

A Runtime Framework for Partial Replication and Its Application for On-Demand Data Exploration *

Sivaramakrishnan Narayanan, Umit Catalyurek, Tahsin Kurc,
Vijay Shiv Kumar, Joel Saltz

Department of Biomedical Informatics, The Ohio State University
Columbus, OH, 43210

{krishnan,umit,kurc,vijayskumar,jsaltz}@bmi.osu.edu

Keywords: data replication, data reorganization, data-intensive computing.

Abstract *Data replication can be used to increase data availability and to improve I/O performance of read-intensive applications. We investigate support for partial data replication and data reordering within a generic runtime framework to provide efficient support for range queries over scientific datasets on parallel and distributed storage systems. Replication enables usage of additional I/O bandwidth to answer queries. Re-organization of data can reduce spurious I/O in storage nodes. An experimental evaluation of the framework is presented using large datasets generated in oil reservoir simulation studies.*

1 Introduction

In many fields of science, engineering, and medicine, research is increasingly becoming data driven. Datasets may be generated from large-scale numerical simulations running on high-performance machines. Such datasets can also arise in imaging or sensor data associated with geophysical sensors, satellites, microscopy or imaging devices used in materials science. Increasing resolution of advanced sensors and more powerful computers have made it possible to generate collections of these datasets that can easily surpass tens or hundreds of terabytes in size. The analysis of these datasets usually involves extracting a region, expressed as a range query.

Disk space is inexpensive and is continuously getting cheaper. Through the collective use of multiple disks in the system, storage clusters provide substantial storage space

and high I/O bandwidth. However, software system support for applications still remains a challenging problem. A critical step in the analysis of data is to extract the data of interest from large and potentially distributed datasets and move it from storage clusters to compute clusters for processing. Several optimizations may be used to minimize the time spent for retrieving the data of interest. Datasets could be partitioned and declustered across multiple disks to take advantage of aggregate storage space and I/O bandwidth [1–3]. Indices could be built to speed up searches for data elements that intersect a given query [4].

As an example application scenario, let us consider analysis of very large simulation datasets from a long-running environmental simulation application. For instance, a reactive chemical transport simulation for subsurface waterways simulates multiple chemicals and fluid velocity in a reservoir [5]. For large scale studies, the size of a single dataset can reach tens of terabytes. Common analysis scenarios involve queries for economic evaluation as well as technical evaluation, such as determination of representative realizations and identification of areas of bypassed oil. Such analysis scenarios would represent different types of access into the dataset. In that case, replicating and reorganizing data on disk can result in performance gains. Full data replication considering query access patterns has also been discussed in literature [6]. The size of datasets under consideration precludes this approach. In this work, we consider replicating the 'popular' regions of the dataset on additional storage nodes. We investigate efficient support for choosing and creating partial replicas and for query execution with replicas within a generic runtime framework. An experimental evaluation of the framework is presented using large datasets generated in oil reservoir simulation studies.

*This research was supported in part by the National Science Foundation under Grants #ACI-9619020 (UC Subcontract #10152408), #EIA-0121177, #ACI-0203846, #ACI-0130437, #ANI-0330612, #ACI-9982087, #CCF-0342615, #CNS-0406386, #CNS-0426241, Lawrence Livermore National Laboratory under Grant #B517095 (UC Subcontract #10184497), NIH NIBIB BISTI #P20EB000591, Ohio Board of Regents BRTTC #BRTT02-0003.

2 Problem Definition

Replication can be done at different levels. The granularity could be at the 1) dataset level, 2) file level, 3) query level, 4) chunk level, and 5) element (tuple) level. In other words, at the coarsest granularity, the entire dataset is replicated. This may be unfeasible for large datasets and may also be unnecessary as not all data elements in a dataset are accessed with the same frequency. At the other extreme, single data elements can be replicated. However, for large datasets, it may be expensive to monitor accesses to all data elements, even though this may minimize storage space needed for data replication.

