ACE: Algorithm-independent Acceleration and Parallelization of Clustering Implementations

Muyeed Ahmed and Iulian Neamtiu

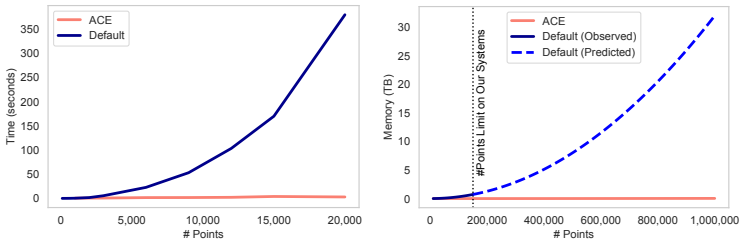
New Jersey Institute of Technology, Newark, NJ, USA
{ma234,ineamtiu}@njit.edu

Abstract. Clustering is a key technique in a wide range of data analysis tasks. However, algorithms that ensure stable, deterministic, accurate clustering are computationally expensive, having superlinear complexity for both memory and time. Therefore, even when using HPC hardware, there are hard limits to dataset sizes that can be clustered, as clustering implementations can run out of memory or take unacceptably long. We introduce an approach called ACE that applies *algorithm-independent, black-box parallelization* to superlinear sequential clustering algorithms, thereby making the clustering of substantial datasets feasible, even on commodity desktop/laptop systems. ACE starts by partitioning data to fit onto a given machine, and via divide-and-conquer, reduce the complexity of clustering steps. Next, ACE uses parallel, automated hyperparameter search to find optimal parameters for the current dataset. Finally, ACE aggregates intermediate results effectively and efficiently so that the final clustering output does not sacrifice clustering quality compared to the original algorithm. An evaluation on four popular clustering algorithms – Affinity Propagation, DBSCAN, Hierarchical Agglomerative Clustering, and Spectral Clustering – shows that ACE substantially reduces memory requirements and achieves linear processing time. ACE was able to process an entire suite of 164 datasets, including substantial datasets with 1.4M points or 1,000 dimensions, whereas the default implementations failed to process between 15 and 149 datasets from the suite. Moreover, for those datasets that could be processed by the default implementations, ACE achieved a 1.13x–102x time reduction.

Keywords: Machine Learning, Clustering, Automatic Parallelization

1 Introduction

Clustering, or cluster analysis, is an unsupervised learning technique for grouping similar objects (represented as points in a multidimensional space) into groups called *clusters*. Clustering is appealing as it does not require labels or ground truth, and is used in a wide range of fields, e.g., medical, finance, or manufacturing. A fundamental obstacle to using precise and stable clustering algorithms such as DBSCAN, Affinity Propagation, or Hierarchical Agglomerative Clustering, is their computational complexity, e.g., $O(n * log\, n)$–$O(n^3)$ time and $O(n^2)$



(a) AP: time comparison on dataset numer.ai.

(b) SC: memory usage comparison for all datasets.

Fig. 1. Time and memory comparison of default vs. ACE.

memory. As datasets continue to grow in size and complexity (number of points n, number of dimensions d), computational resources become severely inadequate, with even "resource frugal" algorithms unable to process commonly used datasets (100,000–1,000,000 points) on contemporary commodity hardware. In contrast, ACE clusters datasets with $n > 1M$ and $n*d = 100M$ in 6–40 minutes.

Prior efforts on accelerating clustering have focused on data parallelization (data/feature partitioning), white-box approaches (e.g., algorithm-specific parallelization), and hardware acceleration [6, 16, 17, 22, 3, 21]. These techniques share three main disadvantages: they require source code, are labor-intensive, and, being designed for specific algorithms and implementations, do not generalize.

To address these issues, we introduce ACE ("Algorithm-independent Clustering AccEleration and paralellization"). ACE takes a sequential, resource-intensive clustering "task" (clustering implementation and input dataset) and turns it into a runtime-efficient, memory-thrifty, parallel task. ACE parallelizes implementations regardless of the underlying algorithm's structure, and without requiring access, or changes, to source code.

Availability. We made ACE public: https://github.com/Anonymous-User-7/ACE

To motivate our approach, we illustrate the lack of scalability on two algorithms, Affinity Propagation (AP) and Spectral Clustering (SC): when using default implementations, runtime and memory requirements can quickly get out of hand even for moderately-sized datasets.

Runtime. To assess the impact of dataset size on runtime, we ran AP on subsets with sizes ranging from 1,000 to 20,000 data points, extracted from the numer.ai [1] dataset. Figure 1(a) illustrates how runtime grows superlinearly with dataset size. For example, when clustering a 5,000-point subset, the algorithm executed in 18.35 seconds. However, when the subset size doubled (10,000 points), time went up almost 4x, to 71.68 seconds. The algorithm took 10,611 seconds (≈3 hours), to cluster the entire 96,320-point dataset (148x increase in time for a 9.6x size increase). This superlinear runtime increase with set size highlights the challenge presented by large datasets. In comparison, ACE completed *in just* 12.6 *seconds, i.e., a speedup of* 842x, and the increase in runtime w.r.t. the number of points n is linear with a very low slope, rather than superlinear.

