Parallelizing Maximal Clique Enumeration on Modern Manycore Processors

Jovan Blanuša*†, Radu Stoica*, Paolo Ienne†, Kubilay Atasu*
*IBM Research - Zürich, †Ecole Polytechnique Fédérale de Lausanne (EPFL)
{jov,rst}@zurich.ibm.com, paolo.ienne@epfl.ch, kat@zurich.ibm.com

***Abstract*—Many fundamental graph mining problems, such as maximal clique enumeration and subgraph isomorphism, can be solved using combinatorial algorithms that are naturally expressed in a recursive form. However, recursive graph mining algorithms suffer from a high algorithmic complexity and long execution times. Moreover, because the recursive nature of these algorithms causes unpredictable execution and memory access patterns, parallelizing them on modern computer architectures poses challenges. In this work, we describe an efficient manycore CPU implementation of maximal clique enumeration (MCE), a basic building block of several social and biological network mining algorithms. First, we improve the single-thread performance of MCE by accelerating its computation-intensive kernels through cache-conscious data structures and vector instructions. Then, we develop a multi-core solution and eliminate its scalability bottlenecks by minimizing the scheduling and the memory-management overheads. On highly-parallel modern CPUs, we demonstrate an up to 19-fold performance improvement compared to a state-of-the-art multi-core implementation of MCE.**

***Index Terms*—graph mining, maximal clique enumeration**

I. Introduction

Subgraph patterns in graph datasets, such as communities, clusters, and motifs, are fundamental concepts used in a wide range of graph mining applications in various fields [1]. However, extracting subgraph patterns often requires executing recursive algorithms that lead to a combinatorial explosion of the search space, resulting in long run-times. It is becoming increasingly challenging for applications to provide real-time insights as the data set sizes continue to grow [2]. An open research question is how to efficiently execute such recursive graph mining algorithms on modern computer architectures.

Trends in multi-core CPUs offer unique opportunities to accelerate graph mining algorithms. Modern manycore CPUs have several architectural features that make them well-suited for running this type of problems. Firstly, such algorithms access memory in an unpredictable manner, which often leads to cache misses and memory access stalls. Today's processors have a large number of cores, each capable of executing multiple simultaneous threads. The hardware can hide the latency of memory accesses by transparently switching between these threads (i.e. the memory accesses can be overlapped). Second, modern CPUs have high-bandwidth memory interconnects that increase the rate at which the data can be processed. Third, wide vector instructions can be leveraged to significantly accelerate the data-parallel operations.

Parallelizing recursive graph algorithms is not straightforward. Real-world graphs exhibit irregular connectivity patterns and operating on them introduces scattered memory accesses that result in cache misses and data transfers across NUMA (Non-Uniform Memory Access) domains. The growing number of physical cores exacerbates all NUMA effects [3] and imposes high penalties when accessing remote data. In addition, since the recursion tree is discovered dynamically and its shape cannot be known in advance, distributing the work evenly across all available hardware resource units (threads, cores, and memory regions) is challenging. Poor load balancing can result in unnecessary bottlenecks and decreased performance.

In this work, we describe how to efficiently parallelize the maximal clique enumeration (MCE), a fundamental graph mining algorithm. MCE is a building block of many different graph mining applications, such as detection of communities in social networks [4], prediction of protein functions in protein interaction networks [5], and prediction of how epidemics spread [6]. Other graph mining algorithms, such as subgraph isomorphism [7], frequent subgraph mining [8], and maximum clique finding [9], are similar to MCE in terms of the way solutions are incrementally constructed and the type of processing that dominates their execution time (i.e., random accesses to adjacency lists of the graphs and intersections of vertex sets).

We first optimize the single thread performance of MCE and then develop scalable parallel implementations. We accelerate the single thread performance by optimizing vertex set intersections through two orthogonal approaches: i) we use cache-optimized data structures and vector instructions, and ii) we reduce the sizes of the sets via subgraph-centric optimizations. We then parallelize MCE across multiple CPU cores by i) dynamic load-balancing across the cores via work stealing, and ii) a NUMA-aware subgraph-centric partitioning of the input graph. Lastly, we address the scalability bottlenecks encountered, namely the memory management and the task scheduling overheads. Overall, we show that our architecture-conscious design leads to an order of magnitude speed-up with respect to a recently-proposed multi-core solution [10], [11].

