Comput Sci Res Dev (2011) 26: 275–283
DOI 10.1007/s00450-011-0154-4

SPECIAL ISSUE PAPER

A system level view of Petascale I/O on IBM Blue Gene/P

Wolfgang Frings · Michael Hennecke

Published online: 8 April 2011

Abstract Petascale supercomputers rely on highly efficient Petascale I/O subsystems. This work describes the tuning and scaling behavior of the GPFS parallel file system on JUGENE, the largest IBM Blue Gene/P installation worldwide and the first PetaFlop/s HPC resource within the European PRACE Research Infrastructure.

Keywords Parallel I/O · Blue Gene · GPFS

1 Introduction and background

The first Petascale supercomputers have now been in production for a while. The Jülich Supercomputing Centre operates the largest IBM* System Blue Gene*/P [1] supercomputer worldwide, nicknamed "JUGENE". It has a peak performance of 1 PetaFlop/s, 288k compute cores, and 144 TiB of main memory. Real world applications from a broad range of science areas have been scaled to the full size of this capability system, as demonstrated by the annual JSC extreme scaling workshops [5, 6].

This work was supported by the FZJ/IBM Exascale Innovation Center, EIC cooperation agreement T/Z1213.02.09.
*IBM, Blue Gene and GPFS are trademarks of IBM in USA and/or other countries. Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

W. Frings (✉)
Forschungszentrum Jülich GmbH (FZJ),
Wilhelm-Johnen-Strasse, 52425 Jülich, Germany
e-mail: w.frings@fz-juelich.de

M. Hennecke
IBM Deutschland GmbH, Karl-Arnold-Platz 1a,
40474 Düsseldorf, Germany
e-mail: hennecke@de.ibm.com

JUGENE is supported by a storage cluster based on the IBM General Parallel File System (GPFS*) [7], with an aggregate bandwidth of 66 GB/s and more than 4 PB of usable capacity [8]. The Blue Gene/P architecture employs an hierarchical I/O model, in which the compute nodes are connected to the external storage cluster by a much smaller number of Blue Gene/P I/O nodes (IONs), using a 10 GbE TCP/IP network. The JUGENE machine is configured with 8 IONs per rack, or 576 IONs in the full system partition of 288k cores.

This article describes our experiences with Petascale I/O in this production environment, in particular the effects of the hierarchical I/O model and the unprecedented scalability in terms of number of MPI tasks. Our work is focusing on tuning the IBM supported Blue Gene/P and GPFS software products which are used in FZJ's operational environment. See [9] for a discussion of enhancements of Blue Gene/P I/O forwarding within a research software environment.

A parallel file system like GPFS is the minimum software layer between the application and the storage hardware. It provides a file system layer on top of the underlying storage hardware, offers a POSIX API, and provides large aggregate bandwidth for large, contiguous I/O by striping across many disks, servers, and network links. Beyond the file system layer, the following areas are important:

1.1 High-level I/O libraries

From the application perspective, sequential as well as parallel I/O requires a mapping of the application specific data structures to the file system level [10]. High-level I/O libraries like HDF5 and NetCDF provide a self-describing, portable data format. Their APIs are much richer than POSIX, in particular allowing the expression of non-contiguous I/O and collective I/O within their respective

data formats. Parallelism is supported through PnetCDF [11], PHDF5 [12], and netCDF-4 [13] which in turn are based on MPI-IO [14] as an intermediate layer. Eventually, this software stack issues POSIX I/O requests against the parallel file system. From a performance viewpoint, the I/O optimizations in the MPI-IO layer are utilizing the knowledge conveyed through its richer API, and transform the application I/O requests into a form which is better suited for the underlying POSIX API and parallel file system characteristics: *fewer* and *larger* I/O requests than what is expressed at the higher abstraction levels. In this work we only focus on I/O performance at the POSIX level as the common low-level API.

1.2 File organization: task-local versus shared files

Application programmers will generally approach parallel I/O from either of two sides: If the applications already use the sequential version of a high-level I/O library, parallelism is introduced through the parallel versions of those libraries, with MPI-IO as the intermediate layer and parallel I/O to one shared file. If the original application is based on sequential POSIX I/O, parallelization naturally evolves through several stages:

- One task performs all I/O.
  This is trivial to implement, but obviously does not scale. On Blue Gene this mode is not realistic, as a single ION does not have the network bandwidth, memory capacity and CPU power to perform all the I/O for thousands of MPI tasks.
- Each MPI task accesses its own file.
  This is an obvious match for the domain decomposition used in many distributed memory applications. The I/O is embarrassingly parallel, so the bandwidth should scale perfectly with the number of nodes. The most obvious disadvantage at Petascale is the excessive number of files: This causes both manageability issues (a single run may generate millions of files), as well as metadata performance issues when creating the files. Other disadvantages are the fact that the file layout is for a fixed number of tasks (so jobs of other size or post-processing requires a reorganization of the files), and that task-local files prevent any kind of optimization across task boundaries.
- All MPI tasks access one shared file.
  This is easiest to manage, and using one file also allows to maintain the same data format as the serial program. For a fully POSIX compliant file system like GPFS, it is safe to perform shared file I/O using the POSIX API. Otherwise, another API like MPI-IO may be required which can guarantee consistency and atomicity at a higher layer.

