Published in final edited form as:

Proc Int Workshops Parallel Proc. 2021 August ; 2021: . doi:10.1145/3458744.3473365.

Towards Faster Execution of Ensemble ML Bootstrap Based Techniques

Vinay B Gavirangaswamy,

Department of Computer Science, Western Michigan University

Vasilije Perovi ,

Department of Mathematics, University of Rhode Island

Ajay Gupta,

Department of Computer Science, Western Michigan University

Hisham M Saleh

Department of Computer Science, Western Michigan University

Abstract

Algorithms for ensemble methods (EM) based on bootstrap aggregation often perform copious amount of redundant computations (RC) thus limiting their practicality. Given this constraint, we propose a framework that views these algorithms as a collection of computational units (cu), a tightly coupled set of both mathematical operations *and* data. This view facilitates a reduction in RC (RRC), thereby allowing for faster execution plans. Inspired by the floor tiling approach in VLSI, we look to engineer solutions for RRC while possibly reconfiguring the underlying computing system's compiler technology stack. We start by showing that under the assumption that the computational system has unbounded but finite memory (i.e., the memory is large enough to hold all intermediate values) and that each cu has a uniform cost, our approach reduces to a well-studied *directed bandwidth problem* for the directed acyclic graphs (DAGs). Next, we consider a more realistic scenario where the computing system has limited memory and concurrent execution while still assuming a uniform cost. Using a new notion of (r,s) set cover of a DAG (nodes representing computational units and edges representing their interdependencies) we formulate the problem of reducing redundant computational steps in EM as a variation of a directed bandwidth problem. We show that the graph's minimum bandwidth is closely related to memory requirements for studying RRC. Finally, our preliminary experimental results are supportive of the proposed approach for RRC and promising that it can be applied to a broader set of algorithms in decision sciences.

v5gavirangas@wmich.edu .

Publisher's Disclaimer: Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Keywords

automatic redundancy reduction; compiler; DAG; dependence analysis; machine learning; parallel processing; theory of computation

1 INTRODUCTION & PROBLEM STATEMENT

Today, we live in a world with an enormous amount of data being available in connection to nearly any aspect of human life. Contrary to popular belief, the mere existence of such data does not necessarily translate into a better decision making, though the ability to process and analyze it typically does. Over the last few decades, advances in parallel processing technologies have made Machine Learning (ML) possible at an industrial scale, thus enabling scientists to make “better” data-driven decisions and also providing a foundation for an automated decision-making. Majority of the contemporary technologies used to scale ML workloads exploit tasks, data, or both in parallelism. In general, advances in technological stacks for computation, such as processors, peripherals, hardware, and software technologies, have led to higher speedup and efficiency in executing computations. Much of this effort in improving computing technology is, in fact, in the direction of finding new avenues to increase speedup, which are broadly governed by Amdahl’s [1] and Moore’s Laws [11].

In contrast to the traditional approaches of task, data, and model parallelisms when assessing an algorithm’s performance, we hypothesize that the nature of an underlying dataset also plays an important role. More specifically, for a given ML algorithm, we look to improve its computational performance by exploiting hidden structures in the underlying datasets that cause the algorithm to perform numerous redundant computations. In support of this point, we limit our discussion to reinforcement learning algorithms based on an ensemble of clustering and classification techniques. These algorithms fundamentally translate to applying mathematical/statistical operations over some datasets. Datasets mainly come in two different forms, streamed or stored, and computations drawing similar inferences over different datasets can vary significantly. Our proposed scheme assumes that the input data is known. This perspective allows in identifying redundant computations that would otherwise are tedious to spot.

1.1 Redundant Computations (RC)

A skeleton for a typical machine learning algorithm belonging to the class of the ensemble methods can be found in Algorithm 1. Applications of this type of algorithm include satellite image analysis, natural language processing on web text, and analysis of experimental data in life sciences, to name a few.

Algorithm 1: General skeleton of an ensemble commonly used in data analytics

Step 1: Create bootstrap datasets .

Step 2: Apply clustering operation on bootstrap datasets .

Step 3: Aggregate results from Step 2.

Step 4: Learn from Step 3.

Step 5: Summarize learning .

The primary source of redundant computations performed by an ensemble is Step 2 (see Algorithm 1 and Figure 1). On one hand, bootstrap datasets are often very similar¹ and thus performing computations over them undoubtedly leads to redundancy. On the other hand, even in the case of a single dataset, redundant computations are being performed among individual components of an ensemble. Figure 1 depicts a typical ensemble clustering algorithm, where the steps with the same color denote examples where redundant computations arise (while it's not critical to be able to read text in Figure 1, one can do so by zooming on the e-copy of the paper). Our goal in this paper consists of identifying and removing these kinds of redundancies as a way to increase the speedup in algorithm's implementation. Ultimately, we strive to be able to automatically identify redundancies and eliminate them by reusing the redundant computations once and storing them in memory. However, it is not feasible to keep all partial results in memory indefinitely, so we specify a bound on how long they are kept. It is not hard to see that this is a NP-hard problem (as, e.g., a sub-problem, namely linear graph bandwidth is a known NP-hard problem), but in Section 2 we show how it can be transformed to a variation of a graph bandwidth problem, for which we can develop heuristics to remove redundancies partially.

We conclude this section with a simple and concrete example of computing distance between vectors in $\mathbb{R}^2$. Note that this type of computation is representative of types of redundant computations one may encounter in the same-color boxes in Figure 1.

Example 1.1. For brevity, the program in Algorithm 2 is referred to as Dist-program throughout the rest of the paper. Dist-program captures the essence of finding similarity between two objects. The program operates over three variables derived from the Cartesian plane. It defines three methods, one of each for multiplication, addition, and subtraction of two numbers. In Figure 2 we display the assembly code generated by GCC for x86-64 in case of Dist-Program and use its structure to facilitate problem formulation. More specifically, the colored sections of code in the first column of Figure 2 identify aggregations of Computational Units (cu). The second column in Figure 2 shows removal of RC by

¹By “similar” we mean datasets that are close to each other with respect to some measure, e.g., classical set intersection or in case of images the mean square error.

reusing results stored from the previous computations instead of repeatedly recomputing. A more formal exposition of the problem is in Section 2.

