Multi-Threaded Hierarchical Clustering by Parallel Nearest-Neighbor Chaining

Yongkweon Jeon, *Student Member, IEEE* and Sungroh Yoon, *Senior Member, IEEE*

Abstract—Hierarchical agglomerative clustering (HAC) is a clustering method widely used in various disciplines from astronomy to zoology. HAC is useful for discovering hierarchical structure embedded in input data. The cost of executing HAC on large data is typically high, due to the need for maintaining global inter-cluster distance information throughout the execution. To address this issue, we propose a new parallelization scheme for multi-threaded shared-memory machines based on the concept of nearest-neighbor (NN) chains. The proposed multi-threaded algorithm allocates available threads into two groups, one for managing NN chains and the other for updating distance information. In-depth analysis of our approach gives insight into the ideal configuration of threads and theoretical performance bounds. We evaluate our proposed method by testing it with multiple public datasets and comparing its performance with that of several alternatives. In our test, the proposed method completes hierarchical clustering 3.09-51.79 times faster than the alternatives. Our test results also reveal the effects of performance-limiting factors such as starvation in chain growing, overhead incurred from using synchronization locks, and hardware aspects including memory-bandwidth saturation. According to our evaluation, the proposed scheme is effective in improving the HAC algorithm, achieving significant gains over the alternatives in terms of runtime and scalability.

Index Terms—Hierarchical clustering, unsupervised learning, parallelization, multi-threading, multi-core CPU

1 Introduction

Hierarchical agglomerative clustering (HAC) is one of the most popular clustering methods that can find hierarchical structure hidden in input data [1], [2], [3]. In many disciplines from astronomy to zoology, researchers frequently rely on HAC for data analysis.

As shown in Fig. 1, the idea of HAC is straightforward. Initially, the pairwise distance between all objects (each of which can be considered as a singleton cluster) is measured and stored in a distance matrix. The closest pair is then merged into a new cluster, and then the distance matrix is updated to include the distance between the new cluster and all other clusters. These merge and update operations are repeated until all of the objects belong to a single cluster.

The output from HAC is a binary tree called a *dendrogram*, which hierarchically represents the order of merge operations (Fig. 1). In a dendrogram, leaves represent data objects (e.g., objects 1-5 in Fig. 1), and an internal node corresponds to the merge operation of two child clusters. The leaves of the tree rooted at an internal node form a cluster (e.g., clusters A-D in Fig. 1). The height of an internal node is proportional to the distance between its child clusters. Flat clusters can be obtained from a dendrogram on cutting it by a horizontal line (e.g., the line in Fig. 1b gives clusters B and C).

To perform HAC, one needs to define the notion of distance, or so-called *linkage*, between two clusters. For the single linkage, the inter-cluster distance is defined as that between the endpoints of the shortest line connecting two clusters. For the complete linkage, the inter-cluster distance is defined as that between the endpoints of the longest straight line connecting two clusters. For the average linkage, the average distance is computed between cluster members. For the centroid linkage, the distance between the centers of two clusters is defined as the inter-cluster distance.

Despite the simplicity of the underlying algorithm, running HAC on large datasets is challenging due to the limited scalability which originates from the need to maintain inter-cluster distance information on a global level.

The worst-case time complexity of a naïve implementation of the conventional HAC algorithm would be $O(n^3)$, since there are $O(n)$ iterations, each of which involves $O(n^2)$ operations to find the closest pair in the distance matrix. Optimization techniques were developed to reduce the complexity, and HAC schemes with quadratic time complexity are now available for the centroid [4], complete [5], average [6], and single [7] linkage methods.

In this paper, we propose a novel parallelization scheme for running HAC on multi-threaded shared-memory machines. Our approach is based on the NN-chain algorithm [8], [9], which exploits a chain structure in order to more efficiently manage inter-cluster proximity information.

In its original form, the NN-chain algorithm considers only one chain at a time. However, given that maintaining a chain corresponds to searching for clusters that can be merged, there exist ample opportunities for parallelization. Our approach not only exploits these opportunities but further mitigates the impact of starvation and deadlock events which pose a challenge to effective parallelization of the

• *The authors are with the Department of Electrical and Computer Engineering, Seoul National University, Seoul 151-744, Republic of Korea. E-mail: dragwon@snu.ac.kr, sryoon@snu.ac.kr.*

Manuscript received 12 Sept. 2013; revised 25 Aug. 2014; accepted 26 Aug. 2014. Date of publication 4 Sept. 2014; date of current version 7 Aug. 2015. Recommended for acceptance by F. Mueller.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2014.2355205

1045-9219 © 2014 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

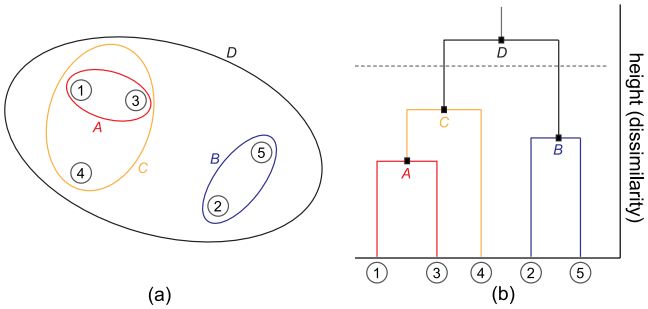


Fig. 1. Hierarchical agglomerative clustering. (a) example data. (b) resulting dendrogram.

NN-chain algorithm. In-depth analysis of our parallelization scheme reveals its theoretical performance limits and ideal thread configurations for maximum performance. According to our experiments, the proposed method completes hierarchical clustering 3.09-51.79 times faster than the alternatives. The results of our analysis demonstrate the effectiveness of the proposed method and suggest directions for further improvements.

The remainder of this paper is organized as follows: Section 2 reviews fundamental materials on hierarchical clustering and describes the NN-chain algorithm. The details of the proposed multi-threaded HAC methodology are explained in Section 3. Section 4 presents the experimental results, and Section 5 concludes the paper. Additional details and related work can be found in the supplementary file, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2014.2355205.

2 Background

The scope of this paper is on the parallelization of HAC for multi-threaded shared-memory machines. In this section, we present background materials on HAC and review related work, in order to facilitate further descriptions of the proposed methodology and to put it in the proper context. The supplementary file, available online, contains an additional survey of clustering methods and parallelization techniques.

2.1 Inter-Cluster Distance Measurement

Given two data objects i and j, let $\delta(i, j)$ denote the distance between i and j. For cluster analysis, we need to define how to measure the distance between two clusters, since clusters in general contain more than one object. In this context, the term *linkage* refers to the method for measuring the inter-cluster distance. The same HAC algorithm will produce different results if different linkages are used.

Let I and J denote two distinct clusters and $d(I, J)$ the distance between them. Widely used linkages include the following [1], [2], [3]:

$$\text{single:} \quad d(I,J) = \min_{i\in I, j\in J} \delta(i,j), \tag{1}$$

$$\text{complete:} \quad d(I,J) = \max_{i\in I, j\in J} \delta(i,j), \tag{2}$$

$$\text{average:} \quad d(I,J) = \frac{1}{|I||J|}\sum_{i\in I}\sum_{j\in J}\delta(i,j), \tag{3}$$

$$\text{centroid:} \quad d(I,J) = \delta(I_c, J_c), \tag{4}$$

$$\text{Ward's:} \quad d(I,J) = \varepsilon(I\cup J) - \{\varepsilon(I) + \varepsilon(J)\}, \tag{5}$$

TABLE 1
Coefficients for the Lance-Williams Formula [10]

Linkage	α_i	α_j	β	γ
single	$1/2$	$1/2$	0	$-1/2$
complete	$1/2$	$1/2$	0	$1/2$
average	$\frac{\lvert I\rvert}{\lvert I\rvert+\lvert J\rvert}$	$\frac{\lvert J\rvert}{\lvert I\rvert+\lvert J\rvert}$	0	0
centroid	$\frac{\lvert I\rvert}{\lvert I\rvert+\lvert J\rvert}$	$\frac{\lvert J\rvert}{\lvert I\rvert+\lvert J\rvert}$	$-\frac{\lvert I\rvert\lvert J\rvert}{(\lvert I\rvert+\lvert J\rvert)^2}$	0
Ward's	$\frac{\lvert I\rvert+\lvert K\rvert}{\lvert I\rvert+\lvert J\rvert+\lvert K\rvert}$	$\frac{\lvert J\rvert+\lvert K\rvert}{\lvert I\rvert+\lvert J\rvert+\lvert K\rvert}$	$-\frac{\lvert K\rvert}{\lvert I\rvert+\lvert J\rvert+\lvert K\rvert}$	0
WPGMA	$1/2$	$1/2$	0	0
WPGMC	$1/2$	$1/2$	$-1/4$	0

where I_c represents the centroid (i.e., mean vector) of I, and $\varepsilon(\cdot)$ denotes the error sum of squares given by

$$\varepsilon(I) = \sum_{i\in I}\{\delta(i, I_c)\}^2. \tag{6}$$

The average and centroid linkages are also known as *unweighted pair group method with arithmetic mean* and *unweighted pair group method using centroids*, respectively. Other linkages include the *weighted pair group method with arithmetic mean* (WPGMA) and the median linkage, which is also known as the *weighted pair group method using centroids* (WPGMC).

Assuming that two clusters I and J have been merged into a new cluster during the HAC procedure, and given another arbitrary cluster K, it is possible to update the distance between the new cluster and K in $O(1)$ time using the Lance-Williams formula [10]:

$$\begin{aligned} d(I\cup J, K) = {} & \alpha_i d(I,K) + \alpha_j d(J,K) \\ & + \beta d(I,J) + \gamma\,|\,d(I,K) - d(J,K)|, \end{aligned} \tag{7}$$

where coefficients α_i, α_j, β, and γ vary depending on the linkage used, as listed in Table 1.

2.2 HAC by Chaining Nearest Neighbors (NNs)

Fig. 2 compares the conventional HAC with the NN-chain algorithm [8], [9]. The conventional HAC globally selects and merges the closest pair of clusters. In contrast, the NN-chain algorithm searches for the closest pair in a local structure termed a *chain*. The (locally) closest pair found in a chain is called the *reciprocal nearest neighbor* (RNN) pair. More formal definitions involved with the NN-chain algorithm will follow in Section 3.2.

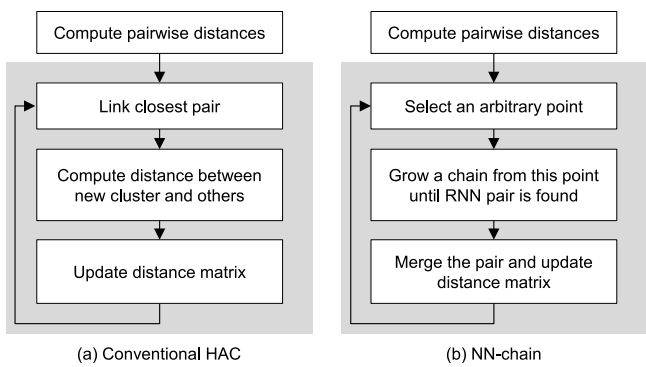


Fig. 2. Algorithm comparison.

