Roomy: A System for Space Limited Computations

Dan Kunkle

Ph.D. Student
College of Computer and Information Science
Northeastern University

Advisor: Gene Cooperman

PASCO '10: July 21, 2010

Outline

Outline

Problem Statement

Goal: solve **space limited problems** without significantly increasing hardware costs or radically altering algorithms and data structures.

A **space limited problem** is one where existing solutions quickly exceed available memory.

Solution: **Roomy**

- A new **programming model** that extends a programming language with transparent disk-based computing support.
- An open source **C/C++ library** implementing this new programming language extension.

Definition: Parallel Disk-based Computation

Parallel disk-based computation: using disks as the main working memory of a computation, instead of RAM. This provides several orders of magnitude more space for the same price.

Performance Issues and Solutions

- **Bandwidth:** the bandwidth of a disk is roughly 50 times less than RAM (100 MB/s versus 5 GB/s).
  Solution: use many disks in parallel.
- **Latency:** even worse, the latency of disk is many orders of magnitude worse than RAM.
  Solution: avoid latency penalties by using streaming access.

Other Approaches to Space-limited Problems

Other approaches to space limited problems include:

- **New algorithmic techniques** that reduce space usage (e.g., Bloom filters).
  Issue: usually problem specific; not always applicable
- **Increase RAM** using large shared-memory machines
  Issue: expensive (non-commodity hardware)
- **Distributed memory** clusters
  Issue: RAM per CPU is the same – still runs out of RAM quickly
- **Disks of a single machine**
  Issue: low bandwidth relative to RAM

Implications of Disk-based Computation

By replacing RAM with disks

- A cluster of 50 computers, each with 8 cores and 1 TB of disk space, can substitute for a shared memory computer with 400 cores and a single 50 TB memory subsystem.

Algorithm and Software Engineering Issues

- Unfortunately, writing programs that use many disks in parallel and avoid using random access is often a difficult task.
- Our group has over five years of case histories applying this to computational group theory – but each case requires months of development and debugging.
  - Rubik's Cube in 26 moves, 2007, 8 TB of aggregate storage.

Outline

Goals of Roomy

The primary goals of Roomy are:

- **Minimally invasive**: common data structures in user sequential code are replaced by Roomy data structures (lists, arrays, and hash tables).
- **Performance:** the interface biases programmers toward approaches with high performance parallel disk-based implementations.
- **Choice of architectures:** can used shared or distributed memory; locally attached disks or storage area networks (SAN).
- **Scalability:** the size of data structures is limited only by aggregate disk space; performance generally scales linearly with increasing parallelism.

Design of Roomy

Applications

A.I search (pancake sorting, Rubik's Cube)

SAT solver

Binary decision diagrams

Explicit state model checking

Algorithm Library

breadth-first search
parallel depth-first search
dynamic programming

API

RoomyList:
add, remove
addAll, removeAll
removeDupes
map, reduce

RoomyArray:
update, predicates
delayed read
map, reduce

Foundation

file management
remote I/O
external sorting
synchronization and barriers

Roomy Programming Model

The Roomy programming model:

- Provides **basic data structures** (arrays, unordered lists, and hash tables).
- Transparently **distributes data** structures across many disks and performs operations on that data in parallel.
- Immediately processes **streaming access operators**.
- Delays processing **random access operators** until they can be performed efficiently in batch (e.g., collecting and sorting updates to an array).

