J Supercomput (2015) 71:2365–2380
DOI 10.1007/s11227-015-1385-0

A comparative study of the parallel wavelet-based clustering algorithm on three-dimensional dataset

Ahmet Artu Yıldırım · Dan Watson

Published online: 5 February 2015

Abstract Cluster analysis—as a technique for grouping a set of objects into similar clusters—is an integral part of data analysis and has received wide interest among data mining specialists. The parallel wavelet-based clustering algorithm using discrete wavelet transforms has been shown to extract the approximation component of the input data on which objects of the clusters are detected based on the object connectivity property. However, this algorithm suffers from inefficient I/O operations and performance degradation due to redundant data processing. We address these issues to improve the parallel algorithm's efficiency and extend the algorithm further by investigating two merging techniques (both merge-table and priority-queue based approaches), and apply them on three-dimensional data. In this study, we compare two parallel WaveCluster algorithms and a parallel K-means algorithm to evaluate the implemented algorithms' effectiveness.

Keywords Parallel clustering · Discrete wavelet transform · Improved parallel WaveCluster algorithm

1 Introduction

Cluster analysis is a widely-used technique that is employed to map a set of objects into groups (i.e., clusters) based on their similar properties, such as spatial and temporal similarity. Sheikholeslami et al. [7] introduced a novel unsupervised clustering approach, called WaveCluster, utilizing a discrete wavelet transform (DWT) that

A. A. Yıldırım (✉) · D. Watson
Department of Computer Science, Utah State University, Logan, UT 84322, USA
e-mail: ahmetartu@aggiemail.usu.edu

D. Watson
e-mail: dan.watson@usu.edu

enables data analysts to perform clustering in a multi-level fashion. This method has the ability to discover clusters with arbitrary shapes and can deal with outliers effectively.

We previously reported a parallel wavelet-based clustering algorithm to process large datasets using a message-passing library (MPI) for distributed memory architectures [12]. While this algorithm achieves linear behavior in terms of algorithm complexity, it suffers from inefficient I/O operations over two-dimensional datasets and performance degradation due to redundant data processing. In this study, we introduce two parallel wavelet-based clustering algorithms that address these issues by benefiting collective MPI I/O capabilities and efficient usage of data structures. Additionally, we cluster three-dimensional datasets with the new algorithms.

To illustrate the effectiveness of this approach, a comparison is performed between our two new parallel WaveCluster algorithms and parallel K-means clustering algorithm. Because one of the fundamental reasons of employing parallelism is to overcome space restrictions by exploiting aggregate memory on the distributed memory systems, and to obtain scalable performance, we take these two important metrics into consideration to measure the performance of the parallel algorithms.

The rest of the paper is organized as follows. Section 2 gives a brief explanation of wavelet transforms from the point of mathematical and practical views. We give the main components of the WaveCluster algorithm and its main phases in Sect. 3. The algorithmic approaches used in the implementation of the parallel algorithms are explained in detail in Sect. 4. Finally, Sects. 5 and 6 provide experimental results with the analysis of the algorithms and concluding remarks, respectively.

2 Wavelet transforms

Wavelet transforms (WTs) are a mathematical technique to analyze non-stationary data to extract frequency information at different resolution scales from the original signal. WTs are commonly used in variety of areas such as in image compression [4], speech recognition [10], and in the analysis of DNA sequences [1]. WTs can be considered a complementary approach to Fourier transforms (FTs). One disadvantage of FTs is that they cannot determine which frequency band exists at any specific time interval in the signal. Short-time Fourier transforms (STFTs) might provide a remedy for time-frequency analysis by dividing the signal into successive segments and then performing an FT on each one, but STFTs suffer from the requirement of choosing the 'right' window width, where a narrow window leads to good time resolution but poor frequency resolution, and vice versa. A more detailed discussion of this effect can be found in [2,6,8].

The main idea of the Wavelet transform is based on the dilation and translation of the wavelet Ψ ('small wave' function) (i.e., the mother wavelet) continuously over the signal $f(t)$ [11];

$$W_f(s,\tau) = \int_{-\infty}^{\infty} f(t)\Psi^*_{s,\tau}(t)\mathrm{d}t \tag{1}$$

The mother wavelet Ψ is defined as;

$$\Psi_{s,\tau}(t) = \frac{1}{\sqrt{s}}\Psi\left(\frac{t-\tau}{s}\right) \tag{2}$$

where s is the dilation or scaling factor determining the window width, and the factor τ manages the translation of the wavelet function Ψ. $\frac{1}{\sqrt{s}}$ is for energy normalization.

There are many mother wavelet functions, such as Haar, Daubechies, Morlet and Mexican hat. Some mother wavelets are depicted in Fig. 1. In order to be regarded as a mother wavelet, the function must satisfy the two conditions in Eq. (3a), which indicates the net area of the corresponding wavelet graph must be zero, but the absolute area cannot be zero (3a). Thus, it must be an oscillating function and have unit energy [9].

$$\int_{-\infty}^{\infty} \Psi(t)\,\mathrm{d}t = 0 \tag{3a}$$

$$\int_{-\infty}^{\infty} |\Psi(t)|^2\,\mathrm{d}t = 1 \tag{3b}$$

In this paper, the approximation coefficients will be referred to as the 'low-frequency component', and the term 'high-frequency component' will be used interchangeably as the detail coefficients of the signal. By means of the mother wavelet function, detail coefficients can be detected at different time intervals. However, we need another function used to retrieve average coefficients of the signal, which is referred to as the scaling function (i.e., the father wavelet) Φ. The father wavelet is orthogonal to the mother wavelet in that both wavelet functions form the basis for the multi-resolution analysis. In fact, the father wavelet is not considered as a 'wavelet function' because it does not satisfy the condition (Eq. 3a). However, by the combination of both wavelet functions, we gain the ability to decompose the signal into low-frequency and high-frequency components.