Memory. Figure 1(b) illustrates superlinear memory demands and the ensuing severe limitations on dataset size. We show the minimum memory required for Spectral Clustering (SC) as the number of points varies from 10,000 to 1,000,000. The superlinear memory demand is apparent from the figure: even for 10,000 points, SC requires a minimum of 320GB of memory, but this requirement escalates to 32TB when processing a dataset containing 1,000,000 data points (1M for short). *In contrast,* ACE *could seamlessly fit the 1M clustering task onto commodity desktop/laptop systems or typical datacenter nodes.* With a maximum partition size set to 1,000 ACE only requires 32MB+ε memory for each partition. ACE automatically adjusts the number of concurrently processed partitions to fit into, and efficiently occupy, the machine's physical memory.

In terms of memory, on a 1M dataset, Hierarchical Agglomerative Clustering (HAC) would require more than *16TB* (Section 4.1), whereas ACE uses less than 20GB. Even assuming access to an HPC system with more than 16TB of memory, Affinity Propagation (AP) would take more than 14 days on a 1M dataset. In contrast, when running ACE on 117 datasets that exceeded 1M points, the average runtime on our slowest system was less than 14 minutes.

Our experiments demonstrate that ACE is particularly effective for memory- and time-intensive algorithms such as AP and SC. These algorithms' default implementations failed to cluster 136 and 139 datasets, respectively (out of 164) due to running out of memory, whereas ACE successfully ran all 164 and proved to be 14x–102x faster. While HAC's implementation was significantly faster than AP/SC, ACE was able to further improve its efficiency, by 6x and allowed HAC to run on all datasets whereas the default implementation failed to run on 132–136 datasets due to unmet memory demands. Finally, DBSCAN ran out of memory for 2–15 datasets (depending on system), whereas ACE clustered all datasets, and improved DBSCAN's performance by about 2x. ACE supports any type of clustering algorithm and is compatible with any numerical dataset.

In Section 2, we present our experimental setup. In Section 3, we discuss ACE's design and implementation. We evaluate ACE on the four algorithms, in terms of runtime and memory (Section 4) as well as accuracy (Section 5). In summary, this paper makes the following contributions:

- An algorithm-independent tool, ACE, enabling parallelization of clustering tasks without accessing or altering the source code of underlying algorithms.
- An approach for auto-tuning partition size and hyperparameters, and an approach for merging clusters from different partitions.
- Experimental results confirming that ACE achieves substantial reductions in runtime and memory usage.

2 Experimental Setup

Algorithms. We studied four clustering algorithms, Affinity Propagation (AP) [10], Density-Based Spatial Clustering (DBSCAN) [9], Hierarchical Agglomerative Clustering (HAC) [4] and Spectral Clustering (SC) [20], as implemented in the Scikit-learn toolkit.

Table 1. System Configurations

System	CPU	Frequency (GHz)	Cores	RAM (GB)	Swap (GB)
Sys1	Mac M2 Pro	3.5GHz	12	32	≤1,000
Sys2	Intel Xeon E5-2697	2.6GHz	14	64	128
Sys3	Intel Xeon W-2145	3.7GHz	8	256	512
Sys4	Intel Core i7-6950X	3.0GHz	10	128	256

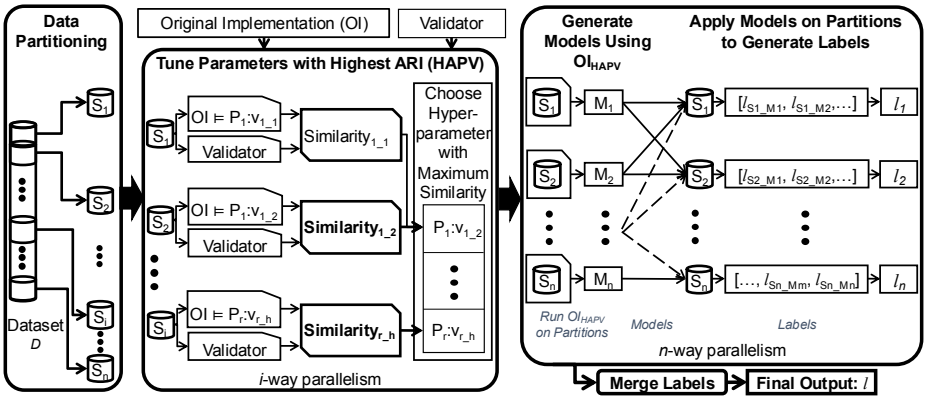


Fig. 2. ACE architecture.

Datasets. We used 164 datasets from OpenML [2], with a number of points varying between 10,000–1,455,525. These datasets come from a wide range of domains, e.g., medical, engineering, finance, social media. On average, datasets have 507,468 instances and 28 attributes. We used classification datasets as their labels allow us to gauge accuracy; the number of clusters ranged from 2 to 1,454.

