Writing Parallel Libraries with MPI - Common Practice, Issues, and Extensions

Torsten Hoefler and Marc Snir

University of Illinois, Urbana, IL 61801, USA, {htor,snir}@illinois.edu

Abstract. Modular programming is an important software design concept. We discuss principles for programming parallel libraries, show several successful library implementations, and introduce a taxonomy for existing parallel libraries. We derive common requirements that parallel libraries pose on the programming framework. We then show how those requirements are supported in the Message Passing Interface (MPI) standard. We also note several potential pitfalls for library implementers using MPI. Finally, we conclude with a discussion of state-of-the art of parallel library programming and we provide some guidelines for library designers.

1 Introduction

Modular and abstract structured programming is an important software-development concept. Libraries are commonly used to implement those techniques in practice. They are designed to be called from a general purpose language and provide certain functions. Libraries can be used to simplify the software development process by hiding the complexity of designing an efficient and reusable collection of (parallel) algorithms. High-performance libraries often provide performance portability and hide the complexity of architecture-specific optimization details. Library reuse has been found to improve productivity and reduce bugs [2, 17].

In this work, we discuss principles for designing and developing parallel libraries in the context of the Message Passing Interface (MPI) [18]. We recapitulate the features that have been introduced 18 years ago [21, 6], add newly found principles and interface issues, and discuss lessons learned. We also analyze current practice and how state of the art libraries use the provided abstractions.

We show that the key concepts are widely used while other concepts, such as process topologies and datatypes, did not find very wide adoption yet. In the following sections, we describe principles for modular distributed memory programming, introduce a taxonomy for parallel libraries, discuss several example libraries, derive common requirements to support modular programming, show how MPI supports those requirements, discuss common pitfalls in MPI library programming, and close with a discussion of common practice.

2 Modular Distributed Memory Programming

Modular programming plays an important role in “Component-based software engineering (CBSE)”, which suggests to program by composing large-scale components. Korson and McGregor [12] identify ten generally desirable attributes that each serial and parallel library should bear: wide domain coverage, consistency, easy-to-learn, easy-to-use, component efficiency, extensibility, integrability, intuitive, robust, well-supported.

Those principles are also valid in distributed memory programming. The main difference is that distributed and parallel programming requires to control multiple resources (processing elements). Several major language techniques to support the development of distributed libraries have been identified in the development of the Eiffel language [16]. The list includes (among others) several items that are relevant for the development of parallel libraries:

classes reusable components should be organized around data structures rather than action structures
information hiding libraries may use each others facilities, but internal structures remain hidden and "cross talk" between modules is avoided
assertions characterize semantic properties of a library by assertions
inheritance can serve as module inclusion facility and subtyping mechanism
composability especially performance composability and functional orthogonality. This requires to query relevant state of some objects.

Writing distributed libraries offers a large number of possibilities because spatial resource sharing can be used in addition to temporal sharing with multiple actors. We identify the following main library types that are commonly used today: (1) spatial libraries use a subset of processes to implement a certain functionality and leave the remaining processes to the user (e.g., ADLB [15]), (2) collective loosely-synchronous libraries are called "in order" (but not synchronously) from a statically specified process group, and (3) collective asynchronous libraries are called by a static group of processes but perform their functions asynchronously.

2.1 A Taxonomy for Parallel Libraries

We classify existing parallel application libraries into four groups:

Computational Libraries provide full computations to the user, for example the solution of PDEs, n-body interactions, or linear algebra problems. Example libraries include PETSc [1], ScaLAPACK, PMTL [14], PBGL [4], PPM [20].
Communication Libraries offer new high-level communication functions such as new collective communications (e.g., LibNBC [11]), domain-specific communication patterns (e.g., the MPI Process Group in PBGL), or Active Messages (e.g., AM++ [23]).
Programming Model Libraries offer a different (often limited) programming model such as the master/slave model (e.g., ADLB [15]) or fine-grained objects (e.g., AP [24]).
System and Utility Libraries offer helper functionality to interface different architectural subsystems that are often outside the scope of MPI (e.g., LibTopoMap [10], HDF5 [3]) or language bindings (e.g., Boost.MPI, C# [5]).

