Comparison of the HPC and Big Data Java Libraries Spark, PCJ and APGAS

Jonas Posner, Lukas Reitz, and Claudia Fohry
Research Group Programming Languages / Methodologies
University of Kassel
Kassel, Germany
{jonas.posner | lukas.reitz | fohry}@uni-kassel.de

***Abstract*—Although Java is rarely used in HPC, there are a few notable libraries. Use of Java may help to bridge the gap between HPC and big data processing.**

This paper compares the big data library Spark, and the HPC libraries PCJ and APGAS, regarding productivity and performance. We refer to Java versions of all libraries. For APGAS, we include both the original version and an own extension by locality-flexible tasks. We consider three benchmarks: Calculation of π from HPC, Unbalanced Tree Search (UTS) from HPC, and WordCount from the big data domain.

In performance measurements with up to 144 workers, the extended APGAS library was the clear winner. With 144 workers, APGAS programs were up to a factor of more than two faster than Spark programs, and up to about 30% faster than PCJ programs. Regarding productivity, the extended APGAS programs consistently needed the lowest number of different library constructs. Spark ranged second in productivity, and PCJ third.

***Index Terms*—Java, Big Data, HPC, Parallel Computing, Spark, PCJ, APGAS**

I. Introduction

Convergence between high performance computing (HPC) and big data processing is a hot topic in current research. Big data programming systems have their strengths in fault tolerance and programming productivity, whereas HPC systems have their strengths in performance and algorithmic flexibility (e. g. [1], [2]). One hindrance to common approaches is the use of different programming languages in the two communities. Typical HPC applications use C/C++ in combination with MPI and/or OpenMP. Typical big data applications, in contrast, use JVM-based languages such as Java or Scala with frameworks such as Hadoop [3] or Spark [4]. While Java is far from prominent in HPC, there are a few notable Java-based libraries such as PCJ [5] and APGAS [6].

The gap between HPC and big data processing can be bridged with interfaces such as *Spark+MPI* [1], *SWAT* [7] and *Alchemist* [8]. These interfaces come at a cost in terms of development effort and computing time. Therefore, use of a unified environment would be more appealing. This paper explores the perspective of a common Java foundation, by comparing the libraries Spark, PCJ and APGAS, regarding productivity and performance.

Spark [4] is an open-source, distributed, multi-threaded, fault-tolerant library for big data processing, and widely used in this domain. Like Hadoop [3], Spark implements the MapReduce pattern [9], but maintains data in memory instead on disc. For storing massive amounts of data, Spark introduces a data structure called Resilient Distributed Dataset (RDD) [10]. Algorithms are implemented via *transformations*, which produce RDDs, and *actions*, which extract a result from an RDD. If MapReduce is not well suited for a particular problem, Spark can cause a significant overhead [8].

The PCJ library for Java [5] won the HPC Challenge Class 2 Best Productivity Award on Supercomputing in 2014, and achieves a better performance than MPI with Java bindings [11]. In some situations, however, the performance of PCJ is up to three times below that of MPI with C bindings [11].

PCJ implements the Partitioned Global Address Space (PGAS) model, which has been designed with productivity and portability in mind. PGAS describes an architecture, such as a cluster of multicore nodes, as a collection of *places*. A place is defined as the combination of a memory partition and computing resources, called *workers*. Typically, places correspond to cluster nodes, and workers correspond to CPU cores. Each place can access every memory partition, but local accesses are faster than remote ones.

PCJ adopts the Single Program Multiple Data (SPMD) execution style, i.e., all workers are invoked at program startup and carry out the same code. Variables are private to each worker, such that different workers can follow different code paths. PCJ provides several methods to exchange data between workers in a synchronous or asynchronous way.

The APGAS library for Java [6] adds asynchronism to the PGAS model by adopting a task-based approach. Its parallelization and distribution concepts are exactly the same as those of IBM's parallel language X10. Program execution starts with a single task on place 0. Later, any task can spawn any number of child tasks dynamically. Task spawning can be synchronous or asynchronous, i.e., the parent task either waits, or does not wait, for the termination of a child. In either case, the programmer must specify an execution place for each task. Inside each place, tasks are automatically scheduled to workers. Thus, depending on worker availability, a generated task may run immediately or later. The place-internal scheduler is implemented with Java's Fork/Join-Pool [12].

In recent work [13], [14], we extended the APGAS library by locality-flexible tasks. These tasks are spawned and man-

aged in the same way as the standard asynchronous tasks of APGAS, except that the programmer does not specify an execution place. Instead, locality-flexible tasks are subject to *system-wide* automatic load balancing. We implemented the concept by extending the Fork/Join pools by an inter-place work stealing scheme that realizes the lifeline-based global load balancing (GLB) algorithm [15]. We denote the original APGAS version by $APGAS_{stat}$, and our extension by $APGAS_{dyn}$.

This paper compares Spark, PCJ, APGAS_{stat} and APGAS_{dyn}. Although part of the libraries can also be used with other languages, we always refer to the Java versions for a meaningful comparison. Use of Java may set Spark at a disadvantage, since its native language is Scala. For the comparison, we selected benchmarks from both HPC and big data processing. Moreover, we took care to implement the same algorithm in each system. Three benchmarks were included:

- Pi: Monte-Carlo estimation of π.
- UTS: Unbalanced Tree Search [16]. This benchmark dynamically generates a highly irregular tree and counts the number of nodes.
- WordCount: This canonical MapReduce example counts the occurrences of each word in a number of input files.

