Scalable I/O Tracing and Analysis

Karthik Vijayakumar [1] Frank Mueller [1] Xiaosong Ma [1,2] Philip C. Roth [2]

[1] Department of Computer Science, North Carolina State University, Raleigh, NC 27695-7534
[2] Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN 37831

Abstract

As supercomputer performance approached and then surpassed the petaflop level, I/O performance has become a major performance bottleneck for many scientific applications. Several tools exist to collect I/O traces to assist in the analysis of I/O performance problems. However, these tools either produce extremely large trace files that complicate performance analysis, or sacrifice accuracy to collect high-level statistical information. We propose a multi-level trace generator tool, ScalaIOTrace, that collects traces at several levels in the HPC I/O stack. ScalaIOTrace features aggressive trace compression that generates trace files of near constant size for regular I/O patterns and orders of magnitudes smaller for less regular ones. This enables the collection of I/O and communication traces of applications running on thousands of processors.

Our contributions also include automated trace analysis to collect selected statistical information of I/O calls by parsing the compressed trace on-the-fly and time-accurate replay of communication events with MPI-IO calls. We evaluated our approach with the Parallel Ocean Program (POP) climate simulation and the FLASH parallel I/O benchmark. POP uses NetCDF as an I/O library while FLASH I/O uses the parallel HDF5 I/O library, which internally maps onto MPI-IO. We collected MPI-IO and low-level POSIX I/O traces to study application I/O behavior. Our results show constant size trace files of only 145KB irrespective of the number of nodes for FLASH I/O benchmark, which exhibits regular I/O and communication pattern. For POP, we observe up to two orders of magnitude reduction in trace file sizes compared to flat traces. Statistical information gathered reveals insight on the number of I/O and communication calls issued in the POP and FLASH I/O. Such concise traces are unprecedented for isolated I/O and combined I/O plus communication tracing.

1. Introduction

Analyzing I/O behavior of parallel applications is a difficult problem. As shown in Figure 1, the compute node I/O stack is increasingly complex, complicating investigation of the I/O behavior of applications. This is especially true for parallel I/O, due to the multi-level layering of I/O modules and architectural variations in storage systems. Although these layers provide essential abstractions to the end user by hiding complex implementation details behind simpler interfaces, these details ultimately impact the actual behavior of any I/O call issued at high-level layers. These abstractions may also hide any inherent performance bottleneck caused by poor implementation at lower layers, such as low-level synchronization issues.

Supercomputing PDSW'09, Nov. 15, 2009. Portland, OR, USA.
Copyright © 2009 ACM ISBN 978-1-60558-883-4/09/11...$10.00.

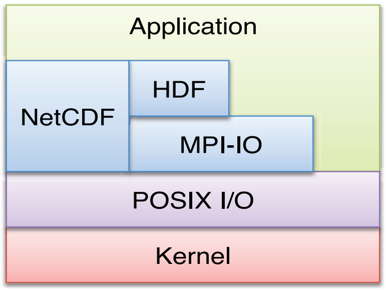

Figure 1. Typical Compute Node I/O Stack

Several studies have been conducted to analyze the I/O behavior of parallel applications. Because that behavior can be difficult to understand due to the complex interactions between the various layers of the I/O software stack, application I/O kernels and I/O event trace analysis have been designed to simplify the analysis. Application I/O kernels like Flash I/O [1] are abstracted from a full application, retaining the application's I/O behavior while simplifying or eliminating its computation and communication. However, keeping such I/O kernels up-to-date requires substantial effort and manpower particularly when the original application changes. I/O event tracing provides an alternative to application I/O kernels. I/O traces can be readily generated by instrumenting an application with an event trace capture library and then simply running an application. However, the resulting traces can be large and remain architecture specific, raising scalability concerns on large-scale machines. The trace file size is particularly an issue with I/O behavior monitoring, as such trace file accesses bring disturbance to the I/O system and may distort timing and performance information recorded from the application while our approach is minimally intrusive.

