HPCx
CAPABILITY COMPUTING

Investigating MPI-IO on HPCx

Elena Breitmoser

June 17, 2003

Abstract

The purpose of this report is to investigate parallel I/O on HPCx, to compare its performance with serial IO performance. Using a simple benchmark program, we demonstrate that the use of MPI-IO is beneficial, resulting in approximately twice the bandwidth of standard serial I/O, whilst maintaining processor-number independent files. We also investigate the usage of MPI specific hints and directives on parallel I/O performance, and conclude that in most cases the default settings are optimal.

1 Introduction

Nearly every code is faced with the need to read in data and write the calculated results to file. This can either be achieved by using serial or parallel I/O. In the case of serial I/O, the simplest approach is to do it all through the master process, which then has to distribute the data to all other processors which require it. The results have to be collected on the master process which is responsible for writing it to file. We will present the I/O performance for an example of this type serial–I/O code on HPCx.

Another option is parallel I/O. This choice offers several advantages over serial I/O. Parallel I/O, for example, allows the reading of a single file without having to worry about how many processes were used to write it—each processor can read its bit without the need for communication between a master process and all the other processes. Parallel I/O also avoids having to collect all data onto the master process before writing it to file. MPI–IO is usually a very neat way to code the I/O and will often allow more portability than hand–written I/O. In general, it is rather difficult to achieve decomposition–independent file formats for 2D or 3D processor topologies without using MPI–IO. Though, in general, its performance is not necessarily better than for serial I/O. But in contrary to what most people might expect, MPI–IO performs better than serial I/O for large data sets on a larger number of processors on HPCx.

In this report we investigate I/O–files up to about 65 Gb [1]. We use a

[1] 1 Gb = 10^9 bytes.

3D array and a 3D domain decomposition. The local array includes a halo region which is not written to file. Using the right `filetype`s for parallel I/O enables the user to write data to disk in the right order, to do so in one single call and to avoid having to separate the halo from the rest of the array in the writing phase.

In order to achieve best performance for parallel I/O several recommendations can be given in principle:

- Use collective write and read routines (see [3])
- Use the *native* data representation in the MPI–subroutine where the fileview is set (unless data conversion is required)
- Use MPI–IO *hints* and the MPI–directive `MPI-MODE_UNIQUE_OPEN` where appropriate.

In this report we compare parallel I/O performance for the default setting on HPCx with the performance obtained by using the IBM–specific MPI–IO hints and the MPI–directive mentioned above.

In the following Section we give a brief summary of the IBM specific MPI–IO hints and compiler directives investigated. In Section 3 the experiments performed are described, the results presented and discussed. We summarize the investigated cases and the draw final conclusions in Section 4. The main parallel I/O code fragments are shown in the Appendix A and the obtained benchmarking results can be found in Appendix B.

2 IBM specific MPI-IO hints and MPI directives

Much of the optimization that MPI-IO can provide depends on the effective use of appropriate user–defined file views. This file view defines, in an MPI subroutine, which data are visible to each process. In addition, there is an MPI–specific *info–object* variable for MPI subroutines. This provides a mechanism to pass a package of options to a routine. An info–object is defined by the `MPI_INFO_CREATE` routine. If no options are to be set, `MPI_INFO_NULL` can be used. By calling the `MPI_INFO_SET` routine this info can be set by the user. Possible options are MPI–IO hints. IBM specific MPI–IO hints tailor the use of the file system to the application needs, thereby improving performance. An inappropriate use of hints may degrade performance (see [1]). A list of supported file hints for MPI-IO can be found in the MPI Subroutine Reference [2]. A discussion about MPI-IO hints and directives and their effective use is presented in [1] and [3].

In this report the hints `IBM_io_buffer_size` and `IBM_largeblock_io` are investigated. According to [2], an increase in the `IBM_io_buffer_size` can improve performance when using files of hundreds of megabytes, particularly if the code uses collective data access operations such as `MPI_file_write_all`. `IBM_largeblock_io` should be beneficial to applications where each task accesses a large, contiguous chunk of the file, or in which the file is divided into distinct regions that are accessed by separate tasks. Note that by default, `IBM_largeblock_io` is disabled. The usage of `IBM_io_buffer_size` and `IBM_largeblock_io` are mutually exclusive.

In addition, the directive MPI_MODE_UNIQUE_OPEN is investigated. This directive activates GPFS file partitioning. It can be switched on in the MPI_FILE_OPEN routine. This will most benefit applications which write several small data records out of each GPFS block they access. The potential cost is that when any node other than the one granted rights by the GPFS file partitioning map tries to access a block, GPFS must ship the data. In contrast to hints, MPI_MODE_UNIQUE_OPEN is a directive and cannot be ignored when used.

The environment variable MP_EAGER_LIMIT affects the latency of MPI messages. Its value determines the message size above which the rendezvous protocol is used. For small messages (below this threshold) the header and the message data are sent together. For large messages only the header is sent initially, and the message data only follows on receipt of an acknowledgement from the receiver. Since the maximum value of MP_EAGER_LIMIT is 64Kb, this only affects small messages. In the cases studied here, the messages are much larger than this.

