Roomy: A New Approach to Parallel Disk-based Computation

Thesis Proposal
College of Computer and Information Science
Northeastern University

Daniel Kunkle

April 16, 2009

1 Introduction

Disk-based computation provides a major new application of disks in addition to the three traditional uses: file systems, databases, and virtual memory. Our working group has coined the term *disk-based computation* to describe our five year effort for using parallel disks in scientific computations. Disk-based computation makes use of the many disks already available in a computational cluster. In doing so, we elevate the disk to a level normally reserved for RAM. This provides an application with several orders of magnitude more working space for the same price. These parallel disk-based methods are often based on lower level external memory algorithms, such as those surveyed by Vitter [54].

There are two fundamental challenges in implementing disk-based storage as an extension of main memory:

1. **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). Our solution is to use many disks in parallel, achieving an aggregate bandwidth comparable to RAM.

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

Unfortunately, writing programs that use many disks in parallel and avoid using random access is often a difficult task. A fact that we encountered in our projects using disk-based computation [34, 35, 36, 47, 48].

To avoid the months-long process of writing and debugging such programs, we propose *Roomy*: a new programming language extension for writing parallel disk-based applications. The primary goal of Roomy is to provide programmers with a familiar and general programming model that allows for high performance in the underlying parallel disk-based computation.

The core of the proposed research is in developing a library of general algorithms and larger applications on top of the Roomy development platform. In doing so, the central questions we seek to answer are:

What is the class of applications for which parallel disk-based computing is practical?

How can existing sequential algorithms and software be adapted to take advantage of parallel disk-based computing?

In particular, we are interested in methods for adding latency avoidance and latency tolerance to applications that make heavy use of random access patterns. Roomy does not eliminate random access data structures and operations from the programming model, since this would eliminate a large class of existing algorithms and software from consideration. Instead, we separate the issuing of a random access operation from the execution of that operation. By increasing the time between these two events, we hope to collect a large number of such operations, and complete them efficiently in batch. Hence, much of the research question revolves around the tension between reduced algorithmic efficiency as delayed operations are introduced, and the benefits of access to a thousand times more storage.

Roomy is implemented as a library for C/C++ that provides programmers with a small number of simple data structures (arrays and lists) and associated operations. 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. All aspects of the parallelism and remote I/O are hidden within the Roomy library.

An initial version of Roomy has been developed and demonstrated on several small applications. For example, we recently used Roomy to perform a complete breadth-first search of the 13-pancake sorting problem, which was the largest such example before the 14 and 15 pancake versions were solved in 2008 [31]. Using Roomy, the entire application took only one day of programming, less than 200 lines of code, and solved the problem in 5 hours using the locally attached disks of a 30 node cluster.

1.1 Contributions of Research

The research problem is two-fold. First, an efficient implementation of a language will be developed to make the disk storage easily accessible to application writers. In this sense, there is an analogy with the Linda language [1]. Just as Linda makes available coordinated access to its tuple space, the proposed Roomy language will make available coordinated access to data streams from files across the network. Just as Linda required significant research to provide efficient performance [4, 8, 37], Roomy will require significant research to determine the right systems constructs to efficiently support its streams in a distributed fashion.

Second, this language will be used as a development platform for creating new parallel disk-based applications. Table 1 gives examples of applications that are currently space-limited, and would therefore make for good potential Roomy applications.

We propose to concentrate especially on algorithms for formal verification, denoted as applications 1, 2, and 3 in Table 1 (see Section 5.2 for more details). By concentrating on a single theme, there will be less time spent on learning the application-specific background needed to write competitive disk-based versions of well-known applications.

1.2 Design of Roomy

In order to support the applications in Table 1, Roomy must satisfy four goals:

1. Provide the most general programming model possible that still biases the applications programmer toward high performance for the underlying parallel disk-based computation.

	Discipline	Example Application
1.	Verification	Symbolic Computation using BDDs
2.	Verification	Explicit State Verification
3.	Verification	SAT Solvers (as used in Bounded Model Checking)
4.	Comp. Group Theory	Search and Enumeration in Mathematical Structures
5.	Coding Theory	Search for New Codes
6.	Security	Exhaustive Search for Passwords
7.	Semantic Web	RDF query language; OWL Web Ontology Language
8,	Artificial Intelligence	Planning
9.	Proteomics	Protein folding via a kinetic network model
10.	Operations Research	Branch-and-Bound
11.	Operations Research	Integer Programming (applic. of Branch-and-Bound)
12.	Economics	Dynamic Programming
13.	Numerical Analysis	ATLAS, PHiPAC, FFTW, and other adaptive software
14.	Engineering	Sensor Data
15.	A.I. Search	Rubik's Cube

Table 1: A selection of applications for which disk-based parallel computation can serve as an enabling technology

2. The use of full parallelism; providing not only the use of parallel disks (e.g., as in RAID), but also parallel processing.

3. Allow for a wide range of architectures, for example: a single shared-memory machine with one or more disks; a cluster of machines with locally attached disks; or a compute cluster with storage area network (SAN).

4. Provide fault tolerance; a necessity as computations are scaled up to run for many hours or days on systems with many hundreds or thousands of components.

