

Graph Analysis with High-Performance Computing

Large, complex graphs arise in many settings including the Internet, social networks, and communication networks. To study such data sets, the authors explored the use of high-performance computing (HPC) for graph algorithms. They found that the challenges in these applications are quite different from those arising in traditional HPC applications and that massively multithreaded machines are well suited for graph problems.

Graphs are among the most widely used combinatorial tools in computing. In science and engineering, they describe the structure of sparse matrices, facilitate load balancing in parallel computations, help researchers study molecular structures, and help us mesh complex geometries. We can also use graphs to model distribution networks, economies, and epidemics, study social systems, and describe sets of protein interactions.

Graphs are applicable in such diverse settings because they're an abstract way of describing interactions between entities. A graph consists of a set of entities known as *vertices* and a set of pairwise relationships between entities known as *edges*. Many refinements and augmentations of this basic model produce vertices and edges with additional properties.

A typical step in a graph algorithm involves visiting a vertex v and then visiting v 's neighbors (the set of vertices connected to v by an edge). For some graphs—such as those that describe a finite difference matrix's nonzero structure—this set of neighbors can have a regular and predictable structure. We can exploit this structure and design

data structures that improve cache performance when accessing v 's neighbors. However, in many emerging applications such as social and economic modeling, the graph has very little exploitable structure. In fact, in such settings, v 's neighbors can be widely dispersed in global memory. This leads to data access patterns that make very poor use of memory hierarchies, which in turn can result in idle processors. Because access patterns are data dependent (that is, they're a function of the graph's edge structure), standard prefetching techniques are often ineffective. In addition, graph algorithms typically have very little work to do when visiting a vertex, so there's little computation for each memory access. For all these reasons, graph computations often achieve a low percentage of theoretical peak performance on traditional processors. Similar challenges plague many other combinatorial applications as well.

However, as graph applications grow in breadth and size, we've witnessed a real need for effective graph algorithm parallelization, even though parallelism presents yet another set of challenges for graph algorithms. An expansive literature on algorithms is designed for theoretical parallel random access machines (PRAMs), but these models aren't realistic, and there are comparatively few success stories for practical parallel graph implementations. In this article, we argue that this gap reflects a mismatch between the demands of graph algorithms and the capabilities of mainstream parallel

computer architectures. Graphs in scientific computing often reflect a physical object's geometry, so we can partition them among a parallel machine's processors in such a way that few edges cross between processors. However, this isn't true of the more abstract graphs that arise in some emerging applications. In addition, parallelism in graph algorithms tends to be fine-grained, with the degree of parallelism varying over the course of the algorithm. This type of parallelism isn't supported in traditional parallel architectures and programming models. To overcome these challenges, we recently started developing graph algorithms on a nontraditional, massively multithreaded supercomputer.

The High-Performance Computing Landscape

By far, the most popular class of parallel machines is distributed-memory computers, which consist of a set of commodity processors connected by a network. These machines are relatively inexpensive, but they're very effective on many scientific problems.

Distributed-memory machines are generally programmed with explicit message passing via the message-passing interface (MPI). With MPI, the user divides the data among the processors and determines which processor performs which tasks; the processors exchange data via user-controlled messages. Although high performance is achievable for many applications, the detailed control of data partitioning and communication can be tedious and error-prone.

MPI programs are typically written in a bulk-synchronous style, in which processors alternate between working independently on local data and participating in collective communication operations. By grouping data exchanges into large, collective operations, the overall latency cost shrinks substantially at the expense of algorithmic flexibility. Data can be transmitted only at pauses between computational steps, and the lack of transmission on demand makes it difficult to exploit fine-grained parallelism in an application. This problem is particularly acute in many basic graph algorithms.

