Practical Parallel Algorithms for Near-Optimal Densest Subgraphs on Massive Graphs

Pattara Sukprasert*† Quanquan C. Liu‡ Laxman Dhulipala§ Julian Shun¶

Abstract

The densest subgraph problem has received significant attention, both in theory and in practice, due to its applications in problems such as community detection, social network analysis, and spam detection. Due to the high cost of obtaining exact solutions, much attention has focused on designing approximate densest subgraph algorithms. However, existing approaches are not able to scale to massive graphs with billions of edges.

In this paper, we introduce a new framework that combines approximate densest subgraph algorithms with a pruning optimization. We design new parallel variants of the state-of-the-art sequential `Greedy++` algorithm, and plug it into our framework in conjunction with a parallel pruning technique based on k-core decomposition to obtain parallel $(1+\varepsilon)$-approximate densest subgraph algorithms. On a single thread, our algorithms achieve 2.6–34× speedup over `Greedy++`, and obtain up to 22.37× self-relative parallel speedup on a 30-core machine with two-way hyper-threading. Compared with the state-of-the-art parallel algorithm by Harb et al. [NeurIPS'22] , we achieve up to a 114× speedup on the same machine. Finally, against the recent sequential algorithm of Xu et al. [PACMMOD'23] , we achieve up to a 25.9× speedup. The scalability of our algorithms enables us to obtain near-optimal density statistics on the `hyperlink2012` (with roughly 113 billion edges) and `clueweb` (with roughly 37 billion edges) graphs for the first time in the literature.

1 Introduction

The densest subgraph problem is a fundamental problem in graph mining that has been studied extensively for decades, both because of its theoretical challenges and its practical importance. The numerous applications of the problem include community detection and visualization in social networks [1, 15, 27, 31, 33, 42], motif discovery in protein and DNA [21, 25, 45], and pattern identification [2, 22, 29].

Significant effort has been made in the theoretical computer science community in computing exact and approximate densest subgraphs under various models of computation, in particular in the static [10, 11, 13, 32, 46], streaming [7], distributed [4, 26, 44], parallel [5, 19, 17, 28], dynamic [7, 12, 14, 43], and privacy-preserving [20, 24, 39] settings. However, despite a plethora of theoretical improvements on these fronts, there still does not exist practical near-optimal densest subgraph algorithms that can scale up to the largest publicly-available graphs with tens to hundreds of billions of edges. In particular, for the largest such graphs, `hyperlink2012` (with roughly 113 billion edges) and `clueweb` (with roughly 37 billion edges), no previous approximations for the densest subgraph were known that are better than a 2-approximation.

There are two typical approaches for solving the densest subgraph problem exactly. The first is to solve a combinatorial optimization problem using a linear program solver. The other is to set up a flow network with size polynomial in the size of the original graph, and then run a maximum flow algorithm on it. However, the caveat to both approaches is that they are not scalable to modern massive graphs; namely, both approaches have large polynomial runtimes and the best theoretical algorithms for these approaches are often not practical. Because of this bottleneck, many have instead investigated approaches for approximate densest subgraphs.

The best-known approximation algorithms for the densest subgraph problem fall into two categories. The first category contains parallel approximation algorithms, which work by iteratively removing carefully chosen subsets of low-degree vertices while computing the density of the induced subgraph of the remaining vertices; then, the induced subgraph with the largest density is taken as the approximate densest subgraph [5, 7, 11] using poly$(\log n)$ rounds of peeling vertices with degree smaller than some threshold. Unfortunately, such methods give $(2+\varepsilon)$-approximations at

*Databricks, San Francisco, CA

†A significant part of this work was done while P.S. was a Ph.D. candidate and Q.C. Liu was a postdoc at Northwestern University.

‡Simons Institute at UC Berkeley, Berkeley, CA

§University of Maryland, College Park, MD

¶MIT CSAIL, Cambridge, MA

best and no one has thus far made such methods work in poly$(\log n)$ rounds and give better approximations.

The second category consists of algorithms obtained from the *multiplicative weight update (MWU)* method. The multiplicative weight update framework approximately solves an optimization problem by using expert oracles to update the weights assigned to the variables multiplicatively and iteratively over several rounds depending on how the experts performed in previous rounds. The MWU framework allows for obtaining $(1 + \varepsilon)$-approximate densest subgraphs in poly$(\log n)$ iterations; however, it requires more work than the peeling algorithm per iteration to update the weights of the variables. As such, neither approach is particularly scalable to massive graphs.

In terms of practical solutions, Boob et al. [10] present a fast, sequential, iterative peeling algorithm called `Greedy++` that combines peeling with the MWU framework. Chekuri et al. [13] show that running `Greedy++` for $\Theta(\frac{\Delta \log n}{\rho^* \varepsilon^2})$ iterations results in a $(1+\varepsilon)$-approximation of the densest subgraph, where ρ^* is the density of the densest subgraph. However, `Greedy++` is not parallel, and does not take advantage of modern multi-core and multiprocessor architectures. Recently, Harb et al. [28] proposed an iterative algorithm based on *projections* that solves a quadratic objective function with linear constraints derived from the dual of the densest subgraph linear program of Charikar [11]. For a graph with m edges and maximum degree Δ, they prove that their algorithm converges to a $(1 + \varepsilon)$-approximation in $O(\sqrt{m\Delta}/\varepsilon)$ iterations, where each iteration takes $O(m)$ work.

Xu et al. [48] recently introduce a framework for a generalized version of the densest subgraph problem that includes variants like the densest-at-least-k-subgraph problem. Their framework alternates between iteratively using maximum flow to obtain denser subgraphs and then peeling according to the k-core to shrink the graph for the next maximum flow iteration. However, their algorithm is not parallel and, thus, cannot scale to the largest publicly available graphs. Parallel implementations exist that give 2-approximations on the densest subgraph [18, 19, 36], but such algorithms and implementations achieve worse theoretical approximation guarantees than our $(1 + \varepsilon)$-approximation algorithms. We also demonstrate that they obtain worse empirical approximations.

In our work, we design fast practical algorithms that simultaneously make use of parallelism as well as the closely related concept of the *k-core decomposition.* The k-core decomposition decomposes the graph into *k-cores* for different values of k. Within the induced subgraph of each k-core, each vertex has degree at least k. It is a well-known fact that the density of the densest subgraph is within a factor of 2 of the maximum core value. However, it is less clear how to make use of this fact in creating scalable algorithms for the largest publicly-available graphs. In this paper, we design a pruning framework that, combined with our parallel densest subgraph subroutines, results in both theoretical as well as practical improvements over the state-of-the-art. The main idea of our framework is to iteratively prune the graph using lower bounds on the density of densest subgraph computed from our parallel densest subgraph subroutines, while preserving the densest subgraph.

The concept of using pruning to obtain a smaller subgraph from which to approximate the densest subgraph is also used in some recent works [23, 48]. However, in their works, the pruning procedures they use are *inherently sequential.* Compared to previous work, we introduce a *parallel, iterative* pruning approach in this paper and demonstrate via our comprehensive experimentation that our algorithms are more efficient and more scalable than all previous baselines.

Specifically, we give parallel peeling-based MWU and sorting-based MWU iterative algorithms that use pruning and are based on `Greedy++` [10]. Our algorithms achieve the same theoretical number of iterations as Chekuri et al. [13], but is more amenable to parallelization. Experimentally, on an 30-core machine with hyperthreading, our parallel sorting-based algorithm outperforms our parallel peeling-based algorithm, as well as previous state-of-the-art algorithms on most graphs. For instance, compared with the state-of-the-art parallel algorithm by Harb et al. [28], we achieve up to a 114× speedup on the same machine.

Leveraging the scalability of our parallel algorithms, we provide a number of previously unknown graph statistics and graph mining results on the largest of today's publicly available graphs, `hyperlink2012` and `clueweb`, using commodity multicore machines. We also provide statistics (such as the empirical *width*) that may prove to be interesting and useful in aiding future work on this topic.

2 Preliminaries

Given an undirected, unweighted graph $G = (V, E)$, let $n = |V|$ and $m = |E|$. Let $\deg_G(v)$ be the degree of vertex v in G. We define the ***density*** of G to be $\rho(G) = \frac{|E|}{|V|}$. The goal of the ***densest subgraph*** problem is to find a subgraph $S \subseteq G$, such that $\rho(S)$ is maximized. We will use S^* to denote a densest subgraph of G with maximum density ρ^*.

A central structure that we study is the k-core of an undirected graph. We now define k-core formally.

