Position Papers of the Federated Conference on Computer Science and Information Systems pp. 133–140
DOI: 10.15439/2016F52
ACSIS, Vol. 9. ISSN 2300-5963

A Parallel MPI I/O Solution Supported by Byte-addressable Non-volatile RAM Distributed Cache

Artur Malinowski*, Paweł Czarnul*, Piotr Dorożyński*,
Krzysztof Czuryło†, Łukasz Dorau†, Maciej Maciejewski† and Paweł Skowron†
*Gdansk University of Technology, Gdansk, Poland
Email: artur.malinowski@pg.gda.pl, pczarnul@eti.pg.gda.pl, piotr.dorozynski@pg.gda.pl
†Intel Technology Poland Sp. z o.o., Gdansk, Poland
Email: {krzysztof.czurylo, lukasz.dorau, maciej.maciejewski, pawel.skowron}@intel.com

Abstract—**While many scientific, large-scale applications are data-intensive, fast and efficient I/O operations have become of key importance for HPC environments. We propose an MPI I/O extension based on in-system distributed cache with data located in Non-volatile Random Access Memory (NVRAM) available in each cluster node. The presented architecture makes effective use of NVRAM properties such as persistence and byte-level access behind the MPI I/O API. Another advantage of the proposed solution is making development of a parallel application easy and efficient as a programmer just needs to use the well known MPI I/O data model and API while efficient file access is automatically provided without a need for application level optimizations like avoiding frequent operations on a small data. Results of experiments obtained with three different applications suggest, that the extension significantly reduces file access time, especially for small I/O operations. By locating cache facilities on computing nodes, the extension decreases load of file system servers and makes I/O scalable.**

I. Introduction

SIZES of high performance computing systems are steadily growing. The currently most powerful cluster Tianhe-2 on the TOP500[1] list features 3120000 cores and 1,024,000 GB total memory. It should also be noted that while clusters are larger and larger and potentially allow for higher speed-ups, there are more and more cores and nodes involved in processing and the probability of failure increases. From this point of view, especially in terms of data processed by such applications, there is a need for reliable and large storage solutions that would support execution of such applications. The Message Passing Interface (MPI) standard [1], [2] includes an MPI I/O part that specifies an API for a parallel application to read and write a single file from many processes. Firstly, the API allows both reading/writing data from individual processes or in a collective manner. Secondly, it allows using explicit offsets, individual or shared file pointers.

Within this paper, motivated by possibility of better I/O performance thanks to NVRAM in HPC environments, we propose wrappers over selected MPI I/O API functions using distributed persistent memories in cluster nodes and then compare the performance of the proposed solution with hardware-based simulation of persistent memory to performance of the same MPI application using OrangeFS in a real cluster environment.

The research in the paper was supported by a Grant from Intel Technology Poland.

[1]http://top500.org/system/177999

II. Related Work

While great effort is put into increasing computational power of supercomputers, many data-intensive applications suffer from insufficient I/O operations performance. Speeding up access to storage devices by applying best practices widely proposed in data centers [3], [4], [5] introduces additional overhead for development process, connected with tuning up the application both for each MPI implementation and Parallel File System (PFS). This leads to the conclusion, that a reasonable way would be to apply a generic solution that is suitable for many applications.

Many performance oriented MPI I/O solutions base on the idea of sieving, prefetching and caching data in RAM. ROMIO, a popular MPI I/O implementation, introduces Two-Phase I/O – an algorithm that attempts to merge non-contiguous requests into larger and more contiguous [6]. Tsujita, Y. et al. obtained remarkable improvements by extending Two-Phase I/O even further, by using multiple threads [7]. Other researchers are focused on new MPI I/O implementations [8] or improvements in PFS [9][10]. The extensions of this kind, however, do not consider possibilities offered by emerging hardware technologies.

A significant group of proposed PFS improvement ideas, that could be easily customized to benefit from NVRAM properties, concerns, among other things, cooperative caching. In 1994, Michael D. Dahlin et. al. prepared a survey of different cooperative caching algorithms and showed performance impact of their incorporation into file systems [11]. The described caching strategies are the base for many modern approaches. Our solution differs from others e.g. zFS file system [12] or Novel Distributed Memory File System [13] in management strategy, as we want to avoid central management because of its poor scalability while increasing the number of cluster nodes. Several papers have proposed extending MPI I/O with cooperative caching algorithms that do not rely on central entity. AHPIOS [14] is an ad-hoc file system, that can be used alternatively to popular PFS implementations. Its main features are tight integration with an MPI application (the client application communicates with a file system using an MPI communicator), minimal configuration and a single instance of global registry with size reduced by keeping metadata minimal. On the other hand, there is no easy method to access created files outside of MPI. Our solution may seem to have a lot in common with another research project, with the mechanism described by Wei-keng Liao et. al. [15], but partially different assumptions (mainly limited amount of memory dedicated to cache storage) led to other technical details. Differences particularly involve splitting cache into pages replaced in our approach by using constant blocks, a complex mechanism of locking unnecessary in our architecture because of request queuing, and a single thread responsible for handling multiple files – our solution serves multiple files simultaneously by taking advantage of a single thread per file.

