A Parallel I/O Test Suite

D. Lancaster[1], C. Addison[2], and T. Oliver[2]

[1] University of Southampton, Southampton, U.K.
[2] Fujitsu European Centre for Information Technology, Uxbridge, U.K.
addison@fecit.co.uk,
WWW page: http://www.fecit.co.uk

Abstract. Amongst its many features, MPI-2 offers the first standard high-performance I/O interface. While this enables a parallel and I/O intensive code to run on multiple platforms, achieving consistent performance levels will be more difficult than many users imagine. To better understand the I/O performance issues in MPI-2, *fecit* and Southampton have developed a parallel I/O test suite for Fujitsu Japan. The design of this test suite is presented, along with a discussion concerning the need for a validation suite for MPI-2 as a whole.

1 Introduction

There are few who will dispute the significant benefit that MPI has brought to those developing software for distributed memory parallel systems. Part of the reason for the success of MPI was the presence of high quality, free distributions, such as MPICH [1], that worked well on a range of architectures.

As with any successful software product, there were features that were intentionally omitted. These included:

– support for parallel I/O operations,
– single-sided communications,
– dynamic processes.

The MPI-2 standard [2] was developed to address these “omissions” and several other important issues, such as language bindings for C++ and Fortran 90.

It is the provision of support for parallel I/O, through the set of routines collectively called MPI-I/O, that is the most exciting of MPI-2’s new features. I/O has long been a largely underdeveloped field of computing. This is particularly true in parallel computing, where users were obliged to use vendor specific I/O models that did not always address the major difficulties. MPI-I/O offers the first standard high-performance I/O interface. That deserves considerable credit, but it must also be viewed as just the start of the process.

Coping with I/O is a non-trivial task and MPI-I/O has a rich feature set. A degree of experimentation is required to understand how the different MPI-I/O routines function together. Fortunately, there are portable implementations of MPI-I/O freely available, so that users can experiment with these to gain some of this understanding. This process is aided by the availability of the MPI-I/O test codes developed at Lawrence Livermore National Laboratory [3].

Performance tuning, both from the point of view of using MPI-I/O and from the point of view of supporting MPI-I/O, requires an additional set of tools. *fecit* and the University of Southampton have developed an I/O test suite for Fujitsu Japan to address part of this need.

2 Design principles of the I/O test suite

The Parallel I/O Test Suite is a timely tool to investigate new developments in parallel I/O. In order to best address this goal, tools rather than benchmarks are emphasised, so the suite is directly under the control of the user, and thus implicitly less "automatic". The analysis requirements are greater than for a simple benchmark so the data gathering is separated from analysis to simplify the tests themselves, yet allow analysis of varying degrees of sophistication. Past benchmarking offerings have been criticised as being too complicated for most users, so simplicity has been required of both the tests themselves and of the run procedure.

Notable features of the Test Suite are:

– The tests are organised into low-level and kernel classes.
– Tools rather than benchmarks are emphasised.
– The tests are simple and the timings have clear significance.
– Data gathering is separated from analysis.
– No model for the behaviour of the data is assumed.
– Curves of data are generated rather than single numbers.
– The tests are not intended to comprise a validation suite for MPI-I/O and do not exercise all the advanced features of MPI-I/O.

These features are discussed in detail below along with some of the design criteria for the analysis stage. Some of the design choices arise from evaluation of serial I/O tests (for example [4]) and MPI-1 benchmark suites [5, 6].

2.1 Lessons from serial I/O testing

Some common issues in I/O testing that have had a bearing on the whole suite of tests have become clear in the context of serial I/O testing [4]. The most important issue that must be faced relates to the effect of the hierarchy of intermediate cache levels between the CPU and the final output device. Even when a write request has completed, one cannot be sure that the information resides on disk instead of in some cache. One strategy to ensure that the limiting bandwidth is measured and that the information is on disk, is to write sufficiently large files that fill up and overwhelm the cache. This strategy is not very efficient, since it requires considerable time to write the large files required. It can also be regarded as artificial to bypass the cache because it is an intrinsic part of the I/O system. A partial resolution to this issue is to measure a curve of bandwidths for different filesizes.

