Distributed data processing on microcomputers with Ascheduler and Apache Spark

Vladimir Korkhov[1], Ivan Gankevich[1], Oleg Iakushkin[1], Dmitry Gushchanskiy[1], Dmitry Khmel[1], Andrey Ivashchenko[1], Alexander Pyayt[2], Sergey Zobnin[2], and Alexander Loginov[2]

[1] Saint Petersburg State University,
7/9 Universitetskaya nab., St. Petersburg, 199034, Russia
v.korkhov@spbu.ru
[2] Siemens LLC, St. Petersburg, Russia

Abstract. Modern architectures of data acquisition and processing often consider low-cost and low-power devices that can be bound together to form a distributed infrastructure. In this paper we overview possibilities to organize a distributed computing testbed based on microcomputers similar to Raspberry Pi and Intel Edison. The goal of the research is to investigate and develop a scheduler for orchestrating distributed data processing and general purpose computations on such unreliable and resource-constrained hardware. Also we consider integration of the scheduler with well-known distributed data processing framework Apache Spark. We outline the project carried out in collaboration with Siemens LLC to compare different configurations of the hardware and software deployment and evaluate performance and applicability of the tools to the testbed.

Keywords: microcomputers, scheduling, Apache Spark, Raspberry Pi, fault tolerance, high availability

1 Introduction

The problem of building distributed computing infrastructures for data collection and processing has been around for many years. One of the well-known technologies for building large-scale computing infrastructures is grid computing. It provides means to connect heterogeneous, dynamic resources into a single metacomputer. However, being focused on high-performance computing systems, grid technologies do not suit well other classes of basic hardware. One of such examples are low-performance, low-cost unreliable microcomputers similar to Raspberry Pi or Intel Edison, sometimes also called System-on-Chip (SoC) devices. To be able to execute distributed applications over a set of such devices extensive fault-tolerance support is needed along with low resource usage profile of the middleware.

In this paper we discuss an approach to orchestrate distributed computing and data processing on microcomputers with help of custom scheduler focused on

fault tolerance and dynamic rescheduling of computational kernels that represent the application. This scheduler, which is named Ascheduler, provides its own low-level API to create and manage computational kernels. Currently the Ascheduler is a closed-source project built on the ideas and approaches presented in [5–7].

In addition, the scheduler has been integrated into Apache Spark [1] data processing framework instead of the default scheduler used by Spark. This opened possibilities to use a wide range of existing Spark-based programs on the underlying microcomputer infrastructure controlled by the Ascheduler.

The project aimed to solve the following main tasks:

- Develop automatic failover and high-availability mechanisms for computer system.
- Develop automatic elasticity mechanism for computer system.
- Enable adjusting application algorithm precision taking into account current number of healthy cluster nodes.
- Adjust load distribution taking into account actual and heterogeneous monitoring data from cluster nodes.
- Adjust micro-kernel execution order to minimise peak memory footprint of cluster nodes.

The task of data processing on resource-constrained and unreliable hardware emerges within the framework of sensor real-time near-field data processing. The implementation of the system, allowing to carry out the processing in the field, will allow one to quickly respond to sudden changes in sensor readings and reduce the time of decision-making. The implementation of general-purpose computations in such a system allows one to use the same hardware and software system for a diverse high-tech equipment.

The paper is organised as follows: Section 2 presents an overview of related work on using microcomputers for building distributed data processing systems with Hadoop and Spark; Section 3 presents the architecture of our solution; Section 4 explains how Ascheduler is integrated with Apache Spark; Section 5 presents experimental evaluation; Section 6 discusses the results and Section 7 concludes the paper.

2 Related work

There are a number of publications which report on successful deployments of Hadoop and Spark on various resource-constrained platforms:

- Hadoop on Raspberry Pi [3];
- Hadoop on Raspberry Pi [4];
- Spark on Raspberry Pi [8];
- Spark on Cubieboard [9].

These papers outline common problems and solutions when running Hadoop/Spark on resource-constrained systems. These are:

– large memory footprint problems,
– too slow/resource-hungry Java VM,
– overheating problems.