  Note that shared file *write* operations require coordination at the metadata level as inode and indirect block updates have to be synchronized. At Petascale, this may reveal effects which are neglectable at smaller scales.

As long as the I/O pattern is predominantly contiguous large block I/O, parallelizing POSIX I/O as described above will generally produce good performance with relatively low porting effort. When small, non-contiguous or collective I/O dominates, employing a higher-level API will be beneficial.

1.3 Avoiding overload

Another important aspect of parallel I/O is that overloading the storage infrastructure should be avoided. It is very common that HPC clusters can generate I/O at a much higher rate than what the storage can consume. From a scaling perspective, using more compute nodes than what is required to saturate the file system should not incur large performance degradations. On Blue Gene/P with its hierarchical I/O model, performance scaling on the ION level is also important: How many compute tasks per ION are needed to saturate the ION, and how big is the performance degradation when increasing the number of compute tasks per ION beyond that point?

In the following sections we will investigate these questions in detail, focusing on the POSIX I/O level. In addition, FZJ has developed SIONlib [15] to address some of the limitations mentioned above. SIONlib is a lightweight I/O library which maintains the POSIX API, but under the covers enables several optimizations including the aggregation of a logical view of task-local files into one file per ION, and control over how many MPI tasks per ION perform the actual I/O. SIONlib is described elsewhere [16], but as it is used in our performance studies we briefly summarize it in Sect. 5.

2 Maximizing storage subsystem bandwidth

Storage subsystem bandwidth is increasing from generation to generation, while individual disk drive performance is no longer improving significantly. Especially with nearline disk drives, sustaining the higher per-controller bandwidth of recent storage subsystems requires larger and larger GPFS block sizes. Assuming that sufficiently many disk drives are populated, the sustained sequential I/O bandwidth that can be achieved with a particular disk storage subsystem depends on the proper alignment of three main size parameters:

1. the GPFS file system block size,
2. the RAID stripe size, and
3. the application I/O size.

It is a general performance best practice to align the RAID stripe size and application I/O size with the file system block size. Otherwise, a read-modify-write penalty is

Table 1 Bandwidth dependency on GPFS block size

GPFS block size (MiB)	Read bandwidth (GB/s)	Write bandwidth (GB/s)
1	4.0–5.3	1.2–4.0
2	6.1–7.4	2.0–5.7
4	8.6–9.3	3.4–8.2

incurred when an application write is performed which is smaller than the file system block size, or when a GPFS block is written to disk which is smaller than the LUN's RAID full stripe size.

One building block of our GPFS storage cluster [8] contains a pair of IBM DS5300 disk storage subsystems with RAID6 arrays (8+2P) on SATA disks and has a hardware read bandwidth of 9.6 GB/s, with a slightly lower write bandwidth. Table 1 shows the sustainable read and write bandwidth of one building block, for GPFS block sizes 1 MiB to 4 MiB (measured on the GPFS servers using the `gpfsperf` [18] benchmark). The indicated bandwidth ranges show the effect of varying the application I/O size and RAID stripe size from 1 MiB to 4 MiB for a given GPFS block size. The highest bandwidth is only achieved with 4 MiB GPFS block size and when all of the above-mentioned performance best practices are followed.

Over the years, the central GPFS storage cluster at FZJ has been migrated through several generations of disk storage hardware, while keeping the GPFS file systems online. The GPFS scratch file system $WORK has been originally created with a 2 MiB block size, which was sufficient to saturate the less powerful storage subsystems at that time. To benefit from the 25% (read) to 40% (write) performance increase on DS5300 hardware when going from a 2 MiB GPFS block size to 4 MiB, the $WORK file system had to be recreated at 4 MiB block size (the GPFS block size cannot be changed once the file system has been created). This was the only tuning activity in the context of the JUGENE Petascale upgrade which was not transparent to the users. Doubling the file system blocksize directly affects tuning on the Blue Gene/P side, as will become obvious in Sect. 3.

3 Tuning the I/O data path on Blue Gene

In HPC clusters, GPFS is usually set up using Network Shared Disk (NSD) servers. Disks are attached to the NSD servers (with redundant connections within a building block), and are accessed by the NSD clients through a TCP/IP network as shown in Fig. 1. The user applications are executed on compute nodes, which are also running the operating system as well as the GPFS and NSD client code.