Definition 1. (k-core) *A k-core $core(G, k)$ of G is*

Symbol	Meaning
$G = (V, E)$	undirected, unweighted input graph
n, m	number of vertices, edges resp.
$\deg(v)$	current degree of vertex v
Δ	current maximum degree of graph
c_p	Peeling complexity
$\rho(G)$	current density of graph G
ρ^*	maximum induced subgraph density of graph G
$\tilde{\rho}$	the best density found in our algorithms
$core(G, k)$	k-core of G
$core(v)$	core number of v
k_{max}	max non-empty core number
$\ell(v)$	current load of vertex v

Table 1: Common notation used throughout the paper.

defined to be a maximal vertex-induced subgraph $S \subseteq G$ such that $\deg_S(v) \geq k$ *for any* $v \in V(S)$.

It is well known that to find $core(G, k)$, one can repeatedly *peel*[1] an arbitrary vertex v from G so long as $\deg_G(v) < k$. This process terminates when all remaining vertices have degree at least k, or the graph becomes empty. If the remaining graph is not empty, then it is the unique subgraph, $core(G, k)$. Next, we define the *coreness* or *core number* of a vertex v:

Definition 2. (Core number) *For any vertex v, we let $core(v) = k$ if k is the maximum integer such that v is in $core(G, k)$.*

An easy modification of the peeling algorithm described above yields $core(v)$ for all vertices v. We call this peeling-based algorithm ***Coreness***. In this algorithm, we pick a vertex with minimum degree and peel it one at a time until there are no vertices left. Let D be a variable that represents the maximum degree of peeled vertices at the time we peel them. Initially, $D = 0$. Once v is about to be peeled, we set $D \leftarrow \max(D, \deg_G(v))$. We then set $core(v) \leftarrow D$ and peel v from G. We refer to the ordering of vertices that we peel in this process as a ***degeneracy ordering*** of the graph, which is unique up to permuting vertices in the order with the same coreness.

We also use the following notion of c-approximate k-core decomposition, which can be computed more efficiently than exact k-core.

Definition 3. (c-Approx k-Core Decomposition) *A **c-approximate k-core decomposition** is a partition of vertices into layers, such that a vertex v is in approximate core $\hat{k}(v)$, denoted $apxcore(G, \hat{k}(v))$, only if $\frac{k(v)}{c} \leq \hat{k}(v) \leq ck(v)$, where $k(v)$ is the coreness of v.*

[1]Throughout this paper, we say a vertex v is *peeled* from G when v and all its adjacent edges are deleted.

Later on, we will want to find an ordering that is similar to the degeneracy ordering, but certain *loads* of vertices are also given as input. Let $\ell(v)$ be *load* of v. At each step, we peel the vertex that minimizes the term $\ell(v) + \deg_G(v)$. Note that after v is peeled, the induced degrees $\deg_G(v')$ of v's neighbors v' are decreased. As a special case, we obtain the degeneracy ordering by setting $\ell(v) = 0$ for all v. For the remainder of this paper, we refer to the ordering obtained using $\ell(v) + \deg_G(v)$ as the ***load ordering***.

Model Definitions. We analyze the theoretical efficiency of our parallel algorithms in the ***work-depth*** model [16, 30]. In this model, the ***work*** is the total number of operations executed by the algorithm and the ***depth*** (parallel time) is the longest chain of sequential dependencies. We assume that concurrent reads and writes are supported in $O(1)$ work/depth. A ***work-efficient*** parallel algorithm is one with work that asymptotically matches the best-known sequential time complexity for the problem. All of our algorithms presented in this paper are work-efficient. We say that a bound holds ***with high probability (whp)*** if it holds with probability at least $1 - 1/n^c$ for any $c \geq 1$.

We use the following parallel primitives in our algorithms: ***ParFor***, ***SuffixSum***, ***FindMax***, ***Bucketing***, and ***IntegerSort***. Each primitive takes a sequence A of length n. ***ParFor*** is a parallel version of a for-loop that we use to apply a function f to each element in the sequence. If a function f takes $O(t)$ work and $O(d)$ depth, then ***ParFor*** takes $O(tn)$ work and $O(d)$ depth. ***SuffixSum*** returns a sequence B where $B[j] = \sum_{i=j}^{n} A[i]$. ***FindMax*** returns an element with maximum value among those in the sequence. ***SuffixSum*** and ***FindMax*** can be implemented to take $O(n)$ work and $O(\log n)$ depth. ***IntegerSort*** returns a sequence in sorted order (either non-increasing or non-decreasing order) according to integer keys. We use two different implementations of ***IntegerSort***: the first is an algorithm by Raman [40] which takes $O(n \log \log n)$ expected work and $O(\log n)$ depth whp, and the second is a folklore algorithm that takes $O(n/\varepsilon)$ work and $O(n^\varepsilon)$ depth for $0 < \varepsilon < 1$ [47]. The decision to use one of these two sorting algorithms depends on whether work or depth is more important. We state the complexity of our algorithm in both ways when necessary.

2.1 Pruning with Cores

In this section, we describe a pruning idea that takes an input graph G and outputs a subgraph $H \subseteq G$ such that (1) H is smaller than G and (2) any densest subgraph $S^* \subseteq G$ is in H. We begin with a property that relates a graph's density and its vertices' degrees.

Lemma 2.1. (Folklore, (see, e.g., [13])) *Given*

$G = (V, E)$, if there is a vertex v with degree $\deg_G(v) < \rho(G)$, then $G' = G \setminus \{v\}$ is a graph with density $\rho(G') > \rho(G)$.

Proof. It holds that

$$\begin{aligned}\rho(G) &= \frac{|E(G')| + \deg_G(v)}{|V|} \\ &= \frac{|V|-1}{|V|} \cdot \frac{|E(G')|}{|V|-1} + \frac{\deg_G(v)}{|V|} \\ &= \frac{|V|-1}{|V|} \cdot \rho(G') + \frac{\deg_G(v)}{|V|} \\ &= x \cdot \rho(G') + (1-x) \cdot \deg_G(v),\end{aligned}$$

for some real number $x \in (0, 1)$. The last line can be viewed as a weighted average between $\rho(G')$ and $\deg_G(v)$. Since $\deg_G(v) < \rho(G)$, it has to be the case that $\rho(G') > \rho(G)$ so that their average becomes $\rho(G)$. $\square$

As a corollary, any vertex in a densest subgraph has induced degree at least $\rho(S^*)$.

Corollary 2.1. *Let S^* be the densest subgraph. Then for any $v \in V(S^*)$, $\deg_G(v) \geq \deg_{S^*}(v) \geq \lceil \rho^* \rceil$.*

Corollary 2.1 follows immediately from Lemma 2.1 because vertices v with $\deg_G(v) < \rho^*$ can be peeled while increasing the density of the remaining subgraph. Thus, a natural procedure that we have for obtaining the densest subgraph is to iteratively remove any vertex v that has degree less than the current density of the subgraph. Notice that this process is very similar to the algorithm for computing $core(G, k)$ described in Section 2. In fact, we can relate k-core to the densest subgraph.

Lemma 2.2. *For some $k \leq \lceil \rho^* \rceil$, let $C = core(G, k)$ be the k-core of G. It must be the case that $S^* \subseteq C$.*

Proof. We prove this by contradiction. Assume that $S^* \setminus C$ is non-empty (i.e., there is a vertex in S^* but not in C). Let $H = S^* \cup C$. Notice that, for any vertex $v \in S^* \cup C$, it holds that $\deg_H(v) \geq k$—if $v \in S^*$, then $\deg_H(v) \geq \deg_{S^*}(v) \geq \lceil \rho^* \rceil \geq k$ by Corollary 2.1, and if $v \in C$, then $\deg_H(v) \geq \deg_C(v) \geq k$. Hence, $S^* \cup C$ is a k-core with more vertices than C, implying that C is not maximal, which is a contradiction. $\square$

Corollary 2.2. (Folklore) *Let k_{max} be the maximum integer such that the $core(G, k_{max})$ is not empty. Let C be the $\lceil \frac{k_{max}}{2} \rceil$-core. Then $S^* \subseteq C$.*

Proof. For any v in $C = core(G, k_{max})$, we have $\deg_S(v) \geq k_{max}$. Hence,

$$\rho(C) = |E(C)|/|V(C)| \geq \frac{\sum_{v \in V_C} \deg_C(v)/2}{|V(C)|} \geq k_{max}/2.$$

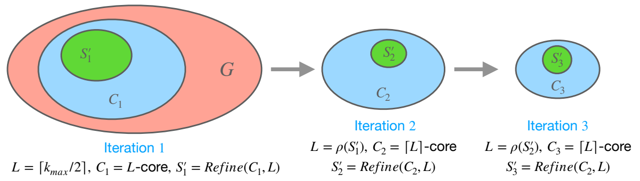
S'_1
C_1
G
S'_2
C_2
S'_3
C_3
Iteration 1
$L = \lceil k_{max}/2 \rceil$, $C_1 = L$-core, $S'_1 = Refine(C_1, L)$
Iteration 2
$L = \rho(S'_1)$, $C_2 = \lceil L \rceil$-core
$S'_2 = Refine(C_2, L)$
Iteration 3
$L = \rho(S'_2)$, $C_3 = \lceil L \rceil$-core
$S'_3 = Refine(C_2, L)$

Figure 1: Example illustrating the Pruning-and-Refining Framework (Algorithm 1). The i-th iteration of the algorithm computes a lower bound L on the density, computes the $C_i = \lceil L \rceil$-th core of G, and then applies a $Refine$ algorithm on C_i to compute a new subgraph S'_i. In the example, the density of each successive S'_i is increasing, and the cores C_i decrease in size.