Example: Delayed Processing of Hash Table Insertions

Elements to insert in hash table

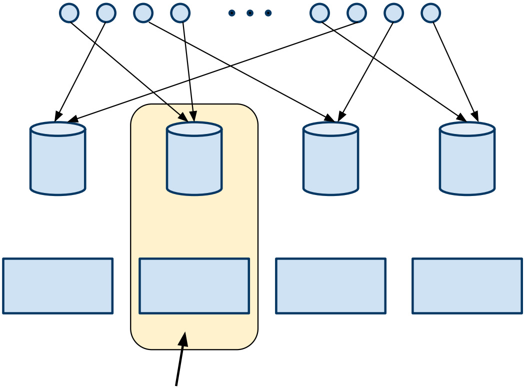
- Buffer to disk until many insertions are made

Store elements in buckets

- Stored on disk
- One bucket per hash table chunk

Roomy Hash Table

- Stored on disk
- Divided into RAM-sized chunks



Process each hash table chunk independently: load the chunk into RAM and perform element insertions.

Programming Interface

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*.
- **RoomyList**: a dynamically sized, unordered list of elements.

There are two types of Roomy operations: **delayed** and **immediate**.

- Operations requiring random access are delayed.
- Other operations are performed immediately.

Processing of delayed operations is initiated explicitly by the user, by making a call to **synchronize** a data structure.

RoomyArray Data Structure

RoomyArray Delayed Operations

- **access** – apply a user-defined function to an element
- **update** – update an element using a user-defined function

RoomyArray Immediate Operations

- **sync** – process outstanding delayed operations
- **size** – return the number of elements
- **map** – apply a user-defined function to each element
- **reduce** – return a value based on a combination of all elements
- **predicateCount** – return the number of elements that satisfy a property

RoomyHashTable Data Structure

RoomyHashTable Delayed Operations

- **insert** – insert a (key, value) pair in the table
- **remove** – remove a (key, value) pair from the table
- **access** – apply a user-defined function to a (key, value) pair
- **update** – update the value of a (key, value) pair

RoomyHashTable Immediate Operations (*gray = same as RoomyArray*)

- **sync** – process outstanding delayed operations
- **size** – return the number of elements
- **map** – apply a user-defined function to each element
- **reduce** – return a value based on a combination of all elements
- **predicateCount** – return the number of elements that satisfy a property

RoomyList Data Structure

RoomyList Delayed Operations

- **add** – add an element to the list
- **remove** – remove all occurrences of an element from the list

RoomyList Immediate Operations (*gray = same as RoomyArray*)

- **addAll** – add all elements from one list to another
- **removeAll** – remove all elements in one list from another
- **removeDupes** – remove duplicate elements from a list
- **sync** – process outstanding delayed operations
- **size** – return the number of elements
- **map** – apply a user-defined function to each element
- **reduce** – return a value based on a combination of all elements
- **predicateCount** – return the number of elements that satisfy a property

Outline

Example Programming Constructs

The use of data structures similar to traditional programming models allows many common **programming constructs** to be implemented in Roomy.

This section will give Roomy code for:

- map
- reduce
- predicates
- permutation multiplication
- set operations
- chain reduction
- pair reduction
- breadth-first search

Programming Construct: Map (complete code)

Map: apply a function to every element of a data structure

Example: add all elements in a RoomyArray to a RoomyList

```
RoomyArray* ra;
RoomyList* rl;

// Function to map over ra.
void mapFunc(uint64 i, void* val) {
  RoomyList_add(rl, val);
}

int main(int argc, char **argv) {
  Roomy_init(&argc, &argv);
  ra = RoomyArray_makeBytes("array", sizeof(uint64), 100);
  rl = RoomyList_make("list", sizeof(uint64));
  /* ... code that modifies ra ... */
  RoomyArray_map(ra, mapFunc);  // execute map
  RoomyList_sync(rl);  // sync rl to complete delayed 'add' ops
  Roomy_finalize();
}
```

Programming Construct: Map (abridged)

Map: apply a function to every element of a data structure

Example: add all elements in a RoomyArray to a RoomyList

```
RoomyArray* ra;
RoomyList* rl;

// Function to map over ra.
void mapFunc(uint64 i, void* val) {
  RoomyList_add(rl, val);
}

RoomyArray_map(ra, mapFunc);  // execute map
RoomyList_sync(rl);  // sync rl to complete delayed 'add' ops
```