When capturing data about program and system events, most tracing tools also capture time stamp information, which is essential to analyze bottlenecks and to facilitate what-if analyses that assess the impact of changing key system architecture aspects such as network bandwidth or latency. There can be three types of traces based on whether and how timing information is collected: (a) time-stamped traces and (b) time-aggregate traces and (c) time-agnostic traces. Time-stamped traces provide exact timing information of each trace event, time-aggregate traces provide statistical time-stamp information per event and time-agnostic traces simply record events without collecting any timing information. Our approach initially captures the actual delta time between any two consecutive trace events during event recording and also preserves the order of events [16]. In SPMD programs, fine-grained clock skews and diverging computational progress often cause delta times to differ from one iteration to another as well as across nodes, and even the smallest differences would prevent compression. To keep the trace file size at bay, we not only compress repeating events but also aggregate timing information in the form of histogram bins. Nonetheless, we maintain information about timing outliers in terms of the observed delta time as well as the responsible node, which is sufficient to determine the cause of communication inefficiencies, such as load imbalance.

Contributions: This paper contributes ScalaIOTrace, a toolset for scalable I/O trace collection and effective replay that performs efficient and flexible trace analysis independent from the original application. Our contributions include:

- We provide the capability to record traces at several layers of abstraction in the software stack of system and library I/O interfaces;
- we create novel capabilities to automate trace analysis for collecting statistical information from the traces;
- we support I/O replay capability in ScalaIOTrace optionally combined with MPI communication replay;
- we provide an extensible approach to trace events and facilitate future user enhancements, customization and integration with other performance analysis tools.

To evaluate the benefits of ScalaIOTrace, we conducted experiments using Parallel Ocean Program (POP) application and FLASH I/O benchmark to collect I/O traces at multiple layers. We collected statistical information on file read/write and communication operations of POP and FLASH I/O.

2. Related Work

Many event trace capture and analysis tools have been developed, such as TAU [17], Vampir [12], Paraver [15] and SCALASCA [5]. These tools support the collection of detailed data about program events, including communication and I/O events. Most of these tools collect traces using a library instrumentation in a similar fashion as our work, but few employ an event trace compression mechanism to control trace data volume. One recent exception are tools supporting the Open Trace Format (OTF) [8]. OTF uses zlib compression on blocks of data, which limits compression effectiveness and also precludes any analysis on the compressed format. During trace analysis or replay, the original traces have to be reconstructed, which is again inefficient and does not scale. Our work performs trace analysis without the need to reconstruct the original trace from the compressed trace and without losing any structural information even in its compressed representation.

Gao *et al*. [4] developed an event trace compression technique that performs static analysis on the application binary and collects loops and functions as structures. Along with the structure, a path grammar is constructed. Path grammars are then utilized to encode paths taken during execution. These structures are compressed individually and stored. Even the iteration count is stored along with the compressed structure traces. This loosely resembles the RSD and PRSD technique used in our work [6, 10, 13, 16]. Unlike their work, our tool does not require the construction of grammars for individual applications separately. Our work employs a generalized trace compression approach based on call path stacks. It is sufficient to link the tool library along with the application to collect traces. This generalization also enables comparative trace studies between two different applications.

Lu and Shen [9] proposed multi-layer event tracing and analysis to identify the system layers responsible for performance bottlenecks. Their evaluation was limited to very small systems (at most 16 nodes) and their approach does not employ any compression mechanism. With traces collected from many different layers of I/O subsystems, there will be unmanageable increases in the trace file size when 100s or 1000s of nodes are used. They have also provided interesting results on performance issues on parallel file system due to constraints at the operating system level using trace analysis. In contrast, our tool enables trace analysis using compressed traces and with even the potential to automate trace analysis by performing a stack walk to identify the layering information using recorded stack signatures along with event traces.

