Specification of the Parallel Input/Output Test Suite Version 1.0

David Lancaster

Electronics & Computer Science
University of Southampton
Southampton SO17 1BJ, U.K.
(djl@ecs.soton.ac.uk)

A Report Prepared for the Fujitsu European Centre for Information Technology (FECIT).

Contents

1 Goals and Motivations

The technology of parallel processing has matured to the stage where for many computationally intensive problems it is the method of choice. However, parallel I/O is still in a rapid state of change and the Parallel I/O Test Suite is a timely tool to investigate the new developments. One of the important recent developments has been the publication of the MPI-2 standard [1] which contains a chapter on I/O (henceforth referred to as MPI-I/O) and presently this is the best way of portably implementing parallel I/O at the application programming level. MPI-1 has been widely taken up in the community and provided MPI-2 is similarly adopted it will be possible, for the first time, to make comparisons of parallel I/O between widely dissimilar systems. The test suite will therefore remain firmly within the context of MPI and besides testing the intrinsic I/O performance the suite will implicitly be testing aspects of the MPI implementation such as the MPI library and the compilers.

This suite of programs is intended to test parallel I/O at several different levels organised using the standard approach of “Low-Level” and “Kernel” classes. Firstly, a parallel I/O system needs carefully instrumented, low-level test programs to facilitate improvements at both the MPI-I/O implementation level and the file system level. The programs in this class are known as “Low-Level Tests” and the overriding design criterion is that they be simple and make clearly defined measurements. Most applications will make more complex use of I/O and to reflect this there is an additional classe of test called “Kernel”. “Kernel” programs identify a set of characteristic I/O behaviours that recur in many typical applications. These essential parts are abstracted as free-standing programs and allow the characteristic behaviours to be investigated in detail. This allows the relative efficiency of different typical characteristic I/O behaviours to be measured and provides an evaluation framework against which future I/O intensive applications can be assessed.

The tests are used to generate clearly defined timing measurements, which are analyzed by the tools discussed in section 7. These measurements and the results of analysis will address the parallel I/O at levels running from filesystem and MPI implementation, to the application level. The levels broadly correspond to the two classes of test. The questions the test suite sets out to answer and the kinds of problems it should help solve are listed below:

- Performance tuning and debugging of filesystems and implementations.
- Comparison between different systems and implementations (especially in the context of the characteristic kernel types)
- Performance tuning of applications within the MPI context.
- Measuring parameters needed to predict performance.

In summary, the analysis of the low level class tests will lead to better understanding of I/O at file system and MPI implementation levels, thus leading the way to improvements. The analysis of the kernel class tests will aid tuning at the application level.

In the following section we describe in broad terms how the test suite is designed to achieve these goals. We concentrate mainly on the design of the tests themselves, but proper analysis of the test measurements is critical for effective use of the suite, so in section 3 the general analysis strategy is outlined. Conventions we apply to the whole suite of tests are listed in section 4. The aim and structure of each of the tests are specified in sections 5 to 6. A schematic description of the important parameters for each test is given in each case, but for full details of all the parameters and the procedure for running the tests, see appendix B and the “Users Guide”. The analysis tools are described in detail in section 7 along with examples and concrete instances of what measurements can be made and what questions may be answered.

Because attention is focussed on testing the rather new field of parallel I/O, in this version we concentrate on the tests that address fundamental issues. These are the low-level class of tests and in the kernel class, the tests performing I/O of multidimensional arrays.

2 General Approach

In order to best address the goals, we emphasise tools rather than benchmarks so the suite is more under the control of the user, and thus implicitly less “automatic”. With this in mind, the analysis requirements are greater than for a simple benchmark so we separate data gathering from analysis in order to simplify the tests themselves, yet allow analysis at varying degrees of sophistication. Learning from criticisms of past benchmarking activity, we insist on simplicity both of the tests themselves and of the run procedure. We note that the tests are not intended to comprise a validation suite for MPI-I/O.

Each of these design choices is expanded upon in the subsequent sections.

2.1 Low-Level and Kernel Classes

As discussed above, we have decided to classify the tests as either Low-Level or Kernel classes along the lines of PARKBENCH [3], with the intention of testing the system at different levels.

- Low-Level : Measures fundamental parameters and checks essential features of the implementation. Allows performance bottle necks to be identified in conjunction with expected performance levels.
- Kernel : Tests the MPI implementation at a more advanced level with a wider range more characteristic of real applications. Allows comparisons between implementations.

2.2 Performance Tools

The list of goals makes it clear that the suite is not intended as a benchmark suite although further work could provide the basis for one. 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 will therefore 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. As an illustration, section 7 provides an example showing several stages of possible analysis.

2.3 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 [4]. 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 – at least for the Low-Level class).

Simplicity is also important in the structure of the suite. For example, a user only interested in running Low-Level tests should be able do so despite having difficulty compiling the compact applications.

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.

2.4 Lessons from Serial I/O Testing

Some common issues in I/O testing that will have a bearing on the whole suite of tests have become clear in the context of serial I/O testing [7]. 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. There is a well-known strategy to guarantee that the information is on disc allowing the limiting bandwidth to be measured: one simply writes sufficiently large files as to fill up and overwhelm the cache. This strategy is not very efficient since it requires considerable time to write the large files required, and moreover it can be regarded as artificial to bypass the cache because it is an intrinsic part of the I/O system. The resolution of this issue is to measure a curve of bandwidths for different filesizes, the large filesize limiting value providing a useful reference that is indeed relevant for many applications. We expand further on this issue when we discuss the analysis stage in the next section.

Some other points can be learned from the experience with serial tests. When mixing writing and reading, the same cache effects found in the write tests can be seen when reading data that was written not long ago and 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 resolve 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.

Parallel I/O testing is not a very developed art mainly because until the advent of MPI-I/O, there was no possibility of writing portable tests. The BTIO [8] test used a pre-standard version of MPI and bears some similarity to our kernel tests.

3 Overview of Analysis

Some of the criteria that we have chosen to design the analysis stage are discussed below. We emphasise the choices made for the Low-Level class which generates more data and consequently requires more extensive analysis. More detailed descriptions of the actual steps taken in performing analysis are given in section 7.

3.1 Separate Data Gathering from Analysis

We shall separate the running of the benchmark from the analysis of the data it provides. 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.

3.2 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, then one can proceed with more complicated tests and more sophisticated levels of analysis. The validity of any analytic model should be checked at each new level.

3.3 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, for example 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.

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.

3.4 Data from Low-Level Tests

For the low-level tests considerable depth of analysis is possible and the data we collect is more than just a single number per filesize and blocksize. We have found in preliminary work 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 is because, in the presence of a cache, modern operating systems can cause strong fluctuations in the time taken for successive writes even on a quiescent machine. Although one might finally derive a single number describing the bandwidth at that blocksize, the additional data we gather provides interesting information about the system. For example one can check that the machine is as dedicated as expected and the data also provides information about startup effects. The quantity and potential complexity of data gathered in this way allows for different levels of analysis in the low-level class. These issues are explained more fully with examples in section 7.

4 Conventions

The test suite should not be regarded as a set of individual programs: there is an overall software engineering philosophy which is formalised in the set of conventions or idioms: the specific elements of software style, defined for the suite. We first list some general conventions before discussing the the environment idiom and some idioms related to errors, file formats and run rules. Some of these conventions are expanded upon in greater detail in the appendices. The low-level class of tests requires some additional idioms which are discussed in the following section.

4.1 General Idioms

- Classification: Classify the different levels of I/O test programs in the suite along the lines used by PARKBENCH. That is, as low-level and kernel.
- Filesizes: In all cases we expect the filesystem to be large since we are interested in the performance of applications requiring substantial I/O transfers. Based on real world considerations we anticipate memory size per processor to be at least 64Mbytes and bandwidths to be at least 2Mbytes/sec. Typical filesizes will be up to the order of Gbytes, though for efficiency this should be reduced as much as possible compatible with not compromising the measurements.
- Measurements.
    - Timing: For portability we will use the `MPI_Wtime` function that returns a (double) time in seconds. The resolution is determined using `MPI_Wticks` and we insist on a minimum resolution of 10^{-2} seconds. Although we eventually transform to rates measured in Mbytes/s, the raw timing data is directly useful. It often happens that for small buffer sizes the time of each write call is almost independent of the amount of data written. To investigate issues such as this, it is best to output the raw timing data rather than making some transformation within the test program.
    - Trace: All measurements will be performed with Trace off.
    - Work Units: Where appropriate use we will use Mbytes. All data lengths are multiples of 8 bytes.
    - Preallocation: Space for files is always preallocated using the MPI routine `MPI_file_preallocate`. This routine ensures that storage space is allocated for the first x requested bytes of the file. Regions of the file that have previously been written are unaffected. For newly allocated regions of the file, `MPI_file_preallocate` has the same effect as writing undefined data. If x is larger than the current filesize, the filesize increases to x. If x is less than or equal to the current file size, the filesize is unchanged. The time required to preallocate is measured and reported.
    - Validation: All runs are timestamped and automatic checks are made where possible.
    - Irregularity: The tests are to be run on a dedicated machine and consist of only a single application. The parallel I/O load induced by multiple executing jobs will not be considered in this version of the test suite.

- Language: The suite is written in F90. As a matter of style, new code is restricted to the F[5] subset of F90.

- Error Idiom: The suite has a well organised set of descriptive error messages that in the event of a failure are passed through the layers of wrapper to the user. MPI provides error facilities for I/O and the tests are designed to catch these errors and continue executing.

