I/O-Efficient Statistical Computing with RIOT ⋆

Yi Zhang* Weiping Zhang† Jun Yang‡
Department of Computer Science, Duke University, Durham, NC 27708, USA
{*yizhang, ‡junyang}@cs.duke.edu, †wz14@duke.edu

Abstract— **Statistical analysis of massive data is becoming indispensable to science, commerce, and society today. Such analysis requires efficient, flexible storage support and special optimization techniques. In this demo, we present RIOT (*R with I/O Transparency*), a system that extends R, a popular computing environment for statistical data analysis. RIOT makes R programs I/O-efficient in a way transparent to users. It features a flexible array storage manager and an optimization engine suitable for statistical and numerical operations. RIOT also seamlessly integrates with external database systems, offering additional opportunities for processing data that reside in databases by blurring the boundary between database and host-language processing. This demo will show how statistical computation can be effectively and efficiently handled by RIOT.**

I. Introduction

Recent technological advances have enabled collection of massive amounts of data in science, commerce, and society. These large, high-resolution datasets have brought us closer than ever before to solving important problems such as decoding human genomes and understanding financial phenomena. Across application domains, much of advanced data analysis is done with programs custom-developed by statisticians, scientists, and engineers. They rely heavily on numerical and statistical computing environments such as R (`http://www.r-project.org`) and MATLAB, which provide a high level of abstraction to simplify programming of numerical and statistical computation. Most users learn to program at the level of vectors and matrices instead of using explicit loops to iterate through arrays.

Such computing environments, however, are seriously challenged by the ever-growing size of data, because they typically assume that all data fits in main memory. If the physical memory cannot hold all data, the operating system's virtual memory mechanism starts to swap data to and from disk, often causing thrashing. When performance degrades because of I/Os, users usually rewrite their programs to explicitly manage I/Os using lower-level languages like C or FORTRAN. This approach requires significant effort and expertise, however.

There have been many approaches towards making data-intensive programs I/O-efficient without placing too much burden on users. One approach is I/O-efficient libraries (e.g., SOLAR [1]) that provide, for example, efficient out-of-core matrix multiplication routines. However, we have observed that it is not enough to simply provide efficient implementations of individual operations—many sources of I/O-inefficiency in programs remain at a higher, inter-operation level [2]. For example, it is important to reduce the amount of intermediate results being passed between operations, and be able to defer and reorder operations in optimizing I/Os and computation.

Most I/O-efficient libraries are equipped with special array storage solutions. Dense multidimensional arrays are often partitioned into (hyper)rectangular *chunks*, which are then laid out on disk in particular orders (e.g., [3], [4]). For sparse arrays, only nonzero elements and their indices need to be stored. Some compression may be employed, e.g., the Compressed Column Storage in MATLAB. Multidimensional index structures such as UB-tree [5] can also be used. However, to the best of our knowledge, none of the existing array storage solutions can adapt to the *varying* sparsity over both time and different regions of arrays. For instance, if some region of a matrix is becoming dense due to successive insertions, the storage representation (for this region only) should switch to a dense one to reduce space and improve access performance.

Database systems have also been used for managing large datasets. Designed to be I/O-efficient, they feature a high-level language (SQL) that enables advanced optimization. Most numerical and statistical computing environments provide ways to connect to databases, but SQL is awkward for capturing nontrivial computation and advanced array layouts, and general-purpose database systems are highly inefficient for dense arrays [4], [2]. While there has been work on making database systems more efficient for array-based storage and computation, much of that work is highly database-centric. Effective users must become SQL experts. Unless all their computational needs can be completely satisfied by a database system, they are faced with the difficult challenge of deciding what processing should be done by the database versus the host programming language.