We compare the libraries with respect to programmer productivity and performance. Regarding productivity, we first detail our subjective impressions from developing the benchmarks. The discussion is supported by codes, which are depicted in full for Pi. Second, productivity is assessed with two objective metrics: lines of code (LOC), and number of different library constructs used (NLC). The second metric indicates learning overhead and complexity. Overall, we found APGAS, and especially APGAS_{dyn}, to be the most productive system, followed by Spark and PCJ.

Performance measurements were conducted on an Infiniband-connected cluster with 12 homogeneous 12-core nodes. We measured both intra-node and inter-node performance. Results show APGAS_{dyn} as a clear winner. With 144 workers, APGAS_{dyn} was between 5.9% and 17.11% faster than APGAS_{stat}, between 8.76% and 56.81% faster than Spark, and between 9.28% and 28.88% faster than PCJ. In summary, our results suggest that APGAS_{dyn} may be a strong candidate for both HPC and big data processing.

The paper is structured as follows. Section II provides background on Spark, PCJ and the APGAS variants. Then, Section III describes and discusses our performance measurements. Section IV is devoted to productivity, and includes both personal impressions and metrics. The paper finishes with related work in Section V, and conclusions in Section VI.

II. Background

A. Spark

Spark [4] is an open-source, distributed, multi-threaded, in-memory, fault-tolerant library for data-intensive cluster computing. The library was initially released in 2010. Meanwhile, it is maintained by the Apache Software Foundation, and is online available as a repository [17]. Examples of typical Spark applications include iterative processing in machine learning, and interactive data analysis. Spark is implemented in Scala, but can also be used with Java, Python and R.

Like Hadoop [3], Spark implements the MapReduce pattern [9]. It addresses Hadoop's I/O performance bottleneck by maintaining data in memory rather than on disk. This yields a speedup by a factor of up to 100 for iterative algorithms [4].

Spark's primary data abstraction is called Resilient Distributed Dataset (RDD) [10]. An RDD is a resilient, immutable, and distributed data structure. It contains a set of elements and provides operations for

- producing new RDDs (called *transformations*), and
- computing return values (called *actions*).

Common examples of transformations and actions are the well-known operations map and reduce, respectively.

The creation of RDDs is lazy, which prevents unnecessary memory usage and minimizes computations. Creations are only triggered when an action is called. Fault tolerance for RDDs is achieved by storing the operation sequence and recomputing lost data after failures. This technique, called *ancestry tracking*, does not require backups.

A Spark program starts by creating a `SparkContext` object, to which a `SparkConf` object is passed. The `SparkContext` object lives in an extra JVM, which is called *driver*. The `SparkConf` object contains information about the execution, e. g. on how many JVMs the program should run, how many JVMs are mapped to each node, the number of worker threads per JVM, and an upper bound on memory usage. JVMs are called *executors*. Each executor has the same number of workers.

Since the driver executes the `main` method, transformations and actions are called within the driver's JVM. When an action is called, the driver splits all computations of the required operations into tasks and distributes them over executors. The result of the action is returned to the driver.

An RDD can be created, e. g., by passing a local data collection to method `parallelize()` of the `SparkContext` object, or by passing URIs of text files to function `textFile()`. Further transformations include:

- `map()`: Applies a passed function to each data element and returns the resulting RDD.
- `flatMap()`: Similar to `map()`, but the passed function may produce for each data element multiple new elements instead of a single one.
- `filter()`: Returns a new RDD containing the data elements that match a passed `boolean` function.
- `reduceByKey()`: Returns a new RDD, in which each executor's subset of the data is reduced to a single value.

Actions include:

- `count()`: Returns the number of data elements in an RDD.
- `collect()`: Returns an array with all data elements of an RDD.

- `reduce()`: Pulls all data elements to the driver, aggregates them with a passed function, and returns a single value.

Spark offers several deployment options. For instance, standalone mode is the simplest option on a private cluster, and Mesos [18] is a dedicated cluster manager.

B. PCJ

The PCJ library [5] implements the PGAS model in Java and is available as an open-source repository [19]. PCJ is targeted at large-scale HPC applications, but was also observed to be suitable for big data applications [20]. The library is shipped as a single jar file without dependencies on other libraries. This is an advantage over the Spark distribution, which involves many dependencies, and over the APGAS distribution, which involves a few. Currently, PCJ offers no fault tolerance mechanism, but it is in development [21].

Execution units in PCJ are called *workers*. Technically, a worker is realized by a Java thread that maintains its own local memory. Each place runs in a separate JVM. Different places can use a different number of workers. The numbers of places and workers are configurable, see below.

At program startup, all workers run in parallel. Each worker executes the same `main` method, which is specified in a `Startpoint` interface. Still, different workers can follow different code paths.

Variables can be declared as shared to permit access by other workers. For the declaration, the variables must simultaneously be fields of an `enum`, and of a class that implements `StartPoint`. Moreover, the `enum` must be annotated with `@Storage(Class)`, and the class should be annotated with `@RegisterStorage(Enum)`.