II. Accelerating Maximal Clique Enumeration

One of the most popular maximal clique enumeration algorithms is the Bron-Kerbosch (BK) algorithm [12], which performs a backtracking search to list all maximal cliques. Tomita et al. [13] improved the original BK algorithm by using a new pivoting technique that leads to a more efficient pruning of the search tree. Eppstein et al. [14] further improved this algorithm for sparse graphs by using the *degeneracy order* of

978-1-7281-7445-7/20/$31.00 ©2020 IEEE
DOI 10.1109/IPDPSW50202.2020.00047

Algorithm 1: Bron-Kerbosch w. degeneracy ordering

```
1  Function BKDegeneracy (Graph G(V, E))
2      Order vertices in G using degeneracy ordering;
       // The outer loop
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);
7  Function BKPivot (Sets R, P, X, graph G)
       // Backtrack
8      if P = ∅ then
9          if X = ∅ then Report R as a maximal clique;
10         return;
       // Find pivot
11     foreach vertex v : P ∪ X do
12         t_v = |P ∩ N_G(v)|;
13     pivot = arg max_v(t_v);
       // Recursive calls
14     foreach vertex v : P/N_G(pivot) do
15         BKPivot(R + {v}, P ∩ N_G(v), X ∩ N_G(v), G);
16         P = P − {v};
17         X = X + {v};
```

the vertices when constructing the recursion tree. In this paper, we present an efficient parallel implementation of Eppstein's version of the BK algorithm, which is given in Algorithm 1.

The BK algorithm maintains three sets of vertices: clique set *R*, candidate set *P*, and exclude set *X*. The algorithm searches for the maximal cliques containing all of the vertices from *R*, some vertices from *P*, and none from *X*. At each recursive call, the set *R* is expanded by a vertex *v* from the set *P*, and the sets *P* and *X* are intersected with the neighborhood of *v*. If both *P* and *X* are empty, *R* is reported as a maximal clique.

The *degeneracy* of a graph is the smallest value *d* such that each nonempty subgraph of it has a vertex with at most *d* edges [14]. When the degeneracy of a graph is *d*, its vertices can be ordered in such a way that each vertex has at most *d* neighbors that appear later in this order. Processing the vertices in the resulting *degeneracy order* in the outer loop of the BK algorithm reduces the search space by bounding the size of the initial candidate set *P* to d. Given a graph with n vertices and degeneracy d, the worst-case complexity of Eppstein et al.'s algorithm is $O(dn3^{d/3})$, which is linear in n and exponential in d, whereas the worst-case complexity of Tomita et al.'s algorithm [13] is $O(3^{n/3})$, which is exponential in n.

A. Data-parallel set intersections

The set intersection operations are the dominant part of the BK algorithm [15]. Performing set intersections is required both when determining the pivot vertex (line 12 in *BKPivot*) and when constructing the new *P* and *X* sets (line 15 in *BKPivot*). To improve the performance, it is crucial to reduce the time spent on these operations. We accelerate set intersections by implementing a cache-friendly data structure called

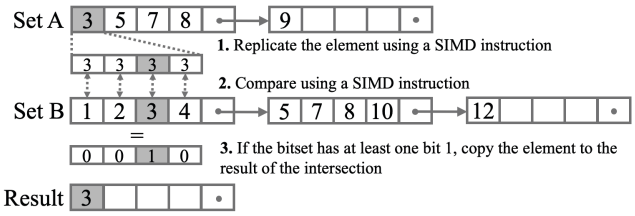


Fig. 1: Data-parallel set intersections using CAlist.

cache-aligned list (*CAlist*), which reduces the cache misses and increases the processing rates by using vector instructions.

We represent sets as ordered lists of 32-bit vertex IDs and parallelize the set intersections using SIMD (Single Instruction Multiple Data) instructions, which enable us to perform several comparisons in only a few clock cycles. Sets are stored in lists of cache-aligned buckets, where each bucket fits into an L2 cache line and stores several vertex IDs as well as a pointer to the next bucket. An intersection between two sets is performed by iterating through the smaller set and checking whether the vertices of the smaller set also exist in the larger set.

Figure 1 illustrates the way we search for a vertex of set *A* in set *B*. Whether vertex 3 of set *A* exists in the first bucket of set *B* can be determined using two SIMD instructions. The first SIMD instruction replicates the vertex ID in a vector, whose size is the same as the size of the bucket. The second SIMD instruction compares this vector with the contents of the bucket. As a result, we obtain a bit vector that indicates the position of vertex 3 inside the current bucket. If the resulting bit vector is nonzero, vertex 3 is in the intersection of the two sets. In this particular example, we see that the next element of set *A* (i.e., vertex 5) cannot exist in the first bucket of set *B* because it has a value greater than the value of the last element of the bucket. Therefore, we can simply skip to the next bucket of set *B* and repeat the same steps for vertex 5.

