Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Evaluating Application I/O Optimization by I/O Forwarding Layers

Jan Christian Meyer [*a], Jørn Amundsen[a] , Xavier Saez[b]

a Norwegian University of Science and Technology (NTNU), Trondheim, NO-7491, Norway
b Barcelona Supercomputing Center, c/ Jordi Girona, 29,08034 Barcelona,Spain

Abstract

Rewriting application I/O for performance and scalability on petaflops machines easily becomes a formidable task in terms of man-hour effort. Furthermore, on HPC systems the gap of compute to I/O capability in Tflop/s vs GByte/s has increased by a factor of 10 in recent years. It makes the insertion of I/O forwarding software layers between the application and file system layer increasingly feasible from a performance point of view. This whitepaper describes the work on evaluating the IOFSL I/O Forwarding and Scalability Layer on PRACE applications. The results show that the approach is relevant, but is presently made infeasible by the associated overhead and issues with the software stack.

1. Introduction

The growing performance gap between the computational rate and hierarchical memory of HPC platforms makes avoiding data movement bottlenecks crucial to application performance and scalability. As memory speed and capacity vary inversely, use of the high volume storage at the bottom of the memory hierarchy comes at high run-time cost. Unlike the higher levels of the hierarchy, however, file system I/O also provides persistent storage, which requires application control. Thus, automation techniques such as caching, prefetching and demand paging do not apply unmodified, creating the challenge of how to support programmable high performance parallel I/O while abstracting system-specific performance details.

Parallel file systems typically address this problem by concealing a client/server architecture at the OS kernel level, transforming low-level file I/O into requests from a network service instead of directly interfacing with the storage devices. I/O forwarding layers provide a similar, additional level of this software stack, intercepting I/O requests by user processes, and forwarding them to a dedicated I/O server which can perform system calls on behalf of the caller. At large scale, this potentially benefits both application program and system, from the reduction in the overall number of system calls, permitting the forwarding server to coalesce large numbers of small requests into a smaller number of larger ones, and otherwise address issues with application I/O characteristics which are beyond the conservative assumptions required by system call semantics.

The I/O Forwarding and Scalability Layer (IOFSL) provides both MPI-IO and POSIX interfaces between application code and the forwarding server, showing micro -benchmark advantages over the native parallel file systems on a Linux cluster with PVFS2 over ext3, and two Blue Gene/P installations [1,2]. This whitepaper evaluates the use of this framework on a recent Altix ICE system, applying it to the EUTERPE plasma physics application. The goal is to use this experimental setting to evaluate the scalability of utilizing the framework as an alternative to rewriting application I/O.

* Corresponding author. *E-mail address*: jan.christian.meyer@ntnu.no

2. Methodology

In addition to hardware details, the performance parameter space governing forwarded parallel I/O reflects design decisions in both the parallel file system design, platform- and installation-specific configuration details, as well as both statically and dynamically configured parameters of the I/O forwarding server software. An exhaustive evaluation of IOFSL is infeasible within the scope of this study. This section details restrictions introduced to make testing feasible, and justifies them with regard to use case. Because the IOFSL software is intended to provide I/O optimization transparently, the primary test objective is to minimize modifications to the provided application and input set.

2.1 Forwarding server configuration

The IOFSL 'iofwd' server v.1.0.0 relies on a combination of 5 mandatory and 4 optional software components in addition to itself. Attempts were made to minimize the complexity of the installation, in the interest of controlling the experimental setting. The mandatory dependencies were satisfied using the flex v2.5.35 and bison 2.3 text analysis generators, Boost C++ libraries v. 1.47.0, the BMI v. 2.8.0 component of PVFS, and the mpich2 1.4.1p1 implementation of MPI and ROMIO I/O modules. The optional additions of a FUSE client interface and OpenPA v. 1.0.3 were also integrated. The former was added because EUTERPE requires a POSIX I/O interface. The latter was added because without it, the internals of the I/O forwarding server falls back on expensive atomic operations which eclipse the cost of the application load already at modest scale. All components were configured using default options, unless otherwise specified by the IOFSL installation instructions [3].

