I/O BENCHMARKING OF DATA INTENSIVE APPLICATIONS

Olga Mordvinova[1,3], Thomas Ludwig[2] and Christian Bartholomä[3]

[1] Ruprecht-Karls-Universität, Institute for Computer Science
69120 Heidelberg, Germany
E-mail: mordvinova@informatik.uni-heidelberg.de
[2] Universität Hamburg and German Climate Computing Center,
20146 Hamburg, Germany
E-mail: ludwig@dkrz.de
[3] SAP AG, oCTO TREX, 69190 Walldorf, Germany
E-mail: christian.barthlomae@sap.com

Abstract. The increasing computerization of the society over the last decade led to the increased data volumes stored over the world. The need to handle and store these massive amounts of data, arising from diverse sources as scientific records, web pages, or social networks has created a new class of application – data intensive applications. Usually designed up to the specific application requirements, one of these most challenging questions is choice of the appropriate back-end. The I/O benchmarking tools can easy this decision process. However, despite of its high variety, there is a lack of portable and easily adaptable benchmarks that can correspond to the real application behavior. The programmable I/O benchmark *Parabench* tries to close this gap. Its input is based on access patterns, which can be adjusted to the application, for which the system is to be used. Our work concentrates on ability of *Parabench* in mimicking real applications. We describe here its capabilities to handle MPI-I/O and POSIX and present a modeling example of a data intensive application from the field of business intelligence.

Keywords: I/O Benchmarking, Data Intensive Applications, Access Patterns, POSIX I/O, MPI-I/O

1. Introduction

Today’s applications have to face the issue of constantly and fast growing data amount. According to [11] only on the field of data warehouses over the last decade, the largest data warehouse have increased from 5 to 100 terabytes. Another prominent example is Google, which data repository stores over 3 billion web pages [15] with the increasing tendency [6]. Big demands on storing, managing, and processing data created a new form of high-performance computing facility that focuses on data, rather than raw computation, as the core focus of the system. Such data intensive systems ([6] calls them data intensive super computer systems) works with massive amounts of data arising from such diverse sources as social networks, web pages, online transaction records or scientific data records e.g. from the field of medicine. Up to the scenario the I/O if such data intensive application is done via POSIX or parallel I/O interfaces.

One of the important questions while designing a data intensive application is choice of the suitable back-end. Cluster file systems are a popular option for the back-end of storing mass data [22]. Choosing an appropriate cluster file system for a specific data intensive application regarding its specific I/O requirements is challenging. For example, in the database scenario, application has usually to handle update of large datasets, whereas in the field of natural science large amounts of data may have been written in massive parallel fashion. The I/O benchmarking tools can easy this decision process. Available benchmarks are quite dissimilar [1–5, 13, 14, 19, 21 24, 35]. They vary in access pattern coverage, interface support, reporting and timing mechanisms. Considering these design goals we distinguish *application-based I/O* benchmarks and *synthetic I/O* benchmarks.

The *application-based I/O* benchmarks are derived from an application or a group of applications. They allow making studies of the architectural system performance under realistic I/O and communication requirements. Being very specialized, it can be difficult for non specialists to build and run such benchmarks. Moreover, they may not be distributed widely if they reveal sensitive algorithms or data. MADbench [4, 5] is an example for this benchmark group, which emulates a scientific cosmology application. Another example is the MPI-IO benchmark NAS BTIO [35]. It is derived from the compute benchmark BT that employs a fairly complex domain decomposition called multi-partitioning. The solution matrix of this calculation process is written to a file after every five time-steps and ordered by certain criteria at the end of the operation. Thus, BTIO only examines one workload and a very limited number of possibilities of MPI-IO.

The *synthetic I/O* benchmarks measure overall I/O system performance using standard or customized I/O access patterns. Benchmarks for local and network file systems IOZone [19] and FileBench [13] are the prominent examples for the synthetic benchmarks. However, due to lack of parallel implementations, they are not used for on the HPC field. Synthetic tools LLNL IOR [1] and b_eff_io [2, 31] support not only POSIX but also MPI-IO. IOR exercises concurrent read/write on one file or on separate files (one-file-per-processor). The benchmark is highly parameterized and offers a wide variety of access patterns. b_eff_io allows a very precise test configuration by using different parameter groups [31]. Its main purpose is to give a limited statement about I/O performance after a defined (usually quite short) time period in which production system is used for testing. Even if b_eff_io is a powerful benchmark, it is challenging

© Olga Mordvinova, Thomas Ludwig, Christian Bartholomä, 2010

to add new access patterns to its framework. There are some disadvantages in the approach of synthetic I/O. Even if the synthetic benchmarking tools are usually easier to build and run than application-based benchmarks, they may not accurately replicate specific workloads [5]. Furthermore, every benchmark provides its own pattern set, such that the result comparison between them is hardly possible.

