GREEDYML: A PARALLEL ALGORITHM FOR MAXIMIZING SUBMODULAR FUNCTIONS

SHIVARAM GOPAL*, S M FERDOUS†, HEMANTA K. MAJI*, AND ALEX POTHEN*

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 applications in areas 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 could exceed the memory available on a processor, and the processor which performs the accumulation could become a computational bottleneck.

Here, 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 massive 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 can be 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.

Keywords: Combinatorial optimization, submodular functions, parallel algorithms, approximation algorithms, data summarization.
Mathematics Subject Classification: 90C27, 68W10.

1. Introduction.

We describe a scalable parallel approximation algorithm for maximizing monotone submodular functions subject to hereditary constraints on distributed memory multiprocessors. We build on an earlier distributed approximation algorithm which has limited parallelism and higher memory requirements. Although this problem is NP-hard (the objective function is nonlinear), a Greedy algorithm that maximizes the marginal gain (defined later) at each step is $(1 - 1/\mathrm{e}) \approx 0.63$-approximate for cardinality constraints and 1/2-approximate for matroid constraints; here e is Euler's number.

Combinatorial optimization with a submodular objective function (rather than a linear objective function) leads to diversity in the computed solution, since at each step the algorithm chooses an element with the least properties in common with the current solution set. A broad collection of optimization problems could be modeled using submodular functions, including data and document summarization [24], load balancing parallel computations in quantum chemistry [9], sensor selection [6], resource allocation [27], active learning [11], interpretability of neural networks [7], influence maximization in social networks [13], diverse recommendation [5] etc. Submodular optimization problems often have efficient approximation algorithms to solve them, since submodular functions have properties that make them discrete analogs of both convex and concave continuous functions. Surveys discussing submodular optimization formulations, algorithms, and computational experiments include Tohidi et al. [28] 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 un-

*Purdue University.
†Pacific Northwest National Lab

der 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 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, this merging step serializes both the computation and communication and is a bottleneck when scaled to more machines.

The new GREEDYML algorithm brings additional parallelism to this step and can lower the memory and time required to solve the problem. We randomly partition the data among all the processors, which constitute the leaves of an *accumulation tree*, and then merge partial solutions at multiple levels in the tree. We prove that the GREEDYML algorithm has a worst-case approximation guarantee of $1/(L+1)$ of the serial GREEDY algorithm, where L is the total number of accumulation levels in the accumulation tree. Using the BSP model, we analyze the time complexity of both computation and communication steps in 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 the maximum k-set cover problem, the maximum k-vertex dominating set in graphs, and exemplar-based clustering (modeled by the k-medoid problem); all of these problems arise in data reduction or summarization. We experiment on massive data sets with millions of elements that exceed the memory constraints (a few GBs) on a single processor.

We demonstrate how solutions may be computed using the parallel algorithm by organizing the accumulation tree to have more levels to adapt to the memory available on a processor. This strategy also enables us to solve for larger values of the parameter k in the problems discussed above, which corresponds to the size of the solution sought. We show that the number of function evaluations on the critical path of the accumulation tree, and hence the run time, could be reduced when the parallel algorithm is employed. Also, we do not observe the deterioration in objective function values expected from the worst-case approximation ratio of the GREEDYML algorithm, and the observed approximation quality of the computed solutions closely matches those obtained by the RANDGREEDI algorithm on these problems.

2. Background and Related Work.

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

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

A submodular function f is *monotone* if for every $X \subseteq Y \subseteq W$, we have $f(X) \leqslant f(Y)$. The *constrained submodular maximization* problem maximizes a submodular function subject to certain constraints:

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

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 ($\mathcal{C} = \{A \subseteq W : |A| \leqslant k\}$) and matroid constraints ($\mathcal{C}$ corresponds to the collection of independent sets of a matroid).

2.2. Lovász extension. For the analysis of our algorithm, we use the *Lovász extension* [20], a relaxation of submodular functions. A submodular function f can be viewed as a function defined over the vertices of the unit hypercube, $f : \{0,1\}^n \to \mathbb{R}^+$, by identifying sets $V \subseteq W$ with binary vectors of length $n = |W|$ in which the i^{th} component is 1 if $i \in V$, and 0 otherwise. The *Lovász extension* [20] $\widehat{f} : [0,1]^n \to \mathbb{R}^+$ is a convex extension that extends f over the entire hypercube, which is given by

$$\widehat{f}(x) = \mathop{\mathbb{E}}_{\theta \in \mathcal{U}[0,1]} [\, f(\{i : x_i \geqslant \theta\}) \,].$$

Here, θ is uniformly random in $[0,1]$. For any submodular function f, the Lovász extension $\widehat{f}$ satisfies the following properties [20]:

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

An α-*approximation* algorithm (where $\alpha \in [0,1)$) for maximizing a submodular function $f : 2^W \to \mathbb{R}^+$ subject to a hereditary constraint $\mathcal{C}$ produces a solution $S \subseteq W$ with $S \in \mathcal{C}$, satisfying $f(S) \geqslant \alpha \cdot f(S^*)$, where S^* is an optimal solution.

2.3. The GREEDI and RANDGREEDI Algorithms. The GREEDY algorithm (shown in Algorithm 2.1) for maximizing submodular functions subject to constraints is an iterative algorithm that starts with an empty solution. Given the current solution, an element is *feasible* if it can be added to the solution without violating the constraints. In each iteration, the GREEDY algorithm chooses a feasible element $e \in V$ that maximizes the marginal gain, $f(S \cup \{e\}) - f(S)$, w.r.t. the current solution S. The algorithm terminates when the maximum marginal gain is zero or all elements in the ground set have been considered.

Algorithm 2.1 GREEDY Algorithm

```
1:  procedure GREEDY (V: Dataset)
2:      S ← ∅
3:      while True do
4:          E ← {e ∈ V \ S : S ∪ {e} ∈ C}
5:          e' ← arg max_{e∈E} f(S ∪ {e})
6:          if f(S ∪ {e'}) = f(S) or E = ∅ then
7:              break
8:          end if
9:          S ← S ∪ {e'}
10:     end while
11:     return S
12: end procedure
```

We now discuss the GREEDI and RANDGREEDI, which are the state-of-the-art distributed algorithms for constrained submodular maximization. The GREEDI algorithm [24] partitions the data *arbitrarily* on available machines, and on each partition, it runs the GREEDY algorithm in parallel to compute a *local* solution. These solutions are then sent to a single *global* machine, where they are accumulated. The GREEDY algorithm is then again executed on the accumulated data to get 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

Algorithm 2.2 RandGreedi framework for maximizing constrained submodular function

1: **procedure** RandGreedi(V: Dataset, m: number of machines)
2: $S \leftarrow \emptyset$
3: Let $\{P_0, P_1, \ldots, P_{m-1}\}$ 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** $\arg\max \{f(T), f(S_1), f(S_2), \ldots, f(S_{m-1})\}$
8: **end procedure**

worst-case approximation guarantee of $1/\Theta(\min(\sqrt{k}, m))$, where m is the number of machines.