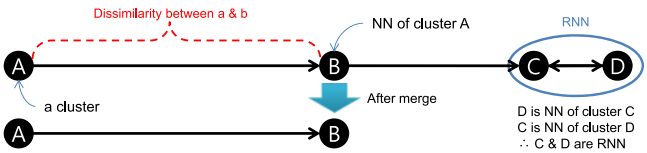


Fig. 3. Concept of NN-chain algorithm. The vertices A, B, C, and D on the chain represent clusters. The arrow from cluster A to cluster B indicates that B is the NN of A. Clusters C and D are the RNNs to each other. After merging the RNNs, the algorithm removes them from the chain and proceeds to the next iteration.

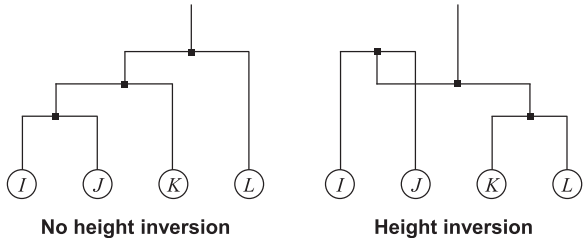


Fig. 4. Example of height inversion problem.

The NN-chain algorithm starts a chain from an arbitrary (singleton) cluster and grows the chain until discovering an RNN pair at the growing end of the chain (Fig. 3). The algorithm then merges the two clusters forming the discovered RNN pair into a new cluster and removes them from the chain. The algorithm then updates the pairwise distance information between this new cluster and the remaining clusters. If there remain vertices in the chain, the algorithm starts regrowing it from the point right next to the removed vertices. Otherwise, the algorithm starts a new chain from another randomly selected cluster. The algorithm terminates when all objects are grouped into a single cluster.

For n objects, the worst-case time complexity of the NN-chain algorithm is $O(n^2)$ [11], [12]. HAC needs to consider a total of $(2n-1)$ clusters: n singleton clusters (leaves in a dendrogram) and $(n-1)$ merged clusters (internal nodes in a dendrogram). Each cluster can be inserted into a chain only once, and can be removed from the chain only if that cluster and another cluster form an RNN pair. The two removed clusters are merged into another cluster and are never inserted to the same chain again. In order for a cluster to be inserted into a nonempty NN-chain, the cluster should be the NN of the tail cluster in the chain. Since it takes $O(n)$ to determine if a cluster is the NN of another and there are $O(n)$ clusters considered in HAC, the worst-case complexity of the NN-chain algorithm becomes quadratic in n.

The correctness of the NN-chain algorithm is guaranteed only when height inversion does not occur in the resulting dendrogram [11]. Usually, the height of any internal node in a dendrogram is higher than that of either child. If this is not the case, *height inversion* (Fig. 4) may occur in the dendrogram. Let us assume that there are three clusters I, J and K. Let $d(I,J)$ denote the distance between I and J and let $I \cup J$ represent a new cluster created by merging I and J. The height inversion does not occur if the following condition holds: If $d(I,J) < d(I,K)$ or $d(I,J) < d(J,K)$, then $d(I,J) < d(I \cup J, K)$. The occurrence of height inversion depends on the linkage used. If we use the average, complete, Ward's, or single linkage, then the above condition always holds [11]. The above condition is not always satisfied for the centroid linkage, and thus we cannot use it with the NN-chain algorithm.

2.3 Approaches to Parallelizing HAC

Olsen proposed a parallel HAC implementation based on a shared-memory abstract machine called *parallel random access machine* (PRAM) [13]. Rajasekaran presented a method based on concurrent-read concurrent-write PRAM processors [14]. Li et al. proposed a graph-theoretic algorithm based on exclusive-read exclusive-write PRAM processors [15]. Note that the PRAM-based algorithms are rather conceptual and hard to implement in practice.

Rasmussen and Willett implemented the SLINK algorithm [7] (a time-optimal version of the single linkage) and the Ward's-linkage-based HAC algorithm [16] in single-instruction multiple-data (SIMD) array processors [17]. Shalom et al. parallelized the conventional HAC algorithm on general-purpose graphic processing units (GPGPUs) [18]. Du and Lin parallelized the single-linkage-based HAC algorithm based on message passing interface (MPI) for multiple-instruction multiple-data environments [19]. For gene-expression analysis [20], [21], they achieved up to 25 times speed up over the sequential version using 48 processors. Rose described a deterministic-annealing algorithm suitable for parallelizing hierarchical clustering [22].

In computational biology and bioinformatics, the problem of phylogenetic-tree construction [23] is closely related to HAC. A genetic evolution history can be represented by a dendrogram-like phylogenetic tree. The popular neighbor-joining algorithm [24] is in essence a greedy HAC approach and has been parallelized using MPI [25] and GPGPU [26] techniques. The randomized accelerated maximum likelihood method [27] is another phylogenetic-tree construction approach that has been parallelized in various ways [27], [28], [29].

More comprehensive reviews of the parallelization attempts for accelerating HAC are available in [13], [14]. Some of these existing approaches will be compared with our method in Section 4.

3 Proposed Methodology

3.1 Motivation and Overview

Consider the example dendrogram in Fig. 1b. Since merging objects 1 and 3 is independent of merging objects 2 and 5, we can construct the left subtree (objects 1, 3, and 4) and the right subtree (objects 2 and 5) in parallel. Although some complications reside under the hood, this type of parallelism is what we can exploit to accelerate hierarchical clustering.

As informally shown in Fig. 5a, each iteration of sequential HAC algorithms merges only one pair of closest clusters at a time. Conventional parallel HAC algorithms (Fig. 5b) parallelize the process of finding the closest pair through local searches and parallel reduction, but still find only the single globally closest pair in each iteration. Fig. 6 depicts the idea of nearest-neighbor chaining. The NN-chain algorithm provides more parallelization opportunities than existing parallel algorithms since it can consider the merging of multiple pairs by growing one chain per thread simultaneously.

Fig. 7 shows the overall flow of the proposed approach. The two main tasks in our approach are growing NN chains

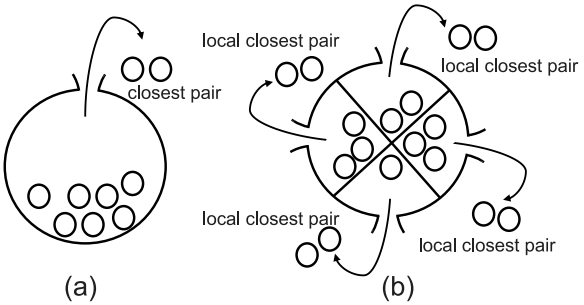


Fig. 5. Conventional HAC: (a) sequential (b) parallel.

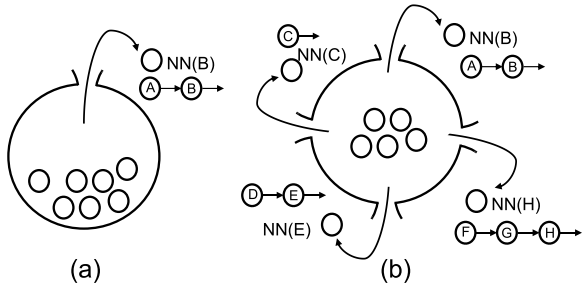


Fig. 6. NN-chain HAC: (a) sequential (b) parallel.

and updating distance information. This section explains each of these tasks in detail, beginning with definitions and assumptions.

Note that the exactness of the sequential HAC algorithm is preserved after parallelization by our NN-chain-based approach.

3.2 Definitions and Assumptions

3.2.1 Basic Definitions

Definition 1. *Given a cluster denoted by A, $nn(A)$ is the nearest neighbor of A,* i.e., *the cluster closest to A.*

Definition 2. *Clusters A and B are called a* reciprocal NN *pair if and only if $nn(A) = B$ and $nn(B) = A$.*

Definition 3. *A* chain *is a directed graph $G = (V, E)$ in which V is a set of vertices (each corresponding to some cluster) and $E = \{v \rightarrow w | v, w \in V \text{ and } C_w = nn(C_v)\}$, where C_v and C_w denote the clusters corresponding to v and w, respectively.*

Definition 4. *Given a chain $v \rightarrow \cdots \rightarrow s \rightarrow t$, vertices s and t are called* second-to-terminal *and* terminal, *respectively.*[1]

3.2.2 Thread Partitioning versus Pooling

Let us assume that a total of N threads are available for parallelization. We define *thread partitioning* as follows: Out of N threads, some threads are assigned to the task of chain growing and the remaining threads are assigned to updating the distance matrix. The role of a thread does not change. An alternative is *thread pooling*, namely utilizing N threads without distinction: A thread is used either for chain growing or distance matrix-updating as needed, and its role changes dynamically.

As presented in Section 4, thread partitioning is better suited to our methodology. In the description of the proposed approach, we therefore assign N_g out of N threads to growing chains (the right flowchart in Fig. 7), where each of the N_g threads is dedicated to the growth of a single chain. We then utilize the remaining $N_u (= N - N_g)$ threads to update the distance matrix (the left flowchart in Fig. 7).

3.3 Data Containers

The basic processing unit of the algorithm is a cluster. The pairwise distance between clusters is stored in the matrix **M**. Multiple clusters form a chain. We use four types of data containers to store clusters and chains.

1. When there exists only one vertex in the chain, no second-to-terminal vertex is defined.

1) `cluster_list` stores clusters that do not belong to any chain. It is initially filled with singleton clusters and later contains larger clusters created by merging RNN pairs. When we need to insert a cluster stored in `cluster_list` back into a chain, we remove the cluster from `cluster_list`, thus ensuring that a cluster exists in either a chain or `cluster_list`.
2) `dependency_queue` is maintained for each cluster and is indexed by cluster ID. For instance, `dependency_queue[`A`]` contains the chains waiting for cluster `A` to become part of an RNN pair and be removed from the chain. For a terminal vertex t in a chain, if $nn(C_t)$ does not belong to this chain, then it cannot be further grown. We add it to `dependency_queue[`$nn(C_t)$`]` where its processing is ceased.
3) `chain_queue` stores the chains that have been removed from `dependency_queue` because they no longer have dependency on other chains.
4) `update_queue` queues the chains to be shortened by removal of the RNN pairs. For each of such chains, **M** needs to be updated.

Note that, at any moment, a cluster either exists in a chain or is stored in `cluster_list`. A chain that is not assigned to a working thread exists in either `chain_queue`, `dependency_queue`, or `update_queue`.