These works do not report any particular problem with Java on resource-constrained platforms and all of them use standard JRE. Neither they report any overheating or large memory footprint problems (although, Raspberry Pi, for example, does not have a cooler). However, all the papers deal with system boards in laboratory or similar environments, where these problems are non-existent. Additionally, the authors run only simple tests to demonstrate that the system is working, and no production-grade application is studied nor large-scale performance tests performed. Using Java and standard JRE for scheduler development seems rational for simple workloads, however, large workloads may require additional boards to cope with memory footprint or boost processing power.

3 Architecture

3.1 Architecture overview

The core concepts and architecture used for the implementation of Ascheduler are described in detail in [5–7]. Here we summarise the most important aspects relevant to the current testbed implementation.

To solve the problem of fault-tolerance of slave cluster nodes we use a simple restart: try to re-execute the task from the failed node on a healthy one. To solve the problem of high-availability of the master node we use replication: copy minimum necessary amount of state to restart the task on the backup node. When the master node fails, its role is delegated to the backup node, and task execution continues. When the backup node fails, the master node restarts the current stage of the task. The most important feature of the approach used in Ascheduler is to ensure master node fault-tolerance without any external controller (e.g. Zookeeper in Hadoop/Spark ecosystem).

Cluster nodes are combined into a tree hierarchy that is used to uniquely determine the master, backup and slave nodes roles without a conflict [7].

Each node may perform any combination of roles at the same time, but can not be both master and backup. The initial construction of the hierarchy is carried out automatically, and the node's position in the hierarchy is solely determined by the position of its IP-addresses in a subnet.

When any cluster node fails or a new one joins the cluster, the hierarchy is rebuilt automatically.

The elasticity of the computer system is provided by dividing each task on a large number of subtasks (called micro-kernels), between which hierarchical links are established. All micro-kernels are processed asynchronously, which makes it possible to distribute them on the cluster nodes and processor cores, balancing the load. Typically, the amount of micro-kernels in a problem exceeds the total number of nodes/cores in the cluster, so the order of their processing can

be optimised so as to minimise memory footprint, or to minimise power consumption by grouping all of the micro-kernels on a small number of nodes, or to ensure the maximum speed of task execution, distributing micro-kernels across all nodes in the cluster. If the cluster capacity is not enough to handle the current data flow/volume of data, micro-kernel pools on the cluster nodes overflow, and excessive kernels may be transferred to a more powerful remote server/cluster. The amount of data, that must be replicated to the backup node to ensure the high-availability, equals to the amount of RAM occupied by a kernel, and can be controlled by the programmer.

Figure 1 shows the schematic view of the system.

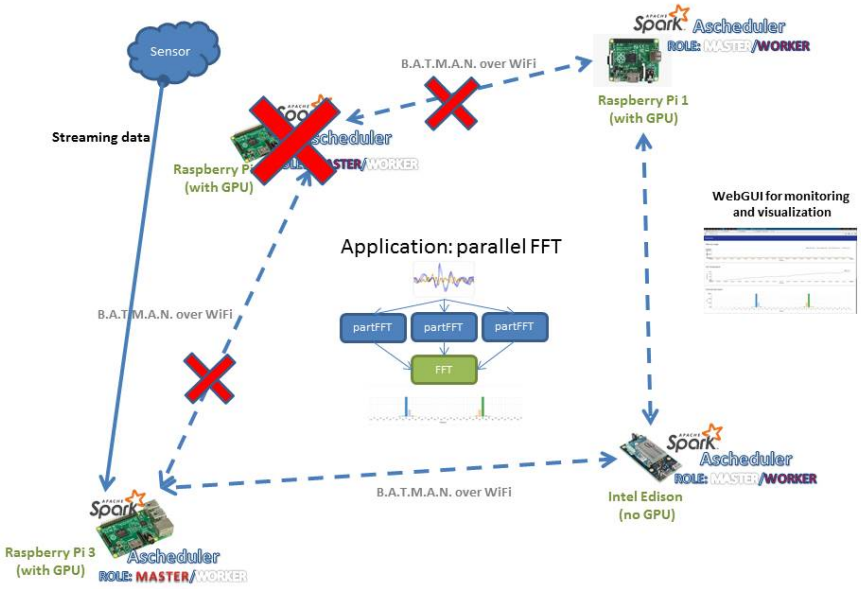

Fig. 1. Schematic view.