General GPFS tuning advice can be found in the GPFS manuals [17, 18] and in the HPC Central Wiki [20]. The

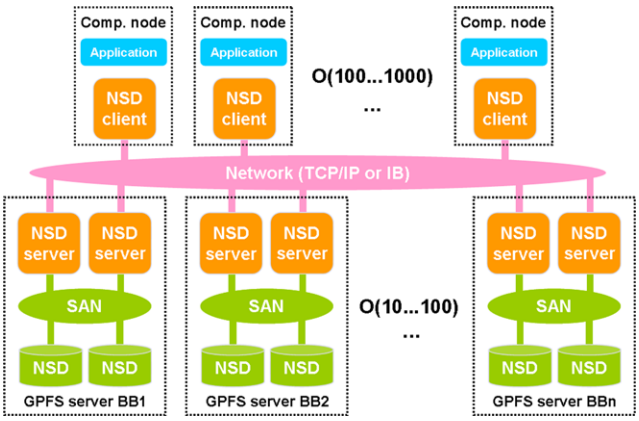

Fig. 1 General GPFS cluster architecture

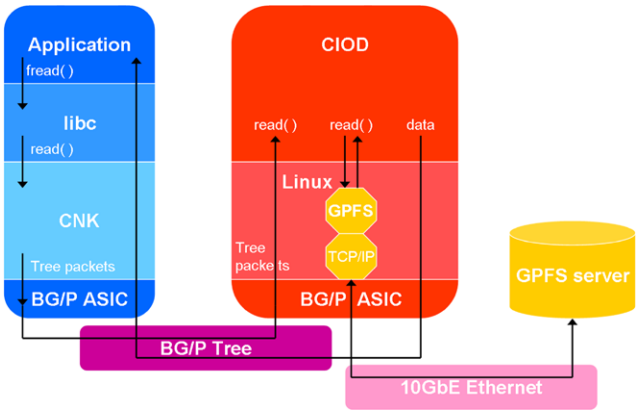

Fig. 2 I/O function shipping on Blue Gene/P

guiding principle is that the whole data path from the application to the disk needs to be optimized to avoid performance bottlenecks. Here we focus on the additional aspects introduced by the Blue Gene I/O architecture.

3.1 GPFS architecture on Blue Gene

The Blue Gene/P system design [1] is very similar to Blue Gene/L [2–4]. Blue Gene compute nodes (CNs) run the user application on a lightweight compute node kernel (CNK), and do *not* have direct TCP/IP connection to an external network. As shown in Fig. 2, all application I/O function calls are *function shipped* to an intermediate layer of Blue Gene I/O nodes (IONs), using the Blue Gene collective (or tree) network. The IONs *do* have external TCP/IP connectivity and run a full Linux* kernel. The control and I/O daemon (CIOD) on the ION is responsible to process the I/O requests which are function shipped from the CNs, and passes them on to the file system layer on the ION. GPFS on Blue Gene uses the GPFS multi-cluster functionality [19] to provide access from the disk-less Blue Gene IONs to an external GPFS storage cluster [22].

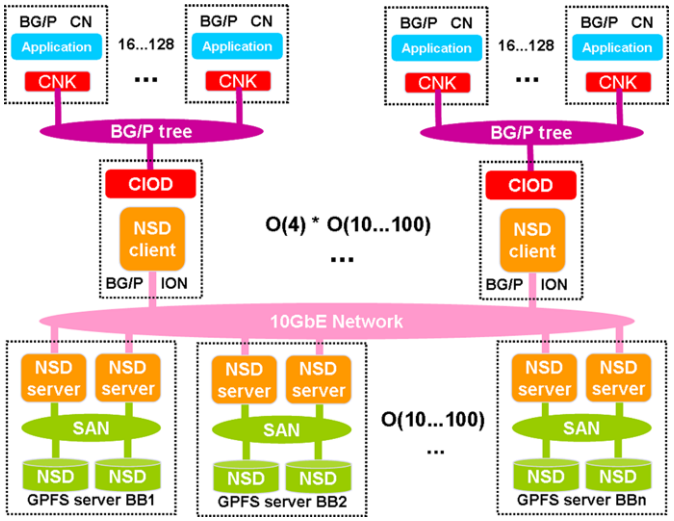

Fig. 3 GPFS architecture for Blue Gene/P

Figure 3 outlines the GPFS architecture on Blue Gene, including the ION layer. The GPFS and NSD client code runs on the IONs, not on the compute nodes. Blue Gene/P supports 8, 16, 32 or 64 IONs per rack with 1024 compute nodes, with each ION providing one 10GbE port. Each compute node can run one MPI task with up to 4 threads (SMP mode) to four single-threaded MPI tasks (virtual node mode or VN mode), so the ratio of MPI tasks per ION can range from 16–64 (for 64 IONs per rack) to as much as 128–512 (for 8 IONs per rack).

In Sect. 3.2 we will show that the Blue Gene/P IONs *cannot* drive file system traffic at 10GbE line rate: The best achievable bandwidth per ION is roughly 450 MB/s for read and 350 MB/s for write in SMP mode, and about 50 MB/s less for VN mode. So in order to consume the GPFS bandwidth provided by a number N of external GPFS servers (which usually *can* drive 10 GbE at line rate), it is necessary to use about $3N$ IONs on the Blue Gene side.