Although the Greedi algorithm performs well in practice [24], its approximation ratio is not a constant but depends on k. To improve the approximation guarantee of Greedi algorithm, Barbosa et al. proposed the RandGreedi algorithm [2]. By partitioning the data uniformly at random on machines, RandGreedi achieves an expected approximation guarantee of $\frac{1}{2}(1 - 1/\mathrm{e})$ 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 2.2. Note that for a cardinality constraint, both Greedi and RandGreedi perform $O(nk(k+m))$ calls to the objective function and has $O(mk)$ elements communicated to the single central 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. This single accumulation step 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.

2.4. Other Related Work. Kumar et al. [17] have developed the sample and prune algorithm which achieves an expected approximation ratio of $1/(2 + \varepsilon)$ for k-cardinality constraints, using $O(1/\delta)$ rounds, when the memory per machine is $O(kn^{\delta} \log n)$, where $\delta > 0$ is a parameter, and n is the number of elements in the ground set. Barbosa et al. [2] have compared their RandGreedi algorithm with this one and show that the former performs better than the latter for the practical quality of the computed approximate solution. They observed this even though the sample and prune algorithm has a better worst-case approximation ratio in expectation.

More recent work on distributed submodular maximization uses the multi-linear extension to map the submodular optimization problem into a continuous domain. This line of work [4, 25, 26] typically performs a gradient ascent on each local machine and builds a consensus solution in each round, which improves the approxima-

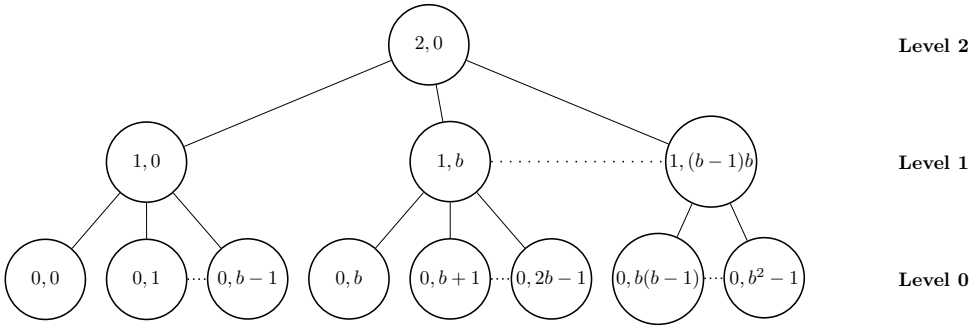

Fig. 1: An accumulation tree with $L = 2$ levels, $m = b^2$ machines, and a branching factor b. Each node has a label of the form (ℓ, id). Here there are b nodes as children at each level, but when there are fewer than b^L leaf nodes, then the number of children at levels closer to the root may be fewer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

tion factor to $(1 - 1/\mathrm{e})$. However, we believe these latter papers represent primarily a theoretical contribution rather than one that leads to practical algorithms. The reason is the high (exponential) cost of computing a single gradient by sampling many points; even randomized approximations of gradient computations are expensive. Most of these algorithms are not implemented and the ones with implementation solve problems with only a hundred elements in the data set [26].

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. We assume that there are m machines identified by the set of ids: $\{0, 1, \ldots, m - 1\}$.

Accumulation Tree. An accumulation tree (T) is defined by the number of machines (m), and branching factor (b). It has the same structure as a complete b-ary tree with m leaves which means all the leaves are at the same depth. The tree nodes correspond to processors along with the corresponding subset of data accessible to them. The edges of the tree determine the accumulation pattern of the intermediate solutions. The final solution is generated on the root node of T. Thus, the branching factor b of the tree indicates the maximum number of nodes that send data to its parent. For each internal node of the tree, we attempt to have exactly b children. Note that since we plan to construct a complete b-ary tree, in the case where m is not multiple of b, in each level of the tree, there could be at most one node whose arity is less than b. The number of accumulation levels, L (i.e., the height of the tree minus 1) is $\lceil \log_b m \rceil$.

To uniquely identify a node in the tree, we will assign an identifier (ℓ, id) to each node of T, where ℓ represents the accumulation level of the node and id represents the machine id corresponding to the node. The id for each leaf node is the id of the machine that the leaf node corresponds to. All the leaf nodes are at level 0. Each internal node receives the lowest id of its children, i.e., any node (l, i) has node $(l + 1, \lfloor i/b^{l+1} \rfloor * b^{l+1})$ as the parent. Therefore the root node will always have level L with id value 0. Also, 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.

Figure 1 shows an example of a generic accumulation tree with b^2 leaves and branching factor b. The number of accumulation levels is the level of the root. Here

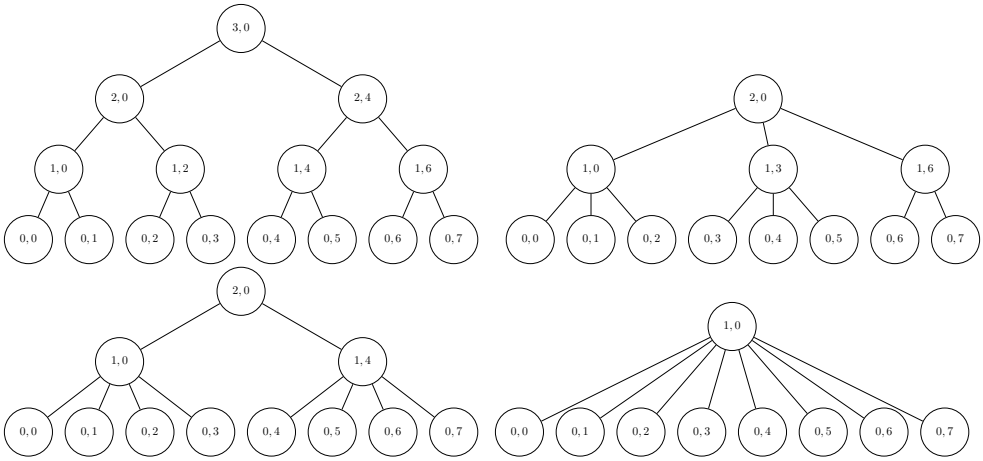

Fig. 2: Accumulation tree with 8 machines and branching factors 2 (top-left), 3 (top-right), 4 (bottom-left), and 8 (bottom-right). The level inside a node represents the identification of the node.

we have $L = \lceil \log_b b^2 \rceil = 2$. Figure 2 shows accumulation trees with 8 leaves and with branching factors 2, 3, 4, and 8. The trees with branching factors 2 and 8 have the same branching factor for every internal node as these trees have b^L nodes. But the tree with branching factor 3 has the last node in level 1 with only 2 children. Similarly, the tree with branching factor 4 has the last node in level 2 i.e. the root with 2 children. Observe that the *id* parameter remains the same in multiple nodes that are involved in computations at multiple levels. For this paper, we show analysis by keeping the branching factor constant across all levels.

