. The total time is the sum of these two results. □

The next lemma offers the result for the time complexity of the overall algorithm for two significant cases.

LEMMA 2. *Let D_0 be the time to execute the BKPivot function without its recursive calls. Then, the time it takes for the BK algorithm with degeneracy ordering to compute all maximal cliques of a graph G is:*

$$D(G) = \begin{cases} O(\Delta_{avg} n 3^{d/3}), & \text{for } D_0 = O(p^2 x) \\ O(\Delta_{avg}\Delta_{max} n 3^{d/3}), & \text{for } D_0 = O(p^2 x \Delta_{max}). \end{cases}$$

PROOF. Using Lemma 5 of Eppstein et al. [26], execution time of the *BKPivot* function including its child recursive calls is $O(x3^{p/3})$ when $D_0 = O(p^2 x)$ and $O(x\Delta_{max}3^{p/3})$

Table 2: Graph properties. Graph sizes are in MB. Graphs larger than 1 GB are considered large.

small graphs	n	Δ_{avg}	d	h	Δ_{max}	size	large graphs	n	Δ_{avg}	d	h	Δ_{max}	size
wiki-talk (wt)	2.4 M	3.9	131	1055	100 k	64	**orkut (or)**	3.1 M	76.3	253	1638	33 k	1740
b-anon (ba)	2.9 M	14.3	63	722	4.4 k	80	**sinaweibo (sw)**	59 M	8.9	193	5902	278 k	3891
as-skitter (as)	1.7 M	13.1	111	982	35 k	143	**aff-orkut (ao)**	8.7 M	74.9	471	6064	318 k	4915
livejournal (lj)	4.0 M	13.9	213	558	2.6 k	393	**clueweb (cw)**	148 M	6.1	192	2783	308 k	7373
topcats (tc)	1.8 M	28.4	99	1457	238 k	403	**wiki-link (wl)**	26 M	41.9	1120	5908	4.2 M	9728
pokec (pk)	1.6 M	27.3	47	492	15 k	405	**friendster (fr)**	66 M	54.5	304	2958	5.2 k	30720

when $D_0 = O(p^2 x \Delta_{max})$. The total cost of all the invocations of *BKPivot* in the *BKDegeneracy* function is

$$\sum_{v} O(x3^{p/3}) \leq O(m3^{d/3}) = O(\Delta_{avg} n 3^{d/3}), \qquad (2)$$

when $D_0 = O(p^2 x)$. Similarly, the total execution time is $O(\Delta_{avg}\Delta_{max} n 3^{d/3})$ when $D_0 = O(p^2 x \Delta_{max})$. □

Hash-join-based set intersections require a first set S_a to be hashed. The second set S_b is traversed while performing lookups in S_a. Assuming each lookup takes $O(1)$ time in the worst case, the total time needed for the intersection is $O(|S_b|)$. Constant worst-case lookup time can be achieved using cuckoo hashing [42], hopscotch hashing [32], and several different perfect hashing algorithms [6, 12, 27].

In the hash-join-based BK algorithm, the adjacency list of each vertex is stored in a dedicated hash-table. Given a graph with n vertices and m edges, construction of the hash tables can be achieved in $O(n+m)$ expected time and space complexity, which can be approximated as $O(\Delta_{avg} n)$ when $\Delta_{avg} >= 1$. This pre-processing overhead is significantly lower than the complexity of the BK algorithm.

THEOREM 1. *The BK algorithm computes all maximal cliques of a graph G in $O(\Delta_{avg} n 3^{d/3})$ time using degeneracy ordering and hash-join-based set intersections.*

PROOF. Given that $I(a, b) = O(a)$ when using hash joins with the second set hashed, the complexity of the *BKPivot* function without its child recursive calls is $O(p(p+x))$ using Lemma 1. The total execution time of the algorithm is obtained by applying Lemma 2 with $D_0 = O(p^2 x)$ given that $O(p(p + x)) \subset O(p^2 x)$. □

Note that Theorem 1 gives a tighter lower bound than $O(dn3^{d/3})$ by Eppstein et al. [26] because $O(\Delta_{avg}) \subset O(d)$. Other hashing algorithms, such as linear probing and double hashing [35], offer weaker bounds on the complexity of hash table lookups. While on average each lookup takes constant time, in the worst-case a lookup can require a linear scan of the hash table. As a result, performing $|S_b|$ lookups in S_a takes $O(|S_a||S_b|)$ time in the worst case, which increases the time complexity of the BK algorithm by Δ_{max} times.

Merge-join-based set intersections require both sets to be sorted. We iterate through both sets in a sequential fashion, looking for common elements. In the worst case, a merge-join-based set intersection performs $O(|S_a| + |S_b|)$ comparisons. In the merge-join-based BK algorithm, sorting the adjacency lists of the input graph is done as a preprocessing step. Because the result of each intersection is also sorted, all the sets remain sorted during the execution of the algorithm without any additional sorting overhead.

THEOREM 2. *The BK algorithm computes all maximal cliques of a graph G in $O(\Delta_{avg}\Delta_{max} n 3^{d/3})$ time using degeneracy ordering and merge-join-based set intersections.*

PROOF. Because the size of the largest adjacency list accessed can be as large as Δ_{max} and given that $I(a, b) = O(a + b)$ in the merge-join case, the complexity of *BKPivot* without its child recursive calls is $O((p+x)(p+\Delta_{max}))$ using Lemma 1. By applying the Lemma 2 for $D_0 = O(p^2 x \Delta_{max})$ given that $O((p + x)(p + \Delta_{max})) \subset O(p^2 x \Delta_{max})$, we obtain the total execution time of the algorithm. □

In summary, merge-join-based set-intersections lead to a Δ_{max} times higher asymptotic time complexity than hash-join-based set-intersections.

3.2 Effect of recursive subgraph creation

Eppstein et al. [26] contributed a subgraph-based BK algorithm, which creates a new subgraph denoted as $H_{P,X}$ before each recursive call. These calls then use their respective subgraphs instead of the original graph G. $H_{P,X}$ is a subgraph of G induced by the vertex set $P \cup X$. However, the edges that exist between the vertices in X in G are not included in $H_{P,X}$. Algorithm 3 shows the *BKSubgraph* function, which replaces the *BKPivot* function of the original BK algorithm, wherein a new function called *createHpxSubgraph* is introduced to create $H_{P,X}$. Note that our formulation of the *BKSubgraph* function given in Algorithm 3 is slightly different from the one given by Eppstein et al. We create only one subgraph per recursive call whereas Eppstein et al.'s formulation [26] creates $O(p)$ subgraphs per call; thus our formulation incurs a lower overhead per call. Note also that the *BKDegeneracy* function has to invoke *BKSubgraph* instead of *BKPivot* when using recursive subgraph creation.

Algorithm 3: Bron-Kerbosch w. subgraph creation

```
1  Function BKSubgraph(Sets R, P, X, graph G)
2      if P = ∅ then
3          if X = ∅ then Report R as a maximal clique;
4          return;
5      SG = createHpxSubgraph(P, X, G);
6      pivot = getPivot(P, X, SG);
7      foreach vertex v : P/N_SG(pivot) do
8          P' = P ∩ N_SG(v); X' = X ∩ N_SG(v);
9          BKSubgraph(R + {v}, P', X', SG);
10         P = P − {v}; X = X + {v};
11 Function createHpxSubgraph(Sets P, X, graph G)
12     foreach vertex u : P do
13         N_SG(u) = (P ∪ X) ∩ N_G(u);
14         foreach vertex v : N_SG(u) do
15             if v is in X then
16                 N_SG(v) = N_SG(v) + {u};
17     return SG
```

In this section, we evaluate the impact of recursive subgraph creation on the time complexity of the BK algorithm.

We evaluate this impact in combination with both merge-join-based and hash-join-based set-intersection algorithms.

Merge join with recursive subgraph creation approach combines the subgraph-based BK algorithm with merge-join-based set intersections. Because the subgraphs shrink with each recursive call, using subgraphs that have smaller adjacency lists than the original graph leads to faster execution of merge-join-based set intersections.

