Query-Log Aware Replicated Declustering

Ata Turk, K. Yasin Oktay and Cevdet Aykanat

Abstract—Data declustering and replication can be used to reduce I/O times related with processing of data intensive queries. Declustering parallelizes the query retrieval process by distributing the data items requested by queries among several disks. Replication enables alternative disk choices for individual disk items and thus provides better query parallelism options. In general, existing replicated declustering schemes do not consider query log information and try to optimize all possible queries for a specific query type, such as range or spatial queries. In such schemes, it is assumed that two or more copies of all data items are to be generated and scheduling of these copies to disks are discussed. However, in some applications, generation of even two copies of all of the data items is not feasible, since data items tend to have very large sizes. In this work we assume that there is a given limit on disk capacities and thus on replication amounts. We utilize existing query-log information to propose a selective replicated declustering scheme, in which we select the data items to be replicated and decide on their scheduling onto disks while respecting disk capacities. We propose and implement an iterative improvement algorithm to obtain a two-way replicated declustering and use this algorithm in a recursive framework to generate a multi-way replicated declustering. Then we improve the obtained multi-way replicated declustering by efficient refinement heuristics. Experiments conducted on realistic datasets show that the proposed scheme yields better performance results compared to existing replicated declustering schemes.

Index Terms—Declustering, replication, parallel disk architectures, iterative improvement heuristics.

1 Introduction

In this section we present related work about declustering and replication and briefly list our contributions.

1.1 Related work

Data declustering is a data scattering technique used in parallel-disk architectures to improve query response time performances of I/O intensive applications. The aim in declustering is to optimize the processing time of each query requested from a parallel-disk architecture. This is achieved by reducing the number of disk accesses performed by a single disk of the architecture while answering a single query. Declustering has been shown to be an NP-complete problem in some contexts [1], [2].

Declustering is widely investigated in applications where large spatial data are queried. In such applications, queries are in the form of ranges requesting neighboring data points, and hence, related declustering schemes try to scatter neighboring data items into separate disks instead of exploiting query log information. For a good survey of declustering schemes optimized for range queries see [3] and the citations within.

There are some applications that also query very large data items in a random fashion and in such applications utilization of query log information is of essence for efficient declustering [1], [2], [4]. In [1], the declustering problem with a given query distribution is modeled as a max-cut partitioning of a weighted similarity graph, where data items are represented by vertices and an edge between two vertices implies that corresponding data items appear in at least one common query. In [2] and [4], the deficiencies of the weighted similarity graph model are addressed and hypergraph models which encode the total I/O cost correctly are proposed.

Data replication is a widely applied technique in various application areas such as distributed data management [5] and information retrieval [6], [7] to achieve fault tolerance and fault recovery. Data replication can also be exploited to achieve higher I/O parallelism in a declustering system [8]. However, while performing replication, one has to be careful about consistency considerations, which arise in update and delete operations. Furthermore, write operations tend to slow down when there is replication. Finally, replication means extra storage requirement and there are applications with very large data sizes where even two-copy replication is not feasible. Thus, if possible, unnecessary replication has to be avoided and techniques that enable replication under given size constraints must be studied.

When there is data replication, the problem of query scheduling has to be addressed as well. That is, when a query arrives, we have to decide which replicas will be used to answer the query. A maximum-flow formulation is proposed in [9] to solve this scheduling problem optimally. There are replicated declustering schemes that aim to minimize this scheduling overhead [10], [11], while minimizing I/O costs. A variation of this problem arises when replicas are assumed to be distributed over different sites, where each site hosts a parallel-disk architecture [12]. This variation can be modeled as a maximum flow problem as well.

- *Ata Turk, and Cevdet Aykanat are with the Computer Engineering Department of Bilkent University. E-mail: {atat, aykanat}@cs.bilkent.edu.tr*
- *K. Yasin Oktay is currently with the Computer Science Dept. of University of California, Irvine. He was with the Comp. Eng. Dept. of Bilkent University during the studies of this work. E-mail: koktay@ics.uci.edu*

Most of the existing replicated declustering schemes proposed for range queries are discussed in [13], [14]. There are some replicated declustering schemes proposed for arbitrary queries as well ([15]–[16]). All of these schemes ([13]–[16]) assume items with equal sizes and they also assume that all data items will be requested equally likely and thus generate equal number of replicas for all data items. Furthermore, they replicate all data items two or more times.

In [15], Random Duplicate Assignment (RDA) scheme is proposed. RDA stores a data item on two disks chosen randomly from the set of disks and it is shown that the retrieval cost of random allocation is at most one more than the optimal cost with high probability (when there are at least two-copies of all data items). In [12], [17], Orthogonal Assignment (OA) is proposed. OA is a two-copy replication scheme for arbitrary queries and if the two disks that a data item is replicated at are considered as a pair, each pair appears only once in the disk allocation of OA. In [16], Design Theoretic Assignment (DTA) is proposed. DTA uses the blocks of a $(K, c, 1)$ design for c-copy replicated declustering using K disks. A block and its rotations can be used to determine the disks on which the data items are stored. Even though both OA and DTA can be modified to achieve selective replication, they do not utilize query log information. However, with the increasing usage in GIS and spatial database systems, such information is becoming highly available, and it is desirable for a replication scheme to be able to utilize this information. A simple motivating example for utilizing query-logs can be found in Section 1 of the Appendix.

TABLE 1: The notations used in the paper

Symbol	Description
$\mathcal{D}$	Dataset
Q	Query set
K	Total number of disks
$\mathcal{D}_k$	Set of data items assigned to disk k
C_{max}	Maximum storage capacity of a disk
d_i	A data item in the dataset
$s(d_i)$	Storage requirement for data item d_i
q	A query in the query set
$\|q\|$	Number of data items requested by q
$f(q)$	Frequency of q in Q
$r(q)$	Response time for q
$t_k(q)$	Response time of disk k for q
$S_{opt}(q)$	Optimal scheduling for q
$r_{opt}(q)$	Optimal response time for q
R_K	A K-way replicated declustering
$Tr(R_K, Q)$	Parallel response time of R_K for Q
$Tr_{opt}(Q)$	Optimal parallel response time for Q
$TrO(R_K, Q)$	Parallel response time overhead of R_K for Q
$g_m(d)$	In a two-way replicated declustering phase, reduction to be observed in the overall query processing cost, if d is moved to the other disk.
$g_r(d)$	In a two-way replicated declustering phase, reduction to be observed in the overall query processing cost, if d is replicated in both disks.
$g_{u_X}(d)$	In a two-way replicated declustering phase, reduction to be observed in the overall query processing cost, if a replica of d is deleted from disk $\mathcal{D}_X$.
$vg(d)$	Number of queries requesting d such that the disk(s) that d resides in serve(s) more than optimal number of data items for these queries.
$g_m(d, k)$	In a K-way replicated declustering phase, reduction to be observed in the overall query processing cost, if d is moved to disk k.
$g_r(d, k)$	In a K-way replicated declustering phase, reduction to be observed in the overall query processing cost, if d is replicated in disk k.

