pCFS vs. PVFS: Comparing a Highly-Available Symmetrical Parallel Cluster File System with an Asymmetrical Parallel File System

Paulo A. Lopes and Pedro D. Medeiros

CITI and Department of Informatics,
Faculty of Science and Technology, Universidade Nova de Lisboa
2829-516 Monte de Caparica, Portugal
{pal,pm}@di.fct.unl.pt

Abstract. pCFS is a highly available parallel, symmetrical (where nodes perform both compute and I/O work) cluster file system that we have designed to run in medium-sized clusters. In this paper, using exactly the same hardware and Linux version across all nodes we compare pCFS with two distinct configurations of PVFS: one using internal disks, and therefore not able to provide any tolerance against disk and/or I/O node failures, and another where PVFS I/O servers access LUNs in a disk array and thus provide high availability (in the following named HA-PVFS). We start by measuring I/O bandwidth and CPU consumption of PVFS and HA-PVFS setups; then, the same set of tests is performed with pCFS. We conclude that, when using the same hardware, pCFS compares very favourably with HA-PVFS, offering the same or higher I/O bandwidths at a much lower CPU consumption.

Keywords: Shared-disk cluster file systems, performance, parallel file systems.

1 Introduction

In asymmetrical file systems, some nodes must be set aside as I/O or metadata servers while others are used as compute nodes, while in symmetrical file systems all nodes run the same set of services and may simultaneously perform both I/O and computational tasks.

In this paper, we argue that symmetrical cluster file systems using shared disks over a SAN may, in medium-sized clusters (with, say, a few dozens of nodes) achieve levels of performance comparable with those of asymmetrical parallel file systems such as PVFS [1] and Lustre [2].

Cluster file systems (CFS) such as GFS [3], and OCFS [4] have been strong in IT, data centre, and other non-HPC environments that require high-availability (HA) file systems that can tolerate node failures. However, these CFSs are known not to exhibit high performance, a situation that can be traced down to their objectives, design and attempt to retain the so called POSIX single-node equivalent semantics [5]. We deem that today, where even in a single node computer running Linux' `ext3` with processes read/write sharing a file the `write()` is not atomic with respect to the

P. D'Ambra, M. Guarracino, and D. Talia (Eds.): Euro-Par 2010, Part I, LNCS 6271, pp. 131–142, 2010.

`read()`, the application programmer should not rely on the POSIX single-node semantics but use appropriate mechanisms such as file locking.

We have proposed [6, 7] and implemented a highly available Parallel Cluster File System (pCFS) in a way as to maintain clusterwide coherency when sharing a file across nodes with very low overhead, provided that standard POSIX locks (placed with `fcntl` calls) are used [8]. Currently, pCFS is implemented through modifications to GFS' kernel module with the addition of two new kernel modules per node, and a single cluster-wide user level daemon (no modifications whatsoever have been made to the Linux VFS layer). We are able to provide exactly (within 1%) the same performance as GFS [16] for files that are read-shared across nodes, and large performance gains when files are read/write (or write/write) shared across cluster nodes, for non-overlapping requests (a quite common pattern in scientific codes). These gains are noticeable higher for smaller I/Os: when write/write sharing files across nodes, GFS starts at 0.1 MB/s for 4KB records and slowly rises up to 34 MB/s for 4MB records, while pCFS starts at 55 MB/s and quickly rises to 63 MB/s; CPU consumption is about the same, with pCFS at 4% and GFS at 3.5% per node.

In this paper, using exactly the same hardware and operating system version across all nodes we compare pCFS with two distinct configurations of PVFS – one using internal disks, and thus not able to provide any tolerance against disk and/or I/O node failures, and another where PVFS I/O servers access LUNs in a disk array and therefore can, in the event of failures, have their services restarted in another node in a matter of minutes (in the following named HA-PVFS).

2 Paper Organisation

The remainder of this paper is organised as follows. In section 3 we present pCFS' concepts and architecture, and how it was designed to deliver both high availability and high bandwidth; we also introduce the region lock concept and how it should be used by the application programmer. Section 4 describes the rationale for the tests, test methodology, and the benchmarks we have developed. Section 5 describes the infrastructure we used to test all file systems we have evaluated against pCFS: NFS, PVFS and GFS; we also report the peak and sustained bandwidths for the network and array subsystems, as well as CPU consumption recorded for these tests. Section 6 reports benchmark results for PVFS, HA-PVFS and pCFS, thus preparing for the conclusions and future work, drawn in section 7.