When mixing writing and reading, the same cache effects found in the write tests can be seen when reading data that was recently written and that still resides on the cache. This pattern of many reads and writes naturally occurs in the low-level tests because of the need for self checking. A useful feature of MPI is the `MPI_file_sync` call which flushes the cache to the storage device, and can be used between write and read calls. This call alone does not remove the possibility that a copy of the written data is still in cache. A further point is that compilers are sometimes capable of optimising code so as not to actually perform a read if the data is not used, this means that the data read must always be "touched" in some way.

Another relevant issue that arises in serial I/O testing is the fluctuations in repeated measurements. This is present in all modern systems, even on a dedicated system. This will be discussed at greater length below in section 2.8.

2.2 Low-level and kernel classes

The tests are classified as either Low-Level or Kernel along the lines of PARKBENCH [6], with the intention of testing the system at different levels.

Lowlevel: Measures fundamental parameters and checks essential features of the implementation. Allows performance bottlenecks to be identified in conjunction with expected performance levels. The tests in this class are:

- `single` measures the bandwidth of a single process to write and read a disk file.
- `multiple` tests multiple processes operating on a disk file.
- `singleI` is the baseline for asynchronous I/O where I/O is interleaved with computation.
- `multipleI` is the multi-process equivalent to `singleI`.
- `sinunix` is similar to `single` except Fortran I/O calls are made rather than MPI-I/O.

Kernel: Tests the MPI implementation at a more advanced level with a wider range more characteristic of real applications. Allows comparisons between implementations. The tests in this class are:

- `matrix2D` and `matrix3D` measure I/O on regular multidimensional arrays. Motivating applications include simulations of 2 or 3 dimensional physical systems, for example computational fluid dynamics, seismic data processing and electronic structure calculations.
- `nonseq` measures non-sequential I/O such as arises in database applications, medical image management and the recovery of partial mapping images.
- `gatherscat2D` tests Gather/Scatter combined with I/O. This series of steps is often employed when running applications on parallel machines with limited I/O capability. It is interesting to compare multidimensional array I/O rates using this method with fully parallel output.
- `sharedfp` tests I/O using a shared filepointer. This is frequently necessary when writing a log file or when checkpointing.
- `transpose` measures I/O when performing an out-of-core transpose operation on large arrays. Transpose operations are frequently used when performing multidimensional Fourier transforms.

Attention has focussed on tests that address fundamental issues. These are the low-level class of tests and the matrix tests in the kernel class.

2.3 Performance tools

The requirements for a detailed investigation and analysis of the parallel I/O system are not compatible with an automated style of benchmark which supplies a single number characterising performance at the push of a button. We anticipate that the tests will be employed by someone who already has a reasonable knowledge of the system under test and who can estimate the range of parameter values that are of interest. The tests therefore do not self-scale over the huge ranges potentially possible and the user is fully in control of selecting parameter values. As his or her understanding of the system evolves, the user will be able to use more of the tests to concentrate attention on the parameter space of greatest interest.

2.4 Simplicity of tests and their usage

It is important to build upon past work in similar areas. For example, PARKBENCH has been criticised on the basis of relevancy, expense and ease of use [7]. We insist upon well-defined goals that justify the run time required and in addition we require simplicity of the tests, which must be:

– Easy to understand
– Easy to use
– Have a low overhead (and therefore execute quickly)
– Small (in terms of size of code)

A primary reason for simple code is to clearly expose the precise significance of the timing measurements. The best way to fully understand the meaning of the timings is to look in the code at the location of the timer calls and this is most easily achieved when the code is simple.

The suite interface is designed to be convenient to use and the choice of which tests to run along with the parameters to use is made with a system of keywords in an input file. This system allows many tests, or one test with several parameters, to be submitted together.

