An Extensible Framework for Distributed Testing of MPI Implementations

Joshua Hursey[1], Ethan Mallove[2], Jeffrey M. Squyres[3], Andrew Lumsdaine[1]

[1]Indiana University
Open Systems Laboratory
Bloomington, IN USA
{jjhursey, lums}@osl.iu.edu

[2]Sun Microsystems, Inc.
Burlington, MA USA
ethan.mallove@sun.com

[3]Cisco, Inc.
San Jose, CA USA
jsquyres@cisco.com

Abstract. Complex code bases require continual testing to ensure that both new development and routine maintenance do not create unintended side effects. Automation of regression testing is a common mechanism to ensure consistency, accuracy, and repeatability of results. The MPI Testing Tool (MTT) is a flexible framework specifically designed for testing MPI implementations across multiple organizations and environments. The MTT offers a unique combination of features not available in any individual testing framework, including a built-in multiplicative effect for creating and running tests, historical correctness and performance analysis, and support for multiple cluster resource managers.

1 Introduction

High quality MPI implementations are software packages so large and complex that automated testing is *required* to effectively develop and maintain them. Performance is just as important as correctness in MPI implementations, and therefore must be an integral part of the regression testing assessment. However, the number of individual tests taken in combination with portability requirements, scalability needs, and runtime parameters generates an enormous set of testing dimensions. The resulting testing space is so large that no single organization can fully test an MPI implementation. Therefore, a testing framework suitable for MPI implementations must be able to combine testing results from multiple organizations to generate a complete view of the testing coverage.

Many MPI test suites and benchmarks already exist that can verify the correctness and performance of an MPI implementation. Additionally, MPI implementation projects tend to have their own internal collection of tests. However, running a large set of tests manually on a regular basis is problematic; human error and changing underlying environments will cause repeatability issues.

A good method for regression testing in large software projects is to incorporate automated testing and reporting, run on a regular basis. Abstractly, a testing framework is required to: obtain and build the software to test; obtain and build individual tests; run all tests variations; and report both detailed and aggregated testing results. Additionally, since the High Performance Computing (HPC) community produces open source implementations of MPI that include contributions from many different organizations, MPI implementation testing methodology and technology must also:

- Be freely available to minimize the deployment cost.
- Easily incorporate thousands of existing MPI tests.
- Support simultaneous distributed testing across multiple sites, including operating behind organizational security boundaries (e.g., firewalls).
- Support on-demand reporting, specialization, and email reports.
- Support execution of parallel tests, and therefore also support a variety of cluster resource managers.

We have therefore created the MPI Testing Tool (MTT), an MPI implementation-agnostic testing tool to satisfy these needs, and have prototyped its use in the Open MPI project [1].

The rest of this paper is organized as follows: related work is presented in Section 2. Section 3 describes the MPI Testing Tool (MTT) in more detail. Section 4 presents MTT experiences with the Open MPI project. Finally, we present conclusions and a selection of future work items in Section 5.

2 Related Work

There is a large field of research surrounding optimal testing techniques, but only a few of those ideas seem to have any impact on the software engineering process used to develop and maintain large software systems [2, 3]. To maintain the high quality of large software systems, continual regression testing is required. Onoma et al. [4] describe vital components of an effective software testing suite. The MTT is designed to satisfy these requirements for MPI implementations.

TET [5] is a widely adopted tool among hundreds of free and open source testing frameworks. However, TET does not provide mechanisms for obtaining the software to be tested, therefore requiring an additional layer of software to determine whether new versions are available to prevent duplicate testing results. TET also has a crude reporting mechanism which requires searching through flat file logs for test results. Although logs can be exported to a database for historical data mining, no front-end is provided for querying the test results.

Perfbase [6] presents a front-end to a SQL database for storing historical performance data. Perfbase does not provide a framework for obtaining, building, and testing software, but rather focuses on archiving and querying test results. Although the first generation of MTT used Perfbase as a back-end data store, MTT evolved its own data store mechanisms due to inflexibility of Perfbase's storage and retrieval model.

DejaGNU [7] is a testing framework from which testing harnesses and suites can be derived. Similar to TET, no rich reporting mechanisms are provided, and native support for parallelism is not included. DejaGNU also requires individualized test suite scripts for each test conducted.

Many other testing harnesses are available, including Kitware's Dart system, the Boost C++ regression testing system, Mozilla Tinderbox and Testopia, and the buildbot project. However, none of the products and projects surveyed met the full set of requirements needed for testing MPI implementations in a distributed, scalable fashion.

3 The MPI Testing Tool (MTT)

The MTT was created to solve many of the issues cited above.