The trade-offs among these design goals drive the design and implementation of Roomy. They also help in distinguishing Roomy from related work, covered in Section 2.

The overall design of Roomy has four layers: foundation; application programming interface; algorithm library; and applications. Figure 1 shows the relationship between each of these layers, along with examples of the components contained within each layer.

The first two layers are the core development platform for Roomy applications and provide the basic Roomy data structures and operations. We currently have an initial implementation of these two layers and have done some initial testing with small applications. The current components, and future improvements, are discussed in Section 4.

The core of the proposed research is in developing a library of general algorithms and larger applications on top of the Roomy development platform. Section 5 discusses some candidate algorithms and high-impact applications.

1.3 Supporting Hardware Trends

Current trends in the memory hierarchy are forcing us to drastically re-think how we write software that uses large working sets. Specifically, slower progress in RAM chip densities relative to the

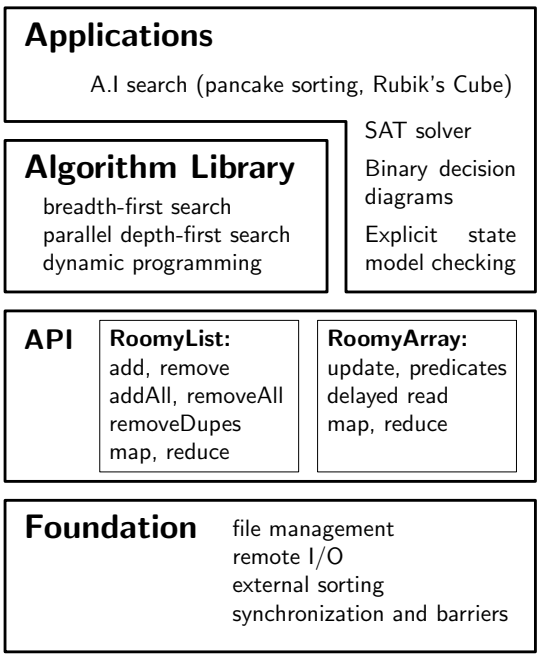

Figure 1: The layered design of Roomy.

number of cores per CPU chip is causing applications to become increasingly space-limited. In many cases, the increasing number of CPU-cores in future systems will quickly create quantities of data that exceed the capacity of RAM on commodity motherboards.

One possible solution to this problem is to use shared memory computers, which typically increase available RAM, thereby balancing the corresponding growth in CPU cores. Unfortunately, the use of such non-commodity hardware in this approach can significantly increase the cost of the system. Further, it will continue to suffer as CPU-performance growth exceeds RAM-density growth. Finally, it forces an end-user to pay for more cores even though the application is not CPU-limited.

Alternatively, distributed shared memory (DSM) systems can increase the total amount of available RAM using commodity hardware. However, each individual node in the DSM cluster increases not only the available RAM, but also the number of CPU cores, leaving the basic problem unsolved.

Compared to RAM, disks currently provide more than 100 times as much space per dollar. So, if we can successfully replace RAM with disks in space-limited applications, we can solve problems of much larger scale for the same system price. Further, in the near future, the growth in disk capacity will continue to exceed the growth in RAM capacity, increasing the advantage of this method.

1.4 Structure of Proposal

Section 2 describes related work. Section 3 gives an example usage of Roomy: breadth-first search. Section 4 describes the Roomy development platform. Section 5 examines the core research question of determining what class of applications is best suited for parallel disk-based computation, and proposes some promising candidate applications. Finally, Section 6 presents a schedule of

milestones, leading to the completion of the proposed dissertation work.

2 Related Work

Currently, there are two broad approaches to disk-based computing, as classified by their primary goal. They can be distinguished as:

- **Purpose-driven systems (large scale data processing):** These systems are primarily motivated by a practical need to processes very large data sets, such as processing data from the world wide web for web search systems. They focus primarily on issues of scalability, such as fault tolerance and efficient large scale parallelism.
- **Libraries of theoretically optimal algorithms:** These systems are primarily motivated by the development of external memory complexity models and algorithms. Such systems attempt to close the gap between the large number of theoretical external memory algorithms and the relatively small number of efficient implementations of those algorithms.

Like systems in the first category, Roomy is designed with a specific purpose in mind: to provide space-limited applications with significantly more working memory. The space limited applications of interest typically have a relatively small amount of input data, but quickly produce a large amount of intermediate data, before providing a final answer. In contrast, other systems typically focus on applications that process very large input data sets.

In many cases, it may be possible to use other systems for the applications that Roomy is targeting (and vice versa). However, the programming model of Roomy is significantly different from other systems, and makes it a more natural fit for applications that produce large intermediate data structures.

We now discuss the related work from each of the two classes of systems above. We then briefly cover some previous work that solved specific problems using disk-based methods, and finish with a note on virtual memory.

Large scale data processing systems. Several recent disk-based computing systems have been developed with scalability as the primary design objective. Google's MapReduce [13] was one of the first such systems, designed to scale to thousands of compute nodes and petabytes of data. To facilitate performance and fault tolerance, the MapReduce programming model is restricted to just two functions: *map* and *reduce* (as implied by the name). Other Google projects related to MapReduce include: the underlying Google File System (GFS) [21]; BigTable [10], a management system for structured data; and Sawzall [44], an interpreted programming language for MapReduce-style computations.

