

GREEDYML: A Parallel Algorithm for Maximizing Constrained Submodular Functions

Shivaram Gopal 


Purdue University, West Lafayette, IN, USA

S M Ferdous 

Pacific Northwest National Laboratory, Richland, WA, USA

Alex Pothen 

Purdue University, West Lafayette, IN, USA

Hemanta Maji 

Purdue University, West Lafayette, IN, USA

Abstract

We describe a parallel approximation algorithm for maximizing monotone submodular functions subject to hereditary constraints on distributed memory multiprocessors. Our work is motivated by the need to solve submodular optimization problems on massive data sets, for practical contexts such as data summarization, machine learning, and graph sparsification.

Our work builds on the randomized distributed RANDGREEDI algorithm, proposed by Barbosa, Ene, Nguyen, and Ward (2015). This algorithm computes a distributed solution by randomly partitioning the data among all the processors and then employing *a single* accumulation step in which all processors send their partial solutions to one processor. However, for large problems, the accumulation step exceeds the memory available on a processor, and the processor which performs the accumulation becomes a computational bottleneck.

Hence we propose a generalization of the RANDGREEDI algorithm that employs multiple accumulation steps to reduce the memory required. We analyze the approximation ratio and the time complexity of the algorithm (in the BSP model). We evaluate the new GREEDYML algorithm on three classes of problems, and report results from large-scale data sets with millions of elements. The results show that the GREEDYML algorithm can solve problems where the sequential GREEDY and distributed RANDGREEDI algorithms fail due to memory constraints. For certain computationally intensive problems, the GREEDYML algorithm is faster than the RANDGREEDI algorithm. The observed approximation quality of the solutions computed by the GREEDYML algorithm closely matches those obtained by the RANDGREEDI algorithm on these problems.

2012 ACM Subject Classification Computing methodologies → Distributed algorithms; Theory of computation → Distributed algorithms; Theory of computation → Facility location and clustering; Theory of computation → Packing and covering problems; Theory of computation → Nearest neighbor algorithms; Theory of computation → Divide and conquer; Theory of computation → Sparsification and spanners; Theory of computation → Discrete optimization; Computing methodologies → Feature selection

Keywords and phrases Combinatorial optimization, submodular functions, distributed algorithms, approximation algorithms, data summarization

Digital Object Identifier 10.4230/LIPIcs.SEA.2025.19

Supplementary Material *Software (Source Code)*: <https://github.com/smferdous1/Multi-Rand-Greedi.git>, archived at `swb:1:dir:ac12481ab5a7ec665040a687e43a3bd5e744c4ed`

Funding *S M Ferdous*: Laboratory Directed Research and Development Program at PNNL.

Alex Pothen: U.S. Department of Energy grant SC-0022260

Hemanta Maji: NSF CNS-2055605 and CCF-2327981



© Shivaram Gopal, S M Ferdous, Alex Pothen, and Hemanta Maji;
licensed under Creative Commons License CC-BY 4.0

23rd International Symposium on Experimental Algorithms (SEA 2025).

Editors: Petra Mutzel and Nicola Prezza; Article No. 19; pp. 19:1–19:21

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

We describe GREEDYML, a parallel approximation algorithm for maximizing monotone submodular functions subject to hereditary constraints on distributed memory multiprocessors. GREEDYML is built on an earlier distributed approximation algorithm, which has limited parallelism and higher memory requirements. Maximizing a submodular function under constraints is NP-hard, but a natural iterative GREEDY algorithm exists that selects elements based on the marginal gain (defined later) and is $(1 - 1/e) \approx 0.63$ -approximate for cardinality constraints and $1/2$ -approximate for matroid constraints; here e is Euler's number.

Maximizing a submodular function (rather than a linear objective function) promotes diversity in the computed solution since at each step the algorithm augments its current solution with an element with the least properties in common with the current solution. A broad collection of practical problems are modeled using submodular functions, including data and document summarization [23], load balancing parallel computations in quantum chemistry [9], sensor placement [6], resource allocation [28], active learning [11], interpretability of neural networks [8], influence maximization in social networks [13], diverse recommendation [5], etc. Surveys discussing submodular optimization formulations, algorithms, and computational experiments include Tohidi et al. [29] and Krause and Golovin [14].

Our algorithm builds on the RANDGREEDI framework [2], a state-of-the-art randomized distributed algorithm for monotone submodular function maximization under hereditary constraints, which has an approximation ratio half that of the GREEDY algorithm. The RANDGREEDI algorithm randomly partitions the data among all the processors, runs the standard GREEDY algorithm on each partition independently in parallel, and then executes a *single accumulation step* in which all processors send their partial solutions to one processor. However, this accumulation step could exceed the memory available on a processor when the memory is small relative to the size of the data, or when solutions are large. Additionally, the accumulation serializes both the computation and communication and is a bottleneck when scaled to many machines.

Our GREEDYML algorithm brings additional parallelism to this step and can lower the memory and running time by introducing hierarchical accumulation organized through an *accumulation tree*. Similar to RANDGREEDI, we randomly partition the data among all the processors, which constitute the leaves of the accumulation tree. We merge partial solutions at multiple levels in the tree, and the final solution is computed at the root. We prove that the GREEDYML algorithm has a worst-case expected approximation guarantee of $(\alpha b)/(m + b)$, where α is the approximation guarantee for the GREEDY algorithm, b is the branching factor in the tree (the maximum number of children of an internal node), and m is the number of machines (leaves in the accumulation tree). Using the BSP model, we also analyze the time and communication complexity of the GREEDYML and RANDGREEDI algorithms and show that the former has lower computation and communication costs than the latter.

We evaluate the parallel algorithms on three representative and practical submodular function maximization problems: maximum k -set cover, maximum k -vertex dominating set in graphs, and exemplar-based clustering (modeled by the k -medoid problem). We experiment on large data sets with millions of elements that exceed the memory constraints (a few GBs) on a single processor, and discuss how to choose the accumulation tree to have more levels to adapt to the small memory available on a processor. This strategy also enables us to solve for larger values of the parameter k , which corresponds to the size of the solution sought. We also show that the number of function evaluations on the critical path of the

accumulation tree, and hence the run time, could be reduced by the parallel algorithm. In most cases, we find the quality of the solutions computed by GREEDYML closely matches those obtained by the RANDGREEDI algorithm on these problems despite having a worse expected approximation guarantee.

2 Background and Related Work

2.1 Submodular functions

A set function $f: 2^W \rightarrow \mathbb{R}^+$, defined on the power set of a ground set W , is *submodular* if it satisfies the *diminishing marginal gain* property. That is,

$$f(X \cup \{w\}) - f(X) \geq f(Y \cup \{w\}) - f(Y), \text{ for all } X \subseteq Y \subseteq W \text{ and } w \in W \setminus Y.$$

A submodular function f is *monotone* if for every $X \subseteq Y \subseteq W$, we have $f(X) \leq f(Y)$. The *constrained submodular maximization* problem is defined as follows.

$$\max f(S) \text{ subject to } S \in \mathcal{C}, \text{ where } \mathcal{C} \subseteq 2^W \text{ is the family of feasible solutions.}$$

We consider *hereditary constraints*: i.e., for every set $S \in \mathcal{C}$, every subset of S is also in \mathcal{C} . The hereditary family of constraints includes various common ones such as *cardinality constraints* ($\mathcal{C} = \{A \subseteq W : |A| \leq k\}$) and *matroid constraints* (\mathcal{C} corresponds to the collection of independent sets of a matroid).

Lovász extension. For the analysis of our algorithm, we use the *Lovász extension* [21], a relaxation of submodular functions. A submodular function f can be viewed as a function defined over the vertices of the unit hypercube, $f: \{0, 1\}^n \rightarrow \mathbb{R}^+$, by identifying sets $V \subseteq W$ with binary vectors of length $w = |W|$ in which the i^{th} component is 1 if $i \in V$, and 0 otherwise. The *Lovász extension* [21] $\hat{f}: [0, 1]^w \rightarrow \mathbb{R}^+$ is a convex extension that extends f over the entire hypercube and given by, $\hat{f}(x) = \mathbb{E}_{\theta \in \mathcal{U}[0,1]} [f(\{i : x_i \geq \theta\})]$. Here, θ is uniformly

random in $[0, 1]$. The Lovász extension \hat{f} satisfies the following properties [21]:

1. $\hat{f}(1_S) = f(S)$, for all $S \subseteq V$ where $1_S \in [0, 1]^w$ is a vector containing 1 for the elements in S and 0 otherwise,
2. $\hat{f}(x)$ is convex, and
3. $\hat{f}(c \cdot x) \geq c \cdot \hat{f}(x)$, for any $c \in [0, 1]$.