To make our solution appealing to the majority of users in statistical and numerical computing communities, we have proposed to make it completely *transparent* to users how efficient I/O is supported [2]. The resulting prototype system, RIOT-DB, brings I/O-efficiency to R programs without forcing them to be rewritten. RIOT-DB uses a database system for backend processing. While RIOT-DB demonstrated the feasibility of transparent I/O-efficiency and the potential of database-style inter-operator optimizations, it also revealed significant deficiencies of database systems in handling numerical computation.

In this demo, we show the next generation of the RIOT system. RIOT achieves the same transparency as RIOT-DB, but it is more efficient and flexible in many significant ways.

⋆This work is supported by NSF Award IIS-0916027 and REU Supplement to Award IIS-0713498.

First, RIOT replaces the previous database backend with an intelligent array storage manager that can efficiently handle arrays with varying sparsity. Second, RIOT uses an expression algebra to represent computation, which is more expressive than SQL views used by RIOT-DB. The optimization and execution engines of RIOT are tailored towards numerical computation. Finally, although RIOT does not rely on a database backend, RIOT knows when it is beneficial to push computation down to a database system for data that originally resides in a database. Together, these features make RIOT an appealing platform for I/O-efficient statistical computing.

II. Design and Implementation

A. RIOT Architecture

We take a minimally invasive approach in building RIOT. Instead of rebuilding R from scratch to make it I/O-efficient, we build RIOT as an R package using R's extensibility features, and avoid modifying the core R code whenever possible. RIOT can be dynamically plugged into an R environment, and immediately adds I/O-efficiency to R programs. The decision to be modular and minimally invasive makes it easy to apply our techniques to other platforms such as MATLAB. The overall architecture of RIOT is shown in Figure 1. Next we highlight the main aspects of RIOT.

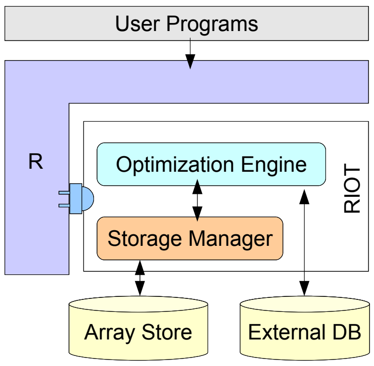

Fig. 1. RIOT architecture.

B. Interfacing with R

RIOT takes the same approach as RIOT-DB [2] when interfacing with R. To be more specific, RIOT defines new data types that correspond to R's built-in vectors, matrices, and arrays (with an arbitrary number of dimensions). These new types implement the same interfaces as their built-in counterparts, and users need not know whether an object has a RIOT type or a built-in type. R's *generics* mechanism enables this transparency. Analogous to *method overloading* in object-oriented programming languages like C++, a generic function in R has polymorphic behavior depending on the types of its arguments. By overloading common functions and operators for RIOT types, we effectively replace R's default methods for handling large data with our more efficient methods. For details and examples, please refer to [2].

We note that the object-oriented programming facilities used above are found in other popular environments as well (e.g., MATLAB), so this approach is portable.

C. Optimization Engine

A baseline solution is to replace R's default implementation of various functions and operators with calls to an I/O-efficient library. For example, to add two large disk-resident vectors without consuming too much memory, we can implement the + operator by a loop that reads the two vectors and produces the result one element at a time. However, this approach exploits I/O efficiency only at the per-operation level.

By contrast, RIOT takes a more aggressive approach. RIOT uses an expression algebra to represent any piece of R code with an acyclic directed graph. Each RIOT object is mapped to a node in the DAG. The result of an operation on RIOT objects becomes a new node (created by the overloaded function), which encapsulates the computation involved in generating this result. However, no computation actually takes place yet—not until the time when evaluation is forced (e.g., by a `print` statement). This approach enables *deferred evaluation*, which is critical to high-level, inter-operation optimization. With pipelined execution, we avoid materializing large intermediate results. We can also selectively evaluate expressions and reordered evaluation for improved I/O efficiency.

To highlight some of the unique aspects of RIOT's optimization engine, consider the following least squares problem.

