

Roomy: A System for Space Limited Computations

Daniel Kunkle
Northeastern University
360 Huntington Ave.
Boston, Massachusetts 02115
kunkle@ccs.neu.edu

ABSTRACT

There are numerous examples of problems in symbolic algebra in which the required storage grows far beyond the limitations even of the distributed RAM of a cluster. Often this limitation determines how large a problem one can solve in practice. Roomy provides a minimally invasive system to modify the code for such a computation, in order to use the local disks of a cluster or a SAN as a transparent extension of RAM.

Roomy is implemented as a C/C++ library. It provides some simple data structures (arrays, unordered lists, and hash tables). Some typical programming constructs that one might employ in Roomy are: map, reduce, duplicate elimination, chain reduction, pair reduction, and breadth-first search. All aspects of parallelism and remote I/O are hidden within the Roomy library.

Categories and Subject Descriptors: D.3.3 [Programming Languages]: Language Constructs and Features — *Dynamic storage management*; E.1 [Data Structures]: *Distributed data structures*

General Terms: Algorithms, Languages

Keywords: parallel, disk-based, programming model, open source library

1. INTRODUCTION

This paper provides a brief introduction to Roomy [1], a new programming model and open source library for parallel disk-based computation. The primary purpose of Roomy is to solve space limited problems without significantly increasing hardware costs or radically altering existing algorithms and data structures.

Roomy uses disks as the main working memory of a computation, instead of RAM. These disks can be disks attached to a single shared-memory system, a storage area network (SAN), or the locally attached disks of a compute cluster. Particularly in the case of using the local disks of a cluster, disks are often underutilized and can provide several order of

magnitude more working memory than RAM for essentially no extra cost.

There are two fundamental challenges in using disk-based storage as main memory:

Bandwidth: roughly, the bandwidth of a single disk is 50 times less than that of a single RAM subsystem (100 MB/s versus 5 GB/s). The solution is to use many disks in parallel, achieving an aggregate bandwidth comparable to RAM.

Latency: even worse than bandwidth, the latency of disk is many orders of magnitude worse than RAM. The solution is to avoid latency penalties by using streaming data access, instead of costly random access.

Roomy hides from the programmer both the complexity inherent in parallelism and the techniques needed to convert random access patterns into streaming access patterns. In doing so, the programming model presented to the user closely resembles that of traditional RAM-based serial computation.

Some previous work that provided a foundation for the development of Roomy include: the use of parallel disks to prove that 26 moves suffice to solve Rubik's Cube [2]; and disk-based methods for enumerating large implicit state spaces [4, 5]. The largest Roomy-based project to date is a package for manipulating large binary decision diagrams [3], which appears in the same proceedings as this paper, and includes an experimental analysis of the package.

The rest of this paper briefly describes the data structures provided by Roomy, some example programming constructs that can be implemented using Roomy, and some general performance considerations. Complete documentation, and instructions for obtaining the Roomy open source library, can be found on the Web at roomy.sourceforge.net.

2. ROOMY DATA STRUCTURES

Roomy data structures are transparently distributed across many disks, and the operations on these data structures are transparently parallelized across the many compute nodes of a cluster. Currently, there are three Roomy data structures:

RoomyArray: a fixed size, indexed array of elements (elements can be as small as one bit).

RoomyHashTable: a dynamically sized structure mapping *keys* to *values*.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PASCO 2010, 21–23 July 2010, Grenoble, France.

Copyright 2010 ACM 978-1-4503-0067-4/10/0007 ...\$10.00.

RoomyList: a dynamically sized, unordered list of elements.

There are two types of Roomy operations: delayed and immediate. If an operation requires random access, it is delayed. Otherwise, it is performed immediately. To initiate the processing of delayed operations for a given Roomy data structure, the programmer makes an explicit call to *synchronize* that data structure. By delaying random access operations they can be collected and performed more efficiently in batch.

Table 1 describes some of the basic Roomy operations. Some operations are specific to one type of Roomy data structure, while others apply to all three. The operations are also identified as either immediate (I) or delayed (D).

For performance reasons, it is often best to use a **RoomyArray** or **RoomyHashTable** instead of a **RoomyList**, where possible. Computations using **RoomyLists** are often dominated by the time to sort the list and any delayed operations. **RoomyArrays** and **RoomyHashTables** avoid sorting by organizing data into *buckets*, based on indices or keys.

