A Parallel Algorithm for Maximizing Submodular Functions

Anonymous author
Anonymous affiliation

Abstract

We describe a parallel approximation algorithm for maximizing monotone submodular functions subject to hereditary constraints on distributed memory multiprocessors, building on an earlier distributed approximation algorithm with limited parallelism and higher memory requirements. 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 GreedyL algorithm on three classes of problems, and report results from massive data sets with millions of elements. The results show that the GreedyL algorithm can solve problems where the Greedy and RandGreedi algorithms fail due to memory constraints. For certain computationally intensive problems, the GreedyL algorithm can be faster than the RandGreedi algorithm. The observed approximation quality of the solutions computed by the GreedyL algorithm closely matches those obtained by the RandGreedi algorithm on these problems.

2012 ACM Subject Classification Computing methodologies; Theory of computation → Theory and algorithms for application domains

Keywords and phrases submodular optimization, approximation algorithm, parallel algorithm, data summarization

Digital Object Identifier 10.4230/LIPIcs.ESA.2023.1
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 older 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 marginal gain at each step is \((1 - 1/e) \approx 0.63\)-approximate for cardinality constraints and \(1/2\) for matroid constraints; here \(e\) is Euler’s number.

We build on the RandGreedi framework [2], a state-of-the-art randomized distributed algorithm for 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 step could exceed the memory available on a processor when the memory is small relative to the size of the data, or there is a need for solutions of larger sizes. Additionally, this merging step serializes the computation and is a bottleneck when scaled to more machines.

The new GreedyL 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 are the leaves of an accumulation tree, and then merge partial solutions at multiple levels in the tree. We prove that the GreedyL algorithm has a worst-case approximation guarantee of \(1/L\) of the serial greedy algorithm, where \(L\) is the total number of levels in the accumulation tree. Using the BSP model, we analyze the time complexity of both computation and communication steps in the GreedyL and RandGreedi algorithms, and show that the former has smaller 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 the \(k\)-medoid problem or exemplar-based clustering. 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 solution size \(k\). 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 GreedyL algorithm, and the observed approximation quality of the computed solutions closely matches those obtained by the RandGreedi algorithm on these problems.

2 Background

A set function \(f: 2^W \to \mathbb{R}^+\) defined on subsets of a ground set \(W\) is submodular if it satisfies the diminishing gains property.

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

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 maximizes a submodular objective function
We consider these solutions are then sent to a single machine, where they are accumulated. The Greedy algorithm has a worst-case approximation guarantee of $1/\Theta(\min(\sqrt{k}, m))$, where $m$ is the number of machines and $k$ is the selection size. It performs $O(nk(k + m))$ oracle calls to the objective function and has $O(mk)$ elements communicated to the single central machine.

---

**Algorithm 1** RandGreedi framework for maximizing constrained submodular function

1: procedure Randomized Greedi (V: Dataset, $m$: number of machines)
2:      $S \leftarrow \emptyset$
3:      Partition the elements in $V$ into $m$ subsets $P_0, \ldots, P_{m-1}$ uniformly at random
4:      for $i = 0 \ldots m - 1$ in parallel do
5:         Run Greedy($P_i$) on machine $i$ to compute the solution $S_i$
6:      $S \leftarrow S \cup S_i$
7:      end for
8:      Run Greedy($S$) to compute the solution $T$
9:      return $\arg \max \{ f(T), f(S_1), f(S_2), \ldots, f(S_{m-1}) \}$
10: end procedure

subject to certain constraints:

$$\max f(A) \text{ subject to } A \in \mathcal{C}, \text{ where } \mathcal{C} \subseteq 2^W$$

is the family of feasible solutions.

We consider hereditary constraints: i.e., for every set $A \in \mathcal{C}$, every subset of $A$ is also in $\mathcal{C}$.

We can view a submodular function $f$ 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 [8] $f^- : [0, 1]^n \to \mathbb{R}^+$ is a convex extension that extends this function over the entire hypercube.

$$f^-(x) = \mathbb{E}_{\theta \in [0,1]} [ f(\{ i : x_i \geq \theta \}) ] .$$

For any submodular function $f$, the Lovász extension $f^-$ satisfies the following properties:

1. $f^-(1_S) = f(S)$, for all $S \subseteq V$,
2. $f^-(x)$ is convex, and
3. $f^-(c \cdot x) \geq c \cdot f^-(x)$, for any $c \in [0, 1]$. We do not compute the Lovász extension in our algorithm, but use it to establish its approximation property.

An $\alpha$-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 $f(S) \geq \alpha \cdot f(S^*)$, where $S^*$ is an optimal solution.

### 2.1 The Greedi and RandGreedi Algorithms

The Greedi algorithm 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 Greedi algorithm chooses a feasible element $e$ 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 negative or all elements in the ground set have been considered. The pseudo-code for the algorithm is included in the Appendix as Algorithm 3).

