Computer Science • 18(3) 2017 http://dx.doi.org/10.7494/csci.2017.18.3.2077

Artur Malinowski

USING REDIS SUPPORTED BY NVRAM IN HPC APPLICATIONS

Abstract *Nowadays, the efficiency of storage systems is a bottleneck in many modern HPC clusters. High performance in the traditional approach – processing using files – is often difficult to obtain because of a model's complexity and its read/write patterns. An alternative approach is to apply a key-value database, which usually has low latency and scales well. On the other hand, many key-value stores suffer from a limitation of memory capacity and vulnerability to serious failures, which is caused by processing in RAM. Moreover, some research suggests that scientific data models are not applicable to the storage structures of key-value databases. In this paper, the author proposes a resolution to the above-mentioned issues by replacing RAM with NVRAM. A practical example is based on Redis NoSQL. The article also contains three domain-specific APIs that show the idea behind transforming from an HPC data model to Redis structures as well as two micro-benchmarks results.*

Keywords high-performance computing, storage systems, NoSQL, NVRAM

Citation Computer Science 18(3) 2017: 287–300

1. Introduction

The computational power of supercomputers is constantly growing. Today, the most-powerful machine in the Top500 list (June 2016 edition) is composed of more than 10,000,000 cores and can achieve performance at a level above 90,000 Tflop/s [19]. Unfortunately, increasing the number of operations per second using CPUs and GPUs is not sufficient for providing a well-prepared environment for high-performance computations (HPC). Other elements that also play a key role in providing system efficiency are, for instance, network, failure prevention strategies, and – especially with often-data-intensive HPC applications – storage.

In a typical cluster environment, storage is provided by a parallel file system (PFS); e.g., General Parallel File System (GPFS), Lustre, or OrangeFS. Such a file system is usually distributed among several server machines to balance the load between them. With high-end hardware, a scalable network, and software customization, modern PFS can reach an aggregated transfer rate of 1Tb/s [16]; however, this still could be a bottleneck for many applications. Storage systems based on files are convenient for cluster administrators – a PFS compatible with POSIX could be mounted in Unix/Linux as a regular file system and accessed remotely when operating on files before and after computations. Modern solutions, like VeilFS [4] or OneData [20, 23], are even able to hide various storage systems located in different locations behind a common interface. Another advantage of using an PFS is its integration with libraries helpful in application development. Widely used in HPC, Message Passing Interface (MPI) includes an MPI I/O – parallel input and output API that covers PFS under the layer of functions that allows for the simultaneous accessing of files from multiple processes in a cluster [6]. MPI is only an API, but there are several mature implementations that offers highly optimized MPI I/O.

Many data structures are inefficient when stored in files, and their straightforward implementation could be the cause of a drop in performance. It is possible to create a fast implementation of operations, like dynamic changes in graph topology or adding/removing elements of a list, but this is usually connected with additional developer effort. An alternative for using files that also allow for sharing data between many processes on multiple nodes is a key-value store. Typically, this is a database that is responsible for the management of an associative array. Today, there are a plenty of implementations, many of which are focused on performance and ease of use. Although widely used in web applications or big data processing, it could be also applied in various HPC scenarios.

Some popular key-value stores are well-known for their great performance based on operating in volatile RAM. However, the capacity of RAM could be insufficient in many cases. Moreover, with its high possibility of failure occurrence and relatively high cost of computations, replacement of a file system with an in-memory database in HPC would introduce an additional risk of losing one's results. In a typical example, such a risk could be lowered by periodically saving the application state into persistent memory (checkpointing). In this situation, the final performance gain from storage

system replacement would be decreased by costly checkpointing. An ideal solution for this problem would be basing a key-value database on fast and capacious non-volatile random-access memory (NVRAM).

This paper aims to propose an alternative for the widely used file based storage systems in HPC applications. The motivation of searching for such a solution is as follows:

- storing data in files for more-complicated or dynamic data structures is neither convenient nor efficient,
- parallel file systems often create a bottleneck in HPC, so it is valuable to verify a different approach that also focuses on performance and scalability,
- today's well-known storage solutions are based on fast, volatile, size-limited RAM and slower block devices like SSDs and HDDs; however, we can expect new devices that could potentially eliminate some storage system weaknesses in the immediate future.

