Los Alamos Parallel I/O Performance Measurement & Testing Project: Final Report

David Bloomquist
Liz Windt

June 19, 2008

Table of Contents

Table of Contents..2
Abstract..3
Introduction..3
Requirements..3
Test Suite..3
Test Method..5
Test Output...6
Design...6
Results..6
Conclusion..7

Abstract

Los Alamos National Labs is a major player in the high performance computing world. They have several large computer clusters that are used for widely varying projects. All of these projects use the clusters for massively parallel computing tasks. One of the most important tasks for the programs is to store the information computed. This is done using the I/O capabilities of an MPI-2 standards compliant library. However, different projects use different parts of the I/O facilities. Los Alamos National Labs is looking to improve its parallel I/O testing library. Currently, the tests consist of a single program that uses a fraction of the MPI I/O function calls. The MPI I/O tests need to be expanded and improved. The purpose of the tests is to check the integrity of the MPI I/O libraries in use.

A test file will be created and written out to disk. After a check to make sure that the write was successful, the file will be read back in. Upon read, two tests will be performed. First, the return value of the function will be checked for error. Second, the integrity of the file will be checked. This basic testing pattern will be used with several different methods of writing files.

The first tier of the project is to get these tests to report success or failure of the I/O operation. This is used to test the general integrity of the MPI I/O installation. The second tier is to have the test programs display performance information; such as, time and bandwidth. This will be used to assess any changes to the MPI I/O framework over time.

Introduction

Message passing interface input and output (MPI I/O) allows multiple processes to write and read simultaneously to the same file. There is a defined standard of MPI and several implementations. Los Alamos National Labs uses the OpenMPI implementation for several different projects and new projects are constantly being developed. These projects rely on the MPI I/O libraries to work successfully. However, there is not currently a way to test all of the MPI I/O functionality ensure that it is working.

A new test suite was developed to evaluate the functionality of the MPI I/O implementation. It will check that each function in the MPI 2 standard relating to MPI I/O, and determine if the I/O is working properly.

Requirements

Test suite

This program needed to be a lightweight test suite that can be run daily for routine system testing as well as on-demand following system upgrades and other such events. Both C and Fortran interfaces need to be tested eventually;

however, a C-only implementation is the goal for this 6-week project. "Lightweight" is defined as:

- Short runtime (~20 minutes or less) when run on 4 processes
- Simple startup (script submission, automated via cron, and so on.)
- Easy to understand output (PASS/FAIL type messages for first version, some measurement numbers in future versions

Despite it being lightweight it still must test as much as possible of the MPI-2 I/O specification. Realizing that most MPI implementations do not yet support the entire specification this test suite should be able to report those features that are not implemented.

The following MPI collective I/O functions are tested:

- MPI_File_open / MPI_File_close
- MPI_File_delete
- MPI_File_write / MPI_File_read
- MPI_File_write_all / MPI_File_read_all
- MPI_File_write_all_begin / MPI_File_read_all_begin
- MPI_File_write_all_end / MPI_File_read_all_end
- MPI_File_write_at / MPI_File_read_at
- MPI_File_write_at_all / MPI_File_read_at_all
- MPI_File_write_at_all_begin / MPI_File_read_at_all_begin
- MPI_File_write_at_all_end / MPI_File_read_at_all_end
- MPI_File_write_ordered / MPI_File_read_ordered
- MPI_File_write_ordered_begin / MPI_File_read_ordered_begin
- MPI_File_write_ordered_end / MPI_File_read_ordered_end
- MPI_File_write_shared / MPI_File_read_shared
- MPI_File_iwrite / MPI_File_iread
- MPI_File_iwrite_at / MPI_File_iread_at
- MPI_File_iwrite_shared / MPI_File_iread_shared
- MPI_File_preallocate
- MPI_File_create_errhandler
- MPI_File_get_amode
- MPI_File_get_atomicity / MPI_File_set_atomicity
- MPI_File_get_byte_offset
- MPI_File_get_errhandler / MPI_File_set_errhandler
- MPI_File_get_group
- MPI_File_get_info / MPI_File_set_info
- MPI_File_get_position / MPI_File_get_position_shared
- MPI_File_get_size / MPI_File_set_size

- MPI_File_get_type_extent
- MPI_File_get_view / MPI_File_set_view
- MPI_File_seek / MPI_File_seek_shared
- MPI_File_sync

Testing Method
Each of the above functions returns an error code. The error codes are examined for correctness for a basic pass/fail evaluation. In addition, the write/read operations should be performed in pairs such that any data that is written to disk is read back in and compared to the original data for accuracy.