3. PROGRAMMING CONSTRUCTS

Because Roomy provides data structures and operations similar to traditional programming models, many common programming constructs can be implemented in Roomy without significant modification. The one major difference is in the use of delayed random operations. To ensure efficient computation, it is important to maximize the number of delayed random operations issued before they are executed (by calling *sync* on the data structure).

Below are Roomy implementations of six programming constructs: map, reduce, set operations, chain reduction, pair reduction, and breadth-first search. Both map and reduce are primitive operations in Roomy. The others are built using Roomy primitives.

First, note that the code given here uses a simplified syntax. For example, the *doUpdate* method from the *chain reduction* programming construct below would be implemented in Roomy as:

```
void doUpdate(uint64 localIndex, void* localVal,
             void* remoteVal) {
    *(int*)localVal =
        *(int*)localVal + *(int*)remoteVal;
}
```

The simplified version given here eliminates the type casting, and appears as:

```
int doUpdate(int localIndex, int localVal,
            int remoteVal) {
    return localVal + remoteVal;
}
```

A future C++ version of Roomy is planned that would use templates to make the simplified version legal code.

See the online Roomy documentation and API [1] for the exact syntax and function definitions.

Map.

The *map* operator applies a user-defined function to every element of a Roomy data structure. As an example, the following converts a **RoomyArray** into a **RoomyHashTable**, with array indices as keys and the associated elements as values.

```
RoomyArray ra; // elements of type T
RoomyHashTable rht; // pairs of type (int, T)

// Function to map over RoomyArray ra.
void makePair(int i, T element) {
    RoomyHashTable_insert(rht, i, element);
}

// Perform map, then complete delayed inserts
RoomyArray_map(ra, makePair);
RoomyHashTable_sync(rht);
```

Reduce.

The *reduce* operator produces a result based on a combination of all elements in a data structure. It requires two user-defined functions. The first combines a partially computed result and an element of the list. The second combines two partially computed results. The order of reductions is not guaranteed. Hence, these functions must be associative and commutative, or else the result is undefined.

As an example, the following computes the sum of squares of the elements in a **RoomyList**.

```
RoomyList rl; // elements of type int

// Function to add square of an element to sum.
int mergeElt(int sum, int element) {
    return sum + element * element;
}

// Function to compute sum of two partial answers.
int mergeResults(int sum1, int sum2) {
    return sum1 + sum2;
}

int sum =
    RoomyList_reduce(rl, mergeElt, mergeResults);
```

The type of the result does not necessarily have to be the same as the type of the elements in the list, as it is in this case. For example, the result could be the *k* largest elements of the list.

Set Operations.

Roomy can support certain set operations through the use of a **RoomyList**. Some of these operations (particularly intersection) are sub-optimal when built using the current set of primitives. Future work is planned to add a native **RoomySet** data structure.

A **RoomyList** can be converted to a set by removing duplicates.

```
RoomyList A; // can contain duplicate elements
RoomyList_removeDups(A); // now a set
```

Performing set union, $A = A \cup B$, is also simple.

```
RoomyList A, B;
RoomyList_addAll(A, B);
RoomyList_removeDups(A);
```

Set difference, $A = A - B$, is performed by using just the *removeAll* operation, assuming *A* and *B* are already sets.

```
RoomyList A, B;
RoomyList_removeAll(A, B);
```

Finally, set intersection is implemented as a union, followed set differences: $C = (A + B) - (A - B) - (B - A)$. Set intersection may become a Roomy primitive in the future.

Table 1: Some basic Roomy operations. If an operation is specific to one type of data structure, it is listed under RoomyArray, RoomyHashTable, or RoomyList. Otherwise, it is listed as “common to all”. Also, the type of each operation is given as either immediate (I) or delayed (D).