The Greedi algorithm [11] partitions the data arbitrarily on available machines, and on each partition, it runs the Greedi algorithm in parallel to compute a local solution. These solutions are then sent to a single machine, where they are accumulated. The Greedi algorithm is then executed once again on the accumulated data to get a global solution. The final solution is the best solution among all the local and global solutions. The Greedi algorithm has a worst-case approximation guarantee of $1/\Theta(\min(\sqrt{k}, m))$, where $m$ is the number of machines and $k$ is the selection size. It performs $O(nk(k + m))$ oracle calls to the objective function and has $O(mk)$ elements communicated to the single central machine.
Although the Greedi algorithm performs well in practice [11], its approximation ratio depends inversely on the number of machines. 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 $\frac{1}{2}(1 - 1/e)$ for cardinality and $1/4$ for matroid constraints. We present the pseudocode of RandGreedi framework in Algorithm 1.

### 3 Description of Our Algorithm

We describe an 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. The tree nodes correspond to processors along with their subset of data, and the edges determine the accumulation pattern of the partial solutions. Figure 1 shows an example of an accumulation tree. The number of levels in a tree corresponds to the level value of the root, which is denoted by $L$. The branching factor of the tree indicates the maximum number of nodes that send data to its parent. We can 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. We can index each node in the tree by its level $\ell$ and machine identifier $id$ with the tuple $(\ell, id)$.

Observe that the $id$ of a node remains the same if it is involved in computations at multiple levels. For this paper, we prove the result for one branching factor across all levels. When the branching factor is $b$, but the number of processors active at some level is not a multiple of $b$, there will be fewer than $b$ children for the last node at the next level.

**Data Accessibility.** We use $P_{id}$ to denote the elements assigned to machine $id$. To clearly 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 node $(\ell, id)$ and consists of all the elements assigned to its descendants: $V_{\ell, id} = \bigcup_{i=0}^{b^\ell - 1} P_{id+i}$.

**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 $E_V$.

**Recurrence Relation.** Figure 2 shows a recurrence relation defined on a node in the accumulation tree that 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 $(\ell, id)$ tuples that do not correspond to nodes in the tree (at higher levels). We call our algorithm associated with the recurrence relation as the GreedyL algorithm.

We can compare it with the RandGreedi algorithm by looking at the recurrence relation at level one. Our 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 it does not affect the approximation ratio.
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 $(\ell, id)$. 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 $b$.

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

Figure 2 The recurrence relation for the multilevel Greedy$L$ which is defined for each node in the accumulation tree. We denote the random subset assigned to machine $id$ by $P_{\text{id}}$.

When we wish to indicate the data set that a node in the tree and its descendants work with, we add one more argument to Greedy$L(\ell, id)$, and write Greedy$L(V_{\ell, id}, \ell, id)$. When we perform a union operation on this data set with some set $B$, and execute the Greedy$L$ algorithm on the union, i.e., Greedy$L(V_{\ell, id} \cup B, \ell, id)$, then elements in $B$ are assigned randomly to the leaves of the subtree rooted at node $(\ell, id)$ and the algorithm is run with the updated data sets. Lemma 1 compares executions of the algorithm when this union operation happens for a special set $B$. We use the same random tape to couple the executions. Therefore, the lemma is not an expectation result.

Lemma 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 $(\ell, id)$, and consider adding elements of $B \subseteq W$ to this node. If we have Greedy$L(V \cup \{e\}, \ell, id) = \text{GreedyL}(V, \ell, id)$, for each element $e \in B$, then Greedy$L(V \cup B, \ell, id) = \text{GreedyL}(V, \ell, id)$.

The proof is similar to a result in Barbosa et al. [2], and we include it (for completeness) in the Appendix.

4 Analysis of Our Algorithm

This section proves the expected approximation ratio of our Greedy$L$ algorithm in Theorem 4. To do so, we need two Lemmas. Lemma 2 provides a lower bound on the expected function value of the solution of the Greedy$L$ algorithm at an internal node in the accumulation tree from a child of the node. Lemma 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. Both Lemmas provide bounds on the quality of the
A Parallel Algorithm for Maximizing Submodular Functions

computed solutions in terms of the optimal solution at the parent node in the accumulation tree. They depend on the probability distribution defined below.

Let \( p_{\ell,id} : V_{\ell,id} \rightarrow [0,1] \) 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:

\[
p_{\ell,id}(e) = \begin{cases} 
\Pr_{A \sim V_{\ell,id}(1/b)} [e \in \text{GreedyL}(A \cup \{e\}, \ell - 1, id)], & \text{if } e \in OPT_{\ell,id} \\
0, & \text{otherwise}.
\end{cases}
\]

For any internal node \((\ell, id)\), the distribution \( p_{\ell,id} \) defines the probability that each element of \( OPT_{\ell,id} \) is in the solution of the Greedy algorithm of a child when it is accessible to the child node.