3 The pCFS Proposal

As stated in the introduction, pCFS strives to take advantage on the reliability of cluster file systems, starting at the hardware level with the use of disk arrays and continuing with the adoption of a symmetrical architecture that allows any node equal access to file system volumes. pCFS' main ideas include:

- Fine-grain locking [8], whereby processes running across distinct nodes may define non-overlapping byte-range regions in a file (instead of the whole file) and access them in parallel, reading and writing over those regions at the infrastructure's full speed (provided that no major metadata changes are required).

- Cooperative caching, a technique that has been used in file systems for distributed disks [9] but, as far as we know, was never used either in SAN based cluster file systems or in parallel file systems. As a result, pCFS may use all infrastructures (LAN and SAN) to move data.

3.1 Programming with pCFS

Our proposal for pCFS requires that programming should not deviate from the use of the standard POSIX API; in fact, we merely propose a few additional option flags for the `open()` call, to cater for the cluster-wide sharing intents, and a new way of looking at the semantics of existent locking primitives [8]. Both were the result of some observations on currently available file systems, namely that there is no way of specifying the degree of sharing for a file at open time in the POSIX API (which in a distributed file system, results in all sorts of tricks being used to implement it), and that our region concept is closely related to the one of mandatory byte-range locks.

The introduction of these flags was carried out without violating our design premises, namely "no VFS changes": all code was confined to the GFS layer (fortunately Linux does not check all flag combinations and these flags do "flow in"). And, furthermore, they have a very important side effect: the user may choose between GFS or pCFS behaviour just by omitting or including these flags, i.e., `open(…,stdflags)` will trigger a GFS open while, e.g., `open(…,stdflags|O_CLSTSOPEN)` will trigger a pCFS open. The simplicity of this process is also highly beneficial to the debugging and benchmarking tasks.

3.2 pCFS Prototype Implementation

Developing from scratch a new shared disk file system with the proposed features would be a huge task, unattainable in the realm of our small group; conversely, a feature-light implementation would preclude a fair head-to-head comparison against other file systems. Therefore, we decided to build on the work in [6], keeping GFS as the basis for our prototype. The term "prototype" clearly states we're not aiming either a full or a production-quality implementation; our primary objective is to show that pCFS, while retaining GFS' strengths, can efficiently support HPC applications,

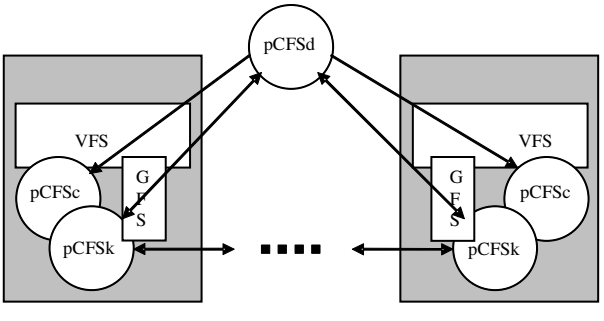

Fig. 1. Current pCFS architecture showing two nodes and their module interconnections

so the prototype specifically target regular files accessed through the usual read/write and other POSIX calls, eschewing direct I/O and memory-mapped operations. Furthermore, no modifications were made in code paths that handle directories, special files, journaling, etc., i.e., no attempt was made to speedup metadata operations (such as file creation, deletion or lookup).

4 Tests and Testing Methodology

The rationale for the set of tests we will perform is the following: in a very crude statement, this work is about access patterns that arise in typical HPC applications when processes, running in different nodes, do share files.

4.1. Introduction

Carrying out short, easily reproducible, and yet meaningful tests is therefore of primary importance; but, regrettably, popular I/O benchmarking applications cannot be used here; as an example, we refer two widely used ones: Bonnie++ and IOzone; the former was designed to test file system performance of single node architectures. However, IOzone [10] can be used on multiple nodes, and furthermore has an option, `-W`, briefly referred in the documentation as "lock files when reading or writing"; unfortunately, looking at the program's source code, we found that it uses `fcntl()` calls with arguments to lock/unlock the file as a whole, so it is worthless for us.