We define three entities; *data element*, *chunk* and *dataset*. The data element is the equivalent of a tuple in a database. A chunk is the smallest unit of I/O. It is a grouping of data elements. A dataset is a collection of chunks. Let $\mathcal{D} = \{C_1, C_2, \dots, C_n\}$ be a dataset with n chunks. Each chunk C_i is a set of data elements $\{t_{i1}, t_{i2}, \dots, t_{ik}\}$ where k is the size of the chunk. Each chunk is associated with an I/O cost $g(C_i)$. $\bigcup \mathcal{D}$ denotes the set of all data elements in \mathcal{D} . A partial replica \mathcal{P} of \mathcal{D} is a non-empty dataset $\{P_1, P_2, \dots, P_m\}$ such that $\bigcup \mathcal{P} \subseteq \bigcup \mathcal{D}$.

Queries to datasets are usually defined by a declarative language like SQL. In this work, regardless of the language the query has been defined in, a query $Q \subseteq \bigcup \mathcal{D}$. We define three problems in the context of partial replication.

Definition 2.1 (Replica Decision Problem) *Given a dataset \mathcal{D} , a cost function g and a set of all possible queries \mathcal{Q} with probability distribution f , find a partial replica set $\{\mathcal{P}_0, \mathcal{P}_1, \dots, \mathcal{P}_P\}$ such that the weighted average execution time (WAET) for queries in \mathcal{Q} is minimized. For a query set \mathcal{Q} ,*

$$WAET(\mathcal{Q}) = \frac{1}{|\mathcal{Q}|} \times \sum_{Q \in \mathcal{Q}} f(Q) \times QueryCost(Q) \quad (1)$$

In this problem, two separate questions, what-to-replicate and where-to-replicate, need to be solved together.

Definition 2.2 (Replication Problem) *Given a dataset \mathcal{D} and metadata for partial replication set $\{\mathcal{P}_1, \mathcal{P}_2, \dots, \mathcal{P}_P\}$, optimize the execution of repartitioning, reordering, and transfer operations on data chunks in \mathcal{D} and in $\{\mathcal{P}_1, \mathcal{P}_2, \dots, \mathcal{P}_P\}$ so that the total execution time is minimized.*

Definition 2.3 (Query Execution Problem) *Given a dataset \mathcal{D} and its partial replicas $\{\mathcal{P}_0, \mathcal{P}_1, \dots, \mathcal{P}_P\}$ where $(\mathcal{D} = \mathcal{P}_0)$ and a query $Q \subseteq \bigcup \mathcal{D}$, determine $\{A_1, A_2, \dots, A_P\}$ where $A_i \subseteq \mathcal{P}_i$ satisfying $Q \subseteq \bigcup \{A_1, A_2, \dots, A_P\}$ such that the query cost is*

minimized.

$$QueryCost(Q) = \sum_{i=1}^P \sum_{C \in A_i} g(C) \quad (2)$$

The problem is to find the set of chunks to be retrieved from each replica so that query execution cost is minimized and the query is satisfied.

3 A Middleware Framework for Partial Replication with Data Reorganization

We develop a runtime framework to facilitate efficient execution of partial replication of scientific datasets on a distributed storage and processing platform. Since some of the tasks of data replication and query execution involve application-specific or user-defined data structures and algorithms, the framework is designed as a set of customizable services. It is implemented as an extension to the STORM [7, 8] middleware framework. Scientific datasets are usually stored in a set of flat files with application-specific file formats. STORM is a service-based, distributed middleware that provides basic database support for 1) *selection of the data of interest* from scientific datasets stored in files and 2) *transfer of selected data from storage nodes to compute nodes for processing*. The data of interest is selected based on attribute values or ranges of values.

