A NEW APPROACH TO MPI COLLECTIVE COMMUNICATION IMPLEMENTATIONS

Torsten Hoefler,[1,4] Jeffrey M. Squyres,[2] Graham Fagg,[3] George Bosilca,[3] Wolfgang Rehm,[4] and Andrew Lumsdaine[1]

[1]*Indiana University, Open Systems Lab, Bloomington, IN 47404 USA*

{htor,lums}@cs.indiana.edu

[2]*Cisco Systems, San Jose, CA 95134 USA*

jsquyres@cisco.com

[3]*University of Tennessee, Dept. of Computer Science, Knoxville, TN 37996 USA*

{fagg,bosilca}@cs.utk.edu

[4]*Technical University of Chemnitz, Dept. of Computer Science, Chemnitz 09107 Germany*

{htor,rehm}@cs.tu-chemnitz.de

Abstract

Recent research into the optimization of collective MPI operations has resulted in a wide variety of algorithms and corresponding implementations, each typically only applicable in a relatively narrow scope: on a specific architecture, on a specific network, with a specific number of processes, with a specific data size and/or data-type – or any combination of these (or other) factors. This situation presents an enormous challenge to portable MPI implementations which are expected to provide optimized collective operation performance on *all* platforms. Many portable implementations have attempted to provide a token number of algorithms that are intended to realize good performance on most systems. However, many platform configurations are still left without well-tuned collective operations. This paper presents a proposal for a framework that will allow a wide variety of collective algorithm implementations and a flexible, multi-tiered selection process for choosing which implementation to use when an application invokes an MPI collective function.

Keywords: Collective Operation, Message Passing (MPI), Automatic Selection, Framework, Open MPI

1. Introduction

The performance of collective operations is crucial for the runtime and scalability of many applications [Rabenseifner, R., 1999]. Decades of collective communication research have yielded a wide variety of algorithms tuned for specific architectures, networks, number of participants, and message sizes. The choice of optimal algorithm to use therefore not only depends on the system that the application is running on, but also the parameters of the collective function that was invoked (e.g., number of peers, data size, data type). The sheer number of algorithms available becomes a fundamental problem when optimizing a portable Message Passing Interface (MPI) library – how should it choose which algorithm to use at runtime?

Our work aims at providing the capability to automatically select the optimal collective implementation for each system and MPI argument set. Such an approach can potentially result in a large performance gain for each collective function invocation [Pjesivac-Grbovic, J. et. al., 2005; Hoefler, T. et. al., 2005; Mitra et al., 1995].

Predictive performance models of point to point communications (such as LogP [Culler, D. et. al., 1993] or LogGP [Alexandrov, A. et. al., 1995]) can return a reasonable approximation of collective communication performance upon which we can base the selection of the collective implementation. Hence, invoking modeling functions at runtime to estimate the algorithm performance is one approach to determine which should be used.

However, such modeling techniques are not necessarily relevant for hardware-assisted collective operations (or other implementations not based on point-to-point operations). Indeed, hardware-based collectives typically outperform even the best software-based algorithms; it is a reasonable optimization to directly invoke available hardware-based collectives and bypass any modeling evaluation.

Based on these considerations and the ideas proposed in [Squyres, J. M. et. al., 2004], we present the design of a next-generation collective framework with the following goals:

1. Enable fine-grained algorithm selection such that a selection atom is an individual function.
2. Perform efficient run-time decisions based on the MPI function arguments.
3. Enable a “fast path” for trivial decisions (e.g., hardware implementations).
4. Enforce a modular approach, preserving the simplicity of adding (and removing) algorithms – especially by third parties.
5. Enable *all* algorithms – even those added by third parties – to be automatically used by user applications, testing, and benchmarking tools.

The rest of this paper is divided as follows: Section 2 discusses related work. Section 3 describes the architecture of our approach. The logic for selecting which algorithm to use is described in Section 4, followed by an analysis of its applicability to a set of real world applications. The last section draws conclusions and points out further work.