3.2 Hardware

Microcomputers used in the testbed:

- Raspberry Pi 3 Model B (2 pieces)
- Raspberry Pi 1
- Intel Edison
- Orange Pi (2 pieces)

3.3 Scheduler core and API

The Ascheduler has layered architecture, as discussed in [5–7]:

- Physical layer. Consists of nodes and direct/routed network links.
- Daemon layer. Consists of daemon processes residing on cluster nodes and hierarchical (master/slave) links between them.
- Kernel layer. Consists of kernels and hierarchical (parent/child) links between them.

Master and slave roles are dynamically assigned to daemon processes, any physical cluster node may become master or slave. Dynamic reassignment uses leader election algorithm that does not require periodic broadcasting of messages, and the role is derived from node's IP address. Detailed explanation of the algorithm is provided in [5].

Software implementation of Ascheduler consists of three main components (Fig. 2):

- Task scheduler core (which is used to compose distributed applications).
- Scheduler daemon based on the core.
- A driver which integrates scheduler into Apache Spark.

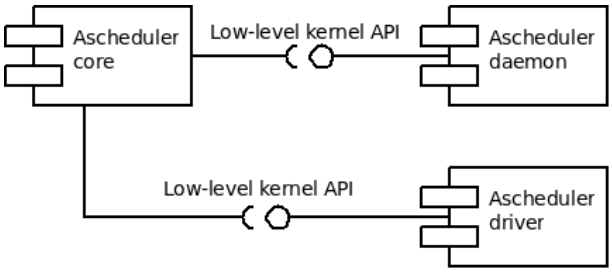

Fig. 2. Scheduler components.

Task scheduler core. The core provides classes and methods to simplify development of distributed applications and middleware. The main focus of this package is to make distributed application resilient to failures, i.e. make it fault tolerant and highly available, and do it transparently to a programmer.

All classes are divided into two layers: the lower layer consists of classes for single node applications, and the upper layer consists of classes for applications that run on an arbitrary number of nodes. There are two kinds of tightly coupled entities in the package — *kernels* and *pipelines* — which are used together to compose a programme. Kernels implement control flow logic in their `act` and `react` methods and store the state of the current control flow branch. Both

logic and state are implemented by a programmer. In `act` method some function is either sequentially computed or decomposed into subtasks (represented by another set of kernels) which are subsequently sent to a pipeline. In `react` method subordinate kernels that returned from the pipeline are processed by their parent. Calls to act and react methods are asynchronous and are made within threads spawned by a pipeline. For each kernel `act` is called only once, and for multiple kernels the calls are done in parallel to each other, whereas `react` method is called once for each subordinate kernel, and all the calls are made in the same thread to prevent race conditions (for different parent kernels different threads may be used).

Pipelines implement asynchronous calls to `act` and `react`, and try to make as many parallel calls as possible considering concurrency of the platform (no. of cores per node and no. of nodes in a cluster). A pipeline consists of a kernel pool, which contains all the subordinate kernels sent by their parents, and a thread pool that processes kernels in accordance with rules outlined in the previous paragraph. A separate pipeline exists for each compute device: There are pipelines for parallel processing, schedule-based processing (periodic and delayed tasks), and a proxy pipeline for processing kernels on other cluster nodes.

In principle, kernels and pipelines machinery reflect the one of procedures and call stacks, with the advantage that kernel methods are called asynchronously and in parallel to each other. The stack, which ordinarily stores local variables, is modelled by fields of a kernel. The sequence of processor instructions before nested procedure calls is modelled by act method, and sequence of processor instructions after the calls is modelled by react method. The procedure calls themselves are modelled by constructing and sending subordinate kernels to the pipeline. Two methods are necessary because calls are asynchronous and one must wait before subordinate kernels complete their work. Pipelines allow circumventing active wait, and call correct kernel methods by analysing their internal state.