Measuring Clustering Quality. For measuring clustering similarity or accuracy, we use the Adjusted Rand Index (ARI) [13]. ARI is an intuitive and effective scalar metric for clustering similarity: given two clusterings U and V of the same dataset, $ARI(U, V)$ can range from -1 (strongest disagreement) to $+1$ (perfect agreement). To gauge accuracy of a clustering C, we measure $ARI(C, G)$ where G is the ground truth (classification labels). We also used the Silhouette Score [19] to measure clustering quality, with values ranging from -1 to 1 (higher values indicate clearer, more compact clusters).

Hardware Configurations. For our experiments we used four hardware configurations, detailed in Table 1. In the remainder of the paper we will refer to these systems as *Sys1* through *Sys4*. The systems span the gamut from a "pro" laptop (Apple MacBook with an M2 Pro chip, 32GB RAM), to a workstation (Intel i7-6950x, 128GB RAM), and typical data center servers, e.g., 8–14 core Intel Xeon servers with 64–256 GB RAM.

3 ACE Architecture

Figure 2 shows ACE's architecture. We designed ACE to operate in a black-box (no source code required) and algorithm-independent manner: ACE only requires the black-box implementation and parameter information (parameter names and their ranges). ACE takes the original implementation OI of a clustering algorithm and dataset D as inputs, and produces l (i.e., a clustering of D) as output. ACE consists of four main parts. First, ACE divides the dataset D into multiple chunks or partitions ($[S1, S2, \ldots]$). Next, ACE determines the parameter values with highest accuracy (HAPV) for D through parallel execution with different parameter settings. Then, ACE concurrently runs the algorithm on each partition using the HAPV. Finally, ACE merges the output of the individual data partitions ($[l_1, l_2, \ldots, l_n]$), and returns the clustering output (l). The average time ACE spent at each stage is shown in the following table.

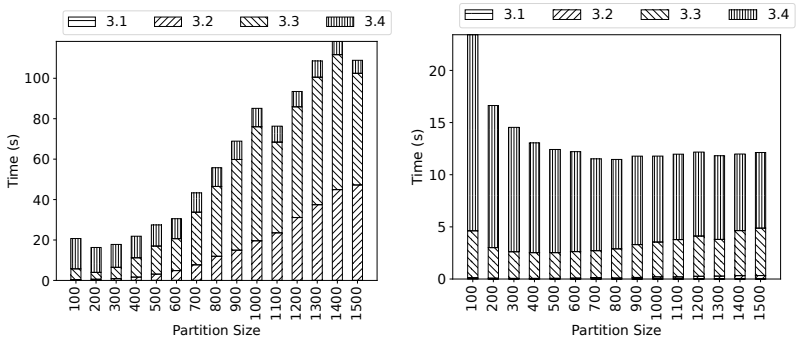
	AP	DBSCAN	HAC	SC
3.1 Data Partitioning	0.00088%	0.15%	0.029%	0.00014%
3.2 Tuning Parameters for HAPV	23.02%	15.24%	1.71%	87.11%
3.3 Generating Labels	66.27%	13.68%	28.35%	11.5%
3.4 Merging Labels	10.7%	70.93%	69.91%	1.44%

Note the different time balances, e.g., merging takes most (70%) of the time for DBSCAN and HAC, while labeling and HAPV dominate in AP and SC, respectively. These findings underscore ACE's adaptability to various algorithms.

3.1 Data Partitioning

ACE starts by partitioning[1] the dataset, which (1) decomposes a superlinear task via divide-and-conquer, (2) enables parallelization of a sequential task, and (3) limits resource use. Partitioning is done after shuffling the order in which the points appear in the input file, to reduce bias and ensure uniformity. Partitioning is a balancing act between efficiency and memory: a small partition size reduces memory demands, but might increase running time. We illustrate this trade-off in Figure 3, showing how runtime varies with partition size, and how long each of ACE's four stages takes. Figure 3(a) shows this distribution for the AP algorithm while Figure 3(b) shows the distribution for HAC (DBSCAN is similar to HAC, and SC is similar to AP). In both cases, the first phase, data partitioning, takes less than 1% of the overall time hence is not discernible. The time for the next stage, parameter tuning, increases linearly, albeit with a steeper slope for AP; label generation shows a similar behavior. The final stage, merging, takes a similar time for both algorithms across varying partition sizes. The difference between the two figures arises from AP taking more time to generate labels, compared to HAC. For both algorithms, with small partition sizes, the merging time dominates; however, as partition sizes increase, parameter tuning and label generation times dominate. While users can tailor the partition

[1] Where partitioning is defined as expected: no partitions share a common point, i.e., $S_a \cap S_b = \emptyset$; and the partitions cover the whole dataset D, i.e., $S_1 \cup S_2 \cup \ldots \cup S_n = D$.



(a) AP (b) HAC

Fig. 3. Runtime of ACE's four stages (3.1: Data Partitioning, 3.2: Tuning Parameters for HAPV, 3.3: Generating Labels, 3.4: Merging Labels) with varying partition sizes.