Next, we state and prove Lemma 2 that relates the expected solution of the Greedy algorithm at a child node with the optimal solution at the parent node when the approximation ratio of the GreedyL algorithm at the child is \( \beta \).

**Lemma 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 set \( V_c \subset V_n \) denote the elements considered in forming \( S_c \). If \( E_{V_c}[f(S_c)] \geq \beta \cdot f(OPT_{\ell-1,id_c}) \), then

\[
E_{V_n}[f(S_c)] \geq \beta \cdot f(OPT_{\ell-1,id_c} - p_{\ell,id}).
\]

**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.,

\[ O_c = \{ e \in OPT_{\ell,id} : e \notin \text{GreedyL}(V_c \cup \{e\}, \ell', id) \}. \]

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

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

From Lemma 1, we know that \( \text{GreedyL}(V_c \cup O_c, \ell - 1, id_c) = \text{GreedyL}(V_c, \ell - 1, id_c) \).

Since the rejected set \( O_c \subseteq OPT_{\ell,id} \) and the constraints are hereditary, \( O_c \subseteq C \). Therefore,

\[
f(OPT_{\ell-1,id_c}) \geq f(O_c).
\]

Then from the condition of Lemma 2, we have

\[
E_{V_n}[f(S_c)] \geq \beta \cdot E_{V_n}[f(OPT_{\ell-1,id_c})]
\]

and

\[
E_{V_n}[f(S_c)] \geq \beta \cdot E_{V_n}[f(OPT_{\ell-1,id_c})] \quad \text{((} \because V_c \subset V_n \text{)).}
\]

Therefore, we have

\[
E_{V_n}[f(S_c)] \geq \beta \cdot f^{-1}(E_{V_n}[1_{O_c}]) \geq \beta \cdot E_{V_n}[f(O_c)] \quad \text{((} \because \text{Eqn. (2), Lovász (2))}
\]

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

**Lemma 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 \( S = \text{Greedy}(D) \) be the solution from the Greedy algorithm on the set \( D \). If \( \text{Greedy} \) is an \( \alpha \)-approximate algorithm, then

\[
E_{V_n}[f(S)] \geq \alpha \cdot f^{-1}(p_{\ell,id}).
\]
Proof. We first prove a preliminary result on set $D$. Consider an element $e \in D \cap \text{OPT}_{\ell, \text{id}}$ present in some solution $S_c$ from a child $c$. Then

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

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

$$\Pr[e \in S_c | e \in V_c] = \Pr[B \cup \text{V}_{\ell, \text{id}}(1/b)| e \in \text{GREEDY}(B \cup \{e\}, \ell - 1, \text{id})] = p_{\ell, \text{id}}(e).$$

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

$$\Pr(D \cap \text{OPT}_{\ell, \text{id}}) = p_{\ell, \text{id}}.$$  (3)

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

$$f(S) \geq \alpha \cdot f(D \cap \text{OPT}_{\ell, \text{id}}).$$

$$\mathbb{E}_{V_{\ell, \text{id}}} [f(S)] \geq \mathbb{E}_{V_{\ell, \text{id}}} [\alpha \cdot f(D \cap \text{OPT}_{\ell, \text{id}})]$$

$$\geq \alpha \cdot \mathbb{E}_{V_{\ell, \text{id}}} [f(D \cap \text{OPT}_{\ell, \text{id}})]$$

$$\geq \alpha \cdot f^* (\mathbb{E}_{V_{\ell, \text{id}}} [1_{D \cap \text{OPT}_{\ell, \text{id}}}])$$

$$= \alpha \cdot f^* (p_{\ell, \text{id}}).$$

(\because \text{Lovász Ext. (2)})

(\because \text{Eqn.3.})

$$\mathbb{E}_{V_{\ell, \text{id}}} [f(\text{GREEDY}(V_{\ell, \text{id}}, \ell, \text{id}))] \geq \frac{\alpha}{(\ell + 1)} f(\text{OPT}_{\ell, \text{id}}).$$  (4)

Theorem 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 $V_{\ell, \text{id}} \subseteq W$ denote the subset of $W$ accessible to a node $(\ell, \text{id})$. Let $\text{OPT}_{\ell, \text{id}}$ be an optimal solution computed from the subset $V_{\ell, \text{id}}$ for the submodular function $f$ with constraints $\mathcal{C}$. If Greedy is an $\alpha$-approximate algorithm, then

$$\mathbb{E}_{V_{\ell, \text{id}}} [f(\text{GREEDY}(V_{\ell, \text{id}}, \ell, \text{id}))] \geq \frac{\alpha}{(\ell + 1)} f(\text{OPT}_{\ell, \text{id}}).$$

Proof. We prove this theorem by induction on the level $\ell$.

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 $V_{\ell, \text{id}}$. The result follows since the Greedy algorithm has the approximation ratio $\alpha$.