The 10 GbE network ports to connect the IONs are a significant cost factor. Especially in large Blue Gene/P installations, it is therefore typical to only use the minimum of 8 IONs/rack, which will often still result in vastly more ION network bandwidth than the available disk bandwidth. For example, the full-system JUGENE partition with 72 racks has 576 IONs with a peak TCP/IP bandwidth of 200–260 GB/s, while the storage cluster can provide only around 66 GB/s.

3.2 GPFS performance tuning on Blue Gene

On Blue Gene, the key difference to a general HPC cluster is the fact that a single GPFS daemon on an ION needs to concurrently serve a much larger number of active I/O streams (512 for VN mode and 8 IONs per rack). Our tuning efforts targeted some resource limitations of the IONs caused by the fact that the IONs are running a 32-bit Linux kernel, and have a limited memory capacity (Blue Gene/P nodes are available with 2 GiB or 4 GiB of memory). This affects the CIOD function shipping infrastructure, and the GPFS daemon itself.

3.2.1 *CIOD tuning on the IONs*

The main CIOD tunable is the size of the buffers which are allocated on the IONs to hold the I/O which is function shipped from the compute nodes, CIOD_RDWR_BUFFER_SIZE [21]. Following GPFS best practices, these buffers should have the same size as the GPFS file system block size to avoid fragmentation. CIOD always allocates four buffers per compute node to be able to handle Blue Gene/P VN mode jobs with four MPI tasks per compute node. If an application I/O request is larger than the CIOD buffer size, it is fragmented and the fragments are function shipped to the ION where they are passed on to the file system (without re-assembling the fragments, as there isn't enough memory to do so).

Figure 4 shows the aggregate bandwidth of a single ION as a function of the number of MPI tasks performing I/O through this ION. The data is from an IOR [23, 24] benchmark in a GPFS file system with 4 MiB block size, using POSIX individual file I/O with an application I/O size of 4 MiB.

Initially, the memory layout of the 32-bit Linux kernel on the IONs prevented the allocation of 4 MiB CIOD buffers with 8 IONs/rack. Figure 4 clearly shows the strong effect of a too small CIOD buffer size on the achievable bandwidth. Changing the memory layout of the IONs' Linux kernel to support 512 CIOD buffers of size 4 MiB was key to achieve the full GPFS bandwidth on file systems with 4 MiB block size.