Thus, $\rho^* \geq \rho(C) \geq k_{max}/2$. It follows from Lemma 2.2 that S^* is contained in the $\lceil \frac{k_{max}}{2} \rceil$-core. $\square$

Similarly, the largest non-empty c-approximate k-core, $\hat{k}_{max}$, also gives us a lower bound on ρ^*, in terms of the density of a (potentially larger) approximate core with smaller approximate core number:

Corollary 2.3. *Let $\hat{k}_{max}$ be the maximum integer such that the $apxcore(G, \hat{k}_{max})$ is not empty. Let C be the $\lceil \frac{\hat{k}_{max}}{2c} \rceil$-approximate core. Then $S^* \subseteq C$.*

Proof. The proof is identical to that of Corollary 2.2; the only difference is that since the core is approximate, the lower bound on $\deg_S(v)$ for any v in $apxcore(G, \hat{k}_{max})$, is $\hat{k}_{max}/c$. $\square$

3 Pruning-and-Refining Framework

Based on the properties described in Section 2, any algorithm that yields a lower bound on ρ^* can be used for pruning the graph while retaining the densest subgraph. The main idea of the framework is as follows. Let L be a lower bound on ρ^*. We can prune the input graph G by computing G', which is the $\lceil L \rceil$-core of G and then search for the densest subgraph in G' instead of G. This process can be repeated multiple times, making it useful in algorithms that iteratively refine (tighten) the lower bounds for ρ^* over a sequence of steps.

To the best of our knowledge, the idea of using cores to prune the graph adaptively while refining the approximate densest subgraph solution has not been done in the literature. The closest idea is from Fang et al. [23] and Xu et al. [48]. In [23], their pruning rules first compute ***Coreness***, and then inspect connected components from the $\lceil \frac{k_{max}}{2} \rceil$-core. They take the maximum density found among the connected components as a lower bound and use k_{max} as an upper bound. They then run a flow-based algorithm on each connected component separately. Note that flow-based algorithm can only tell if a graph has a subgraph of a specific density $\tilde{\rho}$, so a binary search over the optimal density is required to solve the densest subgraph problem. Their pruning rules do not help much if there is only a single component in the $\lceil \frac{k_{max}}{2} \rceil$-core. Then, in [48], they give a sequential pruning-based algorithm based on flow where their implementation prunes the graph at the beginning and runs a flow-based algorithm to find approximate densities.

3.1 Framework Overview We apply this idea in an algorithmic framework for computing an approximate densest subgraph, which is shown in Algorithm 1. The pseudocode uses *exact pruning*, i.e., it uses the value of the exact k_{max}-core, but we also describe how to use approximate k-cores below. On Lines 1–4, we compute the lower bound L by applying Corollary 2.2 and either an exact k-core algorithm or an approximate k-core algorithm. Both algorithms take $O(m+n)$ work, but approximate k-core has provably poly-logarithmic depth. For exact k-core, we use the bucketing-based k-core implementation of [18, 19]. The algorithm iteratively peels all vertices with degree at most d in parallel, starting with $d = 0$, and incrementing d whenever there are no more vertices with degree at most d. The algorithm takes $O(m + n)$ expected work and $O(c_p \log n)$ depth with high probability, where c_p is the ***peeling complexity***, which is defined as the number of iterations needed to completely peel the graph. For approximate k-core, we use the implementation of Liu et al. [35], which gives a $(2 + \delta)$-approximation to all core numbers and takes $O(m + n)$ expected work and $O(\log^3 n)$ depth whp. Line 2 shows the lower bound L given k_{max}, but we can alternatively compute the lower bound for the approximate k-core approach using Corollary 2.3. Given the coreness values in *cores*, we extract the j-core from G using $getCore(G, cores, j)$ on Line 3.

On Lines 5–10, we iterate for T rounds, where each round calls a function $Refine$ which computes subgraphs with potentially higher density. Note that for algorithms that we use in our paper, our $Refine$ step on Line 6 is oblivious to the parameter L. However, knowing L might be useful for other algorithms (e.g., flow-based algorithms). After each refinement step, we then get a potentially better solution, which we memorize in Line 7–9. In line 10, we leverage this lower bound L by shrinking G to be $\lceil L \rceil-$core. By Lemma 2.2, the densest subgraph $\lceil L \rceil$-core contains densest subgraph S^*. We then return the approximate densest subgraph on Line 11. In Section 3.2, we describe various options for the $Refine$ function.

Algorithm 1: Pruning-and-Refining Framework

```
Input  : an input graph $G = (V, E)$, number
         of iterations $T$
Output: an approximate densest subgraph $S$
1  $cores, k_{max} \leftarrow Coreness(G)$
2  $L \leftarrow \lceil k_{max}/2 \rceil$
3  $G \leftarrow getCore(G, cores, L)$
4  $S \leftarrow G$                    // Initial pruning
5  for $i$=1 to $T$ do
6      $S' \leftarrow Refine(G, L)$  // Refine candidate
        subgraph
7      if $\rho(S') > \rho(S)$ then
8          $S \leftarrow S'$
9          $L \leftarrow \max(L, \rho(S))$
10         $G \leftarrow getCore(G, cores, \lceil L \rceil)$
11 return $S$
```

3.2 Refinement Algorithms Next, we describe algorithms that can be used for the $Refine$ function in Algorithm 1. We first describe two existing sequential algorithms, the peeling algorithm and `Greedy++`, and then introduce our parallel algorithms.

Peeling Algorithm [11]. At each step, we compute the density of the current graph. Then, we pick a vertex v with the minimum induced degree and remove it from the graph. We continue until there are no vertices remaining, and return the subgraph with the maximum density found in this process. The peeling algorithm can be parallelized [18], but can have linear depth in the worst case. Charikar [11] proves that the subgraph returned by this peeling algorithm has a density at least half the optimum density, i.e., it gives a 2-approximation to the densest subgraph.

`Greedy++` [10]. Algorithm 2 presents a greedy load-based densest subgraph algorithm, for which the state-of-the-art `Greedy++` algorithm is a special case. Initially, each vertex v is associated with a load $\ell(v) = 0$ (Lines 2–3). The algorithm runs for T iterations (Lines 4–11). On each iteration, we compute the degeneracy order O with respect to the load ℓ on graph H to obtain the ordered set of vertices $v_1, \ldots, v_n$ (Line 6). Then, on Lines 7–11, we peel vertices in this order. When v_i is peeled, we compare the density of the remaining subgraph to the density of the best subgraph found so far, and save the denser of the two. We also update the loads, setting $\ell(v_i) \leftarrow \ell(v_i) + \deg^*_H(v_i)$, where $\deg^*_H(v_i)$ here is the induced degree of v_i when it is peeled. We return the best subgraph found after $T = \Theta(\frac{\Delta \log n}{\rho^* \varepsilon^2})$ iterations, where Δ is the maximum degree and $0 < \varepsilon < 1$ is an adjustable parameter. This algorithm yields a $(1+\varepsilon)$-approximation of the densest subgraph as shown in [13].

Algorithm 2: Greedy Load-Based Densest Subgraph

```
   Input  : an input graph $G = (V, E)$, number
            of iterations $T$, ordering function $O$
   Output: an approximate densest subgraph $S$
1  for $v$ in $V$ do
2  |  $\ell(v) \leftarrow 0$
3  $H = (V_H, E_H) \leftarrow (V, E)$
4  $S \leftarrow G$
5  for $i$=1 to $T$ do
6  |  let $v_1, \ldots, v_n$ be the ordering provided by
   |   the function $O$
7  |  for $j$=1 to $n$ do
8  |  |  if $\rho(H) > \rho(S)$ then
9  |  |  |  $S \leftarrow H$
10 |  |  $\ell(v_j) \leftarrow \ell(v_j) + \deg^*_H(v_j)$
   |  |     // $\deg^*_H(v_j)$ is degree of $v_j$ when
   |  |   peeled
11 |  |  $H \leftarrow H \setminus \{v_j\}$
12 return $S$
```

The first iteration of `Greedy++` is exactly the peeling algorithm of Charikar.

The original `Greedy++` algorithm is implemented in a way where the degeneracy ordering and the update steps are fused together. It will become clear once we introduce our algorithm below why we decouple these two steps for obtaining greater parallelization.

