User Guide for 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)

Technical Report ****

May 4, 1999

Contents

1 Introduction **3**

2 Installing and Running the Test Suite **4**
2.1 Distribution and Installation 4
2.2 Running 5

3 General Issues **6**
3.1 Conventions 6
3.2 MPI-I/O and Filesystems 7
3.3 File Formats 7

4 Test Parameters and Output: the Keyword System **8**
4.1 Input Parameter Keywords 8
4.1.1 Standard Input Keywords 8
4.1.2 Reserved Keywords, MPI Hints 9
4.1.3 Input File Format 10
4.2 Output keywords 10
4.2.1 Output File Format 12

5 Low-Level Class **13**
5.1 Low-Level Tests, Keywords 13
5.2 Single Test: `single` 15
5.3 Multiple Test: `multiple` 18
5.4 Asynchronous Tests 20
5.4.1 `singleI` 21
5.4.2 `multipleI` 23
5.5 Fortran I/O Test: `sinunix` 25

6 Kernel Class **26**
6.1 Characteristic I/O Patterns 26
6.2 I/O of regular multidimensional arrays 27
6.2.1 2D matrix I/O: `matrix2D` 29
6.2.2 3D matrix I/O: `matrix3D` 32
6.3 Gather/Scatter and I/O: `gatherscat2D` 34
6.4 Transposed I/O: `transpose` 36
6.5 Non-Sequential I/O: `nonseq` 39
6.6 I/O with a Shared Filepointer: `sharedfp` 42

7 Analysis **44**
7.1 Analysis of Low-Level Tests 44
7.1.1 Tools for raw data 44
7.1.2 Tools for data distributions 46
7.1.3 Tools for performance curves 47
7.2 Analysis of Kernel Tests 48
7.2.1 matrix tests 49
7.2.2 `nonseq` test 49

7.2.3 `sharedfp` test 50

A Reserved Keywords, MPI Hints **52**

B NQS Script Example **54**

C Input File Example **54**

D Output File Example **55**

References **58**

1 Introduction

The Parallel I/O Test Suite comprises a set of tests of I/O performance of parallel filesystems and is a timely tool to investigate new developments in parallel I/O. The tests are written using the MPI-I/O routines standardised in the I/O chapter of the MPI-2 specification [1] and is therefore portable across all platforms which support a parallel filesystem with MPI-I/O interface. Full discussion of the goals and motivations of the test suite along with a description of the overall structure is given in the most recent version of the test suite specification document [2]. Notable features are:

- The tests are simple and the timings have clear significance.
- Data gathering is separated from analysis.
- No model for the behaviour of the data is assumed.
- Curves of data are generated rather than single numbers.
- Tools rather than benchmarks are emphasised.
- The tests are written in F90.
- The tests are not intended to comprise a validation suite for MPI-I/O.

There are many well known tests of I/O performance for serial machines, for example [3], and we have built on experience from this area. For parallel I/O we are only aware of the BTIO test [4], which is a fairly high level test for the characteristic I/O patterns of CFD applications. In this suite, attention has been focussed on tests that address fundamental issues. These are the low-level class of tests and the matrix tests in the kernel class.

This guide gives a full description of the final state of the tests, and concentrates on the way that the tests are to be practically employed. We start with advice on how to install and run the tests in section 2. This is followed by a concise overview of the design choices underlying the suite which are explained at greater length in [2]. To control the tests it is important to understand the system of keywords used to pass parameters to the tests and to report measurement results. This system is explained in section 4. The heart of this User Guide is contained in sections 5 and 6 which describe each test in detail and give precise definitions of the timing measurements. Tools provided with the suite for analysis are described in section 7. The appendices contain examples of an NQS submission script, a test input file and an output file.

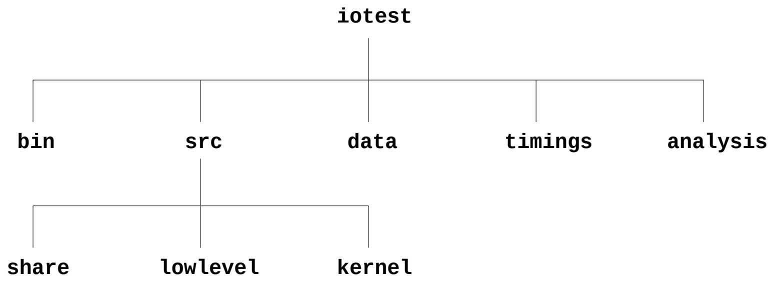

Figure 1: Directory structure of the initial distribution.

2 Installing and Running the Test Suite

2.1 Distribution and Installation

The suite is distributed as a compressed tar file. Unpack the archive into a directory of your choice, `iotest` by typing (on a UNIX machine),

```
uncompress iotest.tar.Z | tar -xf
```

When the distribution is unpacked it generates the directory tree shown in figure 1.

Instructions for installation are given in a README file to be found in the root directory of the tree. The locations of the MPI-2 library and other utilities are set by editing the `iotest/Makevars` file:

MPIR_HOME: path to the root directory of the MPI installation

MPILIB_DIRS: path to the MPI library directory

MPIINCLUDE_DIRS: path to the MPI include directory

FC: name of the Fortran90 compiler

The MPI library is the library for full MPI-2 as the suite uses some calls such as `MPI_alltoallw` that are not part of the I/O chapter. The flags for compiler optimisations are also set in this file, and will require local information to complete.

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, so various Makefile targets are possible:

`all`: make all the executable test programs (this is the default)

`iot_all`: make executable test program named `iot_all`

`iot_lowlevel`: make executable test program named `iot_lowlevel`

`iot_kernel`: make executable test program named `iot_kernel`

`install`: copy executables to `bin` directory

`clean`: remove all object files and other intermediate files

`veryclean`: perform clean and remove executables from `src`

Working in the iotest directory, compile the tests required (*eg* make iot_all). Installation is completed with the command: make install, which places the executables in the iotest/bin directory.

2.2 Running

The procedure for running the tests will be machine dependent but we give some guides for the case where jobs are submitted via an NQS queue. Usually the executable test program iot_all is run, and the choice of tests and parameters is made in a short file named iotparams.in. The format for this input file is discussed at length in this guide. In the course of the run the tests generate the I/O files containing random data and these files may be inspected after completion of the test. The output of the tests, consisting of timing and other data is placed in another file, usually in the timings directory.

On machines which operate an NQS queue, the job script, iotest/bin/job, is provided as a template. This should be edited to specify the location of the executable program and the mpirun (or mpiexec) command and if necessary, the cd command should be altered to change to the data directory. The text of this example NQS job script is given in appendix B. To submit an NQS job, work in the iotest/bin directory and type: qsub -lP nprocs job, where nprocs is the number of processors required.

The particular test and parameters are chosen by editing the iotparams.in file in the iotest/data directory. The text of this template is given in appendix C and the format using keywords is discussed at length in section 4. As explained later, 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 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. In the event that not all classes have been compiled, for example if only lowlevel class test are to be run using the executable iot_lowlevel, then the iotparams.in file can only contain information about this class of tests.

Before running the tests, any setup required for the parallel filesystem must be done. For example, the Fujitsu SPFS system needs a configuration file to specify which I/O devices are to be used. The filename of the I/O file is specified in the iotparams.in file.

The test output is usually placed in a file in the iotest/timings directory, and the name of this file is specified in the iotparams.in file. Typical names are all.out or lowlevel.out depending on which executable is run. The output file is standard ascii and contains all timing data generated by the test. The file may be inspected using a standard text viewer. This facility is useful as a preliminary check, but for full analysis automatic tools as described in section 7 should be employed. A small amount of diagnostic information, indicating the progress of the tests, is output to the stdout steam and test errors go to the same place.

3 General Issues

In this section we discuss some of the conventions used in the design of the suite. A full discussion is provided in the specification [2], only a brief summary relating directly to the concerns of the user is provided here. We emphasise practical issues such as the run rules and the file formats. Some of these conventions are expanded upon in greater detail in the appendices.

3.1 Conventions

- Timing: For portability we use the `MPI_Wtime` function that returns a (F90 double precision) time in seconds. The resolution is determined using `MPI_Wticks`. This value is reported and the user should check that adequate resolution is offered.

- Preallocation: Space for files may be preallocated using `MPI_file_preallocate`. The time required to preallocate is measured and reported.

- Keyword system: A keyword system is used both to supply parameters and to output timing measurements from the tests.

- Irregularity: The tests are to be run on a dedicated machine and consist of only a single application. The CPU use on special I/O nodes, if they exist, should be monitored independently.

- Language: The suite is written in F90, and as a matter of style it is restricted to the F[7] subset of F90.

- Errors: 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.

- Data Units: We write and read bits in the form of double precision floating point numbers. The MPI etype is `MPI_double_precision` which in F90 is usually an 8 byte number. The numbers are randomly chosen from a distribution (uniform [0-1]), but subject to the needs of checking.

- Filetype: 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. In the kernel class both these restrictions are relaxed.

- The way in which these tests are run must be thought through carefully to ensure efficiency. The best way to do this is to run some short preliminary tests to choose an appropriate range of `filesizes` and `blocksizes` for more substantial tests.

- Validation: All runs are timestamped. Self checking is accomplished by using the MPI read tests to check earlier MPI writes. The checking is generally statistical rather than of every number written.

3.2 MPI-I/O and Filesystems

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 low-level tests may be run several times with I/O to different filesystems. The user specifies the filesystem by the filename and directory path where the I/O data file is to be placed. There may be other machine dependent parameters that have to be set for a given filesystem. Some of these addition parameters may be set using MPI hints such as striping factors. The use of these hints is allowed and encouraged in the test suite. 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 A. Other parameters required by a filesystem may not be amenable to control in this way, in which case they should be recorded separately.