In this section we present the replica service that has been implemented within the STORM framework to facilitate data replication and execution of queries with replicas. This service consists of three sub-services. The *replica identification* service is tasked with selecting the subset of a dataset for replication. The *replica creation* service handles the creation of a replica from the data subset identified by the replica identification service. The *query execution* service is responsible for execution of a query when there are partial replicas. It determines replicas or subsets of replicas that can be used to answer the query efficiently.

3.1 Replica Identification

Identification of data subset for replication can be done at the chunk level. The responsibility of the first sub-service is to identify which chunks of a dataset will be replicated. This could be accomplished in several ways. A possible way is to determine a region of interest given by a representative query and replicate a subset of chunks in that region. Another way is to associate a popularity counter with each chunk based on query history. A subset of the most popular chunks may be chosen for replication. Our implementation uses the first technique. The size of the subset is controlled by replication ratio k . The selection of k depends on availability of time and resources. We study the effect of varying k in our experiments.

3.2 Replica Creation

The replica creation sub-service is tasked to retrieve chunks identified by the identification sub-service, repartition and reorder the data elements, compute the declustering of the data, copy the data to destination nodes, create indices for the replica, and update the metadata of the original data to reflect the existence of the replica.

In this paper we investigate two possible repartitioning/reordering strategies. The first strategy, *uniform partitioning*, evenly partitions the domain encompassing the chunks selected for replication into equal multi-dimensional rectangles, each of which corresponds to a new chunk. Note that although the dimensions of rectangles are the same, the number of elements in each chunk may be different because of a possible non-uniform distribution of data elements in multi-dimensional space. To address this, the second method employs a multi-stage adaptive bisection (referred to here as *recursive partitioning*). The algorithm starts from one of the dimensions and divides it into sections such that each section has equal number of data elements. Next, the algorithm partitions each section along the next dimension such that each subsection has equal number of data elements. The algorithm proceeds until all or user-defined set of dimensions are covered. Each of the resulting subsections (multi-dimensional rectangles) corresponds to a new chunk. In the current implementation, once new chunks have been created, a declustering of the chunks to storage nodes is computed using a Hilbert-curve-based declustering algorithm [2]. After declustering, indices are built on the set of new chunks and they are registered as a new dataset.

An extension to this would be reordering data within replica chunks and splitting them into smaller chunks. Hierarchical chunking can help reduce spurious I/O especially when output chunks are large. For example, data within a chunk could be ordered by a computed attribute and smaller fixed-size chunks created. Such chunking will reduce the execution of the range queries involving that computed attribute. Another example is sub-sampling queries of the form $x \% 2 == 0$ can be answered more efficiently if output chunks are ordered on by $x \% 2$ with a small fixed-size.

3.3 Query Execution

When partial replication optimizations are employed, a query can be answered from various data subsets and replicas of datasets. In addition, it is possible that no single data subset can fully answer the query. Even more, if data replicas have been created using repartitioning with reordering, there will not be one-to-one mapping between chunks in the original dataset and those in the replicas. It is query execution sub-service’s job to solve this complex problem.

In the step 1, query box is intersected against the original dataset. This step is actually an index lookup, where an effi-

Inputs:

- Q : Query box.
- $\mathcal{D} = \{C_1, C_2, \dots, C_n\}$: original dataset.
- \mathcal{R} set of replicas.

Algorithm:

- 1 $\mathcal{O} \leftarrow \{C_i | C_i \in \mathcal{D} \text{ and } C_i \cap Q \neq \emptyset\}$
 - 2 Sort \mathcal{R} in decreasing order of suitability for Q
 - 3 **while** \mathcal{O} is not empty **AND** \mathcal{R} is not empty
 - 3.1 $R \leftarrow \text{dequeue}(\mathcal{R})$
 - 3.2 Let $RC = \{C_i | C_i \in \mathcal{D} \text{ was used to create replica } R\}$
 - 3.3 **if** $RC \cap \mathcal{O} \neq \emptyset$
 - 3.3.1 Process chunks in $RC \cap \mathcal{O}$ from replica R
 - 3.3.2 $\mathcal{O} \leftarrow \mathcal{O} - RC$
 - 4 **endwhile**
 - 5 **if** \mathcal{O} is not empty
 - 5.1 Process chunks in \mathcal{O} from \mathcal{D}
-