Some of the example libraries and their used abstractions are discussed in the next section.

2.2 Example of Libraries

We now describe some example libraries that we categorized in our taxonomy that either utilize MPI to implement parallelism or integrate with MPI to provide additional functionality. This collection is not supposed to be a complete listing of all important parallel libraries. It merely illustrates one example for each type of parallel library and shows which abstractions have been chosen to implement parallel libraries with MPI.

PETSc The PETSc library [1] offers algorithms and data structures for the efficient parallel solution of PDEs. PETSc provides abstractions of distributed datatypes (vectors and matrices) that are scoped by MPI communicators and hides the communication details from the user. The passed communicator is copied and cached as attribute to ensure isolation. PETSc uses advanced MPI features such as nonblocking communication and persistent specification of communication patterns while hiding the message passing details and data distribution from the user. It also offers asynchronous interfaces to communication calls, such as VectScatterBegin() and VecScatterEnd() to expose the overlap to the user.

PBGL The Parallel Boost Graph Library [4] is a generic C++ library to implement graph algorithms on distributed memory. The implementation bases on *lifting* the requirements of a serial code to base a parallel implementation on it. The main abstractions are the process group to organize communications, the distributed property map to implement a communication abstraction, and a distributed queue to manage computation and detect termination. The library offers a generic interface to the user and uses Boost.MPI to interface MPI.

PMTL The Parallel Matrix Template Library [14] is, like the PBGL, a generic C++ library. It uses distributed vectors and matrices to express parallel linear algebra computations. As for PBGL, the concepts completely hide the underlying communication and enable optimized implementations.

PPM The Parallel Particle Mesh Library [20] provides domain decomposition and automatic communication support for the simulation of continuous systems. The library offers a high-level application oriented interface which is close to a domain-specific language for such simulations. It offers support for advanced functions of MPI.

ADLB The Asynchronous Dynamic Load Balancing Library, developed at Argonne [15], offers a simplified programming model to the user. The master/slave model consists of essentially three function calls and the scalable distribution of work and parallelization of the server process is done by the library. The library expects an initialized MPI and uses communicators during its init call to divide the job into master and slave groups. The master group "stays" within ADLB while the user has full access to the slave group to issue ADLB calls to the master. The library has been used with multi-threaded processes on BlueGene/P to achieve 95% on-node speedup.

LibNBC LibNBC [11] is an asynchronous communication library that provides a portable implementation of all MPI-specified collective operations with a non-blocking interface. It uses advanced MPI features to provide a high-performing interface to those functions and faces several of the issues discussed in Sections 3.3 and 3.4. LibNBC also offers an InfiniBand-optimized transport layer [9] that faces problems described in Section 3.5.

LibTopoMap LibTopoMap [10] provides portable topology mapping functionality for the distributed graph topology interface specified in MPI-2.2 [18]. It replaces the distributed graph topology interface on top of MPI and caches the new communicator and new numbering as attribute of the old communicator. This shows that a complete modular implementation of the Topology functionality in MPI is possible.

HDF5 The HDF5 library [3] offers an abstract data model for storing and managing scientific data. It offers support to specify arbitrary data layouts and its parallel version relies heavily on MPI datatypes and MPI-IO. As a system library, it faces several problems as discussed in Section 3.4.

2.3 Common Requirements of Parallel Libraries

Based on the survey of libraries, we distill the common requirements for a parallel runtime environment such as MPI. Parallel libraries require performance, scalability, usability, error handling, isolation (a "safe and private" communication space) for point-to-point and collective communication, and virtualized process naming (e.g., topologies or a virtual one-dimensional namespace). In addition, high-quality programming frameworks may offer topology mapping, fault-tolerance, and data management support to libraries.

