J Supercomput
DOI 10.1007/s11227-011-0618-0

Approximative distance computation by random hashing

Selim Mimaroglu · Murat Yagci · Dan A. Simovici

Abstract We propose an approximate computation technique for inter-object distances of binary data sets. Our approach is based on locality sensitive hashing. We randomly select a number of projections of the data set and group objects into buckets based on the hash values of these projections. For each pair of objects, occurrences in the same bucket are counted and the exact Hamming distance is approximated based on the number of co-occurrences in all buckets. We parallelize the computation using mainly two schemes. The first assigns each random subspace to a processor for calculating the local co-occurrence matrix, where all the local co-occurrence matrices are combined into the final co-occurrence matrix. The second method provides the same distance approximation in longer runtimes by limiting the total message size in a parallel computing environment, which is especially useful for very large data sets generating immense message traffic. Our methods produce very accurate results, scale up well with the number of objects, and tolerate processor failures. Experimental evaluations on supercomputers and workstations with several processors demonstrate the usefulness of our methods.

1 Introduction

Locality Sensitive Hashing (LSH), introduced in [1] and [2], can be used for an approximate calculation of distances between the tuples of a data set by using random-

S. Mimaroglu (✉)
Computer Engineering Department, Bahcesehir University, Istanbul, Turkey
e-mail: selim.mimaroglu@bahcesehir.edu.tr

M. Yagci
e-mail: yagcim@ieee.org

D.A. Simovici
Department of Computer Science, University of Massachusetts at Boston, Boston, MA, USA
e-mail: dsim@cs.umb.edu

Published online: 26 May 2011

ized hash functions. A close variant of LSH which works best with the Hamming distance is described in [3]. LSH is used for clustering the Web in [4]. In [5], it is used to enhance the agglomerative hierarchical clustering of the single link method [6]. Both of these techniques rely on the same idea provided by LSH: *Close objects are likely to collide under a high number of randomly chosen hashing functions*. Both of these techniques compute the real distances between objects residing in the same blocks. The clustering algorithms proposed in [4] and [5] focus on finding the approximate set of near neighbors $\mathsf{ANN}(\mathbf{u})$ of an object $\mathbf{u}$, followed by finding real near neighbors of $\mathbf{u}$ by computing the actual distances $d(\mathbf{u}, \mathbf{v})$ for all $\mathbf{v} \in \mathsf{ANN}(\mathbf{u})$. Note that some of the real neighbors of $\mathbf{u}$ may be missed because LSH does not guarantee to put all the close objects in the same blocks.

We propose a method for approximating the distance matrix for data sets of bit vectors. The core idea is to randomly choose m k-dimensional subspaces and consider a bucket for each possible bit vector in this subspace. Then the vectors are hashed into the matching buckets and, for each pair of tuples, the occurrences in the same bucket are counted. The exact Hamming distance is approximated based on the portion of co-occurrences in the m subspaces. Next, we parallelize the computation using two schemes. The first assigns each subspace to a single processor calculating its parts of the co-occurrence matrix and afterward adds up the complete co-occurrence matrix over all subspaces. The second method exchanges results between each processor during computation.

Our data set is a binary table $\mathcal{D}$, having N distinct tuples and a set I that consists of n distinct attributes. A set $K \subseteq I$ with k attributes, designated as a *probe* and chosen randomly, defines a random hashing function h_K by assigning to a tuple $\mathbf{t}_j$ the numerical binary equivalent of the projection of $\mathbf{t}_j$ on the set K, $\mathbf{t}_j[K]$. Each hashing function produces a partition of the set of tuples; each block of this partition consists of tuples that collide under that hashing function.

Parallel and distributed computing techniques are able to solve big and complicated problems by using a variety of divide-and-conquer techniques. In this paper, we introduce several parallel data mining programming methodologies that are applicable in two widely used architectures: shared disk cluster environment, and shared memory architectures [7].

Preliminary results are presented in [8], and the paper is structured as follows. Section 2 examines the relation between randomly generated hash function collisions and distances. In Sect. 3, we present the algorithms and implementation guidelines. Experimental setup and test results are presented in Sect. 4. A final section contains our conclusions and future scope.

2 Collisions and distances

In this section, we examine the relation between randomly generated hash function collisions and inter-object distances.

2.1 Representation of hash function

A *binary data collection* is a sequence $\mathcal{D} = (\mathbf{t}_1, \ldots, \mathbf{t}_N)$ of tuples, where $\mathbf{t}_j \in \{0, 1\}^n$ and $n = |I|$, the cardinality of set of attributes.

Fig. 1 A binary collection and the hashing function h_K for $K = \{i_1, i_3, i_5\}$

(a) $\mathcal{D}$

j	i_1	i_2	i_3	i_4	i_5
1	1	0	0	1	1
2	0	1	1	0	0
3	1	0	1	0	0
4	1	1	0	1	0
5	0	1	1	1	1
6	0	0	1	1	1
7	1	0	1	0	1
8	1	1	0	0	1
9	0	1	1	1	0

(b) $\mathcal{D}[K]$

j	$h_K(\mathbf{t}_j)$
1	5
2	2
3	6
4	4
5	3
6	3
7	7
8	5
9	2

Fig. 2 Partition created by h_K for $K = \{i_1, i_3, i_5\}$. Each block shows its descriptor, and the corresponding collection of tuples

000	*001*	*010*	*011*
{}	{}	{2,9}	{5,6}
100	*101*	*110*	*111*
{4}	{1,8}	{3}	{7}

Let $K = \{i_1, \dots, i_k\} \subseteq \{1, \dots, n\}$ define a *probe*. The *projection of a tuple* $\mathbf{t}_j$ on K is the tuple $\mathbf{t}_j[K] = (t_{ji_1}, \dots, t_{ji_k})$. The *$K$-projection of the binary data collection* $\mathcal{D}$ is the binary data collection $\mathcal{D}[K] = (\mathbf{t}_1[K], \dots, \mathbf{t}_N[K])$.

Each k-projection of $\mathcal{D}$ generates a function $h_K : \{1, \dots, N\} \longrightarrow \mathbb{N}$, where $h_K(\mathbf{t}_j)$ is the binary equivalent of the sequence $\mathbf{t}_j[K]$. This can be seen in Fig. 1, where the binary data collection shown in Part (a) generates the function $h_{\{i_1,i_3,i_5\}}$ given in Part (b) of the figure. In other words, $h_{\{i_1,i_3,i_5\}}$ creates a partition (clustering), with 8 blocks (clusters); 2 of which are empty as shown in Fig. 2.

