MPIEcho: A Framework for Transparent MPI Task Replication

Guy Cobb[1], Barry Rountree[2], Henry Tufo[1],
Martin Schulz[2], Todd Gamblin[2], and Bronis de Supinski[2]

University of Colorado at Boulder
Department of Computer Science
Technical Report

CU-CS-1082-11

June 2011

University of Colorado at Boulder
Department of Computer Science
430 UCB
Boulder, Colorado 80309-0430

www.cs.colorado.edu

MPIEcho: A Framework for Transparent MPI Task Replication

Guy Cobb[1], Barry Rountree[2], Henry Tufo[1],
Martin Schulz[2], Todd Gamblin[2], Bronis de Supinski[2]

June 11, 2011

[1] University of Colorado
[2] Lawrence Livermore National Laboratory

Abstract

In this paper we describe *MPIEcho*: a profiling–layer library for replicating MPI ranks independently of application parallelism. This replication effectively breaks the tight coupling between an application's understanding of the parallel topology and that provided by the underlying MPI implementation, allowing a variety of use cases such as fault detection, independent parallelization of profiling and analysis tools, etc. We describe the architecture of *MPIEcho*, the current implementation status of the project for the MPI–2 standard, and then use the MPI suite of the NAS parallel benchmarks to show that *MPIEcho* correctly preserves application semantics when rank replication is used. We then discuss a wide variety of use cases and present a road map for investigating several of them.

1 Introduction

Understanding the complex and often nondeterministic behavior of massively parallel applications presents a unique set of challenges to the developer. Unfortunately many debugging, profiling, and communication analysis tools are heavyweight in nature and may affect the application runtime characteristics. One strategy to address this issue is to parallelize the execution of these tools independently of the application parallelism, thus minimizing the instrumentation overhead for an individual task.

A parallel application using the Message Passing Interface (MPI) decomposes a problem among a collection of tasks (a.k.a. "ranks"), the number and relative ordering of which is tightly coupled with the logic of most MPI applications. We break this tight coupling by providing arbitrary replication (cloning) of MPI tasks on a rank-by-rank basis in a way that is transparent to the application. Such replication allows parallelization that is orthogonal to existing application parallelism, enabling a wide variety of potential uses which we discuss in Section 2.

In implementing this concept we followed some basic design criteria: Each clone of a particular task should have no understanding at the application level that it is a clone; only the MPI library and tools written to use *MPIEcho* should know of the replication. The per–task replication should also be variable to accommodate a wide variety of use cases. Several challenges must be overcome to ensure this preservation, such as ensuring all replicas receive messages in the same order. This preservation is important because: 1) fixed precision floating point arithmetic is not commutative, and 2) application control flow is often dependent on message delivery order. Nonblocking operations represent a special set of challenges in preserving message ordering and are discussed further in Section4.3.

MPIEcho can be viewed as the major component of a complete process isolation framework for distributed memory MPI applications. In order to provide complete process isolation we will need to wrap non–MPI system calls to ensure only one rank of each clone group performs the operation (socket connections, non–MPI file and terminal I/O, etc.). However, this work is a significant component of such a potential framework, and a surprising number of applications work well if MPI is the only layer that understands task replication. A selection of these applications is discussed in Section 5.

The remainder of this paper is organized as follows. Section 2 describes a selection of use cases currently being implemented. This is followed by a discussion of our architecture in Section 3. We then discuss details of our implementation and our strategy for the remaining features of the MPI–2 standard in Section 4. With our architecture in place, we demonstrate application correctness using the full MPI suite of the NAS parallel benchmarks in Section 5.

2 Potential Uses

In this section we discuss a variety of potential use cases for the library. A companion document to this paper [10] describes the potential use cases in further detail and provides a performance evaluation of the library.

2.1 Parallelizing Heavyweight Tools

The independent parallelization of *MPIEcho* is useful if heavyweight tools are used to investigate anomalous application behavior. For example, the Valgrind [5] project can be used to detect common memory errors in an application (memory leaks, buffer overruns, etc.). If such a tool can be configured to monitor only a small portion of the overall memory space then we can use *MPIEcho*/Valgrind to split the monitoring among a collection of clones. Each clone should have the same memory contents and instruction stream, so we have effectively used independent parallelization minimized the walltime cost of memory monitoring.