The research will cover the justification of using key-value storage in scientific computing, taking into consideration new system properties related to emerging memory technologies. The proposed architecture and practical examples will focus on Redis software supported by NVRAM. According to the author's knowledge, the combination of software and hardware components discussed in this paper is novel for HPC usage (as is its proof-of-concept validation). Finally, the author will prove that, in many cases, key-value storage is enough for providing a data, performing computations, and storing a result in HPC.

2. Related work

Key-value databases have recently become very popular; this is connected to the NoSQL phenomenon triggered primarily by Web 2.0 platforms, cloud computing, and big data processing [14]. As an alternative to relational database management systems (RDBMS), NoSQLs do not impose any structure for data and often do not support ACID (atomic, consistent, isolated, durable) transactions.

As a result, key-value NoSQLs are much more flexible, usually offer higher query speed, and provide high concurrency [8]. Studies show that combining HPC with grid and cloud computing is not only feasible [13], but applying NoSQL into an HPC environment could also be beneficial [9]. Typical examples of such stores are the Oracle NoSQL Database[1] and Redis[2].

One of the most-desirable features of the high throughput computing systems used in science is their scalability; this is meant as the capability of handling an increased load when providing bigger resources, typically by adding hardware. Practically,

[1] `http://www.oracle.com/technetwork/database/database-technologies/nosqldb/overview/index.html`

[2] `http://redis.io/`

this is often achieved by working in a distributed environment. Key-value stores that are also distributed are well-known in literature as distributed hash tables (DHT). Many DHTs have been proposed that have reported good performance. One of these (Tapestry) focuses on routing messages directly to the closest copy of an object or service [21, 22]. The system is decentralized, self-organized, and able to recover from failures by changing the route to the node that contains redundant data. Several applications use Tapestry (i.e., SpamWatch[3], a distributed spam-filtering system, or OceanStore[4], a distributed storage utility), but its use has not been reported in HPC.

Several DHT implementations were designed to fulfill the HPC needs. D1HT, a solution that opted to trade off latency for bandwidth, has been tested on a large environment that consists of 200 physical nodes. Results show that D1HT has low latency, high scalability, and could be used in many applications [15]. On the other hand, D1HT is not persistent; therefore, the data would be lost in the case of a major system failure (for instance, a power failure). The Zero hop Distributed Hash Table (ZHT), another DHT reported to work well with HPC applications, provides an option to store its state into persistent memory [12]. ZHT offers only four basic operations: `insert(key, value)`, `append(key, value)`, `lookup(key)`, and `remove(key)`, which makes the storage of more complex data structures inconvenient.

Some research also includes criticism of applying NoSQLs in HPC. In article *Scientific Computing Doesn't Need noSQL*, D. Buttler justifies the thesis included in the title by showing that some objects could not be easily transformed into NoSQL data model [2]. He mentions scalars, vectors, topology, algebra, and calculus operations as examples. While it is justified when using NoSQL directly, the author believes that many objects could be stored in such databases in a convenient way by using an additional layer with the API of a specific domain.

As stated in the introduction, databases operating on RAM could be insufficient in HPC, so there is a need for better memory technology. A survey prepared by M.H. Kryder and C.S. Kim suggests that one of the described NVRAM technologies is likely to succeed in 2020 [11]. Some researchers assumed that, in the immediate future, the next-generation memory would became phase-change memory (PCM) [7, 18]. Another NVRAM memory technology (3D XPoint, announced by Intel and Micron) was expected to appear on market in 2016 with much better properties than the alternatives [5, 10, 17]. So far, the NVRAM devices are still yet to be published. Performance of currently released devices is comparable more to SSD than to RAM. In accordance with the previously mentioned predictions, it is a reasonable assumption that the problem with volatility and limited capacity of RAM will soon be solved.

