Parallel Processing Letters, Vol. 17, No. 1 (2007) 5–20

World Scientific
www.worldscientific.com

CHALLENGES IN PARALLEL GRAPH PROCESSING

ANDREW LUMSDAINE and DOUGLAS GREGOR
Indiana University
Bloomington, Indiana 47401 USA

and

BRUCE HENDRICKSON and JONATHAN BERRY
Sandia National Laboratories
Albuquerque, New Mexico 87185 USA

Received December 2006
Revised January 2007
Communicated by Guest Editors

ABSTRACT

Graph algorithms are becoming increasingly important for solving many problems in scientific computing, data mining and other domains. As these problems grow in scale, parallel computing resources are required to meet their computational and memory requirements. Unfortunately, the algorithms, software, and hardware that have worked well for developing mainstream parallel scientific applications are not necessarily effective for large-scale graph problems. In this paper we present the inter-relationships between graph problems, software, and parallel hardware in the current state of the art and discuss how those issues present inherent challenges in solving large-scale graph problems. The range of these challenges suggests a research agenda for the development of scalable high-performance software for graph problems.

Keywords: Parallel architectures, graph algorithms, graph theory, multithreading, distributed memory, shortest paths

1. Introduction

Graphs provide a very flexible abstraction for describing relationships between discrete objects. Many practical problems in scientific computing, data analysis and other areas can be modeled in their essential form by graphs and solved with appropriate graph algorithms. As graph problems grow larger in scale and more ambitious in their complexity, they easily outgrow the computation and memory capacities of single processors. Given the success of parallel computing in many areas of scientific computing, parallel processing appears to be necessary to overcome the resource limitations of single processors in graph computations.

Applications perform and scale well when the overall computational problem-solving approach is well balanced—that is, when the problem to be solved, the algorithm used to solve it, the software used to express the algorithm and the hard-

ware on which the software is run are all well-matched to each other. To a large extent, the success of parallel scientific computing has been due to the fact that these aspects are well-matched for typical scientific applications. Common idioms for solving typical problems in scientific domains (which tend to involve solving systems of partial differential equations) have evolved and become standard practice in the scientific computing community. Similarly, the hardware platforms and programming models that work well for typical problems have also become commonplace. Machine rooms around the world contain commodity clusters running codes programmed with MPI.

Unfortunately, the algorithms, software, and hardware that have worked well for developing mainstream parallel scientific applications are not necessarily effective for large-scale graph problems. Graph problems have some inherent characteristics that make them poorly matched to current computational problem-solving approaches. In particular, the following properties of graph problems present significant challenges for efficient parallelism.

- **Data-driven computations.** Graph computations are often completely data-driven. The computations performed by a graph algorithm are dictated by the vertex and edge (node and link) structure of the graph on which it is operating rather than being directly expressed in code. As a result, parallelism based on partitioning of computation can be difficult to express because the structure of computations in the algorithm is not known *a priori.*

- **Unstructured problems.** The data in graph problems are typically unstructured and highly irregular. Similar to the difficulties encountered in parallelizing a graph problem based on its computational structure, the irregular structure of graph data makes it difficult to extract parallelism by partitioning the problem data. Scalability can be quite limited by unbalanced computational loads resulting from poorly partitioned data.

- **Poor locality.** Because graphs represent the relationships between entities and because these relationships may be irregular and unstructured, the computations and data access patterns tend not to have very much locality. This is particularly true for graphs that come from data analysis. Performance in contemporary processors is predicated upon exploiting locality. Thus, high performance can be hard to obtain for graph algorithms, even on serial machines.

- **High data access to computation ratio.** Graph algorithms are often based on exploring the structure of a graph in preference to performing large numbers of computations on the graph data. As a result, there is a higher ratio of data access to computation than for scientific computing applications. Since these accesses tend to have a low amount of exploitable locality, runtime can be dominated by the wait for memory fetches.

In this paper we discuss, from several perspectives, the impediments to high performance graph implementations, and some of their possible solutions. In the next section we briefly survey the landscape of high performance architectures. In §3 we

discuss how these different architectures map onto the needs of graph algorithms. In §4 we review some ongoing activities to develop parallel graph software. Then in §5 we report on some performance results on different architectures. We then discuss some possible architectural trends that may impact graph algorithm performance in §6, and suggest some areas for further work in §7.

2. Parallel Architectures & Programming Models

An assortment of different parallel computers with varying capabilities are commercially available. Most machines are built out of traditional microprocessors, with several layers of hierarchical memory. Fast memory is used to store values from memory addresses that are likely to be accessed soon, thereby reducing latency. For many applications, this is a very effective way to improve performance, but as we will argue in §3, it is not particularly effective for operations on unstructured graphs. These applications tend to be highly latency dominated and often obtain extremely low utilization on traditional microprocessors [16,17].

2.1. Distributed memory machines

The most widespread class of parallel machines are *distributed memory computers*. These machines are usually made predominantly of commodity parts, consisting merely of a set of processors and memory connected by some high speed network. The commodity nature of these machines makes them inexpensive, and they are very effective on many scientific problems and on problems that are trivially parallel.