Inductive case, $\ell = \ell' + 1$: We first obtain a relation for the quality of the solutions at level $\ell'$ compared to the quality of an optimal solution. For each child $c$, let $S_c$ be a solution computed by the Greedy algorithm from the data $V_c \subseteq V_{\ell, \text{id}}$.

From the induction hypothesis applied to child $c = (\ell', \text{id})$, the approximation ratio obtained as a result of the computation Greedy$(V_c, \ell', \text{id})$ is $\alpha/((\ell' + 1) = \alpha/\ell$. This implies that $\mathbb{E}_{V_{\ell', \text{id}}} [f(S_c)] \geq \frac{\alpha}{\ell} \cdot f^*(1_{\text{OPT}_{\ell', \text{id}}})$. Therefore we can apply Lemma 2 to get

$$\mathbb{E}_{V_{\ell', \text{id}}} [f(S_c)] \geq \frac{\alpha}{\ell} \cdot f^*(1_{\text{OPT}_{\ell', \text{id}}} - p_{\ell, \text{id}}).$$  (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 3, we have

$$\mathbb{E}_{V_{\ell, \text{id}}} [f(S)] \geq \alpha \cdot f^*(p_{\ell, \text{id}}).$$  (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. 5 and Eqn. 6 to find lower bounds for $T$.

$$\mathbb{E}_{V_{\ell, \text{id}}} [f(T)] \geq \alpha \cdot f^*(p_{\ell, \text{id}}) \text{ and } \mathbb{E}_{V_{\ell, \text{id}}} [f(T)] \geq \frac{\alpha}{\ell} \cdot f^*(1_{\text{OPT}_{\ell, \text{id}}} - p_{\ell, \text{id}}).$$

By multiplying the second inequality by $\ell$ and then adding it to the first, we get
A Parallel Algorithm for Maximizing Submodular Functions

Algorithm 2: Our Randomized Multi-level GREEDYL Algorithm

1: procedure GreedyL(V: Dataset, b: branching factor, m: number of machines, r: random tape)
2: Let \{P_0, P_1, \ldots, P_{m-1}\} be uniform random partition of V using r.
3: for i = 1 \ldots m - 1 in parallel do
4: \( \ell = \text{level}(id, b) \)
5: Run GreedyL'(V_i, \ell, b, i) to obtain \( S_i \) on machine \( i \)
6: end for
7: Run GreedyL'(V_0, \lfloor \log_b m \rfloor, b, 0) to obtain \( S_0 \) on machine 0
8: return \( S_0 \)
9: end procedure

Algorithm 2 describes our multilevel distributed algorithm in two functions. The first function \textsc{GreedyL} is a wrapper function that sets up the environment to run the distributed algorithm. The second function \textsc{GreedyL'} is the iterative implementation of the recurrence relation that runs on each machine. The \textsc{GreedyL'} procedure is an iterative implementation of the recurrence relation 2 run 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 highest active level for the machine.

Pseudocode. Algorithm 2 describes our multilevel distributed algorithm in two functions. The first function \textsc{GreedyL} is a wrapper function that sets up the environment to run the distributed algorithm. The second function \textsc{GreedyL'} is the iterative implementation of the recurrence relation that runs on each machine. The \textsc{GreedyL'} procedure is an iterative implementation of the recurrence relation 2 run 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 highest active level for the machine.
the ground set by \( n \), the selection size by \( k \), the number of machines by \( m \), and the number of levels in the accumulation tree by \( L \).

The number of objective function calls made by the Greedy algorithm is \( O(nk) \) since \( k \) elements are selected to be in the solution, and we may need to do \( O(n) \) marginal gain for each of them. The RandGreedi algorithm makes \( O(k(n/m + mk)) \) function calls, where the second term comes from the accumulation step. The GreedyL algorithm makes \( O(k(n/m + Lbk)) \) calls, where the branching factor \( b = \lceil m^{1/L} \rceil \).

We note that the time complexity of a function call depends on the specific function being computed. E.g., in the \( k \)-coverage and the dominating set problems, computing a function costs \( O(\Delta) \); here \( \Delta \) is the size of the largest itemset for \( k \)-coverage and the maximum degree of a vertex for the 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 GreedyL algorithm.

The \( k \)-medoid problem computes a local objective function value and has a complexity of \( O(n'\Delta) \) where \( \Delta \) 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 GreedyL 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 \( O(k\Delta) \). Since at each level, a parent node receives messages from 6 children, the communication complexity is \( O(k\Delta Lb) \). Therefore the communication complexity for the Greedi algorithm is \( O(k\Delta m) \) and for the GreedyL algorithm is \( O(k\Delta L \lceil m^{1/L} \rceil) \). We summarize these results Table 1 in the Appendix.