LEMMA 3. *createHpxSubgraph can be executed in $O(p(p+x))$ time using merge-join-based set intersections.*

PROOF. The *createHpxSubgraph* function iterates through all the vertices in P. For each vertex $u \in P$, it determines the adjacency list $N_{SG}(u)$ of the subgraph SG by intersecting $P \cup X$ with the adjacency list $N_G(u)$ of the original graph G. Because the size of $N_G(u)$ is $O(p+x)$, the time needed to create $N_{SG}(u)$ for each $u \in P$ using merge-join-based set intersections is $O(p+x)$. The adjacency lists $N_{SG}(v)$ of $v \in X$ are initially empty. Whenever we find a vertex $v \in X$ in the previously created adjacency list $N_{SG}(u)$, we update $N_{SG}(v)$ by inserting u into it. This insertion can be done in $O(1)$ time by appending the vertex u at the end of $N_{SG}(v)$ because both the adjacency lists and the sets P are always stored in a sorted fashion and traversed as such in merge-join-based implementations. There are $O(p+x)$ vertices to be examined in $N_{SG}(u)$. Therefore, for each $u \in P$, the time needed to update the adjacency lists $N_{SG}(v)$ of $v \in X$ is $O(p+x)$. Since p iterations are performed in the outer loop of the *createHpxSubgraph* function, the function executes in $O(p(p+x))$ time. □

THEOREM 3. *The BK algorithm computes all maximal cliques of a graph G in $O(\Delta_{avg} n 3^{d/3})$ time using degeneracy ordering and merge-join-based set intersections in combination with recursive subgraph creation.*

PROOF. Since we use the $H_{P,X}$ subgraph instead of the original graph in the *BKSubgraph* function, the size of the largest adjacency list used in that function is $p+x$. Therefore, the time needed to execute *BKSubgraph* without the *createHpxSubgraph* function can be obtained using Lemma 1, where $\Delta = p+x$ and $I(a,b) = O(a+b)$, which results in $O(p(p+x))$ time. Recalling from Lemma 3 that subgraph creation takes $O(p(p+x))$ time, the total time needed to execute *BKSubgraph* without its recursive calls is $O(p(p+x))$. Similar to the proof of Theorem 1, the overall complexity can be obtained by using Lemma 2 with $D_0 = O(p^2 x)$. □

In summary, recursive subgraph creation enables the BK algorithm based on merge joins to achieve the same time complexity bound as the hash-join-based BK algorithm.

Hash join with recursive subgraph creation method combines hash-join-based set intersections with subgraph-based BK algorithm. Because the hash-join-based implementation operates on hashed adjacency lists, creating a new subgraph requires construction of several new hash tables that store the adjacency lists of the subgraph. Note that the worst-case complexity of constructing a hash table is quadratic in its size given that the worst-case complexity of inserting an element into a hash table is linear in the number of elements for most hashing algorithms.

THEOREM 4. *The BK algorithm computes all maximal cliques of a graph G in $O(\Delta_{avg}\Delta_{max} n 3^{d/3})$ time using degeneracy ordering and hash-join-based set intersections in combination with recursive subgraph creation.*

PROOF. The intersections shown in the line 13 of Alg. 3 can be performed in $O(p+x)$ time using the hash-join approach. Therefore, computing all the adjacency lists of a subgraph takes $O(p(p+x))$ time. However, we also have to construct hash tables that store the adjacency lists of the subgraph. Given that a $H_{P,X}$ subgraph has p adjacency lists with at most $p+x$ elements and x adjacency lists with at most p elements, constructing all the hash tables takes $O\left((p+x)^2 + xp^2\right) = O\left(p(p+x)^2\right)$ time because of the quadratic complexity of hash table construction. The total time needed to execute *createHpxSubgraph* is $O\left(p(p+x)^2\right)$.

Using Lemma 1 with $I(a,b) = O(a)$, the execution time of *BKSubgraph* without the *createHpxSubgraph* function is $O(p(p+x))$. Adding it to the time to compute the *createHpxSubgraph* function, the execution time of the *BKSubgraph* function becomes $O\left(p(p+x)^2\right)$. The total execution time of the BK algorithm is obtained by applying Lemma 2 with $D_0 = O(p^2 x \Delta_{max})$ given that x is at most Δ_{max} and $O(p(p+x)^2) \subset O(p^2 x \Delta_{max})$. □

Theorem 4 shows that recursive subgraph creation increases the worst-case time complexity when using hash-join-based set intersections. However, there exist hashing algorithms that support insertions in constant worst-case time complexity with high probability [3,4,7,24,29], leading to a linear worst-case hash table construction complexity. Using such algorithms would reduce the complexity of the BK algorithm with hash-join-based set intersections and recursive subgraph creation to $O(\Delta_{avg} n 3^{d/3})$.

3.3 Space complexity

In this section, we perform a space complexity analysis of the BK algorithm assuming the degeneracy ordering of the vertices. In particular, we analyze the impact of recursive subgraph creation on the peak dynamic memory usage.

At each recursive step, the BK algorithm computes new P, R, and X sets. After that, the BK algorithm invokes a child recursive call and passes the new sets as its parameters. Note that the maximum clique size is upper bounded by $d+1$. Therefore, storing these three sets requires $O(p+x+d) = O(\Delta_{max})$ space. When the recursion tree is explored in a depth-first-search (DFS) order, the space used for storing the intermediate results is limited to the current execution depth. In addition, when using the degeneracy ordering, the maximum recursion depth is d. Thus, the peak memory consumption of the single-threaded execution of the BK algorithm is $O(d\Delta_{max})$ without taking into account the space needed to store the input graph and the cliques found.

Recursive subgraph creation increases the memory usage further. Each $H_{P,X}$ subgraph uses $O(p(p+x)+xp) = O(p(p+x)) = O(d\Delta_{max})$ space because it connects either two vertices of P or one from P and one from X [26]. Creating a new subgraph for each recursive call increases the peak memory consumption to $O(d \times d\Delta_{max}) = O(d^2\Delta_{max})$. Therefore, recursive subgraph creation can increase the dynamic memory usage by up to d times. In summary, even though recursive subgraph creation reduces the time complexity of the BK algorithm that uses merge-join-based set intersections, it increases its dynamic memory usage.

The results of our analysis are given in Figure 2. Note that when exploring the recursion tree in the DFS order, the dynamic memory usage of the BK algorithm is independent of n. It depends only on the recursion depth d and Δ_{max}.

3.4 Parallel time and space complexity

The BK algorithm can be parallelized by considering each recursive call as an independent unit of work (i.e., task). Given N hardware threads, N DFS-based explorations of the recursion tree can be performed concurrently. Based on Brent's theorem, we have $T_N \leq T_1/N + T_\infty$, where T_1 is the execution time using a single thread, T_∞ is the execution time using infinitely many threads, and T_N is the execution time using N threads [13]. In the case of the BK algorithm with degeneracy ordering, $T_1 = O(\Delta_{avg} n 3^{d/3})$ and $T_\infty = O(d)$ because d is the maximum depth of the recursion tree. Thus, the execution time using N threads of execution is

$$T_N = O\left(\frac{\Delta_{avg} n}{N} 3^{d/3} + d\right). \tag{3}$$

As a result, we expect the performance of parallel implementations that take advantage of dynamic task scheduling frameworks [10, 36, 44] to scale linearly with the number of threads (strong scaling). In addition, when d is constant and the number of threads (N) is a linear function of n, the worst-case time complexity is also a constant. Hence, a weak performance scaling can be achieved as well.

When we execute the BK algorithm on N threads, we explore N different DFS paths of the recursion tree in parallel. Because the memory consumption of each thread is equal to the memory usage of a single DFS path, the space needed for executing the parallel algorithm is up to N times larger than the single-threaded results given in Section 3.3. We conclude that the peak memory consumption is $O(Nd\Delta_{max})$ without subgraphs while it is $O(Nd^2\Delta_{max})$ with subgraphs. Considering both hash-join-based and merge-join-based set intersections, we summarize these results in Figure 2.

3.5 Arbitrary vertex orderings

In this section, we derive new complexity bounds for arbitrary vertex orderings. When using arbitrary vertex orderings, the size of set P is no longer upper bounded by d but by the maximum vertex degree Δ_{max}. Based on this observation, the time complexity of the BK algorithm that uses arbitrary vertex ordering is $O(n3^{\Delta_{max}/3})$, because each invocation of the *BKPivot* function takes $O(3^{\Delta_{max}/3})$ time.

A better complexity bound can be derived for scale-free graphs. In scale-free graphs, the probability of a vertex having degree Δ is $P(\Delta) \sim \Delta^{-\gamma}$, where the γ parameter is typically between 2 and 3 [5]. Many real-world graphs are scale free, such as the World Wide Web, protein-interaction, and email networks. These graphs have a limited number of highly-connected vertices. When the BK algorithm is executed on scale-free graphs, most of its execution time is spent when starting from those highly-connected vertices.