Programming Construct: Reduce

Reduce: produce a result based on a combination of all elements in a data structure

Example: compute the sum of squares of the elements in a `RoomyList`

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

// Add square of an element to sum.
void mergeElt(int* sum, int* element) {
  *sum += *e * *e;
}

// Compute sum of two partial answers.
void mergeResults(int* sum1, int* sum2) {
  *sum1 += *sum2;
}

int sum = 0;
RoomyList_reduce(rl, &sum, sizeof(int), mergeElt, mergeResults);
```

Programming Construct: Predicates

Predicates: count the number of elements in a data structure that satisfy a Boolean function

Example: count the number of elements in a RoomyList greater than 42

```
RoomyList* rl;

// Predicate: return 1 if element is greater than 42
uint8 predFunc(int* val) {
  if ( *val > 42 )
    return 1;
  else
    return 0;
}

RoomyList_attachPredicate(rl, predFunc);
// ... code that modifies rl ...
uint64 gt42 = RoomyList_predicateCount(rl, predFunc);
```

Programming Construct: Permutation Multiplication

Permutation multiplication: arrays X, Y, Z of length N.

```
for i = 0 to N-1:  Z[i] = Y[X[i]]
```

```
RoomyArray *X, *Y, *Z;

// access X[i]
void accessX(uint64 i, uint64* x_i) {
    RoomyArray_access(Y, *x_i, &i, accessY);
}
// access Y[X[i]]
void accessY(uint64 x_i, uint64* y_x_i, uint64* i) {
    RoomyArray_update(Z, *i, y_x_i, setZ);
}
// set Z[i] = Y[X[i]]
void setZ(uint64 i, uint64* z_i, uint64* y_x_i, uint64* z_i_NEW) {
    *z_i_NEW = *y_x_i;
}

RoomyArray_map(X, accessX); // access X[i]
RoomyArray_sync(Y);         // access Y[X[i]]
RoomyArray_sync(Z);         // set Z[i] = Y[X[i]]
```

Programming Construct: Set Operations

Set operations: sets can be represented using a RoomyList

- A RoomySet data structure is planned for the future.

Convert list to set

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

Union: $A = A \cup B$

```
RoomyList *A, *B;
RoomyList_addAll(A, B);
RoomyList_removeDupes(A);
```

Difference: $A = A - B$

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

Programming Construct: Set Operations

Intersection: $C = A \cap B$

- Implemented as $C = (A \cup B) - (A - B) - (B - A)$

```
// 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);
```

Programming Construct: Chain Reduction

Chain reduction: combine each element in a sequence with the element after it

Example: given an array a of N integers,

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

where $a[i]$ on the right is the value *before* update

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

// Function to be mapped over ra, issues updates
void callUpdate(uint64 iMinus1, int* val_iMinus1) {
  uint64 i = iMinus1 + 1;
  if (i < N) RoomyArray_update(ra, i, val_iMinus1, doUpdate);
}

// Function to complete updates
void doUpdate(uint64 i, int* val_i, int* val_iMinus1,
              int* val_i_NEW) {
  *val_i_NEW = *val_i + *val_iMinus1;
}

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

Programming Construct: Pair Reduction

Pair reduction: apply a function to each pair of elements.

For an array a of length N:

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

Programming Construct: Pair Reduction

Example: insert each pair of elements from a RoomyArray in a RoomyList

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

// Map function, sends access to all other elts
void callAccess(uint64 outerIndex, int* outerVal) {
  for innerIndex = 0 to N-1
    RoomyArray_access(ra, innerIndex, outerVal, doAccess);
}

// Access function, adds a pair to the list
void doAccess(uint64 innerIndex, int* innerVal, int* outerVal) {
  RoomyList_add(rl, new Pair(*innerVal, *outerVal));
}

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

Programming Construct: Breadth-first Search

Breadth-first search: enumerate all elements of a graph, exploring elements closer to the starting point first.

- The graph is **implicit**, defined by a starting element and a generating function that returns the neighbors of a given element.

Initialize search