2.2 Formulation of approximate distance

Let the Hamming distance between two data set tuples $\mathbf{u}, \mathbf{v} \in \{0, 1\}^n$ given by

$$d(\mathbf{u}, \mathbf{v}) = |\{i \in \{1, \dots, n\} : u_i \neq v_i\}|,$$

where $\mathbf{u} = (u_1, \dots, u_n)$ and $\mathbf{v} = (v_1, \dots, v_n)$.

Suppose that the set of attributes K that defines a probe is chosen at random. There are $\binom{n}{k}$ possible choices where $|K| = k$. A collision takes place between two tuples $\mathbf{u}$ and $\mathbf{v}$, if the chosen k attributes are among the $n - d$ attributes on which $\mathbf{u}$ and $\mathbf{v}$ are equal, where $d = d(\mathbf{u}, \mathbf{v})$ is the Hamming distance between $\mathbf{u}$ and $\mathbf{v}$. There are $\binom{n-d}{k}$ such choices for the set of attributes, I. Therefore, for any two tuples $\mathbf{u}, \mathbf{v}$ of $\mathcal{D}$, the collision probability for h_K, that is, the probability that $h_K(\mathbf{u}) = h_K(\mathbf{v})$ is

$$p = \frac{\binom{n-d}{k}}{\binom{n}{k}}.$$

For example, for the data set shown in Fig. 1 and another K such that $k = 2$, the probability $h_K(\mathbf{t}_6)$ collides with $h_K(\mathbf{t}_5)$ is 0.6. Similarly, probability of collision of $h_K(\mathbf{t}_6)$ with $h_K(\mathbf{t}_9)$ is 0.3. These results are intuitive since the distance between $\mathbf{t}_6$ and $\mathbf{t}_9$ is greater, therefore, their collision probability is lower.

If m probes are chosen at random, then $C(\mathbf{u}, \mathbf{v})$, the total number of collisions that occur in this experiment is a binomially distributed variable with the distribution $B(m, p)$. Thus, the expected number of collisions is

$$E(C(\mathbf{u}, \mathbf{v})) = m \frac{\binom{n-d}{k}}{\binom{n}{k}}$$

for $k \leq n - d$, which typically is the case. If $k > n - d$, a collision is impossible and $p = 0$. It is clear that the smaller the distance $d(\mathbf{u}, \mathbf{v})$, the larger the number of collisions will be.

Using Stirling's formula, we can write

$$\begin{aligned}
\frac{\binom{n-d}{k}}{\binom{n}{k}} &= \frac{\frac{(n-d)!}{k!(n-d-k)!}}{\frac{n!}{k!(n-k)!}} \\
&= \frac{(n-d)!}{n!} \frac{(n-k)!}{(n-d-k)!} \\
&\approx \frac{(n-d)^{n-d+0.5}(n-k)^{(n-k+0.5)}}{n^{(n+0.5)}(n-d-k)^{n-d-k+0.5}} \\
&= \left(\frac{n^2 - nd - nk + dk}{n^2 - nd - nk}\right)^{n+0.5} \\
&\quad \times \left(\frac{n-d-k}{n-d}\right)^d \cdot \left(1 - \frac{d}{n-k}\right)^k.
\end{aligned}$$

For moderately large values of n, the first two factors are close to 1. Thus, the expected value of the number of collisions is

$$E(C(\mathbf{u}, \mathbf{v})) \approx m \cdot \left(1 - \frac{d}{n-k}\right)^k.$$

Let $c(\mathbf{u}, \mathbf{v}) = E(C(\mathbf{u}, \mathbf{v}))/m$ be the *relative number of collisions*. Then we estimate the distance between $\mathbf{u}$ and $\mathbf{v}$ as

$$d(\mathbf{u}, \mathbf{v}) \approx (n-k)\left(1 - c(\mathbf{u}, \mathbf{v})^{\frac{1}{k}}\right). \tag{1}$$

2.3 On the computation time of collisions

Assume that m probes with k attributes are applied, where $1 \leq m$ and $1 \leq k \leq n$. Since we deal with binary data, each attribute may take a value of either 0 or 1. Therefore, the partition that corresponds to a k-probe may contain up to 2^k blocks.

Let $c_1, \ldots, c_{2^k}$ be the sizes of the blocks that correspond to a k-probe. For each block of the partition, we need to update the number of collisions of pairs. Therefore, for a block of size c_i, we need to perform $\binom{c_i}{2}$ updates of the pair counters. For example, for a block having three tuples $\{\mathbf{t}_1, \mathbf{t}_2, \mathbf{t}_5\}$, the collision counts of the pairs: $(\mathbf{t}_1, \mathbf{t}_2)$, $(\mathbf{t}_1, \mathbf{t}_5)$, $(\mathbf{t}_2, \mathbf{t}_5)$ are increased by one. The average size of a block is $\frac{N}{2^k}$, so the average total time required is

$$\begin{aligned}\sum_{i=1}^{2^k} \binom{c_i}{2} &= \frac{1}{2}\left(\sum_{i=1}^{2^k} {c_i}^2 - c_i\right)\\ &= \frac{1}{2}\left(2^k\left(\frac{N}{2^k}\right)^2 - 2^k\left(\frac{N}{2^k}\right)\right) = \frac{N^2}{2^{k+1}} - \frac{N}{2}.\end{aligned}$$

The process has to be repeated for each of m probes and this requires an average time proportional to

$$m\left(\frac{N^2}{2^{k+1}} - \frac{N}{2}\right). \tag{2}$$

3 Algorithms and implementation guidelines

A clustering (or partition), which corresponds to a probe, is represented by a Java or a C++ class that is parameterized by the projection size. A clustering includes clusters (or blocks) and members of these clusters.

To produce a clustering $\mathcal{P}$, k attributes are randomly selected and the tuples are projected on the selected set of attributes. A cluster $\mathcal{C}$ consists of tuples that have the same projection $\mathbf{p} \in \{0, 1\}^k$ on the set of attributes that constitutes the probe. The value of bit vector $\mathbf{p}$ is the descriptor of the cluster. The cluster itself is represented by a bit vector $\mathbf{b}_{\mathcal{C}} \in \{0, 1\}^N$, where $(\mathbf{b}_{\mathcal{C}})_j = 1$ if and only if the tuple $\mathbf{t}_j$ belongs to cluster $\mathcal{C}$. Clusters do not overlap. The identifiers of the tuples are placed into appropriate clusters according to the descriptors.