Figure 1. Query execution algorithm

cient tree based index, such as R-Tree [4], can be used. This identifies the chunks in the original dataset that intersect the query. Replicas are considered in decreasing order of their suitability to the query. For e.g., Replicas partitioned along dimensions specified in the query may be considered earlier. For each replica, the algorithm checks if any chunk that has been used to create this replica can be used to answer the query in step 3.3. In step 3.3.1 we process data elements from replica R , using STORM’s existing services as the replica is registered as a dataset. In step 3.3.2 we compute the remaining set of chunks that needs to be processed from the original dataset. After all replicas are processed, the remaining original chunks are retrieved and processed to answer the portions of the query that have not been answered by replicas.

4 Related Work

In the context of replication, most of the previous work targets issues like data availability during disk and/or network failures, as well as to speed up I/O performance by creating exact replicas of the datasets [9–14]. File level and dataset level replication and replica management have been well studied topics [11, 12]. In the area of partial replication [9, 13, 14], the focus has been on creating exact replicas of portions of a dataset to achieve better I/O performance. In this work, however, the goal is to investigate the partial replication of datasets by restructuring and reordering the data.

The Livny Problem presented in [15] relates to the optimization of transfer of files to a central location and is a sub-problem in this context. The query execution problem presented in this paper resembles the Plank-Beck Problem

in [15, 16]: “Given a single large file divided into k ordered segments of uniform length in which each segment has r replicas distributed across hosts, minimize the time necessary to deliver each segment in order”¹ [15]. In the Plank-Beck problem, segment replicas are identical. Thus, there is one-to-many relation between the original data and replicas, whereas in our problem the relation is more complex many-to-many. The second difference is that the segments should be delivered in order as per the Plank-Beck problem [15]. Our problem definition is not strict about the ordering.

Several research projects have looked at improving I/O performance using different declustering techniques [1–3]. Parallel file systems and I/O libraries have also been a widely studied research topic, and many such systems and libraries have been developed [17–19]. These systems mainly focus on supporting regular strided access to uniformly distributed datasets, such as images, maps, and dense multi-dimensional arrays.

In data caching context, Dahlin et al. [20] have evaluated algorithms that make use of remote memory in a cluster of workstations. Franklin et al. [21] have studied ways for augmenting the server memory by utilizing client resources. Venkataraman et al. [22] have proposed a new buffer management policy, which approximates a global LRU page replacement policy. These approaches can be employed for management and replacement of replicas. The main difference is while data caching maintains output of queries already executed in the system, data replication aims to improve query execution performance by replicating, reorganizing, and redistributing input data.

5 Experimental Results

We have evaluated our methods using a large dataset with characteristics similar to those created by oil reservoir management applications. All of the experiments were carried out on 8 nodes in a Linux cluster where each node has a PIII 933MHz CPU, 512 MB main memory, and three 100GB IDE disks. The nodes are inter-connected via a Switched Fast Ethernet.

A typical study of oil reservoir management [23] involves a large collection of simulations (also referred to as realizations) that model the effects of varying oil reservoir properties (e.g., permeability, oil/water ratio, etc.) over a long period of time. Simulations are carried out on a three-dimensional grid. At each time step, the value of seventeen separate variables and cell locations in 3-dimensional space are output for each cell in the grid. If the simulation is run in

¹It is stated as “Given a piece of content (e.g. a file) that is striped and replicated in the wide-area, how can that content best be delivered to a client?” in [16]

parallel, the data for different parts of the domain can reside on separate disks or nodes.

