Available online at www.sciencedirect.com
ScienceDirect
Procedia Computer Science 1 (2012) 2125–2134
Procedia Computer Science
www.elsevier.com/locate/procedia

International Conference on Computational Science, ICCS 2010

I/O performance evaluation with *Parabench* – programmable I/O benchmark

Olga Mordvinova[a], Dennis Runz[a], Julian M. Kunkel[b], Thomas Ludwig[c]

[a]*Ruprecht-Karls-Universität Heidelberg, Department of Computer Science*
Im Neuenheimer Feld 348, 69120 Heidelberg, Germany
[b]*German Climate Computing Center*
Bundesstraße 45a, 20146 Hamburg, Germany
[c]*Universität Hamburg and German Climate Computing Center, Department of Informatics*
Bundesstraße 45a, 20146 Hamburg, Germany

Abstract

Choosing an appropriate cluster file system for a specific high performance computing application is challenging and depends mainly on the specific application I/O needs. There is a wide variety of I/O requirements: Some implementations require reading and writing large datasets, others out-of-core data access, or they have database access requirements. Application access patterns reflect different I/O behavior and can be used for performance testing.

This paper presents the programmable I/O benchmarking tool *Parabench*. It has access patterns as input, which can be adapted to mimic behavior for a rich set of applications. Using this benchmarking tool, composed patterns can be automatically tested and easily compared on different local and cluster file systems. Here we introduce the design of the proposed benchmark, focusing on the *Parabench* programming language, which was developed for flexible pattern creation. We also demonstrate here an exemplary usage of *Parabench* and its capabilities to handle the POSIX and MPI-IO interfaces.

Keywords: I/O Performance Evaluation, Access Pattern Modeling, MPI-IO, POSIX

1. Introduction

The broad range of applications in high-performance computing (HPC) have a large range of I/O requirements: some applications need to read and write large amounts of data in a serial fashion, some perform large out-of-core simulations, and others maintain large databases and have to update large datasets. Considering this specific I/O behavior is essential when looking for a suitable back-end, which is usually a cluster file system.

I/O benchmarking tools help during this decision making process. However, the available benchmarks to test parallel I/O are quite dissimilar. They typically have various reporting and timing mechanisms as well as varying

Email addresses: mordvinova@informatik.uni-heidelberg.de (Olga Mordvinova), dennis.runz@gmx.de (Dennis Runz), kunkel@dkrz.de (Julian M. Kunkel), ludwig@dkrz.de (Thomas Ludwig)
URL: http://pvs.informatik.uni-heidelberg.de (Olga Mordvinova), http://www.dkrz.de (Julian M. Kunkel), http://www.dkrz.de and http://wr.informatik.uni-hamburg.de (Thomas Ludwig)

1877-0509
doi:10.1016/j.procs.2010.04.238

access pattern coverage. Results are not standardized and not easy to compare with each other, which complicates assessment of I/O performance.

The motivation for this work was to develop a generic I/O performance analysis tool, that closes this gap. With its interpreter based approach, *Parabench* is such a tool. Unlike most of the I/O benchmark tools currently available, e.g. [1–5], *Parabench* is exceptional because its behavior can be programmed by adjusting the input file access patterns.

Advantages of this solution over 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. The *Parabench* code itself has few dependencies, whereas the application of interest might involve a high build complexity due to library dependencies. Therefore, it is much easier to port and evaluate I/O performance with *Parabench*. Since it has a modular architecture and is Open Source, our benchmark can be easily extended and adjusted by the community.

The remainder of this paper is organized as follows. After reviewing the latest concepts in the performance evaluation field in section 2, we introduce the design of the proposed concept and outline the *Parabench* programming language, which was designed for flexible pattern composition. In section 4, we present an exemplary use of *Parabench* and its capabilities to handle the POSIX and MPI-IO interfaces. To show this, we conducted some tests on a testbed with the file systems PVFS2 and ext3: synthetic tests for the parallel I/O and emulation of a specific data intensive application on the field of business intelligence. In section 5, we summarize our work and conclude with an outlook on future development.

2. State-of-the-art and Related Work

There are several tools to examine I/O performance on HPC systems. They differ in pattern coverage (wide range of access patterns or only specific workloads), evaluated intefaces (POSIX, MPI-IO, or both), or test scenario (synthetic tests or emulation of specific, real I/O workloads). We summarize prominent examples as follows.