The number of clusterings m is determined by the user and passed as an argument to the implementation. Both the number of clusterings (which equals to the number of probes m) and the width k of the probes are set to positive integers by the user. Note that even for small values of k, the probability of selecting the same probe twice is rather small because there are $\binom{n}{k}$ possible probes and m is typically much smaller than $\binom{n}{k}$.

All clusterings are populated in one scan of the database as follows. Each clustering may have at most 2^k non-empty clusters. First, empty clusterings are initialized, then each tuple in the database $\mathcal{D}$ is passed to all the clusterings. Each clustering projects the tuple on its own randomly selected attributes and then places the tuple in the appropriate cluster according to the cluster descriptors.

For example, assume a clustering projects on first, fifth, and tenth attributes, then the tuple **1**001**0**1101**0**, is placed in cluster 4 of this clustering. Similarly, if another clustering projects on fourth, sixth, and seventh attributes, then the same tuple

Fig. 3 (**a**) Simultaneous Occurrence Matrix (SOM). (**b**) Corresponding Approximate Distance Matrix (ADM)

(a)

	1	**2**	**3**	**4**	**5**
1	3	0	2	0	2
2	0	3	1	3	1
3	2	1	3	1	1
4	0	3	1	3	1
5	2	1	1	1	3

(b)

	1	**2**	**3**	**4**	**5**
1	0	3	1	3	1
2	3	0	2	0	2
3	1	2	0	2	2
4	3	0	2	0	2
5	1	2	2	2	0

100**1011**010, is placed in cluster 7 of this clustering. This computation takes place for each data tuple. In one scan of database $\mathcal{D}$, m clusterings, each having 2^k clusters can be generated efficiently.

3.1 Simultaneous occurrences

We use an $N \times N$ matrix referred to as the *Simultaneous Occurrence Matrix (SOM)* to keep track of the number of collisions of each possible data tuple pairs. After obtaining the clusterings, each cluster in every clustering is scanned once and the SOM cell, corresponding to each pair $(\mathbf{u}, \mathbf{v})$ of tuples in $\mathcal{D}$ that co-occur in the same cluster, is incremented by 1. Note that there are at most $m2^k$ clusters to scan.

As an example, assume that we have a 5-tuple database $\mathcal{D}$ with $n = 4$ attributes, the cardinality of the probes is $k = 1$, and we have $m = 3$ clusterings, whose bit vectors are

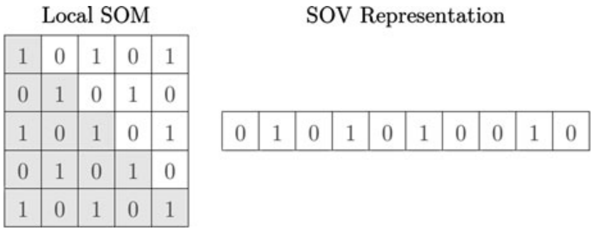
$$\begin{pmatrix}0\\1\\1\\1\\0\end{pmatrix}, \quad \begin{pmatrix}0\\1\\0\\1\\0\end{pmatrix}, \quad \text{and} \quad \begin{pmatrix}1\\0\\1\\0\\0\end{pmatrix}.$$

The first bit vector indicates that the two buckets that correspond to the probe, contain tuples $\{u_1, u_5\}$ and $\{u_2, u_3, u_4\}$ respectively; the other bit vectors are constructed in a similar way. Figure 3(a) shows the Simultaneous Occurrence Matrix (SOM) based on the information in all probes. For example, tuples 2 and 4 occur in the same clusters 3 times. Using the SOM, the Approximate Distance Matrix (ADM) shown in Fig. 3(b) can be computed using Formula (1).

3.2 Parallelization of computation

In the distributed parallel computing environment (in our case, a Beowulf cluster), each worker node reads the database file from the shared disk and creates a collection of bit vectors representing the projected columns. Tuples having the same values on the projected columns are placed in the same clusters. At each computing node, a local binary simultaneous occurrence matrix is filled with 1 s representing collision between a particular pair of tuples. To reduce the message size among multiple processors, we use the upper right half of SOM, which is referred to as Simultaneous Occurrence Vector (SOV) and represented in Fig. 4. Kambadur et al. [9] and [10]

Fig. 4 Local SOM and its SOV representation



offer solutions for efficient use of data structures in a message passing environment. In order to keep the SOV size reasonably low, we use type **char** (8 bits). Note that the number of simultaneous occurrences cannot be greater than the number of nodes in the parallel computing environment. There are 124 nodes in our cluster, therefore, in the resulting SOV the largest value may be 124 and this number can be represented by 8 bits.

Algorithm 1 is for computing SOVs in the cluster environment. This algorithm is graphically represented in Fig. 5. SOVs obtained from each worker node are summed into a resultant SOV. Instead of summing all the SOVs sequentially in the master node, we use the built-in MPI function *MPI_Reduce*. MPI_Reduce is handled in parallel, and in logarithmic time, therefore, it is very advantageous. Full details of the MPI_Reduce depends on the message size and the MPI implementation. Using [11] and [12], a conceptual cost model for MPI collective operations is developed and shown on an MPI_Reduce example in Fig. 6. An alternative idea is to compress each SOV, and decompress the SOVs at receiving nodes, then perform summation. For achieving this, along with compression functions, MPI_Pack and MPI_Unpack routines are used. SOVs gets much smaller when they are compressed. However, there is a large overhead to compress each SOV at the sender node, and to decompress each SOV at the receiver side. Note that the alternative approach reduces the total message size in the cluster noticeably, but it incurs overhead for constantly compressing and decompressing. Experimental results showed us that using MPI_Reduce with original SOVs yields better performance.

As a final procedure for computing the ADM, Formula (1) is applied to the summed SOV as shown in Algorithm 2: Instead of performing this operation sequentially on one node, we divide SOV into m equal fragments and apply Formula (1) on m different worker nodes for the corresponding fragments. For distributing SOV fragments into m worker nodes, we use MPI_Scatter. For collecting the fragments in the master node, MPI_Gather is used.