Scheduler daemon. The purpose of the daemon is to accept tasks from the driver and launch applications in child processes to run these tasks. Each task is wrapped in a kernel, which is used to create a new child process. All subsequent tasks are sent to the newly created process via shared memory pages, and results are sent back via the same interface. The same protocol is used to exchange kernels between parent and child processes and between different cluster nodes. This allows scheduler daemon to distribute kernels between cluster nodes without knowing exact Java classes that implement kernel interface.

Scheduler daemon is a thin layer on top of the core classes which adds a set of configuration options, automatically discovers other daemons over local area network and launches child processes for each application to process tasks from the driver.

Apache Spark integration driver. The purpose of the driver is to send Apache Spark tasks to scheduler daemon for execution. The driver connects to

an instance of the scheduler daemon via its own protocol (the same protocol that is used to send kernels), wraps each task in a kernel and sends them to the daemon. The driver is implemented using the same set of core classes. This allows testing the driver without a scheduler (replace integration tests with unit tests) as well as using the driver without a scheduler, i.e. process all kernels locally, on the same node where Spark client runs.

Fault tolerance and high availability. The scheduler has fault tolerance and high availability built into its low-level core API. Every failed kernel is restarted on healthy node or on its parent node, however, failure is detected only for kernels that are sent from one node to another (local kernels are not considered). High availability is provided by replicating master kernel to a subordinate node. When any of the replicas fails, another one is used in place. Detailed explanation of the fail over algorithm is provided in [7].

Security. Scheduler driver is able to communicate with scheduler daemons in local area network. Inter-daemon messaging is not encrypted or signed in any way, assuming that local area network is secure. There is also no protection from Internet "noise". Submission of the task to a remote cluster can be done via SSH (Secure Shell) connection/tunnel which is *de facto* standard way of communication between Linux/UNIX servers. So, scheduler security is based on the assumption that it is deployed in secure local area network. Every job is run from the same user, as there is no portable way to switch process owner in Java.

3.4 Ascheduler integration with Spark

Starting with the version 2.0, custom schedulers can be integrated in Spark via implementation of three interfaces. For better understanding of Spark classes and their interconnections please refer to Mastering Apache Spark 2.0 [10] and source code of Spark classes available at `https://github.com/apache/spark`, as sometimes there are useful information in code comments. Class diagram of all implemented Apache Spark interfaces as well as wrapper classes is shown in figure 3.

3.5 Communication

The aim of the project was to build a wireless microcomputer cluster. To create a Wi-Fi based ad hoc network mesh we have chosen a protocol with a driver and API: B.A.T.M.A.N. (Better Approach To Mobile Adhoc Networking mesh protocol) [2]. B.A.T.M.A.N. helps organizing and routing wireless ad-hoc networks that are unstructured, dynamically change their topology, and are based on an inherently unreliable medium. Additionally, B.A.T.M.A.N. provides means to collect the knowledge about the network topology, state and quality of the links — this information is used by Ascheduler to make scheduling decisions aware of physical network topology and links.