For this reasons there are efforts to develop portable benchmarking tools with adjustable input. This approach combines the realism of application code with the convenience of synthetic benchmarks. The file system test program BWT [14] is an example of tool with adjustable input. Its parameters for executing tests have to be specified in input files and cover the majority of file system access patterns. BWT supports POSIX and limited parallel I/O commands. The process coordination for parallel I/O is implemented via barrier using IP multicast. Even in an early stage of implementation, BWT provides an approach close to the benchmark presented here [26].

The presented in this work benchmarking tool *Parabench* receives access patterns as input, which can be adjusted. Advantages of this approach in comparison to just running the application directly on the cluster are that these patterns can be shared without licensing or confidence problems, that might come along with the application code. Employment of this tool can also save effort in installing and setting up application as well as potential costs for individual test program or test program adjustment. The *Parabench* code itself has few dependencies; in contrast the interesting application might be hard to build on a different cluster. All these advantages are especially useful in testing and further comparing a closed source applications like considered here online analytic processor SAP NetWeaver BI Accelerator [7, 32].

The following paper concentrates on I/O benchmarking of data intensive workloads with pattern based benchmarking tool *Parabench*. To make the testing process more comprehensive, we introduce in section 2 benchmarks design and explain how we can compose a test program with the *Parabench* Programming Language. Section 3 contains a main contribution of this paper, i.e. that our approach does offer an easy and well usable platform for I/O benchmarking of data intensive applications. We demonstrate this on example of an application from the field of business intelligence. In section 4 we summarize our work and conclude with an outlook on future *Parabench* development.

2. Parabench

Design and Deployment. While designing *Parabench* we focused on development of tool with adjustable access pattern input [26], with the goal to use this tool for I/O behavior modeling of a wide range of applications. To make access patterns creation more flexible, we designed an own formal language – the *Parabench* Programming Language (PPL). To meet portability demands, we implemented *Parabench* in the C programming language. The modular design of our tool allows its easily extension and upgrade. There are three main components of *Parabench*: scanner, parser, and interpreter (see figure 1).

The scanner layer reads given input source files written in PPL and reduces the recognized lexical patterns to tokens. We use the open source scanner tool Flex [12] to generate a fast and flexible scanner layer for *Parabench*. The parser layer provides grammar parsing of the token stream received from the scanner. As output it generates a parse list for the interpreter. To build the parser layer, the open source parser generator Bison [9] is used. Since the interpreting program has qualities of a Lookahead-LR-Parser the PPL is defined as a context free grammar. The interpreter layer analyses and processes the list received from the parser. We implement the interpretation by using structures from the Gnome Library [16]: we used *GList* as stacks to implement the control flow constructs like loops and group blocks. To allow efficient lookups of user defined structures like variables and groups hash tables were used.

Figure 1 demonstrates the I/O benchmarking process with *Parabench*. Here we emulate the application I/O; however, the composition of any synthetic tests is also possible. The benchmark receives as input files with instruction written in PPL. The input files can be easily composed manually. However this option is not acceptable composing complex patterns, e.g. while emulating I/O of the real, data intensive applications. For this reasons, we also implemented a translator utility outside of the *Parabench* kernel. This utility is a regular expressions based parser, which, after a short setup, is able to convert a large amount of trace formats to the PPL. The input files are processed by the *Parabench*, i.e. scanner, parser, and interpreter and dumped to the result ACSII files.

Test Composition. The *Parabench* Programming Language (PPL) allows designing access patterns. To achieve this, we implemented three language construct groups: I/O language constructs, control flow language constructs, and auxiliary language constructs. Here we show the usage of PPL on some test program examples. The detailed PPL specification is done in [26].

The I/O language constructs provide a basis for a proper access pattern composition and can be classified into POSIX I/O [18] and parallel I/O, for which we use MPI-IO [17, 25]. For both interfaces we differ: explicit file handle, when open and close of the file has to be issued manually; implicit file handle, when user just specifies file or folder names. The supported I/O commands for POSIX are: `fopen`, `fclose`, `fread`, `fwrite` for the explicit file handle and `read`, `write`, `append`, `rename`, `create`, `delete`, `mkdir`, `rmdir`, `lookup`, `stat` for the implicit file handle. Listing 1 shows an example for a PPL stress test program for POSIX I/O. The basic I/O commands, here for the explicit file handle, are done on a set of created files and repeated `$num` times.