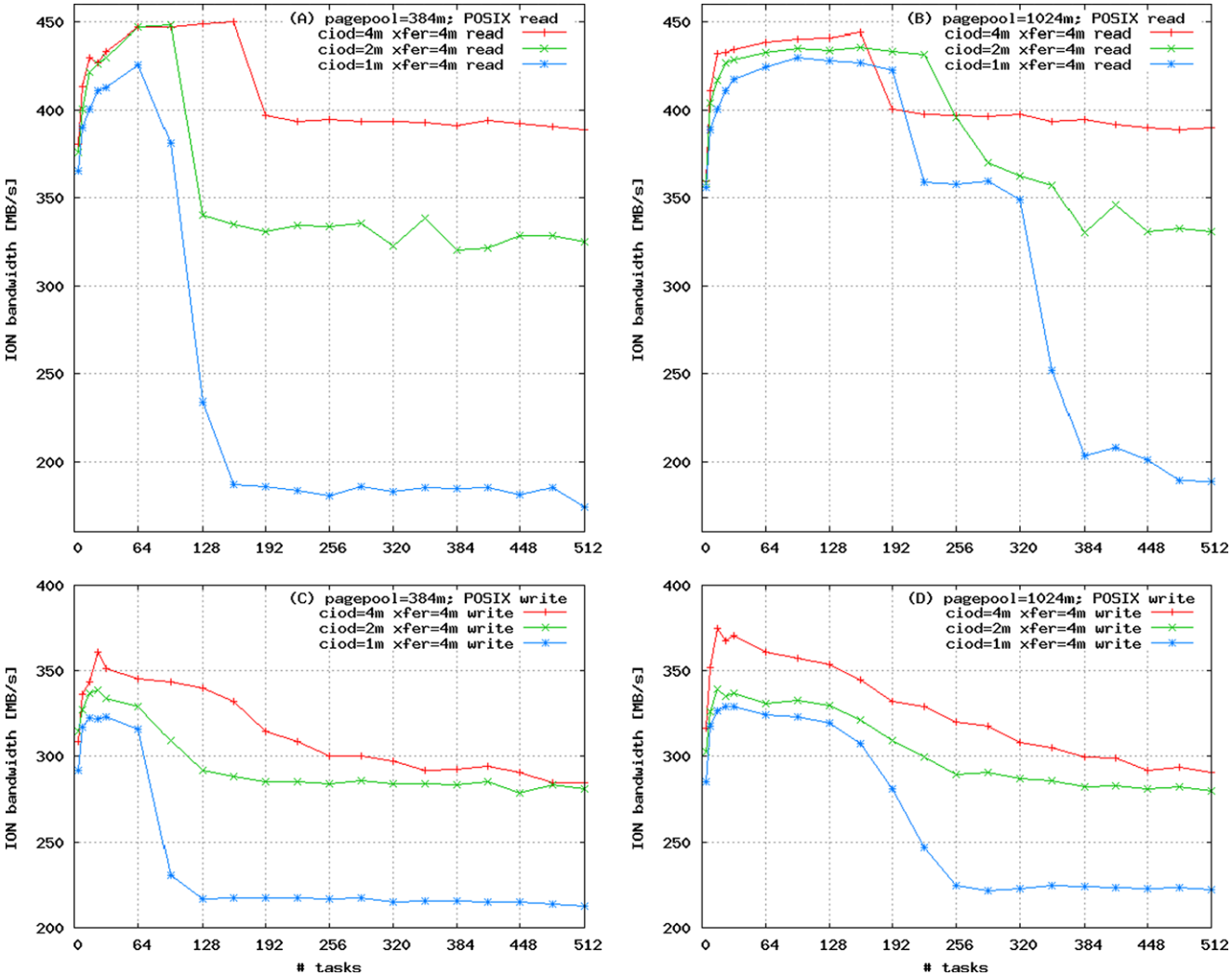

Fig. 4 Single ION bandwidth as a function of the number of MPI tasks per ION. IOR POSIX individual file I/O, 4 MiB GPFS block size, 4 MiB I/O size, varying GPFS pagepool and CIOD buffer size

3.2.2 GPFS tuning on the IONs

GPFS on the IONs needs sufficiently many buffers in the GPFS *pagepool* [17] to support a large streaming I/O bandwidth. Originally, GPFS on Blue Gene/P supported a maximum pagepool of 384 MiB. This translates to only 96 buffers of size 4 MiB, whereas at 8 IONs/rack each ION needs to support between 128 (SMP mode) and 512 (VN mode) active I/O streams. With an optimized memory layout, GPFS on Blue Gene/P now supports a pagepool of up to 1024 MiB (or up to 256 buffers of size 4 MiB). While it would be beneficial to increase this even further, on Blue Gene/P the total ION memory is only 4 GiB (of which 2 GiB are already consumed by CIOD buffers when using 8 IONs/rack and 4 MiB block size), and some non-pinned memory is needed as well, so there is little room to expand the pagepool beyond 1 GiB.

In Fig. 4, comparing (A) and (C) with (B) and (D) shows the performance impact of increasing the pagepool size from 384 MiB to 1024 MiB. When CIOD buffers of size 4 MiB are used, the performance is roughly identical. But when the CIOD buffer size is smaller than the GPFS block size, the larger pagepool shifts the onset of performance degradations to the region of larger numbers of MPI tasks per ION. This is the expected behavior, as a larger pagepool allows to cache more of the file system blocks that have only been partially read or written yet (caused by the fragmentation of the I/O requests at the CIOD layer).

When the CIOD buffer size is set to the optimal 4 MiB but the application transfer size is reduced from 4 MiB to 2 MiB or 1 MiB, the resulting bandwidth plots are almost identical to Fig. 4 (with *ciod* and *xfer* reversed). So even when the CIOD buffer size is identical to the file system block size, a large pagepool is important to sustain higher

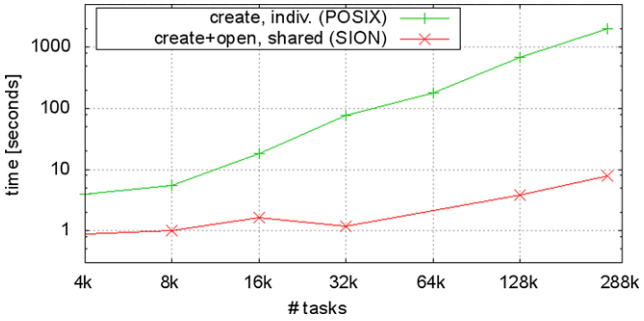

Fig. 5 Performance of parallel file creation: One file per MPI task (POSIX) versus one file per ION (SION)

bandwidth for I/O requests which are smaller than the file system blocksize.

4 Impediments to scaling

Ideally, the I/O bandwidth of a Blue Gene/P partition containing multiple I/O nodes should scale out linearly with the number of IONs, up to the limits imposed by the aggregate performance of the provisioned storage hardware. While this is a reasonable assumption for raw bandwidth scaling, in Sect. 1 we have pointed out two potential bottlenecks: performance of parallel file creates and shared-file write performance.

