Flexible collective communication tuning architecture applied to Open MPI

Graham E. Fagg[1], Jelena Pjesivac-Grbovic[1], George Bosilca[1], Thara Angskun[1], Jack J. Dongarra[1], and Emmanuel Jeannot[2]

[1] Dept. of Computer Science, 1122 Volunteer Blvd., Suite 413, The University of Tennessee, Knoxville, TN 37996-3450, USA
[2] INRIA LORIA, Campus Scientifique, BP 239, 54506 Vandœuvre les Nancy, France

1 Abstract

Abstract. Collective communications are invaluable to modern high performance applications, although most users of these communication patterns do not always want to know their inner most working. The implementation of the collectives are often left to the middle-ware developer such as those providing an MPI library. As many of these libraries are designed to be both generic and portable the MPI developers commonly offer internal tuning options suitable only for knowledgeable users that allow some level of customization. The work presented in this paper aims not only to provide a very efficient set of collective operations for use with the Open MPI implementation but also to make the control and tuning of them straightforward and flexible. Additionally this paper demonstrates a novel example of the proposed frameworks flexibility, by dynamically tuning a MPI_Alltoallv algorithm during runtime.

2 Introduction

Collective (group) communications are of paramount importance to HPC users due to the extent on which developers rely on them for optimal performance[8]. In many cases obtaining optimal performance requires deep internal knowledge of the collective algorithms and the target architectures which many users may not either have access to or have no understanding of. The reasons for these gaps are many. The implementation of the collectives are often left to the middleware developers such as those providing an MPI library. As many of these libraries are designed to be both generic and portable the MPI developers are left in the difficuilt position of deciding just how to implement the basic operations in such a way that they meet the needs of all possible users without knowing just how they will be utilised.

Previous generations of collective implementations usually offered a number of possibly optimal low level implementations and some kind of a fixed decision on when to use one version or the other, in a hope that this would cover most usage cases. Although much pervious work has focused on either measurement (instrumentation) or modelling to make these decisions (ACCT, ATCC OCC, LogGP, Magpie etc) rather than on how to either incorporate them into a runtime system, or make them more accessable.

In many cases making the underlying decisions accessable either directly to knowledgable users, or via automated tools is enought to correct for any [performance] problems with the default decisions implemented by the MPI implementors.

This paper describes current work on the tuned collectives module developed by the University of Tennessee for distribution within the Open MPI 1.1 release. Some sections of the research shown here (i.e. dynamic rule bases) are still however experimental and may never be officially distributed with Open MPI. This paper is ordered as follows: Section 3 detailed related work in collective communications and control. Section 4 details the Open MPI MCA architecture and Section 5 describes the tuned collectives component design and performance, section 6 shows how dynamic rules can be used to implement runtime tuning and section 7 concludes the paper and lists future work. Results are included in each of the relavent sections.

3 Related work

All MPI implementations support MPI collective operations as defined in the MPI 1.2 specification [7]. Many of the portable implementations support a static choice mechanism such as LAM/MPI, MPICH [5], FT-MPI [4] and the basic collectives component [13] of Open MPI [10]. In many cases these implementations are tuned primarily for closely coupled systems and clusters and the decision functions are buried deep inside the implementations. System that are designed for Grid and wide-area use also have to differentiate between various collective algorithms but at a much higher level, such as when implementing hierarchical topologies to hide latency effects. Systems such a Magpie [11], PACX-MPI [3, 2] and MPICH-G2 all use various metrics to control which algorithms are used. Although these systems do not explicitly export control of these parameters to users, their code structure does allow these points to be more easily found than with closely coupled systems.