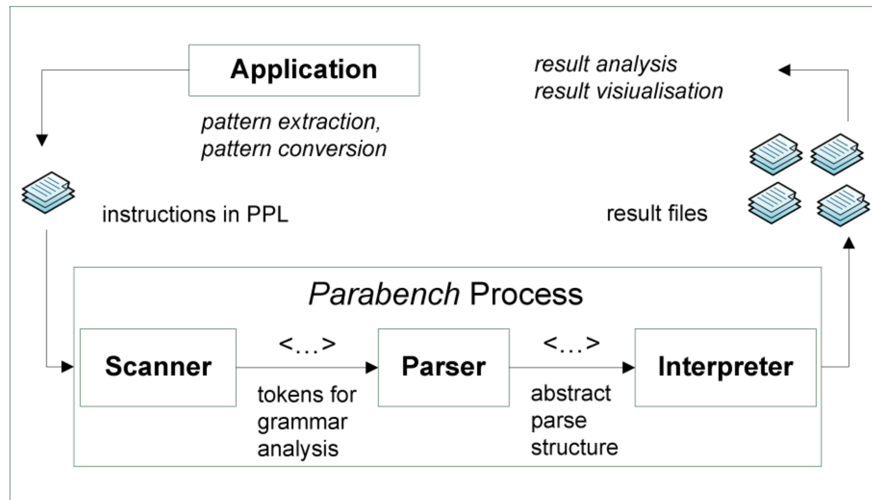

Figure 1: I/O benchmarking process with Parabench

Listing 1: Exemplary PPL test program for POSIX I/O

```
define param "num" $num "10000"
$env = "./env-posixio-test";
$file = "file.dat";
mkdir($env);

print "POSIX-IO stress test START";

time["POSIX-IO stress test"] {
        repeat $i $num write("$env/$file-$i", 1024);
        repeat $i $num append("$env/$file-$i", 1024);
        repeat $i $num read("$env/$file-$i");
        repeat $i $num lookup("$env/$file-$i");
        repeat $i $num mkdir("$env/dir-$i");
        repeat $i $num rmdir("$env/dir-$i");
        repeat $i $num stat("$env/$file-$i");
        repeat $i $num create("$env/$file-$ia");
        repeat $i $num delete("$env/$file-$ia");
}
print "POSIX-IO stress test STOP";
```

For the parallel I/O we support `pfopen`, `pfclose`, `pfread`, `pfwrite` for the explicit, and `pread`, `pwrite` for the implicit file handle. In patterns we can specify file data accessible for each process, which is internally realized by an MPI File view set on the file according to the pattern specification. Currently the array data type (i.e. each process gets a chunk of data in round-robin manner) and individual file pointers are implemented.

To realize the parallel I/O access patterns, we implemented the construct `define pattern`. It requires following parameter values: identifier name, number of iterations, number of elements, and the level of parallelism. The supported levels of parallelism, shown in figure 2, are *level 0* (non-collective calls, contiguous data access), *level 1* (collective calls, contiguous data access), *level 2* (non-collective calls, non-contiguous data access) and *level 3* (collective calls, non-contiguous data access).

The control flow language constructs are `repeat`, `barrier` and `group`. With construct `define groups`, process groups can be defined and therefore create more complex test programs. We distinguish between disjoint and overlapping groups. By providing the option `D` in the pattern definition disjoint groups can be created. This means, that no other groups overlap, and ungrouped processes are not part of the defined group. To limit code execution of certain processes to a defined group, the `group` statement is used. The `barrier` construct allows group specific process synchronization. The `repeat` statement enables execution of code blocks in a loop and reduces replication in the test specification.

The auxiliary language constructs are `time` and `print`. The `time` statement measures the wall clock time of the enclosed code block. To ease evaluating measurement results, a text label can be assigned to each time command. The `print` construct outputs text on standard output.

Listing 2 exemplifies the usage of all language construct groups on example of a small test program for the parallel I/O. Here we create a two disjoint process groups “writers” and “readers”, for which two parallel patterns are defined with 10 iterations, 100*1024*1024 elements per process, and different levels of parallelism. This patterns are applied on two different files (`file-lvl2.dat` and `file-lvl2.dat`) stored in `$env` directory. We archive process synchronization by means of `barrier;` time measurements are done by means of `time` command, which is put before the particular I/O command.

