The FAST Algorithm for Submodular Maximization

Adam Breuer [1] **Eric Balkanski** [1] **Yaron Singer** [1]

Abstract

In this paper we describe a new parallel algorithm called Fast Adaptive Sequencing Technique (FAST) for maximizing a monotone submodular function under a cardinality constraint k. This algorithm achieves the optimal $1 - 1/e$ approximation guarantee and is orders of magnitude faster than the state-of-the-art on a variety of experiments over real-world data sets. Following recent work by Balkanski & Singer (2018a), there has been a great deal of research on algorithms whose theoretical parallel runtime is exponentially faster than algorithms used for submodular maximization over the past 40 years. However, while these new algorithms are fast in terms of asymptotic worst-case guarantees, it is computationally infeasible to use them in practice even on small data sets because the number of rounds and queries they require depend on large constants and high-degree polynomials in terms of precision and confidence. The design principles behind the FAST algorithm we present here are a significant departure from those of recent theoretically fast algorithms. Rather than optimize for asymptotic theoretical guarantees, the design of FAST introduces several new techniques that achieve remarkable practical and theoretical parallel runtimes. The approximation guarantee obtained by FAST is arbitrarily close to $1 - 1/e$, and its asymptotic parallel runtime (adaptivity) is $\mathcal{O}(\log(n)\log^2(\log k))$ using $\mathcal{O}(n \log\log(k))$ total queries. We show that FAST is orders of magnitude faster than any algorithm for submodular maximization we are aware of, including hyper-optimized parallel versions of state-of-the-art serial algorithms, by running experiments on large data sets.

[1]Harvard University, Cambridge, MA. Correspondence to: Adam Breuer <breuer@g.harvard.edu>, Eric Balkanski <ebalkans@gmail.com>, Yaron Singer <yaron@seas.harvard.edu>.

Proceedings of the 37^{th} International Conference on Machine Learning, Online, PMLR 119, 2020. Copyright 2020 by the author(s).

1. Introduction

In this paper we describe a fast parallel algorithm for submodular maximization.[1] Informally, a function is submodular if it exhibits a natural diminishing returns property. For the canonical problem of maximizing a monotone submodular function under a cardinality constraint, it is well known that the greedy algorithm, which iteratively adds elements whose marginal contribution is largest to the solution, obtains a $1 - 1/e$ approximation guarantee (Nemhauser et al., 1978) which is optimal for polynomial-time algorithms (Nemhauser & Wolsey, 1978). The greedy algorithm and other submodular maximization techniques are heavily used in machine learning and data mining as many fundamental objectives such as entropy, mutual information, graphs cuts, diversity, and set cover are submodular.

In recent years there has been a great deal of progress on fast algorithms for submodular maximization designed to accelerate computation on large data sets. The first line of work considers *serial* algorithms where queries can be evaluated on a single processor (Leskovec et al., 2007; Badanidiyuru & Vondrák, 2014; Mirzasoleiman et al., 2015; 2016; Ene & Nguyen, 2019b;c). For serial algorithms the state-of-the-art for maximization under a cardinality constraint is the *lazier-than-lazy-greedy* (LTLG) algorithm which returns a solution that is in expectation arbitrarily close to the optimal $1 - 1/e$ and does so in a linear number of queries (Mirzasoleiman et al., 2015). This algorithm is a stochastic greedy algorithm coupled with lazy updates, which not only performs well in terms of the quality of the solution it returns, but is also very fast in practice.

Accelerating computation beyond linear runtime requires *parallelization*. The parallel runtime of blackbox optimization is measured by *adaptivity*, which is the number of sequential rounds an algorithm requires when polynomially-many queries can be executed in parallel in every round. For maximizing a submodular function defined over a ground set of n elements under a cardinality constraint k, the adaptivity of the naive greedy algorithm is $\mathcal{O}(k)$, which in the worst case is $\mathcal{O}(n)$. Until recently no algorithm was known to have better parallel runtime than that of greedy.

A very recent line of work initiated by Balkanski &

[1]Code is available from www.adambreuer.com/code.

Algorithm	*rounds*	*queries*	*time (sec)*
AMORTIZED-FILTERING (Balkanski et al., 2019a)	961	2124351	20.29
BINARY-SEARCH-MAXIMIZATION (Fahrbach et al., 2019a)	8744	2552028	24.64
RANDOMIZED-PARALLEL-GREEDY (Chekuri & Quanrud, 2019b)	92	148642	4.11
PARALLEL-LTLG (Mirzasoleiman et al., 2015)	200	856	0.15
FAST	**9**	**1598**	**0.033**

Singer (2018a) develops techniques for designing constant approximation algorithms for submodular maximization whose parallel runtime is *logarithmic* (Balkanski & Singer, 2018b; Balkanski et al., 2018; Ene & Nguyen, 2019a; Fahrbach et al., 2019a;b; Kazemi et al., 2019; Chekuri & Quanrud, 2019a;b; Balkanski et al., 2019a;b; Ene et al., 2019; Chen et al., 2019; Esfandiari et al., 2019; Qian & Singer, 2019). In particular, Balkanski & Singer (2018a) describe a technique called *adaptive sampling* that obtains in $\mathcal{O}(\log n)$ rounds a $1/3$ approximation for maximizing a monotone submodular function under a cardinality constraint. This technique can be used to obtain an approximation arbitrarily close to the optimal $1-1/e$ in $\mathcal{O}(\log n)$ rounds (Balkanski et al., 2019a; Ene & Nguyen, 2019a).