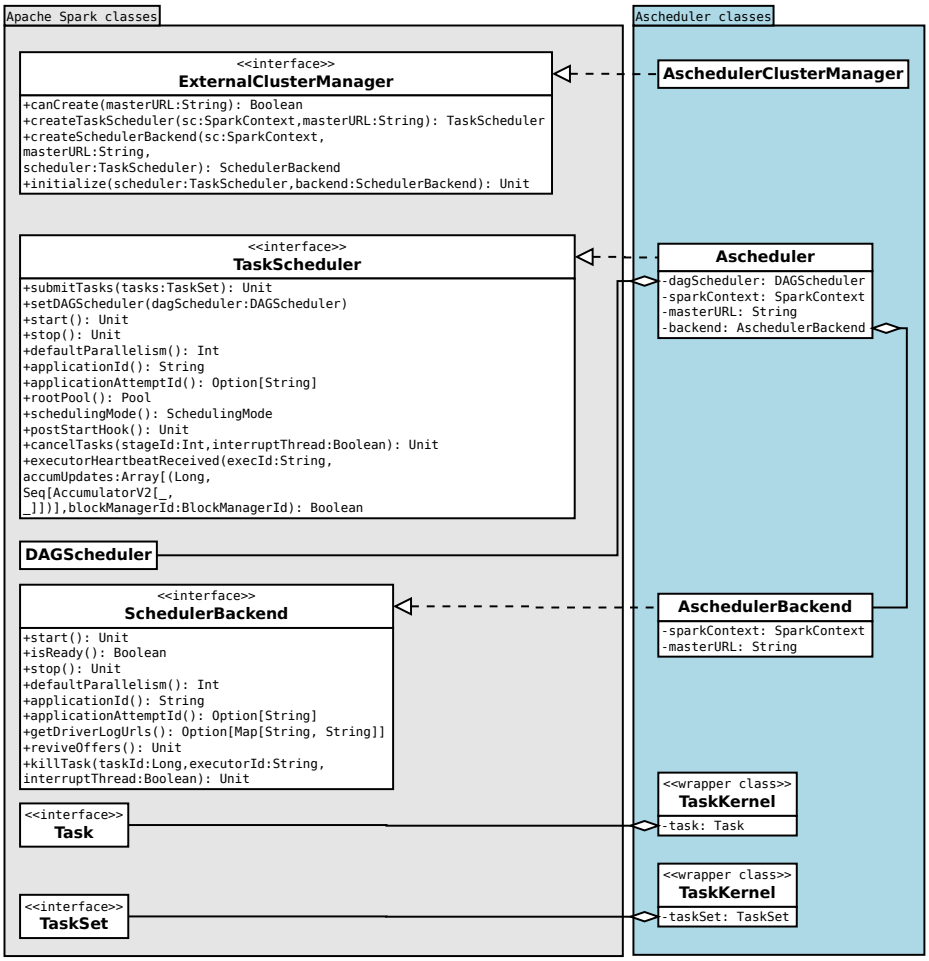

Fig. 3. Apache Spark integration

4 Creating Apache Spark applications for running with Ascheduler

Apache Spark connects to Ascheduler via an implementation of interfaces for custom schedulers. Ascheduler works with Spark version 2.0.2 only. Since Ascheduler integration required access to classes and interfaces considered private in Apache Spark, work of Ascheduler with another versions of Spark is not guaranteed.

Ascheduler integration with Spark has been implemented in a way that allows using Spark functionality disregarding the choice of the scheduler. If Spark is used with several schedulers, the user might want to explicitly choose the scheduling mode. It can be done by creating `SparkContext` from `SparkConf` with method `setMaster(masterURL)` invoked. Here `masterURL` corresponds to particular scheduler with parameters. For Ascheduler string value `ascheduler` could be used for the cluster mode and `ascheduler-local` — for the local mode. Spark driver for Ascheduler has more masterURL options, because of some hardcoded Spark limitations that have to be bypassed:

- `local-ascheduler` for using cluster Ascheduler from Spark shell
- `local-ascheduler-local` for using local Ascheduler from Spark shell
- `local[0_0]-ascheduler-local` for using Spark Streaming with Ascheduler in local version.

Spark programs running on Ascheduler were tested both on local and cluster versions. Spark with Ascheduler supports a wide range of standard operations and functions, such as:

- running both in Spark shell and as standalone applications;
- operating on Resilient Distributed Datasets (RDDs): mapping, reducing, grouping operations;
- partition-wise transformations on RRDs: controllable re-partitioning, shuffling, persisting RDDs, calling functions for partitions;
- Multi-RDD operations: union, subtracting, zipping one RDD with another;
- Broadcasting shared variables among executors;
- Accumulators and task metrics based on them;
- Spark Streaming with rerunning nodes (master included) in case of failure.