4.2 MPI-I/O and Filesystems, the Environment Idiom

A significant feature of MPI is that it hides physical details of the machine, such as whether a processor contains the hardware necessary for performing direct disk I/O. We exploit this feature by writing the tests using only standard MPI which therefore allows a clean separation from any machine-dependent features supplied by the user. The prime example of a machine dependent parameter is the name of the filesystem where test files are to be written or read. The totality of such information constitutes the environmental idiom.

The low-level tests may be run several times with different file arrangements, corresponding to several possible environments. The environments or output patterns are specified by the user through entries in an input parameters file. The entry will supply the directory path information necessary to write and read files in the proper locations.

It is tempting to have a general architecture in mind when giving guidance about where files should be written, and we shall sometimes refer to "local" disks in the sense appropriate to an architecture consisting of a bunch of workstations. But it is important to stress that this nomenclature does not fit all architectures, for example there may not be any local disks on a particular processor (*eg* the Fujitsu VP700). Potential problems that might occur with other architectures need to be identified and recognised at an early stage. We shall return to this point when we discuss the tests in detail, and provide guidelines for what environments we expect to be useful.

As a generalisation of the environment idiom, MPI allows various hints, such as striping factors to be supplied by the user. The use of these hints will be allowed in the test suite, but must be reported. The user supplies the hints through the keyword mechanism of the input file and the reserved keywords for MPI file hints are given in Appendix B.

Although the tests have been designed with hard disk storage devices principally in mind, other storage devices can also be tested. For example, tests could be repeated for solid-state storage devices, if they exist.

4.3 Test Suite Organisation

The test suite will be organised using a hierarchy of wrappers as described in Appendix A. One of the motivating factors for this form of organisation is ease of use: the user should be able to run a Low-Level test soon after receiving the distribution and several tests can be run automatically in succession.

Compilation will have the option of generating separate executables, for tests in each of the classes or of generating a single executable that can run any of the tests. This may partially avoid problems if for example, the user only wants to run a single small test.

A series of tests, or a single test with a series of parameter values can be run. The method of doing this is explained in appendix B. Guidelines on a sensible sequence of tests to run are given in section 7.

4.4 File Formats

The tests are run either by a single executable which covers all tests, or by one of three executables covering a class (Low-Level, Kernel) of tests. The information of which particular tests to run, along with their option and parameter values, are passed to the executable via an input file called “iotparams.in”.

The information is passed using a system of keywords which allows the files to be easily understood and also facilitates some of the analysis. The output file is in a similar format. The keyword system is explained in greater detail in appendix B where lists of keywords are given.

4.4.1 Input File Format

The input file consists of a series of information blocks describing a particular test. Each block contains:

- Class name (Low-Level, Kernel)
- Test name
- General file hints (reserved keywords from MPI standard, includes filename)
- Specific options or parameters for that test (keywords for that test)
- Comments (any line starting with a hash symbol)

Each test has its own set of required parameters so there are keywords associated with every test and these are listed in in appendix B. An example input file is shown in Appendix C.

4.4.2 Output File Format

The output file contains the timing data measured by the test and all diagnostic information for the test. Again it consists of a sequence of information blocks describing each test run, each block has a header containing the diagnostic information and then the data. An important design consideration is that the information blocks in these files can be concatenated or split up at will to form new files.

The header contains a copy of the input file information along with some runtime information, such as timestamp and number of processes. The format of the data section depends on the test. In the case that a test is to be run with a series of parameters, the output file may contain several subsections running through the parameter values given in the header. Each subsection will have a sub-header containing the parameter value for that subsection.

5 Low-Level Class

Low-Level tests measure the most basic I/O timing parameters available from an MPI test. The tests read and write a steam of bytes in blocksize quantities.

The basic I/O tests of reading and writing are simple in concept and the low-level tests need to reflect this. The primary requirements that they be easy to understand and that the raw data have a clear meaning are fulfilled by displaying the heart of the code which will basically be a loop timing successive reads or writes. We expect that anyone using the tests will have a knowledge of MPI-I/O and could indeed have written their own low-level tests which would be similar since there is little latitude in how these tests can be written. Simplicity of the tests, both in the sense of simple algorithms and in the sense of short programs, is vital in order that the code can be inspected and the precise meaning of the timing measurements be determined. In section 5.2, a short code fragment is provided for the simplest low-level test to illustrate this.

The low-level tests are basic write/read tests of bandwidth, but in a multiprocessing environment, various configurations are possible. We will consider two ways in which a file may be read/written, and these will form the two basic low-level tests. These two tests follow directly from the two simple ways in which a file may be opened in MPI: either with MPI_COMM_SELF or with MPI_COMM_WORLD, the file is opened *individually* by each processor, or is opened *collectively* by all the processors (note that MPI has a more precise definition of collective). In our tests we shall simplify this distinction even further and in the first case only consider an MPI program running on a *single* processor whereas the file will be opened by *multiple* processors in the second case. The tests will be called `single` and `multiple` respectively. Another pair of low-level tests called `singleI` and `multipleI` will test asynchronous I/O. As we have already mentioned, the actual location of the file in a file system is given by the environment provided by the user and although for given architectures there will be natural choices for where the file should be written in each test, which we shall point out, the choice finally depends on the particular way the machine is set up.

The low-level tests must be repeated with various choices for the blocksize parameter. For the multiple test running on several processors as described further on, several parallel sets of data from each processor must be analyzed together and this will require further analysis tools.

We anticipate that the main interest will concentrate on the transfer of fairly large size blocksizes, so the incorporation of a timer call for each transfer will not cause any appreciable overhead and there will be no need to analyse the effect of the timer call. Of course, this assumption can be verified on every system tested.

5.1 Approach for Low-Level Tests

5.1.1 Compare with standard UNIX I/O

It is clearly vital to compare the results of the low-level tests with I/O tests that do not use MPI. We are not able to provide such tests since they will be different depending on the site. Nonetheless, we have written the low-level tests in a transparent fashion in order to expose the essential part of the code and we expect that this can be modified in order to make comparison tests. As an example, we do provide one simple modification, `sinunix`, of the “single” test using standard UNIX read and write operations in FORTRAN.

5.1.2 Data Analysis

For the reasons explained in section 3.4, the low-level tests will measure the time for each I/O command in a loop and an example of the code is given in section 5.2. They therefore generate a considerable amount of data which in keeping with the design choices, must be analysed separately. Indeed, it is the quantity and potential complexity of data gathered in this way that allows for different levels of analysis.

A single bandwidth describing the I/O rate at the selected blocksize is simply obtained by averaging the data. This recovers the result that would have been obtained by placing timing markers outside the loop. Other types of analysis which may reveal information about cache sizes and startup effects are discussed in section 7. No particular model is assumed for fitting the data.

5.1.3 Idioms for Low-Level Tests

There are some additional idioms that are necessary for the Low-Level class of tests. The way in which these tests are run must be carefully specified to ensure efficiency. Two main points arise, the range of values to test and the issue of self checking. In order to choose appropriate parameters, it is sensible to run some short preliminary tests before the main ones.

- We write and read bits in the form of double precision floating point numbers. The MPI etype is `MPI_double_precision`. The numbers are randomly chosen from a distribution (uniform [0-1]), but subject to the needs of checking (see below).

- The low-level class does not test advanced features of MPI, so the `filetype` will not contain holes and will always be a contiguous set of `MPI_double_precision` data units. Similarly, offsets will be evaluated explicitly and we shall not use any MPI call which updates its own file pointer.

- As emphasised earlier, the user is responsible for choosing the parameter range of the test. This is the prime area where efficiency savings can be made.

    - The reason we write large files is to ensure that the cache is no longer affecting the bandwidth, and that we therefore have a reproducible reference measurement. As soon as we are in a regime where we are sure that this is the case, no further information is obtained if we continue to extend the file. The data scatter makes it difficult to know when this asymptotic regime starts, nonetheless it should be possible to determine it approximately in preliminary tests. An example is given in section 7.

    - Appropriate choice of blocksizes is also important because the potential range over which we might test is very wide: from 1kB up to at least 2MB.

- Self checking in this context would normally be to use a standard UNIX command to read and check the file written with MPI. This is very time consuming for the size of files that are required, and given the reliability of modern machines it is probably unnecessary. It is more efficient to use the MPI read tests to check the earlier MPI writes. Furthermore, it is acceptable to perform the checking statistically rather than to test every number written. The method of doing this must be matched with the way the buffer is filled with random numbers.

- Fuzzy blocksizes. It is sometimes helpful to use varying blocksizes that will not lie on machine blocksize boundaries. Each subsequent I/O request has a slightly different blocksize chosen from a probability distribution centered around the mean blocksize. This can give rise to smoother behavior as the mean blocksize is varied. In the interests of simplicity we shall not employ this technique in the first release of the test suite.
- Striping. The MPI-hints that are most relevant to the low-level class are the ones that relate to the striping parameters. The effect of varying these should be investigated. The keywords and precise meanings are given in Appendix B.2.

5.2 Single Test: `single`

The "single" case is a very limited test, it does not employ the parallelism of the machine and simply measures the bandwidth of a single processor to write and read a disk file. The test is intended as a reference against which the results of the full multiple test should be compared. A problem occurs with the environment because one would like to measure the bandwidth for the processor to write to its "local" disk (one that is closely attached to the processor), but the existence of this kind of architecture cannot be guaranteed. A more severe problem is that often the submission procedure does not allow one to specify which physical processor to use for the run. This complicates the selection of a "local" disk, since it must be done before submission, and different machines have different ways of dealing with the problem. For these reasons it is important to report the details of the set up and also the processor that was eventually used for the run. In practice we do not anticipate problems because the appropriate disk to write to will often be obvious, and should coincide with the one used for the multiple test in order that meaningful comparisons can be made.