There are several platforms that provide storage and processing features that could be potentially extended with NVRAM. The Apache Spark cluster-computing

[3] `http://www.zhoufeng.net/eng/spamwatch/`
[4] `https://oceanstore.cs.berkeley.edu/`

framework is an example that has been widely tested in practice (i.e., for matrix computations [1] or graph processing [3]). A great advantage of Spark is its modular architecture, which means it contains Spark Core (foundation of the project), Spark Streaming (event processing that can utilize not only plain TCP/IP but also more-comprehensive solutions; i.e., ZeroMQ or Apache Kafka), or a distributed storage system (with the support of many providers like the Hadoop Distributed File System (HDFS) or Cassandra NoSQL), among others. Multiple layers and various components offer the possibility of NVRAM integration – not only with storage, but also with processing or messaging. Although promising, the framework currently does not support NVRAM.

Moreover, it is designed to work with modern programming languages like Java, Python, and Scala, which could be inconvenient for typical HPC applications based on C/C++ or Fortran with MPI or OpenMP support.

According to the author's knowledge at the moment of writing the paper, the only NoSQL database that supports NVRAM features is a special version of Redis[5]. Moreover, there have also been several attempts to create a software layer that will cover Redis behind an easy-to-use domain-specific interface like Redis Graph – a module that implements a graph database on top of Redis[6].

3. Proposed architecture

3.1. Software components

In this section, the author proposes the exemplary architecture of an HPC environment supported by a key-value database. As a key-value store, Redis 3.1.103 enhanced to use NVRAM is applied. This Redis extension internally uses the libpmemobj library[7] – a solution that turns a file located in an NVRAM device into a persistent object store. The project is in the initial phase and currently supports only a limited number of Redis structures. In response to demand for a storage throughput, Redis can be run as a single instance or in distribution using Redis Cluster[8]. The exemplary architecture is illustrated in Figure 1.

The Redis database can be used in many programming languages used in HPC; e.g., C, C++, Python, and Matlab. For testing purposes, the author uses the C language along with MPI. Redis access is provided using hiredis client[9]. Redis storage is covered by a specific API that is different for each application domain; e.g., graph processing, natural language processing, particle movement simulation. The whole technology stack is presented in Figure 2.

[5] `https://github.com/pmem/redis`
[6] `https://github.com/swilly22/redis-graph`
[7] `http://pmem.io/nvml/libpmemobj/`
[8] `http://redis.io/topics/cluster-spec`, `http://redis.io/topics/cluster-tutorial`
[9] `https://github.com/redis/hiredis`

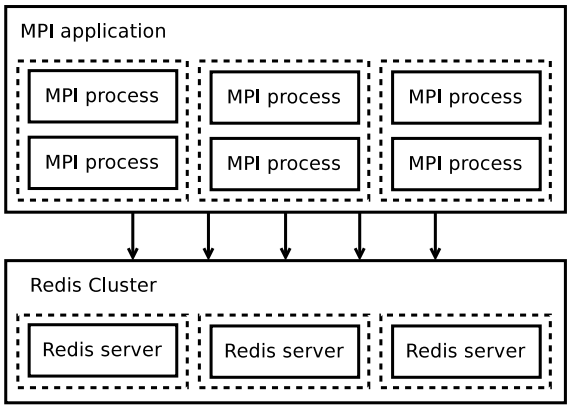

Figure 1. Top view of exemplary architecture with multiple nodes. Dashed rectangles indicate separate physical machines.

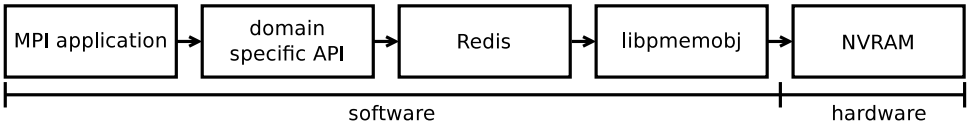

Figure 2. Technology stack used for testing. MPI application calls domain-specific API – abstract layer that covers Redis. Redis uses NVRAM as its persistent storage.