### 5 Related Work

Barbosa et al. [2] have compared the RandGreedi algorithm with the sample and prune algorithm of Kumar et al. [7], and shown that the former performs better than the latter for practical quality of the computed approximate solution. We observe this even though the second algorithm has a better worst-case approximation ratio in expectation. (The sample and prune algorithm achieves an expected approximation 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.)

More recent work on distributed submodular maximization uses the multi-linear extension to map the problem into a continuous domain. The authors of these papers perform a gradient ascent on each local machine and build a consensus solution in each round, and improve the approximation factor to \( (1 - 1/e) \) [3, 12, 14].

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 ones with implementation solve problems with hundred elements in the data set [14].

One potential criticism of our parallel algorithm is that the worst-case approximation ratios achieved by it are lower, \( \frac{1}{4}(1 - 1/e) \), than the RandGreedi algorithm \( (\frac{1}{4}(1 - 1/e)) \), when the number of levels \( L > 2 \). However, our results show that the approximation ratios achieved in practice on several test problems do not show this loss in approximation quality.
with levels. This observation agrees with the one of Barbosa et al. mentioned earlier in this section, and previous results on many matching and edge cover problems (e.g., see the surveys by Hougardy [5] and Pothen, Ferdous and Manne [13]). The latter article concludes that simpler algorithms with 1/2- or 2/3 – ε-approximation ratios for edge-weighted matching, or 1/2- or 2/3-approximation ratios for vertex-weighted matching, achieve better practical approximation quality than a more involved (1 – ε)-approximation algorithm. The former algorithms have more parallelism, easier to implement, and are practically orders of magnitude faster.

6 Experiments

We evaluate our algorithm for maximizing monotone submodular functions subject to a cardinality constraint on three different classes of problems described later. We ran our codes on an AMD Epyc 7662, 2.0 GHz processor with 128 cores, with 2 GB of memory per core. Our codes are written in C++11, gcc version 9.3.0, using the O3 optimization flag. We used MPI as the communication library in the parallel code.

We run these experiments on accumulation trees with differing numbers of machines (m) and varying numbers 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 since we are mapping the data to machines uniformly at random.

Whenever memory constraints allow, we compare our results with the GREEDY algorithm that runs on a single machine. We use the Lazy Greedy [10] variant that is faster in practice than the standard GREEDY 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.

The first problem we consider is maximum k-cover. Given a collection of subsets of a ground set and an integer k, the goal is to select k of these subsets and cover as many elements as possible. The second problem is the k-vertex dominating set, which is a special case of the set covering problem on graphs. Each vertex u dominates its neighboring vertices, and we select k vertices to dominate as many vertices as possible. The third is the k-medoid problem [6]. Given a collection of elements in a ground set W, and a distance d(e, v) defined on every pair of elements, the goal is to select a subset V ⊆ W of size k to minimize \( f(V) = \sum_{e \in W} \min_{v \in V} d(v, e) \). In the data set we use for this problem, an element is an image and the distance between two images is the Euclidean distance between vector representations of the pixels in the images.

We organize the rest of this section into four experiments that illustrate the properties of the GREEDY algorithm.

6.1 Machines with varying memory sizes

![Figure 3](image-url) Results from GREEDYL for k-vertex dominating set on the Friendster dataset with three different machine organizations. The number of machines m and the memory size per machine are varied, and the accumulation tree is selected correspondingly.
The first experiment demonstrates how the GreedyL algorithm solves problems on parallel machines when the memory is insufficient for the RandGreedi and Greedy algorithms. We consider the $k$-dominating set problem on graphs, and first report results on the Friendster dataset [16] that represents a directed graph between friends on the social networking website. It contains 117.8 Million nodes and 2.59 Billion directed edges. Due to the size of this graph, the Greedy algorithm needs at least 32 GB of memory. 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 run this problem only on 8 machines, each with 4 GB of memory, since in the accumulation step, one machine receives partial solutions of size 512 MB each from 8 machines. Since the GreedyL algorithm has multiple levels of accumulation, it can run on 16 machines with only 2 GB memory, using $L = 2$ and $b = 4$; it can also run on 32 machines with only 1 GB memory, using $L = 5$ and $b = 2$.

We report results from these three machine configurations in the subfigures of Figure 3. We report the function evaluations on a critical path: a path from a leaf to the root of the accumulation tree. Due to parallelism in the computation, this variation is less significant than the variation in the total number of function evaluations (for the latter, see Figure 8 in the Appendix.) We also report the objective function values normalized by those obtained from the serial Greedy algorithm. Our results show that objective function values computed by the GreedyL algorithm (the 2 and 1 GB results) are insensitive to the number of levels in the tree. We observe similar trends for the webdocs [9] and road_usa [1] datasets, shown in Figure 7 in the Appendix. 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.

### 6.2 Machines with highly restricted memory

![Figure 4](image-url) Results from GreedyL for $k$-vertex dominating set on the road_usa dataset on 16 machines with restricted memory. Figures show the accumulation trees needed for specific $k$ values.

