Data-Intensive Scalable Computing for Scientific Applications

Increasingly, scientific computing applications must accumulate and manage massive datasets, as well as perform sophisticated computations over these data. Such applications call for data-intensive scalable computer (DISC) systems, which differ in fundamental ways from existing high-performance computing systems.

The ever-increasing volumes of scientific data being generated, collected, and stored call for a new class of high-performance computing facility that places emphasis on *data*, rather than raw computation, as the core focus of the system. The system is responsible for the acquisition, updating, sharing, and archiving of the data, and it supports sophisticated forms of computation over the data. I refer to such systems as *data-intensive scalable computer* (DISC) systems. I believe that DISC systems will yield breakthroughs in many scientific disciplines.

Much of the inspiration for DISC systems comes from the server infrastructures that have been developed to support Internet-based services, such as Web search, social networks, electronic mail, and online shopping. Companies such as Google, Yahoo, Facebook, and Amazon have created computer systems of unprecedented scale, providing high-level storage and computational capacity and high-level availability, while always keeping a close eye on cost.

Scientific computing has traditionally focused on performing numerically intense computations, such as running large-scale simulations to predict a system's behavior based on some underlying engineering or scientific model. Some have observed, however, that scientific research increasingly relies on computing over large datasets, sometimes referring to this as "e-Science."[1] The scientific world has become increasingly data intensive, as sensor, networking, and storage technologies make it possible to collect and store information ranging from DNA sequences to telescope imagery. In fact, the machines performing large-scale simulations also generate huge volumes of data that must be further analyzed to determine what phenomena they predict.

My claim is that systems designed along the lines of the DISC systems used in the Internet-services industry are well suited to perform the data management and analysis tasks required to support scientific data analysis. By colocating storage and computation, by providing high levels of scalability, and by supporting programming models that allow applications to be written in high-level, machine-independent forms, DISC systems will let us create capabilities well beyond those currently available to most scientists. This claim has been validated over the past several years by many scientists running programs on systems ranging from 10 to 20 processors mounted in a single rack to thousand-processor systems made available by Internet-services companies.

Here, I present the case for DISC as an important direction for large-scale computing systems. I do this through a survey of recent work in creating systems that are both easier to program than most other large-scale computing

1521-9615/11/$26.00 © 2011 IEEE
Copublished by the IEEE CS and the AIP

Randal E. Bryant
Carnegie Mellon University

systems and can support data analysis tasks in scientific domains.

Motivating Applications

The following applications come from very different scientific domains, but they have in common the central role of data in their computation.

The first example application infers biological function from genomic sequences. Increasingly, computational biology involves comparing genomic data from different species and from different organisms of the same species to determine how information is encoded in DNA. Ever-larger datasets are being collected as new sequences are discovered, and new forms of derived data are computed.[2]

For example, the National Center for Biotechnology Innovation maintains the GenBank database of nucleotide sequences, which has been doubling in size every 10 months. As of August 2009, it contained over 250 billion nucleotide bases from more than 150,000 distinct organisms. Although the total volume of data is only approximately one terabyte, the computations performed are very demanding. In addition, the amount of genetic information available to researchers will increase rapidly once it becomes commercially feasible to sequence the DNA of individual organisms—such as to enable *pharmacogenomics*, which predicts a patient's response to different drugs based on his or her genetic makeup.

Increasingly, computational biology involves comparing genomic data from different species and from different organisms of the same species to determine how information is encoded in DNA.

A second example concerns discovering astronomical phenomena from telescope image data. Astronomers and computer scientists working together have successfully used data warehouse technology to store, manage, and provide access to data collected by large-scale survey telescopes.[3] Their systems support not only hundreds of researchers but also thousands of amateur scientists worldwide. As telescope resolutions have increased, the data volumes being collected have outstripped the earlier systems' database technology. Shifting to DISC technology will enable them to continue scaling as data needs increase, while also enabling more sophisticated computations on the data.