3.2. Domain-specific APIs – case studies

3.2.1. Graph processing

For directed graph processing In API, each vertex is stored as a set, while the edges are represented by elements of this set. Basic operations that show the idea behind this API are as follows:

1. `getNeighbors(graph, vertexId)`
   - returned type: `integer[]`,
   - Redis pseudocode: `SMEMBERS graph:vertexId`,
   - complexity: $O(n)$, where n is the number of a vertex's neighbors;
2. `addEdge(graph, srcVertexId, dstVertexId)`
   - returned type: `void`,
   - Redis pseudocode: `SADD graph:srcVertexId dstVertexId`,
   - complexity: $O(1)$;
3. `removeEdge(graph, srcVertexId, dstVertexId)`
   - returned type: `void`,
   - Redis pseudocode: `SREM graph:srcVertexId dstVertexId`,
   - complexity: $O(1)$;

4. `removeVertex(graph, vertexId)`
   - returned type: `void`,
   - Redis pseudocode:
     ```
     for each i in SMEMBERS graph:vertexId
         SREM graph:i vertexId
     DEL graph:vertexId,
     ```
   - complexity: $O(n)$, where n is the number of a vertex's neighbors;
5. `isConnected(graph, srcVertexId, dstVertexId)`
   - returned type: `boolean`,
   - Redis pseudocode: `SISMEMBER graph:srcVertexId dstVertexId`,
   - complexity: $O(1)$.

3.2.2. Counting n-grams

An N-gram is a sequence of n items (e.g., letters, syllables, words) from a particular text; it is mainly used in natural language processing for predicting the next item in a sequence. The selected API functions are as follows:

1. `incNGramCount(ngram)`
   - returned type: `void`,
   - Redis pseudocode: `INCR ngram`,
   - complexity: $O(1)$;
2. `incNGramCount(ngram, count)`
   - returned type: `void`,
   - Redis pseudocode: `INCR ngram count`,
   - complexity: $O(1)$;
3. `getNGramCount(ngram)`
   - returned type: `integer`,
   - Redis pseudocode: `get ngram`,
   - complexity: $O(1)$.

3.2.3. Processing vectors

The third API is used for operating on a vector. This implementation is efficient only for a subset of vector operations, as it uses lists in Redis (accessing the n-th element of a list has a complexity equal to $O(n)$). Several selected API functions are as follows:

1. `createVector(vector, elements)`
   - returned type: `void`,
   - Redis pseudocode: `RPUSH vector elements`,
   - complexity: $O(n)$, where n is the number of elements;

2. `getVector(vector)`
   - returned type: `integer[]` or `float[]` or `double[]` or `string[]`,
   - Redis pseudocode: `LRANGE vector 0 -1`,
   - complexity: $O(n)$, where n is the number of elements;
3. `getVectorElement(vector, index)`
   - returned type: `integer` or `float` or `double` or `string`,
   - Redis pseudocode: `LINDEX vector index`,
   - complexity: $O(1)$;
4. `setVectorElement(vector, index, element)`
   - returned type: void,
   - Redis pseudocode: `LSET vector index element`,
   - complexity: $O(n)$, where n is the length of the vector.

4. Experiments

Experiments focus on proving that Redis is able to handle requests quickly, even under a heavy load. For testing purposes, three micro-benchmarks were prepared. First, the application's task is distributed n-gram counting. Each process scans the text, retrieves the n-grams, and increments a global counter stored in Redis. The text sample for each process contains about 10,000 n-grams ($+/-2\%$). The second micro-benchmark is a simple directed graph generator.

Each process selects two random vertices (according to uniform distribution) and then – if it does not exists – adds an edge between them. At the end, a single process is supposed to generate 50,000 edges. The third application focuses on different sizes of input/output data. It simply stores and loads the data chunks multiple times.

4.1. Testbed environment

All of the tests were performed using 96 nodes of a K2 cluster, each node equipped with:

- 2 x Intel Xeon E5345 (together 8 physical cores),
- 8GB of RAM,
- 10Gb/s Ethernet connection.

An additional single node with parameters as specified above was dedicated for the Redis server. As NVRAM devices are not available on the market yet, Redis operated on RAM. The cluster is managed by Rocks Clusters, and the computing nodes work under CentOS 6.5. As an MPI implementation, MPICH 3.2 was used.

