ISSN 1547-4771, Physics of Particles and Nuclei Letters, 2017, Vol. 14, No. 7, pp. 1001–1007. © Pleiades Publishing, Ltd., 2017.

COMPUTER TECHNOLOGIES IN PHYSICS

Middleware for Big Data Processing: Test Results[1]

I. Gankevich*, V. Gaiduchok, V. Korkhov, A. Degtyarev, and A. Bogdanov

St. Petersburg State University, St. Petersburg, 199034 Russia

**e-mail: i.gankevich@spbu.ru*

Received December 9, 2016

Abstract—Dealing with large volumes of data is resource-consuming work which is more and more often delegated not only to a single computer but also to a whole distributed computing system at once. As the number of computers in a distributed system increases, the amount of effort put into effective management of the system grows. When the system reaches some critical size, much effort should be put into improving its fault tolerance. It is difficult to estimate when some particular distributed system needs such facilities for a given workload, so instead they should be implemented in a middleware which works efficiently with a distributed system of any size. It is also difficult to estimate whether a volume of data is large or not, so the middleware should also work with data of any volume. In other words, the purpose of the middleware is to provide facilities that adapt distributed computing system for a given workload. In this paper we introduce such middleware appliance. Tests show that this middleware is well-suited for typical HPC and big data workloads and its performance is comparable with well-known alternatives.

DOI: 10.1134/S1547477117070068

1. INTRODUCTION

Today's methods and technologies for big data processing are focused on some particular use cases: many of them are as simple as parallel fault-tolerant implementations of well-known algorithms from functional languages [12–14], others are reimplementations of well-established structured data querying techniques [15–17]. Although, most of the technologies share the same job scheduler [18], each of them requires its own distributed platform (built on top of the scheduler) to operate. Authors' experience shows that installing, configuring, securing and maintaining such systems requires significant effort, not to mention that the usage of compute and storage resources is not always optimal as you need to allocate a separate machine (or two for fault tolerance) for each service. Moreover, simple data processing methods implemented in these technologies require much programming effort to become usable for data consolidation tasks.

These findings led to an idea of making a scheduler to run not only batch jobs, but also services for data processing using the same API. This change makes the scheduler a central part of the distributed system which is responsible for fault tolerance of each service as well as each job.The approach makes it possible to push fault-tolerance, synchronisation and other common distributed application mechanics to the code base of the scheduler which makes implementation of each service light-weight and lessen the whole system administration burden.

This paper is a generalisation and interpretation of results obtained by the authors during last two years. Additional details, more experiments and results can be found in some previous works [19–21] which dealt with particular aspects of the middleware system.

2. RELATED WORK

There are multiple ongoing efforts to make writing and running distributed applications simple [18, 22, 23], however, these efforts are not consolidated and mainly focused on some particular aspect of distributed operation. For example, in Etcd and ZooKeeper [22, 23] it is possible to reliably store configuration items and application state, but there is no API for distributing applications and providing fault tolerance for them. It is assumed that application writers would implement their own mechanism for fault tolerance using this distributed store. In YARN [18] there is no standard way of implementing services, i.e. there is no service discovery, service endpoints and supporting APIs. So, these technologies are integral parts of the middleware appliance for writing and running distributed applications, but they do not constitute the whole.

Another important aspect of a distributed system is asycnhronous operation, which is typically implemented in an application event-driven architecture (which consists of thread/process pool and task/event queue). This architecture has been used extensively to create desktop applications with graphical user interface since MVC paradigm [10] was developed and nowadays it is also used to compose enterprise application components into a unified system with message

[1] The article is published in the original.

queues [11], however, it is rarely used in scientific applications. One example of such usage is GotoBLAS2 library [8, 9]. Although, it is not clear from the referenced papers, analysis of its source code shows that this library uses a specialised server object to schedule matrix and vector operations' kernels and compute them in parallel. The total number of CPUs is defined at compile time and they are assumed to be homogeneous. There is a notion of a queue implemented as a linked list of objects where each object specifies a routine to be executed and data to be processed and also a number of CPUs to execute it on. Server processes these objects in parallel, and each kernel can be executed in synchronous (blocking) and asynchronous (non-blocking) mode. GotoBLAS2 library exhibits competitive performance compared to other BLAS implementations [8, 9] and it is a good example of viability of event-driven approach in scientific applications. Considering this, an event-driven system can be seen as a generalisation of this approach to a broader set of applications.