Popular I/O benchmarks for local and network file systems like IOZone [1] or FileBench [2] are tools with synthetic I/O access patterns. Generally they are not relevant to HPC applications as they either do not have parallel implementations, or they exercise access patterns that are not common in that application field.

The IOR [3] benchmark is also a synthetic benchmark that supports POSIX and MPI-IO. POSIX handles files primarily as a stream of bytes, while MPI-IO offers mechanisms that allow transparent representation of file regions as a data type. IOR exercises concurrent read/write on one file or on separate files (one-file-per-processor). The benchmark is highly parameterized and can mimic a wide variety of access patterns. However, it is difficult to relate the data collected from it back to the original application requirements [5].

b_eff_io [4] brings together several file access patterns. Similar to IOR, the benchmark makes the test configuration precise by using different parameter groups [6, 7], but its main requirement is to give a statement about I/O performance after a defined time period to reduce the time spent on a production system. Even if b_eff_io is a powerful benchmark, adding new access patterns to the framework was found to be troublesome. It is also difficult to relate its performance back to applications, or to compare results with other available benchmarks.

In comparison to previous tools the MADbench2 [5] benchmark is an application-derived approach. Derived from a scientific application on the field of Cosmic Microwave Background data analysis, it allows studies to be made of the architectural-system performance under realistic I/O demands and communication patterns. MADbench2 emulates only this particular I/O behavior, and can only be used for testing applications with similar I/O requirements and therefore it can not be used generally.

Although it is in an early stage of implementation, the file system I/O Test Program BWT [8] provides a possibility to adjust testing use cases for POSIX I/O. The parameters for executing tests have to be specified in input files and cover the majority of file system access patterns. Although this program is a step forward to provide a standardized test suite, its support of parallel I/O commands is limited: process coordination is done via the barrier command, which is implemented by using IP multicast. Published shortly after our project was started, it provides a close approach to *Parabench*.

Existing testing tools differ in their design goals and tested file access patterns. Test tools, based on synthetic patterns, are well portable, but they do not reflect the real I/O workload. Some of them even provide obsolete patterns. In contrast, application driven benchmarks work with real access patterns, but are designed for a specific workload

and are hard to change. Concluding, there is still a lack of portable benchmarks working with application specific access patterns. The main contribution of this paper is therefore the development of a portable I/O benchmarking tool *Parabench*, which input is easy adaptable to several application needs and which results are easy to compare. The designed benchmark programming language allows easy modeling of a wide range of input patterns, that the benchmark can be easily adapted to resemble application behavior, for which the system is to be used.