Data Accessibility. We use P_{id} to denote the elements assigned to machine id. To indicate the data *accessible* to a particular node in the tree, we describe a set for the input data set as $V_{\ell,id}$. It corresponds to all the data used to compute the solution at the node (ℓ, id) and consists of all the elements assigned to its descendants:

$$V_{\ell,id} = \bigcup_{i=0}^{\min(b^\ell - 1, m - id)} P_{id+i}.$$

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 3 shows a recurrence relation defined for every node in the accumulation tree and will be the basis for our 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

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

Fig. 3: 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}.

that do not correspond to nodes in the tree (at higher levels). We call our algorithm associated with the recurrence relation as the GreedyML algorithm.

We can compare it with the RandGreedi algorithm by looking at the recurrence relation at level one. Our recurrence relation takes the arg max for the accumulated solution and *one* solution from the previous level. However, the RandGreedi algorithm takes the arg max of the accumulated solution and the *best* solution from the children. Our choice reduces the computation at the internal node. We show that this modification produces the same approximation ratio as the RandGreedi algorithm.

Pseudocode. Algorithm 3.1 describes our multilevel distributed algorithm in two functions. The first function 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 3 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.

4. Analysis of Our Algorithm. In this section, we will derive the expected approximation ratio of the GreedyML algorithm. We will then describe the three submodular functions we experiment with and derive their computation and communication complexities.

4.1. Expected Approximation Ratio. This subsection proves the expected approximation ratio of our GreedyML algorithm in Theorem 4.4. To do so, we need three Lemmas. The first Lemma characterizes elements that do not change the solution computed by the GreedyML algorithm. We need some preliminary notation. When we wish to indicate the data set that a node in the tree and its descendants work with, we add an argument to $\textsc{GreedyML}(\ell, id)$, and write $\textsc{GreedyML}(V_{\ell,id}, \ell, id)$. When we perform a union operation on this data set with some set B, and execute the GreedyML algorithm on the union, i.e., $\textsc{GreedyML}(V_{\ell,id} \cup B, \ell, id)$, then elements in B are assigned randomly to the leaves of the subtree rooted at node (ℓ, id) and the algorithm is run with the updated data sets. Lemma 4.1 compares executions of the algorithm when this union operation is performed for a special set B. It states that if adding an individual element of a set, B, to the input of the GreedyML does not change the solution set then adding the whole set, B to the input will also have no effect on the solution. Here we consider executions that use the same random tape, number of machines, and branching factor. We use the same random tape to couple

Algorithm 3.1 Our Randomized Multi-level GreedyML Algorithm

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

procedure GreedyML$'$(P: Partial Data-set, ℓ: levels; b: branching factor, id: machine ID)
  $S = $ Greedy(P)
  $S_{prev} = S$
  for $i = 1 \dots \ell$ **do**
    if $id \neq parent(id, i)$ **then**
      Send S_{prev} to $parent(id, i)$ ▷ $parent(id,i) = b^i \cdot \lfloor id/b^i \rfloor$
      break
    end if
    $D = S_{prev}$ ▷ Prepare D for current iteration
    for $j = 1 \dots b-1$ **do**
      Receive D_j from $child(id, i, j)$ ▷ $child(id,i,j) = id + j \cdot b^{i-1}$
      $D = D \cup D_j$
    end for
    Run Greedy(D) to obtain S
    $S_{prev} = \arg\max\{f(S), f(S_{prev})\}$
  end for
  return S_{prev}
end procedure

the executions. Therefore, the result of the Lemma is not over the expectation of the random tape.

Lemma 4.1. *Let $T(m, L, b)$ be an accumulation tree. Consider a universal set W, and a random tape r_W that maps elements of W to the leaves of T. Let $V \subseteq W$ denote the set of elements accessible to a node (ℓ, id), and consider adding elements of $B \subseteq W$ to this node. If we have* GreedyML $(V \cup \{e\}, \ell, id) =$ GreedyML(V, ℓ, id), *for each element $e \in B$, then* GreedyML$(V \cup B, \ell, id) =$ GreedyML(V, ℓ, id).

Note: Function calls in this analysis use the same random tape for assigning elements; hence elements are assigned uniformly at random to the machines, but they use the same random assignment in all runs involving V; $V \cup \{e\}, \forall e \in B$; and $V \cup B$.

Proof. If possible, let GreedyML$(V \cup B, \ell, id) \neq$ GreedyML(V, ℓ, id). Let e be the first element of B to be selected by the Greedy algorithm at the final level. Let i be the level in which e was in the input in some machine but not selected in a solution for the next level in GreedyML$(V \cup \{e\}, \ell, id)$. Since e is the first element of B that was selected by the Greedy algorithm, the elements chosen before it at level i in GreedyML$(V \cup B, \ell, id)$ are the same ones chosen before it at level i in GreedyML$(V \cup \{e\}, \ell, id)$. Since it was not selected in GreedyML$(V \cup \{e\}, \ell, id)$ it will not be selected in GreedyML$(V \cup B, \ell, id)$. This is a contradiction since e must

be selected at every level to be present in the final solution. ∎

Now we turn to the two Lemmas that provide bounds on the quality of the computed solutions in terms of the optimal solution at an internal node in the accumulation tree.

Lemma 4.2 provides a lower bound on the expected function value of the solution of the GreedyML algorithm from a child of the internal node. Lemma 4.3 provides a lower bound on the expected function value of the solution set from the Greedy algorithm executed at each internal node on the union of the partial solutions from its children. These Lemmas depend on the probability distribution defined below.

Let $p_{\ell,id}\colon V_{\ell,id} \to [0,1]$ be a probability distribution over the elements in $V_{\ell,id}$, which we shall define below. Here $A \sim V_{\ell,id}(1/b)$ denotes a random subset of $V_{\ell,id}$ such that each element is independently present in A with probability $1/b$. This probability corresponds to the distribution from the random tape because each element is present with the same likelihood from any child of the node. Let $OPT_{\ell,id}$ be an optimal solution of the constrained submodular maximization problem when the input data is $V_{\ell,id}$.

The probability $p_{\ell,id}$ is defined as follows:

$$p_{\ell,id}(e) = \begin{cases} \Pr\limits_{A\sim V_{\ell,id}(1/b)} \left[e \in \textsc{GreedyML}(A \cup \{e\}, \ell-1, id)\right], & \text{if } e \in OPT_{\ell,id}; \\ 0, & \text{otherwise.} \end{cases}$$

For any internal node (ℓ, id), the distribution $p_{\ell,id}$ defines the probability that each element of $OPT_{\ell,id}$

is in the solution of the GreedyML algorithm of a child when it is accessible to the child node.