Although the tests have been designed with hard disk storage devices principally in mind, other storage devices (provided they allow writing as well as reading) can also be tested. For example, tests could be used for solid-state storage devices.

3.3 File Formats

The tests are run either by a single executable `iot_all` which covers all tests, or by one of the executables `iot_lowlevel, iot_kernel` covering a particular class 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 the following section.

Standard Input keywords	
`timingsfilename`	string, for timing output filename
`class`	string, (`lowlevel` or `kernel`)
`testname`	string, one of the tests listed below
`filename`	string, for I/O data filename
`preallocate`	Logical (`t/f`), *Optional* default `t`
`debug`	Logical (`t/f`), *Optional* default `f`

4 Test Parameters and Output: the Keyword System

Test parameters are supplied using a system of keywords in the `iotparams.in` file. The timings resulting from the test are output in a similar manner. All parameters and timing values are identified with keywords using the following pattern:

`keyword value`

Where `keyword` is one of the keywords defined in this document. The input keywords are used to supply parameters and a different set of keywords is required for each test. The `value` may for example be a string, floating point number or set of integers depending on the keyword. The allowed values associated with each keyword are defined below and in the full description of each test. Once the parameters are read into the test program, they are stored in an MPI-Info object as defined in the MPI-2 standard. Output keywords also depend on the test, the precise significance of the timing associated with each output keyword is defined in this document.

4.1 Input Parameter Keywords

Each test in the suite requires its own specific set of keywords and associated values which are read from the `iotparams.in` file located in the `iotest/data` directory. Some of the keywords are more general and may appear in any test. These are the standard keywords and `MPI_Hints` which are also supported as keywords. We describe these generic keywords below before giving the full definition of the appropriate set of keywords and values for each test in the later part of this document.

Not all relevant parameters can be specified using keywords, 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. Of more concern are other parameters that are fixed at submission time, for example the amount of memory can be requested in the NQS submission method. This information may be important, but it is not available in the timings output file and should be recorded separately.

4.1.1 Standard Input Keywords

The list of standard input keywords is given in the table above. Test parameters that are not *required* are marked as *optional* and defaults are specified.

The `timingsfilename` keyword is only specified once in the input file at the beginning of a series of tests. This string is *Required* and sets the location of the results file, typically it is placed in the `/timings` directory.

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 parameters for a particular test. These are the keywords `class` and `testname` which select which test to run. They are both *Required*. The value of the `class` keyword can be either, `lowlevel` or `kernel` and defines the class of the test. The `testname` keyword selects the test and the string value must be one of the following names of the tests:

`lowlevel: single, multiple, singleI, multipleI, sinunix`

`kernel: matrix2D, matrix3D, gatherscat2D, transpose, nonseq, sharedfp`

The `filename` keyword is *required* in order to specify the location of the I/O test file. It specifies the file system, directory and filename where the data is to be written and read. Some discussion of this keyword was given in the last section.

The two optional keywords `preallocate` and `debug` are both logical variables intended to switch on or off a certain feature. Preallocation of files using the `MPI_file_preallocate` call explained in section 3, is enabled/disabled with the `preallocate` keyword. The `debug` keyword is used for development. These two standard keywords are always available and will not be listed under each test.

4.1.2 Reserved Keywords, MPI Hints

Besides the keywords defined by the test, there are a set of MPI hints that may affect I/O. These hints are passed to the tests in exactly the same way as the test keywords. A full list of the hint keywords is given in appendix A, they are all of the *Optional* class.

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.

All the MPI hints may be supplied to the tests, and all are passed on to the MPI implementation. At the practical level, we do not anticipate that all of these keywords will have an effect in the first generations of MPI implementations. However the following two MPI hints function with the Pallas implementations on the Fujitsu platform.

`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.

Other MPI hint keywords that control important optimisations may be implemented in the future. In the text we have indicated which optimisations should be employed in a given test.

4.1.3 Input File Format

The format of an input file consists of the following pieces:

Standard keywords: The `timingsfilename`, `testname` and `filename` keywords, along with any optional keywords.

MPI Hint keywords: General file hints consisting of the reserved keywords from the MPI standard.

Specific test keywords: Specific options or parameters for that test. The keywords associated with each test are listed in the section describing that test.

Comments: any line starting with a hash symbol.

One test may be run, or a series of tests may be submitted together by combining the keywords in a series of blocks for each test. In the latter case, the `timingsfilename` specifying where the results file is to be located is given only once.