An α -approximation algorithm ($\alpha \in [0, 1]$) for constrained submodular maximization produces a feasible solution $S \subseteq W$, satisfying $f(S) \geq \alpha \cdot f(S^*)$, where S^* is an optimal solution.

2.2 Related Work

GREEDI and RANDGREEDI. The iterative GREEDY algorithm for maximizing constrained submodular functions starts with an empty solution. Given any current solution S , an element is *feasible* if it can be added to the solution without violating the constraints. Given a dataset V and a current solution S , the GREEDY algorithm in each iteration chooses a feasible element $e \in V$ that maximizes the *marginal gain*, $f(S \cup \{e\}) - f(S)$. The algorithm terminates when the maximum marginal gain is zero or all feasible elements have been considered.

We now discuss the GREEDI and RANDGREEDI algorithms, which are the state-of-the-art distributed algorithms for constrained submodular maximization. The GREEDI algorithm [23] partitions the data *arbitrarily* on available machines, and on each machine, it runs the GREEDY algorithm independently to compute a *local* solution. These solutions are then accumulated to a single *global* machine. The GREEDY algorithm is executed again on the accumulated data to obtain a *global* solution. The final solution is the best solution among all the local and global solutions. For a cardinality constraint, where k is the solution size, the GREEDI algorithm has a worst-case approximation guarantee of $1/\Theta(\min(\sqrt{k}, m))$, where m is the number of machines.

Although GREEDI performs well in practice [23], its approximation ratio is not a constant but depends on k and m . Improving on this work, Barbosa et al. [2] proposed the RANDGREEDI algorithm, which partitions the data uniformly at random on machines and achieves an *expected* approximation guarantee of $\frac{1}{2}(1 - 1/e)$ for cardinality and $1/4$ for matroid constraints. In general, it has an approximation ratio of $\alpha/2$ where α is the approximation ratio of the GREEDY algorithm used at the local and global machines. We present the pseudocode of RANDGREEDI framework in Algorithm 1.

■ **Algorithm 1** RANDGREEDI framework for maximizing constrained submodular function.

```

1: procedure RANDGREEDI( $V$ : Dataset,  $m$ : number of machines)
2:    $S \leftarrow \emptyset$ 
3:   Let  $\{P_0, P_1, \dots, P_{m-1}\}$  be an uniform random partition of  $V$ .
4:   Run GREEDY( $P_i$ ) on each machine  $i \in [0, m - 1]$  to compute the solution  $S_i$ 
5:   Place  $S = \bigcup_i S_i$  on machine 0
6:   Run GREEDY( $S$ ) to compute the solution  $T$  on machine 0
7:   return  $\arg \max \{f(T), f(S_1), f(S_2), \dots, f(S_{m-1})\}$ 
8: end procedure

```

Note that for a cardinality constraint, both GREEDI and RANDGREEDI perform $O(nk(k + m))$ calls to the objective function and communicate $O(mk)$ elements to the global machine where n is the number of elements in the ground set, m is the number of machines, and k is solution size.

Both GREEDI and RANDGREEDI require a single global accumulation from the solutions generated in local machines that can quickly become dominating since the runtime, memory, and complexity of this global aggregation grows linearly with the number of machines. We propose to alleviate this by introducing a hierarchical aggregation strategy that maintains an accumulation tree. Our GREEDYML framework generalizes the RANDGREEDI from a single accumulation to a multi-level accumulation. The number of partial solutions to be aggregated depends on the branching factor of the tree, which can be a constant. Thus, the number of accumulation levels grows logarithmically with the number of machines, and the total aggregation is not likely to become a memory, runtime, and communication bottleneck with the increase in the number of machines. We refer to Appendix A for the detailed complexity comparisons of the RANDGREEDI and our GREEDYML algorithm.

Other work. Early approaches on distributed submodular maximization includes the $1/(2 + \epsilon)$ -approximate SAMPLE AND PRUNE algorithm by Kumar et al. [17], which requires $O(1/\delta)$ rounds assuming $O(kn^\delta \log n)$ memory per machines. Here, $\delta > 0$ is a user parameter. GREEDI [23] and RANDGREEDI [2] are shown to be more efficient in practice than the SAMPLE AND PRUNE algorithm.

More recent distributed approaches [4, 24, 25] use the multi-linear extension to map the problem into a continuous function. They typically perform a gradient ascent on each local machine and build a consensus solution in each round, which improves the approximation factor to $(1 - 1/e)$. However, we do not believe that these approaches are practical since they involve expensive gradient computations (could be exponential-time). Most of these algorithms are not implemented, and the one reported implementation solves problems with only a few hundred elements in the data set [25].

A shared-memory parallel algorithm, the FAST algorithm [3], uses adaptive sequencing to speed up the GREEDY algorithm. Adaptive sequencing is a technique to add several elements in one round to the current solution. First, all elements with large marginal gains with respect to the current solution are selected. They are then randomly permuted, and prefixes with large average marginal gains are considered. The subset that is added is chosen to be a largest subset with sufficiently high average marginal gain. These sequencing rounds are repeated until the solution has the desired cardinality. However, this algorithm does not scale to larger numbers of machines and does not work in memory-restricted contexts. Parallelism in this algorithm occurs in setting a number of threshold values to determine what constitutes a sufficiently large marginal gain.

A more recent algorithm is the distributed DASH algorithm [7], which replaces the GREEDY algorithm in the RANDGREEDI framework with a similar adaptive sequencing algorithm. This algorithm has the same memory and processor bottlenecks as RANDGREEDI. It can be used instead of the greedy algorithm in the GREEDYML framework proposed here.

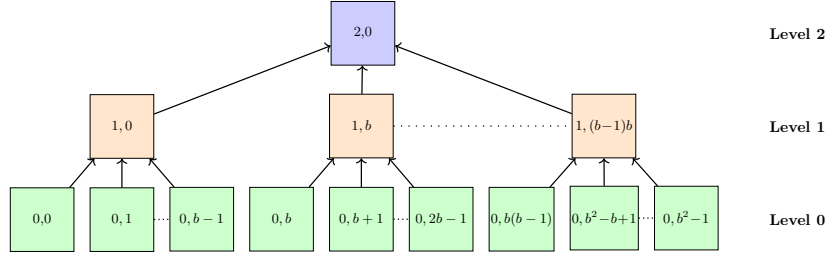
3 Description of Our Algorithm

We describe and analyze our algorithm that generalizes the RANDGREEDI algorithm from a single accumulation step to multiple accumulation steps. Each accumulation step corresponds to a level in an *accumulation tree*, which we describe next.

3.1 Data Structure and Preliminaries

Accumulation tree. Given a problem and an algorithm to solve it, its memory requirements (to store the data and associated data structures), and machines of a specified memory size, we choose the number of machines m needed to solve the problem. We identify the m machines by the set of ids: $\{0, 1, \dots, m - 1\}$. These machines are organized into a tree structure that we call the *accumulation tree*, since it governs how local solutions are merged hierarchically to compute the final solution. Every machine is a leaf in the tree, and the interior nodes of the tree correspond to *subsets* of machines. The machine corresponding to an interior node is the one that corresponds to its left-most child. Figure 1 shows an example of a generic accumulation tree with b^2 leaves, where the maximum number of children of an interior node, its *branching factor*, is b . Each node in the tree is identified by a tuple, its level in the tree, and the ID of the machine corresponding to the node.

The problem is partitioned randomly among the machines (leaves of the tree). The size of the subset of the data assigned to a machine (and associated data structures) must fit within the memory available on a machine. Each leaf computes solutions from its subset of the data, and shares this solution with its parent node. Each interior node collects solutions from its children, unions them all together, and then computes a solution from this latter data. Every interior node must also have sufficient memory for the union of solutions from its children. It then obtains the best solution from among the solutions it received from its children and the solution it computed at this step, and communicates it to its parent. The root node finally



■ **Figure 1** An accumulation tree with $L = 2$ levels, $m = b^2$ machines, and a branching factor b . Each node has a label of the form (ℓ, id) . Here there are b nodes as children at each level, but when there are fewer than b^L leaf nodes, then the number of children at levels closer to the root may be fewer than b .

$$\text{GREEDYML}(\ell, id) = \begin{cases} \text{GREEDY}(P_{id}) & \ell = 0 \\ \arg \max \begin{cases} \text{GREEDY} \left(\bigcup_{i \in \{0,1,\dots,b-1\}} \text{GREEDYML}(\ell-1, id + i \cdot b^{\ell-1}) \right) \\ \text{GREEDYML}(\ell-1, id) \end{cases} & id \bmod b^\ell = 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

■ **Figure 2** The recurrence relation for the multilevel GREEDYML which is defined for each node in the accumulation tree. We denote the random subset assigned to machine id by P_{id} .

reports the best solution from among the local solutions of its children and the solution it computed from the union of the local solutions. Thus, the edges of the tree determine the accumulation pattern of the intermediate solutions. The number of accumulation levels (i.e., one minus the height of the tree), denoted by L , is $\lceil \log_b m \rceil$. When m is less than b^L , nodes at the higher levels may have fewer than b children. We characterize an accumulation tree T by the triple $T(m, L, b)$, where m is the number of leaves (machines), L is the number of levels, and b is the branching factor.