This approach extends to other heavyweight debugging and profiling tools that can be parallelized in some way. This provides not only the benefit of faster analysis time, but also minimizes instrumentation overhead per task. Tools that rely on source–to–source or binary instrumentation are effectively changing the application being measured. The most obvious effect is that the caching behavior and memory layout of an instrumented application is potentially very different than the original binary, potentially causing errors seen in the original application to disappear or change.

MPIEcho is useful in this case because it can minimize the per–task instrumentation by parallelizing it across a set of clones, thereby ensuring the instrumented application is as close as possible to the original uninstrumented binary. *MPIEcho* is implemented as a shared library which effectively separates it from instrumentation performed on the application code.

2.2 Hardware Counter Collection

Modern microprocessors include a set of "hardware counters" that track occurrences of specific hardware events (cache misses, floating point instructions, cycles stalled waiting for memory reads, etc.). These counters are useful for performance analysis and tuning of applications, and the popular PAPI project [1] provides a standard interface for accessing these hardware counters. One of the major limitations of application profiling with hardware counters is that it is difficult to know in advance where a performance bottleneck might be, and so often a very large set of performance counters must be gathered and analyzed. Unfortunately only a few of the many counters available for collection can be collected at the same time due to hardware limitations. *MPIEcho* can address this issue by replicating the ranks to be profiled and having each clone collect a different set of PAPI counters. Using this approach, with a single run the entire set of possible counters can be collected for all ranks. This would greatly simplify the initial data collection phase of program analysis (both in terms of programmer effort and collection time).

2.3 Fault Detection

A natural use for *MPIEcho* is in detecting transient faults (bit flips in registers, corruption of unprotected memory structures, etc) occurring from a variety of sources, from cosmic background radiation and voltage aberrations to subtle manufacturing defects. *MPIEcho* can identify and correct silent data corruptions by comparing the data buffers provided as parameters to MPI functions. Using *MPIEcho* to replicate each rank would allow data corruptions from such events to be detected; providing two replicas would allow for data recovery in a "triple modular redundancy" (TMR) scheme. In effect, *MPIEcho* transforms each call to an MPI routine into a consistency checker for the application, transparently identifying and correcting errors.

Note that this is fault tolerance for a very specific fault model: a "fail–continue" model focused on silent data corruptions rather than hard node failures. *MPIEcho* can also turn a "fail–continue" model into a "fail–stop" model by terminating the application on error detection. This requires less resources than the TMR scheme, but still ensures applications that finish have no corruptions in data buffers seen by MPI routines.

2.4 Sensitivity Analysis

In simulation–based science it is important to understand how sensitive an application's output is to variations in input, message ordering, operating system jitter, or other sources of variation. Using *MPIEcho* it is possible to slightly permute these values either at initialization time (`MPI_Init()`) or during program execution in an MPI call. Each clone can have a slight variation introduced, and differences in either data or behavior (calculated from a timeline of MPI calls) can be analyzed. We are exploring the potential applications of *MPIEcho* to such sensitivity analysis and the more comprehensive approaches of uncertainty quantification (UQ).

3 Architecture

An MPI application's understanding of the parallel topology comes from information retrieved from the MPI library at runtime. Fortunately, the MPI standard provides a profiling interface (PMPI) that can be used to intercept such calls. *MPIEcho* is implemented using this profiling layer, providing clear separation between the application and MPI library parallel configurations. The profiling layer also allows us integrate *MPIEcho* with an existing application without making any changes to the source code; simply change the application Makefile to include the *MPIEcho* library and rebuild.

In *MPIEcho* there is a concept of the "real" rank and size (that seen by the MPI library) and the "application" rank and size (that seen by the application), with *MPIEcho* provides the mapping between the two. This is shown graphically in Figure 1. We see in the diagram that there are 15 real ranks provided by the underlying MPI library, but the application thinks there are only 8 ranks. Real ranks 0-4 all behave as rank 0, and ranks 8-11 behave as application rank 4. Any additional parallelism or other features can be split across real ranks 1-4 or 9-11 without affecting application behavior.