Partitioned Global Address-Space Computing

MPI isn't the only way to program distributed-memory parallel computers. An important alternative that's better suited for fine-grained parallelism is to use a partitioned global address-space language, epitomized by Unified Parallel C (UPC).¹ In a UPC program, the programmer is still responsible for distinguishing between local and global data, but the language supports operations

on remote memory locations with simple syntax. This support for a global address space also facilitates writing programs with complex data access patterns. UPC sits on top of a communication layer that allows for more fine-grained communication than MPI and so can sometimes achieve higher performance. However, as with MPI, the number of control threads is constant in a UPC program—that is, it's generally equal to the number of processors or cores. As we argue later, the lack of dynamic threads is a significant impediment to the development of high-performing graph software.

Shared-Memory Computers

UPC provides a software illusion of globally addressable memory on distributed-memory hardware, but the hardware can also provide support for a global address space. We can categorize shared-memory computers in various ways—specifically, let's look at cache-coherent and massively multithreaded machines.

Cache-coherent parallel computers. In symmetric multiprocessors (SMPs), global memory is universally accessible to each processor. The most common ways to program these machines are via OpenMP² or with a threading approach.³ An SMP provides hardware support for access to addresses in global memory so that threads can quickly retrieve any address in the machine. This allows for higher performance on highly unstructured problems than is possible on distributed-memory machines. The latency challenge is addressed by using faster hardware to access memory, but SMPs have some inherent performance limitations: in a multiprocessor machine with multiple caches, for example, the cache-coherence problem is a significant challenge. It adds overhead, which degrades performance, even for problems in which reads are much more common than writes.

A second performance challenge in SMPs is the protocol for thread synchronization and scheduling. If several threads try to access the same memory region, the system must apply some protocol to ensure correct program execution. Some threads might be blocked for a period of time—current versions of OpenMP require the number of threads to equal the number of processors, so a blocked thread corresponds to an idle processor. Although a more dynamic threading model might appear in future versions of OpenMP, this problem currently causes significant performance challenges for graph algorithms.

Massively multithreaded architectures. Massively

multithreaded machines, such as the Cray MTA-2⁴ and its successor the XMT, address the latency challenge in a very different manner than other architectures. Instead of trying to reduce latency for single-memory access, the MTA-2 tries to tolerate it by ensuring that the processor has other work to do while waiting for a memory request to be satisfied. Each processor can have a large number of outstanding memory requests—it has hardware support for many concurrent threads and switches between them in a single clock cycle. Thus, when a memory request is issued, the processor immediately switches its attention to another thread that's ready to execute. In this way, the processor tolerates latency and isn't stalled waiting for memory.

However, this execution model depends on the availability of numerous fine-grained, hardware-supported threads to keep the processor occupied. We can write many graph algorithms in a thread-rich style, but the likelihood of access contention increases with the number of threads. The MTA-2 addresses this problem by supporting word-level synchronization primitives—each word of memory can be locked independently, thus the locks have a minimal impact on the execution of other threads.

Another unusual feature of the MTA-2 is its support for fast and dynamic thread creation and destruction. The programmer needn't limit the program to a fixed degree of parallelism but can instead let the data determine the number of threads. The MTA-2 supports a virtualization of threads, which it then maps onto physical processors to facilitate adaptive parallelism and dynamic load balancing.

However, massively multithreaded machines also have significant drawbacks. Because the processors are custom and not commodity, they're more expensive and have a much slower clock rate than mainstream microprocessors—for instance, MTA-2 processors have a clock rate of only 220 MHz, more than an order of magnitude slower than state-of-the-art commodity microprocessors. Furthermore, the MTA-2's programming model, although simple and elegant, isn't portable to other parallel architectures.

Software

The different architectures we've discussed so far all have their own programming models: explicit message passing with MPI is the most portable and widely used paradigm, OpenMP is restricted to shared-memory machines but has some portability, and the MTA-2 programming model is unique to Cray's line of massively multithreaded machines. Naturally, these differences raise sig-

nificant impediments to cross-architectural comparisons. One way to alleviate such problems is to use generic programming libraries that hide machine-specific details.