Example 1. *Suppose a response variable is modeled as a linear combination of n covariates plus some Gaussian noise: $y = \mathbf{w}'\mathbf{x} + \epsilon$, where $\mathbf{x} = (x_1, \dots, x_n)'$ is the covariate vector, $\mathbf{w}$ the weight vector, and ϵ a zero-mean Gaussian random variable. We are given m observations $y_1, \dots, y_m$ (which we group into a column vector $\mathbf{y}$) and their corresponding inputs $\mathbf{x}_1, \dots, \mathbf{x}_m$ (which we group into a $m \times n$ matrix $\mathbf{X}$). It is known that the maximum likelihood estimation of $\mathbf{w}$ is exactly the $\mathbf{w}$ that minimizes the sum-of-squares error function $E(\mathbf{w}) = \sum_{i=1}^{m}(y_i - \mathbf{w}'\mathbf{x}_i)^2$, and the solution is $\mathbf{w}^* = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}$. Suppose both n and m are very large, and we are only interested in w_1. The R code would read:*

```
w <- solve(t(X)%*%X) %*% t(X) %*% y; print(w[1])
```

For the R code in Example 1, RIOT creates one node for each intermediate result object, ending up with an expression DAG as shown in Figure 2(a). RIOT performs several optimizations when executing this DAG. First, the large vector `w` as an intermediate result need not be materialized on disk. Second, since the expression DAG is constructed without performing any actual computation, RIOT will effectively compute only `w[1]` in the end, saving both computation and I/O. Third, RIOT will rearrange the order of matrix multiplications. Let $\mathbf{Z} = (\mathbf{X}'\mathbf{X})^{-1}$, then RIOT will compute $\mathbf{Z}(\mathbf{X}'\mathbf{y})$ instead of $(\mathbf{Z}\mathbf{X}')\mathbf{y}$, as shown in Figure 2(b). This optimization reduces the number of scalar multiplications from $n^2m + nm$ to $2nm$ (ignoring the selective evaluation of `w[1]`). Lastly, RIOT can perform shared scans of data when possible (similar to QPipe [6]). Note that in the expression DAG there

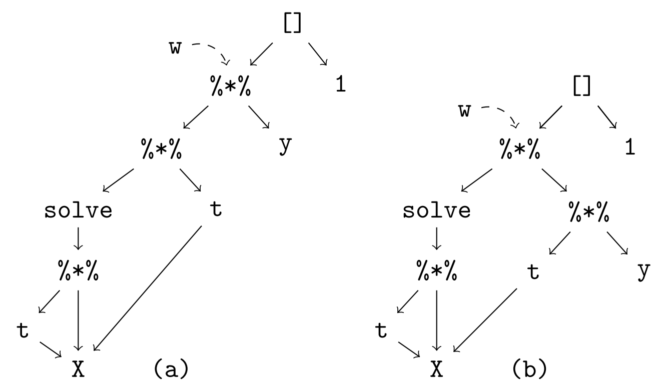

Fig. 2. Expression DAG for the Linear Least Square program.

are two identical transpose operators applied on X. Although not true in this particular example, it is often the case that processing of multiple operators can be shared, significantly reducing I/O.

Some of the optimization opportunities listed above have also been achieved by RIOT-DB, namely avoiding large intermediate results and selective evaluation. The rest, though, is difficult for a relational database backend, primarily because SQL is too low-level for representing many linear algebra operations. RIOT's optimization is much more effective through the use of the high-level semantics of these operations.

RIOT's optimization also covers physical data layouts and related algorithms. Different layouts have dramatic impact on the performance of numerical algorithms [2]. By default, RIOT uses *linearization* based on space-filling curves, and builds indexes on disk-resident data (details below). For maximum flexibility, RIOT also allows programmers to explicitly override its optimizations.