There are two distinct classifications for tasks in an MPI application with replication in place: "clones" and "representatives". The application has no knowledge of this classification; it is only relevant at the *MPIEcho* library level. We refer to the copies of a particular task as clones; each has their own real rank but all share the same application rank. That is, each one behaves at the application level as if they are the only instance of that particular (application) rank. A single clone from each set is chosen as the representative for that group, and along with the un–replicated ranks are referred to as "representatives". Two different types of internal communicators are created to manage the replication structure in *MPIEcho*: a clone communicator for each clone group and a

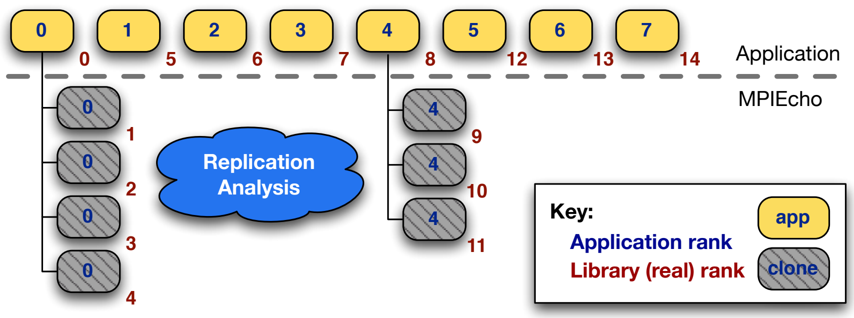

Figure 1: Application–level and library–level ranks

"representative" communicator to replace `MPI_COMM_WORLD`.

We preserve the semantics of collective operations using a two-stage approach: The first stage consists of performing the collective operation on the representative communicator. At this point the representative ranks all have the correct values. In the second stage, the representative for each clone group will broadcast the results to their clones. This process is depicted for a reduction operation in Figure 2.

Our two–stage approach preserves message ordering in two ways: First, only representatives participate in the MPI operation and so normal application–level message ordering is preserved. Second, the broadcast operation from the representatives acts as a synchronization point for the clones, ensuring they all see the same value at the same time. Using the underlying collectives provided by the MPI implementation also lowers the overhead of *MPIEcho* when compared to re–implementing each operation as sequence of point–to–point operations.

Point-to-point operations are also done using the same approach: send to the representative, then that representative broadcasts to their clones.

4 Implementation Details and Current Status for MPI–2

The operations defined in the MPI–2 standard [8] can be generally grouped into the following categories:

- Derived Communicators and Groups
- Point–to–Point Communication
- Nonblocking Communication
- Collectives

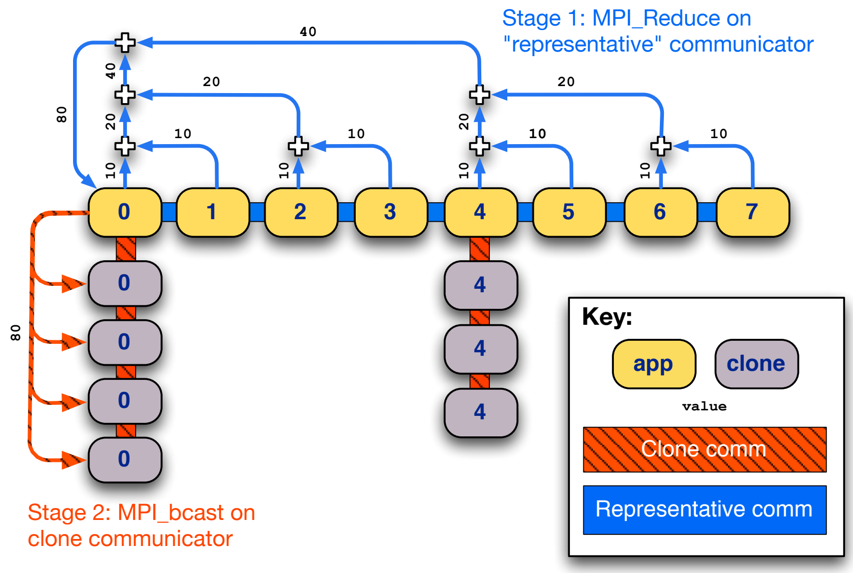

Figure 2: Two-stage reduction operation

- One–Sided Communication
- File I/O
- Intercommunicators
- Derived Datatypes
- Error Handlers
- Generalized Requests
- Process Creation and Management