It should be noted that the FUSE client in this configuration failed during simultaneous accesses to a directory controlled by one instance of it, indicating a race condition. Determining its cause is beyond the scope of this work, but the observation has methodological impact in that it requires application code to establish one client per parallel task which performs I/O. For HPC architectures with loosely coupled SMP nodes, this restricts applications to an I/O equivalent of *hybrid-masteronly* parallelism [4], where one thread assumes responsibility for remote access on behalf of a group which shares memory.

2.2 EUTERPE configuration

EUTERPE exposes several forms of parallelism. Aside from being an MPI code, it features some sections of OpenMP parallelism, as well as supporting several combinations of third-party FFT and sparse linear algebra libraries. Because of the restriction on shared POSIX I/O use of a single directory, testing with MPI would require that multiple ranks at the level of one shared memory node can execute with environments of separate working directory paths, although with identical contents. Although this is technically possible, attempts to test any MPI application in this mode show that the 'hydra' process manager included in the employed mpich-2 package does not admit it, restricting configurations to library combinations which work using a single I/O thread per node.

An EUTERPE configuration with OpenMP enabled fits this category, but cursory source code examination shows that OpenMP use is restricted to 24 loops in 2 files. The configuration provided in the unmodified code from the use case employs the Intel compiler tool chain and math kernel libraries. Satisfying this, version 11.1.073 of the tools reports that all the relevant loops are successfully identified during compilation. Testing this configuration with the Linux 'mpstat' tool collecting utilization through a set of 10 1-second intervals during a preliminary run, indicates the benefit to be barely observable on our test system. Altering the sparse linear algebra library choice to PetSc [5] permits using threaded versions of the math kernel libraries. The same mpstat test shows nearly full node utilization during computationally demanding phases of this configuration, suggesting it for further testing.

This configuration is selected without testing further library combinations, based on the assumptions that additional I/O requirements introduced by another combination are not necessary to application function, and that periodic peaks of full utilization makes it improbable that another configuration will further shorten computation time enough to significantly raise the frequency of I/O operations. While experimental verification of these assumptions is feasible, it will not be examined further as results are contrasted with ideal scaling in Section 3.

2.3 Test platform

The test system employed is an SGI Altix ICE 8200 machine, featuring 256 nodes of dual 4-core Xeon 5570 CPUs, clocked at 2.93 GHz and with hyper-threading enabled. Each node features 24GB of memory, and dual InifiniBand interconnect ports. The interconnect is hierarchically organized, with 4 rack leader nodes each controlling 4 rack units of 16 nodes. It features the Lustre parallel file system, with x86_64 Linux v. 2.6.32.46 at the time of testing.

2.4 Performance model and validation

The combination of systems described in Sections 2.1-2.3 restricts the parameter space of testing to running EUTERPE with up to 256 I/O clients, producing the combined load resulting from a maximum of 16 threads per client, for a total of 4096. Performance estimates are made relative to a basis of 8-node/64-thread runs, because the supplied test problem is compute-bound at this scale, making I/O loads below it less relevant. Scalability is addressed in the strong mode, because of the natural mapping to the fixed-size problem provided by the use case. Scaling proceeds by powers of 2 in the number of clients/threads, because this maps naturally to the execution platform, and provides sufficient variation in I/O characteristics to bring out the impact of introducing I/O forwarding. The experimental method is strongly influenced by an interaction between the IOFSL FUSE client and EUTERPE. Particularly, the Intel Fortran compiler translates Fortran I/O intrinsic statements into a series of system I/O calls which cause the client to fail. The exact cause is not determined, but a Fortran program which only opens and closes an I/O unit to store a text string is sufficient to reproduce the issue. This program fails identically to EUTERPE when translated with the Intel compiler, while translating it with the GNU Fortran compiler creates a version which works with the FUSE client. Compiling the provided version of EUTERPE using GNU Fortran does not yield a working program without source code modification, which puts the pursuit of this solution beyond the scope of this work.