Algorithm 1 Discrete Wavelet Transform (DWT) algorithm

Require: Signal X with size $N = 2^{level}$ where $level$ is integer
Ensure: List of low-frequency components A and high-frequency components D
1: $A_0 \leftarrow X$
2: $D \leftarrow \emptyset$
3: **for** $j = 1 \rightarrow level$ **do**
4: $\quad [A_j, D_j] \leftarrow DWT_{\Phi,\Psi}(A_{j-1})$
5: **end for**

Algorithm 1 shows the procedure to compute discrete wavelet transforms (DWTs) that return the list of low-frequency components and high-frequency components. In DWTs, the size of the input signal must be power of two in each dimension to perfectly

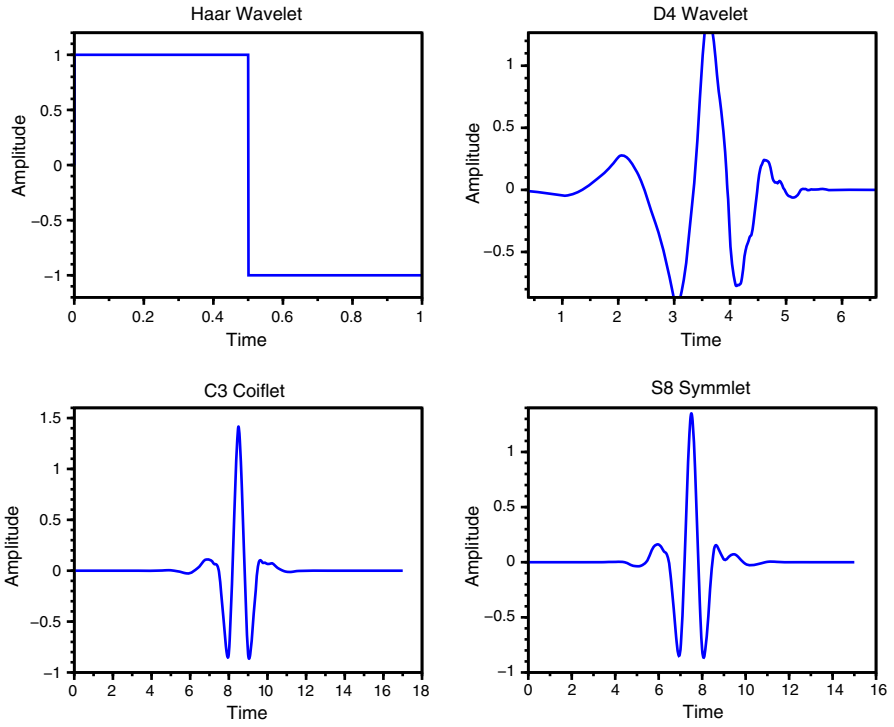

Fig. 1 Some mother wavelet functions: Haar, Daubechies 4, C3 Coiflet, S8 Symlet

decompose the signal, while this is not a requirement in continuous wavelet transforms (CWTs) due to continuous nature of the signal and the factors of s and τ. A_j denotes the approximation coefficients and D_j denotes detail coefficients at level j. After each level, the components of A_j and D_j are extracted using A_{j-1} in a recursive manner where A_0 refers to the original input signal.

Figure 2 shows the decomposition of a sample non-stationary signal by means of Daubechies wavelet function. Because the width of the wavelet window is doubled at each level, the time resolution is halved because of downsampling by two; thus we obtain coarser representations of the original signal; however, the frequency resolution is doubled in that the approximation components contain the lowest half of the frequency and the detail components take the other half.

3 Wavelet-based clustering algorithm

Cluster analysis is a widely-used technique that is employed to map a set of objects into groups (i.e., clusters) based on their similar properties, such as spatial and temporal similarity. Sheikholeslami et al. [7] introduced a novel unsupervised clustering approach, called WaveCluster, using DWTs that enable data analysts to perform clustering in a multi-level fashion. The method can discover clusters with arbitrary