Next, we state and prove Lemma 4.2 that relates the expected solution of the GreedyML algorithm at a child node with the optimal solution at the node when the approximation ratio of the GreedyML algorithm at the child is β.

Lemma 4.2. *Let $c = (\ell-1, id_c)$ be a child of an internal node $n = (\ell, id)$ of the accumulation tree. Let S_c be the solution computed from child c, and $V_c \subset V_n$ denote the elements considered in forming S_c. If $\mathbb{E}_{V_c}[f(S_c)] \geqslant \beta \cdot f(OPT_{\ell-1,id_c})$, then*

$$\mathop{\mathbb{E}}_{V_n}[f(S_c)] \geqslant \beta \cdot \widehat{f}(1_{OPT_{\ell,id}} - p_{\ell,id}).$$

Proof. We first construct a subset of $OPT_{\ell,id}$ 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.,

$$O_c = \{e \in OPT_{\ell,id} : e \notin \textsc{GreedyML}(V_c \cup \{e\}, \ell', id)\}.$$

To clarify further, O_c is a randomized set dependent on the tape $r_{V_{\ell,id}}$. Since the distribution of V_c is the same as $V_{\ell,id}(1/b)$ for each element e in $OPT_{\ell,id}$,

$$\Pr[e \in O_c] = 1 - \Pr[e \notin O_c] = 1 - p_{\ell,id}(e). \tag{4.1}$$

From Lemma 4.1, we know that $\textsc{GreedyML}(V_c \cup O_c, \ell-1, id_c) = \textsc{GreedyML}(V_c, \ell-1, id_c)$. Since the rejected set $O_c \subseteq OPT_{\ell,id}$ and the constraints are hereditary, $O_c \in \mathcal{C}$ (i.e O_c is a feasible solution of child node c).

$$f(OPT_{\ell-1,id_c}) \geqslant f(O_c). \tag{4.2}$$

Then from the condition of Lemma 4.2, we have

$$\begin{aligned}
\mathbb{E}_{V_n}[\mathbb{E}_{V_c} f(S_c)] &\geqslant \beta \cdot \mathbb{E}_{V_n}[f(OPT_{\ell-1,id_c})] \\
\mathbb{E}_{V_n}[f(S_c)] &\geqslant \beta \cdot \mathbb{E}_{V_n}[f(OPT_{\ell-1,id_c})] && [V_c \subset V_n] \\
&\geqslant \beta \cdot \widehat{f}(\mathbb{E}_{V_n}[1_{O_c}]) && [\text{Eqn.}(4.2)] \\
&\geqslant \beta \cdot \mathbb{E}_{V_n}[f(O_c)] && [\text{Lovász } (2), 2.2] \\
&\geqslant \beta \cdot f^-(\mathbb{E}_{V_n}[1_{O_c}]) = \beta \cdot \widehat{f}(1_{OPT_{\ell,id}} - p_{\ell,id}) && [\text{Eqn.}(4.1)]. \quad \square
\end{aligned}$$

Now we show how the solution of the GREEDY algorithm that runs at each internal node compares with the optimal solution at the internal node.

LEMMA 4.3. *For an internal node $n = (\ell, id)$, let D be the union of all the solutions computed by the children of node n in the accumulation tree. Let $S =$ GREEDY(D) be the solution from the Greedy algorithm on the set D. If* GREEDY *is an α-approximate algorithm, then*

$$\mathbb{E}_{V_n}[f(S)] \geqslant \alpha \cdot f^-(p_{\ell,id}).$$

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

$$\Pr[e \in S_c | e \in V_c] = \Pr[e \in \text{GREEDYML}(V_c, \ell - 1, id) | e \in V_c].$$

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

$$\Pr[e \in S_c | e \in V_c] = \Pr_{B \sim V_{\ell,id}(1/b)}[e \in \text{GREEDYML}(B \cup \{e\}, \ell - 1, id)] = p_{\ell,id}(e).$$

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

$$\Pr(D \cap OPT_{\ell,id}) = p_{\ell,id}. \tag{4.3}$$

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

$$\begin{aligned}
f(S) &\geqslant \alpha \cdot f(D \cap OPT_{\ell,id}) \\
\mathbb{E}_{V_n}[f(S)] &\geqslant \mathbb{E}_{V_n}[\alpha \cdot f(D \cap OPT_{\ell,id})] \\
&\geqslant \alpha \cdot \mathbb{E}_{V_n}[f(D \cap OPT_{\ell,id})] \\
&\geqslant \alpha \cdot \widehat{f}(\mathbb{E}_{V_n}[1_{D \cap OPT_{\ell,id}}]) && [\text{Lovász Ext. } (2), 2.2] \\
&= \alpha \cdot \widehat{f}(p_{\ell,id}). && [\text{Eqn.4.3}]. \quad \square
\end{aligned} \tag{4.4}$$

THEOREM 4.4. *Let $T(m, L, b)$ be an accumulation tree. For a universal set W and random tape r_W that maps elements of W to the leaves of the tree T, let $V_{\ell,id} \subseteq W$ denote the subset of W accessible to a node (ℓ, id). Let $OPT_{\ell,id}$ be an optimal solution computed from the subset $V_{\ell,id}$ for the submodular function f with constraints $\mathcal{C}$. If* GREEDY *is an α-approximate algorithm, then*

$$\mathbb{E}_{V_{\ell,id}}[f(\text{GREEDYML}(V_{\ell,id}, \ell, id))] \geqslant \frac{\alpha}{(\ell+1)} f(OPT_{\ell,id}).$$

Proof. We prove this theorem by induction on the level ℓ.

Base case, $\ell = 0$: Here, there is no accumulation step, and we obtain the solution from a single node. Thus we run the GREEDY algorithm on $V_{\ell,id}$. The result follows since the GREEDY algorithm has the approximation ratio α.

Inductive case, $\ell = \ell' + 1$: We first obtain a relation for the quality of the solutions at level ℓ' compared to the quality of an optimal solution.