Another topic, that is also important for parallel, especially long running, applications is checkpointing [16][17]. An application can save its work on a disk or in persistent memory and consequently it can restart from the last known state in case of a failure. I/O bandwidth is an important factor in reducing execution time. In 2013, Rajachandrasekar et. al. proposed CRUISE – in memory file system that speeds up checkpointing [18]. In this system, each write request data is initially stored in a pre-allocated persistent memory region, and after flushed to a PFS or a local file system asynchronously. Performance results presented by the authors were really promising, nevertheless, checkpointing has different characteristics than operating on a file while performing computations, therefore it cannot be applied to our solution. In many cases, the main disparity is connected to a single process operating on a single file, what reduces complexity of routines responsible for simultaneous accesses. Other differences in assumptions, that make checkpointing optimizations inadequate in our solution, are: strong spatial locality of requests (accessed data is rather continuous), usually central management of a checkpoint and focusing much more on optimizations of output operations.

I/O operations are strongly related to storage hardware. In 2009, Mark H. Kryder and Chang Soo Kim evaluated several memory technologies that are expected to be an alternative for hard disk drives (HDD) in 2020 [19]. Some investigated solutions have interesting properties such as non-volatility and random access (NVRAM), fast read/write access time and high density (which affects final capacity of a device) while the price could be still reasonable compared to HDD. In 2015, Intel Corp. and Micron Technology unveiled 3D XPoint – non-volatile memory technology expected to be up to 1,000 times faster than NAND, 10 times denser than DRAM with latency of tens of nanoseconds and possible to be used as system memory [20], [21]. With declared relatively low price and expected market release in 2016 [22], 3D XPoint announcements show, that NVRAM has a potential to be a true alternative for existing storage technologies soon.

Many MPI I/O extensions benefit from particular storage hardware properties. Shuibing He et al. implemented Solid State Drive (SSD) cache that improved throughput of PFS [23][24], however, block data access and long latency of SSDs cause, that the solution is not able to benefit from all properties available in NVRAM. Evaluation of NVRAM role in data-intensive scientific applications was presented by Dong Li et al. [25] and – independently – Brian Van Essen et al [26], but papers are based on single node analysis and are focused rather on extending system memory (heap, stack, global data segments) than speeding up I/O operations in a distributed environment. Active NVRAM for I/O staging proposed by S. Kannan et al. [27], [28] benefits from NVRAM located within each computing node speeding access to PFS up. While this solution could be useful in the case considered in this paper, it does not fulfill our requirement of minimal application modification understood as keeping the proposed extension compatible with MPI I/O API. Moreover, the presented experimental results suggest, that this solution is not beneficial for small data sizes – we assume, that our extension is convenient for developers in the way it allows to access even very small data efficiently.

III. Motivations

In view of the existing solutions and recent developments in the area of non-volatile RAM, new solutions could be proposed for parallel applications that could potentially increase both performance and ease of development of applications processing potentially large data sets. Specifically:

1) Performance. It is possible to use a collection of distributed persistent memories in cluster nodes as an additional layer of cache between an underlying file system and an application. It can serve as an intermediate layer able to store large data sets (larger than in the combined RAM of cluster nodes) with persistence and possibility to recover from persistent memory should a failure occur. Thanks to the relatively low latency of persistent memory, this should allow a solution with better performance than traditional file systems, especially if an application would perform reads or writes to far away spaced locations in a file.
2) Ease of development/programming/data model. Such a solution with a proper API could in fact be regarded as a shared (distributed in the underlying implementation), large memory with persistence and, what is important, byte level access which is a property of persistent memory. While block level access would still yield better performance, even accesses using small blocks or even bytes could yield much better performance than traditionally used file systems for parallel applications.

IV. Proposed Solution

A. Design

This solution is not another MPI I/O implementation – the contribution of the paper is a set of wrappers over MPI I/O

API functions that creates in-app distributed cache between application and particular MPI I/O implementation. Source code is written in C using MPI, POSIX Threads API and the libpmem library responsible for low level NVRAM memory support [29]. Wrappers incorporate NVRAM usage behind the MPI I/O API.

The extension requires a specific architecture as shown in Fig. 1. We assume a cluster with interconnected nodes each of which allow running processes of an application in parallel. Each node must be equipped with its own NVRAM storage, where the cache data is stored. All computing nodes, as in the regular MPI I/O, must have access to a remote file using a distributed file system.

A considered file to be accessed using the MPI I/O API is split into n continuous parts, where n is the number of nodes. Each part is managed by a single, independent cache manager running on a dedicated thread as presented in Fig. 2. Cache managers are mainly responsible for:

- prefetching whole data part – prefetching all of the required data is possible due to the assumption that the size of a file is limited to the sum of all NVRAM capacities in a cluster;
- synchronizing data between cache and file system – occurs only when the file is being opened, closed or the synchronization is called explicitly by the application;
- serving read/write requests from all of the processes in application. Each process is able to determine which cache manager it should contact. Same file locations can be accessed by all processes, which, in case of write requests may require synchronization at the application level.

The proposed solution has no central management. Each process knows exactly which cache manager holds the data, so no additional entity, like dispatcher, is required. Because each cache part is continuous – instead of being split into blocks – metadata is kept to a minimum. The cache manager does not perform any staging optimization – each request is served as fast as possible by making use of NVRAM byte-addressing and low latency compared to HDD or SSD. Processing without a central entity allows to avoid potential bottlenecks, while simplifying data access, and, rather than introducing smart but costly data management, it saves CPU time, reduces latency and makes the solution more independent of specific data patterns.

Although the proposed extension is not a file system, it can serve multiple files simultaneously. Each opened file has its own part of allocated NVRAM, a dedicated thread for a cache manager and an MPI communicator. Required metadata (e.g. file path, file size, communicator handler) is stored within a separate file handler that is returned by a call to the `MPI_File_open` function.