Algorithm 2: Pseudocode for Calculating Distances(Dist - Program)

```

1: Input:  $a, b, c \in \mathbb{R}^2$ .
2: Output:  $dac$  - squared Euclidean dist. b/w  $a$  and  $c$ ,
            $davgc$  - squared Euclidean dist. b/w  $(a+b)$  and  $c$ .
3:  $dac = (a_1 - c_1) \cdot (a_1 - c_1) + (a_2 - c_2) \cdot (a_2 - c_2)$ 
4:  $davgc = (a_1 - c_1 + b_1) \cdot (a_1 - c_1 + b_1) + (a_2 - c_2 + b_2) \cdot (a_2 - c_2 + b_2)$ 

```

1.2 Computational Units and Execution Plans

To get a sense of what kind of improvements one may expect when it comes to reducing redundant computations, we consider performing pairwise comparison between *all* elements in sets $A = \{2, 3, 4, 5\}$, $B = \{2, 3, 5, 6, 8, 9, 13\}$, and $C = \{12, 43, 7, 89, 23, 55\}$. There are total of 17 elements in these three sets and the number of pairwise comparisons one would have to perform is $^{17}C_2 = 136$.

In Figure 3, Scenario 1 is a simple case of sequential execution of three computational workloads S_1 , S_2 , and S_3 . Where S_1 is a pairwise comparison among elements of A (6), A to B (28) and A to C (24), S_2 is between B to B (21) and B to C (42), and S_3 is between elements of C (15). Scenario 2 is an example of running workloads that have a portion of the computational units (cu) in common. The highlighted section in Scenario 2 represents a portion of common cu's, i.e., $S_4 \cap S_5$ represents pairwise comparisons for elements in $A - (A \cap B)$, $B - (A \cap B)$ and C with set resulting from $A \cap B$. Number of elements in each set for Scenario 2 are $|A \cap B| = 3$ represented by $S_4 \cap S_5$ performing 33 pairwise comparisons. Remaining workloads in the figure follow similar notations for elements $|A - (A \cap B)| = 1$ ($S_4 = 13$), $|B - (A \cap B)| = 4$ ($S_5 = 30$), and $|C| = 6$ ($S_6 = 15$). Hence the total comparisons necessary for Scenario 2 is 91, making the cost of running workloads in this scenario smaller than that of Scenario 1. Consequently, in this example the speedup can be improved by a factor of $136/91 \approx 1.5$ for Scenario 2. This is in addition to the speedup achieved through parallelization of the algorithm itself. These scenarios are designed to draw distinction between improvements from traditional parallelism and approach which also considers redundancies involved.

2 PROBLEM FORMULATION

In this section, we define a new theoretical framework that takes advantage of opportunities to reduce redundant computations in algorithms for ensemble methods based on bootstrap aggregation and ultimately improves the overall algorithms' efficiency with respect to both computational time *and* resources required. Our approach to this engineering problem is to view it as a multistep optimization problem as described in Sections 2.1-2.4.

Key components of this framework worth highlighting include:

- i. defining the notion of a *computational unit* (cu) – an atomic unit consisting of both the data *and* the mathematical computations operating on that data.
- ii. generalizing the notion of a set cover over all computational units and discussing approaches to finding them.
- iii. proposing a floor tile-based execution plan (FTiP) of computational units conformable with a set cover.

To ease our way into the abstract definition of the proposed framework, we consider the notion of computational units (cu's) in terms of steps in Dist-Program given in Algorithm 2.

Example 2.1. Let $\mathcal{D}^{(t)} = \{a, b, c\} \subset \mathbb{R}^2$ be the dataset on which Dist-Program operates and let $\mathcal{P}^{(t)}$ be the *power set* of $\mathcal{D}^{(t)}$, i.e., $\mathcal{P}^{(t)}$ is the set of *all* subsets of $\mathcal{D}^{(t)}$. Since $\mathcal{D}^{(t)}$ contains three elements, the total number of subsets of $\mathcal{D}^{(t)}$ is $2^3 = 8$ and they are given below.

# elt's.	subsets of $\mathcal{D}^{(t)}$
0	$\mathcal{D}_1^{(t)} = \emptyset$
1	$\mathcal{D}_2^{(t)} = \{a\}, \mathcal{D}_3^{(t)} = \{b\}, \mathcal{D}_4^{(t)} = \{c\},$
2	$\mathcal{D}_5^{(t)} = \{a, b\}, \mathcal{D}_6^{(t)} = \{a, c\}, \mathcal{D}_7^{(t)} = \{b, c\},$
3	$\mathcal{D}_8^{(t)} = \{a, b, c\}$

Now let $m_j^{i,t} = \text{null}$ denote the j^{th} *computational unit acting on subset* $\mathcal{D}_i^{(t)}$. Then for $i = 1, 2, 3, 4, 5, 7$, we have that $m_j^{i,t} = \text{null}$, i.e., there are no cu's acting solely on subsets $\mathcal{D}_i^{(t)}$ produced by Dist-Program. On the other hand, the nontrivial cu's corresponding to the implementation outlined in the first column of Figure 2 are given in Figure 4, where arrows from left to right indicate the order in which cu's have to be executed.

It is important to emphasize that nontrivial cu's consist of either mathematical operations acting directly on a subset of data, e.g., $m_1^{6,t}$, or a combination of mathematical operations and data representing the union of datasets with the individual cu's. For example, subsets $\mathcal{D}_6^{(t)}$ and $\mathcal{D}_8^{(t)}$ are associated with computational units $m_1^{6,t}$ and $m_1^{8,t}$ respectively, and so a new computational unit $m_1^{6,t} + b_1$ is associated with the subset $\mathcal{D}_6^{(t)} \cup \mathcal{D}_3^{(t)} = \mathcal{D}_8^{(t)}$.

Finally, from Figure 4 we see that Dist-Program is determined completely by those 18 cu's and ten dependencies indicated by arrows. Given that the size of the example is small, one can easily identify pairs of *redundant* cu's in Figure 4. Consequently, Dist-Program can be captured entirely with a smaller set of cu's, denoted by $\tilde{m}$, which are given in Figure 5, along with the dependencies with respect to the relabeled cu's.