Distributed memory machines are most commonly programmed by explicit message passing. In this programming model, the user is responsible for dividing the data up among the memories of the different processors and for determining which processor performs which tasks. Typically, an *owner-computes* model is used in which the processor that owns data also performs any operations on that data. Data are exchanged between processors by user-controlled messages, usually with the MPI communication library [14]. As the details of the computation and communication are almost completely in the user's hands, many applications can achieve high performance this way. However, the detailed control of data partitioning and communication can be tedious and error prone.

In addition to making use of a memory hierarchy, message passing programs often reduce latency by amortization. Typically, programs are written in a Bulk-Synchronous style, in which processors alternate between working independently on local data, and participating in collective communication operations [20]. By grouping data exchanges into large, collective operations, the overall latency cost is substantially reduced. However, this comes at the expense of algorithmic flexibility. Data cannot be transmitted on demand, but only at the pauses between computational steps. This can be deleterious for load balancing, and makes it difficult to exploit fine-grained parallelism in an application.

In principle, message passing programs need not be bulk synchronous. Asynchronous messages can be interleaved with computation in an arbitrary manner. However, most versions of MPI are optimized for two-sided communication in which sends and receives must be paired (i.e., it is not possible to write to or read from

another processor's memory without the program on that processor being involved). Without more efficient support for one-sided communication, distributed memory machines are not well suited to fine-grained parallelism.

2.2. Partitioned global address space computing

MPI is not the only way to program distributed memory parallel computers. An important alternative, that is better suited to fine-grained parallelism, is to use a *partitioned global address space* language such as by UPC [6]. 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 facilitates writing programs with complex data access patterns. UPC sits on top of a communication layer that supports one-sided communication. Thus, fine-grained parallel programs are easier to write and can achieve higher performance than with MPI. However, as with MPI, in a UPC program the number of threads of control is constant, generally equal to the number of processors. As we argue below, the lack of dynamic threads is a significant impediment to the development of high performing graph software.

2.3. Shared-memory computers

UPC provides a software illusion of globally addressable memory on distributed memory hardware. Support for a global address space can also be provided in hardware. Such *shared memory* computers can be categorized in various ways. Here we consider cache-coherent machines and massively multithreaded machines.

2.3.1. Cache-coherent parallel computers

In symmetric multiprocessors (SMPs), global memory is universally accessible by each processor. UPC can be used to program these machines, but the most common approach is OpenMP [5], or a threading approach like POSIX threads [18]. The key feature of an SMP is that it provides hardware support for access to addresses in global memory, so any address in the machine can be retrieved directly and relatively quickly. As a result, highly unstructured problems may achieve higher performance than is possible on distributed memory machines. Thus, SMPs essentially address the latency challenge with faster hardware for accessing memory. However, SMPs have some inherent performance limitations.

As discussed above, most processors have a memory hierarchy in which a small amount of data is kept in faster memory, or *cache*, for quick access. When a datum is updated, the new value may not be immediately propagated to main memory, but may instead reside only in cache for a while. When only a single processor is involved, it is not difficult to ensure that a read operation gets the most current value. But in a multiprocessor machine with multiple caches, the cache-coherence problem is a significant challenge. There are a variety of methods to address this problem, each with its advantages and disadvantages. However, each such approach adds overhead which can degrade performance. Even for problems in which reads are much more prevalent than writes, cache coherence protocols have an impact on scalability.

A second performance challenge in SMPs is the protocol for thread synchronization and scheduling. If several threads are trying to access the same region of memory, the system must apply some protocol to ensure correct program execution. As a result, some threads may 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. A more dynamic threading model may appear in future versions of OpenMP.

2.3.2. Massively multithreaded architectures

The massively multithreaded Cray MTA-2 [2] addresses the latency challenge in a substantially different manner than other architectures. Instead of trying to reduce the latency of single memory access, the MTA-2 tries to *tolerate* latency by ensuring that a 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. The processor has hardware support for many concurrent threads and is able to switch between them in a single clock cycle. Thus, when a memory request is issued, the processor can immediately switch its attention to another thread whose memory request has arrived. In this way, the processor tolerates latency and is not stalled waiting for memory, given sufficient concurrency.

This execution model depends upon the availability of a large number of threads to keep the processor occupied. As we will discuss in §3, many graph algorithms can be written in a thread-rich style. However, with a large number of threads, the likelihood of access contention increases. The MTA-2 addresses this problem by supporting word-level synchronization primitives. Each word of memory can be locked independently. Thus, 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 need not 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 thread processing elements called streams. This facilitates adaptive parallelism and dynamic load balancing.