3 The Loosely Synchronous Model in MPI

We now discuss how MPI supports parallel libraries by providing many of the required features listed in Section 2.3. The loosely synchronous model for parallel libraries is specified in Section 6.9 of the MPI standard [18]. In this model, all processes in a communicator invoke parallel subroutines in the same order. Those processes do not have to synchronize before the invocation.

We now discuss the main concepts in MPI that support the development of parallel libraries.

Communication Contexts in the form of MPI communicators are the most important concept for libraries. Communicators offer spatial and temporal isolation because they can specify disjoint process groups and isolate communication on overlapping process groups. This "communication privatization" is similar to the important "data privatization" in object oriented languages. Section 3.2 discusses potential issues with reentrant libraries. Each communicator has an associated **Process Group** that offers a virtual one-dimensional namespace for processes. Communicators also provide a scope for collective communication and support the concept of "functional and spatial composability" [16].

Virtual Topologies allow for domain-specific process naming schemes that can be passed to and queried by libraries. This extends the simple one-dimensional naming of process groups to arbitrary Cartesian naming schemes or general graph topologies (which can be enumerated by graph traversals, such as Breadth First Search).

Attribute Caching can be used to associate state to communicator objects. MPI allows to attach arbitrary data to communicators, windows, and datatypes in order to pass context or state information between library calls. MPI guarantees that this information can be quickly retrieved and is consistent. It also offers functionality to strictly control the inheritance of attributes in communicator copy functions. This allows to mimic the concept of "inheritance" [16] of general object oriented programming.

Datatypes defines an interface to exchange the layout of data structures for communications between libraries and user applications. MPI offers the required functions to create private copies of datatypes (MPI_Type_dup) and manipulate them. It also offers functions to query the composition of existing datatypes and serialize or deserialize (MPI_Pack/MPI_Unpack) them into/from buffers. Those abstractions support the abstract definition of datatypes in libraries.

MPI's Modular Design allows to implement full sections of the MPI standard as separate libraries (e.g., Sections 5 (Collective Communication), 7 (Process Topologies), and 13 (I/O) can solely be implemented with the core functionality of MPI 1). This supports and encourages the implementation of external communication libraries, such as LibTopoMap or LibNBC.

3.1 Where it Breaks

MPI's support for parallel libraries is comprehensive. However, library writers have to exercise care when using several functionalities in MPI and define external contracts with the users of the library.

The most prominent example is multi-threading. If a library requires a certain thread level, e.g., MPI_THREAD_MULTIPLE, then the user must ensure that MPI is initialized with this thread level. This can be tricky if multiple parallel libraries are used in a single program and can lead to performance degradation if the thread support is only needed for small parts of the code.

A second limitation applies to the MPI_Info values that can be specified during the creation of several objects. Specified values cannot be queried or reset by libraries and need to be communicated out-of-band or enforced via external "contracts". This can influence performance, or even correctness if specified info arguments change the object's semantics (e.g., `no_locks`).

In the following sections, we discuss several application-specific issues and limitations that library-writers may be confronted with.

3.2 Reentrant Libraries

A parallel library invocation will be passed a communicator argument that indicates the group of processes performing the call. A well designed library will pass this communicator as an explicit argument. The library needs a communication context that is distinct from the communication contexts of the invoking code. This is usually done by creating a communicator private to the library that is a duplicate of the argument communicator shows how this private communicator can be cached with the communicator argument, so that the private communicator is created only at the first invocation.

This method provides a static communication context, shared by all library instances. It ensures that sends inside the library cannot match receives outside the library, and vice-versa; but it does not ensure that a send performed by one instance of the library be matched by a receive in another instance.. Such a library is *nonreentrant*: it requires that only one invocation instance be active on a communicator at the same time (no recursion, no new invocation before a previous one completed at all processes). : One can build reentrant libraries in various ways: E.g., by having a barrier, either at entry or at exit (which may have severe impact in performance due to unnecessary synchronization), by creating a new communicator instance for each invocation (several communicators could be pre-dup'd and managed in a stack-like manner as attributes), or by imposing a communication discipline that avoids out-of-order message matching: no wildcard source receives (MPI_ANY_SOURCE), no cancel operations and messages produced within a dynamic scope are consumed within the same dynamic scope.