4.2. Results

4.2.1. Various numbers of processes

Figure 3 presents the results for the n-gram counting micro-benchmark, Figure 4 shows the results for the random graph generator. In both experiments, execution

time grows linearly according to the number of processes (which proves that about a hundred nodes, each running eight processes, was not able to overload a single Redis server). Because of oversubscribing (assigning more processes than physical cores on a single node), the number of processes was limited to 800, as it caused a performance drop on the computing nodes.

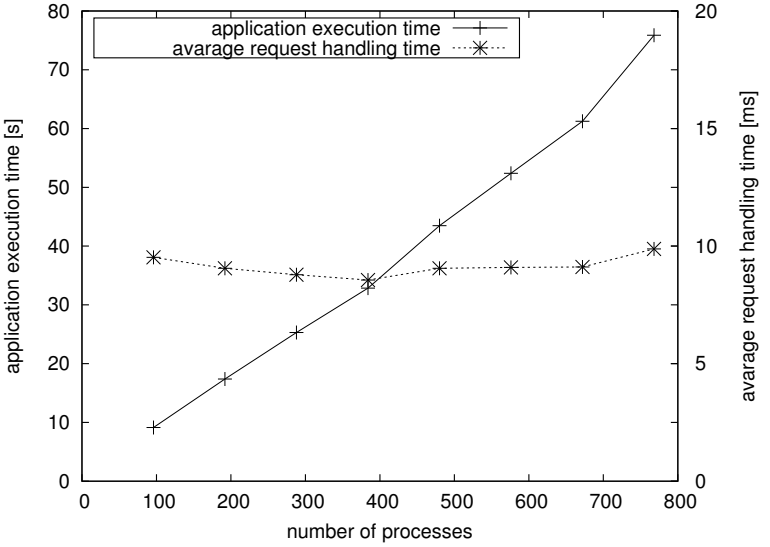

Figure 3. Performance results of counting n-grams.

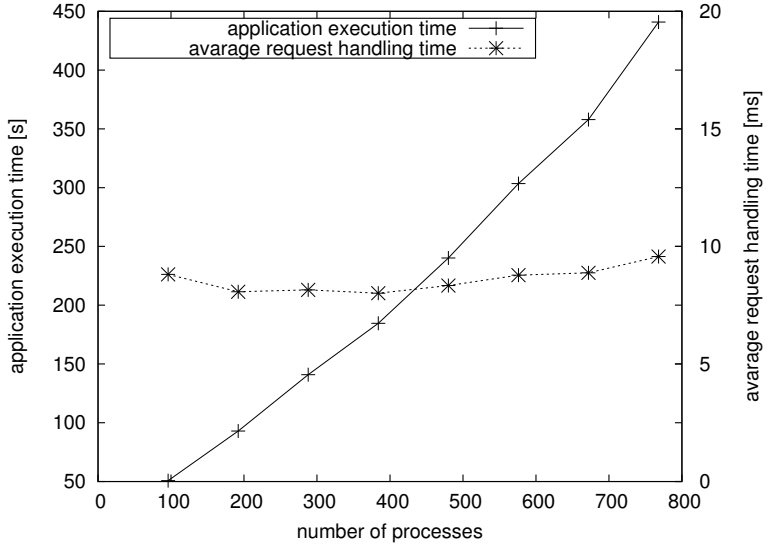

Figure 4. Performance results of constructing a directed graph model.

For both experiments, the average handle times of a single request were calculated. Figure 3 and Figure 4 show that the time of processing a single request was constant and lower than 10ms. It should be also noted that, on the target environment, this value would depend on the performance of the NVRAM devices.

4.2.2. Various data sizes

The next two charts show the performance results for an application that loads (Fig. 5) and stores (Fig. 6) data chunks of different sizes in Redis. The application was executed on 100 nodes with 800 processes, and each process accessed the data 50,000 times. As illustrated with figures, write accesses up to 128 bytes are served really fast – the average time of a synchronous store operation for a single data chunk was lower than 9 ms. The situation is similar with loading the data from Redis – processing was really efficient for chunks up to 512 bytes, with an average processing time lower than 9 ms. Further increasing the accessed data size introduces a noticeable delay in processing; for instance, reading or writing 2kB of data is served within about 14 ms. On the other hand, request processing time grows logarithmically with a single data chunk size, which seems to be acceptable for many applications. It should be also noted that bigger HPC data structures should be decomposed into smaller elements as shown in exemplary domain-specific APIs. If it is not possible (i.e., when processing large continuous blocks of data like images or geographic maps), traditional solutions with processing files are more recommended.