Details of the communication between workers are hidden from PCJ programmers. Instead, the PCJ runtime automatically determines whether a communication is place-internal or global, and selects an appropriate communication mechanism. For place-internal communication, it deploys Java's concurrency constructs, and for global communication it deploys network sockets. Global communication is further optimized by arranging places in a graph.

The PCJ library includes a launcher, which starts applications on multiple cluster nodes. It takes as input a text file that contains a list of nodes with duplicates. The launcher starts one place for each of the different entries in the list, and one worker for each individual entry. PCJ includes the following methods and classes:

- `deploy()`: Deploys a PCJ application across a cluster. The method passes a text file to the launcher, as described before.
- `myID()`: Returns the id of the calling worker. The ids are consecutive numbers starting with 0.
- `threadCount()`: Returns the total number of workers system-wide.
- `getNodeCount()`: Returns the number of places.
- `getNodeId()`: Returns the id of the calling place.
- `barrier()`: Synchronizes all workers. When a worker reaches this call, it stops execution and only resumes when all workers have reached the same line in their respective codes. All workers must execute this line. A variant of the method supports pairwise synchronization of two workers.
- `get()`: Synchronously reads the value of a shared variable from the local memory of a particular worker.
- `put()`: Synchronously stores a value into a shared variable in the local memory of a particular worker.
- `asyncGet()`: Asynchronous variant of `get()`. Returns a `PcjFuture` object.
- `asyncPut()`: Asynchronous variant of `put()`. Returns a `PcjFuture` object.
- `PcjFuture`: Provides a `get()` method, which waits for the completion of the underlying operation, and returns the result (if any).
- `broadcast()`: Sends a value to all workers and writes it into their respective instances of a shared variable with the same name, which is passed as a parameter.
- `waitFor()`: Waits until the value of a given shared variable has changed.

C. APGAS

The "APGAS for Java" library [6] was developed in a branch of IBM's X10 language project [22]. APGAS brings the parallelization and distribution concepts of X10 to Java and "supports resilient, elastic, parallel, distributed programming on clusters of JVMs" [6]. A similar library exists for Scala [23].

As mentioned in Section I, in recent work we extended APGAS by locality-flexible tasks that are subject to system-wide automatic load balancing [13], [14]. In the following, we first describe the original APGAS library $APGAS_{stat}$, and then our extension APGAS_{dyn}. Since $APGAS_{stat}$ is a subset of APGAS_{dyn}, we use the abbreviation APGAS when referring to common functionalities.

APGAS realizes an asynchronous variant of the PGAS model. Computations are encapsulated in lightweight asynchronous tasks (which were called activities in [6]). Like in PCJ, Programmers have full control over the mapping of data and tasks to places.

Internally, each place maintains a task pool, which is an instance of Java's Fork/Join-Pool. Thus, like in PCJ, there is a fixed number of workers, which correspond to Java threads. Unlike in PCJ, each APGAS place has the same number of workers. Spawned tasks are inserted into the task pool of a particular place, which must be specified by the programmer. Since Java's Fork/Join-Pool assigns tasks in arbitrary order, a task may be executed instantly or at any time later.

If a task is mapped to a remote place, final and effectively final variables from the origin place are copied and sent along transparently, if accessed remotely. Exceptions that are raised on the remote place can be caught on the origin place by a surrounding `try-catch` block, if a synchronization point is available.

Distributed collections of at most one object per place can be created with the `GlobalRef<T>` construct. If a `GlobalRef` object is dereferenced at a particular place by calling `get()`, the respective local object (if any) is returned.

APGAS does not explicitly offer constructs for intra-place concurrency control, but relies on Java's extensive facilities such as the `synchronized` keyword.

APGAS provides different launchers for deploying an application on multiple nodes. For instance, the `SSHLauncher` starts all places per `ssh`. The physical nodes to be used can be passed in a text file. The number of workers per place can be configured by providing an option at program start. Each place is realized by a single JVM, and multiple places can be mapped to the same node. The JVMs are interconnected with the help of the Hazelcast open-source Java library [24].

APGAS includes the following constructs:

- `places()`: Returns a list of all places.
- `localWorkers()` : Returns the number of workers per place.
- `here()`: Returns the current place.
- `async()`: Provides the simplest way to create a new task. The task is inserted into the local Fork/Join pool of the calling place. A call of this construct returns immediately and has no return value.
- `at()`: Creates a new task and inserts it into the local Fork/Join pool of a specified remote place. A call of this construct blocks until the task has returned. The construct permits a return value.
- `asyncAt()`: Like `at`, this construct creates a new task and inserts it into the local Fork/Join pool of a specified remote place. Unlike `at`, the construct returns immediately and does not allow a return value.
- `finish()`: This construct defines a code block, in which `async`- and `asyncAt`-tasks can be spawned. At the end of the block, program execution suspends until all spawned tasks, including recursively spawned ones, have been processed. Inside a `finish` block, exceptions are accumulated and can be caught by a surrounding `try-catch` block.

APGAS$_{dyn}$ extends the open-source code of the official APGAS repository [25]. We maintain an own APGAS repository, which is a fork of the official repository, plus our novel features and some bug fixes [26].