In principle, the results of this measurement should be the same if N processors each simultaneously write a file to their "local" disk because the network is not used, and in some circumstances this could provide an additional check. Another similar test would be to run on N processors and write successively from each processor which would automatically avoid any problems with the scheduler choosing the single processor in an uncontrollable way. But this method is extremely time-consuming, and in keeping with the discussion about simplicity and efficiency we have decided to rely on the user's experience and knowledge of the system and use only the `single` test.

In this "single" case, there is little point in testing optimisations such as collectivity. Although a non-blocking version of this test could be run, a better understanding of asynchronous I/O requires additional low-level tests including a calculational component. The test `singleI` is discussed in a later section.

Although we have described the test in words above, a more precise definition comes from the following Fortran code which forms the heart of the test:

```
offset_niter = 0
told = MPI_Wtime()
do $j = 1$, niter
   offset = offset_niter
   call MPI_File_Write_at(fp, offset,
     buf(1), bsize, MPI_DOUBLE_PRECISION, status, ierror)
   tnew = MPI_Wtime()
   wtime($j$) = tnew - told
   told = tnew
   offset_niter = offset_niter + bsize
enddo
```

In this example, the timing data is stored in the array `wtime` which is written to the output file at the end of the test.

The second part of the test reads the data just written. An `MPI_File_Sync` instruction separates the two parts of the test to ensure that the data is truly on disk. This operation does not however, guarantee that a copy of the data does not still reside on the cache. In the read part of the test, the data must be touched in order to check that it has actually been read. Some self-checking ensuring consistency between the written and read data is performed at this stage.

A schematic description of the most important parameters for this test is given below. A more detailed description of the keywords needed to run the test and the method of submitting several parameter values for successive runs is given in appendix B.

`single(filesize, blocksize)`

`filesize:` We anticipate a typical filesize of 1GB, but provided the bandwidth has reached some asymptotic value there is no point in increasing the filesize further. We suggest that filesizes be increased by factors of 4 between subsequent test runs. Note that traditional UNIX filesystems are limited to 4Gbyte. The tests should not be allowed to take too long without very good reason.

`blocksize:` The blocksize is the size of the buffer written each time the loop is traversed. Suggest sizes: $1, 4, 16, 64 \ldots$ kB (up to several MB). This range is very wide and the tests will take a very long time, so we strongly recommend the user to restrict this range based on her knowledge of the system, and then to make some preliminary exploratory tests that use smaller filesizes in order to find the region of interest for her machine.

5.3 Multiple Test: `multiple`

The "multiple" case begins to test parallel I/O and it is straightforward to open a disk file from all processors using the MPI_COMM_WORLD communicator. The disk where the file is written could be any one of the locally attached disks mentioned above or, if the hardware and software requirements for higher performance exist, it could be striped over a series of disks.

Besides the basic test that uses the blocking `MPI_File_Write` call, this test should be used to investigate optimisations based on the `MPI_File_Write_all` collective (in the MPI sense) call. It is likely that the MPI-hint parameters that control the collectivity should be varied in conjunction with the striping parameters.

The loop forming the heart of the code is precisely the same as in the case of the "single" test, and in fact the main difference between the tests is in the way that the file was opened. In the "multiple" case timing data is gathered independently on each processor, it is automatically gathered together at the end of the test and written to the timing output file.

The possibilities of the environment are not necessarily more complicated than for the "single" test, indeed there may only be one file system appropriate for large files opened from multiple processors in this way. In any event, comparison of the timing data for accessing the file from different nodes will provide interesting information about the system.

The read part of the test follows the same pattern as for the `single` case.

The parameters for this test comprise the block and file size which are subject to the same comments as the single test, a new parameter, the number of processors and a switch for the collective form of the I/O routines. Collectivity and striping hint values should be varied and tested.

`multiple(filesize, blocksize, nprocs, collective)`

`filesize:` We anticipate a typical filesize of 1GB.

`blocksize:` Suggested sizes are: $1, 4, 16, 64 \ldots$ kB, up to several MB, as for the `single` test.

`nprocs:` The number of processors. Note that MPI does not specify whether a particular processor is able to perform I/O or not, but that `num_io_nodes` exists as an MPI hint. An ideal system would show no difference in bandwidth per processor as the number of processors varies. Note that this parameter is usually supplied on the command line when the test is submitted rather than in the input.params file.

`collective:` This switch can be set to test collective or non-collective forms of MPI-I/O.

5.4 Asynchronous Tests

The low-level tests discussed above are based on a loop containing an I/O routine alone. They are therefore not suitable for testing asynchronous I/O because the whole point of the asynchronous approach is to perform I/O while the CPU is occupied with some other computation. This introduces an new variable into the problem: the content and duration of the "other computation".

The asynchronous tests are based on the same structure as the `single` and `multiple` tests, but interleave an additional computation into the central loop. The new tests are called `singleI` and `multipleI` to follow MPI naming conventions. The form of the interleaving computation has provisionally been chosen as a matrix-vector multiplication loop. Some of the issues that are important in choosing the form of interleaving computation are the vectorisability of the code and the memory usage. Exactly how the computation interacts with the asynchronous I/O is a complicated problem. The patterns of real applications are best fit if the data written depends in some way on the results of the computation preceding the write instruction in the loop.

The most important aspect of the interleaving computation is the length of time it takes to run once. This time duration is controlled by user parameters that fix details of the computation and once these parameter is set, the precise time duration is obtained in the first cycles of the test which do not include I/O to time the computation in a synchronous environment. Once these cycles have been completed, the main loop using non-blocking versions of the I/O routines is started. From a comparison of the non-blocking cycles with earlier results on blocking I/O, the effectiveness of the asynchronicity may be deduced. The effectiveness will depend on the parameter that fixes the duration of the computation. If this parameter is chosen so that the computation completes very quickly, the tests will resemble the standard synchronous `single` and `multiple` tests.

5.4.1 `singleI`

The pattern for this test follow the blocking version `single` except for the comments above. In particular this test should be regarded as a single processor reference for asynchronous I/O against which the multiple version can be compared.

The comments about the environment idiom carry over from the blocking test as do the comments concerning MPI-hints. The same techniques are used to separate writing and reading phases of the test and self-checking also occurs as before.

`singleI(filesize, blocksize, tasklength)`

`filesize:` This keyword has the same meaning as in the blocking case and the same values should be used.

`blocksize:` This keyword has the same meaning as in the blocking case and the same values should be used.

`tasklength:` The duration and form of the interleaving computational task are determined by several parameters controling the matrix-vector multiplication and given the overall name `tasklength`. In preliminary use this parameter should be chosen to give a short computation in order to check results against the blocking version.

5.4.2 `multipleI`

The asynchronous version of `multiple` is based on the blocking version `multiple`. The additional parameter describing the duration of the computational task that interleaves the I/O routines takes the same form as in `singleI` above.

The blocking `multiple` test is intended to help test the collective features of the MPI implementation and underlying parallel file system. The asynchronous version known as split-collective can also be tested and a switch is provided for this purpose. The relationship between striping parameters and collective parameters is investigated at the blocking level, so for this test they should be adjusted together.

`multipleI(filesize, blocksize, nprocs, tasklength, collective)`

`filesize:` This keyword has the same meaning as in the blocking case and the same values should be used.

`blocksize:` This keyword has the same meaning as in the blocking case and the same values should be used.

`nprocs:` The number of processors.

`tasklength:` The duration and form of the interleaving computational task are determined by several parameters controling the matrix-vector multiplication and given the overall name `tasklength`. In preliminary use this parameter should be chosen to give a short computation in order to check results against the blocking version.

`collective:` This switch can be set to the collective or non-collective forms of MPI-I/O.

6 Kernel Class

Kernel tests are larger, more complex codes than the Low-Level ones. They represent the I/O-intensive kernels common to a variety of different applications. As such, these tests are characteristic of typical I/O patterns, and allow performance for these characteristic patterns to be investigated in detail. Because of their increased complexity, the kernel tests also exercise more sophisticated features of MPI, and can check the performance of the MPI implementation at these advanced levels.

6.1 Characteristic I/O Patterns

The characteristic I/O patterns that we have identified and which form the basis of the Kernel tests are listed below along with typical applications that use that I/O pattern. Each of the tests will be discussed in detail in the following sections. We emphasise that the kernel tests are synthetic: they are intended to test a typical I/O pattern and are not expected to follow precisely the algorithm of any particular application.

- I/O of regular multidimensional arrays. Simulations of 2 or 3 dimensional physical systems, for example computational fluid dynamics, seismic data processing and electronic structure calculations.
- Non-sequential I/O. Database applications, medical image management and recovery of partial mapping images.
- 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.
- I/O using a shared filepointer. This is frequently necessary when writing a log file or when checkpointing.
- Large FFT's and permutations. The algorithms appropriate for very large, out-of-core FFT's display interesting characteristic patterns. For multidimensional FFT's transpose operations may be necessary.
- Out-of-core solvers. Hartree Fock calculations and large matrix operations such as LU decomposition.
- I/O of data-structures based on irregular meshes. Simulations of physical systems, for example weather forecasting, crash analysis and computational fluid dynamics for irregular structures.