Hadoop [19] is an open source implementation of the MapReduce framework. The three related Hadoop systems corresponding to the above Google systems are: the Hadoop Distributed File System (HDFS); HBase, for structured data; and Pig, a high level data flow language (see [19] for links to each of these projects).

Dryad [28] is similar to MapReduce/Hadoop, but with a more general programming model. Dryad programs are structured as a directed acyclic graph, where the vertices are user defined serial computations, and the edges are data channels. To facilitate the use of this non-standard programming model, additional tools are layered on top of Dryad, such as: DryadLINQ [56], which provides an SQL-like query language; and SCOPE [9], similar to the Sawzall and Pig programming languages mentioned above.

External memory complexity models and algorithms. Some of the techniques used in Roomy are based on existing external memory algorithms. One key example is the use of external sorting, which has been extensively studied. A survey of such external memory algorithms and complexity models is given in [54].

The most applicable complexity model in this case is the Parallel Disk Model (PDM) [55]. PDM quantifies complexity as the number of external memory I/O operations, and is parametrized to handle multiple disks and multiple CPUs.

One of the first software packages to provide efficient external memory algorithms was TPIE (Transparent Parallel I/O Environment) [53, 3]. A more recent example, which is still under active development, is STXXL (an external memory implementation of the C++ standard template library) [14, 15]. Though both of these systems are based on the PDM model, only STXXL allows for the use of multiple disks, and neither allows for multiple CPUs.

Search and enumeration problems. A number of recent parallel disk-based solutions have been proposed for specific problems. Many such applications have been based on breadth-first search, and related enumeration techniques. These include previous projects in our group that have inspired the development of Roomy: a proof that 26 moves suffice for Rubik's Cube [34, 36]; computing permutation representations for very large matrix groups [47]; and direct condensation of large permutation modules [48]. Other applications include finding optimal solutions for sliding tile puzzles [30, 33, 57], Towers of Hanoi [32], and the pancake sorting problem [31].

We present a survey and analysis of the various techniques used to solve these problems in [49]. A more in-depth analysis, as well as a general software package for implicit state space enumeration, are described in [46]. An implementation of breadth-first search is also a part of Roomy's standard algorithm library (see Section 5.1).

Virtual memory. At a high level, virtual memory systems were one of the first advances that attempted to use high latency media as an extension of main memory [16].

However, if the working set of any processes exceeds available RAM, *thrashing* can occur, causing excessive paging and a large decrease in performance. Virtual memory also does not inherently consider parallelism. These limitations are motivation for Roomy and the related work described above.

3 Roomy Example: Breadth-First Search

One of the applications that inspired the development of Roomy was our proof that 26 moves suffice to solve any configuration of Rubik's Cube [34]. One primary component of that proof was a breadth-first search of a graph with approximately 1.5×10^{12} vertices and 2.7×10^{13} edges. That computation completed in 65 hours using 7 TB of disk space and sixteen compute nodes with eight processors each (128 CPUs).

Because of this inspiration, some of the first applications we have tested in Roomy are based on breadth-first search. Figure 2 gives a Roomy implementation of breadth-first search.

This implementation uses three `RoomyList` data structures to store: the current frontier of the search (i.e., the *current level*); the neighbors of the current frontier (i.e., the *next level*); and a list of all previously seen elements, which is used for duplicate detection.

All of the Roomy data structures and methods begin with the word `Roomy`, and are shown in bold in Figure 2. This is a general implementation of breadth-first search, and can be applied to

```
// Function to be mapped over cur level to produce next level
void genNextLev(void* val) {
    /*
     *  User defined code to generate nbrs array inserted here.
     */
    for(int i=0; i<numNbrs; i++) {
        RoomyList_add(nextLevList, nbrs[i]);
    }
}

// Init lists for duplicates, current level, and next level
RoomyList* allLevList = RoomyList_make("allLev", eltSize);
RoomyList* curLevList = RoomyList_make("lev0", eltSize);
RoomyList* nextLevList = RoomyList_make("lev1", eltSize);

// Add start element
RoomyList_add(allLevList, startElt);
RoomyList_add(curLevList, startElt);

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

    // detect duplicates
    RoomyList_removeDupes(nextLevList);
    RoomyList_removeAll(nextLevList, allLevList);
    RoomyList_addAll(allLevList, nextLevList);

    // rotate levels
    RoomyList_destroy(curLevList);
    curLevList = nextLevList;
    nextLevList = RoomyList_make(levName, eltSize);
}
```

Figure 2: A Roomy implementation of breadth-first search. User defined values are underlined; Roomy data structures and functions are bold.

any search space. The user defined portions of the implementation are italicized. These include: the start element, from which the search begins; an element representation, and its associated size; and a function to generate the neighbors of a given element.