Currently, all categories except one–sided communication have been implemented and tested. Support for Fortran MPI applications has been tested with the NAS Parallel Benchmarks and is discussed in Section 5. We use *wrap* [9] to generate method stubs for MPI routines in both C and Fortran, including re–entrant guards for MPI implementations that call from the underlying implementation layer back into the profiling layer. This utility proved to be very useful for generating the large amounts of template code required for a profiling–layer library, saving us a great deal of time.

4.1 Derived Communicators and Groups

In the case of MPI_COMM_WORLD, a static mapping between application and real ranks is created during initialization. Because it is not possible to know a priori what derived communicators an application might construct, dynamically creating rank maps at runtime is not feasible. Our approach then is to eliminate the need for rank mapping in derived communicators by only including representative ranks in the new communicator. Thus, MPI_COMM_WORLD becomes the special case that requires mapping, with derived communicators working with no special logic. We still have clones for the ranks in the derived communicators, and the two–stage mechanism works as before: non–representatives rely on their representatives to provide them the necessary results.

As with communicators, MPI_Group construction only includes representative ranks, with mapping performed only in the case of MPI_COMM_WORLD.

4.2 Point–to–Point Communication

This category consists primarily of the MPI_Send() and MPI_Recv() functions and their buffered analogues. In the non–buffered case the standard two–stage approach is used as described in Section 3.

The operation MPI_Bsend() relies on a user–provided buffer for outgoing messages (specified by MPI_Buffer_attach). In our two–stage approach only the representatives will call the underlying buffer attachment routine because only they participate in the send operation that will use the buffer.

4.3 Nonblocking Communication

Nonblocking MPI operations present a unique set of implementation challenges. In these operations, the initiation and completion are distinct operations. To use our two–stage approach, we need pointers to the original buffers, types, and lengths used in an MPI_Irecv() operation so that the results can be broadcast to the clones. This information is not available in the MPI_Wait()/MPI_Test() calls, so we maintain it in a linked list that these routines traverse and then perform the broadcast to the clones as usual.

MPI also provides persistent nonblocking operations (MPI_Send_init(), etc.) that allow a request handle to be reused. Our context–saving mechanism is used in conjunction with the two–stage approach as in the non–persistent case.

4.4 Collective Operations

Support for collective operations is a natural extension of the two–stage approach described for point–to–point operations. The representative ranks perform the operation and the results are broadcast to the clones.

4.5 One–Sided Communication

One–sided communication is the last major category of MPI operations to be implemented. `Get` operations can be performed using our two–stage approach, but `Put` operations require additional logic: because there is no explicit MPI call on the receive side of a `Put` operation, it the source must update all clones of a particular rank. We use our strategy for nonblocking communication in this case: when `MPI_Win_create` is called to register a block of memory, this block and the corresponding blocks for all clones will be maintained in a linked list. This list will be traversed on the representative and the value put in the registered memory window for each of the clones. This approach is a straightforward extension of our two–stage approach for nonblocking communication and is in the process of being implemented. The NAS benchmarks do not use one–sided communication, so this feature is not one we have focused on for our initial testing.

4.6 File I/O

MPI file I/O is implemented using our two–stage approach: representatives perform the operations and broadcast results to clones. Nonblocking I/O operations follow the semantics of the nonblocking communication: context information is stored and used in the `MPI_Wait()` (and related) calls to distribute results to the clones.

A special category of "split collectives" exists for MPI I/O operations. For example, the `MPI_File_read_all_begin()` and `MPI_File_read_all_end()` operations split a single operation into two collective calls so that file I/O can be overlapped with computations that do not depend on the data being read. Context information is maintained for these split collectives in a manner similar to nonblocking communication approach, but with a slightly simpler implementation due to guarantees made by the MPI standard: no more than one outstanding split collective can be in progress at a time per file handle, so there is no need to further disambiguate requests beyond what file handle they are associated with.