A natural advantage of using NVRAM as a cache storage is its persistent character which is directly linked to the possibility to recover data after failure, but guaranteeing full data consistency in a distributed environment requires further investigation and will be the subject of our next research.

B. *Target applications*

As presented in Fig. 3a, the solution should be most beneficial in applications that access small data chunks (gain from byte addressing) from spread file locations (no drawback from omitting staging phase). Improved performance is a result of fast read and write accesses, but prefetching a large amount of data in the beginning and the need for writing the whole cache back at the time of closing file introduce overhead associated with initialization and de-initialization. This leads to the conclusion, that in order to perform better than the regular MPI I/O, an application has to access data frequently. As shown in Fig. 3b, it could be achieved either in very data-intensive applications, or in long running applications. However, many scientific applications meet these criteria.

Introducing the file size limitation is not an issue, because NVRAM capacity multiplied by number of nodes in modern clusters is expected to be enough for handling files of the sizes comparable to the SSD based solutions. Our extension is also scalable, so it is expected to perform well while increasing the number of processes or nodes. On the other hand, it should be kept in mind that the total number of processes in an application results in a certain number of processed served, on average, by each cache manager in a cluster node.

C. *Implementation*

Making use of proposed extension in an MPI application requires two minor changes in source code. The first one is including `file_io_pmem_wrappers.h` header that allows to transform each native MPI I/O function call into its NVRAM cache counterpart. Due to compatibility of function signatures, calls do not need modifications.

Configuration of the solution is prepared with `MPI_Info` parameters passed to `MPI_File_open`. A minimal configuration requires only one parameter, `pmem_path`, that points to an NVRAM device mounted in a local file system. `MPI_Info` parameters unrecognized by MPI I/O are ignored, so the cache could be switched on and off using an include directive.

The extension spawns additional POSIX threads (cache managers) that use MPI to communicate with the application. Therefore, initialization with `MPI_Init_thread` and `MPI_THREAD_MULTIPLE` support are needed. Algorithm 1 shows the idea behind implementation of the cache manager thread. The listing contains all MPI and NVRAM cache related calls. The thread is created within `MPI_File_open`.

An object, that represents a file opened with MPI I/O, is called a file handler. In our solution, the handler is considered as a pointer to `MPI_File_pmem_structure`. Although the object-oriented programming (OOP) paradigm is not natively provided in the C programming language, we used it by incorporating into structure both data together with a set of pointers to functions. Data stored in the structure includes:

- information about the file e.g. name and size,
- handlers responsible for communication with the cache (MPI communicator),
- cache metadata (number of cache nodes, file offsets handled by each cache manager),

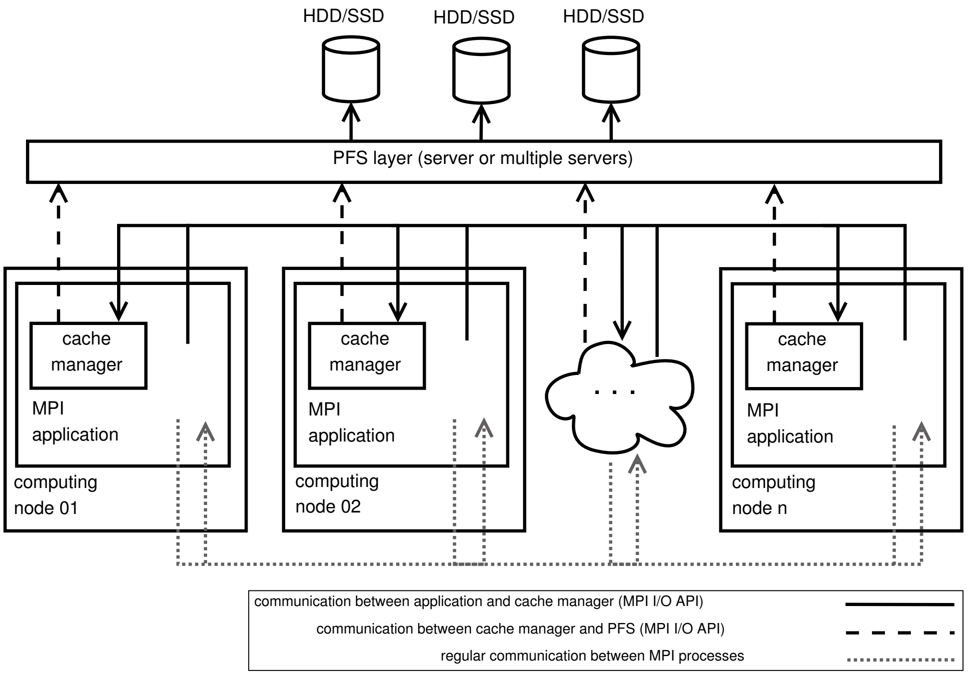

HDD/SSD
HDD/SSD
HDD/SSD
PFS layer (server or multiple servers)
cache manager
MPI application
computing node 01
cache manager
MPI application
computing node 02
. . .
cache manager
MPI application
computing node n
communication between application and cache manager (MPI I/O API)
communication between cache manager and PFS (MPI I/O API)
regular communication between MPI processes

Fig. 1: Architecture of the multi-node system that utilizes the proposed solution. MPI processes are not included in the diagram.