For very large databases, we noticed that the total message size in Algorithm 1 becomes very large since every node has a message of size $|SOV|$ to pass. Message traffic adversely affects the communication, and in rare cases computing environment comes to a halt. Therefore, we created an alternative method as shown in Algorithm 3. In this algorithm, which divides the SOV into equal fragments, each node operates on a $\frac{|SOV|}{m}$ size message, where m is the total number of nodes. The algorithm passes each SOV fragment $m-1$ times in a circular motion between the worker nodes. For achieving this, we created a virtual circular topology of m worker nodes as shown in Fig. 7. In this setting, each node randomly generates a hash function, and creates a fragment of SOV. For example $Node_1$ has the first fragment containing first $\frac{|SOV|}{m}$

```
Input: $\mathcal{D}$: database, $k$: projection size, $m$: number of projections
Output: $SOV$: Simultaneous Occurrence Vector
// There are $m$ worker nodes
foreach worker node do
    Represent $\mathcal{D}$ vertically by a collection of bit vectors in the main memory,
    refer to this bit vector collection as $\mathcal{D}bv$
    On $\mathcal{D}bv$, create a random projection of size $k$ by choosing $k$ columns
    randomly
    Form a clustering $\mathcal{P}$ containing $2^k$ clusters
    // Some of the clusters in $\mathcal{P}$ may be empty
    Place each tuple in $\mathcal{D}bv$ into corresponding cluster according to the values
    in randomly chosen columns
    Initialize a Simultaneous Occurrence Vector SOV
    foreach cluster $\mathcal{C} \in \mathcal{P}$ do
        foreach tuple pair $(\boldsymbol{t}_i, \boldsymbol{t}_j) \in \mathcal{C}$ do
            Set value of $(\mathbf{t}_i, \mathbf{t}_j)$ in SOV to 1
Sum all SOVs in parallel using, MPI_Reduce
return $SOV$
```

Algorithm 1: Parallel Algorithm to Compute Simultaneous Occurrence Vector

```
Input: $SOV$: Simultaneous Occurrence Vector, $m$: number of nodes
Output: $ADM$: Approximate Distance Matrix
// Fragment SOV into $m$ contiguous vectors of size $s$
   each
$s$ = size(SOV) / $m$;
Scatter each fragment $SOV_f$ of size $s$, $f \in \{1, \ldots, m\}$ to m worker nodes;
// Apply distance approximation formula
foreach $SOV_f$, $f \in \{1, \ldots, m\}$ do
    foreach element e in $SOV_f$ do
        if e != 0 then
            $e$ = applyDistanceFormula($e$) ;
// Each $SOV_f$ has become an $ADM_f$
Gather $ADM_f$, $f \in \{1, \ldots, m\}$ in the master node to form $ADM$;
return $ADM$
```

Algorithm 2: Parallel Algorithm for Computing Approximate Distance Matrix (ADM)

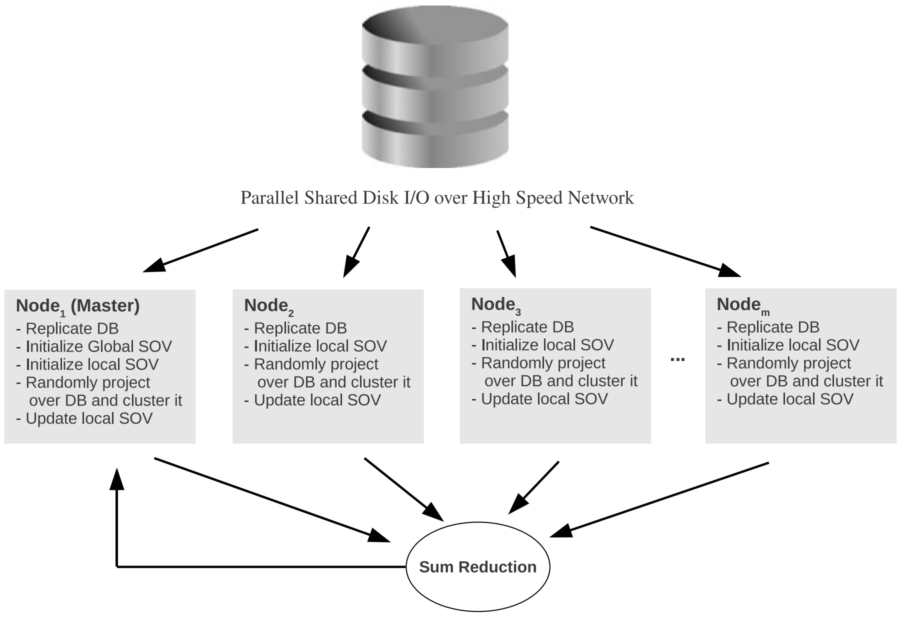


Fig. 5 Computing SOV in cluster environment

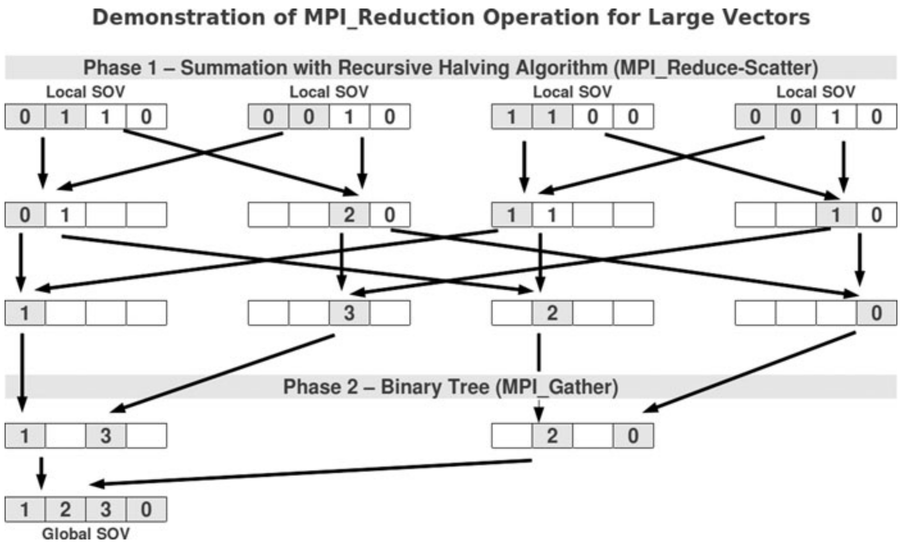

Demonstration of MPI_Reduction Operation for Large Vectors
Phase 1 – Summation with Recursive Halving Algorithm (MPI_Reduce-Scatter)
Local SOV
Local SOV
Local SOV
Local SOV
0 1 1 0
0 0 1 0
1 1 0 0
0 0 1 0
0 1
2 0
1 1
1 0
1
3
2
0
Phase 2 – Binary Tree (MPI_Gather)
1 3
2 0
1 2 3 0
Global SOV

Fig. 6 MPI_Reduce adding 4 SOVs in parallel

entries, and $Node_m$ has the last $\frac{|SOV|}{m}$ entries. After projecting, each node updates the SOV message it has, then passes this SOV message to its right, and receives a new