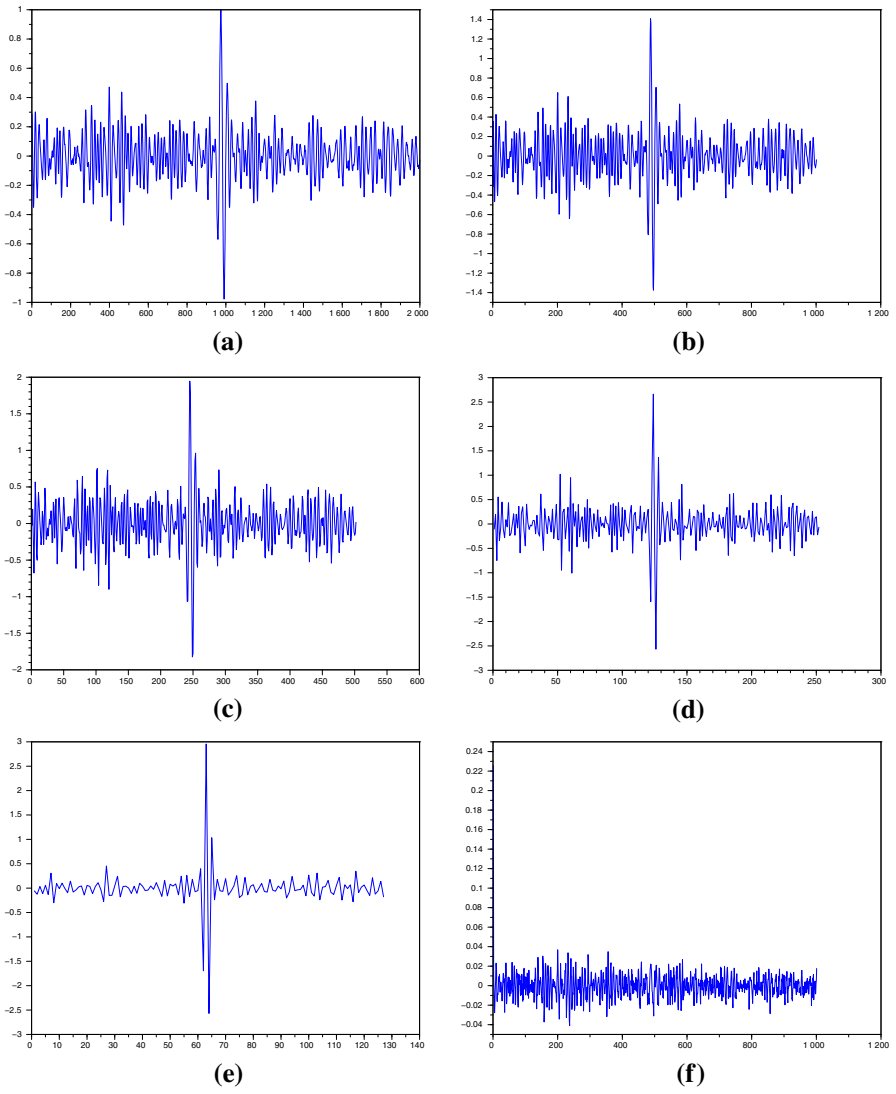

Fig. 2 Wavelet decomposition of a sample non-stationary signal. **a** Original signal (A_0). **b** Approximated coefficients at level 1 (A_1). **c** Approximated coefficients at level 2 (A_2). **d** Approximated coefficients at level 3 (A_3). **e** Approximated coefficients at level 4 (A_4). **f** Detailed coefficients at level 1 (D_1). **g** Detailed coefficients at level 2 (D_2). **h** Detailed coefficients at level 3 (D_3). **i** Detailed coefficients at level 4 (D_4)

shapes and can deal with outliers (data points that don't belong to any cluster) effectively.

WaveCluster defines the notion of a cluster as a dense region consisting of neighboring objects (in the 8-connected neighborhood) in the low-frequency component of the data at level j (A_j). The low-frequency component represents a lower resolution approximation of the original feature space on which connected component labeling

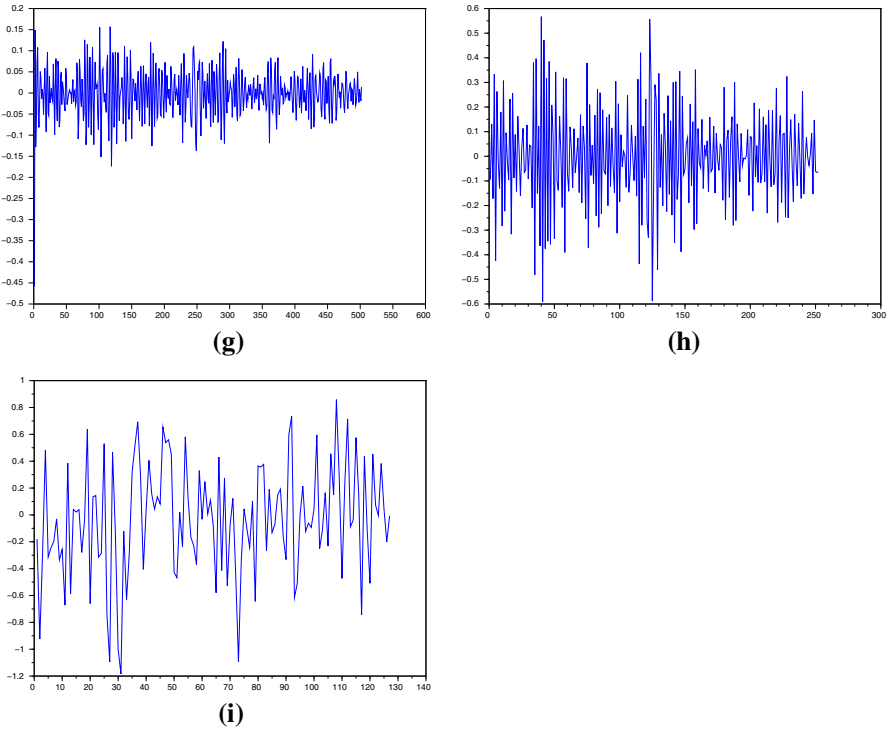

Fig. 2 continued

algorithm is performed to detect clusters at different scales from fine to coarse. Hence, the clustering algorithm gains multi-resolution properties by means of the DWT. The corresponding algorithm with its multi-resolution property is illustrated in Fig. 3. The algorithm discards detail coefficients and uses approximation values (low-frequency component) that are extracted by the low-pass filter operation $L[n]^j$ at level j. The algorithm applies thresholding to the approximation coefficients in the last level to remove the outlier data (i.e, *nodata* vertices whose values are less than threshold).