3. BASIC DESIGN PRINCIPLES AND RATIONALE

Our middleware system is based on several simple principles which govern architectural decisions for it. The principles follow.

(1) The data is considered *big*, if its pre- and post-processing time is much larger than processing time. Pre- and post-processing includes general I/O, compressing/decompressing, encoding/decoding, filtering and other auxiliary operations. It follows that big data does not always have big volume, and tightly-coupled and semi-structured data is also considered *big*. This principle allows to bridge the gap between high-performance computing and big data applications, and use the same API for both. So, the distinct property of big data applications is that their parallel tasks are bound to nodes where input data is located.

The key difference of this definition of "big data" is that it does not try to capture qualitative aspects of big data processing like many other definitions do [24], but it attempts to pin the point where a programmer should care more about data handling code performance than performance of all other code. In other words, for big data problems optimisation of non data handling code does not give large increase in performance. One can argue, that in big data applications all the code does data handling, however, in our view big data problems have much broader scope than problems handled by Hadoop and MapReduce. In other words, big data is not only Hadoop but any application where data processing performance matters. So, our definition captures quantitative aspects of big data, such as volume, connectedness and density. We hope, that this definition is useful for developing big data related middleware as it is easy to mathematically formalise quantitative aspects and use them to derive optimal parameters for the system.

(2) Each dataset has two dynamically adjustable parameters: the number of replicas and the number of chunks. These parameters are capped by physical constraints such as total number of nodes, maximal number of nodes per job etc., and are used to adapt read/write performance of a dataset for a given workload. For example, if a dataset is used for searching, its number of replicas is dynamically increased to trade write performance for read performance, and if this dataset is not accessed for a long period of time its number of replicas is decreased to some minimal number to save disk space. This principle makes the whole system agile and allows automatic archiving of infrequently used data.

(3) The API for parallel processing of big data is based on *micro-kernels* – special event-driven objects implementing callbacks for processing, collecting results from subordinates and reading/writing. These micro-kernels use subordinate kernels to process data in parallel and *always* bind to a compute node where the data chunk (if any) is stored. The API allows expressing problem solution in loosely-coupled parameterised modules, which solve one particular part of the problem; there are no messages in this context, instead a micro-kernel may migrate to another compute node to communicate with its parent or some other micro-kernel.

These principles allow building scalable system with concise API suitable for both high-performance computing and big data applications.

4. IMPLEMENTATION

The distributed system consists of the middleware *core* providing API for transmitting/receiving micro-kernels over the network, and several applications based on this API which discover and update the list of healthy cluster nodes, manage file replicas and expose a web interface for monitoring and querying system status. The service and the applications run on each node of the cluster.

4.1. Middleware Core and API

The core is implemented as a small and light-weight Linux daemon written in C++. It consumes minimal amount of RAM and its source code base contains less than 10000 SLOC. The sole purpose of the core is to provide API for application writers to create, send and receive micro-kernels. So, the core can be seen as a cluster-wide scheduler for micro-kernels.

The feature which distinguishes micro-kernels from other parallel programming techniques is an ability to create *subordinate* kernels to describe computations. Multiple subordinate kernels are always executed in parallel (if the number of processor cores

permits) and their *principal* is deleted only when all of them completed execution. This allows modelling SIMD, MIMD and MISD parallel programming patterns. If the problem can not be described by a parallel algorithm, but there are multiple inputs to it, a micro-kernel can be created for each data file. This effectively duplicates kernel hierarchy for each file automatically creating UNIX-like pipeline. So, the notion of micro-kernel is general enough to express both parallel computations and pipelined execution of programmes.