THEOREM 5. *The BK algorithm computes all maximal cliques of a scale-free graph in $O(3^{\Delta_{max}/3})$ time using an arbitrary ordering of vertices when $\Delta_{max}^\gamma \leq 3^{(\Delta_{max}-1)/3}$.*

PROOF. Each invocation of *BKPivot* in the *BKDegeneracy* function takes $O(3^{\Delta/3})$ time, for an arbitrary vertex ordering. The sum of the cost of all invocations is

$$\sum_v O(3^{\Delta/3}) = O\left(\sum_{\Delta=1}^{\Delta_{max}} N(\Delta) 3^{\Delta/3}\right), \tag{4}$$

where $N(\Delta)$ is the number of vertices with degree Δ. In scale-free graphs, the degrees follow a power-law distribution, i.e., the number of vertices with degree Δ is proportional to $n\Delta^{-\gamma}$. For scale-free graphs, it also holds that $n = O\left(\Delta_{max}^{\gamma-1}\right)$ [5]. By using these properties, we obtain

$$\begin{aligned} O\left(\sum_{\Delta=1}^{\Delta_{max}} N(\Delta) 3^{\Delta/3}\right) &= O\left(n \sum_{\Delta=1}^{\Delta_{max}} \frac{3^{\Delta/3}}{\Delta^\gamma}\right) \\ &\leq O\left(n \frac{3^{\Delta_{max}/3}}{\Delta_{max}^{\gamma-1}}\right) \leq O\left(3^{\Delta_{max}/3}\right), \end{aligned} \tag{5}$$

where $3^{\Delta/3}/\Delta^\gamma \leq 3^{\Delta_{max}/3}/\Delta_{max}^\gamma$ holds for every Δ_{max} if $3^{(\Delta_{max}-1)/3} \geq \Delta_{max}^\gamma$. □

In real-world graphs, the condition $3^{(\Delta_{max}-1)/3} \geq \Delta_{max}^\gamma$ almost always holds. For example, when $\gamma = 3$, the maximum node degree (i.e., Δ_{max}) needs to be larger than 29.

Theorem 5 shows that the worst-case complexity of the BK algorithm using an arbitrary vertex ordering depends only on the time to process the vertex with the maximum degree. Note that Δ_{max} is much smaller than n in real-world graphs even though $\Delta_{max} = O(n)$ (see Table 2). Effectively, we have derived a new bound that is significantly tighter than the $O(3^{n/3})$ bound reported in Tomita et al. [55].

4. VECTORIZED SET INTERSECTIONS

In this section, we briefly describe our SIMD-accelerated hash-join-based set intersection implementation. We show that it outperforms state-of-the-art methods when the sets involved in intersections have disproportionate sizes, which frequently occurs when executing the BK algorithm.

4.1 Our set intersection implementation

We have developed a hash-join-based set intersection algorithm, called *SimpleHashSet*, which constructs hopscotch hash tables [32] and performs SIMD-accelerated table lookups. We use hopscotch hash tables to achieve $O(1)$ worst-case complexity for the lookups. Even though the construction of hopscotch hash tables differs from the construction of the tables used by linear probing implementations, the table lookups can be performed in exactly the same way. Our SIMD implementation of table lookups is based on the SIMD-accelerated linear probing implementation of Polychroniou et al. [43]. However, we support only unique integer keys without payloads. Thus, our design is much simpler than that of Polychroniou et al. [43]. Note that the set difference operations used by the BK algorithm can also be implemented using our SIMD-accelerated hash-table lookups.

We build a dedicated hopscotch hash table for each vertex of the input graph to store its adjacency list. Hopscotch hash tables are constructed in such a way that each key is found within H entries of the address computed by the hash function [32]. In our implementation, H is the number of integer keys that fit into one cache line. When computing the size of a hash table, we multiply the size of the respective adjacency list by two and round it up to the nearest power of two. Note that the hopscotch hash tables require a hash function from a universal family. We take advantage of multiplicative universal hashing, which uses one multiplication, one addition, and one bit-shift operation [23, 59].

Table 3: Hardware platforms used in the paper

platform	Intel KNL [53]	Intel Xeon Skylake
no. cores	64	48
no. threads	256	96
SIMD instr.	AVX-512	AVX-512
memory	110 GB	360 GB
L1d cache	32 KB per core	32 KB per core
L2 cache	1 MB per 2 cores	1 MB per core
L3 cache	none	38.5 MB

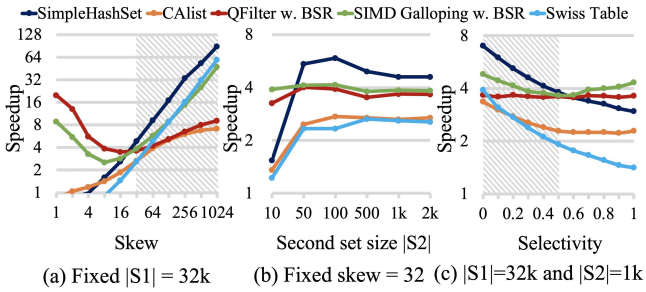


Figure 3: Speedup of SIMD-accelerated set intersection algorithms compared to scalar-merge-based set intersections.

4.2 Comparison to other algorithms

We perform an experimental study of set intersection algorithms *i)* to show the efficiency of our algorithms in comparison to prior solutions and *ii)* to understand when hash joins should be favored over merge joins. Our experiments in this section are performed on Intel Xeon Phi[1] 7210 - Knights Landing (KNL) processor [53], described in Table 3. Intel KNL supports the state-of-the-art AVX-512 instructions, where each vector register enables operations on sixteen 32-bit integer operands in parallel. The following state-of-the-art implementations of set intersection algorithms that use SIMD instructions were used in the comparison.

QFilter is a merge-join-based set intersection algorithm optimized for graph processing by Han et al. [30]. It uses a compressed bit-vector representation of graph vertices, called BSR, and it accelerates the set-intersection operations using 128-bit vector registers and the AVX2 instruction set. The main drawback of this method is that it requires a time-consuming reordering of the input graph vertices in order to achieve high-performance intersections.

Galloping is a merge-join-based intersection algorithm that locates the members of the first set in the second set using binary search, and it can be accelerated using SIMD instructions [38]. Han et al. [30]. optimized this approach to use their compressed bit-vector representation. We refer to this implementation as *SIMD Galloping with BSR*.

CAlist was described in our previous work [8], which is also a merge-join-based set-intersection implementation. It uses cache-aligned linked lists to minimize cache misses and efficiently leverages AVX-512 instructions. CAlist works well when the sets have disproportionate sizes because it can skip redundant comparisons. In addition, it can be easily adapted to take advantage of even wider vector registers.

Swiss Table is a highly-optimized open-source hash table library by Abseil adapted from Google's C++ codebase [1].

We use a microbenchmarking approach, in which we randomly generate and intersect two sets $S1$ and $S2$ and vary their sizes. Within the BK algorithm, $S1$ is an adjacency list of the graph, which is hashed in the hash-join-based case, and $S2$ is either set P or set X. Figure 3 shows the speedup achieved by different set intersection strategies over a scalar merge join strategy that is used as the baseline. We fix the *density* of the sets (i.e., the ratio between the larger set size and the range of the elements in sets [30]) to 0.5 in the case of QFilter and SIMD Galloping with BSR, and to 0.1 in all other cases. The reason for using the higher density in QFilter and SIMD Galloping with BSR cases is because they benefit from graph reordering [30], which increases the densities of the sets and enables more efficient intersections.

Figure 3a compares the performance of different set intersection implementations when the ratio between the set sizes (i.e., *skew*) varies. We fix the *selectivity* of the intersections (i.e., the ratio between the size of the result and the smaller set) to 0.3 because the BK algorithm using scalar-merge-based set intersections spends $50-60\%$ of its set intersection time on set intersections with a selectivity lower than 0.3 on average across all the graphs from Table 2. Then, we fix the size of the set $S1$ to 32000, which is in the order of the average size of set $S1$ observed when processing the large graphs. Note that this average is much larger than Δ_{avg} because the vertices with higher degrees take part in the intersections much more frequently. We then vary the skew between 1 and 1024. We see that *SimpleHashSet* is preferable when the set sizes are disproportionate (the shaded region) whereas QFilter is preferable when the skew is small. (i.e., when $S1$ and $S2$ are similar in size).