A short example input file for one simple test is shown below and a longer example including comments is given in Appendix C.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
# Template for input file
# IOT: MPI IO Test Suite Release 1.0 (11/8/97)
# Copyright Fujitsu Ltd. 1997
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
#
timingsfilename    /home/djl/SUITE/iotest/timings/lowlevel.out
classname         lowlevel
testname          single
filename          /v0/data/lowleveltest.dat
numruns           3
filesize          0.01 0.1 1
blocksize         0.01 0.02 0.04
#
```

4.2 Output keywords

The output format is a standard ascii file and the timings that constitute the output of the tests are identified using a system of keywords similar to that used on input.

The beginning of the output file is just a copy of the input keywords and parameter values along with a timestamp to identify the run. The number of nodes that the test runs on is not given in the `iotparams.in`, but is deduced while running and printed in the output file against the keyword `nprocs`. The names of the nodes used is also recorded.

This part of the output file is followed by the keyword timing values themselves. In the case that several nodes are employed, there are `nprocs` columns of timing values for each keyword. A final timestamp concludes the test.

Standard Output keywords	
`pre_time`	Set up time
`fopen_time`	File open time
`fview_time`	Time to set fileview
`palloc_time`	Preallocatation time
`sync_time`	Synchronisation time
`post_time`	Clean up time
`fclose_time`	File close time
`total_time`	Total time.
`error`	Error if this appears.

When several tests are run together, the output file consists of a series of blocks of the form just described for a single test. This format enables output files to be split up or concatenated at will, which may be convenient at the analysis stage.

Standard output keywords that describe common timing measurements are listed in the table above. These relate to operations that occur in most of the tests and that have clear significance. Although there are some common I/O timing measurements in the lowlevel tests, the precise significance can vary so we defer discussion of these keywords to the sections devoted to each test. A fuller description of the keywords appearing in the table is given below.

`pre_time` The time taken between the start of the test routine to the beginning of the essential timing part. This includes time to to set up buffers, to open the test data file, and to set up filetypes.

`fopen_time` The time taken to open the file.

`fview_time` The time taken to set the fileview. That is the time taken in the routine `MPI_File_Set_View`. This command is included even when the fileview is the default and it is not strictly necessary. In the case of certain `kernel` class tests, this measurement includes the time needed to set up nontrivial filetypes.

`palloc_time` The time taken to preallocate the test data file with `MPI_file_preallocate` as discussed in section 3 on conventions. If `preallocate` is set `f`, then this result is not reported.

`sync_time` The time taken to complete the MPI instruction `MPI_file_synchronise` which only returns when the data is securely on disk. This call is placed between the writing and reading phases of the test.

`post_time` The time taken between the end of the essential timing part of the test and the end of the test routine. This includes time to close the file and to write measurements to the statistics structure.

`fclose_time` This measures the time taken to close the file.

`total_time` The total time taken to complete the `iot_run_test` subroutine part of the test. The total summed time should relate to the time taken for the test to run, which will be reported separately at some sites.

`error` This keyword does not print a value, but its appearance keyword indicates that there has been some error in the test. In this case the diagnostic output from the test (which is sent to `stdout`) should be inspected carefully. This keyword does not print a value.

4.2.1 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 was 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. When multiple processors are used, separate columns of the output file are assigned to the data from each processor. If there are a very large number of processors, then the `IOT_LINE_LENGTH` variable in `share/iot_defs.f` may need to be adjusted. An example output file is shown in Appendix D.

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 stream of bytes in `blocksize` quantities.

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. 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.

The MPI standard provides various I/O routines that are conveniently described using three orthogonal properties: positioning, synchronism and coordination. The precise method of file pointer positioning should not affect the performance so we shall always (though not for kernel tests) use explicit file-pointers. The space is thus reduced to synchronism and coordination. We have separate tests for synchronous and asynchronous tests, but the collective or non-collective choice is implemented by a keyword supplied to the `multiple` and `multipleI` tests.

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.

It is clearly vital to compare the results of the low-level tests with I/O tests that do not use MPI. We supply a test, `sinunix`, that performs standard Fortran I/O from within an MPI wrapper. This is portable and convenient in that the input and output format is the same as for the `single` test, but because of the MPI wrapper, it is not a completely satisfactory reference test.

5.1 Low-Level Tests, Keywords

These basic tests write a single file in `blocksize` quantities. Because these tests are quite similar to one another, several of the keywords are common. For example the parameters `filesize` and `blocksize`. To facilitate repeated tests over a range of parameters, we allow a series of parameters to be specified.

Input Keywords

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

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

`filesize` (floating point values separated by spaces) The filesize values are given in MB (we use the convention that 1 Megabyte is one million bytes rather than 2^{20} bytes).

`blocksize` (floating point values separated by spaces) The blocksize values are also given in Megabytes.

There must be at least `numruns` pairs of values specified in the `filesize` and `blocksize` rows. If more are specified, they are ignored.

Output Keywords

Besides the common output keywords listed in section 4.2 that measure the time for some standard operations, the main I/O timing measurements have keywords specifying whether they time write or read operations. These measurements are fairly similar for all the lowlevel tests, for example `single` and `multiple` use the short keywords:

`w` The time taken to write `blocksize` Megabytes.

`r` The time taken to read `blocksize` Megabytes.

However we postpone the discussion of the precise significance of each measurement to the discussion in the appropriate section for each test.

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 precise understanding of the timing measurements comes from inspecting 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. The result of this checking is reported under the keyword `sumcheck`, which should be zero for a correctly functioning test.

In principle one would like to make a measurement of the bandwidth for the processor to write to its "local" disk (one that is closely attached to the processor). However, the existence of this kind of architecture cannot be guaranteed and the way the filesystems are set up is certainly machine dependent. In practice we do not anticipate problems because the appropriate filesystem and 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.

Keywords

This test only requires the standard input keywords and the common low-level input keywords. The specific input keywords are shown in the table.

Input keywords	`single`
`class`	`lowlevel`
`testname`	`single`
`filename`	string
`numruns`	integer
`filesize`	list of floating point values in MB
`blocksize`	list of floating point values in MB

The values of the `filesize` and `blocksize` keywords determine the parameters `niter` and `bsize` that appear in the code fragment shown above. The relations are explicitly: `blocksize` = `bsize`×`sizeof(MPI_double_precision)`, `filesize` = `niter`×`blocksize`. Note that the values are specified in MB, here defined as 10^6 bytes rather than 2^{20} bytes. The `numruns` value specifies the number of `filesize, blocksize` pairs to test.

We anticipate, for larger blocksizes, a typical filesize of 1GB. But provided the bandwidth has reached some asymptotic value there is no point in increasing the filesize further. The tests should not be allowed to take too long without very good reason.

The blocksize is the size of the buffer written each time the loop is traversed. We suggest sizes: $4, 16, 64$kB . . . (up to maybe 4MB). 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.

We suggest an initial exploratory run with the following parameters.

`numruns`	4
`filesize`	104.8576 13.1072 1.6384 0.2048
`blocksize`	1.048576 0.131072 0.016384 0.002048

Note the way that power of 2 blocks are described explicitly. Providing values like this makes it more likely that the blocks will fit precisely into system buffers, thereby reducing the measurement noise. In each case we have arranged for the loop to be traversed 100 times. The interesting parts of this range should be investigated more closely.

We remind the user that the standard keyword `preallocate` defined in section 4 is optional and may be specified in any test.

Output keywords	single
sumcheck	0.0 for correct functioning
pre_time	Set up time
fopen_time	File open time
palloc_time	Preallocatation time
fview_time	Time to set fileview
w	Time to write one block
sync_time	Synchronisation time
r	Time to read one block
post_time	Clean up time
fclose_time	File close time
total_time	The total time
error	Error if this keyword appears

On output, all the standard low-level output keywords are used and these were explained in detail in 4.2.

The `w` and `r` times are the times for writing and reading each `blocksize` block of data and make the measurement precisely as shown in the code fragment. They will each appear `niter` times, once for each loop.

`sumcheck` appears twice for each test, constituting two different checks. Its value must be zero for a correctly functioning test.

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. Depending on the filesystem, the device where the file is written could be one of the locally attached disks or, if the hardware and software requirements for a parallel filesystem exist, it could be distributed 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. This option is implemented using a keyword switch.

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 is 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 in separate columns of the timing output file.

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

Keywords

The parameters for this test comprise the `blocksize` and `filesize` which are subject to the same comments as the `single` test, and a new keyword which switches between the non-collective and the collective forms of the I/O routines. MPI hint keywords such as striping values should be varied and investigated in this test.

The number of processors is a parameter in this test, but it is usually supplied on the command line rather than through the input file via the keyword system. It is however reported in the output using the keyword `nprocs`. 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.

Input keywords	`multiple`
`class`	`lowlevel`
`testname`	`multiple`
`filename`	string
`numruns`	integer
`filesize`	list of floating point values in MB
`blocksize`	list of floating point values in MB
`collective`	logical (`t/f`)

The `collective` keyword may take values: `t` or `f` to select the `MPI_File_Write_all` or `MPI_File_Write` call, and a similar choice for read. The meaning of the collective optimisation is discussed in the MPI-2 standard, but may be implemented in various ways.

We suggest similar size filesizes and blocksizes to the ones used in the `single` test for initial explorations.

On output, the keywords are used with precisely the same significance as for the `single` test. All the timing values are measured independently on each processor and are reported in separate columns of the output file. Each column referring to a different processor. In the case of large numbers of processors certain internal parameters defined in the `share/iot_defs.f` file may require adjustment.

The presence of MPI barriers allows us to synchronise the timings on each processor. Barriers are placed at the start of both the write and the read loop. The first barrier separates the setup and file opening (timed using the `pre_time` keyword) from the write loop. The second barrier appears after the synchronisation call that separates the writing and reading phases. The synchronisation of the start of each I/O loop allows us to synchronise the main timing data which is reported with the `w` and the `r` keywords.

The executable programs (wrappers) also contain barriers both before and after the submission of the `iot_run_test` subroutine which is the subroutine that actually performs the test. The time spent in this subroutine is measured using the keyword `total_time`, and as a result of the barriers, will take the same value on all processors.

Output keywords	`multiple`
`nprocs`	Number of processors
`sumcheck`	0.0 for correct functioning
`pre_time`	Set up time
`fopen_time`	File open time
`palloc_time`	Preallocatation time
`fview_time`	Time to set fileview
`w`	Time to write one block
`sync_time`	Synchronisation time
`r`	Time to read one block
`post_time`	Clean up time
`fclose_time`	File close time
`total_time`	The total time
`error`	Error if this keyword appears

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 new variables into the problem: the form 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 been chosen as series of matrix-vector multiplications. The parameters defining the inner and outer loops as well as the repeat count are parameters that can be adjusted with keywords. The essential part of the code is shown below:

```
do k = 1, repeat
   do j = 1, outerloop
      do i = 1, innerloop
        y(i) = y(i) + a(i,j)*x(j)
      enddo
   enddo
enddo
```

The vectors `x,y` and the matrix `a` are initialised with random numbers and all arrays are Fortran `double_precision`.

The issues that are important in choosing the form of interleaving computation are the vectorisability of the code and the memory usage. Matrix-vector multiplication is often used as a benchmark test for floating point performance. This is not an appropriate place to give a full report, but we may summarise that the vectorisability is determined by the parameter `innerloop`. Provided this parameter is not too large, one can regard the other parameter, `outerloop`, as being a way of adjusting the memory required.

The most important aspect of the interleaving computation is the length of time it takes to run one time. Once the vectorisability and memory usage of the computation have been fixed using `innerloop` and `outerloop`, this computational time is controlled by the `repeat` keyword. The precise time duration is obtained in the first cycles of the test which omit the I/O operation and time the computation separately in a synchronous environment. Once these reference cycles have been completed, the main timing loop consists of the computation and the non-blocking I/O routine. The patterns of real applications are best fit by ensuring that the data written or read depends in some way on the results of the computation preceding the I/O instruction in the loop.

From a comparison of the non-blocking timings provided by this test with blocking I/O timings from earlier lowlevel tests, the effectiveness of the asynchronicity may be deduced. The efficiency 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 follows 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 method of timing is slightly different from the blocking `single` case because three routines are involved in the loop and we require three timing calls. A code fragment that illustrates the precise definition is shown below. The `MPI_Wait` call ensures that the asynchronous write has completed at the end of each loop.

```
offset_niter = 0
told = MPI_Wtime()
do j = 1, niter
   offset = offset_niter
   call MPI_File_Iwrite_at(fp, offset,
     buf(1), bsize, etype, request, ierr)
   tpre = MPI_Wtime()
   call comp_task(repeat,innerloop,outerloop)
   tcomp = MPI_Wtime() - tpre
   wtasktime(j) = tcomp
   call MPI_Wait(request, status, ierr)
   tnew = MPI_Wtime()
   writetime(j) = tnew - told - tcomp
   told = tnew
   offset_niter = offset_niter + bsize
enddo
```

The time taken to perform the computational task and the time for the write are measured separately and temporarily stored in the `wtasktime` and `writetime` arrays for later output with keywords `wt` and `wtI` respectively.

This loop is prefaced by a short loop just timing the computational task without any I/O, and using the keyword `t`. The length of this preliminary loop is determined by the `taskloops` parameter. As usual the writing and reading phases are separated by a synchronisation call and self-checking also occurs as before.

Keywords

The only additional parameters beyond those needed in the blocking version are those used to describe the extra computation. These are the `innerloop` and `outerloop` parameters that characterise the form of the computation; for example its vectorisability and memory usage. A further parameter, `repeat`, fixes the number of iterations of another loop that repeats the the multiplication as described in the code on the previous page. The duration of the interleaving computational task is determined by this parameter and it controls the time taken in a linear manner. In practice it is convenient to fix the form of computation using `innerloop` and `outerloop` and then to perform a series of tests with different values of the `repeat`. In preliminary use `repeat` should be chosen small (even 0) in order to check results against the blocking version.

Input keywords	singleI
class	lowlevel
testname	singleI
filename	string
numruns	integer
filesize	list of floating point values in MB
blocksize	list of floating point values in MB
taskloops	integer
repeat	integer
innerloop	integer
outerloop	integer

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

Output keywords	singleI
sumcheck	0.0 for correct functioning
pre_time	Set up time
fopen_time	File open time
palloc_time	Preallocatation time
fview_time	Time to set fileview
t	Task time in isolation
wI	Time to write one block asynchronously
wtI	Task time while writing asynchronously
sync_time	Synchronisation time
rI	Time to read one block asynchronously
rtI	Task time while reading asynchronously
post_time	Clean up time
fclose_time	File close time
total_time	The total time.
error	Error if this keyword appears

The time taken for the computational task without any I/O is first measured (in a loop of length `taskloops`) and reported with the keyword `t`. The effect of the parameters `innerloop`, `outerloop` and `repeat` on this measurement should be checked. Normally the time does not scale linearly with the `innerloops` parameter due to vectorisation, but provided this parameter is sufficiently large, scaling is linear with respect to the other two parameters. Then, referring to the code fragment, `wtI` and `wI` signify the times taken for the computation and for the asynchronous write respectively. The matching keywords for the read phase of the test are denoted, `rtI` and `rI`.

5.4.2 multipleI

The asynchronous version is based on the blocking version `multiple`. The additional parameters 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 able to test the collective features of the MPI implementation and underlying parallel file system. The asynchronous version known as split-collective can also be tested using `multipleI`, and a switch is provided for this purpose. When the collective optimisation is used, the method of timing is similar to the ordinary case, but is clarified in the following code fragment.