The work of Spark with Ascheduler and any of Spark packages except Spark Streaming is not guaranteed. With those exceptions, any Spark application is expected to work with Ascheduler as a task scheduling base.

5 Evaluation

The application used for evaluation is an example of real-time micro-batch processing using Ascheduler and Apache Spark. The application consists of two entities: a periodic signal generator and its processor. The generator creates batches of values of a superposition of harmonic signals and sends them for processing

via a network socket and for output via a websocket. The processor receives the batches from the raw socket, applies adaptive Fast Fourier Transform (FFT) on the signal and sends the result into output via a WebSocket. Both outputs are available on the system monitoring page.

In this experiment we benchmark two implementations of FFT demo application on two platforms using two schedulers (fig. 4). The first implementation is based on Spark Streaming API, the second is based on Ascheduler API. The first platform (left column) is Intel Edison, the second (right column) is commodity Intel Core i5. The first scheduler is Spark Standalone in local mode, the second scheduler is Ascheduler in local mode. Cluster versions are not benchmarked in this experiment. In each run demo application computes spectrum of 25KHz signal in real time for 5 minutes. Time of each spectrum computation is recorded as a point in a graph. Since demo application automatically downsamples input signal when processing is slow, we measure overall throughput by dividing the number of processed points by time taken to process them. The results are presented in fig. 4 for each run and summarised in table 1.

Platform	Scheduler	API	Average throughput, points/s
Intel Edison	Spark	Spark	375
Intel Edison	Ascheduler	Spark	995
Intel Edison	Ascheduler	Ascheduler	517 676
Intel Core i5-4200H	Spark	Spark	487 594
Intel Core i5-4200H	Ascheduler	Spark	511 618
Intel Core i5-4200H	Ascheduler	Ascheduler	5 046 540

Table 1. Comparing performance of Ascheduler and Spark schedulers.

6 Discussion

Graphs show that Spark API is incapable of processing 25KHz input signal on Intel Edison platform. Ascheduler scheduler outperforms Spark standalone by a factor of 3 on Intel Edison but still more performance is needed to process 25KHz signal. Direct use of Ashcheduler API on Intel Edison finally solves the problem, allowing to process 500KHz input signal. On commodity Intel Core i5 platform there is no significant difference between performance of Spark standalone scheduler and Ascheduler when using Spark API, however, direct use of Ascheduler API gives tenfold increase in performance: it is capable of processing 5GHz input signal.

7 Conclusions

The following was achieved as the final outcomes of the project:

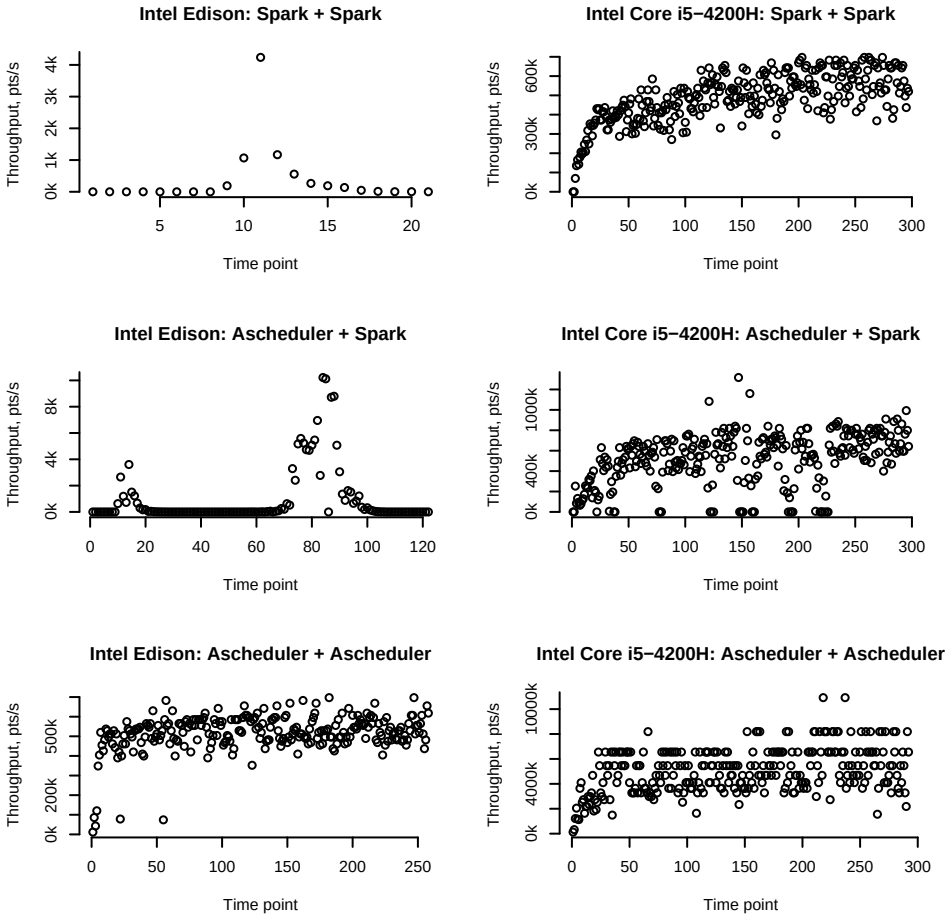

Fig. 4. Comparing performance of Ascheduler and Spark schedulers.

- Ascheduler — fault-tolerant scheduler implemented in Java, running standalone or with Apache Spark (with Spark Streaming supported)
- Master-node fault tolerance is supported by Ascheduler.
- Dynamic resource discovery, composition and re-configuration of distributed cluster.
- Optimised for running on unreliable and resource-constrained microcomputer hardware.
- Running in heterogeneous and dynamic hardware and networking environment.
- Integrated microcomputer and cluster monitoring API.
- Transparent monitoring and visualization with web-based UI.
- Distributed FFT application (with GPGPU support if available) with streaming input and dynamic graphical output.

Acknowledgments

The research was supported by Siemens LLC.

References

1. Apache spark official website. http://spark.apache.org/
2. B.A.T.M.A.N. official web page. https://www.open-mesh.org/projects/open-mesh/wiki
3. Cox, S.J., Cox, J.T., Boardman, R.P., Johnston, S.J., Scott, M., Obrien, N.S.: Iridis-pi: a low-cost, compact demonstration cluster. Cluster Computing 17(2), 349–358 (2014)
4. Fox, K., Mongan, W.M., Popyack, J.: Raspberry hadoopi: a low-cost, hands-on laboratory in big data and analytics. In: SIGCSE. p. 687 (2015)
5. Gankevich, I., Tipikin, Y., Gaiduchok, V.: Subordination: Cluster management without distributed consensus. In: High Performance Computing & Simulation (HPCS), 2015 International Conference on. pp. 639–642. IEEE (2015)
6. Gankevich, I., Tipikin, Y., Korkhov, V., Gaiduchok, V.: Factory: Non-stop batch jobs without checkpointing. In: High Performance Computing & Simulation (HPCS), 2016 International Conference on. pp. 979–984. IEEE (2016)
7. Gankevich, I., Tipikin, Y., Korkhov, V., Gaiduchok, V., Degtyarev, A., Bogdanov, A.: Factory: Master node high-availability for big data applications and beyond. In: International Conference on Computational Science and Its Applications. pp. 379–389. Springer (2016)
8. Hajji, W., Tso, F.P.: Understanding the performance of low power raspberry pi cloud for big data. Electronics 5(2), 29 (2016)
9. Kaewkasi, C., Srisuruk, W.: A study of big data processing constraints on a low-power hadoop cluster. In: Computer Science and Engineering Conference (ICSEC), 2014 International. pp. 267–272. IEEE (2014)
10. Laskowski, J.: Mastering apache spark 2.0. https://www.gitbook.com/book/jaceklaskowski/mastering-apache-spark/details