Although conceptually attractive, massively multithreaded machines also have significant drawbacks. Because the processors are custom and not commodity, they are more expensive and have a much slower clock than mainstream microprocessors. Furthermore, the programming model of the MTA-2, while simple and elegant, is non-standard. The combination of high cost, non-standard programming model, and lower peak performance numbers for compute-bound computations have apparently precluded commercial success for the MTA-2. However, the machine's unique ability to handle large-scale graph informatics problems has given rise to a successor: the XMT (formerly called Eldorado) [7]. This new architecture combines MTA processors with commodity networks augmented to support shared memory. The predicted scalability of graph codes on the XMT is discussed in [19], which provides evidence that MTA-like scaling may be possible through at least 512 processors. The XMT can be built with up to 8192 processors. Cray also intends to include massive multithreading in its heterogeneous architectures of the future.

3. Mapping Parallel Graph Algorithms to Hardware

Although numerous theoretical papers have been written on parallel graph algorithms, many using the PRAM model [12], there are far fewer papers describing actual implementations or libraries. High performance parallel graph algorithms have proven to be difficult to develop, and a number of hardware and software related challenges must be addressed.

- **Task granularity.** A fundamental question in designing a graph algorithm is where to introduce parallelism. For some algorithms, there is a significant degree of coarse-grained parallelism. For example, some centrality computations involve solving many shortest path problems. Each shortest path problem could be a separate task. However for most graph operations, particularly those with linear or near linear runtime, parallelism can only be found on a more fine-grained level. Machines that facilitate fine-grained parallelism will be better suited to running such algorithms.

- **Memory contention.** In systems which support a global address space, multiple processes or threads can try to simultaneously access the same memory. This can significantly reduce performance. The importance of this problem grows with increasing degrees of parallelism and is therefore most acute for multithreaded systems. Conveniently, for many applications in informatics, the graph can be considered to be a read-only object, and read-only contention is less problematic. A graph query algorithm will not write to the graph itself, but it will allocate and write to its own local data structures and contention must be carefully handled for these operations.

- **Load balancing.** The vertices being visited in a graph algorithm may have some spatial locality in global memory. In an owner-computes model, this means that some processors will have more work to do than others. Methods for reassigning this work may improve performance. Note that this issue is less pronounced on shared-memory machines because work can be migrated without explicitly moving data. Temporal aspects of load balance can also be important. A single graph algorithm might have a time-varying amount of work to do. For example, in breadth-first search, early stages may have few vertices to visit, while later stages may have significantly more. Dynamic thread creation and assignment is one mechanism for dealing with varying workloads.

- **Simultaneous queries.** In some data mining applications, a large graph may be used by a team of analysts, and each may be submitting queries at any time. This provides the potential for an additional degree of parallelism. In such a setting, a system that gracefully and efficiently handles streams of queries is highly preferable to one that can only respond to a single query at a time. In contrast, virtually all of the literature is concerned with the time required to perform a single graph algorithm. A better model would focus on throughput, rewarding systems that can work on multiple queries simultaneously.

- **Software development.** Solutions to many of the challenges alluded to above can, and should, be encapsulated within a carefully architected software framework. Software design issues will be addressed more completely in §4, but flexibility, extensibility, portability and maintainability are all important considerations. As informatics applications often involve iterative algorithmic exploration, the software should also support rapid prototyping.

In the following sub-sections we discuss these issues with respect to mapping graph algorithms to particular classes of parallel hardware.

3.1. Distributed-memory architectures

For many of the issues related to parallel graph algorithms, distributed-memory, message-passing machines have problems. They are the least amenable to fine-grained parallelism, and the most complex platform on which to perform dynamic load balancing. They are also the most challenging architectures upon which to build a simultaneous query processing system. On the other hand, the principal appeal of message passing machines is that they are ubiquitous and comparatively inexpensive. Code written in MPI will run on almost all parallel platforms.

Distributed memory requires that the edges and vertices of a graph be partitioned among processors. If a processor owns a vertex, it needs to have a mechanism for finding that vertex's neighbors. One method for doing this that is widely employed in scientific applications is to keep a local data structure with information about all the vertices that are adjacent to the vertices local to a process. Keeping information about these *ghost vertices* works well for applications in which the graph can be partitioned in such a way that few edges cross between processors, and such graphs predominate in scientific applications. However, for less structured graphs like those common in informatics applications, the set of adjacent vertices can be much larger than the number of local vertices. Thus, the use of ghost vertices can have a seriously detrimental impact upon memory usage. Even with ghost vertices, the total amount of memory required should be proportional to the number of edges in the graph. But for a fixed amount of space, a shared memory approach will allow for the storage of a larger graph than distributed memory due to the (potentially quite large) storage impact of ghost vertices. In addition, high-degree vertices cause problems in distributed memory, as they may overwhelm the memory available on a single processor.

An alternative to ghost cells is to use a hashing scheme to assign vertices to processors. Although hashing can result in memory savings compared to ghost cells, it can incur significant computational overhead.

3.2. Partitioned global address space computing

Partitioned global address space languages provide a global address space computing model while running on inexpensive, distributed memory hardware. The global address space obviates the need for ghost cells, and facilitates finer grained parallelism and dynamic load balancing. Latencies are reduced by one-sided communication.