The first pattern, I/O of multidimensional arrays, is the most important in practice. Whereas the Low-Level tests wrote a stream of bytes with trivial structure, these tests perform I/O of arrays with a multidimensional structure that often corresponds to some physical geometry. This is a very common I/O requirement, and there are many sub-varieties depending for example on grid regularity, array dimension and the way the geometry is mapped onto the processors. Collective I/O routines are often designed with this type of I/O pattern in mind, so these tests will be particularly relevant for investigating this optimisation. In a major set of the kernel tests we consider I/O of different dimension arrays and each test has a wide set of parameters. The user of the tests will select which dimension kernel is of greatest interest to him or her. The 3D array test, `matrix3D`, maximizes the difficulties of the array nature of the I/O while remaining a standard step widely used in practical applications. It therefore stresses the aspects of the I/O most relevant to this pattern. The analysis is however necessarily more involved than in the 2D case. It has also been argued that the 2D case already contains the essential aspects of the pattern [9] and we therefore recommend that the 2 dimensional matrix I/O test `matrix2D` be used first.

The last two items listed certainly refer to interesting and important real applications. Synthetic kernel versions containing the characteristic I/O patterns for these applications are harder to identify. The reasons are slightly different in each case: the range of different problems requiring out-of-core solutions is wide and the I/O pattern for Hartree Fock is distinct from that needed for large matrix operations; and the way that data on irregular meshes is presented is not standardised. Although these problems could be avoided by choosing one particular example of each application and considering the tests as compact applications, we feel that the potential for useful kernel tests exists and that these tests would be more helpful for investigating the system. These kernel tests must be well thought out, which requires expertise in each field and considerable work in selecting the correct algorithms to develop the precise details of the tests. We therefore postpone the introduction of these two kernel tests until a later version of the test suite, but meanwhile we give a preliminary overview of each characteristic pattern.

The kernel tests enable the efficiency of I/O with a characteristic pattern to be determined and compared between different systems and in some cases, between different patterns. This information will provide insight into the likely behaviour of full applications with that I/O pattern. The type of analysis required is therefore rather different from the case of the low-level class of tests and this is reflected in the tools available. Advanced features of the MPI implementation are also tested by the kernel class and this allows problems at this level of the implementation to be identified.

We anticipate that some exploration of the parameter space will already have been done using the low-level class before Kernel tests are run. The interesting regions will therefore be known and the kernel tests need not be run for excessively large range of parameters.

6.2 I/O of regular multidimensional arrays

Applications that simulate physical systems using a discrete lattice basis come in many varieties with their own particular field structures, depending for example on the choice of grid and the way the complete physical system is distributed on the parallel machine. Rather than attempt to rigidly follow one of these applications, the kernel tests take a more synthetic approach that should be generally applicable by isolating the essence of the problem. A compact application involving I/O of arrays will be considered as a test in that class.

Evidence that this type of data structure is common comes from the 18 major application suites selected for investigation by the Scalable I/O consortium (SIO) [10]: 12 of these applications perform I/O of regular arrays (an additional 4 use irregular mesh structures). Note that 3D is not the maximum dimension used in practice, the seismic data processing application employs arrays of up to 5 dimensions. Also note that beyond 2D, the access patterns observed in practice on parallel file servers do not change in character, only in degree [9].

We consider I/O of regular arrays uniformly distributed on the machine. Two separate tests that consider the 2D and 3D cases are provided The arrays are square of size $N \times N$ (or $N \times N \times N$ in the 3D case). The processes are viewed as a regular logical grid $px \times py$ (or $px \times py \times pz$), and the matrix or 3D array is distributed uniformly across this grid, each processor containing a subarray of size $N/px \times N/py$, ($N/px \times N/py \times N/pz$). More complicated schemes are sometimes employed, for example a cyclic distribution of smaller arrays can improve load balancing in certain applications, but in the interests of simplicity we remain with this pattern.

In contrast to the low-level tests where the main user-supplied parameters where the number of processes, the filesize and the blocksize, here the main parameters are the problemsize N and px, py (subject to the restriction that $px \times py$ is the total number of processes). The filesize is given by N^2 times the size of the `MPI_double_precision` data structure, and is reported in the output. N should be divideable by px and py. The parameter space still contains the filesize, but new parameters that describe the way the physical system is distributed on the machine, replace the blocksize. In 3D, pz is an additional parameter and the expression for the filesize is suitably modified.

The test program measures how long it takes to write and read the complete array. The file containing the array is in the format that a single processor would write it, that is, in column order (for FORTRAN). The information can be read on a different number of processes from that it was written from. MPI provides various ways of constructing fileviews that allow the processes to write their subarrays in the correct parts of the file for overall consistency. These aspects of MPI are tested in these kernels. The organisation of the array is compatible with the form normally expected in High Performance Fortran.

Because of the importance of this I/O pattern, there has been considerable research on optimising it. The most important optimisation is to make the I/O collective, that is, coordinated and conforming to the underlying data storage pattern. Many small non-contiguous accesses are replaced by a few large contiguous accesses. Sometimes collective optimisations are implemented at a fairly high level (eg Panda [11]), which would be above the level visible to MPI. There is however, no difficulty in implementing these optimisations at the MPI level and indeed they are expected in the MPI standard. In the MPI standard, hints can be supplied both for generic collective optimisation and also for specific optimisations for multidimensional arrays. These latter hints contain the word "chunked". The interplay between the underlying hardware and the way in which collective optimisations are implemented is fascinating. Indeed it is rather the aim of the test suite to investigate this interplay than for us to anticipate the connections.

One of the important roles of the multidimensional array I/O kernel tests is to investigate the effectiveness of the collective optimisation. In doing this, there is considerable dependence on parameters. Both the explicit parameters of the test and the parameters that are supplied to MPI via the MPI-hints are crucial to proper use of the tests. In particular the keywords, relating to collective buffering such as `collective_buffering`, and `cb_block_size` as well as the keywords containing "chunked" must be investigated in these tests. The full list and meanings of these keywords are given in the Appendix B.2. Guides as to appropriate values to chose for these parameters are given in the detailed discussion of each test. The hints associated with different striping parameters should already have been investigated using the Low-Level tests, but they should naturally be matched to the other parameters associated with collective output. In the analysis section we describe the tools developed to understand the results of these tests and also give some examples of the possible influence of the parameters.

6.2.1 2D matrix I/O: `matrix2D`

This test reads and writes a complete, distributed square 2D matrix. The matrix is distributed over the processes in a uniform manner, each process containing a $N/px \times N/py$ subarray. The matrix must be divided exactly into sub-matrices with no remaining elements.

Applications that write 2D matrices are not rare, and this test will be directly relevant to them. More generally, the `matrix2D` test is the simplest multidimensional array test to analyse, and if the parallel file system has many variable parameters, it is recommended that this test should be used before proceeding to the 3D version.

The test program measures how long it takes to write and read the complete matrix. These timing measurements are single measurements in contrast to the low-level case where many timings were generated. They thus include the effect of the initial cache dominated period, but this effect should be well understood from earlier use of the low-level class tests. Besides measuring the total read and write timings on each processor, it reproduces some of the low-level timings such as the time taken to open and to close the test data file. Details of all output information, layout and keywords are described in the appendix.

As a general guide, the problemsize (related to the filesize) should be chosen to investigate the characteristic I/O regions identified in the low-level tests. These will usually be the plateau ranges dominated by cache and disk I/O respectively. It is more difficult to give guides as appropriate ranges of the other parameters relating to the data distribution and to the optimisations. Certainly curves should be generated, but the dimension of the parameter space is large if the collective and chunked optimisations are invoked. We expect that for good performance, the values of all the parameters will be closely related. Regions of bad performance are also of interest and in particular it is interesting to check bandwidths when the chunk parameters are not aligned with the subarray sizes. It is one of the aims of the test suite to investigate the effectiveness of various types of hardware and optimisation software and how these will affect performance profiles.

In order not to over-complicate the analysis, the I/O operations of the sub-matrix on each process are done with a single MPI command. This is possible through the use of advanced MPI filetype mechanisms. It is this single operation that is timed on each process and forms the output of the test. Because this single instruction is the heart of the test, and there is no explicit loop, there is no point in considering an asynchronous version of this test. Non-blocking I/O is best tested at low-level using the `singleI` and `multipleI` tests.

`matrix2D(problemsize, xproc, yproc)`

`problemsize` For problem size, N, the square matrix is of size $N \times N$ elements. The actual file size is N^2 times the size of the `MPI_double_precision` data structure. The value of N should be selected to work in the cache and in the disk regime identified as plateau regions in the Low-Level tests.

`xproc`, `yproc` Process grid configurations. For each test run a pair of numbers are taken: one from `xproc`, px, and one from `yproc`, py. This pair of numbers specifies the number of processes in a row, px (or column, py), of the logical grid of processes. The user must ensure that $px * py$ is the total number of processes.

`collective:` This switch can be set to the collective or non-collective forms of MPI-I/O.

The MPI-hint parameters that are especially relevant for this test are the ones concerning collectivity: `collective_buffering`, `cb_block_size`, `cb_buffer_size` and `cb_nodes` and also the ones concerning multidimensional array I/O: `chunked`, `chunked_item` and `chunked_size`. Full descriptions are provided in Appendix B.2.

A series of tests can be conducted as explained in the appendix, but the total number of processors should remain fixed. This may be hard to achieve in practice.

6.2.2 3D matrix I/O: `matrix3D`

This test reads and writes a complete, distributed 3D array. The matrix is distributed over the processes in a uniform manner, each process containing a $N/px \times N/py \times N/pz$ subarray. The array must be divided exactly into sub-arrays with no remaining elements.