1.1. From theory to practice

The focus of the work on adaptivity described above has largely been on conceptual and theoretical contributions: achieving strong approximation guarantees under various constraints with runtimes that are exponentially faster under worst case theoretical analysis. From a practitioner's perspective however, even the state-of-the-art algorithms in this genre are infeasible for large data sets. The logarithmic parallel runtime of these algorithms carries extremely large constants and polynomial dependencies on precision and confidence parameters that are hidden in their asymptotic analysis. In terms of sample complexity alone, obtaining (for example) a $1-1/e-0.1$ approximation with 95% confidence for maximizing a submodular function under cardinality constraint k requires evaluating at least 10^8 (Balkanski et al., 2019a) or 10^6 (Fahrbach et al., 2019a; Chekuri & Quanrud, 2019b) samples of sets of size approximately $\frac{k}{\log n}$ in every round. Even if one heuristically uses a single sample in every round, other sources of inefficiencies that we discuss throughout the paper prevent these algorithms from being applied even on moderate-sized data sets. The question is then whether the plethora of breakthrough techniques in this line of work of exponentially faster algorithms for submodular maximization can lead to algorithms that are fast in practice for large problem instances.

1.2. Our contribution

In this paper we design a new algorithm called Fast Adaptive Sequencing Technique (FAST) for maximizing a monotone submodular function under a cardinality constraint k. FAST has an approximation ratio that is arbitrarily close to $1-1/e$, is $\mathcal{O}(\log(n)\log^2(\log k))$ adaptive, and uses a total of $\mathcal{O}(n\log\log(k))$ queries. The main contribution is not in the algorithm's asymptotic guarantees, but in its design that is extremely efficient both in terms of its non-asymptotic worst case query complexity and number of rounds, and in terms of its practical runtime. In terms of *actual* query complexity and practical runtime, this algorithm outperforms all algorithms for submodular maximization we are aware of, including hyper-optimized versions of LTLG. To be more concrete, we give a brief experimental comparison in the table above for a movie recommendation objective on $n = 500$ movies against optimized implementations of algorithms with the same adaptivity and approximation (experiment details in Section 4).[2]

FAST achieves its speedup by thoughtful design that results in frugal worst case query complexity as well as several heuristics used for practical speedups. From a purely analytical perspective, FAST improves the ε dependency in the linear term of the query complexity of at least $\tilde{\mathcal{O}}(\varepsilon^{-5}n)$ in Balkanski et al. (2019a) and Ene & Nguyen (2019a) and $\tilde{\mathcal{O}}(\varepsilon^{-3}n)$ in Fahrbach et al. (2019a) to $\tilde{\mathcal{O}}(\varepsilon^{-2}n)$. We provide the first non-asymptotic bounds on the query and adaptive complexity of an algorithm with sublinear adaptivity, showing dependency on small constants. Our algorithm uses adaptive sequencing (Balkanski et al., 2019b) and multiple optimizations to improve the query complexity and runtime.

1.3. Paper organization

We introduce the main ideas and decisions behind the design of FAST in Section 2. We describe and analyze guarantees in Section 3. We discuss experiments in Section 4.

[2]To obtain these values, we set all algorithms to guarantee a $1-1/e-0.1$ approximation with probability 0.95 except LTLG, which has this guarantee in expectation, and we set $k = 200$.

2. FAST Overview

Before describing the algorithm, we give an overview of the major ideas and discuss how they circumvent the bottlenecks for practical implementation of existing logarithmic adaptivity algorithms.

Adaptive sequencing vs. adaptive sampling. The large majority of low-adaptivity algorithms use *adaptive sampling* (Balkanski & Singer, 2018a; Ene & Nguyen, 2019a; Fahrbach et al., 2019a;b; Balkanski et al., 2018; 2019a; Kazemi et al., 2019), a technique introduced in Balkanski & Singer (2018a). These algorithms sample a large number of sets of elements at every iteration to estimate (1) the expected marginal contribution of a random set R to the current solution S and (2) the expected marginal contributions of each element a to $R \cup S$. These estimates, which rely on concentration arguments, are then used to either add a random set R to S or discard elements with low expected marginal contribution to $R \cup S$.

In contrast, the *adaptive sequencing* technique which was recently introduced in Balkanski et al. (2019b) generates at every iteration a *single* random sequence $(a_1, \dots, a_{|X|})$ of the elements X not yet discarded. A prefix $A_{i^\star} = (a_1, \dots, a_{i^\star})$ of the sequence is then added to the solution S, where $i^\star$ is the largest position i such that a large fraction of the elements in X have high marginal contribution to $S \cup A_{i-1}$. Elements with low marginal contribution to the new solution S are then discarded from X.

The first choice we made was to use an adaptive sequencing technique rather than adaptive sampling.

- **Dependence on large polynomials in ε.** Adaptive sampling algorithms crucially rely on sampling, and as a result their query complexity has high polynomial dependency on ε (e.g. at least $\mathcal{O}(\varepsilon^{-5}n)$ in Balkanski et al. (2019a) and Ene & Nguyen (2019a)). Due to these ε dependencies, the query complexity blows up with any reasonable value for ε. In contrast, adaptive *sequencing* generates a *single* random sequence at every iteration. Therefore, in the term that is linear in n we can obtain an ε dependence that is only $\tilde{\mathcal{O}}(\varepsilon^{-2})$.
- **Dependence on large constants.** The asymptotic query complexity of previous algorithms depends on very large constants (e.g. at least 60000 in Balkanski et al. (2019a) and Ene & Nguyen (2019a)) making them impractical. As we tried to optimize constants for adaptive sampling, we found that due to the sampling and the requirement to maintain strong theoretical guarantees, the constants cascade and grow through multiple parts of the analysis. In principle, adaptive *sequencing* does not rely on sampling, which dramatically reduces its dependency on constants.

Negotiating the adaptive complexity with the query complexity. The vanilla version of our algorithm, whose description and analysis are in Appendix A, has at most $\varepsilon^{-2} \log n$ adaptive rounds and uses a total of $\varepsilon^{-2} nk$ queries to obtain a $1 - 1/e - \frac{3}{2}\varepsilon$ approximation, without additional dependence on constants or lower order terms. In our actual algorithm, we trade a small factor in adaptive complexity for a substantial improvement in query complexity. We do this in the following manner:

- **Search for estimates of `OPT`.** All algorithms with logarithmic adaptivity require a good estimate of `OPT`, which can be obtained by running $\varepsilon^{-1} \log k$ instances of the algorithms with different guesses of `OPT` in parallel, so that one guess is guaranteed to be a good approximation to `OPT`.[3] We accelerate this search by binary searching over the guesses of `OPT`. A main difficulty when using this binary search is that the approximation guarantee of the solution obtained with each guess of `OPT` needs to hold with high probability, instead of in expectation, to obtain any guarantee for the global solution.