When operating on the large graphs, the BK algorithm that uses scalar-merge-based set intersections spends more than 80% of its intersection time on intersecting sets with a skew larger than 32. Thus, in Figure 3b, we fix the skew to $|S1|/|S2| = 32$ while keeping the selectivity at 0.3 and we vary the size of both sets proportionally until $|S1| = 64000$. It is clear that *SimpleHashSet* outperforms all other set intersections when $|S2|$ is not extremely small. In Figure 3c, we keep the skew at $|S1|/|S2| = 32$, fix the size of $S1$ to 32000, and vary the selectivity. Averaged across the large graphs from Table 2, the BK algorithm that uses the scalar merge spends almost 70% of its intersection time executing the intersections with a selectivity lower than 0.5. Our *SimpleHashSet* is faster than the other set intersection algorithms exactly in that region.

In conclusion, hash-join-based set intersections are preferable to merge-join-based ones when the set sizes involved in the intersections are highly skewed, which is often the case for the BK algorithm. Furthermore, our *SimpleHashSet* method is competitive against state-of-the-art set intersection methods such as QFilter without requiring compressed bit-vector representations such as BSR.

5. MANYCORE IMPLEMENTATION

In this section, we build on the theoretical results of Section 3 and develop a shared-memory parallel implementation of the BK algorithm that exploits task-level parallelism and guarantees a worst-case complexity of $O(Nd\Delta_{max})$ on the peak dynamic memory consumption. First, we introduce the software framework we use to exploit the task-level parallelism dynamically and describe our initial parallel implementation of the BK algorithm. Next, we discuss the optimizations that minimize the dynamic memory usage and maximize the scalability of our software implementation.

[1]Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

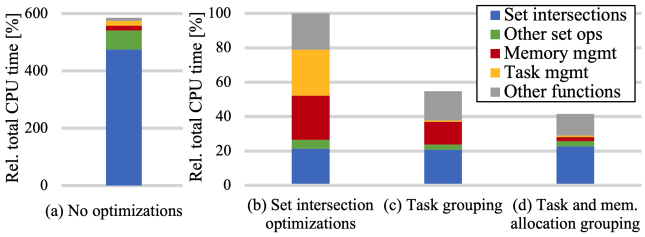


Figure 4: Impact of various optimizations on the total CPU time used by the manycore BK implementation when processing the *orkut* graph. In (a), we use scalar-merge-based intersections. In (b), we use our *SimpleHashSet*.

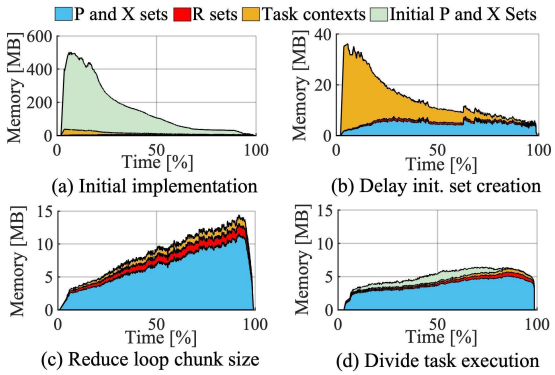


Figure 5: Dynamic memory usage over time for the *orkut* graph. Our optimizations reduce the peak dynamic memory usage by 80× while affecting the runtime only marginally.

The experiments presented in this section are performed on the Intel KNL platform (see Table 3) using 256 hardware threads. We use Intel VTune Amplifier version 2018.3 to obtain the execution time breakdown. Tools such as Intel VTune and Valgrind Massif [51] can be used to perform dynamic memory usage profiling. However, they are extremely slow in doing so. Therefore, we extract the dynamic memory usage profile by instrumenting our code. We keep track of the dynamic memory allocations and deallocations performed by individual threads on the relevant data structures, and periodically sample their sum. A new sample is collected when a certain number of memory allocations have been performed, which is tracked by an atomic counter.

5.1 Leveraging task parallelism frameworks

We use the Intel *Threading Building Blocks* (TBB) software framework [36] for exploiting the task parallelism. TBB implements a dynamic scheduler that can dispatch tasks to parallel worker threads, where the load balance across these threads is achieved via the work-stealing approach [11]. Each worker thread is typically pinned to a hardware thread. Additionally, TBB offers a scalable memory allocator that reduces the overhead of concurrent memory allocations.

Our initial manycore implementation wraps each recursive call to the *BKPivot* function of Algorithm 1 in a task. In each iteration of its `foreach` loop, memory for the new sets $P' = P \cap N_G(v)$ and $X' = X \cap N_G(v)$ is allocated, and a new task is *spawned* with these sets as parameters. We refer to the original task as the parent task, and to the spawned tasks as child tasks. When the `foreach` loop completes, we wait for all the child tasks to complete before destroying the parent task. Note that, the *BKDegeneracy* function, which is the root of the recursion tree, is also implemented as a task, and the iterations of its `foreach` loop are executed in parallel using TBB's `parallel_for` loop construct. When recursive subgraph creation is enabled, each task creates a $H_{P,X}$ subgraph from the P and X sets as described in Algorithm 3. Parallel DFS exploration of the recursion tree is enforced by the TBB scheduler by default. Each worker thread simply prioritizes the task it spawned most recently.

Figure 4 shows the execution time breakdown of our manycore BK implementation. In the unoptimized case shown in Figure 4a, set intersections based on the scalar merge approach dominate the execution time. However, using our *SimpleHashSet*, described in Section 4, accelerates set intersections by 22×, as shown in Figure 4b. This result matches the results of Figure 3a, where *SimpleHashSet* is up to 128 times faster than the scalar merge approach for the cases that frequently occur in the BK algorithm. However, once the set intersection implementation is optimized, the task and memory management overheads can constitute up to 50% of the total CPU time, as shown in Figure 4b. In addition, our initial implementation does not achieve the ideal space complexity results reported in Section 3.4, and uses more memory than necessary. In this section, we discuss these problems in detail and offer our solutions.

5.2 Minimizing dynamic memory usage

The initial manycore implementation of the BK algorithm described in the previous section uses more dynamic memory than what is predicted in Section 3.4. The main reason is that a task executing on a TBB thread spawns all its child tasks and allocates memory for them while it is still executing. After the completion of the current task, the thread can switch to one of these child tasks. However, the remaining child tasks occupy memory that is not yet being used, causing two main problems: 1) Dynamic memory usage depends on the number of vertices n. 2) The memory used by the *BKPivot* task can be up to d times larger than necessary.

The first problem is caused by the `parallel_for` that implements the `foreach` loop of the *BKDegeneracy* function. The default behaviour of TBB's `parallel_for` loop is to heuristically group its iterations into *chunks*, and then a worker thread sequentially executes an entire chunk before starting to work on another chunk [58]. Each iteration of the main loop of the *BKDegeneracy* function spawns a task with an initial pair of P and X sets (lines 4-6 of Algorithm 2). By default, TBB does not limit the chunk size, so the `parallel_for` loop might have a chunk made of up to n loop iterations. A thread executing that chunk would then allocate $O\left(n(\Delta_{max} + c)\right)$ memory, assuming that c is the memory used by the task context in addition to the sets. The resulting memory allocations could easily become the dominant component of the dynamic memory usage when processing large graphs. Figure 5a shows that the initial sets consume a significant amount of memory in the case of the *orkut* graph. One way to solve this problem is to delay creation of the initial sets to the start of the corresponding *BKPivot* function, which ensures that each thread creates at most one pair of the initial sets. By doing so, we reduce the dynamic memory usage by 15× on the *orkut* graph, compared to the initial implementation (Figure 5b). However, the task contexts created in the `parallel_for` loop of the *BKDegeneracy* function still represent a large part of the

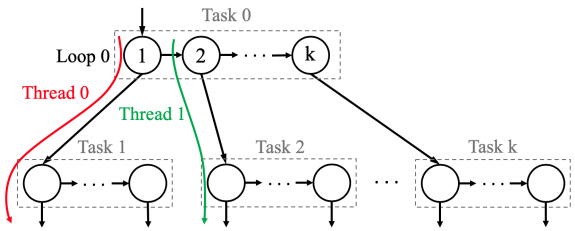


Figure 6: Reducing dynamic memory usage by enabling different threads to execute different loop iterations (circles).

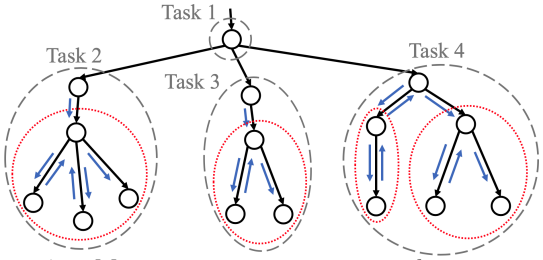


Figure 7: Manycore optimizations: task grouping and memory allocation grouping. Circles represent recursive calls, dashed ellipses the tasks, and dotted ellipses the recursive calls that share the same pre-allocated memory region.

execution time. This problem can be solved by limiting the chunk size to one, which results in each chunk creating exactly one task. As Figure 5c shows, this optimization further reduces the dynamic memory usage to less than half. Note that now the space complexity no longer depends on n.

