Active Pebbles: A Programming Model For Highly Parallel Fine-Grained Data-Driven Computations

Jeremiah Willcock
Indiana University
150 S. Woodlawn Ave
Bloomington, IN 47405, USA
jewillco@cs.indiana.edu

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

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

Abstract

A variety of programming models exist to support large-scale, distributed memory, parallel computation. These programming models have historically targeted coarse-grained applications with natural locality such as those found in a variety of scientific simulations of the physical world. Fine-grained, irregular, and unstructured applications such as those found in biology, social network analysis, and graph theory are less well supported. We propose Active Pebbles, a programming model which allows these applications to be expressed naturally; an accompanying execution model ensures performance and scalability.

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

General Terms Performance, Design

Keywords Irregular applications, programming models, active messages

1. Introduction

High-performance computing (HPC) traditionally focuses on problems with a static communication and computation schedule. This is reflected in today's de facto benchmark (HPL, High-Performance Linpack) and programming model (MPI, Message Passing Interface). Many problems, such as the solution of partial differential equations as used in numerous simulation techniques, fit this category. Less regular computations, such as adaptive mesh refinement, often use unstructured meshes to better represent physical phenomena. The resulting unstructured grids require more dynamic communication and computation schemes. However, such physical models still have a natural locality related to the properties of the modeled systems. Recently, a new class of dynamic and unstructured problems has emerged from areas such as biology, social network modeling, and graph theory. The most difficult instances of these problems require fine-grained, unstructured, data-driven accesses, as can be found in the solution of many graph problems such as betweenness centrality [3] or parallel shortest path search [5].

Traditional coarse-grained HPC applications can be expressed efficiently in the bulk synchronous parallel (BSP) model. However, the irregular, data-driven structure of graph applications prevents them from being statically coarsened to fit the BSP model well. Approaches which perform coarsening at run-time group operations based on concurrent execution and not necessarily actual computational dependencies. The fine-grained computational dependencies of these irregular, data-driven problems necessitate a data-driven programming model. We propose Active Pebbles (AP), a flexible programming model for fine-grained data-driven computations. AP allows the programmer to implement algorithms at their *natural* granularities (e.g., with billions of vertices, each of only a few bytes).

The AP programming model is related to Active Messages (AM) in that it utilizes fine-grained method invocation targeting globally-addressable objects. The two main differences between AP and AM are that AP implements transparent object addressing (and allows various object distribution schemes) and that objects are handled in bulk, i.e., they have no individual identity and may be processed in parallel. This enables the implementation of fine-grained algorithms at their natural levels of granularity and allows AP to fully capture their fine-grained dependency structures.

We also specify an execution model for AP which is crucial to achieving high performance at large process counts. In order to achieve this, AP exploits four main techniques:

1. Message Coalescing
2. Active Routing
3. Message Reductions
4. Termination Detection

The programming model ensures easy and abstract specification of parallel algorithms and is independent of the underlying execution architecture. However, the flexible execution model ensures the performance and scalability of the final AP application. The philosophy of the execution model is to transform the fine-grained unstructured specification into a representation that efficiently leverages the underlying architecture. Message coalescing can be used to adapt the message size to the network, active routing can transform all-to-all communication patterns into optimized collective patterns (similar to MPI collective operations), and message reductions can be used to spread the computational load and reduce network traffic.

2. Related Work

Several parallel programming models and languages have been designed in the recent past to address the challenges of data-intensive computing. PGAS languages such as Unified Parallel C [7] and Co-Array Fortran [6] allow shared access to distributed data, but they provide no method for invoking remote computations. Charm++ allows message-driven programming of relatively coarse-grained objects and supports advanced techniques such as redistribution and migration. Recently developed HPCS languages like Chapel [1] and X10 [2] allow the expression of distributed computations. However, it is unclear if these languages readily support fine-grained applications, e.g., billions of "places" in X10. We claim that if they were to support a similar fine-grained model

PPoPP'11, February 12–16, 2011, San Antonio, Texas, USA.
ACM 978-1-4503-0119-0/11/02.

then they would need to use the same techniques as we define in our execution model.

3. The Active Pebble Model

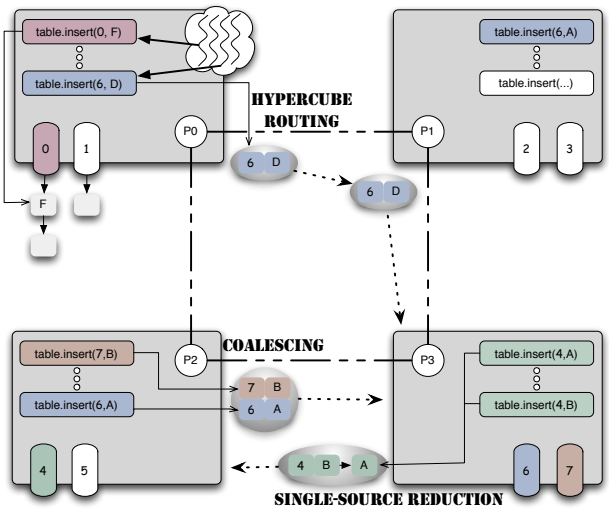

Figure 1. Overview of the Active Pebbles programming model.