To address this problem, a manual, static analysis of the EUTERPE code reveals that its I/O characteristics can be replicated by a performance model. The execution path taken by the use case input reaches 4 sources of I/O activity: an initialization phase, periodic updates of a histogram file throughout execution, periodically storing restart states, and a finalization which includes storing a restart state. We restrict attention to the periodic behavior, because its aggregate cost grows in proportion to application run time, while initialization and finalization are constant. The periods of the histogram file updates and restart files are independent, and both explicitly controlled by program input, in terms of multiples of synchronized time steps. The provided problem set runs for 50 time steps, updating histograms on multiples of 5 and writing restart files on multiples of 20. This pattern is used for validation.

The fundamental I/O mode of the application is to access files per MPI process, so the load imposed by execution can be determined by monitoring the size of these files during execution. The histogram files prove to have a regular size, with a small size offset from initialization, followed by a sequence of regular, fixed-size appends. For power-of-two runs, the appended data size varies exactly inversely with the number of processes. Restart file sizes reflect the state of ions in the simulated plasma, displaying inherent variability. Particle-in-cell methods show inherently different load balances between their particle and grid domains, making an analytical expression to cover all cases infeasible. The test problem, however, displays variability of only 1% between tasks in 256-way runs. As an approximation, file sizes from process rank 0 are taken as representative, and files are assumed to scale linearly. The greatest observed deviation was in the file size of the (128/2048) run, which differed from the expected factor 2 by 4×10^{-4}. Problems of greater imbalance may invalidate the model.

As computation time influences I/O demand only indirectly, it is approximated using the average of observed computation step times from runs, averaged over steps which do not contain I/O. For modeling purposes, any deviation form this is classified as I/O. Time steps also contain communication and synchronization, but it is not necessary to discriminate these from computation in order to estimate file system access costs. Moreover, append and overwrite access modes are treated uniformly, and file sizes are assumed to be sufficient that write bandwidth dominates latency. This results in a model function parametric in step count s, process count p, aggregate file system write bandwidth $w(p)$, and average compute step cost $comp(p)$ as per equations (1a-1c).

$$cost(s) = comp(p) \ | \ s \bmod 5 = 1,2,3,4 \quad (1a)$$

$$cost(s) = comp(p) + 50727424 / (p/8) * w(p) \ | \ s \bmod 20 = 5, 10, 15 \quad (1b)$$

$$cost(s) = comp(p) + (1200032096 + 50727424) / (p/8) * w(p) \ | \ s \bmod 20 = 0 \quad (1c)$$

Eq. 1a is the basic computation step cost, Eq. 1b offsets it by a cost estimate based on scaling the histogram byte count from the (8/128) run, and Eq. 1c adds the estimate of the restart files, scaled from the smallest case. The model's purpose is to estimate write bandwidth without running EUTERPE, to evaluate potential performance with IOFSL. The IOR file system benchmark [6] permits configuration to per-process file access of given sizes, and is used to emulate write behavior for the restart file size. Table 1 summarizes the results from these IOR tests.

Table 1. IOR mean aggregate write rates for native Lustre installation.

Process count	Per-process file size [MB]	IOR mean aggregate write rate [MB/s], 10 repetitions
8	1144	2399.61
16	572	4105.29
32	286	4373.24
64	143	4741.83
128	72	4794.08
256	36	4662.29

It is noteworthy that the rates reported in Table 1 are quite high, which is attributed to the fact that the file sizes in question are quite manageable within the memory constraints of the system. It is probable that these results do not reflect a stress test of the storage system so much as its buffering mechanisms, but as the purpose of the test is to emulate application behavior with and without IOFSL, this should not be detrimental to the comparison as long as the conditions are identical between the two. Inserting the measurements into the model equations, Figs. 1,2 plot the model estimates of time estimates for compute steps 1-49, shown with execution time as reported by EUTERPE.