2.1 I/O on HPCx

On HPCx there are a total of four LPARs which deal purely with I/O: the disks are attached to these four LPARs. These I/O LPARs are connected to the rest of the system via the same switch which connects the compute LPARs. The maximum capcity of a switch connection is, in practice, around 350 Mb/s per second, so the I/O bandwidth from a single LPAR cannot exceed 350Mb/s, and the aggregate I/O bandwidth cannot exceed 1400 Mb/s. It is important to note that the limits on I/O bandwidth are due to the switch capcity, not to limits on the acess rate to the disks themselves.

3 Benchmarking

In this report serial I/O, MPI–IO with default settings and MPI–IO using MPI–directives and hints are investigated. A FORTRAN 90 code is used for this purpose. The bandwidth is summed over the number of processors. A small array (of local size 100^3) and a large array (of local size 400^3) are investigated. They are both of type MPI_DOUBLE_PRECISION. The small array corresponds to 8 Mb/processor (Mb/PE), the large one to 512 Mb/PE. Each processor calculates a local array of that size. The 3D data is distributed in a 3D domain decomposition across the processes. The local arrays include halos in 3D which are not written to file. The total filesize is the local array size multiplied by the number of processes used (up to a maximum total filesize of 65 Gb when parallel I/O is used). It is important to note that the writing and reading is done in the same code. First the results are written to disk then they are read back in again.

For parallel I/O, this locally contiguous 3D array is mapped to the output file such that it will be stored there in globally column major order. In order to achieve this mapping an appropriate MPI–filetype is defined so that the local data can be written out with one call. The code investigated relies on parallel I/O and on the collective, blocking MPI_file_write_all. For each benchmark, MPI_BARRIER is called before beginning the timing,

making each task's first call to the MPI–write or read well synchronised. Since in MPI semantics, the return from an MPI–write operation does not guarantee that the data have been committed to disk, the file is closed and `MPI_BARRIER` is called again before taking the end time stamp. It is sufficient to include the `close`–file–statement both for the parallel and the serial I/O in the timing. The essential parts of the parallel code can be found in Appendix A.

We also use a serial code where all I/O is done from a single processor. All locally calculated data are send to one processor which is responsible for writing to disk and reading from file. This second code is used to compare performance between serial and parallel I/O. For parallel I/O, the native data representation is used: for serial I/O, unformatted data output.

The various experiments performed are presented in the following.

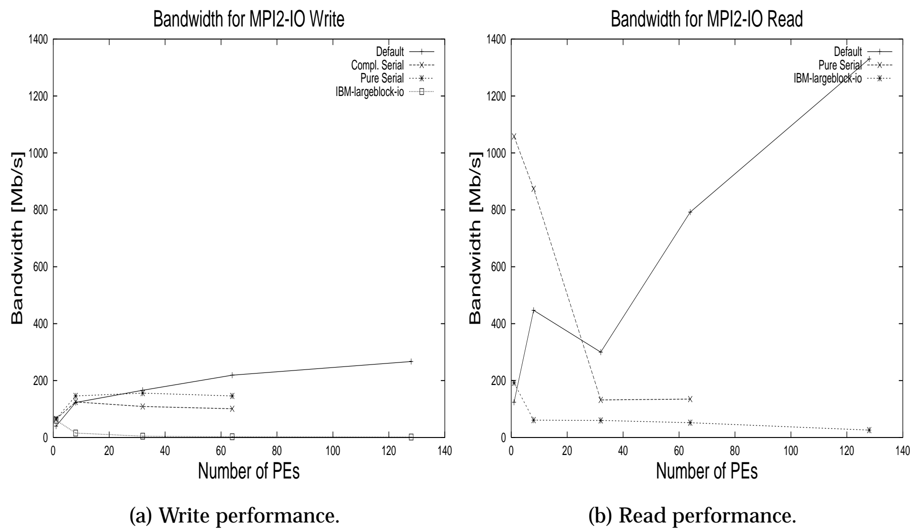
3.1 Serial I/O performance

In our version of the code in which the I/O is performed serially all writing to disk and reading from file is done by a single, master, process within one step. We do not investigate more sophisticated solutions for serial I/O where, for example, attempts are made to achieve decomposition independence. Point–to–point communications, i.e. blocking `MPI_SEND`s and `MPI_RECV`, are used. The local 3D data on each process are sent to the receiving master process from where it is written to file. The data are read back in from file by the master process only. We time the write performance both including and excluding the inter–processor–communications (`MPI_SEND`s and `MPI_RECV`). For the read we only time the process of reading the data back in from file.

The results for serial I/O for both array sizes are presented in Tables 1 and 2 for writing and reading, respectively. For large arrays, only results on one processor can be obtained, since otherwise more than 1 Gb of data has to be stored on one processor to do the I/O. At the moment, there is a memory limit of 900 Mb per process, (7.2 Gb divided by the number of processes per node), hence one PE gets access to the whole memory on its LPAR only when code is only run serially. This is why we only show the scaling for the small array in Fig. 1, where it can be compared directly to the performance for parallel I/O with default settings (as described in Chapter 3.2) for writing (Fig. 1(a)) and reading (Fig. 1(b)). We do not consider solutions in which each processor sends its chunk to the master, the master writes it to disk andafterwards gets the next from the next processor in turn, thus avoiding the 900 Mb/PE limit. Such a solution is only readily applicable to 1D decompositions.

As can be expected, the inter processor communications take some additional time, hence the pure write performance is better than the complete write performance including the inter processor communications.