Another option is to use real applications; for example, an MPI application such as one we have developed in-house to process tomography images: it accesses the image file in big, disjoint regions, for reading and writing. However, we cannot use it (yet) for pCFS testing, as usage of MPI over pCFS requires a new ROMIO [17] driver (to cater for pCFS open extensions, etc.).

4.2. The Benchmarking Application

So the only remaining option left was to develop our own benchmarking application, and that's just what we have done. It is a fairly simple application, composed of a `controller` and a set of `exerciser` programs.

The `controller` runs in a node and accepts a string as its sole parameter; the string is a sequence of characters, `S` and `P`, which specifies that an exerciser should be fired (along with others) either sequentially (`S`), or in parallel (`P`). A few examples are: `SSS`, where three exercisers will be fired in sequence; `SPPS`, where a first exerciser will start and, when it finishes, two will be fired in parallel; then, when they are both done, a fourth one will be run.

An exerciser is an I/O program that reads or writes; it accepts as arguments the file size, the buffer size, the total number of exercisers that will be used in the test, and its `id` number. There are six versions of the exercisers; we'll just show the reader's list, the writers being symmetrical: there's a simple reader (`rdr`), a reader which performs full region locks before it starts reading (`rdr-lck`), and a reader which performs a per-record lock/read/unlock sequence (`rdr-sml-lck`).

This application can be used to exercise a broad range of situations, such as modelling I/O behaviour from parallel applications; for example, when a MPI application performs I/O over NFS, the ROMIO library uses a per-call lock/read/unlock sequence that we can accurately reproduce with the `*-sml-lck` exercisers. We can also, to some extent, simulate multiple file access by streaming over file regions that are very far from each other (the drawback is that simulation on a single file does not properly exercise the metadata part: for reading, it may profit too much from metadata caching while, for writing, there may be some lock contention); however, in our tests, we do not try to simulate accesses to multiple files.

5 Test Infrastructure

pCFS, PVFS and PVFS-HA tests were carried out in a configuration with six IBM x335 nodes, each with 4 GB of memory and two Intel Xeon processors; nodes 1, 2 and 3 had them running at 2.6 GHz while nodes 4, 5 and 6, which were also connected to the Fibre Channel (FC) SAN, had them running at 3.06 GHz. We have characterised all major subsystems: in node GbE NICs (Broadcom 5703) and FC HBAs (QLA-2200F), switches (GbE SMC 8624T and FC IBM 3534-F08) and disk array (IBM FAStT-200), measuring both bandwidths and CPU consumption as seen by nodes. All nodes were running CentOS Linux 2.6.18-92.1.18.el5.

GbE Network. The network was exercised with `netperf` [11]; in summary, we have, for the "slow" (2.66 GHz) nodes a TCP bandwidth of 975.5 Mb/s, a CPU usage of 37.6%, and a rate of 16.2k interrupts per second (IPS) issued by the NIC. For the "fast" (3.06 GHz) nodes both the TCP bandwidth, at 975.4 Mb/s, and the interrupt rate, at 16.9k IPS, are quite similar, the difference being the CPU usage, at 27.9%.

FC infrastructure and disk array. We wanted to assess several configurations, trying to get the best base level one to supports the typical HPC environment – large files, often accessed sequentially or in segmented mode (each node accessing different regions). Tests were carried out with a program we have developed ourselves because widely used file benchmark applications such as, again, IOzone, did not provide the features we needed, such as the ability of using direct I/O on raw devices.

Our application performs as follows: a) it starts by sequentially reading 32MB from the raw device opened with `O_DIRECT` to bypass the Linux page cache; b) for each data size, a cache-fill run is executed – and this also touches the page-aligned pages in the user buffer, preparing it for the next page fault free runs; c) the file is re-read with a given record size – typically starting at 4KB and going up to, at least, 1MB – and each run is separately timed; finally, size increased by 1MB (or 2MB for larger file sizes) and the above steps are repeated. Our tests show that the array does not degrade its bandwidth when using both Storage Processors (SP) in parallel; in fact, each SP is able to deliver a maximum of 75MB/s from its cache, and each disk contributes with a sustained bandwidth of 45MB/s. The array's total is fine at 150 MB/s when reading from both caches (and we concluded that the usable cache is circa 70MB per SP) and sustains 90MB/s when reading from both disks in parallel, a good value for a low cost, entry level disk array. At the host we recorded, a maximum CPU usage of 1.6% and a peak of 50k blocks read per second per disk drive.