  Even though the guarantees on the marginal contributions obtained from each element added to the solution only hold in expectation for adaptive sequencing, we obtain high probability guarantees for the global solution by generalizing the robust guarantees obtained in Hassidim & Singer (2017) so that they also apply to adaptive sequencing. In the practical speedups below, we discuss how we often only need a single iteration of this binary search in practice;

- **Search for position $i^\star$.** To find the position $i^\star$, which is the largest position $i \in [k]$ in the sequence such that a large fraction of not-yet-discarded elements have high marginal contribution to $S \cup A_{i-1}$, the vanilla adaptive sequencing technique queries the marginal contribution of all elements in X at each of the k positions. This search for $i^\star$ causes the $\mathcal{O}(nk)$ query complexity.

  Instead, similarly to the search of `OPT`, we binary search over a set of $\varepsilon^{-1} \log k$ geometrically increasing values i that correspond to guesses of $i^\star$. This improves the $\mathcal{O}(nk)$ dependency on n and k in the query complexity to $\mathcal{O}(n \log(\log k))$. Then, at any step of the binary search over a position i, instead of evaluating the marginal contribution of all elements in X to $S \cup A_{i-1}$, we only evaluate a small sample of elements. In the practical speedups below, we discuss how we can often skip this binary search for $i^\star$ in practice.

[3] Fahrbach et al. (2019a) do some preprocessing to estimate `OPT`, but it is estimated within some very large constant.

Practical speedups. We include several ideas which result in considerable speedups in practice without sacrificing approximation, adaptivity, or query complexity guarantees:

- **Preprocessing the sequence.** At the outset of each iteration of the algorithm, before searching for a prefix $A_{i^\star}$ to add to the solution S, we first use a preprocessing step that adds high value elements from the sequence to S. Specifically, we add to the solution S all sequence elements a_i that have high marginal contribution to $S \cup A_{i-1}$.

  After adding these high-value elements, we discard surviving elements in X that have low marginal contribution to the new solution S. In the case where this step discards a large fraction of surviving elements from X, we can also skip this iteration's binary search for $i^\star$ and continue to the next iteration without adding a prefix to S;

- **Number of elements added per iteration.** An adaptive sampling algorithm which samples sets of size s adds at most s elements to the current solution at each iteration. In contrast, adaptive sequencing and the preprocessing step described above often allow our algorithm to add a very large number of elements to the current solution at each iteration in practice;

- **Single iteration of the binary search for `OPT`.** Even with binary search, running multiple instances of the algorithm with different guesses of `OPT` is undesirable. We describe a technique that often needs only a single guess of `OPT`. This guess is the sum $v = \max_{|S| \leq k} \sum_{a \in S} f(a)$ of the k highest valued singletons, which is an upper bound on `OPT`. If the solution S obtained with that guess v has value $f(S) \geq (1 - 1/e)v$, then, since $v \geq$ `OPT`, S is guaranteed to obtain a $1 - 1/e$ approximation and the algorithm does not need to continue the binary search. Note that with a single guess of `OPT`, the robust guarantees for the binary search are not needed, which improves the sample complexity to $m = \frac{2+\varepsilon}{\varepsilon^2(1-3\varepsilon)} \log(2\delta^{-1})$;

- **Lazy updates.** There are many situations where lazy evaluations of marginal contributions can be performed (Minoux, 1978; Mirzasoleiman et al., 2015). Since we never discard elements from the solution S, the marginal contributions of elements a to S are non-increasing at every iteration by submodularity. Elements with low marginal contribution c to the current solution at some iteration are ignored until the threshold t is lowered to $t \leq c$. Lazy updates also accelerate the binary search over $i^\star$.

3. The Algorithm

We describe the FAST-FULL algorithm (Algorithm 1). The main part of the algorithm is the FAST subroutine (Algorithm 2), which is instantiated with different guesses of `OPT`. These guesses $v \in V$ of `OPT` are geometrically increasing from $\max_{a \in N} f(a)$ to $\max_{|S| \leq k} \sum_{a \in S} f(a)$ by a $(1-\varepsilon)^{-1}$ factor, so V contains a value that is a $1 - \varepsilon$ approximation to `OPT`. The algorithm binary searches over guesses for the largest guess v that obtains a solution S that is a $1 - 1/e$ approximation to v.

Algorithm 1 FAST-FULL: the full algorithm

```
input function $f$, cardinality constraint $k$, parameter $\varepsilon$
  $V \leftarrow$ GEOM$(\max_a f(a), \max_{|S| \leq k} \sum_{a \in S} f(a), 1 - \varepsilon)$
  $v^\star \leftarrow$ B-SEARCH$(\max\{v \in V : f(S_v) \geq (1 - 1/e)v\})$
      where $S_v \leftarrow$ FAST$(v)$
  return $S_{v^\star}$
```

FAST generates at every iteration a uniformly random sequence $a_1, \ldots, a_{|X|}$ of the elements X not yet discarded. After the preprocessing step which adds to S elements guaranteed to have high marginal contribution, the algorithm identifies a position $i^\star$ in this sequence which determines the prefix $A_{i^\star-1}$ that is added to the current solution S. Position $i^\star$ is defined as the largest position such that there is a large fraction of elements in X with high contribution to $S \cup A_{i^\star-1}$. To find $i^\star$, we binary search over geometrically increasing positions $i \in I \subseteq [k]$. At each position i, we only evaluate the contributions of elements $a \in R$, where R is a uniformly random subset of X of size m, instead of all elements X.