My final example focuses on applications that analyze large-scale simulation results. Computational science often involves performing simulations of physical phenomena, based on hypothesized models, to see what effects these models predict and whether they match observed data. Large-scale simulations can produce multiple terabytes of data, making it a major challenge to validate, analyze, and visualize the results.

For example, the recent Millennium XXL run modeled the evolution of the universe by representing it as $6{,}720^3$ (around 3×10^{11}) particles and simulating its behavior over time. All told, the computation lasted for 9.3 days, running on a 12,288-core supercomputer. Storing the complete data for the simulation would have required 700 terabytes. Instead, they saved the values for only four time steps, reducing the required storage to 70 terabytes. Even so, analyzing and understanding the generated data poses a major challenge in data-intensive computing.

These and many other tasks involve collecting and maintaining very large datasets and applying vast amounts of computational power to the data. An increasing number of data-intensive computational problems are arising as technology for capturing and storing data becomes more widespread and economical, and as the Web provides a mechanism to retrieve data from around the world. Quite importantly, the relevant datasets aren't static; researchers must update them regularly, and new data derived from computations over the raw information must also be updated and saved.

Traditional High-Performance Computing Systems

Over the past 60 years, the term "scientific computing" has been associated primarily with applications that require large and complex numerical calculations, such as those to run simulations. A class of machines—commonly referred to as *high-performance computing systems*, or HPC—has evolved to support such computations. Most of the world's supercomputers, for example, fall into this category. In the following, I briefly describe how typical HPC systems are structured, programmed, and operated, and describe their strengths and shortcomings.

A typical HPC system consists of a set of processing *nodes*, each containing a small number of microprocessors, random-access memory, and

a network interface. A high-bandwidth interconnection network links the nodes together. High-capacity storage is provided by a separate *storage server*, such that transferring data between processors and disk storage requires streaming data over the interconnection network between the nodes and the storage server.

In most HPC environments, application programs operate the nodes in a tightly synchronized fashion, with a careful orchestration of the computation and communication. For example, a typical simulation operates according to a *bulk synchronous* programming model,[4] where the system to be simulated is partitioned into a number of regions, each of which is mapped onto one of the nodes. Simulation proceeds by a series of time steps. For each step, each node computes the behavior of its region for some brief time duration and then communicates boundary values with the nodes simulating adjacent regions.

Long-running programs periodically save the entire computation's state by *checkpointing*—copying data from the nodes' random-access memories to the storage server. When a computational error occurs—for example, due to a hardware or software failure—the system state is restored to that of the most recent checkpoint by loading the node memories with data from the storage server. Any computation performed since the time of the last checkpoint is therefore wasted.

Although these HPC systems have yielded many important scientific insights, this style of system design clearly lacks scalability—building ever-larger versions of such machines has become increasingly difficult and less cost effective. Indeed, while supercomputers keep getting bigger and have become a source of national pride (as indicated by large amount of attention paid to the TOP500 supercomputer ranking[5]), there are many scientific applications that don't map well onto these machines. Among their shortcomings are that they have limited ability to work with datasets that can't be kept resident within the nodes' random-access memories because of the partitioning between the machine's computational resources and the storage server's resources.

Also, they require careful tuning of the software to keep the machine running at high efficiency. This tuning leads to programs where a large fraction of the code and the programming effort must be dedicated to managing the computation and communication. Although there have been several attempts to create languages that allow programmers to describe computations in more abstract forms,[6] most applications are still written using low-level notations to enable more control over resource utilization.

In addition, hardware or software errors are costly with these systems, due to the high cost of checkpointing (typically requiring several minutes) and to the wasted computation incurred when rolling back to the most recent checkpoint. As a consequence, achieving acceptable reliability requires using carefully engineered and conservatively designed hardware. The resulting cost-performance is therefore well below that of a typical desktop machine. Reliability considerations are becoming a dominant challenge in creating next-generation HPC systems. As the number of components—processors, memories, routers, network cables, and power supplies—increase, the probability of having at least one failed component increases dramatically.

