Expressing Graph Algorithms Using Generalized Active Messages

Nick Edmonds
Indiana University
Bloomington, IN 47405
ngedmond@cs.indiana.edu

Jeremiah Willcock
Indiana University
Bloomington, IN 47405
jewillco@cs.indiana.edu

Andrew Lumsdaine
Indiana University
Bloomington, IN 47405
lums@cs.indiana.edu

Abstract

Recently, graph computation has emerged as an important class of high-performance computing application whose characteristics differ markedly from those of traditional, compute-bound, kernels. Libraries such as BLAS, LAPACK, and others have been successful in codifying best practices in numerical computing. The data-driven nature of graph applications necessitates a more complex application stack incorporating runtime optimization. In this paper, we present a method of phrasing graph algorithms as collections of asynchronous, concurrently executing, concise code fragments which may be invoked both locally and in remote address spaces. A runtime layer performs a number of dynamic optimizations, including message coalescing, message combining, and software routing. Practical implementations and performance results are provided for a number of representative algorithms.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming

Keywords Parallel Graph Algorithms; Active Messages; Parallel Programming Models

1. Introduction

Graph problems have a number of inherent characteristics that distinguish them from traditional scientific applications [1]. Graph computations are often completely *data-driven*: they are dictated by the vertex and edge structure of the graph rather than being expressed directly in code. Execution paths and data locations are therefore highly unpredictable. Moreover, the connectivity of many graphs is not determined by 3D space (as is the case for discretized PDEs), resulting in data dependencies and computations with *poor locality*. Partitioning in such situations is computationally impractical since no good separators may exist [2, 3] and scalability can be significantly limited by the resulting unbalanced computational loads. Finally, graph algorithms are often based on exploring the structure of a graph rather than performing large numbers of computations on the graph data, which results in *fine-grained data accesses* and a *high ratio of data accesses to computation*. Memory and communication latency can dominate such computations.

The standard parallelization approach for scientific applications follows the SPMD model which partitions the problem data among a number of processes and then uses a bulk-synchronous-parallel [4] pattern to effect the overall computation. Although this approach has been tremendously successful for scientific applications based on discretized PDEs, it is not well-suited for graph-based, data-intensive applications [1]. To address these issues, we have developed an approach for portably expressing high-performance graph algorithms based on fine-grained generalized active messages, as provided by the Active Pebbles programming model [5].

Formulating graph algorithms using active messages has the dual advantages of being both conceptually simple while not over-constraining implementations. Active messages allow the user to modify algorithm code in an understandable fashion (individual vertex and edge operations) rather than requiring the use of complicated, vendor-tuned primitives which operate at a coarser level.

With this approach, we are able to capture the fine-grained dependency structure of graph computations at runtime, exposing maximal parallelism. Separating the expression of an algorithm (the code in the active messages) from the implementation (the messages themselves) enables performance portability across a variety of platforms without modifying the algorithm specification. This allows the runtime system to adapt algorithms to the hardware using coalescing and other transformations. Deferring these decisions until runtime is especially appropriate for graph algorithms as the structure of the graph determines the computational structure of the algorithm and thus both are discovered dynamically.

Finally, the active message phrasing permits both shared- and distributed-memory parallelism, and is amenable to acceleration. A single algorithm specification may be executed using an active message library in a distributed memory environment, a threading library (possibly combined with work-stealing) in a shared memory environment, and a hardware-specific programming environment targeting various kinds of accelerators. Most importantly, arbitrary combinations of these hardware environments are supported.

2. Programming Model

Message passing, an effective programming model for regular HPC applications, provides a clear separation of address spaces and makes all communication explicit. The Message Passing Interface (MPI) is the de facto standard for programming such systems. However, graph applications need shared access to data structures which naturally cross address spaces. A number of choices exist for how to implement fine-grained, irregular remote memory access. The key requirement with regard to graph applications is that the remote memory updates performed by one process must be atomic with regard to those performed by other processes and must support "read-modify-write" operations (e.g., compare-and-swap, fetch-and-add, etc.). More importantly, only the process performing the updates has knowledge of which regions of memory are being updated and thus the process whose memory is the target of these updates cannot perform any sequencing or arbitration of the updates. Finally, some algorithms require dependent updates to multiple, non-contiguous locations in memory, which provides perhaps the greatest challenge to a programming model.