Generic programming, for example, underlies the C++ Standard Template Library,⁵ the Boost C++ Libraries, and, in particular, the Boost Graph Library (BGL).⁶ This programming paradigm involves implementing concepts such as iterators, which use language features such as templates. Generic programming libraries are efficient as well: BGL algorithms, for example, implement the *visitor pattern*, a software methodology that lets programmers provide custom routines to be executed when predefined events occur. With the visitor pattern, we can implement BGL algorithms without worrying about low-level details—we can represent graphs with adjacency matrices, adjacency lists, or some other data structure, yet the same algorithm code will run on any of these.

The BGL's generic nature also makes it extensible in a high-performance computing context. BGL algorithms can run on any graph representation that exports a certain interface, thus they can run on distributed data structures that exploit cluster architectures and export this interface. The Parallel BGL (PBGL) provides a suite of such data structures.⁷ In its purest sense, the PBGL provides a way to run serial graph algorithms on very large problem instances that require large clusters' distributed memory for storage. Although researchers have implemented inherently parallel algorithms in the PBGL, barriers still exist to consistently achieving strong scaling of running time (running faster on the same problem instance when more processors are used) for graph algorithms on distributed-memory architectures.

To leverage the massively multithreaded architectures we described earlier, we extended a small subset of the BGL into the Multi-Threaded Graph Library (MTGL).⁸ This library retains the BGL's look and feel, yet encapsulates the use of nonstandard features such as compiler directives for parallelization and word-level synchronization operators. We still use the visitor pattern to give algorithm programmers entry points for custom computation. Although much of the architecture-specific detail is encapsulated in software abstractions, the custom routines provided must still be thread safe, which requires a higher level of programmer expertise.

Algorithmic Results

To compare graph algorithm implementations on different platforms, let's review some recent work (more details appear elsewhere⁹). We consider two

fundamental graph algorithms: $s - t$ connectivity and single-source shortest paths (SSSPs). In $s - t$ connectivity, the goal is to find a path from vertex s to vertex t that traverses the fewest possible edges. In SSSPs, each edge has a length, so the goal is to find the shortest-length path from a specific vertex to all other vertices in the graph.

Data

Graphs associated with physical simulations often have a structure induced from physical geometry—edges tend to connect vertices that are geometrically nearby. However, the growing field of informatics is characterized by data sets of relationships deduced by analyzing information rather than by modeling physical objects. A canonical example is the network of relationships between people in a population (social network). Stanley Milgram’s small-world experiment,¹⁰ which led to the “six degrees of separation” principle, found that by following just a few edges in a social network, we might end up anywhere. However, informatics data sets with this small-world property lack spatial locality and are therefore more challenging to map to distributed-memory parallel computers. Another common characteristic of informatics graphs is an inverse power law degree distribution—the vast majority of entities in these networks tend to be connected to just a few other entities, whereas a few “high degree” entities are connected to an enormous number of other entities.

Let’s focus here on two different, purely synthetic classes of graphs: Erdős-Rényi³ random graphs and a class of inverse power law graphs constructed by recursively adding adjacencies to a matrix in an intentionally uneven way (RMAT).¹¹ We can construct an Erdős-Rényi graph by assigning a uniform edge probability to each possible edge and then using a random-number generator to determine which edges exist. RMAT graph construction involves recursively partitioning an adjacency matrix and then unevenly assigning neighbor relationships. Unlike Erdős-Rényi graphs, RMAT graphs have an inverse power law degree distribution.

However, among the machines we’ll discuss shortly, only the MTA-2 has a programming model and architecture sufficiently robust to easily test instances of inverse power law graphs with close to a billion edges. Andy Yoo and his colleagues’ work¹² was limited to Erdős-Rényi graphs, and the PBGL’s current RMAT generator doesn’t scale to large instances. We know of no distributed-memory results for gigascale inverse power law graphs. Given this limitation, we only describe results for Erdős-Rényi

graphs and note that the MTA-2 performance on like-sized RMAT graphs is almost identical.