Fig. 1. Traces vs. estimates of (a) 8-32 clients; (b) 64 clients

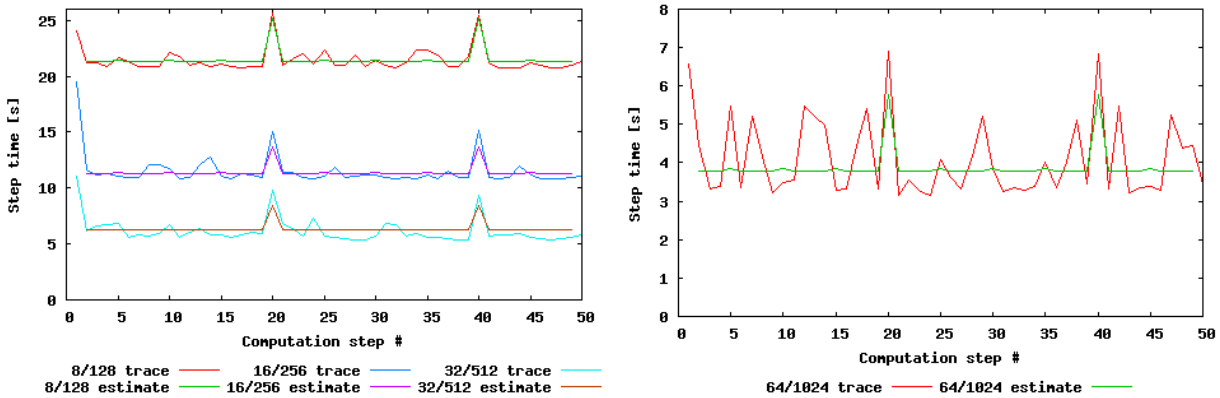
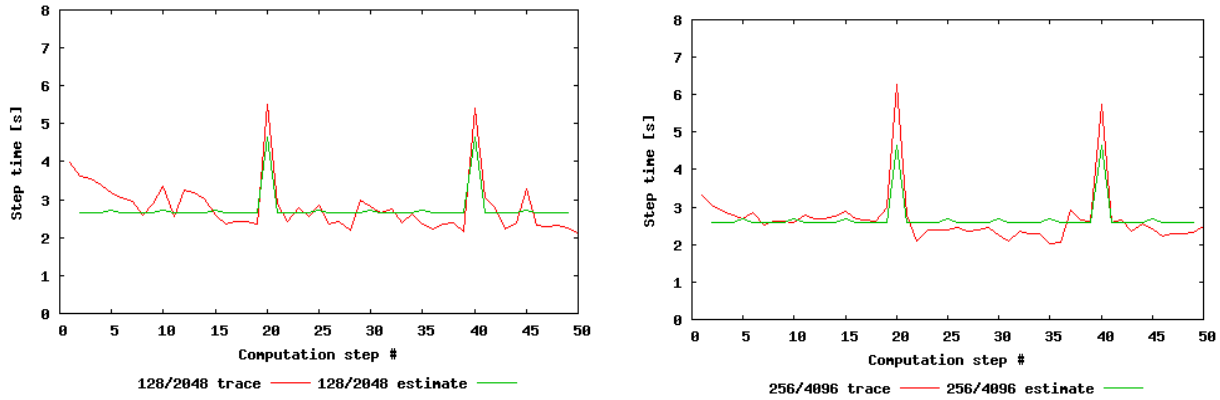


Fig. 2. Traces vs. estimates of (a) 128 clients; (b) 256 clients



Model accuracy is imperfect, but comparison shows that the peaks at checkpoint steps appear to be consistently underestimated by a constant term, histogram I/O load is smaller than the variability between time steps of actual runs, and the periodic restart file writing accounts for the I/O which will first become a bottleneck.