In our implementation, we use the Haar wavelet [3], whose scaling function is described as:

$$\phi(t) = \begin{cases} 1 & 0 \leq t < 1, \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

The Connected Component Labeling algorithm (CCL) is illustrated in Algorithm 2. The algorithm traverses through a list of vertices V in the WaveCluster algorithm such that the list is substituted by the low-frequency component of the input data. In the initialization phase between Line 1 and 4 of the algorithm, all vertices are marked as unvisited and their cluster numbers are set to a special value *nocluster* indicating no vertex is associated with any cluster. The algorithm uses stack S to avoid recursive calls and to keep track of the connected vertices. When there is no element in the stack (stack is empty) in Line 6, the algorithm finds another vertex that is not visited yet

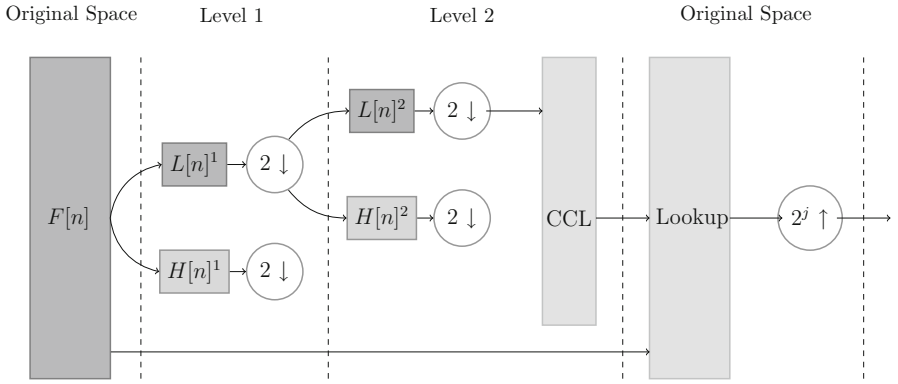

Fig. 3 Multi-resolution property of the WaveCluster algorithm at level 2 on one-dimensional data

and has no *nodata* value. In Line 12, the algorithm pushes the adjacent vertices v_k to the stack where each vertex has no *nodata* value. Thus, all connected vertices are traversed and assigned a unique cluster number to represent distinct clusters over V. The algorithmic complexity of the CCL algorithm is $O(N)$ where N is the size of the input list.

Algorithm 2 Connected Component Labeling (CCL) algorithm

Require: V is a list of vertices with size N where the vertices $v_i, v_j, v_k \in V$ and vertex v_k is adjacent to v_j. i, j and k are indices that identify the vertices over V.
Ensure: C stores the cluster numbers of the vertices where the value *nocluster* indicates the vertex v_i is not associated with any cluster.
1: **for** $i = 1 \rightarrow N$ **do**
2: $visited[i] \leftarrow false$
3: $C[i] \leftarrow nocluster$
4: **end for**
5: $clusternumber \leftarrow 1$
6: **for all** $v_i \in V$ *where* $visited[i] \neq true$ *and* $v_i \neq nodata$ **do**
7: $S \leftarrow push(S, v_i)$
8: **while** S *is not empty* **do**
9: $v_j \leftarrow pop(S)$
10: $visited[j] \leftarrow true$
11: $C[j] \leftarrow clusternumber$
12: $S \leftarrow push(S, v_k)$ *where* v_k *is neighbor of* v_j *and* $v_k \neq nodata$
13: **end while**
14: $clusternumber \leftarrow clusternumber + 1$
15: **end for**

In the final phase, mapping the units in the transformed feature space to the original feature space, a lookup procedure is carried out that one object in the low-frequency component at level j corresponds to $2^{(j \times d)}$ objects in the original signal where d is the dimension of the data. If the object is not represented in the transformed feature space (outlier), the object in the original space is assigned a *nocluster* value.

4 Parallel wavelet-based clustering algorithms

We previously reported parallel wavelet-based clustering algorithm using a message-passing library (MPI) for the distributed memory architecture. While the algorithm possesses linear behavior in terms of algorithm complexity, it suffers from the inefficient I/O operations and performance degradation due to redundant data processing. In this study, we address those issues by improving two parallel WaveCluster algorithms that are based on merge-tables and priority-queues, respectively. The two parallel algorithms differ from the way they handle the merging phase, which is fundamental in the context of parallelism.

In the previous algorithm [12], the master processor reads the whole two-dimensional input dataset and then distributes each stripe of data to the slave processors. This approach leads to inefficient I/O operation. In order to minimize I/O times, both algorithms benefit from collective MPI I/O capabilities in which each processor reads its local data in parallel collectively via MPI_File_Read function. We also extend the study by processing larger and three-dimensional datasets.

4.1 Priority-queue based parallel WaveCluster algorithm

We implemented a new parallel WaveCluster algorithm whose merging phase is performed using priority-queue data structure. As algorithmic improvements, in the merging phase of the previous parallel WaveCluster algorithm [12], the master processor creates the merging table with respect to the adjacency relations of bordering data for each slave processor. Then, the algorithm distributes the result that each processor updates its local clustering numbers using the merging table. This approach leads to inefficiency due to the idle time of slave processors during the execution of merging phase at master processor. We employ cooperative merging operation between adjacent processors to alleviate this execution inefficiency. In this approach, each processor maintains its local clustering result. Ultimately, this leads to a globally correct result as ghost regions are swapped among adjacent processors.

Algorithm 3 Parallel WaveCluster Algorithm using priority-queue structure; where ρ is target wavelet level and i denotes MPI process number i, that is, $A_{j,i}$ is the partition of low-frequency component in level j, C_i is the output of the CCL algorithm at the first invocation, and then the output of $Merge$ algorithm in loop, CR_i is the final clustering result of partition i, all processed by MPI processes in parallel

1: $A_{0,i} \leftarrow loadLocalDomain\ (A_0,\ i)$
2: $A_{\rho,i} \leftarrow DWT(A_{0,i},\ \rho,\ threshold)$
3: $C_i \leftarrow CCL(A_{\rho,i})$
4: **repeat**
5: $\quad G_i \leftarrow swapGhostRegion()$
6: $\quad C_i \leftarrow Merge(C_i, G_i)$
7: **until** *no cluster value is changed globally*
8: $CR_i \leftarrow lookup(A_{0,i},\ C_i,\ \rho)$
9: $writeClusteringResult(CR_i, i)$

