New Execution Models are Required for Big Data at Exascale

Andrew Lumsdaine, lums@crest.iu.edu
Center for Research in Extreme Scale Technologies, Indiana University

Abstract
Computing on Big Data involves algorithms whose performance characteristics are fundamentally different from those of traditional scientific computing applications. Supercomputers are programmed today using execution models, such as CSP (and its primary realization, MPI), that are designed and optimized for traditional applications, but those models have weaknesses when applied to Big Data applications. For example, MPI's performance characteristics and operations emphasize synchronization between nodes within an application, and its implementations heavily optimize collective communication. Big Data algorithms, however, are highly asynchronous; although they can be implemented using models such as MPI, that style is not natural. These applications also benefit from specific optimizations of their messaging, such as duplicate message elimination, that do not apply to traditional applications and thus are not incorporated into MPI. Therefore, new execution models are necessary for Big Data applications, providing these optimizations as well as interfaces designed to simplify the expression of these applications. We illustrate these issues with the Active Pebbles programming and execution model, which provides a natural way to express irregular applications and execute them with high performance, even on current systems, but with the potential for further improvements on future hardware designed for advanced execution models and Big Data.

1 Introduction

Big Data is becoming an increasing part of scientific computing workloads, and many of the algorithms on Big Data are data-driven and irregular. Here we use the term "irregular" to mean that an algorithm accesses data in arbitrary locations scattered throughout memory with access patterns (typically data-driven) that are not predictable in advance. Examples of such algorithms include graph algorithms (breadth-first search, shortest paths, connected components), machine learning algorithms (junction tree learning, frequent itemsets, clustering), and mathematical optimization (linear programming, semidefinite programming). On the other hand, traditional supercomputer workloads often involve dense and/or sparse linear algebra in the service of solving partial differential equations; even when matrices in use are sparse, they are well-behaved because they typically represent discretizations of objects in a low-dimensional space. Big Data inputs often come from a high-dimensional space, if they can be viewed as having a geometry at all.

These different types of algorithms lead to different utilization of system resources and different performance characteristics. Dense linear algebra stresses the floating-point computation of a system; memory and communication bandwidths are important, but latencies are much less important. Sparse linear algebra includes much more random access to memory, and so latencies start to become a problem; also, they include much less computation per data element. On the other hand, because of the spatial origins of many sparse linear algebra problems, they can be partitioned fairly easily among nodes of a parallel computer, and communication can be localized. As opposed to these two problem classes, irregular applications do little or no floating-point computation per data element, and useful partitioning is impractical or even impossible. Their memory access patterns are also much more random and highly dependent on the input data. Communication patterns often involve many small messages, and every node is likely to communicate with all others. Memory performance is usually limited by latency in these applications, and communication can be limited by either bandwidth or latency (and may switch between these during the run of a single algorithm).

We argue that the different performance characteristics of irregular applications necessitate the use of different execution models for them as well. MPI is primarily designed to facilitate traditional scientific computing applications, and does well at this objective. Although it is also capable of expressing irregular applications, its encouraged programming style and common implementations' performance characteristics do not natively provide high performance on these applications. The simplest, and thus traditional, model to use for MPI programming is single-program, multiple-data (SPMD) processes and a bulk synchronous parallel (BSP) [8] structure for communication. In such a model, the processes making up an application run largely the same code, and are synchronized so they are nearly at the same place in the code at any given time. Communication occurs among all processes at the same time, and is often separated from computation (non-blocking collectives [5] weaken this separation, however). In irregular applications, on the other hand, synchronizing tasks too frequently leads to a large drop in performance. The types of structuring and reasoning about programs supported by SPMD and BSP do not apply to these applications. Although they can be shoehorned into MPI (and in fact, the AM++ framework described later is implemented using MPI), the programming style required is not natural, leading to both performance and maintainability problems.

2 Alternative Execution Models

Several execution models have been proposed as alternatives to MPI for high-performance computing. Among these are partitioned global address space (PGAS) models, and approaches based primarily on active messages such as Charm++ [6], ParalleX [4], ARMI [7], and Active Pebbles [10]. Newer models combine these approaches, in languages such as X10 [2] and Chapel [1] with both PGAS and active-messaging constructs.