The development of APGAS$_{dyn}$ was motivated by the existence of tasks that may run equally well on any resource of the overall system. We call them *locality-flexible*. In APGAS$_{stat}$, a programmer has to to specify an execution place for such tasks, which causes unneeded programming effort and restricts the flexibility of the runtime system, APGAS$_{dyn}$ introduces constructs for submitting and managing locality-flexible tasks. These tasks are automatically scheduled over all places by work stealing, such that load balancing is achieved without any effort for the programmer. Additionally, locality-flexible tasks can be canceled.

APGAS$_{dyn}$ includes the following constructs:

- `asyncAny()`: Creates a locality-flexible task and inserts it into the local task pool of the calling place. The task can later be stolen away to other places by the runtime system.
- `staticAsyncAny()`: Creates a user-defined number of `asyncAny`-tasks and initially distributes them evenly over all places.
- `finishAsyncAny()`: Defines a `finish` block, which terminates only when all `asyncAny`-tasks spawned inside, including recursively spawned ones, have been processed.
- `mergeAsyncAny()`: Merges a passed value into the partial result of the local worker. Such partial results are maintained by all workers and form the basis for computing a global result by reduction afterwards. The passed value can be of a primitive type, or be an object of a user class that extends `ResultAsyncAny<T>`.
- `reduceAsyncAny()`: Computes and returns the current global result by reduction over all partial worker results.
- `cancelableAsyncAny()`: Resembles `asyncAny`, but the created task can be canceled later.
- `cancelAllCancelableAsyncAny()`: Cancels all unprocessed cancelable `asyncAny`-tasks and prohibits spawning new ones.

III. Performance Evaluation

In the following, we describe the hardware and software setup for our experiments. Afterwards, we present each benchmark and its experimental results. The last paragraph discusses the results.

A. Setup

Experiments were conducted on a cluster with 12 homogeneous nodes [27]. Each node comprises two 6-core Intel Xeon E5-2643 v4 CPUs, and 256 GB of main memory. All nodes are interconnected with Infiniband. The cluster was used exclusively, i.e., there were no other applications running concurrently. The cluster uses operating system CentOS in version 7.2, workload manager Slurm in version 17.11, and the GPFS distributed file system in version 4.2.3-8. For Spark, we used version 2.3.0. For PCJ, we used version 5.0.6 in its latest available revision (May 29, 2018) from the official repository [19]. As mentioned before, we maintain our own APGAS repository [26]. Both APGAS$_{stat}$ and APGAS$_{dyn}$ were deployed from there in their latest revisions (August 14, 2018). As mentioned in Section II-C, APGAS relies on Hazelcast for networking, which we used in version 3.10. Since Spark is not compatible with the current Java release, we deployed Java in version 1.8.0_172 for all Spark benchmarks. However, for PCJ and APGAS, we deployed the most recent Java release 10.0.1. We did not specifically configure the JVMs, but used the default settings of the respective Java versions.

In a first group of experiments, we measured intra-node performance. We utilized one JVM, and varied the number of workers from 1 to 12. For Spark, we also started the driver on

the same node. Then, we measured inter-node performance, for which we varied the number of nodes from 1 to 12. Here, one JVM with 12 workers was run on each node. Therefore, we started up to 144 workers. For Spark, the driver was placed on one of the 12 nodes.

For each benchmark, we used strong scaling (fixed global problem size). Each configuration was executed five times and the average value of these runs is reported. The numbers given below include the entire program execution time, except for the initialization time of the runtime system. Tables I–IV depict our results in seconds for the different benchmarks and libraries.

To jump ahead briefly, APGAS$_{\boldsymbol{dyn}}$ performs best in most cases. Therefore, to illustrate performance differences more clearly, Figures 1–4 depict the *overhead* of the other libraries compared to APGAS$_{\boldsymbol{dyn}}$. The overhead is specified as percentage, and is calculated with the formula $\texttt{time}_x$ / $\texttt{time}_{APGAS_{dyn}} - 1$, where $x \in \{Spark, PCJ, APGAS_{stat}\}$.

B. The Pi benchmark

The approximation of π by a Monte Carlo algorithm is a simple computation-intensive benchmark. It generates n points inside a unit square and counts the number of points that fall inside the corresponding unit circle. Afterwards the value of π is calculated with the formula $4 * numInside/n$.

Parallelization of this algorithm is straightforward: The overall work is split into independent tasks, which are distributed over all resources. Each task generates a specified number of random points, and returns a single `long` value, which expresses the number of points inside the circle. The overall number of points inside the circle is computed by summation.

The number of random points per task is calculated by the formula $n/(numWorker \cdot tasksPerWorker)$, where $tasksPerWorker$ is a parameter. We set $n = 2^{40}$ and $tasksPerWorker = 64$, which we experimentally determined as the best value for all systems.

Figure 1 depicts the overheads over APGAS$_{dyn}$ as percentage, and Table I shows the absolute execution times in seconds. Overall, APGAS$_{dyn}$ has the best performance, and a speedup of 127.30 with 144 workers.

Spark has a large overhead over APGAS$_{dyn}$ for low worker counts, e. g. 57.57% with two workers. Between 10 and 144 workers, the overhead of Spark ranges between 2.63% and 9.60%.

PCJ has an overhead over APGAS$_{dyn}$ of at most 10.84% with 72 workers. With 5 workers, PCJ performs 0.42% better than APGAS$_{dyn}$. Between 10 and 144 workers, the overhead of PCJ ranges between 6.99% and 10.84%.