The results for the serial performance for small arrays can be compared directly to the ones for the parallel I/O by comparing Table 1 with Table 5 (writing) and Table 2 with Table 6 (reading). On one PE the serial write performance is more than 50% better than the parallel one. On 8 PEs the



(a) Write performance.

(b) Read performance.

Figure 1: Performance for a 3D array of 8 Mb/PE on different numbers of processors. The curves represent the median values over 10 runs for the parallel default setting, serial I/O including the time needed for inter processor communications, serial I/O for the pure writing and parallel I/O with $IBM_largeblock_io$.

serial and the parallel bandwidths are identical, while for 64PEs, the parallel bandwidth is twice the serial bandwidth. The read performance for the serial I/O is much better than the parallel one on 1 PE (nearly by a factor of 10). On 8 PEs it is only better by a factor of 2. On 32 PEs MPI–IO is definitely preferable over serial I/O since it is by a factor of two faster, and on 64 PEs it is a factor of six faster.

Comparing the last row in Tables 1 and 2 for the serial I/O with the first row in Tables 3 and 4 for the parallel I/O, clearly shows that, although the serial I/O is slightly better than parallel I/O, the results are rather close to each other and time is no reason to favour serial I/O to parallel I/O even on 1 PE for large arrays of 512 Mb/PE.

One might expect a constant bandwidth for serial I/O. That is not what we find in this study. The numbers for write–performance vary up to a factor of nearly 2 and for read–performance even up to a factor of 8. The read bandwidth remains constant from 32 to 64 PEs. The serial write bandwidth provides a baseline for standard FORTRAN I/O on HPCx. The maximum value at which it saturates is below 200 Mb/s. The read bandwidth clearly shows the effect of the disk cache. This is due to the fact that the read is performed in the same code as the write. Hence, for sufficiently small data sets which still fit into disk cache a bandwidth higher than the expected maximum bandwidth can be achieved. For a larger number of PEs (32 or over) the amount of data exceeds the size of the disk cache, which results in a sudden drop in read bandwidth down to the standard FORTRAN I/O value below 200 Mb/s already observed for writes.

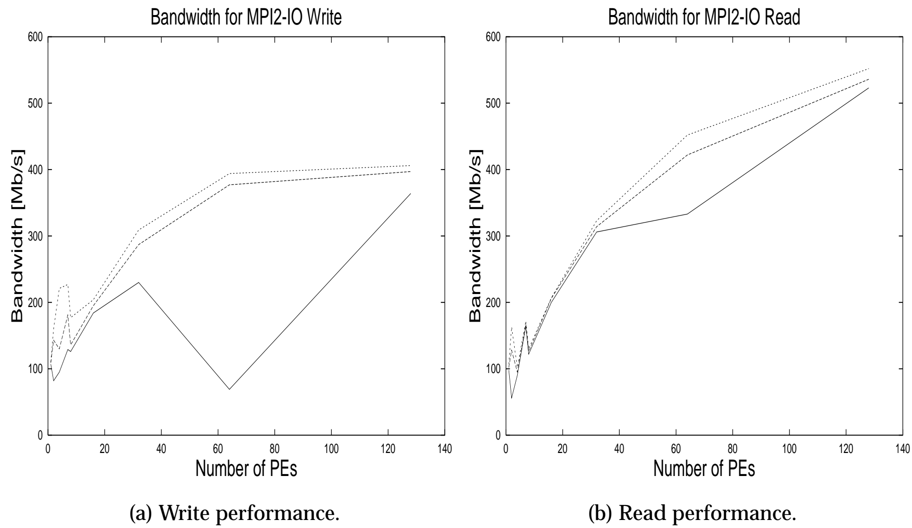
This investigation shows clearly that MPI–IO performs better than serial I/O for large arrays the more PEs are used.

3.2 MPI–IO with default settings

MPI–IO performance for both writing and reading with the default setting for the info–object are investigated first. This means no special hints (MPI_INFO_NULL is used) or directives are set in this case.

To observe the scaling of the code it was run on a various numbers of processors from 1 to 128. The efficiency can be calculated by dividing the parallel bandwidth on a given number of processors by the bandwidth obtained on one processor times the number of processors used. Where not mentioned otherwise, an average over 10 runs is taken for all the tests. The median bandwidth for large arrays is presented in Fig. 2 for writing (a) and reading (b) and in Tables 3 and 4. The Figures show the minimum, maximum and median values of the bandwidth in Mb/s, because of the rather high standard deviation for most runs. The Tables give the numerical values of the arithmetic mean, the standard deviation and the median of the bandwidth, as well as the efficiency. The results for the small array are displayed in Fig. 3 for writing (a) and reading (b) and Tables 5 and 6.