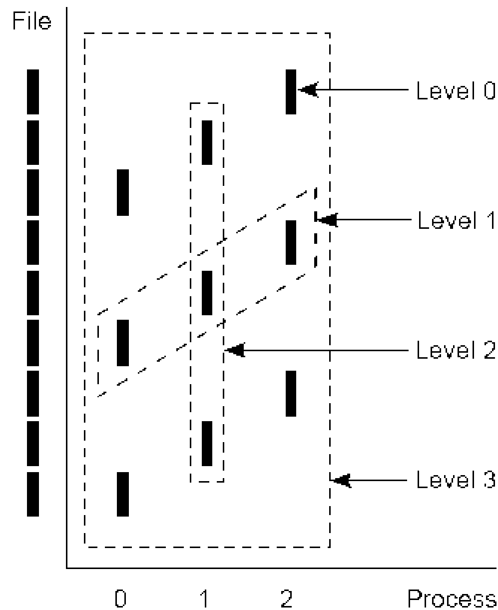

Figure 2: Supported levels of parallelism

Listing 2: Exemplary PPL test program for parallel I/O

```
define groups {"writers":D, "readers":D };
define pattern {"pattern-lvl2", 10, (100*1024*1024), 2};
define pattern {"pattern-lvl3", 10, (100*1024*1024), 3};

$env = "./env-mpiio-test"; mkdir($env);

print "MPI-IO test START";

group " writers" {
time ["write"] repeat 10 append ("$$rank",10*1024*1024);
time ["pwrite-lvl2"] pwrite ("$env/file-lvl2.dat", "pattern-lvl2");
time ["pwrite-lvl3"] pwrite("$env/ file-lvl3.dat", "pattern-lvl3");

}

barrier;

group "readers" {
$fh = fopen ("$$rank", "r+");
time ["read"] repeat 100 fread ($fh , 10*1024*1024) ;
fclose ( $fh);
time ["pread-lvl2"] pread ("$env/file-lvl2.dat", "pattern-lvl2");
time ["pread-lvl3"] pread ("$env/file-lvl3.dat", "pattern-lvl3");
}

print "MPI-IO test STOP";
```

Concluding, by means of the *Parabench* Programming Language we can easily compose any I/O test programs. Knowing particular I/O behavior we are able to design more complex tests, which can be used for studying and benchmarking of any application.

3. Evaluation

To demonstrate testing capability of *Parabench* we performed tests, which reflect behavior of data intensive workload, i.e. I/O emulation of a real, data intensive application. The application I/O tests were performed on two cluster file systems. The test environment was the cluster of the Research Group Parallel and Distributed Systems (PVS) at the Heidelberg University. It is a 32 Bit Ubuntu 8.04 cluster, consisting of 8 nodes with commodity-of-the-shelf (COTS) components and Gigabit Ethernet interconnect. For a qualitative evaluation precise hardware details are unnecessary and thus spared.

Testing OLAP Engine I/O. The tests with real patterns were based on the I/O of an exemplarily online analytic processing (OLAP) engine. For better understanding, we explain here its main features and access patterns more in detail.

OLAP systems have to handle growing volumes of time-critical data and reliably deliver fast response times for complex or ad hoc queries. The SAP NetWeaver BW Accelerator (BWA) [7, 32] is a prominent example of an in-memory column-oriented OLAP engine for use in business warehouse and business intelligence applications (see figure 3). BWA can be considered as a data intensive application: running on a large cluster it is able to load and process up to 25 TB of data [8]. Due to it architectural features specific for the main memory processing like high data compression, usage optimized data structures, column-orientation, OLAP engines like BWA can achieve desirable performance. Distributed server architecture allows parallel processing, scalability, reliability, and load balancing. There are several possibilities how BWA stores data: network attached storage (NAS) with NFS [30], storage area network (SAN) with a cluster files systems like GPFS [33], or the integrated distributed persistence layer [27] with software based fault tolerance. The last persistency option runs without central storage installation, just over the node local file system. To analyze the engines I/O behavior, i.e. its access patterns, we used the distributed persistence setup, because there is a separate data server process traces, which are easy to handle [27].