We synthetically generated a 0.35TB size dataset, with the same domain partitioning as the simulator code, so that we can manage the distribution of attributes for a more controlled experimental environment. Each grid point is stored as a tuple and each tuple consists of 21 attributes. 16 time steps worth of data is created using a grid of size $1024 \times 1024 \times 256$. The data is partitioned into $8 \times 8 \times 256 \times 1$ size chunks on X , Y , Z and $TIME$ dimensions (attributes). Each chunk is roughly 1.3MB in size. The metadata associated with a chunk includes lower and upper bounds of each attribute along with a $(filename, offset, size)$ triple that is required to retrieve it. The chunks were declustered across the data nodes using a Hilbert curve traversal along X and Y axes. Each node maintains a local index of its chunks’ metadata. The index takes a query as input and returns the list of chunks whose bounds intersect with the range of the query. The attributes that we will focus on are $SOIL$ (oil saturation), which has a uniform distribution in $[0, 1]$, and the VX (oil velocity in x-dimension) attribute which has a standard normal distribution. Queries will be of the form $Q = [(l_X, l_Y, l_T, l_{SOIL}, l_{VX}) : (h_X, h_Y, h_T, h_{SOIL}, h_{VX})]$.

We used a single representative query to determine a region of interest. As described in Section 3.1, we consider the chunks whose bounds intersect with the query. Assume there are N such chunks. For a replication ratio of k , we choose Nk random chunks from this set for replication. Original chunking of the data is based on X , Y , Z and $TIME$ attributes. We have created replicas of this dataset by repartitioning using $SOIL$ and VX attributes. Two partitioning schemes have been compared here: uniform partitioning and recursive partitioning.

Uniform partitioning is a naive method that is insensitive to the actual distribution of the attribute(s). It involves dividing up the domain of the attributes into regular, fixed size partitions. To compute the recursive partition, we randomly sampled 0.1% of the data in the chosen chunks and recursively divided the space along $SOIL$ and VX dimension. In both cases we created 100×10 partitions along $SOIL$ and VX dimensions.

The end-to-end execution time of the replica creation process for replication ratios 0.25, 0.50, 0.75 and 1.0 are shown in Figure 2. This corresponds to replica sizes of 10.5GB, 21GB, 31.5GB and 42GB. As seen in the figure, replica creation time is linear to the ratio of replication. Note that since the partition is global, there is an exchange of data among the nodes and hence the process is bounded by the network bandwidth. The number of tuples replicated increases linearly with increasing replication ratio.

The recursive partitioning technique is more expensive than the naive uniform partitioning method because of retrieving and processing sample data. However, as Figure 3

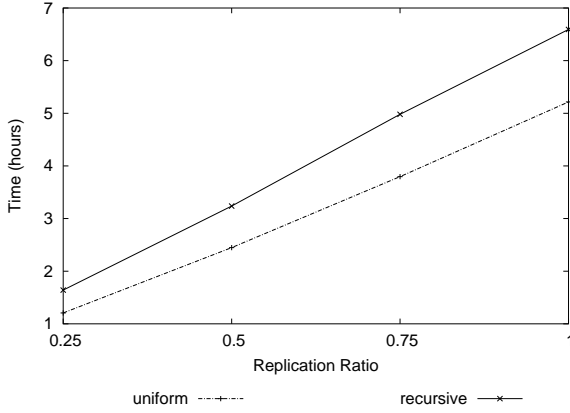


Figure 2. Total execution time of replica creation process.

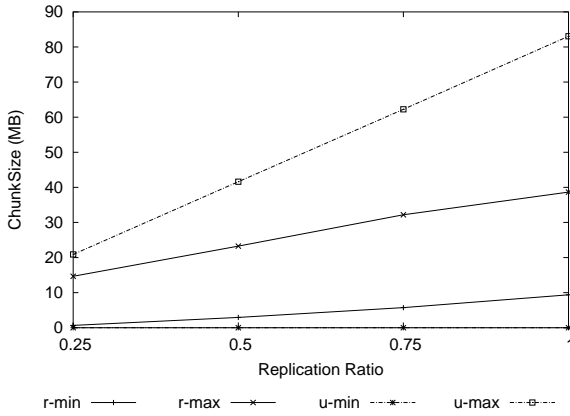


Figure 3. Min and Max chunk sizes in the uniform and recursive partitioning.