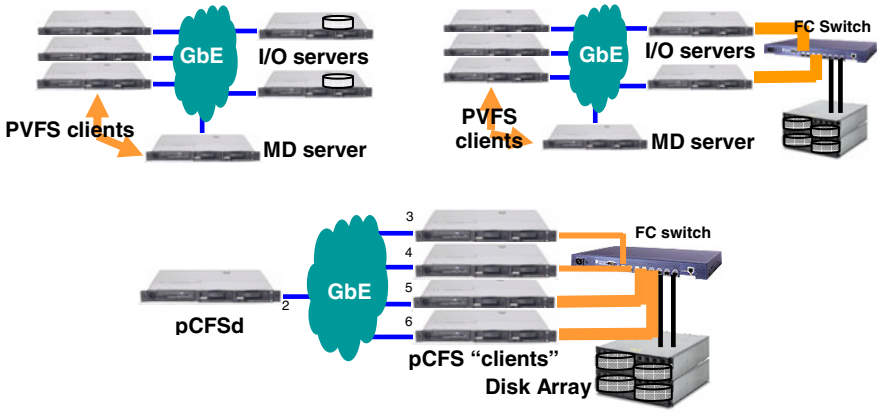

Fig. 2. Test configurations, all using the same hardware: PVFS is using internal disks (*left*), while HA-PVFS (*right*) and pCFS (*bottom*), are both using the external disk array.

File system test configurations. Fig. 2 shows the configurations we used to benchmark each file system. The same hardware and OS release was used throughout all tests; for both PVFS (we used version 2.7.0) and HA-PVFS, a total of six nodes was used, while for pCFS only five were used. Tests with the disk array used four disks, balanced across both SPs; the only slight imbalance is in the PVFS test with internal disks, where only two disks were used.

6 File System Tests: pCFS vs. PVFS

Results gathered in our tests are reported here, divided into two groups: the first one for reading tests and the second one for those tests were at least one node is writing. A single, large file (16 GB unless otherwise noted) is used, and all accesses are sequential, starting from a given offset, and moving forward – we call it accessing a segment, or region. All tests were run three times, and, unless otherwise noted, between runs the systems under test were rebooted, to clear their page caches. pCFS tests were run with region locks (both "big" and "small"), therefore guaranteeing data consistency; PVFS, however, does not need locking to access non-overlapping regions, so locks were not used – and, furthermore, PVFS does not yet support file locking on its POSIX API, which was the one used for all tests.

On each node, the exerciser process (reader or writer) is provided with a starting offset, record and region sizes; then, when fired, it will proceed accessing its file region. Readers may overlap their regions, but writers may not, even with readers.

6.1. Read-Only Tests

Fig. 3 below shows that bandwidth, both for pCFS and PVFS, increases with the number of clients; however it's easy to notice one of PVFS' weaknesses: absolute performance is quite sensitive to the record size used for I/O (see section 7, at the end, for a discussion on this issue).

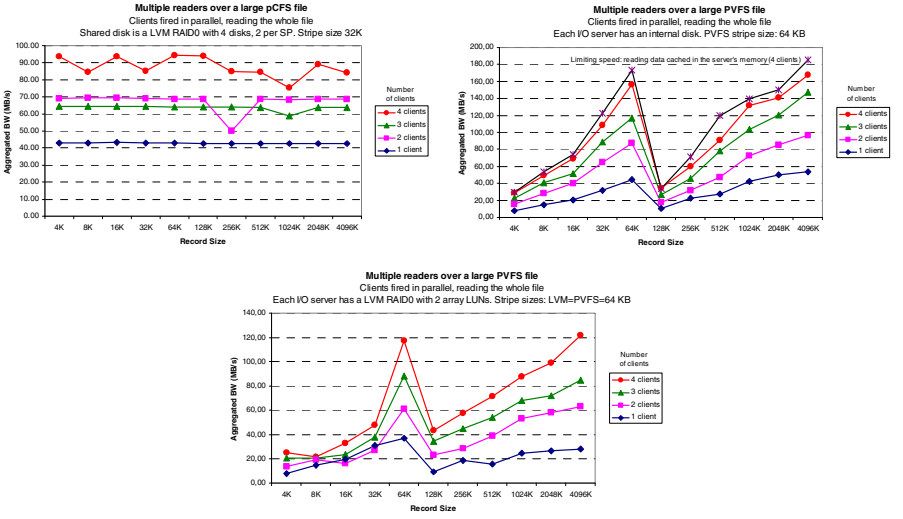

Fig. 3. Full file scan readers, large file: pCFS (*left*) PVFS (*right*) and HA-PVFS (*down, centre*)