```
offset_niter = 0
told = MPI_Wtime()
do j = 1, niter
   offset = offset_niter + offset_proc
   call MPI_file_write_at_all_begin(fp, offset,
     buf(1), bsize, etype, ierr)
   tpre = MPI_Wtime()
   call comp_task(repeat,innerloop,outerloop)
   tcomp = MPI_Wtime() - tpre
   wtasktime(j) = tcomp
   call MPI_file_write_at_all_end(fp, offset,
     buf(1), bsize, etype, ierr)
   tnew = MPI_Wtime()
   writetime(j) = tnew - told - tcomp
   told = tnew
   offset_niter = offset_niter + nprocs*bsize
enddo
```

The relationship between striping parameters and collective parameters is best investigated at the blocking level, so for this test they should be adjusted together.

The layout of the non-collective version of the code is identical to that shown in the `singleI` test. Barriers are placed in the locations described in the `multiple` test, allowing the timing of the I/O loops on different processors to begin at the same time.

Keywords

The input keywords are the same as those required for `singleI` except for the additional keyword `collective` that acts as a switch for the collective optimisation in the same way as it did for the `multiple` test.

Input keywords	`multipleI`
`class`	`lowlevel`
`testname`	`multipleI`
`filename`	string
`numruns`	integer
`filesize`	list of floating point values in MB
`blocksize`	list of floating point values in MB
`collective`	logical (`t/f`)
`taskloops`	integer
`repeat`	integer
`innerloop`	integer
`outerloop`	integer

The output keywords are the same as the ones defined above for `singleI`. Each timing measurement is made independently on each processor and is reported in a separate column of the output file.

Output keywords	`multipleI`
`nprocs`	Number of processors
`sumcheck`	0.0 for correct functioning
`pre_time`	Set up time
`fopen_time`	File open time
`palloc_time`	Preallocatation time
`fview_time`	Time to set fileview
`t`	Task time in isolation
`wI`	Time to write one block asynchronously
`wtI`	Task time asynchronously
`sync_time`	Synchronisation time
`rI`	Time to read one block asynchronously
`rtI`	Task time asynchronously
`post_time`	Clean up time
`fclose_time`	File close time
`total_time`	The total time.
`error`	Error if this keyword appears

5.5 Fortran I/O Test: `sinunix`

In order to compare with I/O not using MPI-2 routines we provide a test that uses standard Fortran I/O. For simplicity of use and portability this test works within an MPI wrapper in exactly the same way as all the other tests. The test follows exactly the pattern of the `single` test For example in the code fragment shown for `single`, the only difference is to replace the MPI-I/O calls with the appropriate Fortran I/O calls.

For various reasons, the MPI wrapper and the limitations of Fortran I/O, this test is not completely satisfactory as a non-MPI reference test. We are not able to provide such stand-alone tests since they will be different depending on the site. Nonetheless, the low-level tests have been written in a transparent fashion in order to expose the essential part of the code and we expect that they can be modified in order to write site specific comparison tests.

Keywords

The keywords are the same as for the `single` test. Note that the way that filenames are specified for MPI-I/O may be different from the way that should be used in this test.

Input keywords	`sinunix`
`class`	`lowlevel`
`testname`	`sinunix`
`filename`	string
`numruns`	integer
`filesize`	list of floating point values in MB
`blocksize`	list of floating point values in MB

In this test the file cannot be preallocated.

Output keywords	`sinunix`
`sumcheck`	0.0 for correct functioning
`pre_time`	Set up time
`fopen_time`	File open time
`w`	Time to write one block
`r`	Time to read one block
`post_time`	Clean up time
`fclose_time`	File close time
`total_time`	The total time.
`error`	Error if this keyword appears

Rather than a synchronisation call, the file is closed and opened to ensure that all data is on disk.

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 [8]. 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.

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.

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.
- Transpose operations are frequently used when performing multidimensional Fourier transforms. For very large arrays, it may be appropriate to do this out-of-core.

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 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 provides 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.

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.

In the following, short code fragments are used as the best way to describe the function of the tests and to be precise about the significance of the measurements. These code fragments ignore some issues that are not relevant to understanding these points and are therefore slightly different from the code to be found in the tests themselves.

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.

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 of size `xsize`×`ysize` (or `xsize`×`ysize`×`zsize` in the 3D case). The processes are viewed as a regular logical grid `xproc` × `yproc` (or `xproc` × `yproc` × `zproc`), and the matrix or 3D array is distributed uniformly across this grid, each processor containing a subarray of size `xsize`/`xproc` × `ysize`/`yproc`, (`xsize`/`xproc` × `ysize`/`yproc` × `zsize`/`zproc`). 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 were the number of processes, the filesize and the blocksize, here the main parameters are the problemsizes, `xsize` and `ysize` (`zsize`) and `xproc`, `yproc` (`zproc`). These parameters are subject to the restriction that `xproc` × `yproc`(×`zproc`) is not greater than the total number of processes. The filesize is given by `xsize`×`ysize`(×`zsize`) times the size of the `MPI_double_precision` data structure, and is reported in the output. The array sizes should be divideable by the number of processors in each dimension so that each processor has an identical load. In fact this requirement is imposed to avoid complicating the analysis, and the sizes of the subarrays are the integer part of `xsize`/`xproc` etc.

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 using a different number of processes from the number 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. We employ these non-trivial fileviews, so the actual write instruction for the full array is a single line of code and this is what is timed.

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 [9]), 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`.

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 A. 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.

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 `xsize/xproc` $\times$ `ysize/yproc`, subarray. In the interests of simple analysis, the matrix will be truncated to divide 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 any initial buffer 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, `matrix2D` reproduces some of the low-level timings such as the time taken to open and to close a test data file.

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 buffered and direct 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 bandwidths should be checked when the chunk parameters are not aligned with the subarray sizes.

The choice of process grid strongly affects the performance. It is better to divide the matrix in the x-direction rather than the y-direction in order to allow longer columns with contiguous memory to be passed to the disc. However, in setting `yproc` $= 1$, the I/O pattern can become so simple that it resembles the stream I/O considered by the low-level tests.

In designing the test there was a question of whether to use the Cartesian topology generator of MPI. If optimisations exist, then the nodes can be reordered so that the Cartesian grid fits the underlying hardware in such a way that allows nearest neighbour communication to be optimised. At present this choice has not been made.

The I/O operations of the sub-matrix on each process are performed 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. The central part of the code is shown below:

```
told = MPI_Wtime()
call MPI_file_write(fp, buf, xsize/xproc, coltype,
  status, ierr)
tnew = MPI_Wtime()
writetime = tnew - told
```

Here `buf` is the two dimensional array containing the sub-matrix data. The `coltype` is one of the special filetypes constructed in order to deal efficiently with two dimensional arrays. It contains holes. In this test we use the individual file pointer version of the I/O routine. A collective version also exists with the `MPI_file_write_all` routine. In order to gather more statistics, a `repeat` parameter may be specified for the number of times to repeat the measurement. MPI barriers are placed both before this segment of code and before the reading phase to ensure that the measurements on different processors start synchronously.

Because there is only one I/O call in the heart of the test, and there is no explicit loop, it is not so useful to consider an asynchronous version of this test. Non-blocking I/O is best tested at low-level using the `singleI` and `multipleI` tests.

Self checking for these tests is thorough and all elements of the matrix that have been written and read are checked against the original matrix. If the check fails then the test aborts.

Keywords

In submitting this test a series of different grids can be investigated and the total number of processors needed for some of the grids may be smaller than the number of processors originally requested. In this case the unused processors are idle.

Input keywords	`matrix2D`
`class`	`kernel`
`testname`	`matrix2D`
`filename`	string
`repeat`	integer
`numsizes`	integer
`xsize`	integer
`ysize`	integer
`numprocgrids`	integer
`xproc`	integer
`yproc`	integer
`collective`	logical (`t/f`)

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

`xsize`, `ysize` The full matrix is of size `xsize`×`ysize`, and usually a square matrix is chosen. Each element is an `MPI_double_precision` data structure. The values of `xsize` and `ysize` should be selected to work in the characteristic regions identified in the low-level tests.

`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`, and one from `yproc`. This pair of numbers specifies the number of processes in a row, `xproc` (or column, `yproc`), of the logical grid of processes. The user must ensure that xproc∗yproc is less than or equal to the total number of processors requested when the job is submitted.

repeat (integer) The number of times to repeat the measurement with a particular set of parameters.

collective (logical t/f, default f) To select the MPI_File_Write_all or MPI_File_Write call, and a similar choice for read.

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 A.

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. The code which defines the I/O timing measurement was given above.

For each test run the following data is output:

Output keywords	matrix2D
nprocs	Number of processors
filesize	Size of matrix (bytes)
pre_time	Set up time
fopen_time	File open time
palloc_time	Preallocatation time
fview_time	Time to set fileview
palloc	Preallocatation time
write	Time to write complete matrix
sync_time	Synchronisation time
read	Time to read complete matrix
post_time	Clean up time
fclose_time	File close time
total_time	The total time.
error	Error if this keyword appears

The total size of the matrix in bytes is reported as filesize, but no blocksize parameter is output.

In this case the fview_time includes the time needed to set up the datatypes required to write and read the matrix.

6.2.2 3D matrix I/O: `matrix3D`

This test reads and writes a complete, distributed 3D array. The array is distributed over the processes in a uniform manner, each process containing a `xsize/xproc`$\times$`ysize/yproc`$\times$ `zsize/zproc` subarray. The array is divided exactly into sub-arrays with no remaining elements. The central code fragment is exactly the same as for the `matrix2D` test.

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, though if `yproc` = `zproc` = 1, the I/O pattern is simple.

Keywords

This test is very similar to the previous one, it only requires two further parameters, `zsize` and `zproc`, that appear in the same way as the x and y parameters in 2-dimensions.

Input keywords	`matrix3D`
`class`	`kernel`
`testname`	`matrix3D`
`filename`	string
`repeat`	integer
`numsizes`	integer
`xsize`	integer
`ysize`	integer
`zsize`	integer
`numprocgrids`	integer
`xproc`	integer
`yproc`	integer
`zproc`	integer
`collective`	logical (`t/f`)

For each test run three numbers are taken: one from `xproc`, one from `yproc`, and one from `zproc`. These numbers specify the number of processes in each of three axes of the logical 3D grid of processes. The user must ensure that `xproc*yproc*zproc` is not greater than the total number of processes.

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

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

Output keywords	`matrix3D`
`nprocs`	Number of processors
`filesize`	Size of matrix (bytes)
`pre_time`	Set up time
`fopen_time`	File open time
`palloc_time`	Preallocatation time
`fview_time`	Time to set fileview
`palloc`	Preallocatation time
`write`	Time to write complete matrix
`sync_time`	Synchronisation time
`read`	Time to read complete matrix
`post_time`	Clean up time
`fclose_time`	File close time
`total_time`	The total time.
`error`	Error if this keyword appears

6.3 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 old-fashioned serial rates with fully parallel output using the `matrix` tests. 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 message traffic to be investigated and will measure the efficiency of the network as well as the I/O performance of the single node.

A short code fragment is shown below:

```
told = MPI_Wtime()
call MPI_alltoallw()
tnew = MPI_Wtime()
gathertime = tnew - told
told = tnew

if (node == 0) then
   call MPI_file_write(fp, fulla,
     xsize*ysize, etype, status, ierr)
   tnew = MPI_Wtime()
   writetime = tnew - told
end if
```

The gather and scatter calls are made using the `MPI_alltoallw` routine (we have not shown all the arguments needed). The reading phase follows, but because the I/O only takes place from one node, no additional barriers are inserted. In this test the barriers appear before the gather and scatter phases which are timed separately on each processor. Since the full matrix is written/read from one node, the I/O timings will not depend on the way that the matrix is divided between processors. Gather and scatter times will however depend on the distribution parameters `xproc` and `yproc`.

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.

Self checking is though as in all matrix based tests. The test aborts if the gathered, written, read and scattered information is different from the original.

Keywords

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

Input keywords	`gatherscat2D`
`class`	`kernel`
`testname`	`gatherscat2D`
`filename`	string
`repeat`	integer
`numsizes`	integer
`xsize`	integer
`ysize`	integer
`numprocgrids`	integer
`xproc`	integer
`yproc`	integer
`collective`	logical (`t`/`f`)

I/O is from one process only, so only one column of timing data is written using the same keywords as for the `matrix2D` test. Additional timing parameters are needed to specify the time taken to gather/scatter the data.

Output keywords	`gatherscat2D`
`nprocs`	Number of processors
`filesize`	Size of matrix (bytes)
`pre_time`	Set up time
`fopen_time`	File open time
`palloc_time`	Preallocatation time
`fview_time`	Time to set fileview
`dtype_time`	Time to set datatypes
`gather`	Time to gather matrix
`write`	Time to write complete matrix
`sync_time`	Synchronisation time
`read`	Time to read complete matrix
`scatter`	Time to scatter matrix
`post_time`	Clean up time
`fclose_time`	File close time
`total_time`	The total time.
`error`	Error if this keyword appears

`gather`, `scatter` These keywords measure the time required to gather or scatter the data to or from the root node. As shown in the code fragment, this is performed using an `MPI_alltoallw` routine, and barriers are placed before each of these calls in order to synchronise the timings on all processors.

`dtype_time` The time to set up the datatypes needed to perform the gather and scatter.

`fview_time` The filetype is also nontrivial and this time includes the time required to set it up and make the `MPI_file_set_view` call.

6.4 Transposed I/O: `transpose`

The test of a matrix transpose operation is motivated by a well known algorithm for performing multidimensional Fourier transforms: in two dimensions for example, a Fourier transform of an `xsize` $\times$`ysize` matrix may be accomplished by first performing `xsize` one-dimensional FFT's down the columns of the matrix, and then doing `ysize` one-dimensional FFT's along the rows of the matrix. In the first phase of the computation, the optimal layout of the data is to distribute the `xsize` columns evenly amongst `nprocs` processors in such a way that elements of the columns appear in contiguous memory locations. This is automatic in a Fortran array. In the second phase of the computation, it is advantageous to put the elements of the rows in contiguous memory locations on the same processor. This necessitates a transpose operation on the two-dimensional array.

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.

Even without the FFT motivation, the characteristic transpose pattern provides a more severe test of the I/O system than the straightforward matrix pattern. It is interesting to measure the performance under such a load.

The kernel test `transpose` first writes a distributed 2D array to a file then reads it back as a transpose. The array distribution is the same as that used in the `matrix2D` test, though with the restriction `yproc` = 1. 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. The method we choose to perform the internal transpose is to use the MPI collective communication routine `MPI_alltoallw`. Although this is simple and provided the implementation is well optimised for this pattern it should be reasonably efficient, it is possible that faster methods based on point to point communication exist [10].

To avoid excessive complication we do not consider the most general pattern in which the matrix can be split into sub-matrices. Instead of the patterns described in the section referring to the `matrix2D` test, we only consider the subclass of patterns in which `xproc = nprocs`, `yproc` = 1. This simplifies the algorithms considerably, yet does not affect the essence of the problem.

A short code fragment is shown below illustrating how simple it is to perform the external transpose. All the work goes into setting up the appropriate filetypes, and is hidden. Note that the fileview is changed between the writing and reading phases. This call also resets the filepointer.

```
told = MPI_Wtime()
call MPI_file_write(fp, buf, xsize/nprocs, coltype,
  status, ierr)
tnew = MPI_Wtime()
writetime = tnew - told

call MPI_file_set_view(fp, offset_zero, etype,
  subtransmattype, datarep, info, ierr)
told = MPI_Wtime()
call MPI_file_read(fp, bufT, 1, strvectype,
  status, ierr)
tnew = MPI_Wtime()
readtime = tnew - told
```

The internal transpose using message passing is performed after the external transpose using the file system. This is done using a single call to `MPI_alltoallw`. As was the case for the external transpose, all the non-trivial work is in setting up the datatypes.

The internal and external transposes are fully checked against each other and the test fails if the results are different. This provides quite a stringent test that the fileview and datatype mechanisms are working correctly.

Keywords

Most of the test keywords are the same as those used in other 2D matrix based tests such as `matrix2D`, however the way that the matrix is divided only depends on the number of processors. The `xproc` and `yproc` keywords are dispensed with and the matrix is always divided in the x-direction with, `xproc = nprocs`, `yproc` = 1.

Input keywords	`transpose`
`class`	`kernel`
`testname`	`transpose`
`filename`	string
`repeat`	integer
`numsizes`	integer
`xsize`	integer
`ysize`	integer
`collective`	logical (`t/f`)

A practical point in this test is to first run it with small matrices because the efficiency may be very low, and it can take a long time to read transposed matrices even of size 100^2.

The output from this test uses exactly the same keywords as the matrix tests. Some additional keywords are used to describe the time taken to perform the transpose using message passing. Barriers are located at the start of the external and the internal transpose and do not occur within each section of code.

Output keywords `transpose`	
`nprocs`	Number of processors
`filesize`	Size of matrix (bytes)
`pre_time`	Set up time
`fopen_time`	File open time
`palloc_time`	Preallocatation time
`fview_time`	Time to set fileview
`write`	Time to write complete matrix
`sync_time`	Synchronisation time
`read`	Time to read transposed matrix
`post_time`	Time between external and internal
`fclose_time`	File close time
`int_pre`	Prepare internal transpose
`dtype_time`	Time to set datatypes
`internal`	Time to transpose internally
`int_post`	After internal transpose
`total_time`	The total time.
`error`	Error if this keyword appears

The significance of the `write` and `read` are clear from the code shown. For the internal transpose there are several new keywords:

`int_pre` Time taken to preparing for the internal transpose, this includes setting up datatypes.

`internal` This measurement is the actual time for the `MPI_alltoallw` call that accomplishes the internal transpose.

`int_post` This is the time after the internal transpose to the end of the test routine.

For this test it is especially interesting to monitor the times to set up nontrivial data structures.

`dtype_time` The time needed to set up appropriate datatypes for the internal transpose. Most of `int_pre` is for this purpose.

`fview_time` In the external part of the test, this keyword appears twice. Once to report the time to set up the filetypes and set the fileview for writing the matrix and a second time between writing and reading in order to set new filetypes for the transpose and to reset the fileview.

6.5 Non-Sequential I/O: nonseq

The nonseq test is 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 and the matrix tests.

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 have opted for an approach which is based on a simplified and general algorithm, but allows 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 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. A brief piece of descriptive code is shown below:

```
updates = 0
told = MPI_Wtime()
do j=1,trials
   offset = fsize*iotrand(IOT_RSEED)
   call MPI_file_read_at(fp, offset,
     rbuf(1), bsize, etype, status, ierr)
   tnew = MPI_Wtime()
   readtime(j) = tnew - told
!  Touch data
   told = tnew
   rand = iotrand(IOT_RSEED)
   if (rand < update_frac) then
      updates = updates + 1
      call MPI_file_write_at(fp, offset,
        wbuf(1), bsize, etype, status, ierr)
      tnew = MPI_Wtime()
      updatetime(updates) = tnew - told
      told = tnew
   endif
enddo
```

The file is filled with random data before the main part of the test. The offset at which to read a block of data is set to a random value (aligned with the blocksize). and the time to read a block is measured. As usual, the data is touched in order to ensure that it has really been read. In update_frac of the cases, the same block is changed by being rewritten. The time for this is also measured. The random seeds are different on each processor so each node searches, reads and updates independently. For this reason, the quantity of data from each processor may vary slightly in which case some columns of data are zero filled.

Keywords

Input keywords	nonseq
class	kernel
testname	nonseq
filename	string
filesize	floating point value in MB
numblocksize	integer
blocksize	list of floating point values in MB
numupdate	integer
update_frac	floating point

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

The blocksize values are given in Megabytes. In contrast to the low-level class, this parameter need not become very large and a good choice is to set it equal to the pagesize.

The `numupdate` keyword specifies the number of update fractions given in the list following the `update_frac` keyword. This floating point fraction (value 0 to 1) specifies the fraction of blocks that after reading, are to be modified and rewritten to the file.

It is unlikely that this application will benefit from the collective optimisation and we have not included this option.

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 update writes since these operations will be interleaved. The standard keyword "`r`" is used for reads and outputs the `readtime` array shown in the code segment. The new keyword "`u`" indicates an update and corresponds to the `updatetime` array shown in the code.

Output keywords	nonseq
nprocs	Number of processors
pre_time	Set up time
fopen_time	File open time
palloc_time	Preallocatation time
fview_time	Time to set fileview
write	Time to write the file
r	Time to read
u	Time to write the update
post_time	Clean up time
fclose_time	File close time
total_time	The total time.
error	Error if this keyword appears

The time taken to write the file before the main part of the test is measure and reported with the `write` keyword. Only the total time to write the complete file is measured, and this time should agree with values deduced from the lowlevel tests.

The fileview is trivial in this test, but the `fview_time` is still measured as it was for the lowlevel class. Synchronisation time is not important and is not measured in this test.h There is no self checking in this test.

6.6 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 is also relevant when checkpointing.

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 lying between `blockmin` and `blockmax`. The quantity of data actually written will therefore lie between these extremes and have mean $(\texttt{blockmin} + \texttt{blockmax})/2$ megabytes. 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 can therefore be used.

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

A short code fragment is shown below. Note that independent random seeds are used for each process so the size of the blocks of data are not the same across processes. The loop parameter `niter` is determined by the average blocksize as, $niter = 2 \times filesize/(blockmin + blockmax)$. This parameter must be large for a useful test, in which case the final file will be similar in size to the requested `filesize` value.

```
bsum = 0
told = MPI_Wtime()
do j = 1, maxniter
   bdsize = int(bmaxsize*iotrand(IOT_RSEED),IDBL) + 1
   bsize(j) = bdsize
   bsum = bsum + bdsize
   if (bsum*nprocs > filedsize) exit
   call MPI_file_write_shared(fp,
     wbuf(1), bdsize, etype, status, ierr)
   tnew = MPI_Wtime()
   writetime(j) = tnew - told
   told = tnew
enddo
```

Keywords

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

It is interesting to also consider the collective version of the shared filepointer routine called `MPI_file_write_ordered`. As suggested by the name, the accesses are ordered by the ranks of the processes. The number of calls to this routine are the same on each processor, but the amount of data written The keyword `collective` (default `f`) may be used to select this option.

Input keywords	sharedfp
`class`	`kernel`
`testname`	`sharedfp`
`filename`	string
`filesize`	floating point value in MB
`numblocksize`	integer
`blockmin`	list of floating point values in MB
`blockmax`	list of floating point values in MB
`collective`	logical (`t/f`)

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

`numblocksize` (integer) This specifies the number of `blockmin/blockmax` pairs to test from the following lists.

`blockmax`, `blockmin`: The uniform distribution used to select the blocksizes lies between `blockmin` and `blockmax` bytes. These parameters should be varied. Often `blockmin` will be set to zero.

Output from this test includes the blocksize actually written as stored in the `bsize` array in the code above. The timing of the write call is also measured in the usual way as shown in the code. The keywords `b` and `w` referring to these respective measurements correspond to the temporary arrays `bsize()` and `writetime()` shown in the code.

A barrier is placed before the code segment shown allowing measurements to be synchronised.

Output keywords	sharedfp
`nprocs`	Number of processors
`pre_time`	Set up time
`fopen_time`	File open time
`palloc_time`	Preallocatation time
`fview_time`	Time to set fileview
`b`	Number of bytes written, integer
`w`	Time to write these bytes
`post_time`	Clean up time
`fclose_time`	File close time
`total_time`	The total time.
`error`	Error if this keyword appears

There is no self checking because the file is not reread. Synchronisation time is not measured in this test.

7 Analysis

The tests generate raw timing data with very clear and precise meaning. To interpret this data so as to address the problems of I/O performance, 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 this section we give a brief description of the analysis tools available and examples of the kind of output they generate. For a more though description of the use of the suite in evaluating the performance of a particular platform see [11].

At the present stage the tools are presented in the form of a series of `awk` scripts that process the output files to calculate averages and generate graphs using `gnuplot`. A unified graphical tool might be more convenient, but the script approach offers simplicity and flexibility in devising new methods of analysis. All these script tools may be found in the `analysis` subdirectory of the distribution.

One of the first steps in analysis is to have at hand some idea of the hardware performance. This is usually available in manuals. A measurement of the extent to which this performance is achievable in non-MPI tests is vital to provide a reference comparison with the results of the low-level tests. As part of the suite, we have provided the `sinunix` test to perform this measurement. This test is not optimised for any site and it may be advisable to supplement it with the user's own tests.

One simple utility tool, `checktest`, is provided to ensure that the checksum values are correct and that no `error` flags have been set. This tool also reports on the contents of the output file. The syntax of this tool is:

```
checktest file.out
```

where `file.out` is the output file to be checked.

7.1 Analysis of Low-Level Tests

As explained in section 5 the lowlevel tests generate more than than just a single number per test. Here we illustrate how the kind of data generated by a lowlevel test may be processed by the analysis tools to measure interesting underlying parameters that characterise the machine.

7.1.1 Tools for raw data

The significance of the timing measurements produced in the `single` test is apparent from the code fragment shown in section 5.2. The `rawdata` tool makes a plot of the successive times versus the loop counter, which measures the extent of the file as it grows. This provides a detailed picture of the I/O process and an example is shown in figure 2.

This picture appears complicated because of the strong fluctuations in successive timings, and also because there are two regimes, an initial fast phase and a slower phase which can be sustained. We emphasise that fluctuations in the rate are not rare, and that if anything, figure 2 shows a particularly clean example.

The two regimes, initial and sustained, are clearly separated and correspond to a buffer (of about 13MB) becoming full. When taking averages, it is important to specify over what

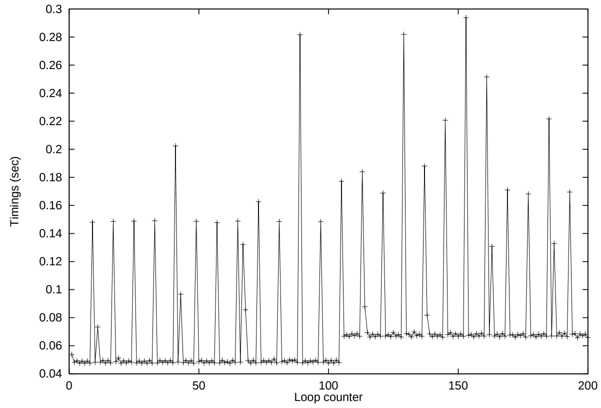

Figure 2: Raw I/O times using `single` on a Fujitsu VX-4. File incremented in blocks of size 0.131MB. The initial and sustained regions are clear as is the periodic structure.

regime they are taken. The relevance of the averages to rates achievable in full applications will depend on the regime.

The fluctuations display an impressive periodicity between the successive longer time peaks amounting to 1MB. The origin of this structure could lie in buffering, scheduling, the replacement order of virtual pages or some other effect. The pattern provides valuable information about the detailed working of the I/O system and although this may be interesting in optimising performance it requires a deep knowledge of the implementation and platform to make use of it. We merely note that the test suite is able to detect this structure, our main purpose in showing figure 2 was to illustrate the difficulty of taking and interpreting averages. In the next section we shall proceed to consider rates that are averaged over the course of many periods, but it is clear that the average will certainly not correspond to a typical measurement. It is even harder to decide on an appropriate measure of the error.

We emphasise that unless this kind of observation of the raw data is made, and the effect of any startup and scatter understood, there is little sense in taking averages. Understanding the raw data may be more significant than having accurate averaged data when attempting to predict I/O times within applications.

The raw timing data is useful in its original form, rather than in the form of rates in MBytes/s, when comparing small buffer sizes. It often happens that in this limit the time of each write call is almost independent of the amount of data written and to investigate such issues the unprocessed times are most appropriate.

`rawdata`

This tool selects a particular test in an output file and generates a postscript file of the raw data corresponding to a given keyword. It may be used for tests run on multiple processors, in which case it generates several graphs each displaying the timing of a separate process. The syntax is as follows:

```
rawdata file.out testname filesize blocksize key
```

`file.out`: The name of the timings output file obtained from running one or several tests from the suite. This is usually placed in the `timings` directory and is often called all.out.

`testname`: The particular test to be analysed. This will usually be one of the `lowlevel` class tests listed earlier.

`filesize`: The filesize parameter, identifying the particular test run to be analysed. Given in megabytes exactly in the same form as in the `iotparams.in` input file.

`blocksize`: The blocksize parameter, identifying the particular test run to be analysed. Given in megabytes as in the input files.

`key`: The keyword specifying which measurement is to be analysed. For the blocking tests and for `sinunix` this will be `w` or `r` for writing or reading. For the asynchronous tests, other possibilities are available: `t, wtI, rtI, wI, rI`. In fact, any of the output keywords corresponding to a loop in the test may be used.

The postscript files are placed in the file `testname.ps` where `testname` is the parameter supplied to the tool. In the case of tests run on multiple processors the postscript file will contain several pages. All pages are labeled.

An example which would be appropriate for generating the graph shown in the figure is:

```
rawdata lowlevel.out single 26.2144 0.131072 w
```

`allrawdata`

Sometimes when a run contains several tests with different parameters, it is convenient to analyse all results with the same key together. This may be done with the tool `allrawdata`. The syntax is similar to that of `rawdata`, but the identifying parameters `filesize` and `blocksize` are not necessary. This tool is therefore useful even for kernel tests such as `nonseq`. `allrawdata` generates temporary files called `testname.tmp1`, `testname.tmp2` *etc* which can be used as the basis of further analysis. For example, these temporary files can be used when partial averages are needed.

7.1.2 Tools for data distributions

`distribution`

A plot of the distribution of the timings can be generated using this tool. This may be convenient when the rawdata plot contains more noise than the one shown in figure 2. An example is shown in 3 for the same set of data. When more noise is present the peaks would be broadened, but the overall picture would remain.

The plots generated by this tool may expose some structure which could provide clues about how the I/O is performed. If noise is present they may also be useful for estimating the errors present in averages.

`distribution`

The frequency distribution of a data set may be plotted using this tool. Because we may want to look at the distribution within some restricted range of the loop parameter, as input we use the temporary files generated by `allrawdata`. This tool is a simple `awk` script and the syntax is:

```
distribution -v B=# file.tmp
```

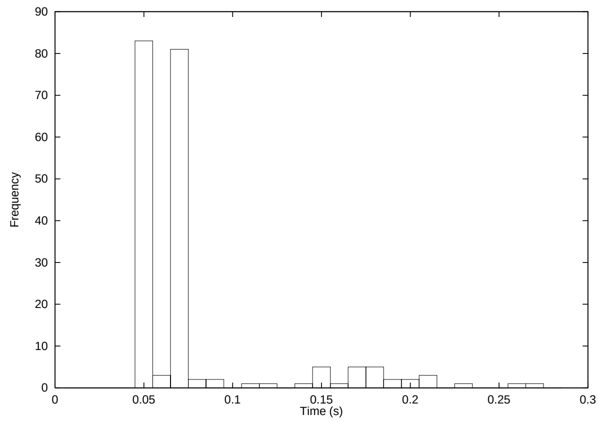

Figure 3: Distribution of I/O times of the plot shown in figure 2. Obtained using the `distribution` tool.

The size of the bins is specified using the `B=` format as shown in the line above. The value is given in seconds.

The tool may be used as a pipe while working within `gnuplot`. For example the following command was used to generate figure 3.

```
gnuplot> plot '< distribution -v B=0.02 single.tmp5' with boxes
```

For data consisting of several columns referring to measurements on different processors, the tool requires minor modification or the `.tmp` files should be edited.

7.1.3 Tools for performance curves

Performance curves of the kind discussed by Chen and Patterson [3] show how the I/O rate varies with filesize and blocksize. Each point on the curve is an average of the raw measurements, and it is apparent from figure 2 that this average will depend on how it is taken.

Provided the data is reasonably clean and does not drift over long periods, one may legitimately calculate an averaged sustained 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 useful reference figure. As emphasised earlier, it does not necessarily correspond to any typical measurement. Any averages over smaller files, and any averages that include the initial period should be compared with this reference bandwidth. The traditional average obtained by placing timing calls outside the I/O loop can be recovered by summing the individual block times and creating an average using the total amount of data written or read.

An example of performance curves of rate against blocksize are shown in figure 4. The curve shows an average taken only in the sustained region. Because of the peaked distribution shown in figure 3, Gaussian error bars do not capture the possible range of measurement. The errors shown in figure 4 are based on the difference between the average and the minimum timings. The interpretation is that the rate is frequently at the top of the error bar, but occasionally it is much lower than the bottom error bar.

Curves of this type form the basis for analysing filesystem performance. For example, the drop in performance for blocksizes below the knee at 32kB in figure 4 should be understood. The changes in the multiple version of the curves as various filesystem parameters

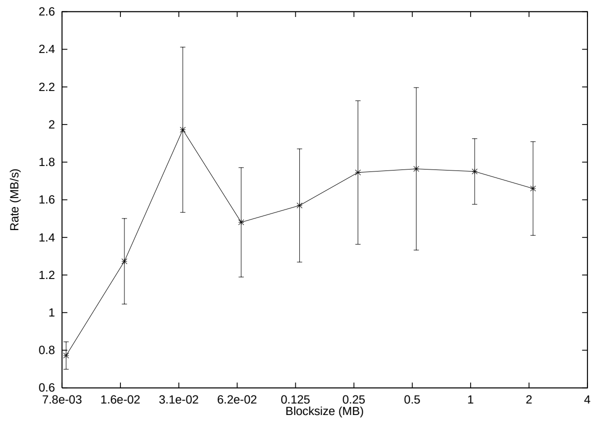

Figure 4: Averaged write rates for the `single` test on a Fujitsu VX-4. See text for a discussion of the significance of the error bars.

(such as those relating to striping) are varied will help to identify the best way to use these optimisations.

`allav`

As first step in constructing averages we provide the analysis tool `allav`. This calculates a simple average over the complete data range, which may be convenient on first investigating data.

```
allav file.out testname key
```

The parameters have the same meaning as for the `rawdata` tool. The averages along with maxima, minima, standard deviation and other measurements are placed in the file `testname.avs`.

As with the `rawdata` tool, the `allav` tool may be used for any lowlevel test and also with kernel tests. Care should be exercised in what quantities are averaged.

The averages are still only of the original timings and to evaluate rates for a performance curve, further calculations must be made. The format of the `.avs` files is intended to help this last stage.

To take account of the discussion above, more sophisticated work must be done to select the precise region in which averages are to be taken, and to evaluate errors. For example, averages may be taken using the temporary files generated by `allrawdata`. These are named `testname.tmp1`, `testname.tmp2` *etc.* They may be truncated appropriately and the averages performed using some other tool.

7.2 Analysis of Kernel Tests

The tools `allrawdata` and `allav` that we have already discussed may still be used for some of the kernel tests. However, some additional specialised tools are required in certain cases.

7.2.1 matrix tests

All the matrix based tests `matrix2D`, `matrix3D`, `gatherscat2D` and `transpose` generate similar kinds of information. Instead of the central loops of the lowlevel tests, single measurements are taken, but there is a repeat parameter. Because not as much data is generated as for the lowlevel tests, at the initial stage of looking at the raw data, there is less need for a tool. The output file can be inspected directly or `grep` can be used. At the level of taking averages, because of the different output pattern, the `kallav` tool replaces `allav`.

As an example, the `matrix2D` test produces a single timing measurement on each processor for each problemsize and matrix distribution. The low-level class of tests may already have isolated startup effects which will be incorporated in these measurements. The initial visual check can be used to see how well the measurements on different processors agree. It is also possible to get a rough idea of how the distribution pattern affects performance. For more precise information, we follow the approach recommended in the lowlevel class and use `kallav` to generate a performance curves for different size matrices. These curves should be compared with the low-level performance curves to see how well the filesystem copes with the matrix I/O pattern. Then a series of such curves should be generated for different optimisations. The individual optimisation parameters will have effects that are highly correlated. For example the values of collective buffer size, chunk size and striping size should be related.

`kallav`

This tool takes simple-minded averages over the full data set, in this context meaning over all the repeats of the measurement.