4.7 Intercomunicators

The semantics of intercommunicator construction are a natural extension of those for intracommunicator construction described in Section 4.1. Representative ranks are included in the newly created intracommunicator, with clone ranks receiving the results of operations performed on that intracommunicator using the broadcast mechanism.

4.8 Derived Datatypes

Construction of derived datatypes does not explicitly perform any communication (note the lack of communicators as function parameters), and so no special logic is required.

4.9 Error Handlers

Error handlers can be specified for communicators, file operations, or one–sided communication and present a special challenge for *MPIEcho*. Creation and assignment of error handlers is supported in *MPIEcho* for communicators and file operations (one–sided communication is a work in progress), but with a caveat: because only the representative ranks participate in MPI operations, they are also the only ranks that will invoke the error handler. If the error handler provided by the user alters the application state then it is entirely possible for the clones to become inconsistent with their representatives. It is most likely highly application–specific what effect error handlers may have on correct application behavior.

4.10 Generalized Requests

Generalized requests allow developers to define new nonblocking requests, executed in a separate thread to ensure progress independent of primary application execution. In principle, *MPIEcho* supports the generalized request mechanism specified by MPI, but it is operating system dependent whether this mechanism works in practice. The MPI standard does not specify the underlying mechanism the user application must use to ensure concurrent execution of the generalized request (threads, etc.), which prevents a priori guarantees of correctness being made for all platforms. If each instance of the generalized request is independent, then *MPIEcho* generalized requests will function correctly. If not, then the behavior of *MPIEcho* is undefined. As mentioned earlier, a completely transparent replication mechanism would require all system calls to be wrapped as well to ensure complete process independence.

4.11 Process Creation and Management

The MPI–2 standard provides applications the ability to create and terminate processes via the `MPI_Comm_spawn()` and related routines. This functionality is supported partially in *MPIEcho*: the calls to these MPI routines are implemented and the representative ranks will spawn the requested number of processes, but the semantics of this operation in the context of replication have yet to be determined. For example: `MPI_UNIVERSE_SIZE` can be queried to determine the number of processes that the runtime system will allow to be spawned, so should this number consider the replication configuration currently in use? The answer to this question depends on whether the spawned processes should use the same replication configuration as the "parent" application. A great variety of replication scenarios are possible with *MPIEcho* and we are currently deciding what behavior makes the most sense.

5 Test Cases

It is important to verify that *MPIEcho* preserves application behavior under a variety of replication configurations. Our primary test suite for this is the NAS Parallel benchmarks[4] (version 3.3) from NASA (in addition to various internal regression tests). This suite of benchmarks provides good coverage of the basic MPI operations and includes built–in verification of results. Many of the benchmarks in the suite are written in Fortran, providing a good test of the Fortran to C bindings in various MPI implementations, Currently the full suite of MPI benchmarks in the NAS benchmark suite execute correctly with *MPIEcho* under a variety of cloning scenarios. This suite includes:

- EP: "Embarrassingly Parallel" benchmark
- LU: Lower-upper symmetric Gauss-Seidel
- IS: Integer Sorting
- MG: Multigrid
- CG: Conjugate Gradient
- FT: Fast Fourier Transform
- BT: Block Tridiagonal
- SP: Scalar Pentadiagonal

Verification runs were performed with the most current versions of the OpenMPI and MVAPICH2 MPI libraries on the Atlas system at Lawrence Livermore National Laboratory. Currently all benchmark tests pass successfully under all attempted replication scenarios.

6 Related Work

While other projects exist that allow rank replication and are similar in some respects to *MPIEcho*, it is important to note that these projects are targeted exclusively for fault tolerance and not the general independent parallelization targeted by *MPIEcho*.

Architecturally, the most closely related project to *MPIEcho* is the rMPI project described in [7] which focuses on providing fault tolerance through the replication of ranks. The replication used in rMPI is fixed, with each application task having a single redundant copy. All messages are duplicated upon sending, and an set of algorithms is used to ensure proper message ordering. The high-order bits of the tag field are used to disambiguate messages intended for the primary and redundant tasks. Point–to–point operations are used for collectives, avoiding the problem of needing to perform communicator reconstruction in the event of a node failure. Unfortunately this (potentially) incurs a performance penalty because it bypasses the built-in collectives.