4 Current collectives framework in Open MPI

4.1 Open MPI collective framework and basic components

The current Open MPI[10] architecture is a component based system, and is called the Modular Component Architecture (MCA). The MCA architecture was designed to allow for a customized (and optimized) MPI implementation that is built from a range of possible components at runtime, allowing for a well architect ed code base that is both easy to test across multiple configurations and easy to integrate into a new platform. The architectures design is a follow up to the SSI system[12] originally developed for the LAM7. The system consists of a MCA framework which loads components (shared objects) during MPI_Init.

If any of these components can be utilized (they can disqualify themselves via a query function) they become modules (a component instance coupled with resources such as allocated memory). Many of the subsystems within Open MPI such as low level point-to-point messaging, collective communication, MPI-2 I/O, and topology support are all built as components that can be requested by the user at MPIRUN time.

The Open MPI 1.0 release supplied a number of MPI components for collective communication that each contained a complete set of MPI 1.2 collective operations. The components being: *basic*, *shm* and *self.*

The shm component contains collectives for use when Open MPI is running completely on a shared memory system. The self component is a special feature within MPI for use on the MPI_COMM_SELF communicator. The basic component is the default component used when not using either shared memory or self. The basic component contains at least one implementations per collective operation. For broadcast and reduce it offers two implementations, one linear and the other using a binary tree. Further details of the Open MPI collective framework can be found in [13].

5 New tuned collectives and decision module

The new tuned collectives module from the University of Tennessee, Knoxville (UTK) has a number of goals, and aims to support the following:

1. Multiple collective implementations
2. Multiple logical topologies
3. Wide range fully tunable parameters
4. Efficient default decisions
5. Compile new decision functions into the component
6. Specify a simple decision map file selectively for any parameter set
7. Provide a means to dynamically alter a decision function completely

Items (1-3) are paramount for any collective implementation to be able to provide performance on an unknown system that the developer has no direct access to. Item (4) is required to allow users to just down load and install Open MPI and get reasonable performance. Item (5) is for more knowledgeable users who wish to change the default decision and allow for the fastest use of that new decision without fully replacing the current default rules. If a comprehensive benchmarking of the Open MPI collectives module has been completed, then the output from this could be feed back into the MPI runtime (item 6) and used instead of the default rule base. The final item is quite unusual and allows for the entire (or part of) the rule base to be changed during runtime. This in effect allows for adaptive behavior of the decision functions. This is explored later in section 6.

5.1 Collective algorithms and parameters

Previous studies of MPI collectives have shown that no single algorithm or topology is optimal and that the variations in network topology, interconnection technology, system buffering and so on, all effect the performance of a collective operation [9]. Thus, the tuned module supports a wide number of algorithms for performing MPI collective operations. Some of these implementations are relay on fixed communication topologies such as the Bruck and recursive doubling, others are general enough to handle almost any topology i.e. trees with varying fan-outs, pipelines etc. Another additional parameter implemented in the tuned collectives module is segment size. In an attempt to increase performance by utilizing as many communication links as possible we have modified most algorithms to segment the users data into smaller blocks (*segments*). This allows the algorithm to effectively pipeline all transfers. The segment size is however not a simple factor of network MTU, sender overhead gap etc, and has to be benchmarked to find optimal values.

5.2 Default tuned decision function

For the module to be named *tuned* implies that it is in fact tuned for some system somewhere. In fact it has been tuned for a cluster of AMD64 processors communicating across a Gigabit Ethernet interconnect located at UTK. The tuning was performed using an exhaustive benchmarking technique as part of the OCC[6] and Harness/FT-MPI[4] projects. (The module shares almost the same decision functions as FT-MPI although they both implement slightly different ranges of algorithms).

A comparison of the tuned component compared to the basic collectives module in Open MPI, MPICH2 and FT-MPI is shown in figures 1 a & b. The first figure shows absolute performances and the second is normalized to the optimal of the 4 systems (i.e. the Y-Axis shows how much slower the others are compared to the best for that message size and operation).