Indeed, while supercomputers keep getting bigger and have become a source of national pride, there are many scientific applications that don't map well onto these machines.

Data-Intensive Scalable Computing Systems

I now contrast the systems used for large-scale, computationally intense applications with those suitable for collecting, managing, and computing over very large datasets.

Key Principles

A data-intensive scalable computing system can range from a single rack of processors to a building filled with 100,000 processors, dwarfing the size of even the world's largest supercomputers. I've enumerated four key principles that I believe DISC systems should employ to support scalable, data-intensive computing:

1. *Intrinsic rather than extrinsic data.* Collecting and maintaining data should be the system's duties rather than those of individual users. The system must retrieve new and updated information over networks and perform derived computations as background tasks. Users should be able to use rich queries based on content and identity to access the data. Reliability mechanisms (such as replication and error correction) should be incorporated

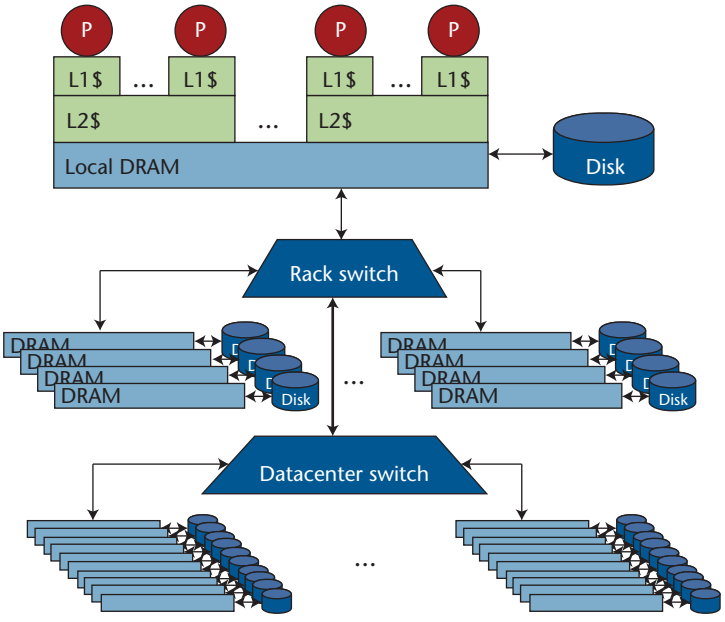

Figure 1. Hierarchical structure of a typical warehouse-scale computer.[8] The example system includes 2,400 nodes, organized as a hierarchy of 30 racks that each have 80 server nodes. — *Image courtesy Morgan & Claypool publishers; used with permission.*

to ensure data integrity and availability as part of the system function.

2. *High-level programming models for expressing computations over the data.* The application developer should be provided with powerful, high-level programming primitives that express natural forms of parallelism and that don't specify a particular machine configuration (such as the number of processing elements). It should be the job of the compiler and runtime system to map these computations onto the machine efficiently.
3. *Interactive access.* Users should be able to execute programs interactively and with widely varying computation and storage requirements. The system should respond to user queries and simple computations on the stored data in less than one second, while more involved computations could take longer but not degrade performance for the queries and simple computations of others. To support interactive computation on a DISC system, there must be some over-provisioning of resources. I believe that the consequent increased cost of computing resources can be justified based on the increased productivity of system users.
4. *Scalable mechanisms to ensure high reliability and availability.* A DISC system should employ nonstop reliability mechanisms, where all original and intermediate data are stored in redundant forms and selective recomputation can be performed in the event of component or data failures. Furthermore, the machine should automatically diagnose and disable failed components; these would be replaced only when enough had accumulated to impair system performance. The machine should be available on a 24/7 basis, with hardware and software replacements and upgrades performed on the live system.