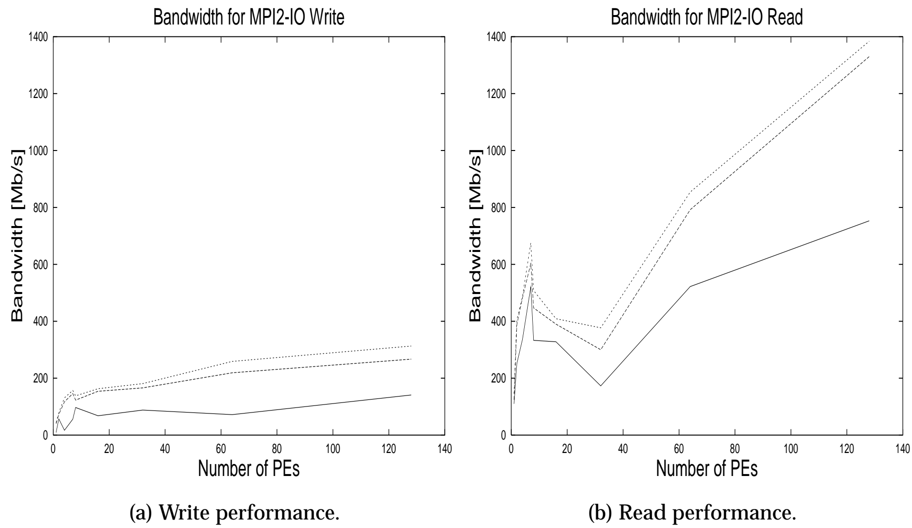
For the large array, writing is as fast or slightly faster than reading on between 1–8 processors, and vice versa on 16–128 processors. For the small array, reading is always faster than writing, by up to a factor of five. It is quite remarkable how high the standard deviation turns out to be for some cases.



(a) Write performance.

(b) Read performance.

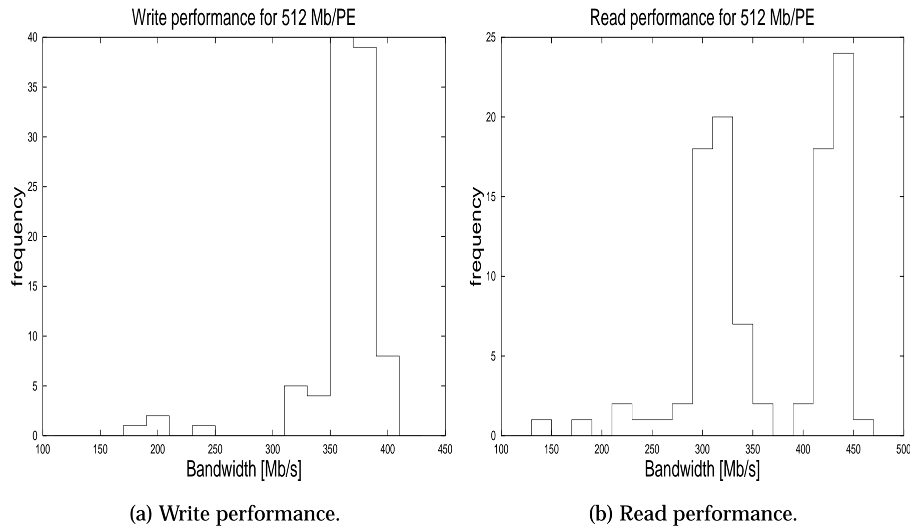
Figure 2: Performance for a 3D array of 512 Mb/PE on different numbers of processors. The curves represent the maximum, median and minimum out of 10 runs.



(a) Write performance.

(b) Read performance.

Figure 3: Performance for a 3D array of 8 Mb/PE on different numbers of processors. The curves represent the maximum, median and minimum out of 10 runs.



(a) Write performance.

(b) Read performance.

Figure 4: Performance for a 3D array of 512 Mb/PE on 64 processors. The histogram shows the bandwidth frequency for in total 100 runs.

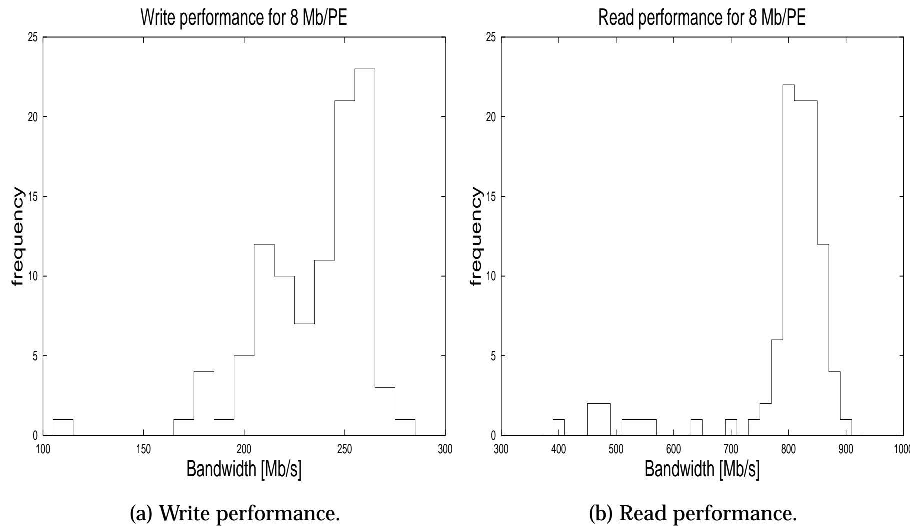
In order to get an impression about how much the time might differ for identical runs due to the usage of the machine by other users or possible system operations, we present the data for the large and the small array sizes on 64 processes averaged over 100 runs. The results for the large array are shown in a histogram in Fig.4 for writing (a) and reading (b), the corresponding numerical values can be found in Table 7. The results for the small array are shown in Fig. 5 for writing (a) and reading (b) and Table 7 as well. For both writes and reads and for small and large array sizes, the results are widely scattered.