High-degree nodes are a challenge for distributed-memory machines for several reasons. A standard scientific computing practice for distributed-memory platforms is to store “ghost nodes” on each processor to represent the neighbors of all the processor’s graph vertices. With ghost nodes, vertices can traverse their neighbors, know which of them are stored remotely, and avoid some remote communication. However, this simple strategy doesn’t scale to large instances of graphs with inverse power law distributions because a single processor can’t be expected to store ghost nodes for high-degree nodes’ neighbors. As an alternative, Yoo and his colleagues avoided using ghost nodes, but in their approach, high-degree vertices require very large message buffers. Ghost nodes limit memory scalability and help runtime scalability, which creates a fundamental tension.

$s - t$ Connectivity

For $s - t$ connectivity, we can consider the following simple algorithm: given two vertices s and t , find s ’s neighbors and see if any of them is t . If not, then find t ’s neighbors and see if any of them is one of the vertices in s ’s expanding frontier. Repeat this process by expanding the smaller of the frontiers of s and t until the two frontiers intersect (see Figure 1). Yoo and his colleagues used an algorithm like this on Erdős-Rényi graphs of 3.2 billion nodes and reported their results from the 32,768 processors of the world’s largest distributed-memory machine, BlueGene/L. Notably, this implementation was memory efficient because it didn’t use ghost nodes. However, as we’ll discuss, this came at the cost of significantly reduced performance.

For a fixed-size problem, Yoo and his colleagues reported a speedup of roughly 65 on 450 processors. They also reported runtimes for a series of scaled problems in which the graph size grew with the number of processors. However, because the amount of work in $s - t$ connectivity grows less quickly than the graph’s size, the scalability assessment requires some care.

On an Erdős-Rényi graph, it’s straightforward enough to analyze the expected number of vertices to be visited to find the shortest path between s and t , but if we apply the algorithm in Figure 1 to the instances Yoo and his colleagues studied, the number of vertices visited should be roughly 177 times larger for the graph on 32,768 nodes than for the graph on one node. With the runtime growing by a factor of three, this suggests an overall speedup factor of approximately 60 times. Unfortunately,

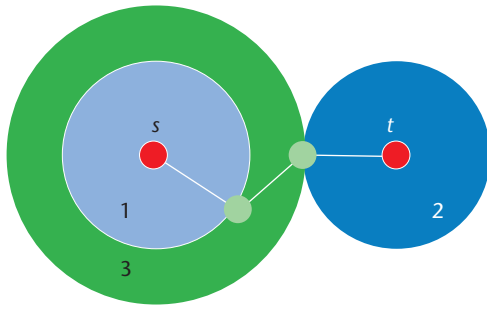


Figure 1. Simple connectivity algorithm. For two vertices s and t , (1) find s 's neighbors and see if t is one of them, (2) find t 's neighbors and see if s or one of its neighbors is one of them, or (3) alternate back to s and expand its frontier one more level.

Yoo and his colleagues didn't test their code on RMAT instances, in part because of concerns about message-buffer sizes.

However, David Bader and Kamesh Madduri¹³ implemented the same $s-t$ connectivity algorithm on the MTA-2 and achieved a speedup factor of roughly 28 on 40 processors for both Erdős-Rényi and RMAT instances. A simple counting argument based on the number of vertices touched during the $s-t$ connectivity algorithm suggests that the computation done by 32,768 processors on BlueGene/L could be done by five to 10 processors of an MTA-2 with sufficient memory.

Andrew Lumsdaine and his colleagues⁹ implemented the same algorithm in the PBGL and achieved excellent single-processor performance. They used compact data structures to improve cache utilization, resulting in single-processor performance comparable to that obtained by Yoo and his colleagues with BlueGene/L. However, Lumsdaine and his group weren't able to reduce runtime as processors were added, even with the use of ghost nodes. Fundamentally, there isn't much work to do in an $s-t$ connectivity algorithm: even if the graph is large, only a small subset of vertices must be visited for Erdős-Rényi graphs. Thus, it's difficult to outperform a fast serial algorithm.