Many of the above items are interrelated due to the hardware technology's underlying nature. Magnetic disk storage is currently the best technology for storing large amounts of data with reasonable cost and access rates. Locating the data within a large-scale system therefore leads to a system organization consisting of a number of nodes, each with a few processors; a few disk drives; and an interface to a high-speed, local network. Compared to traditional HPC machines—where the disk drives are in a separate storage server—incorporating disk drives within the processing nodes dramatically changes the system characteristics. Disk drives have widely varying access times, making it more difficult to tightly synchronize the nodes' activities. This reduces the effectiveness of the low-level programming models used on existing HPC machines. Disk drives also have a much higher failure rate than purely electronic components,[7] increasing the need for fault tolerance mechanisms.

Internet-services companies (such as Google, Yahoo, Facebook, and Amazon) have devised approaches for constructing large-scale computing facilities that fulfill many of these proposed DISC system characteristics. By layering sophisticated software on top of relatively generic hardware, they achieve high degrees of scalability and cost-performance.

Hardware Configuration

Although the detailed designs of systems for large-scale, data-intensive computing vary across companies and even from one installation to another, they follow a general model recently labeled as *warehouse-scale computers*.[8] This term emphasizes the fact that these machines are typically constructed by filling large buildings with many racks of server machines, often packaged inside of shipping containers so the facility can be expanded by simply adding more racks or containers.

Figure 1 shows the hierarchical structure for a typical cluster in a warehouse-scale machine. It consists of 2,400 nodes, organized as a hierarchy

of 30 racks, each containing 80 server nodes. Each node contains a single, multicore CPU and several disk drives, having a total capacity of 2 terabytes. In aggregate, this would create a system with 30 terabytes of DRAM and 4.8 petabytes of disk storage. The servers within a rack are connected through a 1-gigabit-per-second (Gbps) Ethernet switch, while the racks are connected to each other through an Ethernet switch supporting either 1 gigabyte per second (GBps) or 10 Gbps.

One key aspect of this hardware design is that it's based entirely on technology found in data centers operated by companies around the world. Hardware is chosen on the basis of its price-performance, rather than on performance alone. By using commodity parts, the facility can take advantage of the huge economies of scale that the global IT industry provides.

System and Application Software

At Google[9] and other Internet-services companies, the set of disk drives in a cluster is managed as a distributed file system with a single namespace and multiple copies (typically three) of each file stored on different drives. This replication ensures that at least one copy of a file will be available despite failures of individual disk drives or even entire racks. In addition, the system can often optimize a node's file access by using the nearest copy. These file systems illustrate a key feature of DISC systems: they use system software to create a reliable system from a set of unreliable hardware components.

Google has also demonstrated that application programs for DISC systems can be written using a machine-independent, data-parallel programming model. They have implemented a programming abstraction, *MapReduce*,[10] that supports powerful parallel computation over large amounts of data. MapReduce can be viewed as operating over the data contained in one set of files to generate data in a new set of files. To perform a MapReduce operation, the user specifies two functions: a *mapping* function that generates values and associated keys from an input file, and a *reduction* function that describes how all data associated with each key should be combined. MapReduce is used by Google to compute statistics about documents, to create the index structures used by its search engine, and to implement their PageRank algorithm[11] for quantifying the relative importance of different Web documents.

The runtime system implements MapReduce, handling details of scheduling, load balancing, and error recovery.[12] The overall execution involves partitioning the mapping and reduction operations into tasks, where the total number of tasks is much greater than the number of CPU cores. (For example, one report from Google states that a single computation may consist of 200,000 mapping tasks and 4,000 reduction tasks executing on 2,000 processors.[12]) Each task involves reading data from a file, performing some computation, and writing the result as another file. A centralized scheduler orchestrates the task executions, dynamically mapping them in ways that try to locate a task near a copy of its input file, while also balancing the load across the nodes. Between the mapping and reduction steps, the system performs a *shuffle* to group the map outputs according to their keys. This involves either a large parallel sort, or a hash mapping.