2. Related Work

Many research groups inherently limit the selection problem by implementing only a subset of the standardized collective operations to fit their particular needs and assume that those algorithms are globally applicable [Huse, 1999; Chan, E.W. et. al., 2004]. Some MPI implementations (as MPICH [Gropp, W. et. al., 1996], MPICH2 [MPICH2 Developers, 2006], LAM/MPI [Burns, 1994]) do their selection of the collective implementation to use either statically at compile time or based on a limited number of arguments at runtime. The selection decision is typically based on the communicator and/or data size and does not take into account network characteristics (such as bandwidth and/or latency) and ignores the physical network topology.

FT-MPI [Fagg, G.E. et. al., 2004] and current generations of Open MPI [Gabriel and et al., 2004] base their decisions on an augmented set of parameters which include the network characteristics. However, in order to make the right selection, a decision table must be built prior to the execution by a benchmarking tool. This input file has to be generated by an external tool after running intensive set of benchmarks. The cost of building the decision table on the full set of possible combination of arguments can be prohibitive (especially for large clusters); a subset of all available nodes and/or algorithms may need to be used, leading to the selection of a sub-optimal algorithm in some cases. Even though this approach can provide an increase in performance, it is difficult to add a new algorithm since both the decision function and the benchmark tool have to be modified in order to include the new algorithm.

Similar modular approaches were described by Vadhiyar et. al. [Vadhiyar, S.S. et. al., 2000] and Hartmann et. al. [Hartmann, O. et. al., 2006]. Both propose methods and show the potential benefits of selecting between collective algorithms during runtime. However, these approaches are limited to a small set of implemented algorithms and not easily extensible by third party implementers.

3. Framework Architecture

We propose a hierarchical framework architecture composed of collective components, collective modules and collective functions. A collective component is the software entity which is provided by the module implementer and it generates communicator specific modules on request (called query in the following). Each component is loaded, queried and unloaded by the framework. A collective module is a software instance of a collective operation bound to a specific communicator. A collective component may return an arbitrary number of collective modules during the query. A collective module may have one or more (opaque) collective functions to perform the collective operation available. Additionally, each module defines an evaluation function which returns a collective function pointer and an estimated time for each MPI argument set.

We divide the architecture into three main parts. The software architecture defines the nesting of software entities. The usage and interaction of the software entities during the program run is defined in the runtime architecture. The decision architecture, which can be omitted with the "fast path", defines the decision logic used during function invocation and possible optimizations.

Software Architecture

The software architecture is explained by example in Fig. 1. This example shows only a subset of all available collectives. However, a basic implementation of all collectives is provided with the framework, therefore at least one collective function is available at any time. The example shows two available collective components, called "Component A" and "Component B". Both components are loaded by the framework during start-up and maintained on

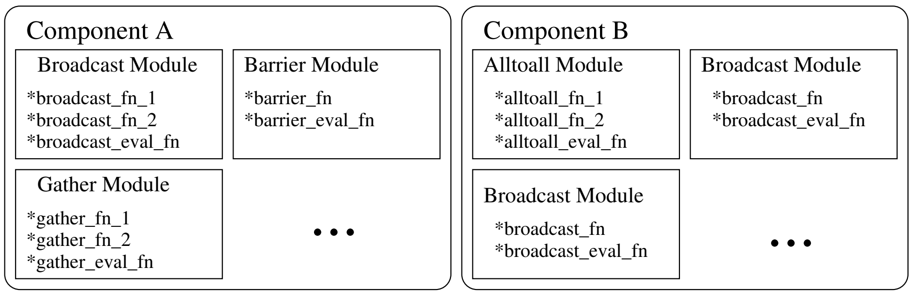

Figure 1. Software Architecture

a list of active components. The initialization of the framework during start-up (MPI_INIT), where all available components are loaded and initialized, is shown in Fig. 2. The user can select specific components via framework pa-