3.4 Basic Chain Operations

A chain $v \rightarrow \cdots \rightarrow r \rightarrow s \rightarrow t$ is grown or shrunk only from the terminal vertex t. Depending on the identity of $A = nn(C_t)$, the chain can be either grown or shrunk. If $A \neq C_s$, the chain grows to $v \rightarrow \cdots \rightarrow t \rightarrow u$, where $C_u = A$. On the other hand, if $A = C_s$, then the chain shrinks to $v \rightarrow \cdots \rightarrow r$, triggering the following operations:

- We remove the RNN pair (C_s and C_t) from the chain and insert the resulting shrunken chain into `chain_queue`.
- We combine C_s and C_t into a new cluster and insert it into `cluster_list` after updating **M** to incorporate the distance between the new cluster and the other clusters.
- C_s and C_t are invalidated and no longer stored since only the new cluster will be used henceforth.
- We remove those chains that are stored in `dependency_queue[`C_s`]` and `dependency_queue[`C_t`]`, since there exist no dependency of these chains on C_s and C_t after invalidation. We then move these chains to `chain_queue`.

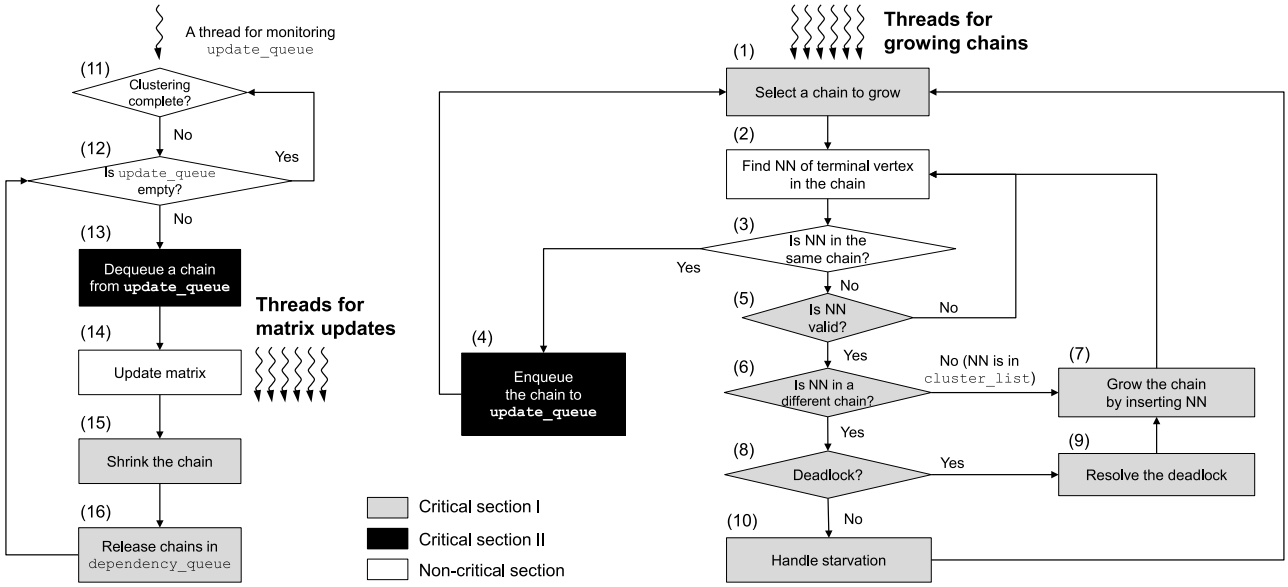


Fig. 7. Overall flow of the proposed approach.

3.5 Details of Each Step in Fig. 7

Step 1: We select an arbitrary cluster from `cluster_list` from which to start a new chain. If `cluster_list` is empty, we dequeue a chain from `chain_queue`. Note that starting from a chain in `chain_queue` rather than a cluster in `cluster_list` does not affect the correctness of the algorithm, since in both ways we are starting from an arbitrary cluster independent of the others.

Step 2: We find $nn(C_t)$ for the terminal vertex t in the current chain. $nn(C_t)$ may exist either in the current chain or in the other chains. If $nn(C_t)$ exists in the current chain, then $nn(C_t)$ must be C_s, where s is the second-to-terminal vertex in the current chain (otherwise, t cannot be terminal).

Steps 3-4: If $nn(C_t) = C_s$, this implies that we have found an RNN pair, which will become a new cluster. We enqueue the current chain (without shrinking it yet) to `update_queue` in step 4 and go back to step 1.

Step 5: If the $nn(C_t)$ cluster found in step 2 is marked invalid (i.e., it was already merged into a new cluster), then we return to step 2. Otherwise, we move to step 6.

Steps 6-7: Even though $nn(C_t)$ is valid, if it is already included in another chain, then we cannot include $nn(C_t)$ in the current chain. If this is the case, then we move to step 8. Otherwise, the $nn(C_t)$ must be a cluster in `cluster_list`. In step 7, we remove that chain from `cluster_list` and then grow the current chain by inserting $nn(C_t)$.

Steps 8-9: If the $nn(C_t)$ cluster found in step 2 is valid (i.e., has not been previously merged) but belongs to another chain, then this results in either starvation or a deadlock. Step 8 checks if there is a deadlock. If so, we resolve it in step 9 according to the method described in Section 3.6. Otherwise, we proceed to step 10.

Step 10: As stated above, reaching this step implies that there is starvation in the current chain. It cannot be grown unless $nn(C_t)$ in another chain is removed therefrom. Thus, we insert the current chain to `dependency_queue` $[nn(C_t)]$ and go back to step 1.

Steps 11-14: These steps are toward updating the distance matrix in parallel with growing chains. In steps 11-12, one of the N_u threads monitors `update_queue`. In steps 13-14, a chain with an RNN pair inside is dequeued, and the N_u threads update $\mathbf{M}$ in parallel for the new cluster represented by the RNN. Consider the example in Fig. 8. If clusters A and B are merged, then the distance values inside the dotted box need to be updated. Multiple threads can do this update in parallel, although note that we cannot update $\mathbf{M}$ for multiple RNN pairs simultaneously. For instance, if clusters E and F in Fig. 8 are also to be merged, then the values in the solid box, as well as those in the dotted box, need to be updated. However, the values inside the two boxes may not be updated at the same time, in order to maintain data coherence.

Step 15: We shrink the chain dequeued in step 13 and insert the resulting new cluster into `cluster_list`. If this chain still has a vertex, then it is inserted into `chain_queue`, so that it can be picked up later in step 1.

Step 16: Let A and B denote the two clusters that form the RNN pair removed from the chain in step 15. We remove the chains stored in `dependency_queue`$[A]$ and `dependency_queue`$[B]$ and insert these chains into `chain_queue`.

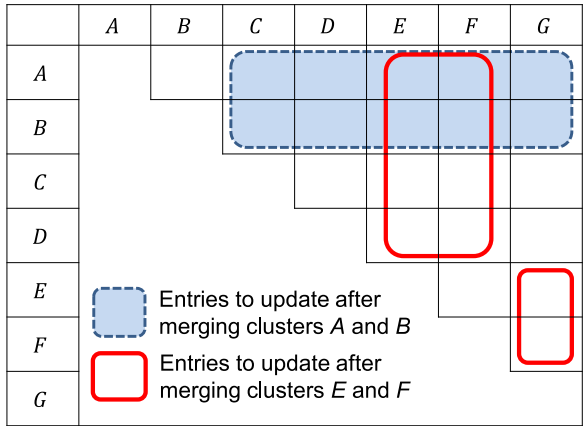


Fig. 8. Example dissimilarity matrix (A-G: cluster IDs).

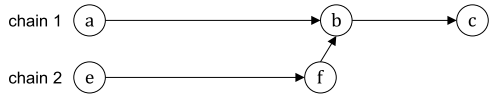


Fig. 9. Example of starvation.

3.6 Starvation and Deadlock

The existence of starvation (Fig. 9) and deadlock (Fig. 10) events poses a challenge to effective parallelization of the NN-chain algorithm. To avoid starvation and deadlocks, we partition the algorithm steps into critical and non-critical sections. While this affects the parallelization efficiency, partitioning is essential to ensure the correctness of the algorithm.

Starvation occurs if a chain has dependency on another chain. For example, in Fig. 9, chain 2 cannot grow unless b in chain 1 is removed from it. We put chain 2 to `dependency_queue[`C_b`]`, and wait for C_b to be removed from chain 1 and to form a new cluster.

A deadlock occurs if there is mutual dependency between two chains. The example in Fig. 10 shows the situation where $nn(C_c) = C_d$ and $nn(C_d) = C_c$. Here, we have found $nn(C_c) = C_d$, but it already belongs to chain 2. Thus, we cannot insert d into chain 1. Likewise, chain 2 cannot grow to c, since it belongs to chain 1. To address this deadlock situation, we remove d from chain 2 and insert d into chain 1.

Note that it is impossible to have mutual dependencies among three chains. Consider the example shown in Fig. 11. Assume that $d(C_c, C_d)$, $d(C_d, C_i)$, and $d(C_i, C_c)$ are all different. Although if such were the case, the scenario shown in the figure cannot happen, since it premises the following inequality:

$$d(C_c, C_d) > d(C_d, C_i) > d(C_i, C_c) > d(C_c, C_d).$$

Therefore, $d(C_c, C_d)$, $d(C_d, C_i)$ and $d(C_i, C_c)$ should be equal if we are to see the scenario depicted in the figure. If this is the case, then by design of the algorithm, chains 1 and 2 are put into a deadlock momentarily and then resolved, while chain 3 suffers from starvation briefly and can start growing after the merging of C_c and C_d.

3.7 Remarks

Step 1, steps 5-10, and steps 15 and 16 altogether form a critical section, as marked in Fig. 7, due to the following: First, we disallow multiple threads to select the same cluster or chain. To do otherwise would create redundancy and waste of resources, or lead to inconsistency. Second, we should remove the possibility of this cluster being included in other chains by another thread on the selection of an arbitrary cluster in step 1. Third, we need to maintain the coherency of the `chain_queue`. A thread running step 1 can dequeue a chain from `chain_queue`, while a thread running steps 15-16 enqueues a chain to `chain_queue` simultaneously.

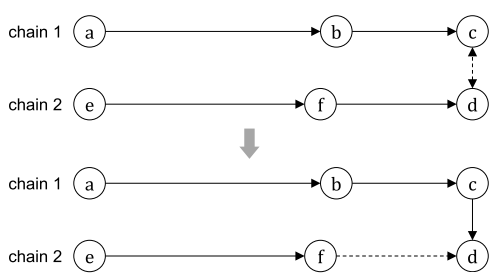


Fig. 10. Example of deadlock (mutual dependency) and its resolution.

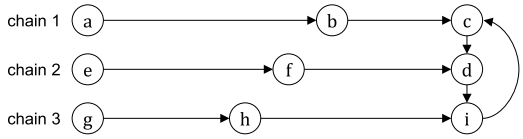


Fig. 11. Dependency among three chains (note that this cannot happen in reality).

Ideally, an invalid cluster should not be searched as $nn(C_t)$ in step 2. However, steps 2 and 16 could be executed by different threads simultaneously, and it is possible that step 2 finds a cluster as $nn(C_t)$ before step 16 marks this cluster as invalid. Preventing this from happening requires creating another critical section, which may lower performance. We thus choose to use the approach as described in Section 3.5.