```
// 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);
```

Programming Construct: Breadth-first Search

Perform search

```
// 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_removeDupes(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);
}
```

Outline

KEY #1: Parallelism

Multi-process and Multi-threading:

- It is anticipated that most applications will use **one Roomy process** per compute node.
- If disk bandwidth is not fully utilized, and there is excess CPU power, one node can start *multiple Roomy processes*.
- Roomy is multi-threaded, but **user code is usually serial**.
- Currently, user code can be multi-threaded, but only if one thread makes all calls to Roomy.

KEY #2: Data Structure Size

Maximum data structure size is not limited by Roomy.

- The maximum size of a Roomy data structure is limited only by **aggregate available disk space**.
- Typically, each Roomy process stores about the same amount of data.
- If nodes have significantly different amounts of free space, multiple Roomy processes can be started on nodes with more space.

KEY #3: Choice of Roomy Data Structure

RoomyArray is often the most efficient data structure.

+ Minimizes data stored (elements can be as small as one bit)
+ Does not need hash function to determine element location
+ Does not use sorting
- Fixed size

RoomyHashTable is good when keys cannot be mapped to integers, or structure size is not predetermined.

+ Variable size
+ Arbitrary types for keys
+ Does not use sorting
- Empty *slots* take up additional space
- Hash function adds some CPU overhead

RoomyList should usually be chosen only if there is no alternative solution.

+ Variable size
+ No need for element indexes or keys
- Sorting causes a significant slowdown
- Hash function adds some CPU overhead

KEY #4: Synchronization Costs

Minimizing Synchronization Costs:

- The number of `sync` operations should be minimized.
- i.e., The number of outstanding delayed operations per `sync` should be maximized.
- Synchronizing cost is due to:
  - A small number of delayed operations causes random access.
  - A large number of delayed operations requires the entire data structure to be accessed.
  - All compute nodes must wait for all others to finish.

KEY #5: Load Balancing

Load Balancing:

- The even distribution of **data** is handled by Roomy.
  - RoomyArray element at index i is stored on node $i \bmod N$.
  - RoomyHashTable and RoomyList elements are distributed using a hash function.
- The **load** can become unbalanced if there are a small number of *hot* elements.
- Load balancing is important because all nodes must wait for the slowest node on a `sync`.