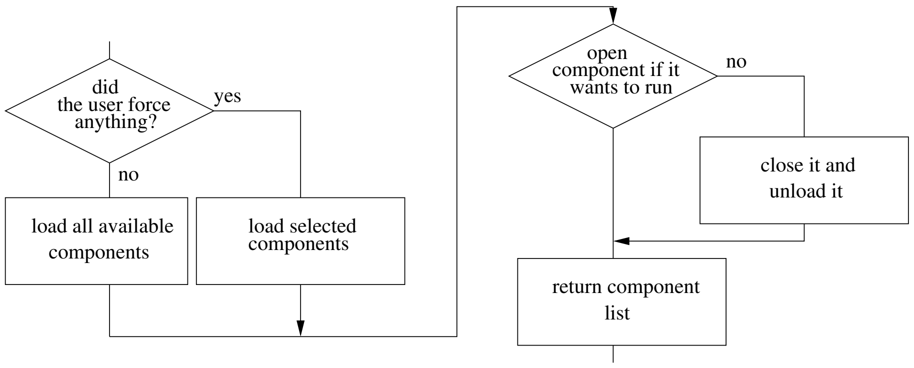
did the user force anything?
yes
no
load all available components
load selected components
open component if it wants to run
no
close it and unload it
return component list

Figure 2. Actions during MPI_INIT

rameters. Each loaded component may disable itself during start-up if not all requirements (e.g., special hardware) are met. Fig. 1 shows that implementations for MPI_BCAST, MPI_BARRIER, MPI_GATHER, and MPI_ALLTOALL are available to the framework. All available components are queried with each communicator during its construction, including the default communicators MPI_COMM_WORLD and MPI_COMM_SELF. This procedure is shown in Fig. 3. Each component returns an array of available modules to the frame-

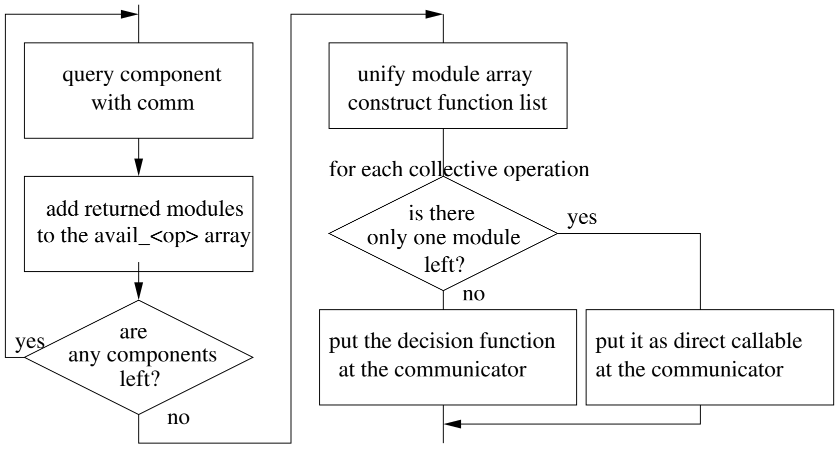
query component with comm
add returned modules to the avail_<op> array
are any components left?
yes
no
unify module array construct function list
for each collective operation
is there only one module left?
yes
no
put the decision function at the communicator
put it as direct callable at the communicator

Figure 3. Actions during Communicator Construction

work which adds the modules to a list of runnable modules (`avail_<op>`) on each communicator. A unification, typically represented by a global operation, of this list ensures that all selectable modules are available on all nodes of this communicator (some of them may not have the right hardware requirements). Finally, the runtime architecture of this communicator is initialized by the framework. This architecture is described in detail in the next section.

Runtime Architecture

Each instantiated module offers an evaluation function to the framework. This evaluation function returns the function pointer to the fastest internal implementation. This means that more than one implementation may exist inside a single module. Our example in Fig. 1 depicts a single MPI_BCAST implementation and two opaque MPI_BCAST functions implemented in “Component A”. This shows that the module is allowed to implement opaque functions and to select between them independently of the framework. This offers the possibility to implement a more sophisticated selection inside a single module if the module implementer is able to simplify the decision. This reduces the number of modules, the memory footprint, and the decision costs which are discussed later. However, the component is free to return any number of collective modules for a single collective operation. So does “Component B” and offers two distinct MPI_BCAST implementations which can be turned into two MPI_BCAST modules.