1.2 Contributions

In this work we present a selective and query-log aware replication scheme which works in conjunction with declustering. The proposed scheme utilizes the query log information to minimize the aggregate parallel query response time while obeying given replication constraints due to disk sizes. There are no restrictions on the replication counts of individual data items. That is, some data items may be replicated more than once while some other data items may not even be replicated at all.

We first propose an iterative-improvement-based replicated two-way declustering algorithm. In this algorithm, in addition to the replication operation that we proposed in [18], we successfully incorporate unreplication operation to the replicated two-way declustering algorithm to prevent unnecessary replications. We also provide simple closed-form expressions for computing the cost of a query in a two-way replicated declustering. Utilizing these expressions, we avoid usage of expensive network-flow based algorithms for the construction of optimal query schedules. By recursively applying our two-way replicated declustering algorithm we obtain a K-way replicated declustering. Our unreplication algorithm prevents unnecessary replications to advance to the next levels in the recursive framework.

We then propose an efficient multi-way replicated refinement heuristic that considerably improves the obtained K-way replicated declustering via multi-way move and multi-way replication operations. In this iterative algorithm, we adapt a novel idea about multi-way move operations and obtain an efficient greedy multi-way move/replication scheme. We also present an efficient scheme to avoid the necessity of computing the optimal schedules of all queries at each iteration of our multi-way refinement algorithm. The proposed scheme enables us to compute the optimal schedules of all queries just once, at the beginning of the multi-way refinement, and then update the schedules incrementally according to the performed operations.

The rest of the paper is organized as follows. Section 2 presents the notation and the definition of the problem. The proposed scheme is presented in Section 3. In Section 4, we experiment and compare our proposed approaches with two state-of-the-art replications schemes. We conclude in Section 5.

2 Notation and Definitions

We are given a dataset $\mathcal{D}$ with $|\mathcal{D}|$ indivisible data items and a query set Q with $|Q|$ queries, where a query $q \in Q$

requests a subset of data items, i.e., $q \subseteq \mathcal{D}$. Each data item $d \in \mathcal{D}$ can represent a spatial object, a multi-dimensional vector or a cluster of data records depending on the application. $s(d)$ indicates the storage requirement for d and $s(\mathcal{D}') = \sum_{d \in \mathcal{D}'} s(d)$ indicates the storage requirement for data subset $\mathcal{D}'$. Query information can be extracted by either application usage prediction or mining existing query logs, with the assumption that future queries will be similar to older ones. In a few applications, it is more appropriate to apply declustering such that items that have common features are stored on separate disks [19], [20], [21]. However, even in such applications, each query can be considered as a set of features and the discussions in the following sections still hold.

In a given query set Q, two data items are said to be neighbor if they are requested together by at least one query. Each query q is associated with a relative frequency $f(q)$ which indicates the probability that q will be requested. Query frequencies can be extracted from the query log. We assume that all disks are homogeneous and the retrieval time of all data items on all disks are equal and can be accepted as one for practical purposes.

Definition K-Way Replicated Declustering: Given a set $\mathcal{D}$ of data items, K homogeneous disks with storage capacity C_{max}, and a maximum allowable replication ratio r, $R_K = \{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_K\}$ is said to be a K-way replicated declustering of $\mathcal{D}$, where $\mathcal{D}_k \subseteq \mathcal{D}$ for $1 \le k \le K$, $\cup_{k=1}^{K} \mathcal{D}_k = \mathcal{D}$, and R_K satisfies the following feasibility conditions for $1 \le k \le K$, when each decluster $\mathcal{D}_k$ is assigned to a separate disk:

- Disk capacity constraint: $s(\mathcal{D}_k) \le C_{max}$
- Replication constraint: $\sum_{1 \le k \le K} s(\mathcal{D}_k) \le (1+r) \times s(\mathcal{D})$.

The optimal schedule for a query q minimizes the maximum number of data items requested from a disk for q. Given a replicated declustering R_K and a query q, an optimal schedule $S_{opt}(q)$ for q can be calculated by a network-flow based algorithm [9] in $O(|q|^2 \times K)$ time, if we assume homogeneous data item retrieval times. $S_{opt}(q)$ indicates which replicas of the data items will be accessed during processing q.

Definition Given a replicated declustering R_K, a query q and an optimal schedule $S_{opt}(q)$ for q, response time $r(q)$ for q is:

$$r(q) = \max_{1 \le k \le K} \{t_k(q)\}, \tag{1}$$

where $t_k(q)$ denotes the total retrieval time of data items from disk $\mathcal{D}_k$ that are requested by q. Under homogeneous data item retrieval times assumption, $t_k(q)$ can also be considered as the number of data items retrieved from $\mathcal{D}_k$ for q.

Definition The total parallel response time of a replicated declustering R_K for a query set Q is:

$$Tr(R_K, Q) = \sum_{q \in Q} f(q) r(q). \tag{2}$$

Definition A replicated declustering R_K is said to be *strictly optimal* for a query set Q iff it is optimal for every query $q \in Q$, i.e., $r(q) = r_{opt}(q), \forall\, q \in Q$, where

$$r_{opt}(q) = \lceil |q|/K \rceil. \tag{3}$$

Total parallel response time of a strictly optimal replicated declustering is called $Tr_{opt}(Q)$ and is:

$$Tr_{opt}(Q) = \sum_{q \in Q} f(q) r_{opt}(q). \tag{4}$$

Definition The total parallel response time overhead of a replicated declustering R_K for a query set Q is:

$$TrO(R_K, Q) = Tr(R_K, Q) - Tr_{opt}(Q). \tag{5}$$

Definition K-Way Replicated Declustering Problem: Given a set $\mathcal{D}$ of data items, a set Q of queries, K homogeneous disks each with a storage capacity of C_{max}, and a maximum allowable replication ratio r, find a K-way replicated declustering R_K of $\mathcal{D}$ that minimizes the total parallel response time $Tr(R_K, Q)$. Note that minimizing $Tr(R_K, Q)$ is equivalent to minimizing $TrO(R_K, Q)$, since $Tr_{opt}(Q)$ is a constant.