shows, there is less of a variation in chunk sizes when recursive partitioning is employed. The variation exists because the partition was based on sample data. One may be able to reduce the variation by choosing more sample data to build the partition on. Thus, there is a tradeoff between speed and accuracy. Uniform partitioning can result in some chunks having a lot of data while some have little or no data. We then decluster the resultant replica chunks using a 2-D Hilbert curve. We noticed that declustering results in an even distribution of tuples among the nodes for both partition techniques despite variations in chunk sizes.

We now focus on how we use these replicas to answer queries. In query execution we have used algorithm displayed in Figure 1. The set of queries we investigate are all within the subregion $R = [(0, 0, 0, 0, -5) : (511, 511, 7, 1.0, 5)]$ and involve a sliding window in the VX dimension. Specifically $Q_i = [(0, 0, 0, 0.0, -5.0 + i) : (511, 511, 7, 1.0, -4.75 + i)]$ where i is the the query number.

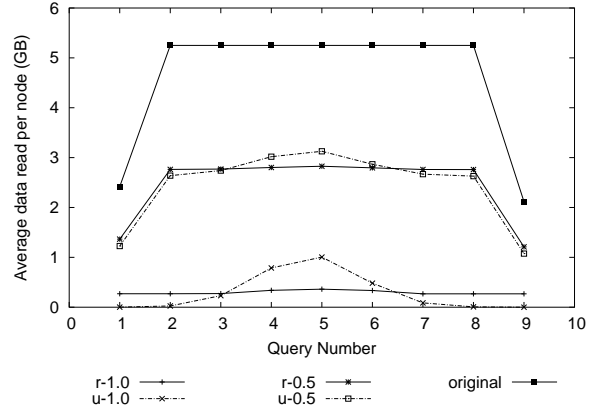


Figure 4. Total I/O to execute Q_i .

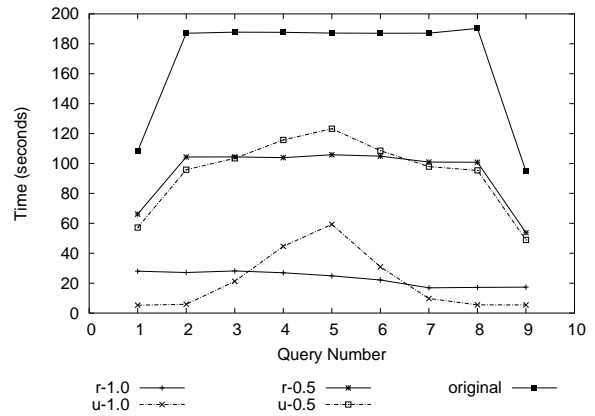


Figure 5. Query execution time with different replicas.

Figure 4 shows the average amount of data read per node to answer the queries. Figure 5 shows the time taken to execute the queries. Performance of replication ratios 0.5 and 1.0 are shown in the above figures. Since the original data was not partitioned on the queried attribute, the amount of spurious I/O is enormous when the original dataset is used. The replicated data is partitioned along *SOIL* and *VX* dimensions using both uniform and recursive partitioning techniques. This decreases spurious I/O and improves query performance.

We can see increased benefits for the sliding window queries as the replication ratio is increased. Comparing the partition techniques, we can see that at queries around the mean (Q_4, Q_5, Q_6), recursive partition does better since the partitions along the *VX* dimension are finer near the dense areas resulting in less spurious I/O. However, for queries at the extreme values of *VX*, uniform partitioning does better since recursive partitions are much coarser near the extreme regions ($Q_1, Q_2, Q_3, Q_7, Q_8, Q_9$) and result in more spurious I/O.