3. Background

Our work builds on ScalaTrace [14], a tool that collects communication traces from large-scale parallel application runs with efficient, lossless online trace compression. It also preserves timing information in the compressed form along with the calling context of MPI calls [16]. In this paper, we examine the efficacy and performance of ScalaIOTrace, which is built on ScalaTrace. ScalaIOTrace collects compressed MPI-IO as well as POSIX I/O system calls.

ScalaTrace uses the MPI Profiling layer (PMPI) [11] through Umpire [18] to intercept MPI calls and collects MPI traces. It performs two types of compression: *intra-node* and *inter-node*, motivated by the temporal and spatial similarity in parallel simulations' execution due to the SPMD programming paradigm. Intranode compression exploits the repetitive nature of timestep simulation and is performed on-the-fly during trace collection. Internode compression, in contrast, exploits the homogeneity in behavior among different processes. It is performed before MPI_Finalize and compresses traces collected from all nodes into a single file.

ScalaTrace compression algorithms and data structures are discussed in detail in previous work [14][16]. Here, we briefly introduce several of its key ideas and techniques relevant to I/O tracing. ScalaTrace captures MPI events in innermost loop as Regular Section Descriptors (RSD) [6] while power-RSDs capture RSDs in higher-level loop nests in a constant size [10]. Consider the example in the following code snippet:

```
for( i = 0; i < 10; i++ ) {
   MPI_File_open(...); // Open 10 different files
   for( j = 0; j < 100; j++) {
      compute1();
      MPI_File_write(...); // Write call
   }
   MPI_File_close(...);
}
```

I/O trace compression with PRSDs results in the following tuples: RSD1: $<$ 100, MPI_File_write $>$ denotes a loop with 100 iterations of MPI_File_write and PRSD1: $<$ 10, MPI_File_open, RSD1, MPI_File_close $>$ represents the embedding of the inner loop in the outer one consisting of 10 iterations with additional I/O operations.

Another important feature of ScalaTrace is the time preservation of captured traces. Instead of recording absolute timestamps, the tool records delta time of computation between adjacent communication calls. During RSD formation, instead of accumulating exact delta timestamps, statistical histogram bins are utilized to concisely represent timing details across the loop. These bins are comprised of statistical timing data (minimum, maximum, average and standard deviation). More details on collecting statistical timing information are provided in [16].

4. Multilevel I/O Trace Collection

File I/O operations in large-scale scientific applications typically go through a multi-level software stack, such as the one depicted in Figure 1. A parallel application often performs file I/O through high-level scientific data format libraries, such as HDF5 [2] and netCDF [3], where shared file access capabilities may be built on general-purpose parallel I/O libraries such as MPI-IO. Applications may also directly use MPI-IO interfaces to perform parallel I/O, where the I/O operations are passed to the parallel file system clients running on each compute node. Eventually, I/O calls are made through the system I/O libraries.

With such an increasingly deep I/O stack, I/O performance depends not just on application access behavior, but also on the interaction between different abstraction layers. It is important to isolate an application's behavior at a certain level, or to correlate activities at multiple levels. With our prototype system, we make initial effort to expose these layers and enable the analysis of multi-level traces in a scalable way, to better understand I/O behavior of an application on a specific architecture.

In our prototype implementation, ScalaIOTrace collects traces of MPI-IO and low-level POSIX I/O function calls. Both MPI-IO and POSIX calls represent the high and low levels of the I/O software stack. ScalaIOTrace can, of course, be extended further to collect traces at any layer.

4.1 MPI-IO Trace Generation

At the surface, the methodology to collect MPI-IO traces resembles trace collection of MPI communication calls. ScalaIOTrace contains an interposition engine based on the Umpire tool [18] to automatically create wrappers for trace events from a specification of the corresponding instrumentation actions. The wrapper engine generates modules that assist in trace collection (header files, a wrapper file containing all MPI function overrides, and lookup files containing details of MPI functions used internally).

However, certain I/O function parameters, such as file name, offset and MPI_File opaque objects, require special handling to achieve scalable trace compression, as detailed below.