Data Structure	Name	Type	Description
RoomyArray	access	D	apply a user-defined function to an element
	update	D	update an element using a user-defined function
RoomyHashTable	insert	D	insert a given (key, value) pair in the table
	remove	D	given a key, remove the corresponding (key, value) pair from the table
	access	D	given a key, apply a user-defined function to the corresponding value
	update	D	given a key, update a the corresponding value using a user-defined function
RoomyList	add	D	add a single element to the list
	remove	D	remove all occurrences of a single element from the list
	addAll	I	adds all elements from one list to another
	removeAll	I	removes all elements in one list from another
	removeDupes	I	removes duplicate elements from a list
Common to all	sync	I	process all outstanding delayed operations for the data structure
	size	I	returns the number of elements in the data structure
	map	I	applies a user-defined function to each element
	reduce	I	applies a user-defined function to each element and returns a value (e.g. the ten largest elements of the list)
	predicateCount	I	returns the number of elements that satisfy a given property (Note: this does not require a separate scan, the count is kept current as the data is modified)

```
// input sets
RoomyList A, B;
// initially empty sets
RoomyList AandB, AminusB, BminusA, C;

// create three temporary sets
RoomyList_addAll(AandB, A);
RoomyList_addAll(AandB, B);
RoomyList_removeDupes(AandB);
RoomyList_addAll(AminusB, A);
RoomyList_removeAll(AminusB, B);
RoomyList_addAll(BminusA, B);
RoomyList_removeAll(BminusA, A);

// compute intersection
RoomyList_addAll(C, AandB);
RoomyList_removeAll(C, AminusB);
RoomyList_removeAll(C, BminusA);
```

Chain Reduction.

Chain reduction combines each element in a sequence with the element after it. In this example, we compute the following function for an array of integers a of length N

```
for i = 1 to N-1
  a[i] = a[i] + a[i-1]
```

where all array elements on the right-hand side are accessed before updating any array elements on the left-hand side.

In the following code, `val_i` represents $a[i]$ and `val_iMinus1` represents $a[i-1]$.

```
RoomyArray ra; // array of ints, length N

// Function to complete updates
int doUpdate(int i, int val_i, int val_iMinus1) {
  return val_i + val_iMinus1;
}

// Function to be mapped over ra, issues updates
void callUpdate(int iMinus1, int val_iMinus1) {
  int i = iMinus1 + 1;
  if i < N
```

```
RoomyArray_update(
  ra, i, val_iMinus1, doUpdate);
}

RoomyArray_map(ra, callUpdate); // issue updates
RoomyArray_sync(ra);           // complete updates
```

The computation is deterministic. The new array values are based only on the old array values because Roomy guarantees that none of the delayed update operations are executed until `sync` is called. The code above is implemented internally through a traditional scatter-gather operation.

Parallel Prefix.

The chain reduction programming construct can also be used as the basis for a parallel prefix computation. At a high level, the parallel prefix computation is defined as

```
for (k = 1; k < N; k = k * 2)
  if i-k >= 0
    a[i] = a[i] + a[i-k];
```

Pair Reduction.

Pair reduction applies a function to each pair of elements in a collection. For an array a of length N , pair reduction is defined as

```
for i = 0 to N-1
  for j = 0 to N-1
    f(a[i], a[j]);
```

The following example inserts each pair of elements from a `RoomyArray` into a `RoomyList`. The variable `outerVal` represents $a[i]$ and the variable `innerVal` represents $a[j]$.

```
RoomyArray ra; // array of int, length N
RoomyList rl; // list containing Pair(int, int)

// Access function, adds a pair to the list
void doAccess(int innerIndex, int innerVal,
              int outerVal) {
```

```

RoomyList_add(
    rl, new Pair(innerVal, outerVal));
}

// Map function, sends access to all other elts
void callAccess(int outerIndex, int outerVal) {
    for innerIndex = 0 to N-1
        RoomyArray_access(
            ra, innerIndex, outerVal, doAccess);
}

RoomyArray_map(ra, callAccess);
RoomyArray_sync(ra); // perform delayed accesses
RoomyList_sync(rl); // perform delayed adds

```

One can think of the `RoomyArray_map` method as the outer loop, the `callAccess` method as the inner loop, and the `doAccess` method as the function being applied to each pair of elements.

Breadth-first Search.