3.8 Complexity Analysis

Due to the need for storing the pairwise distance matrix, the worst-case space-complexity of the proposed algorithm is $O(n^2)$, where n is the number of input objects. The worst-case time complexity is $O(n^2/P)$, where P is the degree of parallelization.

For a system with $N = (N_g + N_u)$ threads (i.e., N_g threads for growing chains and N_u for updating the distance matrix), P is given by

$$P = \alpha(N_g + N_u), \tag{8}$$

where α is a non-deterministic factor whose range is $\alpha_L \leq \alpha \leq 1$ with the low limit α_L given by

$$\alpha_L = \frac{1 + N_u}{N} = 1 - \frac{N_g - 1}{N}. \tag{9}$$

Note that $\alpha = 1$ when all threads in the system are utilized, whereas $\alpha = \alpha_L$ when N_u threads work on updating the matrix and only one thread out of N_g is active for chain growing. This can happen when all the N_g threads look for the same cluster (say cluster C_*) as the NN of their terminal clusters. In this case, C_* is inserted into only one chain, and the other chains encounter starvation and are inserted into `dependency_queue`. The threads in charge of growing these chains become to spend $O(n)$ time on redundantly searching for C_*, and the situation is equivalent to using only a single thread. That is, $P = \alpha(N_g + N_u) = N_u + 1$, which gives $\alpha_L = (1 + N_u)/N$.

3.8.1 Ideal Ratio of N_g to N_u

The choice of how to partition N into N_g and N_u naturally affects the performance. For the sake of our theoretical analysis, we assume that there is no dependency among chains (e.g., no starvation) and we do not consider hardware effects such as memory bandwidth saturation.

As explained in Section 2.2, HAC considers a total of $(2n - 1)$ clusters. In the NN-chain algorithm, every cluster needs to be inserted into a chain once. Each insertion

TABLE 2
Datasets Used for Experiments

ID	Description	# obj.	# dim.
PROT	Protein microenvironment	100,000[a]	264 [30]
SMD	Prostate cancer genes	41,421	18 [31]
UCI1	MAGIC gamma telescope	19,020	10 [32]
UCI2	Relative location of CT slices	53,500	385 [32]
UCI3	KEGG metabolic reaction net	65,554	28 [32]
UCI4	Pseudo-periodic time series	100,000	10 [32]
UCI5	MiniBooNE particle identification	130,064	50 [32]
RAND	Random integers bet'n $[0, 255]$	100,000	128

[a]*The original data set has 1,992,567 vectors; 100,000 vectors were randomly chosen.*

operation takes $O(n)$ time. The process of checking the RNN relationship incurs a total of $(n-1)$ merge operations, each of which also takes $O(n)$ time. Assuming that N_g threads are in charge of chain growing, it therefore takes $O(3n^2/N_g)$ time.

For each of the $(n-1)$ merge operations, $O(2n)$ time is required to access the distance between each of the two merged clusters and the other clusters. Thus, with N_u threads for updating the distance matrix, the matrix update procedure takes $O(2n^2/N_u)$ time in total.

It would be ideal if the chain-growing and matrix-updating parts of the algorithm take the same amount of time, thus incurring no idle time. In this respect, the ideal ratio between N_g and N_u would be $N_g : N_u = 3 : 2 = 1.5 : 1$. Plugging this ratio into Eq. (9) gives $\alpha_L = 1/N + 0.4$, which sets the lower bound of P to $1 + 0.4N$. Although in reality, the existence of starvation and hardware limitations negatively affect the scalability, as demonstrated in Section 4.9.

4 Results and Discussion

4.1 Setup and Notations

We implemented our approach in C++ and compiled it with icc 13.0 under an Ubuntu 12.04 linux environment. The experiments are carried out on a machine equipped with two Intel Xeon E5-2620 (2.0 GHz, 6 cores, QuickPath Interconnect [33] 7.2 GT/s) CPUs with 64 GB memory. In experiments for measuring runtime, we repeat every run twenty times and take the average value. We exclude the time for calculating pairwise distance, which is common among all the compared methods.

Table 2 lists the datasets used in our experiments. Due to the space limit, except for Figs. 18, 20, and 21 this section presents only the plots for the largest data (PROT). The observations and discussions regarding PROT presented in this section also hold for the other data.

To analyze the design decisions made and uncover the factors that limit the performance of our approach (Sections 4.2, 4.3, 4.4, 4.5, 4.6, and 4.7), we also prepared modified versions of the proposed algorithm, as listed in Table 3. For the performance comparison in Section 4.8, we utilized the existing HAC methodologies in Table 3. More details of each of the methods listed in the table will be presented later with relevant experimental results.

4.2 Runtime Profiling

To locate the parallelization targets, we carry out runtime profiling of SEQ, the sequential version of the proposed algorithm implemented based on [34]. For profiling, we use a single thread to run the SEQ algorithm and measure the runtime over input data (the PROT dataset) containing different numbers of objects (from 10 to 100K with increment of 10K). Fig. 12 shows the result on a logarithmic scale. According to our experiments, the operations to grow chains and update the distance matrix account for the majority of the runtime, which is also observed for the other datasets besides PROT. This profiling result justifies our decision of choosing these two tasks for parallelization, as described in Section 3.

In Table 3 and the figure captions in this section, IDEAL represents the hypothetical parallelization method whose runtime is exactly that of SEQ divided by the number of threads available for parallelization. In terms of Eq. (8), $\alpha = 1$ and $P = N$ for IDEAL.

4.3 Performance-Limiting Factors

To facilitate further explanation of our experimental results, we discuss the factors that affect parallelization and limit the performance of the proposed algorithm. These factors originate from both algorithmic (e.g., the chain lock, the matrix lock, and the dependency between chains leading to starvation) and hardware aspects (e.g., saturation in the memory bandwidth).

Recall that there exist two types of locks used by the proposed algorithm (Fig. 7). One is called the *chain lock*, which creates 'Critical section I' for chain growing; the other is called the *matrix lock*, which introduces 'Critical section II' for matrix-updating.

We need the chain lock to ensure that a cluster is inserted into only one chain. Allowing a cluster to be inserted into multiple chains will incur redundancy because the tails of

TABLE 3
The Clustering Methods Tested and Compared in Section 4

Acronym	Description	Appearance
PROPOSED	proposed method (thread-partitioning, matrix lock, chain lock)	Fig. 13, 14, 16, 17, 18, 19, 20, 21
SEQ	sequential version of proposed method (no multithreading, no lock)	Fig. 12, 13, 14, 18, 19, 20, 21
POOL	thread-pool version of proposed method	Fig. 13, 14
POOL_NO_MLOCK	thread-pool, no-matrix-lock version of proposed method	Fig. 13, 14
IDEAL	perfect parallelization (runtime: that of SEQ divided by # threads)	Fig. 13, 14, 19
GENERALSEQ	sequential version of priority-queue-based general HAC implementation [13]	Fig. 18, 20, 21
GENERALPAR	multithreaded version of GENERALSEQ [13]	Fig. 18, 20, 21
MTLB	linkage function included in the MATLAB package	Fig. 18, 20, 21
PARADIGGPU	GPGPU-parallelized implementation of paradigmatic HAC algorithm [18]	Fig. 18, 20, 21

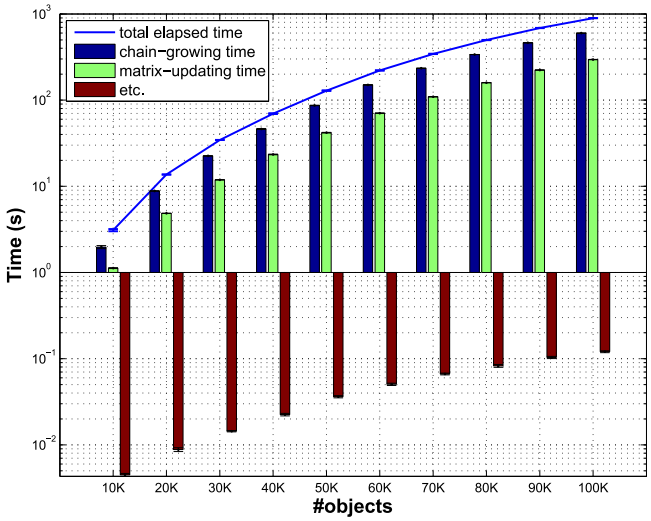


Fig. 12. Runtime profiling with a single thread used.

these chains will be identical. We expect that the overhead incurred by the chain lock would be relatively small because it takes $O(1)$ operations to insert a cluster into a chain.

We need the matrix lock since only one thread at a time is allowed to update the distance matrix for the sake of coherence (Fig. 8). The operations for updating the matrix take $O(n)$ time, and we expect that the overhead by using the matrix lock would be larger than that for using the chain lock.

Starvation occurs when the NN of the terminal cluster of a chain becomes to be part of another chain. In general, we cannot help encountering starvation since we grow multiple chains simultaneously.

Since clustering is a memory-intensive application, each thread in the system running the proposed approach will incur frequent memory access. As the number of threads increase for multi-threading, the memory bandwidth of the system may not fulfill the combined requests from multiple threads. This is another rate-limiting factor that hinders linear scale-up of the performance.

In the rest of this section, we present experimental results and related discussion for investigating the effect of each of these performance-limiting factors.

4.4 Effects of Locks and Starvation

We consider the modified versions of the proposed algorithm listed in Table 3. In particular, POOL uses thread pooling, while POOL_NO_MLOCK also uses thread pooling but is implemented without the matrix lock. POOL_NO_MLOCK will give incorrect results due to the absence of locking but is helpful for studying the performance-limiting factors.

Fig. 13a shows the runtime of the modified implementations for processing different numbers of objects in PROT. For comparison, the plot also includes the curves for other versions of the algorithm: sequential (SEQ), thread-pooling version with locks (POOL), thread-partitioning version with locks (PROPOSED), thread-pooling version without the matrix lock (POOL_NO_MLOCK), and ideality (IDEAL).