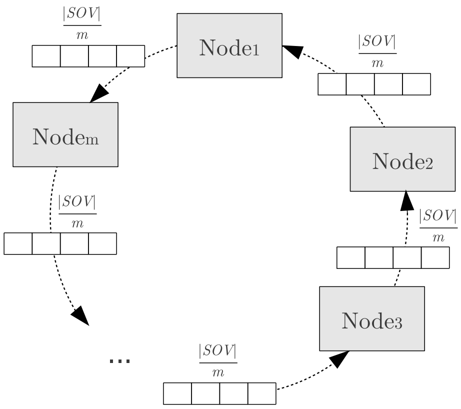


Fig. 7 Alternative approach for computing SOVs in virtual circular topology

```
Input: $\mathcal{D}$: database, $k$: projection size, $m$: number of projections
Output: $SOV$: Simultaneous Occurrence Vector
// There are $m$ worker nodes
foreach worker node do
    Represent $\mathcal{D}$ vertically by a collection of bit vectors in the main memory,
    refer to this bit vector collection as $\mathcal{D}bv$ ;
    On $\mathcal{D}bv$, create a random projection of size $k$ by choosing $k$ columns
    randomly;
    Form a clustering $\mathcal{P}$ containing $2^k$ clusters ;
    // Some of the clusters in $\mathcal{P}$ may be empty
    Place each tuple in $\mathcal{D}bv$ into corresponding cluster according to the values
    in randomly chosen columns;
    Initialize a Simultaneous Occurrence Vector SOV fragment;
    Update the SOV fragment at hand ;

for 1 to m-1 do
    foreach worker node do
        Send SOV fragment to the right node ;
        Update SOV fragment received from left node ;

Combine all the SOV fragments into a global SOV ;
```
Algorithm 3: Parallel Algorithm to Compute Simultaneous Occurrence Vector with Circular Topology

SOV message from its left. The circular message passing between nodes is completed in $m - 1$ iterations. This design makes sure that at anytime in the system maximum message size is $|SOV|$, which makes it suitable for larger databases.

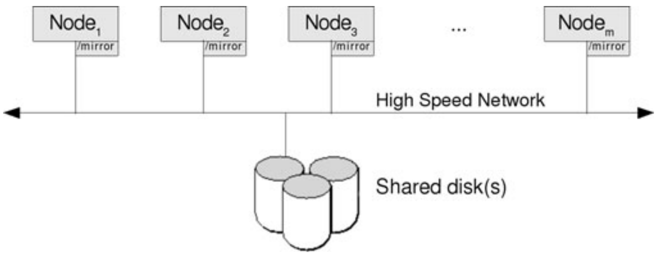


Fig. 8 Topology of the beowulf cluster having 124 nodes

4 Experimental results

4.1 Testing environment

Our primary testing environment is a Beowulf cluster having 124 nodes and infiniband connectivity. Each node has a 64-bit processor with 4 GB to 8 GB of main memory. The cluster is equipped with parallel file system, message passing interface (MPI) [13, 14], Linux operating system, and 62 dual core 1.0-GHz AMD Opteron processors.

Our choice of programming language on the cluster is C++. For conducting experiments, we used MPICH2 [15], along with the gcc version 4.2. compiler, and Boost library [16]. In the cluster, whose topology is given in Fig. 8, workload balancing is performed manually.

Since we are also interested in implementing our algorithms on multi-core platforms with a relatively smaller number of processors (which are widely available), we used as a secondary experimental environment an Apple–Mac Pro having 2 Intel 3.0 GHz Xeon quad-core 64 bit processors. This server has 8 cores and 16 GB of total main memory. On this shared memory system, we use Java threads, and these threads are converted to operating system native threads by the compiler. We relied on the operating system (Mac OS X Leopard) to distribute the work evenly.

In the Beowulf cluster, we relied mostly on MPI libraries for achieving reliability, and synchronization. In the shared memory environment, we used locks, and atomic operations (where possible) to solve similar problems.

The test data sets we used are randomly generated using independent uniform distributions for each bit and consisted of bit vectors of length 20, unless otherwise indicated. The average density of the 1 s is 50% in each vector.

4.2 Runtime results

Using Algorithms 1 and 2 subsequently, Fig. 9 presents the runtime for creating the approximate distance matrix (ADM) on a database having 10,000 tuples. It is important to note that in all approximate distance computation experiments presented in this section, increasing the number of nodes produces more accurate ADMs rather than reducing the runtime. Communication and computation runtime results in Fig. 9 demonstrate that our implementation runs fully in parallel. By using more nodes in

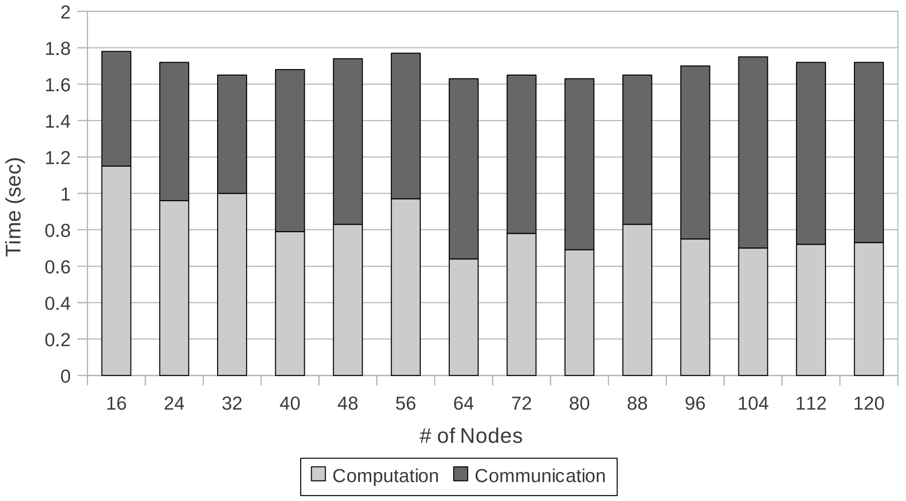


Fig. 9 Runtime on 10,000 tuples using Algorithms 1 and 2 in the cluster environment

the computing environment, we obtain more accurate approximations without increasing the runtime. Running our algorithm on 70 nodes and above achieves accuracy results greater than 90% as shown in Fig. 14.