It is recommended that these tests be employed after the 2D matrix test results have been understood. The parameters are almost identical to the 2D version and the timing measurements are also very similar. The analysis is likely to be more complicated than for the 2D case.

```
matrix3D(problemsize, xproc, yproc, zproc)
```

`problemsize` For problem size, N, the array is of size $N \times N \times N$ elements. The actual file size is N^3 times the size of the `MPI_double_precision` data structure.

`xproc`, `yproc`, `zproc` Process grid configurations. For each test run three numbers are taken: one from `xproc`, px, one from `yproc`, py and one from `zproc`, pz. These numbers specify the number of processes in each of three axes of the logical 3D grid of processes. The user must ensure that $px * py * pz$ is the total number of processes.

`collective:` This switch can be set to the collective or non-collective forms of MPI-I/O.

The MPI-hint parameters especially relevant for this test are the same as the ones listed under the `matrix2D` test.

6.3 Non-Sequential I/O: `nonseq`

The `nonseq` test is to be based on the I/O patterns found in database searching applications. The essential point is that these require non-sequential access to the file, need many seek operations and therefore test rather different I/O aspects from the low-level class of tests.

Database applications appear in the list of SIO selected programs in the Earth Sciences category under Land cover dynamics and Data Analysis and knowledge discovery.

Real database applications sometimes operate in a page mode and some of the pages must be updated. It is difficult to strike a balance between a real application which will have its individual profile for the proportion of pages updated and a generic kernel test that should not become too specialised in attempting to model a particular application.

We shall opt for an approach which is based on a simplified and general algorithm, but allowing sufficient parameters so that with appropriate choices the pattern of behaviour of a real application may be approached. The kernel `nonseq`, locates and reads blocksize pieces of data from a large file in deterministic pseudo-random order. The data is modified and replaced in some fixed fraction of the cases.

The timing measurement will be of each seek and blocksize read or written. The test will therefore generate considerable data, as in the case of the low-level tests.

```
nonseq(filesize, blocksize, nprocs, update_frac)
```

`filesize:` The filesize should be large compatible with the system capabilities – say 1GB, but there is no need to rerun the test for more than one value of this parameter.

`blocksize:` Choosing the pagesize of the machine would be an appropriate choice. In contrast to the low-level class, this parameter need not become very large.

`nprocs:` The number of processors.

`update_frac:` This floating point fraction (value 0 to 1) specifies the fraction of blocks read that are to be modified and rewritten to the file. This parameter must be varied as its effect is likely to be highly nonlinear.

It is unlikely that this application will benefit from the optimisations of collective buffering or special multidimensional array handling.

A series of tests with different parameters can be conducted using the additional keywords as explained in the appendix.

6.4 Gather/Scatter and I/O: `gatherscat2D`

For many reasons, such as postprocessing requirements and the physical I/O capabilities of the machine, a non-parallel form of output is often employed at present. This amounts to a gather MPI call, followed by a write from that processor. In the read cycle, data is first read onto a single processor and then distributed around the machine using the scatter instruction.

The kernel test based on this form of I/O will be used to compare rates with fully parallel output using the `matrix` tests. This is interesting, especially in the context of the IFS weather code. because it allows kernel level comparison of old-fashioned serial and new parallel I/O. Whether this test is useful or not in facilitating this kind of comparison will depend on the parallel filesystem in use.

This kernel also allows the interplay of message traffic and I/O to be investigated and will measure the efficiency of the network as well as the I/O performance of the single node.

Presently we only specify a 2D version, but a 3D version may be useful in future to make comparisons with the results of `matrix3D`. The parameters of this test follow closely those of the matrix test, and this should aid direct comparison.

```
gatherscat2D(problemsize, xproc, yproc)
```

`problemsize` For problem size, N, the square matrix is of size $N \times N$ elements. The actual file size is N^2 times the size of the `MPI_double_precision` data structure.

`xproc, yproc` Process grid configurations. For each test run a pair of numbers are taken: one from `xproc`, px, and one from `yproc`, py. This pair of numbers specifies the number of processes in a row, px (or column, py), of the logical grid of processes. The user must ensure that $px * py =$ total number of processes.

The effect of the optimisations of collective buffering and multidimensional array handling are not likely to be as pronounced in this test as for the full matrix tests. Nevertheless their effect should be investigated.

6.5 I/O with a Shared Filepointer: `sharedfp`

The shared file pointer of MPI is useful when all processors need to write random quantities of data to a file in arbitrary order. This is typically the case when writing a log file and can also be relevant when checkpointing.

Checkpointing was identified in many of the applications considered by SIO as an important I/O bottleneck. This is also the case in the IFS application.

The test is a modification of the `multiple` test that goes through a write loop on each process. The filepointer is shared and the quantity of data written in each call to the write routine is independently randomly selected from a uniform distribution of ranging between `blockmin` and `blockmax`. This test will also be interesting from the point of view of instrumentation and checking the load balance. The test will only perform write operations since the way that checkpointing data is read can require some special data structure and depend on the precise application. The `access_type` hint should therefore be used.

The measurements will give the time for the write instruction and the amount of data written. An analysis tool is necessary to manage this information. The operations performed by this test may benefit from collective buffering.

```
sharedfp(filesize, blockmin, blockmax, nprocs)
```

`filesize:` This test should be run with one large filesize. The test will complete when the next write would exceed this filesize.

`blockmin, blockmax:` Range of the uniform distribution used to select the blocksizes. The parameter `blockmin` will often be set to zero.

`nprocs:` The number of processors.

A series of tests can be conducted as explained in the appendix.

6.6 Transposed I/O: `transpose`

FFT's present a very important class of computation and they often require permuted data array structures. One type of operation necessary is the transpose since multidimensional FFT's need both row and column ordered data. For example, distributed array transpose operations are used in the seismic data processing application considered by the SIO in order to change the state of the I/O and data decompositions. Also, the 3-D Navier Stokes Turbulence simulation software considered by the SIO uses a pseudo-spectral method. Flow gradients are computed using a Fourier transform and in order to transform the entire 3-D array, a transpose must be performed.

Although the transpose is usually performed by communicating subarrays through the interconnect, there is some interest in performing the operation by writing and reading from disk. For example, in a seismic application, this might be because FFT's of a whole data set must be performed before proceeding to the next stage of the computation.

The kernel test `transpose` first writes a distributed 2D array to a file then reads it back as a transpose. The array distribution is as used in the `matrix` tests and the same parameters are used to describe the array as in that case. The data generated is the time for writing and transposed reading along the lines of the measurements taken in the `matrix2D` test.

An interesting comparison can be made in this test. The time required for the transpose using the method described above can be compared with the time taken to perform the operation using message passing. We therefore also measure this quantity which aids self checking.

`transpose(problemsize, nprocs)`

`problemsize` For problem size, N, the 2D array is of size $N \times N$ elements. The actual file size is N^2 times the size of the `MPI_double_precision` data structure.

`nprocs` The process grid configuration is determined by the number of processes. With the same system as for other matrix tests, `yproc` = 1, `xproc` = `nprocs`.

`collective:` This switch can be set to the collective or non-collective forms of MPI I/O.

Any effect of the MPI hint `access_style` can be checked in this context of this test.

6.7 Other Kernel Tests

In the introduction to the Kernel test class several classes of important applications with apparently similar I/O requirements were identified. The precise synthetic pattern to choose for a kernel test had not however been decided. We discuss two additional kernel tests below, but postpone details until a future version of the suite.

6.7.1 Out-of-core

"Out-of-core solvers" form an important class of applications with significant I/O requirements. Although increases in main memory size often allow problems that used to require out-of-core solutions to be done directly; new problems arise with even greater memory requirements. It should be recognised however, that out-of-core solvers are by no means a homogeneous class and the range of different problems requiring out-of-core solutions is wide. For example, the I/O pattern for Hartree Fock is distinct from that needed for large matrix operations.

It is not apparent that there is any synthetic kernel that can capture the characteristics for all out-of-core problems. At present, our inclination is to select a simple mathematical operation and perform it out-of-core. A natural choice is one of the matrix operations. The best algorithms to use for out-of-core matrix operations are certainly not the same as would be used in a direct solution. The best algorithm must be chosen for this test if it is to be effective, and to be confident of this aspect requires further work.

At present the LU decomposition problem has been identified as a good choice to implement as a kernel test. One advantage is that a public code is available for this problem, which would have to be assessed for algorithmic effectiveness and efficiency.

6.7.2 Irregular mesh

When the arrays generated by applications that simulate physical systems are dense and regular there is no difficulty in devising kernel tests such as the `matrix` tests. However, in an important class of applications with significant I/O needs, the arrays are sparse or may be based on irregular mesh structures. Unfortunately the organisation of mesh data structures is not standard, neither is the kind or sparse array standard. The labeling of the elements is important and can be managed in a variety of different ways usually using pointers.

To devise an effective kernel test in these circumstances requires further knowledge of a wide class of these applications. To proceed, common feature of the I/O patterns must be identified. It appears that I/O considerations may be restricting the kinds of algorithm employed in current applications.

7 Analysis and Usage

The tests generate raw timing data with very clear and precise meaning. To interpret this data so as to address the problems listed in section 1, a separate analysis stage is needed. The range of analysis tools available is designed to throughly test the system and is thus more complicated than would be found in a simple benchmark suite. For this reason, some guidance is useful for organising a suitable series of tests and for using the analysis tools.

In the case of the low-level class, the analysis consists of several stages starting with an exploration of the raw data. No assumptions about the way the data are expected to behave are imposed and only by looking at the raw data can one test whether a particular assumption is valid. Provided the data do look reasonable, one can proceed to more sophisticated levels of analysis. A suggested series of analysis steps is outlined in section 7.2. In future releases of the test suite we expect that this may form part of the "Users Guide".