For each child c, let S_c be a solution computed by the GREEDYML algorithm from the data $V_c \subset V_{\ell,id}$;

From the **induction hypothesis** applied to child $c = (\ell', id)$, the approximation ratio obtained as a result of the computation GREEDYML(V_c, ℓ', id) is $\alpha/(\ell'+1) = \alpha/\ell$. This implies that $\mathbb{E}_{V_{l',id}}[f(S_c)] \geqslant \frac{\alpha}{\ell} \cdot f^{-}(1_{OPT_{\ell',id}})$. Therefore we can apply Lemma 4.2 to get

$$\mathbb{E}_{V_{\ell,id}}[f(S_c)] \geqslant \frac{\alpha}{\ell} \cdot \widehat{f}(1_{OPT_{\ell,id}} - p_{\ell,id}). \tag{4.5}$$

After obtaining the solutions from the children, we get the solution S computed by the GREEDY algorithm on the union of these solution sets. From Lemma 4.3, we have

$$\mathbb{E}_{V_{\ell,id}}[f(S)] \geqslant \alpha \cdot \widehat{f}(p_{\ell,id}). \tag{4.6}$$

Now we obtain the relation between the solution at level $\ell' + 1$ and the optimal solution. Let the solution set at level $\ell' + 1$ be T. We have $T = \arg\max\{f(S), f(S_c)\}$. Then, we can use the lower bounds calculated earlier in Eqn. 4.5 and Eqn. 4.6 to find lower bounds for T.

$$\mathbb{E}_{V_{\ell,id}}[f(T)] \geqslant \alpha \cdot \widehat{f}(p_{\ell,id}) \text{ and } \mathbb{E}_{V_{\ell,id}}[f(T)] \geqslant \frac{\alpha}{\ell} \cdot \widehat{f}(1_{OPT_{\ell,id}} - p_{\ell,id}).$$

By multiplying the second inequality by ℓ and then adding it to the first, we get

$$\begin{aligned}(\ell+1)\mathbb{E}_{V_{\ell,id}}[f(T)] &\geqslant \alpha \cdot (\widehat{f}(1_{OPT_{\ell,id}} - p_{\ell,id}) + \widehat{f}(p_{\ell,id})) \\ &= \alpha \cdot \widehat{f}(1_{OPT_{\ell,id}}). \qquad \text{[Lovász Ext. (2), 2.2]}\end{aligned}$$

Dividing by by $\ell+1$, and substituting from Lovász Ext. (1), 2.2 we conclude that the algorithm is $\alpha/(\ell+1)$-approximate. ∎

4.2. Submodular Functions and Complexity. Here, we describe three submodular functions that we consider in our experiments and then discuss their computational and communication complexities.

Our algorithm can handle any hereditary constraint, but in our experiments, we consider only cardinality constraints to keep the computations simple. (More general constraints involve additional computations to check if an element can be added to the current solution set and satisfy the constraints.) Cardinality constraints are widely used in various applications such as sensor placement [16], text, image, and document summarization [18, 19], and information gathering [15]. The problem of maximizing a submodular function under cardinality constraints can be expressed as follows.

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

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. The first problem we consider is the *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, where we want to select a set of exemplars (cluster centers) that are representatives of a large dataset. Given a collection of elements in a ground set V, and a dissimilarity $d(u, v)$, 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., $L(S) = \dfrac{1}{|V|} \sum_{u \in V} \min_{v \in S} d(u, v)$. Following [24], 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 representing the exemplars that maximize $f(S)$.

Next, we will analyze the computational and communication complexity of our GreedyML algorithm using the bulk synchronous parallel (BSP) model of parallel computation [29]. For the analysis, we will 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 (shown in Algorithm 2.1 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 ($\leqslant \delta$), 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 1.

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^2m	$k^2\lceil m^{1/L}\rceil$
	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 1: 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.

5. Experimental setup. We conduct experiments to evaluate our algorithms using different accumulation tree structures and compare them with Greedy and RandGreedi to assess the quality, runtime, and memory footprints of these algorithms. All the algorithms are executed on the Bell community cluster [22] of Purdue University 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 cores on a node are organized hierarchically: four cores form a core complex, two core complexes form a core complex die, eight core complex dies form a socket, and two sockets constitute a node. Unfortunately, there are only 16 memory controllers for the 128 cores, and hence in this NUMA architecture, memory contention is an issue on cores within a node. To simulate a completely 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 run time 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 in each machine, and for the quality, we show the objective function value of the corresponding submodular function. 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 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 nodes of the accumulation tree with $mid = 0$ since this node participates in the function calls from all levels of the tree.

Function	Dataset	$n = \|V\|$	$\sum_u \delta(u)$	avg. $\delta(u)$
k-dominating set	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

Table 2: Properties of Datasets used in the experiments. $\delta(u)$ is 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.

Datasets. In this paper, we limit our experiments to cardinality constraints using three different submodular functions described in detail in Section 4.2.

Our benchmark dataset is shown in Table 2. They are grouped based on the objective function and are sorted by the size of the dataset in each group. For the k-dominating set, our testbed consists of the Friendster social network graph [30] and a collection of road networks from **DIMACS10** dataset. We chose these graphs since they have relatively small average vertex degrees, leading to large vertex-dominating sets. For the k-cover objective, our datasets come from the **Frequent Itemset Mining Dataset Repository** [10] which contains popular benchmarks for set covers. We choose *webdocs*[21], *retail* [3], and *kosarak*. For the k-medoid problem, we use the **Tiny ImageNet** dataset [8], which contains $100,000$ images with 200 different classes and 500 images from each class. Each image is 64×64 pixels in size.

MPI Implementation. Our codes are implemented using `C++11`, and compiled with `g++9.3.0`, using the `O3` optimization flag. Our implementation of the GREEDY algorithm uses the Lazy Greedy [23] 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 Open MPI implementation for the 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 3.1). 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.

6. Experimental Results. The experiments are executed with different accumulation trees that vary in the number of machines (m) and the number of levels (L) and 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 as stated in Section 5. Whenever memory constraints allow, we compare our results with the sequential GREEDY algorithm that achieves $(1-1/e)$ approximation guarantee.

Recall that our GREEDYML algorithm generalizes the RANDGREEDI algorithm

by allowing multiple levels in the accumulation tree, thus removing the bottleneck of a single aggregation. In the following, we verify this through a series of experiments.

In Section 6.1, we assess the performance of our algorithm using different accumulation tree structures. We fix the number of machines and construct the best parameters of the accumulation tree for our dataset. Additionally, the experiment also demonstrates that the number of function calls in the critical path is a good estimate of the parallel runtime. In Section 6.2, we show the memory benefit of our GreedyML w.r.t RandGreedi with two experiments. In Section 6.2.1, we impose a limit of 100 MB for each node and vary k, the selection size. This also simulates how the new algorithm can find applications in the *edge-computing* context. In Section 6.2.2, we fix the k value 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 this problem with negligible drop in accuracy. For these experiments, we will choose the computational tree with the lowest depth that can be used with the memory limit and k values.

In Section 6.3, we perform a scaling experiment by varying the number of machines and using the tallest tree by setting a branching factor of two for the accumulation tree. We specifically 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. We also show how our algorithm alleviates this bottleneck. Finally, in Section 6.4, we perform experiments for the k-medoid objective function 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.