However, this approach is not a panacea. Performance can be very sensitive to data layout, since the graph remains partitioned and non-local accesses are expensive. Also, UPC remains a boutique language with limited support and portability.

3.3. Cache-coherent, shared-memory computers

SMPs retain the advantages of partitioned global address space computing, but since they provide hardware support for global accesses, they have lower latencies. However, with only a single application thread per processor and complex memory access patterns, the processor will frequently stall while waiting for memory. The highly unstructured access patterns associated with complex graphs means that cache is of limited value, yet the cost of cache-coherence cannot be avoided. This problem is partially mitigated if the graph is read-only. Nonetheless, contention-prone accesses to local data structures need to be handled carefully. SMPs lack the word-level synchronization primitive provided by massively multithreaded machines, so scalability limits due to contention will be more substantial. Economically speaking, SMPs are significantly more expensive than distributed memory machines on a per-processor basis.

3.4. Massively multithreaded machines

Massively multithreaded machines are conceptually attractive for graph algorithms for a variety of reasons. They can support both coarse and fine-grained parallelism. They facilitate light-weight load-balancing and simultaneous queries. They provide a global address space without the complexity and performance cost of cache-coherence. The ability to dynamically adjust the degree of parallelism is very convenient for expressing algorithms, particularly for applications like graph algorithms in which the parallelism is data dependent and difficult to predict prior to runtime.

As with all the other global address space approaches, care must be taken to ensure that algorithms are written in a thread-safe manner. The principal additional challenge of massively multithreaded algorithms is that the number of threads can be much larger than the number of processors, and so memory contention issues are more significant. As a result, careful attention to algorithms and data structures is required.

Since current massively multithreaded machines are not mainstream, they are comparatively expensive and have slower clock speeds than machines built with commodity processors. As we discuss in §6, there is some possibility that this will change in the future, but for now the uncertain future of this architecture makes any commitment to massively multithreaded software risky.

4. Software Approaches

4.1. Parallel Boost Graph Library

The Parallel Boost Graph Library (Parallel BGL) [10,9] is a library of parallel graph algorithms and accompanying data structures. Designed using the Generic

Programming paradigm, the Parallel BGL separates the implementation of parallel algorithms from the underlying data structures and communication mechanisms. Thus, the implementation of a given *generic* algorithm within the Parallel BGL—say, a breadth-first search—can operate on any graph data structure and communicate over any communication medium so long as the data structure and medium meet certain essential abstract requirements.

By abstracting away the reliance on a particular communication medium, the same algorithm in the Parallel BGL can execute on distributed-memory clusters using MPI (relying on message passing for communication) or SMPs using Pthreads (relying on shared memory and locking for communication). Thus, architecture-specific optimizations (such as the introduction of lock-free data structures or MPI one-sided communication) can be separated from the algorithm description, greatly easing the process of porting to and tuning for a new parallel-computer architecture.

Generic algorithms in the Parallel BGL place as few requirements on the communication infrastructure as possible, to permit maximum reusability. However, there exist certain algorithms for which the optimal implementations vary widely from one architecture to the next. In these cases, multiple algorithm implementations may be required to account for radical differences in architecture, such as the distinction between course-grained parallelism that performs well on clusters and some SMPs and fine-grained parallelism that performs well on massively multi-threaded architectures like the MTA-2. The vast majority of the development effort for the Parallel BGL has focused on course-grained parallelism using MPI, although in the future we intend to branch out to other models of parallelism.

4.2. Multi-Threaded Graph Library

The simple programming model of the MTA-2 and XMT machines is well-suited for the Generic Programming paradigm in principle. However, software library development efforts on these platforms are in their infancy. For example, the C++ Standard Template Library (STL) is available, but it runs *in serial*. Massively multithreaded codes are only useful with sufficient parallelism. Thread-safe versions of STL and its Boost extensions, which would provide the foundation for a Multithreaded Boost Graph Library, are future work.

The *MultiThreaded Graph Library* [4], inspired by the serial Boost Graph Library, is being developed at Sandia National Laboratories to provide a near-term generic programming capability for implementing graph algorithms on massively multithreaded machines. Like the Parallel BGL, underlying data structures are leveraged to abstract parallelism away from the programmer. However, unlike the Parallel BGL, the primary algorithm design methodology is not to provide parallelized data structures for serial graph algorithms. The key to performance on MTA/XMT machines is keeping processors busy, and in practice this often reduces to performing many communicating, asynchronous, fine-grained tasks concurrently. The MTGL provides a flexible engine to control this style of parallelism.

The MTGL was developed to facilitate data mining on semantic graphs, i.e., graphs with vertex and edge types. Semantic graph computations involve extensive use of filters during graph searches. The MTGL provides mechanisms for these filtering operations. Furthermore, the XMT usage model will allow many users to

run algorithms concurrently on the same graph. The MTGL is designed to support this usage model.