5.3 Architecture and calling sequence

The overall architecture of the tuned collectives component is governed by both the MCA framework and the structure of the communicator data structure. As the upper level MPI architecture calls the function pointer in the communicator directly, this forces the first level function in the component to have the same argument list as that of the MPI API, i.e. no extra arguments. As discussed above many of the backend implementations of the collectives require extra parameters, such as topology and segment size. We resolve this issue by using at least a two level architecture. The first level takes normal MPI arguments, decides which algorithm/implementation to use, creates any additional parameters and then invokes it, passing any results or errors back to the MPI layer. I.e. the first level function is both a decision function and a dispatch function. The second or lower layer is the implementation layer, and contains the actual algorithm themselves.

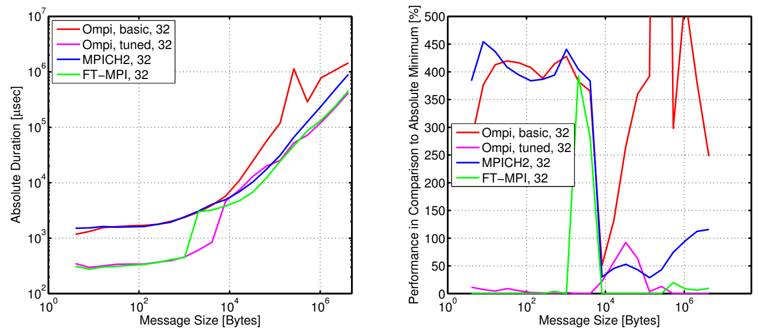

Fig. 1. Absolute and Relative (compared to best) performance of 4 collective implementations, Open MPI (basic,tuned), MPICH2 and FT-MPI

Adding this additional layer of redirection allows the component complete flexibility in how it handles requests, as all functions except the decision/dispatch are hidden from the above layers. The component additionally implements some support functions to create and manage virtual topologies. These topologies are cached on either the component, or on each instance of the module as configured by the user.

5.4 User overrides

One of the goals of the tuned module was to allow the user to completely control the collective algorithms used during runtime. From the architecture it is clear that the upper level MPI API does not offer any methods of informing the component of any changes (except through MPI attributes) as the decision/dispatch function has the same arguments as the collective calls. This issue is resolved by the MCA framework, which allows for the passing of key:value pairs from the environment into the runtime. These values can then be looked up by name.

To avoid incurring any kind of performance penalty during normal usage, these overrides are not checked for unless a special trigger value known as *mca_coll_tuned_use_dynamic_rules* is set. When this value is set, the default compiled in decision routines are replaced by other routines that check for all the possible collective control parameters. To further reduce overheads, these parameters are only looked up at the MCA level during communicator creation time, and their values are then cached on each communicators collective module data segment.

Forcing choices The simplest choice that the user might want is the ability to completely override the collective component and choose a particular algorithm and its operating parameters (such as topology and segmentation sizes) directly. In the tuned component this is known as *forcing* a decision on the component, and it can be performed on as many or as few MPI collectives as required. The example below illustrates how the user can force the use of a Bruck based barrier operation from the command line.

```
host% mpirun -np N -mca coll_tuned_use_dynamic_rules 1
        -mca coll_tuned_barrier_algorithm 4 myapp.bin
```

The range of possible algorithms available for any collective can obtained from the system by running the Open MPI system utility *ompi_info* with the arguments *-mca coll_tuned_use_dynamic_rules 1 -param coll all*. The possible range for an MPI Broadcast is currently:

```
MCA coll: information "coll_tuned_bcast_algorithm_count" (value: "6")
            Number of bcast algorithms available
MCA coll: parameter "coll_tuned_bcast_algorithm" (current value: "0")
    Which bcast algorithm is used. Can be locked down to choice of:
0 ignore, 1 basic linear, 2 chain, 3: pipeline, 4: split binary tree,
5: binary tree, 6: BM tree.
```

It is important to note that the value *0* forces the component to default to the built in compiled decision rules. Further control parameters exist that control both topology and message transfers such as *coll_tuned_bcast_algorithm_segmentsize*, *coll_tuned_bcast_algorithm_tree_fanout* and *X_chain_fanout*. These parameter names are common to most collective operations.

