Design of a Large-Scale Hybrid-Parallel Graph Library

Nicholas Edmonds,* Jeremiah Willcock, and Andrew Lumsdaine
Indiana University
150 S. Woodlawn Ave
Bloomington, IN 47405, USA
{ngedmond,jewillco,lums}@cs.indiana.edu

Torsten Hoefler
University of Illinois at
Urbana-Champaign
1205 W. Clark St.
Urbana, IL 61801, USA
htor@illinois.edu

Abstract

The focus of traditional scientific computing has been in solving large systems of PDEs (and the corresponding linear algebra problems that they induce). Hardware architectures, computer systems, and software platforms have evolved together to efficiently support solving these kinds of problems. Similar attention has not been devoted to solving large-scale graph problems. Recently this class of applications has seen increased attention. The irregular, non-local, and dynamic characteristics of these problems require new programming techniques to adapt them to modern HPC systems offering multiple levels of parallelism. We describe a library for implementing graph algorithms based on asynchronous execution of fine-grained, concurrent operations. Prototype implementations of two graph kernels which combine lightweight graph metadata transactions with generalized active messages demonstrate that it is possible to implement graph applications which efficiently leverage both shared- and distributed-memory parallelism.

1 Introduction

Graph applications are members of a new class of data-intensive applications within high-performance computing (HPC) which differ from traditional compute-intensive applications in a number of important ways. Traditional compute-intensive applications, for example, those based on discretized systems of PDEs, possess natural locality due to the local nature of the underlying operators. Coarse-grained approaches based on the BSP "compute-communicate" model thus tend to yield scalable solutions. The communication operations required are also coarse, involving only a small number of local peers.

In contrast, data-intensive problems such as graph applications possess no underlying natural locality which can be determined analytically. The locality information is **irregular**, and embedded directly in the data itself. This locality information describes a dependency graph for computations which is **non-local**, i.e. it does not have good separators [7]. To complicate matters further, data-intensive problems tend to be **fine-grained**, i.e. they possess a large number of small objects. Performing efficient static coalescing of these objects into larger, coarser-grained objects is hampered by the irregularity of the problems.

Efficiently mapping graph problems to traditional HPC hardware designed to solve coarse-grained, compute-intensive problems with good locality can (and has) been done by hand, albeit with great difficulty. These solutions required all layers of the solution stack to be developed by the application developer, and would need to be re-implemented from scratch for new platforms. By observing common techniques across a number of implementations we have designed a library which will allow the modular implementation of parallel graph algorithms that efficiently leverage both shared- and distributed-memory parallelism.

At the core of this new library is the ability to efficiently execute and manage asynchronous, independent, lightweight *handlers* across address spaces. These handlers are invoked remotely using a generalized form of Active Messages [9] described in detail in [10]. In order to allow for parallel execution of

*Student author

these handlers, lightweight memory transactions on non-contiguous graph metadata are supported.

2 Design

Many aspects of the new library described here are based on our experience with the Parallel Boost Graph Library (Parallel BGL) [3]. The intent is that this design will serve as the next generation of the Parallel BGL. This design focuses on two primary goals:

1. To support operating on very large-scale graphs on distributed-memory systems with thousands or tens of thousands of nodes.
2. To leverage parallelism at all levels, both across nodes and within them (i.e. via threads).

2.1 Handling Data Dependencies

Graph algorithms depend on three primary types of data: the graph data itself (vertices and edges) describing adjacency information, metadata describing the vertices and edges, and external data structures used for control flow (such as queues, lists, etc.). In order to support distributed-memory computation all of these data structures must be partitioned and assigned to processing elements (PEs). Finding minimal graph separators is a well known NP-hard problem. Some random graph classes lack good separators entirely [2], while commonly used methods tend to produce unbalanced cuts on interesting graph classes [4, 6]. This suggests that the approximately balanced partitions required to distribute the graph amongst a large number of PEs are likely to cut a large number of edges. The practical implications of this fact are that graph operations which depend on metadata from adjacent vertices and edges are likely to require remote memory accesses.

According to the "owner computes" paradigm, computations which update metadata associated with a vertex or edge are performed by the PE which owns that vertex or edge. A common method of dealing with data dependencies is to cache copies of remote data in ghost cells locally and define a consistency model that allows up-to-date values of these remotely owned elements to be accessed when necessary. This approach treats the graph metadata like an object-based Distributed Shared Memory with weak consistency.