3.3 Nonblocking Libraries

Nonblocking or asynchronous libraries pose the challenge of progress and control transfer. We differentiate between "manual" progress (the user periodically transfer control to the library, for the library to progress, cf. coroutines) and "asynchronous" progress (the library spawns an asynchronous activity, e.g., a thread) [8]. Manual progress is required on some HPC systems because of limitations on multithreading, limitations on signaling between the communication hardware and (user) threads, and lack of an appropriate scheduling policy.

MPI-2.2 offers generalized requests to integrate completion checks of operations in nonblocking library routines with the usual MPI completion calls (e.g., MPI_Test). However, the specification requires asynchronous progress and does not work on systems where manual progress is needed. A simple fix for this, which adds manual progress facilities to generalized requests, has been proposed for MPI-3 [13].

The parallel invocation method described in Section 3.3 requires that the communicator argument be duplicated, at least at the first library invocation on the communicator. However, MPI_Comm_dup is a blocking collective routine and may require synchronization. This makes it impossible to implement pure nonblocking collective libraries. The alternative of Initializing each communicator before using it with a library is an unnecessary burden for library users even though it is common practice (e.g., in ScaLAPACK). A nonblocking MPI_Comm_dup call would solve this problem.

3.4 Complex Communication Operations

Libraries are often used to implement new, higher-level communication operations. We already discussed issues with nonblocking interfaces of such libraries, however, implementers need to consider two more potential hurdles.

If the library on top of MPI-2.1 was to perform a reduction with either a predefined or user-defined MPI operation, then the library needed to implement the reduction operation itself (since the new library cannot access the function pointer associated with an MPI_Op). MPI-2.2 introduces MPI_Reduce_local to solve this problem. MPI_Reduce_local performs a single binary reduction with an MPI_Op handle as it would be performed by a collective reduction operation. It is recommended to use this functionality to implement reduction communication operations, such as nonblocking MPI_Reduce on top of MPI.

3.5 Process Synchronization Outside of MPI

The MPI standard does not specify the interaction of MPI with other, potentially synchronizing, communication mechanisms outside of MPI. This can pose problems when such operations are mixed, e.g., if communication libraries are tuned for low-level transport interfaces [9]. The implementer has to ensure that all communication interfaces make progress. However, MPI may require manual progress but does not offer an explicit progress call. This may be emulated (rather inelegantly) by calling MPI_Iprobe in a progress loop.

4 Hybrid Programming

Hybrid Programming mixes MPI with other programming models such as Pthreads, OpenMP, or PGAS models. The implementations of runtimes for those models often use external communication layers and may suffer from issues discussed in Section 3.5. However, the interaction between different parallelization schemes can have more complex effects. We discuss two issues with the interaction of MPI and threads. We remark that the discussion is not limited to threads and applies to other models, such as PGAS, or languages, such as C#.

4.1 Thread-safe Message Probing

MPI offers a mechanism (MPI_Probe/MPI_Iprobe) to peek into the receive queue and query the size of found messages before posting the receive. This enables the reception of dynamically-sized messages. However, this also creates problems in the context of multiple threads [5] since one thread can query the message and another thread can receive it (the queue is a global shared object). A matched probe call that removes the message from the queue while peeking has been proposed to MPI-3 to solve this problem [7]. This addition enables low-overhead probing for threaded libraries and languages.

4.2 Control Transfer and Threading

Threaded libraries pose additional problems for the interfaces. This is because threaded libraries encapsulate resource requirements in addition to functionality.

For single-threaded libraries, the control is handed from a single thread running on a single processing element (PE) to a single thread. In multi-threaded environments, we differentiate four scenarios:

1. A single application thread calls a single-threaded library.
2. A single application thread calls a multi-threaded library.
3. Multiple application threads call a single-threaded library.
4. Multiple application threads call a multi-threaded library.