Recovery from hard node failures is currently outside the scope of potential applications for *MPIEcho*. The primary reason for this is that recovery from hard node failures requires the ability to construct a new communicator that does not include failed node. Communicator construction in MPI is defined in terms of existing communicators, so an MPI implementation would need to provide a new `MPI_COMM_WORLD` as well as information about what MPI tasks failed. This is unfortunately not a feature provided by current mainstream MPI implementations. The goals of *MPIEcho* are to be applicable to a wide variety of potential uses on any MPI implementation, and so until an update to the MPI standard provides this feature we restrict ourselves to the fail–continue fault model.

The *r*MPI and *MPIEcho* projects are similar, but have different purposes and implementation approaches. In contrast to *r*MPI where the replication topology is fixed and highly optimized for fault tolerance, the *MPIEcho* topology is arbitrary and can vary on a per-task basis. Currently *r*MPI is tied to a particular version of MPICH[2] due to use of internal functions in that MPI implementation. A design goal for *MPIEcho* was to work on any MPI implementation that provides the profiling interface, and it has been tested on both OpenMPI and MVAPICH[3].

A closely related project to *r*MPI is the MR–MPI[6] project from Oak Ridge National Laboratory. This project also focuses on fault tolerance by means of rank duplication. Messages from a sender are duplicated to each of the replicated ranks, and varying replication degrees are supposed in contrast to the fixed replication degree of *r*MPI project. Another feature of the MR–MPI project is that it is based on the MPI profiling layer, and as such is not tied to any particular MPI implementation.

7 Conclusions and Future Work

In this paper we have described *MPIEcho*, a profiling–layer library we have created for transparently replicating MPI ranks in an existing parallel application. *MPIEcho* requires no application code changes, requiring only a re–link with the *MPIEcho* library. We have described the architecture and implementation challenges in creating *MPIEcho*. The *MPIEcho* library is nearly feature–complete for the MPI–2 standard, with one–sided communication being the only remaining major feature of MPI–2 to be implemented.

We have demonstrated the correctness of the library using the NAS parallel benchmark suite; a companion document provides further information on the correctness and performance of the library under a variety of replication scenarios.

Architecturally–related projects are discussed, with a focus on their similarities and differences. Our goal is to provide a more general–purpose framework for orthogonal parallelism of tools, and our design philosophy is built around this idea, distinguishing *MPIEcho* from the existing rank replication projects focused on fault tolerance. A variety of use cases currently under investigation was described, and we are confident that a great many more uses for *MPIEcho* will emerge in the near future. We hope to expand on several of these use cases and demonstrate *MPIEcho*'s effectiveness in the near future.

References

[1] `http://icl.cs.utk.edu/papi/`, PAPI: Performance Application Programming Interface

[2] `http://www.mcs.anl.gov/research/projects/mpich2/`, MPICH2: A high-performance and widely portable implementation of MPI

[3] `http://www.mcs.anl.gov/research/projects/mpich2/`, MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE

[4] NPB: NAS parallel benchmarks, `http://www.nas.nasa.gov/Resources/Software/npb.html`

[5] Valgrind, `http://valgrind.org/`, Valgrind: A suite of tools for debugging and profiling

[6] Engelmann, C., Böhm, S.: In: Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN) 2011. Innsbruck, Austria

[7] Ferreira, K., Riesen, R., Oldfield, R., Stearley, J., Laros, J., Pedretti, K., Kordenbrock, T., Brightwell, R.: Increasing fault resiliency in a message-passing environment. Tech. rep., Sandia National Laboratory (2009)

[8] Forum, M.P.I.: MPI: A Message-Passing Interface Standard, Version 2.1. High Performance Computing Center Stuttgart (HLRS) (June 2008)

[9] Gamblin, T.: `https://github.com/tgamblin/wrap`, Wrap: An MPI Wrapper Generator

[10] Rountree, B., Cobb, G., Gamblin, T., Schulz, M., Supinski, B.D.: Parallelizing Heavyweight Debugging Tools with MPIecho. In: Proceedings of the First International Workshop on High-performance Infrastructure for Scalable Tools (2011)