Algorithm 2 FAST: the Fast Adaptive Sequencing Technique algorithm

```
input $f$, constraint $k$, guess $v$ for OPT, parameter $\varepsilon$
  $S \leftarrow \emptyset$
  while $|S| < k$ and number of iterations $< \varepsilon^{-1}$ do
    $X \leftarrow N, t \leftarrow (1 - \varepsilon)(v - f(S))/k$
    while $X \neq \emptyset$ and $|S| < k$ do
      $a_1, \ldots, a_{|X|} \leftarrow$ SEQUENCE$(X, |X|)$
      $A_i \leftarrow a_1, \ldots, a_i$
      $S \leftarrow S \cup \{a_i : f_{S \cup A_{i-1}}(a_i) \geq t\}$
      $X_0 \leftarrow \{a \in X : f_S(a) \geq t\}$
      if $|X_0| \leq (1 - \varepsilon)|X|$ then
        $X \leftarrow X_0$ and continue to next iteration
      $R \leftarrow$ SAMPLE$(X, m)$,
      $I \leftarrow$ GEOM$(1, k - |S|, 1 - \varepsilon)$
      $R_i \leftarrow \{a \in R : f_{S \cup A_{i-1}}(a) \geq t\}$ , for $i \in I$
      $i^\star \leftarrow$ B-SEARCH$(\max\{i : |R_i| \geq (1 - 2\varepsilon)|R|\})$
      $S \leftarrow S \cup A_{i^\star}$
  return $S$
```

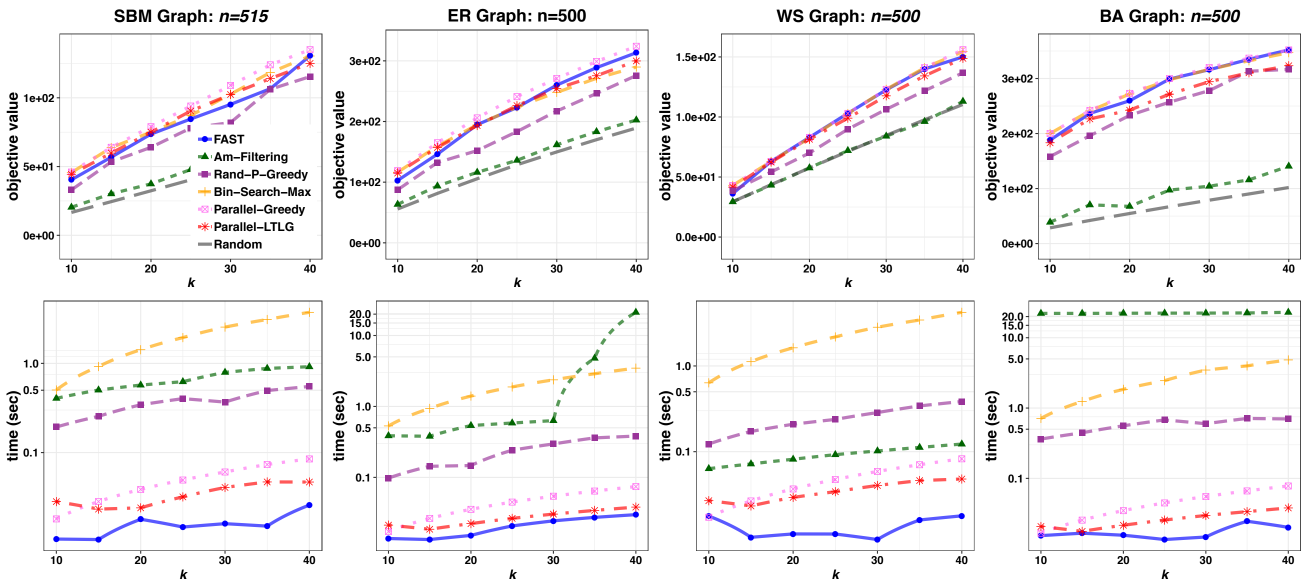

Figure 1. Experiment Set 1.a: Fast *(blue) vs. low-adaptivity algorithms and* Parallel-LTLG *on graphs (time axis* ***log-scaled****).*

3.1. Analysis

We show that Fast obtains a $1 - 1/e - \varepsilon$ approximation w.p. $1-\delta$ and that it has $\tilde{O}(\varepsilon^{-2}\log n)$ adaptive complexity and $\tilde{O}(\varepsilon^{-2}n + \varepsilon^{-4}\log(n)\log(\delta^{-1}))$ query complexity.

Theorem 1. *Assume* $k \geq \frac{2\log(2\delta^{-1}\ell)}{\varepsilon^2(1-5\varepsilon)}$ *and* $\varepsilon \in (0, 0.1)$, *where* $\ell = \log(\frac{\log k}{\varepsilon})$. *Then,* Fast *with sample complexity* $m = \frac{2+\varepsilon}{\varepsilon^2(1-3\varepsilon)}\log(\frac{4\ell\log n}{\delta\varepsilon^2})$ *has at most* $\varepsilon^{-2}\log(n)\ell^2$ *adaptive rounds,* $2\varepsilon^{-2}\ell n + \varepsilon^{-4}\log(n)\ell^2 m$ *queries, and achieves a* $1 - \frac{1}{e} - 4\varepsilon$ *approximation with probability* $1-\delta$.

We defer the analysis to Appendix B. The main part of it is for the approximation guarantee, which consists of two cases depending on the condition which breaks the outer-loop. Lemma 3 shows that when there are ε^{-1} iterations of the outer-loop, the set of elements added to S at every iteration of the outer-loop contributes $\varepsilon^{-1}(\texttt{OPT} - f(S))$. Lemma 5 shows that for the case where $|S| = k$, the expected contribution of each element a_i added to S is arbitrarily close to $(\texttt{OPT} - f(S))/k$. For each solution S_v, we need the approximation guarantee to hold with high probability instead of in expectation to be able to binary search over guesses for $\texttt{OPT}$, which we obtain in Lemma 7 by generalizing the robust guarantees of Hassidim & Singer (2017) in Lemma 6. The main observation to obtain the adaptive complexity (Lemma 6) is that, by definition of $i^\star$, at least an ε fraction of the surviving elements in X are discarded at every iteration with high probability.[4] For the query complexity (Lemma 7), we note that there are $|X| + m\ell$ function evaluations per iteration.