  - Watch for other causes of *slow nodes*, particularly certain hardware problems (e.g. a disk with high error rate).

KEY #6: Peak Disk Usage

Peak Disk Usage:

- Is one of the statistics printed by `Roomy_printStats`.
- All Roomy data is stored on disk.
  - data structures
  - delayed operations (includes an 8-byte index for RoomyArrays)
- Disk space can be freed by synchronizing delayed operations.

KEY #7: Peak RAM Usage

Peak RAM Usage:

- Typically, **buffers for delayed operations** are the bulk of RAM usage.
- To minimize RAM usage: minimize the number of Roomy data structures that have delayed operations outstanding at one time.
- A future version of Roomy is planned that uses free **RAM as a cache** for frequently used data.

KEY #8: Local vs. Shared Disks

Local Disks vs. Storage Area Network (SAN):

- Local disks
  - \+ processing of delayed operations does not use network
  - \+ typically higher performance
  - \- less reliable than an array of disks
- SAN
  - \+ may provide significantly more disk space
  - \+ more reliable (e.g., RAID)
  - \- possible lower performance: may be used by many other users; can cause a network bottleneck

KEY #9: Network Bottlenecks

Aggregate network bandwidth should be at least as large as aggregate disk bandwidth.

- All delayed operations are written to disk once and read from disk once.
- With local disk, delayed operations cross the network once.
- With a SAN, delayed operations cross the network twice.

KEY #10: Other High Latency Storage

Roomy is appropriate for any high-latency storage:

- Solid state drives (SSD), i.e. *flash storage*, provides much better random read performance than disk, but still has very bad random write performance
- Distributed RAM can also be high latency due to the network.
    - Roomy can be run with a RAM disk for distributed memory computations.

Outline

Pancake Sorting Problem

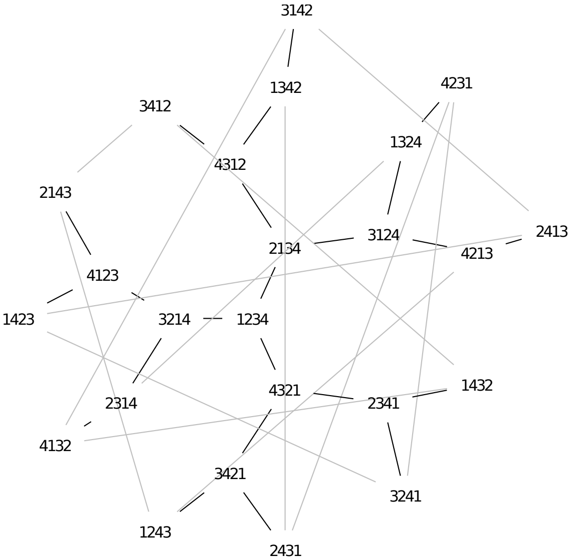
Pancake sorting: Sort using prefix reversal. Goal is to minimize the number of reversals used.

Example

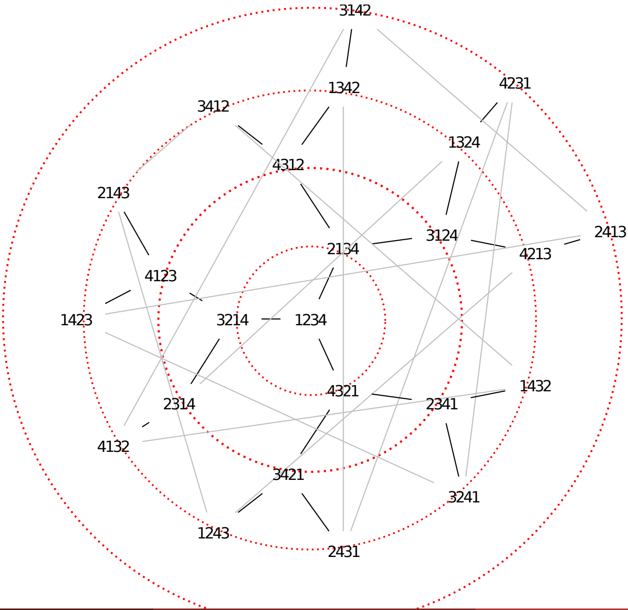
<u>31</u>42
<u>134</u>2
<u>4312</u>
<u>21</u>34
1234

Question: what is the maximum number of reversals needed to sort N elements?

Pancake Sorting Graph



Pancake Sorting Graph: Breadth-first Search Levels



Experimental Results for Pancake Sorting

Roomy was used for a breadth-first search of the 13-pancake graph.

- The graph has approximately **6.2 billion vertices** and **74 billion edges**.
- The computation completed in **11.5 minutes** using 64 compute nodes.
- Peak disk usage was **200 GB**.
- Average disk bandwidth was over **1.5 GB/s**.