MPI-2 One-Sided operations and Partitioned Global Address Space (PGAS) models attempt to fill the gap left by two-sided MPI message passing by allowing transparent access to remote memory in an emulated global address space. However, mechanisms for concurrency control are limited to locks and critical sections; some models support weak predefined atomic operations (e.g., *MPI_Accumulate()*). Stronger atomic operations (e.g., compare and swap, fetch and add) and user-defined atomic operations are either not supported in current versions or do not perform well. Thus, we claim that these approaches do not provide the appropriate primitives for fine-grained graph applications. The designers of those approaches have also realized these limitations in the original models, leading to proposals such as UPC queues [6] and Global Futures [7] to add more sophisticated primitive operations (including active messages in some cases) to otherwise PGAS programming models. The Chapel and X10 languages in particular have direct support for active messages.

PPoPP'13, February 23–27, 2013, Shenzhen, China.
ACM 978-1-4503-1922/13/02.

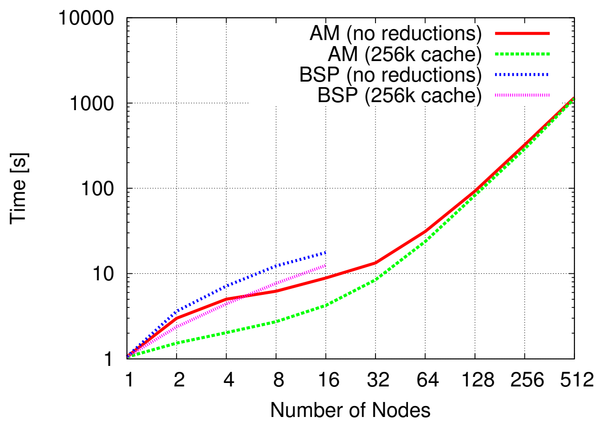

Figure 1. Δ-stepping shortest paths (Graph 500, 2^{16} vertices per node, average degree 16, 2^{18}-element reduction caches).

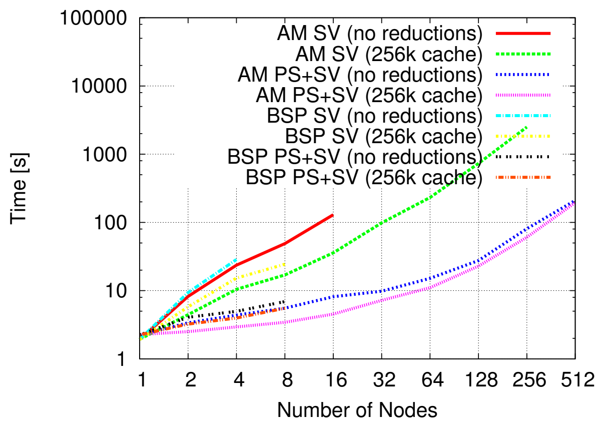

Figure 2. Shiloach-Vishkin connected components (Erdős-Rényi, 2^{18} vertices per node, average degree 2, 2^{18}-element reduction caches).[1]

3. Evaluation

We have applied the active message abstraction to the design a library of graph algorithms utilizing the Active Pebbles (AP) programming and execution model. We used Challenger, a 13.6 TF/s, 1 rack Blue Gene/P with 1024 compute nodes. Each node has 4 PowerPC 450 CPUs and 2 GiB of RAM. Our experiments used Version 1.0 Release 4.2 of the Blue Gene/P driver, IBM MPI, and `g++` 4.3.2 as the compiler (including as the back-end compiler for MPI).