3. Results and discussion

Combining the average computation step time of EUTERPE with the I/O cost predictions obtained from the native Lustre file system as in Fig. 3 and Tab. 2 shows that the cost of the checkpoint steps scales with the cost of the computation step, stagnating beyond 128 clients. At this point, the computation step cost accounts for ~56% of a checkpoint step. This suggests that the application will not become I/O bound until the latency of file creation number of files produced inflates the I/O cost beyond the bandwidth requirement of writing, and that the frequency of checkpointing can control the fraction of total execution time spent on I/O.

Table 2. I/O cost relative to computation step cost

cpu#/thread#	8/128	16/256	32/512	64/1024	128/2048	256/4096
I/O cost	3.97s	2.32s	2.18s	2.01s	1.99s	2.04s
Computation	21.28s	11.31s	6.22s	3.77s	2.65s	2.59s
I/O / total	15%	17%	26%	34%	42%	44%

Fig. 3. Step time with and without I/O, empirically observed computational scaling

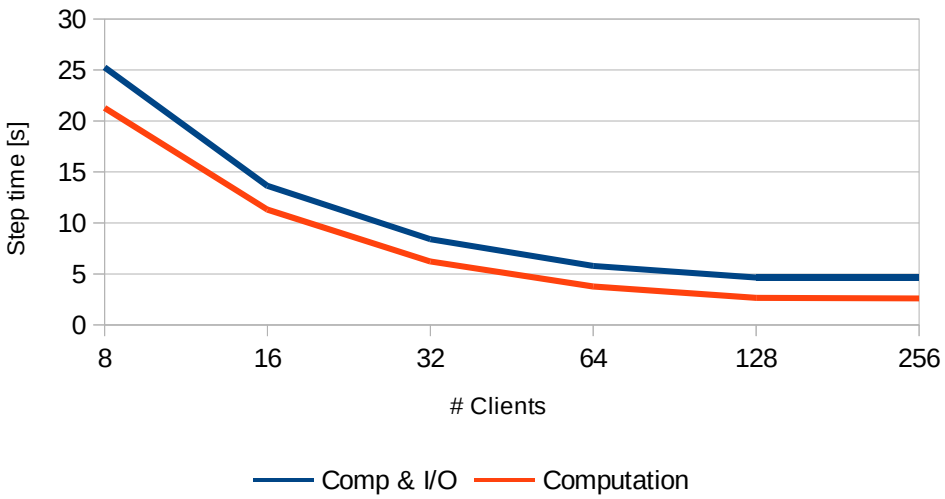
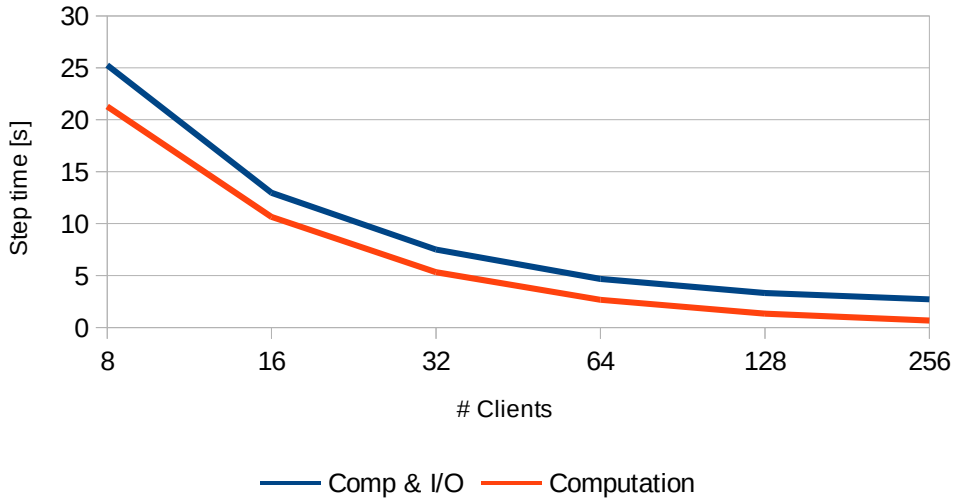