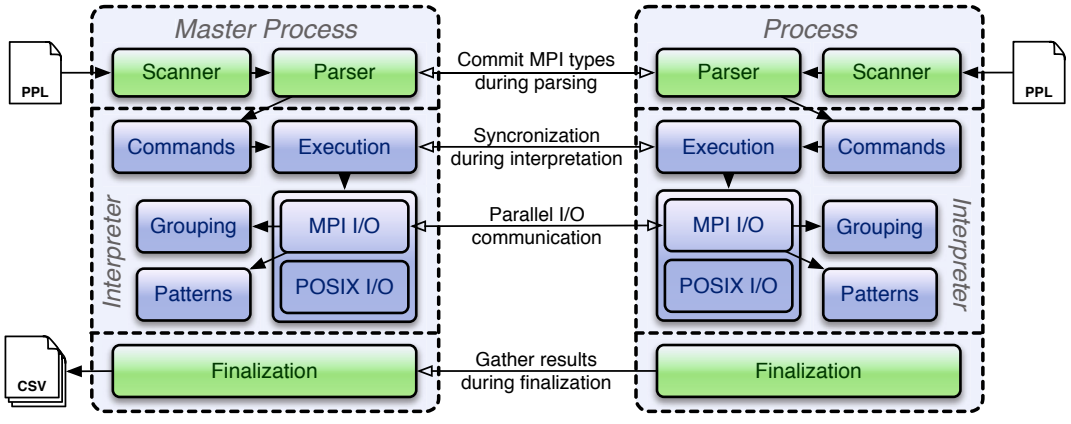
3. Parabench

3.1. Benchmark Design

To achieve our design goal, development of a portable and easy-to-use benchmarking tool with adjustable test patterns, we implemented *Parabench* as an interpreter of access patterns in the C programming language. Since we wanted to create a tool, that enables a wide range of flexibility, we decided to design our own programming language – the *Parabench* programming language (PPL). PPL covers different I/O patterns and provides support for MPI and parallel I/O on cluster and parallel computers. To process PPL, beside the interpreter layer there is also a layer for language lexical analysis (scanner), and a layer for its grammatical analysis (parser). The scanner, based on the open source scanner Flex [9], recognizes lexical patterns and reduces them to tokens. The parser receives a sequence of tokens from the scanner and generates a parse list for the interpreter. To make the design less complex, and because *Parabench* has only linear grammar constructs, we hold the output as a list and not as a tree. We build the parser with the open source parser generator Bison [10]. Finally, the interpreter processes the generated parse list. We implement the interpretation by using structures from the Gnome Library [11]. Because of this modularity in design the tool becomes more clear and easily upgradeable.

Figure 1 shows design and functioning of *Parabench*. The input files, composed in PPL, are processed and dumped to the result files in CSV[1] format, which can be post processed by any analysis tool. There is also the *Parabench* communication scheme with MPI in figure 1, when several benchmark processes are in use. In the multiprocess scenario, all *Parabench* processes execute commands equally and the master process gathers the results. This communication scheme is chosen to make output also to the prompt possible.

Figure 1: *Parabench* design and *Parabench* communication, when several clients are involved



[1] Comma-Separated Values

3.2. Benchmark Programming Language

To achieve flexibility and extensibility we designed the *Parabench* programming language (PPL) to control benchmark workflow and design necessary access patterns. Listing 1 demonstrates the PPL grammar definition, which is done according to the syntax of the popular in software development Go programming language [12]. Single braces ({) indicate that the rule can be repeated many times and quoted literals ("{") represent plain text tokens in the programming language. `String` is a regular expression for a standard C-type string, while `Variable` and `Number` are expressions for integers or variables defined either by user or locally. For the `IntegerExpression` statement we can use any arithmetic expressions in infix notation like `+ - * / ^ %`. The `Expression` is a placeholder for any kind of expressions. Dots (...) indicate that further definitions are spared for readability reasons and the `String` type parameter for the `barrier` statements is used to refer to process groups.

Listing 1: Simplified PPL grammar definition

```
Program = Defines StatementList

UserVariable = "$"[A-Za-z][A-Za-z0-9]*
InternalVariable = "$$"[A-Za-z][A-Za-z0-9]*
Variable =  UserVariable | InternalVariable

GroupTag = Number
SubgroupTag = Number
Group = String ":" GroupTag "/" SubgroupTag
GroupList = Group { "," Group }

Defines = { DefineGroups | DefineParameters | DefinePattern }
DefineGroups = "define groups" "{" GroupList "}"
DefineParameters = "define params" String Variable String
DefinePattern = "define pattern" "{" String, Number, Number, Number "}"

Expression = IntExpression | StringExpression
IntExpression = Number "+" Number | ...
StringExpression = String "+" String | ...

Statement = RepeatStatement | TimeStatement | GroupStatement | Assign |
   Command | Function
StatementList = { Statement }
Block = "{" StatementList "}"

Label = "[" String "]" | /* empty */

RepeatStatement = "repeat" Variable IntExpression Block
TimeStatement = "time" Label Block | "time" Label Statement
GroupStatement = "group" String Block
Assign = Variable "=" Expression ";"

ParameterList = Expression { "," Expression } | /* empty */

Command = CommandIdentifier "(" ParameterList ");"
CommandIdentifier = "read" | "write" | ...

Function = Variable "=" FunctionIdentifier "(" ParameterList ");"
FunctionIdentifier = "fopen" | "pfopen" | ...
```

We classify two groups of the *Parabench* programming language constructs: I/O language constructs and auxiliary language constructs.

3.2.1. I/O Language Constructs

The I/O language constructs provide a basis for a proper access pattern composition and can be classified into POSIX I/O [13] and parallel I/O, for which MPI-IO [14, 15] is used. For both interfaces we distinguish explicit file handles i.e. when an open and close needs to be issued manually, and implicit file handles i.e. when the user just specifies file or folder names (see table 1). The provided constructs for explicit file handles are prefixed with `f`.