Obviously, using no lock can reduce the runtime, as is represented by the gap between POOL and POOL_NO_MLOCK. Still, there is a performance gap between POOL_NO_MLOCK and IDEAL. This gap is owing to other performance-limiting factors than the matrix lock, such as the chain lock and

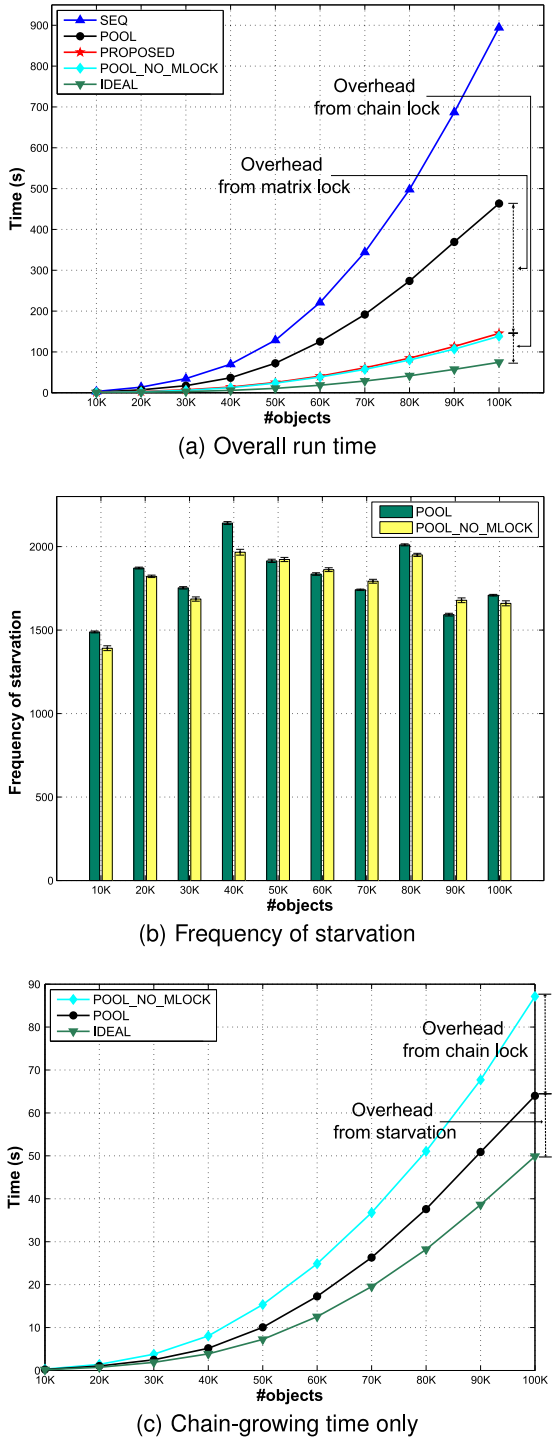
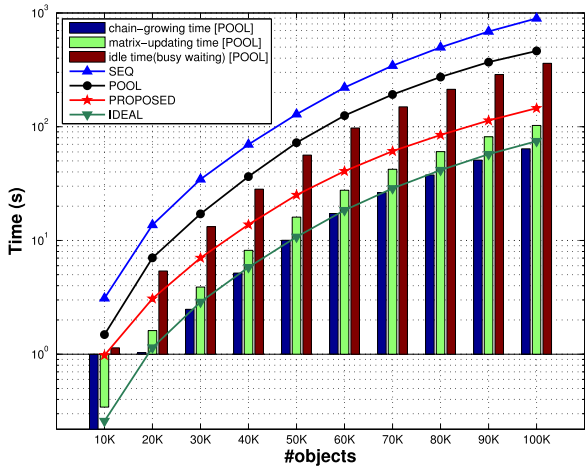


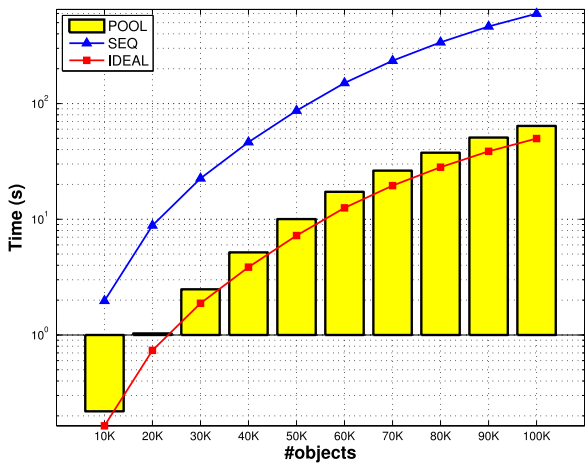
Fig. 13. Effects of omitting the matrix lock on the overall runtime, the frequency of starvation, and the chain-growing time.

starvation. The effect of omitting the matrix lock on starvation turns out to be insignificant, as shown in Fig. 13b, which shows the number of starvation occurrences for POOL and POOL_NO_MLOCK. Consequently, we conjecture that the difference between POOL and POOL_NO_MLOCK (Fig. 13a) is mainly due to the use of the chain lock.

To see the effect of not using the matrix lock on the chain growth, we separate the chain-growing portion from the overall runtime shown in Fig. 13a and plot the chain-growing time in Fig. 13c. Interestingly, the chain-growing time of POOL_NO_MLOCK is rather higher than that of POOL. Removing



(a) Overall run time



(b) Chain-growing time only

Fig. 14. Comparison of runtime between thread-pooling and thread-partitioning versions of the algorithm.

the matrix lock results in many threads pending on the chain lock, thus increasing the chain-growing time. In Fig. 13c, the difference between POOL_NO_MLOCK and POOL is thus due to the overhead incurred from the chain lock. The gap between POOL and IDEAL appearing in Fig. 13c is mainly owing to the chains suffering from starvation.

4.5 Thread Partitioning Improves Performance

Fig. 14a shows the runtime of different versions of the algorithm: sequential (SEQ), thread-pooling (POOL), and thread-partitioning (PROPOSED). The dataset used is PROT with the number of objects varying from 10 to 100K. These figures also include a curve representing ideality (IDEAL), which shows the lower limit of the runtime.

We observe that the thread-partitioning version outperforms the thread-pooling version by a large margin. To understand the root-cause of this performance improvement, we take a closer look at the runtime breakdown of POOL, as indicated by the three bars[2] corresponding to each object size in Fig. 14a: chain-growing time, matrix-updating time, and idle time (i.e., when waiting for the matrix

2. These breakdown times are separately measured and do not sum up to the total runtime labeled POOL because the matrix-update and the chain-growth tasks can occur concurrently in time.

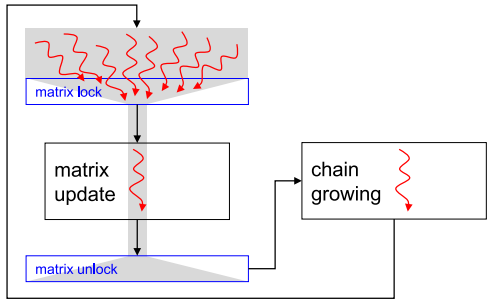


Fig. 15. Conceptual illustration of the problem of thread skewing in the thread-pooling version of the algorithm.

lock to be released). As expected, the overhead incurred from using the matrix lock is significant, and the idle time of threads awaiting resolution of the matrix lock is more than three times the combined matrix-updating and chain-growing time. In contrast, the chain-growth operation takes similar time to ideality as shown in Fig. 14b, suggesting that the overhead incurred from using the chain lock is negligible.

To summarize, the problem with the thread-pooling version is that too many threads are pending on the release of the matrix lock while many less threads work on chain growing, as informally illustrated in Fig. 15. We alleviate this problem of thread skewing by employing the thread-partitioning idea mentioned above. By dedicating some threads to chain growing and others to matrix-updating, we can reduce the problem of thread skewing, as demonstrated by the runtime result in Fig. 14.

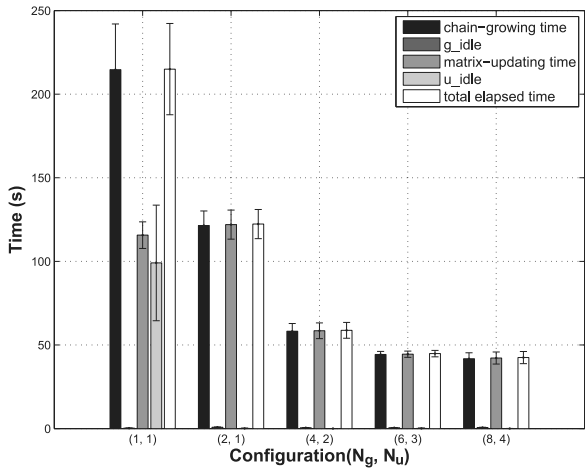
The question remains of how to choose the size of each thread partition, namely, how to distribute N threads between N_g and N_u. The next section presents experimental results pertaining to this issue.

4.6 Effects of Configuring Thread Partitions

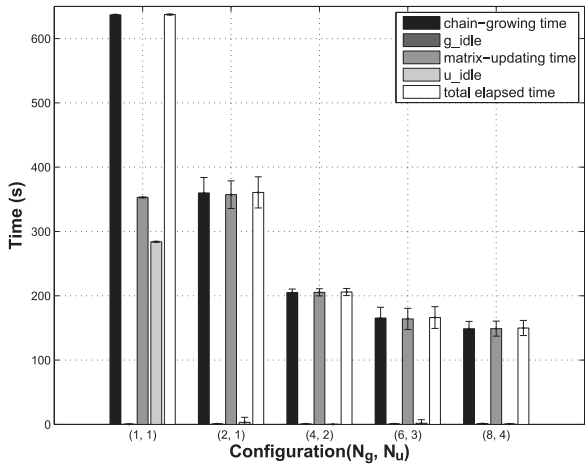
If the number of chains that need updates increases faster than the matrix-update rates, then threads in the chain-growing group can be put on hold due to the lack of chains available to grow. In the opposite case, `update_queue` becomes empty frequently and threads in the matrix-update group may become idle. We thus carried out experiments to determine the optimal configuration of (N_g, N_u) that minimizes the waiting time of both thread groups.

Fig. 16a shows the variation in runtime of the PROPOSED algorithm to process 60 K objects in PROT for five different (N_g, N_u) compositions: $(1,1)$, $(2,1)$, $(4,2)$, $(6,3)$, and $(8,4)$. Fig. 16b shows the same information for processing 100 K objects. For each configuration, the plot shows the total elapsed time and its breakdown: chain-growing time, matrix-updating time, the idle time of the chain-growing group (labeled 'g_idle'), and the idle time of the matrix-update group (labeled 'u_idle'). Ideally, the idle times should be zero, and the total elapsed time, chain-growing time, and matrix-update time should be identical.

For the $(1,1)$ configuration, we observe that there is significant idle time for the threads in the matrix-update group and that the chain-growing speed is slower than the matrix-update speed. Slow chain growing naturally reduces the rate of discovery of chains to be updated. The result is that the influx of RNN pairs to `update_queue` is slower than

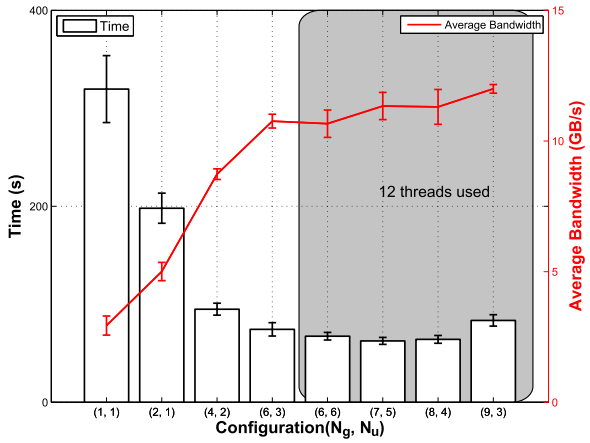


(a) #obj: 60K

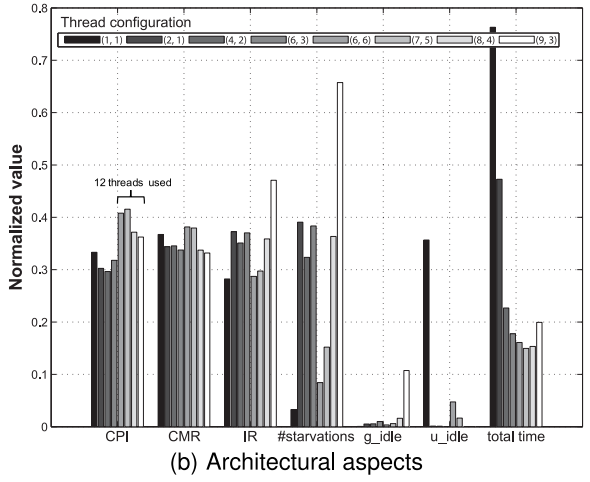


(b) #obj: 100K

Fig. 16. Runtime variation according to the (N_g, N_u) configurations.