5. Performance

Preliminary performance evaluations for parallel graph algorithms on various architectures illustrate some of the trade-offs in the design and implementation of parallel graph algorithms. We consider two similar graph problems: unweighted $s-t$ shortest paths, which determines the shortest path from an arbitrary vertex s to another vertex t in an unweighted graph, and single-source shortest paths, which determines the shortest path from an arbitrary vertex s to every other vertex in a graph with arbitrary, non-negative edge weights.

We report results for distributed memory runs of the Parallel Boost Graph Library [9] implementation on the Odin cluster at Indiana University, a 128-node cluster where each node contained two 2.0GHz AMD Opteron processors connected via Infiniband. The Parallel BGL tests were compiled with the Intel compiler version 9.0 with flags `-O3 -xW -tpp7 -i_dynamic -fno-alias`, the Parallel BGL 0.6.0 beta, and Open MPI 1.2 beta 3 [8].

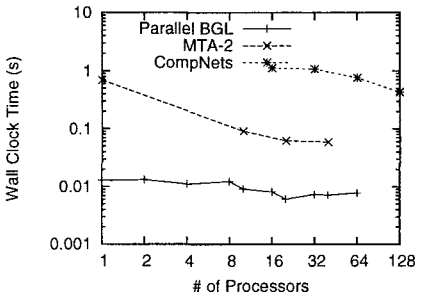
Our MTA-2 implementation are the C codes of [13] (not MTGL), and our results were gathered on a 40-processor MTA-2 system with 160GB uniform shared memory, where each processor has a clock speed of 220Mhz. The MTA-2 benchmarks were compiled with the MTA-2 C compiler (Cray Programming Environment 2.0.3) with the flag `-par`.

5.1. $s-t$ Shortest Paths

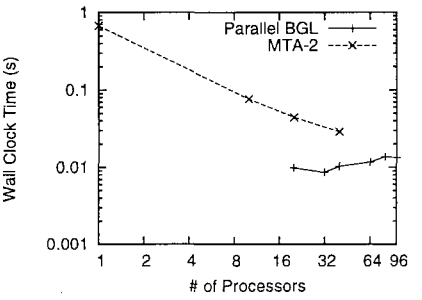
The $s-t$ shortest paths problem is essentially a bidirectional breadth-first search, with one search originating at the source and the other at the target. When the two searches collide at a vertex, a path from s to t has been found. Figure 1 illustrates the performance of the two different implementations of this algorithm on Erdös-Renyi random graphs of two different sizes. The MTA-2 implementation is described in [3]. In addition, we present results from CompNets [21], another distributed-memory parallel implementation of $s-t$ shortest paths using MPI. The CompNets results were gathered on a 128-node cluster, where each node contained two 3.06GHz Xeon processors connected by Myrinet.

Figure 1a shows the performance of the three implementations on random graphs with 2^{25} vertices and 2^{28} edges. CompNets and the MTA-2 show some scaling, but with these relatively small graphs, none of the implementations scale particularly well. Figure 1b shows performance results for much larger graphs, with 2^{27} vertices and 2^{30} edges. On graphs of this size, the Parallel BGL requires at least 20 compute nodes to store the graph and algorithm-specific data structures, while the shared-memory MTA-2 can store and process the graph for any number of processors. The MTA-2 shows excellent scalability on this problem, achieving a speedup of 25 on 40 processors. The Parallel BGL, while solving the same problem two to three times faster than the MTA-2, begins scaling inversely after 32 processors.

The $s-t$ shortest paths problem has somewhat unique performance characteristics because a typical $s-t$ path query only touches a small portion of the



(a) 2^{25} vertices, 2^{28} edges



(b) 2^{27} vertices, 2^{30} edges

Figure 1: Performance of the $s-t$ shortest paths implementations for distributed memory and the MTA-2 on random graphs.

graph. With the graphs in Figure 1, the queries only touched about 80,000 vertices ($< 0.06\%$ of the graph). Thus, despite the large graph size, there is only a small amount of work being performed by each processor. The good raw performance of the Parallel BGL implementation was assisted by improved memory locality due to a compressed representation of the primary bookkeeping data structure. This improves microprocessor utilization relative to a naïve implementation.

However, the small number of touched vertices in this problem negatively impacts the scalability of the Parallel BGL implementation. As additional processors are introduced, the time spent managing ghost cells and synchronizing communication among the processors increases. As a result, this overhead becomes comparable to the actual computation of interest because the local computation is so small. On the other hand, the MTA-2's use of fine-grained parallelism ensures that the workload is balanced among all of the MTA-2's processors. Thus, even when only a small portion of the graph is accessed, the MTA-2 shows excellent scalability.