Regarding file names, we consider several widely used approaches for performing periodic I/O in parallel applications. In many applications, all processes send output data to a root process (process 0), which then performs I/O. Alternatively, all processes may use parallel I/O to write one or more shared checkpoint/snapshot files, either with collective or individual I/O calls. In a third and currently less common approach, each process creates its own output file. The checkpoint files and/or snapshot files are written periodically, typically once per c (where c is some constant) timesteps, identified by a timestep number in the file name. In case separate files are created per process, files are typically differentiated by encoding the process/node rank in file names. For efficient intra- and inter-node trace compression, ScalaIOTrace parses the file names to identify the "static" and "dynamic" component strings. For example, file names checkpoint-001-0.nc, checkpoint-001-1.nc, etc. will be recognized as having static components of "checkpoint-001-" and ".nc". During compression, file names from disjoint nodes are merged into a single event if the static file name components match. Process ranks can be substituted by RSDs, which are expanded during replay (see below).

Similarly, ScalaIOTrace needs to identify and merge parallel MPI-IO accesses to shared files across I/O timesteps and across processes. For example, assume each of 10 nodes writes a disjoint range of 1000 bytes in a shared file, *i.e.* node 0 accesses the byte range of [0,999], node 1 accesses [1000,1999], etc. ScalaIOTrace encodes such access pattern into three fields (start position, stride, and the total number of elements) during intra- and inter-node compression, so that multiple file accesses with the same call stack will be compressed into a single event.

Finally, MPI_File handles representing file objects are opaque pointers handled internally by the MPI library and do not exhibit repetitive patterns. ScalaIOTrace stores these handles in a buffer, added upon the file open operation. Subsequent accesses to open files are recorded by referencing the corresponding handle's offset in the buffer rather than the handle itself. This allows us to compress and replay the I/O traces appropriately.

Apart from MPI-IO calls, ScalaIOTrace provides support to record traces for creating custom data types, such as MPI_Type_create_darray, which are widely used in collective file accesses. These custom data type handlers are also opaque pointers and are treated in the same manner as file handlers.

4.2 POSIX I/O Trace Generation

ScalaIOTrace also collects traces from POSIX I/O calls. Compared to MPI-IO, POSIX I/O calls belong to the lower level in the I/O stack and potentially can provide more information on the actual requests made to the parallel file system. Tracing POSIX calls can help in identifying performance bottlenecks in the middle and lower I/O stack layers, as well as in capturing I/O activities that do not go through a higher-level I/O library. Many of ScalaIOTrace's techniques in MPI-IO trace collection, compression, and replay can be applied to POSIX I/O as well. We discuss several POSIX-specific design and implementation issues below.

We exploited GNU linker's link time function interposition facility to provide instrumentation for POSIX I/O calls and to collect traces. The "–wrap" option enables function calls, such as open, write, etc., to be redirected to corresponding interposition functions (*e.g.*, __wrap_open()). The interposition wrappers implement trace collection and call the corresponding native (actual) function (*e.g.*, __real_open()). ScalaIOTrace provides a separate library for POSIX wrappers, which, together with the ScalaIOTrace library, is statically linked with the application using the link switch "–wrap" to signify which I/O functions are interposed.

Most of the function parameters in POSIX calls are similar to those of MPI-IO calls (*e.g.*, file name, number of bytes read/written). During our initial testing with ScalaIOTrace, we discovered that several files had been opened even before application execution and thus prior to our I/O interpositioning. These activities are accessing the internally managed resources and sockets opened by the MPI runtime system. We observed many I/O calls on these files during the initialization phase to coordinate application execution and system activity. Since these I/O calls were outside the application scope, we filtered them out by recording the traces with the files opened only after MPI_Init.