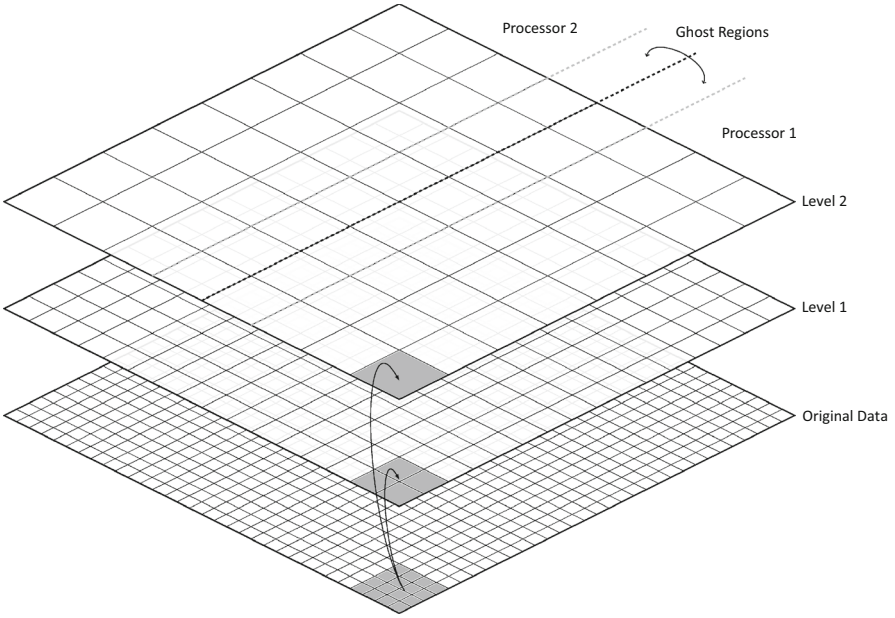

Fig. 4 Parallel WaveCluster Algorithm with ghost data illustration using two processors

The main algorithm is given in Algorithm 3. All operations in the algorithm figure are performed in parallel. The partitions of $A_0 = A_{0,1} \cup A_{0,2} \cup \cdots \cup A_{0,n}$ are distributed evenly among the MPI processes in a striped fashion where n is the number of MPI processes. Each process performs discrete wavelet transform using its own input partition data independently and returns the local low-frequency component with level ρ in Line 2. The value of input vertex over A_0 has either 1 or 0 that indicates whether there is a vertex or not at a particular index at A_0. The algorithm uses $threshold$ value to remove the outlier vertices. The algorithm considers all vertices in the domain to be 'outliers' when threshold value is 1. When the value is 0, the algorithm does not discard any vertex from the domain. In Line 3 CCL function finds the connected vertices using local transformed feature space.

Subsequent calls, between Lines 4 and 7, deal with merging of the connected vertices that might possibly run through the process borders. The parallel merge phase is performed to merge the clusters globally, and propagates the minimum cluster number through the connected vertices of the local domain. In this phase, the parallel algorithm first calls $swapGhostRegions$ in Line 5, whose goal is to exchange ghost data among the neighboring processes, as illustrated in Fig. 4. The ghost region is a low-frequency component with 1-vertex thickness that is adjacent to the local low-frequency component of the neighboring MPI process. By swapping the ghost regions, the algorithm propagates the minimum cluster number of the connected vertices through the processes, which possibly can span the whole domain. Then, the process calls $merge$ function in Line 6 to merge the clusters in the local transformed domain using the latest ghost data.

Algorithm 4 Priority-queue based cluster merging algorithm

Require: C is a list of local cluster numbers of the vertices with size N where cluster numbers of $c_i, c_j, c_k, c_g, \in C$ for the vertices v_i, v_j, v_k and v_g respectively. G is the ghost region retrieved from the neighboring MPI processes where the vertex $v_g \in G$, PQ is the priority-queue structure whose function $pop(PQ)$ returns and removes the vertex with minimum cluster number at the priority-queue, S is stack structure.

Ensure: The algorithm returns the updated list of clustering result C

1: $ischanged \leftarrow false$
2: **for** $i = 1 \rightarrow N$ **do**
3: $\quad visited[i] \leftarrow false$
4: **end for**
5: $PQ \leftarrow push(PQ, G)\ where\ v_g \in G\ and\ c_g \neq nocluster$
6: **while** $PQ\ is\ not\ empty$ **do**
7: $\quad v_g \leftarrow pop(PQ)$
8: $\quad S \leftarrow push(S, c_i)\ where\ c_i\ is\ neighbor\ of\ v_g\ and\ c_i \neq nocluster$
9: $\quad$ **while** $S\ is\ not\ empty$ **do**
10: $\quad\quad c_j \leftarrow pop(S)$
11: $\quad\quad visited[j] \leftarrow true$
12: $\quad\quad$ **if** $c_k < c_j$ **then**
13: $\quad\quad\quad C[j] \leftarrow c_k$
14: $\quad\quad\quad ischanged \leftarrow true$
15: $\quad\quad\quad S \leftarrow push(S, c_k)\ where\ v_k\ is\ neighbor\ of\ v_j\ and\ c_k \neq nocluster$
16: $\quad\quad$ **end if**
17: $\quad$ **end while**
18: **end while**

In the *merge* function (shown in detail in Algorithm 4), we use priority-queue structure PQ to retrieve the minimum cluster number on the ghost data (Line 7), and stack S to propagate the cluster number among the connected vertices. If the cluster number of the popped ghost vertex is smaller than the neighboring vertex of the local domain (Line 12), the merge function updates all the connected vertices' cluster numbers and then marks those vertices as visited, which insures that those vertices are never visited for one iteration of the merging phase only. This merging phase runs until all clusters are detected globally, keeping track with the variable $ischanged$ locally and by reducing all $ischanged$ variables to the single $globalischanged$ variable via MPI_Reduce function globally.