Although both CompNets and the Parallel BGL are MPI-based implementations of $s-t$ shortest paths, their approaches to solving graph problems are quite different. CompNets pays careful attention to memory scalability: the size of communication buffers remains constant even as the graph size grows, and CompNets does not use ghost cells, for which the memory requirements would typically increase both with the graph size and the number of processors. By focusing on memory scalability, Yoo et al. were able to solve the $s-t$ shortest paths problem on a graph with $4B$ vertices and $20B$ edges in 1.5 seconds using 32,768 processors of Blue Gene/L [21]. The Parallel BGL, on the other hand, makes heavy use of ghost cells, which can drastically improve performance: on 100 nodes of the Odin cluster, the Parallel BGL solved the $s-t$ shortest paths problem for the same graph in 0.43 seconds. Figure 1a also shows how ghost cells result in improved performance on small graphs. The use of ghost cells comes at a cost, however: the amount of memory required for a given computation depends heavily on the structure and size of the graph itself, and may easily grow large enough to exhaust local memory. There are many trade-offs in the design of distributed-memory parallel graph algorithms, with CompNets and

the Parallel BGL having taken different stances regarding memory scalability and the use of ghost cells.

5.2. Single-Source Shortest Paths

We report on two different parallel implementations of the Δ-stepping algorithm [15] for single-source shortest paths with non-negative edge weights. Δ-stepping is a variant of Dijkstra's algorithm, wherein the priority queue of Dijkstra's algorithm is replaced with an approximate bucket structure. The bucket structure is an array of buckets in which the i^{th} bucket contains all vertices whose tentative distance from the source is in the range $[i\Delta, (i+1)\Delta)$, where Δ is a positive real number. Parallelization of the Δ-stepping algorithm involves relaxing outgoing edges from all of the vertices in the current bucket simultaneously (potentially causing re-insertions into that bucket). Edges are classified into *light* edges (those with weights $\leq \Delta$) and *heavy* edges (those with weights $> \Delta$). By relaxing the light edges (those that can cause re-insertion) first, the algorithm ensures that the work of relaxing heavy edges need only occur once.

Figure 2 illustrates the performance of the MTA-2 and Parallel BGL implementations of Δ-stepping on a random graph with $n = 2^{28}$ vertices and $m = 2^{30}$ edges. The MTA-2 implementation of Δ-stepping is described in [13]. The Parallel BGL implementation is a direct translation of the distributed-memory formulation of the Δ-stepping algorithm [15]. Following the experimental study on Δ values for the MTA-2 implementation in [13], both implementations normalize edge weights in $[0, 1)$ and let $\Delta = n/m = 0.25$.

Both implementations of Δ-stepping scale extremely well on this problem. For equivalent performance, the Opteron implementation requires about 10 times as many processors as the MTA-2 implementation. This result, combined with the disparity in clock rates between the MTA-2 and Opteron processors, illustrates that the MTA-2 is achieving a much higher processor utilization than the Opterons, owing to its support for latency tolerance. On the other hand, while clusters with thousands of processors are becoming commonplace, the largest MTA-2 ever built contains only 40 processors (although future XMT machines may be significantly larger).

Figure 3 illustrates the performance of the MTA-2 implementation of Δ-stepping on two different graph classes, Erdös-Renyi random graphs and scale-free graphs, which have significantly different properties. Both graphs have 2^{28} vertices and 2^{30} edges. While the edges in a random graph tend to be evenly distributed throughout, the distribution of edges in a scale-free graph follows a power law, e.g., few vertices have a very high degree whereas the vast majority of vertices have a very low degree. With distributed-memory implementations, these imbalances have an impact both on performance (relaxing edges for high-degree vertices requires much more work than for low-degree vertices) and on the ability to solve certain problems (since memory utilization can vary widely from one compute node to another). With the fine-grained parallelism and shared address space of the MTA-2, the performance of the Δ-stepping algorithm is nearly identical from one graph class to another.

fine-grained parallelism and shared address space of the MTA-2, the performance of the Δ-stepping algorithm is nearly identical from one graph class to another.

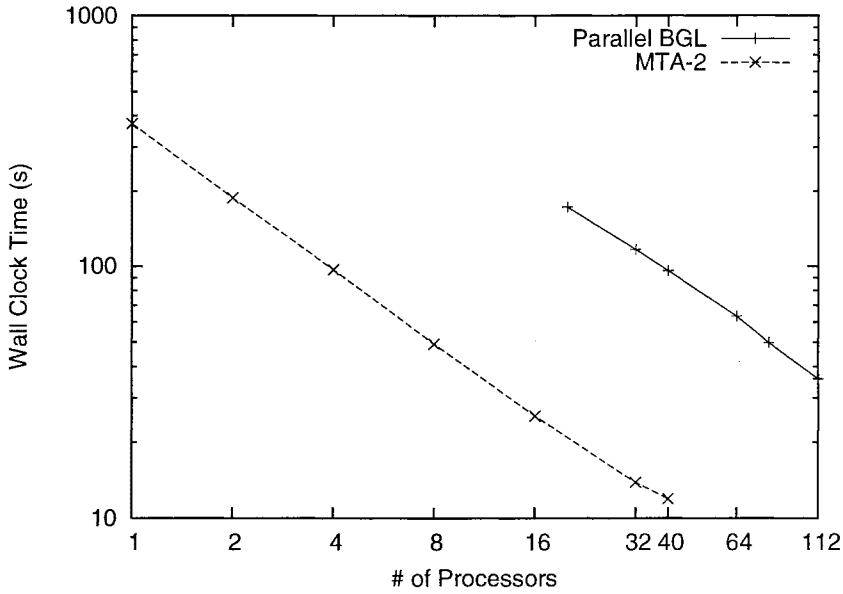


Figure 2: Performance of the Δ-stepping implementations for distributed memory and the MTA-2 on random graphs with 2^{28} vertices and 2^{30} edges.