We conclude this section with Algorithm 3 that generalizes the discussion from Example 2.1 and whose steps are described in more details in subsequent sections.

Algorithm 3: Reduction in Redundant Computations(RRC)

- Step 1:** Decompose Ensemble Clustering ML algorithm into collection of computational units $\mathcal{M}$.
- Step 2:** Identify and collect all *distinct* computational units from $\mathcal{M}$ into $\widetilde{\mathcal{M}}$ along with interdependencies.
- Step 3:** Find an aggregation of cu's, $\mathfrak{C}$, with execution cost lower than that of $\mathcal{M}$.
- Step 4:** FTiP, Floor Tile Planning - Find an efficient execution plan for $\mathfrak{C}$ on the compute resource.
-

2.1 Step 1: Creation of Computational Units from Ensemble ML Algorithms

We start by establishing some notation. A typical ML ensemble algorithm based on clustering consists of multiple *distinct* clustering algorithms denoted here by B_i , $i = 1, 2, \dots$. Furthermore, each algorithm B_i can be tuned to produce additional variations, e.g., depending on the number of parameters. We denote these variations by $\beta_{i,1}, \beta_{i,2}, \dots, \beta_{i,p(i)}$, where $p(i)$ is a positive integer depending On i . For the sake of reducing clutter, whenever there is no confusion, we write p instead of $p(i)$.

Next, each variation of clustering algorithms operates on subsets of the given dataset $\mathcal{D}$. These subsets are bootstrapped from $\mathcal{D}$ and are denoted by $\mathcal{D}^{(t)}$, $t = 1, \dots, k$. The *power set* of $\mathcal{D}^{(t)}$, denoted by $\mathcal{P}^{(t)}$, is the set of all subsets of $\mathcal{D}^{(t)}$ and has the cardinality $q(t) = 2^{|\mathcal{D}^{(t)}|}$. Here we emphasize that $\mathcal{D}^{(t)}$ is to be viewed as a collection of *specified* objects, which are then used to construct the corresponding power set. For example, if $\mathcal{D}^{(t)}$ is a matrix, then $\mathcal{D}^{(t)}$ can be viewed as either a collection of numbers or column vectors or row vectors. Each of these viewpoints has its advantages and leads to different power sets. This added flexibility in how the data is viewed allows for the analysis of *computational units* at different levels of granularity.

Finally, each variation of all clustering algorithms acts in some way on all subsets $\mathcal{D}_j^{(t)}$, for $j = 1, \dots, q(t)$ and $t = 1, \dots, k$. This results in creation of *computational units* (cu's) that are either trivial or nontrivial (see Example 2.1). In any case, since there is a finite number of clustering algorithm variations acting on a finite number of data subsets, the total number of all associated computational units is *finite*. Independently how these cu's are labeled (possibly by generalizing the labeling scheme from Example 2.1), they can be relabeled as $m_1, m_2, \dots, m_\omega$ for some positive integer ω . Of course, each of the relabeled cu's is accompanied with the relabeled data $\mathcal{D}_\omega$ which corresponds to some $\mathcal{D}_s^{(t)}$, $1 \leq t \leq k$ and $1 \leq s \leq q(t)$. Then the collection of all these cu's, $\mathcal{M}$,

$$\mathcal{M} = \{m_1, m_2, \dots, m_\omega\}, \quad (1)$$

together with the interdependencies among cu's, completely determine the original ML ensemble algorithm.

2.2 Step 2: Reducing Redundant Computations

As seen in Section 1.1, the structure of ensemble ML algorithms (bootstrapping and similar computational steps) inherently leads to redundant cu's. Analogous to Example 2.1, an ML algorithm characterized by $\mathcal{M}$ in (1) can in fact be described by a *smaller* set of *distinct* cu's, denoted by $\widetilde{\mathcal{M}}$. That is, there exists a positive integer $\widetilde{\omega} \leq \omega$ such that the given ML algorithm previously described by (1) can now be described by a smaller set of *distinct* cu's given by

$$\widetilde{\mathcal{M}} = \{\widetilde{m}_1, \widetilde{m}_2, \dots, \widetilde{m}_{\widetilde{\omega}}\} \subseteq \mathcal{M}. \quad (2)$$

Note that we need to allow for a possibility that we may only want or are able to remove a portion of all the redundancies from $\mathcal{M}$, i.e., we may consider a set of cu's $\mathcal{M}_a$ such that $\widetilde{\mathcal{M}} \subseteq \mathcal{M}_a \subseteq \mathcal{M}$. Now in order for both $\mathcal{M}$ and $\widetilde{\mathcal{M}}$ (or $\mathcal{M}_a$) to characterize the same ML ensemble algorithm, one also has to account for all the interdependencies between cu's. These dependencies put additional constraints on the cu's execution order. To that end, we let $\widetilde{\mathcal{N}}$ be the collection of dependencies between cu's in $\widetilde{\mathcal{M}}$. More specifically,

$$\widetilde{\mathcal{N}} = \{\widetilde{n}_j : j = 1, 2, \dots, \widetilde{\omega}\},$$

where each $\widetilde{n}_j$ is a sequence of length $\widetilde{\omega}$ and the c^{th} entry of sequence $\widetilde{n}_j$, denoted by $\widetilde{n}_j(c)$, is equal to 1 if $\widetilde{m}_c$ has to be executed before $\widetilde{m}_j$ and $c \neq j$; otherwise $\widetilde{n}_j(c) = 0$. Then the number of cu's on which $\widetilde{m}_j$ depends on is given by $|\widetilde{n}_j| = \sum_{c=1}^{\widetilde{\omega}} \widetilde{n}_j(c)$. The notions of $\mathcal{N}$ and $\mathcal{N}_a$ are defined in a similar way, but with respect to $\mathcal{M}$ and $\mathcal{M}_a$, respectively.