The maximum write performance achieved is 400 Mb/s for 512 Mb/PE on 128 PEs and the maximum read performance 1330 Mb/s for 8 Mb/PE on 128 PEs. In both cases this outperforms the serial I/O and allows the writing and reading of bigger array sizes in the right order without using more sophisticated I/O coding techniques.

3.3 `IBM_io_buffer_size` and `MPI_MODE_UNIQUE_OPEN`

The test cases presented in this Chapter include the hint `IBM_io_buffer_size`. We set `IBM_io_buffer_size` to 1 Mb, 16 Mb, 64 Mb and 128 Mb,respectively, where 128 Mb is the maximum possible and 16 Mb is the default on HPCx. We also represent the bandwidths for another set of runs for which the compiler directive `MPI_MODE_UNIQUE_OPEN` is added on top of `IBM_io_buffer_size`. All experiments are performed on 8 and 32 PEs.

The results for large array sizes are shown in Figure 6 for the write (a)



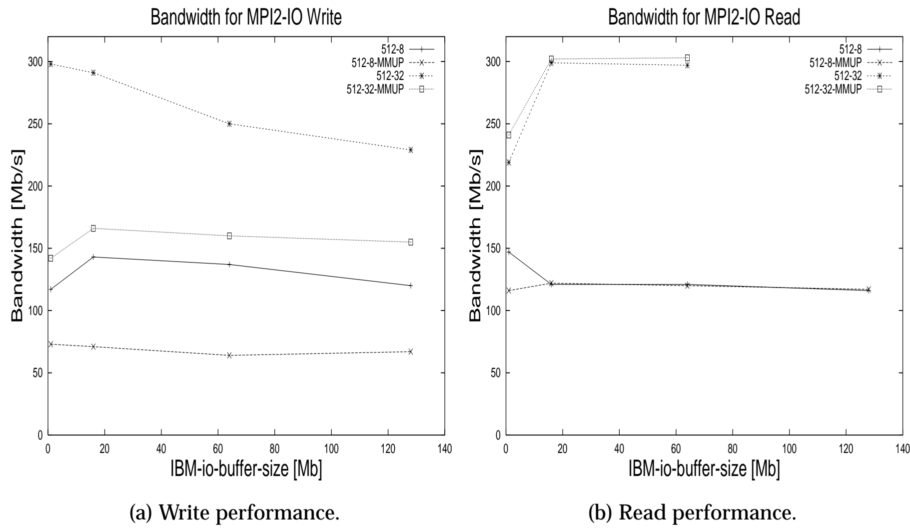
(a) Write performance. (b) Read performance.

Figure 5: Performance for a 3D array of 8 Mb/PE on 64 processors. The histogram shows the bandwidth frequency for in total 100 runs.

and read (b) performance on 8 and 32 processors both with and without including the directive `MPI_MODE_UNIQUE_OPEN`. The numerical values can be found in Tables 8 and 9 for the writing and reading, respectively.

The results for small array sizes are shown in Figure 7 for the write (a) and read (b) performance on 8 and 32 processors both with and without including the directive `MPI_MODE_UNIQUE_OPEN`. Again, the numerical values can be found in Tables 8 and 9 for the writing and reading, respectively.

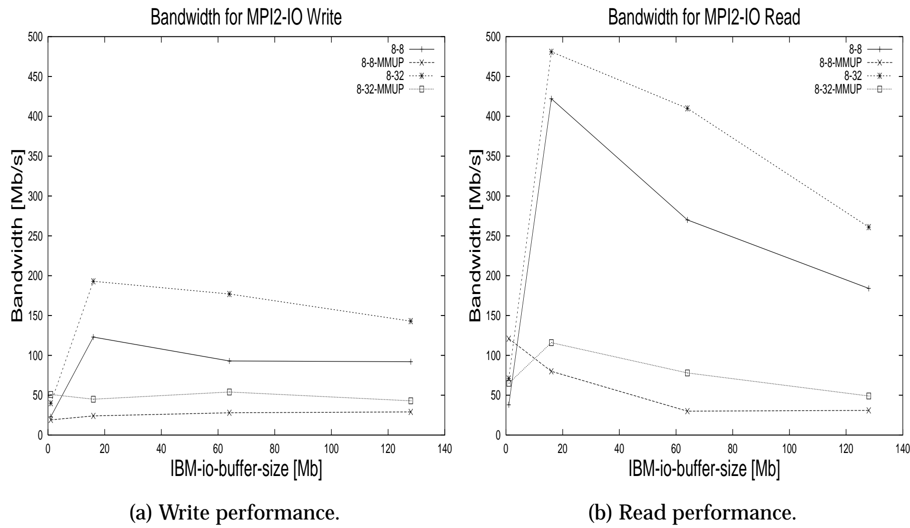
All results for `IBM_io_buffer_size=16` Mb should correspond to the default setting and agree with the results obtained in Section 3.2. In fact, they all agree within a few percent except the result for a small array on 32 PEs where the median for the default bandwidth is 300 Mb/s but 481 Mb/s when using the hint `IBM_io_buffer_size`. It can be seen that for large array sizes including the MPI–directive `MPI_MODE_UNIQUE_OPEN` always gives worse or similar (but never better) results for both writing and reading than omitting it. The only exception is the read bandwidth for large arrays on 32 PEs for all values of `IBM_io_buffer_size`. Here, the maximum performance improvement is 10%. The bad effect of this directive is worse for writing than for reading. Omitting the directive speeds up the writing process by a factor of 1.5–2. For large array sizes and `IBM_io_buffer_size>=64` Mb HPCx runs out of memory for reading with MPI–IO on 32 PEs. In general, it is not recommended for large array sizes to use the MPI–directive since it does not improve performance and even decreases performance in most cases. Performance gains, if any, are very small. In most cases the pure default setting gives best performance.