6 Conclusion and Future Work

We have investigated the benefits of using partial replicas of very large scientific datasets and the support for efficient partial replication. We have presented a framework for partial replication and query execution with replicas. We described an implementation using a middleware infrastructure of the framework for multi-dimensional datasets and range queries. Our experiments on large datasets generated in oil reservoir simulation studies show that replicas do not need to be created on faster storage mediums to increase query performance. Creating replicas with data reorganization can improve the performance even if the replica is created on the same storage medium that stores the original data.

As a future work, we plan to investigate mathematical models of cost and benefits of creating partial replicas. If the cost and benefits can be modeled, it would be possible to create an automated system which will mine query history and decide to create partial replicas automatically.

References

- [1] David DeWitt and Jim Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, June 1992.
- [2] Christos Faloutsos and Pravin Bhagwat. Declustering using fractals. In *Proceedings of the 2nd International Conference on Parallel and Distributed Information Systems*, pages 18–25, January 1993.
- [3] Bongki Moon and Joel H. Saltz. Scalability analysis of declustering methods for multidimensional range queries. *IEEE Transactions on Knowledge and Data Engineering*, 10(2):310–327, March/April 1998.
- [4] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of SIGMOD’84*, pages 47–57. ACM Press, May 1984.
- [5] Tahsin Kurc, Umit Catalyurek, Chialin Chang, Alan Sussman, and Joel Saltz. Exploration and visualization of very large datasets with the Active Data Repository. Technical Report CS-TR-4208 and UMIACS-TR-2001-04, University of Maryland, Department of Computer Science and UMIACS, January 2001. Also appears in *IEEE Computer Graphics and Applications*, July/August 2001.
- [6] Sarawagi and Stonebraker. Efficient organization of large multidimensional arrays. In *ICDE: 10th International Conference on Data Engineering*. IEEE Computer Society Technical Committee on Data Engineering, 1994.
- [7] Sivaramakrishnan Narayanan, Umit Catalyurek, Tahsin Kurc, Xi Zhang, and Joel Saltz. Applying database support for large scale data driven science in distributed environments. In *Proceedings of the Fourth International Workshop on Grid Computing (Grid 2003)*, pages 141–148, Phoenix, Arizona, Nov 2003.
- [8] Sivaramakrishnan Narayanan, Tahsin Kurc, Umit Catalyurek, and Joel Saltz. Database support for data-driven scientific applications in the grid. *Parallel Processing Letters*, 13(2):245–271, 2003.
- [9] Ali Saman Tosun and Hakan Ferhatosmanoglu. Optimal parallel i/o using replication. In *Proceedings of International Workshops on Parallel Processing ICPP*, Vancouver, Canada, August 2002.
- [10] Peter Sanders, Sebastian Egner, and Jan Korst. Fast concurrent access to parallel disks. In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, pages 849–858. Society for Industrial and Applied Mathematics, 2000.
- [11] K. Ranganathan and I. Foster. Identifying dynamic replication strategies for high performance data grids. In *Proceedings of International Workshop on Grid Computing*, Denver, CO, November 2002.
- [12] Ann Chervenak, Ewa Deelman, Ian Foster, Leanne Guy, Wolfgang Hoschek, Adriana Iamnitchi, Carl Kesselman, Peter Kunszt, Matei Ripeanu, Bob Schwartzkopf, Heinz Stockinger, Kurt Stockinger, and Brian Tierney. Giggie: a framework for constructing scalable replica location services. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–17. IEEE Computer Society Press, 2002.
- [13] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–186, June 1994.
- [14] Chung-Min Chen and Christine T. Cheng. Replication and retrieval strategies of multidimensional data on parallel disks. In *Proceedings of the twelfth international conference on Information and knowledge management*, pages 32–39. ACM Press, 2003.
- [15] M. Allen and R. Wolski. The livny and plank-beck problems: Studies in data movement on the computational grid. In *Supercomputing 2003*, November 2003.
- [16] J. S. Plank, S. Atchley, Y. Ding, and M. Beck. Algorithms for high performance, wide-area distributed file downloads. *Parallel Processing Letters*, 13(2):207–224, June 2003.
- [17] John M. May. *Parallel I/O for High Performance Computing*. Morgan Kaufmann Publishers, 2000.
- [18] P. H. Carns, W. B. Ligon, R. B. Ross, and R. Thakur. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, October 2000.