Table 1: *Parabench* I/O language constructs

POSIX I/O	

implicit handle	`read(<filename>, <size>, [<offset>])`
	`write(<filename>, <size>, [<offset>])`
	`append(<filename>, <size>)`
	`rename(<filename>, <filename>)`
	`create(<filename>)`, `delete(<filename>)`
	`lookup(<filename>)` , `stat(<filename>)`
	`mkdir(<dirname>)` , `rmdir(<dirname>)`

explicit handle	`<handle> = fopen(<filename>, <mode>)`
	`fclose(<handle>)`
	`fread(<handle>, <size>, [<offset>])`
	`fwrite(<handle>, <size>, [<offset>])`

MPI I/O	
implicit handle	`pread(<filename>, <mode>, <pattern>, [<group>])` `pwrite(<filename>, <mode>, <pattern>, [<group>])`

explicit handle	`<handle> = pfopen(<filename>, <mode>, [<group>])`
	`pfclose(<handle>)`
	`pfread(<handle>, <pattern>)`
	`pfwrite(<handle>, <pattern>)`

File data accessible for each process, can be specified in a pattern. Internally an MPI file view is set on the file according to the pattern specification. Currently we support the array data type, i.e. each process gets a chunk of data in round-robin fashion. Also, only individual file pointers are implemented. In the future we will extend the support.

As shown in table 1, the parallel read and write commands expect a pattern as parameter, which describes how the parallel I/O is achieved. This pattern is defined using `define pattern` construct by an identifier name, the number of iterations, the number of elements and the level of parallelism, which can be *level 0* (non-collective calls, contiguous data access), *1* (collective calls, contiguous data access), *2* (non-collective calls, non-contiguous data access) or *3* (collective calls, non-contiguous data access). Listing 2 gives an example of pattern definitions, each with 100 iterations and $10 * 1024 * 1024$ elements per process for a *level 3* parallelism. As a result, each process gets a 10 MB part of the file, since one element has the size of 1 byte. The total file size depends on the number of processes N, attending in the read or write call ($N * 10$ MB).

To create more complex test programs, processes can be grouped and commands can be executed on process groups. Groups can be defined with the construct `define groups`. It allows having single groups (neither other groups overlap nor ungrouped processes are part of) and disjoint groups of processes by specifying an option after definition. Disjoint groups can also have disjoint subgroups. In the listing 2, the option `:D` defines two disjoint groups "writers" and "readers". The size of each group is set by command line options. The `barrier` construct is used for synchronization of processes or process groups.

3.2.2. Auxiliary Language Constructs

The auxiliary language constructs are `repeat`, `time` and `print`. Control flow statement `repeat` enables execution of code blocks in a loop and reduces replication in the test specification. The construct `time` allows time measurement of every command of the programming language. These commands have a special syntactic role, since they can be used as a prefix of every other command or as a code block. For easier tracking of timing commands, it is possible to assign text labels. The auxiliary construct `print` controls text output on stdout. Listing 2 demonstrates a small parallel test program, where all groups of PPL constructs are used.

Listing 2: Exemplary parallel program in PPL

```
define groups {"writers":D, "readers":D};
define pattern {"100x10MiB", 100, (10*1024*1024), 3};

group "writers" {
  time["write"] repeat 100 append("file$$rank", 10*1024*1024);
  time["pwrite"] pwrite("pfile", "100x10MiB");
}

barrier;

group "readers" {
  $fh = fopen("file$$rank", "r+");
  time["read"] repeat 100 fread($fh, 10*1024*1024);
  fclose($fh);
  time["pread"] pread("pfile", "100x10MiB");
}
```

4. Evaluation

To show *Parabench* usage and its ability to work with different I/O interfaces we performed two experiments: a simulation of I/O behavior of a state-of-the-art online analytic processing engine SAP NetWeaver BW Accelerator [16] over POSIX, and synthetic I/O tests with MPI-IO.

Our testbed was the cluster systems of the Research Group Parallel and Distributed Systems of the Ruprecht-Karls-Universität Heidelberg[2]. This is a 32 bit Ubuntu 8.04 cluster consisting of 7 nodes with COTS components and Gigabit Ethernet interconnect. For a qualitative evaluation precise hardware details are unnecessary and thus spared.