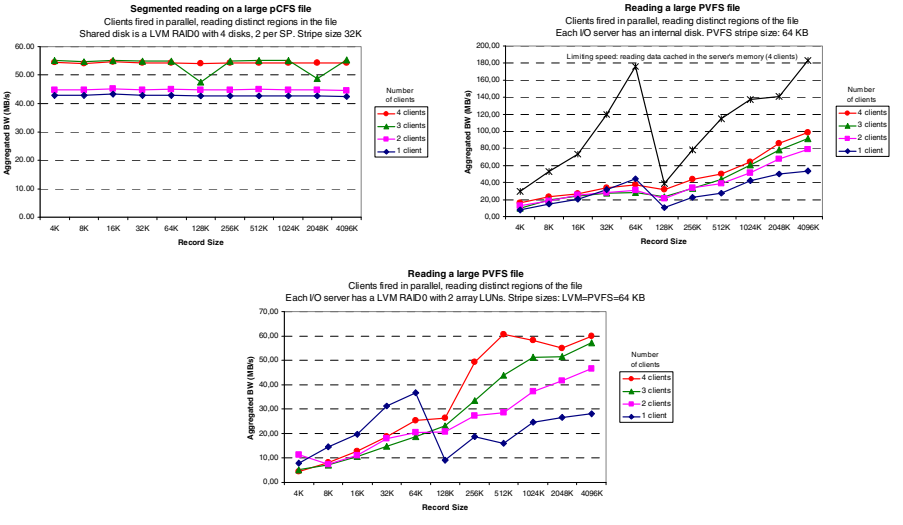

Fig. 4. Read sharing a large file with pCFS (*left*) PVFS (*right*) and HA-PVFS (*down, centred*). Disjoint segments are accessed by each node.

The results for segmented access, where the file is split into equally sized segments and each reader (in its node) accesses its own segment, are reported in Fig. 4. We see quite a performance drop caused by too much disk seeking, and we can now clearly observe the peak performance, designated “Limiting speed” in some graphs, that we can get when we read (or write) from (to) the PVFS I/O server’s cache (to get data for the cached BW plot, we read/wrote an amount of data which was small enough to fit in the node’s caches, and we did not reboot the systems between runs).

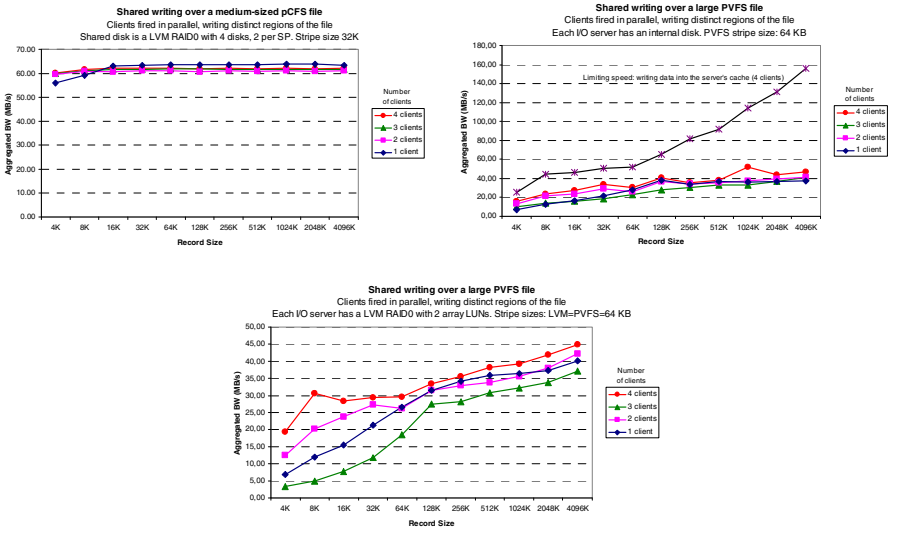

Fig. 5. Write sharing a file with pCFS (*left*) PVFS (*right*) and HA-PVFS (*down, centred*). Disjoint segments are accessed by each node.