This experiment shows how we can use the GreedyL algorithm to solve problems with large partial solution sizes when the machines available have extremely low memory, as would happen in an edge computing context. Again we consider the $k$-vertex dominating set, and first report results on the road_usa dataset [1]. It has 23.9 Million vertices and 57.7 Million edges. We chose this graph since it has a relatively small average vertex degree, leading to large vertex-dominating sets. The experiment is run on a fixed architecture of 16 machines with a simulated environment having extremely limited memory of 100 MB.

The first subfigure in Figure 4 plots the number of function evaluations with varying values of $k$. The RandGreedi algorithm has $b = 16$ since there are 16 machines when $k = 128$. When $k$ increases, the memory requirement of the accumulation step exceeds the limited memory size of 100 MB on a machine, and the RandGreedi algorithm fails.
Instead, we use the GreedyL algorithm using multiple levels in the accumulation tree, with the number of levels increasing from 1 to 3. The results show that the number of function evaluations in the Greedy algorithm is greater than the number of function evaluations on the critical path in the GreedyL algorithm, as we have seen in the earlier experiment. The former does not vary much with \( k \); however, it increases with \( k \) for the GreedyL algorithm since the partial solutions are larger, the number of levels increases, and \( b \) decreases. The second subfigure in Figure 4 plots the objective function values with varying \( k \). The results show that the RandGreedi and GreedyL algorithms attain function values comparable to that of the serial Greedy algorithm. Similar trends can be observed for other graphs in the summary of results shown in Figure 5.

### 6.3 Machines with no memory restrictions

![Figure 5 Results from GreedyL for \( k \)-vertex dominating set on the different road datasets and \( k \)-cover on different set cover benchmark datasets on 32 machines.](image)

For the third experiment, we consider the situation where there are no restrictions on memory on the machines, and hence we can vary the selection set sizes \( k \), and the number of levels and branching factors in the accumulation trees. Further, we obtain results for six problems from two distinct problem sets, and in Figure 5, we provide summary results on the number of function evaluations and the running times.

The first set of problems is the set of road maps from the DIMACS10 [1] dataset, containing roadusa, belgium_osa, and roadcentral graphs. The second problem set is the Itemset Mining dataset [9] used for benchmarking set covers, containing the webdocs, retail, and korasak datasets. We report the geometric means over the six problems for the various quantities of interest.

The first subfigure shows the number of submodular objective function evaluations in the critical path of the GreedyL and RandGreedi algorithms, relative to the Greedy algorithm, as the number of levels and the parameter \( k \) are varied. When \( k \) is small there is no significant change in the number of function calls, since a significant amount of work is performed at the leaves. For higher values of \( k \), we can observe that the GreedyL algorithm has fewer function evaluations relative to the RandGreedi algorithm (\( L = 1, b = 32 \)). This is because the accumulation step for the RandGreedi algorithm does a significant amount of work with increasing \( k \). This cost can be reduced by selecting different accumulation trees in the GreedyL algorithm (e.g., \( L = 2, b = 8 \)). The run time and function values for these two accumulation trees are provided in Table 4 and Table 3 in the Appendix.

The second subfigure chooses \( k = 32,000 \), and plots a scaled value of the function calls in the critical path and the run times for different \((L, b)\) pairs. We have scaled each value by the mean values over all the \((L, b)\) pairs shown to bring the number of function calls and the times to the same scale. 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; hence communication cost remains low even when $k$ is large.

The objective function values obtained by the GreedyL 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 $k$-set coverage problem (Figure 8 in the Appendix). However, GreedyL obtains objective function values that are about 20\% higher than the RandGreedi and GreedyL algorithms.

6.4 The $k$-Medoid Problem

![Figure 6](image)

Figure 6 Results from GreedyL for the $k$-medoid problem on the Tiny image data set. The two plots show the objective function values and time when the gain is computed on each machine with the addition of 1000 randomly selected images to the data set on the machine, and computed without adding any images.

In the fourth experiment, we consider the $k$-medoid problem where the objective function is global, and hence can only be approximated with local information. Additionally the function computation is compute intensive. In the image data we work on, the volume of communication is also high since each image has a large number of features (pixels).

We use images from the tiny image dataset [4] (we used the training set data) which contains 200 different classes and 500 images from each class. Each image is $64 \times 64$ pixels in size. Barbosa et al. [2] and Mirzasoleiman et al. [11] have also reported results on this problem using smaller images ($32 \times 32$ pixels).

The computation of the objective function $f(V)$ requires access to all of $W$, or it can be approximated in a distributed manner, which would make it expensive. Mirzasoleiman et al. show that computing $f(D)$ for some subset $D \subseteq V$ chosen uniformly at random gives a provably good approximation to the function value $f(V)$ (Theorem 10, [11]). They use this to justify computing the objective function values locally on each machine. They have also added subsets of randomly chosen images to the machines to provide better approximations of the objective function. We have followed these techniques in our multilevel GreedyL algorithm.