D. Storage Manager

Previous studies have found relational database systems to be inadequate in storing arrays [4], [2], especially dense ones. The main reason is that the relational model does not exploit the fact that arrays are *ordered* collections of data; storing array indices for a dense array wastes storage and only slows down access. As for a sparse array, it is unnecessary to store the zero-valued elements that occupy most of the array; storing only the nonzero elements together with their array indices is more efficient. An index such as a B-tree can be used for fast access to elements. Even though database systems are better at handling sparse arrays, it is still difficult to support various array layouts—i.e., linearization of elements of multidimensional arrays on disk with proper clustering to facilitate efficient access.

Another problem is that array sparsity may vary over time and over different regions of the array. For example, an initially sparse matrix may be gradually populated with new data until a subregion becomes dense. In this case, the subregion should use a dense representation to avoid storing the array indices.

RIOT addresses the above problem by implementing a novel storage manager that handles arrays on the entire dense-sparse continuum. Given an n-dimensional array, an *linearization function* $f : \mathbb{N}^n \to \mathbb{N}$ maps n-dimensional array indices in the form $(i_1, \ldots, i_n)$ to scalars representing addresses inside a linear addressable storage container (e.g., a file on disk). Each nonzero element in the array is then represented by a pair (key, data), where key is the translated address. By default, RIOT does linearization based on space filling curves (e.g., Z-order curve), which preserves the proximity of elements in the high-dimensional index space. RIOT also supports the conventional row- and column-major linearizations when users specify so.

The linearized representation can be indexed by a B-tree for efficient access and update, as is done in UB-trees [5]. However, this idea alone does not lead to an efficient solution for dense regions of an array. Therefore, we propose a new index structure that adapts to varying sparsity of arrays. Our index structure has a similar internal organization to a UB-tree, but differs at the leaf level. First, we support two storage formats for leaf nodes: *sparse* and *dense*. A sparse-format node stores (key, data) pairs and can hold up to m such entries. By contrast, a dense-format node drops the keys and sequentially lays out only the data elements (thus zero-valued elements can no longer be omitted). A dense-format node can hold a maximum of αm entries, where $\alpha > 1$. If data insertion causes a sparse-format leaf node to overflow, and all keys in that node span a range smaller than αm, the node will switch to the dense format without splitting; the keys will be removed and the remaining values packed in order, improving space utilization and access performance. The opposite can happen if an originally dense node has fewer than m entries left due to deletions (or rather, zeroing out of elements), and an insertion expands the range of its keys to beyond αm. Second, we have designed new splitting algorithms to maintain high space and query efficiency, by exploiting unique properties of the address domain of array indices (e.g., it is finite and discrete). The details are beyond the scope of this paper.

E. Integration with External Databases

It is common to have programs to bring large amounts of data from a database system through SQL queries into arrays for further processing. The standard way of handling this situation is to execute the SQL queries and bring all the results into R. The cost of moving data can be very high. However, it may be more efficient to leave data in the external database and *push* additional computation specified in R down to the database, if such computation can be executed efficiently by the database and/or doing so reduces the volume of data to be copied into R. Conversely, it is also conceivable that we should perform only subqueries in the database and execute the remaining query operators in R. To this end, RIOT covers relational operators in its expression algebra, and it models the cost of executing queries in the external database. Hence, RIOT is able to decide what computation to push down to the database, blurring the boundary between database and host-language processing.

III. Demonstration Scenarios

The purpose of our demo is to show the transparency and I/O efficiency of RIOT. Our demo will be run on a Solaris virtual machine because of the specific Solaris instrumentation tools we use for performance tracking; RIOT itself is portable across operating systems.

The demo will be centered around a short, realistic piece of R code that performs a statistical analysis task and operates on a large dataset. The same code snippet (with little modification) will be run on three different systems: plain R, RIOT-DB, and RIOT. We expect the performance differences to establish a clear advantage of RIOT.