[4]To obtain the adaptivity r with probability 1 and the approximation guarantee w.p. $1-\delta$, the algorithm declares failure after r rounds and accounts for this failure probability in δ.

4. Experiments

Our goal in this section is to show that in practice, Fast finds solutions whose value meets or exceeds alternatives in less parallel runtime than both state-of-the-art low-adaptivity algorithms and Lazier-than-Lazy-Greedy (LTLG). To accomplish this, we build optimized parallel MPI implementations of Fast, other low-adaptivity algorithms, and LTLG, which is widely regarded as the fastest algorithm for submodular maximization in practice. We then use 95 Intel Skylake-SP 3.1 GHz processors on AWS to compare the algorithms' runtime over a variety of objectives defined on 8 real and synthetic datasets. We measure runtime using a rigorous measure of parallel time (see Appendix C.8). Appendices C.1, C.4, C.9, and C.6 contain detailed descriptions of the benchmarks, objectives, implementations, hardware, and experimental setup on AWS.[5]

We conduct two sets of experiments. The first set compares Fast to previous low-adaptivity algorithms on 8 objectives. Since previous algorithms all have practically intractable sample complexity, we grossly reduce their sample complexity to only 95 samples per iteration so that each processor performs a single function evaluation per iteration. This reduction, which we discuss in detail below, gives these algorithms a large runtime advantage over Fast, which computes its full theoretical sample complexity in all experiments. This is practically feasible for Fast

[5]Code is available from www.adambreuer.com/code.

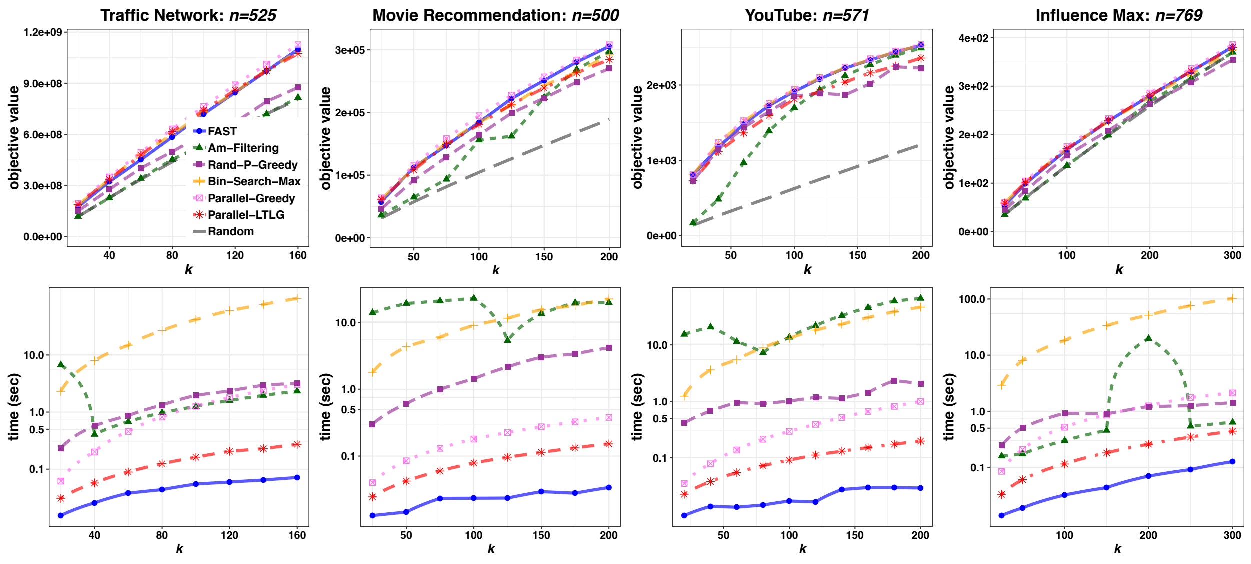

Figure 2. Experiment Set 1.b: FAST *(blue) vs. low-adaptivity algorithms and* PARALLEL-LTLG *on real data (time axis **log-scaled**).*

because FAST samples *elements*, not *sets* of elements like previous algorithms. Despite the large advantage this setup gives to the previous low-adaptivity algorithms, FAST is consistently one to three orders of magnitude faster.

The second set of experiments compares FAST to PARALLEL-LAZIER-THAN-LAZY-GREEDY (PARALLEL-LTLG) on large-scale data sets. We scale up the 8 objectives to be defined on synthetic data with $n = 100000$ and real data with up to $n = 26000$ and various k ranging from $k = 25$ to $k = 25000$. We find that FAST is consistently 1.5 to 27 times faster than PARALLEL-LTLG, and its runtime advantage increases in k. These fast relative runtimes are a loose lower bound on FAST's performance advantage, as FAST can reap additional speedups by adding up to n processors, whereas PARALLEL-LTLG performs at most $n \log(\varepsilon^{-1})/k$ function evaluations per iteration, so using over 95 processors often does not help. In Section 4.1, we show that on many objectives FAST is faster even with only a single processor.

4.1. Experiments set 1: FAST vs. low-adaptivity algorithms

Our first set of experiments compares FAST to state-of-the-art low-adaptivity algorithms. To accomplish this, we built optimized parallel MPI versions of each of the following algorithms: RANDOMIZED-PARALLEL-GREEDY (Chekuri & Quanrud, 2019b), BINARY-SEARCH-MAXIMIZATION (Fahrbach et al., 2019a), and AMORTIZED-FILTERING (Balkanski et al., 2019a). For any given $\varepsilon > 0$ all these algorithms achieve a $1 - 1/e - \varepsilon$ approximation in $\mathcal{O}(\text{poly}(\varepsilon^{-1}) \log n)$ rounds.