4.1 Performance of parallel file creates

Besides the manageability issues of handling millions of files, one additional issue with POSIX individual file I/O is the overhead during file creation. Figure 5 compares the time to create and open files in the same directory. For a 64-rack partition on JUGENE, the parallel creation of 256k individual files took approximately 33 minutes whereas creating and opening 512 shared SION files (one shared file per ION) took less than 10 seconds. This effect of metadata contention during file creation is caused by the parallel access to the directory: 256k directory entries need to be added, which link the files' names to their inode numbers. Although GPFS optimizes the parallel access by *fine grain directory locking* (FGDL) which only locks individual directory blocks rather than the whole directory for write updates, the time for file creation is not negligible if the application uses more than a few thousand tasks. The most likely bottleneck in this scenario is contention on the directory's *metanode*. For each open file or directory, GPFS assigns one node to coordinate all metadata updates. Parallel file creates of this magnitude may overwhelm the metanode (which will usually be the GPFS client which first opened the directory).

The workaround for this performance issue is to precreate one subdirectory per MPI task, and create the files

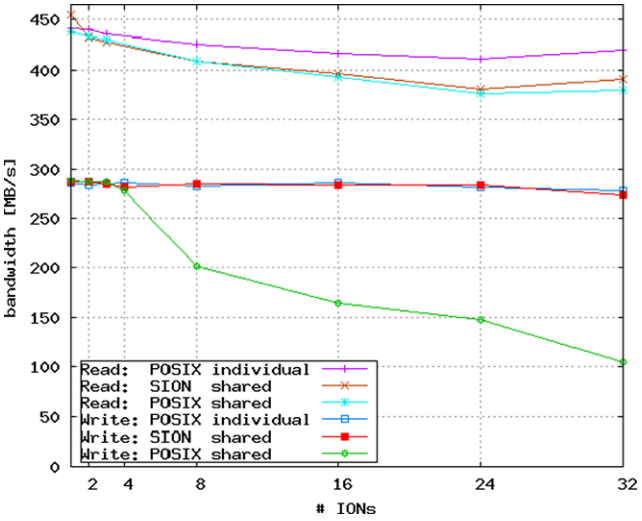

Fig. 6 Bandwidth per ION when scaling the number of IONs (VN mode). One file per task (POSIX individual), one file per ION (SION shared), and one shared file (POSIX shared)

in those per-task subdirectories. But at Petascale, this technique only multiplies the manageability issues of using task-local files as creating one subdirectory per task doubles the number of files/inodes.

4.2 Shared file write performance

For parallel I/O to individual files, there is no dependency between the tasks after the initial parallel file create, therefore we expect no decay of the I/O bandwidth per ION when increasing the number of IONs. The "POSIX individual" measurements in Fig. 6 confirm this expectation.

In contrast, parallel I/O to a shared file uses a common file as a resource accessed concurrently by all tasks. The "POSIX shared" measurements in Fig. 6 show a pronounced bandwidth degradation when writing from several IONs into one shared file. The reason for this degradation is the file metadata management. While block allocation happens in parallel, it is necessary to store the references to new file blocks in the file's inode structure (inode and associated indirect blocks). These updates are managed by the per-file *metanode* (usually the first GPFS client opening the file), and at Petascale this apparently causes contention.

On a 32-bit Linux kernel there is a maximum of 164 GPFS threads to perform both sequential/prefetching I/O (`prefetchThreads`) and random I/O (`worker1Threads`). Changing the default setup to allocate more threads to random I/O slightly reduces the amount of shared file write degradation. Other effects are still under investigation.

In the meantime, the FZJ library SIONlib can be used to circumvent this degradation. SIONlib uses a transparent optimization which maps all task-local files which use the

same ION into a single shared SION file. At 8 IONs per rack and VN mode, 512 MPI tasks performing task-local I/O will actually use one shared file, and only one ION will perform I/O against this file. This case of one physical file per ION is situated between the two extremes of one shared file and one file per MPI task, and combines the advantages of both: Parallel file create time is no concern (only 576 files are created for 288k tasks), and the "SION shared" measurements in Fig. 6 show that the write bandwidth on multiple IONs is comparable to individual files.

5 The SIONlib library

The objective of SIONlib [16] is to make massively parallel I/O to task-local files such as checkpoints, scratch files, or log files more efficient. Situated as an additional software layer between a parallel application and the underlying parallel file system, the main idea of SIONlib is to map a large collection of logical task-local files onto physical shared files. As already shown in previous sections this avoids metadata contention during file creation and simplifies file management operations. SIONlib can be thought of as a very simple application-level file system with an API and command-line utilities to access individual logical files. The programming interface of SIONlib is laid out as an extension of the POSIX or ANSIC I/O interface, requiring only very few source code changes for applications already using these APIs. Existing ANSI C or POSIX read and write calls can be retained. To allow parallel codes written in Fortran to take advantage of the SIONlib library, a Fortran language binding is supplied in addition to the C API. Although by design not tied to a specific parallel programming interface, the current version of SIONlib supports MPI, OpenMP and hybrid MPI + OpenMP programs. The internal metadata exchange, needed to collect and write information about the size and location of data in the shared files, is done via the same parallel interface. Optimizations implemented in SIONlib to support fast parallel I/O to task local files from large number of tasks are the automatic block alignment, the handling of extension, which allows to store more than one chunk per task in a shared SION file, the transparent handling of shared files which are split into a couple of physical files, and the support of collective I/O operations.

