The Case for Collective Pattern Specification

Torsten Hoefler
University of Illinois at Urbana-Champaign
htor@illinois.edu

Jeremiah Willcock Arun Chauhan
Andrew Lumsdaine
Indiana University
{jewillco,achauhan,lums}@cs.indiana.edu

Abstract

Many scientific applications are written in a Bulk Synchronous Parallel style, in which regions of pure computation are separated by communication operations. Unless an existing MPI collective operation can be used, these communication operations are usually written as separate message sends and receives, making analysis and optimization difficult. This style of communication also reduces readability and maintainability by hiding the overall collective pattern in a maze of individual messages. Instead we advocate directly specifying (collective) communication operations in a domain-specific language. We further classify applications by their communication patterns, in particular with respect to different nodes' knowledge of the applications' patterns and the frequency of pattern changes, showing how a language would benefit each of these classes of applications and the requirements for such a language.

Categories and Subject Descriptors D.1.3 [*Programming techniques*]: Concurrent programming—Parallel programming; D.3.3 [*Programming languages*]: Language constructs and features—Concurrent programming structures

General Terms Languages, performance

Keywords Message passing, collective communication

1. Introduction

We argue that current semantics of message passing applications are too limited to express many application communication patterns directly, and consequently performance can suffer in HPC applications. Many applications have structured communication patterns that cannot be expressed at a high level, forcing users to implement their communications using low-level point-to-point primitives. The Message Passing Interface (MPI) [12], often regarded as *"the assembly language of parallel computing,"* is the de facto standard interface for parallel high-performance computing. However, we argue that MPI is more than an assembly language because, while it offers numerous low-level routines such as send and receive, it also supports high-level communication constructs and patterns through collective operations. These collective operations do not express single messages, but rather model group operations that can employ very complex communication patterns in their optimized execution [2, 9, 11]. MPI's abstractions and its capability for defining complex process topologies (Cartesian or general graphs) represent features of a high-level language.

AMP '10 June 6, 2010, Toronto, Ontario, Canada
ACM .

The seventeen collective operations in MPI 2.2 support many communication patterns found in scientific applications today. However, this set is limited to collectives in which all processes have the same view of the operation. Typical point-to-point communication patterns, such as nearest-neighbor exchanges, do not fit this scheme because the local views of the processes might not be uniform. Such operations are typically implemented with explicit point-to-point communications, that is, at the *"assembly level."* We note that the most general collective call, MPI_Alltoallv, is able to express most patterns, but this operation is neither scalable [1] nor easy to optimize. It also does not support reduction operations.

In this paper, we adopt the abstract perspective that many common communication patterns—even those that cannot be modeled by MPI collective operations—are, conceptually, collective communications. If these patterns are implemented by means of point-to-point operations, one loses abstract information about the concurrent communication and restricts optimization possibilities [6]. Also, point-to-point techniques force the user to descend into the lowest level of the MPI interface and to specify the pattern manually, which might lead to less maintainable source code [8].

We assume that most applications are written in a Bulk Synchronous Parallel (BSP) model [14] in that they execute in phases or supersteps. From this perspective, it seems beneficial to enable the supersteps' communication to be defined as collective operations. We argue that this technique could benefit a large class of applications and lead to higher performance and simpler implementations. In addition, several optimizations, such as automatic communication/computation overlap and relaxed synchronization, could be achieved with simple code transformations.

We discuss and characterize the communication patterns of typical scalable parallel applications that are implemented in the message passing model. We classify the patterns into five categories and provide guidelines for how each category should be supported by the middleware and handled by the programmer. We show that the current MPI interface does not support the specification of many of those patterns as collective operations.

2. Communication Patterns

Multiple *parallel design patterns* exist for the parallel implementation of common algorithms, for example, [3, 13]. In contrast to those, our work focuses on the algorithms' communication patterns and their efficient and easy specification and implementation; we thus classify applications based on their communication patterns. This classification serves as a set of parallel communication patterns, and a parallel language or environment should strive to support those patterns at the highest possible level. We describe five different patterns, evaluating MPI's expressiveness for each.