Observe that the id parameter remains the same in multiple nodes that are involved in computations at multiple levels. For our analysis, we keep the branching factor constant across all levels.

Randomness. The randomness in the algorithm is *only* in the initial placement of the data on the machines, and we use a random tape to encapsulate this. The random tape r_W has a randomized entry for each element in W to indicate the machine containing that element. Any expectation results proved henceforth are over the choice of this random tape. Moreover, if the data accessible to a node is V , we consider the randomness over just r_V . Whenever the expectation is over r_V , we denote the expectation as \mathbb{E}_V .

Recurrence relation. Figure 2 shows the recurrence relation that forms the basis of the GREEDYML algorithm, defined for every node in the accumulation tree; it will be the basis for the multilevel distributed algorithm. At level 0 (leaves), the recurrence function returns the GREEDY solution of the random subset of data P_{id} assigned to it. At other levels (internal nodes), it returns the better among the GREEDY solution computed from the union of the received solution sets of its children and its solution from its previous level. It is undefined for (ℓ, id) tuples that do not correspond to nodes in the tree (at higher levels). The detailed pseudocode of our algorithm is presented in Algorithm 2.

3.2 Pseudocode of GREEDYML

Algorithm 2 describes our multilevel distributed algorithm using two procedures. The first procedure GREEDYML is a wrapper function that sets up the environment to run the distributed algorithm. The second function GREEDYML' is the iterative implementation of the recurrence relation that runs on each machine. The wrapper function partitions the data into m subsets and assigns them to the machines (Line 2). Then each machine runs the GREEDYML' function on the subset assigned to it (Line 5, Line 7). The wrapper function uses and returns the solution from machine 0 (Line 8) as it is the root of the accumulation tree.

The GREEDYML' procedure is an iterative implementation of the recurrence relation 2 that runs on every machine. Each machine checks whether it needs to be active at a particular level (Line 5) and decides whether it needs to receive from (Line 11) or send to other machines (Line 6). The function returns the solution from the last level of the machine.

■ **Algorithm 2** Our Randomized Multi-level GREEDYML Algorithm.

```

1: procedure GREEDYML( $V$ : Dataset,  $b$ : branching factor,  $m$ : number of machines,  $r$ : random
   tape)
2:   Let  $\{P_0, P_1, \dots, P_{m-1}\}$  be uniform random partition of  $V$  using  $r$ .
3:   for  $i = 1 \dots m - 1$  in parallel do                                 $\triangleright$  Run GREEDYML' on all machines except 0
4:      $\ell = \text{level}(i, b)$                                                $\triangleright \text{level}(i, b) = \max_l \{l : id \bmod b^l \text{ is } 0\}$ 
5:     Run GREEDYML'( $V_i, \ell, b, i$ ) to obtain  $S_i$  on machine  $i$ 
6:   end for
7:   Run GREEDYML'( $V_0, \lceil \log_b m \rceil, b, 0$ ) to obtain  $S_0$  on machine 0
8:   return  $S_0$ 
9: end procedure

1: procedure GREEDYML'( $P$ : Partial Data-set,  $\ell$ : levels;  $b$ : branching factor,  $id$ : machine ID)
2:    $S = \text{GREEDY}(P)$ 
3:    $S_{prev} = S$ 
4:   for  $i = 1 \dots \ell$  do
5:     if  $id \neq \text{parent}(id, i)$  then
6:       Send  $S_{prev}$  to  $\text{parent}(id, i)$                                  $\triangleright \text{parent}(id, i) = b^i \cdot \lfloor id/b^i \rfloor$ 
7:       break
8:     end if
9:      $D = S_{prev}$                                                          $\triangleright$  Prepare  $D$  for current iteration
10:    for  $j = 1 \dots b - 1$  do
11:      Receive  $D_j$  from  $\text{child}(id, i, j)$                              $\triangleright \text{child}(id, i, j) = id + j \cdot b^{i-1}$ 
12:       $D = D \cup D_j$ 
13:    end for
14:    Run GREEDY( $D$ ) to obtain  $S$ 
15:     $S_{prev} = \arg \max \{f(S), f(S_{prev})\}$ 
16:  end for
17:  return  $S_{prev}$ 
18: end procedure

```

4 Analysis of Our Algorithm

We prove the *expected approximation ratio* of GREEDYML algorithm in Theorem 4 using three Lemmas. We restate a result from [2] that applies to the leaves of our accumulation tree and characterizes elements that do not change the solution computed by the GREEDY algorithm.

► **Lemma 1** ([2]). *If we have $\text{GREEDY}(V \cup \{e\}) = \text{GREEDY}(V)$, for each element $e \in B$, then $\text{GREEDY}(V \cup B) = \text{GREEDY}(V)$.*

The next two Lemmas connect the quality of the computed solutions to the optimal solution at the internal nodes (in level one) of the accumulation tree. Lemma 2 provides a lower bound on the expected function value of the *individual* solutions of the GREEDY algorithm received from the leaf nodes, while Lemma 3 analyzes the expected function value of the GREEDY execution over the *accumulated* partial solutions.

Let $p: V \rightarrow [0, 1]$ be a probability distribution over the elements in V , and $A \sim V(1/m)$ be a random subset of V such that each element is independently present in A with probability $1/m$. The probability p is defined as follows:

$$p(e) = \begin{cases} \Pr_{A \sim V(1/m)}[e \in \text{GREEDY}(A \cup \{e\})], & \text{if } e \in \text{OPT}; \\ 0, & \text{otherwise.} \end{cases}$$

For any leaf node, the distribution p defines the probability that each element of OPT is in the solution of the GREEDY algorithm when it is placed in the node.

► **Lemma 2.** *Let c be a leaf node of the accumulation tree, S_c be the solution computed from c , and $V_c \subset V_n$ be the elements considered in forming S_c . If GREEDY is an α -approximate algorithm, then $\mathbb{E}_{V_n}[f(S_c)] \geq \alpha \cdot \hat{f}(1_{\text{OPT}} - p)$.*

Proof. We first construct a subset of OPT that contains all the elements that do not appear in S_c when added to some leaf node in the subtree rooted at child c . Let O_c be the rejected set that can be added to V_c without changing S_c ; i.e., $O_c = \{e \in \text{OPT} : e \notin \text{GREEDY}(V_c \cup \{e\})\}$. Therefore, $\Pr[e \in O_c] = 1 - \Pr[e \notin O_c] = 1 - p(e)$.

From Lemma 1, we know that $\text{GREEDY}(V_c \cup O_c) = \text{GREEDY}(V_c)$. Since the rejected set $O_c \subseteq \text{OPT}$ and the constraints are hereditary, $O_c \in \mathcal{C}$ (i.e., O_c is a feasible solution of child node c). Then from the condition of Lemma 2, we have

$$\begin{aligned} f(S_c) &\geq \alpha \cdot f(O_c) \\ \mathbb{E}[f(S_c)] &\geq \alpha \cdot \mathbb{E}[f(O_c)] = \alpha \cdot \hat{f}(\mathbb{E}_{V_n}[1_{O_c}]) = \alpha \cdot \hat{f}(1_{\text{OPT}_{\ell, id}} - p_{\ell, id}). \end{aligned}$$

◀

► **Lemma 3.** *Let D be the union of all the solutions computed by the b children of an internal node $(1, id)$ in the accumulation tree, and S be the solution from the GREEDY algorithm on the set D . If GREEDY is an α -approximate algorithm, then $\mathbb{E}_{V_n}[f(S)] \geq \frac{b * \alpha}{m} \cdot \hat{f}(p)$.*