The feature that makes this implementation different from other similar approaches is that both processors and disks work in parallel throughout the programme execution. Such behaviour is achieved with assigning a separate thread pool for each device and placing kernels in the queue for the corresponding device. As kernels that read from the disk complete, they produce kernels for CPUs to process this data and place them into the CPU kernel queue. In similar way, when data processing kernels complete, they place tasks to write the data into the disk kernel queue. Similarly, via a separate kernel queue network devices send and receive the data from a remote node. So, each device has its own thread pool with a kernel queue, and all of them work in parallel by placing kernels in each other's kernel queues.

The feature that makes micro-kernels suitable for big data processing is an ability to bind to a compute node where an abstract resource is located. For now the resource is simply a file stored on local file system. Files are indexed locally and the resulting index is replicated to a principal node (principal and subordinate nodes are determined by discovery algorithm, see Section 2.2). Hierarchy of nodes is used to efficiently search for files: the search starts locally, and if the file is not found, the search is repeated recursively on principal node.

The API is event-driven and built around only one base class which makes it easily extensible. Derived classes are expected to override some of the callbacks (for computing, collecting results, reading/writing). If there is a desire to implement new feature, another callback method can be added to the base class (Listing 1).

```
struct Micro_kernel {
/// Modify state or create sub-tasks.
void act();
/// Collect results from sub-kernels.
void react(Micro_kernel* sub);
/// I/O
void read(Stream); void write(Stream);
/// Resource binding.
Resource resource();
};
```

Listing 1. Micro-kernel base class.

4.2. Node Discovery

Node discovery application uses consensus-free algorithm developed in [19] which distinguishes it from other similar programmes. The application ranks nodes in the network according to some static criteria (the simpliest ranking criteria is based on the position of the node's IP address in network IP address range), it then partitions ranked list of nodes into multiple hierarchical layers and selects the node with the highest rank from the closest layer as the leader. That way cluster nodes form a subordination tree, topology of which is changed only when a node leaves or joins the cluster. The advantage of this algorithm over distributed consensus algorithms is that it scales well (see Section 3.3) with the size of the cluster: as the number of nodes increases more layers are introduced into the subordination tree thus forming *leaders' framework* that manages the whole cluster.

Multiple levels of subordination are beneficial for a range of management tasks, especially for resource monitoring and usage accounting. Typical monitoring task involves probing each cluster node to determine its state (offline, online, needs service etc.). Usually probing is done from one node, and in case of a large cluster introducing intermediate nodes that collect data and send it to master, helps distribute the load. Each level of hierarchy adds another layer of intermediate nodes, so the data can be collected efficiently.

5. EVALUATION

The event-driven architecture was tested on the example of hydrodynamics simulation programme that solves a real-world problem [4–7]. The problem consists of generating real ocean wavy surface and computing pressure under this surface to measure impact of the external excitations on marine object. The program is well-balanced in terms of processor load and for the purpose of evaluation it was implemented with introduced event-driven approach and the resulting implementation was compared to existing non event–driven approach in terms of performance and programming effort.

The event-driven architecture makes it easy to write logs which in turn can be used to make visualization of control flow in a program. Each server maintains its own log file and when some event occurs it is logged in this file accompanied by a time stamp and a server identifier. Having such files available, it is straight-forward to reconstruct a sequence of events occurring during program execution and to establish connections between these events (to dynamically draw graph of tasks as they are executed). Such graphs are used in this section to demonstrate results of experiments.

Generation of wavy surface is implemented as a transformation of white noise, autoregressive model is used to generate ocean waves and pressures are com-

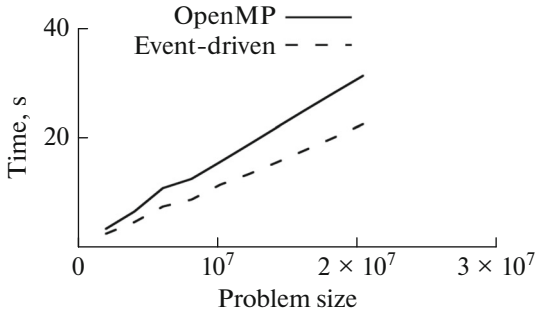

Fig. 1. Performance comparison of OpenMP and event-driven implementations.

puted using analytical formula. The program consists of preprocessing phase, main computer-intensive phase and post-processing phase. The programme begins with solving Yule–Walker equations to determine autoregressive coefficients and variance of white noise. Then white noise is generated and is transformed to a wavy surface. Finally, the surface is trimmed and written to output stream. Generation of a wavy surface is the most computer-intensive phase and consumes over 80% of programme execution time (Fig. 2) for moderate wavy surface sizes and this time does not scale with a surface size. So, the program spends most the time in the main phase generating wavy surface (this phase is marked with [$G0$, $G1$] interval in the graphs). The hardware used in the experiments is listed in Table 2. The program was tested in a number of experiments and finally compared to other parallel programming techniques.