5. Trace Analysis

Our prototype ScalaIOTrace implementation not only supports scalable tracing, it also supports a scalable replay engine. Given a single, compressed trace file, the replay engine allows all I/O and communication calls to be reissued without trace decompressing while preserving event ordering. For this process, the replay engine runs as an MPI job with the same number of tasks as its original application. It then replays I/O and communication events in each node with their original parameters except for actual file content / message payloads. Instead, a random buffer of the same size as the original file/message buffer is utilized. Additionally, computation time on each node is simulated by a delay between any two I/O or communication calls based on recorded delta time. As delta time is arranged in histograms, delays can be replayed such that the computational time is resembled during replay through randomized distribution of delta time delays over histogram information.

Based on this specialized replay facility, we designed a generic and novel automated post-mortem trace analysis framework. Our design provides generic event handlers for all recorded trace functions. These handles can then be interposed or substituted by user-specific code when the trace is traversed in our generic analysis engine. While trace analysis could also be performed in a more conventional manner during application execution with interposed events within ScalaIOTrace, post-mortem generic trace analysis provides several advantages. Conventional trace analysis requires a priori knowledge about performance problems in order to collect and correlate the subset of I/O and communication events that may contribute to performance problems or application characterization footprints. Short of such a prior knowledge, conventional trace analysis is generally repeated with different refinement steps, which requires a separate application execution each time. In contrast, our generic post-mortem trace analysis facilitates the detection of anomalies or identification of communication patterns in the applications by iterative refinement without re-running the application. Instead, different trace analysis interposition functions at the replay level operate directly on compressed traces, and by omitting time-accurate replay, can be traversed rapidly taking only a fraction of the original application's execution time.

In this work, we exploit generic trace replay to demonstrate its capabilities in one specialization example. By providing aggregation interposition functionality, we obtain statistical details on the number of I/O operations and collective/blocking/non-blocking communication calls across all nodes. This could easily be refined for group-specific analysis of MPI communicators or subsets of traced events originating from certain code sections based on per-call stack backtrace information.

6. Experimental Results

We evaluated our ScalaIOTrace prototype in two aspects: (1) its effectiveness of communication-I/O trace file compression, and (2) its capability of collecting statistical information on I/O and communication activities via replaying the compressed traces. For such evaluation, we experimented with (1) a complex parallel I/O benchmark, Flash I/O [1], which is closely modeled after the FLASH astrophysics code, and (2) a production-scale climate simulation application, the Parallel Ocean Program (POP) [7]. More details on these workloads are given below.

Our experiments were conducted on Jaguar, the Cray XT4 system at ORNL. Each of its compute nodes contain a 2.1 GHz quad-core AMD Opteron 1354 processor and 8GB of DDR2 memory. The login nodes run a full-featured Linux version while the compute nodes run the Compute Node Linux microkernel.

We collected trace that are (i) uncompressed, (ii) compressed within each individual nodes (intra-node) and (iii) compressed both within nodes and across all nodes resulting in a single file (inter-node). Trace file sizes are assessed under strong scaling, where we vary the number of nodes while keeping the overall problem size fixed. Note that the effectiveness of trace compression should be evaluated considering the application's programming model and computation behavior (such as regular vs. irregular). For example, for regular SPMD codes, good compression across all nodes is expected, as each of these nodes will be running the same program working on disjoint yet similar datasets.

Regarding statistical information collection, we demonstrate that our proof-of-concept prototype is able to perform post-mortem analysis to examine the traces, such as how many calls have been issued and how I/O operations behave under strong scaling. While such statistics are common information items, ScalaIOTrace has the unique advantage of avoiding either rerunning the application or going through large-sized trace files. Tool features can be easily extended in the future to plug in user supplied trace processing routines for customized analysis.

6.1 Flash I/O Benchmark Results

Flash I/O simulates the I/O behavior of the FLASH application, which is a block-structured adaptive mesh hydrodynamics code [1]. The original benchmark distribution contains simulation using two I/O methods, one with parallel HDF5 [2] and one with serial f77 binary. For our experiments, we used a distribution modified by Argonne National Laboratory, which is compatible with newer versions of HDF5 as the API of newer versions of HDF5 is not backward compatible with older versions. Parallel HDF5 internally uses MPI-IO. As a result, all nodes are involved in I/O workload.