3 Proposed Approach

We propose a two-phase approach for solving the K-way replicated declustering problem. In the first phase, we use a recursive replicated declustering heuristic to obtain a K-way replicated declustering. We should note that, by allowing imbalanced two-way declusters in this phase, we are able to obtain K-way declusterings for arbitrary K values. In the second phase, we use a refinement heuristic to improve the K-way replicated declustering obtained in the first phase. In the following two subsections we provide the details of operations performed in these phases. The reader is referred to Section 4 of the Appendix for a detailed complexity analysis of the recursive replicated declustering and multi-way replicated refinement phases.

3.1 Recursive replicated declustering phase

The objective in the recursive replicated declustering phase is to evenly distribute the data items of queries at each two-way replicated declustering step of the recursive framework. That is, at each two-way step, we try to attain optimal response time $r_{opt}(q) = \lceil |q|/2 \rceil$ for each query q as much as possible. This objective is somewhat restrictive and it will not completely model the minimization of the objective function for the K-way replicated declustering problem. But it is expected to produce a “good” initial K-way replicated declustering for the multi-way refinement phase. The even query distribution obtained after the recursive replicated declustering phase is assumed to avoid a bad locally optimal declustering by providing flexibility in the search space of the multi-way refinement scheme.

3.1.1 Two-way replicated declustering

The core of our recursive replicated declustering algorithm is a two-way replicated declustering algorithm. In this algorithm, we start with a given (and possibly randomly generated) initial feasible two-way declustering of the dataset $\mathcal{D}$, say $R_2 = \{\mathcal{D}_A, \mathcal{D}_B\}$, and iteratively improve R_2 by three refinement operations defined over the data items: Namely *move*, *replication* and *unreplication* operations. In order to perform these three operations we consider four different gain values for each data item d:

- move gain ($g_m(d)$): the reduction to be observed in the overall query processing cost, if d is moved to the other disk,
- replication gain ($g_r(d)$): the reduction to be observed in the overall query processing cost, if d is replicated to the other disk,
- unreplication-from-A gain ($g_{u_A}(d)$): the reduction to be observed in the overall query processing cost, if a replica of d is deleted from $\mathcal{D}_A$,
- unreplication-from-B gain ($g_{u_B}(d)$): the reduction to be observed in the overall query processing cost, if a replica of d is deleted from $\mathcal{D}_B$.

Unreplication gains are only meaningful for data items that are replicated. Similarly, in a two-way declustering, move and replication gains are only meaningful for data items that are not replicated. Thus, for any data item, only two gain values need to be maintained.

A two-way replicated declustering $R_2 = \{\mathcal{D}_A, \mathcal{D}_B\}$ can be considered as partitioning the dataset $\mathcal{D}$ into three mutually disjoint parts: A, B, and AB, where part A is composed of the data items that are only stored in disk $\mathcal{D}_A$, part B is composed of the data items that are only stored in disk $\mathcal{D}_B$, and part AB is composed of the data items that are replicated. In this view,

$$\mathcal{D}_A = A \cup AB \quad \text{and} \quad \mathcal{D}_\mathrm{B} = \mathrm{B} \cup \mathrm{AB}. \tag{6}$$

A variable $State(d)$ is kept to store the part information of each data item d.

For each query q, we maintain a 3-tuple

$$dist(q) = (|q_A| : |q_B| : |q_{AB}|), \tag{7}$$

where $|q_A|$, $|q_B|$, and $|q_{AB}|$ indicate the number of data items of q in parts A, B, and AB, respectively. That is,

$$q_A = q \cap A \ , \ \ \mathrm{q_B = q \cap B \ \ and \ \ q_{AB} = q \cap AB}. \tag{8}$$

The total number of data items requested by query q is equal to: $|q| = |q_A| + |q_B| + |q_{AB}|$.

Using the above notation, the retrieval times of a given query q from disks $\mathcal{D}_A$ and $\mathcal{D}_B$ can be written as follows, without loss of generality assuming that $|q_A| \geq |q_B|$:

$$\begin{aligned} t_A(q) &= \begin{cases} \lceil |q|/2 \rceil & \text{if } |q_{AB}| \geq (|q_A| - |q_B|) - 1 \\ |q_A| & \text{otherwise} \end{cases} \\ t_B(q) &= \begin{cases} \lfloor |q|/2 \rfloor & \text{if } |q_{AB}| \geq (|q_A| - |q_B|) - 1 \\ |q_B| + |q_{AB}| & \text{otherwise} \end{cases} \end{aligned} \tag{9}$$

Here the "$|q_{AB}| \geq (|q_A| - |q_B|) - 1$" condition corresponds to the case in which there are enough number of replicated data items requested by q that can be utilized to achieve even distribution of q among $\mathcal{D}_A$ and $\mathcal{D}_B$. The "otherwise" condition corresponds to the case for which even distribution of q among the disks is not possible. In the former case, the replicated data items requested by q will be retrieved from $\mathcal{D}_A$ and $\mathcal{D}_B$ in an appropriate manner to attain even distribution, whereas in the latter case, all of the replicated data items requested by q will be retrieved from $\mathcal{D}_B$ to minimize the cost of query q. Hence, for a two-way replicated declustering R_2=$\{\mathcal{D}_A, \mathcal{D}_B\}$, the cost $r(q)$ of q can be computed with the following closed-form expression:

$$r(q) = \begin{cases} \lceil |q|/2 \rceil & \text{if } |q_{AB}| \geq (||q_A| - |q_B||) - 1 \\ max(t_A(q), t_B(q)) & \text{otherwise} \end{cases} \tag{10}$$

The simple closed-form expressions given in Equations 8, 9, and 10 for computing $r(q)$ enable us to avoid constructing the optimal schedules for the queries throughout the iterations of the two-way replicated declustering algorithm. That is, $r(q)$ in Equation 10 gives the cost of query q that can be attained by an optimal schedule for q, without constructing $S_{opt}(q)$ through costly network-flow based algorithms.