(a) Write performance.

(b) Read performance.

Figure 6: Performance for a 3D array of 512 Mb/PE for four different values of $IBM_io_buffer_size$ and with and without $MPI_MODE_UNIQUE_OPEN$. The curves always show the median out of 10 runs. In the legend the first number denotes the array size in Mb/PE, the second the number of processors used and the extension 'MMUP' shows that $MPI_MODE_UNIQUE_OPEN$ is used.



(a) Write performance.

(b) Read performance.

Figure 7: Performance for a 3D array of 8 Mb/PE for four different values of $IBM_io_buffer_size$ and with and without $MPI_MODE_UNIQUE_OPEN$. The curves always show the median out of 10 runs. In the legend the first number denotes the array size in Mb/PE, the second the number of processors used and the extension 'MMUP' shows that $MPI_MODE_UNIQUE_OPEN$ is used.

In some cases setting IBM_io_buffer_size = 1Mb improves performance compared to the default case. Setting IBM_io_buffer_size > 16Mb does not seem to be useful.

For small arrays the write bandwidth is better the more PEs are used, if the MPI–directive is omitted and the default settings are used. Including the directive reduces even the performance on 32 PEs below the one on 8 PEs without the directive. The only exception is the write bandwidth achieved for IBM_io_buffer_size = 1Mb with the directive used and run on 32 PEs. Again, for small arrays the read performance is better the more PEs are used, if the MPI–directive is omitted and the default settings are used. The only exception is the read bandwidth achieved for IBM_io_buffer_size = 1Mb with the directive used and run on 8 PEs which outperforms all other cases for this hint. Also for small arrays, the usage of the MPI–directive usually decreases performance and the best value for the hint is the default setting.

3.4 MPI–IO with IBM_largeblock_io

The last case investigated includes IBM_largeblock_io as the only info–object. The results for write and read performance for both small and large arrays can be seen in Tables 10 and 11. In Fig. 1 (a) and (b) the write and read performance can be compared directly to the results obtained for parallel default setting I/O and serial I/O.

Using IBM_largeblock_io fails for large array reads (a segmentation fault is produced). For all other cases, the usage of this hint decreases performances considerably when run on more than one processor as can be seen in Tables 10 and 11. While on one processor the bandwidths are always a bit better for all I/O and array sizes, they are not only always below the value without using the hint but even drop the more processors are used.

4 Summary

In this report parallel I/O (MPI-IO) on HPCx is investigated for small and large array sizes, i.e. 8 and 512 Mb/PE, up to 65 Gb in total. For this purpose the influence of IBM–specific hints like IBM_io_buffer_size and IBM_largeblock_io and the compiler directive MPI_MODE_UNIQUE_OPEN on the benchmarks are investigated and compared to the results obtained for a default setting (corresponding to an info–object of MPI_INFO_NULL). We also compare parallel I/O performance to serial I/O performance.

This investigation showed that MPI–IO performs not only equally well but even better than serial I/O when used for arrays of a total size of 256 Mb (or over) on 32 processors or over. The higher the number of processors the more it outperforms serial I/O. Parallel write bandwidth is more then twice as high than the serial one. Parallel read bandwidth is more than 5 times more than serial bandwidth. The baseline figure for standard serial FORTRAN I/O is a maximum bandwidth of less than 200 Mb/s.

Hence, any I/O performance above this value favours MPI–IO. The maximum parallel write bandwidth achieved in this investigation is 400 Mb/s for 512 Mb/PE on 128 PEs. The maximum parallel read bandwidth found is 1330 Mb/s for 8 Mb/PE on 128 PEs. This has to be compared to a maximum serial write performance of 132 Mb/s for 8 MB/PE on 8 PEs and 1058 Mb/s for 8 Mb/PE on one PE.

In addition, using MPI–IO is a rather neat and simple way to deal with a code's I/O. If serial I/O is done in the rather straight–forward approach of having the master processor doing the I/O for all data in one go, serial I/O is not an option on HPCx if the total data size exceeds the memory limit of 900 Mb/PE. We would recommend using MPI–IO instead of serial I/O on HPCx, but without changing the default settings for the MPI hints or directives. Only in exceptional cases does the use of the hints improve the performance at all. In most cases it decreased performance. If a user wants to run a code for a specific number of processors only and a fixed array size, it might be worth experimenting to find the best choice for hints and directives. In most cases it does not seem to be worth the effort.