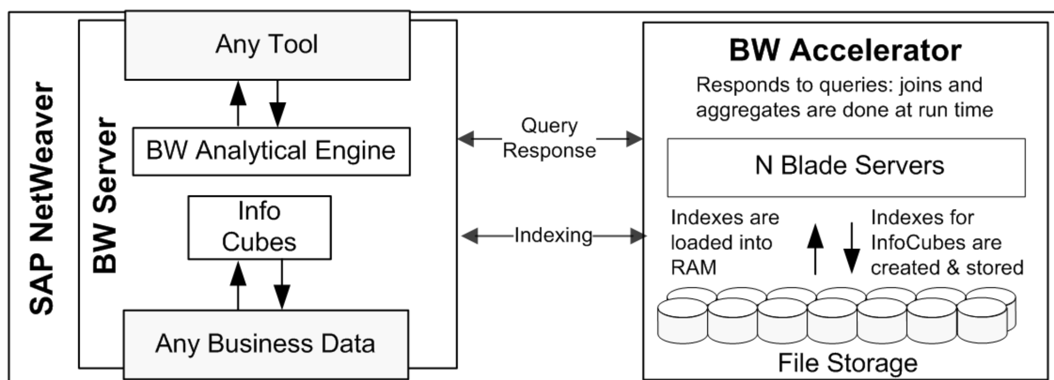

Figure 3: OLAP engine architecture [27]

All the basic operations in the BWA – creation, deletion, and indexing – are done on the index level. This is the minimum logical unit of the application [27]. Being hierarchically organized in namespaces, indexes contain: content files with transactional data, temporary index files with uncompressed indexes, attribute files or index columns, and configuration files. The index size varies up to the data model, application semantic and is limited by the aggregated size of the RAM in the cluster (indexes up to 2.9 TB are already used [8]).

At index creation, the initial index directory structure is built; index configuration files are created and modified. During index deletion, application configuration data is deleted, and particular directories and files are removed. Thus, there are operation on data and metadata while index creation and deletion. During the indexing, index is filled with data, which is first stored in temporary files. Afterwards this data is compacted. At the same time temporary data is read in parallel and written to the attribute files or columns used for the further querying. The application persistence calls are realized through three internal interfaces build on the top of POSIX: `bufferedFiles` (prefix `bf`) for attribute, temporary and configuration files, `contentStore` (prefix `cs`) for content files, and `storageLocation` for handling persistency on index level.

Figure 4 demonstrates the I/O calls distribution for *index create*, *index delete*, and *indexing* operations. While *index create* and *index delete* operation there are mainly metadata calls. While *indexing* a read operation dominates. Large amount of writes reflects the first phase of indexing, where data is written to the temporary files.