4.1. OLAP Engine over POSIX

Systems for online analytic processing (OLAP) need to handle growing volumes of time-critical data and also deliver fast response times for complex or ad hoc queries. Modern memory-based OLAP engines are designed to meet these challenges. By holding and processing data in main memory, they can execute queries over large volumes of structured data and reliably deliver subsecond response times. OLAP engines do not belong to the actual HPC area, but this is a data intensive (data intensive supercomputing [17]) application. To achieve desirable performance, OLAP engines like [16] provide a distributed server architecture and run on a cluster. As basis for persistence [16] uses reliable HPC storage technologies like network attached storage (NAS) or storage area network (SAN) with a cluster file system (e. g. GPFS [18]). Another persistence option is an integrated distributed persistence layer with proper redundancy and high availability mechanisms [19]. This distributed persistence layer runs on a node's local file system. We used this configuration on ext3 for our experiment, to show the POSIX-I/O capabilities of *Parabench*.

The minimum logical units of operation of the engine are indexes [20]. Indexes are semantically linked and hierarchically organized in namespaces. The most interesting I/O operations are: creation and deletion of indexes,

[2] `http://pvs.informatik.uni-heidelberg.de`

and indexing – the process of filling an index with data. All these operations have complex I/O patterns reflecting application semantics. We could analyze them using the application's built-in tracing capability of the engine's data server [19]. The relevant information was exported to files and by means of a parser script converted to the PPL test programs, used for the following tests.

During index creation several directories and files are created, in which application relevant data is written. Table 2 shows the lower I/O operations during creation and deletion of 100 indexes, measured in IOop/s (I/O operation per second) on a single host.

Table 2: Performance for creating and deleting of 100 indexes on a single host [IOop/s], emulated with *Parabench*

Write	Append	Read	Lookup	Delete	Mkdir	Rmdir	Create	Rename
2900	1700	500	300	3300	401	200	1200	800

Table 3 presents results of indexing a huge data cube [20] in a small cluster with 7 hosts. Each data server had an overall volume of approx. 3 GiB, where 279155 write, 117317 read and 1485 meta data operations have been performed by all hosts together. We modeled the load imbalances of the 7 different data servers by using one group for each process. Since we traced every data server individually, each of them had its own set of instructions, which were executed concurrently.

Table 3: Distributed indexing performance with 7 hosts, emulated with *Parabench*

Data Server	Open/Close [IOop/s]	Write [IOop/s]	Read [IOop/s]	Delete [IOop/s]	Rename [IOop/s]	Mkdir [IOop/s]	Total Time [s]
Host 1	433	297	136	1	1	0.20	145.962
Host 2	607	416	191	1	1	0.05	103.637
Host 3	595	408	187	1	1	0.07	105.841
Host 4	527	406	121	1	1	0.06	97.340
Host 5	612	420	192	1	1	0.05	102.180
Host 6	662	458	204	3	1	0.06	101.655
Host 7	601	412	189	1	1	0.05	51.901

This experiment shows the POSIX interface support of *Parabench*. We also see its capability in modeling such complex patterns as OLAP engines and in providing a basis for performance analysis and performance optimization. An obvious advantage of such a modeling is the possibility to port constructed tests to other systems without installing the application itself and taking care for necessary test data. Furthermore, it makes testing and comparison of proprietary systems like SAP NetWeaver BW Accelerator [16] possible.

In this development stage, there are still some questions about the validity of the presented simulation. We consider, that by executing the test with the same parameters (same access patterns with same parameters and amount of processed data, same order, same client number) the validity is given implicitly. To evaluate the 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 OLAP 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.

4.2. MPI-IO

For demonstrating the MPI-IO capabilities, we measured parallel reads and writes with different numbers of processes and file access levels. The used test cluster is set up with PVFS2 [21], where three nodes are configured as data and meta data servers. We executed *Parabench* on the remaining four nodes to have disjoint sets of nodes acting as I/O and client nodes.

We tested two different patterns, where the data in memory and in the files is contiguous. Each process writes and reads 100 MiB of data with the granularities $E * S$, where E represents the number of elements and S the size of one element. The granularities `102400*1KiB` and `10*10MiB` were tested. Listing 3 shows an excerpt from the PPL program, used to create the MPI-IO performance example. First, all parallel I/O patterns are defined followed by parallel writes and reads for both patterns in *level 0* to *level 3* of parallelism.