5.1. Evaluation for an HPC Application

In the experiment overall performance of the event-driven approach was tested and it was found to be superior when solving problems producing large volumes of data. In the previous research it was found that OpenMP is the best performing technology for the wavy ocean surface generation [5], so the experiment consisted of comparing its performance to the performance of event-driven approach on a set of input data. A range of sizes of a wavy surface was the only parameter that was varied among subsequent program runs. As a result of the experiment, the event-driven approach was found to have higher performance than OpenMP technology and the more the size of the problem is the bigger performance gap becomes (Fig. 1). Also event plot in Fig. 2 of the run with the largest problem size shows that high performance is achieved with overlapping of parallel computation of a wavy surface (interval [$G0$, $G1$]) and output of resulting wavy surface parts to the storage device (interval [$W0$, $W1$]). It can be seen that there is no such overlap in OpenMP implementation and output begins at point $W0$ right after the generation of wavy surface ends at point $G1$. In contrast, there is a significant overlap in event-driven implementation and in that case wavy surface generation and data output end almost simultaneously at points $G1$ and $W1$ respectively. So, the approach with pipelined execution of parallelised computational steps achieves better performance than sequential execution of the same steps.

Although OpenMP technology allows constructing pipelines, it is not easy to combine a pipeline with parallel execution of tasks. In fact such combination is possible if a thread-safe queue is implemented to communicate threads generating ocean surface to a thread writing data to disk. Then using omp section work of each thread can be implemented. However, implementation of parallelexecution within omp section requires support for nesting omp parallel directives. So, combining pipeline with parallel execution is complicated in OpenMP implementation requiring the use a thread-safe queue which is not present in OpenMP standard.

5.2. API Overheads

The experiment consisted of measuring different types of overheads including profiling, load balancing, queuing and other overheads so that real performance of event-driven system can be estimated. In this experiment, the same technique was used to obtain mea-

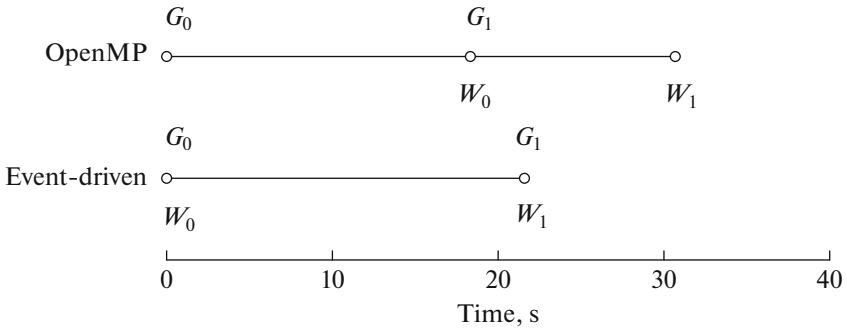

Fig. 2. Event plot showing overlap of parallel computation [$G0$, $G1$] and data output [$W0$, $W1$] in event-driven implementation. There is no overlap in OpenMP implementation.

Table 1. Breakdown of wall clock time for an event-driven system. Time is shown as a percentage of the total program execution time. Experiments for 4 cores were conducted on the System I and experiments for 24 and 48 cores were conducted on the System II from Table 2

	Time spent, %		
	4 cores	24 cores	48 cores
Problem solution	84	37	21
Stale time	16	63	79
Overhead	0.0724	0.0325	0.0225

surements: every function causing overhead was instrumented and also the total time spent executing tasks and total program execution time was measured. As a result, the total overhead was estimated to be less than 0.1% for different numbers of cores (Table 1). Also the results showed that reduction time is smaller than the total time spent solving production tasks in all cases (Table 1). It is typical of generator programs to spend more time solving data generating tasks than solving data processing tasks; in a data-centric program specializing in data processing this relation can be different. Finally, it is evident from the results that the more cores are present in the system, the more stale time is introduced into the program. This behaviour is caused by imbalance between processor performance and performance of a storage device for this particular computational problem. So, event-driven system do not incur much overhead even on systems with large number of cores.