It is clear that optimizing the cost function given below at each two-way replicated declustering step will optimize the "goodness" criteria explained at Section 3.1:

$$cost(R_2) = \sum_{q \in Q} f(q)(r(q) - \lceil |q|/2 \rceil) \tag{11}$$

Our overall two-way replicated declustering algorithm works as a sequence of two-way refinement passes performed over all data items. In each pass, we start with computing the initial operation gains of all data items. Then, we iteratively perform the following computations: find the data item and the operation that produces the highest reduction in the cost; perform that operation; update gain values of neighboring data items; lock the selected data item to further processing to prevent thrashing.

We perform these computations until there are no remaining data items to process. We restore the declustering to the state where the best reduction is obtained during the pass and we start a new pass over the data items if the obtained improvement in the current pass is above a threshold or if the number of passes performed is below some predetermined number. Once we obtain a two-way declustering, we can recursively apply our two-way declustering algorithm on each of these declusters to obtain any number of declusters. A running example demonstrating move, replication and unreplication gain updates can be found in Section 2.1 of the Appendix.

All operations are kept in priority queues keyed according to their gain values. The priority queues are implemented as binary heaps. For a two-way declustering, we maintain six heaps: Two heaps for storing the move operations of data items from part A to B and from part B to A, two heaps for storing the replication operations of data items from part A to B and from part B to A, and two heaps for storing the unreplication operations of replicated data items from part A and from part B.

In our two-way replicated declustering algorithm, we start with calculating the initial move, replication and unreplication gains of all data items (Appendix, Algorithm 2) . After initializing the gains, we retrieve the highest gains and the associated data items for each operation type and by comparing these gains we select the best operation to perform. If there are any possible unreplication operations which do not increase the total cost of the system (i.e., with zero unreplication gain), those unreplication operations are performed first. After we finish possible unreplications, we compare the gains to be obtained by move and replication operations. If the gains are the same, we prefer to perform move operations. Recall that each data item is eligible for two types of operations and thus has two related gain values. So, after deciding on the best operation to perform, we remove the data item from the two related heaps by extractMax and delete operations.

After performing an operation (move, replication or unreplication) on a data item d^* , we may need to update the gains of operations related with the data items that are neighbor to d^* (Appendix, Algorithms 3, 4, and 5). For any data item d, we have $g_r(d) \geq g_m(d)$, hence, in a pass, the number of replication operations tend to outweigh the number of move operations. A similar problem had been observed when replication was used for clustering purposes in the VLSI literature and one of the solutions proposed was the gradient methodology [22]. We adopt this methodology by permitting solely move and unreplication operations until the improvement obtained drops below a certain threshold and only after that we perform replication operations.

3.1.2 Query splitting

At the end of a two-way replicated declustering $R_2 = \{\mathcal{D}_A, \mathcal{D}_B\}$ of a dataset and query set pair $\{\mathcal{D}, Q\}$, we split the queries of Q among the obtained two sub-datasets as evenly as possible so that split queries correctly represent the optimizations performed during that two-way replicated declustering step. That is, an R_2 is decoded as splitting each query $q \in Q$ into two sub-queries

$$q' \subseteq q \cap \mathcal{D}_A \text{ and } q'' \subseteq q \cap \mathcal{D}_B, \tag{12}$$

such that the difference $||q'|-|q''||$ is minimized. The split queries q' and q'' are added to sub-query sets Q_A and Q_B, respectively so that further two-way declustering operations can be recursively performed on $\{\mathcal{D}_A, Q_A\}$ and $\{\mathcal{D}_B, Q_B\}$ pairs.

Recall that the optimizations performed during a two-way replicated declustering assume that queries will have optimal schedules with regard to that of two-way replicated declustering, and even splitting of queries ensures that. Also recall that constructing the optimal schedule of a query q in a replicated declustering system requires network-flow-based algorithms. However, for two-way replicated declustering this feat can be achieved by utilizing the item distribution $dist(q)$ of q and the value of $r(q)$, which can be computed via the closed form definitions given in Equations 7–10. We know that in an optimal splitting according to the optimal schedule, the size of q' should be $|q'| = t_A(q)$ and the size of q'' should be $|q''| = t_B(q)$.

Consider the three-way partition of query q into q_A, q_B, and q_{AB} (according to Equation 8) induced by the two-way replicated declustering. It is clear that data items in q_A will go into q' and data items in q_B will go into q'', so all that remains is to decide on the splitting of the data items in q_{AB} according to an optimal scheduling. Let us call the replicated data items that will go into q' as q'_{AB} and the replicated data items that will go into q'' as q''_{AB}. That is,

$$q' = q_A \cup q'_{AB} \text{ and } q'' = q_B \cup q''_{AB}, \tag{13}$$

Since we want to enforce a splitting such that $|q'| = t_A(q)$ and $|q''| = t_B(q)$, we can say that

$$|q'_{AB}| = t_A(q) - |q_A| \text{ and } |q''_{AB}| = t_B(q) - |q_B| = |q_{AB}| - |q'_{AB}|. \tag{14}$$

Any splitting of the data items in q_{AB} that respects the size constraints given in Equation 14 satisfies the optimality condition. In our studies we assign the first $t_A(q) - |q_A|$ items of q_{AB} to q' and the remaining items of q_{AB} to q''. Other assignment schemes can be explored for better performance results.