(a) Time and bandwidth demands



(b) Architectural aspects

Fig. 17. Saturated memory bandwidth limits performance improvements from multi-threading (#obj: 70K).

the outflux. Consequently, due to the increased chance that `update_queue` becomes empty, the threads in the matrix-update group are likely to be idle.

To accelerate chain growth, we allocate one more thread to chain growing and try the $(2, 1)$ configuration. As is evident from the plot, this configuration successfully reduces the idle time of updating threads. The other configurations maintaining the $N_g = 2N_u$ relationship are also shown to be effective. The next section considers additional architectural aspects to further optimize the configuration of N_g and N_u.

4.7 Effects of Bandwidth Saturation

In Fig. 16, we observe that the runtime steeply decreases as the thread configuration goes from $(1, 1)$ via $(2, 1)$ to $(4, 2)$. In contrast, the relative improvements in runtime are diminished as we go from $(4, 2)$ via $(6, 3)$ to $(8, 4)$. To understand why, we carry out profiling of architectural aspects using Intel Vtune^{TM} Amplifier XE (http://software.intel.com).

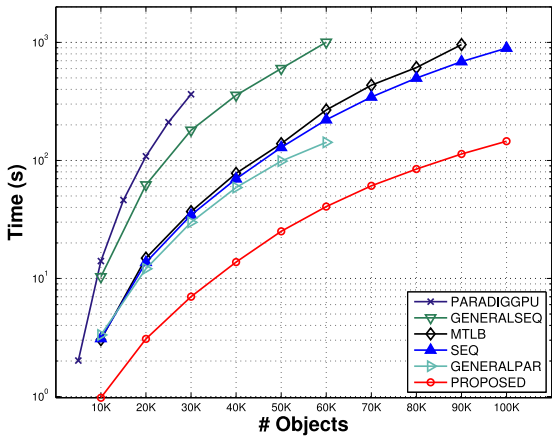
Fig. 17a shows how the runtime of PROPOSED and the average memory bandwidth utilization vary across eight different thread configurations. The input dataset used is PROT. As we go from $(1, 1) \rightarrow (2, 1) \rightarrow (4, 2)$, the average bandwidth utilization increases, and the runtime noticeably decreases. However, as we go from $(4, 2) \rightarrow (6, 3) \rightarrow (6, 6)$, the memory bandwidth becomes almost saturated, and the average bandwidth utilization fails to increase significantly. We believe that this saturation in memory bandwidth is the main cause of the saturated performance.

Fig. 17b shows how the number of chains in starvation and the architectural aspects including cycles per instruction (CPI), cache miss rate (CMR), and instructions retired (IR) vary over the eight thread configurations. The CPI values of the four configurations using all twelve threads available in the system [i.e., $(6, 6)$, $(7, 5)$, $(8, 4)$, and $(9, 3)$] are higher than those of the other configurations. This increase in CPI is understandable considering that the nearly saturated memory bandwidth will slow down data transfers required for executing instructions.

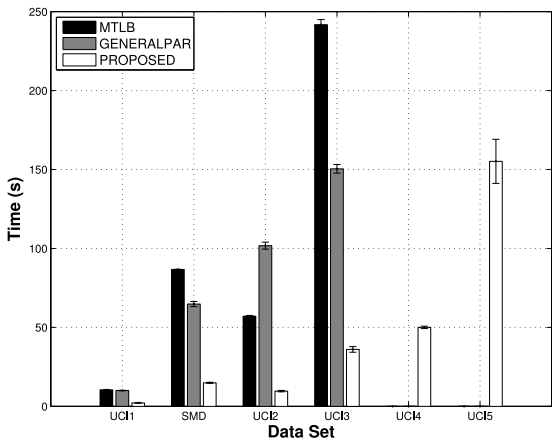
Among these four configurations, the CPI values of the $(8, 4)$ and $(9, 3)$ configurations are lower than those of $(6, 6)$ and $(7, 5)$. Indeed, allocating more threads for chain growing will increase the frequency of starvation and also increase IR, but CMR tends rather to decrease given that starvation corresponds to an attempt to access the same memory location.

In the $(2, 1)$, $(4, 2)$, and $(6, 3)$ configurations, the memory bandwidth is not saturated, and CPI can be kept small. This enables a high chain growth rate to be maintained, making the frequency of starvation relatively high.

Another key feature observed in Fig. 17a is that, for the configurations using all twelve threads, a greater proportion



(a) PROT ($N_g : N_u = 7 : 5$)



(b) Additional datasets ($N_g : N_u = 7 : 5$)

Fig. 18. Runtime comparison between the proposed method and other existing approaches.

of threads allocated to chain-growing leads to an increased utilization of memory bandwidth. This is understandable considering that each chain-growing thread on its own needs to explore $O(n)$ search space to search for the NN of the responsible chain, whereas multiple matrix-updating threads can collectively work on the $O(n)$ task of matrix-updating. Thus, using more chain-growing threads requires higher bandwidth. On the other hand, as observed earlier, allocating a small number of threads to chain-growing tends to leave matrix-updating threads busy waiting for the matrix lock. We therefore see a tradeoff between the idle time of matrix-updating threads and the required memory bandwidth.

Empirically, we find that the $(7, 5)$ configuration is the optimal choice in this particular case. Using this configuration, we can minimize the overhead incurred from the matrix lock and make the performance of the proposed approach (PROPOSED) nearly identical to that of the matrix-lock-free version (POOL_NO_MLOCK) in Fig. 13a. Note that this ratio of N_g to N_u ($7 : 5 = 1.4 : 1$) is close to the theoretical optimum ($1.5 : 1$), as explained in Section 3.8.

4.8 Comparison with Other Approaches

Fig. 18a shows the runtime of some various approaches to HAC: the sequential and the multi-threaded implementations of the proposed approach (labeled SEQ and PROPOSED, respectively), the sequential and the multi-threaded implementations of the priority-queue-based so-called *general* algorithm [13] (labeled GENERALSEQ and GENERALPAR, respectively), a GPGPU-based approach [18] (labeled PARADIGGPU), and the 2012b version of the MATLAB (http://mathworks.com) package (labeled MTLB). The linkage used is the average linkage in all the cases.

We include GENERALPAR in comparison as the best existing HAC parallel implementation in terms of exactness and compatibility with multiple linkage methods. Recently, GPGPU-based data processing is becoming popular, and we also include PARADIGGPU to see its effectiveness. We consider MTLB on the basis of its efficiency and popularity for data analysis. Additional discussions on GENERALPAR and its comparison to the proposed method are available in the supplementary file, available online.

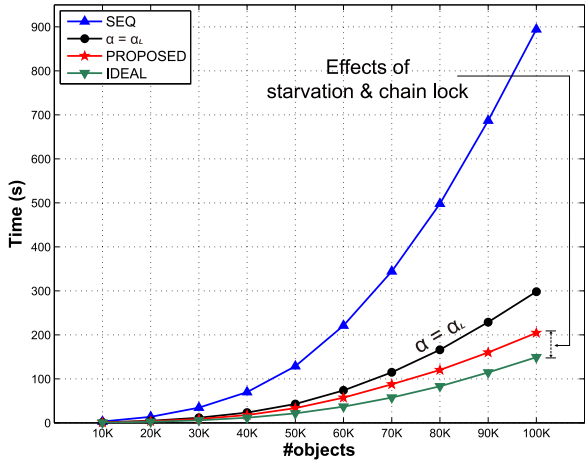
GENERALSEQ and GENERALPAR are implemented using a priority queue into which the NN of each cluster is inserted and sorted automatically in terms of the distance. The worst-case time complexity of both GENERALSEQ and GENERALPAR is $(n^2 \log n/P)$ where n is the number of objects and P represents the degree of parallelization (i.e., the number of threads) as mentioned in Section 3.8. The priority-queue-based implementations require additional $O(n^2)$ space to store a priority queue. Due to this increased space complexity, we can run GENERALSEQ and GENERALPAR for only up to 60 K objects.

PARADIGGPU utilizes massively parallel computing resources in GPGPU but the baseline algorithm (also known as the *paradigmatic* HAC algorithm) that PARADIGGPU parallelizes is suboptimal in the sense that the algorithm searches for the closest pair in every single iteration. The worst-case time complexity of the PARADIGGPU approach is $O(n^3/P)$. Moreover, GPGPU cards typically have relatively limited memory compared to multi-core CPU systems (the Nvidia GTX580 card used is equipped with 3 GB GDDR memory). In this regard, the PARADIGGPU approach tends to run slower than the alternatives and also has limited scalability (only up to 30 K objects in this particular experiment).

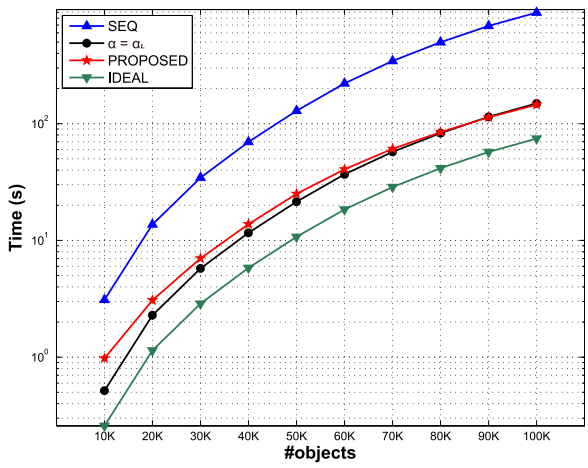
For the MTLB approach, we utilize the linkage function available in the MATLAB package. Turning on the built-in parallelization option in MATLAB does not provide noticeable change in the runtime, because the linkage function in MATLAB is provided as an unparallelized MATLAB executable (mex) function. The scalability of MTLB is limited by the memory installed in the system, and in our experiments, MTLB could process up to 90 K objects.