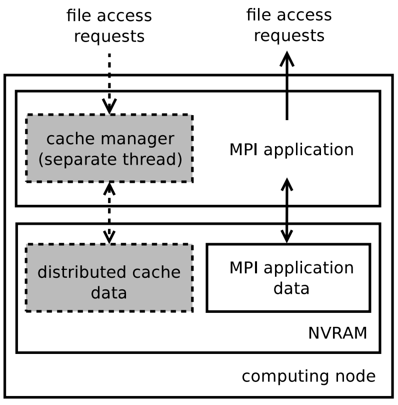


Fig. 2: Architecture of proposed solution within a single node. Gray components and dashed connections are transparent to MPI application developer

- some additional parameters e.g. related with failure recovery.

A set of pointers to functions contain a single counterpart function for each MPI I/O routine. Combining a file with functions allows for choosing different strategies for different files which could be potentially beneficial when extending the method further.

Source code of the solution released under the BSD license, software documentation and examples are available on GitHub platform [2].

[2]https://github.com/pmem/mpi-pmem-ext

V. Experiments

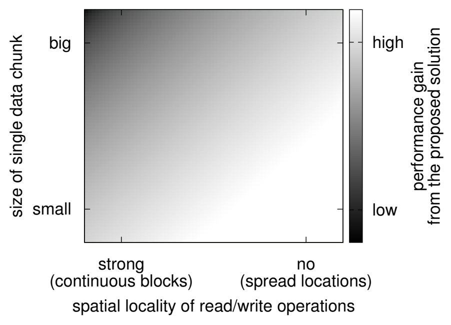
A. Testbed Environment

All of the tests were performed on an eight-node cluster, each node equipped with two Intel® Xeon® E5-4620 CPUs, as well as 32GB of RAM, storage on SSD + HDD together with both 10 Gigabit Ethernet (10GbE) and Infiniband connections.

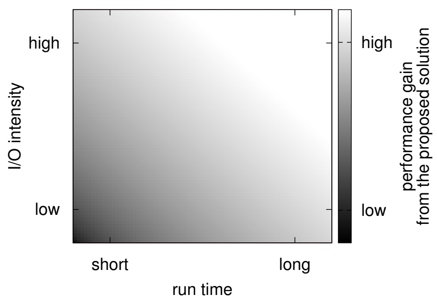
A single node was responsible for application execution, gathering the results and hosting a PFS server, the other 7 nodes for parallel application execution. OrangeFS 2.8.7 (former: PVFS2)[3] compiled with Infiniband support was chosen as PFS, because of relatively good performance [30]. MPICH 3.1.4 with ROMIO[4] was installed on seven computing nodes as an MPI IO implementation. OrangeFS stored both data and metadata on SSD. Nodes were communicating with each other using 10GbE, all of the Infiniband bandwidth was used to provide fast file access. In most experiments each computing node ran 15 processes – with 16 physical cores on a single node it left a spare core for a PFS thread.

RAM in each of seven computing nodes was split into two parts: regular system memory (15GB) and storage for NVRAM simulation (17GB). Amount of NVRAM memory does not influence performance, because the cache manager would use only as much NVRAM, as the size of its cache part. The NVRAM simulation part was visible in the operating system as an ext4 file partition using The Persistent Memory Driver and ext4 Direct Access (DAX)[5]. DAX provided a way

[3]http://www.orangefs.org/
[4]https://www.mpich.org/
[5]https://www.kernel.org/doc/Documentation/filesystems/dax.txt


size of single data chunk
big
small
high
low
performance gain from the proposed solution
strong (continuous blocks)
no (spread locations)
spatial locality of read/write operations

(a)



(b)

Fig. 3: Plots present properties of the applications, that potentially would benefit most from the proposed solution

to use NVRAM through file system omitting paging, caching etc., so we noted performance comparable to RAM. To obtain expected NVRAM properties, we used a hardware simulator configured with three parameters:

- latency – additional latency to access the data (default: 600ns),
- commit latency – time required to ensure that saved data is flushed on device (default: 2000ns),
- bandwidth (default: 9.5GB/s).

If it is not explicitly stated in the experiment description, the simulator was configured with default values.

B. Results

1) Rompio benchmark and tests: Rompio[6], software developed at Los Alamos National Laboratory, is a file I/O performance benchmark with MPI support. Rompio was chosen because it is able not only to provide a final bandwidth, but also intermediate values (i.e. time of opening or closing a file) useful in performance tuning.

Fig. 4 shows execution times of read and write operations separately, both for NVRAM cache based extension and regular MPI I/O. In this test, the proposed extension is better for small data chunks (up to 1024B), while the regular MPI implementation has better results for larger data.

2) Discussion: The reason for execution time growth that occurs in this case is related to the specific design of the benchmark i.e. the size of a file increases linearly with the size of data chunk, the number of operations and the number of processes. A significant part of execution time of the code using this solution is consumed on opening and closing the file, as it prefetches data into cache, so this extension benefits mostly for long-running applications with many read and accesses on an open file. While the benchmark is very configurable, it does not allow to use a fixed size of a file.