At open-time, SIONlib uses the information about the size of data which will be written by one task to optimize the access to the shared file. For example SIONlib automatically aligns boundaries between sections written by different tasks to the boundaries of the file system blocks. This guarantees that a simultaneous parallel access to the same file system block from different tasks can never happen. Bandwidth degradation could also be seen if chunks are separated to different file system blocks but the task I/O is not aligned, writing for example some bytes of data at the end of one file system block and another part at the beginning of the next file system block. To optimize such aligned write operations, the newest version of SIONlib provides internal buffering—similar to the ANSI C I/O functions—in a buffer which has the same size as the file system block. Furthermore, the library provides collective write and read functions to support applications writing only small chunks of data. To allow a broad range of applications to take advantage of SIONlib, a fully documented version has been made available at [15].

6 Full system bandwidth scaling

The large FZJ scratch file system $WORK provides an aggregate bandwidth of about 33 GB/s (50% of the total bandwidth of the storage cluster). Given the performance characteristics shown in Sect. 3, we expect that roughly 12 to 16 racks of Blue Gene/P are needed to saturate the bandwidth of this file system (48k to 64k MPI tasks in VN mode). For smaller partitions the aggregate bandwidth should scale linearly with the partition size, and for larger partitions it should ideally stay flat at roughly the file system limit.

Figure 7 shows the measured scaling behavior, using SIONlib with logical task-local I/O but physically one file per ION. We observe the expected linear scaling up to the file system saturation at around 48k to 64k tasks, and a write performance drop of about 10% beyond that point which is still being investigated. The different curves show the effect of varying the number of tasks per ION which perform the actual I/O. Using 512 tasks per ION (VN mode) exhibits a larger performance hit, all curves with 256 or less tasks per ION are roughly comparable.

7 Summary and conclusions

In this paper we have described the hierarchical I/O model of the Blue Gene architecture, and have shown how to tune the I/O nodes to support the resulting high number of I/O streams per ION. While the technical details are specific to Blue Gene/P, this is indicative of the general trend of increasing numbers of cores per OS instance or parallel file system service instance.

GPFS has always distributed metadata load e.g. by utilizing distributed token servers and dynamically assigning the metanode roles for *different* files to different nodes. Our scaling studies on up to 288k cores suggest that the performance of an individual node which is performing metadata management functions for *one specific file* is becoming a gating factor when tens or hundreds of thousands of concurrent updates are happening against that single file. Performance tracing and tuning is ongoing to better understand and address these effects.

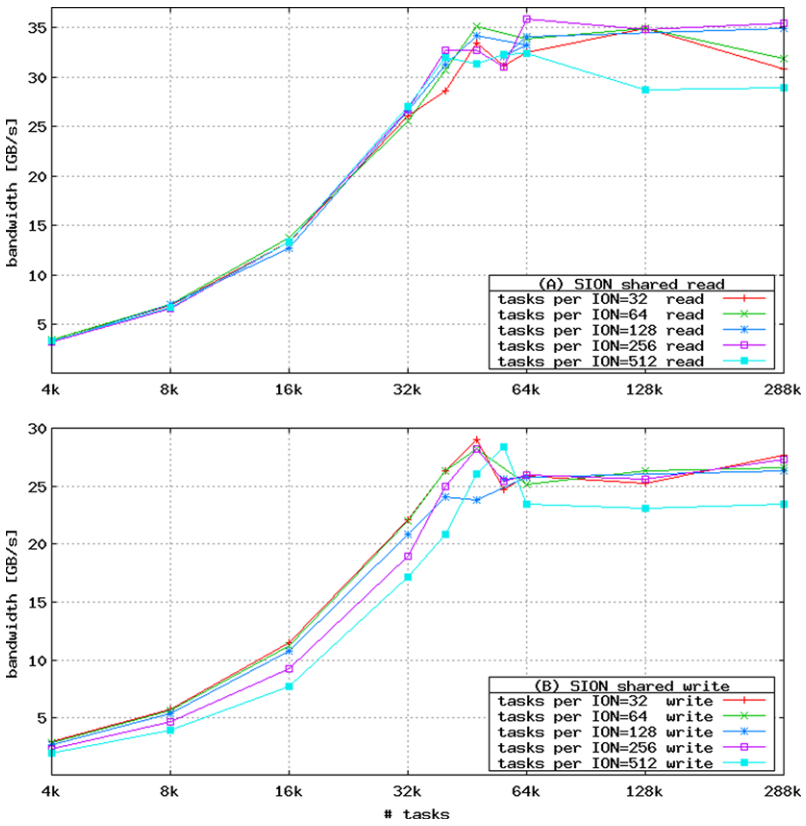

Fig. 7 Total bandwidth scaling with the number of MPI tasks. SIONlib read/write tests, one file per ION

The optimizations in SIONlib provide an easy to implement solution for POSIX-style I/O: With the help of per-ION container files for task-local I/O, and by controlling the number of tasks per ION which perform the I/O, we are able to reach and sustain the file system's peak bandwidth for up to 288k tasks. In the future, similar optimizations could also be implemented within the MPI-IO layer.