In the experiments shown in Fig. 18a, the PROPOSED approach substantially outperforms the alternatives in terms of runtime: 14.37-51.79 times faster than PARADIGGPU, 10.57-25.89 times than GENERALSEQ, 3.39-4.27 times than GENERALPAR, 3.09-8.44 times than MTLB, and 3.17-6.15 times than SEQ.

Fig. 18b shows the runtime of MTLB, GENERALPAR, and PROPOSED on processing each of the six different datasets listed in Table 2. We repeat each measurement twenty times and show the average runtime in Fig. 18b. For the UCI4 and UCI5 datasets, the alternatives do not terminate, and their runtime is not shown in the plot. For the other datasets, PROPOSED outperformed GENERALPAR and MTLB by a large margin: PROPOSED is 4.17-10.56 times faster than GENERALPAR and 4.84-6.71 times than MTLB.



(a) $(N_g, N_u) = (4, 2)$



(b) $(N_g, N_u) = (7, 5)$

Fig. 19. Comparison of empirical results and theoretical bounds on runtime: (a) no memory bandwidth saturation (b) memory bandwith saturation occurs.

4.9 Comparison with Theoretical Bounds

Fig. 19 shows how the PROPOSED algorithm performs with respect to the theoretical bounds developed in Section 3.8. Recall that the performance of PROPOSED depends on P, and at the lower bound of P, we obtain the worst performance, occurring for when $\alpha = \alpha_L$.

Using all of the available twelve threads in our experimental system saturates the memory bandwidth, as shown in Fig. 17a. Since the theoretical complexity analysis described in Section 3.8 assumes no such hardware effects, we used only six threads to avoid memory bandwidth saturation in our experiments comparing the empirical and theoretically predicted runtime.

Fig. 19a shows the empirical runtime of SEQ and PROPOSED along with the theoretical lower (i.e., IDEAL) and upper (i.e., the curve labeled $\alpha = \alpha_L$) bounds of the runtime. The thread configuration used is $(N_g, N_u) = (4, 2)$. We observe that the runtime of PROPOSED lies between the bounds. The additional time over the ideality is mainly due to overhead incurred from starvation and chain-lock.

When we use all of the twelve threads, the runtime of PROPOSED is higher than that predicted by the theoretical analysis, as shown in Fig. 19b. This result demonstrates the limitation of our theoretical analysis of the performance bound. Considering the effects of hardware limitations may allow more accurate analysis, but it would be difficult to estimate such effects, which vary between datasets and from one machine to another.

4.10 Practical Guideline for Choosing $N_g : N_u$

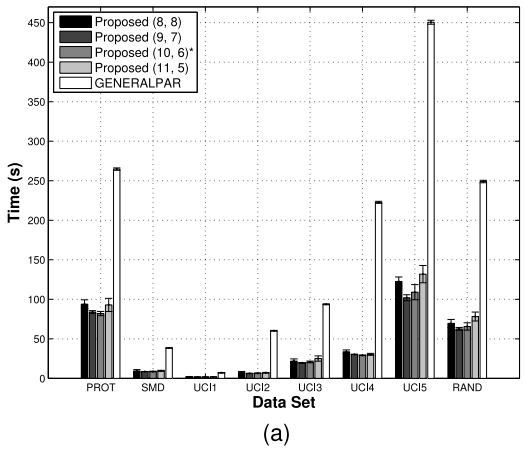
The theoretically ideal ratio of N_g to N_u, namely $N_g : N_u = 1.5 : 1$, holds when there is no limitations from hardware perspectives (e.g., memory bandwidth saturation) and software perspectives (e.g., starvation and deadlock). The thread ratio used in the previous sections is based on this theoretical analysis.

In reality, due to the existence of various nonidealities, the best-working ratio would change from machine to machine. To come up with a practical guideline for deciding the ratio for reasonable thread partitioning results, we perform additional experiments. In addition to the experimental setup described in Section 4.1, we try two more machine configurations (Table 4). These two configurations are called CONFIG1 and CONFIG2. In addition, we test a dataset named RAND that contains 100,000 vectors of 128 randomly integers randomly sampled from [0, 255].

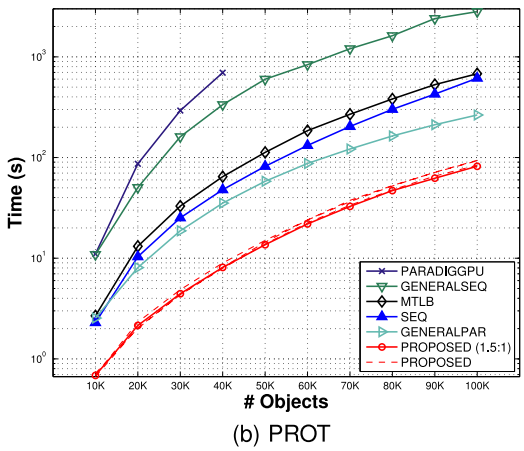
Figs. 20 and 21 show the results from our experiments using CONFIG1 and CONFIG2, respectively. For both, we test the proposed method and GENERALPAR with the eight datasets listed in Table 2. For the proposed method, we try different ratios of N_g to N_u: for CONFIG 1, $(N_g, N_u) = (8, 8)$, $(9, 7)$, $(10, 6)$, and $(11, 5)$; for CONFIG2, $(N_g, N_u) = (16, 16)$, $(17, 15)$, $(18, 14)$, $(19, 13)$, $(20, 12)$, and $(21, 11)$. Among these ratios, note that $(10, 6)$ and $(19, 13)$ are the closest to the

TABLE 4
Specification of Machine Configurations Used in Section 4.10 (Figs. 20 and 21)

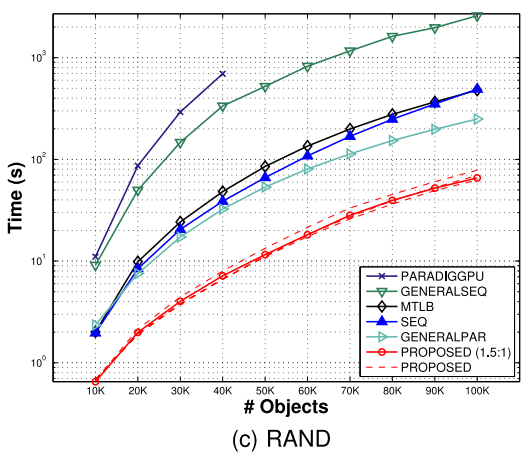
Configuration	CONFIG1	CONFIG2
CPU	Intel Xeon E5-2650 (2.0 GHz, 8 Cores, QPI 8 GT/s) ×2	Intel Xeon E5-4620 (2.2 GHz, 8 Cores, QPI 7.2 GT/s) × 4
RAM	256 GB	512GB
Max memory bandwidth	51.2 GB/s × 2 (ways) = 102.4 GB/s	42.6 GB/s × 4 (ways) = 170.4 GB/s
Maximum FLOPs/cycle	8	8
Maximum FLOPs	256 GFLOPs	563.2 GFLOPs
Memory bandwidth/FLOPs	102.4/256 = 0.4	170.4/563.2 = 0.28
OS	Ubuntu 12.04	Ubuntu 12.04
Compiler	icc 13.0	icc 13.0
GPU	Nvidia Geforce Titan, 6GB GDDR	Nvidia Geforce Titan, 6GB GDDR



(a)

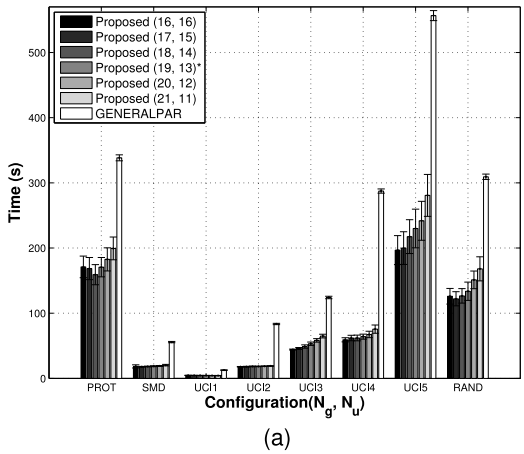


(b) PROT

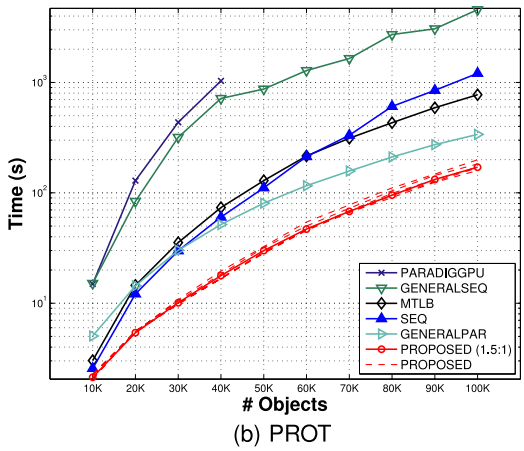


(c) RAND

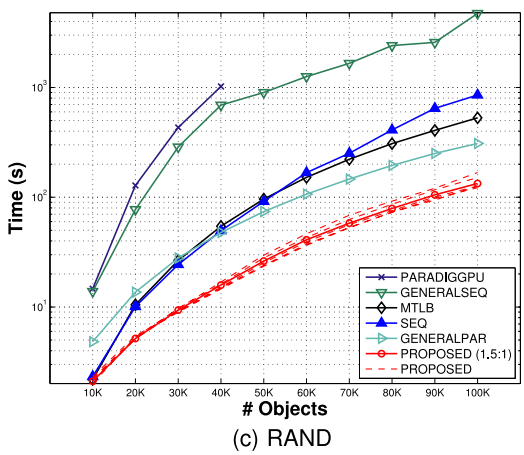
Fig. 20. Runtime comparison between the proposed method and other existing approaches (configuration: CONFIG1).



(a)



(b) PROT



(c) RAND

Fig. 21. Runtime comparison between the proposed method and other existing approaches (configuration: CONFIG2).

theoretically ideal ratio of N_g to N_u for CONFIG1 and CONFIG2, respectively.

Fig. 20a shows the runtime of the proposed method for different ratios of N_g to N_u (approximately from $1:1$ to $2:1$) measured using different datasets. For comparison, we also show the runtime of GENERALPAR on the same plot. Using ratios of $9:7$ and $10:6$, which are close to the theoretically ideal ratio $1.5:1$, gives the best results. For all the ratios used, the proposed method outperforms GENERALPAR significantly.

Figs. 20b and 20b present more detailed measurements of run-time over the PROT and RAND datasets. In these plots, the performance obtained by using the ideal ratio of N_g to N_u $(1.5:1)$ is denoted by a red solid line. For the other ratios of N_g to N_u used are marked by red dotted line. This result shows that the ideal ratio does not deviate significantly from the theoretical analysis.