Bandwidth calculated with values that neglect time consumed by opening and closing the file is presented in Fig. 5 which shows much better values for the NVRAM based

[6]http://www.osti.gov/scitech/biblio/1231008-rompio

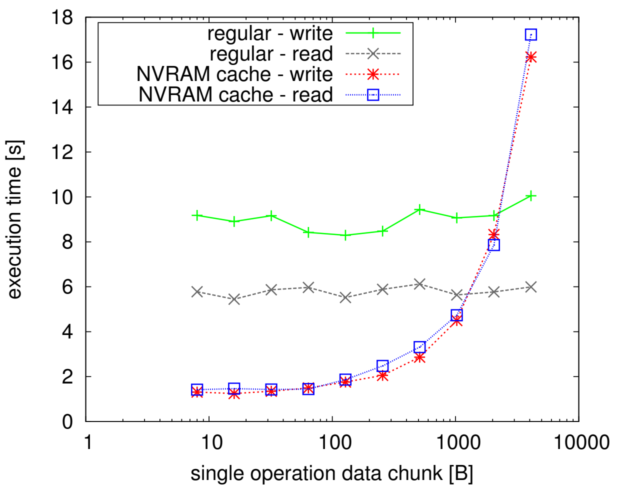


Fig. 4: Rompio benchmark execution time results

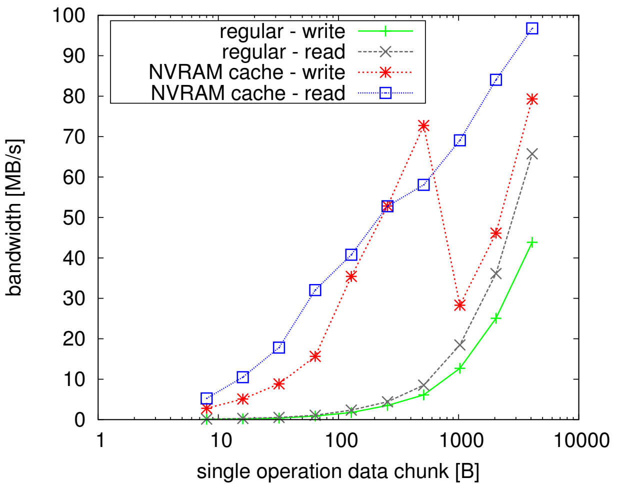


Fig. 5: Rompio benchmark bandwidth results

Algorithm 1 Cache manager routine

```
init_cache(); // allocate NVRAM memory

// prefetch part of file, that
// cache manager is responsible for
MPI_File_read_at();

while (true) {

  MPI_Probe();
  switch (probe_status.MPI_TAG) {

    case READ_AT_REQUEST_TAG:
      MPI_Recv(); // get read request
      read_from_cache();
      MPI_Send(); // send bytes from cache
      break;

    case WRITE_AT_REQUEST_TAG:
      MPI_Recv(); // get write request
      write_into_cache();
      cache_flush(); // flush into NVRAM
      break;

    case SYNC_TAG:
      MPI_Recv(); // get sync request
      MPI_File_write_at(); // flush
                           // into PFS
      break;

    case SHUTDOWN_TAG:
      MPI_Recv(); // get shutdown request
      MPI_File_write_at(); // flush
                           // into PFS
      deinit_cache();
      return;

    // another cases
  }
}
```

solution. As a consequence, in order to achieve a better overall execution time compared to a standard solution, as shown in Fig. 4, the NVRAM based proposed solution needs a high ratio of the time spent on read/write operations compared to the initialization/finalization time spent on open/close operations. The large bandwidth drop between data chunk of 512B and 1024B is caused by inefficiency of asynchronous writing. In the proposed extension, small write requests end immediately after being submitted and then the cache manager is performing an actual writing procedure. However, for constant requests frequency, writing bigger chunks consumes more time, so consecutive requests have to queue.

3) 2D map search and tests: 2D map search is a geometric SPMD type application for searching throughout a 2D map stored in a file. The goal of this application is as follows: search throughout the map for pixels that meet certain criteria and – after a pixel/object meeting a criterion has been found – a part of its immediate surrounding in a selected direction is searched up to a predefined radius or until a given number of pixels meeting another criterion is found. An exemplary application of such an algorithm may be searching for spreading of pollution in farmlands with wind blowing in a certain direction. The application can read the data byte by byte (naive approach, but fastest in development) or use block reading with blocks of a predefined size.

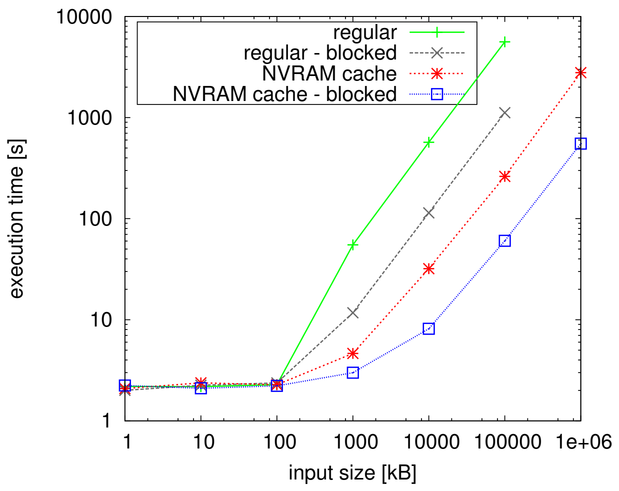


Fig. 6: 2D map search results according to input size (105 processes, 512B block size)