Proof. We first show a preliminary result on the union set D . Consider an element $e \in D \cap \text{OPT}$ present in some solution S_c from a child c . Then,

$$\Pr[e \in S_c | e \in V_c] = \Pr[e \in \text{GREEDY}(V_c) | e \in V_c].$$

Since the distribution of $V_c \sim V(1/m)$ conditioned on $e \in V_c$ is identical to the distribution of $B \cup \{e\}$, where $B \sim V(1/m)$, we have,

$$\Pr[e \in S_c | e \in V_c] = \Pr_{B \sim V(1/m)}[e \in \text{GREEDY}(B \cup \{e\})] = p(e).$$

Since this result holds for every child c , and each subset V_c is disjoint from the corresponding subsets mapped to the other children, we have

$$\Pr(e \in D \cap \text{OPT}) = \sum_i \Pr[e \in S_{c_i} \cap \text{OPT} | e \in V_{c_i}] \Pr[e \in V_{c_i}] = \sum_i p * 1/m = bp/m.$$

Now, we are ready to prove the Lemma. The subset $D \cap OPT_{\ell, id} \in \mathcal{C}$, since it is a subset of $OPT_{\ell, id}$ and the constraints are hereditary. Further, since the GREEDY algorithm is α -approximate, we have

$$\begin{aligned}
 f(S) &\geq \alpha \cdot f(D \cap OPT_{\ell, id}) \\
 \mathbb{E}_{V_n}[f(S)] &\geq \mathbb{E}_{V_n}[\alpha \cdot f(D \cap OPT)] \\
 &\geq \alpha \cdot \widehat{f}(\mathbb{E}_{V_n}[1_{D \cap OPT}]) && [\text{Lovász Ext. (2), 2.1}] \\
 &= \alpha \cdot \widehat{f}(bp/m) = \alpha b/m \cdot \widehat{f}(p). && [\text{Lovász Ext. (3)2.1}] \tag{1}
 \end{aligned}$$

◀

► **Theorem 4.** *Let $T(m, L, b)$ be an accumulation tree, V be the ground set, and r_V be a random mapping of elements of V to the leaves of the tree T . Let OPT be an optimal solution computed from V for the constrained submodular function f . If GREEDY is an α -approximate algorithm, then $\mathbb{E}[f(\text{GREEDYML}(V))] \geq \frac{b \cdot \alpha}{(m+b)} f(OPT)$.*

Proof. We concentrate on a node at level 1, where after obtaining the partial solutions from the children of this node, we compute the GREEDY on the union of these partial solutions. Let S_c be any of the partial solutions, S be the union of these partial solutions, and T be $\arg \max\{f(S), f(S_c)\}$. From Lemma 2 and Lemma 3,

$$\mathbb{E}[f(T)] \geq \frac{b\alpha}{m} \cdot \widehat{f}(p) \text{ and } \mathbb{E}[f(T)] \geq \alpha \cdot \widehat{f}(1_{OPT} - p).$$

By multiplying the first inequality by m/b and then adding it to the second, we get

$$\begin{aligned}
 (m/b + 1)\mathbb{E}[f(T)] &\geq \alpha \cdot (\widehat{f}(1_{OPT} - p) + \widehat{f}(p)) = \alpha \cdot \widehat{f}(1_{OPT}) && [\text{Lovász Ext. (2), 2.1}] \\
 \mathbb{E}[f(T)] &\geq \frac{\alpha}{(m/b + 1)} \cdot \widehat{f}(1_{OPT}).
 \end{aligned}$$

The theorem follows since the solution quality can only improve at higher levels of the tree. ◀

5 Experimentation

5.1 Experimental setup

We conduct experiments to demonstrate that our algorithms are capable of overcoming the memory limitations of the GREEDY and RANDGREEDI algorithms and can solve large-scale constrained submodular maximization problems. We also compare these algorithms with respect to runtimes and the quality of solutions.

All the algorithms are executed on a cluster computer with 448 nodes, each of which is an AMD EPYC 7662 node with 256 GB of total memory shared by the 128 cores. Each core operates at 2.0 GHz frequency. The nodes are connected with a 100 Gbps HDR Infiniband network. To simulate a distributed environment on this cluster, we needed to ensure that the memory is not shared between nodes. Therefore, in what follows, a machine will denote one node with just one core assigned for computation, but having access to all 256 GB of memory. We also found that this made the runtime results more reproducible.

For our experimental evaluation, we report the *runtime* and *quality* of the algorithms being compared. For runtime, we exclude the file reading time on each machine, and for the quality, we show the objective function value of the corresponding submodular function.

■ **Table 1** Properties of datasets used in the experiments. Here $\delta(u)$ represents: the number of neighbors of vertex u for the k -dominating set problem, the cardinality of the subset u for the k -cover problem, and the size of the vector representation of the pixels of image u for the k -medoid problem.

Function	Dataset	$n = V $	$\sum_u \delta(u)$	avg. $\delta(u)$
k -dominating set	AGATHA_2015	183,964,077	11,588,725,964	63.32
	MOLIERE_2016	30,239,687	6,669,254,694	220.54
	com-Friendster	65,608,366	1,806,067,135	27.52
	road_usa	23,947,347	57,708,624	2.41
	road_central	14,081,816	33,866,826	2.41
	belgium_osm	1,441,295	3,099,940	2.14
k -cover	webdocs	1,692,082	299,887,139	177.22
	kosarak	990,002	8,018,988	8.09
	retail	88,162	908,576	10.31
k -medoid	Tiny ImageNet	100,000	1,228,800,000	12,288

Since the RANDGREEDI and GREEDYML are distributed algorithms, we also report the *number of function calls in the critical path* of the computational tree, which represents the parallel runtime of the algorithm. Given an accumulation tree, the number of function calls in the critical path refers to the maximum number of function calls that the algorithm makes along a path from the leaf to the root. In our implementation, this quantity can be captured by the number of function calls made by the nodes of the accumulation tree with $id = 0$ since this node participates in the function calls from all levels of the tree.

Datasets. In this paper, we limit our experiments to cardinality constraints using three different submodular functions described in detail in Appendix A. Other hereditary constraints add more computation to the problem and will greatly increase the run times of the experiments.

Our benchmark dataset is shown in Table 1. They are grouped based on the objective function and are sorted by the $\sum_u \delta(u)$ values within each group (see the Table for a definition). For the k -dominating set, our testbed consists of the Friendster social network graph [31], a collection of road networks from *DIMACS10* dataset, and the *Sybrandt* dataset. We chose road graphs since they have relatively small average vertex degrees, leading to large vertex-dominating sets. We chose the *Sybrandt* collection [26][27] since it is a huge data set of machine learning graphs. For the k -cover objective, we use popular set cover datasets from the *Frequent Itemset Mining Dataset Repository* [10]. For the k -medoid problem, we use the *Tiny ImageNet* dataset [18].

MPI Implementation. GREEDYML is implemented using C++11, and compiled with g++9.3.0, using the O3 optimization flag. We use the Lazy Greedy [22] variant that has the same approximation guarantee as the GREEDY but is faster in practice since it potentially reduces the number of function evaluations needed to choose the next element (by using the monotone decreasing gain property of submodular functions). Our implementation of the GREEDYML algorithm uses the OpenMPI library for inter-node communication. We use the MPI_Gather and MPI_Gatherv primitives to receive all the solution sets from the children (Line 11 in Algorithm 2). We generated custom MPI_Comm communicators to enable this communication using MPI_Group primitives. Customized communicators are required since every machine has different children at each level. Additionally, we use the MPI_Barrier primitive to synchronize all the computations at each level.

5.2 Experimental Results

The experiments are executed with different accumulation trees that vary in the number of machines (m), the number of levels (L), and the branching factors (b), to assess their performance. We repeat each experiment six times and report the geometric mean of the results. Unless otherwise stated, a machine in our experiments represents a node in the cluster with only one core assigned for computation. Whenever memory constraints allow, we compare our results with the sequential GREEDY algorithm that achieves a $(1 - 1/e)$ approximation guarantee.

5.2.1 Experiments with memory limit

Here we show the memory advantage of our GREEDYML algorithm w.r.t RANDGREEDI with two experiments. In the first one, we impose a limit of 100 MB of space for each node and vary k , the solution size. This also simulates how the new algorithm can find applications in the *edge-computing* context. We also fix k and vary the memory limits, necessitating different numbers of nodes to fit the data in the leaves. We observe the quality and runtime of different accumulation tree structures in these two experiments. Both these experiments are designed to show that the RANDGREEDI algorithm quickly runs out of memory with increasing m and k , and by choosing an appropriate accumulation tree, our GREEDYML algorithm can solve the problem with negligible drop in accuracy. For these experiments, we will choose the shortest accumulation tree that can be used with the memory limit and k values.