Example 2.2. Going back to the Example 2.1 and Figure 5, we see that $\widetilde{n}_i = (0, 0, \dots, 0)$ for $i = 1, 3$. When $i = 4, 5, 6, 10$ we have

$$\begin{aligned} \widetilde{n}_4 &= (0, 0, 1, 0, 0, 0, 0, 0, 0, 0), & \widetilde{n}_5 &= (0, 1, 0, 1, 0, 0, 0, 0, 0, 0), \\ \widetilde{n}_6 &= (1, 0, 0, 0, 0, 0, 0, 0, 0, 0), & \widetilde{n}_{10} &= (0, 0, 0, 0, 0, 0, 1, 0, 1, 0). \end{aligned}$$

In other words, there are no restrictions on execution of $\widetilde{m}_1$ and $\widetilde{m}_3$, though in order to execute $\widetilde{m}_5$, $\widetilde{m}_2$ and $\widetilde{m}_4$ must be executed before.

The notion of dependency lists and order of precedence for cu's execution can also be successfully described by *interaction graphs* (Figure 6). Nodes in the graph represent cu's in $\widetilde{\mathcal{M}}$ while directed edges represent execution dependencies [10]. All interaction graphs defined in this way fall in the family of *directed acyclic graphs (DAGs)* [10]. In fact, two ways of capturing interdependencies in $\widetilde{\mathcal{M}}$, via $\widetilde{\mathcal{N}}$ or the associated directed acyclic graph $\widetilde{\mathcal{G}}$,

can be used interchangeably. For reminder of this paper we focus our attention on $\widetilde{\mathcal{G}}$, though we emphasize that the notion of $\widetilde{\mathcal{N}}$ could prove more advantageous in future studies when developing advanced execution plans using linear programming techniques. We adopt the *convention* that if the pair $(\mathcal{M}, \mathcal{G})$ determines an ML algorithm, then the pairs $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{G}})$ and $\mathcal{M}_a, \mathcal{G}_a$ also determine the same algorithm, where $\widetilde{\mathcal{M}} \subseteq \mathcal{M}_a \subseteq \mathcal{M}$ and $\mathcal{G}_a$ is the associated interaction graph corresponding to $\mathcal{M}_a$.

In our formulation, identification of cu's is to reduce redundant computations in the ensemble clustering algorithm applied on a specified dataset. Characteristics of redundant computation are different and based on the programming paradigm and level of abstraction. Many contemporary processors base their hardware design and implementation on executing machine code, which uses an imperative programming paradigm. However, according to our definition, multiple instances of one or more computational step(s), the data yet not necessarily the variable define cu. Towards this, we hypothesize that those compiler technologies that can be applied after input data is given (inspector-executor) or during program execution (interference graph) are relevant. Our hypothetical compiler technology stack will have a customized low-level optimizer, an enhanced/modified version of the standard partial redundancy eliminator. The inspector-executor paradigm could capture some of the modified redundancy eliminator's internal workings and applied to given input data.

The internals of concrete implementation of this idea in a compiler stack is out of scope in the current work and is a topic for future investigation. Once we have labels assigned to the ensemble ML algorithm, one can follow an approach in pattern detection and classification applied in this context. Mainly, the algorithm would identify a recurring subset of labels across all the cu's. This problem boils down to finding repeated patterns among labels, and substantial research is available in this direction. Interleaving sections of labels that occur in between sections of labels that repeat from separate cu. Identified aggregation of labels for computational steps forms our $\mathcal{M}$. Furthermore, a subset of unique computational units in $\mathcal{M}$ forms $\widetilde{\mathcal{M}}$.

2.3 Step 3: Finding a set cover of $\widetilde{\mathcal{M}}$

In the previous two sections we have identified the pairs $(\mathcal{M}, \mathcal{G})$ and $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{G}})$ that represent the same ensemble ML algorithm. Under ideal conditions, given a machine with infinite resources in terms of compute cycles and memory, executing $\widetilde{\mathcal{M}}$ would be the most efficient execution plan for a given ML algorithm. With this in mind, we consider a function $cc()$ that takes any aggregation of cu's as input and returns the total cost required for its execution. If memoization is used to substitute previously computed results for repeated cu's in $\mathcal{M}$, then $cc(\widetilde{\mathcal{M}}) \leq cc(\mathcal{M})$ since $|\widetilde{\mathcal{M}}| \leq |\mathcal{M}|$ and the bandwidth of $\widetilde{\mathcal{G}}$ is bounded above by the bandwidth of $\mathcal{G}$ (further details in (3)). The cost incurred due to memoization is negligible because of infinite resources. Consequently, the lower bound on the possible reduction in the overall compute cost is achieved by $\widetilde{\mathcal{M}}$. On the other hand, since we are trying to improve on the $cc(\mathcal{M})$, we only focus on aggregations of cu's with total compute cost bounded from above by $cc(\mathcal{M})$.

The ideal case of having a machine with infinite resources, as discussed above, is just not realistic. Consequently, when looking for an aggregation of cu's that determine a given ML algorithm whose total compute cost is less than $cc(\mathcal{M})$, one has to look beyond $\widetilde{\mathcal{M}}$. Thus, in order to accommodate restrictions put by limited computer resources, we have identified following types of machines and the corresponding limitations imposed on such aggregation of cu's.

- *Machine with infinite memory and restrictive compute:* With this kind of machine all computed cu's in $\widetilde{\mathcal{M}}$ can be stored in memory to use as and when required. Memory operations do not put overhead but only on the ability to execute cu's. Cost function $cc()$ is in terms of the number of compute cycles required to execute each cu and the ability to execute more than one cu concurrently (i.e., parallelizations using traditional methods). Towards this end we simplify our formulation by assuming that each cu has a uniform compute cycle cost.
- *Machine with infinite compute and restrictive memory:* With this type of machine one ignores the compute cost of executing cu and number of cu's that can be executed simultaneously is infinite. However, once a cu is executed its result must be kept for subsequent use. The system puts an upper bound on memory, in terms of the number of cu's that can be propagated across compute cycles.
- *Machine with restrictive compute and restrictive memory:* This type of a machine combines restrictions from the two previously discussed and is the most realistic.

Now the kind of aggregations of cu's that are suitable for execution on a machine with limitations as discussed above are best described by a new notion given in Definition 2.3.