The second problem happens because each *BKPivot* task spawns $O(|P|) \subset O(d)$ child tasks after creating the respective P and X sets, which leads to usage of $O(d)$-fold more dynamic memory than necessary within recursive calls. Then, the space complexity becomes d times higher than what is predicted in Section 3.4. To solve this issue, we enable different threads to execute different iterations of the `foreach` loop shown in line 6 of the Algorithm 1 (see Figure 6). After executing a loop iteration, the worker thread can either switch to the next loop iteration or to its child task. Using TBB's *scheduler bypass* feature, we force the thread to switch to the child task (Thread 0 of Figure 6) and return the context of the parent task to the scheduler. Another thread can later continue executing the parent task by picking up its context from the scheduler (Thread 1 of Figure 6). This approach is similar to continuation stealing introduced by *Cilk-5* [28, 37]. By executing the child task before the next loop iteration, each worker thread spawns only one task at a time instead of spawning O(d) tasks at once. Figure 5d shows that this optimization further reduces the dynamic memory usage by 50% when processing the *orkut* graph.

As a result of the previous optimizations, our manycore implementation of the BK algorithm uses $O(Nd\Delta_{max})$ space for dynamic memory as we predict in Section 3.4. The dynamic memory usage is reduced by 80× in the case of the *orkut* graph (see Figure 5), and between 30× to 180× when processing the small graphs. The highest dynamic memory usage measured when processing these seven graphs is around 10 MB, which is significantly lower than the cumulative cache capacity of the Intel KNL processors (see Table 3).

5.3 Task grouping

If more time is spent on managing tasks rather than executing them, the manycore implementation will not scale well. As Figure 4b suggests, more than 25% of the time is spent on task management. One way of reducing the task management overheads is to group several recursive calls in a single task. However, grouping too many calls in a task can lead to load imbalances, which increases the idle time of the worker threads. In this section, we describe a task grouping heuristic that aims to marginalize the impact of task management on the runtime without causing resource underutilization. This goal is fulfilled by creating sufficiently complex tasks.

Theorem 1 shows that the complexity of executing a recursive call without its child recursive calls depends on the size of the sets P and X as $O(|P|(|P| + |X|))$. Both sets typically become smaller as we move deeper in the recursion tree. Therefore, we heuristically restrict task grouping only to the recursive calls near the bottom of the recursion tree. We create the task groups implicitly by not spawning new tasks if the cardinality of the corresponding $P \cup X$ set is smaller than a task threshold t_t, and execute the following recursive call sequentially instead, as depicted in Figure 7. Figure 4c shows that the task management overheads become negligible after the optimization. In addition, the memory management overheads are also indirectly reduced. We will study the choice of the empirical parameter t_t in Section 5.5 and show that it is not particularly critical.

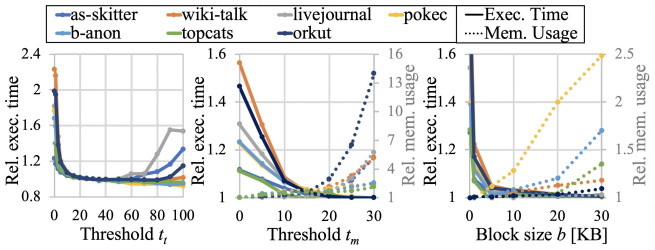
This optimization has no side effect on the peak space complexity because the recursive calls within a task are implicitly executed in DFS order by our C++ implementation. Thus, the peak space complexity bounds derived for DFS-based processing in Section 3.4 apply without any changes.

5.4 Memory allocation grouping

Frequent memory allocations and deallocations by multiple threads can cause contention, lead to performance overheads, and limit scalability. TBB's scalable memory allocator alleviates such problems, but it cannot eliminate them completely. The main reason is the frequent dynamic allocations and deallocations of the P and X sets. Given that the majority of the recursive calls are short-lived, especially towards the leaves of the recursion tree, memory allocation can easily become a scalability bottleneck. Figure 4c shows that, after task grouping, memory management overheads still take up to 25% of the total CPU time.

We introduce a memory allocation grouping method to reduce the memory management overheads of our implementation. To reduce the number of memory allocations, we create a large block of memory, in which the sets created by several consecutive recursive calls are placed. Considering that the sets become smaller and the memory allocations become more frequent as we move deeper in the recursion tree, grouping the memory allocations that originate near the bottom of the tree is a necessity. Similarly to our task grouping approach, we introduce a memory threshold t_m and group memory allocations of a recursive call and all its child calls when $|P| + |X| \leq t_m$. We constrain t_m to be smaller than the task threshold t_t to ensure that memory blocks are not shared by different threads and that there is no need to synchronize the accesses to these blocks.

We denote the size of a pre-allocated memory block as b. When b is not large enough to accommodate all the sets, a



(a) Varying task threshold (b) Varying mem. threshold (c) Varying block size

Figure 8: Sensitivity of our manycore implementation to the parameters t_t, t_m and b. Graph (a) shows the execution time relative to $t_t = 30$. Graph (b) shows the execution time relative to $t_m = 30$ and the peak memory usage relative to $t_m = 0$. Graph (c) shows the execution time relative to $b = 30$ KB and the peak memory usage relative to $b = 100$ B.

singly-linked list of such blocks is created. Using a too large b increases the memory usage whereas using a too small one increases the number of allocations and negatively impacts the performance. Figure 4d shows that our memory allocation grouping method using $t_m = 20$ and $b = 20$ KB virtually eliminates the memory management overheads.

5.5 Sensitivity analysis

This section evaluates the impact of the manycore implementation parameters t_t, t_m, and b on execution time and memory usage. Experiments are performed on Intel KNL using 256 hardware threads and cover all small graphs and one large graph (i.e., *orkut*) from Table 2.

First, we evaluate the task-grouping optimization in isolation. Figure 8a shows the impact of the task threshold t_t on the execution time. In all the cases evaluated, the best performance is achieved when t_t is between 20 and 50. Note that t_t does not influence the dynamic memory usage. Next, we set $t_t = 30$ and evaluate the memory-allocation-grouping optimization. In Figure 8b, we set $b = 20$ KB and vary the memory threshold t_m. Across all the graphs tested, the lowest execution time and dynamic memory usage combinations are achieved when the range of t_m is between 10 and 20. In fact, the highest performance is achieved when $t_m = 30$. However, its dynamic memory usage can be up to 13× higher than that of the baseline (i.e., $t_m = 0$), which disables the memory grouping optimization. A good trade-off is achieved when $t_m = 20$. In this case, the execution time is within 5% of the optimal and the memory usage is at most 3× higher than that of the baseline. Finally, in Figure 8c we set $t_m = 20$ and vary the block size b. The execution times decrease as we increase b and reach their optimal values at $b = 20$ KB. Note that the dynamic memory usage increases linearly with the block size after a certain point. When $b = 20$ KB, the memory usage can be up to 2× higher than that of the baseline that does not use memory pre-allocation (i.e., $b = 0$), which is not a significant overhead.

Thus, our experiments suggest that none of the empirical parameters is particularly critical: *i)* the results are consistent across all the graphs we use and *ii)* for each parameter there exists a reasonable range where the optimizations are similarly effective as with the very best values. Therefore, we set $t_t = 30$ for task grouping and $b = 20$ KB, $t_m = 20$ for memory allocation grouping. We use these values in all the experiments performed in the remainder of this paper.

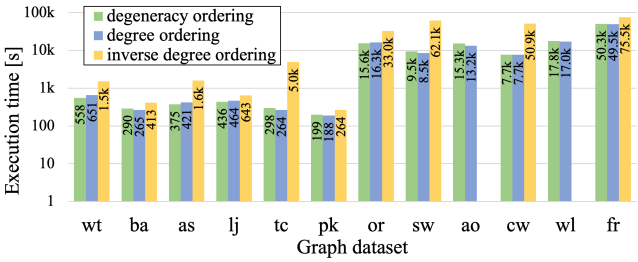


Figure 9: Impact of various vertex ordering techniques on the single-threaded performance of the BK algorithm when using Intel KNL. The experiments using the inverse degree ordering did not succeed in under 48 h for *ao* and *wl* graphs.

6. EXPERIMENTAL RESULTS

In this section, we first show the impact of our algorithmic and implementation choices as well as our optimizations on execution time, memory consumption, and manycore scalability. Then, we compare our optimized implementation with state-of-the-art references in terms of both single-threaded and multi-threaded performance.