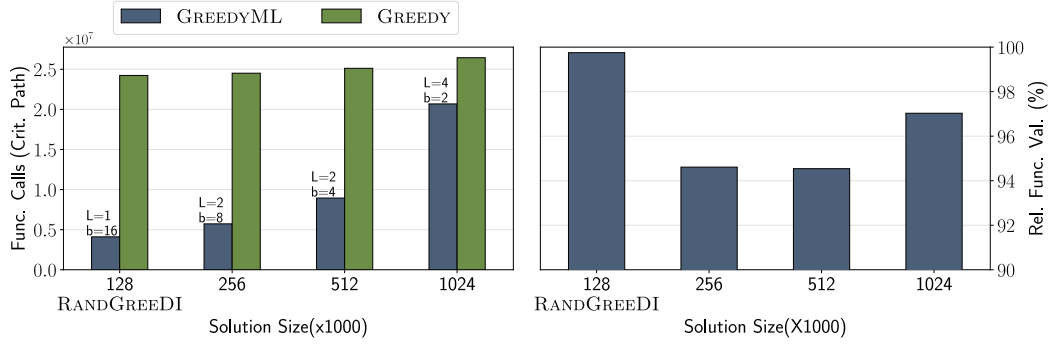
Varying k . For this experiment, we use 16 machines with a limit on the available memory of 100 MB per machine and vary k from 128,000 to 1,024,000 for the k -dominating set problem on the `road_usa` [1] dataset. The small memory limit in this experiment can also be motivated from an *edge computing* context.

The left plot in Figure 3 shows the number of function calls with varying values of k for the GREEDY (green bars) and GREEDYML algorithms (blue bars). Note that when $L = 1$ (the left-most bar in the Figure), the GREEDYML algorithm corresponds to the RANDGREEDI algorithm. For the GREEDYML (and the RANDGREEDI), we are interested in the number of function calls in the critical path since it represents the parallel runtime of the algorithm. With our memory limits, only $k = 128,000$ instances can be solved using the RANDGREEDI algorithm.

As we increase k , we can generate solutions using our GREEDYML with different accumulation trees. The corresponding lowest-depth accumulation tree with the number of levels and branching factor is shown on top of the blue bars. The result shows that the number of function evaluations on the critical path in the GREEDYML algorithm is smaller than the number of function evaluations in the sequential GREEDY algorithm. While the number of function calls for accumulation trees with smaller b values is larger than RANDGREEDI, we see that GREEDYML can solve the problems with larger k values in the same machine setup, which was not possible with RANDGREEDI. But it comes with a trade-off on parallel runtime. We observe that as we make the branching factor smaller, the number of function calls in the critical path increases, suggesting that it is sufficient to choose the accumulation trees with the largest branching factor (thus the lowest depth tree) whenever the memory allows it.

The right plot of Figure 3 shows the relative objective function value, i.e., the relative number of vertices covered by the dominating set compared to the GREEDY algorithm, with

19:12 GreedyML: Maximizing Constrained Submodular Functions



■ **Figure 3** Results from GREEDYML for the k -dominating set problem on the road_usa dataset on 16 nodes with varying k . The pair (L, b) shows the number of levels and branching factors chosen for specific k values. The function values are relative to the GREEDY algorithm. Note that the leftmost bars in both plots represent the RANDGREEDI results.

■ **Table 2** Results for k -dominating set on the Friendster, road_usa and webdocs datasets. The memory size per machine is varied for the Friendster dataset. The number of machines m and the accumulation tree are selected based on the size of the data and the size of the solutions to get three different machine configurations. We report the function values relative to the GREEDY algorithm and the execution time in seconds. Note that the 4GB entries run with $L = 1$ corresponding to the RANDGREEDI (RG) algorithm. We use the same three machine organizations for the road_usa and webdocs datasets to show they follow similar trends in solution quality and execution time.

Dataset	Alg.	Mem. Limit	m	b	L	Rel. Func.(%)	Time (s.)
Friendster	RG	4GB	8	8	1	99.959	61.994
	GML	2GB	16	4	2	99.903	61.352
	GML	1GB	32	2	5	99.793	79.997
MOLIERE_2016	RG	8GB	8	8	1	99.257	121.318
	GML	4GB	16	4	2	99.106	108.764
	GML	2GB	32	2	5	98.990	161.139
AGATHA_2015	RG	12GB	8	8	1	99.996	94.122
	GML	6GB	16	4	2	99.995	99.574
	GML	3GB	32	2	5	99.989	104.156

varying k . The figure shows that the RANDGREEDI and GREEDYML algorithms attain quality at most 6% less than the serial GREEDY algorithm. Similar trends can be observed for other datasets.

Varying memory limits. This experiment demonstrates that the memory efficiency of the GREEDYML algorithm enables us to solve problems on parallel machines, whereas the RANDGREEDI and GREEDY cannot solve them due to insufficient memory. Unlike the previous experiment (Varying k), where we selected the accumulation trees based on k , here, we fix k and choose accumulation trees based on the memory available on the machines. We consider the k -dominating set problem and report results on the Friendster [31], AGATHA_2015[27], and MOLIERE_2016[26] dataset in Table 2. For the Friendster dataset, we choose k such that the k -dominating set requires 512 MB, roughly a factor of 64 smaller than the original graph. The RANDGREEDI algorithm (the first row) can execute this problem only on 8 machines, each with 4 GB of memory, since in the accumulation step, one

machine receives solutions of size 512 MB each from 8 machines. The GREEDYML algorithm, having multiple levels of accumulation, can run on 16 machines with only 2 GB memory, using $L = 2$ and $b = 4$. Furthermore, it can also run on 32 machines with only 1 GB memory, using $L = 5$ and $b = 2$. We repeat the same experiment for the other two datasets with these three machine configurations, with corresponding memory restrictions.

We show relative quality and running time for the three datasets from these configurations in Table 2. Our results show that function values computed by the GREEDYML algorithm (the 2 and 1 GB results) are insensitive to the number of levels in the tree. As expected, increasing the number of levels in the accumulation tree increases the execution times due to the communication and synchronization costs involved. However, aggregating at multiple levels enables us to solve large problems by overcoming memory constraints. So, in this scenario, it is sufficient to select the number of machines depending on the size of the dataset and then select the branching factor such that the accumulation step does not exceed the memory limits. We also notice that the RANDGREEDI algorithm has an inherently serial accumulation step, and the GREEDYML algorithm provides a mechanism to parallelize it.

5.2.2 Selecting the Accumulation tree

Now we show how the accumulation tree may be chosen to reduce runtime (or a proxy, the number of function calls in the critical path) when the number of machines is fixed.

In this experiment, we show results for the k -dominating set and k -coverage problem by fixing the number of machines and varying branching factors, the number of levels in the accumulation tree, and the solution size k .

In Figure 4, we provide summary results on the number of function evaluations in the critical path relative to the GREEDY algorithm and the running times by taking a geometric mean over all nine datasets.

Three subfigures (top left, top right, and bottom left) of Figure 4 show the *execution time* in seconds for the GREEDYML and RANDGREEDI algorithms, as the number of levels and the parameter k are varied. When k is small (top left), there is less variation in the execution time since work performed on the leaves dominates overall time. As k increases (bottom left), the GREEDYML algorithm becomes faster than the RANDGREEDI algorithm ($L = 1, b = 32$). Note that although Figure 4 presents the geometric mean results over all nine datasets, the runtime and the function values for the individual datasets follow the same trend. The largest and smallest reduction in runtime we observe is on the *belgium_osm* and *kosarak* datasets with a reduction of around 22% and 1%, respectively, for all k values.

The bottom right plot fixes $k = 32,000$ and shows the *number of function calls* in the critical path of the accumulation tree relative to the GREEDY algorithm for different (L, b) pairs. Here, the leftmost bar represents the RANDGREEDI algorithm. We observe that the relative number of function calls for RANDGREEDI is around 70% of GREEDY, whereas the GREEDYML (with $L = 2$ and $b = 8$) reduces it by 15 percent. From Table 5, the number of function calls at a leaf node is $O(nk/m)$, while at an accumulation node, it is $O(mk^2)$, for the RANDGREEDI algorithm. Hence, the accumulation node dominates the computation since it has a quadratic dependence on k , becoming a bottleneck for large k values. This plot also shows that the number of function calls is a good indicator of the algorithm's run time and that the cost of function evaluations dominates the overall time. The other factor affecting run time is communication costs, which are relatively small and grow with the number of levels when k is very large.