6. Looking Forward

Several hardware trends can be discerned that will likely have an impact upon high performance graph capabilities in the near future.

The Cray XMT (formerly called Eldorado) computer is a follow-on to the MTA-2 [7]. The introduction of the XMT may make massively multithreaded architectures more widely available for further study. The XMT processors are similar to their MTA-2 counterparts, but with a 500MHz clock rate. Although the XMT network will be lower performing than that of the MTA-2, its 3D mesh/torus interconnect will allow for the construction of larger machines. Bandwidth will be reduced, latency will increase, and with a poorer communication to computation ratio, it will be harder to achieve peak performance on the XMT than on the MTA-2. However, the XMT is far more economical that the MTA-2, and its ability to scale to 8000 processors and 128TB of memory makes it feasible for very large data. Unlike the MTA-2, an XMT processor can access local memory more quickly than remote memory. Even without taking advantage of this opportunity to exploit locality, performance simulations for the XMT indicate that a 512-processor XMT should be around 50% efficient on memory-intensive graph operations [4]. Locality management techniques borrowed from current distributed-memory approaches should help to achieve better efficiencies and to scale up to the XMT's full complement of processors. In addition, Cray's plans for future heterogeneous machines with multithreaded processors will make massive multithreading a more mainstream technology for the high performance computing community.

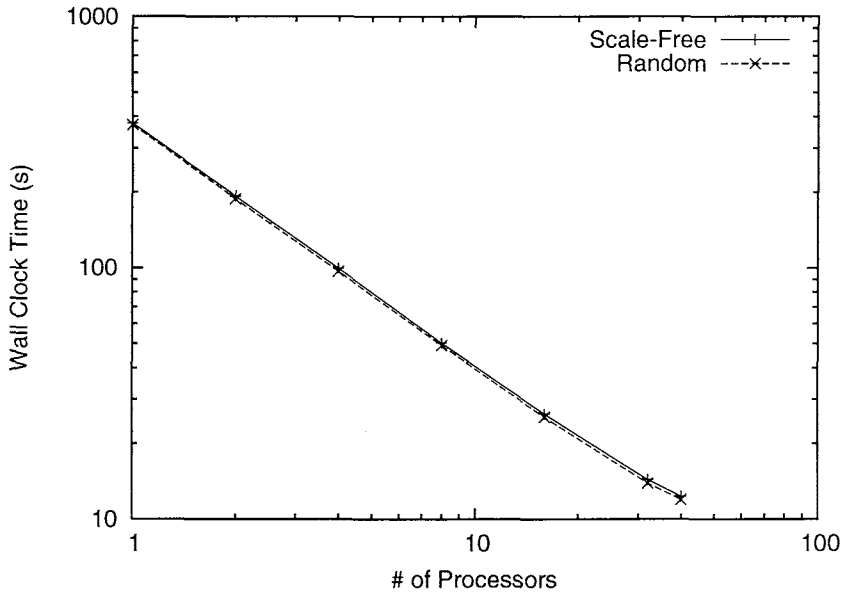


Figure 3: : Performance of the MTA-2 implementation of Δ-stepping on scale-free and random graphs with 2^{28} vertices and 2^{30} edges.

A broader trend in the architecture community is the coming ubiquity of multicore processors. Two and four core processors are common now, and many more, perhaps hundreds, will likely be available in the future. The impact of multicore technology will depend upon the manner in which certain technical issues are resolved. Bandwidth and latency for memory accesses will continue to be important factors in graph algorithm performance. Multicore chips may also motivate greater interest in multithreaded technology, which should allow techniques from the MTA to be leveraged to improve the efficiency of graph computations on all platforms.

In distributed-memory architectures, one-sided communication protocols are becoming widely available in both hardware (e.g., Infiniband's RDMA support) and in software (e.g., various MPI implementations support the MPI-2 one-sided operations [11]), which may help reduce latency within distributed graph applications. Advances in distributed shared memory systems (e.g., [1]) may make it possible to implement efficient parallel graph algorithms in distributed memory with less effort than today's approaches based on explicit message passing programming.

Further ahead, processor-in-memory (PIM) technology may prove to be important for graph algorithms. With PIM, the processor and memory are manufactured together, so a processor can access near-by memory very quickly. PIM machines can be very fast if computations on memory are performed by nearby processors. In a PIM system, threads will travel to processors that are near the memory the thread uses. Since threads for graph operations tend to be small, and thread migration is an efficient, unidirectional process, PIM machines have the potential to reduce latency and bandwidth challenges for graph algorithms.