```
kallav file.out testname key
```

These parameters have the same significance as for earlier tools. The `key` may be chosen to be any one of the keywords associated with the matrix tests. It is not restricted to `write`, for example the average synchronisation time can be calculated using `sync_time` as key.

The averages of the timings are placed in the `testname.avs` file with similar format to that generated by the `allav` tool.

7.2.2 `nonseq` test

The purpose of this test is to provide a some performance data for non-sequential I/O. At the most basic level a direct comparison of averaged rates with similar quantities calculated from sequential data (from one of the lowlevel tests) is sufficient. After first inspecting the data with the `allrawdata` tool to ensure that averages will make sense, the computation may be done using the `allav` tool already discussed. These averages should of course be compared at the same `blocksize`. Typically non-sequential reads are considerably slower than sequential ones.

At a more sophisticated level it is instructive to see how the I/O times are distributed, and to compare the distribution with the sequential version. For example, the distribution of read times on the Fujitsu VX-4 has the characteristic shape shown in figure 5.

No special tool is required to make figures of this kind, the `distribution` tool, acting on the temporary file produced by `allrawdata` is sufficient. When using multiple processors, the temporary files need slightly more manipulation before processing with `distribution`.

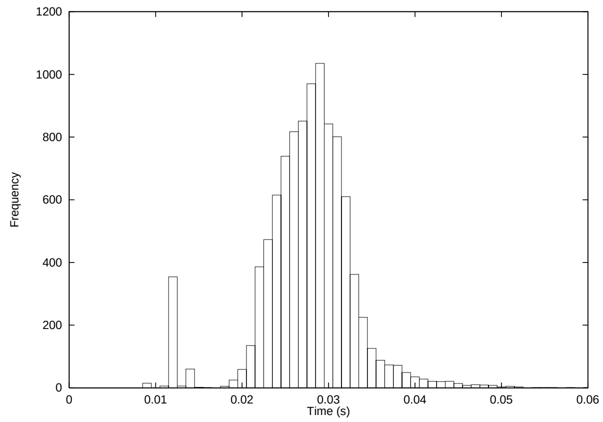

Figure 5: Distribution of read times (sec) for the `nonseq` test.

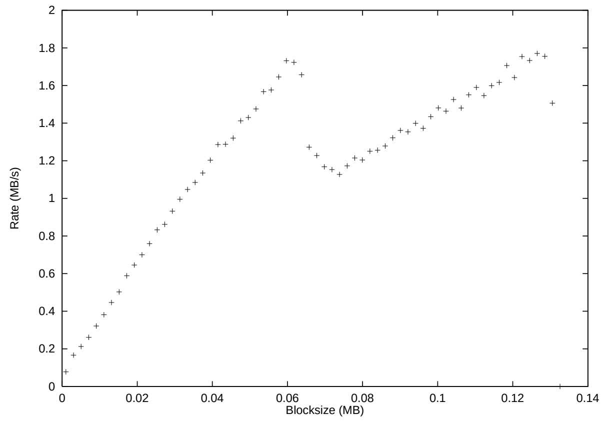

Figure 6: Averaged rates for various blocksizes. `sharedfp` test.

7.2.3 `sharedfp` test

In this test, random quantities of data are written using the shared file pointer of MPI. None of the tools we have discussed so far are useful in this case because they can only consider measurements with one keyword. Here both the amount of data actually written (keyword `b`) and the time taken to actually write it (keyword `w`) are needed for any sensible analysis.

Two tools, `awkallsfp` and `awkdistsfp` are provided. The first tool produces a temporary file containing the `b` and `w` information in a format convenient for subsequent processing with `awkdistsfp`. An example of a plot generated in this way is shown in figure 6.

The figure 6 shows the binned averaged rate for blocksizes up to 128kB. The information is similar to that contained in the performance curves, but over a smaller range of blocksizes (between `blockmin` and `blockmax`). It refers to rates obtained with the shared filepointer routines and should be compared with performance curves based on the low-level tests. These plots can expose interesting effects such as the dip at 64kB in the figure. The effect of optimisations may be deduce from a comparison of curves.

```
awkallsfp file
```

This tool merely selects the data with the `b` and `w` keywords and reformats it in files `sharedfp.tmp1 etc.` The reformatted information is presented with two columns per processor, giving the blocksize and write time respectively.

```
awkallsfp file.out
```

No `testname` or `key` is needed in this case.

The `.tmp` files may need editing before being passed to the next stage of processing.

`awkdistsfp`

The tool `awkdistsfp` evaluates individual rates for each write operation from the reformatted data. These are then placed in bins to create a distribution of rates. Note that this is different from calculating average rates from the total times and volumes in each bin.

The `awkdistsfp` tool has the same syntax as the general `distribution` tool discussed earlier.

```
awkdistsfp -v B=# file.tmp
```

The size of the bins is specified as shown in the line above, and is given in bytes.

To use this tool as a filter while working within `gnuplot` one may use the command (for example):

```
gnuplot> plot '< awkdistsfp -v B=2024 sharedfp.tmp1'
```

The same tools can be used for multiple processors, but in the case of the `awkdistsfp` tool, this first requires some minor modifications.

Acknowledgments

I am indebted to Tim Oliver who has written a considerable part of the test suite. I would also like to thank, Panos Melas, Dave Snelling, Ed Zaluska and Lloyd Lewins.

A 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) 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 NQS Script Example

An example NQS job script is shown below. It contains some lines that are specific to the MPI-I/O implementation on the VX-4. These should be regarded as illustrating the format and can be altered where necessary.

```
#!/usr/bin/sh
## @$-q normal # default queue
# @$-q bench # bench queue
## @$-mb -me   # send mail at start and end
# @$-lM 150MB  # memory for NQS job
## @$-P mpiio # MPI-IO charging project
# @$-lT 50 # maximum time per process
# @$
VPP_MBX_SIZE=8192000; export VPP_MBX_SIZE
cd $IOT_HOME/data
/home/oliver/pallas.new/lib/uxpv/mplib2/mpiexec -n 2 $IOT_HOME/bin/iot_lowlevel
```

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
##################################
%
#
# Timingsfilename 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.
#
classname          lowlevel
testname           single
filename           lowleveltest.dat
#
# Options valid for this test
#
# Number of filesize/blocksize pairs to read from list
numruns            2
# List of filesizes (in MB = 1000000 bytes)
filesize           0.01 0.1 1 10 100
# List of blocksizes (in MB = 1000000 bytes)
```

```
blocksize         0.005 0.05
#
############################
# Classname and testname for the second set of tests
#
classname         lowlevel
testname          multiple
#
filename          lowleveltest.dat
preallocate       t
#
# Options valid for this test
#
collective        t
striping_factor   2
striping_unit     131072
numruns           1
filesize          0.5
blocksize         0.1
#
```

D Output File Example

An example of an output file which corresponds to the results from the tests specified in the input file above. This is run on two processors.

Diagnostic information is also output from the test in the `stdout` stream.

```
## Time stamp ##################################################
timestamp          19980122 180638.040 +0000

## General run information ###########################################

nodename           vx4_pe001.
mpi_wtick                0.16249833E-05

## Test input arguments ##############################################

nprocs                                 1
classname          lowlevel
testname           single
filename           lowleveltest.tst
filesize           0.01 0.1 1 10 100
blocksize          0.005 0.05
numruns            2

## Test output information ###########################################

filesize                           10000
blocksize                           5000
```

```
niter                              2
fopen_time              0.11510979E+01
palloc_time             0.22923125E-02
fview_time              0.56950003E-03
pre_time                0.11555678E+01
sync_time               0.13776750E-01
fclose_time             0.30158437E-01
sumcheck                0.00000000E+00
sumcheck                0.00000000E+00
w                       0.31544375E-02
w                       0.29346875E-02
r                       0.26913750E-02
r                       0.27487500E-02
post_time               0.30201625E-01
total_time              0.12133739E+01

## Test output information ###########################################

filesize                      100000
blocksize                      50000
niter                              2
fopen_time              0.53478662E+00
palloc_time             0.22298750E-02
fview_time              0.31793752E-03
pre_time                0.93187075E+00
sync_time               0.17416125E-01
fclose_time             0.29096875E-01
sumcheck                0.00000000E+00
sumcheck                0.00000000E+00
w                       0.55810625E-02
w                       0.65319375E-02
r                       0.37397500E-02
r                       0.40741250E-02
post_time               0.29158875E-01
total_time              0.10006086E+01
## Time stamp ##################################################
timestamp           19980122 180640.290 +0000

## Time stamp ##################################################
timestamp           19980122 180640.370 +0000

## General run information ###########################################

nodename            vx4_pe001.          vx4_pe003.
mpi_wtick               0.16249833E-05

## Test input arguments ##############################################

nprocs                             2
classname           lowlevel
```

```
testname            multiple
filename            lowleveltest.tst
striping_factor     2
striping_unit       131072
collective          t
filesize            0.5
blocksize           0.1
preallocate         t
numruns             1

## Test output information ###########################################

filesize                          500000              500000
blocksize                         100000              100000
niter                                  2                   2
fopen_time                0.14868291E+01      0.14868437E+01
palloc_time               0.61417500E-02      0.61361875E-02
fview_time                0.58437500E-03      0.55256253E-03
pre_time                  0.22692001E+01      0.22715652E+01
sync_time                 0.35472000E-01      0.31329375E-01
fclose_time               0.44092937E-01      0.38785250E-01
sumcheck                  0.00000000E+00      0.00000000E+00
sumcheck                  0.00000000E+00      0.00000000E+00
w                         0.12539438E-01      0.25290688E-01
w                         0.23637250E-01      0.15018500E-01
r                         0.10360562E-01      0.17218875E-01
r                         0.89921250E-02      0.74486875E-02
post_time                 0.44171250E-01      0.38860812E-01
total_time                0.24109413E+01      0.24130340E+01
## Time stamp ##################################################
timestamp           19980122 180642.810 +0000
```

References

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

[2] D. Lancaster, *Specification of the Parallel Input/Output Test Suite Version 1.0* Report prepared for Fujitsu European Centre for Information Technology, August 1997.

[3] 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.

[4] 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`

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

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

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

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

[9] 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.

[10] J. Choi, J.J. Dongarra, D.W. Walker, *Parallel Matrix Transpose Algorithms on Distributed Memory Concurrent Computers*, October 1993. Available with PUMMA part of SCALAPACK.

[11] D. Lancaster, *Parallel I/O Performance Evaluation of the VPP700*, Report prepared for Fujitsu European Centre for Information Technology, February 1998.