2.1 Compile-time static

In applications with compile-time static communication patterns, the communication structure is defined by the application's source code and is constant across all inputs for a given number of processes. These codes tend to use simple, for example, Cartesian, n-dimensional domain decomposition strategies. Codes that solely

use non-vector collective communication, such as parallel fast Fourier transforms, also fit in this category. These applications are typically BSP-style, with communication occurring collectively at given points throughout execution.

Implementation in MPI. Applications in this class often use either dense collective operations or point-to-point communication along Cartesian grids. Dense collective operations are part of MPI, and often optimized by the MPI implementation, sometimes with acceleration in network hardware. Cartesian topologies, represented through appropriate MPI communicators, can be mapped to the topology of a system's interconnection network but still enforce explicit point-to-point messaging along the edges.

Evaluation. The expressiveness of MPI for these applications depends on the exact communication pattern. Global, dense collectives are currently supported by MPI, with non-blocking collectives (used to overlap communication and computation) projected to arrive in MPI 3.0 [7]. Nearest-neighbor communications, on the other hand, cannot be expressed collectively. They must be written using point-to-point communication operations, which forces programmers to fall back to the "assembler" level, in MPI 2.2; the MPI_Sendrecv function can aid in avoiding deadlocks, but that function still creates independent send and receive operations. The MPI Forum is currently working on adding explicit support for this use case (*"sparse collectives"* [6]).

2.2 Run-time static

Applications in the run-time static class have communication patterns that are input-dependent, but the pattern is fixed during a single application run but cannot be compiled statically. These applications typically use unstructured meshes, with techniques such as graph partitioning to find good parallel distributions of the input problem. As in the compile-time static class of applications, typical communication operations are collective (BSP-style) in computations such as sparse matrix-vector multiplication and elementwise vector operations (often including reductions).

Implementation in MPI. This type of communication can be mapped to MPI constructs, but not as directly as for compile-time static patterns. A mesh in a file can be decomposed into load-balanced subproblems using a graph partitioning algorithm. The edges cut by the partition can be used to create a communication schedule, which can be mapped onto the system topology by an MPI graph communicator. Although the interface for constructing graph communicators in MPI 2.1 is not scalable, MPI 2.2 adds a scalable interface for use with larger systems [12, § 7.5.4]. Given a topology mapping to the actual network, data can either be read on the appropriate processor or redistributed after it has been loaded.

Evaluation. Operations such as vector dot products and norms can employ dense reductions, and convergence testing can use global reductions or broadcasts; MPI directly supports all of these. The unstructured nearest-neighbor communication involved in a sparse matrix-vector multiplication, on the other hand, cannot be expressed efficiently in MPI 2.2 without the manual use of point-to-point operations. The aforementioned sparse collective proposal for MPI 3.0 would enable high-level handling of this use case.

2.3 Run-time flexible

In this class of application, the communication pattern changes during the computation but the changes are relatively rare. Typical examples are applications using adaptive mesh refinement that reuse the same mesh multiple times before the re-meshing step. As in the previous two types of applications, run-time flexible applications are typically BSP-style and communications occur collectively.

Implementation in MPI. A similar approach can be used to implement rarely changing patterns as is used for static patterns. For changing graph structures, such as adaptive mesh refinement (AMR), less expensive partitioning algorithms may be used for the periodic repartitioning steps. MPI graph topologies may still be useful if the same communication schedule is reused often enough.

Evaluation. Run-time flexible applications map to MPI in basically the same ways as run-time static applications. However, one must consider the tradeoff between the costs and the benefits of optimizations. Operations involving all processes symmetrically can usually be done with MPI collectives, but the main body of each algorithm cannot be, as in the run-time static case.