6.1 Experimental setup

In the experiments we use two platforms: Intel KNL and Intel Xeon Skylake. We developed our code on Intel KNL and ran most of the analyses there; yet, for completeness, we ran the scalability analysis and the comparisons to competing implementations also on Intel Xeon Skylake processors available in Google Cloud's Compute Engine. The properties of these platforms are summarized in Table 3. We build our code, which is available online[2], using GCC v8.3.1 with -O3 optimization flag. To exploit task parallelism, we use version 2019_U9 of the Intel TBB framework. As already mentioned, the graph datasets we use are obtained from the *Network Data Repository* [46] and *SNAP* [39] (see Table 2). In our experiments, the input graph is preloaded in the main memory and interleaved across all NUMA regions. We remove all self loops from the graphs and transform all directed edges to undirected edges. When evaluating the performance of the MCE implementations, we do not store the maximal cliques found, but we simply count them.

6.2 Evaluation of vertex ordering strategies

Figure 9 shows the impact of different vertex ordering strategies on the single-threaded execution time of the BK algorithm on Intel KNL. We evaluated three main strategies: *i)* degeneracy ordering [26], *ii)* degree ordering (ascending order), and *iii)* inverse degree ordering (descending order). The inverse degree ordering strategy provides us with a lower bound of the worst-case behavior of arbitrary vertex orderings covered in Section 3.5. As expected, this strategy leads to the worst performance results, resulting in an up to 16× slow-down with respect to degeneracy ordering, this confirms the results of our theoretical analysis provided in Table 1. However, despite the better worst-case complexity bound it achieves, the degeneracy ordering does not always lead to a better practical performance than the degree ordering. We believe that further theoretical analysis could shed more light on this empirical observation.

[2] `github.com/accelerated-graph-mining/hash-join-mce`

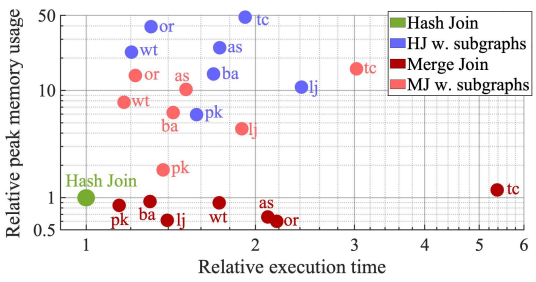
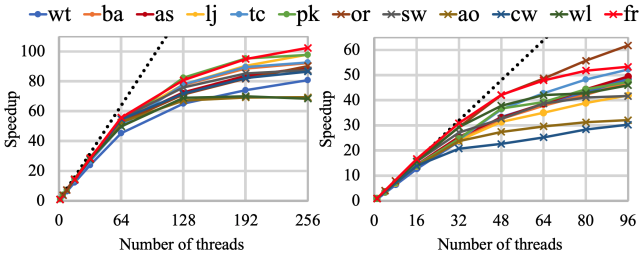


Figure 10: Impact of (i) hash-join- vs. merge-join-based intersection algorithms and of (ii) recursive subgraph creation on runtime and memory usage of the BK algorithm.



(a) KNL: 64 x 4 threads (b) Xeon Skylake: 48 x 2 threads

Figure 11: Performance scaling on modern many-core processors: speed-ups are relative to single-threaded execution.

6.3 Hash joins versus merge joins

In this section, we evaluate the impact of set intersection algorithms on the overall performance and memory usage of the BK algorithm. The hash-join-based set intersections are implemented using *SimpleHashSet* described in Section 4, and the merge-join-based set intersections are implemented using CAlist [8]. We also evaluate the impact of using recursive subgraph creation described in Section 3.1. However, instead of creating a subgraph per call, we create a subgraph per task to limit the overhead of subgraph creation.

The experiments are performed on Intel KNL using all 256 hardware threads. In Figure 10, we evaluate all six small graphs and one large graph from Table 2 and show the execution time and the peak dynamic memory usage results of different set intersection methods. We repeat these experiments both with and without recursive subgraph creation. The results given in Figure 10 are relative to our hash-join-based BK implementation that does not use recursive subgraph creation. Peak dynamic memory usage is measured as described in Section 5, where we track dynamic memory allocations and deallocations of each thread and sample the peak after a certain number of allocations. Note that these results do not include the memory used by the input graphs.

We see that the merge-join-based approach benefits from creating subgraphs. Its execution time is reduced as much as 44% for the *tc* graph, but at the cost of increased memory usage. In all the cases, our hash-join-based BK implementation that does not create subgraphs is Pareto optimal as predicted by our theoretical analysis (see Figure 2). On the other hand, also as predicted by our theoretical analysis, the hash-join-based BK implementation does not benefit from recursive subgraph creation. Note that the subgraphs created by the hash-join-based implementation use more memory than those created by the merge-join-based implementation because the adjacency lists of the subgraphs are stored as hash tables in the hash-join case, and our *SimpleHashSet* implementation sets the size of the hash tables to twice the size of the adjacency lists to minimize hash conflicts.

6.4 Scalability analysis

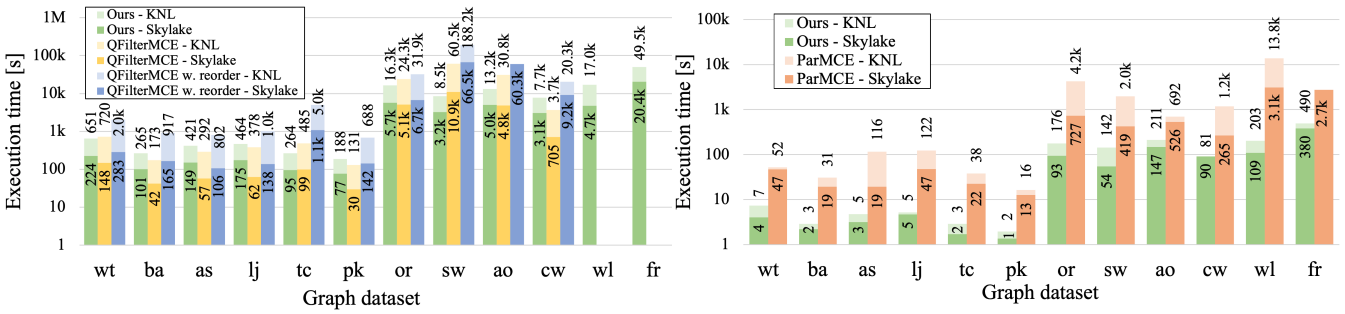
The scalability analysis is carried out on both hardware platforms shown in Table 3 using the graphs from Table 2. Figure 11a shows the performance improvements we achieve with respect to single-threaded execution when increasing the number of threads on the KNL architecture. We observe an almost linear performance scaling up to 64 threads, which is the number of physical cores available. After 64 threads, the performance scales sublinearly because the threads running on the same core start sharing hardware resources, e.g., SIMD units. Using two hardware threads per core improves the performance by only 40%. Using all 256 hardware threads, we achieve up to 100× speedup compared to single-threaded execution. Figure 11b shows the performance scaling when using Xeon Skylake processors. We observe up to 60× speedup using 96 hardware threads.

6.5 Comparisons with the state of the art

We compare our BK implementation with the following state-of-the-art MCE implementations: *i)* **QFilterMCE** by Han et al. [30] is an optimized single-threaded implementation that uses QFilter and SIMD Galloping with BSR methods, described in Section 4, to accelerate set intersections. QFilterMCE uses compressed bit-vectors for representing the graph vertices and requires a preprocessing step that reorders the vertices in order to perform more efficient set intersections. *ii)* **ParMCE** by Das et al. [22] is an optimized shared-memory parallel C++ implementation that uses the Intel TBB library for parallelization. In the comparisons, we use our BK implementation based on degree ordering because the performance of the BK algorithm based on degree ordering is similar to the one that uses degeneracy ordering as shown in Figure. 9. In addition, computing the degree order does not require any advanced preprocessing.

We compare the single-threaded execution of our implementation with QFilterMCE. Figure 12a shows that our implementation, which uses the *SimpleHashSet* algorithm given in Section 4 to accelerate set intersections, displays a competitive performance to that of QFilterMCE even though our solution does not require any preprocessing. On average, taking into the account both the time to preprocess the graph and to execute QFilterMCE, our solution is faster than QFilterMCE by 4.1× on KNL and by 2.7× on Xeon Skylake. In addition, the QFilter preprocessing did not finish under 48 h on KNL for the *ao* and *wl* graphs, and under 24 h on Xeon Skylake for the *wl* graph. It also failed to execute for the *fr* graph on both platforms. Note that we reused the preprocessing results computed by Xeon Skylake for the *ao* graph when executing QFilterMCE on KNL.