Single-Source Shortest Paths

A fundamental problem in graph theory is that of finding SSSPs. Given a single starting vertex, SSSP algorithms compute a shortest path to each vertex in the graph, as Figure 2 shows. A classical algorithm by Edsger Dijkstra solves this problem by sequentially finding and "settling" the closest unsettled vertex to the source.¹⁴ This elegant algorithm is inherently serial, but several variations

of it attempt to find and exploit parallelism by identifying vertices that might be settled simultaneously. Such algorithms are highly sensitive to the type of graph processed, and some graph types, such as road networks, don't offer enough parallelism for these schemes to work well. However, in the case of Erdős-Rényi random graphs and RMAT graphs, researchers have obtained some positive results. Perhaps the most notable of these came from Madduri and his colleagues,¹⁵ who used an implementation of Ulrich Meyer and Peter Sanders' delta-stepping algorithm¹⁶ to find the SSSP on an RMAT graph of roughly 1 billion edges in approximately 10 seconds on a 40 processor MTA-2. The single MTA-2 processor time for the same instance was 371 seconds, yielding a parallel speedup factor of roughly 30 times.


Lumsdaine and his colleagues developed a PBGL version of the same algorithm on an Opteron cluster, for which they reported performance at roughly an order of magnitude slower than the MTA-2 performance. The Opterons in their experiment had 2.0-GHz clocks, and the MTA-2 processors clocked at 220 MHz, suggesting that the MTA-2 was roughly two orders more efficient than the Opterons for this problem. It's also worth noting that the researchers' PBGL code used ghost nodes, making it less memory efficient than the MTA-2 software.

However, unlike in the $s-t$ connectivity study, the PBGL implementation of delta stepping displayed excellent scalability, which speaks well for the PBGL's generic programming software model. The PBGL delta-stepping implementation required roughly one day of programming effort and one day of benchmarking effort; pre-PBGL distributed graph algorithm implementations of similar complexity often required orders of magnitude more development effort.

As combinatorial algorithms become increasingly important in science, engineering, and other applications, their distinctive computational requirements will grow in significance. We focused here on the challenges of graph algorithms, but we believe that many combinatorial (and other) algorithms will face similar challenges. Unlike most scientific computing kernels, graph algorithms exhibit complex memory access patterns and limited amounts of actual processing. Consequently, their performance is determined by the computer's ability to access memory, not by actual processor speed. Complex data dependencies and dynamic fine-

grained parallelism often result in poor parallel performance on traditional machines.

Although graphs might be an extreme case, we believe a broad trend exists in the scientific computing community toward increasingly complex and memory-limited simulations: unstructured grids involve much more complex memory access patterns than structured ones, adaptive grids are even more challenging and lead to dynamic parallelism, and multiphase and multiphysics simulations add an additional degree of dynamism. These complex calculations generally achieve a very low percentage of peak performance on a single processor and exhibit poor parallel scalability.

We believe our work with massively multithreaded machines suggests an alternative, with the potential to significantly improve the performance of challenging computations. Thanks to the continued forward march of Moore's law, current microprocessors have silicon to spare. We believe this space could be used to support massive multithreading, resulting in processors and parallel machines that are applicable to a much broader range of applications than current offerings. 

Acknowledgments

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the US Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AI85000. This work was funded under the Laboratory Directed Research and Development program.