The performance of APGAS$_{stat}$ differs only slightly from that of APGAS$_{dyn}$. Sometimes, APGAS$_{stat}$ is better, by a maximum of 4.82% with 4 workers. With more than 60 workers, APGAS$_{stat}$ is consistently slower than APGAS$_{dyn}$, with an overhead of at most 6.27% with 144 workers.

C. Unbalanced Tree Search

The Unbalanced Tree Search (UTS) benchmark [16] generates a highly irregular tree from SHA1 values. It starts with the node descriptor of the root node of the tree. Node descriptors contain all information that is necessary to construct the children, and thus the subtree, of the corresponding node. The work load is not initially known, because processing a node descriptor can generate an unknown number of children. The result of a run is a single `long` value, which indicates the number of processed tree nodes.

In previous work [14], UTS was ported from X10's official GLB examples repository [22] to APGAS$_{dyn}$. The APGAS$_{dyn}$ algorithm starts with a single task that processes the root node. The same task continues processing the root's children, grandchildren etc., in a depth-first manner. As soon as 511 tree nodes have been generated, it spawns a second task, such that both tasks are responsible for half of the generated tree nodes. The computation continues, and whenever 511 tree nodes have been generated, the current task is split into two.

Since Spark, PCJ and APGAS$_{stat}$ do not support automatic system-wide load balancing, we initially calculate the tree sequentially up to a certain depth, called $treeDepth$. The resulting tree nodes are then split into $numWorker * tasksPerWorker$ tasks, which are distributed evenly to all workers. We set both $tasksPerWorker = 64$ and $seqTreeDepth = 6$. These values were determined experimentally, and perform best for all systems.

We run UTS with the following parameters [16]:

- geometric tree shape,
- branching factor = 4,
- random seed = 19,
- tree depth = 16.

Figure 2 depicts the overheads over APGAS$_{dyn}$ as percentage, while Table II lists the measured execution times in seconds.

Overall, APGAS$_{dyn}$ has the best performance, and a speedup of 89.97 with 144 workers.

Spark has its lowest overhead of 10.84% over APGAS$_{dyn}$ with one worker. For multiple JVMs, the overhead varies between 38.32% (122 workers) and 57.10% (48 workers).

PCJ performs better than APGAS$_{dyn}$ by 0.49% with one worker. Otherwise, PCJ performs worse. For multiple JVMs, the overhead varies between 40.60% (144 workers) and 89.65% (24 workers).

APGAS$_{stat}$'s overhead over APGAS$_{dyn}$ is rather low inside a place. On multiple JVMs, it varies between 20.62% (144 workers) and 68.41% (24 workers).

D. WordCount

WordCount is a canonical map-reduce application. A specified number of input files are read line-by-line, and the occurrences of each word are counted and stored as individual key/value pairs. In the parallel variant, reading and counting are distributed, and partial results are computed locally first. The reduction step accumulates the partial results. The result of a run is a key/value map.