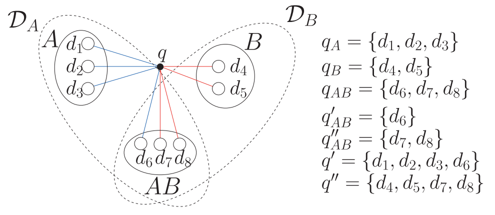

$\mathcal{D}_A$
$\mathcal{D}_B$
A
B
AB
q
d_1
d_2
d_3
d_4
d_5
d_6 d_7 d_8
$q_A = \{d_1, d_2, d_3\}$
$q_B = \{d_4, d_5\}$
$q_{AB} = \{d_6, d_7, d_8\}$
$q'_{AB} = \{d_6\}$
$q''_{AB} = \{d_7, d_8\}$
$q' = \{d_1, d_2, d_3, d_6\}$
$q'' = \{d_4, d_5, d_7, d_8\}$

Fig. 1: Splitting of a query q according to a two-way replicated declustering $R_2 = \{\mathcal{D}_A, \mathcal{D}_B\}$.

A sample splitting of a query q with eight data items is given in Fig. 1. According to Equation 9, for q, $t_A(q) = 4$ and $t_B(q) = 4$, hence $|q'| = 4$ and $|q''| = 4$. Since, for q, $|q_A| = 3$, $|q_B| = 2$, $|q_{AB}| = 3$, by Equations 13 and 14, we can say that $|q'_{AB}| = t_A(q) - |q_A| = 1$ and $|q''_{AB}| = t_B(q) - |q_B| = 2$. Any splitting of q_{AB} according to these size constraints satisfies the optimality condition, and according to our assignment scheme $q'_{AB} = \{d_6\}$ and $q''_{AB} = \{d_7, d_8\}$. Hence, $q' = q_A \cup q'_{AB} = \{d_1, d_2, d_3, d_6\}$ and $q'' = q_B \cup q''_{AB} = \{d_4, d_5, d_7, d_8\}$.

3.2 Multi-way replicated refinement

Our multi-way replicated refinement scheme starts with the K-way replicated declustering of the dataset $\mathcal{D}$, say $R_K = \{\mathcal{D}_1, \ldots, \mathcal{D}_K\}$, generated by the recursive replicated declustering scheme described in Section 3.1. We iteratively improve R_K by multi-way refinement operations K-way *move* and K-way *replication*. In order to perform these operations we maintain the following gain values for each data item d:

- K-way move gain ($g_m(d,k)$): the reduction to be observed in the overall query processing cost, if d is moved to disk k,
- K-way replication gain ($g_r(d,k)$): the reduction to be observed in the overall query processing cost, if d is replicated in disk k.

If we were to maintain the above gain values for all data items, we would need approximately $2 \times (K-1)$ gain values for each data item, because a data item can be moved or replicated from its current source disk(s) to any of the disks that does not already store it. Instead of this expensive schema, we adapt an efficient greedy approach that was proposed for unreplicated declustering in [4] to support multi-way refinement and we develop a multi-way refinement heuristic suitable for replicated declustering. Our heuristic can perform multi-way move and replication operations. The approach in [4] was based on the observation that a move operation can be viewed as a two-stage process, where in the first stage the data item d^* to be moved is assumed to leave the source disk and in the second stage d^* arrives at the destination disk. The first stage represents the decrease in the load of the source disk due to the relief in processing of the queries related with d^*, resulting with a decrease in the cost. The second stage represents the increase in the load of the destination disk due to the excess in processing of the queries related with d^*, resulting with an increase in the cost. Here we extend this efficient greedy approach to support both multi-way move and replication selection operations. Our adapted schema requires maintenance of only a single gain value (virtual leave gain) for each data item d.

Virtual leave gain $vg(d)$ indicates the number of queries requesting d such that the disk(s) that d resides in serve(s) more than optimal number of data items for these queries. That is, the virtual leave gain of a data item d that resides on disk D_s is:

$$vg(d) = \sum_{q \in Q_+(d,s)} f(q), \text{where} \tag{15}$$

$$Q_+(d,s) = \{q \in Q : d \in q \wedge t_s(q) > r_{opt}(q)\} \tag{16}$$

That is, each query q that requests data item d contributes $f(q)$ to $vg(d)$, if the number of data items in q that are retrieved from disk D_s is greater than the optimal response time $r_{opt}(q)$ of q. This means that it is possible to improve the distribution of query q through moving or replicating data item d to an appropriate destination disk D_z. Thus, virtual leave gain is an upper bound on the actual move or replication gain. We should note here that our definition of virtual leave gain is different from that of [4] in order to support correct computation of multi-way move and multi-way replication operations. A running example demonstrating virtual leave gain updates can be found in Section 2.2 of the Appendix.

Our overall K-way replicated declustering refinement algorithm works as a sequence of multi-way passes performed over all data items. Before starting the multi-way refinement passes, as a preprocessing step, we compute the optimal schedules for all queries once and maintain these schedules in a data structure called $OptSched$. The process of initial optimal schedule calculation is performed using network-flow based algorithms [9]. $OptSched$ is composed of $|Q|$ arrays, where the ith array is of size $|q_i|$ and stores from which disks the data items of q_i are answered in the optimal scheduling. $OptSched$ is kept both to identify bottleneck disks for queries and also to report the actual aggregate parallel response time of the replicated declustering produced by the recursive declustering phase. A bottleneck disk for a query q is the disk from which q requests the maximum number of data items (and hence determines response time $r(q)$).

In a multi-way refinement pass, we start with computing the virtual leave gains of all data items (Appendix, Algorithm 6). At each iteration of a pass, a data item d^* with the highest virtual leave gain is selected. The $K-1$ move and $K-1$ replication gains associated with d^* are computed (Appendix, Algorithm 7), the best operation associated with d^* is selected and performed if it has a positive actual gain and if it obeys the given capacity constraints, and then the virtual leave gain values of the neighboring data items of d^* are updated (Appendix, Algorithm 8). Also the optimal schedules of each query that requests d^* is considered for update in constant time by investigating possible changes in the bottleneck disk of that query. We perform these passes until the obtained improvement is below a certain threshold or we reach a predetermined number of passes.

4 Experimental Results

In this section, we present the results of experiments conducted to compare the performance of the proposed Selective Replicated Assignment (SRA) scheme against the state-of-the-art Random Duplicate Assignment (RDA) and Orthogonal Assignment (OA) schemes. RDA and OA are selected since they are known to perform good for arbitrary queries. Also it is possible to modify these approaches for selective replication. We modified both RDA and OA to support partial replication, and improved RDA such that it utilizes query logs and selects the most frequently requested data items and replicates them at random disks. We call this modified version the Most Frequent Assignment (MFA) scheme.