The combination of the MapReduce execution model, a distributed file system, and dynamic scheduling is the key for achieving scalability and fault tolerance. By defining each operation as a mapping between sets of files in a reliable file system, the system can recover from a fault by recomputing just those tasks that fail. In a sense, the system performs localized checkpointing with each task. The dynamic scheduling automatically controls the mapping of the computations onto the processor resources. This eliminates the need for the programmer to carefully tune an application program to a particular machine configuration, and it also automatically adapts to variations, such as those caused by unpredictable disk access and network communication latencies. It also reduces the cost of recovering from a fault; tasks that must be recomputed are simply added to the task queue and handled as part of the normal execution.

These file systems illustrate a key feature of DISC systems: they use system software to create a reliable system from a set of unreliable hardware components.

The open source Apache/Hadoop project (http://hadoop.apache.org) provides capabilities similar to Google's file system and MapReduce programming model. Apache/Hadoop is finding widespread use within companies, research organizations, and educational institutions. It can run on configurations as small as a single processor (useful for program development) and as large as systems consisting of thousands of processors. To perform a MapReduce operation with Hadoop, a programmer provides

two Java classes: one to define the mapping operation and one to define the reduction operation. Compared to other systems for developing parallel programs, those based on MapReduce express the overall computation at a much higher level and in a more machine-independent form.

Assessing MapReduce

The MapReduce programming model provides many advantages for expressing data-intensive applications. It leads to natural forms of parallelism. It also enables a mapping that combines dynamic scheduling and selective recomputation, which lets the system automatically balance the loads on processors and readily tolerate failures. Further, MapReduce has proven remarkably expressive, due to the generality of expressing matching between data elements and program-defined keys. For example, a number of graph algorithms and database operations can be defined in MapReduce terms.[13]

On the other hand, MapReduce is by no means the universal solution to all parallel programming applications, and existing implementations of MapReduce have significant sources of inefficiency. One limitation of the MapReduce model is that it requires performing computation over an entire dataset. For example, it would be very inefficient in supporting simple query operations, where the goal is to identify attributes involving just a few data elements. A second limitation is that it limits the forms of abstractions used in expressing programs. An application must be organized as a sequence of MapReduce operations. This works well in some cases but can be very awkward in others.

Among the limitations of current MapReduce implementations, one of the biggest is that it involves a high degree of global synchronization. Although there's ample asynchrony in performing a single MapReduce operation, typical applications involve sequences of MapReduce steps, each of which can begin only after the preceding one has completed. This is especially inefficient for iterative computations, where the iterations can converge much more rapidly for one region of the data than for another. MapReduce steps must continue over the entire dataset until the last region converges. A second inefficiency in current implementations of MapReduce is that each MapReduce step involves a considerable amount of disk I/O. Always mapping from one file to another has advantages in terms of fault tolerance, but it can be very inefficient for many computations.

To see the impact of some of these limitations, consider the following scheme for implementing a bulk-synchronous parallel simulation with MapReduce. The target system would be divided into regions, with the state of each region stored in a separate file. A single MapReduce step would then run the simulation for one time step:

- *The map phase* would perform the computations internal to each region.
- The *shuffle phase* would gather the internal and boundary data for each region.
- *The reduce phase* would merge each region's data to compute the new region state.

This would indeed be an elegant way to express the computation, but its performance for existing implementations of MapReduce would fall well short of that achieved by a more traditional HPC system implementation. Most significantly, it would involve reading the state of the entire simulation from a set of disk files at the beginning of each time step and then writing it to a set of disk files at the end. The exact performance penalty incurred by this much higher amount of disk I/O would depend on many factors of the two implementations, but, given the slow performance of disks compared to keeping the data in memory, the MapReduce version could easily run several orders of magnitude slower than the conventional implementation.