Pure PGAS approaches are unsuitable for many irregular applications, even when they support fine-grained memory access and communication; the main reason is that their synchronization constructs are too limited. For example, atomic operations (if provided) are limited to fetch-and-add, compare-and-swap, and other fixed, simple

operations. Graph applications, on the other hand, need operations such as "do a fetch-and-min on this location, and if it succeeds, push an element into a queue." That latter operation may require memory access and is unlikely to be handled by any traditional atomic operation. The approach to resolve this problem is to incorporate active messages and other generalized atomic operations into PGAS models, as is done in Global Futures [3]. The other major approach to new execution models, and the one we advocate, is to use abstractions based on active messages [9]. In such a model, messages are sent explicitly, but received implicitly; instead, a receiving process registers a handler that is then called automatically when a message arrives. These models are highly asynchronous, at a cost of extra complexity. In particular, we advocate models inspired by the features of Active Pebbles [10]; those features are specifically targeted to irregular applications.

3 Active Pebbles

Active Pebbles defines a programming and execution model intended for irregular applications, with optimizations to help those applications to run well on current hardware. The programming model gives a way to express applications with fine granularities directly, without the user needing to coarsen message or computation granularities manually. The execution model then applies optimizations in domain- and system-specific ways to achieve better performance. The execution model includes four main optimizations:

1. **Message coalescing**: Although many current networks are not built for large numbers of tiny messages, message coalescing groups these messages into larger ones that can be transferred with higher bandwidths, at the cost of higher latencies.
2. **Active routing**: All-to-all communication in combination with message coalescing has non-scalable memory requirements. Thus, Active Pebbles allows messages to be routed in application- and system-specific ways to create a synthetic software network. This feature reduces memory usage while also making message reductions more effective. Coalesced groups of messages are split and recombined at intermediate nodes, providing the benefits of coalescing at this level as well.
3. **Message reductions**: Some algorithms have semantics allowing multiple messages relating to the same object to be combined in some way. For example, duplicate messages can be removed, or data values can be summed. Active Pebbles provides message reductions that can be applied at the source and at intermediate hops of message processing; caches (write-through and/or write-back) are used to identify a subset of messages that can be removed from further processing.
4. **Termination detection**: The Active Pebbles programming model allows message handlers to send messages, to unlimited depths. Thus, standard termination detection algorithms are used to safely detect quiescence when algorithms require full barrier synchronization.

These optimizations are synergistic: they combine to produce a final result that is better than the sum of the individual optimizations. For example, routing and reductions combine to synthesize tree-based reduction operations from messages written as point-to-point by the user. The features of Active Pebbles are shown in Figure 1.

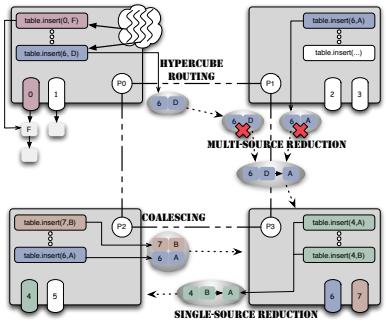

Fig. 1: Synergistic effects of Active Pebbles execution model optimizations.

4 Conclusion

Big Data problems have computational and storage needs that differ greatly from those of traditional scientific computing, and thus the programming and execution models used for high performance and software maintainability on traditional applications provide a poor fit for this new class of applications. We propose message-driven execution models, such as Active Pebbles, ParalleX, and Charm++ as better alternatives for Big Data. These models provide semantics and optimizations specific to Big Data applications, giving them better performance than older models, even on existing supercomputers. Hardware support for advanced execution models in future systems would enhance performance on irregular applications even further.

References

[1] D. Callahan, B. L. Chamberlain, et al. The Cascade High Productivity Language. In *Intl. Workshop on High-Level Parallel Programming Models and Supportive Environments*, pp. 52–60. April 2004.

[2] P. Charles, C. Grothoff, et al. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA*. 2005.

[3] D. Chavarria-Miranda, S. Krishnamoorthy, et al. Global Futures: A multithreaded execution model for Global Arrays-based applications. In *CCGRID*, pp. 393–401. 2012.

[4] G. R. Gao, T. Sterling, et al. ParalleX: A study of a new parallel computation model. In *IPDPS*. 2007.

[5] T. Hoefler, A. Lumsdaine, et al. Implementation and performance analysis of non-blocking collective operations for MPI. In *Supercomputing*. Nov. 2007.

[6] L. V. Kalé and S. Krishnan. CHARM++: a portable concurrent object oriented system based on C++. *SIGPLAN Not.*, 28(10):91–108, 1993.

[7] N. Thomas, S. Saunders, et al. ARMI: A high level communication library for STAPL. *Parallel Processing Letters*, 16(02):261–280, 2006.

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

[9] T. von Eicken, D. E. Culler, et al. Active messages: a mechanism for integrated communication and computation. In *International Symposium on Computer Architecture*, pp. 256–266. 1992.

[10] J. Willcock, T. Hoefler, et al. Active Pebbles: Parallel programming for data-driven applications. In *International Conference on Supercomputing*. Tucson, Arizona, May 2011.