At its core, the MTT is a testing engine that executes the following phases:

1. **MPI Get**: Obtain MPI implementations. Implementations are obtained from the Internet (via HTTP/S, FTP, Subversion), local copies (e.g., tarball or directory), or by reference to a working installation.
2. **MPI Install**: Specify how to build/install MPI implementations.
3. **Test Get**: Obtain test suite source codes, similar to the MPI Get phase.
4. **Test Build**: Compile test suites against all successfully installed MPI implementations.
5. **Test Run**: Run individual tests in each successfully built test suite.

The MPI Install, Test Build, and Test Run phase results are stored in a central database (or other user specified mechanism). All MTT tests will generate one of four possible results: pass, fail, timeout, or skip. Each test defines specific criteria indicating success. For example, an MPI implementation must compile and install successfully to qualify as "pass." A test is "failed" if the test completes but the "pass" criteria is not met. A test is killed and declared a "timeout" if its execution did not finish within the allotted time period. A test is declared "skip" if it elects not to run. For example, an InfiniBand-specific test may opt to be skipped if there are no InfiniBand networks available.

Phases are templates which allow multiple executions of each step based on parametrization. Later phases are combined with all successful invocations of prior phase invocations, creating a natural multiplicative effect.

- M: MPI implementations
- I: MPI installations, each of which are applied against all M MPI implementations.
- N_t: Individual tests, grouped by test suite (t), each of which is compiled against all $(M \times I)$ MPI installations.
- R_t: Run parameters specified for tests, each of which is applied against $(M \times I \times N_t)$ tests.

Fig. 1 shows the general sequence of the phases as well as their relationships to each other and the natural multiplicative effect. The figure shows one MPI implementation that is installed two different ways (e.g., with two different compiler suites). Each installation is then used to build two tests; each test is run two different ways. Hence, the total number of reported tests will follow the equation:

$$\begin{aligned} &num\_MPI\_installs + num\_test\_builds + num\_test\_runs \\ &\qquad (M \times I) + \quad (M \times I \times N_t) \quad + (M \times I \times N_t \times R_t) \end{aligned}$$

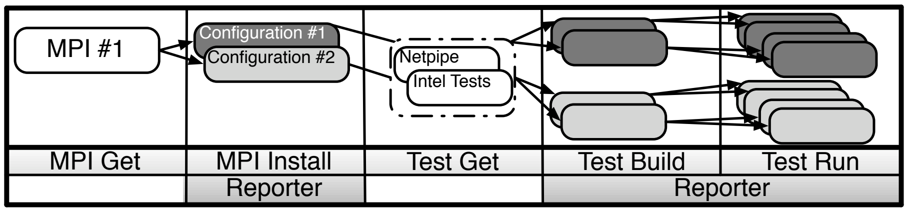

Fig. 1. MTT phase relationship diagram. Phases labeled with "Reporter" contain tests that are saved to a back-end data store such as a database.

Therefore Fig. 1 shows a total of 14 tests (2 MPI installs + 4 test builds + 8 test runs). While the multiplicative effect is a deliberate design decision, users must be careful to not create test sets that incur prohibitively long run times.

3.1 Configuration

The MTT test engine is configured by an INI-style text file specified on the command line. In the INI file format, sections are denoted with strings inside brackets and parameters are specified as `key = value` pairs. Fig. 2 shows a sample fragment of an MTT INI configuration file.

A typical configuration file contains a global parameters section, one or more MPI Details sections, and one or more sections for each of the execution phases.

```
[MPI Get: Open MPI nightly trunk]
module = SVN
svn_url = http://svn.open−mpi.org/svn/ompi/trunk

[MPI Get: Open MPI v1.2 snapshots]
module = OMPI_Snapshot
ompi_snapshot_url = http://www.open−mpi.org/nightly/v1.2

[MPI Install: GNU compilers]
mpi_get = Open MPI nightly trunk,Open MPI v1.2 snapshots
module = OMPI

[MPI Install: Intel compilers]
mpi_get = Open MPI nightly trunk,Open MPI v1.2 snapshots
module = OMPI
ompi_configure_arguments = CC=icc CXX=icpc F77=ifort FC=ifort CFLAGS=−g

[MPI Details: Open MPI]
exec = mpirun −−mca btl self,&enumerate(“tcp”, “openib”) \
    −np &test_np() &test_executable()
```

Fig. 2. Fragment of a simplified MTT configuration file. Two MPI Get phases are paired with two MPI Install phases, resulting in four MPI installations. The MPI Details section templates an execution command for invoking MPI tests. This example shows at least two executions for each test: one each for TCP and OpenFabrics networks.

