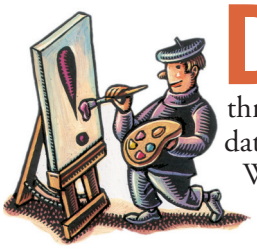
Daniel Kunkle and Gene Cooperman

Solving Rubik's Cube: Disk Is the New RAM

Substituting disk for RAM, disk-based computation is a way to increase working memory and achieve results that are not otherwise economical.



Disk-based computation represents a major new use of disks, in addition to the three historical uses: file systems, databases, and virtual memory. We recently demonstrated the importance of this fourth case by showing progress on a 25-year-old conjecture: determine how many moves suffice to solve Rubik's Cube. We chose Rubik's Cube because it has long served as a computationally challenging problem in which practitioners from a variety of disciplines have tested the efficacy of their techniques.

Our working group coined the term "disk-based computation" to describe our five-year effort to make use of parallel disks in scientific computation, including the many disks already available in a computational cluster. In doing so, the humble disk is elevated to a status normally reserved for RAM. RAM equivalence gives an application several orders of magnitude more working space for the same financial price. Such parallel disk-based methods are often based on lower-level external memory algorithms (such as those surveyed in [3]).

Our work reached the mainstream media in 2007 when we showed that Rubik's Cube can be solved in 26 moves or less [1]. At its heart, our computation simply enumerates and stores possible configurations of the puzzle. But, with more than 4.3×10^{19} possible configurations, proving that 26 moves suffice requires many terabytes of main memory. It was only our insight that "disk is the new RAM" that enabled us to overcome this memory barrier.

Rubik's Cube is an example of a large enumeration problem for which disk-based computation may lead to breakthroughs in many different problem domains, including group theory, hardware and software verification, coding theory, and constraint satisfaction. In them, one has an initial state, a method to produce neighboring states, and a need to store all reachable states. New powerful multicore computers are beginning to allow us to generate neighboring states faster than ever before. However, the ability to do so often means we also reach the limits of RAM more quickly than ever before.

Limiting ourselves to 4GB of main memory per computer is an arbitrary restriction not required by current technology. We are all conditioned by decades of history to regard disk as a hopelessly slow cousin to RAM. However, a simple back-of-the-envelope calculation shows this does not have to be so. The bandwidth of commodity disks is on the order of 100MB/s. A computer cluster with 50 disks provides 50 times the aggregate bandwidth, or 5GB/s, which is close to the bandwidth of commodity RAM. Thus 50 local disks provide the moral equivalent of a single extremely large RAM subsystem.

Viewed this way, a 50-node scientific computing cluster would be able to perform like a powerful parallel computer endowed with a single 10TB RAM subsystem. Justifying the use of distributed

LISA HANEY

Despite the fact that RAM stands for random access memory, we would almost never use the "new RAM" (disk) in random-access mode.

disks as a multi-terabyte main memory requires a small amount of math, as well as several somewhat larger caveats. A typical scientific computing cluster includes 200GB of often-unclaimed disk space per computer. A 50-node cluster provides 10TB of disk. As a nice side benefit, in today's commodity computer market, this 10TB of idle local disk space is essentially free.

How can we treat 10TB of disk space as if it were RAM? The answer depends on consideration of disk bandwidth, disk latency, and network bandwidth:

Thesis. Because 50 disks provide approximately the same bandwidth as a single RAM subsystem, the local disks of a computer cluster can be regarded as if they were a single very large RAM subsystem;

Caveat 1. Disk latency is much more limiting than disk bandwidth. Therefore, despite the fact that RAM stands for random access memory, we would almost never use the "new RAM" (disk) in random-access mode. The old-fashioned RAM already serves as our random-access cache;

Caveat 2. The new RAM is distributed across the local-area network. The aggregate network bandwidth of a cluster (even gigabit Ethernet) may not fully support the ideal 5GB/s aggregate bandwidth of the new RAM. Parallel algorithms must therefore be restructured to emphasize local access over network access. (This restriction is familiar to practitioners, who have long been aware of the impossibility of accessing traditional remote RAM at full speed over the network.)

Testbed

The details of the Rubik's Cube computation illustrate the benefits of disk-based computation. Whereas people usually solve Rubik's Cube in four or five stages, each involving fewer than one million combinations, the large main memory of disk-based computation allows a programmer to provide a two-stage solution where the largest subproblem involves 10^{14} combinations.