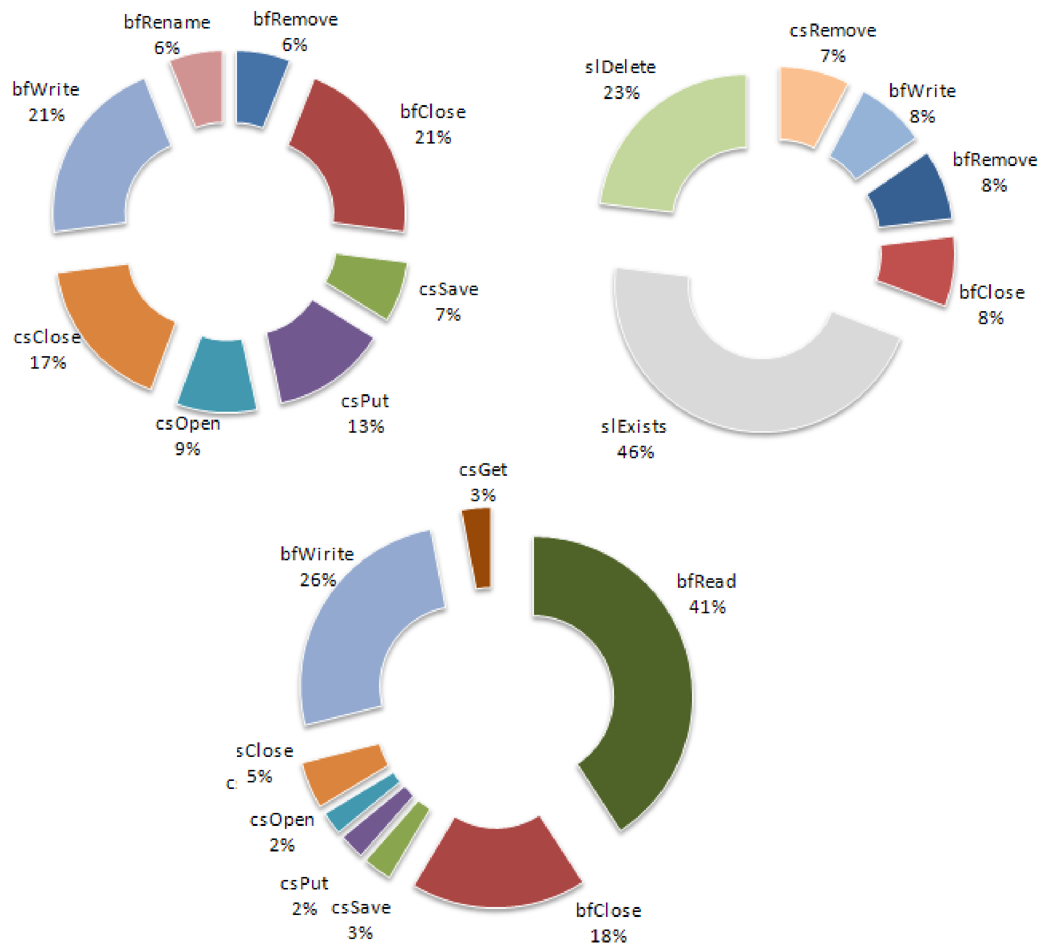

Figure 4: I/O calls distribution by percentage during *index create* (left above), *index delete* (right above) and *indexing* (below) operations

To evaluate the I/O performance of described application regarding a particular file system, it has to be installed on a cluster, where the particular file system is running. There are some challenges in this option. There are licensing and confidence problems relating to the proprietary application code. Second, the engine, certified only for a couple of file systems, possibly does not run on some recently developed systems properly and we may have to adjust it first. Finally, it is a challenge in installing engine itself and its required dependencies. Using *Parabench* this challenges can be resolved. We extracted described I/O patterns from the data server [27] traces of the engine, converted them without modifying these order to the PPL and composed tests for the OLAP basic operations *index create*, *index delete* and *indexing*.

Two cluster file systems were chosen for the evaluation: the parallel file system OCFS2 [28] and the object based distributed file system Ceph [34]. The goal was to see the influence of different file system design approaches on the I/O performance of chosen application. As a shared disk cluster file system, OCFS2 has to run on SAN/NAS or iSCSI. Because the PVS-cluster does not provide central storage, its abstraction from the cluster node devices with Network Block Device (NBD) [29] was created. OCFS2 provides symmetric file system architecture, i.e. every installed node has services for data and metadata processing. We made a comparable setup on Ceph. There is one monitor, one metadata server (MDS) and one object storage device (OSD) on each node. Testing OLAP I/O, we reserved first four nodes of the PVS cluster for exclusively acting as I/O nodes, whereas the remaining four nodes were used as client nodes. In the following, we call the I/O nodes as servers and the nodes, where the *Parabench* tool was started, as clients.

Table shows received results while basic index operations (*index create*, *index delete* and *indexing*) for setup with four servers and two clients per server. For the indexing operation, we also investigated the performance dependency for read and write operations of a single client (`s.read` and `s.write`) on different client-server setups, see figure 5. All tests were performed with 500 indexes. The results show, Ceph performs better on writes but significantly worse during metadata operations, in particular during *index delete*. The reason for this could be synchronization effort between distributed MDS (for every ODS one MDS was used). The default Ceph setup provides only one MDS. The lower write performance of OCFS2 could be explained by the overhead of the NBD, which combines the distributed hard drives in the cluster to one logical volume. By tuning file system setup, e.g. usage of central storage for OCFS2 and asymmetric metadata management for Ceph, we may get better results.

This evaluation shows the capability of the benchmarking tool *Parabench* in modeling such a complex patterns as OLAP engines and in providing a basis for performance analysis and performance optimization. In this stage of benchmark development there are still some questions regarding the validity of the presented simulation. The benchmark validity for comparison of two or more different file systems regarding this specific I/O load is given implicitly by following facts: test patterns in PPL are executed in the same order with same parameters, the same amount of data was written with the same client number. To evaluate difference between real application I/O and I/O simulation by *Parabench*, we need to repeat our tests on a cluster, certified and verified for the BWA application. Here we plan to determine the confidence interval of deviance to real application and evaluate impact of possible error sources like access pattern definition or access pattern processing.