Definition 2.3. Let $\widetilde{\mathcal{M}} = \{\widetilde{m}_1, \widetilde{m}_2, \dots, \widetilde{m}_{1\overline{0}}\}$ be a set of distinct cu's and $\widetilde{\mathcal{G}}$ a directed acyclic (interaction) graph corresponding to $\widetilde{\mathcal{M}}$. An *ordered list*

$$\mathcal{G} = (\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_\delta), \quad \mathcal{A}_i \subseteq \widetilde{\mathcal{M}}, \quad i = 1, 2, \dots, \delta,$$

is said to be an (r, s) *set cover of $\widetilde{\mathcal{M}}$* , where r and s are positive integers, if for all $i = 1, 2, \dots, \delta$ sets $\mathcal{A}_i$ are nonempty and the following conditions are satisfied:

- i. $\cup_{i=1}^{\delta} \mathcal{A}_i = \widetilde{\mathcal{M}}$,
- ii. cardinality of $\mathcal{A}_i$ is at most r , i.e., $|\mathcal{A}_i| \leq r$, and
- iii. an arbitrary $m_\alpha \in \mathcal{A}_i$ can be executed given a subset, possibly empty, of tasks from $\mathcal{A}_1, \mathcal{A}_{i-1}, \dots, \mathcal{A}_{i-s}$.

If one is to consider $\mathfrak{C}$ in Definition 2.3 as a collection of subsets of $\widetilde{\mathcal{M}}$ instead of an ordered list while satisfying only condition (i), then $\mathfrak{C}$ would be just a *set cover of $\widetilde{\mathcal{M}}$* in the classical sense [14].

A natural question to ask is for what integers r and s $\widetilde{\mathcal{M}}$ has an (r,s) set cover. Clearly since r represents a lower bound on cardinality of $\mathcal{A}_i$'s, it must be that $r \geq 1$. Then it is trivially true that $\mathfrak{C} = (\widetilde{\mathcal{M}})$ is $(1, |\widetilde{\mathcal{M}}|)$ set cover of $\widetilde{\mathcal{M}}$. Furthermore, if $\mathfrak{C}$ is an (r_1, s_1) set cover of $\widetilde{\mathcal{M}}$, then $\mathfrak{C}$ is also an (r_2, s_2) set cover of $\widetilde{\mathcal{M}}$ for all $r_2 \geq r_1$ and $s_2 \geq s_1$. Thinking in terms of r and s as machine's computational and storage capacities, respectively, leads us to the following fundamental problem:

Given $r \geq 1$, what is the smallest s for which (r,s) set cover of $\widetilde{\mathcal{M}}$ exists?

So far in the literature, to our knowledge, only partial solutions to this problem are known. For example, if $r=1$, then this problem relates to the well studied *directed bandwidth problem*, see [2] and references therein. More specifically, for a given graph $H = (V, E)$, where V is set of vertices and E is a set of edges, a *layout of H* is a one-to-one mapping $\sigma: V \rightarrow |V|$. Then the *bandwidth of layout σ with respect to H* , denoted by $Bw(H, \sigma)$, is

$$Bw(H, \sigma) = \max\{|\sigma(u) - \sigma(v)| : (u, v) \in E\}.$$

Finally, the *bandwidth of H* is defined as

$$Bw(H) = \min\{Bw(H, \sigma) : \sigma \text{ is a layout of } H\}. \quad (3)$$

The notion of bandwidth also generalizes to directed graphs, though in more than one way, see [2, Sec. 9] and [9] and references therein. Following the exposition in [2], given a directed acyclic graph $\widetilde{\mathcal{G}}$ associated with a collection of cu's in $\widetilde{\mathcal{M}}$, the *directed bandwidth of $\widetilde{\mathcal{G}}$* , $\overrightarrow{Bw}(\widetilde{\mathcal{G}})$ is given by

$$\begin{aligned} \overrightarrow{Bw}(\widetilde{\mathcal{G}}) = \min\{Bw(\widetilde{\mathcal{G}}, \sigma) : \sigma \text{ is a layout of } \widetilde{\mathcal{G}} \text{ and} \\ \sigma(m_\alpha) < \sigma(m_\beta) \text{ for all directed arcs from } m_\alpha \text{ to } m_\beta\}. \end{aligned}$$

It turns out that given $r=1$, (r,s) set cover of $\widetilde{\mathcal{M}}$ exists for all $s \geq \overrightarrow{Bw}(\widetilde{\mathcal{G}})$, where the inequality is sharp. Determining $\overrightarrow{Bw}(\widetilde{\mathcal{G}})$ is generally an NP-complete problem and is closely related to the DAG scheduling problem [10]. On the other hand, determining if the graph's bandwidth is bounded above by a fixed/constant number k can be done in polynomial time [12].

On the other hand, if $r > 1$ we might be interested in the smallest s , let's denote it by $\hat{s}$, for which (r,s) set cover of $\widetilde{\mathcal{M}}$ exists. To get an upper bound on $\hat{s}$, we start with a layout that realizes $\overrightarrow{Bw}(\widetilde{\mathcal{G}})$ and then group the first r cu's together, and the next r , and the next r , and so on. In this case, the maximum difference in labels of adjacent cu's would either be the ceiling of $\overrightarrow{Bw}(\widetilde{\mathcal{G}})/r$ or one less than that. Further analysis is required to determine if this is the

optimal $\hat{s}$ for a given computational method, e.g., SIMD, MIMD, on a specific compute device, which is beyond the scope of the current discussion.

2.4 Step 4: FTiP algorithm in case of infinite memory and restricted r

Examples in Figure 3 discuss deterministic scenarios. However, in reality, machine learning algorithms are much more difficult, e.g., Figure 7 shows part of an interaction graph constructed from cu's in a k -means clustering algorithm. Executing abstracted computational units from the ensemble learning algorithm's redundant computations involves accounting for spatial and temporal dependencies in the cu's. We can get ordering of the cu's from Section 2.3. We can also get the value of s , the minimum memory size for a machine that can accommodate the efficient execution of a given ensemble ML algorithm, but executing them efficiently poses a further challenge.

Computing computational units on a machine with a compute capability of r per cycle is an optimization question. Inspired by the vastly studied VLSI (Very Large Scale Integration) field, we are exploring methods to adopt techniques that aggregate cu's into conceptual sets of floors. More formally, for a given ordering of computational units of different weights representing their computing cost, we have to find an aggregation of cu's as disjoint sets, where each set's total compute cost is at most r with or without unbounded memory.