`GreedySorting++` (our algorithm). Our second algorithm uses a simpler method for computing O than `Greedy++`, in that it orders vertices based on their loads at the *beginning* of the iteration. The motivation for this algorithm is that sorting is highly parallelizable and is also faster in practice than the iterative peeling process used in `Greedy++` (which has linear depth). Therefore, on Line 6 of Algorithm 2, we compute $v_1, \ldots, v_n$, such that $\ell(v_1) \leq \ell(v_2) \leq \ldots \leq \ell(v_n)$. Because of the way we decouple the ordering and update steps in `Greedy++`, Line 6 is the only difference between the two algorithms. Next, we argue that `GreedySorting++` has the same guarantees in terms of the approximation and number of rounds as `Greedy++`.

THEOREM 4. *For $T = \Theta\left(\frac{\Delta \log n}{\rho^* \varepsilon^2}\right)$, `GreedySorting++` outputs a $(1+\varepsilon)$-approximation to the densest subgraph problem.*

Proof. The proof follows almost immediately from Section 4 of [13]. To prove that `Greedy++` works, they define an exponential-sized linear program where each variable corresponds to one possible peeling order (i.e., a permutation). They then utilize the multiplicative weight update (MWU) framework on the linear program.[2] By the way the formulate their linear program, the subproblem that we need to solve w.r.t. MWU is to find a good ordering. Lemma 4.6 of [13] shows that the ordering obtained with `Greedy++` is a *good approximate ordering.* The proof of Lemma 4.6 work for any order $v_1, \ldots, v_n$ that satisfies the following property: $\ell(v_i) \leq \ell(v_j) + \Delta$ if $i < j$. This is true for the ordering used in `GreedySorting++`, where we sort by the initial load of the vertices. Hence, by plugging in this ordering, all of the proofs in [13] go through. □

Algorithm 3: Parallel Density and Load Computation

```
   Input  : an input graph $G = (V, E)$, an
            ordering $v_1, \ldots v_n$, current loads $\ell(\cdot)$
   Output: updated loads $\ell(\cdot)$, best density
            found $\rho_{max}$
1  $A$ := array of size $n$
2  parfor $e = (v_j, v_k)$ in $E$ do
3  |  $A[\min(j,k)] \leftarrow A[\min(j,k)] + 1$
4  $B := SuffixSum(A)$
5  parfor $i = 1$ to $n$ do
6  |  $B[i] \leftarrow B[i]/(n - i + 1)$
7  $\rho_{max} \leftarrow \max_i B[i]$ // maximum density in
    this iteration
8  parfor $v_i$ in $V$ do
9  |  $\ell(v_i) \leftarrow \ell(v_i) + A[i]$ // update loads
10 return $\ell, \rho_{max}$
```

3.3 Parallel Implementation

In this subsection, we present our parallelizations of `Greedy++` and `GreedySorting++`. We still run both algorithms for T iterations, one iteration at a time, and our aim is to achieve low depth within each iteration.

Parallelization of Algorithm 2. We first describe how to parallelize all parts of Algorithm 2 except for Line 6. Let $v_1, \ldots, v_n$ be an ordering of vertices for peeling. On the i'th iteration of the for-loop on Line 7, the induced subgraph of G that we use is $S_i = G(v_i, \ldots, v_n)$. This holds for all $1 \leq i \leq n$. For any edge $e = (v_j, v_k)$, e will contribute to the density of S_i if and only if $i \leq j$ and $i \leq k$.

Our implementation for computing the densities and updating the loads in parallel is shown in Algorithm 3. We first initialize an empty array A of size n (Line 1). Then, for each edge $e = (v_j, v_k)$, we add 1 to $A[\min(j,k)]$ (Lines 2–3). Let B be the suffix sum array of A (Line 4). Then, $B[i]$ corresponds to the number of edges remaining in the graph after vertices $v_1, \ldots, v_{i-1}$ are peeled. To see why it is the case, let us consider the remaining subgraph after $v_1, \ldots, v_{i-1}$ are peeled.

[2]See, e.g., [3] for a survey on this topic.

Consider an edge $e = (v_j, v_k)$. Edge e will appear in this subgraph if and only if both v_j and v_k are not yet peeled, i.e., $i \leq j$ and $i \leq k$. We add 1 to $A[\min(j,k)]$ to account for the presence of e in the subgraphs induced by $v_{i \leq \min(j,k)}, \dots, v_n$. We then compute the densities in parallel and take the maximum density on Lines 5–7. We update the loads in parallel on Lines 8–9. The work and the depth of this implementation are $O(n+m)$ and $O(\log n)$, respectively. As we described in Section 2, ***ParFor***, ***SuffixSum***, and ***FindMax*** all take linear work. ***SuffixSum*** and ***FindMax*** have $O(\log n)$ depth, and ***ParFor*** have $O(1)$ depth, so our depth bound follows.

`ParallelGreedy++`. In order to parallelize `Greedy++`, what is left for us is to parallelize the computation of the degeneracy ordering, which can be computed with a k-core decomposition algorithm in $O(m+n)$ expected work and $O(c_p \log n)$ depth, with high probability. When we peel multiple vertices at the same time, the degeneracy ordering can be different from the order obtained sequentially. The reason is that when a vertex is peeled in the sequential algorithm, it affects its neighbors' degrees immediately. However, in the parallel version, this effect is delayed until the end of the peeling step where multiple vertices may be peeled together. We claim that this does not significantly affect the order. Consider a pair of vertices v_i and v_j. To make the proof in Theorem 4 go through, it suffices to show that $i < j$ implies $\ell(v_i) \leq \ell(v_j) + \Delta$. We prove the contrapositive. Suppose $\ell(v_i) > \ell(v_j) + \Delta$. Because the number of neighbors of both v_i and v_j are bounded by Δ, it is the case that, even if all of v_i's neighbors are peeled and none of v_j's neighbors are peeled, v_i will still be peeled after v_j, implying that $i > j$. Therefore, for $i < j$ we have that $\ell(v_i) \leq \ell(v_j) + \Delta$.

THEOREM 5. *For $T = \Theta\left(\frac{\Delta \log n}{\rho^* \varepsilon^2}\right)$, our parallel algorithm **`ParallelGreedy++`** outputs a $(1+\varepsilon)$-approximation to the densest subgraph problem. Moreover, each iteration takes $O(n+m)$ expected work and $O(c_p \log n)$ depth with high probability.*

`ParallelGreedySorting++`. We replace the degeneracy order in `ParallelGreedy++` with parallel ***IntegerSort*** to obtain `ParallelGreedySorting++`. As discussed in Section 2, integer sorting takes $O(n \log\log n)$ expected work and $O(\log n)$ depth whp, or $O(n/\varepsilon)$ work and $O(n^\varepsilon)$ depth for any $0 < \varepsilon < 1$. We proved earlier that sorting does not affect the approximation guarantee of the algorithm. Therefore, we have the following theorem.

THEOREM 6. *For $T = \Theta\left(\frac{\Delta \log n}{\rho^* \varepsilon^2}\right)$, our parallel **`ParallelGreedySorting++`** outputs a $(1+\varepsilon)$-approximation to the densest subgraph problem. Moreover, each iteration takes either $O(n \log\log n + m)$ expected work and $O(\log n)$ depth whp or $O(n+m)$ work and $O(n^\varepsilon)$ depth for any $0 < \varepsilon < 1$.*

Note that for our algorithms, the number of steps T depends on ρ^*, but we can choose T based on k_{max} instead, as it is within a factor of 2 of ρ^*.

Combining Refinement with Pruning. Using the framework in Algorithm 1, we can combine a pruning method with *one iteration* of `Greedy++`, `GreedySorting++`, `ParallelGreedy++`, and `ParallelGreedySorting++` as the *Refine* function. For example, we can combine approximate k-core for pruning with one iteration of `ParallelGreedySorting++`, which would give an algorithm with either $O(T(n \log\log n + m))$ expected work and $O(T \log n + \log^3 n)$ depth or $O(T(n+m))$ work and $O(T \log n + n^\varepsilon)$ depth for any $0 < \varepsilon < 1$.

Remark. While pruning gives speedups in practice, it does not improve the theoretical complexity of the algorithm, as there exists a graph where the core that we prune down to covers most of the original graph. However, as we observe in our experiments below, pruning results in massive improvements in runtime in practice as most real-world graphs exhibit a densest subgraph that is a small percentage of the input (in some graphs, the densest subgraph contains fewer than 1% of the vertices in the input).

4 Experiments

In this section, we implement and benchmark different instantiations of the Pruning-and-Refining framework on real-world datasets. We also compare our algorithms with existing algorithms. We demonstrate that our approach is practical and is scalable to the largest publicly-available graphs. In addition, we also provide interesting statistical data on large-scale graphs, in particular, near-optimal densities of the larger graphs, previously not reported to such accuracy in literature. All of our code is provided at this link.[3] There is some experimental data omitted due to space constraints. They are included in the full version of our paper.