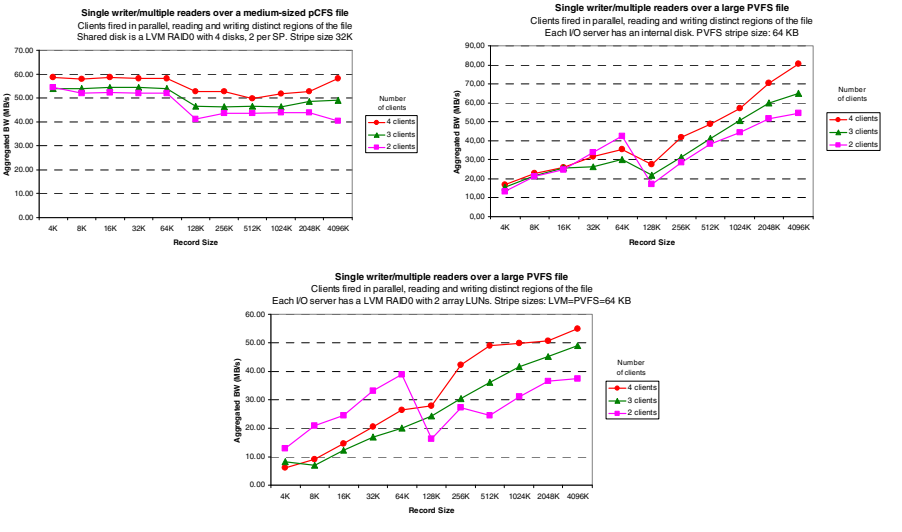

Fig. 6. Single writer/multiple readers with pCFS (left), PVFS (*right*) and HA-PVFS (*down*)

6.2 Write Tests

Tests reported here are those were at least one node is a writer; pCFS tests in Figs. 5 and 6 are performed with a single “big region” lock per process.

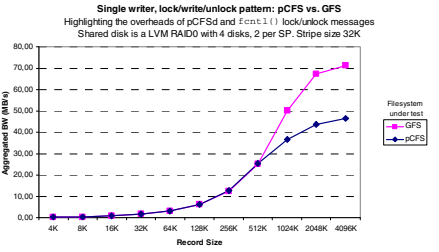

Fig. 7. Per-call lock/write/unlock pattern, showing that the current pCFS prototype not only does not improve, but may even lag behind standard GFS

While writing to large pCFS files we uncovered a bug related to memory (page cache) pressure, which triggers a kernel panic; so, we reduced the file size in order to keep the size of the region accessed on each node slightly less than the node's memory; to flag it, we insert the "medium sized pCFS file" label on the affected graphs (Figs. 5 and 6).

Looking at Figs. 7 and 8 we can explain why pCFS' performance is still low for small record I/O sizes when using a per-I/O call lock/unlock: for each call, there is Distributed Lock Manager (DLM [12]) traffic among all nodes, even when there is a single node writing over the file; the delay imposed by this communication pattern does create a start/stop effect that degrades the bandwidth.

We think a solution may be found within the region mechanism: we may use either a) dynamic regions that will cover large areas and contract when needed to fulfil some request, or b) elect one node as file owner (an extension of the mechanism we