It should also be noted that we are physically limited by the network bandwidth. In Algorithm 1, each node computes a local SOV, and then these SOVs are sent over the network and merged into a final SOV in the master node. The size of each SOV is $\frac{N(N-1)}{2}$ where N is the number of tuples in the database. In a relatively smaller database with 10,000 tuples, there are around 50 million entries in the SOV (each entry is 1 byte long for saving space). Each node sends its local SOV through the high speed infiniband network as a 50-MB message. Therefore, when running on, e.g. 100 nodes, total message size (total network traffic) on the system goes up to $100 * 50$ MB $=5$ GB. Although, the projection size does not effect the execution time, the messages are reflected as overhead.

In our experimental setup, computing the ADMs of databases having 40,000 tuples or more by using the Algorithms 1 and 2 became problematic due to the total message size. Figure 10 shows that for 40,000 tuples only 32 nodes could be utilized: It was impossible to use more nodes because of the network bottleneck and memory requirements. Clearly, for very large databases we need to use the alternative approach presented in Algorithm 3 in Sect. 3.2. Although this approach is somewhat more time consuming compared to Algorithm 1, due to $m-1$ iterations and communication overhead, it is less sensitive to total message size and memory requirements by keeping the network traffic constant and equal to $|SOV|$. Figure 11 shows that this approach can handle a database with 40,000 tuples as opposed to the results in Fig. 10. Note that increasing the number of nodes does not increase runtime but accuracy, which means the algorithm scales. Figure 11 also shows the time spent for communication and computation.

On our secondary platform (an Apple–Mac Pro) which has only 8 nodes, each clustering is implemented as a Java thread which is converted to an operating system

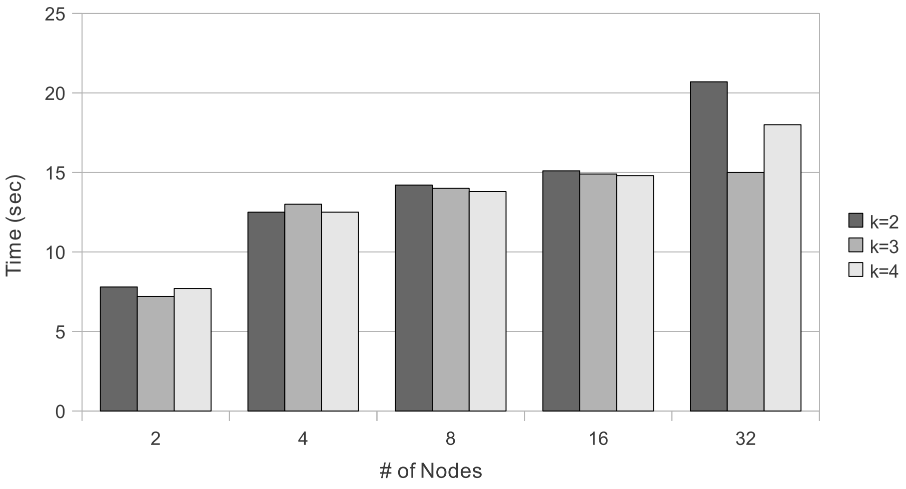


Fig. 10 Runtime on 40,000 tuples with different projections using Algorithms 1 and 2 in the cluster environment

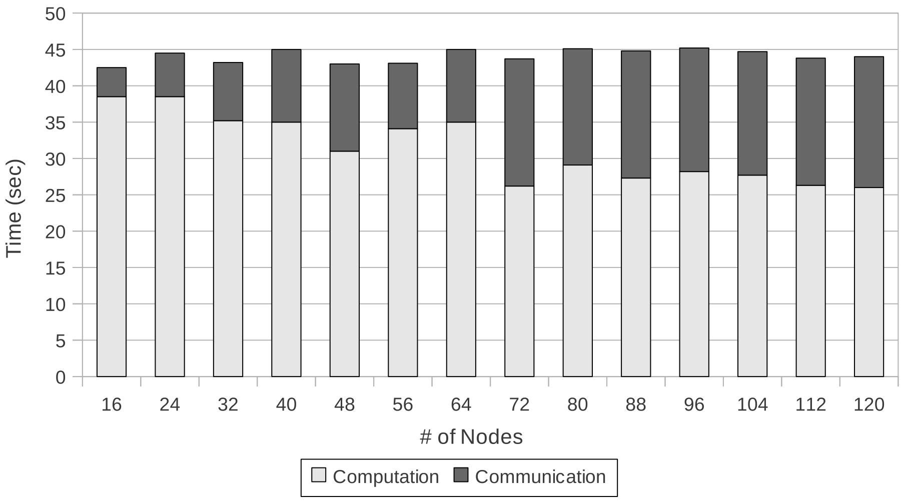


Fig. 11 Runtime on 40,000 tuples using Algorithms 3 and 2 (circular topology) in the cluster environment

native thread by the compiler. On a database having 15,000 tuples, we computed approximate distance matrices for $k = 4$ and varying number of nodes (probes). In Fig. 12, we report the total execution time. The results are as expected: total time stays almost stable when increasing number of nodes.

4.3 Accuracy

To evaluate the accuracy of our approximation, we used the cophenetic correlation coefficient [17]. This coefficient takes its maximum value at 1, where a higher value

Fig. 12 Runtime on 15,000 tuples in the secondary testing environment

Nodes	Total time (sec)
2	2.2
3	2.1
4	2.1
5	2.3
6	2.3
7	2.4
8	2.5

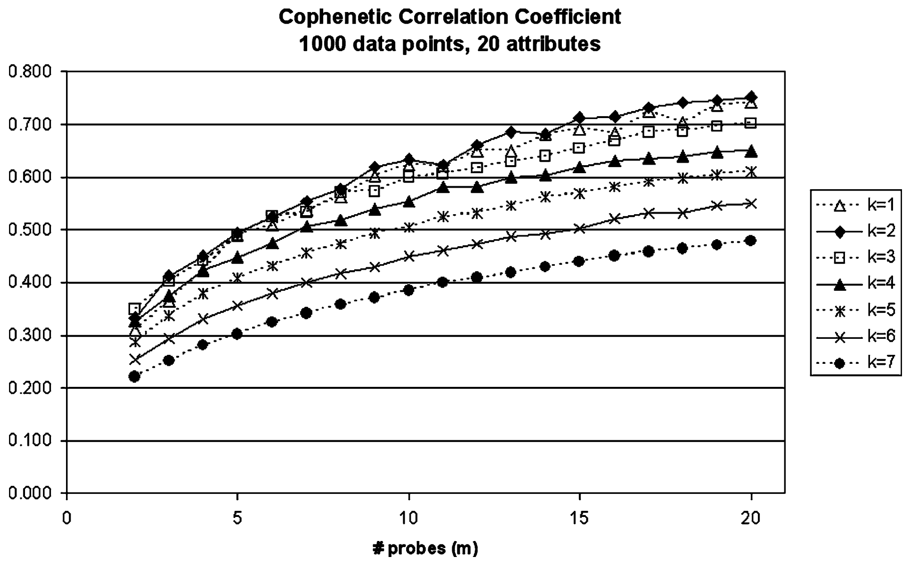


Fig. 13 Cophenetic correlation coefficient on 1,000 tuples having 20 attributes and for varying k and m