Implementations. We implement `Greedy++`, `GreedySorting++`, and their parallel instantiations as our refinement algorithms. We consider two algorithms for pruning: pruning using exact k-cores, and pruning using approximate k-cores. Both pruning algorithms are parallel and are modular across all of the refinement algorithms. We use the exact k-core decomposition

[3]https://github.com/PattaraS/gbbs/tree/ALENEX

algorithm from Dhulipala et al. [18] and the approximate k-core decomposition algorithm from Liu et al. [35] for our pruning step. The algorithm of Liu et al. [35] allows us to specify the approximation ratio (c in Definition 3). When c is higher, the algorithm tends to be more parallelizable but will be less accurate. We run our experiments with $c = 1.5$. We also combine approximate k-core decomposition with exact k-core decomposition for further speedups. In particular, our combined pruning algorithm first uses approximate k-core decomposition to shrink the graph and then uses exact k-core decomposition for greater accuracy in the refine step on the smaller graph. Such a procedure results in speedups for a peeling-based algorithm like `ParallelGreedy++` since in each iteration, the algorithm already performs much of the necessary work (with minimal modification) to find the k-core decomposition. We name our algorithms as follows:

1. `PaRGreedy++`: is `ParallelGreedy++` combined with our Pruning-and-Refining framework with exact k-core pruning.
2. `PaRSorting++`: is `ParallelGreedySorting++` combined with our Pruning-and-Refining framework with exact k-core pruning.
3. `PaRApxGreedy++`: is `ParallelGreedy++` combined with our Pruning-and-Refining framework with approximate k-core pruning followed by exact k-core pruning.
4. `PaRApxSorting++`: is `ParallelGreedySorting++` combined with our Pruning-and-Refining framework with approximate k-core pruning.

We present experimental results for each of our methods and also compare with existing implementations.

Existing Algorithms. We compare with the state-of-the-art $(1 + \varepsilon)$-approximation algorithms from the sequential algorithms of Fang et al. (`CoreExact` [23] and `CoreApp` [23]), the sequential algorithm of Boob et al. (`Greedy++` [10]), the parallel algorithm of Harb et al. (`FISTA` [28], `Frank-Wolfe` [17], and `MWU` [4]), and the sequential algorithm of Xu et al. [48]. We also compare with the state-of-the-art parallel 2-approximation algorithms of Luo et al. [36] and Dhulipala et al. [19].

Fang et al. [23] and Xu et al. [48] implement variants of maximum flow algorithms to find the densest subgraph. Their `CoreExact` [23] and `cCoreExact` [48] implementations are exactly the flow algorithm with binary search over the density. They perform a density lower-bound estimation using cores and approximate maximum flow, which we described in more detail in Section 2.1. If a graph has many connected components, then a subgraph with maximum density lies exclusively in one component. Hence, they run the flow algorithm on each connected component separately. The densities of cores are then used to determine the lower bounds and upper bounds of the binary search needed for the flow computation. `cCoreExact` [48] improves `CoreExact` [23] by using cores to shrink the input graph. Their approximation algorithm, `CoreApp` [23] is an algorithm that finds $core(G, k_{max})$ directly. Once the maximum core is found, they return the component with the highest density. This algorithm yields a 2-approximation for the densest subgraph problem. `cCoreG++` [48] is their implementation of `Greedy++` [10] that uses one iteration of a k-core decomposition algorithm to shrink the input graph.

Harb et al. [28] propose a gradient descent based algorithm called `FISTA` [28], where the number of iterations needed is $O(\sqrt{\Delta m}/\varepsilon)$. They use accelerated proximal gradient descent, which is faster than the standard gradient descent approach [6, 38]. The algorithm runs in iterations, where each iteration can be made parallel. The output in each iteration is a feasible solution to a linear program for the densest subgraph problem. They then use `Greedy++`-inspired rounding, which they call fractional peeling, to round the linear program solution into an integral solution.

Finally, we benchmark against recent parallel algorithms, `Julienne` [19] and `PKMC` [36], for computing the exact k-core decomposition. Then, the maximum core gives a 2-approximation of the densest subgraph. Although these algorithms achieve parallelism, their approximations are worse than the $(1 + \varepsilon)$-approximation algorithms, both theoretically and empirically.

Setup. We use `c2-standard-60` Google Cloud instances (3.1 GHz Intel Xeon Cascade Lake CPUs with a total of 30 cores with two-way hyper-threading, and 236 GiB RAM) and `m1-megamem-96` Google Cloud instances (2.0 GHz Intel Xeon Skylake CPUs with a total of 48 cores with two-way hyper-threading, and 1433.6 GB RAM). We use hyper-threading in our parallel experiments by default. Our programs are written in C++. We use parallel primitives from the `GBBS` [18] and `Parlay` [8] libraries. The source code is compiled using g++ (version 10) with the -O3 flag. We terminate experiments that take over 1 hour. We run each experiment for three times and take the average for the runtime and accuracy analyses. Using enough threads, with our framework, `Greedy++`, `GreedySorting++`, and their parallel versions finished within 1 hour for all of our experiments. However, all other $(1+\varepsilon)$-approximation algorithms took longer than 1 hour on all of the large graphs (`clueweb`, `twitter`, `friendster`, and `hyperlink2012`), so we omit the entries for these datasets for these algorithms.

Datasets. We run our experiments on various synthetic and real-world datasets. The real-world datasets are obtained from SNAP [34] (`cahepth`, `ascaida`, `hepph`, `dblp`, `wiki`, `youtube`, `stackoverflow`, `livejournal`, `orkut`, `twitter`, and `friendster`), Network Repository [41] (`brain`), Lemur project at CMU [9] (`clueweb`), and WebDataCommons [37] (`hyperlink2012`). The `hyperlink2012` graph is the largest publicly-available real-world graph today. `closecliques` is a synthetic dataset designed to be challenging for `Greedy++` [10, 13, 28]. We remove self-loop and zero-degree vertices from all graphs, and symmetrize any directed graphs. We run most of our experiments on `c2-standard-60` machines. However, on the larger graphs (namely, `twitter`, `friendster`, `clueweb`, and `hyperlink2012`), we use `m1-megamem-96` machines as more memory is required. The sizes of our inputs and their maximum core values is included in our full paper.

Overview of Results. We show the following experimental results in this section.

- Our pruning strategy is very efficient in practice as our pruned graph contains 175$\times$ fewer edges on average and 3,596$\times$ fewer vertices on average.
- Our algorithms, similar to the state of the art, take only a few iterations to converge. `PaRSorting++` takes more iterations, but it still converges to < 1.01-approximation within 10–20 iterations and each iteration is significantly faster than all other algorithms.
- Our algorithms are faster than existing algorithms by a large margin.
- Our algorithms are highly parallelizable, achieving up to 22.37$\times$ self-relative parallel speedup on a 30-core machine with two-way hyperthreading.
- We measure empirical "width", which is a parameter that correlates to the number of iterations needed to converge. We observe that the empirical width is much smaller than the upper bound used to analyze the algorithm. This may lead to more fine-grained analyses of many MWU-inspired algorithms.

4.1 Core-Based Pruning

In this section, we present experimental results related to various different pruning methods using exact and approximate k-core decomposition (and combinations thereof).

Pruning with $core(G, \lceil \frac{k_{max}}{2} \rceil)$. We first study the benefit of performing pruning using the *exact* k-core computation. The data for this experiment across all graphs can be found in our full paper. For the real-world graphs, the cores contain between 2–282$\times$ fewer edges than the actual graphs (48.3$\times$ fewer on average), and between 4.5–14227$\times$ fewer vertices (2420$\times$ fewer on average). The only exception is the `brain` dataset, where the core is half the size of the actual graph. Even in this case, the number of vertices left in the core is around 25% of the original graph. For the synthetic dataset `closecliques`, the input is designed so that the maximum-core is identical to the original graph, so there is no benefit to pruning. However, this situation is very unlikely to occur in real-world datasets.

Due to the significant reduction in graph sizes in terms of both the number of vertices and number of edges, using $core(G, \lceil \frac{k_{max}}{2} \rceil)$ is almost always preferable over using G, especially since computing all cores of G (a linear-work algorithm, with reasonably high parallelism in practice [18]) is inexpensive compared to the cost of running any of the refinement algorithms, which mostly require super-linear work. To summarize, we find that pruning is nearly always beneficial and should be applied prior to refinement.

