Parallelizing Maximal Clique Enumeration on Modern Manycore Processors

Jovan Blanuša†, Radu Stoica†, Paolo Ienne†, Kubilay Atasu‡
†IBM Research - Zürich, ‡École 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
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 \( v \).

At each recursive call, the next 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 (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 BK Pivot function is defined as a task. In each iteration of the foreach loop shown in line 14 of Algorithm 1, memory for the subsets \( P' = \overline{P} \cap N_G(v) \) and \( X' = \overline{X} \cap N_G(v) \) 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 BK Degeneracy 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].

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 2018.3 of 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

<table>
<thead>
<tr>
<th>dataset</th>
<th>#vertices (M)</th>
<th>#edges (M)</th>
<th>degeneracy</th>
<th>size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>wiki-Talk</td>
<td>2.4</td>
<td>1131</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>B-anon</td>
<td>2.9</td>
<td>1111</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>as-skitter</td>
<td>1.7</td>
<td>1111</td>
<td>143</td>
<td></td>
</tr>
<tr>
<td>livejournal</td>
<td>4.0</td>
<td>213</td>
<td>393</td>
<td></td>
</tr>
<tr>
<td>wiki-topcats</td>
<td>1.8</td>
<td>99</td>
<td>405</td>
<td></td>
</tr>
<tr>
<td>Pokec</td>
<td>1.6</td>
<td>47</td>
<td>405</td>
<td></td>
</tr>
<tr>
<td>orkut</td>
<td>3.1</td>
<td>253</td>
<td>1740</td>
<td></td>
</tr>
</tbody>
</table>

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