Each Roomy data structure is automatically distributed across all disks, and the Roomy methods are automatically executed in parallel. Further, all of the Roomy operations utilize streaming-only access, avoiding any random access to Roomy data structures. Any operations that would normally require random access are delayed until many such operations are collected. These collected operations can then be performed efficiently, usually by sorting the updates and scanning the associated data structure.

4 Development Platform

This section discuss the bottom two layers of the Roomy design: the foundation, and the application programming interface (see Figure 1). Together, they provide a development platform for parallel disk-based computations, and facilitate our core research goals.

We discuss each of the two layers in turn, including an overview of the current implementation and proposed future improvements.

4.1 Foundation

4.1.1 Foundation Components: Currently Implemented

File management. All Roomy data structures are stored as a set of files distributed across disk-based storage. Additionally, all delayed Roomy operations are buffered to disk for later batch processing. This component handles the naming and sizing of files for these data structures and delayed operations.

Remote I/O. Each element of a Roomy data structure, and the associated operations performed on it, are "owned" by a single node of the compute cluster. If the data is distributed across the locally attached disks of the cluster, each node must be able to write data to a remote node for later processing. This is done using MPI, and each node has a thread dedicated to handling the remote I/O messages. In cases where a global file system is used (e.g. a SAN), each node can directly write this data to the shared storage and avoid using additional remote I/O messages.

Synchronization and barriers. Roomy uses barriers to synchronize parallel operations. These barriers are implemented using MPI messages, where one of the Roomy processes acts as a coordinator. Roomy also uses special *lock files* to ensure that any local or remote I/O has completed before completing the barrier. Roomy is designed to minimize the the use of these barrier, and most operations can be performed without any coordination between processes.

External sorting. One of the most important operations in Roomy (and in disk-based computations in general) is external sorting [29]. Because random access is so prohibitively expensive when using high-latency storage, it is usually replaced by a sequence of *delay*, *sort*, and *scan* operations. Using this method, the random operations are buffered until a large number of them are ready to be processed in batch. They are then sorted according to the order of the elements of the data structure they are accessing (e.g. the index of the elements of an array). Finally, the sorted operations and the data structure are scanned sequentially to complete the process.

In-RAM data structures. Roomy uses a number of custom data structures for managing in-RAM data. The following lists provides some examples, and the primary functions they serve.

- Arrays: bit packing for elements less than 1 byte in size; index out of bounds error conditions; memory allocation.
- Hash Tables: a basic chaining hash table; hash functions.
- Buffers: read/write buffers for iterating over disk-based data.

Memory management. This component adjusts the sizes of in-RAM Roomy data structures to ensure that their aggregate size, along with any RAM being consumed by user code, does not exceed the maximum allowed. Additionally, it must provide enough RAM for read/write buffers to guarantee large streaming I/O operations, and enough buffer space to make external sorting efficient.

4.1.2 Foundation Components: To be Implemented

High-performance sorting. External sorting is one of the workhorses of the Roomy algorithmic library. Currently, Roomy uses a custom implementation of external sort: a form of external merge sort, using quicksort for the in-RAM portion of the algorithm. This is a standard approach, and the implementation has not been significantly optimized. Because the time for external sorting is likely to dominate many Roomy-based applications, we plan to leverage the significant existing work in this area in future versions of Roomy.

Specifically, we will evaluate existing high-performance sorting systems and integrate that work with Roomy. Along with the related work discussed in Section 2, one good source of such systems are the top entries to the yearly sorting competition hosted at [42] (with awards being presented yearly at the ACM SIGMOD conference). Special attention will be paid to sorting systems that can utilize multiple compute cores and multiple disks per node, an architecture that we believe will be common for Roomy-based applications.

RAM as a cache. Currently, all Roomy data structures are stored on disk. In general, random access to these data structures is prohibited, and RAM is used primarily as a buffer for the I/O of large contiguous portions of that data. However, it may possible to accommodate some random access patterns that have sufficient spacial and/or temporal locality. In this case, RAM can be used as a cache for data that is stored elsewhere on disk. The techniques to be used in Roomy will likely be very similar to the page cache management policies in modern operating systems.

It remains to be determined if such a cache can be used effectively if implemented only as an addition to the internal workings of Roomy, or if it is better to expose new functionality in the Roomy API that makes use of this facility.

Lazy allocation. Currently, all Roomy array data structures are fully initialized at the time of creation. That is, files containing all zeroes are written to disk. Additionally, subsequent operations that scan that array can require reading a significant amount of uninitialized data. So, algorithms that perform many such operations on sparsely initialized arrays will pay a significant performance penalty. A lazy allocation strategy will split the array into *chunks* and only store chunks that contain non-zero elements. So, sparsely populated arrays will require less space, and operations on them will be faster.

Dynamic compression. Many disk-based computations are dominated by the time spent on disk I/O. In these cases, compressing disk based data may provide a significant speedup. However, some disk-based computations may still be CPU-bound, in which case compression can cause an overall drop in performance. For compression to be used effectively in Roomy, we will need to dynamically determine whether the computation is CPU-bound or I/O-bound, and use compression only in the latter case. We may also make additions to the API to allow programmers to use compression in specific cases (e.g., to reduce the final size of resulting data structures).