Breadth-first search enumerates all of the elements of a graph, exploring elements closer to the starting point first. In this case, the graph is implicit, defined by a starting element and a generating function that returns the neighbors of a given element.

```

// Lists for all elts, current, and next level
RoomyList* all = RoomyList_make("allLev", eltSize);
RoomyList* cur = RoomyList_make("lev0", eltSize);
RoomyList* next = RoomyList_make("lev1", eltSize);

// Function to produce next level from current
void genNext(T elt) {
    /* User-defined code to compute neighbors ... */
    for nbr in neighbors
        RoomyList_add(next, nbr);
}

// Add start element
RoomyList_add(all, startElt);
RoomyList_add(cur, startElt);

// Generate levels until no new states are found
while(RoomyList_size(cur)) {
    // generate next level from current
    RoomyList_map(cur, genNext);
    RoomyList_sync(next);

    // detect duplicates within next level
    RoomyList_removeDuples(next);

    // detect duplicates from previous levels
    RoomyList_removeAll(next, all);

    // record new elements
    RoomyList_addAll(all, next);

    // rotate levels
    RoomyList_destroy(cur);
    cur = next;
    next = RoomyList_make(levName, eltSize);
}

```

One of the initial tests of Roomy was to use breadth-first search to solve the *pancake sorting problem*. Pancake sorting operates using a sequence of prefix reversals (reversing the order of the first k elements of the sequence). The sequence can be thought of as a stack of pancakes of varying sizes, with the prefix reversal corresponding to flipping the top k pancakes. The goal of the computation is to determine the number of reversals required to sort any sequence of length n .

Using Roomy, the entire application took less than one day of programming and less than 200 lines of code. Breadth-first search was implemented using a `RoomyArray`, similar to

the `RoomyList`-based version presented above. It was able to solve the 13-pancake problem in 70 minutes using the locally attached disks of a 30 node cluster.

Three different solutions to the pancake sorting problem, each using one of the three Roomy data structures, is available in the Roomy online documentation [1].

4. PERFORMANCE CONSIDERATIONS

Parallelism: It is anticipated that most applications will use one Roomy process per compute node. In some cases, however, disk bandwidth may not be fully utilized by a single process. In this case, each compute node can start several Roomy processes. Alternatively, the user application itself can be multi-threaded, as long as a single thread issues all Roomy operations. A future version of Roomy is planned that provides full multi-threading support.

Maximum data structure size: The maximum size of a Roomy data structure is limited only by the aggregate available disk space. The use of load balancing techniques ensures that each Roomy process stores approximately the same amount of data. In cases where compute nodes have significantly different amounts of free space, aggregate space can be increased by starting additional Roomy processes on those nodes with more space.

Choice of data structure: Roomy uses two primary methods for converting random access patterns into streaming access: buckets and sorting. The bucket-based method is used by `RoomyArrays` and `RoomyHashTables`. This method splits the data structure into RAM-sized *chunks*, and co-locates delayed operations with the corresponding chunk. The sorting method is used by `RoomyLists`, where there is no index or key that can be used to define buckets. In this case, both the list and the delayed operations are maintained in a sorted order. Because the cost of sorting often dominates the running time of `RoomyList`-based programs, it is recommended that a `RoomyArray` or `RoomyHashTable` be used instead, where possible.

5. ACKNOWLEDGMENTS

Many thanks to Gene Cooperman and Vlad Slavici, whose input has greatly improved both Roomy and this paper.

6. REFERENCES

- [1] Daniel Kunkle. Roomy: A C/C++ library for parallel disk-based computation, 2010. <http://roomy.sourceforge.net/>.
- [2] Daniel Kunkle and Gene Cooperman. Harnessing parallel disks to solve Rubik's cube. *Journal of Symbolic Computation*, 44(7):872–890, 2009.
- [3] Daniel Kunkle, Vlad Slavici, and Gene Cooperman. Parallel disk-based computation for large, monolithic binary decision diagrams. In *Parallel Symbolic Computation (PASCO '10)*. ACM Press, 2010.
- [4] Eric Robinson. *Large Implicit State Space Enumeration: Overcoming Memory and Disk Limitations*. PhD thesis, Northeastern University, Boston, MA, 2008.
- [5] Eric Robinson, Daniel Kunkle, and Gene Cooperman. A comparative analysis of parallel disk-based methods for enumerating implicit graphs. In *Parallel Symbolic Computation (PASCO '07)*, pages 78–87. ACM Press, 2007.