Figs. 21a and 21b present the results obtained from using CONFIG2. In this configuration, using the $1:1$ ratio gives better performance than the ratios close to the ideal ratio. We believe that this is because bandwidth saturation occurs at a different ratio of N_g to N_u, given that more cores share the (almost similar) memory bandwidth in CONFIG2 than in

CONFIG1. Given the two thread groups, the chain-growing group normally requires more memory bandwidth. If bandwidth saturation occurs, it is better to reduce the portion of chain-growing threads to alleviate the bandwidth saturation problem. Consequently, using the $1:1$ ratio of N_g to N_u is expected to run faster than using $1.5:1$ ratio, as observed from the experimental results in Fig. 21.

To sum up, it is recommended to set the N_g to N_u ratio to a value between $N_g : N_u = 1:1$ and $1.5:1$ as a practical guideline.

4.11 Remarks on Other Parallelization Options

In lieu of multi-core CPUs, we may also consider using GPGPUs to parallelize the NN-chain algorithm. The matrix update part contains ample SIMD type parallelism and might be able to better exploit GPGPUs compared to the chain growing part. However, it is difficult to enforce coalesced memory access, which is often required to fully utilize the power of GPGPUs. For instance, in the example shown in Fig. 8, it is difficult to coalesce all matrix update related memory access. At the time of writing, no GPGPU in the market supports the notion of critical sections. Additionally, different threads will be handling chains of different lengths, which is not ideal for parallelization by GPGPUs. Taken together, we believe that using multicore CPUs is better for parallelizing the NN-chain algorithm, although the situation may change in the future with the advancement of GPU technology. Due to the noncontinuous memory access involved in matrix updates, exploiting spatial locality remains difficult even for CPUs. However, using CPUs gives better results than using GPGPUs in this study, suggesting that the memory transfer issue is more significant in GPGPUs.

5 Conclusions

We have proposed an approach to parallelization of hierarchical clustering based on the nearest-neighbor chain algorithm. Our algorithm grows multiple chains simultaneously by partitioning available threads into two groups, one for growing chains and the other for updating the distance matrix. From theoretical analysis of our approach we provide the optimal ratio for thread partitioning, and theoretical performance bounds on parallelization.

We carry out an extensive set of experiments to test our approach with various data sets. The optimal thread partitioning ratio determined empirically from our experiments is in reasonable agreement with the theoretically predicted value. Our experimental results also further reveal the effects of performance-limiting factors such as starvation, synchronization lock overhead, and hardware aspects. In the absence of memory bandwidth saturation, the performance of our approach is located between our theoretically predicted upper and lower bounds. Memory bandwidth saturation induced bottlenecking at increasing thread number is revealed as a limiting factor to the scalability of the approach. Nonetheless, our approach is shown to significantly outperform the compared alternatives by 3.09-51.79 times in terms of runtime.

Given the popularity of hierarchical clustering in many disciplines, the proposed method will be helpful for extending the applicability of hierarchical clustering to large-scale data that otherwise cannot be analyzed due to the limited scalability of current approaches.

Acknowledgments

This work was supported in part by the National Research Foundation of Korea (NRF) grant funded by the Ministry of Science, ICT and Future Planning [No. 2011-0009963 and No. 2012-R1A2A4A01008475], in part by the ICT R&D program of MSIP/ITP [14-824-09-014, Basic Software Research in Human-level Lifelong Machine Learning (Machine Learning Center)], and in part by the Brain Korea 21 Plus Project in 2014. The authors would like to thank Dr. Daniel Mason for proofreading the manuscript and Dr. Daniel Müllner for helpful discussion.

References

[1] P. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining*. Reading, MA, USA: Addison-Wesley, 2006.
[2] J. Han, M. Kamber, and J. Pei, *Data Mining: Concepts and Techniques*, San Mateo, CA, USA: Morgan Kaufmann, 2006.
[3] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques*, San Mateo, CA, USA: Morgan Kaufmann, 2005.
[4] W. Day and H. Edelsbrunner, "Efficient algorithms for agglomerative hierarchical clustering methods," *J. Classification*, vol. 1, no. 1, pp. 7–24, 1984.
[5] D. Defays, "An efficient algorithm for a complete link method," *Comput. J.*, vol. 20, no. 4, pp. 364–366, 1977.
[6] F. Murtagh, "Multidimensional clustering algorithms," in *Compstat Lectures*, vol. 1, Vienna, Austria: Physika-Verlag, 1985.
[7] R. Sibson, "Slink: An optimally efficient algorithm for the single-link cluster method," *Comput. J.*, vol. 16, no. 1, pp. 30–34, 1973.
[8] J. Benzécri, "Construction dune classification ascendante hiérarchique par la recherche en chaîne des voisins réciproques," *Les Cahiers de l'Analyse des Données*, vol. 7, no. 2, pp. 209–218, 1982.
[9] J. Juan, "Programme de classification hiérarchique par l'algorithme de la recherche en chaîne des voisins réciproques," *Les Cahiers de l'Analyse des Données*, vol. 7, pp. 219–225, 1982.
[10] G. N. Lance and W. T. Williams, "A general theory of classificatory sorting strategies: 1. Hierarchical systems," *Comput. J.*, vol. 9, no. 4, pp. 373–380, 1967.
[11] F. Murtagh, "A survey of recent advances in hierarchical clustering algorithms," *Comput. J.*, vol. 26, no. 4, pp. 354–359, 1983.
[12] I. Gronau and S. Moran, "Optimal implementations of UPGMA and other common clustering algorithms," *Inf. Process. Lett.*, vol. 104, no. 6, pp. 205–210, 2007.
[13] C. Olson, "Parallel algorithms for hierarchical clustering," *Parallel Comput.*, vol. 21, no. 8, pp. 1313–1325, 1995.
[14] S. Rajasekaran, "Efficient parallel hierarchical clustering algorithms," *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, no. 6, pp. 497–502, Jun. 2005.
[15] Z. Li, K. Li, D. Xiao, and L. Yang, "An adaptive parallel hierarchical clustering algorithm," in *Proc. 3rd Int. Conf. High Perform. Comput. Commun.*, 2007, pp. 97–107.
[16] J. Ward Jr., "Hierarchical grouping to optimize an objective function," *J. Amer. Statis. Assoc.*, vol. 58, pp. 236–244, 1963.
[17] E. Rasmussen and P. Willett, "Efficiency of hierarchic agglomerative clustering using the ICL distributed array processor," *J. Doc.*, vol. 45, no. 1, pp. 1–24, 1989.
[18] S. Shalom, M. Dash, and M. Tue, "An approach for fast hierarchical agglomerative clustering using graphics processors with cuda," in *Proc. 14th Pac.-Asia Conf. Adv. Knowl. Discovery Data Mining*, 2010, pp. 35–42.
[19] Z. Du and F. Lin, "A novel parallelization approach for hierarchical clustering," *Parallel Comput.*, vol. 31, no. 5, pp. 523–527, 2005.
[20] S. Yoon, C. Nardini, L. Benini, and G. De Micheli, "Discovering coherent biclusters from gene expression data using zero-suppressed binary decision diagrams," *IEEE/ACM Trans. Comput. Biol. Bioinformat.*, vol. 2, no. 4, pp. 339–354, Oct.–Dec. 2005.
[21] S. Yoon, L. Benini, and G. De Micheli, "Co-clustering: A versatile tool for data analysis in biomedical informatics," *IEEE Trans. Inf. Technol. Biomed.*, vol. 11, no. 4, pp. 493–494, Jul. 2007.

[22] K. Rose, "Deterministic annealing for clustering, compression, classification, regression, and related optimization problems," *Proc. IEEE*, vol. 86, no. 11, pp. 2210–2239, Nov. 1998.
[23] D. Gusfield, *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge, U.K.: Cambridge Univ. Press, 1997.
[24] N. Saitou and M. Nei, "The neighbor-joining method: A new method for reconstructing phylogenetic trees." *Mol. Biol. Evol.*, vol. 4, no. 4, pp. 406–425, 1987.
[25] R. McLay, D. Stanzione, S. Mckay, and T. Wheeler, "A scalable parallel implementation of the neighbor joining algorithm for phylogenetic trees," presented at the ICCABS Conf., Orlando, FL, USA, 2011.
[26] Y. Liu, B. Schmidt, and D. L. Maskell, "Parallel reconstruction of neighbor-joining trees for large multiple sequence alignments using cuda," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, 2009, pp. 1–8.
[27] A. Stamatakis, "RAxML-VI-HPC: Maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models," *Bioinformatics*, vol. 22, no. 21, pp. 2688–2690, 2006.
[28] A. Stamatakis, M. Ott, and T. Ludwig, "RAxML-OMP: An efficient program for phylogenetic inference on SMPs," in *Proc. 8th Int. Conf. Parallel Comput. Technol.*, 2005, pp. 288–302.
[29] A. Stamatakis, "RAxML version 8: A tool for phylogenetic analysis and post-analysis of large phylogenies," *Bioinformatics*, vol. 30, no. 9, pp. 1312–1313, 2014.
[30] S. Yoon, J. Ebert, E. Chung, G. De Micheli, and R. Altman, "Clustering protein environments for function prediction: Finding prosite motifs in 3D," *BMC Bioinformat.*, vol. 8, no. Suppl 4, p. S10, 2007.
[31] M. Thompson, J. Lapointe, Y. Choi, D. Ong, J. Higgins, J. Brooks, and J. Pollack, "Identification of candidate prostate cancer genes through comparative expression-profiling of seminal vesicle," *Prostate*, vol. 68, no. 11, pp. 1248–1256, 2008.
[32] A. Asuncion and D. Newman. (2007). UCI Machine Learning Repository [Online]. Available: http://www.ics.uci.edu/~mlearn/MLRepository.html
[33] D. Ziakas, A. Baum, R. A. Maddox, and R. J. Safranek, "Intel® quickpath interconnect architectural features supporting scalable system architectures," in *Proc. IEEE 18th Annu. Symp. High Perform. Interconnects*, 2010, pp. 1–6.
[34] D. Müllner, "fastcluster: Fast hierarchical, agglomerative clustering routines for R and Python," *J. Statistical Software*, vol. 53, no. 9, pp. 1–18, 2013.

Yongkweon Jeon (S'14) received the BS and MS degrees in electrical engineering from Korea University, Korea, in 2009 and 2013, respectively. He is currently working toward the PhD degree in electrical and computer engineering at Seoul National University, Korea. His research interests include design of parallel algorithms and computer architecture.

Sungroh Yoon (S'99–M'06–SM'11) received the BS degree in electrical engineering from Seoul National University, Korea, in 1996, and the MS and PhD degrees in electrical engineering from Stanford University, CA, in 2002 and 2006, respectively. From 2006 to 2007, he was with Intel Corporation, Santa Clara, CA. Previously, he held research positions with Stanford University, CA, and Synopsys, Inc., Mountain View, CA. He was an assistant professor with the School of Electrical Engineering, Korea University from 2007 to 2012. He is currently an associate professor with the Department of Electrical and Computer Engineering, Seoul National University, Korea. He received the 2013 IEEE/IEEK Joint Award for Young IT Engineers. His research interests include parallel processing, big-data analytics, and high-performance bioinformatics. He is a senior member of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.