Parallel Computing 37 (2011) 576–591

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Tuning collective communication for Partitioned Global Address Space programming models

Rajesh Nishtala [b,*], Yili Zheng [a], Paul H. Hargrove [a], Katherine A. Yelick [a,b]

[a] CRD/NERSC, Lawrence Berkeley National Laboratory Berkeley, Berkeley, CA 94720
[b] Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, USA

ARTICLE INFO

Article history:
Available online 12 June 2011

Keywords:
Partitioned Global Address Space languages
Collective communication
One-sided communication

ABSTRACT

Partitioned Global Address Space (PGAS) languages offer programmers the convenience of a shared memory programming style combined with locality control necessary to run on large-scale distributed memory systems. Even within a PGAS language programmers often need to perform global communication operations such as broadcasts or reductions, which are best performed as collective operations in which a group of threads work together to perform the operation. In this paper we consider the problem of implementing collective communication within PGAS languages and explore some of the design trade-offs in both the interface and implementation. In particular, PGAS collectives have semantic issues that are different than in send–receive style message passing programs, and different implementation approaches that take advantage of the one-sided communication style in these languages. We present an implementation framework for PGAS collectives as part of the GASNet communication layer, which supports shared memory, distributed memory and hybrids. The framework supports a broad set of algorithms for each collective, over which the implementation may be automatically tuned. Finally, we demonstrate the benefit of optimized GASNet collectives using application benchmarks written in UPC, and demonstrate that the GASNet collectives can deliver scalable performance on a variety of state-of-the-art parallel machines including a Cray XT4, an IBM BlueGene/P, and a Sun Constellation system with InfiniBand interconnect.

Published by Elsevier B.V.

1. Introduction

Partitioned Global Address Space (PGAS) languages combine the convenience of shared memory programming with the locality and data layout control of message passing. These languages run well on both shared memory and distributed memory hardware, taking advantage of shared memory support when it exists and leveraging the partitioned nature of the address space for performance on distributed memory supercomputers and clusters. The main difference between PGAS programming and the conventional message passing programming model, such as MPI, is that PGAS languages provide a global view of the memory across nodes and support one-sided communication, *read* and *write* (or *get* and *put*), for shared data. In contrast, in the two-sided communication model in MPI a process can only see its own local memory space and needs matching *send* and *receive* operations between itself and a remote process to transfer data. MPI is widely used due to its high quality implementations, portability, and good performance scalability, but the two-sided communication model is inconvenient for expressing asynchronous communication patterns in applications like histogram construction, mesh generation, and certain graph algorithms, in which only one side knows about a communication event. A global address space and one-sided

* Corresponding author.
E-mail addresses: rajeshn@cs.berkeley.edu (R. Nishtala), yzheng@lbl.gov (Y. Zheng), phhargrove@lbl.gov (P.H. Hargrove), kayelick@lbl.gov (K.A. Yelick).

0167-8191/$ - see front matter Published by Elsevier B.V.
doi:10.1016/j.parco.2011.05.006

communication primitives make developing such applications easier. In addition, by decoupling the transfer of data from the notification to a remote process, the one-sided communication can take advantage of Remote Direct Memory Access (RDMA) hardware and avoids some of the overheads of message and tag matching that are inherent to the two-sided model.

There are many different variations of PGAS languages and each has different design goals and programming styles but all of them fundamentally share the idea of a global address space in which a thread has affinity to part of the global address space. PGAS programming language implementations share a common set of runtime needs including: one-sided communication, collective communication, active messages, memory allocation, and the management of processes and threads. The GASNet communication library [1] is a portable runtime system for enabling PGAS implementations. GASNet is used to implement many PGAS programming languages including: Co-Array Fortran [2], UPC [3], Titanium [4], Cray Chapel [5], in addition to some research prototypes.

One-sided communication involves only a single application level process and two-sided communication involves two, but it is often useful in both models to have operations involving a large set of processes working collectively to perform a global communication operation. Common *collective* operations such as broadcast and reduction could be performed by accessing shared variables in a global address space, but this approach is neither convenient nor scalable. To aid in productivity and performance, many languages provide a standard set of such collective operations. These operations encapsulate common data movement and inter-processor communication patterns, such as broadcasting an array to all the other processors or having all processors exchange data with every other processor. The abstraction is intended to shift the responsibility of optimizing these common operations away from the application writer, who is probably an application domain expert, into the hands of the implementers of the runtime systems. While this problem has been well studied in the two-sided communication model community, the primary focus of this paper will be to understand and improve the performance and productivity benefits of collective operations in PGAS languages such as UPC [3], Titanium [4], and Co-Array Fortran [2], with the emphasis on UPC. The semantic differences between one- and two-sided communication pose interesting and novel opportunities for defining and tuning collectives.

PGAS collectives are different than MPI collectives in several aspects:

- MPI collectives are node-centric, while PGAS collectives are data-centric. For example, in MPI broadcast, the user specifies the source by providing the rank of of root. In contrast, in a UPC broadcast, the user specifies the source by providing the pointer to the shared data buffer and the runtime determines the location of the data automatically.
- MPI collectives implementations are often built on top of two-sided communication primitives (send and receive) whereas PGAS collectives are often built on top of one-sided communication primitives (put and get).
- PGAS collectives have an added semantic complexity in that one process may be writing to remote data that is involved in a concurrent collective operation. In UPC, for example, a set of synchronization modes are used to limit such behavior.

To benefit as many PGAS implementations as possible, we have designed, implemented, and tuned collective functions for PGAS languages in the GASNet system. Because GASNet is designed as a low-level system software library, programmers usually use GASNet indirectly by programming in one of the PGAS languages that implemented on top of GASNet. In this work, we use Berkeley UPC to implement application level benchmarks to evaluate the performance of the optimized GASNet collectives implementation.

In this paper we make the following contributions: (i) explore the interface issues with collective communication in PGAS language, which includes trade-offs in performance and simplicity; (ii) describe the implementation of a collective communication library for PGAS languages as part of the GASNet communication library; (iii) enumerate some of the implementation approaches for various collectives and the need to automatically tune over these implementation to select a reasonable one for a given machine and usage scenario; (iv) evaluate the collectives on some of the largest computer systems available today, including a Cray XT4, an IBM BlueGene/P, as well as a large Sun cluster with an InfiniBand interconnect.

2. Designing a PGAS collectives interface

Many parallel algorithms perform global communication operations, such as broadcasts and reductions, that involve data movement across all threads. In PGAS languages, because all threads have access to shared data, the first design question is whether these global operations should be performed as a *collective* operation in which all threads participate together, can a single thread invoke a global operation. In PGAS languages, a single thread has access to all data in the shared space, operations on data spread across the machine can be performed by a single thread. Nevertheless, UPC, like MPI, provides these global operations as collectives. This simple model provides a good abstraction of current machines and requires minimal runtime support, albeit at some loss of convenience. Several questions remain about the collectives interface and implementation strategy.

One key part of the collectives interface that we do not discuss here in detail is the exact set of collective operations. GASNet and UPC provide both rooted collectives, such as broadcasts and reductions with a single source or destination thread that acts as the root, as well as non-rooted collectives that perform operations such as all-to-all communication patterns. Some of the collectives perform computation, e.g., scans and reductions, whereas others simply move data. These are described for UPC and GASNet in their respective specifications [3,1].