In the Low-Level class, simplicity of the tests themselves is particularly important. One might imagine that low-level codes would inevitably be small, easy and quick to run and that difficulty would only arise later when the same simplicity requirements are imposed on more complicated codes. In fact, because of the large quantities of data that must be written to test the system in a genuine manner and the wide range of parameter values that must be checked, even these low-level tests can be very time consuming.

2.5 Separate data gathering from analysis

The running of the tests is separated from the analysis of the data they provide. This approach helps keep the test suite simple yet allows the analysis to be performed at whatever level of complexity is desired. This is necessary because the suite is emphatically for testing rather than benchmarking, so it requires a wider variety and more flexible and sophisticated possibilities of analysis than a benchmark would.

2.6 No data fitting assumptions

Because the suite is intended to provide genuine tests, no model for the behaviour of the data is assumed. For example, PARKBENCH has been criticised because it forces the data to fit a particular model which was not always valid. Only by looking at the raw data can one test whether a particular assumption is valid.

The analysis therefore consists of several stages starting with an exploration of the raw data produced by the simplest low-level tests. The data can provide empirical feedback to analytic models of individual I/O functions. Provided the behaviour can be understood within the context of a target model, one can then proceed with more complicated tests and more sophisticated levels of analysis. The validity of any analytic model must be checked at each new level.

2.7 Curves not single numbers

Although there is little freedom in the basic form of a low-level class I/O test, the choice of parameters, such as the block and file sizes, are of great significance. From experience with serial I/O tests it is clear that a single number describing the bandwidth is not sufficient and that at least one, and probably several, curves are needed to characterise the behaviour of the system. For example, a curve may represent bandwidth as a function of block size, with a different curve plotted for each of several different file sizes. Although we do not adopt any particular data model, we strongly recommend that a full analysis of this type is performed.

Notwithstanding the limitations of a single number, the bandwidth that can be measured by overwhelming any cache mechanism using sufficiently large filesizes is still an important reference. This bandwidth may indeed be relevant to a variety of applications where the cache mechanism is unable to operate effectively but more significantly it acts as a fixed (in the sense that it does not change as the filesize is further increased) reference for the curves which describe the full behaviour of the system.

Another issue is the accuracy of the measurements. An estimate of the accuracy should be calculated and reported as error bars on the curves.

2.8 Data from low-Level tests

For the low-level tests, considerable depth of analysis is possible and the data collected is more than just a single number per filesize and blocksize. Preliminary work has suggested that it is important to measure the time of each write command rather than simply take an implicit average by measuring the time for the full loop. This helps compensate for the fact that modern operating systems can cause large fluctuations in the time taken for successive writes, even on a quiescent machine. Although a single number describing the bandwidth at that blocksize can be derived, the additional data gathered provides interesting information about the system. For example one can check that the machine is as dedicated as expected. The data also provides information about startup effects and this data can be used to produce error bars on the curves. The quantity and potential complexity of data gathered in this way allows for different levels of analysis in the low-level class.

3 Status of the test library

The Parallel I/O Test Suite has only been extensively tested on a prototype MPI-2 implementation on a 4 processor vector system. Preliminary results suggest that most of the test suite works with publically available implementations of MPI-I/O on the UltraSPARC based E10000 (from Sun Microsystems) and the AP3000 (from Fujitsu).

Comprehensive tests on these systems using more processors (allowing a 2-D process grid) and larger problem sizes are planned. Such tests will help quantify limitations of the test suite, for instance, with `gatherscat2D`, memory will become a problem with large matrices.

4 Validating MPI-2 implementations

The emphasis of the project has been to develop a suitable test suite for MPI-I/O programs. There have been many occasions when it would have been useful to have had access to a suite that checked whether part of MPI-2 was supported correctly in an implementation.