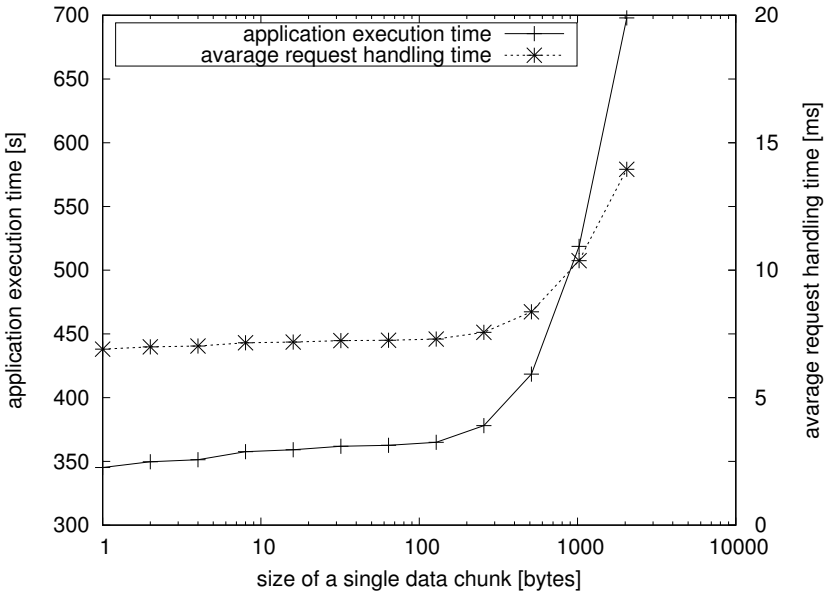

Figure 5. Performance results of loading data chunks from Redis.

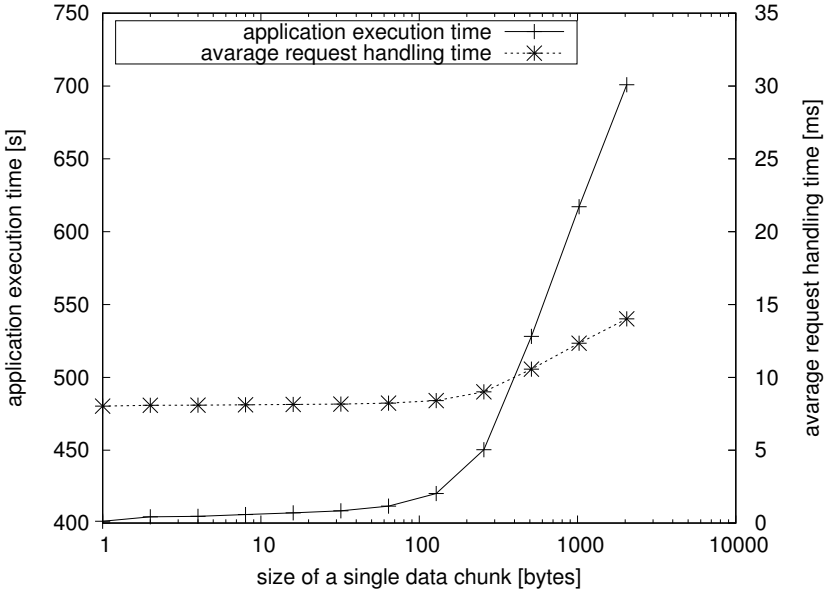

Figure 6. Performance results of storing data chunks into Redis.

5. Conclusion and future work

In this paper, the author proposed using a key-value database in HPC applications. The motivation for this was its good performance, low latency, and high scalability justified with many examples included in related work. The author highlighted the problem of volatility and capacity limitation of memory usually used in key-value storage and proposed solving it by using NVRAM, which will be available in the near future. The research also introduced a basic hardware architecture and required software components including exemplary domain-specific APIs.