We note (not shown in the figure) that generally the objective function values obtained by the GREEDYML algorithm are not sensitive to the choice of the number of levels and the

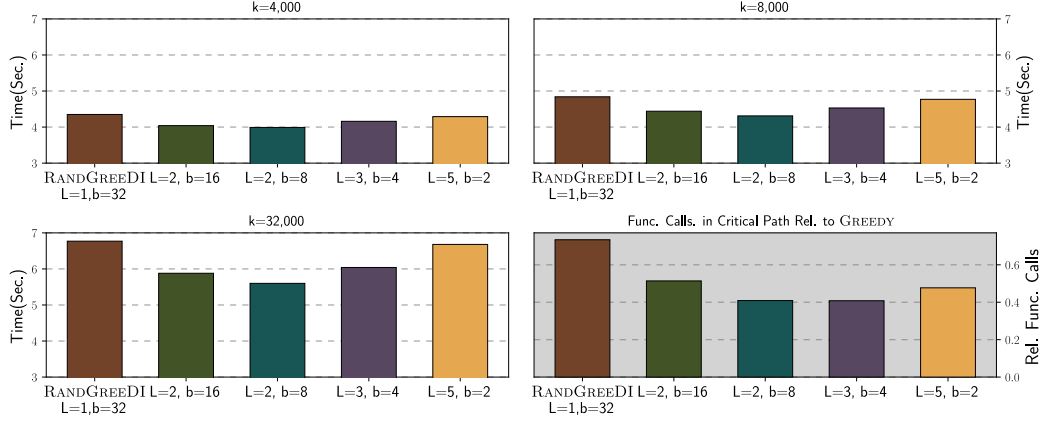


Figure 4 Geometric means of results from GREEDYML for k -dominating set (on road datasets) and for k -cover (on set cover benchmarks) using 32 machines. Different L and b values represent different accumulation tree configurations. The top and bottom left subfigures show execution times for different k values and accumulation trees. The bottom right plot shows the geometric means of the number of function calls in the critical path relative to the GREEDY algorithm for $k = 32,000$.

Table 3 Objective function values relative to the GREEDY algorithm at the first and final accumulation steps for the Friendster dataset with selection size $k = 1000$ and $m = 32$. Recall that L denotes the level of the root node, and b is the branching factor, in the accumulation tree.

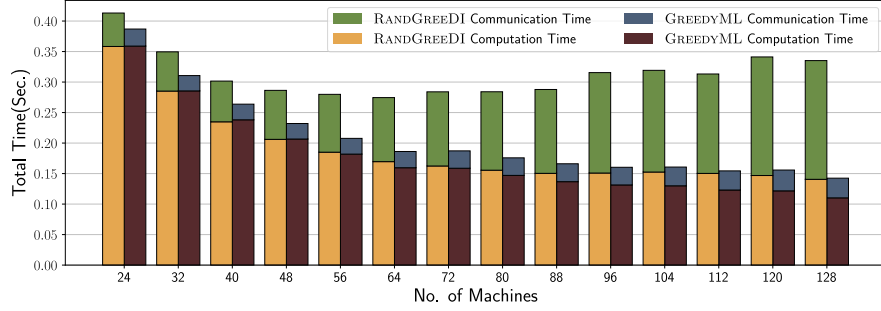
L	b	Accumulation Step	
		First	Final
3	4	0.74111	0.99994
2	8	0.82971	1.00005
2	16	0.91965	0.99994
1	32	1.00003	1.00003

branching factors of the accumulation tree and differ by less than 1% from the values of the RANDGREEDI algorithm. For the webdocs k -coverage problem, however, GREEDY quality is about 20% higher than both the RANDGREEDI and GREEDYML.

In Table 3, we report the improvement in objective function value when accumulating over multiple levels, over choosing to stop at level one of accumulation. We use the maximum k -cover function for the Friendster dataset with $k = 1000$ for different branching factors at the first and final accumulation levels. We observe that the objective values at the highest accumulation level are not very sensitive to the tree parameters, contrary to their sensitivity to the approximation ratio derived in Theorem 4.

5.2.3 Scaling results

Here we perform a strong scaling experiment to show how computation and communication times vary for the RANDGREEDI and GREEDYML algorithms. For the latter algorithm, we use the tallest accumulation tree by using a branching factor of two, thereby increasing the number of accumulation steps. Our results will show that even though the RANDGREEDI algorithm has a low asymptotic communication cost, it can become a bottleneck when scaled to a large number of machines, and that our algorithm alleviates this bottleneck.



■ **Figure 5** Strong scaling results of the RANDGREEDI and GREEDYML algorithms for $k = 50$ on Friendster dataset for the k -dominating set problem. We set $b = 2$ for the GREEDYML algorithm.

Next, we show how the GREEDYML algorithm alleviates the scaling bottlenecks of the RANDGREEDI algorithm using the k -dominating set problem on the Friendster dataset. We set the branching factor $b = 2$ for the GREEDYML algorithm since this has the highest number of levels and, thus, the lowest approximation guarantee. We compare communication and computation times against the RANDGREEDI algorithm from 8 to 128 machines with $k = 50$.

In Figure 5, we plot the total execution time by stacking communication and computation times for the two algorithms. For RANDGREEDI, the communication time scales poorly since it increases linearly with the number of machines, $O(km)$ (See Table 5). But, for GREEDYML algorithm (with a constant branching factor, $b = 2$, $L = \log_2 m$), the communication cost is $O(k \log m)$, which grows logarithmically in the number of machines. Figure 5 shows that the total communication times of the GREEDYML algorithm are consistently around 0.25 seconds, whereas the RANDGREEDI increases from 0.05 seconds to 0.2 seconds. We observe that computation times for both RANDGREEDI and GREEDYML change similarly with m , indicating that the majority of the computation work is performed at the leaf nodes. For computation time, we observe a slightly worse scaling of RANDGREEDI compared to GREEDYML, again because the central node becomes a computational bottleneck as m increases. Similar to other experiments, we observe (not shown in the plot) an almost identical quality in the solutions, where the GREEDYML solution has a quality reduced by less than 1% from that of the RANDGREEDI algorithm.

5.2.4 The k -medoid problem

In this final subsection, we perform experiments for the k -medoid objective function (one that is computationally more expensive than the others we have used here) and show that we can provide a significant speedup by using taller accumulation trees without loss in quality. The k -medoid function is extensively used in machine learning as a solution to exemplar-based clustering problems.

Our dataset consists of the *Tiny ImageNet* dataset [18] containing 100K images (64×64 pixels) with 200 different classes, with 500 images from each class. We convert and normalize the image into a vector and use Euclidean distance to measure dissimilarity. We define an auxiliary vector e_0 as a pixel vector of all zeros. Note that, unlike the other two functions, the k -medoid function requires access to the full dataset to compute the functional value. Since the dataset is distributed, this poses an issue in the experiment. To overcome this, following [23, 2], we calculate the objective function value using only the images available

■ **Table 4** Results from GREEDYML for the k -medoid function on the *Tiny ImageNet* data set using 32 machines organized into different accumulation trees. The table shows the relative function values and speedup compared to the RANDGREEDI algorithm using two different schemes for computing the local objective functions. Higher values are better for both schemes. Recall that L and b are the number of levels and the branching factor, respectively.

L	b	Local Obj.			Added Images		
		Rel.	Func.	Speedup	Rel.	Func.	Speedup
		Val.	(%)		Val.	(%)	
5	2	92.22		2.00	93.69		2.01
3	4	92.21		1.96	92.70		1.94
2	8	92.73		1.95	92.77		1.93
2	16	92.22		1.49	93.34		1.44

locally on each machine. This means the ground set for each machine is just the images present in that machine. Additionally, they [23, 2] have also added subsets of randomly chosen images to the central machine to provide practical quality improvement. We have followed these techniques (*local only* and *local with additional images*) in the experiments for our multilevel GREEDYML algorithm.

In our experiments, we set k to 200 images, fix the number of machines ($m = 32$), and vary the accumulation trees by choosing different L and b . For the variant with additional images, we add 1,000 random images from the original dataset to each accumulation step.

In Table 4, we show the relative objective function values and speedup for different accumulation trees relative to the RANDGREEDI algorithm. We observe that the objective function values for GREEDYML algorithm are almost similar to RANDGREEDI. Our results show that the GREEDYML algorithm becomes gradually faster as we increase the number of levels, with runtime improvement ranging from $1.45 - 2.01\times$. This is because the k -medoid function is computationally intensive, where computation cost increases quadratically with the number of images (Table 5). With $k = 200$ and $m = 32$, the RANDGREEDI algorithm has $km = 6,400$ images at the root node but only $n/m = 313$ images at the leaves; thus, the computation at the root node dominates in cost. On the other hand, as we decrease the branching factor (from $b = 16$ to 2), the number of images (kb) in the interior nodes decreases from 3,200 to 400 for the GREEDYML algorithm. This gradual decrease in compute time is reflected in the total time and in the observed speedup.