7. Conclusion

Large graphs are finding increasing applications in computational biology, unstructured mesh computations, and data analyses of many sorts. We foresee a growing need for high-performance solutions to graph problems. As we have argued, the computational requirements of graph algorithms are significantly different from those of most existing parallel applications, and these differences have consequences for architectures, software libraries and algorithm development.

Much research remains to be done in parallel graph algorithms and software. Although ubiquitous and inexpensive, distributed memory machines are a challenging platform for graph algorithms. Efforts like the Parallel BGL have shown that many graph algorithms can be parallelized on these platforms, but the processor utilization is often low. In distributed memory, a fundamental tension exists between using ghost cells for optimizing runtime and hashing for optimizing memory usage. New, perhaps hybrid, ideas are needed. One possibility would be to build upon the work of Yoo, et al. using matrix-based insights to parallelize breadth-first traversals [21]. Methods for unifying the expression of algorithms across different architectures would also be very beneficial. Further work on distributed memory algorithms and software is needed.

Massive multithreading looks to be an attractive approach for parallelizing graph operations, with the potential for excellent scalability and performance. But this work is really in its infancy and requires new algorithms and improved software. Furthermore, the commercial viability of MTA-like machines is uncertain, which adds risk to this approach.

Despite the significant challenges that graphs pose for parallel machines, we are generally optimistic about the future. The ever-increasing cost of memory access relative to processor speed means that more and more applications are becoming latency dominated. We expect this to result in architectural changes that will be advantageous to graph algorithms. Thanks to Moore's Law, processors have now become small enough that several of them can fit on a chip. We expect the sudden availability of silicon real estate to foster renewed research in methods to improve memory performance, including new forms of multithreading. We are hopeful that our experience with graph algorithms will help motivate the development of technologies that will be of benefit to a broad range of applications.

References

[1] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, 1996.

[2] W. Anderson, P. Briggs, C. S. Hellberg, D. W. Hess, A. Khokhlov, M. Lanzagorta, and R. Rosenberg. Early experiences with scientific programs on the Cray MTA-2. In *Proc. SC'03*, 2003.

[3] D. Bader and K. Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2. In *Proc. 35th Int'l Conf. on Parallel Processing (ICPP)*, Columbus, OH, August 2006. IEEE Computer Society.

[4] J. W. Berry, B. Hendrickson, S. Kahan, and P. Konecny. Graph software development and performance on the MTA-2 and Eldorado. In *Cray User's Group*, May 2006.

[5] L. Dagum and R. Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 05(1):46–55, 1998.

[6] T. A. El-Ghazawi, W. W. Carlson, and J. M. Draper. *UPC Language Specification*, 1.1 edition, May 2003. `http://www.gwu.edu/~upc/downloads/-upc_specs_1.1p2pre1.pdf`.

[7] J. Feo, D. Harper, S. Kahan, and P. Konecny. Eldorado. In *Proc. 2nd Conf. Computing Frontiers*, pages 28–34, 2005.

[8] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[9] D. Gregor, N. Edmonds, B. Barrett, and A. Lumsdaine. The Parallel Boost Graph Library. `http://www.osl.iu.edu/research/pbgl`, 2005.

[10] D. Gregor and A. Lumsdaine. The Parallel BGL: A generic library for distributed graph computations. In *Parallel Object-Oriented Scientific Computing (POOSC)*, July 2005.

[11] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, , and M. Snir. *MPI — The Complete Reference: Volume 2, the MPI-2 Extensions.* MIT Press, 1998.

[12] J. JáJá. *An introduction to parallel algorithms.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.

[13] K. Madduri, D. A. Bader, J. W. Berry, and J. Crobak. Parallel shortest path algorithms for solving large-scale instances. In *9th DIMACS Implementation Challenge: Shortest Paths*, November 2006.

[14] Message Passing Interface Forum. MPI: A Message Passing Interface. In *Proc. of Supercomputing '93*, pages 878–883. IEEE Computer Society Press, November 1993.

[15] U. Meyer and P. Sanders. Delta-stepping: A parallel single source shortest path algorithm. In *ESA '98: Proceedings of the 6th Annual European Symposium on Algorithms*, pages 393–404, London, UK, 1998. Springer-Verlag.

[16] R. C. Murphy, J. Berry, W. McLendon III, B. Hendrickson, D. Gregor, and A. Lumsdaine. DRS: A simple to write yet difficult to execute benchmark. In *Proc. IEEE Intl. Symp. Workload Characterizations*, October 2006.

[17] R. C. Murphy and P. M. Kogge. On the memory access patterns of supercomputer applications: Benchmark selection and its implications. *IEEE Trans. Computers*, 2007. To appear.

[18] *System Application Program Interface (API) [C Language].* Information technology—Portable Operating System Interface (POSIX). IEEE Press, Piscataway, NJ, 1990.

[19] K. Underwood, M. Vance, J. Berry, and B. Hendrickson. Analyzing the scalability of Eldorado. In *Proc. Cray User's Group*, May 2006.

[20] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[21] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek. A scalable distributed parallel breadth-first search algorithm on BlueGene/L. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 25, Washington, DC, USA, 2005. IEEE Computer Society.