Figure 12b compares the execution time of our manycore *SimpleHashSet*-based implementation with ParMCE using 256 hardware threads on KNL and 96 hardware threads on Xeon Skylake. On average, we achieve 14.3× and 8.3× lower execution times than ParMCE on KNL and Skylake, respectively. Note that ParMCE runs out of memory when executing the *fr* graph on KNL and goes into swap space when executing the *wl* graph on KNL. The highest speedups we achieve with respect to ParMCE on KNL and Skylake are 68× and 28×, respectively.



(a) Single-threaded comparisons

(b) Multi-threaded comparisons using all available hardware threads

Figure 12: Comparisons with state of the art on Intel KNL (the top value) and Intel Xeon Skylake (the bottom value).

7. RELATED WORK

Maximal clique enumeration (MCE) represents an important graph mining problem with applications in various fields, such as bioinformatics [62, 63], social network analysis [41], and electronic design automation [45, 57]. The most efficient class of MCE algorithms are based on backtracking search, such as the algorithm from Bron and Kerbosch [15], and its improvements by Tomita et al. [55] and Eppstein et al. [26], which we discuss in more detail in Section 2. These algorithms do not analyze the effect of using different intersection strategies on the overall time complexity. Our work analyzes the effect of using hash and merge joins for intersections on the time and space complexity of the algorithm by Eppstein et al. [26] and reaps the corresponding advantage.

SIMD-accelerated set-intersection algorithms can be used to improve the speed of MCE [30, 34, 49]. Schlegel et al. [49] and Inoue et al. [34] exploit STTNI instructions in order to accelerate set intersections using merge joins. QFilter by Han et al. [30] further improves the performance of set intersections by using a compressed bit-vector representation. QFilter is used for accelerating graph algorithms such as MCE. However, it requires a preprocessing of the input graph in order to achieve a high performance. Our *Simple-Hashset* approach uses hash joins rather than merge joins, and it does not involve any significant preprocessing, yet it achieves a performance comparable to that of QFilter.

Of particular interest to our work are multi-core implementations of MCE [22, 40, 50]. Schmidt et al. [50] present a parallel variant of the MCE algorithm by Bron and Kerbosch [15], and describe a work-stealing method for load balancing. Das et al. [22] present ParMCE, a shared-memory parallel algorithm for MCE, which uses Intel TBB library for load balancing. ParMCE is based on the algorithm by Tomita et al. [55], an improved version of the algorithm by Bron and Kerbosch [15]. Our work further improves the performance by addressing the task and memory management overheads that arise in our TBB-based implementation of MCE. Lessley et al. [40] describe a parallel algorithm based on data-parallel primitives [9], which can be executed on both multi-core CPUs and GPUs by generating the corresponding TBB or CUDA code. However, it explores the search space of MCE in a breadth-first order, which is memory inefficient compared to depth-first-based solutions such as ours. As pointed out by Das et al. [22], the CPU implementation of Lessley et al. [40] fails to execute on even moderately sized graphs such as *wiki-talk* from Table 2. Our work takes into account the dynamic memory usage of multi-core MCE and proposes methods to minimize it.

Distributed implementations of MCE have also been proposed [17, 33, 54]. Svendsen et al. [54] use different vertex ordering strategies for statically balancing the load across the computation nodes. Brighen et al. [14] propose using a vertex-centric framework Giraph [48], which is based on the bulk synchronous parallel (BSP) model [56], for distributed computation of MCE. However, our work uses a framework with dynamic load balancing, which is designed to cope with load balancing and synchronization issues better than the simpler static load balancing and the less specialized BSP on shared-memory manycore processors. The work by Chen et al. [17] uses the idea of recursive subgraph partitioning in order to distribute the work across multiple computing nodes dynamically. This technique aims at a coarser grain parallelism than the one we exploit in our manycore implementation and is essentially orthogonal to our work.

8. CONCLUSIONS

In this paper we explore the use of join algorithms for accelerating set intersections in the MCE algorithm proposed by Eppstein et al. [26]. We theoretically show that the use of hash-join-based set intersections enables Pareto-optimal MCE implementations in terms of time and space complexity compared to various possibilities that use merge-join-based set intersections. Building on this result, we introduce a simple SIMD-accelerated hash-join-based set intersection implementation and use it to accelerate MCE. Using this simple approach, we match the single-threaded performance of an MCE implementation that uses highly-optimized set intersections, which requires some time-consuming preprocessing; our implementation does not suffer from such a requirement. In addition, we contribute a manycore version of MCE that uses a shared-memory parallel processing framework for exploiting task-level parallelism. When implemented in a naive way, the many-core implementation suffers from scalability overheads and poor dynamic memory management. By addressing those issues, we achieve a maximum speedup of 100× compared to the single-threaded case on a machine with 64 physical cores; we outperform a state-of-the-art manycore MCE implementation by an order of magnitude. Our future work will focus on scalable execution of a wide range of recursive graph mining algorithms.

Acknowledgements. We thank the authors of *ParMCE* [21, 22] for sharing the source code of their implementation. The support of Swiss National Science Foundation (project number 172610) for this work is gratefully acknowledged.

9. REFERENCES

[1] Abseil Swiss Tables. `https://abseil.io/blog/20180927-swisstables`.

[2] C. C. Aggarwal and H. Wang, editors. *Managing and Mining Graph Data*, volume 40 of *Advances in Database Systems*. Springer US, Boston, MA, 2010.

[3] Y. Arbitman, M. Naor, and G. Segev. De-amortized Cuckoo Hashing: Provable Worst-Case Performance and Experimental Results. In *Automata, Languages and Programming*, volume 5555, pages 107–118. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. Series Title: Lecture Notes in Computer Science.

[4] Y. Arbitman, M. Naor, and G. Segev. Backyard Cuckoo Hashing: Constant Worst-Case Operations with a Succinct Representation. In *2010 IEEE 51st Annual Symposium on Foundations of Computer Science*, pages 787–796, Las Vegas, NV, USA, Oct. 2010. IEEE.

[5] A.-L. Barabási and M. Pósfai. *Network science*, chapter The scale-free property. Cambridge University Press, Cambridge, United Kingdom, 2016. OCLC: ocn910772793.

[6] D. Belazzougui, F. C. Botelho, and M. Dietzfelbinger. Hash, Displace, and Compress. In *Algorithms - ESA 2009*, volume 5757, pages 682–693. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. Series Title: Lecture Notes in Computer Science.

[7] I. O. Bercea and G. Even. Fully-Dynamic Space-Efficient Dictionaries and Filters with Constant Number of Memory Accesses. *arXiv:1911.05060 [cs]*, Nov. 2019. arXiv: 1911.05060.

[8] J. Blanuša, R. Stoica, P. Ienne, and K. Atasu. Parallelizing Maximal Clique Enumeration on Modern Manycore Processors. In *IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2020, New Orleans, Louisiana, USA, May 18-22, 2020*, pages 211–214. IEEE, 2020.

[9] G. E. Blelloch. *Vector models for data-parallel computing*. Artificial intelligence. MIT Press, Cambridge, Mass, 1990.

[10] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. *Journal of Parallel and Distributed Computing*, 37(1):55–69, Aug. 1996.

[11] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, Sept. 1999.

[12] F. C. Botelho, R. Pagh, and N. Ziviani. Practical perfect hashing in nearly optimal space. *Information Systems*, 38(1):108–131, Mar. 2013.

[13] R. P. Brent. The Parallel Evaluation of General Arithmetic Expressions. *J. ACM*, 21(2):201–206, Apr. 1974.

[14] A. Brighen, H. Slimani, A. Rezgui, and H. Kheddouci. Listing all maximal cliques in large graphs on vertex-centric model. *The Journal of Supercomputing*, Feb. 2019.

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

[16] F. Cazals and C. Karande. A note on the problem of reporting maximal cliques. *Theoretical Computer Science*, 407(1-3):564–568, Nov. 2008.

[17] Q. Chen, C. Fang, Z. Wang, B. Suo, Z. Li, and Z. G. Ives. Parallelizing Maximal Clique Enumeration Over Graph Data. In *Database Systems for Advanced Applications*, volume 9643, pages 249–264. Springer International Publishing, Cham, 2016.

[18] A. Conte, R. Grossi, A. Marino, and L. Versari. Sublinear-Space Bounded-Delay Enumeration for Massive Network Analytics: Maximal Cliques. page 15 pages, 2016.