2.4 Dynamic

Dynamic applications have communication patterns that have little structure, depend on the input, and change very frequently. Typical examples are parallel graph computations, high-impact adaptive mesh refinement applications that re-mesh every iteration, and many n-body methods. Some of these applications are BSP-style, such as partial differential equation solvers on fast changing meshes, breadth-first search, and parallel single-source shortest path algorithms. Those applications benefit from less synchronization: internal synchronization would slow down graph exploration (finding all vertices within a certain number of links of a given vertex), and sparse LU factorization uses mostly local synchronization operations. Because of the very loose synchronization necessary to hide communication latency in these applications, BSP-style approaches are not typically used.

Implementation in MPI. Fully dynamic applications, such as most graph computations, generally do not benefit from graph partitioning at all, so simple distributions are used. Dynamic applications tend to use non-blocking point-to-point operations for maximum concurrency, and purposely minimize synchronization among processes.

Evaluation. These applications only rarely use collective communication, and so hand-written message passing is necessary. Dynamic applications tend to use asynchronous messaging in order to hide latency (overlap communication and computation); the logic involved in managing the many outstanding requests is byzantine. Because MPI requires send and receive buffers to be managed by the application, it must also monitor each operation's completion in order to reclaim its buffers. Dynamic applications generally have communications sent to processes that do not expect them; therefore, MPI_ANY_SOURCE must be used. When a message's size is also unknown, MPI_Probe or MPI_Iprobe can be used to determine it, but these functions cannot be used reliably in the presence of threads [5]. One-sided communication operations or active messages are more suitable for these applications. MPI 2.0 provides simple one-sided operations, but they are inadequate for sophisticated applications such as graph algorithms, while active messages are not currently provided by MPI.

2.5 Massively parallel

This last class is mostly mentioned for completeness. Some parallel applications, such as Monte Carlo simulations, require minimal or no communication. Massively parallel applications in which all nodes do roughly the same amount of work, distributed at the beginning of the program's run, can be viewed as BSP-style with only a few giant supersteps. Many manager-worker applications, on the other hand, do not have global synchronization or communication except at the very end of execution.

Implementation in MPI. Massively parallel applications typically do little communication, and the communication structures are often simple (a few global collectives or a manager-worker

model). Many manager-worker applications are straightforward to implement (e.g., through the use of MPI_ANY_SOURCE to wait for work requests).

Evaluation. MPI works well for these applications. When communication patterns are simple, the applications are similar to those in the compile-time static class but with even less importance placed on message-passing performance.

In general, MPI's collective operations are predefined static patterns that can be plugged into many applications. They are a large step towards higher-level constructs (see Gorlatch [4]), but are only partially applicable to applications that use sparse communications. The MPI Forum has been working on various library additions to support sparse applications. We argue that compiler-aided transformations could also benefit these applications.

3. Our Position: Collective Pattern Specification

The importance of message-passing is undebatable and almost all message-passing models are implemented through libraries (MPI) that are inherently constrained by limited contextual information and requirements to adhere to standard APIs. Communication optimizations that require information on the larger contexts surrounding calls to library functions require error-prone and non-portable manual transformations.

An *declarative* specification that lets users express the *what*, but not the *how*, of communication separates the description of communication from its optimization. Note that unlike other parallel language-based approaches such as PGAS or HPF, this approach encourages programmers to think carefully about data decomposition and accesses, but relieves them of the need to write spaghetti message passing code that mixes computation and communication.

The fundamental communication abstraction in our model is the *collective*: A communication pattern that is run simultaneously by all nodes and that is logically synchronizing between them. The specification syntax uses a small extension to the host language (e.g., C++) while leveraging the full power of the underlying language (e.g., templates).

A compiler will be used to implement the patterns and optimize surrounding code. Compiler techniques were proved effective in communication optimization [10]; in our approach, the compiler optimizes parallel programs specified with a well-designed collective language. Optimization techniques include message coalescing, communication placement, scheduling, global pattern optimizations (e.g., turning a linear gather into a tree pattern), and overlapping communication and computation by automatically turning blocking calls into non-blocking calls. This model can be used to implement four of the five classes of applications given above:

Compile-time static and massively parallel. For this type of application, communication patterns are expressed as collective operations with respect to an arbitrary processor i, quantified over a set of processors. Since the language does not restrict the range of quantification, sparse collectives are specified just as naturally as dense collectives. In both cases, a compiler is able to derive the global pattern and optimize it statically.