To address the different needs of users, a multi-layered validation suite would be needed, with layers such as:

– Baseline level: functional tests to validate correct behaviour of routines used in correct programs. Test sub-classes within this class are necessary. For instance, simple tests would concentrate on a single feature set (such as MPI-I/O or single sided communications) while complex tests might also test for the correct interworking between functionalities (such as MPI-I/O and single sided communications used together). Similar test sets should be developed for each language binding and some tests should involve mixed language programs.
– Robust level: tests to assess whether “sensible” actions are taken (reporting mechanisms etc.) when errors are detected. Tests at the robust level would also determine critical resource limitations and attempt to determine what happens when these limits are exceeded. MPI-I/O contains a large amount of state information (partly because files are involved). Some of the robust tests should determine the correctness of this information and whether the same state information is returned on successive MPI calls.
– Performance level: tests here would not be a full test of all of MPI-2 features. A layered approach is again required, with low-level routines providing users with an understanding of the salient components of performance on a system and more sophisticated tests providing detailed data to better understand some of the subtle performance implications.

The Parallel I/O test suite described here can serve as a model for the latter category. It also addresses the needs for a performance I/O test suite.

There are MPI-1.1 test suites (such as the one available from Intel, [8]) that are quite comprehensive with both C and Fortran 77 interfaces. Since MPI-1.2 is a relatively simple enhancement of MPI-1.1, such existing suites can be improved to cover it relatively cheaply. The test programs from LLNL for MPI-I/O, [3], while not comprehensive, are also a start in the right direction.

So much for the good news. MPI-1.2 defines 129 interfaces while MPI-2 defines a further 188 interfaces and extends another 28. Therefore, assuming that each routine is equally difficult to test, covering the new features of MPI-2 would take about 1.5 times the effort to produce an MPI-1.2 test suite.

In addition, there are C++ bindings to take into account. Some benefits should transfer from producing a validation suite in C to C++, but it would not be trivial. Furthermore, there are a range of quality questions associated with the C++ (and to a lesser extent Fortran 90) bindings that would not be relevant for the C / Fortran 77 bindings. There are also questions relating to heterogeneous environments, particularly in terms of data formats and performance that would be interesting to answer.

Clearly, if some form of validation suite is to be produced in a reasonable time frame, a cooperative venture is required. This not only means pooling resources, but also means building a pool of shared common experience to better target effort on those areas that appear to be most relevant. This is perhaps a lesson that can be learned from the HPF-1 effort. While the HPF-1 standard defined cyclic block distributions and dynamic redistribution, most early versions of HPF either did not support these features or did not support them well. Are there similar features in MPI-2? If so, they could be omitted from early validation suites (this is from a vendor's perspective, an end user who wants to use such features would want them tested).

A comprehensive validation suite provides vendors with a way to provide quality assurance to end users. Being able to buy a validated MPI-2 implementation would be attractive. However, a truly comprehensive suite, with robust tests is more than most organizations are willing to fund. There is also the fact that the specification is not perfect and that different groups might interpret parts of the MPI-2 standard differently. A plan for an evolving suite that started with those features that are of immmediate interest, such as asynchronous I/O, and that grew as the standard was clarified and demand increased would better serve the MPI-2 community.

References

1. W. Gropp, E. Lusk, N. Doss and A. Skjellum, *A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard*,Preprint MCS-P567-0296, July 1996. HTML available at: `http://www-c.mcs.anl.gov/mpi/mpich/`.
2. The MPI-2 standard is available at: `http://www.mpi-forum.org/`.
3. The latest version of the Lawrence Livermore National Laboratory is available at: `http://www.llnl.gov/sccd/lc/piop/`.
4. P.M. Chen and D.A. Patterson, *A New Approach to I/O Performance Evaluation - Self-Scaling I/O Benchmarks, Predicted I/O Performance*, Proc 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Santa Clara, California, pp. 1-12, May 1993.
5. PALLAS MPI benchmarks. Available from PALLAS at: `http://www.pallas.de/`.
6. R. Hockney and M. Berry (Eds.). *Public International Benchmarks for Parallel Computers*, Parkbench Committee Report No. 1, Scientific Programming, 3, pp. 101-146, 1994.
7. Private comments by Charles Grassi (13/5/97).

8. The latest version of the Intel MPI V1.1 Validation Suite is available at `http://www.ssd.intel.com/mpi.html`.