3.1 Active Pebble Terminology

The primary abstractions in the AP model are pebbles and targets. Pebbles are light-weight active messages that operate on targets (which can, transparently, be local or remote). Targets are created by binding together a data object with a message handler, through the use of a distribution object. In Figure 1, targets are the destination buckets of a hash table, and hash values are used directly as destination addresses. The distribution object in this example would map each key i to process $\lfloor i/2 \rfloor$. Each pebble flows through the network from its source rank to its destination rank (determined by the distribution object). Handler functions can be multi-threaded and process incoming pebbles in any order; thus, the Active Pebbles model allows for an optimized implementation on multi-core nodes and accelerators. The techniques for optimized pebble transport and processing are discussed in detail in the following sections.

3.2 Fine-grained Pebble Addressing

Because AP is intended for problems with unstructured data, fine-grained addressing of pebbles based on their targets is essential. By making this addressing transparent to the user, local data can be treated similarly to remote data, avoiding special-casing for local and remote data. The method by which addressing is performed can be specified by the application programmer; both static and dynamic addressing is supported.

3.3 Message Coalescing

Modern networks are poorly suited for sending large numbers of small messages. They become injection-rate-limited before becoming bandwidth-limited. In order to overcome the injection rate limitation and achieve higher bandwidth utilization, message coalescing is necessary. By performing message coalescing at run-time, AP increases both message latency and network bandwidth utilization. The degree of message coalescing is selectable by the application programmer. This in effect coarsens the application at run time to better fit the parameters of the network. Taking the fine-grained application specification provided by the programmer and performing the minimal level of coarsening necessary to achieve good network bandwidth utilization minimizes the increase in message latency as well as enabling the application specification to be portable across hardware platforms.

3.4 Active Routing

In applications with good locality, the communication pattern (which processes communicate with which others) is generally sparse. By contrast, because data-driven applications are fine-grained and unstructured, the communication patterns are often dense. As process counts become large, maintaining $\mathcal{O}(P)$ communication buffers and their associated channels per process becomes infeasible. Active Routing simulates a virtual topology on top of the physical network topology and reduces the number of peers each process communicates with to $\mathcal{O}(\log P)$ or even $\mathcal{O}(1)$.

3.5 Message Reductions

It is often the case that multiple messages are sent to the same target by a single process. In some cases these messages may be redundant, while in other cases they may able to be combined via some reduction operation. Performing these reductions at the sender reduces network bandwidth utilization, as well as processing time at the target. Duplicate message elimination and message reductions are transparent to the application programmer, while the reduction operation to utilize for a given message type is user-specified. The effectiveness of message reductions is directly proportional to the number of messages they operate on, with higher degrees of coalescing increasing the effectiveness of reductions.

3.6 Termination Detection

Because AP utilizes a fully-general messaging model which supports sending messages from message handlers, it is not necessarily the case that the length of the longest chain of recursive messages is known at compile-time. Detecting delivery of all outstanding messages requires distributed termination detection [4]. The most efficient termination detection algorithm depends on whether dependent messages are known to be of a fixed-depth or may be of unbounded depth, as well as available features of the underlying network such as message counting or acknowledgment.

4. Conclusion

Active Pebbles consists of two parts, a *programming model* which allows fine-grained, unstructured, data-driven applications to be expressed at their natural level of granularity, and an *execution model* which maps this fine-grained expression to an efficient implementation. Separating expression from implementation allows the application specification to be portable across hardware platforms. Utilizing the four techniques described, the execution model allows the application specification to be adapted to the properties of the underlying physical network to enable maximum performance.

References

[1] D. Callahan, B. L. Chamberlain, and H. P. Zima. The Cascade High Productivity Language. In *Ninth Intl. Workshop on High-Level Par. Prog. Models and Supportive Environments*, pages 52–60, April 2004.

[2] P. Charles, C. Grothoff, V. A. Saraswat, et al. X10: An object-oriented approach to non-uniform cluster computing. In *Object-Oriented Programming, Systems, Languages and Apps.*, pages 519–538, 2005.

[3] N. Edmonds, T. Hoefler, and A. Lumsdaine. A space-efficient parallel algorithm for computing betweenness centrality in distributed memory. In *Int.l Conf. on High Performance Computing*, Goa, India, Dec. 2010.

[4] F. Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2(3):161–175, 1987.

[5] U. Meyer and P. Sanders. Δ-stepping: A parallelizable shortest path algorithm. *J. Algorithms*, 49(1):114–152, 2003. ISSN 0196-6774.

[6] R. W. Numrich and J. Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[7] UPC Consortium. UPC Language Specifications, v1.2. Technical report, Lawrence Berkeley National Laboratory, 2005. LBNL-59208.