A Parallel code segments

The major parts of the parallel MPI–IO code look like the following:

```
[...]
! 3D domain decomposition, with xndat = 400, yndat = 400, zndat = 400, ndim = 3
[...]
! Create  datatype to describe internal elements of data, ie core excluding halos
  sizes(1)    = xcells + 2
  sizes(2)    = ycells + 2
  sizes(3)    = zcells + 2
  subsizes(1) = xcells
  subsizes(2) = ycells
  subsizes(3) = zcells
  starts(1)   = 1        ! data without halo begins at data(1,1,1),
  starts(2)   = 1        ! uses C-like format, ie starts from zero
  starts(3)   = 1

  call mpi_type_get_extent(MPI_DOUBLE_PRECISION, lb, dblesize, ierr)
  call mpi_type_create_subarray(ndim, sizes, subsizes, starts, &
                                order, MPI_DOUBLE_PRECISION, coretype, ierr)
  call mpi_type_commit(coretype,ierr)
  call mpi_type_get_extent(coretype, lb, extent, ierr)
[...]
! File is basically made up of reals
  etype = MPI_DOUBLE_PRECISION

! create datatype to describe the section of the global file that
! is owned by this process (the ftype)

  sizes(1)    = xcells*dims(1)
  sizes(2)    = ycells*dims(2)
  sizes(3)    = zcells*dims(3)
  subsizes(1) = xcells
  subsizes(2) = ycells
  subsizes(3) = zcells
  starts(1)   = coords(1)*xcells
  starts(2)   = coords(2)*ycells
  starts(3)   = coords(3)*zcells

  call mpi_type_create_subarray(ndim, sizes, subsizes, starts, &
                                order, MPI_DOUBLE_PRECISION, ftype, ierr)
  call mpi_type_commit(ftype,ierr)

  call mpi_type_get_extent(ftype, lb, extent, ierr)
! open the file which sets up the File Handle fh

  call mpi_info_create(finfo,ierr)
! In this example no MPI-directive
```

```
  call mpi_barrier(comm,ierr)
  mpstarttime = MPI_WTIME()
  call mpi_file_open(comm, 'MPIIOfile.dat', &
                     MPI_MODE_CREATE+MPI_MODE_WRONLY,finfo, fh, ierr)
  disp = 0
  call mpi_file_set_view(fh, disp, etype, ftype, 'native', &
                         finfo, ierr)
  call mpi_file_write_all(fh, data, 1, coretype, status, ierr)
  call mpi_file_close(fh, ierr)
  call mpi_barrier(comm,ierr)
  mpendtime = MPI_WTIME()
[...]
```

B Tables

All benchmarking results are shown in the following tables.

Table 1: Benchmarks for serial write performance for a 3D–array. The average is taken over 10 runs. The bandwidth is in Mb/s. The pure writes only include the writing to disk, the complete writes include the inter processor communication as well.

Type	Array [Mb/PE]	No PEs	Arithmet. mean	Standard deviation	Median
Complete	8	1	60	12	65
Pure	8	1	60	12	65
Complete	8	8	124	32	125
Pure	8	8	148	42	146
Complete	8	32	105	21	109
Pure	8	32	149	35	156
Complete	8	64	98	12	101
Pure	8	64	141	23	146
	512	1	151	17	146

Table 2: Benchmarks for serial read performance for a 3D–array of 8 Mb/processor. The average is taken over 10 runs. The bandwidth is in Mb/s.

Array [Mb/PE]	No PEs	Arithmet. mean	Standard deviation	Median
8	1	1009	180	1058
8	8	876	19	874
8	32	133	6	132
8	64	133	11	135
512	1	155	31	154

Table 3: Benchmarks for parallel write performance for a 3D–array of 512 Mb/processor. The average is taken over 10 runs. No special directives or hints used. The bandwidth is in Mb/s.

No PEs	Arithmet. mean	Standard deviation	Median	Efficiency
1	109	1	110	1.00
2	143	27	154	0.70
4	130	47	97	0.23
7	181	40	184	0.24
8	136	15	133	0.15
1 task/8 nodes	398	12	401	0.45
16	194	6	195	0.11
32	287	21	291	0.08
64	343	99	377	0.05
128	397	12	400	0.03

Table 4: Benchmarks for parallel read performance for a 3D–array of 512 Mb/processor. The average is taken over 10 runs. No special directives or hints used. The bandwidth is in Mb/s.

No PEs	Arithmet. mean	Standard deviation	Median	Efficiency
1	104	4	103	1.00
2	128	39	139	0.65
4	95	4	94	0.23
7	168	3	167	0.23
8	126	3	126	0.15
1 task/8 nodes	459	25	472	0.58
16	206	3	207	0.13
32	314	5	314	0.09
64	418	32	424	0.06
128	536	11	537	0.04

References

[1] Prost J.–P., Treumann R, et al.. Towards a High–Performance and Robust Implementation of MPI–IO on top of GPFS
EuroPar2000, Munich, August 2000,in A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1253–1262, 2000. (Springer-Verlag: Berlin).
http://www.llnl.gov/icc/lc/siop/papers/MPIIO_GPFS.pdf

[2] MPI Subroutine Reference
http://hpcf.nersc.gov/vendor_docs/ibm/pe/am107mst118.html

[3] Dick Treumann. Making Effective use of MPI–IO, SP Scicomp-5
http://www.spscicomp.org/ScicomP5/Presentations/Treumann/Daresbury.MPI-IO.pdf

Table 5: Benchmarks for parallel write performance for a 3D–array of 8 Mb/processor. The average is taken over 10 runs. No special directives or hints used. The bandwidth is in Mb/s.

No PEs	Arithmet. mean	Standard deviation	Median	Efficiency
1	37	10	40	1.00
2	72	7	73	0.90
4	107	32	117	0.73
7	139	29	146	0.53
8	122	12	123	0.39
16	146	28	154	0.24
32	156	30	166	0.13
64	208	53	219	0.09
128	263	50	267	0.05

Table 6: Benchmarks for parallel read performance for a 3D–array of 8 Mb/processor. The average is taken over 10 runs. No special directives or hints used. The bandwidth is in Mb/s.