The size of the resulting test files will be determined based on the criteria that the test suite must be able to run under 20 minutes on 4 processes.

Output
The output should be as simple as possible. For each test there is a single line stating the name of the function tested followed by a simple PASS or FAIL statement. An example would be

Test <number> MPI_File_Write() <data size> - **** PASS ****

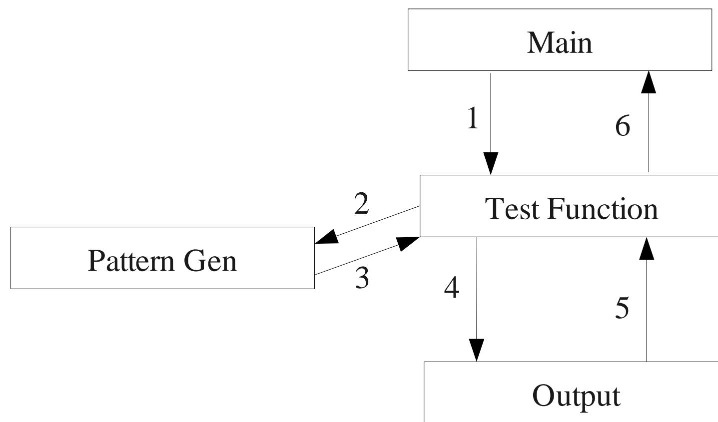
where <number> and <data size> are determined as part of the implementation of the suite.

Following each of the above lines, time permitting, it should also display timing and bandwidth data when implemented.

Design

To comply with the requirements, we wrote a program that will test all of the MPI I/O functions specified in the MPI-2 standards. Los Alamos National Labs currently uses OpenMPI which is the target library, but it should be compatible with all versions. The application is written in C. It prints to standard out to allow the greatest degree of flexibility for use of the results.

The main part of the program calls the individual functions that test the I/O functionality. Each test function called implements a write/read pair of MPI I/O function calls to execute a test. Other non-write/read test functions test one specific I/O call. For example, MPI_File_read() and MPI_File_write() would be in the same test function while MPI_File_seek() would be by itself.



The test functions first create and open a file. Next is the initial write to the file. At this time, it generates the input for the write, based on a function that will generate a pattern that we can later check against. Each function then reads in the file that was just written. Next, the read data is checked for differences from the written data.

We use a fail-fast methodology, in which if any read or write indicates that an error occurs, processing will stop and no more checking will be done. At the end of each function, regardless of the outcomes of the checks, an output function will be called to print the results of the test. Before a return from the program, the file that was created will be removed from the system.

The purpose of the output function is to separate the testing from the output it generates. The output function takes a struct that contains the information to be displayed to the screen. It then prints the information from the struct. The use of a struct will allow for easy modification to the available output as more information is added without compromising the ability of previous tests to function.

The purpose of the pattern generation function is to separate the testing from the patterns used to test. The pattern that we use is a string of numbers starting at 1 and incremented by 1 plus the process rank. This will generate a unique string that will only be read back correctly if the MPI I/O functions operate correctly.

The modular nature of the design should easily accommodative future refinements, requiring a minimal amount of revision.

Results

The combination of an under 20 minute run time and using approximately 100 gigabyte test files ended up being unobtainable (Note: performance testing was done on the CSM Alamode lab not the LANL cluster). Separately they were both possible. To get around this conflict the program allows the user to choose the test file size in gigabytes. The smaller the file the faster the execution time.

Even though MPI 2 is a standard and all implementations should at least “stub” out all the functionality defined by the standard, we discovered that this was not the case. Since we have no way of discovering which functions were missing before compiling on a specific system, this led to last minute changes of the test bed to exclude these calls.

There is one error specific to OpenMPI that we were not able to address. If an MPI call uses a shared file pointer, MPI creates a temporary file. When the working file is closed MPI cleans up the temporary file. However, in OpenMPI each process tries to delete the file. The first process deletes the file, the rest print an error message to standard error. We can not change the behavior of OpenMPI, therefore we can not address this issue. These errors do not cause a problem to our testbed and can safely be ignored.

Conclusion

This program meets all the basic requirements from LANL. It tests all the MPI I/O functionality. The output is clear and concise. The command line argument to change the size of the test file allowed for a compromise between the time and size requirements. The modularity of it should allow the time and bandwidth measurements to be easily added in a future revision.