We also compare these low-adaptivity algorithms to an optimized parallel MPI implementation of LAZIER-THAN-LAZY-GREEDY (LTLG) (Mirzasoleiman et al., 2015) (see Appendix C.11). LTLG is widely regarded as the fastest algorithm for submodular maximization in practice, and it has a $(1-1/e-\varepsilon)$ approximation guarantee in expectation.

For calibration, we also ran (1) PARALLEL-GREEDY, a parallel version of the standard GREEDY algorithm, as a heuristic upper bound for the objective value, as well as (2) RANDOM, an algorithm that simply selects k elements uniformly at random.

A fair comparison of the low-adaptivity algorithms' parallel runtimes and solution values is to run each algorithm with parameters that yield the same guarantees, for example a $1-1/e-\varepsilon$ approximation w.p. $1-\delta$ with $\varepsilon = 0.1$ and $\delta = 0.05$. However, this is infeasible since the other low-adaptivity algorithms all require a practically intractable number of queries to achieve any reasonable guarantees, e.g. every round of AMORTIZED-FILTERING would require at least 10^8 samples, even with $n = 500$.

Dealing with benchmarks' practically intractable query complexity. To run other low-adaptivity algorithms despite their huge sample complexity we made two major modifications:

1. **Accelerating subroutines.** We optimize each of the three other low-adaptivity benchmarks by implementing parallel binary search to replace brute-force search and several other modifications that reduce unnecessary queries (for a full description of these fast implementations, see Appendix C.10). These optimizations result in

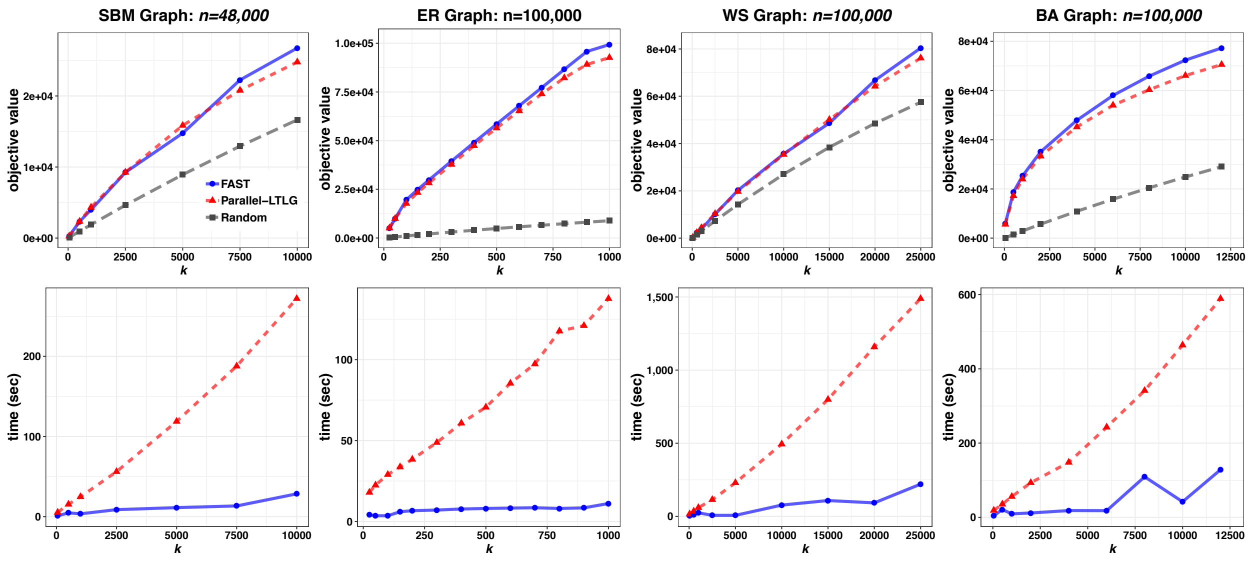

Figure 3. Experiment Set 2.a: FAST *(blue) vs.* PARALLEL-LTLG *(red) on graphs.*

speedups that reduce their runtimes by an order of magnitude in practice, and our implementations are publicly available in our code base. Despite this, it remains practically infeasible to compute these algorithms' high number of samples in practice even on small problems (e.g. $n = 500$ elements);

2. **Using a single query per processor.** Since our interest is in comparing runtime and not quality of approximation, we dramatically lowered the number of queries the three benchmark algorithms require to achieve their guarantees. Specifically, we set the parameters ε and δ for both FAST and the three low-adaptivity benchmarks such that all algorithms guarantee the same $1 - 1/e - 0.1$ approximation with probability 0.95 (see Appendix C.3). However, for the low-adaptivity benchmarks, we reduce their theoretical sample complexity in each round to have exactly **one** sample per processor (instead of their large sample complexity, e.g. 10^8 samples needed for AMORTIZED-FILTERING).

   This reduction in the number of samples per round allows the benchmarks to have each processor perform a single function evaluation per round instead of e.g. $10^8/95$ functions evaluations per processor per round, which 'unfairly' accelerates their runtimes at the expense of their approximations. However, we do *not* perform this reduction for FAST. Instead, we require FAST to compute the *full count* of samples for its guarantees. This is feasible since FAST samples elements rather than sets.

Data sets. Even with these modifications, for tractability we could only use small data sets:

- **Experiments 1.a: synthetic data sets ($n \approx 500$).** To compare the algorithms' runtimes under a range of conditions, we solve max cover on synthetic graphs generated via four different well-studied graph models: *Stochastic Block Model* (SBM); *Erdős Rényi* (ER); *Watts-Strogatz* (WS); and *Barbási-Albert* (BA). See Appendix C.4.1 for additional details;

- **Experiments 1.b: real data sets ($n \approx 500$).** To compare the algorithms' runtimes on real data, we optimize *Sensor Placement* on California roadway traffic data; *Movie Recommendation* on MovieLens data; *Revenue Maximization* on YouTube Network data; and *Influence Maximization* on Facebook Network data. See Appendix C.4.3 for additional details.