Fig. 4 shows the runtime architecture for two communicators, “Communi-

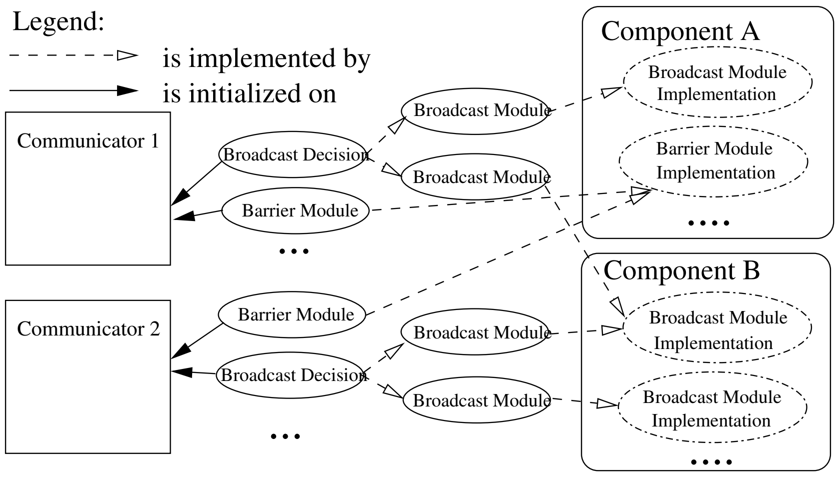

Figure 4. Runtime Architecture

cator 1" and "Communicator 2". All modules returned by queried components are attached to communicator which was used to query the component. The framework maintains a communicator-specific list of available modules per collective operation. Each module implements a single collective operation which meets the fine grained selection criterion in goal 1. The dashed arrows in Fig. 4 point to the collective implementation in the "Component A" or "Component B" component which acts as a code-base for the collective module. This shows that each component can create multiple modules which can be attached to different communicators. Each communicator can manage an arbitrary number of collective modules to perform a collective operation. The module to process a specific collective call is selected depending on the actual MPI arguments during invocation. However, the collective function is called directly if there is only a single module available, or a single module is enforced by the user (cf. Fig. 3). This direct invocation is called "fast path" as it does not introduce any additional overhead.

4. Selection Logic

The example in Fig. 4 shows that there is only a single MPI_BARRIER and MPI_GATHER module available for "Communicator 1". As a result, both operations are called directly using the "fast path" without any selection overhead. However, there are two MPI_BCAST implementations available for this communicator which means that there has to be some intermediate layer to select one of those depending on the arguments. This layer is called selection logic and is implemented in a set of MPI operation specific decision functions. The "fast path" enforces that the function arguments of these decision functions are identical to the actual arguments of the collective functions because the upper layer is not aware of the selection logic. This means that the call to the decision function is completely transparent to the upper layer. The selection logic with the MPI_BCAST decision function is shown in Fig. 4 and the actions performed during the invocation of a collective operation are shown in Fig. 5. The first

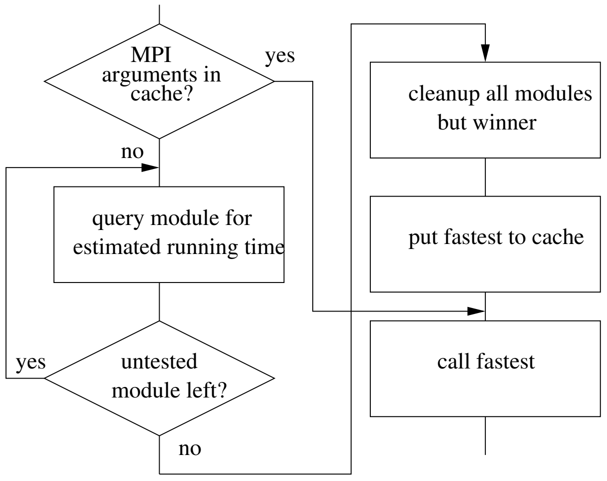

Figure 5. Actions during a collective function call

action is to check if these arguments have already been issued and if the decision result is in the cache. If this is true, the whole decision functionality and the related overhead can be skipped and the fastest function is called directly via its cached function pointer. However, if the arguments have not been called before (or have been evicted from the cache to free memory), the selection needs to be performed for the particular argument set. This means that all runnable modules (modules in the `avail_<op>` array at the communicator) are queried for their estimated running time. The module that returns the lowest running time is added to the cache for future calls and invoked to perform the collective operation.