Fig. 4: Step time with and without I/O, ideal linear computational scaling



The level computation time scaling at the high end of Fig. 3 can be attributed to the numerical intensity of the test problem. For comparison, Fig. 4 shows the development of scaling the base 8/128 case computation time linearly with the number of processors. The development up to the 128/2048 case suggests that actual scaling is near ideal while there is enough computational work to distribute. Although the test problem borderlines being too small to verify it empirically, this clearly shows the tendency that addressing the I/O requirements of EUTERPE is most relevant at large scale with frequent checkpoints. This means that the defining characteristic of the relevant I/O load is a large number of comparatively small files. Tab. 3 shows the IOR benchmark values for the test runs which are closest to displaying this behavior. The FUSE POSIX client figures are omitted for the 256 client case, because the write bandwidth is too low to justify occupying the entire machine with a greater test. The performance log of the server shows that the great majority of the induced overhead is spent writing data, where introduction of an additional software layer has the least potential for altering application behavior.

Table 3. Forwarded write bandwidths with MPI-IO and POSIX clients

Client count	Per-client file size [MB]	IOR mean agg. write rate, MPI-IO, 10 repetitions	IOR mean agg. write rate, POSIX, 1 repetition
128	72	675.78	3.53
256	36	646.00	-

As using MPI-IO with the synthetic benchmark does not suffer from the issue of multiple mount points per node, it admits further testing for charting forwarding server behavior up to the number of physical cores. Results from these runs are presented in Tab. 4. Note that EUTERPE is not tested with the multiple-clients-per-node execution mode which gave these figures.

Table 4. Forwarded write bandwidths with MPI-IO, multiple clients per node

Client count	Per-client file size [MB]	IOR mean aggregate. write rate, MPI-IO, 10 repetitions
512	18	733.93
1024	9	742.17
2048	4	675.27

Comparing the achieved bandwidths in Tables 1, 3 and 4 leaves no question that the overheads associated with rerouting I/O across user-space processes before dispatching it to the file system far outgrows the practical benefit within the parameter space charted using the EUTERPE workload. While there is reason to suspect that there exists a scale at which I/O time is dominated by an execution pattern which could benefit from forwarding, practical use of the tested framework is presently not practical, not only because of the issues with its functionality, but also because it would introduce an intolerable overhead.

4. Conclusions

The conclusion of this study is that while the file creation bottleneck which can be ameliorated by user-space I/O forwarding is observable, using the IOFSL forwarding software layer comes at prohibitive costs. This conclusion is founded on the cost balance observed using nodes with small counts of powerful processing cores, and may not be appropriate for massively parallel, simplified core architectures. Future generations of large-scale platforms may alter the described balance of per-process file sizes and number, and place demands on checkpoint intervals. These considerations suggest that future work to reduce the cost of I/O forwarding is highly relevant, although reported results are negative in the use case tested here.

Acknowledgments

This work was financially supported by the PRACE project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-211528 and FP7-261557.

References

1. V. Vishwanath *et al.*, Accelerating I/O Forwarding in IBM Blue Gene/P Systems, Proceedings of the 2010 International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2010
2. K. Ohta et al., Optimization Techniques at the I/O Forwarding Layer, Proceedings of the 2010 IEEE Intl. Conference on Cluster Computing (CLUSTER), 2003
3. https://trac.mcs.anl.gov/projects/iofsl/ (Last retrieved 27.11.2011)
4. R. Rabenseifner, Hybrid Parallel Programming on HPC Platforms, Proceedings of the Fifth European Workshop on OpenMP, EWOMP'03, 2003.
5. http://www.mcs.anl.gov/petsc/ (Last retrieved 27.11.2011)
6. http://sourceforge.net/projects/ior-sio/ (Last retrieved 27.11.2011)