5.3. Node Discovery Evaluation

Test platform consisted of a multi-processor node, and Linux network namespaces were used to consolidate virtual cluster of varying number of nodes on a physical node. Similar approach was used in a number of works [1–3]. The advantage of it is that the tests can be performed on a single machine, and there is no need to use physical cluster. Tests were repeated multiple times to reduce influences of processes running in background. Each subsequent test run was separated from previous one with a delay to give operating system time to release resources, cleanup files and flush buffers.

Performance test was designed to compare subordination tree build time for two cases. In the first case each node performed full network scan to determine online nodes, choose the leader, and then sent confirmation message to it. In the second case each node used IP mapping to determine the leader without full network scan, and then sent confirmation message to it. So, this test measured the effect of using IP mapping, and only one leader was chosen for all nodes.

Subordination tree test was designed to check that resulting trees for different maximal number of subordinate nodes are stable. For that purpose every change in a tree topology was recorded in a log file, and after 30 seconds every test run was forcibly stopped. The test was performed for 100–500 nodes. For this test additional physical nodes were used to accommodate large number of parallel processes (one node per 100 processes).

Performance test showed that using IP mapping can speed up subordination tree build time by up to 200% for 40 nodes (Fig. 4), and this number increases with the number of nodes. This is expected behaviour since overhead of sending messages to each node is omitted, and predefined mapping is used to find the leader. So, our approach is more efficient than a full scan of a network. The absolute time of this test is expected to increase when executed on real network, and thus performance may increase.

Subordination tree test showed that for each number of nodes stable state can be reached well before 30 seconds (Fig. 5). The absolute time of this test may increase when executed on real network. Resulting subordination tree for 11 nodes is presented in Fig. 3.

Table 2. Testbed setup

	System I	System II
Operating system	Debian 3.2.51-1 x86 64	CentOS 6.5 x86 64
File system	ext4	ext4
Processor	Intel Core 2 Quad Q9650	2 × Intel Xeon E5-2695 v2
Cores frequency, GHz	3.00	2.40
No. of cores	4	24 (48 virtual cores)
RAM capacity, GB	8	256
RAID device		Dell PERC H710 Mini
RAID configuration		RAID10
Storage device	Seagate ST3250318AS	4 × Seagate ST300MM0006
Storage device speed, rpm	7200	2 × 10000

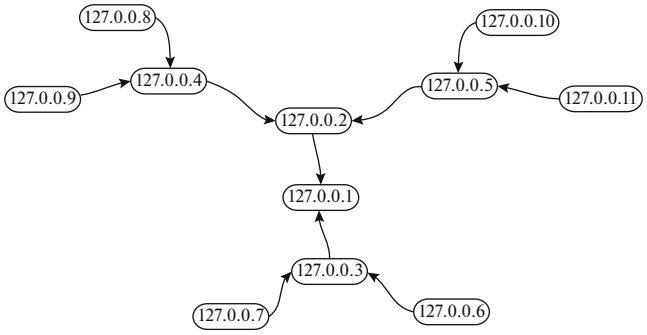

Fig. 3. Resulting subordination tree for 11 nodes.

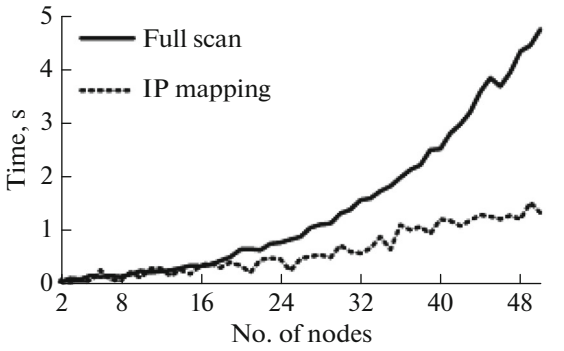

Fig. 4. Time needed to build initial subordination tree with full scan of each IP address in the network and with IP mapping.