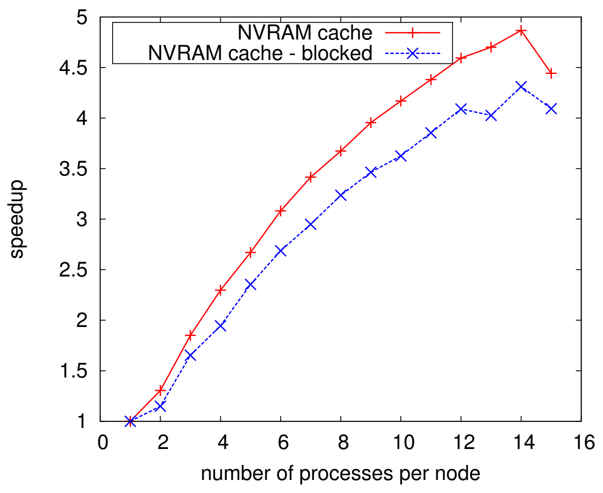


Fig. 7: 2D map search speedup according to number of processes per each of seven nodes (map size: 100MB)

Fig. 6 shows, that for this application the proposed extension performed better than regular MPI I/O, when the size of a file

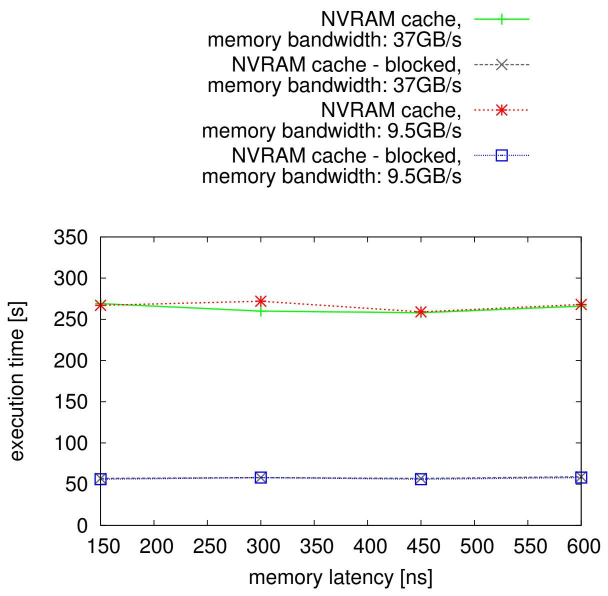


Fig. 8: 2D map search execution time with different NVRAM simulation platform configurations (map size: 100MB)

was greater than 100 KBs (execution time with smaller maps is determined by operations unrelated to I/O). Fig. 7 presents speedup according to the number of processes respectively. 2D map search does not perform any time consuming calculations, execution time is mainly based on I/O operations.

4) Discussion: With regular MPI I/O, the application does not scale because from the PFS perspective, the number and sizes of requests are constant. On the other hand, the proposed extension is scalable – each additional node reduces average load for a single node. Different NVRAM simulation platform configurations do not influence the performance, which is shown in Fig. 8. Taking into consideration file size 100MB, the number of nodes equal to 7, file size per node equal to $\frac{100MB}{7} \approx 14MB$, potential difference between latencies 450ns, for byte level access we can compute the overhead of $450ns \cdot \frac{14MB}{1B} \approx 6.3s$ which constitutes 2.4% of execution time. For 512B blocks we can compute a theoretical overhead of $450ns \cdot \frac{14MB}{512B} \approx 12ms$ while for reference for SSD with 512B block, $0.1ms \cdot \frac{14MB}{512B} \approx 2.7s$. In test runs, we did observe differences in times varying from run to run, in the order of this overhead, coming most likely from file system operations and consequently such overhead is not exposed in the chart.

5) Random walk microbenchmark and tests: A third group of experiments was performed with an application not as data-intensive as Rompio or 2D map search, created in order to check whether the solution could be useful in programs where file operations consume less amount of time compared to the application running time. This microbenchmark is a constrained version of a random walk algorithm. In each step a data chunk is read, the application performs some selected computations (about a million iterations of Collatz conjecture), and the chunk is written back.

6) Discussion: Results presented in Fig. 9 show, that if the ratio of read/write to open/close operations is relatively high, the solution performs better than regular MPI I/O. The dependence shows, that the potential target of the extension is not only a set of data-intensive applications that operate on relatively small data chunks. Programs of a less data-intensive character and operating on bigger data chunks can also benefit from the solution if only they run long enough to compensate the overhead for initialization and de-initialization of NVRAM cache.

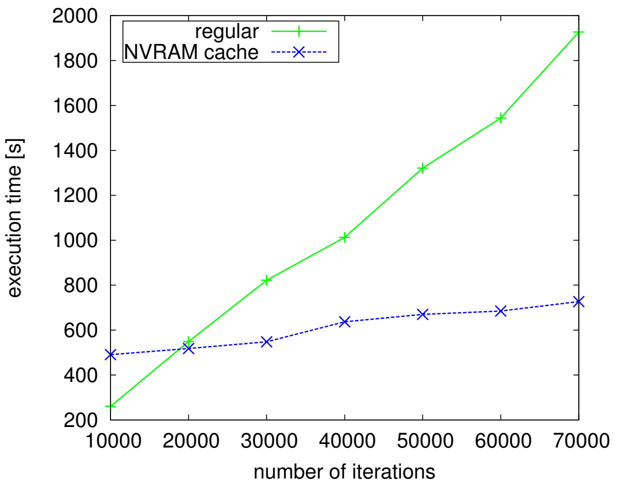


Fig. 9: Random walk microbenchmark execution time results (input file size: 10GB)

VI. Conclusions and Future Work

In this paper, we presented a new parallel MPI I/O solution implemented by our group, including implementation and tests, supported by byte-addressable non-volatile RAM distributed cache. We demonstrated improvements of I/O operations' performance in a cluster environment using NVRAM. We proposed an MPI I/O extension based on distributed cache in NVRAM, not only for improvement of performance, but also to make application the development process easier by allowing accessing small data chunks efficiently using the MPI I/O data model and API. The solution was tested on a cluster equipped with a hardware NVRAM simulator using three different applications: an MPI I/O benchmark, searching throughout a 2D map stored in a file and a microbenchmark based on a random walk algorithm combined with Collatz conjecture. The results confirmed, that in tested applications in a cluster with hardware simulated NVRAM the proposed solution significantly improves performance of small I/O operations, compared to a standard MPI implementation on a typical cluster without NVRAM.