7.1 Analysis Tools

The analysis should be done with a graphic tool that displays graphs and calculates various averages. At the present stage the tool and documentation are not well developed and a series of awk scripts that provide similar functionality by processing the output files to generate gnuplot graphs are provided as a temporary measure.

This analysis of the timing data is supplemented by instrumentation, which addresses different aspects of the problem.

7.1.1 Low-Level Tools

The tools provided for the low-level class of blocking tests are briefly described here, the way they may be used is noted in section 7.2.

`rawdata` This tool makes a plot of the raw timing data versus the loop counter. An example is shown in figure 1. A more advanced version of the test will enable zooming and hence allow restricted regions of the loop counter to be selected and averages to be taken within these regions. The multiple version of this tool will have several horizontal axes each displaying the timing of a separate process.

`distribution` A plot of the distribution of the timings can be generated using this tool. It will be possible to select a range of loop counter in order to construct the distribution. The multiple version of this tool will generate separate curves for each process.

`allav` Curves describing how the I/O rate varies with filesize and blocksize are generated by this tool. The precise way in which averages are to be taken must be determined using the previous tools. Error bars are also plotted.

7.2 Analysis of Low-Level Tests

As explained in the section specifying the low-level class of tests, these tests generate more than than just a single number per test. In this section we give an example of the kind of data gathered by a `single` test, and show how the analysis tools can be used to measure interesting underlying parameters that characterise the machine.

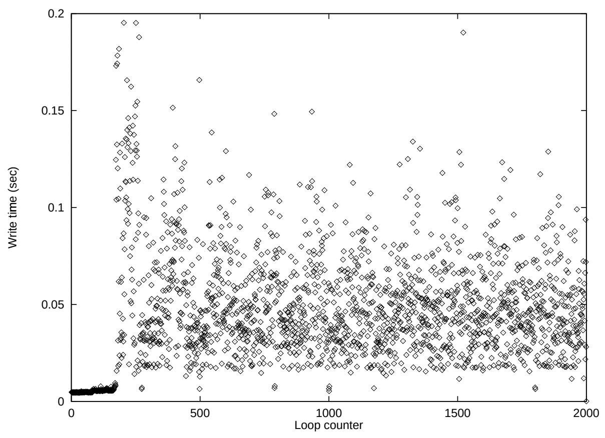

Figure 1: Example of timings for a loop of the type described in the text. Blocksize is 100kB, and a 200MB file is written on a dedicated IBM rs6000 workstation. A startup period and data scatter with some structure are visible.

To remind the reader of the precise meaning of the timing data, consider the following pseudo-code for the heart of a simple write test which measures the times for successive writes of a fixed size buffer.

```
do $j = 1$, niter
   call MPI_Write()
   tnew = MPI_Wtime()
   wtime($j$) = tnew - told      Store successive write times
   told = tnew
enddo
```

The significance of the time differences is manifest. The timings are stored in the array `wtime` and are output at the end of the program to form the raw data produced by the test. To obtain the time that would be measured by timing instructions placed outside the loop, all the individual timings in the array could be summed, and this would constitute an average. The quantity and potential complexity of data gathered in this way is what allows for different levels of analysis.

An overall picture of what is happening may be obtained from plotting the successive times against the loop counter (which measures the extent of the file as it grows). A typical example obtained for a similar program on a workstation is shown in figure 1. This type of figure is generated by the `rawdata` analysis tool. This picture is complicated because of the fluctuations in successive timings, and also because of initial effects such as filling up a cache.

We observe the following points, with their interpretation.

- The first ~ 200 writes occur quickly and correspond to a cache (or if one looks carefully, maybe two caches), being filled up. There is little fluctuation between successive timing measurements in this regime.

- The main part of the plot, from say ~ 300 on, is significantly slower than the initial part, and displays strong fluctuations in timing between successive measurements. This can be

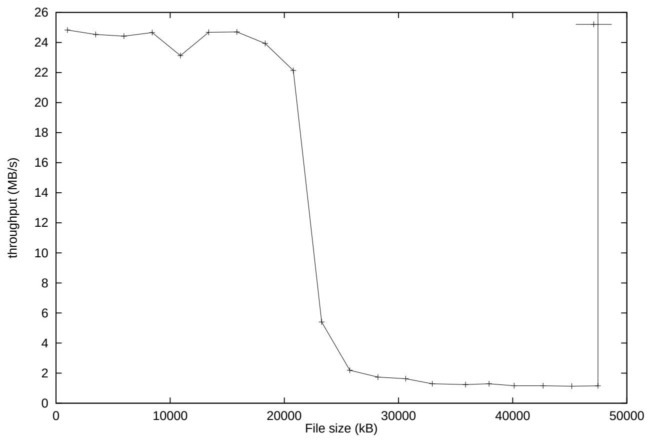

Figure 2: Example of performance curve for a dedicated IBM rs6000 workstation. The plateaus characterise the regions dominated by cache and disk I/O respectively.

interpreted as an effect arising from the cache being full. The CPU must wait until transfers are made out of the cache, and eventually to the disc. The frequency and the blocksize in which this transfer is made depends on the operating system (and possibly also the state of readiness of the disc), and is incommensurate with the rate and blocksize with which the CPU is attempting to fill the cache. It is not easy to develop a model to describe the detailed shape of the fluctuations, especially when the cache has several levels. It may however be possible to construct a model that describes some statistical properties of the data, such as average and maybe also correlations. A useful tool (`distribution`) to analyse the scatter is the distribution of times (usually after removal of the startup period), and this may show some structure which could provide clues about how the I/O is performed.

- There also appears to be a short intermediate period between the two regimes described above, with even longer timings.

Once the degree of data scatter is appreciated, the significance of various averages can be judged. Indeed, we emphasise that unless this kind of observation is made, and the effect of any startup and the origin of any scatter understood, there is little sense in taking averages. For example one can check that the machine is as dedicated as expected. For the `multiple` test, running on several processors, several parallel sets of data from each processor must be analysed together and this requires further analysis tools.

Provided these conditions above are met, and the data is reasonably clean, one may calculate an averaged asymptotic rate after removing any initial period that corresponds to transient startup effects. This bandwidth should be independent of further increases in the file size and is thus a good reference. Any averages over smaller files (shorter writes), and any averages that include the initial period should be compared with this reference bandwidth.

All the above discussion assumes a fixed buffer size, and all tests must be repeated when this parameter is altered. Performance curves of the kind discussed by Chen and Patterson [14] are then constructed using averages constructed subject to all the above caveats. One curve will display the variation of I/O rate with filesize at some fixed blocksize. An approximate version of this curve can be constructed using averages obtained from reduced regions of the raw data for a large filesize. This curve will usually allow the plateau ranges dominated by cache and disk I/O respectively to be identified. A display of this type is shown in figure 2 for the simple example of an IBM workstation. The plateaus are very clear, and indeed the crossover region that corresponds to the effective cache size was apparent from the raw data. More complicated behavior sometimes occurs when there are hierarchies of cache, for examples see [14].

Further characteristic curves show the how the rate varies with blocksize at fixed filesize. These curves will reveal fundamental I/O behaviour. Again they will contain error estimates. The changes in the multiple version of these curves in the case of collective buffering with different buffering parameters will help to identify the best way to use this optimisation. Similarly, the effect of varying the striping parameters should also be viewed at this level.

Although in the performance graphs we transform to rates measured in MBytes/s, the raw timing data is useful in its original form. It often happens that for small buffer sizes the time of each write call is almost independent of the amount of data written. To investigate such issues the raw times are most appropriate.

7.3 Analysis of Kernel Tests

In this section we only consider analysis of the `matrix2D` test. The `matrix2D` test produces a single timing measurement on each process for each problemsize and distribution (parameters px and py). The low-level class of tests have already isolated startup effects which are incorporated in these measurements. As specified earlier, we expect these tests to be run for some fixed problemsizes that correspond to the plateau regions of the low-level performance curves, usually characterised by cache and disk dominated. The parameter space that we wish to investigate describes the distribution of the array and the I/O optimisations and is a large space. The individual parameters will have effects that are highly correlated. For example the values of collective buffer size, chunk size and striping size should be related.

At the present moment, and without detailed knowledge of the MPI implementation, it is difficult to give guidelines for how this space should be most efficiently explored. With experience we will be able to predict the performance profile characteristics to expect from different types of hardware and software. It is not even clear how best to go about tuning such a large number of parameters to obtain optimum performance. Sometimes guidelines may be apparent from the documentation that comes with the MPI implementation. The ultimate aim of tuning for optimum performance would be to give some preliminary idea of the relative efficiencies of different systems for a characteristic I/O pattern. Such crude tests are not however sufficient to distinguish the origin of the efficiency and a full comparison of curves is necessary.

In the above we have discussed the advanced features of the MPI implementation related to optimisations, but there are other MPI features that we would hope to test. One such is the efficiency of the fileview feature.

Acknowledgments

I would like to thank, Panos Melas, Tim Oliver, Dave Snelling and Ed Zaluska.

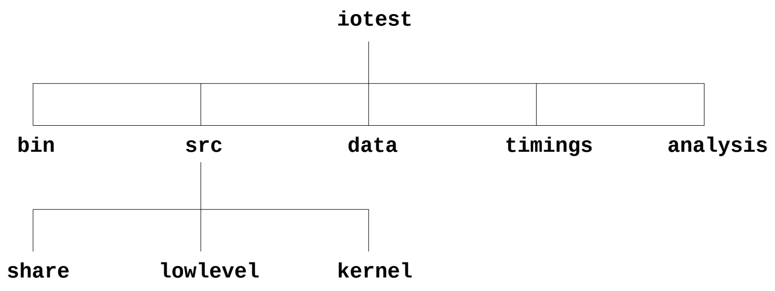

Figure 3: Directory structure of the initial distribution.