The performance of the set intersection operations can be further improved by using bit-vector-based implementations and by reordering the vertices to increase the data locality [15]. Currently, no such optimizations are exploited by our work.

B. Subgraph-centric processing

An orthogonal way of accelerating set intersections is to reduce the number of vertices in the sets. This goal can be achieved by creating a subgraph induced by the neighborhood of a vertex v_i in the *BKDegeneracy* function and forwarding it to the corresponding *BKPivot* function [11]. Creating subgraphs enables the following recursive calls to perform faster set intersections by using smaller adjacency lists.

An additional benefit of subgraph-centric processing is that it improves memory locality and reduces remote memory accesses. Instead of accessing a single large graph distributed across all NUMA domains, each task can access a subgraph local to its memory region. This optimization reduces the latency of accessing memory, makes caching more efficient, and reduces communication across the chip, all of which improve the performance of the BK algorithm on NUMA architectures.

Related subgraph-centric parallelization approaches have also been used in distributed graph processing frameworks [16].

C. Multi-core optimizations

We use the Intel Threading Building Blocks[1] (TBB) library [17] for parallel processing, which enables defining *tasks* as independent units of computation that are separately scheduled for execution on the available hardware threads. TBB uses *work-stealing* scheduling, which performs dynamic load balancing across the threads [18]. In addition, TBB offers a scalable memory allocator, which reduces contention when multiple threads allocate and deallocate memory concurrently.

Initially, each recursive call to the *BKPivot* function is defined as a task. In each iteration of the foreach loop shown in line 14 of Algorithm 1, memory for the subsets $P' = P \cap N_G(v)$ and $X' = X \cap N_G(v)$ is allocated, and a new task is *spawned* with the subsets as parameters. When the foreach loop completes, we wait for all spawned child tasks to return before finishing the current task. The iterations of the foreach loop of the *BKDegeneracy* function are also executed in parallel by spawning a dedicated task for each one.

The scalability of our parallel BK implementations can be limited by task scheduling and memory management overheads. In the following, we discuss our relevant optimizations.

1) Reducing the task scheduling overheads: Grouping multiple recursive calls into a single task reduces the task creation and scheduling overheads. If more time is spent on managing tasks rather than executing them, the multi-core implementation will not scale well. Because the calls typically become shorter lived as we move deeper in the recursion tree, we heuristically restricted task grouping to the recursive calls near the bottom of the recursion tree. However, we also impose a limit on the number of recursive calls that can be combined in a single task to preserve the efficiency of work stealing.

2) Reducing the memory management overheads: Frequent memory allocations and deallocations by multiple threads can cause contention and lead to performance bottlenecks. TBB's scalable memory allocator can alleviate such problems, but it cannot eliminate them completely. The main reason is the frequent dynamic allocation of *P* and *X* sets. Every recursive call needs to create multiple pairs of these sets that might live and be deallocated in different tasks. Our work reduces the memory management overheads by allocating and deallocating memory needed by multiple sets at once. Each task pre-allocates a memory block, in which all the sets created by the task are stored. The memory block is deallocated once all of the sets in the block are no longer needed.

III. Experimental results

In this section we evaluate the performance of our implementation. First, we discuss how each optimization affects the different components of the execution time, such as time spent on set operations, memory access coordination, and task scheduling. Next, we show how well our implementation scales on a modern manycore CPU. Afterwards, we compare our implementation to a state-of-the-art multi-core implementation of the BK algorithm by Das et al. [10], [11].

[1] Intel, Intel Xeon, Intel Xeon Phi, Threading Building Blocks and VTune are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

TABLE I: Properties of the graph datasets.

dataset	#vertices (M)	#edges (M)	degeneracy	size (MB)
wiki-Talk	2.4	5	131	64
B-anon	2.9	21	63	80
as-skitter	1.7	11	111	143
livejournal	4.0	28	213	393
wiki-topcats	1.8	29	99	403
Pokec	1.6	31	47	405
orkut	3.1	117	253	1740

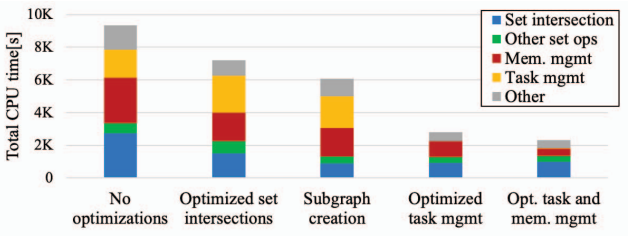


Fig. 2: Impact of each performance optimization on the total CPU time when using 256 threads to mine the *wiki-Talk* graph.