Checkpoint / restart. Many disk-based computations are long running and use computer architectures with many independent components (e.g. nodes in a compute cluster with associated processing, networking, and storage components). In this case, it becomes more likely that some component will fail before the computation finishes.

We propose to address this fault tolerance issue in Roomy at two levels. First, the state of the running computation will be checkpointed and restarted with DMTCP [2], a transparent user-level checkpointing package for distributed applications. This allows any RAM-based computation to restart after a failure by periodically saving the state of the program to disk.

However, this does not protect against the failure of a disk, which is used as primary storage in Roomy. For this, another layer of fault tolerance focused on disk failure is needed. If the compute cluster is using a SAN for storage, the problem of fault tolerance is likely to be handled transparently. If the storage is locally attached, one can use several disks in a RAID configuration at each compute node. Alternatively, Roomy could use a replication strategy, storing all data on multiple nodes. The method used will likely be determined by the hardware available, and so Roomy should be designed to handle each of these cases.

Improved overlap of computation and communication. Currently, Roomy uses asynchronous versions of file I/O and MPI commands in an attempt to overlap those communication routines with computation. However, because these routines are typically implemented using additional operating system defined buffers, they may cause significant overhead. By implementing such asynchronous routines directly in Roomy (using double buffering schemes), this overhead can be reduced.

4.2 Application Programming Interface

A full API for Roomy will be available for public release in the near future. Here, we present a brief overview of the main elements of Roomy.

There are two basic Roomy data structures: arrays and lists. And, there are two basic types of Roomy operations: delayed and immediate. Basically, if an operation requires random access, it is delayed. Otherwise, it is performed immediately.

The following will describe each of these two data structures, and give examples of delayed and immediate operations for each.

We also briefly mention planned additions to the API for gathering and reporting performance statistics.

4.2.1 Roomy Lists

A `RoomyList` is a variable sized container for user defined elements of a given size. Those elements can be built in data types (such as integers), or user defined structures. The only information Roomy needs about these elements is their size. A `RoomyList` can be of any size, up to the limits of

available disk space. For example, the `RoomyList`s in the breadth-first search example in Figure 2 contain elements representing the search space and its current frontier.

`RoomyList` delayed operations.

`add`: add a single element to the list

`remove`: remove all occurrences of a single element from the list

The elements of a `RoomyList` are maintained in sorted order to facilitate other operations based on streaming access. Pending `add`s and `remove`s are merged into the existing list when enough such operations have been buffered to ensure efficient streaming access, or when another operation on the list requires the updates to be synchronized.

`RoomyList` immediate operations.

`size`: returns the number of elements in a list

`addAll`: adds all elements from one list to another

`removeAll`: removes all elements in one list from another

`removeDupes`: removes duplicate elements from a list

`map`: applies a user defined function to each element of a list

`mapAndModify`: applies a user defined function to each element of a list, and optionally modifies the value of each element

`reduce`: applies a user defined function to each element of a list and returns a value (e.g. the ten largest elements of the list)

4.2.2 Roomy Arrays

A `RoomyArray` is a fixed size, indexed sequence of elements of a given size. An element of a `RoomyArray` can be any number of bytes. There are two variants this array that store less than one byte per element: `RoomyBitArray`, which stores one bit per element; and `RoomyBitsArray`, which stores between two and seven bits per element.

`RoomyArray` delayed operations.

`access`: apply a user defined function to the element at a given index

`update`: update the element at a given index using a user defined function

`RoomyArray` immediate operations.

`map`: applies a user defined function to each element of an array

`mapAndModify`: applies a user defined function to each element of an array, and optionally modifies the value of each element

`reduce`: applies a user defined function to each element of an array and returns a value (e.g. the ten largest elements in the array)

The operations for `RoomyArray` and `RoomyBitsArray` are the same, the difference is the representation of the data (e.g., bit packing is done in the case of elements with less than one byte).

There a few differences and additions for `RoomyBitArray` operations: it has `set` and `clear` operations, instead of `update`; and it has bitwise operations, such as `not`, `and`, and `or`.

4.2.3 Statistics Gathering and Reporting

Parallel disk-based computations may have many possible performance bottlenecks, including: CPU, RAM, network, and disks. Which possible bottleneck is the limiting factor for a given computation depends on the given implementation and computing architecture, and may be difficult to determine analytically.

To support this kind of bottleneck analysis, Roomy will provide facilities for gathering and reporting a range performance statistics. This portion of the API does not exist yet, and is part of the proposed future work.

5 Core Research

Again, the two core research questions we seek to answer are:

> *What is the class of applications for which parallel disk-based computing is practical?*
>
> *How can existing sequential algorithms and software be adapted to take advantage of parallel disk-based computing?*

We will answer these questions by developing new parallel disk-based versions of a number of algorithms and applications.

We first mention some of the basic algorithms that will make up a standard library of techniques. We then examine some larger, high-impact applications in the area of formal methods.

5.1 Standard Algorithm Library

The primary criterion for choosing to use Roomy is that the application often fails due to lack of primary storage (i.e., lack of RAM).