A person might first solve the top layer of the Cube (with nine smaller cubies, or individual box-like segments), then the bottom layer, and finally the remaining middle pieces. Solving the bottom and middle layers requires the use of macro moves, or sequences of moves that preserve the previous layers.

The programmer solves each of the two subproblems by performing a breadth-first search over all possible configurations, starting with the solved state. For the smaller of the two subproblems (10^5 configurations), this is easy.

For the larger of the two subproblems (10^{14} configurations), we first used the symmetries of Rubik's Cube to reduce it to 10^{12} configurations. We then analyzed several possible algorithms, settling on the final version, enumerating the 10^{12} configurations in 63 hours with the help of 128 processor cores and 7TB of disk space.

The primary difficulty in trying to extend a naive enumeration algorithm to execute on disk is how to efficiently perform duplicate detection, that is, to determine when a newly generated state has been seen before. This is typically done using a hash table or some other data structure that relies on random access. In the disk-based version, we avoid random access by delaying duplicate detection and collect many new states we check for duplicates in a later phase.

A brief description of the methods we considered when solving Rubik's Cube illustrates the kinds of

data structures and algorithms we have found useful in disk-based computation. The first method is based on external sort—a well-known disk-based sort that avoids random access at the cost of performing several passes through the data. New states discovered during the breadth-first search are saved to disk without checking for duplicates. When an entire level of the search is completed, the new states are externally sorted and merged into a sorted list containing all previously discovered states.

In this way, we eliminate random-access data structures, using sorted lists in their place. Eliminating random access comes at the cost of having to maintain the sorted order of the lists. Further, this method requires that we save all known states. For our Rubik's Cube computation, storing all configurations would require 11TB, not counting the buffer space for newly generated states.

The second method avoids storing all seen states and also removes the need for expensive external sorting operations. Instead of explicitly storing the known states, we use a disk-based table to record the previously discovered states. To avoid random access, we split this table into contiguous pieces such that each piece fits into RAM. When performing duplicate detection, we load one piece of the table into RAM at a time and remove duplicate states that correspond to that portion of the search space.

Even though this method avoids storing explored states, it still requires the storage of the open list of new states from which duplicates have not been removed. For our Rubik's Cube computation, the open list has a maximum size of 50TB. To avoid this limitation, we use a technique we call `implicit open list` to encode the open states using a hash table, rather than an explicit list. This allows us to complete the computation using just 7TB of disk space.

Organizing Principle

A unified framework is required to broaden the appeal of disk-based computation beyond Rubik's Cube. Our team is now searching for an organizing principle that will allow for the construction of a software library or language extension that does for disk-based computation what numerical libraries have done for numerical analysis. As an initial step, we have begun a comparative analysis of eight different techniques for disk-based enumeration [2]. This analysis is based on the methods we used for Rubik's Cube, along with our solutions to several model problems in computational group theory.

The search cuts across many areas of computer science. For example, in systems and architecture, how can we design disk-based computations to balance the use of CPU, RAM, network, and disk? In theory and algorithms, what class of computations can be converted to efficient disk-based computation? In software engineering and programming languages, how can we separate disk-specific data structures and algorithms from problem-specific concerns? By answering such questions, we will advance the use of disk-based computation, enabling solutions to problems requiring even petabytes of memory. C

References

1. Kunkle, D. and Cooperman, G. Twenty-six moves suffice for Rubik's Cube. In *Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation* (Waterloo, Ontario, Canada, July 29–Aug. 1). ACM Press, New York, 2007, 235–242.
2. Robinson, R., Kunkle, D., and Cooperman, G. A comparative analysis of parallel disk-based methods for enumerating implicit graphs. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation* (London, Ontario, Canada, July 27–28). ACM Press, New York, 2007, 78–87.
3. Vitter, J. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys 33,* 2 (June 2001), 209–271.

Daniel Kunkle (kunkle@ccs.neu.edu) is a Ph.D. candidate in computer science in the College of Computer and Information Science at Northeastern University, Boston, MA.

Gene Cooperman (gene@ccs.neu.edu) is a professor in the College of Computer and Information Science at Northeastern University, Boston, MA., where he is also the director of the Institute for Complex Scientific Software and the head of the High Performance Computing Laboratory.

DOI: 10.1145/1330311.1330319