Finally, the lookup procedure is called in Algorithm 3 in Line 8 to map the vertices in the transformed feature space to the original feature space. Those that are considered 'outliers' based on $threshold$ value are marked with special $nocluster$ value and the results are written to the disk in parallel.

4.2 Merge-table based parallel WaveCluster algorithm

The second parallel WaveCluster algorithm is based on merge-table which is similar to the previous algorithm introduced in [12]. We observe that memory consumption on the merging phase is too high due to data sparsity because of the nature of lookup table usage with $O(N)$ space complexity where N is the number of all vertices on input dataset. We addressed this memory consumption issue using a hash table storing only the cluster numbers of vertices which do not have *nodata* values.

The main phases of the merge-table based WaveCluster algorithm are shown on Algorithm 5. The merge-table is the data structure that stores one record for each vertex and each record associates the initial cluster number with the final cluster number. At each iteration to merge the local clustering results, the algorithm repeatedly propagates the minimum cluster number through the connected vertices in the local transformed domain. The algorithm maintains the merge-table using a hash table rather than the lookup table used in the previous implementation to keep track of the proposed cluster number that is smaller than the current cluster number. The hash table key is the initial cluster number and the associated values are the proposed/final cluster number and the coordinate information of the vertex residing on the ghost data. This hash table is initialized before the MPI process starts performing the merging phase. In subsequent iterations, this merge-table is updated based on the ghost data in $updateMergeTable$ function. The merging phase continues until no changes occur in the merge-table globally. Finally, using this merge-table, each process changes the cluster numbers of the connected vertices to reflect the correct cluster numbers in $updateClusterNumbers$, an operation performed using a stack data structure.

Algorithm 5 Merge-table based Parallel WaveCluster Algorithm; where MT is the merge-table

1: $A_{0,i} \leftarrow loadLocalDomain\ (A_0,\ i)$
2: $A_{\rho,i} \leftarrow DWT(A_{0,i},\ \rho,\ threshold)$
3: $C_i \leftarrow CCL(A_{\rho,i})$
4: $MT = initializeMergeTable(C_i)$
5: **repeat**
6: $\quad G_i \leftarrow swapGhostRegion()$
7: $\quad MT \leftarrow updateMergeTable(G_i, MT)$
8: **until** *any merge table is not updated globally*
9: $C_i \leftarrow updateClusterNumbers(C_i, MT)$
10: $CR_i \leftarrow lookup(A_{0,i},\ C_i,\ \rho)$
11: $writeClusteringResult(CR_i, i)$

5 Performance evaluation

In this section, we present the performance comparisons of the two parallel WaveCluster algorithms by means of the elapsed algorithm time and their speed-up ratios for synthetically generated three-dimensional datasets. We implemented a synthetic data generation program that allowed us to minimize application-specific artifacts and concentrate more on properties and benefits of proposed algorithms. Generated input datasets have equal length for each three dimension and power of two as a condition of discrete wavelet transform to fully exploit multi-resolution analysis. The vertex value of dataset is either 1 with probability 0.3 or 0. Thus, the objects over the dataset are evenly distributed. The discussion about the effects of data distribution on the parallel WaveCluster algorithm can be found in [12]. In these experiments, we generated three datasets named Dataset1, Dataset2 and Dataset3 where number of objects—vertices with value 1—are nearly 40, 80 and 161 K, respectively. The details of the datasets are shown in Table 1.

Table 1 Datasets used in the experiments

Name	Size	Number of objects
Dataset1	512 MB	40,268,934
Dataset2	1 GB	80,536,099
Dataset3	2 GB	161,067,172

We compare the parallel WaveCluster algorithms that have better benchmark results with the common parallel clustering algorithm—parallel K-means algorithm. The experiments are conducted on a cluster where each node is equipped with two Quad-Core AMD Opteron(tm) Processor 2376 (8 total cores/node) and 16 GB memory. The interconnection among the nodes is achieved over double data rate (DDR) infiniband. The main performance measures that we consider are then running time of the parallel algorithms and their speed-up ratio with respect to serial execution of the algorithm.

In both parallel WaveCluster algorithms, the input on the original space is initially evenly distributed among p processors in a striped fashion. Figure 5 shows the benchmark results of the two parallel WaveCluster algorithms with different merging approaches called the priority-queue approach and the merge-table approach. The experiments are conducted for wavelet levels 1 and 2 for all datasets. Both parallel WaveCluster algorithms do not scale well for wavelet level 1 because of the high communication time in exchanging the border data. Furthermore, the run time of the algorithm starts to increase when more than 16 processors are used for Dataset1 and Dataset2, and 8 processors for Dataset3. The reason is that the communication time dominates the computation time that is required to obtain a globally correct result. Although the algorithms do not pose effectiveness in the context of parallelism for wavelet level 1, we did not observe this behavior when the wavelet level is 2.

Note that the number of objects to be exchanged between the neighboring processes is decreased by factor of 2^d at each level where d is the dimension of the input dataset. In our experiments, the communication time is decreased by a factor of 8 as wavelet level increases. The communication overhead is reduced with the cost of coarser clustering analysis of original data. This shows that the wavelet level highly significantly affects the scaling behavior of the parallel algorithm due to adverse effect of communication time. While we obtain identical speed-up results on Dataset1 and Dataset2, in the largest one, Dataset3, the merging approach performs better than the priority-queue approach when the wavelet level is 2.