Refinements and Possible Improvements of MapReduce

Several research activities are attempting to address MapReduce's shortcomings and current implementations. Here, I highlight a few such efforts. Several projects involve extending or improving the MapReduce implementation in Hadoop. These have the advantage of building on a large and sophisticated code base, although they're then constrained in the degree to which they can change the underlying computation model. For example, Jaliya Ekanayake and colleagues eliminate some of the disk operations by passing results from one task to another through a memory buffer, rather than by writing and then reading files.[14] In case of failure, recovery requires backing up to an earlier part in the computation, where results have been stored as files, and then recomputing results forward. The frequency of storage to disk must be optimized to balance between the performance under normal conditions and the time to recovery under failure. Passing data through memory buffers also creates a more difficult set of scheduling constraints—it's much better to schedule the consuming task for some data on the same machine on which they were produced.

Another way to reduce the amount of disk I/O and to reduce the degree of global synchronization in Hadoop involves performing localized MapReduce operations in addition to global ones.[15] This works for computations that require performing multiple iterations until a convergence criterion is achieved. For cases in which the solution is unique, augmenting a global MapReduce step with local ones can speed convergence without compromising the results' correctness.

A more ambitious research agenda involves moving away from MapReduce altogether to consider different, but related, strategies. Perhaps the most ambitious effort is Microsoft Research's Dryad Project.[16] Dryad starts with an underlying execution model similar to that of MapReduce: computation is expressed as a set of tasks, each providing a functional mapping from one data element to another. Rather than layering these tasks as mapping, shuffle, and reduction steps, however, Dryad simply views the overall computation as an arbitrary directed acyclic graph. Dryad can readily express a sequence of MapReduce operations, but it can express other styles as well. In addition, Dryad can choose whether to store a data element as a file in a distributed file system, as a file in a local file system, or in a memory buffer. As discussed earlier, recovering from a failure might require recomputing multiple tasks, starting from the last point at which results were stored in reliable files. The system can optimize overall performance by balancing between the cost of storing results to files and the cost of recovering from faults.

Several other projects provide higher levels of programming abstraction by building upon MapReduce and Dryad. These include Google's Sawzall;[17] Pig, developed at Yahoo[18] and included as part of the Apache/Hadoop open source project; and Facebook's Hive.[19] All of these projects use MapReduce as the underlying execution method. Similarly, Microsoft's DryadLINQ programming system builds on Dryad's distributed execution engine and combines it with the .NET Language Integrated Query.[20]

DISC for Science

The ease of deploying Hadoop-based clusters has led to widespread use of the MapReduce programming framework in academia, research labs, and industry. Scientific computing projects that have used MapReduce thus far vary widely in how much of MapReduce's full functionality they use.

On one hand, many of the published applications can be classified as being "embarrassingly parallel." These applications involve simply performing independent computations over many separate datasets. This can be expressed in MapReduce by having the mapping operation perform the desired computation and the reduction be an identity operation. Such programs can take advantage of MapReduce's dynamic scheduling, load balancing, and fault-recovery mechanisms to manage the computations. In this sense, MapReduce is serving as a workload-management system, with the distributed file system providing a mechanism to distribute data to the computations and to collect the results. A slight variant on this application class is to use the reduction operation to collect some aggregate information about different computations' results, for example to compute statistical summaries.

A slightly more sophisticated class of applications makes fuller use of the shuffle operation's ability to redistribute data organized one way in the input files to another way in the output files. Example uses for this capability include finding correlations between two different sets of genomic data,[2] or between different images in astronomic data.[21]

On the other end of the scale, an important applications class involves analyzing the overall structure of a set of elements, as represented by large, sparse graphs. Their computations consist of a series of steps, where each step involves propagating values along a graph's edges and then combining them for each node. The mapping operation performs the propagations, using the destination operation as the key, and the reduction operation combines values on a per-node basis. Examples include the PageRank computation,[11] as well as other graph-mining tasks.[13,22] Although these are typically described as computations over large graphs, they can also be thought of as iterative computations over sparse and irregular matrices. These applications point to a growing trend toward applications that are *both* computationally intensive and data intensive. For example, whereas the PageRank computation finds just the principal eigenvector of a sparse matrix, more sophisticated methods for characterizing graphs require finding the top k eigenvectors.[23] Researchers have proposed a new class of benchmarks, Graph500.[24] To stress the need for high-performance machines that can support applications involving very large and irregularly structured graphs.