Finding minimum bandwidth for the scenarios where computational units do not have uniform cost is a much more difficult problem. Further insights on an FTiP function based on DP (Dynamic Programming) and scenarios for weighted bandwidth set cover are topics for future discussion.

3 SIMULATION STUDY

Current compilation techniques usually treat operations on data to be dynamic or data agnostic. This treatment hides inherent redundancies that exist when the exact computation is performed on the same data value in various places of an EM. Furthermore, analogous redundancies exist in other algorithms in an ensemble when computational steps are viewed across boundaries defined by programming languages. In support of our preliminary work of this paper, we aim to uncover the patterns of these redundancies through a simulation study. We start by considering the k -means algorithm described by the pair $(\mathcal{M}, \mathcal{G})$ (see Section 2.2). Now let $(\mathcal{M}_a, \mathcal{G}_a)$ be a pair that describes the same k -means algorithm. For $(\mathcal{M}_a, \mathcal{G}_a)$, we define the notion of *RR* (redundancy reduction) *factor* as

$$RR(\mathcal{M}_a) = \frac{|\mathcal{M}| - |\mathcal{M}_a|}{|\mathcal{M}|}.$$

Clearly, $\min(RR) = 0$ when $\mathcal{M}$ has no redundancies and $\max(RR)$ is attained when $\mathcal{M}_a = \widetilde{\mathcal{M}}$.

3.1 Experimental Setup

As a tool to track repeated patterns in computational steps of various clustering algorithms used in EM, we created a simulator called FTiPSim [6]. FTiPSim is used to analyze patterns

across local, global, single statement, code blocks, and functions over multiple invocations. The simulator starts by producing the pair $(\mathcal{M}, \mathcal{G})$, where $\mathcal{G}$ describes the interdependencies among analysed cu's, keeps track of the number of cu's, number of dependencies, and computes the minimum bandwidth of $\mathcal{G}$. This treatment closely follows the design in Section 2. In Figure 7(a), we give an example of a graph $\mathcal{G}$ resulting from the partial computation steps of a k -means clustering algorithm at an instance of our simulation. Moreover, Figures 7(b)–(c) contain dependency graphs $\mathcal{G}_a$ and $\widetilde{\mathcal{G}}$ corresponding to partially and fully removed redundancies, $\mathcal{M}_a$ and $\widetilde{\mathcal{M}}$, respectively.

From Figure 7, as expected, we see that eliminating redundancies results in the decreased number of vertices (computational units) and edges (dependencies). However, having a smaller dependency graph $\mathcal{G}_a$ does not guarantee improvements in faster computation of $(\mathcal{M}_a, \mathcal{G}_a)$ or less resource consumption. To address this, we consider an additional measure of $\mathcal{G}_a$, namely its minimum bandwidth, and assess its relation to $RR(\mathcal{M}_a)$.

In Figure 8 we exhibit the above metrics of the pair $(\mathcal{M}, \mathcal{G})$ corresponding to different variations in data, i.e., $|\mathcal{D}^{(t)}| = 8, 16, 32$ while keeping β constant (see Section 2.1). Finally, we calculate the bandwidth of each dependency DAG by adapting the algorithms of Saxe [12] and Gurari & Sudborough [8].

3.2 Experimental Results

Experimental results are furnished in Figure 8; in which chart title indicates the measuring metric. For each y-axis is the actual measure and x-axis represents $\mathcal{M}$, $\mathcal{M}_a$, and $\widetilde{\mathcal{M}}$ for $|\mathcal{D}^{(t)}| = 8, 6, 32$. Our results show that for an ML algorithm described by $(\mathcal{M}, \mathcal{G})$ and its optimized $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{G}})$, the minimum bandwidths of $\mathcal{G}$ and $\widetilde{\mathcal{G}}$ both have an upper and lower bounds. Moreover, for the pair $(\mathcal{M}_a, \mathcal{G}_a)$ that encapsulates the same ML algorithm, we have $RR(\mathcal{M}) \leq RR(\mathcal{M}_a) \leq RR(\widetilde{\mathcal{M}})$. The fact that the bandwidth of $\mathcal{G}_a$ depends on $RR(\mathcal{M}_a)$ suggests alternate execution approaches of $\mathcal{G}_a$, and when mapped to a computational plane, there are different floor tile plans.

Methods used to eliminate redundancy in optimizing compilers are clear evidence of a need for specialized implementation of the compiler technology stack for the commonly occurring scenarios in decision science applications. Currently, compilers only consider redundancy in limited scope. They also do not tend to treat data values as an essential component in redundancy elimination. Our approach gives data values equal weight alongside variables holding these values and operations performed using them. We are currently investigating different variations of algorithms in EM and apply the same framework to other machine learning algorithms. Ultimately, these changes will enable increased use of ensemble ML algorithms for reduced computation time and/or resources, leading to better aid in making well-informed (potentially automated) decisions.

4 CONCLUSION & FUTURE WORK

The work discussed in this paper is an extension of our previous efforts in improving the computational performance of the ensemble clustering ML algorithm used to study decisions under uncertainty on data collected for the Iowa Gambling Task. In general, while most of the prior efforts try to improve a specific clustering technique, we proposed a parallel implementation for all phases of the ensemble clustering ML algorithm for shared [4] and distributed [5] memory systems. For additional details on comparison with other parallel implementations see [3].

In our present work, we explored avenues for further improvements in computational performance in EM algorithms. We identify our work at the intersection of high-performance computation, compiler, and reinforcement learning algorithms and thus also refer the reader to a few most relevant publications by others. In [13], Song and coauthors have proposed an algorithm called GDRA to account for redundancies when using multiprocessor systems. Gordon et al. discussed compiler optimizations at a task level for data parallelism in stream applications [7]. For a more generic research on eliminating resource redundancy see [15] and the references

We propose a novel theoretical framework which can help us identify previously undetected redundancies in ensemble techniques commonly used in data analytics. We define the notion of (r,s) set cover that allows one to attack the problem using graphs and linear programming approaches. In this paper, we solve the RRC problem for a restricted case, in which machine size allows a limited compute capacity but unbounded memory that can store partial results for reuse. We assume data to be known at compile time and thereby part of the program code. We also discuss the applicability of the minimum bandwidth problem to find an (r,s) set cover. Memory requirement defined by s in our formulations is used to determine memory size for a given machine and workload.