implies better correlation. We calculated the cophenetic correlation coefficient between our approximate distance matrix ADM , and the Hamming distance matrix H. The averages of the matrices ADM and H are denoted by $\bar{d}$ and $\bar{h}$, respectively. The coefficient is given by

$$c = \frac{\sum (ADM_{ij} - \bar{d})(H_{ij} - \bar{h})}{\sqrt{\sum (ADM_{ij} - \bar{d})^2 \sum (H_{ij} - \bar{h})^2}}.$$

In Fig. 13, we show the cophenetic correlation coefficient for varying k and m. Note that best experimental results are achieved when $k = 2$. As expected, higher values of m produces better correlations and the coefficient approaches to 1 for reasonable values of m. Figure 14 shows that for 20, 100, and 200 probes, the cophenetic correlation coefficient is 0.773, 0.934, and 0.972, respectively. 200 probes may seem extreme, but each probe scans only 2 attributes out of the total 20 attributes. Therefore, each probe scans 10% of the database, and 200 probes correspond to a total

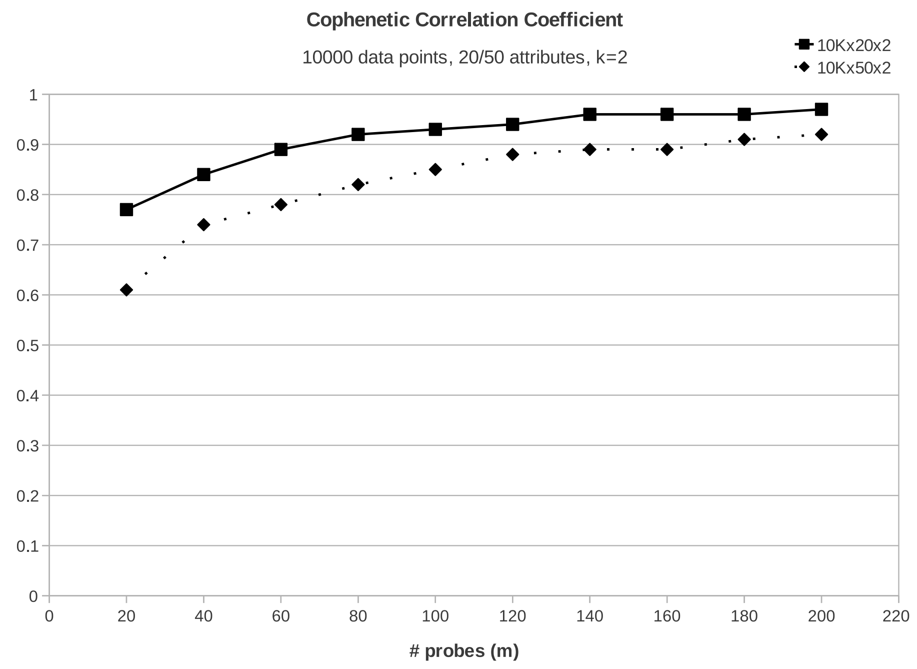


Fig. 14 Cophenetic correlation coefficient on 10,000 tuples having 20 and 50 attributes with $k = 2$ and varying values of m (nodes)

of 20 full scans of the database. On the other hand, to compute a distance matrix of 10,000 tuples, around 5,000 full scans of the database are required.

4.4 Comparative runtimes and complexity analysis

Figure 15 shows runtime results of approximate distance computation in comparison to sequential computation of Hamming distance. We also implemented a parallel computation method for Hamming distance. In both implementations, we use bit sets and take the cardinality of XOR operation on bit sets, which is the fastest way to compute Hamming distance. Only the upper half of the Hamming distance matrix is computed because the matrix is symmetric. When using m nodes, the matrix is divided into m equal parts and distributed to m nodes for computing. Figure 16 presents the comparative parallel runtime results in the cluster environment. For all comparisons, our approximate distance matrix computation is composed of Algorithms 1 and 2 except for 40,000 tuples, and we use Algorithms 3 and 2. We did not provide Fig. 16 as a comparative speedup diagram since the achievement of our algorithm is that it provides almost a constant runtime with increasing number of nodes to improve accuracy of the approximation. Therefore, our major concern is scalability.

Finally, Table 1 shows time complexities of the algorithms comparatively. Generally speaking, complexities of MPI collective operations are logarithmic with the number of inputs which, in our case, is N^2. However, they may vary according to the implementation [11].