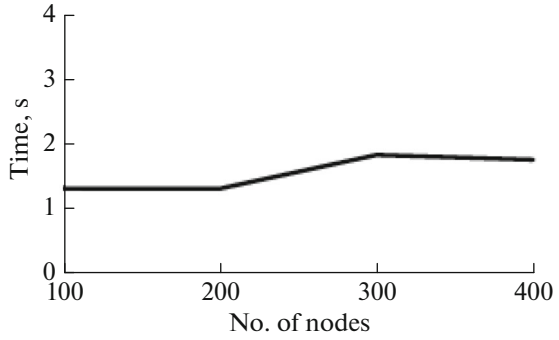

Fig. 5. Time needed to reach stable subordination tree state for largenumber of nodes.

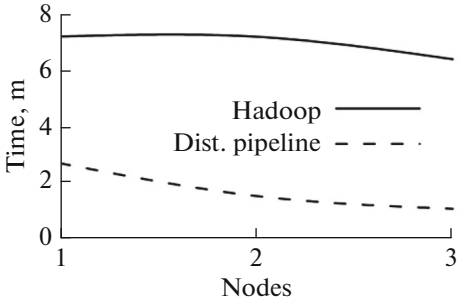

Fig. 6. Performance of Hadoop and distributed pipeline implementations.

5.4. Evaluation for a Big Data Problem

The system setup which was used to test the implementation consisted of commodity hardware (three nodes with System I configuration in Table 2) and open-source software (Hadoop), and the evaluation was divided into two stages. In the first stage Hadoop was installed on each node of the cluster and was configured to use host file system as a source of data so that performance of parallel file system used by default in Hadoop could be factored out from the comparison. To make this possible the whole dataset was replicated on each node and placed in the directory with the same name. In the second stage Hadoop was shut down and replaced by newly developed middleware system and dataset directories were statically distributed to different nodes to nullify the impact of parallel file system on the performance.

In the test it was found that Hadoop implementation has low scalability and maximum performance of approx. 1000 spectra per second and alternative implementation has higher scalability and maximum performance of approx. 7000 spectra per second (Fig. 6). The source of Hadoop inefficiency was found to be temporary data files which are written to disk on each node. These files represent sorted chunks of the key-value array and are part of implementation of merge sort algorithm used to distribute the keys to different nodes. For NDBC dataset the total size of these files exceeds the size of the whole dataset which appears to be the consequence of Hadoop not compressing intermediate data (the initial dataset has compression ratio of 1 : 5). So, the sorting algorithm and inefficient handling of compressed data led to performance degradation and inefficiency of Hadoop for NDBC dataset.

The sorting is not needed to distribute the keys and in the alternative implementation directory hierarchy is used to determine machine for reduction. For each directory a separate task is created which recursively creates tasks for each sub-directory and each file. Since each task can interact with its parent when the reduction phase is reached reduction tasks are created on the machines where parents were executed previously.

6. CONCLUSIONS

Apart from being more efficient than OpenMP the biggest advantage of the event-driven approach is the ease of parallel programming. First of all, what is needed from a programmer is to develop a class to describe each independent task, create objects of that class and submit them to a queue. Programming in such a way does not involve thread and lock management and the system is flexible enough to have even

the tiniest tasks executed in parallel. Second, relieving programmer from thread management makes it easy to debug this system. Each thread maintains its own log and any of both system and user events can be written to it and the sequence of events can be restored after the execution ends. Finally, with event-driven approach it is easy to write load distribution algorithm for your specific problem (or use an existing one). The only thing which is not done automatically is decomposition and composition of micro-kernels, however, this problem requires higher layer of abstraction to solve.

For big data applications no redundant sorting nor any kind of temporary files are used in the implementation which allows it to scale well and show better performance compared to Hadoop approach.

The future work is to extend event-driven approach for distributed and hybrid (GPGPU) systems and to see if it is possible to cover those cases.