Finally, in Fig. 6 (Appendix B), we show 16 out of the 200 images determined to be cluster centers by the GREEDYML and RANDGREEDI algorithms. We can conclude that the submodular k -medoid function can generate a diverse set of exemplar images for this clustering problem.

6 Conclusion and Future work

We have developed a new distributed algorithm, GREEDYML, that enhances the existing distributed algorithm for maximizing a constrained submodular function. We prove GREEDYML is $\alpha \cdot b/(b + m)$ -approximate, but empirically demonstrate that its quality is close to the best approximation algorithms for several practical problems. Our algorithm alleviates the inherent serial computation and communication bottlenecks of the RANDGREEDI algorithm while reducing memory requirements. This enables submodular maximization to be applied to massive-scale problems effectively.

Future work could experiment with other hereditary constraints, such as matroid and p -system constraints. Another direction is to apply GREEDYML to closely related non-monotone and weakly submodular functions. Since our experiments suggest that GREEDYML delivers higher quality solutions than the expected approximation guarantees, one could investigate whether the approximation ratio could be improved.

References

- 1 David A Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. Graph partitioning and graph clustering. In *Contemporary Mathematics*, volume 588. American Mathematical Society, 2012. 10th DIMACS Implementation Challenge Workshop.
- 2 Rafael da Ponte Barbosa, Alina Ene, Huy L. Nguyen, and Justin Ward. The power of randomization: Distributed submodular maximization on massive datasets. In *Proceedings of the 32nd International Conference on Machine Learning*, pages 1236–1244. JMLR.org, 2015. URL: <http://proceedings.mlr.press/v37/barbosa15.html>.
- 3 Adam Breuer, Eric Balkanski, and Yaron Singer. The FAST algorithm for submodular maximization. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 1134–1143. PMLR, 13–18 July 2020. URL: <https://proceedings.mlr.press/v119/breuer20a.html>.
- 4 Chandra Chekuri and Kent Quanrud. Submodular function maximization in parallel via the multilinear relaxation. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 303–322. Society for Industrial and Applied Mathematics, 2019. doi:10.1137/1.9781611975482.20.
- 5 Laming Chen, Guoxin Zhang, and Eric Zhou. Fast greedy MAP inference for determinantal point process to improve recommendation diversity. In *Advances in Neural Information Processing Systems*, pages 5627–5638, 2018. URL: <https://proceedings.neurips.cc/paper/2018/hash/dbbf603ff0e99629dda5d75b6f75f966-Abstract.html>.
- 6 M. Coutino, S. P. Chepuri, and G. Leus. Submodular sparse sensing for Gaussian detection with correlated observations. *IEEE Transactions on Signal Processing*, 66:4025–4039, 2018. doi:10.1109/TSP.2018.2846220.
- 7 Tonmoy Dey, Yixin Chen, and Alan Kuhnle. Dash: A distributed and parallelizable algorithm for size-constrained submodular maximization. *Proceedings of the AAAI Conference on Artificial Intelligence*, 37(4):3941–3948, June 2023. doi:10.1609/aaai.v37i4.25508.
- 8 E. Elenberg, A. G. Dimakis, M. Feldman, and A. Karbasi. Streaming weak submodularity: Interpreting neural networks on the fly. In *Advances in Neural Information Processing Systems*, pages 4044–4054, 2017.
- 9 S M Ferdous, Alex Pothen, Arif Khan, Ajay Panyala, and Mahantesh Halappanavar. A parallel approximation algorithm for maximizing submodular b -matching. In *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA)*, pages 45–56, 2021. doi:10.1137/1.9781611976830.5.
- 10 FIMI. Frequent itemset mining dataset repository. <http://fimi.uantwerpen.be/data/>, 2003.
- 11 D. Golovin and A. Krause. Adaptive submodularity: Theory and applications in active learning and stochastic optimization. *Journal of Artificial Intelligence Research*, 42:427–486, 2011. doi:10.1613/JAIR.3278.
- 12 Leonard Kaufman and Peter J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley, 1990. doi:10.1002/9780470316801.
- 13 David Kempe, Jon Kleinberg, and Éva Tardos. Maximizing the spread of influence through a social network. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 137–146, 2003. doi:10.1145/956750.956769.

- 14 Andreas Krause and Daniel Golovin. Submodular function maximization. In *Tractability: Practical Approaches to Hard Problems*, pages 71–104. Cambridge University Press, 2014. doi:10.1017/CB09781139177801.004.
- 15 Andreas Krause and Carlos Guestrin. Near-optimal observation selection using submodular functions. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence*, volume 7, pages 1650–1654, 2007. URL: <http://www.aaai.org/Library/AAAI/2007/aaai07-265.php>.
- 16 Andreas Krause, Jure Leskovec, Carlos Guestrin, Jeanne VanBriesen, and Christos Faloutsos. Efficient sensor placement optimization for securing large water distribution networks. *Journal of Water Resources Planning and Management*, 134(6):516–526, 2008.
- 17 Ravi Kumar, Benjamin Moseley, Sergei Vassilvitskii, and Andrea Vattani. Fast greedy algorithms in mapreduce and streaming. *ACM Trans. Parallel Comput.*, 2(3):14:1–14:22, 2015. doi:10.1145/2809814.
- 18 Fei-Fei Li and Andrej Karpathy. Tiny imagenet challenge. <http://cs231n.stanford.edu/tiny-imagenet-200.zip>, 2017. [Online; last accessed 13-Mar-2024].
- 19 Hui Lin and Jeff Bilmes. Multi-document summarization via budgeted maximization of submodular functions. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 912–920, 2010. URL: <https://aclanthology.org/N10-1134/>.
- 20 Hui Lin and Jeff Bilmes. A class of submodular functions for document summarization. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 510–520, 2011. URL: <https://aclanthology.org/P11-1052/>.
- 21 L. Lovász. Submodular functions and convexity. In Achim Bachem, Bernhard Korte, and Martin Grötschel, editors, *Mathematical Programming The State of the Art: Bonn 1982*, pages 235–257. Springer Berlin Heidelberg, 1983. doi:10.1007/978-3-642-68874-4_10.
- 22 Michel Minoux. Accelerated greedy algorithms for maximizing submodular set functions. In J. Stoer, editor, *Optimization Techniques*, pages 234–243. Springer, 1978.
- 23 Baharan Mirzasoleiman, Amin Karbasi, Rik Sarkar, and Andreas Krause. Distributed submodular maximization: Identifying representative elements in massive data. In *Advances in Neural Information Processing Systems*, volume 26, pages 2049–2057, 2013. URL: <https://proceedings.neurips.cc/paper/2013/hash/84d2004bf28a2095230e8e14993d398d-Abstract.html>.
- 24 Aryan Mokhtari, Hamed Hassani, and Amin Karbasi. Decentralized submodular maximization: Bridging discrete and continuous settings. In *Proceedings of the 35th International Conference on Machine Learning*, pages 3616–3625, 2018.
- 25 Alexander Robey, Arman Adibi, Brent Schlotfeldt, Hamed Hassani, and George J. Pappas. Optimal algorithms for submodular maximization with distributed constraints. In *Proceedings of the 3rd Conference on Learning for Dynamics and Control*, pages 150–162, 2021. URL: <http://proceedings.mlr.press/v144/robey21a.html>.
- 26 Justin Sybrandt, Michael Shtutman, and Ilya Safro. Moliere: Automatic biomedical hypothesis generation system. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '17*, pages 1633–1642, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3097983.3098057.
- 27 Justin Sybrandt, Ilya Tyagin, Michael Shtutman, and Ilya Safro. Agatha: Automatic graph mining and transformer based hypothesis generation approach. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management, CIKM '20*, pages 2757–2764, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3340531.3412684.
- 28 K. Thekumparampil, A. Thangaraj, and R. Vaze. Combinatorial resource allocation using submodularity of waterfilling. *IEEE Transactions on Wireless Communications*, 15:206–216, 2016. doi:10.1109/TWC.2015.2469291.

- 29 Ehsan Tohidi, Rouhollah Amiri, Mario Coutino, David Gesbert, Geert Leus, and Amin Karbasi. Submodularity in action: From machine learning to signal processing applications. *IEEE Signal Processing Magazine*, 37(5):120–133, 2020. doi:10.1109/MSP.2020.3003836.
- 30 Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990. doi:10.1145/79173.79181.
- 31 Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, pages 1–8, 2012.