- This replicated the best result as of 2006.
- Writing the Roomy program took less than a day.

Level	Number of Elements
0	1
1	12
2	132
3	1451
4	14556
5	130096
6	1030505
7	7046318
8	40309555
9	184992275
10	639768688
11	1525115582
12	2183056185
13	1458670200
14	186883243
15	2001

Distribution of Elements

Binary Decision Diagrams: Problem Statement

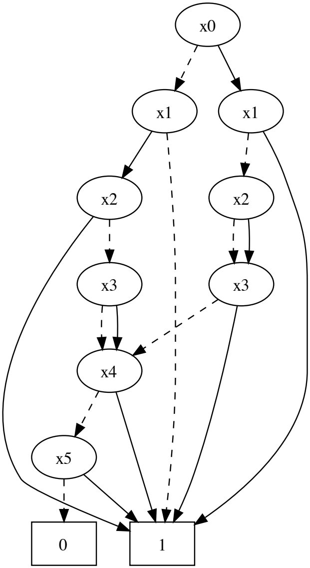
A **binary decision diagram (BDD)** is a compact representation of a Boolean function.

One of the primary practical uses of BDDs is in **symbolic model checking**, particularly **circuit verification**.

Problem: BDD packages typically **run out of space very quickly**.

- can fill RAM in a matter of minutes to hours.
- traditional approaches make heavy use of random access patterns

Example of a Binary Decision Diagram



BDD representing
$(x_0 \vee \neg x_1 \vee x_2 \vee x_4 \vee x_5) \wedge$
$(\neg x_0 \vee x_3 \vee x_1 \vee x_4 \vee x_5)$

Roomy-based BDD package implements three algorithms:

- **apply:** the application of a Boolean operator to two BDDs (and, or, xor, etc.)
- **any-SAT:** return a satisfying assignment
- **SAT-count:** count the number of satisfying assignments

Counting Solutions to the N-Queens Problem

Problem: determine the number of ways N non-attacking queens can be placed on an $N \times N$ chess board.

Size of State Space: $N!$

Boolean Representation

N^2 **variables:** $x_{i,j}$ is true iff there is a queen at row i, column j

N^2 **square constraints:** $S_{i,j}$ is true iff there is a non-attacked queen on i, j

- $S_{i,j} = x_{i,j} \wedge \neg x_{i_1,j_1} \wedge \neg x_{i_2,j_2} \wedge \ldots$

N **row constraints:** R_i is true iff row i has exactly one queen

- $R_i = S_{i,1} \vee S_{i,2} \vee \ldots \vee S_{i,N}$

board constraint: B is true iff the board has one queen in each row

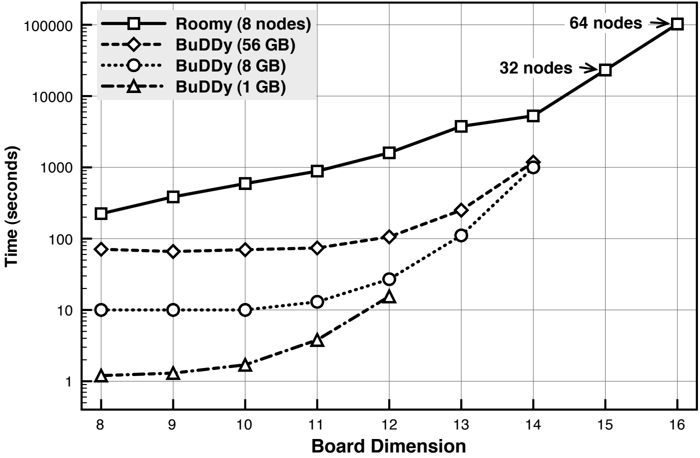
- $B = R_1 \wedge R_2 \wedge \ldots \wedge R_N$

Solution: count the number of satisfying assignments of B

N-Queens Results

Roomy-based package increased size of state space by 240 times over a RAM-based package (BuDDy) using 56 GB of RAM.

See PASCO '10: *Parallel Disk-Based Computation for Large, Monolithic Binary Decision Diagrams*, Kunkle, Slavici, Cooperman.



Outline

Conclusions

- Roomy is a new **programming model** and **open source library** for parallel disk-based computation.
- Roomy can provide **orders of magnitude more space** over RAM-based methods.
- The Roomy programming model extends sequential programs in a **minimally invasive** manner.

See: `roomy.sourceforge.net`

for a beta release of library and user documentation