Figure 2 depicts the size of trace files generated by two approaches over increasing number of nodes, both on log scale. The size of the uncompressed trace files grows linearly with increasing

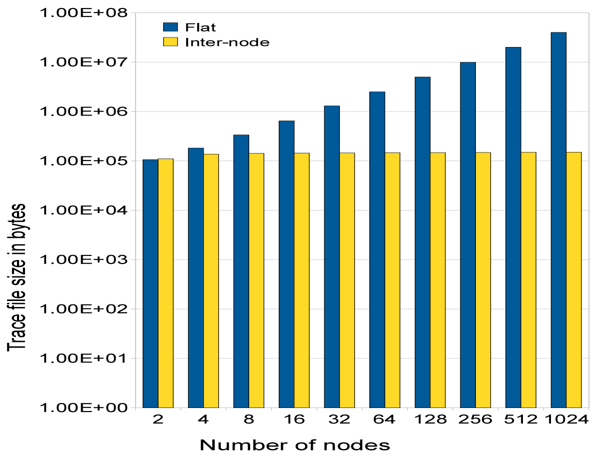

Figure 2. Flash I/O Benchmark

number of nodes. The reason for this behavior is that each node writes its own trace file and the number of files created grows with the increase in the number of nodes. In contrast, the size of the inter-node compressed traces is almost constant under strong scaling. This behavior of perfect inter-node compression is attributed to the SPMD programming style in the Flash I/O benchmark without any data-dependent conditional statements. FLASH I/O does not contain any loops. Hence, intra-node compressed traces are similar to flat traces and thus omitted in the results.

# nodes	MPI-IO at 0	POSIX I/O at 0	Comm. at 0	MPI-IO Other	POSIX I/O Others	Comm. Other
2-1024	194	171	299	85	56	299

Table 1. Number of Multi-Scale MPI/POSIX I/O and Communication Calls for Flash I/O

Table 1 shows the statistical information collected exploiting ScalaIOTrace's generic analysis feature with a specialized module for aggregation of event statistics. It shows that FLASH I/O issues the same number of I/O and communication calls under weak scaling with increasing number of nodes while the overall output file size increase due to the larger number of nodes. Since parallel HDF5 uses MPI-IO, which in turn uses POSIX I/O, separate aggregation results are reported for both MPI-IO and POSIX I/O. We observe that the number of MPI-IO calls for both node 0 and all others is greater than that of corresponding POSIX I/O calls. This is due to MPI_File_set_view calls issued before writing to the file.

6.2 Parallel Ocean Program Results

The Parallel Ocean Program (POP) is an ocean circulation model developed at Los Alamos National Laboratory. It does not use any parallel I/O library. Instead, node 0 collects output data and generate netCDF files. POP can run with two resolutions, a lower resolution grid (1 degree) and a higher resolution one (0.1 degree).

Our experiments exercises the lower resolution 1 degree grid in this work. These experiments gives us the ability to analyze compression effectiveness for production run applications. The problem size in case of a 1 degree grid is 320x384 and the individual block size is 10x12 resulting in a total of 1024 (32x32) blocks distributed to individual nodes. We conducted experiments by varying the maximum number of blocks assigned to each node.

Figure 3 shows the trace file size for different types of trace collections by varying the maximum blocks allocated to nodes.