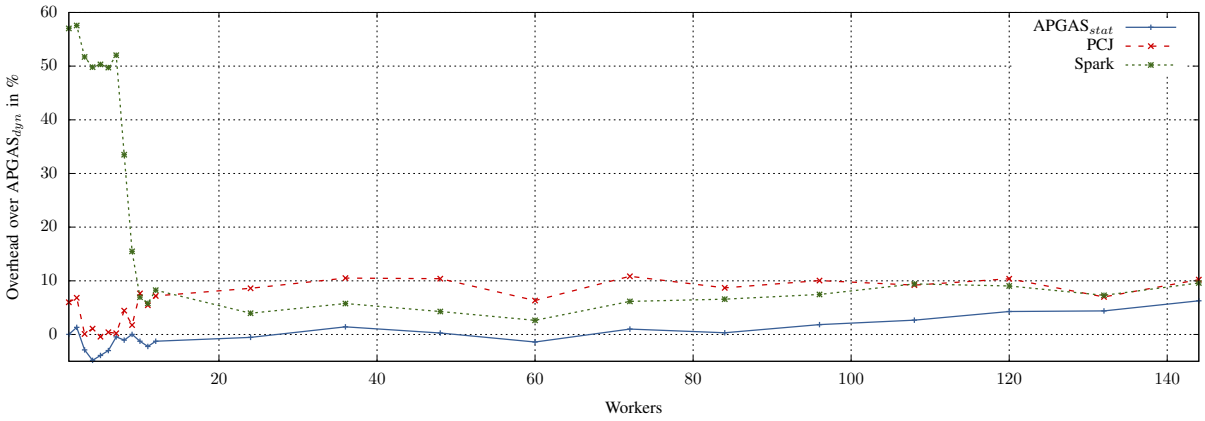

Fig. 1: Pi: Overhead over APGAS$_{dyn}$

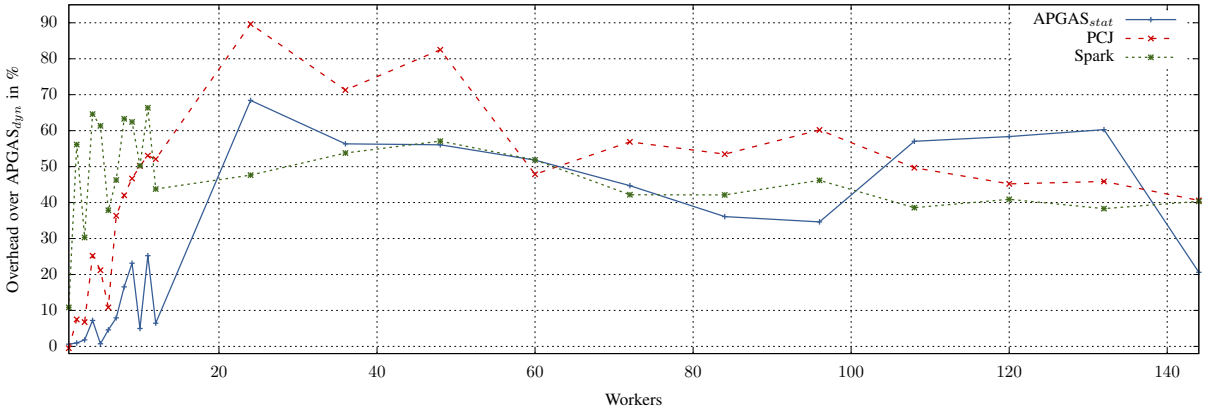

Fig. 2: UTS: Overhead over APGAS$_{dyn}$

Workers	Spark	PCJ	APGAS$_{stat}$	APGAS$_{dyn}$
1	8889.44	6000.49	5662.12	5661.01
2	4468.84	3029.61	2873.45	2836.01
4	2275.39	1535.15	1445.45	1518.72
8	1021.40	799.08	757.03	765.11
12	544.21	538.82	496.28	502.67
24	267.83	279.83	256.23	257.63
48	135.69	143.65	130.49	130.14
96	71.20	72.92	67.48	66.27
144	48.74	49.02	47.26	44.47

Tab. I: Pi: Execution time in seconds

Workers	Spark	PCJ	APGAS$_{stat}$	APGAS$_{dyn}$
1	1429.09	1282.95	1297.22	1289.32
2	1023.32	704.67	661.67	655.44
4	600.29	456.63	390.99	364.70
8	305.75	265.96	218.23	187.25
12	197.88	209.35	146.53	137.65
24	99.94	128.40	114.02	67.70
48	53.87	62.58	53.52	34.29
96	28.83	31.59	26.55	19.72
144	20.11	20.15	17.29	14.33

Tab. II: UTS: Execution time in seconds

Like [20], we selected two novels as input: Lev Tolstoy's *War and Peace* [28] (written in English, 3.3 MB), and Georges des Scudéry's *Artamène ou le Grand Cyrus* [29] (written in French, 10 MB). Both are encoded in UTF-8. In order to achieve a larger amount of data, each file is read 32 768 times, resulting in 105 GB and 320 GB input data, respectively. The count of 32 768 is evenly distributed over all workers, such that each worker reads and processes $32\,768/totalWorkers$ files successively. In APGAS$_{dyn}$, 32 768 locality-flexible tasks are spawned.

Figures 3 and 4 depict the overheads over APGAS$_{dyn}$ as percentage, and Tables III and IV show the absolute execution times in seconds.

When using *War and Peace*, APGAS$_{dyn}$ always has the best performance, and a speedup of 63.80 with 144 workers. When using *Artamène ou le Grand Cyrus*, APGAS$_{dyn}$ again has the best performance, and a speedup of 96.78 on 144 workers. The only exception occurs for one worker, where PCJ is faster by 2.28%.

For the first novel, Spark has an overhead over APGAS$_{dyn}$ of at most 133.54% with 9 workers. With an increasing number of workers the overhead decreases, but with 144 workers it