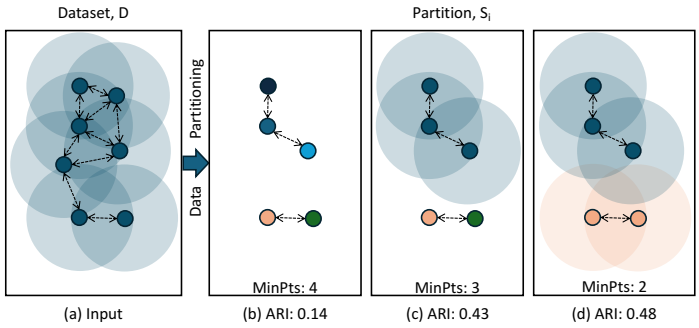


Fig. 4. Automatic parameter tuning via HAPV: improving clustering outcome by adjusting *MinPts* value post-partitioning. *MinPts=2* is chosen as it has the highest ARI.

size to the specific algorithm employed, note that a wide range of partition sizes, 300–1,000, works well across all algorithms. Users can set a fixed partition size (our experiments used 1,000 for all algorithms) or let ACE determine the partition size automatically. Note that even if the chosen partition size is not the optimal size for that algorithm, we are still within a constant factor of the optimal; in contrast, the default implementation might not even run, or could run orders of magnitude slower.

3.2 Parallel Auto-tuning of Parameters for Highest Accuracy

In this stage, ACE conducts a parallel search for parameter values with highest accuracy, named HAPV. We introduce our new HAPV technique, rather than relying on existing optimization strategies such as Grid Search, Random Search, and Bayesian Optimization because they struggle in unsupervised learning scenarios, due to a lack of clear performance metrics and ground truth. Figure 4

illustrates the HAPV process on a DBSCAN parameter. Consider a dataset D, with C_1 being one of its clusters. As ACE runs DBSCAN on partition S_i, one of the hyperparameters it adjusts is *MinPts*, controlling the minimum neighborhood size. Figures 4 (b)-(d) show the outcomes of using three different *MinPts* values and their ARI with respect to the validator. ACE will select *MinPts=2* for HAPV, as it yields the maximum ARI.

Algorithm 1 shows ACE's HAPV pseudocode. First, ACE selects a parameter P_1 with h possible values and runs the algorithm C's original implementation (OI) with different values of P_1 ($[v_{1\_1}, v_{1\_2}, \ldots, v_{1\_h}]$) on h different partitions $[S_1, S_2, \ldots, S_h]$, generates their output labels $[l_{1\_o}, l_{2\_o}, \ldots, l_{h\_o}]$, and calculates the silhouette score. At the same time, ACE runs a *validator*, i.e., another clustering algorithm C_k on the same partitions, with output labels $([l_{1\_k}, l_{2\_k}, \ldots, l_{h\_k}])$. ACE then calculates the ARI of the two outputs and selects the parameter value with maximum ARI ($max([ari(l_{1\_o}, l_{1\_k}), ari(l_{2\_o}, l_{2\_k}), \ldots, ari(l_{m\_o}, l_{h\_k})])$) and has a silhouette score greater than 0, and sets it as the final value for P_1. In parallel, ACE explores parameter P_2's j possible values on the next j partitions, i.e., $[S_{h+1}, S_{h+2}, \ldots, S_{h+j}]$, and selects the best value for P_2. ACE continues the same process for all the other parameters and eventually chooses parameter settings that maximize the accuracy for the dataset. Currently, our HAPV algorithm does not account for coupling effects; we leave automatic discovery of coupling effects to future work. By default, ACE uses KMeans as validator C_k, based on a statistical analysis[2] though users can specify a different clustering algorithm as validator.

Algorithm 1 Determining HAPV

```
Input:
    Data Partitions S = {S_1, S_2, ..., S_n}, Implementation OI, Validator C_k,
    ParameterValues = r parameters, each with h values,
procedure TUNE_HAPV(S, OI, ParameterValues, C_k)
  for all parameters P_i where i = 1, ..., r do
    P_iValues = {v_i_1, v_i_2, ..., v_i_h}
    for all v_i_j in P_iValues doinparallel
      l_d_o = fit OI |= P_i : v_i_j on S_d
      l_d_k = fit C_k on S_d
      ari_i_j = ARI(l_d_o, l_d_k)
      silh_i_j = silhouette_score(l_d_o, S_d)
    end for
    Set the value P_i in HAPV to v_i_j with maximum ari_i_j and silh_i_j > 0
  end for
  Return HAPV
end procedure
```

[2] We used KMeans, AP, HAC, and DBSCAN as validators, and ran pairwise two-means t-tests on their outcomes' accuracy distributions. The p-values indicated that KMeans, AP, and HAC achieve the same effectiveness as validators, with only DBSCAN exhibiting inferior effectiveness.

3.3 Generating Models and Applying on Partitions

Figure 2 shows how ACE generates labels for each partition. In this step, ACE applies the clustering algorithm C using the HAPV-determined parameter values to each partition S_i, and identifies clusters within them. This process is executed concurrently, with clustering application on each partition being performed in parallel. The output consists of n lists of clustering labels ($[l_1, l_2, ...l_n]$), where each list contains the clustering labels within its respective partition.