In our comparisons we used 9 datasets: *Airport*, *Bea*, *Face*, *FR*, *HH*, *Ntar*, *Park*, *Place90*, and *State*. The properties of these datasets are presented in Table 2. The datasets are taken from [4] and divided into 4 groups. *Face* is a collection of gray-scale face images which are

TABLE 2: Properties of datasets.

Class	$\mathcal{D}$ Dataset	$\lvert\mathcal{D}\rvert$	$\lvert\mathcal{Q}\rvert$	Average query size
Image	*Face*	844	1024	23.1
Func. Approx.	*HH*	1638	2000	43.3
	FR	3338	10000	10.0
GIS (Point)	*Airport*	1176	5000	22.8
	Place90	3382	12000	17.9
GIS (Polygon)	*Park*	1022	4000	20.1
	Ntar	8952	10000	29.2
	Bea	10674	20000	26.8
	State	10827	10000	33.5

used to construct an image retrieval system using the algorithm described in [21]. *HH* and *FR* consists of multi-feature point data used for function-approximation experiments [23]. These datasets are indexed into a grid directory with cell size restricted to 16 points as described in [24]. A set of synthetic rectangular and diagonal queries is generated assuming Gaussian distribution for both query sides and centers for each data set. Other datasets consist of GIS data collected from the National Transportation Atlas Databases [25]. *Airport* contains the public use airports and landing facilities in the US. *Place90* contains place locations from the 1990 Census Master Area reference file. *Park* contains the national parks, *Ntar* contains the national transportation analysis regions, *Bea* contains the economic areas, and State contains the US boundaries with integrated shorelines. A set of synthetic rectangular and diagonal queries are generated for the GIS data sets as for the function-approximation data sets. Further details of the datasets and associated query sets can be found in the Appendix.

While testing the performance of MFA and SRA, the query sets for all datasets except *Face* are divided into two equal parts. The first half is used for replication and declustering and the second half is used for testing the performance. The query set for *Face* is composed of all possible queries so it is fully used while declustering and testing of *Face*.

All of the algorithms used in the experiments are implemented in C programming language, and experiments are conducted on a 2GHz Intel Core Duo machine with 2MB L2 cache and 2GB DDR2 667 MHz memory.

Query processing performances of the compared algorithms are tested on K=$16, 24, 32$ disks and the allowed overall replication ratio is varied from 10% to 100%. With 9 different datasets, 3 different disk counts, and 10 different replication ratio values, we present the results of 270 different experiment instances. For each SRA experiment instance, we report the average of 10 runs, since we use randomly generated initial feasible two-way declusterings in our replicated declustering phase.

The query processing performance of a given algorithm is evaluated in terms of the average retrieval overhead per query induced by the resulting replicated declustering. Here, average retrieval overhead per query

TABLE 3: Arithmetic averages of the arO values for K=32 disks over the nine datasets.

	percent distribution of replication ratio among recursive replicated declustering and multi-way refinement phases						
	w.out unrep.	with unreplication					
% rep.	100–0	100–0	80–20	60–40	40–60	20–80	0–100
10%	0.40	0.31	0.29	0.27	0.24	0.23	0.21
20%	0.36	0.28	0.25	0.22	0.19	0.16	0.15
30%	0.32	0.22	0.23	0.19	0.15	0.12	0.11
40%	0.25	0.19	0.20	0.18	0.12	0.09	0.09
50%	0.19	0.15	0.15	0.14	0.10	0.06	0.07
60%	0.15	0.12	0.13	0.13	0.09	0.05	0.06
70%	0.12	0.09	0.11	0.11	0.09	0.04	0.05
80%	0.10	0.07	0.09	0.09	0.08	0.03	0.04
90%	0.07	0.05	0.07	0.07	0.08	0.03	0.03
100%	0.06	0.04	0.06	0.06	0.07	0.02	0.02

(arO) for a given replicated declustering of a dataset and a query set is defined as total response time overhead (Equation 5) divided by the number of queries. That is,

$$arO(Q) = TrO(R_K, Q)/|Q|. \tag{17}$$

In Table 3, we present the arithmetic averages of the average retrieval overhead of SRA over the nine datasets with increasing replication ratio, where the allowed replication ratio is distributed between the recursive replicated declustering and multi-way refinement phases according to the percentage values displayed over the columns. For example, the column header 80–20 indicates that the recursive replicated declustering phase is allowed to utilize 80% of the replications and the multi-way refinement phase is allowed to utilize 20% of the replications. The values in the table indicate the retrieval overhead of the replicated declusterings obtained by SRA under the given replication distribution.

The second column of Table 3 is introduced to justify the usage of unreplication operation in recursive replicated declustering phase. Note that the 100%–0% replication-distribution scheme provides an approach where replication is only performed in recursive replicated declustering phase. The third and second columns of Table 3 show the performance of such a system where unreplication operation is utilized and not utilized, respectively. By comparing these two columns we can observe that embedding unreplication operation always improves the performance of the recursive replicated declustering phase.

As seen in Table 3, especially for low replication ratios (between 10% to 30%), the average results obtained by SRA are best when the given replication amount is fully utilized in the multi-way refinement phase (0%–100% replication-distribution). However, for higher replication ratios (between 40% to 100%), best results are obtained in the 20%–80% replication-distribution scheme. These results indicate that, for small allowed replication ratios, performing replications at a later phase, that is dur-