[19] L. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, Oct. 2004.

[20] L. Danon, A. Ford, T. House, C. Jewell, M. Keeling, G. Roberts, J. Ross, and M. Vernon. Networks and the Epidemiology of Infectious Disease. *Interdisciplinary perspectives on infectious diseases*, 2011:284909, 03 2011.

[21] A. Das, S.-V. Sanei-Mehri, and S. Tirthapura. Shared-Memory Parallel Maximal Clique Enumeration. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*, pages 62–71, Bengaluru, India, Dec. 2018. IEEE.

[22] A. Das, S.-V. Sanei-Mehri, and S. Tirthapura. Shared-memory Parallel Maximal Clique Enumeration from Static and Dynamic Graphs. *ACM Trans. Parallel Comput.*, 7(1):1–28, Apr. 2020.

[23] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A Reliable Randomized Algorithm for the Closest-Pair Problem. *Journal of Algorithms*, 25(1):19–51, Oct. 1997.

[24] M. Dietzfelbinger and F. Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In M. S. Paterson, editor, *Automata, Languages and Programming*, volume 443, pages 6–19. Springer-Verlag, Berlin/Heidelberg, 1990. Series Title: Lecture Notes in Computer Science.

[25] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis. GraMi: frequent subgraph and pattern mining in a single large graph. *PVLDB*, 7(7):517–528, 2014.

[26] D. Eppstein, M. Löffler, and D. Strash. Listing All Maximal Cliques in Sparse Graphs in Near-optimal Time. *arXiv:1006.5440 [cs]*, June 2010. arXiv: 1006.5440.

[27] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a Sparse Table with 0(1) Worst Case Access Time. *J. ACM*, 31(3):538–544, June 1984.

[28] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation - PLDI '98*, pages 212–223, Montreal, Quebec, Canada, 1998. ACM Press.

[29] M. T. Goodrich, D. S. Hirschberg, M. Mitzenmacher, and J. Thaler. Cache-Oblivious Dictionaries and Multimaps with Negligible Failure Probability. In *Design and Analysis of Algorithms*, volume 7659, pages 203–218. Springer Berlin Heidelberg, Berlin,

Heidelberg, 2012. Series Title: Lecture Notes in Computer Science.

[30] S. Han, L. Zou, and J. X. Yu. Speeding Up Set Intersections in Graph Algorithms using SIMD Instructions. In *Proceedings of the 2018 International Conference on Management of Data - SIGMOD '18*, pages 1587–1602, Houston, TX, USA, 2018. ACM Press.

[31] W.-S. Han, J. Lee, and J.-H. Lee. Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases. pages 337–348, 06 2013.

[32] M. Herlihy, N. Shavit, and M. Tzafrir. Hopscotch Hashing. In *Distributed Computing*, volume 5218, pages 350–364. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[33] B. Hou, Z. Wang, Q. Chen, B. Suo, C. Fang, Z. Li, and Z. G. Ives. Efficient Maximal Clique Enumeration Over Graph Data. *Data Sci. Eng.*, 1(4):219–230, Dec. 2016.

[34] H. Inoue, M. Ohara, and K. Taura. Faster set intersection with SIMD instructions by reducing branch mispredictions. *PVLDB*, 8(3):293–304, 2014.

[35] D. E. Knuth. *The art of computer programming. Volume 1, Volume 1,*. 1968. OCLC: 489816776.

[36] A. Kukanov. The Foundations for Scalable Multicore Software in Intel Threading Building Blocks. *ITJ*, 11(04), Nov. 2007.

[37] I.-T. A. Lee, S. Boyd-Wickizer, Z. Huang, and C. E. Leiserson. Using memory mapping to support cactus stacks in work-stealing runtime systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques - PACT '10*, page 411, Vienna, Austria, 2010. ACM Press.

[38] D. Lemire, L. Boytsov, and N. Kurz. SIMD Compression and the Intersection of Sorted Integers. *Softw. Pract. Exper.*, 46(6):723–749, June 2016. arXiv: 1401.6399.

[39] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[40] B. Lessley, T. Perciano, M. Mathai, H. Childs, and E. W. Bethel. Maximal clique enumeration with data-parallel primitives. In *2017 IEEE 7th Symposium on Large Data Analysis and Visualization (LDAV)*, pages 16–25, Phoenix, AZ, Oct. 2017. IEEE.

[41] Z. Lu, J. Wahlström, and A. Nehorai. Community Detection in Complex Networks via Clique Conductance. *Sci Rep*, 8(1):5982, Dec. 2018.

[42] R. Pagh and F. F. Rodler. Cuckoo Hashing. In G. Goos, J. Hartmanis, J. van Leeuwen, and F. M. auf der Heide, editors, *Algorithms — ESA 2001*, volume 2161, pages 121–133. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. Series Title: Lecture Notes in Computer Science.

[43] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking SIMD Vectorization for In-Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data - SIGMOD '15*, pages 1493–1508, Melbourne, Victoria, Australia, 2015. ACM Press.

[44] M. J. Quinn. *Parallel programming in C with MPI and openMP*. McGraw-Hill, Dubuque, Iowa, 2004.

[45] J. Reddington and K. Atasu. Complexity of Computing Convex Subgraphs in Custom Instruction Synthesis. *IEEE Trans. VLSI Syst.*, 20(12):2337–2341, Dec. 2012.

[46] R. A. Rossi and N. K. Ahmed. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*, 2015.

[47] R. A. Rossi, D. F. Gleich, and A. H. Gebremedhin. Parallel Maximum Clique Algorithms with Applications to Network Analysis. *SIAM Journal on Scientific Computing*, 37(5):C589–C616, Jan. 2015.

[48] S. Sakr, F. M. Orakzai, I. Abdelaziz, and Z. Khayyat. *Large-Scale Graph Processing Using Apache Giraph*. Springer International Publishing, Cham, 2016.

[49] B. Schlegel, T. Willhalm, and W. Lehner. Fast sorted-set intersection using simd instructions. In *ADMS@VLDB*, 2011.

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

[51] J. Seward, editor. *Valgrind 3.3: advanced debugging and profiling for Gnu/Linux applications*. Network Theory, Bristol, 1. print edition, 2008. OCLC: 476326409.

[52] K. Shin, T. Eliassi-Rad, and C. Faloutsos. CoreScope: Graph Mining Using k-Core Analysis — Patterns, Anomalies and Algorithms. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 469–478, Barcelona, Spain, Dec. 2016. IEEE.

[53] A. Sodani. Knights landing (KNL): 2nd Generation Intel® Xeon Phi processor. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–24, Cupertino, CA, USA, Aug. 2015. IEEE.

[54] M. Svendsen, A. P. Mukherjee, and S. Tirthapura. Mining maximal cliques from a large graph using MapReduce: Tackling highly uneven subproblem sizes. *Journal of Parallel and Distributed Computing*, 79-80:104–114, May 2015.

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

[56] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, Aug. 1990.

[57] A. K. Verma, P. Brisk, and P. Ienne. Fast, Nearly Optimal ISE Identification With I/O Serialization Through Maximal Clique Enumeration. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, 29(3):341–354, Mar. 2010.

[58] M. Voss, R. Asenjo, and J. Reinders. *Pro TBB: C++ Parallel Programming with Threading Building Blocks*, chapter 16. Apress, Berkeley, CA, 2019.

[59] P. Woelfel. Efficient Strongly Universal and Optimally Universal Hashing. In *Mathematical Foundations of Computer Science 1999*, volume 1672, pages 262–272. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. Series Title: Lecture Notes in Computer Science.

[60] Xifeng Yan and Jiawei Han. gSpan: graph-based substructure pattern mining. In *2002 IEEE*

International Conference on Data Mining, 2002. Proceedings., pages 721–724, Maebashi City, Japan, 2002. IEEE Comput. Soc.

[61] Y. Xu, J. Cheng, A. W.-C. Fu, and Y. Bu. Distributed Maximal Clique Computation. In *2014 IEEE International Congress on Big Data*, pages 160–167, Anchorage, AK, USA, June 2014. IEEE.

[62] L. Yang, X. Zhao, and X. Tang. Predicting Disease-Related Proteins Based on Clique Backbone in Protein-Protein Interaction Network. *Int. J. Biol. Sci.*, 10(7):677–688, 2014.

[63] H. Yu, A. Paccanaro, V. Trifonov, and M. Gerstein. Predicting interactions in protein networks by completing defective cliques. *Bioinformatics*, 22(7):823–829, Apr. 2006.