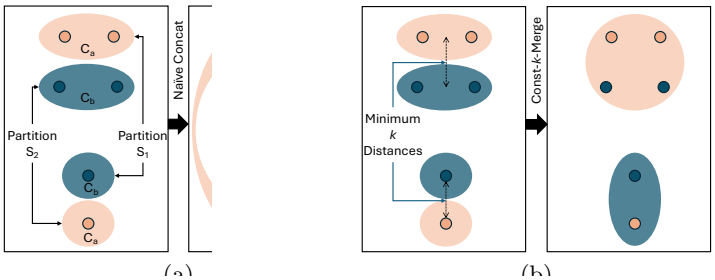
We have introduced an additional sub-step for incremental clustering algorithms, e.g., AP, KMeans, Gaussian Mixture Model (GMM). This involves utilizing the knowledge gained from clustering in one partition to assist in refining the clustering of another partition. Each partition S_i generates a model M_i. These trained models ($[M_1, M_2, ...M_n]$) are then applied to all partitions. For example, applying model M_i on partition S_1 will produce labels $l_{S_1\_M_i}$. Applying all n models on S_1 will yield n sets of labels ($[l_{S_1\_M_1}, l_{S_1\_M_2}, ..., l_{S_1\_M_n}]$). Note that these sets of labels represent various labels assigned to the same partition. These labels are consolidated by selecting, for each data point, the clustering label that appears most frequently across all labels assigned to that data point. This process creates a unified set of labels, denoted as l_1.

3.4 Merging Labels

The steps described so far generate n sets of labels for n partitions. We now discuss how ACE merges the label sets to produce the final output L. This is a crucial step: in contrast to the original algorithms, which have a "global" view, ACE needs to merge the local results into a global clustering. Merging multiple sets of labels is not as trivial as, say, concatenating them, because local clusterings may have labeled clusters differently. For example, let the labels of partition S_1 be $[r_1 : C_a, r_2 : C_a, r_3 : C_b]$ and partition S_2's be $[r_4 : C_a, r_5 : C_b, r_6 : C_b]$, where r_i are the points and the two clusters are C_a and C_b. If we simply aggregated the two outputs, we would place r_1, r_2, r_4 in one cluster and r_3, r_5, r_6 in another cluster. However, this would be incorrect, as cluster C_a of S_1 might be closer to C_b of S_2 than C_a. Figure 5(a) illustrates this problem, where cluster C_a of S_1 is closer to C_b of S_2 than C_a. To determine the correct correlation between the two outputs we devised two merging techniques, described next.

ACE supports both algorithms that require users to specify k (the number of clusters), and those that automatically determine k, as explained next.

Constant-k-Merge. We employ this technique for algorithms such as SC and HAC that require users to specify k, the number of clusters. First, we calculate the centers of all the clusters ($[Ci_1, Ci_2, ..., Ci_k]$) in each partition S_i. Next, we merge two partitions by calculating pairwise $k * k$ distances between their clusters and assigning the same label to the closest cluster. We keep merging two partitions of same size until we merge all the partitions to one output. Figure 5(a) illustrates the formation of four clusters from the two partitions. Using *Constant-k-Merge* with k=2 will create two clusters from the four existing ones.



F S_2,
(2].

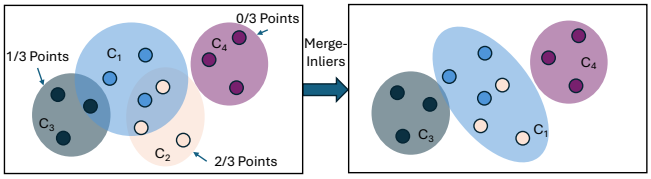


Fig. 6. Merging two clusters C_1 and C_2 using *Merge-Inliers* method.

The distances between the centers of these four clusters are computed, and the two clusters with the shortest distances are merged, as illustrated in Figure 5(b).

Merge-Inliers. Algorithms such as AP and DBSCAN do not require users to provide k; instead they determine the number of clusters based on the structure of the dataset. As a result, each partition creates a different number of clusters, and merging them using *Constant-k-Merge* is not possible.

We address this by applying *anomaly detection* [8] to determine the boundary of a cluster a_{s_i} (the output of a partition S_i). This gives us a model $M_{a\_s\_i}$. Next, we apply the model on the points of a cluster b_{s_j} of another partition S_j, and if a majority (>50%) of the samples are normal, i.e., not anomalies, we conclude that cluster a_{s_i} of S_i and cluster b_{s_j} of S_j belong to the same cluster in the new merged partition S_{ij}. We create a new cluster in S_{ij} for a cluster in S_j if we do not find any clusters in S_i where the number of anomalies is less than 50%.

Figure 6 illustrates this method on four clusters, C_1, C_2, C_3, and C_4, originating from different partitions. Following the generation of a model trained on cluster C_1 to identify the boundary of its inliers, we assess whether any points from other partitions lie within this region. Notably, two out of three points, i.e., more than 50% of the data points within C_2, fall within this inlier region, prompting the decision to merge clusters C_1 and C_2.