We performed comparison experiments between the two parallel WaveCluster algorithms and the parallel K-means algorithm [5]. We have chosen the parallel K-means algorithm as a representative of a classical parallel clustering algorithm. We define the speedup in comparisons as a fraction of the execution times of the parallel K-means algorithm relative to the parallel WaveCluster algorithm for a varying number of processors and wavelet levels. The speed-up equation is shown below:

$$S_p = \frac{T_{pkmeans,\,p}}{T_{pwavecluster,\,p}} \tag{5}$$

The parallel K-means algorithm must be configured to detect K distinct clusters where K is fixed. However, the number of clusters detected by the WaveCluster

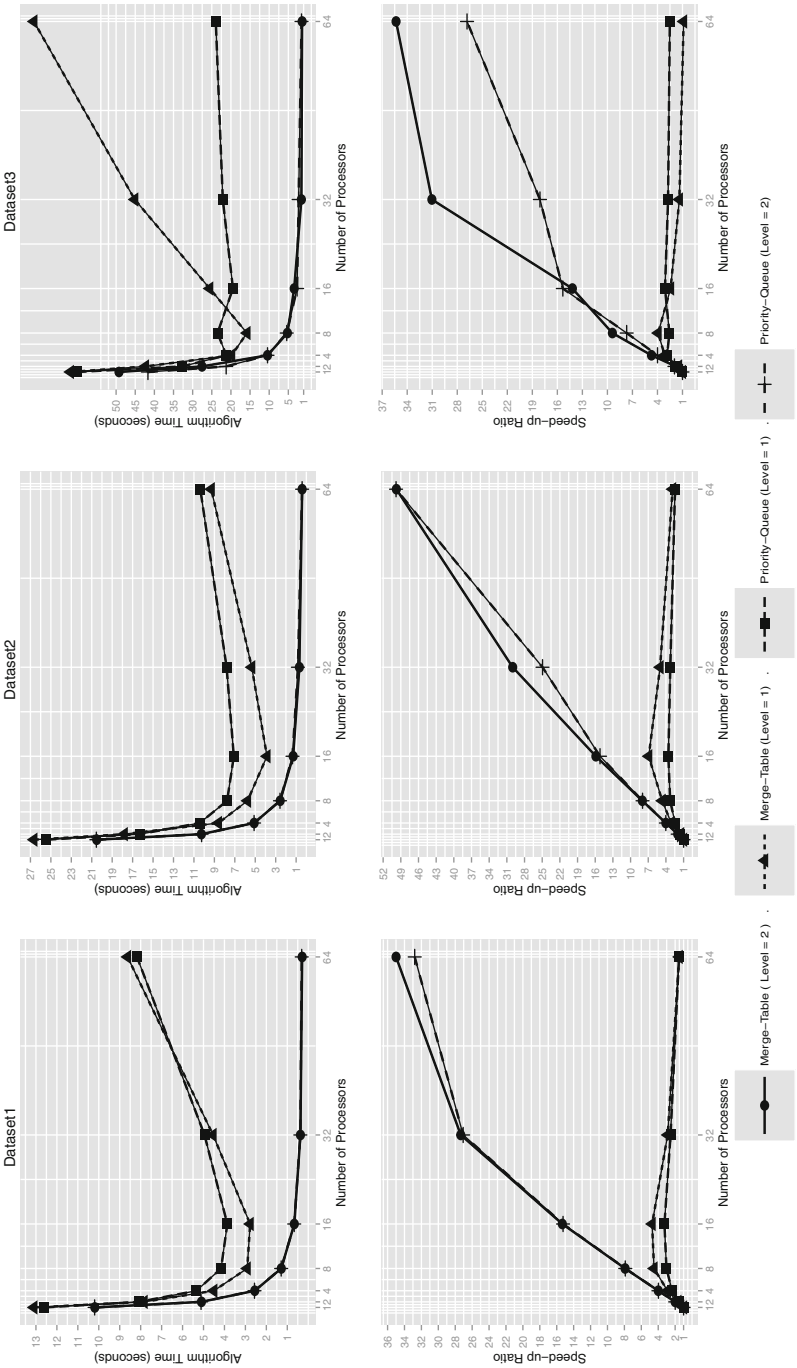

Fig. 5 Parallel WaveCluster Algorithms with varying input dataset files and wavelet levels

Table 2 Execution times (in seconds) of the parallel WaveCluster (PWC) algorithms using priority-queue (PQ) and merge-table (MT) approaches for wavelet levels (ρ) 1 and 2, and the parallel K-means algorithm with a varying number of processors (np) on Dataset1

Exec. time/np	1	2	4	8	16	32	64
P. K-means	2,387.46	395.15	144.31	98.53	45.67	24.98	20.01
PWC-PQ ($\rho = 1$)	12.59	8.06	5.33	4.16	3.86	4.95	8.17
PWC-PQ ($\rho = 2$)	10.19	5.11	2.56	1.29	0.67	0.37	0.31
PWC-MT ($\rho = 1$)	13.10	7.83	4.50	2.91	2.77	4.54	8.62
PWC-MT ($\rho = 2$)	10.18	5.09	2.55	1.28	0.66	0.37	0.29