Run-time static and flexible. Our language extensions will support the use of run-time values to specify communications, allowing patterns to be defined for unstructured applications. Compiler analysis is more difficult for unknown patterns—much of the optimization would need to be delayed until the exact patterns are known. The level of optimization applied would also vary based on the reuse of the pattern. A potential issue for these applications is that the full communication schedule is often not available globally; each node only has the portions relevant to it, and the run-time matching of sends and receives must be definable.

Dynamic. Applications in this class, requiring arbitrary and potentially unstructured communication patterns, do not lend themselves easily to specification in the form of collectives. Our extensions would thus need to provide special mechanisms to specify this type of communication; active messages with different termination detection schemes, or similar, would be one possibility.

We accept that scientific computing needs slow transitions instead of revolutionary new languages due to the high number of existing programs and the large amount of effort that has been spent on them. We therefore propose extensions that can be incrementally retrofitted into existing applications, with compiler assistance to use the extensions to improve performance. Such a high-level specification of communications will be especially important for high performance on petascale and exascale systems.

Acknowledgments

We would like to thank Marc Snir for helpful discussions, as well as William Byrd and Laura Hopkins for comments on the paper. This work was supported by the Lilly Endowment, DOE FASTOS II (LAB 07-23), and NSF grant CNS-0834722.

References

[1] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, S. Kumar, E. Lusk, R. Thakur, and J. L. Träff. MPI on a million processors. In *EuroPVM/MPI*, pages 20–30, 2009. ISBN 978-3-642-03769-6.

[2] J. Bruck, C.-T. Ho, S. Kipnis, and D. Weathersby. Efficient algorithms for all-to-all communications in multi-port message-passing systems. In *SPAA*, pages 298–309, 1994. ISBN 0-89791-671-9.

[3] S. Gorlatch. *Abstract Machine Models for Parallel and Distributed Computing*, pages 147–161. IOS Press, Amsterdam, 1997. ISBN 90-5199-267-X.

[4] S. Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM Trans. Program. Lang. Syst.*, 26(1):47–56, 2004. ISSN 0164-0925.

[5] D. Gregor, T. Hoefler, B. Barrett, and A. Lumsdaine. Fixing probe for multi-threaded MPI applications. Technical Report 674, Indiana University, Jan. 2009.

[6] T. Hoefler and J. L. Träff. Sparse collective operations for MPI. In *International Parallel & Distributed Processing Symposium, HIPS'09 Workshop*, May 2009. ISBN 978-1-4244-3750-4.

[7] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and performance analysis of non-blocking collective operations for MPI. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, 2007.

[8] T. Hoefler, F. Lorenzen, and A. Lumsdaine. Sparse non-blocking collectives in quantum mechanical calculations. In *EuroPVM/MPI*, volume LNCS 5205, pages 55–63, 2008. ISBN 078-3-540-87474-4.

[9] R. M. Karp, A. Sahay, E. E. Santos, and K. E. Schauser. Optimal broadcast and summation in the LogP model. In *Symposium on Parallel Algorithms and Architectures*, pages 142–153, New York, NY, USA, 1993. ACM. ISBN 0-89791-599-2. doi: http://doi.acm.org/10.1145/165231.165250.

[10] A. Karwande, X. Yuan, and D. K. Lowenthal. CC–MPI: a compiled communication capable MPI prototype for Ethernet switched clusters. *SIGPLAN Not.*, 38(10):95–106, 2003. ISSN 0362-1340.

[11] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. *SIGARCH Comput. Archit. News*, 19(2):269–278, 1991. ISSN 0163-5964.

[12] MPI Forum. MPI: A Message-Passing Interface Standard. Version 2.2, September 4th 2009. http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf.

[13] S. Siu, M. De Simone, D. Goswami, and A. Singh. Design patterns for parallel programming. In *Parallel and Distributed Processing Techniques and Applications*, pages 230–240, 1996.

[14] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990. ISSN 0001-0782.