In the case of graph computations this approach has two problems. First, because the dependency structure of the computation cannot be determined analytically it is not necessarily the case that the owner of a piece of data knows what other PEs will need to access it, or when. This means that data dependencies must be satisfied by "pulling," the data from the owner at run-time. Secondly, enforcing consistency requires global synchronization operations because it is not possible to determine analytically which PEs have accessed which data. These synchronization operations group computations into epochs based on the fact that they occur concurrently, regardless of whether they actually share any data dependencies.

Rather than moving dependent data to the computation site, it is often desirable to move the computation to the data it depends on. This approach eliminates the need to cache and maintain non-local data, requires less synchronization, and can reduce latency. With graph applications in mind we have developed a generalized active message framework, AM++ [10], to perform this task. AM++ has a number of desirable features which have been described previously however, the ability for message handlers to themselves generate active messages to unbounded depth is the key feature which enables the library design we propose. This "fire and forget" method of computation is supported by pluggable termination detection algorithms which allow application developers to determine when all handlers have completed. The termination detection process defines epochs as with the coarse-grained approach described previously, but avoids the need to introduce additional epochs to enforce progress or data consistency. This is especially beneficial in unbalanced computations such as graph applications because no process can proceed to the next epoch until the longest running process in the previous epoch has completed.

An additional benefit of using asynchronous active message handlers as the work-distribution model is that the location of the target data can be transparent to the application developer. In the case of local data the handler is invoked locally, in the case of remote data it is invoked remotely using AM++. Locality information is of course still available in the case that

it can be used to optimize communications.

One last feature is essential to scaling graph applications to thousands or tens of thousands of processing elements. Because the graph data is unstructured, it is likely that the resulting communication pattern is dense, that is that each PE communicates with many others. In a system with p PEs, maintaining $\mathcal{O}(p)$ communication buffers and channels per PE is both expensive and non-scalable. Host-based routing allows a virtual topology to be embedded in the physical network topology. Messages are then routed through the (sparser) virtual topology which reduces the number of peers each PE communicates with at the expense of latency.

3 Fine-Grained Parallelism

Distributed-memory HPC systems are increasingly composed of multicore nodes. While it is prohibitively expensive to find independent sets of operations within a graph application, it is possible to enforce the atomic execution of operations at the graph metadata level and thus allow them to be treated as if they were independent. Graph applications require atomic transactions on disjoint, possibly non-contiguous metadata. Software transactional memory (STM) [5, 8] is a complicated and active area of research however, we can make use of the fact that the metadata is related using structural information in the graph itself and thus obviate the need to provide fully-general STM. If the metadata in question is all related to a single vertex or edge, we can simply associate a lock with that vertex or edge. In the case that the metadata is contiguous it may also be possible to apply lock-free techniques. If the metadata in question spans multiple vertices or edges it is necessary to define a shared lock hierarchy that ensures atomic access.

Encapsulating the data access concerns in the metadata layer using memory transactions has two benefits. First, it allows each handler invocation to be treated independently and executed concurrently. Second, it allows the method in which these transactions are performed to be optimized independent of the applications which use them. Combining asynchronous handler invocation with metadata transactions yields transactional semantics on graph metadata distributed across multiple address spaces.

In this approach the application developer defines a set of handlers which are assumed to execute concurrently and optionally, additional control flow which performs termination detection and other coordination tasks if necessary. The progress semantics are such that threads which perform communication operations may execute handlers for active messages received before returning.

Algorithm 1: Coarse-grained depth-limited search.

Input: Vertex v_0, D the maximum distance to explore from v_0, $\texttt{neighbors}(v)$ a function returning the neighboring vertices of v, Q a distributed queue
Output: $\mathrm{d}[v]$ the distance of v from v_0

```
1  ∀ v ∈ V: d[v] = ∞;
2  d[v0] ← 0;
3  Q ← ∅;
4  enqueue(Q, v0);
5  while Q ≠ ∅ do
6  |   u ← dequeue(Q);
7  |   foreach v ∈ neighbors(u) do
8  |   |   if d[u] + 1 < D and d[v] > d[u] + 1 then
9  |   |   |   d[v] ← d[u] + 1;
10 |   |   |   enqueue(v);
```

Algorithm 1 shows an example of a parallel, bounded-depth search on a graph in a coarse-grained fashion using a distributed queue. The distributed queue is composed of one local queue per process; enqueueing a vertex results in the vertex being placed on the local queue of the owning process. In this implementation checking to see if the distributed queue is empty requires checking the local queue for emptiness on each process and thus serves as a synchronization point.