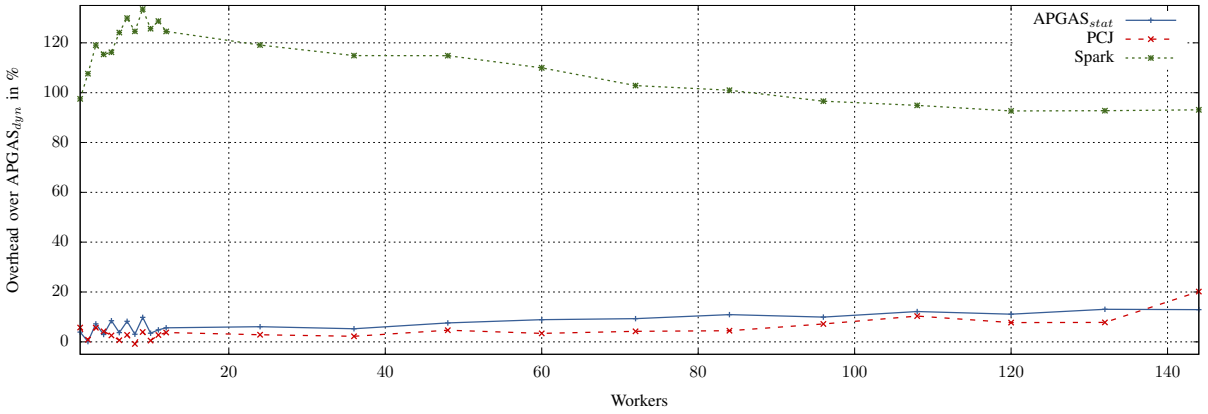

Fig. 3: WordCount using *War and Peace*: Overhead over APGAS$_{dyn}$

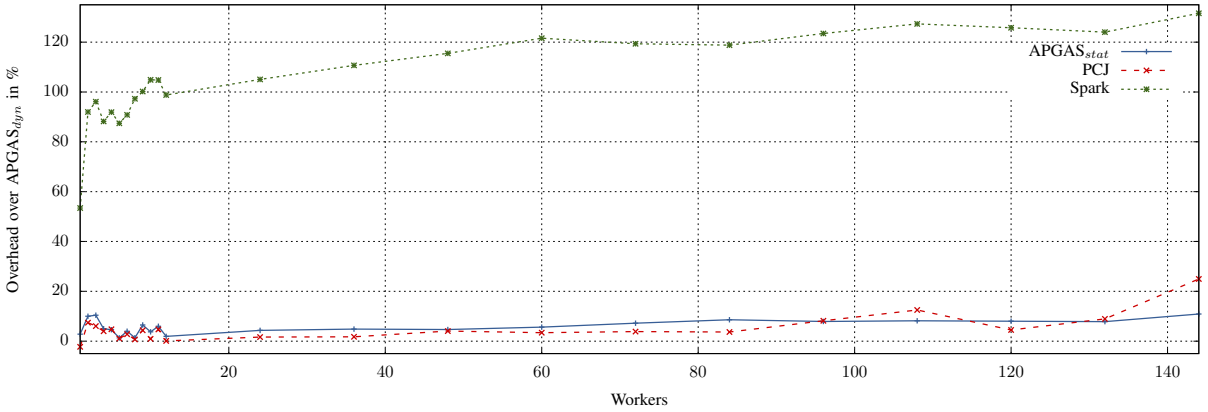

Fig. 4: WordCount using *Artamène ou le Grand Cyrus*: Overhead over APGAS$_{dyn}$

Workers	Spark	PCJ	APGAS$_{stat}$	APGAS$_{dyn}$
1	4136.31	2214.35	2176.38	2094.35
2	2299.44	1117.01	1108.43	1107.50
4	1268.73	614.03	606.41	589.11
8	754.19	332.99	346.04	335.80
12	554.54	256.12	260.81	246.92
24	287.46	134.91	139.14	131.19
48	155.69	75.83	77.95	72.46
96	84.56	46.10	47.28	43.01
144	63.40	39.46	37.70	32.83

Tab. III: WordCount using *War and Peace*: Execution time in seconds

Workers	Spark	PCJ	APGAS$_{stat}$	APGAS$_{dyn}$
1	12713.30	8097.07	8518.37	8286.21
2	7649.71	4279.59	4382.66	3984.65
4	4185.24	2311.64	2334.54	2223.67
8	2375.39	1212.08	1220.22	1203.99
12	1749.49	880.88	896.75	880.01
24	911.87	451.96	463.93	444.72
48	495.04	238.97	240.41	229.71
96	271.33	131.32	131.01	121.41
144	198.28	106.97	94.95	85.62

Tab. IV: WordCount using *Artamène ou le Grand Cyrus*: Execution time in seconds

is still at a high value of 93.11%. Results are similar for the second novel, where Spark's overhead over APGAS$_{dyn}$ increases with the number of workers from 53.43% with 1 worker to 131.58% with 144 workers.

The overheads of PCJ and APGAS$_{stat}$ over APGAS$_{dyn}$ tend to increase with the number of workers. The overhead of PCJ varies between 0.65% (6 workers) and 20.19% (144 workers) for the first novel, and is up to 24.94% (144 workers) for the second novel. The overhead of APGAS$_{stat}$ varies between 0.08% (24 workers) and 13.07% (144 workers) for the first novel, and is up to 10.90% (144 workers) for the second novel.

E. Discussion

Overall, APGAS$_{dyn}$ outperforms the other systems, because it is the only system that provides load balancing at both the intra- and inter-node levels. As expected, the advantage is particularly clear for the dynamic workloads of UTS. For static workloads, like those of Pi and WordCount, the advantage is smaller, but still noticeable.

The PCJ programs can be extended manually by dynamic load balancing. Since, in our experiments, PCJ and APGAS$_{stat}$ achieved a similar performance, we expect that such an extension may at best bring the PCJ performance close to APGAS$_{dyn}$, but at the price of an even lower productivity than described in Section IV.

Spark does not provide appropriate constructs for manually implementing dynamic load balancing. Overall, the Spark programs have the lowest performance. Surprisingly, WordCount, which is a typical big data benchmark, needed approximately twice the time of its APGAS$_{dyn}$ counterpart. We do not know the reason for this result. Possible explanations include the use of an older Java version, deployment of Java instead of Scala, and a rather low machine size.

IV. Productivity of the Libraries

The term programming productivity denotes the efficiency of program development. Of course, productivity is subjective, since it depends on the user to some degree. To be as fair as possible, all benchmarks were developed by the same person, namely the second author of this paper, who had no previous experience with any of the systems. In the following, we describe and discuss his impressions, referring to code examples. Moreover, the discussion includes two objective metrics: number of different library constructs used (NLC), and lines of code (LOC). The NLC metric reflects learning overhead and complexity. For LOC, we only count lines containing code.