ACKNOWLEDGMENTS

The second author was partially supported by the National Institute of General Medical Sciences of the National Institutes of Health under Award Number R15GM120820. The content is solely the author's responsibility and does not necessarily represent the official views of the NIH. The third author was partially supported by the Career Enhancement Grant through the University of Rhode Island.

The authors thank Bill Kinnersley for discussion regarding the directed bandwidth problem.

REFERENCES

- [1]. Amdahl GM 1967. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In Proceedings of the April 18–20, 1967, Spring Joint Computer Conference (AFIPS '67) Association for Computing Machinery, New York, USA, 483–485.
- [2]. Garey MR, Graham RL, Johnson DS, and Knuth DE 1978. Complexity Results for Bandwidth Minimization. SIAM J. Appl. Math 34, 3 (1978), 477–495.
- [3]. Gavirangaswamy VB 2018. High Performance Computing Techniques for Analyzing Risky Decision Making PhD Dissertation, Western Michigan University. (2018).
- [4]. Gavirangaswamy VB and Gupta A 2017. CPU-GPU implementation of ensemble clustering algorithm for increased performance. In 2017 Int. Conf. on Intelligent Communication and Computational Techniques (ICCT) 262–271.

- [5]. Gavirangaswamy VB, Gupta A, and Gupta A 2016. A parallel implementation of reinforced learning model used in analyzing risky decision making. In 2016 Int. Conf. on High Performance Computing Simulation (HPCS) 940–946.
- [6]. B Gavirangaswamy V and Saleh HM 2021. FTiPSim- Floor Tile Planning Simulator https://github.com/vinaybabug/ftip_simulator.
- [7]. Gordon MI, Thies W, and Amarasinghe S 2006. Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs. SIGOPS Oper. Syst. Rev 40, 5 (2006), 151–162.
- [8]. Gurari EM and Sudborough IH 1984. Improved dynamic programming algorithms for bandwidth minimization and the mincut linear arrangement problem. Journal of Algorithms 5, 4 (1984), 531–546.
- [9]. Jain P, Kanesh L, Lochet W, Saurabh S, and Sharma R 2019. Exact and Approximate Digraph Bandwidth. In 39th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 150), Arkadev Chattopadhyay and Paul Gastin (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 18:1–18:15.
- [10]. Kwok Y-K and Ahmad I 1999. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. ACM Comput. Surv 31, 4 (1999), 406–471.
- [11]. Moore G 1975. Progress In Digital Integrated Electronics. In IEEE International Electron Devices Meeting 11–13.
- [12]. B Saxe J 1980. Dynamic-programming algorithms for recognizing small-bandwidth graphs in polynomial time. SIAM Journal on Algebraic Discrete Methods 1, 4 (1980), 363–369.
- [13]. Song I, Yoon W, Jang E, and Choi S 2011. Task Scheduling Algorithm with Minimal Redundant Duplications in Homogeneous Multiprocessor System. In Grid and Distributed Computing, Kim Tai-hoon, Adeli Hojjat, Cho Hyun-seob, Gervasi Osvlado, Yau Stephen S., Kang Byeong-Ho, and Villalba Javier García (Eds.). Springer Berlin Heidelberg, Berlin, 238–245.
- [14]. Vazirani VV 2001. Approximation Algorithms Springer-Verlag, Berlin, Heidelberg.
- [15]. Zhao L, Ren Y, and Sakurai K 2013. Reliable workflow scheduling with less resource redundancy. Parallel Comput 39, 10 (2013), 567–585.

CCS CONCEPTS

• **Theory of computation** → **Models of computation**; *Design and analysis of algorithms*, • **Computing methodologies** → **Machine learning**; *Parallel computing methodologies*.

Author Manuscript

Author Manuscript

Author Manuscript

Author Manuscript

Figure 1:

Highlighted boxes denote an example of steps in ensemble clustering that lead to redundant computations.

Original Machine Assembly Code w/ Color Coding for Redundancies	An Example Optimized Assembly Code
<pre> Omitted code ... mov DWORD PTR [rbp-52], 4 mov DWORD PTR [rbp-56], 0 mov DWORD PTR [rbp-60], 0 mov edx, DWORD PTR [rbp-52] mov eax, DWORD PTR [rbp-24] mov esi, edx mov edi, eax call subtract(int, int) mov ebx, eax mov edx, DWORD PTR [rbp-52] mov eax, DWORD PTR [rbp-24] mov esi, edx mov edi, eax call subtract(int, int) mov esi, ebx mov edi, eax call multiply(int, int) mov ebx, eax mov edx, DWORD PTR [rbp-48] mov eax, DWORD PTR [rbp-20] mov esi, edx mov edi, eax call subtract(int, int) mov r12d, eax mov edx, DWORD PTR [rbp-48] mov eax, DWORD PTR [rbp-20] mov esi, edx mov edi, eax call subtract(int, int) mov esi, r12d mov edi, eax call multiply(int, int) mov esi, ebx mov edi, eax call add(int, int) mov DWORD PTR [rbp-56], eax mov edx, DWORD PTR [rbp-52] mov esi, DWORD PTR [rbp-24] mov edi, eax call subtract(int, int) mov edx, eax mov eax, DWORD PTR [rbp-32] mov esi, eax mov edi, edx call add(int, int) mov ebx, eax mov esi, DWORD PTR [rbp-52] mov edi, eax call subtract(int, int) mov edx, eax mov esi, DWORD PTR [rbp-48] mov edi, eax call add(int, int) mov r12d, eax mov edx, DWORD PTR [rbp-48] mov esi, DWORD PTR [rbp-20] mov edi, esi call subtract(int, int) mov edx, eax mov esi, DWORD PTR [rbp-28] mov esi, eax mov edi, edx call add(int, int) mov r12d, eax mov edx, DWORD PTR [rbp-48] mov esi, DWORD PTR [rbp-20] mov edi, esi call subtract(int, int) mov edx, eax mov esi, DWORD PTR [rbp-48] mov esi, esi call multiply(int, int) mov esi, esi mov edi, eax call add(int, int) mov DWORD PTR [rbp-60], eax mov eax, 0 add rsp, 48 pop rbx pop r12 pop rbp ret </pre>	<pre> Omitted code ... mov DWORD PTR [rbp-52], 4 mov DWORD PTR [rbp-56], 0 mov DWORD PTR [rbp-60], 0 <stored result D, Computational Unit D> mov ebx, eax <stored result D, Computational Unit C> <stored result F, Computational Unit F> mov ebx, eax <stored result B, Computational Unit A> mov r12d, eax <stored result B, Computational Unit A> <stored result F, Computational Unit F> <stored result G, Computational Unit G> mov DWORD PTR [rbp-56], eax <stored result D, Computational Unit K> mov edx, eax <stored result G, Computational Unit O> mov ebx, eax <stored result D, Computational Unit K> mov edx, eax <stored result G, Computational Unit N> <stored result Q, Computational Unit Q> mov ebx, eax <stored result B, Computational Unit I> mov edx, eax <stored result M, Computational Unit M> mov r12d, eax <stored result B, Computational Unit H> mov edx, eax <stored result M, Computational Unit L> <stored result P, Computational Unit P> <stored result N, Computational Unit A> mov DWORD PTR [rbp-60], eax mov eax, 0 add rsp, 48 pop rbx pop r12 pop rbp ret </pre>