I anticipate that more scientists will use MapReduce programming and related methods for implementing different forms of data analysis on scientific data, as more researchers gain access to cluster hardware and large-scale datasets.

Moreover, both the datasets and the computational needs will keep growing at a rapid pace.

As more scientific applications rely on analyzing very large datasets, the need for cost-efficient, large-scale computing capabilities will continue to increase. I've argued that these systems should be constructed and programmed in ways that can readily scale to meet the growing needs. The desire for scalability leads to a very different system style than has traditionally been used for large-scale scientific computing.

One important system concept for scaling parallel computing beyond 100 or so processors is to incorporate error detection and recovery into the runtime system and to isolate programmers from both transient and permanent failures as much as possible. Historically, most work on and implementations of parallel programming assumed that the hardware operates without errors. By assuming instead that every computation or information retrieval step can fail to complete or can return incorrect answers, it's possible to devise strategies to correct or recover from errors that allow the system to operate continuously.

It's important to avoid the tightly synchronized parallel programming notations used for current supercomputers. Instead, programs should dynamically adapt to the available resources and be designed to operate efficiently in a more asynchronous execution environment. The MapReduce programming environment is a first step in this general direction; future research will likely lead to approaches that both expand the generality and improve the efficiency of this computational model.

In describing the ideas behind DISC to other researchers, I've found great enthusiasm among both potential system users and system developers. It's time for the computer science community to step up their level of thinking about the power of data-intensive computing and the scientific advances it can produce. Just as Web search has become an essential tool in the lives of people ranging from schoolchildren to academic researchers to senior citizens, DISC systems could change the face of scientific research worldwide. CiSE

References

1. T. Hey and A. Trefethen, "The Data Deluge: An e-Science Perspective," *Grid Computing: Making the Global Infrastructure a Reality*, F. Berman, G.C. Fix, and A.J G. Hey, eds., 2003, John Wiley & Sons, pp. 809–824.
2. S. Leo, F. Santoni, and G. Zanetti, "Biodoop: Bioinformatics on Hadoop," *Proc. Int'l Conf. Parallel Processing Workshops*, IEEE CS Press, 2009, pp. 415–422.
3. A.S. Szalay et al., "Designing and Mining Multi-Terabyte Astronomy Archives: The Sloan Digital Sky Survey," *Proc. Int'l Conf. Management of Data*, ACM Press, 2000, pp. 451–462.
4. L.G. Valiant, "A Bridging Model for Parallel Computation," *Comm. ACM*, vol. 33, no. 8, 1990, pp. 103–111.
5. H.W. Meuer, "The TOP500 Project: Looking Back over 15 Years of Supercomputing Experience," *Informatik-Spektrum*, vol. 31, no. 3, 2008, pp. 203–222.
6. E. Allen et al., *The Fortress Language Specification*, tech. report, Sun Microsystems, 2007.
7. B. Schroeder and G.A. Gibson, "Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You?" *Proc. Usenix Conf. File and Storage Technologies* (FAST), Usenix Assoc., 2007, pp. 1–16.
8. L.A. Barroso and U. Hölze, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, Morgan & Claypool, 2009.
9. S. Ghemawat, H. Gobioff, and S.T. Leung, "The Google File System," *Proc. Symp. Operating Systems Principles* (SOSP), ACM Press, 2003, pp. 29–43.
10. J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Operating Systems Design and Implementation* (OSDI), Usenix Assoc., 2004; http://labs.google.com/papers/mapreduce-osdi04.pdf.
11. L. Page et al., *The PageRank Citation Ranking: Bringing Order to the Web*, tech. report, Stanford Digital Library Technologies Project, 1998.
12. J. Dean, "Experiences with MapReduce, an Abstraction for Large-Scale Computation," *Proc. ACM Int'l Conf. Parallel Architecture and Compilation Techniques*, ACM Press, 2006; doi:10.1145/1152154.1152155.
13. J. Cohen, "Graph Twiddling in a MapReduce World," *Computing in Science and Eng.*, vol. 11, no. 4, 2009, pp. 29–41.
14. J. Ekanayake, S. Pallickara, and G. Fox, "MapReduce for Data Intensive Scientific Analyses," *Proc. 4th Int'l Conf. eScience*, IEEE Press, 2008, pp. 277–284.
15. K. Kambatla et al., "Asynchronous Algorithms in MapReduce," *Proc. Int'l Conf. Cluster Computing*, IEEE Press, 2010, pp. 245–254.
16. M. Isard et al., "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," *ACM SIGOPS Operating Systems Rev.—EuroSys'07 Conf. Proc.*, ACM Press, vol. 41, no. 3, 2007; doi:10.1145/1272996.1273005.
17. R. Pike et al., "Interpreting the Data: Parallel Analysis with Sawzall," *Scientific Programming*, vol. 13, no. 4, 2005, pp. 277–298.
18. C. Olston et al., "Pig Latin: A Not-So-Foreign Language for Data Processing," *Proc. SIGMOD Int'l*