As a key-value store, Redis enhanced to use NVRAM was proposed. Presented fragments of APIs showed the idea behind transformation from scientific data objects into data structures available on Redis, but the approach could be extended to other similar storage platforms. The last part of the paper was dedicated to experiments with two micro-benchmarks that evaluated the performance of the proposed architecture.

In the near future, the author will focus on the further development of domain-specific APIs and publishing them under one of the open-source licenses. As NVRAM-enabled devices are expected to become available soon, the author also plans to incorporate the presented approach into real-world HPC applications.

References

[1] Bosagh Zadeh R., Meng X., Ulanov A., Yavuz B., Pu L., Venkataraman S., Sparks E., Staple A., Zaharia M.: Matrix Computations and Optimization in Apache Spark. In: *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD'16, pp. 31–38. ACM, New York, 2016. http://dx.doi.org/10.1145/2939672.2939675.

[2] Butler D.M.: Scientific Computing Doesn't Need noSQL. In: *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC'12, pp. 1301–1302. IEEE Computer Society, Washington, DC, USA, 2012. http://dx.doi.org/10.1109/SC.Companion.2012.158.

[3] Carlini E., Dazzi P., Esposito A., Lulli A., Ricci L.: Balanced Graph Partitioning with Apache Spark. In: *Euro-Par 2014: Parallel Processing Workshops: Euro-Par 2014 International Workshops, Porto, Portugal, August 25–26, 2014, Revised Selected Papers, Part I*, pp. 129–140. Springer International Publishing, Cham, 2014. http://dx.doi.org/10.1007/978-3-319-14325-5_12.

[4] Dutka Ł., Słota R., Wrzeszcz M., Król D., Kitowski J.: Uniform and Efficient Access to Data in Organizationally Distributed Environments. In: Bubak M., Kitowski J., Wiatr K. (eds.), *eScience on Distributed Computing Infrastructure: Achievements of PLGrid Plus Domain-Specific Services and Tools*, pp. 178–194. Springer International Publishing, Cham, 2014. http://dx.doi.org/10.1007/978-3-319-10894-0_13.

[5] Foong A., Hady F.: Storage as Fast as Rest of the System. In: *2016 IEEE 8th International Memory Workshop (IMW)*, pp. 1–4, 2016. http://dx.doi.org/10.1109/IMW.2016.7495289.

[6] Forum M.P.I.: MPI: A Message-Passing Interface Standard Version 3.1, 2015. http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[7] Gao S., Xu J., Hrder T., He B., Choi B., Hu H.: PCMLogging: Optimizing Transaction Logging and Recovery Performance with PCM, *IEEE Transactions on Knowledge and Data Engineering*, vol. 27(12), pp. 3332–3346, 2015. http://dx.doi.org/10.1109/TKDE.2015.2453154.

[8] Han J., Haihong E., Le G., Du J.: Survey on NoSQL database. In: *Pervasive Computing and Applications (ICPCA), 2011 6th International Conference on*, pp. 363–366. 2011. http://dx.doi.org/10.1109/ICPCA.2011.6106531.

[9] Hanlon M.R., Dooley R., Mock S., Dahan M., Nuthulapati P., Hurley P.: A Case Study for NoSQL Applications and Performance Benefits: CouchDB vs. Postgres. In: *Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery*. ACM, New York, NY, USA, 2011.

[10] Intel Corporation: Introducing Breakthrough Memory Technology, 2015. http://www.intel.com/content/www/us/en/architecture-and-technology/non-volatile-memory.html.

[11] Kryder M.H., Kim C.S.: After Hard Drives. What Comes Next? *IEEE Transactions on Magnetics*, vol. 45(10), pp. 3406–3413, 2009. http://dx.doi.org/10.1109/TMAG.2009.2024163.

[12] Li T., Verma R., Duan X., Jin H., Raicu I.: Exploring Distributed Hash Tables in HighEnd Computing, *SIGMETRICS Performance Evaluation Review*, vol. 39(3), pp. 128–130, 2011. http://dx.doi.org/10.1145/2160803.2160880.