- [19] Robert Bennett, Kelvin Bryant, Alan Sussman, Raja Das, and Joel Saltz. Jovian: A framework for optimizing parallel I/O. In *Proceedings of the 1994 Scalable Parallel Libraries Conference*, pages 10–20. IEEE Computer Society Press, October 1994.
- [20] Michael Dahlin, Randolph Wang, Thomas Anderson, and David Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proc. Symp. on Operating Systems Design and Implementation*, pages 267–280. ACM Press, 1994.
- [21] Michael J. Franklin, Michael J. Carey, and Miron Livny. Global memory management in client-server DBMS architectures. Technical Report CS-TR-1992-1094, University of Wisconsin, 1992.
- [22] Shivakumar Venkataraman, Jeffrey F. Naughton, and Miron Livny. Remote load-sensitive caching for multi-server database systems. In *Proceedings of the 1998 International Conference on Data Engineering*, pages 514–521. IEEE Computer Society Press, 1998.
- [23] T. Kurc, U. Catalyurek, X. Zhang, J. Saltz, R. Martino, M. Wheeler, M. Peszyńska, A. Sussman, C. Hansen, M. Sen, R. Seifoullaev, P. Stoffa, C. Torres-Verdin, and M. Parashar. A simulation and data analysis system for large scale, data-driven oil reservoir simulation studies. *Concurrency and Computation: Practice and Experience*, accepted for publication., 2004.

Biographies

Sivaramakrishnan Narayanan is a PhD student in the Department of Computer Science and Engineering at The Ohio State University. His research interests include data-intensive computing and scheduling in parallel and distributed environments. He received his B.E.(hons) from Birla Institute of Technology and Science, Pilani, India in 2002.

Umit Catalyurek is an Assistant Professor in the Department of Biomedical Informatics at The Ohio State University. His research interests include graph and hypergraph partitioning algorithms, grid computing, and runtime systems and algorithms for high-performance and data-intensive computing. He received his PhD, M.S. and B.S. in Computer Engineering and Information Science from Bilkent University, Turkey, in 2000, 1994 and 1992, respectively.

Tahsin Kurc is an Assistant Professor in the Department of Biomedical Informatics at the Ohio State University. His research interests include runtime systems for data-intensive computing in parallel and distributed environments, and scientific visualization on parallel

computers. He received his PhD in computer science from Bilkent University, Turkey, in 1997 and his B.S. in electrical and electronics engineering from Middle East Technical University, Turkey, in 1989.

Vijay Shiv Kumar is currently pursuing his PhD at the Department of Computer Science and Engineering, The Ohio State University. His research interests include developing techniques and middleware for efficient storage, retrieval and analysis of large datasets in distributed environments. He received his B.E. in Computer Science and M.Sc. in Chemistry from the Birla Institute of Technology and Science, Pilani, India in 2003.

Joel Saltz is Professor and Chair of the Department of Biomedical Informatics, Professor in the Department of Computer and Information Systems and a Senior Fellow of the Ohio Supercomputer Center. Prior to coming to Ohio State, Dr. Saltz was Professor of Pathology and Informatics in the Department of Pathology at Johns Hopkins Medical School and Professor in the Department of Computer Science at the University of Maryland. He received his M.D. and PhD in computer science from Duke University in 1985 and 1986, respectively. He earned his B.S. in mathematics and physics from University of Michigan in 1978. His research interests are in the development of systems software, databases and compilers for the management, processing and exploration of very large datasets.