Selective file driven decision functions Another alternative to forcing the complete collective operations is to force only parts of the decision space in a semi-fixed manner. An example of such a usage scenario would be in the case of a user having tuned an MPI collective for a range of input parameters (message size, communicator size) either manually or via some automated tool [6]. The user could then tell the MPI collective component to use these values within a set range by supplying a file that contains as many data points as the user knows. To decrease both storage and evaluation time the file contents are stored using a run-length technique that effectively only stores the switching points for each algorithm. An example version for an MPI Alltoall operation is shown below:

```
1          # num of collectives
3          # ID = 3 Alltoall collective (ID in coll_tuned.h)
2          # number of com sizes
1          # comm size 1
1          # number of msg sizes 1
0 1 0 0       # for message size 0, linear 1, topo 0, 0 segmentation
8          # comm size 8
4          # number of msg sizes
0 1 0 0       # for message size 0, linear 1, topo 0, 0 segmentation
32768  2 0 0  # 32k, pairwise 2, no topo or segmentation
262144 1 0 0  # 256k, use linear 1, no topo or segmentation
524288 2 0 0  # message size 512k+, pairwise 2, topo 0, 0 segmentation
# end of first collective
```

6 Runtime dynamic feedback control

Runtime Controllable Tuned Collectives decision functions The ultimate level of dynamic control is to avoid compiling a decision rule set at all, but instead to construct it fully at runtime. As the rule base is then represented by a set of data structures it can be modified on the fly during runtime (with the appropriate locking in place to avoid race conditions). To allow experimentation with this we developed and implemented a dynamic rule base system around the concept of small expression blocks operating on standardized parameters derived from the collective call arguments. The fundamental premise is that any decision table is really an optimizes set of expressions operating on derived arguments that eventually resolve down to a single function call (or function pointer) and any additional parameters (such as segment size, topology) needed. With this in mind we designed a simple rule building block that operates on standardized parameters.

Standardized parameters are those derived from normal MPI collective call argument lists. We examined our current decision functions to see which parameters are most commonly used. These were found to include: data size (*extent * data count*), location of the root, power of two communicator size and so on. To allow for a rule base to be built using these, each parameter type is associated with a well known constant label that can be used during rule creation and modification time.

To allow us to reason about and with these parameters we also defined a number of standard expression operators, again each with a well known constant name. These include: *LT, LTEQ, GT, GTEQ* and *EQ*, meaning Less Than, Less Than and Equal, and so on. The values the operators compare parameters against can be set at rule creation time and later modified. A special requirement that we demonstrate below is that comparison values can be named and their memory address made available via the MPI Attributes interface.

The rule block itself consists of upto seven parts, the first three being: the standardized parameter, the expression operator and the expression value to compare against. For each of the possible outcomes *true* or *false*, the rule must specify one of two entities, either, another rule to evaluate or a terminating function pointer and its additional argument block. In the case another rule is specified, the system will simply evaluate it and continue on. If otherwise a function pointer is specified together with an argument block (containing segment size, topology information such as fan-out/fan-in) the system will simply call the [collective] function and pass its return code back to the MPI layer.

6.1 Overheads of the rule base

Using dynamic rules with multiple function pointer dereferencing and evaluation of the common parameter block does potentially add some overhead to the critical path of the MPI collective invocation. We build a small benchmark code that exhaustively tested a small rule base by taking MPI parameters to a broadcast and invoking a dummy function. The benchmark kept count of both the average depth and the total time for evaluating 180 thousand input parameters. The benchmark was wrote in two parts, one that used the dynamic rule system and the other that used a compiled *if else* statement block in the C language. Both codes were compiled with the same compiler flags (-O). The results were as follows for a 1.4GHz Pentium 4:

```
Static rules:   63.7 nSeconds
Dynamic rules:  125.7 nSeconds
```

These results were for an average depth of 5 rules or just over 12 nSeconds per rule. Considering that efficient small message collectives on clusters are in the order of tens of microseconds the extra overhead is probably insignificant. If the user supplies a hardware assisted collective then they should probably use a different mechanism to invoke it such a direct function pointer in the communicator structure itself.