Results of experiment set 1. Figures 1 and 2 plot all algorithms' solution values and parallel runtimes for various k on synthetic and real data (each point is the mean of 5 trials with the corresponding k). In terms of solution values, across all experiments, values obtained by FAST are nearly indistinguishable from values obtained by GREEDY—the heuristic upper bound. From this comparison, it is clear that FAST does not compromise on the values of its solutions. In terms of runtime, FAST is 36 to 1600 times faster than BINARY-SEARCH-MAXIMIZATION; 7 to 120 times faster than RANDOMIZED-PARALLEL-GREEDY; 4 to 2200 times faster than AMORTIZED-FILTERING; and 1.1 to 7 times faster than PARALLEL-LTLG on the 8 objectives and various k (the time axes of Figures 1 and 2 are *log-scaled*). Appendix C.12 shows that FAST continues to outperform all benchmarks even (1) when we turn off its lazy updates, and (2) when we run all low-adaptivity benchmarks on just

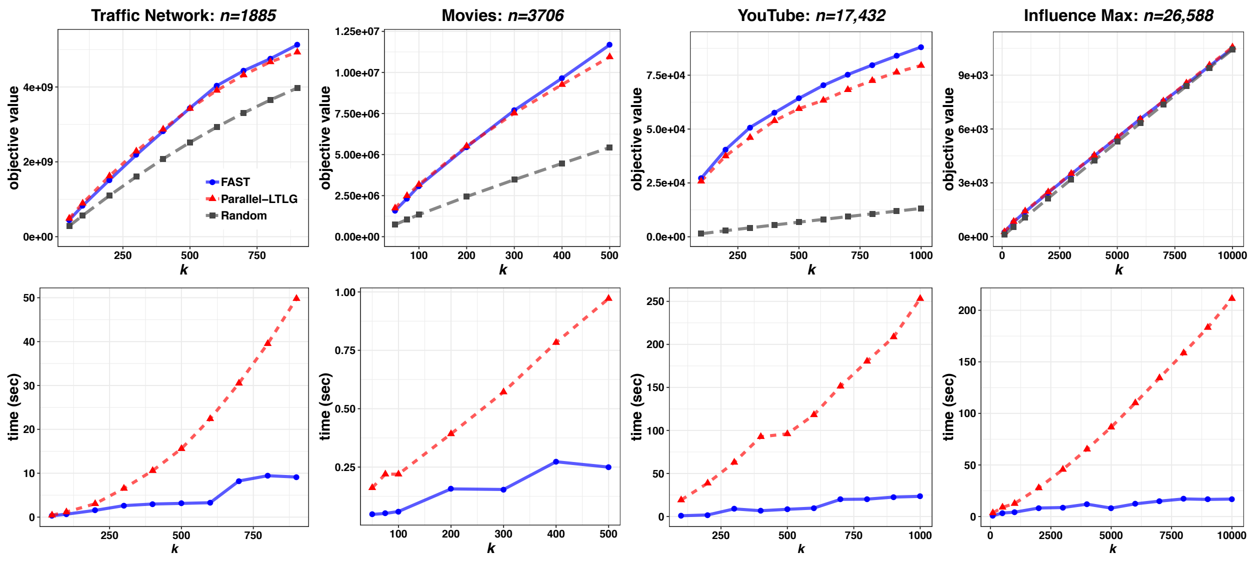

Figure 4. Experiment Set 2.b: FAST *(blue) vs.* PARALLEL-LTLG *(red) on real data.*

a single 'good' guess for OPT. We emphasize that FAST's faster runtimes were obtained despite the fact that the three other low-adaptivity algorithms were run with only a single sample per processor each iteration, rather than the 10^8 or 10^6 samples required for their respective guarantees.

4.2. Experiment set 2: FAST vs. Parallel-Lazier-than-Lazy-Greedy

Our second set of experiments compares FAST to the optimized parallel version of LAZIER-THAN-LAZY-GREEDY (LTLG) (Mirzasoleiman et al., 2015) on large data sets. Specifically, our optimized parallel MPI implementation of LTLG allows us to scale up to random graphs with $n \approx 100000$, large real data with n up to 26000, and various k from 25 to 25000 (see Appendix C.11). For these large experiments, running the parallel GREEDY algorithm is impractical. LTLG has a $(1 - 1/e - \varepsilon)$ approximation guarantee in expectation, so we likewise set both algorithms' parameters ε to guarantee a $(1 - 1/e - 0.1)$ approximation in expectation (see Appendix C.3).

Results of experiment set 2. Figures 3 and 4 plot solution values and runtimes for various k on large experiments with synthetic and real data (each point is the mean of 5 trials). In terms of solution values, while the two algorithms achieved similar solution values across all 8 experiments, FAST obtained slightly higher solution values than PARALLEL-LTLG on most objectives and values of k.

In terms of runtime, FAST was 1.5 to 32 times faster than PARALLEL-LTLG on each of the 8 objectives and all k we tried from $k = 25$ to 25000. More importantly, runtime disparities between FAST and PARALLEL-LTLG increase in larger k, so larger problems exhibit even greater runtime advantages for FAST.

Furthermore, we emphasize that due to the fact that the sample complexity of PARALLEL-LTLG is less than 95 for many experiments, it cannot achieve better runtimes by using more processors, whereas FAST can leverage up to n processors to achieve additional speedups. Therefore, FAST's fast relative runtimes are a loose lower bound for what can be obtained on larger-scale hardware and problems. Figure 5 plots FAST's parallel speedups versus the number of processors we use.

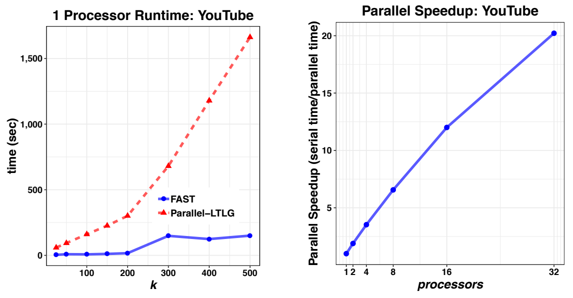

Figure 5. Single processor runtimes for FAST *and* PARALLEL-LTLG*, and parallel speedups vs. number of processors for* FAST *for the YouTube experiment. See Appendix C.14 for details.*

Finally, we note that *even on a single processor*, FAST is faster than LTLG for reasonable values of k on 7 of the 8 objectives due to the fact that FAST often uses fewer queries (see Appendix C.13). For example, Figure 5 plots single processor runtimes for the YouTube experiment.