In the nearest future we plan to extend the method further and test selected optimizations. The next step is to test the solution with more applications. At the time of writing, a simulation of tornadoes moving across an area is being prepared.

We also plan on using this approach for parallelization of processing of many images extending the work performed in [31] for parallelization of image processing within GIMP using an NVRAM-assisted MPI based solution. Although the proposed distributed cache is always persistent and can be recreated after a failure, additional set of tests, performance tuning and further research of data consistency are also planned.

References

[1] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard Version 3.1," June 2015, http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[2] W. Gropp, T. Hoefler, R. Thakur, and E. Lusk, *Using Advanced MPI: Modern Features of the Message-Passing Interface (Scientific and Engineering Computation)*. The MIT Press, 2014, ISBN 978-0262527637.

[3] P. Wautelet, "Best practices for parallel IO and MPI-IO hints," March 2015, http://www.idris.fr/media/docs/docu/idris/idris_patc_hints_proj.pdf.

[4] B. Hadri, "Introduction to Parallel I/O," October 2011, https://www.olcf.ornl.gov/wp-content/uploads/2011/10/Fall_IO.pdf.

[5] N. H.-E. C. Program, "Lustre Best Practices," August 2015, http://www.nas.nasa.gov/hecc/support/kb/lustre-best-practices_226.html.

[6] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective I/O in romio," *Frontiers '99 - Seventh Symposium On Frontiers Massively Parallel Computation, Proc.*, pp. 182–189, 1999. doi: 10.1109/FMPC.1999.750599. [Online]. Available: http://dx.doi.org/10.1109/FMPC.1999.750599

[7] Y. Tsujita, K. Yoshinaga, A. Hori, M. Sato, M. Namiki, and Y. Ishikawa, "Multithreaded Two-Phase I/O: Improving Collective MPI-IO Performance on a Lustre File system," *2014 22nd Euromicro Int. Conference On Parallel, Distributed, Network-based Processing (pdp 2014)*, pp. 232–235, 2014. doi: 10.1109/PDP.2014.46. [Online]. Available: http://dx.doi.org/10.1109/PDP.2014.46

[8] A. Hori, K. Yamamoto, and Y. Ishikawa, "Catwalk-ROMIO: A Cost-Effective MPI-IO," *2011 IEEE 17th Int. Conference On Parallel Distributed Systems (icpads)*, pp. 120–126, 2011. doi: 10.1109/ICPADS.2011.40. [Online]. Available: http://dx.doi.org/10.1109/ICPADS.2011.40

[9] F. Wang, Y. Chen, S. Li, F. Yang, and B. Xiao, "The design of data storage system based on lustre for {EAST}," *Fusion Engineering and Design*, pp. –, 2016. doi: 10.1016/j.fusengdes.2016.04.002. [Online]. Available: http://dx.doi.org/10.1016/j.fusengdes.2016.04.002

[10] S. A. Wright, S. D. Hammond, S. J. Pennycook, I. Miller, J. A. Herdman, and S. A. Jarvis, "Ldplfs: Improving I/O Performance Without Application modification," *2012 IEEE 26th Int. Parallel Distributed Processing Symposium Workshops & Phd Forum (ipdpsw)*, pp. 1352–1359, 2012. doi: 10.1109/IPDPSW.2012.172. [Online]. Available: http://dx.doi.org/10.1109/IPDPSW.2012.172

[11] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson, "Cooperative caching: Using remote client memory to improve file system performance," in *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI '94, 1994. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267638.1267657

[12] A. Teperman and A. Weit, "Improving Performance of Distributed File System Using OSDs and Cooperative Cache," *IBM Haifa Labs*, 2004.

[13] U. Karnani, R. Kalmady, P. Chand, A. Bhattacharjee, and B. S. Jagadeesh, "Design and Implementation of a Novel Distributed Memory File System," ser. Communications in Computer and Information Science, vol. 133, no. III, 2011. doi: 10.1007/978-3-642-17881-8_14 pp. 139–148, 1st International Conference on Computer Science and Information Technology, 2011, India. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-17881-8_14

[14] F. Isailă, J. G. Blas, J. Carretero, W.-k. Liao, and A. Choudhary, "AHPIOS: An MPI-Based Ad Hoc Parallel I/O System," in *Parallel and Distributed Systems, 2008. ICPADS'08. 14th IEEE International Conference on*. IEEE, 2008. doi: 10.1109/ICPADS.2008.50 pp. 253–260. [Online]. Available: http://dx.doi.org/10.1109/ICPADS.2008.50

[15] W.-K. Liao, K. Coloma, A. Choudhary, and L. Ward, "Cooperative Client-Side File Caching for MPI Applications," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 2, pp. 144–154, May 2007. doi: 10.1177/1094342007077857. [Online]. Available: http://dx.doi.org/10.1177/1094342007077857