4 Experiments

We evaluated ACE on four algorithms: AP, DBSCAN, HAC, and SC. AP's implementation requires multiple $O(n^2)$ arrays, which are updated in multiple it-

Table 2. Severe memory limitations with default implementations: dataset limits and number datasets, out of 164, that failed to run.

System	Size limit (#points)			Failed to run (#datasets)			
	AP	HAC	SC	AP	DBSCAN	HAC	SC
Sys1	84,000	179,000	79,000	139	15	134	139
Sys2	80,000	157,000	75,000	139	6	136	139
Sys3	169,000	297,000	158,000	136	2	134	136
Sys4	110,000	223,000	103,000	137	3	134	137

erations; systems that do not have the physical memory for the arrays will either produce an error or take orders of magnitude longer due to swapping. DBSCAN lacks parallel execution support, leading to long runtimes and memory issues, especially on large datasets; for neighborhood computations, the implementation uses either *kd-tree* or *brute-force*, which affects complexity. HAC, similar to AP, faces challenges with memory demands. However, the execution time is much faster than AP. SC is the most time- and space-consuming algorithm, even when adjusting the *eig_tol* hyperparameter that controls the precision of eigen decomposition. ACE addressed these inefficiencies, markedly reducing runtime and memory usage, and allowing substantial datasets to be clustered.

4.1 Memory

The memory complexity of AP/HAC/SC is $O(n^2)$, where n is the number of points in a dataset. Each algorithm creates multiple `float64` (8 bytes) arrays of size $n*n$. For instance AP creates (among other variables) four 2D arrays of size $n*n$, i.e., *similarity matrix* `S`, *availability matrix* `A`, *responsibility matrix* `R`, and an intermediate results matrix `tmp`. Therefore, running AP on a dataset with 1 million points requires $(1,000,000^2)*8*4 = 3.2*10^{13}$ *bytes*, i.e., 32TB of memory. Similarly, HAC and SC create two and four large $n*n$ arrays, respectively (among other variables) hence their minimum memory requirements on a 1M dataset are 16TB and 32TB, respectively. The system thus limits the set size (number of points) that default implementations can process to relatively small datasets. While DBSCAN imposes no bounds on the number of points, we encountered memory issues with large sparse datasets. The space complexity of DBSCAN in default settings is $O(n)$ for *brute-force* and $O(n * log\, n)$ for *kd-tree*; out of 164 datasets, the implementation selected *kd-tree* for 32 datasets.

Table 2 shows the results of running the original implementations of the algorithms on our systems. Columns 2–4 show the maximum dataset size (#points): depending on the system for the three algorithms, this limit was 80,000–169,000 points for AP, 179,000–315,000 for HAC, and 75,000–158,000 points for SC, respectively. Columns 5–8 show the number of datasets, out of 164, that could not be run with the original implementations: 2–139 sets, depending on the system and algorithm. Notably, the original implementations of AP, HAC and SC failed to cluster more than 80% of datasets, due to memory requirements.

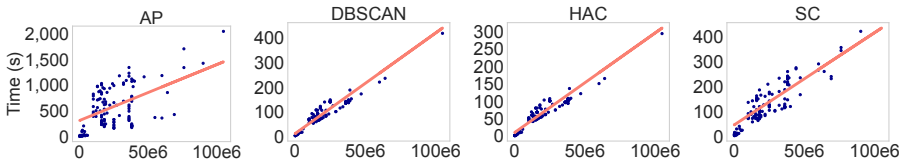


Fig. 7. ACE runtime grows linearly w.r.t. #points (n) $*$ #features (d); the x-axis represents $n * d$ up to 100M, while the y-axis represents time; regression line in red.

In contrast, ACE *ran all four algorithms successfully, on the 164 datasets, on all four systems.*

4.2 Runtime

Scalability. In addition to memory constraints, the time required for clustering can be prohibitive. In the worst case, the time complexities of AP and HAC are $O(n^2 * max\_iter)$ and $O(n^2)$, respectively. DBSCAN's time complexity can be $O(n^2)$ or $O(n * log\, n)$ depending on the distance measure. For SC, while termination depends on reaching convergence, an $O(n^3)$ time complexity lower bound is imposed by calculating the Laplacian matrix. While the number of dimensions d is typically omitted from complexity calculations, we have noticed that in practice d is an important linear factor that should be considered. ACE's time complexity was observed to be linear with respect to $n * d$. In Figure 7 we show ACE's clustering time on Sys1 (other systems' time followed the same trends), for each of the 164 datasets and each algorithm: each blue dot corresponds to a dataset. The regression line, shown in red, indicates ACE's linear scalability.

Table 3. Average Runtime [R_{AP}, R_{DBS}, R_{HAC}, R_{SC}: computed across those datasets that fit into memory and completed execution within 2 hours.]