Listing 3: Synthetic MPI-IO test program in PPL

```
define pattern {"102400*1k-lvl0", 102400, (1 * 1024), 0};
... // Code for level 1 to 3 pattern define spared
define pattern {"10*10m-lvl0", 10, (10 * 1024 * 1024), 0};
... // Code for level 1 to 3 pattern define spared

$p = "pvfs2://pvfs2";

barrier;
// Parallel write for pattern "102400*1k-lvl0"
time["Write 102400*1k L0"] pwrite("$p/lvl0", "102400*1k-lvl0");
... // Code for level 1 to 3 parallel write spared

barrier;
// Parallel read for pattern "102400*1k-lvl0"
time["Read 102400*1k L0"] pread("$p/lvl0", "102400*1k-lvl0");
... // Code for level 1 to 3 parallel read spared

barrier;
// Parallel write for pattern "10*10m-lvl0"
time["Write 10*10m L0"] pwrite("$p/lvl0", "10*10m-lvl0");
... // Code for level 1 to 3 parallel write spared

barrier;
// Parallel read for pattern "10*10m-lvl0"
time["Read 10*10m L0"] pread("$p/lvl0", "10*10m-lvl0");
... // Code for level 1 to 3 parallel read spared
```

For each labeled `time` command *Parabench* creates one result file. Listing 4 exemplifies the results of the first `time` command with the label `"Write 102400*1k L0"` and 16 processes. The first number represents the process rank, followed by the `time` command ID and the wall clock time spent while executing `pwrite` for the pattern `"102400*1k-lvl0"`.

Listing 4: Result file generated by *Parabench* for 16 processes and label `"Write 102400*1k L0"`

```
0 42 750.102450
1 42 853.492694
2 42 867.392600
3 42 828.677785
...
14 42 863.404965
15 42 804.127985
```

The results of the tests described above are illustrated in figures 2 and 3. We can see the impact of different access levels on pattern granularities for both parallel read and write using MPI-IO. There is also an effect of two-phase I/O, write-back and caching, since for simplicity we did not do cold testing. To analyze the results, we imported the generated result files into an spreadsheet application and created the graphs below. In the future we plan to develop a visualization functionality to enable easy and straightforward evaluation of benchmark results.

5. Conclusion and Future Work

We developed a programmable pattern based benchmarking tool for local and cluster file systems. It supports POSIX and parallel I/O. Since it includes an own programming language, *Parabench* allows easy access pattern adjustment. Therefore, this benchmark can be geared toward a rich variety of high-performance computing applications.

Figure 2: MPI write performance with patterns `102400*1KiB` (left) and `10*10MiB` (right)

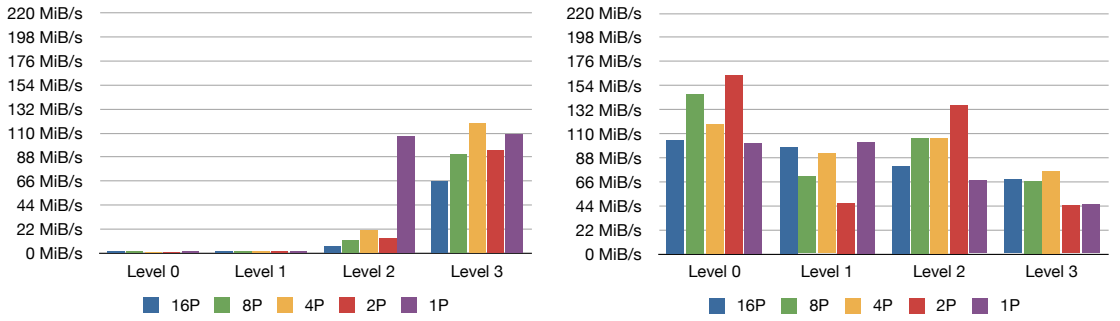
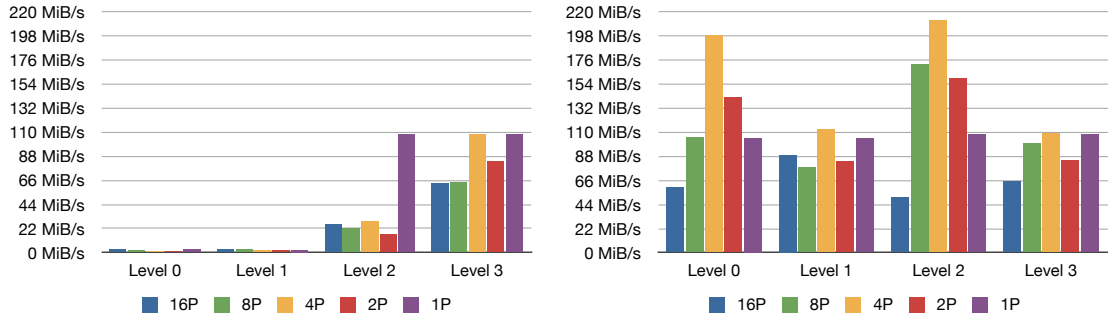


Figure 3: MPI read performance with patterns `102400*1KiB` (left) and `10*10MiB` (right)



We also demonstrated the usage of *Parabench* and its capabilities to handle different I/O interfaces and emulate real application access patterns.