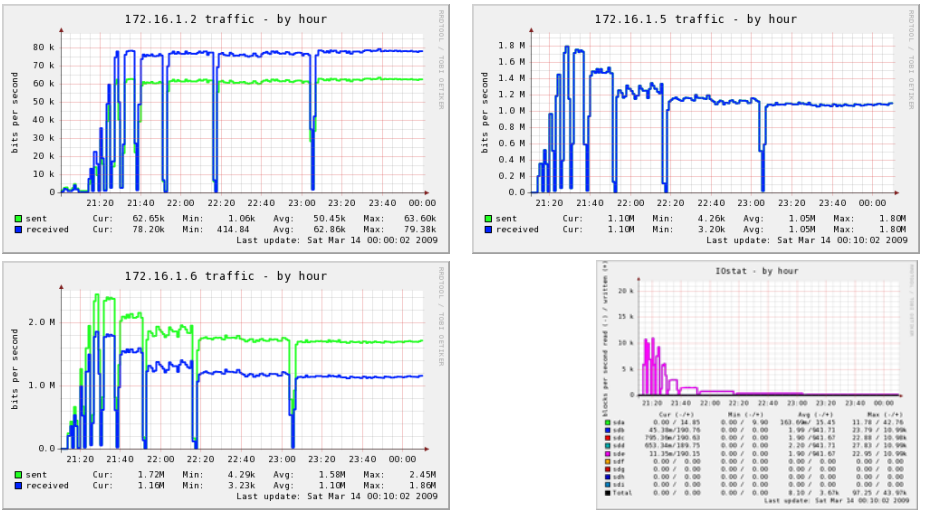

Fig. 8. Per-call lock/unlock – the root cause for low bandwidth is excessive DLM traffic on the node running pCFSd (*upper left*), on the other cluster nodes (*upper right*) and on the node where the file system volume is being accessed (*bottom left and right*)

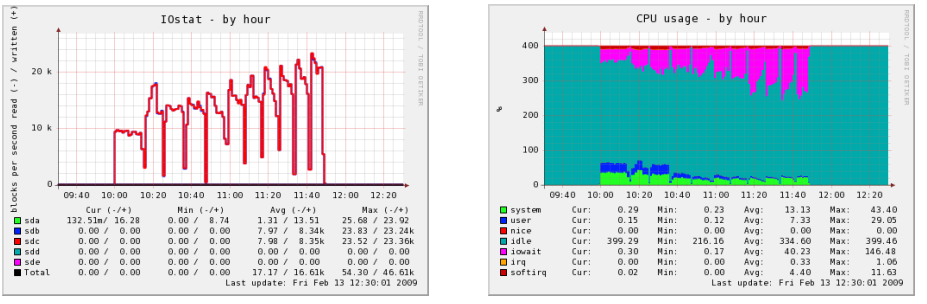

Fig. 9. PVFS CPU usage (*right*) and disk I/O requests issued (*left*) in a single I/O server for the four-writers segmented file access test

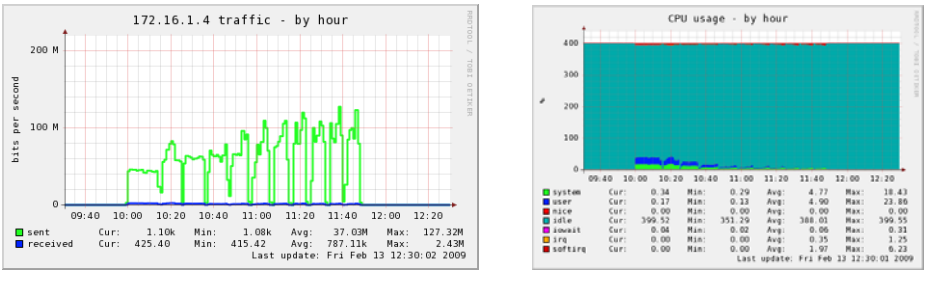

Fig. 10. PVFS CPU usage (*right*) and network bandwidth (*left*) in a single client running the four-writers segmented file access test

proposed and implemented to maintain cache coherence in pages that are shared between regions [8]) and force all other nodes to ship data to/from it.

6.3. pCFS vs. HA-PVFS: Resource Usage

The last step of this report on pCFS vs. PVFS is to present the Munin [13] graphs gathered during the four writer tests, which is the most resource-consuming test. We start with HA-PVFS and show disk I/O results and CPU consumption for a single server (out of two) in Fig. 9, and a single client (out of four) in Fig. 10.

Figures 9 and 10 show HA-PVFS' highest CPU consumption, reached when I/O size is small, at 4KB: each I/O server consumes almost a full CPU (92%), while each client uses 25% of its CPU for I/O chores; for more details see [16], where we also show that the PVFS metadata server uses 10% in this test. As for pCFS, it exhibits a maximum CPU usage of circa 4% per node, as there is no role separation into clients and servers. CPU usage is much less dependent on the I/O size, and so small as to deter us from including the graph.

7 Conclusions and Future Work

Our small test bench shows that when sharing a file using large regions pCFS does (except for the largest buffer sizes) provide a bandwidth which is comparable with

that offered by PVFS' POSIX interface while draining out a lot less CPU. It is, therefore, possible to run applications in the very same nodes that perform the I/O – as it should be for a symmetrical file system. pCFS' performance is quite smooth across the entire buffer size range (a contribution of VFS' read-ahead/write behind mechanisms), contrary to PVFS, which has very low performance for small buffer sizes (due to the latency of the client-request/server-response flow over the network) and may also exhibit performance valleys across the range of I/O sizes (which can be levelled increasing the number of PVFS I/O servers).