[13] Mateescu G., Gentzsch W., Ribbens C.J.: Hybrid Computing–Where {HPC} meets grid and Cloud Computing, *Future Generation Computer Systems*, vol. 27(5), pp. 440–453, 2011. http://dx.doi.org/http://dx.doi.org/10.1016/j.future.2010.11.003.

[14] Mohan C.: History Repeats Itself: Sensible and NonsenSQL Aspects of the NoSQL Hoopla. In: *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT'13, pp. 11–16. ACM, New York, 2013. http://dx.doi.org/10.1145/2452376.2452378.

[15] Monnerat L., Amorim C.L.: An Effective Single-hop Distributed Hash Table with High Lookup Performance and Low Traffic Overhead, *Concurrency and Computation: Practice and Experience*, vol. 27(7), pp. 1767–1788, 2015. http://dx.doi.org/10.1002/cpe.3342.

[16] Oral S., Dillow D.A., Fuller D., Hill J., Leverman D., Vazhkudai S.S., Wang F., Kim Y., Rogers J., Simmons J., Miller R.: OLCFs 1 TB/s, Next-Generation Lustre File System. In: *Proceedings of Cray User Group Conference (CUG 2013)*, 2013.

[17] Patterson D.: Past and Future of Hardware and Architecture. In: *SOSP History Day 2015*, SOSP '15, pp. 9:1–9:63. ACM, New York, 2015. http://dx.doi.org/10.1145/2830903.2830910.

[18] Qiu M., Ming Z., Li J., Gai K., Zong Z.: Phase-Change Memory Optimization for Green Cloud with Genetic Algorithm, *IEEE Transactions on Computers*, vol. 64(12), pp. 3528–3540, 2015. http://dx.doi.org/10.1109/TC.2015.2409857.

[19] Strohmaier E., Dongarra J., Simon H., Meuer M.: TOP 10 Sites for June 2016. https://www.top500.org/lists/2016/06/.

[20] Wrzeszcz M., Trzepla K., Słota R., Zemek K., Lichoń T., Opioła Ł., Nikolow D., Dutka Ł., Słota R., Kitowski J.: Metadata Organization and Management for Globalization of Data Access with Onedata. In: Wyrzykowski R., Deelman E., Dongarra J., Karczewski K., Kitowski J., Wiatr K. (eds.), *Parallel Processing and Applied Mathematics: 11th International Conference, PPAM 2015, Krakow, Poland, September 6–9, 2015. Revised Selected Papers, Part I*, pp. 312–321. Springer International Publishing, Cham, 2016. http://dx.doi.org/10.1007/978-3-319-32149-3_30.

[21] Zhao B.Y., Huang L., Stribling J., Rhea S.C., Joseph A.D., Kubiatowicz J.D.: Tapestry: A Resilient Global-scale Overlay for Service Deployment, *IEEE Journal on Selected Areas in Communications*, vol. 22(1), pp. 41–53, 2006. http://dx.doi.org/10.1109/JSAC.2003.818784.

[22] Zhao B.Y., Kubiatowicz J.D., Joseph A.D.: Tapestry: An Infrastructure for Fault-tolerant Wide-area Location, Technical report, Berkeley, CA, USA, 2001.

[23] Żmuda M., Opioła Ł., Dutka Ł., Słota R., Kitowski J.: Kademlia with Consistency Checks as a Foundation of Borderless Collaboration in Open Science Services. In: Boukhanovsky A., Bubak M., Balakhontceva M. (eds.), *Procedia Computer Science. 5th International Young Scientist Conference on Computational Science, YSC 2016, 26–28 October 2016, Krakow, Poland*, vol. 101, pp. 304–312, 2016. http://dx.doi.org/http://dx.doi.org/10.1016/j.procs.2016.11.036.

Affiliations

Artur Malinowski
Faculty of Electronics, Telecommunications and Informatics,
Gdansk University of Technology, Gdansk, Poland, artur.malinowski@pg.gda.pl

Received: 17.07.2016
Revised: 21.05.2017
Accepted: 22.05.2017