Pruning with Highest Cores. As our algorithms progress, we perform additional pruning to shrink the graph even further. We report the sizes of the final graphs in our full paper. In many cases, the sizes of the final graph (after pruning to the highest cores) are less than half of the sizes of their $\lceil \frac{k_{max}}{2} \rceil$-cores. Across all datasets, we find that iterative pruning yields up to a 30.3$\times$ reduction in the number of vertices over the $core(G, \lceil k_{max}/2 \rceil)$, and up to a 200$\times$ reduction in the number of edges when comparing these quantities in $core(G, \lceil k_{max}/2 \rceil)$ and $core(G, \lceil \tilde{\rho} \rceil)$, where $\tilde{\rho}$ is the best density found by our algorithms. Thus, we see advantages in performing multiple rounds of refinement in certain graphs. Note that, there are cases when this additional pruning step is not helpful, e.g., in `dblp` and `clueweb`. In each of these cases, we notice that the best density found is very close to $\lceil k_{max}/2 \rceil$, so there is no room for pruning opportunities.

4.2 Number of Iterations Versus Density

Next, we study the progress that different refinement algorithms make in our framework vs. other implementations toward finding the maximum density. We perform this experiment on all variants of our algorithms: `PaRGreedy++`, `PaRSorting++`, `PaRApxGreedy++`, and `PaRApxSorting++`; and all other benchmarks that use iterations: `FISTA`, `Greedy++`, `FrankWolfe`, `MWU`, and `cCoreG++`. We also include `PKMC` as a baseline of comparison against a 2-approximation algorithm. All algorithms were run for at least 20 iterations. The results are illustrated in Fig. 2. In fact, most algorithms converge very early with our algorithms `PaRGreedy++` and `PaRApxGreedy++` converging no later than the fastest converging algorithms. In fact, on most graphs, `Greedy++`, `PaRGreedy++`, and `PaRSorting++` took the fewest iterations to converge. Two algorithms, `PaRSorting++` and `MWU`, take more iterations in many graphs. This matches with our understanding of the *width* of MWU as discussed below. Furthermore, `PaRApxGreedy++` and `PaRApxSorting++` exactly match the convergence rates of `PaRGreedy++` and `PaRSorting++`, respectively; such is expected as our use of approximate vs. exact pruning affects the runtime *not* the accuracy. `PaRApxSorting++` needs more iteration to converge since we only use approximate k-core pruning.

4.3 Approximation Ratio We compare the densities returned from various algorithms at iteration 10 with the best density currently known in the literature. Except for `brain`, `twitter`, `friendster`, `clueweb`, and `hyperlink2012`, the best known density is equal to the optimum. To compute the optimum density, we run a linear program solver on $core(G, \lceil \tilde{\rho} \rceil)$.

Except for `FrankWolfe`, all algorithms have approximation ratios less than 1.02 after 10 iterations. Our `PaRGreedy++` algorithm achieves the best approximation ratio after 10 iterations for all four of the largest graphs, `twitter`, `friendster`, `clueweb`, and `hyperlink2012`. For the rest of the graphs, our algorithm achieves an approximation ratio no worse than 1.0001× the smallest approximation ratio. We also include a table that compares densities after iteration 20 in Table 2. After 20 iterations, most approximation ratios are less than 1.001. Our algorithm `PaRGreedy++` obtains the best approximation for 8 out of the 12 tested graphs and obtains an approximation ratio no worse than 1.0000002× the best for the remaining graphs.

4.4 Empirical Widths As mentioned at the end of Section 3.2, in the multiplicative weight update framework, ***width*** is a parameter that is correlated with the number of rounds needed for a solution to converge. In our context, the width ω corresponds to the *maximum increase of a load* of a single vertex in any iteration. See, e.g., [3, 13] for more details on width and its analysis. ω is lower bounded by ρ^* because when we peel the first vertex from the densest subgraph, its degree must be at least ρ^*. The width is also upper bounded by the maximum degree Δ, since the increase of a load of a vertex is bounded by its degree, i.e., $\rho^* \le \omega \le \Delta$. This upper bound is reflected in the $T = O\left(\frac{\Delta \log n}{\rho^* \varepsilon^2}\right)$ iterations needed for our algorithms in the worst case. However, this bound does not reflect reality, as most of our iterative algorithms usually converges in just a few iterations. The bound on the number of iterations could have been $T = O(\frac{\omega \log n}{\rho^* \varepsilon^2})$. This can be significant if $\omega \ll \Delta$. Here, we partially explain this phenomenon by measuring the width empirically. In the full paper, we show information about the width across multiple datasets gathered from our experiments. We observe that the widths for running `PaRGreedy++` are much closer to the best density found, while the widths from `PaRSorting++` are closer to Δ. If we see empirically that $\omega = O(\rho^*)$, then our algorithms should converge in $T = O\left(\frac{\log n}{\varepsilon^2}\right)$ iterations. On the other hand, if $\omega \gg \rho^*$, our algorithms should take more iterations to converge. This supports what we observed in our experiments, and also explains why it takes very few iterations, e.g., fewer than 10–20 iterations, for `PaRGreedy++` to converge.

4.5 Scalability Here, we show the scalability of our algorithms compared to the parallel version of FISTA and the parallel 2-approximation algorithms, `Julienne` and `PKMC` in Fig. 3, which shows the running time of the algorithms (in milliseconds) versus the number of threads used by our algorithms and parallel `FISTA`, `Julienne`, and `PKMC` when each algorithm is run for 5 iterations. We show additional plots for 10 and 20 iterations the full version of our paper. We see that our `PaRSorting++` algorithm achieves greater self-relative speedups than `FISTA` and `PaRGreedy++`. Specifically, `PaRSorting++` achieves up to a 10.6× self-relative speedup (on `livejournal`), while `FISTA` achieves up to a 14× self-relative speedup (on `dblp`) and `PaRGreedy++` achieves up to a 5.51× self-relative speedup (on `orkut`). Furthermore, both of our algorithms take shorter time than parallel `FISTA` regardless of the number of threads. Our `PaRApxGreedy++` and `PaRApxSorting++` achieve the greatest self-relative speedup. Specifically, On `livejournal`, `PaRApxGreedy++` and `PaRApxSorting++` achieves up to a 17.6× and a 20.51× self-relative speedup, respectively. `PaRApxSorting++` achieves greater self-relative speedup than the rest of the implementations for 8 of the 12 tested graphs. Although `PKMC` achieves greater self-relative speedups on `dblp`, `hepph`, `stackoverflow`, and `friendster`, they obtain worse approximations guarantees since they only guarantee a 2-approximation on the density. As we discussed previously, it is much easier to obtain greater parallelism when the approximation guarantee is relaxed to a 2-approximation (we see in Fig. 2 that `PKMC` obtains noticeably worse approximations on most graphs).

4.6 Comparing the Total Running Time We first compare `PaRSorting++` with the algorithms given in [23]. We ran experiments on two of their algorithms, namely, `CoreExact` and `CoreApp`. `CoreExact` took too long to run on most datasets. `CoreApp` is faster than our implementation on some graphs, however, since it uses $core(G, k_{max})$, this algorithm gives a 2-approximation and is less accurate than our algorithms. More details are included in the full version of this paper.

We include plots that compare the total runtime of our algorithms (`PaRGreedy++`,

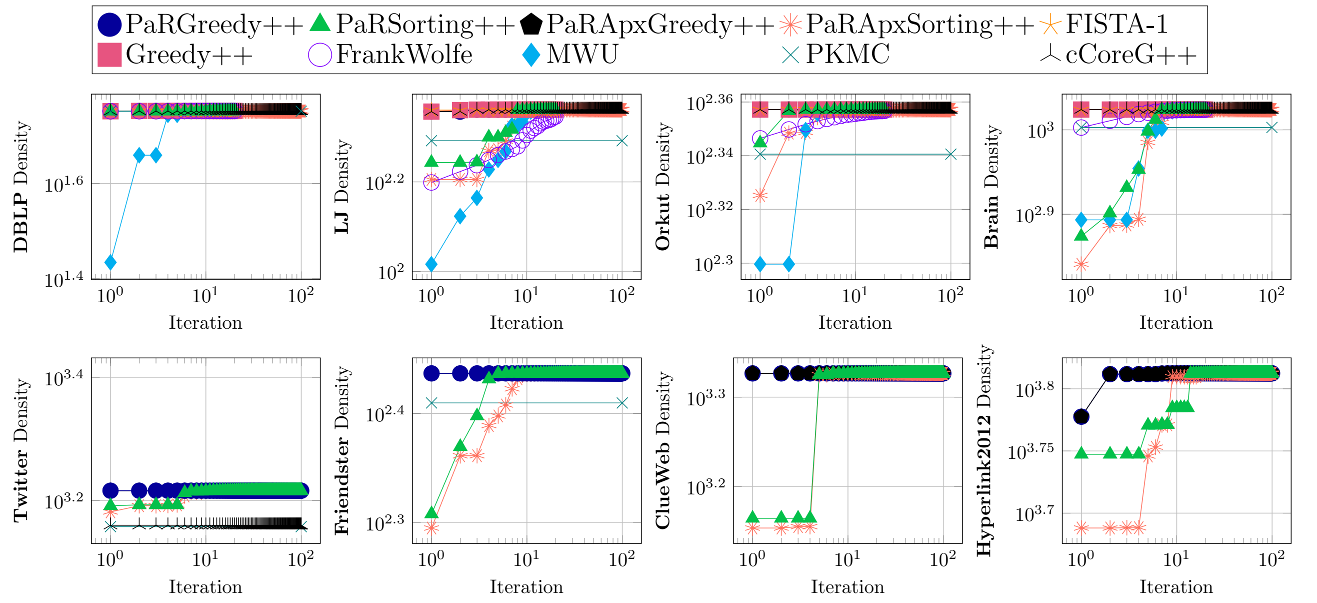

Figure 2: Densities on different iterations for various algorithms. Only our algorithms can successfully process all of the large graphs (bottom row) within the 1 hour limit.