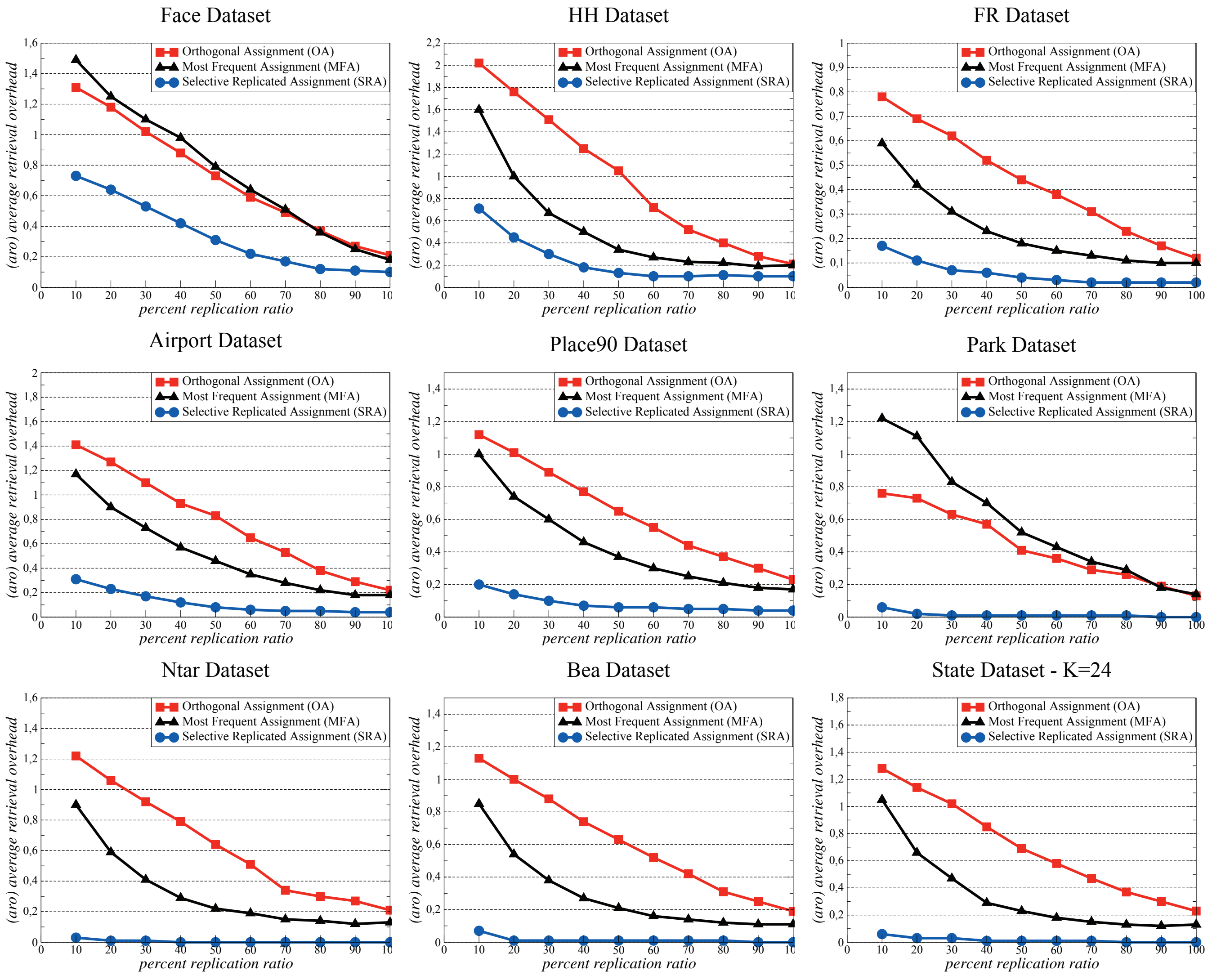


Fig. 2: Average retrieval overhead vs replication ratio figures for the datasets for $K = 24$.

ing the K-way declustering phase, brings more gain, whereas for higher allowed replication ratios, performing a small percent of the replications at an earlier phase, that is during the recursive bipartitioning phase, has a more positive effect on the overall SRA performance. Since 20%–80% replication-distribution scheme has better results for more experiment instances, the results reported for SRA in the following figures are obtained with this replication-distribution scheme. The good results observed for the 0%–100% and 20%–80% replication-distribution schemes point to the success of our multi-way replicated refinement algorithm. The fact that 20%–80% replication-distribution scheme, which is a combination of recursive replicated declustering and multi-way replicated refinement schemes most of the time outperforms the 0%–100% scheme, which is an approach where replication and declustering is decoupled demonstrates the need for our recursive replicated declustering algorithm.

Fig. 2 display the individual performances of the algorithms for K=24 disks over the 9 datasets. Similar detailed analysis for K=16 and K=32 disks can be found in Section 5 of the Appendix. In the figure, variation of the arO values of algorithms with increasing replication ratio is presented. Closer points to x-axis mean better average retrieval times. As seen in the figure, SRA has better (smaller) average retrieval time than MFA and OA for all experiment instances. While comparing MFA with OA, MFA performs much better than OA in seven of the nine datasets. Only in *Face* and *Park* datasets OA performs slightly better than MFA. We observe that with increasing replication amount, the deviation of OA from the strictly optimal declustering decreases linearly, whereas in both MFA and SRA we observe a quadratic decrease. These results point to the importance of using query logs in improving performance, since MFA also makes use of query logs by replicating frequently requested items. An analysis of Fig. 2 reveals that the performance gap between the proposed SRA algorithm and the state-of-the-art MFA and OA algorithms decreases with increasing replication amount. However, as also seen in the figure, SRA still performs considerably better then MFA and OA even for high replication amounts.

An analysis of the arithmetic averages of the average retrieval overheads and the running times of the MFA, OA and SRA over the nine datasets with increasing replication ratio reveals that, for K =16, 24 and 32 disks, even with low replication ratios such as 10%, SRA achieves very low overheads and to achieve similar overheads MFA requires around 70%, whereas OA requires around 90% replication. Detailed experiments supporting these deductions can be found in Section 5 of the Appendix.

5 Conclusions

In this work, we proposed an effective K-way replicated declustering scheme that utilizes a given query distribution. We first propose an iterative-improvement based two-way replicated declustering scheme, which iteratively improves the quality of a two-way replicated declustering. We recursively apply this two-way scheme to obtain a K-way replicated declustering. We then propose an efficient and effective multi-way refinement scheme that can perform multi-way move and replication of data items. With this scheme, we further improve the quality of the obtained K-way declustering and improve balance if possible. Obtained results indicate the merits of utilizing query logs in partial and selective replication. The proposed scheme achieves much better results compared to state-of-the-art replicated declustering schemes, many times achieving optimal overall response time with less than 100% replication ratio.

In this work we assume homogeneous data item retrieval times and homogeneous disks. Heterogeneity in both aspects can be considered for further research studies.