In our experiments, we fix the machines ($m = 32$) and vary the accumulation trees by varying $L$ and $b$. We set the selection size $k$ to 500 images. We also considered the variant described by (Mirzasoleiman et al. [11]), in which 1,000 additional random images from the original dataset were added to each accumulation step.

In the first subfigure of Figure 6, we plot the objective function values for different accumulation trees. The additional images improve the objective functions slightly. The objective function values are quite similar for both RandGreedi and GreedyL algorithms.

In the second subfigure, we plot the run times for the two algorithms. Our results show that the GreedyL algorithm with $L = 5, b = 2$ is six times faster than the RandGreedi algorithm. Note the difference from the $k$-cover problems, where run times depend on $L$ in a more complex way (Figure 5).
A Parallel Algorithm for Maximizing Submodular Functions

References


Algorithm 3 Greedy Algorithm

1: procedure Greedy (V: Dataset)
2: \[ S \leftarrow \emptyset \]
3: while True do
4: \[ E \leftarrow \{ e \in V \setminus S : S \cup \{ e \} \in \mathcal{C} \} \]
5: \[ e' \leftarrow \arg \max_{e \in E} f(S \cup \{ e \}) \]
6: if \[ f(S \cup \{ e' \}) = f(S) \text{ or } E = \emptyset \] then
7: \[ \text{return } S \]
8: end if
9: \[ S \leftarrow S \cup \{ e' \} \]
10: end while
11: end procedure

Lemma 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 \((\ell, id)\), and consider adding elements of \( B \subseteq W \) to this node. If we have \( \text{GreedyL}(V \cup \{ e \}, \ell, id) = \text{GreedyL}(V, \ell, id) \), for each element \( e \in B \), then \( \text{GreedyL}(V \cup B, \ell, id) = \text{GreedyL}(V, \ell, id) \).

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 \( V \cup \{ e \}, \forall e \in B \); and \( V \cup B \).

Proof. If possible, let \( \text{GreedyL}(V \cup B, \ell, id) \neq \text{GreedyL}(V, \ell, id) \). 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 \( \text{GreedyL}(V \cup \{ e \}, \ell, id) \). Since \( e \) is the first element of \( B \) that was selected by the Greedy algorithm, the elements chosen before it at level \( i \) in \( \text{GreedyL}(V \cup B, \ell, id) \) are the same ones chosen before it at level \( i \) in \( \text{GreedyL}(V \cup \{ e \}, \ell, id) \). Since it was not selected in \( \text{GreedyL}(V \cup \{ e \}, \ell, id) \) it will not be selected in \( \text{GreedyL}(V \cup B, \ell, id) \). This is a contradiction since \( e \) needs to be selected at every level to be present in the final solution. \( \blacksquare \)
A Parallel Algorithm for Maximizing Submodular Functions

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Greedy</th>
<th>RandGreedi</th>
<th>GreedyL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements per leaf node</td>
<td>$n$</td>
<td>$n/m$</td>
<td>$n/m$</td>
</tr>
<tr>
<td>Elements per interior node</td>
<td>$0$</td>
<td>$km$</td>
<td>$k \left[ m^{1/L} \right]$</td>
</tr>
<tr>
<td>Total Function Calls</td>
<td>$kn$</td>
<td>$k(n/m + km)$</td>
<td>$k(n/m + Lk \left[ m^{1/L} \right])$</td>
</tr>
</tbody>
</table>

Cost Per call

<table>
<thead>
<tr>
<th>$k$-cover</th>
<th>$\Delta n$</th>
<th>$\Delta k(n/m + km)$</th>
<th>$\Delta k(n/m + Lk \left[ m^{1/L} \right])$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime complexity</td>
<td>$\Delta n$</td>
<td>$\Delta k(n/m + km)$</td>
<td>$\Delta kL \left[ m^{1/L} \right]$</td>
</tr>
<tr>
<td>Communication cost</td>
<td>$0$</td>
<td>$\Delta km$</td>
<td>$\Delta km$</td>
</tr>
</tbody>
</table>

$\Delta$: number of neighbours

<table>
<thead>
<tr>
<th>$k$-medoid</th>
<th>$\Delta n$</th>
<th>$\Delta n/m$</th>
<th>$\Delta n/m$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime complexity</td>
<td>$\Delta n$</td>
<td>$\Delta k(n/m)^2 + (km)^2$</td>
<td>$\Delta kL \left[ m^{1/L} \right]$</td>
</tr>
<tr>
<td>Communication cost</td>
<td>$0$</td>
<td>$\Delta km$</td>
<td>$\Delta km$</td>
</tr>
</tbody>
</table>

$k$: number of features

### Table 1 Complexity Results for $k$-cover and $k$-medoid Problems. Here 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 $L$.