The decision function performs the MPI argument specific selection of collective modules based on querying the evaluation function of each module. The module's evaluation function returns an estimated time in microseconds and a function pointer to its fastest function. Absolute time was chosen as an evaluation criterion because it denotes the least common denominator for our case. This enables the component author to predict or benchmark the running time of all possible collective implementations no matter if it is performed hardware supported or simply on top of point-to-point messages. It is obvious that querying all available modules each time a collective call occurs is extremely costly and can have a catastrophic impact at the application performance. The decision function implements an MPI-argument specific cache which stores the collective function pointer to speed up the critical path to reduce the number of the costly queries. The fastest collective function pointer is added to the cache and called after each evaluation. This introduces two questions: How much overhead does the evaluation add to the collective latency and how cache friendly will an application really be. The overhead of the evaluation and the cache friendliness of three MPI applications are analyzed in Section 5

For example, a direct call occurs to Component A's Barrier Module if the application calls `MPI_Barrier(Communicator 1)`. This shows the "fast path" which is enabled for the barrier call on Communicator 1. The next MPI call of the application is `MPI_Bcast(sbuf, 1, MPI_INT, 0, Communicator 1)` which uses the decision function. This arguments are not yet in the cache (i.e., have not been called before). The decision function queries both Broadcast Modules of Component A and Component B for their fastest function (-pointer) and its estimated running time. The function pointer of the fastest function is inserted into the cache and it is called to perform the collective. If another call to `MPI_Bcast(sbuf, 1, MPI_INT, 0, Communicator 1)` occurs, we already know the fastest function (in the cache) and call it without evaluating all modules. However, if a call to `MPI_Bcast(sbuf, 2, MPI_INT, 0, Communicator 1)` occurs, we have to reevaluate all modules again.

Decision Overhead

The argument cache can be implemented as a collision-free hash-table which has an ideal complexity of $O(1)$. The costly part is if a cache miss occurs (i.e., the called argument set is not in the cache, has not been called before). This results in a serial query to the evaluation functions of all available modules. There are many different ways to implement this evaluation function, we will discuss the costs of two approaches on detail.

Benchmark Based Implementation. The evaluation function could return a time that is based on an actual benchmark which has previously been run on the system. We assume that the benchmark data has a small memory demand and was loaded during startup. The cost will be approximately a indirect function call and several cache misses. The indirect function call costs has been evaluated in [Barrett, B. et. al., 2005] and turns out to be between $2ns$ and $10ns$. We implemented a simple x86 RDTSC based micro benchmark to measure cache miss penalty which was between $0.5\mu s$ and $1.5\mu s$ on all evaluated architectures (Opteron 2.0 GHz, Xeon 2.4 GHz, Athlon MP 1.4 GHz). This shows that each evaluation function call may take some microseconds for a benchmark based implementation.

Model Based Implementation. The time to return could also be calculated using a model function like LogP or LogGP. We can assume that the small set of necessary model parameters are already in the cache. Our micro-benchmark measures access times between $10ns$ and $50ns$ for cached items and a calculation time of $200ns$ up to $500ns$ for the evaluation of a 4th grade polynomial (model function). The overall evaluation should take less than $1\mu s$ for this case.

This shows that well implemented evaluation functions may need up to $5\mu s$ to return the result. This should not hurt the application performance to much, because the expected benefits are higher (previous studies show differences in the millisecond scale for several collective implementations). However, the cache may even speed thing up for repeated arguments. The next section analyzes the cache-friendliness of a small set of applications.