5. Acknowledgements

This research was supported by a Google PhD Fellowship, NSF grant CAREER CCF-1452961, BSF grant 2014389, NSF USICCS proposal 1540428, NSF Grant 164732, a Google research award, and a Facebook research award.

References

Badanidiyuru, A. and Vondrák, J. Fast algorithms for maximizing submodular functions. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pp. 1497–1514, 2014. doi: 10.1137/1.9781611973402.110. URL https://doi.org/10.1137/1.9781611973402.110.

Balkanski, E. and Singer, Y. The adaptive complexity of maximizing a submodular function. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, pp. 1138–1151. ACM, 2018a.

Balkanski, E. and Singer, Y. Approximation guarantees for adaptive sampling. In *International Conference on Machine Learning*, pp. 393–402, 2018b.

Balkanski, E., Breuer, A., and Singer, Y. Non-monotone submodular maximization in exponentially fewer iterations. *NIPS*, 2018.

Balkanski, E., Rubinstein, A., and Singer, Y. An exponential speedup in parallel running time for submodular maximization without loss in approximation. *SODA*, 2019a.

Balkanski, E., Rubinstein, A., and Singer, Y. An optimal approximation for submodular maximization under a matroid constraint in the adaptive complexity model. *STOC*, 2019b.

CalTrans. Pems: California performance measuring system. http://pems.dot.ca.gov/ [accessed: August 1, 2019].

Chekuri, C. and Quanrud, K. Parallelizing greedy for submodular set function maximization in matroids and beyond. *STOC*, 2019a.

Chekuri, C. and Quanrud, K. Submodular function maximization in parallel via the multilinear relaxation. *SODA*, 2019b.

Chen, L., Feldman, M., and Karbasi, A. Unconstrained submodular maximization with constant adaptive complexity. *STOC*, 2019.

Ene, A. and Nguyen, H. L. Submodular maximization with nearly-optimal approximation and adaptivity in nearly-linear time. *SODA*, 2019a.

Ene, A. and Nguyen, H. L. A nearly-linear time algorithm for submodular maximization with a knapsack constraint. *ICALP*, 2019b.

Ene, A. and Nguyen, H. L. Towards nearly-linear time algorithms for submodular maximization with a matroid constraint. *ICALP*, 2019c.

Ene, A., Nguyen, H. L., and Vladu, A. Submodular maximization with matroid and packing constraints in parallel. *STOC*, 2019.

Esfandiari, H., Karbasi, A., and Mirrokni, V. Adaptivity in adaptive submodularity. *arXiv preprint arXiv:1911.03620*, 2019.

Fahrbach, M., Mirrokni, V., and Zadimoghaddam, M. Submodular maximization with optimal approximation, adaptivity and query complexity. *SODA*, 2019a.

Fahrbach, M., Mirrokni, V. S., and Zadimoghaddam, M. Non-monotone submodular maximization with nearly optimal adaptivity and query complexity. *ICML*, 2019b.

Feldman, M., Harshaw, C., and Karbasi, A. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems 42, 1 (2015), 33 pages.*, 2015.

Harper, F. M. and Konstan., J. A. The movielens datasets: History and context. *ACM Transactions on Interactive Intelligent Systems (TiiS) 5, 4, Article 19 (December 2015), 19 pages.*, 2015. doi: http://dx.doi.org/10.1145/2827872.

Hassidim, A. and Singer, Y. Robust guarantees of stochastic greedy algorithms. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 1424–1432. JMLR. org, 2017.

Kazemi, E., Mitrovic, M., Zadimoghaddam, M., Lattanzi, S., and Karbasi, A. Submodular streaming in all its glory: Tight approximation, minimum memory and low adaptive complexity. *arXiv preprint arXiv:1905.00948*, 2019.

Leskovec, J., Krause, A., Guestrin, C., Faloutsos, C., VanBriesen, J. M., and Glance, N. S. Cost-effective outbreak detection in networks. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Jose, California, USA, August 12-15, 2007*, pp. 420–429, 2007. doi: 10.1145/1281192.1281239. URL https://doi.org/10.1145/1281192.1281239.

Minoux, M. Accelerated greedy algorithms for maximizing submodular set functions. In *Optimization techniques*, pp. 234–243. Springer, 1978.

Mirzasoleiman, B., Badanidiyuru, A., Karbasi, A., Vondrák, J., and Krause, A. Lazier than lazy greedy. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

Mirzasoleiman, B., Badanidiyuru, A., and Karbasi, A. Fast constrained submodular maximization: Personalized data summarization. In *ICML*, pp. 1358–1367, 2016.

Nemhauser, G. L. and Wolsey, L. A. Best algorithms for approximating the maximum of a submodular set function. *Mathematics of operations research*, 3(3):177–188, 1978.

Nemhauser, G. L., Wolsey, L. A., and Fisher, M. L. An analysis of approximations for maximizing submodular set functions—i. *Mathematical Programming*, 14(1): 265–294, 1978.

Qian, S. and Singer, Y. Fast parallel algorithms for statistical subset selection problems. In *Advances in Neural Information Processing Systems*, pp. 5073–5082, 2019.

Rossi, R. A. and Ahmed, N. K. The network data repository with interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015. URL `http://networkrepository.com`.

Traud, A. L., Mucha, P. J., and Porter, M. A. Social structure of Facebook networks. *Phys. A*, 391(16):4165–4180, Aug 2012.

Troyanskaya, O., Cantor, M., Sherlock, G., Brown, P., Hastie, T., Tibshirani, R., Botstein, D., and Altman, R. B. Missing value estimation methods for dna microarrays. *Bioinformatics*, 17(6):520–525, 2001.