One such class of applications is those based on breadth-first search. Our laboratory has already shown in a series of papers [11, 34, 36, 47, 48] how to satisfy converting data access patterns from breadth-first search applications into a purely streaming access pattern. Breadth-first search is relatively easy because one can write a new search frontier to disk using a streaming access pattern. However, this still leaves the problem of detecting duplicates. Traditionally, a hash array would be used to determine if a state of the frontier has been seen before.

In our context, *delayed duplicate detection* (DDD) provides a solution that uses streaming access to disk. There are several well understood streaming techniques for delayed duplicate detection, such as: (a) externally sort the frontier in hash array order and compare with the hash array (Sorting-based DDD [49, Section 3.3]); and (b) use a second hash function to direct frontier data into smaller files that can be fully loaded into RAM (Hash-based DDD [49, Section 3.4]).

Figure 2 is an example of Roomy-based breadth-first search. A survey of still more examples can be found in [49], where an abbreviated list of earlier papers by other authors in this area can be found.

Several of these methods for breadth-first search are now included as part of the standard algorithm library for Roomy. As we identify other algorithms that are common in other applications, they will be added to the library. Possible examples include: parallel depth-first search; best-first search; topological sort; and dynamic programming.

5.2 High-Impact Applications

We will choose at least one of the following high-impact applications to implement using the Roomy development platform. These applications will be considered successful or unsuccessful only partially based on their speed. Although we will execute on a computer cluster, our goal is not a speedup over sequential applications. Rather, it is a greatly increased storage ceiling while not incurring too much of a storage slowdown.

If an important application runs for an hour on one computer and then fails due to lack of RAM, then it is perfectly acceptable for the application to run for two hours on a computer cluster (a *slowdown* of two) if it completes with an answer at the end of those two hours.

We present four possible applications related to formal methods, SPIN, Büchi automata, BDDs, and SAT solvers. These four tools provide a range of strengths in formal verification, including: static deadlock analysis and software verification; static livelock analysis; and hardware verification.

The proposed thesis work will including evaluating these possible applications, identifying which are most likely to provide a large benefit when combined with Roomy, and developing one or more Roomy-based tools for formal verification.

SPIN, explicit state model checking. SPIN [50, 27] is a well-known, open-source explicit-state model checker with about fifteen years of history. It has achieved many successes in formal verification of real world software and applications. As algorithms and computational power increase, explicit state model checking is having an ever increasing impact, for example, in such problems as software verification. SPIN generates new states at a rate of approximately 100,000 per second. Although it has a very efficient encoding of the states (typically two bits per state), it nevertheless frequently runs out of storage in a matter of hours. This application is perhaps the most challenging of all, due to risk of a parallel, disk-based implementation disrupting the finely tuned heuristics embedded in the software. Perhaps because of this, a competitive disk-based implementation on a single computer does not even exist. Nevertheless, recent successes by others in a multi-core implementation of SPIN [26, 25] indicate that this area is at least worth investigation.

Büchi automata. The use of Büchi automata for formal verification was opened up by the landmark papers by Vardi and Wolper [52, 12, 20]. Just as SPIN excels at finding such bugs as deadlock, among others, the algorithmic use of Büchi automata provide a state-based model checking mechanism for finding livelock bugs or infinite loops in software programs. The fundamental algorithm involves the use of two depth-first searches, along with a hash array representing each state in a very small number of bits. The latter feature is similar to that of SPIN. The combination of that property with depth-first search has some algorithmic similarities to BDDs (to be discussed next). The use of Büchi automata is an area that has not been explored to the same extent as the SPIN-style explicit state model checking. Hence, there has also been relatively little work on disk-based or parallel implementations. This can make this an "easier" application, in the sense that a first

parallel disk-based can be compared against existing implementations that are not based to the same extent on finely tuned heuristics.

BDD. BDDs (*binary decision diagrams*) are a method of symbolic model checking that has been demonstrated to be very successful for hardware verification. A BDD can fill 4 GB of RAM in minutes to hours. Verification engineers tend to use BDDs until main memory is exhausted, after which, they must consider other options. The fundamental BDD algorithm is a depth-first search algorithm. Information from current nodes are used to prune future branches. The algorithms for manipulating BDDs are very irregular and extremely challenging to parallelize and distribute over clusters because they involve the use of shared hash tables (for memoization) and because the basic algorithms consist of mostly pointer chasing, making them difficult to distribute.

Standard methods for implementing BDDs as parallel depth-first search algorithms have been discussed since 1987 [45]. Variations of them have been used in [24, 22, 23, 39, 51, 43, 7, 41] to parallelize BDD. There has also been a notable use of disk on a single computer by Minato [40]. No previous work simultaneously uses both parallel disks and the parallel nodes of a computer cluster, as proposed here. Yet, without the use of parallel disks, the speed of a single disk is often too slow. Without the use of a parallel algorithm, it is difficult for sequential code to efficiently use parallel disks.

SAT solver. SAT solvers have a wide range of practical applications, including: checking the functional equivalence of two circuits; automatic test pattern generation; planning in artificial intelligence; and model checking [38]. Because of our focus on formal verification, we are specifically interested in the use of SAT solvers in *bounded model checking*, a complementary approach to using BDDs for symbolic model checking [5].