A Submodular Functions and Complexity

Our algorithm can handle any hereditary constraint, but we consider only cardinality constraints in our experiments to keep the run times low. More general constraints involve additional computations to check if an element can be added to the current solution set and satisfy the constraints. They increase the computation time but not the communication time, and we believe the GREEDYML algorithm will perform even better relative to the RANDGREEDI algorithm. Cardinality constraints are widely used in various applications such as sensor placement [16], text, image, and document summarization [19, 20], and information gathering [15]. The problem of maximizing a submodular function under cardinality constraints can be expressed as follows.

$$\begin{aligned} \max_{S \subseteq V} \quad & f(S) \\ \text{s.t.} \quad & |S| \leq k. \end{aligned}$$

Here V is the ground set, f is a non-negative monotone submodular function, and k is the size of the solution set S .

In our experiments, we have considered the following three submodular functions.

k -cover. Given a ground set B , a collection of subsets $V \subseteq 2^B$, and an integer k , the goal is to select a set $S \subseteq V$ containing k of these subsets to maximize $f(S) = |\bigcup_{S_i \in S} S_i|$.

k -dominating set. The k -dominating set problem is a special case of the k -cover problem defined on graphs with the ground set V as the set of vertices. We say a vertex $u \in V$ *dominates* all its adjacent vertices (denoted by $\delta(u)$). Our goal is to select a set S of k vertices to dominate as many vertices as possible, i.e., $f(S) = |\bigcup_{u \in S} \delta(u)|$. The marginal gain of any vertex is the number of vertices in its neighborhood that are not yet dominated. Therefore, the problem shows diminishing marginal gains and is submodular.

k -medoid problem. The k -medoid problem [12] is used to compute exemplar-based clustering, which asks for a set of exemplars (cluster centers) representatives of a large dataset. Given a collection of elements in a ground set V , and a dissimilarity measure $d(u, v)$, we define a loss function (denoted by L) as the average pairwise dissimilarity between the exemplars (S) and the elements of the data set, i.e., $L(S) = 1/|V| \sum_{u \in V} \min_{v \in S} d(u, v)$. Following [23], we turn this loss minimization to a submodular maximization problem by setting $f(S) = L(\{e_0\}) - L(S \cup \{e_0\})$, where e_0 is an auxiliary element specific to the dataset. The goal is to select a subset $S \subseteq V$ of size k that maximizes $f(S)$.

Next, we analyze the computational and communication complexity of our GREEDYML algorithm using the bulk synchronous parallel (BSP) model of parallel computation [30]. We denote the number of elements in the ground set by $n = |V|$, the solution size by k , the number of machines by m , and the number of levels in the accumulation tree by L .

Computational Complexity. The number of objective function calls by the sequential GREEDY algorithm is $O(nk)$, since k elements are selected to be in the solution, and we may need to compute $O(n)$ marginal gains for each of them. Each machine in RANDGREEDI algorithm makes $O(k(n/m + mk))$ function calls, where the second term comes from the accumulation step. Each machine of the GREEDYML algorithm with branching factor b makes $O(k(n/m + Lbk))$ calls. Recall that $L = \lceil \log_b m \rceil$.

We note that the time complexity of a function call depends on the specific function being computed. For example, in the k -coverage and the k -dominating set problems, computing a function costs $O(\delta)$, where δ is the size of the largest itemset for k -coverage, and the maximum degree of a vertex for the vertex dominating set. In both cases, the runtime complexity is $O(\delta k(n/m + mk))$ for the RANDGREEDI, and $O(\delta k(n/m + Lbk))$ for the GREEDYML algorithm. The k -medoid problem computes a local objective function value and has a complexity of $O(n'\delta)$ where δ is the number of features, and n' is the number of elements present in the machine. For the leaves of the accumulation tree, $n' = n/m$, and for interior nodes, $n' = bk$. Therefore its complexity is $O(k\delta((n/m)^2 + (mk)^2))$ for the RANDGREEDI, and $O(k\delta((n/m)^2 + L(bk)^2))$ for the GREEDYML algorithm.

Communication Complexity. Each edge in the accumulation tree represents communication from a machine at a lower level to one at a higher level and contains four messages. They are the indices of the selected elements of size k , the size of the data associated with each selection (proportional to the size of each adjacency list ($\leq \delta$)), the total size of the data elements, and the data associated with each selection. Therefore the total volume of communication is $O(k\delta)$ per child. Since at each level, a parent node receives messages from b children, the communication complexity is $O(k\delta Lb)$ for each parent. Therefore the communication complexity for the RANDGREEDI algorithm is $O(k\delta m)$ and for the GREEDYML algorithm is $O(k\delta L \lceil m^{1/L} \rceil)$. We summarize these results in Table 5.

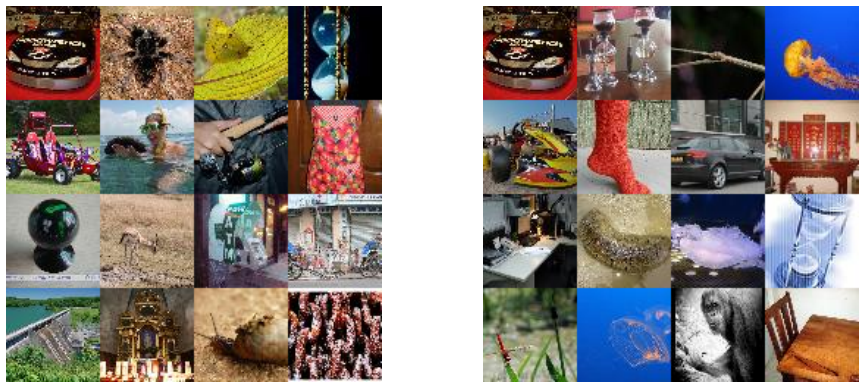
■ **Table 5** Complexity Results of the submodular functions for different algorithms. The number of elements in the ground set is n , the selection size is k , the number of machines is m , and the number of levels in the accumulation tree is L .

Algorithms	Metric	GREEDY	RANDGREEDI	GREEDYML
All	Elements per leaf node	n	n/m	n/m
	Calls per leaf node	nk	nk/m	nk/m
	Elements per interior node	0	km	$k \lceil m^{1/L} \rceil$
	Calls per interior node	0	$k^2 m$	$k^2 \lceil m^{1/L} \rceil$
	Total Function Calls	kn	$k(n/m + km)$	$k(n/m + Lk \lceil m^{1/L} \rceil)$
k -cover / k -dominating set		δ : subset size/number of neighbours		
	Cost Per call	δ	δ	δ
	Computational complexity	δkn	$\delta k(n/m + km)$	$\delta k(n/m + Lk \lceil m^{1/L} \rceil)$
	Communication cost	0	δkm	$\delta kL \lceil m^{1/L} \rceil$
k -medoid		δ : number of features		
	Cost Per call in Leaf node	δn	$\delta n/m$	$\delta n/m$
	Cost Per call in interior node	0	δkm	$\delta k \lceil m^{1/L} \rceil$
	Computational complexity	δkn^2	$\delta k((n/m)^2 + (km)^2)$	$\delta k((n/m)^2 + L(k \lceil m^{1/L} \rceil)^2)$
	Communication cost	0	δkm	$\delta kL \lceil m^{1/L} \rceil$

■ **Table 6** Notations and parameters used in the paper.

Parameter	Description	Parameter	Description
α	Approximation ratio of the GREEDY algorithm	W	Complete universe for the input dataset
b	Branching factor of the accumulation tree	w	Size of the universe W
	Number of leaves in the accumulation tree	V	Input Dataset
m	Numbers of machines used for computation.	n	Size of the input dataset V
L	Number of levels of the accumulation tree	V_c	Dataset corresponding to any node c of the tree
ℓ	level identifier for a node	S	Solution Set
id	Machine identifier for a node.	k	Size of solution
f	Submodular function	OPT	The optimal solution
\hat{f}	Lovász extension of function f		Probability that $e \in OPT$ is selected
P_{id}	Part of the dataset assigned to machine id	$p(e)$	by the GREEDY algorithm when sampled from V .

B Cluster Centers (Images) selected by GREEDYML and RANDGREEDI algorithms



■ **Figure 6** Results from GREEDYML for the k -medoid problem on the Tiny ImageNet dataset on 32 nodes with $k = 200$, and no images added at each accumulation step. The subfigure on the left shows the top 16 images for one of the runs of the GREEDYML algorithm with branching factor $b = 2$, and the subfigure on the right shows the top 16 images for one of the runs of the RANDGREEDI algorithm.