Table. *Index create, delete, indexing* operations: 500 indexes, 4 servers, 2 clients per server

Op	Qnty	Ceph				OCFS2			
		Time [ms]			MiB/s	Time [ms]			MiB/s
		avg	min	max		avg	min	max	
index create									
delete	8000	0.033	0.028	1.188	21.79	0.049	0.032	0.286	12.3
mkdir	4000	3.870	0.521	56.283		1.585	0.471	144.903	
rename	8000	0.809	0.626	106.685		0.164	0.103	88.64	
write	25000	0.105	0.016	7.435		0.185	0.015	229.701	
index delete									
delete	26000	1.249	0,025	80.840		0.284	0.032	108.633	
rmdir	5000	200.485	105.498	1303.360		0.364	0.106	91.671	
indexing									
s.read [1]	40000	0.029	0.026	0.133	3.79	0.037	0.028	14.434	3.08
s.write [1]	84000	0.032	0.021	0.274	2.01	0.021	0.011	2.329	2.97
c.throughput [2]					22.11				13.55

[1] s = single client [2] c = complete

4. Conclusion and Future Work

In this work we presented the pattern based I/O benchmark *Parabench*. On example of a data intensive application we could demonstrate the convenience and usability of the developed tool. Already in this stage of implementation, *Parabench* allows convenient simulation of application I/O. Thereby test case modeling can be done either by automatic extraction of the access pattern or by manual pattern definition. In case of closed source applications we can additionally benefit of *Parabench* advantages and do not take care of licensing or confidence problems that come along with the proprietary code of tested applications.

The next important step in *Parabench* development is its extensive tests on the cluster, where the particular application is running. Based on these results, we will evaluate the proximity of the *Parabench* I/O simulation to the real application I/O, identify and prioritize improvement needs.

Furthermore, there are plenty opportunities for future development, which makes this approach even more compelling. First is extension of the pattern support for the true parallel I/O, which is mandatory for emulation more complex access patterns. Second is improvement of the current parse tree structure and language grammar. This would enable options for later more complex language constructs and therefore emulation of more complex application. We also plan an additional tool with ability for post processing and visualization of the benchmark results, and a graphical interface, which helps while pattern composing and validate input.

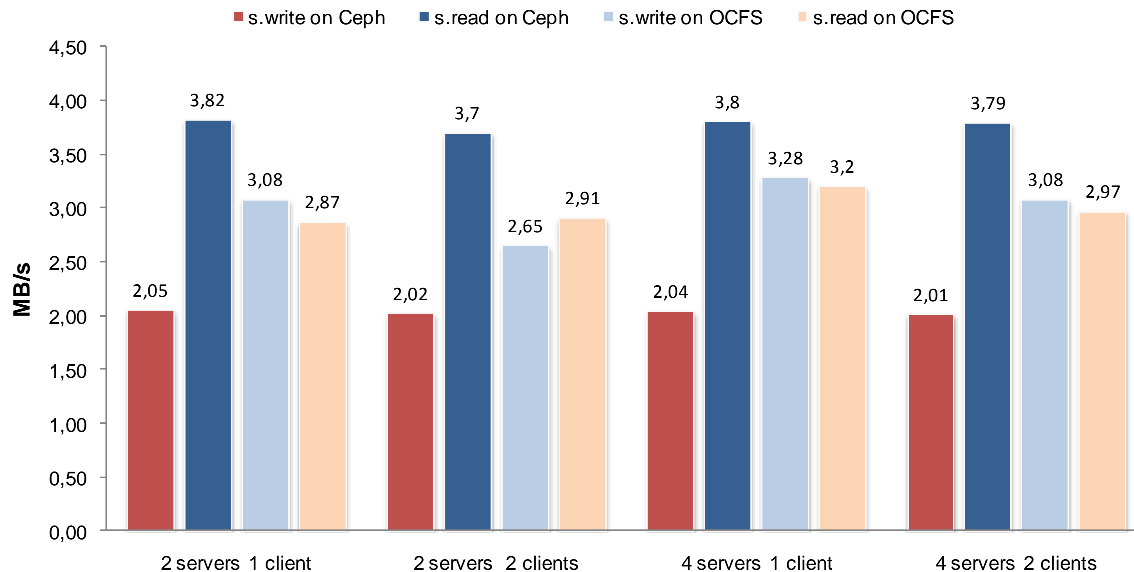

Figure 5: `s.read` and `s.write` performance on Ceph and OCFS while *indexing* with different client-server setups