Some recent work has shown disk-based computation to be effective in certain SAT preconditioning algorithms [6]. These methods seek to reduce the number of clauses a SAT solver must consider, allowing more problems to be solved in RAM. However, these methods do not necessarily scale to large parallel computations, and the SAT solver itself is still restricted to using only RAM. We believe that developing a full parallel disk-based SAT solver based on Roomy can provide significant additional advantages, and we are currently working toward integrating Roomy with the state-of-the-art SAT solver MiniSat [18].

6 Schedule of Milestones

The following gives anticipated completion dates for a series of milestones in the proposed thesis work.

May, 2009: Thesis proposal presentation.

August / September, 2009: Complete feature set; all additions to Roomy API completed (see Section 4.2), and additional standard algorithms are implemented (see Section 5.1).

October, 2009: Lower level optimizations of Roomy completed (see Section 4.1.2).

November / December, 2009: A major application from formal verification implemented on top of Roomy (see Section 5.2).

December, 2009 / January, 2010: A suite of additional applications built on top of Roomy (current efforts include coset enumeration, and formal verification using Murphi [17]).

March, 2010: First version of thesis written.

April / May, 2010: Second version of thesis written.

June, 2010: Thesis defense.

References

[1] Sudhir Ahuja, Nicholas Carriero, and David Gelernter. Linda and friends. *Computer*, 19(8):26–34, 1986.

[2] Jason Ansel, Kapil Arya, and Gene Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *Proc. of IEEE International Parallel and Distributed Processing Symposium (IPDPS '09)*, 2009, to appear.

[3] Lars Arge, Octavian Procopiuc, and Jeffrey Scott Vitter. Implementing i/o-efficient data structures using tpie. In *In Proc. European Symposium on Algorithms*, pages 88–100. Springer, 2002.

[4] David Edward Bakken and Richard D. Schlichting. Supporting fault-tolerant parallel programming in linda. *IEEE Trans. Parallel Distrib. Syst.*, 6(3):287–302, 1995.

[5] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, 2003.

[6] Fernando Brizzolari, Igor Melatti, Enrico Tronci, and Giuseppe Della Penna. Disk based software verification via bounded model checking. In *APSEC '07: Proceedings of the 14th Asia-Pacific Software Engineering Conference*, pages 358–365, Washington, DC, USA, 2007. IEEE Computer Society.

[7] Gianpiero Cabodi, Paolo Camurati, and Stefano Quer. Improved reachability analysis of large finite state machines. In *ICCAD*, pages 354–360, 1996.

[8] Nicholas Carriero and David Gelernter. The S/Net's Linda kernel. *ACM Trans. Comput. Syst.*, 4(2):110–129, 1986.

[9] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, 2008.

[10] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 205–218, Berkeley, CA, USA, 2006. USENIX Association.

[11] G. Cooperman and E. Robinson. Memory-based and disk-based algorithms for very high degree permutation groups. In *Proc. of International Symposium on Symbolic and Algebraic Computation (ISSAC '03)*, pages 66–73. ACM Press, 2003.

[12] C. Courcoubetis, Vardi M. Y, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2/3):275–288, 1992.

[13] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. pages 137–150.

[14] R. Dementiev and L. Kettner. Stxxl: Standard template library for xxl data sets. In *In: Proc. of ESA 2005. Volume 3669 of LNCS*, pages 640–651. Springer, 2005.

[15] R. Dementiev, L. Kettner, and P. Sanders. Stxxl: standard template library for xxl data sets. *Softw. Pract. Exper.*, 38(6):589–637, 2008.

[16] Peter J. Denning. Virtual memory. *ACM Comput. Surv.*, 2(3):153–189, 1970.

[17] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *ICCD '92: Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors*, pages 522–525, Washington, DC, USA, 1992. IEEE Computer Society.

[18] Niklas Eén and Niklas Sörensson. An extensible sat-solver. pages 502–518. 2004.

[19] The Apache Software Foundation. Hadoop, 2009.

[20] R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Fifteenth International Symposium on Protocol Specification, Testing and Verification (PSTV'95)*, volume 38 of *IFIP Conference Proceedings*, pages 3–18. Chapman & Hall, 1996.

[21] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[22] Orna Grumberg, Tamir Heyman, Nili Ifergan, and Assaf Schuster. Achieving speedups in distributed symbolic reachability analysis through asynchronous computation. In Dominique Borrione and Wolfgang J. Paul, editors, *CHARME*, volume 3725 of *Lecture Notes in Computer Science*, pages 129–145. Springer, 2005.

[23] Orna Grumberg, Tamir Heyman, and Assaf Schuster. A work-efficient distributed algorithm for reachability analysis. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 54–66. Springer, 2003.

[24] Orna Grumberg, Tamir Heyman, and Assaf Schuster. A work-efficient distributed algorithm for reachability analysis. *Formal Methods in System Design*, 29(2):157–175, 2006.

[25] Gerard J. Holzmann and Dragan Bosnacki. The design of a multi-core extension to the spin model checker. *Software Engineering, IEEE Transactions*, 33:659–674, October 2007.