For our evaluation, we chose algorithms from each of four representative algorithm classes—wavefront expansion (label-setting and label-correcting), coarsening and refinement, and iterative methods—however in the interest of brevity we present only a subset of the results here. We compare algorithms written in an active message (AM) style to the same algorithms written in a BSP [4] style. Both implementations use AP for the sake of comparison including routing and, where specified, message reductions provided in AP. Our results demonstrate reasonable weak-scaling performance and, more importantly, the ability to complete computations where the BSP form of the algorithm fails due to memory exhaustion (which is indicated by a missing data point). This is due to the fact that in the AM algorithms messages can be executed and retired whereas, in the BSP cases data communicated must be retained until the end of a computation phase *even if it is received earlier*.

Figure 1 shows the weak scaling performance of the Δ-stepping single-source shortest paths algorithm [8], an example of a label-correcting wavefront. Here, the BSP algorithms fail to complete due to memory exhaustion for more than 16 nodes. In addition, the AM formulations out-perform their BSP counterparts in the region where both algorithms run to completion.

The Shiloach-Vishkin connected components algorithm is an example of a coarsening and refinement algorithm. It consists of contraction of components to rooted “stars” via iterative hooking and pointer-doubling steps. We also implement an optimized algorithm which does a parallel exploration from the highest degree vertex (which is likely to be in the giant component), and then applies the Shiloach-Vishkin connected components algorithm to the undiscovered portion of the graph to label the remaining components. We call this variant “Parallel Search + Shiloach-Vishkin” (PS+SV). Figure 2 shows the weak scaling performance of the AM and BSP implementations of both algorithms. The AM implementation out-performs the corresponding BSP implementation in all cases, in addition to completing successfully on more nodes.

Increasingly poor scaling in both sets of experiments as the processor counts increase could be ameliorated by making the choice of coalescing factors and cache sizes dynamic. Additionally, removing routing at smaller scales would improve performance there. Incorporating dynamic variation of these features at runtime is interesting future work.

4. Conclusion

Phrasing graph algorithms as collections of asynchronous, concurrently executing, message-driven fragments of code allows for natural expression of algorithms, flexible implementations leveraging various forms of parallelism, and performance portability—all without modifying the algorithms themselves. Active messages are an effective abstraction for expressing graph applications because they allow the fine-grained dependency structure of the computations to be expressed directly in a form that can be observed dynamically at runtime as it is discovered. At this point a variety of optimizations are available (coalescing, reductions, etc.) that are difficult or impossible to apply effectively at compile time. The active message abstraction allows the specification of graph algorithms to be separated from the details of their execution, yielding flexible and expressive semantics and high performance.

Acknowledgments

This work was supported by NSF grant CCF-11118882. Resources of the Argonne Leadership Computing Facility at Argonne National Laboratory were used, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

References

[1] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, “Challenges in parallel graph processing,” *Parallel Processing Letters*, vol. 17, no. 1, pp. 5–20, 2007.

[2] K. Lang, “Fixing two weaknesses of the spectral method,” in *Neural Information Processing Systems*, 2005.

[3] P. Erdős, R. L. Graham, and E. Szemeredi, “On sparse graphs with dense long paths,” Stanford Univ., Tech. Rep. CS-TR-75-504, 1975.

[4] L. G. Valiant, “A bridging model for parallel computation,” *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[5] J. Willcock, T. Hoefler, N. Edmonds, and A. Lumsdaine, “Active Pebbles: Parallel programming for data-driven applications,” in *International Conference on Supercomputing*, Tucson, Arizona, May 2011.

[6] J. Jose, S. Potluri, M. Luo, S. Sur, and D. K. Panda, “UPC Queues for scalable graph traversals: Design and evaluation on InfiniBand clusters,” in *Conference on PGAS Programming Models*, Oct. 2011.

[7] D. Chavarria-Miranda, S. Krishnamoorthy, and A. Vishnu, “Global Futures: A multithreaded execution model for Global Arrays-based applications,” in *CCGRID*, 2012, pp. 393–401.

[8] U. Meyer and P. Sanders, “Δ-stepping: A parallelizable shortest path algorithm,” *J. Algorithms*, vol. 49, no. 1, pp. 114–152, 2003.

[1] The missing data point for “AM SV (1M cache)” is due to the wall clock time limit on the machine expiring before the algorithm completed, not memory exhaustion as with the missing data points for the BSP algorithms.