Scenario 1 is identical to the single-threaded case while all other scenarios require some kind of resource management. Scenario 2 is simple because the library is the only consumer of PE resources, while Scenario 3 can solved by synchronizing all threads before the library is called (this is commonly used today, e.g., in [15]). Scenario 4 is most tricky and requires advanced resource management.

Resource management can either be performed by the operating system (time multiplexing) or explicitly by the user with ad-hoc mechanisms such as querying the number of available cores and thread-pinning. A promising OS-based space multiplexing (core allocation) approach is proposed in [19].

4.3 Communication Endpoints

Special care has to be taken if the communication layer requires multiple client threads per node in order to achieve full performance. This has to be addressed in hybrid programming by either using multiple threaded MPI processes per node, or a scheme similar to Scenario 4. A proposal for MPI-3.0 [22] shows an extension for MPI to provide multiple logical network endpoints in a threaded hybrid MPI application.

5 Guidelines for Library Designers

We now conclude this work by providing some hints and guidelines for MPI library developers. All those guidelines are in addition to the well-known serial library design rules, such as privatization and abstraction. In general, libraries should utilize the features provided by MPI while paying attention to the pitfalls discussed above. In particular, libraries should use communicators to specify spatial decomposition of the process space and to present safe communication contexts for temporal decomposition. Created communicators and library internal state and data-structures should be cached with the user communicator (which then becomes the central communication object that needs to be passed to every library call, special care has to be taken for reentrant libraries, cf. Section 3.2). Libraries should take advantage of virtual topologies to specify process topologies and possibly perform topology mapping (this may conflict with the user program or other stacked libraries). If library-specific structures are passed to communication functions and from or to the user, then those should be specified with MPI datatypes. Parallel libraries should also handle errors internally and provide library-specific error messages to the user. This can be achieved by attaching a library-specific error handler to the library's private communicator.

Library and communicator initialization can either be done explicitly or implicitly (at first invocation). Communicator initialization must be done collectively and we discuss issues with nonblocking communication in Section 3.3.

5.1 What to Avoid!

Libraries should never use the passed communicators directly (just attached attributes); this includes the global communicator MPI_COMM_WORLD. Synchronization or draining messages at entry or exit from a library call may impose unnecessary overheads and should be avoided. Libraries also don't need to limit themselves to disjoint process groups. Overlapping communicators are managed well within MPI.

5.2 Progress

There is no generally good strategy for highly-performance library progress: The use of asynchronous progress may be too inefficient or even impossible, while the use of manual progress breaks isolation and may lead to deadlock when multiple libraries are composed, with no systematic use of manual progress at each interface. Thus, progress should be ensured for each library separately. Also, repeated library invocations for manual progress add a superfluous overhead on systems with asynchronous progress. The cleaner solution would be to provide adequate asynchronous progress on all systems. Baring this, it is very desirable to provide manual progress calls that are macro-expanded into noops on systems that do not need them.

6 Summary and Conclusions

In this paper, we showed principles for designing parallel libraries, described a taxonomy of existing libraries and several library examples. We then derived general requirements for parallel libraries and described how they are supported in MPI. Furthermore, we show issues with the current MPI specification that may present pitfalls to developers. Finally, we summarize current practice and good practices for designing parallel libraries.

We conclude that MPI is very well suited to support the development and use of parallel libraries. It offers mechanisms for space- and time-multiplexing processes and an object-oriented interface. It is crucial that other parallel programming environments, such as upcoming PGAS languages, provide a similar level of support for library development.

Acknowledgments This work was supported by the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number OCI 07-25070) and the state of Illinois.