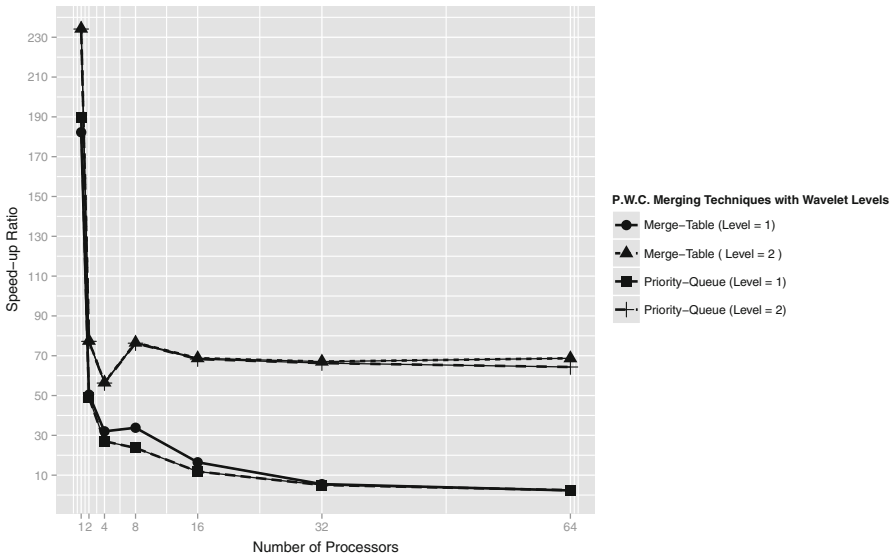

Fig. 6 Speed-up Comparison between parallel WaveCluster algorithms and parallel K-means algorithm; where wavelet levels 1, 2 on Dataset1, $K = 20$

algorithm is determined by the wavelet level and the threshold value which is used to remove outlier objects that do not belong to any clusters. The performance of both parallel WaveCluster algorithms can be better seen in Table 2 for varying processors and wavelet levels. The execution time of parallel K-means algorithm is also included. Note that both parallel WaveCluster algorithms perform significantly better than parallel K-means algorithm on finding clusters. Figure 6 shows that both parallel WaveCluster algorithms are nearly 70 times faster than the parallel K-means algorithm on wavelet level 2 and 8 times faster on wavelet level 1 where a maximum number of processors are utilized. We choose the parameter $K = 20$ in the experiments for K-means algorithm. Because the speed-up values are measured in terms of parallel performance of the algorithms $\left(\frac{T_{pkmeans,\,p}}{T_{pwavecluster,\,p}}\right)$, a slight speed-up drop occurs because parallel K-means algorithm (to which our algorithm is compared) experiences a comparatively greater increase in performance when four processors are used—even

though the wall clock time decreases in both cases. This result shows the effectiveness of the parallel WaveCluster algorithm when compared to the parallel K-means algorithm.

6 Conclusion

Parallel WaveCluster algorithms with their detailed description and comparison study have been presented here. We have improved upon previous work by investigating two different merging techniques, namely priority-queue and merge-table approaches that play important role in the parallel algorithm. These parallel algorithms find distinct clusters using discrete wavelet transformation on distributed memory architectures using the MPI API. Due to high compute times, we did not obtain scalability when wavelet level is 1. However, the scaling behavior is acquired for wavelet level 2.

We have chosen parallel K-means algorithm as a classical parallel clustering algorithm to compare the performance of two parallel WaveCluster algorithms. As an inherent property of the WaveCluster algorithm opposed to the K-means algorithm, it is capable of removing outlier objects that do not belong to any clusters. The effectiveness of the parallel WaveCluster algorithms as compared to the parallel K-means algorithm are shown in the study to have achieved up to $70\times$ speed-up ratio where wavelet level is 2 on Dataset1.

Acknowledgments Compute, storage and other resources from the Division of Research Computing in the Office of Research and Graduate Studies at Utah State University are gratefully acknowledged. We also would like to thank Dr. Wei-keng Liao for providing us with the source code of the parallel K-means algorithm.

References

1. Arneodo A, Bacry E, Graves PV, Muzy JF (1995) Characterizing long-range correlations in DNA sequences from wavelet analysis. Phys Rev Lett 74:3293–3296. doi:10.1103/PhysRevLett.74.3293
2. Cohen L (2000) The uncertainty principles of windowed wave functions. Opt Commun 179(16):221–229. doi:10.1016/S0030-4018(00)00454-5. http://www.sciencedirect.com/science/article/pii/S0030401800004545
3. Haar A (1910) Zur theorie der orthogonalen funktionensysteme. Mathematische Annalen 69(3):331–371. doi:10.1007/BF01456326
4. Lewis AS, Knowles G (1992) Image compression using the 2-D wavelet transform. IEEE Trans Image Process 1(2):244–250. doi:10.1109/83.136601
5. Liu Y, Pisharath J, Liao WK, Memik G, Choudhary A, Dubey P (2004) Performance evaluation and characterization of scalable data mining algorithms. In: Proceedings of IASTED. http://users.eecs.northwestern.edu/wkliao/Kmeans/
6. Loughlin P, Cohen L (2004) The uncertainty principle: global, local, or both? IEEE Trans Signal Process 52(5):1218–1227. doi:10.1109/TSP.2004.826160
7. Sheikholeslami G, Chatterjee S, Zhang A (2000) Wavecluster: a wavelet-based clustering approach for spatial data in very large databases. VLDB J 8(3–4):289–304
8. Shim I, Soraghan JJ, Siew W (2001) Detection of PD utilizing digital signal processing methods. Part 3: open-loop noise reduction. Electr Insul Mag IEEE 17(1):6–13. doi:10.1109/57.901611
9. Torrence C, Compo GP (1998) A practical guide to wavelet analysis. Bull Am Meteorol Soc 79:61–78
10. Tufekci Z, Gowdy J (2000) Feature extraction using discrete wavelet transform for speech recognition. In: Proceedings of the IEEE on Southeastcon 2000, pp 116–123. doi:10.1109/SECON.2000.845444
11. Valens C (1999) A really friendly guide to wavelets. C. Valens@mindless.com 2004

12. Yildirim AA, Ozdogan C (2011) Parallel wavecluster: a linear scaling parallel clustering algorithm implementation with application to very large datasets. J Parallel Distrib Comput 71(7):955–962. doi:10.1016/j.jpdc.2011.03.007