We use the second generation Intel Xeon Phi 7210 processor, i.e., the Knights Landing (KNL) [19]. It is a manycore processor with 64 cores and four NUMA regions (16 cores share a memory region). Each core can execute four simultaneous threads, so the CPU can run 256 simultaneous threads. In addition, Intel KNL features a high-bandwidth memory and the state-of-the-art vector instruction set (i.e., AVX-512).

To build our code, we use version 8.3.1 of the GCC compiler using *-O3* optimization flag. To exploit task parallelism, we use version 2019_U9 of the Intel TBB framework. For profiling, we use Intel VTune Amplifier version 2018.3.

In our set intersection implementations, the bucket size is equal to the L2 cache line size (64B). Each bucket contains 14 vertex identifiers and a pointer to the next bucket. We use Intel's AVX-512 vector instructions to accelerate set intersections. The datasets we use are summarized in Table I and come from the *Network Data Repository* [20] and *SNAP* [2].

Figure 2 shows the impact of each optimization when executing our BK implementation on the *wiki-Talk* [2] graph using 256 threads. Our baseline without optimizations is *std::unordered_set*, which supports lookups in $O(1)$ time complexity. Intersections are performed simply by iterating through the smaller set and performing lookups in the larger set. Our data-parallel set intersection implementation achieves a two-fold speed-up with respect to this unoptimized baseline. The improvement comes both from the use of AVX-512 instructions and the reduction of the number of cache misses. The intersection time is further reduced 1.7 times by creating subgraphs in the first level of the recursion tree. In addition, combining several recursive calls into a single task eliminates the task management overheads. However, this optimization reduces the memory management overheads as well. Because the tasks are dynamically allocated, reducing the frequency of

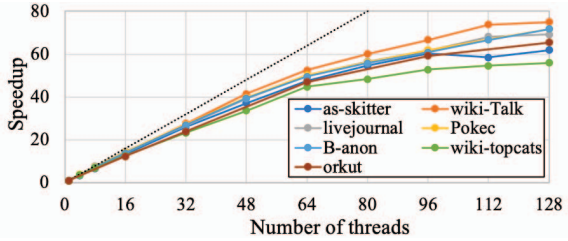
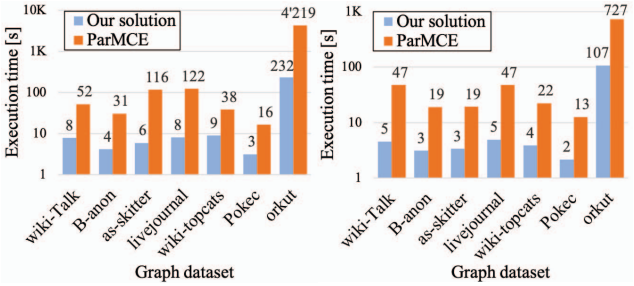


Fig. 3: Speed-up w.r.t. single-threaded execution on KNL.



(a) KNL: 256 threads (b) Xeon Skylake: 96 threads

Fig. 4: Comparisons with the TBB-based implementation of *ParMCE* [11]. The run-times are given above the bars in secs.

task creation indirectly reduces the time to allocate the P and X sets. Lastly, Fig. 2 shows that the time spent on memory management is further reduced by pre-allocating the memory space needed to store all the P and X sets created by a task.

Figure 3 shows the scalability of our implementation by reporting the speedup compared to the single thread case. The algorithm scales almost linearly until 64 threads, which is exactly the number of physical cores of the Intel KNL. After that point, KNL uses simultaneous multi-threading, which improves the performance sublinearly. There is little benefit of using more than 128 threads, so we omit these data points.

Finally, we provide comparisons with the state-of-the-art multi-core implementation by Das et al. [10], [11]. In addition to KNL, we show results on the Intel Xeon Skylake processor, which incorporates 48 physical cores and supports two simultaneous threads per core. This processor also supports the AVX-512 instructions exploited by our work. Fig. 4 shows that the largest performance improvements we achieve on the KNL and Skylake processors are respectively 19- and 9-fold.