Algorithm 3 shows an example of the same parallel, bounded-depth search using the proposed active message-based design. In this implementation the distance updates are assumed to be performed atomically but each instance of the `discover` handler may execute concurrently. The `end_epoch` method returns once all active instances of the `discover` handler have completed. The address space in which the `discover` handler executes is determined by the ownership of the vertex discovered (u).

Algorithm 2: `discover`(v, d_v) – active message handler for vertex discovery

Input: v the vertex to discover, d_v the tentative distance to v, D the maximum distance to explore, `neighbors`(v) a function returning the neighboring vertices of v

1 **if** $d_v < d[v]$ **then**
2     $d[v] \leftarrow d_v$;
3   **if** $d_v + 1 < D$ **then**
4   **foreach** $u \in$ `neighbors`$[v]$ **do**
5   `discover`$(u, d_v + 1)$;

Algorithm 3: Depth-limited search using active messages.

Input: Vertex v_0
Output: $d[v]$ the distance of v from v_0

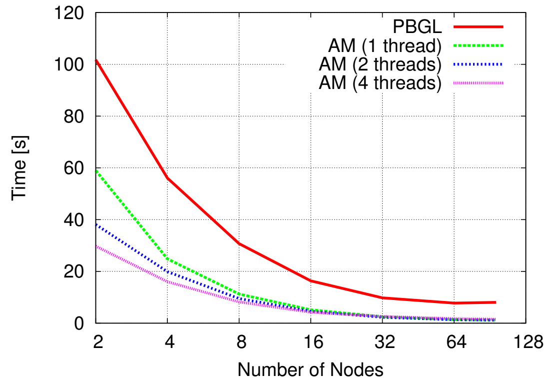
1 $\forall v \in V$: $d[v] = \infty$;
2 `begin_epoch`();
3 `discover`$(v_0, 0)$;
4 `end_epoch`();

4 Evaluation

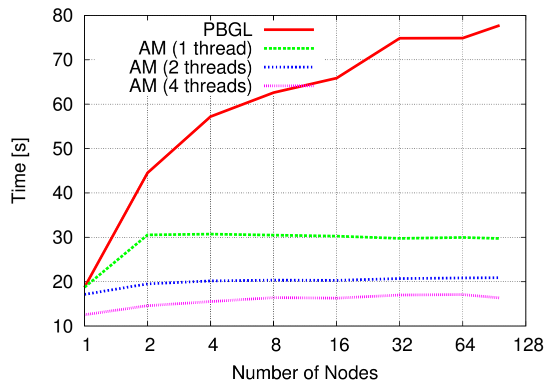
Preliminary evaluations of two graph kernels implemented using the proposed techniques have been performed in order to determine whether the performance goals outlined are feasible. We present performance results on Odin, a 128-node InfiniBand cluster (Single Data Rate). Each node is equipped with two 2 GHz Dual Core Opteron 270 CPUs and 4 GiB RAM. We ran our experiments with a pre-release version of AM++ over Open MPI 1.4.1 and OFED 1.3.1.

In order to assess the performance and scalability of hybrid-parallel graph algorithms implemented using active messages and transactional metadata updates we compared the performance of algorithms implemented in this fashion to the Parallel BGL. The communication layer in the Parallel BGL performs message coalescing, early send/receive, and utilizes asynchronous MPI point-to-point operations and traditional collectives for communication. All results are for Erdős-Rényi random graphs.

Figure 1 shows the performance of the active message (AM) implementation of the Breadth-First Search graph kernel with various numbers of threads compared to the single-threaded Parallel BGL implementation. Figure 1(a) demonstrates that not only is the AM implementation faster than the Parallel BGL implementation on a fixed-size problem but that it benefits from additional threads in all cases except with 4 threads on 32–96 processors, likely due to contention as the amount of work available on each node decreases. Figure 1(b) demonstrates that the AM implementation exhibits no increase in runtime when both the problem size and number of processors available are increased simultaneously. This is expected as BFS performs $\mathcal{O}(|V|)$ work.