References

1. T.A. El-Ghazawi, W.W. Carlson, and J.M. Draper, "UPC Language Specification, 1.1 ed.," May 2003; <http://upc.lbl.gov/docs/user>.
2. L. Dagum and R. Menon, "OpenMP: An Industry-Standard API for Shared-Memory Programming," *IEEE Computational Science and Eng.*, vol. 5, no. 1, 1998, pp. 46–55.
3. *IEEE Standard Portable Operating System Interface for Computer Environments*, IEEE Press, 1988.
4. W. Anderson et al., "Early Experiences with Scientific Programs on the Cray MTA-2," *Proc. Supercomputing 2003*, IEEE CS Press, 2003, pp. 46–58.
5. A. Stepanov and M. Lee, *The Standard Template Library*, tech. report 95-11 (R.1), Hewlett-Packard Labs., 1995.
6. J. Siek, L.-Q. Lee, and A. Lumsdaine, *The Boost Graph Library*, Addison-Wesley, 2002.
7. D. Gregor and A. Lumsdaine, "The Parallel BGL: A Generic Library for Distributed Graph Computations," *Proc. Workshop Parallel Object-Oriented Scientific Computing (POOSC)*, 2005, www.osl.iu.edu/publications/prints/2005/Gregor:POOSC:2005.pdf.
8. J.W. Berry et al., "Software and Algorithms for Graph Queries on Multithreaded Architectures," *Proc. 21st Int'l Parallel and Distributed Processing Symp.*, IEEE Press, 2007, p. 495.
9. A. Lumsdaine et al., "Challenges in Parallel Graph Processing," *Parallel Processing Letters*, vol. 17, no. 1, 2007, pp. 5–20.

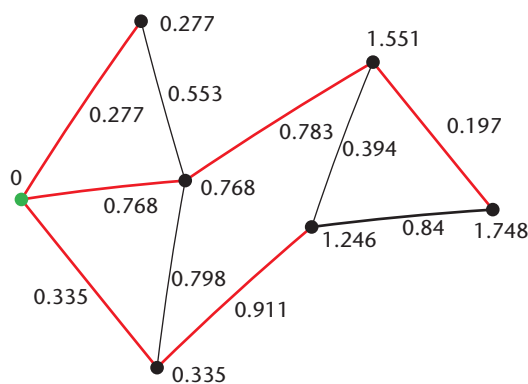


Figure 2. Calling an algorithm for a single-source shortest path (SSSP). The vertices are labeled with their distance from the single source, and the edges are labeled with their lengths. The red edges form an SSSP tree.

10. S. Milgram, "The Small World Phenomenon," *Psychology Today*, vol. 1, 1967, pp. 61–67.
11. D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A Recursive Model for Graph Mining," *Proc. 4th SIAM Int'l Conf. Data Mining*, SIAM Press, 2004.
12. A. Yoo et al., "A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L," *Proc. Supercomputing 2005*, IEEE CS Press, 2005, p. 25.
13. D. Bader and K. Madduri, "Designing Multithreaded Algorithms for Breadth-First Search and ST-Connectivity on the Cray MTA-2," *Proc. 35th Int'l Conf. Parallel Processing (ICPP)*, IEEE CS Press, 2006, pp. 523–530.
14. E. Dijkstra, "A Note on Two Problems in Connection with Graphs," *Numerische Mathematik*, vol. 1, 1959, pp. 269–271.
15. K. Madduri et al., "Parallel Shortest Path Algorithms for Solving Large-Scale Instances," *Proc. 9th DIMACS Implementation Challenge: Shortest Paths*, 2006; <http://dimacs.rutgers.edu/Workshops/Challenge9/papers/madduri.pdf>.
16. U. Meyer and P. Sanders, "Delta-Stepping: A Parallel Single Source Shortest Path Algorithm," *Proc. 6th Ann. European Symp. Algorithms*, Springer-Verlag, 1998, pp. 393–404.

Bruce Hendrickson is a Distinguished Member of Technical Staff in the Informatics and Computer Science Department at Sandia National Laboratories, and an Affiliated Professor of Computer Science at the University of New Mexico. His research interests include combinatorial scientific computing, graph algorithms, and parallel computing. Hendrickson has a PhD in computer science from Cornell University. Contact him at bahendr@sandia.gov.

Jonathan W. Berry is a Principal Member of Technical Staff in the Scalable Algorithms Department at Sandia National Laboratories. His research interests include graph algorithms and software, as well as combinatorial optimization. Berry has a PhD in computer science from Rensselaer Polytechnic Institute. Contact him at jberry@sandia.gov.