Conf. Management of Data, ACM Press, 2008, pp. 1099–1110.
19. A. Thusoo et al., "Hive: A Warehousing Solution over a Map-Reduce Framework," *Proc. Very Large Database Endowment*, ACM Press, vol. 2, no. 1, 2009, pp. 1626–1629; www.vldb.org/pvldb/2/vldb09-938.pdf.
20. M. Isard and Y. Yu, "Distributed Data-Parallel Computing Using a High-Level Programming Language," *Proc. SIGMOD Int'l Conf. Management of Data*, ACM Press, 2009, pp. 987–994.
21. Y.C. Kwon et al., "Scalable Clustering Algorithm for *N*-Body Simulations in a Shared-Nothing Cluster," *Scientific and Statistical Database Management*, LNCS 6187, Springer-Verlag, 2010, pp. 132–150.
22. U. Kang, C.E. Tsourakakis, and C. Faloutsos, "Pegasus: A Peta-Scale Graph Mining System Implementation and Observations," *Proc. Int'l Conf. Data Mining*, IEEE Press, 2009, pp. 229–238.
23. U. Kang, B. Meeder, and C. Faloutsos, "Spectral Analysis for Billion-Scale Graphs: Discoveries and Implementation," *Advances in Knowledge Discovery and Data Mining*, LNCS 6635, Springer-Verlag, 2011, pp. 13–25.
24. M. Anderson, "Better Benchmarking for Supercomputers," *IEEE Spectrum*, vol. 48, no. 1, 2011, pp. 12–14.

Randal E. Bryant *is a professor of computer science and dean of the School of Computer Science at Carnegie Mellon University. His research interests include methods for formally verifying digital hardware and some forms of software; he has developed several techniques to verify circuits by symbolic simulation, with levels of abstraction ranging from transistors to very high-level representations. More recently, he has been involved in several efforts to apply large-scale, parallel computing to data-intensive problems. He coauthored, with David R. O'Hallaron,* Computer Systems: A Programmer's Perspective *(Prentice-Hall, 2nd ed., 2010). Bryant has a PhD in computer science from the Massachusetts Institute of Technology. He's a fellow of IEEE and the ACM, as well as a member of the National Academy of Engineering and the American Academy of Arts and Sciences. Contact him at randy.bryant@cs.cmu.edu.*