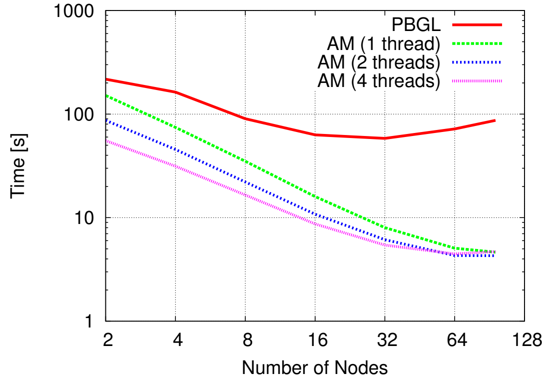
(a) Strong scaling (2^{27} vertices and 2^{29} edges).



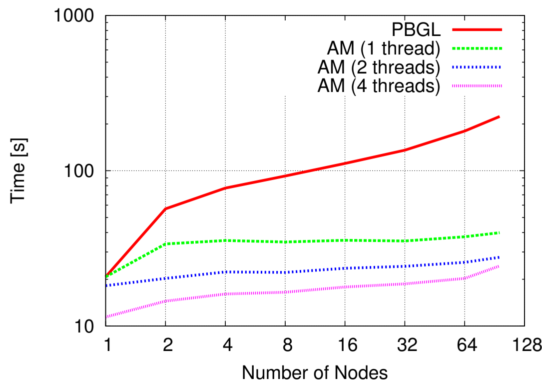
(b) Weak Scaling (2^{25} vertices and 2^{27} edges per node).

Fig. 1: Performance of the Parallel BGL and active message-based implementations on a parallel breadth-first search.

The single-source shortest paths (SSSP) problem requires finding the shortest distance from a single source vertex to all other vertices. A variety of SSSP algorithms exist, Figure 2 shows the performance of Δ-Stepping which has been found to be one of the better-performing parallel algorithms on distributed-



(a) Strong Scaling (2^{27} vertices and 2^{29} edges).



(b) Weak Scaling (2^{24} vertices and 2^{26} edges per node).

Fig. 2: Performance of the Parallel BGL and active message-based implemenations computing single-source shortest paths in parallel using Δ-Stepping.

memory systems [1]. Figure 2(a) shows that while the Parallel BGL implementation scales inversely between 32 and 96 nodes the AM implementation continues to scale as both additional nodes and threads are added in almost all cases. This inverse scaling behavior is likely due to communication overheads dominating performance as the work per processor decreases. In Figure 2(b) the AM implementation exhibits minor increases in runtime as problem size and node counts are increased. This is expected as SSSP performs $\mathcal{O}(|V| \log |V|)$ work while the number of nodes increases linearly. The AM implementation scales with a more gradual slope, as well as outperforming the Parallel BGL implementation.

5 Future Work

Work is currently underway to write a full graph library based on the design developed with the prototype implementations presented. Additionally we have begun work to evaluate the performance of the prototype implementations on larger-scale distributed memory systems, including the IBM Blue Gene/P. We also plan to evaluate the performance of the tools and techniques utilized on systems with more on-node parallelism. Developing new algorithms to fit the programming model exposed by this library is also of continuing interest.

References

[1] N. Edmonds, A. Breuer, D. Gregor, and A. Lumsdaine. Single-source shortest paths with the Parallel Boost Graph Library. In *The Ninth DIMACS Implementation Challenge: The Shortest Path Problem*, Piscataway, NJ, November 2006.

[2] P. Erdős, R. L. Graham, and E. Szemeredi. On sparse graphs with dense long paths. Technical Report CS-TR-75-504, Stanford University, Stanford, CA, USA, 1975.

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

[4] S. Guattery and G. L. Miller. On the quality of spectral separators. *SIAM Journal on Matrix Analysis and Applications*, 19(3):701–719, 1998.

[5] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, 1993.

[6] K. Lang. Fixing two weaknesses of the spectral method. In *Neural Info. Proc. Systems*, 2005.

[7] A. L. Rosenberg and L. S. Heath. *Graph Separators, with Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[8] N. Shavit and D. Touitou. Software transactional memory. In *Principles of Distributed Computing*, pages 204–213. ACM, 1995.

[9] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A mechanism for integrated communication and computation. In *Intl. Symp. on Comp. Arch.*, pages 256–266, 1992.

[10] J. Willcock, T. Hoefler, N. Edmonds, and A. Lumsdaine. AM++: A generalized active message framework. In *Parallel Architectures and Compilation Techniques*, Sept. 2010.