ACKNOWLEDGMENTS

The research was carried out using computational resources of Resource Centre "Computational Centre of Saint Petersburg State University" (T-EDGE96 HPC-0011828-001) and supported by Russian Foundation for Basic Research (projects nos. 16-07-01111, 16-07-00886, 16-07-01113) and St. Petersburg State University (project no. 0.37.155.2014).

REFERENCES

1. B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks* (ACM, 2010), p. 19.
2. N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible network experiments using container-based emulation," in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies* (ACM, 2012), pp. 253–264.
3. B. Heller, "Reproducible network research with high-fidelity emulation," PhD Thesis (Stanford Univ., 2013).
4. A. Degtyarev and A. Reed, "Synoptic and short-term modelling of ocean waves," Int. Shipbuild. Prog. **60**, 523–553 (2013).
5. A. Degtyarev and I. Gankevich, "Wave surface generation using OpenCL, OpenMP and MPI," in *Proceedings of the 8th International Conference on Computer Science and Information Technologies, 2011,* pp. 248–251.
6. A. B. Degtyarev and A. M. Reed, "Modelling of incident waves near the ship's hull (application of autoregressive approach in problems of simulation of rough seas)," in *Proceedings of the 12th International Ship Stability Workshop, 2011.*
7. A. Degtyarev and I. Gankevich, "Evaluation of hydrodynamic pressures for autoregression model of irregular waves," in *Proceedings of the 11th International Conference on Stability of Ships and Ocean Vehicles, Athens, 2012,* pp. 841–852.
8. Goto Kazushige and R. van de Geijn, "Anatomy of high-performance matrix multiplication," ACM Trans. Math. Software **34** (3), 12 (2008).
9. Goto Kazushige and R. van de Geijn, "High-performance implementation of the level-3 blas," ACM Trans. Math. Software **35** (1), 4 (2008).
10. G. E. Krasner, S. T. Pope, et al., "A description of the model-view-controller user interface paradigm in the Smalltalk-80 system," J. Object Oriented Program. **1** (3), 26–49 (1988).
11. S. Vinoski, "Advanced message queuing protocol," Internet Comput. **10** (6), 87–89 (2006).
12. M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (USENIX Association, 2012), p. 2.
13. M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, 2010,* p. 10.
14. J. Dean and G. Sanjay, "MapReduce: simplified data processing on large clusters," Commun. ACM **51**, 107–113 (2008).
15. M. Hausenblas and J. Nadeau, "Apache drill: interactive ad-hoc analysis at scale," Big Data **1.2**, 100–104 (2013).
16. A. Thusoo et al., "Hive: a warehousing solution over a map-reduce framework," in *Proceedings of the VLDB Endowment **2.2*** (2009), pp. 1626–1629.
17. C. Olston et al., "Pig latin: a not-so-foreign language for data processing," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (ACM, 2008).
18. V. K. Vavilapalli et al., "Apache hadoop yarn: yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing* (ACM, 2013).
19. I. Gankevich, Yu. Tipikin, and V. Gaiduchok, "Subordination: cluster management without distributed consensus," in *Proceedings of the International Conference on High Performance Computing Simulation HPCS, 2015,* pp. 639–642.
20. I. Gankevich and A. Degtyarev, "Efficient processing and classification of wave energy spectrum data with a distributed pipeline," Comput. Res. Model. **7**, 517–520 (2015).
21. I. Gankevich, Yu. Tipikin, A. Degtyarev, and V. Korkhov, "Novel approaches for distributing workload on commodity computer systems," in *Proceedings of the International Conference on Computational Science and Its Applications, ICCSA,* Lect. Notes Comput. Sci. **9158**, 259–271 (2015).
22. P. Hunt et al., "ZooKeeper: wait-free coordination for internet-scale systems," in *Proceedings of the USENIX Annual Technical Conference, Boston, MA, USA, June 23–25, 2010,* Vol. 8.
23. CoreOS, Etcd, Fleet. https://coreos.com/.
24. NIST Big Data PWG, NIST Big Data Interoperability Framework, Vol. 1: Definitions, Reference Architecture (2015). doi 10.6028/NIST.SP.1500-1