[16] P. Czarnul and M. Frączak, *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 12th European PVM/MPI Users' Group Meeting Sorrento, Italy, September 18-21, 2005. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, ch. New User-Guided and ckpt-Based Checkpointing Libraries for Parallel MPI Applications,, pp. 351–358. ISBN 978-3-540-31943-6. [Online]. Available: http://dx.doi.org/10.1007/11557265_46

[17] P. Dorożyński, P. Czarnul, A. Malinowski, K. Czuryło, Ł. Dorau, M. Maciejewski, and P. Skowron, "Checkpointing of Parallel MPI Applications using MPI One-sided API with Support for Byte-addressable Non-volatile RAM," *Procedia Computer Science*, vol. 80, pp. 30 – 40, 2016. doi: 10.1016/j.procs.2016.05.295 International Conference on Computational Science 2016, June 2016, USA. [Online]. Available: http://dx.doi.org/10.1016/j.procs.2016.05.295

[18] R. Rajachandrasekar, A. Moody, K. Mohror, and D. Panda, "A 1PB/s File System to Checkpoint Three Million MPI Tasks," June 2013.

[19] M. H. Kryder and C. S. Kim, "After Hard Drives — What Comes Next?" *Magnetics, IEEE Transactions on*, vol. 45, no. 10, pp. 3406–3413, Oct 2009. doi: 10.1109/TMAG.2009.2024163. [Online]. Available: http://dx.doi.org/10.1109/TMAG.2009.2024163

[20] Intel Corporation, "Intel and Micron Produce Breakthrough Memory Technology," July 2015, http://newsroom.intel.com/community/intel_newsroom/blog/2015/07/28/intel-and-micron-produce-breakthrough-memory-technology.

[21] ——, "3D XPoint Technology Revolutionizes Storage Memory," July 2015, http://www.intel.com/content/www/us/en/architecture-and-technology/3d-xpoint-technology-animation.html.

[22] ——, "Introducing Breakthrough Memory Technology," July 2015, http://www.intel.com/content/www/us/en/architecture-and-technology/non-volatile-memory.html.

[23] S. He, X.-H. Sun, and B. Feng, "S4d-cache: Smart Selective SSD Cache for Parallel I/O systems," *2014 Ieee 34th Int. Conference On Distributed Computing Systems (icdcs 2014)*, pp. 514–523, 2014. doi: 10.1109/ICDCS.2014.59. [Online]. Available: http://dx.doi.org/10.1109/ICDCS.2014.59

[24] S. He, Y. Wang, and X.-H. Sun, "Improving Performance of Parallel I/O Systems through Selective and Layout-Aware SSD Cache," *IEEE Transactions on Parallel and Distributed Systems*, 2016. doi: 10.1109/TPDS.2016.2521363. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2016.2521363

[25] D. Li, J. S. Vetter, G. Marin, C. McCurdy, C. Cira, Z. Liu, and W. Yu, "Identifying Opportunities for Byte-Addressable Non-Volatile Memory in Extreme-ScaleScientific applications," *2012 Ieee 26th Int. Parallel Distributed Processing Symposium (ipdps)*, pp. 945–956, 2012. doi: 10.1109/IPDPS.2012.89. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2012.89

[26] B. V. Essen, R. Pearce, S. Ames, and M. Gokhale, "On the role of NVRAM in data-intensive architectures: an evaluation," *2012 Ieee 26th Int. Parallel Distributed Processing Symposium (ipdps)*, pp. 703–714, 2012. doi: 10.1109/IPDPS.2012.69. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2012.69

[27] S. Kannan, A. Gavrilovska, K. Schwan, D. Milojicic, and V. Talwar, "Using Active NVRAM for I/O Staging," in *Proceedings of the 2Nd International Workshop on Petascal Data Analytics: Challenges and Opportunities*, ser. PDAC '11. ACM, 2011. doi: 10.1145/2110205.2110209. ISBN 978-1-4503-1130-4 pp. 15–22. [Online]. Available: http://dx.doi.org/10.1145/2110205.2110209

[28] S. Kannan, D. Milojicic, V. Talwar, A. Gavrilovska, K. Schwan, and H. Abbasi, "Using Active NVRAM for Cloud I/O," in *Proceedings of the 2011 Sixth Open Cirrus Summit*, ser. OCS '11. IEEE Computer Society, 2011. doi: 10.1109/OCS.2011.12. ISBN 978-0-7695-4650-6 pp. 32–36. [Online]. Available: http://dx.doi.org/10.1109/OCS.2011.12

[29] NVM Library team at Intel Corporation, led by Andy Rudoff, "pmem.io Persistent Memory Programming," http://pmem.io/nvml/libpmem/.

[30] J. M. Kunkel and T. Ludwig, "Performance evaluation of the PVFS2 architecture," *15th Euromicro International Conference On Parallel, Distributed And Network-based Processing, Proceedings*, pp. 509–516, 2007.

[31] P. Czarnul, A. Ciereszko, and M. Fraczak, "Towards efficient parallel image processing on cluster grids using gimp," in *Computational Science - ICCS 2004*, ser. Lecture Notes in Computer Science, M. Bubak, G. van Albada, P. Sloot, and J. Dongarra, Eds. Springer Berlin Heidelberg, 2004, vol. 3037, pp. 451–458. ISBN 978-3-540-22115-9. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-24687-9_57