References

[1] D. R. Liu and S. Shekhar, "Partitioning similarity graphs: a framework for declustering problems," *Information Systems*, vol. 21, pp. 475–496, 1996.

[2] D. R. Liu and M. Y. Wu, "A hypergraph based approach to declustering problems," *Distributed and Parallel Databases*, vol. 10(3), pp. 269–288, 2001.

[3] A. S. Tosun, "Threshold-based declustering," *Information Sciences*, vol. 177(5), pp. 1309–1331, 2007.

[4] M. Koyuturk and C. Aykanat, "Iterative-improvement-based declustering heuristics for multi-disk databases," *Information Systems*, vol. 30, pp. 47–70, 2005.

[5] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," in *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 173–182, 1996.

[6] S. Ghemawat, H. Gobio, and S. T. Leung, "The google file system," *ACM SIGOPS Operating Systems Review*, vol. 37(5), pp. 29–43, 2003.

[7] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazons highly available key-value store.," in *Proc. 21st ACM SIGOPS symposium on Operating systems principles*, pp. 205–220, 2007.

[8] A. S. Tosun and H. Ferhatosmanoglu, "Optimal parallel i/o using replication," in *Proc. International Workshops on Parallel Processing*, pp. 748–753, 2002.

[9] L. T. Chen and D. Rotem, "Optimal response time retrieval of replicated data," in *Proc. 13th ACM Sigact-Sigmod-Sigart symposium on principles of database systems*, pp. 36–44, 1994.

[10] K. B. Frikken, "Optimal distributed declustering using replication," in *Proc. 19th International Conference on Database Theory - ICDT*, pp. 144–157, 2005.

[11] H. Ferhatosmanoglu, A. S. Tosun, G. Canahuate, and A. Ramachandran, "Efficient parallel processing of range queries through replicated declustering," *Distributed and Parallel Databases*, vol. 20(2), pp. 117–147, 2006.

[12] A. S. Tosun, "Multi-site retrieval of declustered data," in *Proc. 28th Int'l Conf. Distributed Computing Systems*, pp. 486–493, 2008.

[13] A. S. Tosun, "Analysis and comparison of replicated declustering schemes," *IEEE Trans. Parallel Distributed Systems*, vol. 18(11), pp. 1587–1591, 2007.

[14] A. S. Tosun, "Divide-and-conquer scheme for strictly optimal retrieval of range queries," *ACM Trans. on Storage*, vol. 5(3), 2009.

[15] P. Sanders, S. Egner, and K. Korst, "Fast concurrent access to parallel disks," in *Proc. 11th ACm-SIAM Symp. Discrete Algorithms*, pp. 849–858, 2000.

[16] A. S. Tosun, "Design theoretic approach to replicated declustering," in *Proc. Int'l Conf. Information Technology Coding and Computing*, pp. 226–231, 2005.

[17] A. S. Tosun, "Replicated declustering for arbitrary queries," in *Proc. 19th ACM Symp. Applied Computing*, pp. 748–753, 2004.

[18] K. Y. Oktay, A. Turk, and C. Aykanat, "Selective replicated declustering," in *Proc. 15th International Euro-Par Conference on Parallel Processing*, pp. 375–386, 2009.

[19] T. Kwon and S. Lee, "Load-balanced data placement for variable-rate continuous media retrieval," in *Multimedia Database Systems*, pp. 185–207, 1996.

[20] H. Pang, B. Jose, and M. S. Krishnan, "Resource scheduling in a high-performance multimedia server," *IEEE Trans. on Knowl. and Data Eng.*, vol. 11, no. 2, pp. 303–320, 1999.

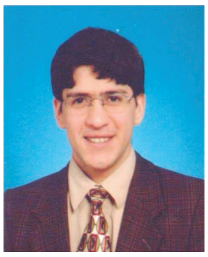
[21] C. E. Jacobs, A. Finkelstein, and D. H. Salesin, "Fast multiresolution image querying," in *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pp. 277–286, 1995.

[22] L.-T. Liu, M.-T. Kuo, S.-C. Huang, and C.-K. Cheng, "A gradient method on the initial partition of fiduccia-mattheyses algorithm," in *IEEE/ACM international conference on Computer-aided design*, pp. 229–234, 1995.

[23] H. A. Guvenir and I. Uysal, "Bilkent university function approximation repository." http://funapp.cs.bilkent.edu.tr, 2000.

[24] S. Hannan, *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[25] "National transportation atlas databases." CD-ROM, 1999. Bureau of Transportation Statistics.



Ata Turk received his B.Sc. and M.Sc. degrees from the Computer Engineering Department of Bilkent University, Turkey, in 2002 and 2004, respectively. He is currently working towards a Ph.D. degree at Bilkent University. His research interests include parallel information retrieval and algorithms.

Kerim Yasin Oktay received his BS degree from Bilkent University, Ankara in computer science. Currently, he is working to complete MS degree and doing research with Prof. Sharad Mehrotra in the Department of Computer Science at University of California, Irvine. His research areas are basically secure database systems and data privacy on the cloud systems.

Dr. Cevdet Aykanat received his B.S. and M.S. degrees from Middle East Technical University, Ankara, Turkey, both in Electrical Engineering, and his Ph.D. degree from Ohio State University, Columbus, US, in Electrical and Computer Engineering. He was a Fulbright scholar during his Ph.D. studies. He worked at the Intel Supercomputer Systems Division, Beaverton, Oregon, US, as a research associate. Since 1989, he has been affiliated with the Department of Computer Engineering, Bilkent University, Ankara, Turkey, where he is currently a professor. His research interests mainly include parallel computing, parallel scientific computing and its combinatorial aspects, parallel computer graphics applications, parallel data mining, graph and hypergraph theoretic models for load balancing, high performance information retrieval systems, parallel and distributed databases, and grid computing. He has (co)authored about 70 technical papers published in academic journals indexed in the ISI, and his publications have received about 560 citations in ISI indexes. He is the recipient of the 1995 Young Investigator Award of The Scientific and Technological Research Council of Turkey and 2007 Parlar Science Award. He was appointed a member of IFIP Working Group 10.3 (Concurrent System Technology) in April 2004, a member of the EU-INTAS Council of Scientists in June 2005, and an Associate Editor of IEEE Transactions of Parallel and Distributed Systems in December 2008.