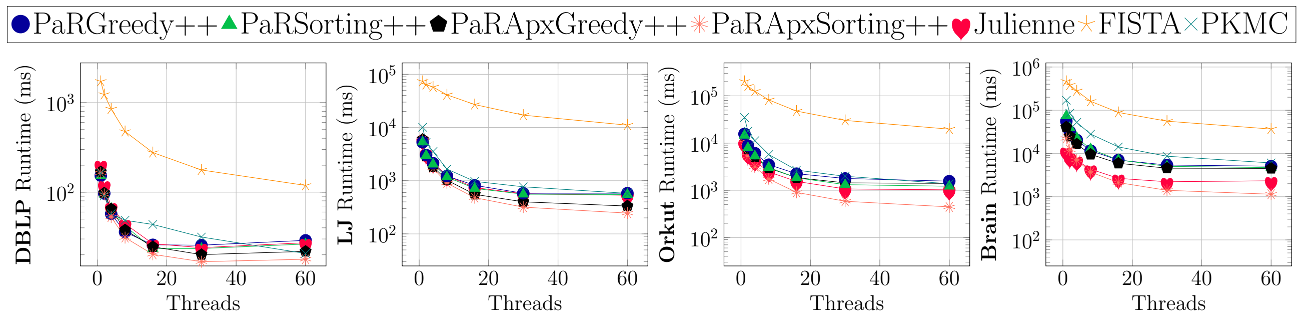

Figure 3: Runtimes (ms) of `PaRGreedy++`, `PaRSorting++`, `PaRApxGreedy++`, `PaRApxSorting++`, `Julienne`, `FISTA`, and `PKMC` versus the number of threads when running for 5 iterations.

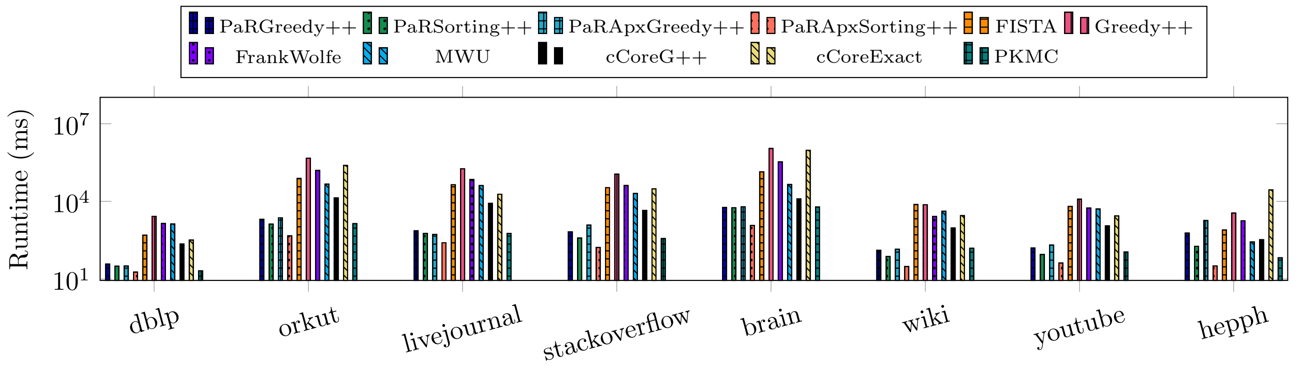

Figure 4: Runtimes of different densest subgraph algorithms on our small graph inputs. The algorithms are run for 20 iterations. Parallel algorithms use 60 hyper-threads.

Graph Dataset	$\tilde{\rho}$	FISTA	MWU	FrankWolfe	Greedy++	PaRGreedy++	PaRSorting++	PaRApxSorting++
hepph*	265.969	1.00043	1.00011	1.00011	1	1	1.00031	1.00001
dblp*	56.565	1	1	1	1	1	1	1
brain	1057.458	1.00011	1.00005	1.00031	1	1	1.00026	1.0001
wiki*	108.59	1	1	1.00723	1	1	1.00002	1
youtube*	45.599	1.00023	1.00104	1.01522	1.00007	1	1.00079	1.00063
stackoverflow*	181.587	1	1.00001	1.0031	1	1	1.00002	1.00001
livejournal*	229.846	1.00113	1.00003	1.03671	1	1	1.00019	1.00006
orkut*	227.874	1	1.00011	1.00123	1	1	1.00026	1.00026
twitter	1643.301	n/a	n/a	n/a	1	1	1.00003	1.00006
friendster	273.519	n/a	n/a	n/a	n/a	1	1	1.00006
clueweb	2122.5	n/a	n/a	n/a	n/a	1	1	1
hyperlink2012	6496.649	n/a	n/a	n/a	n/a	1	1	1.00002

Table 2: Approximation Ratio at the 20th iteration for various algorithms. $\tilde{\rho}$ is the best densest subgraph known in the literature. Ratios are computed as the best currently known density ($\tilde{\rho}$) divided by the density produced by the respective algorithm. Results are indicated as n/a if the corresponding algorithms timeout at 1 hour. Graphs indicated with an $*$ have optimum computed densities. $\tilde{\rho}$ is rounded to 3 decimal places and approximation ratios are rounded to 5 decimal places. PaRApxGreedy++ is omitted since the ratios are identical to PaRGreedy++.

PaRSorting++, PaRApxGreedy++, PaRApxSorting++) with Greedy++ [10], FISTA, FrankWolfe, PKMC, cCoreG++, cCoreExact, and MWU [28] in Fig. 4. The detailed results can be found in our full paper. In short, when measuring the quality of our solutions in running time, our algorithms outperform all existing algorithms by significant margins, and is up to 25.9× faster than the fastest $(1+\varepsilon)$-approximation algorithm for each graph. On many of the graphs that we tested, we achieve a 2× improvement in runtime when using approximate k-core compared to when we use exact.

Large Graph Runtime and Accuracy Results. Even when using multi-threading, our algorithms are the only algorithms to finish processing the large graphs (twitter, friendster, clueweb, and hyperlink2012) within 1 hour. Specifically, for 20 iterations and 60 threads, our fastest algorithms PaRGreedy++ and PaRSorting++ require 10.44, 15.35, 112.64, and 352.65 seconds, and 8.41, 10.54, 83.91, and 270.39 seconds on twitter, friendster, clueweb, and hyperlink2012, respectively. When using approximate k-core, PaRApxGreedy++ and PaRApxSorting++ require 9.77, 15.15, 120.97, and 375.71 seconds, and 8.27, 8.62, 85.29, and 287.27 seconds on twitter, friendster, clueweb, and hyperlink2012, respectively. Moreover, we obtain the best densities known in literature for these graphs (shown in Table 2). For massive real-world graphs, our experiments show that using approximate k-core does not yield much benefit. This is because larger graphs lead to more parallelism in the exact k-core decomposition algorithm, so the exact k-core algorithm exhibits similar parallelism to the approximate k-core algorithm on a 30-core machine.

Initialization time. We measure the initialization time of our algorithms (i.e., the time to perform the k-core decomposition step), and report the results in our full paper. The finding is that significant portion of total runtime is on this initialization step for all of the graphs. This makes sense as the initial graph tends to be much larger than the graph we obtain after pruning. To illustrate this point, for brain, when running on 60 threads, the initialization step takes at least 2655 ms for PaRGreedy++, PaRSorting++, and PaRApxGreedy++, and 1074 ms for PaRApxSorting++. The average time spent on each iteration for these algorithms are 138, 3, 148, and 2.3 ms, respectively. PaRApxSorting++ is still faster than all other algorithms despite running for 300 more iterations.

5 Conclusion

We introduced a framework that combines pruning and refinement for solving the approximate densest subgraph problem. We designed new parallel variants of the sequential Greedy++ algorithm, and achieved state-of-the-art performance by plugging them into our framework. We showed that our algorithms can scale to the large hyperlink2012 and clueweb graphs and obtain near-optimal approximations of their densest subgraphs for the first time in the literature.