Acknowledgements This work summarizes the efforts of a large FZJ/IBM team; special thanks to M. Stephan, J. Docter, O. Mextorf, L. Wollschläger, U. Schmidt (FZJ) and K. Kutzer, K. Petersen, M. Megerian, T. Gooding, M. Mundy, D. McNabb, B. Hartner, G. Shah, K. Gunda, Y. Volobuev (IBM).

References

1. IBM Blue Gene team (2008) Overview of the IBM Blue Gene/P project. IBM J Res Dev 52(1/2):199–220. Online at http://dx.doi.org/10.1147/rd.521.0199
2. Gara A, et al (2005) Overview of the Blue Gene/L system architecture. IBM J Res Dev 49(2/3):213–248. Online at http://dx.doi.org/10.1147/rd.492.0195
3. Coteus P, et al (2005) Packaging the Blue Gene/L supercomputer. IBM J Res Dev 49(2/3):213–248. Online at http://dx.doi.org/10.1147/rd.492.0195
4. Moreira J, et al (2005) Blue Gene/L programming and operating environment. IBM J Res Dev 49(2/3):367–376
5. Mohr B, Frings W (2010) Jülich Blue Gene/P extreme scaling workshop 2009. Technical report FZJ-JSC-IB-2010-02. Online at http://www.fz-juelich.de/jsc/docs/printable/ib/ib-10/ib-2010-02.pdf
6. Mohr B, Frings W (2010) Jülich Blue Gene/P extreme scaling workshop 2010. Technical report FZJ-JSC-IB-2010-03. Online at http://www.fz-juelich.de/jsc/docs/printable/ib/ib-10/ib-2010-03.pdf
7. Schmuck F, Haskin R (2002) GPFS: A shared-disk file system for large computing clusters. In: Proceedings of the first USENIX conference on file and storage technologies. Monterey, CA, January 28–30, 2002, pp 231–244, 2002. Online at http://www.usenix.org/publications/library/proceedings/fast02/
8. Mextorf O, Schmidt U, Wollschläger L, Hennecke M, Kutzer K (2010) Storage and network design for the JUGENE Petaflop system. inSiDE 8(1):62–66 Online at http://inside.hlrs.de/htm/editions.htm
9. Vishwanath V, et al. (2010) Accelerating I/O Forwarding in IBM Blue Gene/P Systems. Online at http://www.mcs.anl.gov/uploads/cels/papers/P1745.pdf
10. Latham R (2008) Parallel I/O for high performance and scalability. In: 14th annual meeting of ScicomP, NY, 19–23 May 2008. Online at http://www.spscicomp.org/ScicomP14/talks/Latham.pdf
11. Parallel-NetCDF: a high performance API for NetCDF file access. Online at http://www.mcs.anl.gov/parallel-netcdf/
12. The HDF Group Parallel HDF5. Online at http://www.hdfgroup.org/HDF5/PHDF5/
13. Unidata. Parallel I/O with netCDF-4. Online at http://www.unidata.ucar.edu/software/netcdf/netcdf-4/

14. Gropp W, Huss-Lederman S, Lumsdaine A, Lusk E, Netzberg B, Saphir W, Snir M (1998) MPI: the complete reference. The MPI-2 extensions, vol 2. MIT-Press, Cambridge, p 10
15. SIONlib: scalable massively parallel I/O to task-local files. Online at http://www.fz-juelich.de/jsc/sionlib/
16. Frings W, Wolf F, Petkov V (2009) Scalable massively parallel I/O to task-local files. In: Proceedings of SC09, Portland, OR, USA, November 14–20. Online at http://dx.doi.org/10.1145/1654059.1654077
17. GPFS Version 3.3 (2009) Concepts, planning, and installation guide. IBM publication GA76-0413-03, September 2009
18. GPFS Version 3.3 (2009) Administration and programming reference. IBM publication SC23-2221-03, September 2009
19. GPFS Version 3.3 Advanced administration guide. IBM publication SC23-5182-03, September 2009
20. IBM developerWorks "HPC Central" Wiki. Online at http://www.ibm.com/developerworks/wikis/display/hpccentral/
21. Lakner G (2010) IBM system Blue Gene solution: Blue Gene/P system administration. Configuring I/O nodes. Chap 12. IBM redbook SG24-7417-03. Online at http://www.redbooks.ibm.com/abstracts/sg247417.html
22. Hennecke M (2006) GPFS multicluster with the IBM System Blue. Gene solution and eHPS clusters. IBM redpaper REDP4168. Online at http://www.redbooks.ibm.com/abstracts/redp4168.html
23. IOR HPC benchmark. Online at http://sourceforge.net/projects/ior-sio/
24. Shan H, Shalf J (2007) Using IOR to analyze the I/O performance for HPC Platforms. LBNL technical report. Online at http://escholarship.org/uc/item/9111c60j