6.2 Feedback controlled New Switchable All2allv Algorithm

To test the dynamic rule base in a novel way we implemented a new variant of the MPI ALLTOALL vector operation (*MPI_Alltoallv*). The ALLTOALLV operation allows different sized messages between each peer of nodes. This new variant is a cross between a fully eager ALLTOALL implementation and a pairwise based approach. A fully eager approach makes sense for small messages and small numbers of nodes. Pairwise is efficient on larger message sizes, but the vector nature of ALLTOALLV means that it can potentially contain both very large and very small messages.

At the beginning of the operation, each process checks the data sizes for each transfer with its peers and if the data sizes are below a certain threshold, it sends them immediately via a non blocking send operation (while conversely posting any nonblocking receives for any incoming short messages). Any messages not sent and received during the eager phase are exchanged during a standard pairwise communication pattern. At the end of the operation all outstanding requests are waited on until complete. If the threshold is zero then the algorithm is a pairwise algorithm, if the threshold is as big as the largest message between peers then the algorithm becomes fully eager. Values of threshold in between can reduce either total time to completion or contention within the network, but modeling the optimal is difficult due to the non uniform data sizes possible.

To overcome the problems associated with choosing a fixed threshold we implemented the threshold test within the dynamic rule system and exported the thresholds address via the MPI attribute system. We then implemented a new version of ALLTOALLV at the MPI profiling layer which attempted to tune the underlying collective by testing different values of the threshold. This was performed by using the best value of the threshold for n iterations and then testing a single new value of the threshold. If this value produced a better time that the previous values it would be kept. As each peer of the ALLTOALLV handles different amounts of data it is impossible for each peer to decide locally which value of the threshold we be kept and which will be rejected. Thus at each *test* iteration we nominated the zero rank process to broadcast if the threshold is kept or rejected. Figure [**?**] shows the time per iteration of an ALLTOALLV collective with dynamic tuning of the eager / pairwise threshold. As can be seen, the tuning system does actually make progress in improving the performance of the collective during runtime. As can also be seen, the tuning system must be capable of handling a *noisy* system as many of the techniques used to produce repeatable results during benchmarking cannot be enforced on a working application.

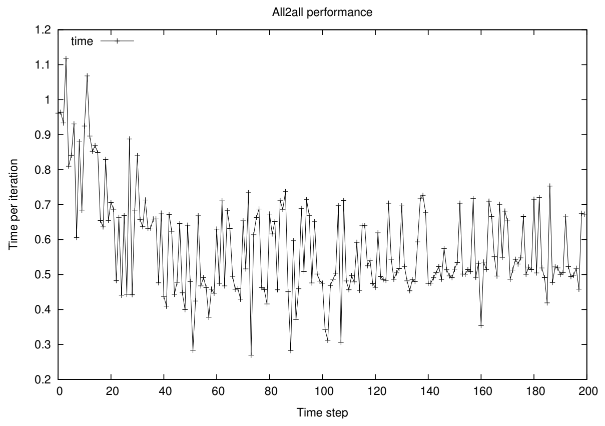

Fig. 2. Time per iteration for the ALLTOALLV stage of a particle simulation while runtime tuning of the threshold between pairwise and eager operations (43 nodes 18MB average data transfer per node, tuning test every 5 time steps)

7 Conclusions and future work

The results presented in this paper show that the flexible control of the Open MPI tuned collectives component do not effect communication performance, and that the component is still competitive with other MPI implementations such as LAM/MPI, FT-MPI and MPICH2. The component allows multiple varied and concurrent methods for the user or system administrator to control selectively the choice of backend collective algorithm and its parameters.