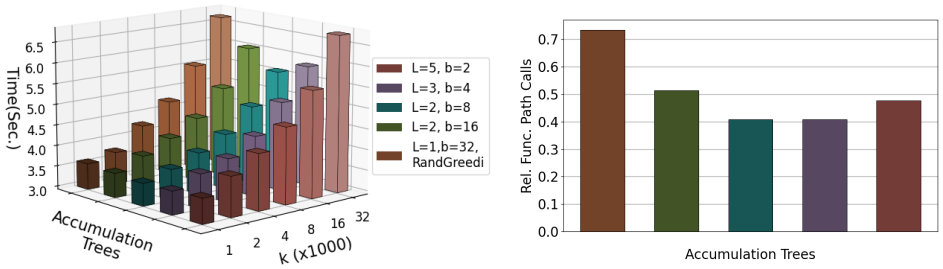

Fig. 4: Geometric means of results from GreedyML for k-dominating set on different road datasets and k-cover on different set cover benchmark datasets on 32 machines. The first subfigure shows the execution times for different k values and accumulation trees. The second subfigure shows the Geometric mean values of the number of function calls in the critical path relative to the Greedy algorithm for $k = 32,000$.

6.1. Accumulation tree parameter selection. Our first experiment explores the effect of choosing different branching factors and different accumulation levels in the accumulation tree for a fixed number of machines. In this experiment, we vary the selection set sizes k for each of these accumulation trees. We obtain results for the six datasets for k-dominating set and k-coverage detailed in Table 2. In Figure 4, we provide summary results on the relative number of function evaluations in the critical

path relative to the GREEDY algorithm and the running times by taking a geometric mean over all the datasets.

The first subfigure shows 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, there is less variation in the execution time, since a significant amount of work is performed at the leaves. As k increases, we can observe that the GREEDYML algorithm runs faster relative to the RANDGREEDI algorithm ($L = 1, b = 32$). Note that, although we present in Figure 4 the geometric mean results over all the six datasets, the runtime and the function values for the individual datasets follow the same trend. The belgium_osm dataset has the largest reduction in run time with a reduction of around 22% and the smallest reduction in runtime is in the kosarak dataset with a reduction of 1% across all k values.

The second subfigure chooses $k = 32,000$ and plots the number of function calls in the critical path of the accumulation tree relative to the GREEDY algorithm for different (L, b) pairs. We observe that the relative number function calls for RANDGREEDI is around 70% of GREEDY, whereas the GREEDYML (with $L = 2$ and $b = 8$) cuts down the time by 15 percent. From Table 1, we can see that the function calls at a leaf node is $O(nk/m)$ whereas the function calls at an accumulation node is $O(mk^2)$ for the RANDGREEDI algorithm. The accumulation node dominates the computation since it has a quadratic dependence on k, becoming a bottleneck for large k values.

This plot shows that the number of calls is a good indicator of the run time of the algorithm and that the cost of function evaluations dominates the time taken by the algorithm. On the other hand, the communication costs are small but, for the GREEDYML, they do grow with the number of levels when k is very large.

We additionally note (not in figure), that the objective function values obtained by the GREEDYML algorithm are not sensitive to the choice of the number of levels and the 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 obtains objective function values that are about 20% higher than both the RANDGREEDI and GREEDYML algorithms.

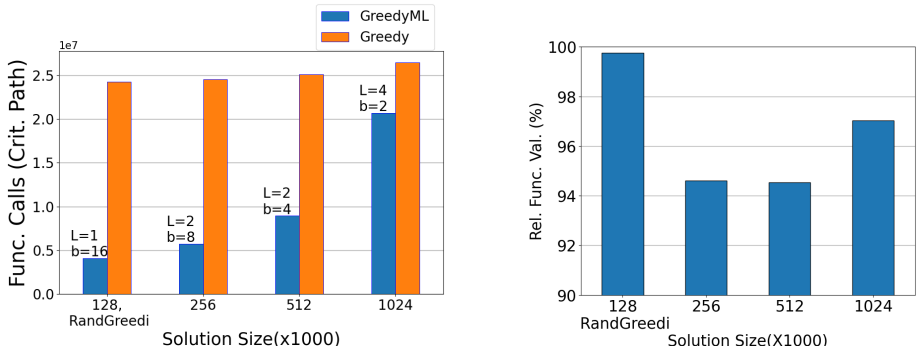

Fig. 5: Results from GREEDYML for the k-dominating set problem on the road_usa dataset on 16 nodes with varying k. The tuple(L,b) shows the number of levels and branching factors chosen for specific k values. The function values are relative to the GREEDY algorithm. Note that the leftmost bars in both plots represent the RANDGREEDI results.

6.2. Experiments with memory limit.

6.2.1. Varying k. For this experiment, we use 16 machines with a limit on available memory of 100 MB per machine and vary k from $128,000$ to $1,024,000$. We consider the k-dominating set problem on the road_usa [1] dataset and large k values are chosen since the graph has an even larger maximum dominating set. Note that the k-values other than $128,000$ cause the RandGreedi algorithm to run out of memory in accumulating all the solutions in the root node. We note that the small memory limit in this experiment can also be motivated from *edge computing* context.

The left plot in Figure 5 shows the number of function calls with varying values of k for the Greedy and GreedyML algorithms. 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 $k = 128,000$ instance can be solved using the RandGreedi algorithm, which is shown in the leftmost blue bar of the plot.

As we increase k, we were able to 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.

For each k value, we also executed the Greedy algorithm shown in the orange 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 calls for accumulation trees with smaller b values is larger than RandGreedi, we can 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 our number of function calls in the critical path increases. That suggests 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 5 shows the relative objective function value, i.e., the relative number of vertices covered by the dominating set compared to the Greedy algorithm, with varying k. The figure shows that the RandGreedi and GreedyML algorithms attain quality at most 6% lesser than the serial Greedy algorithm. Similar trends can be observed for other datasets in the summary of results shown in Figure 4.