System	Default								ACE			
	Timeout (# Datasets)				Mean Time (seconds)				Mean Time seconds (speedup)			
	R_{AP}	R_{DBS}	R_{HAC}	R_{SC}	R_{AP}	R_{DBS}	R_{HAC}	R_{SC}	R_{AP}	R_{DBS}	R_{HAC}	R_{SC}
Sys1	2	0	0	14	564	159	45	1,139	8(70x)	106(1.5x)	7(6.4x)	16(71x)
Sys2	3	0	0	14	1,874	149	111	873	31(60x)	132(1.13x)	19(5.8x)	60(15x)
Sys3	3	0	0	17	1,637	228	77	751	17(96x)	128(1.8x)	13(5.9x)	46(16x)
Sys4	5	0	0	16	1,947	4,146	91	773	31(63x)	114(36x)	16(5.7x)	54(14x)

Acceleration. We now present the results of our acceleration experiments, that compare Default vs. ACE for those datasets that default implementations can cluster. We set a two-hour time limit and introduce a subset of datasets denoted R_{AP}, R_{DBS}, R_{HAC} and R_{SC}, representing datasets that successfully

Table 4. Average Accuracy [R_{AP}, R_{DBS}, R_{HAC}, R_{SC}: computed across those datasets that fit into memory and completed execution within 2 hours.]

Method	Accuracy (ARI) vs Ground Truth			
	R_{AP}	R_{DBS}	R_{HAC}	R_{SC}
Default	0.014	-0.006	**0.025**	-0.0004
ACE	**0.070**	**0.024**	0.016	**0.005**
p-value	0.048	2.46e-7	0.544	0.049

completed execution within the limit under default implementation settings. Table 3's "Timeout" grouped columns show the number of datasets that fit into memory but failed to complete due to the timeout. The "Mean Time" columns juxtapose the mean runtime of default implementation and ACE, when applied to $R_{AP} \ldots R_{SC}$ datasets. We encountered timeouts with AP and SC algorithms but not with DBSCAN and HAC. Notably, with SC, we encountered the most timeouts, 14–17 datasets. In contrast, ACE *successfully and efficiently completed the clustering for all 164 datasets, typically within 15 minutes.*

The table also demonstrates ACE's runtime efficiency improvements. For instance, on *Sys1*, the average runtime of default AP on R_{AP} datasets was 564 seconds; ACE reduced this to 8 seconds, a 70x acceleration. Similarly, *Sys2*, *Sys3*, and *Sys4* showcased significant improvements with ACE, reducing mean runtimes from 1,874 to 31 seconds, 1637 to 17 seconds, and 1,947 to 31 seconds, respectively, for R_{AP} datasets (a 60x–96x acceleration).

With DBSCAN, we saw the least improvement in time on most systems, with ACE only able to reduce time by 11–47%, with a notable exception on *Sys4*, where ACE was able to reduce runtime from 4,145.9 to 114 seconds on average. However, it is important to note that small datasets (generally less than 50,000 points) require increased runtime with ACE compared to default settings on most systems. This is attributed to the additional steps involved in ACE, introducing overhead for each dataset irrespective of size.

The HAC implementation was the fastest among all algorithms: the average runtime across the four systems was 45–111 seconds for R_{HAC}. ACE was able to accelerate this task by about 6x, requiring only 7–19 seconds on average.

As previously mentioned, SC was the slowest algorithm. Across the four systems, only 11 datasets were clustered within the two-hour limit. By adjusting the eigen tolerance parameter to 0.001, SC was able to process 28 datasets, with runtimes ranging between 751–1,139 seconds. In contrast, with ACE, we could process all datasets and achieved a speedup of 14x–71x.

5 Accuracy

ACE is designed to reduce runtime and memory without compromising accuracy. We quantify ACE's impact on accuracy as follows: for all the datasets that could be processed with both default implementations and ACE, we computed the clustering accuracy (ARI compared to ground truth). Next, we ran two-means

t-tests between the default and ACE accuracy distributions. We present the results in Table 4. The first two data rows show the mean accuracy for the default and ACE, respectively (higher is better), while the last row shows the statistical significance, expressed as p-value. Note that for three algorithms (AP, DBSCAN and SC), ACE has achieved a *statistically significant improvement in accuracy* (p-value < 0.05). For HAC, while there is a nominal decrease in mean accuracy, this accuracy is not statistically significant (p-value $= 0.544$). These accuracy improvements, along with substantial reductions in runtime and memory, make ACE a preferable alternative to default implementations.

6 Related Work

Prior work on clustering acceleration has employed data/feature partitioning, hardware acceleration, or algorithm-specific parallelization; these techniques lack generality and require algorithm-specific manual source code changes.

Andrade et al. [5] developed G-DBSCAN, a GPU-accelerated DBSCAN, which constructs a graph and uses BFS for cluster identification. They achieved over *>100x* speedup on 5,000 to 700,000 point datasets. Böhm et al. [7] introduced CUDA-DClust for GPU-accelerated clustering with the index structure built on the CPU; Poudel et al. developed CUDA-DClust+ to construct the index directly on the GPU, reducing this overhead [18]. Li et al. [16] proposed a GPU-accelerated version of K-Means; they created two separate techniques for low and high dimensional datasets. Jin and JaJa [14] developed a heterogeneous CPU-GPU SC algorithm by accelerating the major steps, i.e., constructing the similarity matrix, eigenvector computation, and k-means clustering.