The global parameters section is used to specify testing parameters and user preferences across an entire run of the MTT. MPI Details sections specify how to run executables for a specific MPI. For example, these sections contain nuances such as whether `mpirun` or `mpiexec` should be used, what command line options to use, etc. Each execution phase will also be specified by at least one section in the configuration file. The phase INI sections are comprised of phase-specific parameters, the designation of a plugin module to use, and module-specific parameters. Fig. 2 shows an INI file example that downloads two different versions of Open MPI detailed in the MPI Get phases and compiles both of them with two different compilers (GNU, Intel) detailed in the two MPI Install phases.

Any number of MPI implementations can be specified for download in MPI Get sections. Since each MPI Get section will potentially download a different MPI implementation (and therefore require a different installation process), MPI Install sections must specify which MPI Get section(s) to install.

Phases are linked together in the configuration file by back-referencing one or more prior phase names. For example, lines 10 and 14 in Fig. 2 show the two MPI Install sections back-referencing the "Open MPI nightly trunk" and "Open MPI v1.2 snapshots" MPI Get sections. Hence, these two MPI Install sections will be used to install both MPI Get sections. Similar back-referencing mechanisms are used for the other phases.

MTT will conditionally execute phases based on the outcome of prior phases. For example, the MPI Install phase is only executed in the case where the prior execution of the corresponding MPI Get phase was both successful and yielded a *new* version of the MPI implementation (unless otherwise specified). Similarly, Test Run sections will only execute tests where all prior phases were successful: a new MPI implementation was obtained and successfully installed, and tests were successfully obtained and compiled against the MPI installation.

3.2 Funclets

Additional combinations of testing parameters can be specified via "funclets" in the configuration file. "Funclets" are Perl-like function invocations that provide both conditional logic and and text expansion capabilities in the configuration file, enabling it to serve as a template that is applicable to a variety of different scenarios.

A common use of funclets is to expand a configuration parameter to be an array of values. For example, the `np` parameter in Test Run sections specifies how many processes to run in the test. `np` can be set to one or more integer values. The following example assigns an array of values to the `np` parameter by using three funclets:

```
np = &pow(2, 0, &log(2, &env_max_procs()))
```

– `&env_max_procs()`: Returns the maximum number of processes allowed in this environment (e.g., number of processors available in a SLURM or Torque job, the number of hosts in a hostfile, etc.).
– `&log()`: Returns the log of the first parameter to the second parameter.

– `&pow()`: Returns an array of integer values. The first parameter is the base, the second and third parameters are the minimum and maximum exponents, respectively.

In the above example, when running in a SLURM job of 32 processors, `np` would be assigned an array containing the values 2, 4, 8, 16, and 32. This causes the MTT execution engine to run each test multiple times: one for each value in the array.

3.3 Test Specification

The MTT supports adding tests in a modular and extensible manner. The procedures to obtain and build test suites (or individual tests) are specified in the configuration file. Although the MTT can execute arbitrary shell commands from the configuration file to build test suites, complex build scenarios are typically better performed via MTT plugin modules. Tests to run are also specified in the configuration file; the MTT provides fine-grained control over grouping of tests, pass/fail conditions, timeout values, and other run-time attributes.

3.4 Test Execution

The MPI Details section tells the MTT how to run an executable with a particular MPI implementation. Each MPI Get section forward-references an MPI Details section that describes how to run executables for that MPI. This feature allows the MTT to be MPI-implementation-agnostic.

For example, line 18 in Fig. 2 shows an MPI Details section for Open MPI. The `exec` parameter provides a command line template to run tests. The funclets `&test_np()` and `&test_executable()` are available to "paste in" the values specific to the individual test being invoked. Note, too, the use of the `&enumerate()` funclet. This funclet will return an array of all of the values passed as parameters, effectively causing the `exec` parameter to expand into at least *two* `mpirun` command lines: one with the string "`--mca btl self,tcp`" and another with the string "`--mca btl self,openib`" (forcing Open MPI to use the TCP and OpenFabrics network transports, respectively).

Combining the use of multiple funclets can result in a multiplicative effect. For example, using the same 32 processor SLURM job and funclet-driven `np` value from Section 3.2, the `exec` parameter from Fig. 2 will expand to invoke *ten* command lines for each test: five with the TCP transport (with 2, 4, 8, 16, and 32 processes), and five with the OpenFabrics transport.

3.5 Reporting Testing Results

Fig. 1 illustrates that the Reporter phase is run after the MPI Install, Test Build, and Test Run phases. The Reporter phase writes testing results to a back-end data store such as a central database, but may also log information to local text files (or a terminal).

The MTT features a web interface to the central database to facilitate complex interactive explorations of the testing data, including comparisons of performance and correctness over multiple versions of an MPI implementation. The web interface is specifically designed to aggregate the testing results into high level summary reports that can be used to repetitively narrow searches in order to find specific data points. Such "drill down" methods are commonly used to aid in discovering trends in test failures, displaying historical performance results, comparing results between different configurations, etc.