But one may (and should) pose a string of questions: a) what is the scalability of pCFS, and how well will it handle larger configurations? b) What would be the results, both for pCFS and HA-PVFS, if the disk array was not severely limiting bandwidth any more? c) How does MPI-IO over pCFS compares with PVFS' native MPI interface? And, for the cost-conscious, d) can we compare both in terms of cost/performance?

While we do not get hold of a larger cluster and a better disk array, our answers to (a) and (b) may only be based on knowledge about GFS configurations being used out in the field, as well as on pCFS' design strengths: first, GFS is commercially supported in clusters with a maximum of 16 nodes, but has been reported to run well in larger configurations; furthermore, pCFS does provide an efficient locking protocol which is based on GFS' but supplemented with very low overhead region locks [8], in fact reducing DLM pressure for HPC-style applications which access non-overlapping file segments while delivering very high bandwidths for data intensive, non write--sharing workloads. As for MPI-IO, only testing can confirm, but we believe that pCFS will maintain a good performance overall, its worst case being fine-strided I/O, where it will, at worst, perform like NFS. And, finally, as far as price/performance is concerned, a pCFS-based cluster should cost about the same as a similarly sized production PVFS site, as both need disk arrays to recover for storage and node faults.

pCFS is still a prototype; we are currently re-implementing pCFS module communications on top of TIPC [14] and improving performance for the small sized lock/access/unlock. A more ambitious goal is to fully implement the cooperative cache using the distributed shared memory approach provided by Kerrighed [15].

Acknowledgments. We thank IBM for the SUR grant that allowed us to carry out this work.

References

1. Carns, P., et al.: PVFS: A Parallel File System for Linux Clusters. In: Proceedings of the 4th Annual Linux Showcase and Conference, pp. 317–327 (2000)
2. Braam, P.: The Lustre storage architecture. Technical Report, Cluster File Systems, Inc. (2003), `http://www.lustre.org`
3. Soltis, S.: The Design and Implementation of a Distributed File System based on Shared Network Storage. PhD thesis, University of Minnesota Graduate School (1997)
4. Fasheh, M.: OCFS2: The Oracle Clustered File System, Version 2. In: Proceedings of the Linux Symposium (2006)

5. Schmuk, F., Haskin, R.: GPFS: A Shared-Disk File System for Large Computing Clusters. In: Proceedings of the Conference on File and Storage Technologies (FAST 2002), pp. 231–244 (2002)
6. Lopes, P., Medeiros, P.: pCFS: A Parallel Cluster File System. In: Proceedings of the International Conference ParCO 2005, pp. 515–522 (2005)
7. Lopes, P., Medeiros, P.: Cooperative Caching in the pCFS parallel Cluster File System. In: Proceedings of the 15th IEEE International Symposium on High Performance Distributed Computing (2006)
8. Lopes, P., Medeiros, P.: Enhancing write performance of a shared-disk cluster filesystem through a fine-grained locking strategy. In: Proceedings of the Second International Workshop on High Performance I/O Systems and Data Intensive Computing (HiperIO 2008), Co-located with the IEEE International Conference on Cluster Computing 2008 (2008)
9. Anderson, T., et al.: Serverless Network File Systems. ACM Transactions on Computer Systems 14(1) (1996)
10. Capps, D., et al.: Iozone Filesystem Benchmark, `http://www.iozone.org`
11. Netperf, `http://www.netperf.org`
12. Kronenberg, N., et al.: VAXclusters: A Closely-Coupled Distributed System. ACM Transactions on Computer Systems 4(2), 130–146 (1986)
13. Munin, `http://munin.projects.linpro.no`
14. The Transparent Inter-Process Communication (TIPC), `http://tipc.sourceforge.net`
15. Morin, C., et al.: Towards an efficient single system image cluster operating system. Journal of Future Generation Computer Systems 20(4), 505–521 (2004)
16. Lopes, P.: A Shared-Disk Parallel Cluster File System. PhD Thesis, Department of Informatics, Faculty of Science and Technology, Universidade Nova de Lisboa (2009)
17. Thakur, R., et al.: Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation. Technical Memorandum No. 234, Mathematics and Computer Science Division, Argonne National Laboratory (2004)