The dynamic rule base system adds even more capabilities in terms of runtime control, which was demonstrated by real time tuning of a critical parameter in a new ALLTOALLV algorithm. We are hoping to extend this work in a number of ways. This includes adding more automated tools for file based rule generation (application targeted tuning) and using feedback from switch and network infrastructure to dynamically control algorithm choices during runtime.

Acknowledgments. This material is based upon work supported by "Los Alamos Computer Science Institute (LACSI)", funded by Rice University Subcontract #R7B127 under Regents of the University Subcontract #12783-001-05 49 and "Open MPI Derived Data Type Engine Enhance and Optimization", funded by the Regents of the University of California (LANL) Subcontract #13877-001-05 under DoE/NNSA Prime Contract #W-7405-ENG-36. Infrastructure support was provided by the Grid'5000 project of the ACI GRID of the French Ministry of Research.

References

1. Beck, Dongarra, Fagg, Geist, Gray, Kohl, Migliardi, K. Moore, T. Moore, Papadopoulous, Scott, and Sunderam. *Future Generation Computer Systems*, 15, 1999.
2. Edgar Gabriel and Michael Resch and Thomas Beisel and Rainer Keller. Distributed Computing in a heterogenous computing environment. *Recent Advances in Parallel Virtual Machine and Message Passing Interface.* , Springer, Lecture Notes in Computer Science, 1998.
3. Rainer Keller and Edgar Gabriel and Bettina Krammer and Matthias S. Mueller and Michael M. Resch. Efficient execution of MPI applications on the Grid: porting and optimization issues. *Journal of Grid Computing*, 1(2), pp 133-149, 2003.
4. G. E. Fagg, A. Bukovsky, and J. J. Dongarra. HARNESS and fault tolerant MPI. *Parallel Computing*, 27:1479–1496, 2001.
5. W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.
6. Jelena Pješivac-Grbović and Thara Angskun and George Bosilca and Graham E. Fagg and Edgar Gabriel and Jack J. Dongarra. Performance Analysis of MPI Collective Operations. *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 15* IEEE Computer Society, Washington, DC, USA, 2005.
7. Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*, June 1995. http://www.mpi-forum.org/.
8. Rolf Rabenseifner. Automatic MPI counter profiling of all users: First results on a CRAY T3E 900-512. Proceedings of the Message Passing Interface Developer's and User's Conference, pp. 77–85, 1999.
9. S. S. Vadhiyar, G. E. Fagg, and J. J. Dongarra. Performance modeling for self-adapting collective communications for MPI. In *LACSI Symposium.* Springer, Eldorado Hotel, Santa Fe, NM, Oct. 15-18 2001.
10. E. Gabriel and G.E. Fagg and G. Bosilica and T. Angskun and J. J. Dongarra J.M. Squyres and V. Sahay and P. Kambadur and B. Barrett and A. Lumsdaine and R.H. Castain and D.J. Daniel and R.L. Graham and T.S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungry, 2004.
11. Thilo Kielmann and Rutger F.H. Hofman and Henri E. Bal and Aske Plaat and Raoul A. F. Bhoedjang. MagPIe: MPI's collective communication operations for clustered wide area systems. In ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99), 34(8), pp131–140, May 1999.
12. J.M. Squyres and A. Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting, Springer, LLNCS, Venice, Italy, 2003.*
13. *Jeffrey M. Squyres and Andrew Lumsdaine. The Component Architecture of Open MPI: Enabling Third-Party Collective Algorithms. In* Proceedings, 18th ACM International Conference on Supercomputing, Workshop on Component Models and Systems for Grid Applications *St. Malo, France, pp. 167–185, July 2004.*