References

1. Balay, S., Gropp, W.D., McInnes, L.C., Smith, B.F.: Efficient management of parallelism in object-oriented numerical software libraries, pp. 163–202 (1997)

2. Basili, V.R., Briand, L.C., Melo, W.L.: How reuse influences productivity in object-oriented systems. Commun. ACM 39, 104–116 (October 1996)
3. Folk, M., Heber, G., Koziol, Q., Pourmal, E., Robinson, D.: An overview of the HDF5 technology suite and its applications. In: Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases. pp. 36–47. AD '11, ACM (2011)
4. Gregor, D., Lumsdaine, A.: Lifting sequential graph algorithms for distributed-memory parallel computation. In: Proceedings of OOPSLA'05. pp. 423–437 (2005)
5. Gregor, D., Lumsdaine, A.: Design and implementation of a high-performance MPI for C# and the common language infrastructure. In: Proceedings of PPoPP'08. pp. 133–142. ACM, New York, NY, USA (2008)
6. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: portable parallel programming with the message-passing interface. MIT Press, Cambridge, MA, USA (1994)
7. Hoefler, T., Bronevetsky, G., Barrett, B., de Supinski, B.R., Lumsdaine, A.: Efficient MPI Support for Advanced Hybrid Programming Models. In: EuroMPI'10. vol. LNCS 6305, pp. 50–61. Springer (Sep 2010)
8. Hoefler, T., Lumsdaine, A.: Message Progression in Parallel Computing - To Thread or not to Thread? (09 2008), accepted at the Cluster 2008 Conference
9. Hoefler, T., Lumsdaine, A.: Optimizing non-blocking Collective Operations for InfiniBand. In: Proceedings of IEEE IPDPS'08 (2008)
10. Hoefler, T., Snir, M.: Generic Topology Mapping Strategies for Large-scale Parallel Architectures. In: Proceedings of ICS'11. pp. 75–85. ACM (Jun 2011)
11. Hoefler, T., Lumsdaine, A., Rehm, W.: Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In: Proceedings of SC'07 (2007)
12. Korson, T., McGregor, J.D.: Technical criteria for the specification and evaluation of object-oriented libraries. Softw. Eng. J. 7, 85–94 (March 1992)
13. Latham, R., Gropp, W., Ross, R.B., Thakur, R.: Extending the MPI-2 Generalized Request Interface. In: EuroPVM/MPI'07. LNCS, vol. 4757, pp. 223–232 (2007)
14. Lumsdaine, A., Mccandless, B.C.: Parallel extensions to the matrix template library. In: Parallel Processing for Scientific Computing (1997)
15. Lusk, E.L., Pieper, S.C., Butler, R.M.: More scalability, less pain: A simple programming model and its implementation for extreme computing. In: SciDAC Rev. 17. pp. 30–37 (2010)
16. Meyer, B.: Lessons from the Design of the Eiffel Libraries. Commun. ACM 33(9), 68–88 (1990)
17. Mohagheghi, P., Conradi, R., Killi, O.M., Schwarz, H.: An empirical study of software reuse vs. defect-density and stability. In: Proc. of ICSE'04. pp. 282–292 (2004)
18. 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`
19. Pan, H., Hindman, B., Asanović, K.: Lithe: enabling efficient composition of parallel libraries. In: HotPar'09. pp. 11–11 (2009)
20. Sbalzarini, I.F., Walther, J.H., Bergdorf, M., Hieber, S.E., Kotsalis, E.M., Koumoutsakos, P.: Ppm: a highly efficient parallel particle-mesh library for the simulation of continuum systems. J. Comput. Phys. 215, 566–588 (July 2006)
21. Skjellum, A., Doss, N., Bangalore, P.: Writing libraries in mpi. In: Scalable Parallel Libraries Conference, 1993., Proceedings of the. pp. 166 –173 (oct 1993)
22. Snir, M.: Endpoint proposal for mpi-3.0. Tech. rep. (2010)
23. Willcock, J., Hoefler, T., Edmonds, N., Lumsdaine, A.: AM++: A Generalized Active Message Framework. In: Proccedings of ACM PACT'10 (2010)
24. Willcock, J., Hoefler, T., Edmonds, N., Lumsdaine, A.: Active Pebbles: Parallel Programming for Data-Driven Applications. In: Proc. of ACM ICS'11. pp. 235–245 (2011)