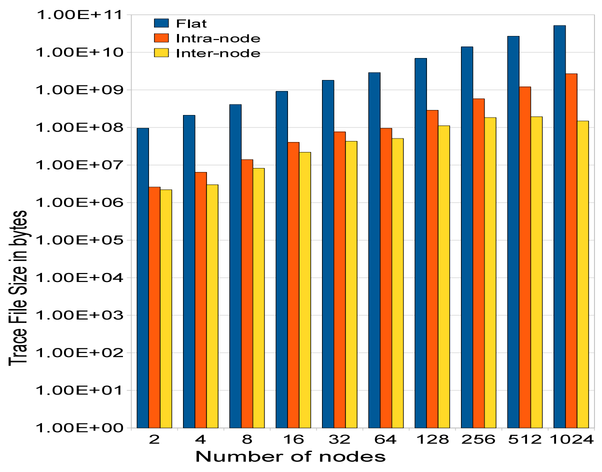

Figure 3. Parallel Ocean Program

The figure shows that intra-node compression alone can reduce the trace file size by an order of magnitude. Not surprising, inter-node compression brings further trace size reduction. However, unlike observed in our previous experiment with FLASH I/O, inter-node compression here fails to obtain near-constant trace file sizes. Instead we see a linear increase in the file size up to 256 nodes. Then, trace file sizes flatten for 512 and 1024 nodes since timestep behavior becomes more regular at 1024 nodes resulting in more effective inter-node compression at this size.

We also found that intra-node compression is not perfect for executions across different timesteps. POP invokes a set of functions in a loop that comprise one timestep iteration. The simulation goes through multiple timesteps based on the number of simulation days specified, where each simulation day consists of 46 timesteps. We collected traces by running POP for 2 simulation days. By trace file analysis, we determined that POP contains a preconditioned conjugate solver problem, which is a typical ϵ convergence problem, where a set of calculations are performed followed by Send/Receive and All_Reduce MPI calls. These calls and calculations iterate until the result converges. This results in different number of loop iterations due to data-dependent convergence points. Hence, the PRSD information varies across different timesteps and produces imperfect timestep loop compression. The trace file size on each node would have been significantly smaller had there been regular behavior for perfect compression across all timesteps.

Since this issue can also occur in other scientific applications, we intend to introduce user-tunable imprecision in trace recording in future work. The user can then specify an error percentage during trace recording to address the compression issue discussed above by merging two PRSD if iteration counts differences do not exceed a given imprecision threshold. This solution generalizes to other cases where a typical outer loop compression fails due to a small number of irregular trace events caused by data-dependent conditionals.

Table 2 shows the statistical information collected similar to the one reported for FLASH I/O benchmark. Since only node 0 performs I/O operations, we report results for I/O operations and collective (Coll.)/blocking (Block.)/non-blocking (NB) communication operations for node 0 separately and the average number of operations for all other nodes. We identified that all of the blocking communication calls at node 0 are I/O induced. This confirms that using any parallel I/O methodology would have definitely reduced the communication overhead involved in I/O.

# nodes	I/O at 0	Coll. at 0	Block. at 0	NB at 0	Coll. Other	Block. Other	NB Other
2	1589	21247	129034	231714	21247	0	385350
4	1573	21257	179284	308952	21257	0	388838
8	1573	21277	210140	308952	21277	0	393393
16	1573	21317	1444912	386190	21317	0	447680
32	1573	21397	858648	386190	21397	0	451373
64	1573	12225	858648	386190	8575	0	382512
128	1573	21877	463372	386190	21877	0	441344
256	1573	22517	470288	386190	22517	0	426550
512	1573	23797	239932	386190	23797	0	424329
1024	1573	26357	240198	386190	26357	0	412485

Table 2. Number of I/O & Communication calls in POP

With more nodes, the average number of non-blocking calls does not change significantly. This shows that the fraction of non-blocking receives relative to the corresponding blocking sends from node 0 is almost negligible. We also observe that the communication overhead increases due to strong scaling to solve the problem across even larger number of nodes. We further infer from trace analysis that communication is performed in sub-groups as the average non-blocking calls in other nodes is greater than that for node 0. We derived from the trace analysis that even collective operations, such as MPI_Allreduce, are performed in sub-groups. This is also the explanation for the difference in collective communication between node 0 and other nodes for the 64-node experiment. Here, the collective operations at node 0 operate in one sub-group while the average number of collectives across all the other nodes is lower (dominated by other sub-groups). We manually verified the correctness of this result.