In the future we will extend the pattern support for truly parallel I/O, for example to allow easy modeling of non-contiguous data and holes in files. This enables us to create more complex patterns with a small amount of basic patterns. There are also more language constructs we want to provide, which will make it possible to mimic even more complex control flows of real applications. A `sleep` command to cause certain processes to wait, a `master` command to let only the master process of a certain group execute special code, or a more advanced timing functionality to eliminate interpreter overhead are just examples here. Another issue is making our benchmarking tool more user-friendly. The current *Parabench* version requires a test file according to the PPL grammar as input. As output it provides ASCII files, for which other post processing and visualization tools currently have to be applied. To make the input easier and less error-prone and to automate the output analysis, a graphical interface has to be developed. Furthermore, we plan to establish an open platform, where patterns can be exchanged and their results can be evaluated and compared.

Acknowledgment

The authors would like to thank the members of the SAP development team TREX, in particular to Christian Bartholomä and Oleksandr Shepil for their technical advices in studing OLAP I/O, and to Roland Kurz and Arne Schwarz for their organizational support. We also thank to Max Gerashchenko and Andrew Ross for their help in improving numerous details in this paper.

References

[1] IOzone Filesystem Benchmark, http://www.iozone.org/.
[2] FileBench, http://hub.opensolaris.org/bin/view/Community+Group+performance/filebench.
[3] The ASCI I/O Stress benchmark, http://www.llnl.gov/asci/purple/benchmarks/limited/ior/.
[4] b_eff_io Benchmark, https://fs.hlrs.de/projects/par/mpi/b_eff_io/.
[5] J. Borrill, L. Oliker, J. Shalf, H. Shan, Investigation of leading HPC I/O performance using a scientific-application derived benchmark, in: SC '07, ACM, 2007, pp. 1–12.
[6] R. Rabenseifner, A. E. Koniges, Effective file-I/O bandwidth benchmark, in: Proc. of Euro-Par '00, Springer-Verlag, 2000, pp. 1273–1283.
[7] R. Rabenseifner, A. E. Koniges, Effective communication and file-I/O bandwidth benchmarks, in: Proc. of PVM/MPI '01, Springer-Verlag, 2001, pp. 24–35.
[8] Filesystem IO Test Program BWT, http://people.web.psi.ch/stadler_h/.
[9] Flex, http://flex.sourceforge.net.
[10] Bison, http://www.gnu.org/software/bison/.
[11] Gnome Library, http://library.gnome.org.
[12] The Go Programming Language, http://golang.org/.
[13] IEEE POSIX Certification Authority, http://standards.ieee.org/regauth/posix/.
[14] Message Passing Interface Forum, MPI: A message-passing interface standard. Version 2.1 (June 2008).
[15] W. D. Gropp, E. L. Lusk, R. B. Ross, R. Thakur, Using MPI-2: Advanced features of the message passing interface, in: CLUSTER, IEEE Computer Society, 2003.
[16] A. Ross, SAP NetWeaver BI Accelerator, Galileo Press, 2009.
[17] R. Bryant, Data-intensive supercomputing: The case for DISC, Tech. rep., CMU (2007).
[18] F. Schmuck, R. Haskin, GPFS: A shared-disk file system for large computing clusters, in: Proc. of FAST '02, USENIX Association, 2002, pp. 231–244.
[19] O. Mordvinova, O. Shepil, T. Ludwig, A. Ross, A strategy for cost efficient distributed data storage for in-memory OLAP, in: Proc. IADIS Int. Conf. Applied Computing 2009, Vol. I, IADIS Press, 2009, pp. 109–117.
[20] T. Legler, W. Lehner, A. Ross, Data mining with the SAP NetWeaver BI Accelerator, in: Proc. of VLDB '06, VLDB Endowment, 2006, pp. 1059–1068.
[21] PVFS2, http://www.pvfs.org/.