6.2.2. Varying Memory Limits. This experiment demonstrates the capability of the GreedyML algorithm to solve a problem with a fixed k value on parallel machines when the memory is insufficient for the RandGreedi and Greedy algorithms. Unlike the experiment in Section 6.2.1, where we selected the accumulation trees based on the k value for the problem, here, we fix k and choose accumulation trees based on the size of memory available on the machines. We consider the k-dominating set problem on graphs, and first report results on the Friendster dataset [30]. We set the cardinality constraint k so that the k-dominating set requires 512 MB, roughly a factor of 64 smaller than the original graph. The RandGreedi algorithm 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 show results from these three machine configurations in Table 3. We report the number of function calls on a critical path and the objective function values normalized

Dataset	Alg.	Mem. Limit	m	b	L	Rel. Func. Val.(%)	Time (sec.)
Friendster	RG	4GB	8	8	1	96.294	69.81
	GML	2GB	16	4	2	96.232	82.92
	GML	1GB	32	2	5	96.175	112.17
road_usa	RG		8	8	1	99.034	1.25
	GML		16	4	2	99.005	1.63
	GML		32	2	5	99.027	3.56
webdocs	RG		8	8	1	79.948	4.50
	GML		16	4	2	78.723	4.72
	GML		32	2	5	79.743	8.59

Table 3: Results from GreedyML (GML) 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 organizations. We report the function values relative to the Greedy algorithm and the execution time in seconds. Note that the 4GB entries run with $L = 1$ and correspond 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.

by those obtained from the serial Greedy algorithm. Our results show that objective function values computed by the GreedyML algorithm (the 2 and 1 GB results) are insensitive to the number of levels in the tree. Similar trends are observed for the webdocs [21] and road_usa [1] datasets when we used the same number of machines and accumulation trees. As we increase the number of machines and levels in the accumulation tree, the execution times (in seconds) increase for this problem due to the communication and synchronization costs involved. However, the larger numbers of machines enable 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.

6.3. Strong Scaling. Next, we show how the GreedyML algorithm serves as a solution to the scaling bottlenecks that arise in the RandGreedi algorithm. For the scaling experiment, we consider 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 this with the RandGreedi algorithm starting from 8 machines to 128 machines with $k = 50$. We compare the total execution time, communication time, and computation time for the GreedyML and the RandGreedi algorithms.

In Figure 6, we plot the total execution time by stacking communication and computation time for the two algorithms. We observe that the communication cost does not scale for the RandGreedi algorithm. From Table 1, we can see that the time spent by the central node collecting the solutions is $O(km)$ and, therefore, increases linearly with the number of machines. In contrast, for GreedyML algorithm (with

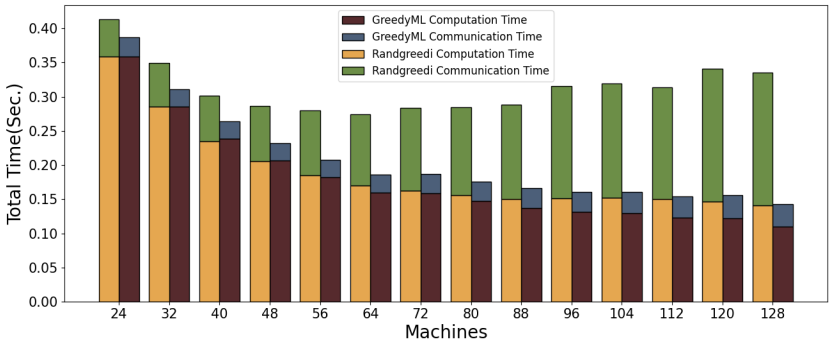

Fig. 6: Strong scaling results of the RandGreedi and GreedyML algorithms for $k = 50$ on Friendster dataset for k-dominating set problem. We set $b = 2$ for the GreedyML algorithm.

L	b	Local Obj.		Added Images	
		Rel. Func. Val. (%)	Speedup	Rel. Func. Val. (%)	Speedup
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

Table 4: Results from GreedyML for the k-medoid function on the **Tiny ImageNet** data set using different accumulation trees. The table shows the relative function values and speedup compared to the RandGreedi algorithm using two different local objective values computation schemes executed on 32 nodes. For both, higher values are better. Here L and b are the number of levels and branching factor, respectively.

a constant branching factor, $b = 2, L = \log_2 m$), the communication cost $O(k \log m)$ which grows logarithmically in the number of machines. The total communication times of the GreedyML algorithm across different machines are consistently around 0.25 seconds, whereas the RandGreedi increases from 0.05 second to 2 seconds linearly. We observe that computation times for both RandGreedi and GreedyML changes 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 see an almost identical quality in the solutions where the GreedyML solution has a quality reducing by less than 1% from the solution of the RandGreedi algorithm.

6.4. The k-Medoid Problem. In our final experiment, we consider the k-medoid function that solves the exemplar-based clustering problem. Our dataset consists of the **Tiny ImageNet** dataset [8], which contains 100,000 images with 200 different classes and 500 images from each class. Each of the images is 64×64 pixels. We flatten each image into a vector of 12,288 length. We then subtracted the mean value and normalized the vector. We compute the dissimilarity between two images as the Euclidean distance between the normalized vector representations. Here, the

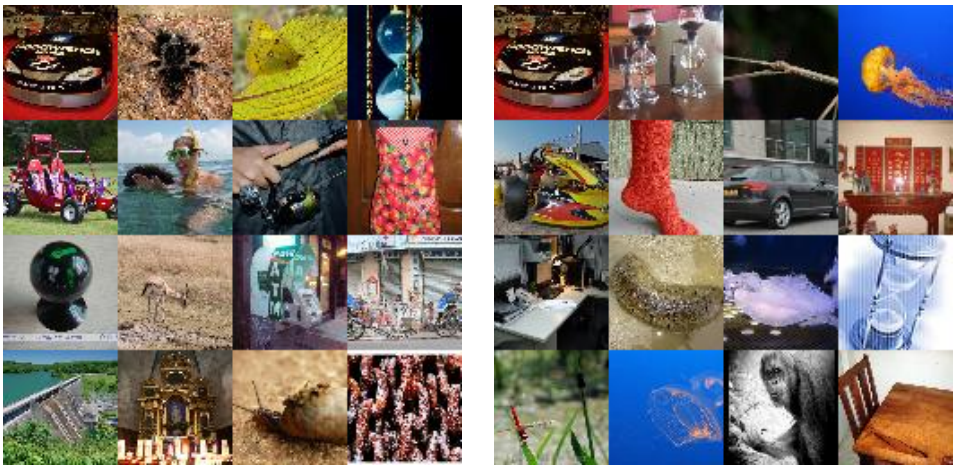

Fig. 7: Results from GreedyML for the k-medoid problem on the Tiny ImageNet dataset on 32 nodes with $k = 200$ with no images added at each accumulation step. The subfigure on the left shows the first 16 image results for one of the runs for the GreedyML algorithm with branching factor $b = 2$, and the subfigure on the right shows the top 16 image results for one of the runs for the RandGreedi algorithm.

auxiliary image e_0 is a pixel vector of all zeros. Note that, unlike the other two functions, the k-medoid function, requires access to the full dataset for computing the functional value. Since the dataset is distributed, this poses an issue in the experiment. To overcome this, following [24, 2], we calculate the objective function value using only the images available *locally* on each machine. This means the ground set for each machine is just the images present in that machine. This is motivated by an analysis from Mirzasoleiman et al. (Theorem 10, [24]) showing that computing $f(S)$ with the ground set as some subset $D \subseteq V$ chosen uniformly at random provides a high-probability additive approximation to the function value $f(S)$ evaluated with ground set V. Additionally, they 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 fix the number of machines ($m = 32$) and vary the accumulation trees by choosing different L and b. We set the solution size k to 200 images. 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 compute-intensive, where computation cost increases quadratically with the number of images (Table 1). 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 also in the observed speedup.