1. *ASCI I/O Stress Benchmark*. http://www.llnl.gov/asci/purple/benchmarks/limited/ior/
2. *b_eff_io Benchmark*. https://fs.hlrs.de/projects/par/mpi//b_eff_io/
3. *Biardzki, Ch., Ludwig*. Th. Analyzing Metadata Performance in Distributed File Systems. In: Proc. of PaCT'09. Springer (2009) 8–18
4. *Borrill, J, Carter, J, Oliker, L., Skinner, D.* Integrated Performance Monitoring of a Cosmology Application on Leading HEC Platforms. In: Proc. of ICPP'05, IEEE Computer Society (2005) 119–128
5. *Borrill, J., Oliker, L., Shalf, J., Shan, H.* Investigation of leading HPC I/O performance using a scientific-application derived benchmark. In: Proc. of SC'07, ACM (2007) 1–12
6. *Bryant, R.A.* Data-Intensive Supercomputing: The Case for DISC. Technical Report, CMU (2007)
7. *Burns, R., Dorin, R.* The SAP NetWeaver BI Accelerator.Transforming Business Intelligence. Technical report, Winter Corporation (2006)
8. *Burns, R.* Large-scale testing of the SAP NetWeaver BI Accelerator on an IBM Platform. Technical report. Winter Corporation (2008)
9. *Bison Manual*. http://www.gnu.org/software/bison/manual/
10. Dean, J., Ghemawat, S. MapReduce: Simplified data processing on large clusters. In Operating Systems Design and Implementation, 2004.
11. *DeWitt, D.J., Madden, S., Stonebraker, M.* How to Build a High-Performance Data Warehouse. Technical article. MIT (2009) http://db.lcs.mit.edu/madden/high_perf.pdf
12. *Flex Manual*. http://flex.sourceforge.net/manual/
13. *FileBench Benchmark*. http://hub.opensolaris.org/bin/view/Community+Group+performance/filebench.
14. *Filesystem IO Test Program BWT*. http://people.web.psi.ch/stadler h/
15. *Ghemawat, S., Gobioff H., Leung, S.-H.* The Google File System. In: Proc. of SOSP'03. ACM (2003) 29–43 http://www.cs.rochester.edu/sosp2003/papers/p125-ghemawat.pdf
16. *Gnome Library*. http://library.gnome.org.
17. *Gropp, W.D., Lusk, E.L., Ross, R.B., Thakur, R. Using* MPI-2: Advanced features of the message passing interface. In: CLUSTER, IEEE Computer Society (2003)
18. *IEEE POSIX Certification Authority*. http://standards.ieee.org/regauth/posix/
19. *IOzone Filesystem Benchmark*. http://www.iozone.org/
20. Isard, M., et al. Dryad: Distributed data-parallel programs from sequential building blocks. In EuroSys 2007, March 2007.
21. *Krietemeyer, M., Versick. D. Tavangarian, D.* The PRIOmark Parallel I/O-Benchmark. In: Proc. of IASTED'05. IASTED Press (2005)
22. *Ludwig, Th.* Research Trends in High Performance Parallel Input/Output for Cluster Environments.In: Procs. of UkrPROG'04.
23. *NASU* (2004) 274–281
24. *May, J.* Pianola: A Script-based I/O Benchmark. In: Proc. of SC'08. IEEE (2008)
25. *Message Passing Interface Forum, MPI*. A message-passing interface standard. Version 2.1 (June 2008).
26. *Mordvinova, O., Runz, D., Kunkel, J.M., Ludwig, T.* I/O performance evaluation with *Parabench* – Programmable I/O Benchmark. In: Proc.of. ICCS'10 [To Appear], ICCS (2010)
27. *Mordvinova, O., Shepil, O., Ludwig, T., Ross, A.* A strategy for cost efficient distributed data storage for in-memory OLAP. In: Proc. IADIS Int. Conf. Applied Computing 2009. Volume I., IADIS Press (2009) 109–117
28. *Mushran, Sunil.* OCFS2 – A cluster file system for Linux. User's guide for release 1.4. http://oss.oracle.com/projects/ocfs2/dist/documentation/v1.4/ocfs2-1_4-usersguide.pdf
29. *Network Block Device*. http://nbd.sourceforge.net/
30. *Network File System*. http://sourceforge.net/projects/nfs/
31. *Rabenseifner R., Koniges, A. E.* Effective communication and file-I/O bandwidth benchmarks. In: Proc. of PVM/MPI'01, Springer (2001) 24–35
32. *Ross, A.* SAP NetWeaver BI Accelerator. Galileo Press (2009)
33. *Schmuck F., Haskin R.* GPFS: A Shared-Disk File System for Large Computing Clusters. In: Proc. of FAST'02, USENIX Association (2002) 231–244
34. *Weil, S. A. et al.* Ceph: A scalable, high-performance distributed file system. In: Proc. 7th Symposium on Operating Systems Design and Implementation, OCDI (2006) 307–320
35. *Wong, P., Van der Wijngaart, R. F.* NAS Parallel Benchmarks I/O Version 2.4, Technical Report NAS-03-002 (2003)