A Test Suite Organisation

When the distribution is unpacked it generates a directory tree as shown in figure 3. Instructions for setting environment variables, setting the location of the MPI-2 library etc. are given in a README file to be found in the root directory of the tree. Compilation uses `make` rather than `gmake`. In order to speed up and simplify the running of the tests, it will be possible to compile separately any of the classes of test instead of having to compile everything. As explained in appendix B, it is possible to run a whole series of tests with different parameter values in one go. To enable this feature, a hierarchical system of wrappers is used to run the tests. In brief, the executable `iot_all` (or for example `iot_low-level` if not all classes have been compiled) calls different tests through the wrapper routine `iot_wrap_<test>`. This routine loops through all the parameter values required for the test and calls the final routine `iot_run_<test>` which actually runs the test using the parameter values that it has been passed.

B Keywords

B.1 File for Inputting Parameters

Because the file hints defined in the MPI standard are necessary parameters, it is a sensible policy to use a file format that identifies *all* parameters with keywords. The pattern is as follows:

```
keyword value
```

Where `keyword` is one of the keywords defined in this document. The keywords include the reserved MPI file-hint keywords, but each test may require additional keywords to define test parameters. The `value` may for example be a string, floating point number or set of comma spaced integers depending on the keyword. The allowed values associated with each keyword are defined in this appendix. Once the parameters are read into the test program, they are stored in an MPI-Info object as defined in the MPI-2 standard.

Not all important parameters can be specified in this way, the most notable example is the number of processors that the test runs on. In general this must be specified when the test is submitted, and the way the submission procedure operates will be strongly site dependent. In fact this example does not cause any difficulty because the information of how many processors are running is available to the test via MPI routines. More worrying are other parameters that are fixed at submission time, for example the amount of memory can be requested in the qsub submission method. This information is important, but it is not clear how it can be retained for the output. One possibility would be to request such parameters as dummy values in the `iotparams.in` file.

B.2 Reserved Keywords, MPI Hints

Hints allow a user to provide information regarding file access patterns and file system specifics to direct optimization. Providing hints may enable an implementation to deliver increased I/O performance or minimize the use of system resources. However, hints do not change the semantics of any of the interfaces. In other words, an implementation is free to ignore all hints.

Some potentially useful hints (info key values) are outlined below. These are taken directly from the I/O chapter of the MPI-2 standard. The following key values are reserved for all info arguments. An implementation is not required to interpret these key values, but if it does interpret the key value, it must provide the functionality described.

These hints are mainly concerned with layout of data on parallel I/O devices, and with access patterns. For each hint name introduced, we describe the purpose of the hint, and the type of the hint value. Some hints must take the same value on all participating processes, this is automatically handled in the contest of the test suite.

`access_style` (list of comma separated strings) This hint specifies the manner in which the file will be accessed until the file is closed. The hint value is a comma separated list of the following: read_once, write_once, read_mostly, write_mostly, sequential, reverse_sequential, and random.

`collective_buffering` (boolean) This hint specifies whether the application will benefit from collective buffering (an optimization performed on collective accesses which coalesces small requests into large disk accesses). Legal values for this key are true and false. Collective buffering parameters are further directed via additional hints.

`cb_block_size` (integer) This hint specifies the block size used for collective buffering file access. Target nodes access data in chunks of this size. The chunks are distributed to target nodes in a round-robin (CYCLIC) pattern.

`cb_buffer_size` (integer) This hint specifies the total buffer space used for collective buffering on each target node; usually a multiple of cb_block_size.

`cb_nodes` (integer) This hint specifies the number of target nodes used for collective buffering.

`chunked` (comma separated list of integers) SAME: This hint specifies that the file consists of a multidimensional array that is often accessed by subarrays. The value for this hint is a comma separated list of array dimensions, starting from the most significant one (for an array stored in row-major order, as in C, the most significant dimension is the first one; for an array stored in column-major order, as in Fortran, the most significant dimension is the last one, and array dimensions should be reversed).

`chunked_item` (comma separated list of integers) : This hint specifies the size of each array entry, in bytes.

`chunked_size` (comma separated list of integers) : This hint specifies the dimensions of the subarrays. This is a comma separated list of array dimensions, starting from the most significant one.

`filename` (string) This hint specifies the file name used when the file was opened. If the implementation is capable of returning the file name of an open file, it will be returned using this key by `MPI_file_get_info`. This key is ignored when passed to `MPI_open`, `MPI_file_set_view`, and `MPI_file_set_info`.

`file_perm` (string) This hint specifies the file permissions to use for file creation. This hint is only useful when passed to `MPI_open` when amode includes `MPI_create`. The value with this key is implementation dependent.

`io_node_list` (list of comma separated strings) This hint specifies the list of I/O devices that should be used to store a file.

`nb_proc` (integer) This hint specifies the number of parallel processes that will typically be assigned to run programs that access this file. This hint is most relevant when the file is created.

`num_io_nodes` (integer) This hint specifies the number of I/O devices in the system. Used to specify the ideal number of I/O devices for this application.

`striping_factor` (integer) This hint specifies the number of I/O devices that the file should be striped across, and is relevant only when the file is created.

`striping_unit` (integer) This hint specifies the suggested striping unit to be used for this file. The striping unit is the amount of consecutive data taken from one I/O device before progressing to the next device, when striping across a number of devices; it is expressed in bytes. This hint is relevant only when the file is created.

At a practical level, we do not anticipate that all of these keywords will have an effect in the first generations of MPI implementations. Other keywords that control important optimisations are likely to be implemented. In the text we have indicated which optimisations should be employed in a given test.

The keyword `filename` must appear in the `iotparams.in` file and its value is discussed at length in the environment idiom.

B.3 Test Keywords

Each test in the suite requires its own specific set of parameters. These are read from the `iotparams.in` file in exactly the same way as the hints information above. The keywords for each test and parameter are defined below.

Two special keywords always appear irrespective of the test, and they must appear (in order) at the beginning of a set of keyword-value pairs defining the parameter for a particular test. These keywords are:

`class` (string) Where the string can be either, “Lowlevel” or “Kernel”. This parameter defines the class of the test.

`testname` (string) The string must be one of the names of the tests listed in the specification, and also listed below with the required keywords and possible associated values.

The keyword `filename` is actually one of the MPI-hints and is always needed to specify the location of the test file. Another parameter that is always available, though not required, specifies the file for writing timing measurements.

`timingsfilename` (string) The default value is “all.out”, and it appears in the `/timings` directory of the distribution.

B.4 Low-Level Tests, Keywords

These basic tests write a single file in blocks. The important parameters are the filesize and blocksize. To facilitate repeated tests over a range of parameters, we use a similar format to SCALAPACK.

Input Keywords

The standard test parameters, `classname`, `testname`, and `filename` are required parameters for all the low-level tests. The standard low-level input keywords are:

`numruns` (integer) This specifies the number of filesize, blocksize pairs to test. If only one pair is to be used in the test, this value must be set to 1.

`filesize` (space separated floating point values) The filesize values are given in MB (1 Megabyte is one million bytes rather than 2^{20} bytes).

`blocksize` (space separated floating point values) The blocksize values are given in Megabytes (1MB is one million bytes).

Output Keywords

The number of nodes that the test runs on is not given in the `iotparams.in` file, but is deduced while running and printed in the output file against the keyword `nprocs`.

Other output keywords that appear in all low-level tests are the ones needed to describe the timing measurements. The main timing parameters appearing in the central loop have keywords specifying whether they time write or read operations and since these appear many times they are given short keywords. Additional timing parameters that relate other operations such as the time needed to open files and set up filetypes before the first write operation.

These standard low-level output keywords are as follows.

`pre_time` The time taken to open the test data file and set up filetypes.

`palloc_time` The time taken to preallocate the test data file using the MPI instruction `MPI_file_preallocate` as discussed in the idioms section.

`w` The time taken to write a block in the loop.

`r` The time taken to read a block in the loop.

`post_time` The time taken to close the test data file.

B.4.1 `single`

Input

The standard keywords: `class, testname` and `filename` must be specified. The keywords; `numruns, filesize`, and `blocksize`, which are the standard low-level input keywords are also required.

Output

On output, all the standard low-level output keywords are used.

B.4.2 `multiple`

Input

The standard keywords: `class, testname` and `filename` must be specified. The stanndard low-level input keywords are also needed. In addition, one other keyword is required:

`collective` (string) The string may take values: "true" or "false". The meaning is clear from the table of the different kinds of MPI read/write displayed in the standard.

Optional MPI-hint keywords associated with collectivity and striping parameters will be relevant.

Output

On output, the standard low-level output keywords are used.

B.4.3 `singleI`

Input

Several additional parameters beyond those needed in the blocking version are necessary.

`tasklength` This stands for a set of parameters that contol the matrix-vector computation. In particular, the duration of the computational task is determined by these parameters. In preliminary use this parameter should be choosen to check results against the blocking version.

Output

On output, several new keywords are needed in order to measure the times in the initial cycles as well as in the non-blocking cycles. In each case both I/O routine and task duration are measured.

`t` The time taken for the task instruction in the loop without any I/O.

`wI` The time taken for the write instruction in the loop during a non-blocking phase of the test.

`rI` The time taken to read in the loop during a non-blocking phase of the test.