He et al. [12] introduced MR-DBSCAN, a scalable DBSCAN algorithm utilizing MapReduce. They fully parallelized critical sub-procedures and introduced a new data partitioning method, ensuring load balancing. MapReduce was also applied to K-Means acceleration using FPGAs [17] and Hadoop [3].

Federated learning work has explored clustering, but not accelerated it [3, 15, 11, 23]. While these methods could potentially be adapted to enhance clustering efficiency in federated learning settings, it is important to note that these techniques might not be compatible with all types of clustering algorithms, particularly those like DBSCAN or AP, which do not use gradient-based updates.

7 Conclusions

ACE addresses the critical roadblocks of runtime and memory for clustering algorithms. Through black-box parallelization, ACE enables tasks with original superlinear complexity to run efficiently, facilitating clustering of large datasets on standard desktop/laptop/datacenter computers. In our comprehensive evaluation on 164 OpenML datasets and four popular algorithms, ACE has significantly reduced memory and accelerated computation, compared to default implementations. By increasing clustering scalability across various domains, ACE paves the way for more efficient data analysis and decision-making processes.

Acknowledgments

We thank Chase Wu and the anonymous reviewers for their valuable feedback on this work. This material is based upon work supported by the National Science Foundation under Grant No. CCF-2007730.

References

1. numerai28.6. https://www.openml.org/search?type=data&id=23517
2. Openml — openml.org. https://www.openml.org, [Accessed 22-01-2024]
3. Anchalia, P.P., Koundinya, A.K., Srinath, N.: Mapreduce design of k-means clustering algorithm. In: ICISA'13. pp. 1–5. IEEE (2013)
4. Anderberg, M.R.: Cluster analysis for applications: probability and mathematical statistics: a series of monographs and textbooks, vol. 19. Academic press (2014)
5. Andrade, G., Ramos, G., Madeira, D., Sachetto, R., Ferreira, R., Rocha, L.: G-dbscan: A gpu accelerated algorithm for density-based clustering (2013)
6. Bhimani, J., Leeser, M., Mi, N.: Accelerating k-means clustering with parallel implementations and gpu computing. In: HPEC'15. pp. 1–6. IEEE (2015)
7. Böhm, C., Noll, R., Plant, C., Wackersreuther, B.: Density-based clustering using graphics processors. In: CIKM'09. pp. 661–670 (2009)
8. Breunig, M.M., Kriegel, H.P., Ng, R.T., Sander, J.: Lof: identifying density-based local outliers. In: ACM SIGMOD. pp. 93–104 (2000)
9. Ester, M., Kriegel, H.P., Sander, J., Xu, X., et al.: A density-based algorithm for discovering clusters in large spatial databases with noise. In: KDD'96 (1996)
10. Frey, B.J., Dueck, D.: Clustering by passing messages between data points. science **315**(5814), 972–976 (2007)
11. Ghosh, A., Chung, J., Yin, D., Ramchandran, K.: An efficient framework for clustered federated learning. NeurIPS'20 **33**, 19586–19597
12. He, Y., Tan, H., Luo, W., Feng, S., Fan, J.: Mr-dbscan: a scalable mapreduce-based dbscan algorithm for heavily skewed data. FCS'14 **8**, 83–99 (2014)
13. Hubert, L., Arabie, P.: Comparing partitions. Journal of Classification **2** (02 1985)
14. Jin, Y., Jaja, J.F.: A high performance implementation of spectral clustering on cpu-gpu platforms. In: IPDPSW'16. pp. 825–834. IEEE (2016)
15. Kumar, H.H., Karthik, V., Nair, M.K.: Federated k-means clustering: A novel edge ai based approach for privacy preservation. In: CCEM'20. pp. 52–56. IEEE
16. Li, Y., Zhao, K., Chu, X., Liu, J.: Speeding up k-means algorithm by gpus. Journal of Computer and System Sciences **79**(2), 216–229 (2013)
17. Li, Z., Jin, J., Wang, L.: High-performance k-means implementation based on a simplified map-reduce architecture. arXiv preprint arXiv:1610.05601 (2016)
18. Poudel, M., Gowanlock, M.: Cuda-dclust+: Revisiting early gpu-accelerated dbscan clustering designs. In: HiPC'21. pp. 354–363. IEEE (2021)
19. Rousseeuw, P.J.: Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. Journal of computational and applied mathematics **20** (1987)
20. Shi, J., Malik, J.: Normalized cuts and image segmentation. IEEE Transactions on pattern analysis and machine intelligence **22**(8), 888–905 (2000)
21. Shi, X.: Parallelizing affinity propagation using graphics processing units for spatial cluster analysis over big geospatial data. In: Geocomputation'15. Springer (2017)
22. Zhao, W., Ma, H., He, Q.: Parallel k-means clustering based on mapreduce. In: CloudCom'09. pp. 674–679. Springer (2009)
23. Zhao, X., Xie, P., Xing, L., Zhang, G., Ma, H.: Clustered federated learning based on momentum gradient descent for heterogeneous data. Electronics **12**(9) (2023)