IV. Conclusions and Future Work

We presented a scalable multi-core implementation of the BK algorithm that achieves an up to 19-fold speed-up compared to a recent solution [11]. Such an improvement was enabled by minimizing the time spent on set intersections in the single-threaded implementation and by eliminating the performance bottlenecks of the multi-threaded implementation.

Currently, we are evaluating the impact of various vertex ordering methods and set-intersection algorithms on both the theoretical complexity and the practical performance of the BK algorithm. Our future work will explore exploitation of high-bandwidth memories to improve the performance further [21].

Acknowledgment

We would like to thank the authors of *ParMCE* [10], [11] for sharing the source code of their TBB-based implementation.

References

[1] C. C. Aggarwal and H. Wang, Eds., *Managing and Mining Graph Data*, ser. Advances in Database Systems. Boston, MA: Springer US, 2010, vol. 40.

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

[3] L. Bergstrom, "Measuring NUMA effects with the STREAM benchmark," *arXiv:1103.3225 [cs]*, pp. 1–11, Mar. 2011.

[4] Z. Lu, J. Wahlström, and A. Nehorai, "Community Detection in Complex Networks via Clique Conductance," *Sci Rep*, vol. 8, no. 1, Dec. 2018.

[5] H. Yu, A. Paccanaro, V. Trifonov, and M. Gerstein, "Predicting interactions in protein networks by completing defective cliques," *Bioinformatics*, vol. 22, no. 7, pp. 823–829, Apr. 2006.

[6] 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*, vol. 2011, p. 284909, 03 2011.

[7] 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*, vol. 26, no. 10, pp. 1367–1372, Oct. 2004.

[8] Xifeng Yan and Jiawei Han, "gSpan: graph-based substructure pattern mining," in *2002 IEEE International Conference on Data Mining, 2002. Proceedings.* Maebashi City, Japan: IEEE Comput. Soc, 2002, pp. 721–724.

[9] R. A. Rossi, D. F. Gleich, and A. H. Gebremedhin, "Parallel Maximum Clique Algorithms with Applications to Network Analysis," *SIAM Journal on Scientific Computing*, vol. 37, no. 5, pp. C589–C616, Jan. 2015.

[10] 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)*. Bengaluru, India: IEEE, Dec. 2018, pp. 62–71.

[11] ——, "Shared-Memory Parallel Maximal Clique Enumeration from Static and Dynamic Graphs," *arXiv:2001.11433 [cs]*, pp. 1–28, Jan. 2020.

[12] C. Bron and J. Kerbosch, "Algorithm 457: finding all cliques of an undirected graph," *Communications of the ACM*, vol. 16, no. 9, pp. 575–577, Sep. 1973.

[13] E. Tomita, A. Tanaka, and H. Takahashi, "The worst-case time complexity for generating all maximal cliques and computational experiments," *Theoretical Computer Science*, vol. 363, no. 1, pp. 28–42, Oct. 2006.

[14] D. Eppstein, M. Löffler, and D. Strash, "Listing all maximal cliques in sparse graphs in near-optimal time," in *Algorithms and Computation*. Springer Berlin Heidelberg, 2010, pp. 1–13.

[15] 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*. Houston, TX, USA: ACM Press, 2018, pp. 1587–1602.

[16] H. Chen, M. Liu, Y. Zhao, X. Yan, D. Yan, and J. Cheng, "G-Miner: an efficient task-oriented graph mining system," in *Proceedings of the Thirteenth EuroSys Conference on - EuroSys '18*. Porto, Portugal: ACM Press, 2018, pp. 1–12.

[17] A. Kukanov, "The Foundations for Scalable Multicore Software in Intel Threading Building Blocks," *ITJ*, vol. 11, no. 04, Nov. 2007.

[18] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, Sep. 1999.

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

[20] R. A. Rossi and N. K. Ahmed, "The Network Data Repository with Interactive Graph Analytics and Visualization," in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, ser. AAAI'15. AAAI Press, 2015, p. 4292–4293.

[21] O. Green, J. Fox, J. Young, J. Shirako, and D. Bader, "Performance Impact of Memory Channels on Sparse and Irregular Algorithms," *arXiv:1910.03679 [cs]*, pp. 1–10, Oct. 2019.