Acknowledgements

This research was supported by DOE Early Career Award #DESC0018947, NSF Awards #CCF-1845763 and #CCF-2103483, Google Faculty Research Award, Google Research Scholar Award, cloud computing credits from Google-MIT, and FinTech@CSAIL Initiative.

References

[1] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani. Large scale networks fingerprinting and visualization using the k-core decomposition. In *Proceedings of the International Conference on Neural Information Processing Systems*, 2005.

[2] A. Angel, N. Koudas, N. Sarkas, D. Srivastava, M. Svendsen, and S. Tirthapura. Dense subgraph maintenance under streaming edge weight updates for real-time story identification. *VLDB J.*, 23(2):175–199, 2014.

[3] S. Arora, E. Hazan, and S. Kale. The multiplicative weights update method: a meta-algorithm and applications. *Theory of Computing*, 8(1):121–164, 2012.

[4] B. Bahmani, A. Goel, and K. Munagala. Efficient primal-dual graph algorithms for MapReduce. In *International Workshop on Algorithms and Models for the Web Graph (WAW)*, pages 59–78, 2014.

[5] B. Bahmani, R. Kumar, and S. Vassilvitskii. Densest subgraph in streaming and MapReduce. *Proceedings of the VLDB Endowment*, 5(5), 2012.

[6] A. Beck and M. Teboulle. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM journal on imaging sciences*, 2(1):183–202, 2009.

[7] S. Bhattacharya, M. Henzinger, D. Nanongkai, and C. Tsourakakis. Space- and time-efficient algorithm for maintaining dense subgraphs on one-pass dynamic streams. In *ACM Symposium on Theory of Computing (STOC)*, pages 173–182, 2015.

[8] G. E. Blelloch, D. Anderson, and L. Dhulipala. ParlayLib - a toolkit for parallel algorithms on shared-memory multicore machines. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*, page 507–509, 2020.

[9] P. Boldi and S. Vigna. The webgraph framework I: compression techniques. In *Proceedings of the International Conference on World Wide Web*, pages 595–602, 2004.

[10] D. Boob, Y. Gao, R. Peng, S. Sawlani, C. Tsourakakis, D. Wang, and J. Wang. Flowless: Extracting densest subgraphs without flow computations. In *Proceedings of The Web Conference 2020*, page 573–583, 2020.

[11] M. Charikar. Greedy approximation algorithms for finding dense components in a graph. In *Approximation Algorithms for Combinatorial Optimization*, pages 84–95, 2000.

[12] C. Chekuri and K. Quanrud. $(1-\epsilon)$-approximate fully dynamic densest subgraph: linear space and faster update time, 2022.

[13] C. Chekuri, K. Quanrud, and M. R. Torres. Densest subgraph: Supermodularity, iterative peeling, and flow. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1531–1555, 2022.

[14] A. B. G. Christiansen, J. Holm, I. van der Hoog, E. Rotenberg, and C. Schwiegelshohn. Adaptive out-orientations with applications. *CoRR*, abs/2209.14087, 2022.

[15] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003.

[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[17] M. Danisch, T.-H. H. Chan, and M. Sozio. Large scale density-friendly graph decomposition via convex programming. In *Proceedings of the International Conference on World Wide Web*, page 233–242, 2017.

[18] L. Dhulipala, G. E. Blelloch, and J. Shun. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 293–304, 2017.

[19] L. Dhulipala, G. E. Blelloch, and J. Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018.

[20] L. Dhulipala, Q. C. Liu, S. Raskhodnikova, J. Shi, J. Shun, and S. Yu. Differential privacy from locally adjustable graph algorithms: k-core decomposition, low out-degree ordering, and densest subgraphs. In *IEEE Annual Symposium on Foundations of Computer Science*, pages 754–765, 2022.

[21] R. Dondi, M. M. Hosseinzadeh, and I. Zoppis. Dense temporal subgraphs in protein-protein interaction networks. In *International Conference on Computational Science*, pages 469–480, 2022.

[22] X. Du, R. Jin, L. Ding, V. E. Lee, and J. H. Thornton. Migration motif: A spatial - temporal pattern mining approach for financial markets. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, page 1135–1144, 2009.

[23] Y. Fang, K. Yu, R. Cheng, L. V. S. Lakshmanan, and X. Lin. Efficient algorithms for densest subgraph discovery. *Proc. VLDB Endow.*, 12(11):1719–1732, 2019.

[24] A. Farhadi, M. T. Hajiaghai, and E. Shi. Differentially private densest subgraph. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 11581–11597, 2022.

[25] E. Fratkin, B. T. Naughton, D. L. Brutlag, and S. Batzoglou. MotifCut: regulatory motifs finding with maximum density subgraphs. In *ISMB*, pages 156–157, 2006.

[26] M. Ghaffari, S. Lattanzi, and S. Mitrović. Improved parallel algorithms for density-based network clustering. In *Proceedings of the International Conference on Machine Learning*, pages 2201–2210, 2019.

[27] A. Gionis, F. Junqueira, V. Leroy, M. Serafini, and I. Weber. Piggybacking on social networks. *Proc. VLDB Endow.*, 6(6):409–420, apr 2013.

[28] E. Harb, K. Quanrud, and C. Chekuri. Faster and scalable algorithms for densest subgraph and decomposition. In *Advances in Neural Information Processing Systems*, 2022.

[29] B. Hooi, H. A. Song, A. Beutel, N. Shah, K. Shin, and C. Faloutsos. FRAUDAR: Bounding graph fraud in the face of camouflage. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, page 895–904, 2016.

[30] J. Jaja. *Introduction to Parallel Algorithms.* Addison-Wesley Professional, 1992.

[31] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-HOP: A high-compression indexing scheme for reachability query. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, page 813–826, 2009.

[32] S. Khuller and B. Saha. On finding dense subgraphs. In *International Colloquium on Automata, Languages and Programming*, pages 597–608, 2009.

[33] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Trawling the web for emerging cyber-communities. *Computer Networks*, 31(11):1481–1493, 1999.

[34] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. June 2014.

[35] Q. C. Liu, J. Shi, S. Yu, L. Dhulipala, and J. Shun. Parallel batch-dynamic algorithms for k-core decomposition and related graph problems. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 191–204, 2022.

[36] W. Luo, Z. Tang, Y. Fang, C. Ma, and X. Zhou. Scalable algorithms for densest subgraph discovery. In *IEEE International Conference on Data Engineering (ICDE)*, pages 287–300. IEEE, 2023.

[37] R. Meusel, S. Vigna, O. Lehmberg, and C. Bizer. The graph structure in the Web - analyzed on different aggregation levels. *J. Web Sci.*, 1:33–47, 2015.

[38] Y. E. Nesterov. A method of solving a convex programming problem with convergence rate $o\left(\frac{1}{k^2}\right)$. In *Doklady Akademii Nauk*, volume 269, pages 543–547. Russian Academy of Sciences, 1983.

[39] D. Nguyen and A. Vullikanti. Differentially private densest subgraph detection. In *Proceedings of the International Conference on Machine Learning*, pages 8140–8151, 2021.

[40] R. Raman. The power of collision: Randomized parallel algorithms for chaining and integer sorting. In *Foundations of Software Technology and Theoretical Computer Science*, pages 161–175, 1990.

[41] R. A. Rossi and N. K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[42] P. Rozenshtein, N. Tatti, and A. Gionis. Discovering dynamic communities in interaction networks. In *Machine Learning and Knowledge Discovery in Databases*, pages 678–693, 2014.

[43] S. Solomon and N. Wein. Improved dynamic graph coloring. *ACM Trans. Algorithms*, 16(3), June 2020.

[44] H.-H. Su and H. T. Vu. Distributed Dense Subgraph Detection and Low Outdegree Orientation. In *International Symposium on Distributed Computing*, pages 15:1–15:18, 2020.

[45] T. Swarnkar, S. N. Simões, A. Anurak, H. Brentani, J. Chatterjee, R. F. Hashimoto, D. C. Martins, and P. Mitra. Identifying dense subgraphs in protein-protein interaction network for gene selection from microarray data. *Netw. Model. Anal. Health Informatics Bioinform.*, 4(1):33, 2015.

[46] N. Tatti and A. Gionis. Density-friendly graph decomposition. In *Proceedings of the International Conference on World Wide Web*, page 1089–1099, 2015.

[47] U. Vishkin. Thinking in parallel: Some basic data-parallel algorithms and techniques. *Parallel Algorithms*, 2010.

[48] Y. Xu, C. Ma, Y. Fang, and Z. Bao. Efficient and effective algorithms for generalized densest subgraph discovery. *Proceedings of the ACM on Management of Data*, 1(2):1–27, 2023.