No PEs	Arithmet. mean	Standard deviation	Median	Efficiency
1	126	11	124	1.00
2	351	62	387	1.55
4	457	57	482	0.98
7	592	51	603	0.70
8	426	57	447	0.45
16	382	25	390	0.19
32	285	62	300	0.08
64	781	24	792	0.10
128	1236	1243	1330	0.08

Table 7: Benchmarks for write and read performance for a 3D–array of 8 Mb/PE and 512 Mb/PE. Here, the average is taken over 100 runs on 64 processors. We calculate the arithmetic mean, the standard deviation and the median of the bandwidth in Mb/s.

Array [Mb/PE]	Task	Arithmet. mean	Standard deviation	Median
8	Write	230	27	239
8	Read	783	99	809
512	Write	351	69	324
512	Read	371	12	372

Table 8: Benchmarks for parallel write performance for a 3D–array. The average is taken over 10 runs. For all runs $IBM_io_buffer_size$ is set to the value given in the table. If $MPI_MODE_UNIQUE_OPEN$ is also used it is denoted by 'MMUP'. The bandwidth is in Mb/s.

$IBM_io_buffer_size$ [Mb]	Array [Mb/PE]	No PEs	Arithmet. mean	Standard deviation	Median
128	512	8	122	9	120
MMUP 128	512	8	72	18	67
64	512	8	145	20	137
MMUP 64	512	8	70	15	64
16	512	8	153	26	143
MMUP 16	512	8	72	12	71
1	512	8	117	7	117
MMUP 1	512	8	82	25	73
128	512	32	220	30	229
MMUP 128	512	32	154	13	155
64	512	32	253	12	250
MMUP 64	512	32	157	11	160
16	512	32	291	5	291
MMUP 16	512	32	170	14	166
1	512	32	274	79	298
MMUP 1	512	32	141	8	142
128	8	8	84	25	92
MMUP 128	8	8	28	6	29
64	8	8	92	3	93
MMUP 64	8	8	26	5	28
16	8	8	125	9	123
MMUP 16	8	8	25	3	24
1	8	8	25	10	23
MMUP 1	8	8	19	2	19
128	8	32	133	32	143
MMUP 128	8	32	43	3	43
64	8	32	165	39	177
MMUP 64	8	32	53	8	54
16	8	32	184	20	193
MMUP 16	8	32	45	5	45
1	8	32	39	4	40
MMUP 1	8	32	49	9	51

Table 9: Benchmarks for parallel read performance for a 3D–array. The average is taken over 10 runs. For all runs $IBM_io_buffer_size$ is set to the value given in the table. If $MPI_MODE_UNIQUE_OPEN$ is also used it is denoted by 'MMUP'. The bandwidth is in Mb/s.

$IBM_io_buffer_size$ [Mb]	Array [Mb/PE]	No PEs	Arithmet. mean	Standard deviation	Median
128	512	8	117	6	116
MMUP 128	512	8	116	3	117
64	512	8	121	3	121
MMUP 64	512	8	120	2	120
16	512	8	120	2	121
MMUP 16	512	8	119	9	122
1	512	8	146	10	147
MMUP 1	512	8	114	4	116
128	512	32	—	—	—
MMUP 128	512	32	—	—	—
64	512	32	297	6	297
MMUP 64	512	32	303	4	303
16	512	32	299	7	299
MMUP 16	512	32	302	6	302
1	512	32	217	14	219
MMUP 1	512	32	240	10	241
128	8	8	213	48	184
MMUP 128	8	8	30	3	31
64	8	8	257	47	270
MMUP 64	8	8	28	5	30
16	8	8	408	60	422
MMUP 16	8	8	77	13	80
1	8	8	41	15	38
MMUP 1	8	8	97	46	121
128	8	32	257	11	261
MMUP 128	8	32	47	5	49
64	8	32	407	31	410
MMUP 64	8	32	77	11	78
16	8	32	483	23	481
MMUP 16	8	32	122	22	116
1	8	32	73	20	71
MMUP 1	8	32	68	20	65

Table 10: Benchmarks for parallel write performance for a 3D–array. The average is taken over 10 runs. Only on 128 PEs the job had to be split into two different batch jobs. For all runs $IBM_largeblock_io$ is set to be true. The bandwidth is in Mb/s.

Array [Mb/PE]	No PEs	Arithmet. mean	Standard deviation	Median
8	1	56.3	12.8	60.9
8	8	15.5	1.0	15.5
8	32	4.5	0.5	4.5
8	64	2.7	0.1	2.7
8	128	1.6	0.0	1.6
512	1	156.4	30.0	163.5
512	8	47.5	1.5	48.0
512	32	30.3	2.1	30.5
512	64	23.5	2.0	23.2
512	128	15.9	4.5	16.2

Table 11: Benchmarks for parallel read performance for a 3D–array. The average is taken over 10 runs. For all runs $IBM_largeblock_io$ is set to be true. The bandwidth is in Mb/s.

Array [Mb/PE]	No PEs	Arithmet. mean	Standard deviation	Median
8	1	188.6	44.1	191.8
8	8	60.7	0.8	60.9
8	32	57.1	7.3	59.5
8	64	52.0	1.4	51.9
8	128	25.6	0.5	25.6