Finally, in Fig. 7, we show 16 out of the 200 images determined to be cluster centers by the GreedyML and RandGreedi algorithms. We can draw the conclusion that the submodular k-medoid function is able to generate a diverse set of exemplar images for this clustering problem.

7. Conclusion and Future work. We have developed a new distributed algorithm that generalizes the existing state-of-the-art algorithm for monotone submodular maximization subject to hereditary constraints. We prove that the new algorithm is $\alpha/(L+1)$ approximate and showed its quality doesn't degrade for the k-cover, k-dominating set, and k-medoid problems. We showed how this new algorithm reduces the inherent serial computation and communication bottlenecks of the RandGreedi algorithm. We also reduce the memory required to solve the problem enabling submodular maximization to be solved in an edge computation context and with larger k values. Finally, We showed a significant speedup in solving the popular exemplar-based clustering problem.

As part of our future work, we plan to run experiments for other hereditary constraints, such as matroid and p-system constraints. We will also explore how this generalization technique can be applied to other classes of NP-Hard problems such as non-monotone submodular functions and weakly-submodular functions.

REFERENCES

[1] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, *Graph partitioning and graph clustering*, in Contemporary Mathematics, vol. 588, American Mathematical Society, 2012. 10th DIMACS Implementation Challenge Workshop.

[2] R. d. Barbosa, A. Ene, H. L. Nguyen, and J. Ward, *The power of randomization: Distributed submodular maximization on massive datasets*, in Proceedings of the 32nd International Conference on Machine Learning, JMLR.org, 2015, pp. 1236–1244.

[3] T. Brijs, G. Swinnen, K. Vanhoof, and G. Wets, *Using association rules for product assortment decisions: A case study*, in Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 1999, pp. 254–260.

[4] C. Chekuri and K. Quanrud, *Submodular function maximization in parallel via the multilinear relaxation*, in Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, 2019, p. 303–322.

[5] L. Chen, G. Zhang, and E. Zhou, *Fast greedy MAP inference for determinantal point process to improve recommendation diversity*, in Advances in Neural Information Processing Systems, 2018, pp. 5627–5638.

[6] M. Coutino, S. P. Chepuri, and G. Leus, *Submodular sparse sensing for Gaussian detection with correlated observations*, IEEE Transactions on Signal Processing, 66 (2018), p. 4025–4039.

[7] 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, 2017, p. 4044–4054.

[8] A. K. Fei-Fei Li, *Tiny imagenet challenge*. http://cs231n.stanford.edu/tiny-imagenet-200.zip, 2017. [Online; last accessed 13-Mar-2024].

[9] S. M. Ferdous, A. Pothen, A. Khan, A. Panyala, and M. Halappanavar, *A parallel approximation algorithm for maximizing submodular b-matching*, in SIAM Conference on Applied and Computational Discrete Algorithms (ACDA), 2021, pp. 45–56.

[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 (2011), p. 427–486.

[12] L. Kaufman and P. J. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis.*, John Wiley, 1990.

[13] D. Kempe, J. Kleinberg, and É. 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, 2003, pp. 137–146.

[14] A. Krause and D. Golovin, *Submodular function maximization*, in Tractability: Practical Approaches to Hard Problems, Cambridge University Press, 2014, pp. 71 – 104.

[15] A. Krause and C. Guestrin, *Near-optimal observation selection using submodular functions*, in Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, vol. 7, 2007, pp. 1650–1654.

[16] A. Krause, J. Leskovec, C. Guestrin, J. VanBriesen, and C. Faloutsos, *Efficient sensor placement optimization for securing large water distribution networks*, Journal of Water Resources Planning and Management, 134 (2008), pp. 516–526.

[17] R. Kumar, B. Moseley, S. Vassilvitskii, and A. Vattani, *Fast greedy algorithms in mapreduce and streaming*, ACM Trans. Parallel Comput., 2 (2015), pp. 14:1–14:22.

[18] H. Lin and J. 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, 2010, pp. 912–920.

[19] H. Lin and J. 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, 2011, pp. 510–520.

[20] L. Lovász, *Submodular functions and convexity*, in Mathematical Programming The State of the Art: Bonn 1982, A. Bachem, B. Korte, and M. Grötschel, eds., Springer Berlin Heidelberg, 1983, pp. 235–257, https://doi.org/10.1007/978-3-642-68874-4_10.

[21] C. Lucchese, S. Orlando, R. Perego, and F. Silvestri, *WebDocs: A real-life huge transactional dataset*, in Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI 04), 2004.

[22] G. McCartney, T. Hacker, and B. Yang, *Empowering Faculty: A Campus Cyberinfrastructure Strategy for Research Communities*, Educause Review, (2014), https://er.educause.edu/articles/2014/7/empowering-faculty-a-campus-cyberinfrastructure-strategy-for-research-communities.

[23] M. Minoux, *Accelerated greedy algorithms for maximizing submodular set functions*, in Optimization Techniques, J. Stoer, ed., Springer, 1978, pp. 234–243.

[24] B. Mirzasoleiman, A. Karbasi, R. Sarkar, and A. Krause, *Distributed submodular maximization: Identifying representative elements in massive data*, in Advances in Neural Information Processing Systems, vol. 26, 2013, pp. 2049–2057.

[25] A. Mokhtari, H. Hassani, and A. Karbasi, *Decentralized submodular maximization: Bridging discrete and continuous settings*, in Proceedings of the 35th International Conference on Machine Learning, 2018, pp. 3616–3625.

[26] A. Robey, A. Adibi, B. Schlotfeldt, H. Hassani, and G. J. Pappas, *Optimal algorithms for submodular maximization with distributed constraints*, in Proceedings of the 3rd Conference on Learning for Dynamics and Control, 2021, pp. 150–162.

[27] K. Thekumparampil, A. Thangaraj, and R. Vaze, *Combinatorial resource allocation using submodularity of waterfilling*, IEEE Transactions on Wireless Communications, 15 (2016), p. 206–216.

[28] E. Tohidi, R. Amiri, M. Coutino, D. Gesbert, G. Leus, and A. Karbasi, *Submodularity in action: From machine learning to signal processing applications*, IEEE Signal Processing Magazine, 37 (2020), pp. 120–133, https://doi.org/10.1109/MSP.2020.3003836.

[29] L. G. Valiant, *A bridging model for parallel computation*, Commun. ACM, 33 (1990), p. 103–111, https://doi.org/10.1145/79173.79181.

[30] J. Yang and J. Leskovec, *Defining and evaluating network communities based on ground-truth*, in Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics, 2012, pp. 1–8.