Additionally the web-based reporter interface provides custom stored queries, allowing developers to easily share views of the testing data. Absolute and relative date range reports are useful when citing a specific testing result and tracking its progress over time, respectively.

4 Case Study: The Open MPI Project

The Open MPI project relies on the MTT for daily correctness and performance regression testing. Open MPI is a portable implementation of the MPI standard that can run in a wide variety of environments; testing it entails traversing a complex parameter space due to the enormous number of possible environments, networks, configurations, and run-time tunable parameters supported.

The enormous parameter space, in combination with limited available testing resources, prevents any one member organization from achieving complete code coverage in their testing. By running the MTT at each Open MPI member organization, the project effectively pools the resources of all members and is able to achieve an adequate level of testing code coverage. This scheme naturally allows each organization to test only the specific configurations that are important to their goals.

Member testing resources range from small to large collections of machines; unscheduled and scheduled environments; with and without firewall restrictions. Each organization has their own site-specific MTT configuration files that detail exactly which scenarios and environments to test.

The Open MPI project has two distinct testing schedules (weekday for "short" 24-hour testing, and weekend for longer / higher process count testing); both follow the same general format: Open MPI snapshot tarballs are generated from the Open MPI Subversion development trunk and release branches and are posted on the Open MPI website. Member organizations use the MTT to download and test the snapshots on their local testing resources.

Several well-known MPI test suites are run against Open MPI via the MTT, including the Intel MPI test suite, the LAM/IBM MPI test suite, the Notre Dame C++ MPI test suite, the Intel MPI Benchmarks (IMB), NetPIPE, etc. Many tests internal to the Open MPI project are also run via the MTT.

As each member's testing completes, results are uploaded to a central database hosted by Indiana University and made available through the MTT's web-based reporter interface. On weekdays, rollup summary reports are e-mailed to the Open MPI development team 12 and 24 hours after the daily cycle begins. Summary reports of weekend-long testing are sent on Monday morning. The e-mail reports, combined with detailed drill-down queries, form the basis of daily discussions among developers, corrections and modifications to recent changes, and decisions about release schedules.

Using the MTT, the Open MPI project has accumulated over 7 million test results between November 2006 and May 2007. Approximately 100,000 tests are run each weekday/weekend cycle, spanning six platforms, six compiler suites, and seven network transports.

5 Conclusions

The MTT successfully supports the active development of the Open MPI project providing correctness and performance regression testing. The MTT provides a full suite of functionality useful in the automated routine testing required of a high quality MPI implementation. By fully automating the testing process, developers spend more time developing software than routinely testing it. Although this paper has focused on the MTT's testing of Open MPI, the MTT is MPI implementation agnostic, and has been used to test other MPI implementations, such as LAM/MPI [8] and MPICH2 [9]. The MTT is available at:
http://www.open-mpi.org/projects/mtt/

5.1 Future Work

While the MTT currently supports many modes of operation, there are several areas where its support could be expanded, including: supporting testing in heterogeneous environments, supporting complex "disconnected" scenarios for environments not directly (or even indirectly) connected to the Internet, exploiting the natural parallelism exhibited by orthogonal steps within the MTT testing cycle to make more efficient use of testing resources, and expanding the MTT to support general middleware testing.

References

[1] Gabriel, E., et al.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary (2004) 97–104

[2] Osterweil, L.: Strategic directions in software quality. ACM Comput. Surv. **28**(4) (1996) 738–750

[3] Harrold, M.J.: Testing: a roadmap. In: ICSE '00: Proceedings of the Conference on The Future of Software Engineering, New York, NY, USA, ACM Press (2000) 61–72

[4] Onoma, A.K., Tsai, W.T., Poonawala, M., Suganuma, H.: Regression testing in an industrial environment. Commun. ACM **41**(5) (1998) 81–86

[5] TET Team: TETware white paper. Technical report, The Open Group (2005) `http://tetworks.opengroup.org/Wpapers/TETwareWhitePaper.htm`.

[6] Worringen, J.: Experiment management and analysis with perfbase. In: IEEE Cluster Computing 2005, IEEE Computer Society (2005) 1–11

[7] Free Software Foundation: DejaGnu (2006) `http://www.gnu.org/software/dejagnu/`.

[8] Squyres, J.M., Lumsdaine, A.: A Component Architecture for LAM/MPI. In: Proceedings, 10th European PVM/MPI Users' Group Meeting. Number 2840 in Lecture Notes in Computer Science, Venice, Italy, Springer-Verlag (2003) 379–387

[9] Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A high-performance, portable implementation of the MPI message passing interface standard. Parallel Computing **22**(6) (1996) 789–828