Installing and Loading RIOT Since an important goal of RIOT is to make it appealing to common R users, we will begin by demonstrating how easy it is to install RIOT into an existing R environment. To save time at the demo, we plan to install plain R and download the RIOT package on our demo machine in advance. At the demo, we show that a single command compiles and installs RIOT into the existing R installation, as with any other R package. We then start R and load the RIOT package using the R `library` function. Data types and methods supported by RIOT are now visible to user programs; the optimization engine and storage manager are ready to accept computational tasks.

Running the Code Snippet We next run our code snippet in plain R, RIOT-DB, and RIOT in turn. Recall that in the case of plain R, the large dataset the program references is brought into the virtual memory system and appears entirely in memory to the program, although swapping can happen due to limited physical memory. RIOT-DB manages the large dataset using a relational database backend and converts operations in the program into SQL queries. By contrast, RIOT works with a custom optimization engine and an efficient array storage manager.

In the demo, we will not attempt to run with very large arrays whose sizes are greater than the actual amount of physical memory on the demo machine, because the running time would be extremely long. Instead, we will simulate a limited-memory environment, by locking down a large portion of the physical memory using the appropriate system calls on Solaris.

To help the audience understand the differences among the compared systems, we show two kinds of information when running the code.

- *Logs*. We have implemented extensive logging in both RIOT and RIOT-DB. While the program is running in either system, logs will reveal what is happening under the hood. For example, for each operator or function evaluated in the program, RIOT-DB will print the corresponding query executed in the database. RIOT, on the other hand, will display the new operator node added to the expression DAG.
- *I/O statistics*. I/O volume is a direct measure of the efficiency of different systems. To measure the I/O volume, we utilize the DTrace facility on Solaris, which dynamically traces activities (e.g., function calls, I/Os) in both user programs and the operating system. We use DTrace to monitor different statistics for the three systems. For plain R, I/Os are caused by the swapping of data into and out of the physical memory. We thus monitor virtual memory paging statistics. For RIOT-DB, virtual memory paging activity is negligible assuming there is enough memory to run R and the database backend; most I/Os are caused by the database server reading and writing its data and index files. Therefore we monitor disk I/O statistics pertaining to database files. For RIOT, I/Os are mainly caused by reading and writing data files (assuming no external databases attached). The disk I/Os related to RIOT managed files are monitored.

Running RIOT as an Integrated System Finally, we demonstrate how RIOT deal with input data residing in an external database. The code snippet is modified so that some arrays are initialized by SQL queries over the external database. From the logs produced by RIOT, we should be able to see how RIOT builds an expression tree involving both database and R operations, and which operations are actually pushed down to the external database for execution. This approach will be compared with the alternative, naïve, approach of simply pulling data out from the database as specified. The I/O difference will show RIOT's flexibility and efficiency in dealing with external data sources.

References

[1] S. Toledo and F. G. Gustavson, "The design and implementation of solar, a portable library for scalable out-of-core linear algebra computations," in *Proceedings of the fourth Workshop on I/O in Parallel and Distributed Systems*, 1996, pp. 28–40.

[2] Y. Zhang, H. Herodotou, and J. Yang, "RIOT: I/O-Efficient Numerical Computing without SQL," in *CIDR*, 2009.

[3] S. Sarawagi and M. Stonebraker, "Efficient organization of large multidimensional arrays," in *ICDE*, 1994.

[4] M. Stonebraker, C. Bear, U. Çetintemel, M. Cherniack, T. Ge, N. Hachem, S. Harizopoulos, J. Lifter, J. Rogers, and S. B. Zdonik, "One size fits all? part 2: Benchmarking studies," in *CIDR*, 2007.

[5] R. Bayer, "The universal B-tree for multidimensional indexing: General concepts," *Lecture Notes in Computer Science*, vol. 1274, pp. 198–209, 1997.

[6] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki, "QPipe: A simultaneously pipelined relational query engine," in *SIGMOD*, 2005.