7. Conclusion

We presented the design and implementation of ScalaIOTrace, a multi-level I/O tracing approach that combines aggressive trace compression, and a generic analysis tool that allows rapid post-mortem traversal of tracing in their compressed format to gather statistics, detect performance bottlenecks or analyze event precedence orders. Experimental results demonstrate the ability to obtain a single, near constant sized trace file of 145KB, irrespective of the number of nodes, capturing I/O and communication of the FLASH I/O benchmark. Trace sizes for POP grow under strong scaling but remain up to two orders of magnitude smaller than without compression. Such concise traces are unprecedented for isolated I/O and combined I/O plus communication tracing. We further identify challenges in compression due to application execution variability and methodologies to address these in future work.

Acknowledgements

This work was supported in part by NSF grants 0237570 (CAREER), 0621470, 0937908, and 0429653. It was also supported by a joint faculty appointment between Oak Ridge National Laboratory and NC State University.

It was also sponsored in part by the Office of Advanced Scientific Computing Research; U.S. Department of Energy. The work was performed in part at the Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725.

This research used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under Contract DE-AC05-00OR22725.

References

[1] FLASH I/O benchmark routine. http://www.ucolick.org/ zingale/flash_benchmark_io.

[2] Hierarchical data format. http://www.hdfgroup.org/HDF5.

[3] network common data form. http://www.unidata.ucar.edu/software/netcdf/.

[4] X. Gao, A. Snavely, and L. Carter. Path grammar guided trace compression and trace approximation. *High-Performance Distributed Computing, International Symposium on*, 0:57–68, 2006.

[5] M. Geimer, F. Wolf, B. J. N. Wylie, E. Abraham, D. Becker, and B. Mohr. The scalasca performance toolset architecture. In *International Workshop on Scalable Tools for High-End Computing*, June 2008.

[6] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[7] P. W. Jones, P. H. Worley, Y. Yoshida, J. B. White, III, and J. Levesque. Practical performance portability in the parallel ocean program (pop): Research articles. *Concurr. Comput. : Pract. Exper.*, 17(10):1317–1327, 2005.

[8] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. E. Nagel. Introducing the open trace format (OTF). In *International Conference on Computational Science*, pages 526–533, May 2006.

[9] P. Lu and K. Shen. Multi-layer event trace analysis for parallel i/o performance tuning. In *ICPP '07: Proceedings of the 2007 International Conference on Parallel Processing*, page 12, Washington, DC, USA, 2007. IEEE Computer Society.

[10] J. Marathe and F. Mueller. Detecting memory performance bottlenecks via binary rewriting. In *Workshop on Binary Translation*, Sept. 2002.

[11] MPI-2: Extensions to the message-passing interface. July 1997.

[12] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, 1996.

[13] M. Noeth, F. Mueller, M. Schulz, and B. R. de Supinski. Scalable compression and replay of communication traces in massively parallel environments. In *International Parallel and Distributed Processing Symposium*, Apr. 2007.

[14] M. Noeth, F. Mueller, M. Schulz, and B. R. de Supinski. Scalatrace: Scalable compression and replay of communication traces in high performance computing. *Journal of Parallel Distributed Computing*, 69(8):969–710, Aug. 2009.

[15] V. Pillet, J. Labarta, T. Cortes, and S. Girona. PARAVER: A tool to visualise and analyze parallel code. In *Proceedings of WoTUG-18: Transputer and occam Developments*, volume 44 of *Transputer and Occam Engineering*, pages 17–31, Apr. 1995.

[16] P. Ratn, F. Mueller, B. R. de Supinski, and M. Schulz. Preserving time in large-scale communication traces. In *International Conference on Supercomputing*, pages 46–55, June 2008.

[17] S. S. Shende and A. D. Malony. The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, 2006.

[18] J. S. Vetter and B. R. de Supinski. Dynamic software testing of mpi applications with umpire. In *Supercomputing*, page 51, 2000.