<table>
<thead>
<tr>
<th>Graph</th>
<th>$n$</th>
<th>$m$</th>
<th>avg. degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Friendster</td>
<td>65,608,366</td>
<td>1,806,067,135</td>
<td>27.52</td>
</tr>
<tr>
<td>road_usa</td>
<td>23,947,347</td>
<td>57,708,624</td>
<td>2.41</td>
</tr>
<tr>
<td>road_central</td>
<td>14,081,816</td>
<td>33,866,826</td>
<td>2.41</td>
</tr>
<tr>
<td>belgium_osm</td>
<td>1,441,295</td>
<td>3,099,940</td>
<td>2.14</td>
</tr>
<tr>
<td>webdocs</td>
<td>1,692,082</td>
<td>299,887,139</td>
<td>177.22</td>
</tr>
<tr>
<td>korasak</td>
<td>990,002</td>
<td>8,018,988</td>
<td>8.09</td>
</tr>
<tr>
<td>retail</td>
<td>88,162</td>
<td>908,576</td>
<td>10.31</td>
</tr>
<tr>
<td>tiny_image</td>
<td>10,000</td>
<td>122,880,000</td>
<td>12,288</td>
</tr>
</tbody>
</table>

### Table 2 Properties of Datasets used in the Experiments.

Figure 7 Results showing similar trends to Friendster dataset in the experiments on machines with varying memory. The selection sizes are 32K for webdocs and road_usa datasets and 1M for the Friendster dataset.
Figure 8 Results from GREEDYL for the Maximum coverage problem on the webdocs dataset. The function values relative to the GREEDY algorithm are plotted against values of set cover size $k$ when 32 processors are employed.

Figure 9 Results from GREEDYL for $k$-vertex dominating set on different road datasets, and $k$-cover on different set cover datasets, both on 32 machines.

<table>
<thead>
<tr>
<th>Graph</th>
<th>$k = 2,000$</th>
<th>$k = 2,000$</th>
<th>$k = 8,000$</th>
<th>$k = 8,000$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$L = 1, b = 32$</td>
<td>$L = 2, b = 8$</td>
<td>$L = 1, b = 32$</td>
<td>$L = 2, b = 8$</td>
</tr>
<tr>
<td>road_usa</td>
<td>10,114</td>
<td>10,128</td>
<td>40,454</td>
<td>40,495</td>
</tr>
<tr>
<td>road_central</td>
<td>10,103</td>
<td>10,138</td>
<td>34,903</td>
<td>34,946</td>
</tr>
<tr>
<td>belgium_osm</td>
<td>6152</td>
<td>6152</td>
<td>19349</td>
<td>19352</td>
</tr>
<tr>
<td>webdocs</td>
<td>2,181,430</td>
<td>2,181,420</td>
<td>2,768,280</td>
<td>2,768,280</td>
</tr>
<tr>
<td>korasak</td>
<td>22,308</td>
<td>22,308</td>
<td>25,216</td>
<td>25,216</td>
</tr>
<tr>
<td>retail</td>
<td>12,353</td>
<td>12,364</td>
<td>16,470</td>
<td>164,70</td>
</tr>
</tbody>
</table>

Table 3 Function Values from RANDGREEDI and GREEDYL algorithms for $k$-vertex dominating set on the different road datasets, and $k$-cover on different itemset data.
### Table 4

Execution Time in seconds from GreedyL and RandGreedi algorithms for $k$-vertex dominating set on the different road datasets, and $k$-cover on different itemset data.

<table>
<thead>
<tr>
<th>Graph</th>
<th>$k = 2,000$</th>
<th>$k = 2,000$</th>
<th>$k = 8,000$</th>
<th>$k = 8,000$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$L = 1, b = 32$</td>
<td>$L = 2, b = 8$</td>
<td>$L = 1, b = 32$</td>
<td>$L = 2, b = 8$</td>
</tr>
<tr>
<td>road_usa</td>
<td>3.15</td>
<td>3.11</td>
<td>3.16</td>
<td>3.12</td>
</tr>
<tr>
<td>road_central</td>
<td>3.15</td>
<td>3.10</td>
<td>3.16</td>
<td>3.11</td>
</tr>
<tr>
<td>belgium_osm</td>
<td>4.4</td>
<td>3.54</td>
<td>5.12</td>
<td>4.49</td>
</tr>
<tr>
<td>webdocs</td>
<td>3.96</td>
<td>3.76</td>
<td>4.83</td>
<td>4.34</td>
</tr>
<tr>
<td>korasak</td>
<td>3.13</td>
<td>3.13</td>
<td>3.34</td>
<td>3.22</td>
</tr>
<tr>
<td>retail</td>
<td>3.07</td>
<td>3.02</td>
<td>3.15</td>
<td>3.09</td>
</tr>
</tbody>
</table>