[26] Gerard J. Holzmann and Dragan Bosnacki. Multi-core model checking with spin. In *21th International Parallel and Distributed Processing Symposium (IPDPS 2007)*, pages 1–8, 2007.

[27] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. New challenges in model checking. In *25 Years of Model Checking — History, Achievements, Perspectives*, volume 5000 of *Lecture Notes in Computer Science*, pages 65–76. Springer, 2008.

[28] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, New York, NY, USA, 2007. ACM.

[29] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998. Section 5.4: External Sorting, pp. 248–379.

[30] R. E. Korf. Best-first frontier search with delayed duplicate detection. In *AAAI*, pages 650–657, 2004.

[31] Richard E. Korf. Minimizing disk I/O in two-bit breadth-first search. In *AAAI*, pages 317–324, 2008.

[32] Richard E. Korf and Ariel Felner. Recent progress in heuristic search: A case study of the four-peg towers of Hanoi problem. In *IJCAI*, pages 2324–2329, 2007.

[33] Richard E. Korf and Peter Schultze. Large-scale parallel breadth-first search. In *AAAI*, pages 1380–1385, 2005.

[34] Daniel Kunkle and Gene Cooperman. Twenty-six moves suffice for Rubik's cube. In *Proc. of International Symposium on Symbolic and Algebraic Computation (ISSAC '07)*. ACM Press, 2007.

[35] Daniel Kunkle and Gene Cooperman. Solving Rubik's cube: Disk is the new RAM. *ACM Commmunications*, 51:31–33, 2008.

[36] Daniel Kunkle and Gene Cooperman. Harnessing parallel disks to solve Rubik's cube. *Journal of Symbolic Computation*, 2009. doi:10.1016/j.jsc.2008.04.013, to appear.

[37] Jerrold S. Leichter and Robert A. Whiteside. Implementing Linda for distributed and parallel processing. In *ICS '89: Proceedings of the 3rd international conference on Supercomputing*, pages 41–49, New York, NY, USA, 1989. ACM.

[38] Joao Marques-Silva. Practical applications of boolean satisfiability. pages 74–80, May 2008.

[39] Kim Milvang-Jensen and Alan J. Hu. BDDNOW: A parallel BDD package. In Ganesh Gopalakrishnan and Phillip J. Windley, editors, *FMCAD*, volume 1522 of *Lecture Notes in Computer Science*, pages 501–507. Springer, 1998.

[40] S.-i. Minato. Streaming BDD manipulation. *IEEE Transactions on Computers*, 51(5):474–485, 2002.

[41] Pradeep K. Nalla, Roland J. Weiss, Prakash M. Peranandam, Jürgen Ruf, Thomas Kropf, and Wolfgang Rosenstiel. Distributed symbolic bounded property checking. *Electr. Notes Theor. Comput. Sci.*, 135(2):47–63, 2006.

[42] Chris Nyberg. Sort benchmark home page, 2009. `http://www.hpl.hp.com/hosted/sortbenchmark/`.

[43] D. Moundanos P. Arunachalam, C. Chase. Distributed binary decision diagrams for verification of large circuits. In *IEEE International Conference on Computer Design (ICCD'96)*, pages 365–370, 1996.

[44] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with sawzall. *Sci. Program.*, 13(4):277–298, 2005.

[45] V.N. Rao and V. Kumar. Parallel depth first search. part i implementation. *International Journal of Parallel Programming*, 16:479–499, 1987.

[46] Eric Robinson. *Large Implicit State Space Enumeration: Overcoming Memory and Disk Limitations*. PhD thesis, Northeastern University, Boston, MA, 2008.

[47] Eric Robinson and Gene Cooperman. A parallel architecture for disk-based computing over the baby monster and other large finite simple groups. In *Proc. of International Symposium on Symbolic and Algebraic Computation (ISSAC '06)*, pages 298–305. ACM Press, 2006.

[48] Eric Robinson, Gene Cooperman, and Jürgen Müller. A disk-based parallel implementation for direct condensation of large permutation modules. In *Proc. of International Symposium on Symbolic and Algebraic Computation (ISSAC '07)*, pages 315–322. ACM Press, 2007.

[49] 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.

[50] Theo C. Ruys and Gerard J. Holzmann. Advanced spin tutorial. In *Model Checking Software, 11th International SPIN Workshop*, volume 2989 of *Lecture Notes in Computer Science*, pages 304–305. Springer, 2004.

[51] Tony Stornetta and Forrest Brewer. Implementation of an efficient parallel BDD package. In *DAC*, pages 641–644, 1996.

[52] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. of the First Symposium on Logic in Computer Science*, pages 322–331, 1986.

[53] Darren Erik Vengroff. A transparent parallel i/o environment. In *In Proc. 1994 DAGS Symposium on Parallel Computation*, pages 117–134, 1994.

[54] Jeffrey Scott Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.

[55] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory i: Two-level memories. *Algorithmica*, 12(2/3):110–147, 1994.

[56] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Symposium on Operating System Design and Implementation (OSDI)*, page 14, San Diego, CA, December 8-10 2008.

[57] Rong Zhou and Eric A. Hansen. Structured duplicate detection in external-memory graph search. In *AAAI*, pages 683–689, 2004.