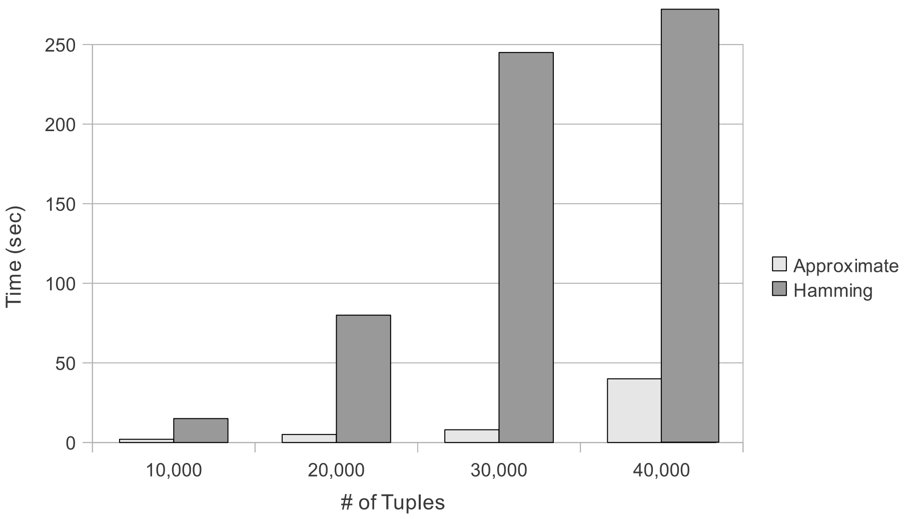


Fig. 15 Runtime comparison with sequential Hamming distance algorithm

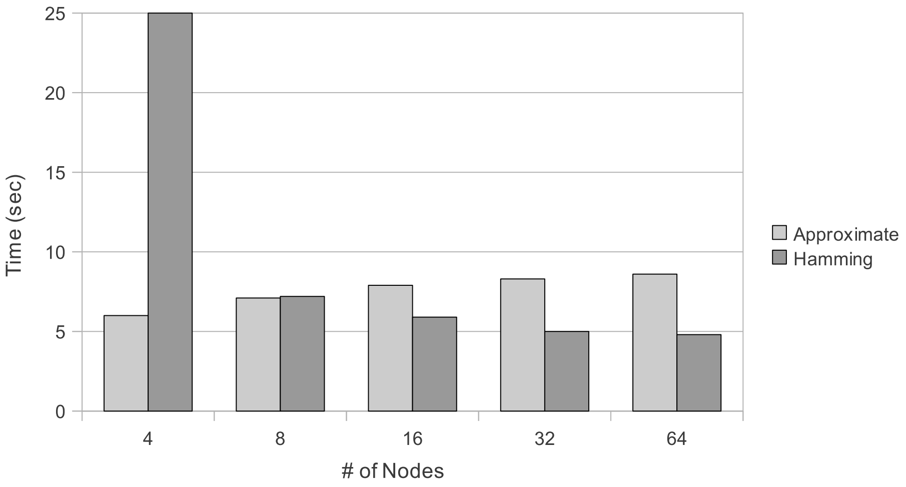


Fig. 16 Runtime comparison with parallel Hamming distance algorithm for data set with 30,000 tuples

5 Conclusions and future scope

Computing the distance matrix of database tuples is a fundamental problem in data clustering. We present an efficient, approximative approach for this computation which relies on randomized hash functions known as Locality Sensitive Hashing (LSH). Implementation guidelines, and several methods that are suitable for distributed and shared memory architectures are discussed. Experimental results clearly demonstrate that our parallel methods are comparably fast and accurate.

Table 1 Comparison of algorithm complexities

Algorithm	Order of complexity	Explanations
Sequential Hamming Distance	$O(nN^2)$	N: cardinality of data set n: cardinality of set of attributes
Sequential Approximate Distance	$m(\frac{N^2}{2^{k+1}} - \frac{N}{2}) + N^2$	See (2). Second term refers to computation of Formula (1)
Parallel Approximate Distance (Algorithms 1 + 2)	$\frac{N^2}{2^{k+1}} - \frac{N}{2} + \alpha \log N^2$	We assume m nodes and include complexity of MPI collective operations
Parallel Approximate Distance (Algorithms 3 + 2)	$(m-1)(\frac{\frac{N^2}{2^{k+1}} - \frac{N}{2}}{m}) + \beta \log N^2$	with added complexity of MPI collective operations
Parallel Hamming Distance	$O(\frac{nN^2}{m}) + \gamma \log N^2$	m is $O(N)$ and complexity of MPI collective operations is added

A very strong point of our method is that when some of the nodes fail, we still get the whole approximate distance matrix, but with reduced accuracy. However, parallel Hamming will leave empty portions in the Hamming distance matrix. Empty portions in the distance matrix get larger when the number of failed nodes increases. Therefore, parallel Hamming leads to incomplete results while we provide complete results in case of network troubles and failed nodes.

Running our algorithm on a few nodes is always faster than both parallel and sequential Hamming distance computations. However, the accuracy may not be very good, but obtained information can provide general idea for exploratory purposes.

Our future scope is to concentrate on more efficient parallel implementations of our method and contemplate developing new parallel clustering ensemble algorithms that will combine several clusterings into a single superior clustering in an efficient manner.

Acknowledgements The authors acknowledge the use of the supercomputing facilities in the College of Science and Mathematics at University of Massachusetts Boston.

References

1. Indyk P, Motwani R (1998) Approximate nearest neighbors: towards removing the curse of dimensionality. In: Proceedings of the thirtieth annual ACM symposium on theory of computing, pp 604–613
2. Andoni A, Indyk P (2008) Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. Commun ACM 51(1):117–122
3. Gionis A, Indyk P, Motwani R (1999) Similarity search in high dimensions via hashing. In: Proceedings of the 25th international conference on very large data bases, pp 518–529
4. Haveliwala T, Gionis A, Indyk P (2000) Scalable techniques for clustering the web. In: WebDB (informal proceedings), vol 129, p 134
5. Koga H, Ishibashi T, Watanabe T (2007) Fast agglomerative hierarchical clustering algorithm using locality-sensitive hashing. Knowl Inf Syst 12(1):25–53
6. Sibson R (1973) Slink: an optimally efficient algorithm for the single-link cluster method. Comput J 16(1):30–34

7. Flynn M (1972) Some computer organizations and their effectiveness. IEEE Trans Comput 21(9):948–960
8. Mimaroglu S, Simovici DA (2008) Approximate computation of object distances by locality-sensitive hashing. In: DMIN, pp 714–718
9. Kambadur P, Gregor D, Lumsdaine A, Dharurkar A (2006) Modernizing the C++ interface to mpi. In: Lecture notes in computer science, vol 4192. Springer, Berlin, p 266
10. Tansey W, Tilevich E (2008) Efficient automated marshaling of C++ data structures for mpi applications. In: IPDPS. IEEE Press, New York, pp 1–12
11. Thakur R, Rabenseifner R, Gropp W (2005) Optimization of collective communication operations in mpich. Int J High Perform Comput Appl 19(1):49
12. Dhillon I, Modha DS (2000) A data-clustering algorithm on distributed memory multiprocessors. In: Lecture notes in computer science, vol 1759. Springer, Berlin, pp 245–260
13. Gropp W, Huss-Lederman S, Lumsdaine A, Lusk E, Nitzberg B, Saphir W, Snir M (1998) Mpi—the complete reference, vol 2, The mpi-2 extensions. ISBN-10:0-262-57123-4
14. Tu B, Fan J, Zhan J, Zhao X (2009) Performance analysis and optimization of MPI collective operations on multi-core clusters. J Supercomput. doi:10.1007/s11227-009-0296-3
15. Gropp W (2002) Mpich2: a new start for mpi implementations. In: Lecture notes in computer science. Springer, Berlin, pp 7–27
16. Karlsson B (2005) Beyond the C++ standard library. Addison-Wesley Professional, New York
17. Sokal R, Rohlf F (1962) The comparison of dendrograms by objective methods. Taxon 11(1):30–40