Analyzing the Cache Friendliness

The usage of the cache (i.e., hit and miss rates) are not easily predictable because they depend entirely on the application. We measured two different applications to measure their cache friendliness. The first is ABINIT (`http://www.abinit.org`) which offers two distinct parallelization schemes, band parallelization and CG+FFT parallelization. The second application is CPMD (`http://www.cpmd.org/`) which is used in its standard configuration. Both applications have been run with a real-world input file and a special library which logs collective calls using the MPI profiling interface. ABINIT issues 295 collective operation calls with 16 different parameter sets (hit rate: 94.6%) for band parallelization. The CG+FFT parallelization uses 53887 collective operations with 75 different argument sets (hit rate: 99.9%). CPMD issues 15428 collective operations with 85 different argument sets (hit rate: 99.4%). Both applications utilize the cache very efficiently.

5. Conclusion and Future Work

We have shown that our new design to select collective implementations during runtime is able to support all kinds of possible collective function implementations. We have also shown that the idea of the MPI argument cache to store the optimal selection will work well with at least some real world application. It is possible to disable the whole selection logic and call every operation via the “fast path”. The selection logic enables scientists to add new collective functionality easily and to use it also in productive environments. Next steps will include the implementation and testing of the proposed approach and the analysis of more real applications for their argument cache friendliness.

Acknowledgments

This work was supported by a grant from the Lilly Endowment and National Science Foundation grant EIA-0202048.

References

Alexandrov, A. et. al. (1995). LogGP: Incorporating Long Messages into the LogP Model. *Journal of Parallel and Distributed Computing*, 44(1):71–79.

Barrett, B. et. al. (2005). Analysis of the Component Architecture Overhead in Open MPI. In *Proc., 12th European PVM/MPI Users' Group Meeting*.

Burns, G. et. al. (1994). LAM: An Open Cluster Environment for MPI. In *Proc. of Supercomputing Symposium*, pages 379–386.

Chan, E.W. et. al. (2004). On optimizing of collective communication. In *Proc. of IEEE International Conference on Cluster Computing*, pages 145–155.

Culler, D. et. al. (1993). LogP: towards a realistic model of parallel computation. In *Principles Practice of Parallel Programming*, pages 1–12.

Fagg, G.E. et. al. (2004). Extending the MPI specification for process fault tolerance on high performance computing systems. In *Proceedings of the International Supercomputer Conference (ICS) 2004*. Primeur.

Gabriel, Edgar and et al. (2004). Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary.

Gropp, W. et. al. (1996). A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828.

Hartmann, O. et. al. (2006). A decomposition approach for optimizing the performance of MPI libraries. In *Proc., 20th International Parallel and Distributed Processing Symposium IPDPS*.

Hoefler, T. et. al. (2005). A practical Approach to the Rating of Barrier Algorithms using the LogP Model and Open MPI. In *Proc. of the 2005 International Conference on Parallel Processing Workshops (ICPP'05)*, pages 562–569.

Huse, Lars Paul (1999). Collective communication on dedicated clusters of workstations. In *Proc. of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in PVM and MPI*, pages 469–476.

Mitra, Prasenjit, Payne, David, Shuler, Lance, van de Geijn, Robert, and Watts, Jerrell (1995). Fast collective communication libraries, please. Technical report, Austin, TX, USA.

MPICH2 Developers (2006). `http://www.mcs.anl.gov/mpi/mpich2/`.

Pjesivac-Grbovic, J. et. al. (2005). Performance Analysis of MPI Collective Operations. In *Proc. of the 19th International Parallel and Distributed Processing Symposium*.

Rabenseifner, R. (1999). Automatic MPI counter profiling of all users: First results on a CRAY T3E 900-512. In *Proc. of the Message Passing Interface Developer's and User's Conference*, pages 77–85.

Squyres, J. M. et. al. (2004). The component architecture of Open MPI: Enabling third-party collective algorithms. In *Proc. 18th ACM International Conference on Supercomputing, Workshop on Component Models and Systems for Grid Applications*, pages 167–185.

Vadhiyar, S.S. et. al. (2000). Automatically tuned collective communications. In *Proc. of the ACM/IEEE conference on Supercomputing (CDROM)*, page 3.