`wtI` The time taken for the task instruction in the loop during a non-blocking write phase of the test.

`rtI` The time taken for the task instruction in the loop during a non-blocking read phase of the test.

B.4.4 `multipleI`

Input

The input keywords controlling `tasklength` are required besides all the standard low-level input keywords.

Output

The output keywords are the same as the ones defined above for `singleI`.

B.5 Kernel Tests, Keywords

The kernel tests are more varied than the low-level tests and there are no common keywords for the class other than those inherited from all tests.

The MPI hint parameters are optional with system dependent defaults.

B.5.1 `matrix2D`

Input

The test is described in detail in the main text, and requires all the following parameters.

`numsizes` (integer) The number of different problemsizes to test. The test is run for each of the `numsizes` problem sizes specified by the following parameters.

`xsize, ysize` (comma separated integers) A list of `numproblemsizes` test problem sizes. For each problem size the matrix is of size `xsize`$\times$`ysize` elements.

`numprocgrids` (integer) The number of different logical process grid configurations to test. The test is run for each of the `numprocgrids` process grid configurations specified by the `xproc` and `yproc` parameters.

`xproc`, `yproc` (integer) Two lines, each with a list of `numprocgrids` process grid configurations. For each test run a pair of numbers are taken: one from `xproc`, px, and one from `yproc`, py. This pair of numbers specifies the number of processes in a row, px (or column, py), of the logical grid of processes. The user must ensure that $px * py$ is the total number of processors.

`collective` (string) The string may take values: “true” or “false”. For testing the optimisations this must be set to “true”.

Output

Some of the output keywords needed to describe the timing measurements are the same as appear in the low-level tests. The main timing measurements have different keywords from the low-level case because there is no explicit loop and they correspond to the times for the complete I/O.

For each test run the following data is output:

`pre_time` The time taken to open the test data file and set up filetypes.

`palloc_time` The time taken to preallocate the test data file using the MPI instruction `MPI_file_preallocate` as discussed in the idioms section.

`write` The time taken to write the matrix to the test data file.

`read` The time taken to read the matrix from the test data file.

`post_time` The time taken to close the test data file.

The total summed time should relate to the time taken for the test to run, which in some sites will be reported separately.

B.5.2 `matrix3D`

Input

This test is very similar to the previous one, it only requires two further parameters, `zsize` and `zproc`, that appear in the following way.

`xsize, ysize, zsize` (comma separated integers) A list of `numproblemsizes` test problem sizes. For each problem size the matrix is of size `xsize`$\times$`ysize`$\times$`zsize` elements.

`xproc`, `yproc`, `zproc` (integer) For each test run three numbers are taken: one from `xproc`, px, one from `yproc`, py, and one from `zproc`, pz. These specify the way the array is distributed on the processes as a 3D grid $px \times py \times pz$ The user must ensure that $px * py * pz$ is the total number of processors.

Output

The output parameters are exactly as for the `matrix2D` test.

B.5.3 `nonseq`

Input

This test requires the following keywords.

`filesize:` (floating point) The filesize should be large compatible with the system capabilities – say 1GB, but there is no need to rerun the test for more than one value of this parameter.

`numblocksize` (integer) This specifies the number of blocksizes to test (starting with the first) from the following list given by the blocksize keyword. This range of blocksizes is tested for every value of the filesize given in the list above

`blocksize` (comma separated floating point values) The blocksize values are given in Megabytes (1MB is one million bytes). Suggest the pagesize. In contrast to the low-level class, this parameter need not become very large.

`numupdate` (integer) This specifies the number of update fractions given in the next list to test (starting with the first).

`update_frac:` (floating point) This floating point fraction (value 0 to 1) specifies the fraction of blocks read that are modified and rewritten to the file. This parameter must be varied as its effect is likely to be very nonlinear.

Output

The output keywords to describe the timing measurements will include the usual set. An additional keyword is needed to distinguish the timing of the reads and the fraction of writes since these operations will be interleaved. The standard keywords "`r`" is used and "`u`" denotes an update.

It is unlikely that this application will benefit from the optimisations of collective buffering or special multidimensional array handling.

B.5.4 `gatherscat2D`

Input

The test parameters are the same as those used in `matrix2D`, and have the same meanings.

`numsizes` (integer)

`xsize`, `ysize` (integers)

`numprocgrids` (integer)

`xproc`, `yproc` (integers)

Output

The output is from one process only, but the output keywords are the same as for the matrix test. One additional timing parameter is needed to specify the time taken to gather/scatter the data. The keywords are `gather` and `scatter`.

The measurements will give the time for the write instruction and the amount of data written. An analysis tool will be necessary to manage this information.

B.5.5 `sharedfp`

Input

Some of these parameters are similar to the ones used in the `multiple` test. The blocksize is not however constant on each write.

`filesize` (floating point) This test should be run with one large filesize. The test will complete when the next write would exceed this filesize.

`numblocksize` (integer) This specifies the number of ranges to test (starting with the first) from the following list.

`blockmin, blockmax` (comma separated floating point) Range of the uniform distribution used to select the blocksizes.

The operations performed by this test may benefit from collective buffering.

Output

Output from this test includes the blocksize actually written besides the usual timing value. The keywords `b` and `w`, are chosen to be short since they will be output often.

`w` Time taken for write operation.

`b` Quantity of data written, measured in bytes as an integer.

B.5.6 `transpose`

Input

The parameters required for this test are the same as those needed for the `matrix2D` test. The matrix is divided according to `ysize` = 1, `xsize` = `nprocs`.

`xsize, ysize` (integer) The 2D array is of size `xsize`×`ysize` elements.

The effects of the MPI hint `access_style` can be checked in this context of this test.

Output

The output from this test uses exactly the same keywords as the matrix tests. One additional keyword is used to describe the time taken to perform the transpose using message passing.

`internal` This measurement provides a comparison to the value of `read` which may be useful.

C Input File Example

An example of an input file which performs two tests, one after the other, is given below. This template also appears in the `iotest/timings` directory of the distribution. All lines starting with a # are comments.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
# Template for input file
# IOT: MPI IO Test Suite Release 1.0 (11/8/97)
# Copyright Fujitsu Ltd.  1997
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#
# Timingfilename only appears once in an input file.
#
timingsfilename /home/djl/SUITE/iotest/timings/lowlevel.out
#
#
# Classname and testname must come at the beginning of each test
section.
#
# Class name
classname Lowlevel
# Test name
testname single
#
filename /L/v/djl/test/lowleveltest.tst
#
# Options valid for this test
#
# Number of filesizes to read from list
numfilesize 3
# List of filesizes (in MB)
```

```
filesize 0.01 0.1 1 10 100
# Number of blocksizes to read from list
numblocksize 2
# List of blocksizes (in MB)
blocksize 0.01 0.02 0.04
#
############################
# Classname and testname for the second set of tests
#
classname Lowlevel
testname multiple
#
# Some MPI file hints (with reserved keywords)
#
filename /L/v/djl/test/lowleveltest.tst
#
collective_buffering true
#
cb_buffer_size 32
#
num_io_nodes 1
#
# Options valid for this test
#
# Number of filesizes to read from list
numfilesize 2
# List of filesizes (in MB)
filesize 0.5 5 50 500
# Number of blocksizes to read from list
numblocksize 1
# List of blocksizes (in MB)
blocksize 0.1
#
collective true
```

D Future Directions

Future versions of the suite will include more tests in the kernel class. The analysis tools will be made more sophisticated.

Experience with the use of the tests on a variety of platforms will enable better guides as to appropriate choices of parameters to be made.

References

[1] The MPI-2 standard is available at: `http://www.mpi-forum.org/`.

[2] VAMPIR and documentation on the product are available from PALLAS at: `http://www.pallas.de/`.

[3] R. Hockney and M. Berry (Eds.). *Public International Benchmarks for Parallel Computers*, Parkbench Committee Report No. 1, Scientific Programming, 3, pp. 101-146, 1994.

[4] Private comments by Charles Grassi (13/5/97).

[5] M. Metcalf and J. Reid, *The F Programming Language*, Oxford Science Publications, OUP 1996.

[6] The list of Compact Applications considered by the Scalable I/O Consortium at Urbana.

`http://www.cacr.caltech.edu/SIO`

`http://www.llnl.gov/liv_comp/siof.html`

[7] D. Lancaster and E.J. Zaluska, *Survey of Parallel Input/Output Testing and Benchmarking*, Report prepared for Fujitsu European Centre for Information Technology, August 1997.

[8] R. Carter, R. Ciotti, S. Fineberg and W. Nitzberg, *NHT-1 I/O Benchmarks*, RND Technical Report RND-92-016.

`http://parallel.nas.nasa.gov/MPI-IO/btio/btio-download.html`

[9] Nils Nieuwejaar and David Kotz. *Low-level interfaces for high-level parallel I/O*, In Proceedings of the IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems, pages 47-62, (1995).

[10] James T. Poole, *Preliminary survey of I/O intensive applications*, Technical Report CCSF-38, Scalable I/O Initiative, Caltech Concurrent Supercomputing Facilities, Caltech, (1994).

[11] K. E. Seamons, Y. Chen, M. Winslett, Y. Cho, S. Kuo, P. Jones, J. Jozwiak, and M. Subramanian. *Fast and easy I/O for arrays in large-scale applications*, At SPDP'95, October 1995.

[12] D. Lancaster, *Report on IFS I/O Patterns*, Report prepared for Fujitsu European Centre for Information Technology, August 1997.

[13] 11th RAPS Workshop. GMD/SCAI St. Augustin. 12-13 May 1997.

[14] 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.