Figure 2:

Example demonstrating redundancies in code due to numerical operations applied over same data, over separate spatial and temporal dependencies.

Scenario 1: Sequential execution of cu's one after another

Scenario 2: Sequential execution of cu's, with $A \cap B(S_5) = 24$ redundancy

Figure 3:

Example scenarios differentiating sequential and redundant execution plans.

$$\begin{array}{l}
\left. \begin{array}{l} m_1^{6,t} = a_1 - c_1 \\ m_2^{6,t} = a_1 - c_1 \end{array} \right\} \longrightarrow m_3^{6,t} = m_1^{6,t} \cdot m_2^{6,t} \\
\left. \begin{array}{l} m_4^{6,t} = a_2 - c_2 \\ m_5^{6,t} = a_2 - c_2 \end{array} \right\} \longrightarrow m_6^{6,t} = m_4^{6,t} \cdot m_5^{6,t} \end{array} \left\} \longrightarrow m_7^{6,t} = m_3^{6,t} + m_6^{6,t}
\right.$$

$$\left. \begin{array}{l} m_7^{6,t} = a_1 - c_1 \\ m_8^{6,t} = a_1 - c_1 \end{array} \right\} \longrightarrow m_1^{8,t} = m_7^{6,t} + b_1 \\
\left. \begin{array}{l} m_9^{6,t} = a_2 - c_2 \\ m_{10}^{6,t} = a_2 - c_2 \end{array} \right\} \longrightarrow m_2^{8,t} = m_8^{6,t} + b_1 \end{array} \left\} \longrightarrow m_5^{8,t} = m_1^{8,t} \cdot m_2^{8,t}
\right.$$

$$\left. \begin{array}{l} m_9^{6,t} = a_2 - c_2 \\ m_{10}^{6,t} = a_2 - c_2 \end{array} \right\} \longrightarrow m_3^{8,t} = m_9^{6,t} + b_2 \\
\left. \begin{array}{l} m_7^{6,t} = a_1 - c_1 \\ m_8^{6,t} = a_1 - c_1 \end{array} \right\} \longrightarrow m_4^{8,t} = m_{10}^{6,t} + b_2 \end{array} \left\} \longrightarrow m_6^{8,t} = m_3^{8,t} \cdot m_4^{8,t}
\right.$$

$$\longrightarrow m_7^{8,t} = m_5^{8,t} + m_6^{8,t}$$

Figure 4:
Nontrivial cu's with repetitions in Dist-Program

Original cu	Relabeled cu	Dependency
$m_1^{6,t} = a_1 - c_1$	$\tilde{m}_1$	none
$m_3^{6,t} = m_1^{6,t} \cdot m_2^{6,t} = m_1^{6,t} \cdot m_1^{6,t}$	$\tilde{m}_2$	$\tilde{m}_1$
$m_4^{6,t} = a_2 - c_2$	$\tilde{m}_3$	none
$m_6^{6,t} = m_4^{6,t} \cdot m_5^{6,t} = m_4^{6,t} \cdot m_4^{6,t}$	$\tilde{m}_4$	$\tilde{m}_3$
$m_7^{6,t} = m_3^{6,t} + m_6^{6,t}$	$\tilde{m}_5$	$\tilde{m}_2, \tilde{m}_4$
$m_1^{8,t} = m_7^{6,t} + b_1 = m_1^{6,t} + b_1$	$\tilde{m}_6$	$\tilde{m}_1$
$m_5^{8,t} = m_1^{8,t} \cdot m_2^{8,t} = m_1^{8,t} \cdot m_1^{8,t}$	$\tilde{m}_7$	$\tilde{m}_6$
$m_3^{8,t} = m_9^{6,t} + b_2 = m_4^{6,t} + b_2$	$\tilde{m}_8$	$\tilde{m}_3$
$m_6^{8,t} = m_3^{8,t} \cdot m_4^{8,t} = m_3^{8,t} \cdot m_3^{8,t}$	$\tilde{m}_9$	$\tilde{m}_8$
$m_7^{8,t} = m_5^{8,t} + m_6^{8,t}$	$\tilde{m}_{10}$	$\tilde{m}_7, \tilde{m}_9$

Figure 5:
Nontrivial cu's without repetitions in Dist-program

Figure 6:
Interaction Graph for Figure 5

Figure 7:

DAGs resulting from cu's in $\mathcal{M}$, $\mathcal{M}_a$ and $\widetilde{\mathcal{M}}$ with $RR(\mathcal{M}) \leq RR(\mathcal{M}_a) \leq RR(\widetilde{\mathcal{M}})$.

Figure 8:

Metrics on DAGs resulting from cu's in $\mathcal{M}$, $\mathcal{M}_a$ and $\tilde{\mathcal{M}}$ for $|D^{(t)}| = 8, 16, 32$; $\mathcal{M}_a$ corresponds to a variation in RR.