Listings 1–4 depict the source codes for Pi. The codes are almost complete, except that a few code snippets have been shortened. In particular, the listings for Spark and the APGAS variants only show the contents of the `main` method, because the associated classes comprise standard elements only. Since PCJ requires class annotations to declare variables as shared, the class is included for PCJ in Listing 1.

Table V reports the NLC values of our codes. As constructs, we count methods, classes etc., as provided by the libraries. Counted constructs are colored in green in Listings 1–4. As shown in Listing 1, the PCJ constructs refer to worker control (e. g. line 29), worker communication (e. g. line 33), shared variable declarations (e. g. line 11), and system control (e. g. line 21). APGAS constructs, as depicted in Listing 2, chiefly refer to task spawning (e. g. line 13) and distributed data structures (e. g. line 7). Finally, as depicted in Listing 4, Spark constructs include transformations (e. g. line 16), actions (e. g. line 24), and system control (e. g. line 8).

The APGAS variants have the lowest NLC value, with a minor advantage for APGAS$_{dyn}$. Spark always ranks third, requiring up to four constructs more than APGAS$_{dyn}$. PCJ always has the highest NLC value, and needs about twice as many constructs as APGAS$_{dyn}$.

One reason for this outcome can be seen in Spark's and PCJ's use of constructs to start the library runtime, see lines 8 and 9 in Listing 4 and lines 2, 21, 22, 25 and 26 in Listing 1, respectively. In contrast, an APGAS program is started automatically when calling the first construct.

	Spark	PCJ	APGAS$_{stat}$	APGAS$_{dyn}$
Pi	7	12	6	7
UTS	6	15	6	5
WordCount	10	12	7	6

Tab. V: Number of different library constructs used (NLC)

	Spark	PCJ	APGAS$_{stat}$	APGAS$_{dyn}$
Pi	29	67	36	31
UTS	28	78	64	38
WordCount	46	76	75	74

Tab. VI: Lines of code (LOC)

```
1  @RegisterStorage(PCJPi.Shared.class)
2  public class PCJPi implements StartPoint {
3
4   public static int n = 0;
5   public static long tasksPerWorker = 0;
6   public long points = 0;
7   public long c = 0;
8   public long tasksPerPlace = 0;
9   public static AtomicLong remainingTasks;
10
11  @Storage(PCJPi.class)
12  enum Shared {
13   c,
14   points,
15   tasksPerPlace
16  }
17
18  public static void main(String[] args) {
19   n = Integer.parseInt(args[1]);
20   tasksPerWorker = Integer.parseInt(args[2]);
21   NodesDescription n =new NodesDescription(args[0]);
22   PCJ.deploy(PCJPi.class, n);
23  }
24
25  @Override
26  public void main() {
27   points = 1L << n;
28   int nodes = PCJ.getNodeCount();
29   int worker = PCJ.threadCount();
30   int workerPerPlace = nodes / worker;
31   tasksPerPlace = workerPerPlace * tasksPerWorker;
32   PCJ.barrier();
33   long myTPP = PCJ.get(0, Shared.tasksPerPlace);
34   remainingTasks = new AtomicLong(myTPP);
35   points = PCJ.get(0, Shared.points);
36   long nAll = points;
37   long pointsPerTask = nAll / (myTPP * nodes);
38   long tmpCount = 0;
39   PCJ.barrier();
40   while (remainingTasks.decrementAndGet() >= 0) {
41    for (long i = 0; i < pointsPerTask; i++) {
42     double x = 2 * randomDouble() - 1.0;
43     double y = 2 * randomDouble() - 1.0;
44     tmpCount += (x * x + y * y <= 1) ? 1 : 0;
45    }
46   }
47   c = tmpCount;
48   PCJ.barrier();
49   if (PCJ.myId() == 0) {
50    PcjFuture<Long> cL[] = new PcjFuture[worker];
51    long c0 = c;
52    for (int p = 1; p < worker; p++) {
53     cL[p] = PCJ.asyncGet(p, Shared.c);
54    }
55    for (int p = 1; p < worker; p++) {
56     c0 = c0 + cL[p].get();
57    }
58    println("Pi is roughly " + 4.0 * c0 / nAll);
59   }
60 }}
```

Lst. 1: PCJ: Code for Pi

```
1  long points = 1L << Integer.parseInt(args[0]);
2  int tasksPerWorker = Integer.parseInt(args[1]);
3  int allWorkers = localWorkers() * places().size();
4  int numTasks = allWorkers * tasksPerWorker;
5  long pointsPerTask = points / numTasks;
6
7  GlobalRef<AtomicLong> result = new GR<>(new AL());
8
9  finish(() -> {
10  for (Place p : places()) {
11   for (int j = 0; j < workerPerPlace; ++j) {
12    for (int t = 0; t < tasksPerWorker; t++) {
13     asyncAt(p, () -> {
14      long tmpCount = 0;
15      for (long i = 0; i < pointsPerTask; ++i) {
16       double x = 2 * randomDouble() - 1.0;
17       double y = 2 * randomDouble() - 1.0;
18       tmpCount += (x * x + y * y <= 1) ? 1 : 0;
19      }
20      long transferCount = tmpCount;
21      asyncAt(result.home(), () -> {
22       result.get().addAndGet(transferCount);
23      });
24     });
25    }
26   }
27  }
28 });
29 long count = result.get().get();
30 println("Pi is roughly " + 4.0 * count / points);
```

Lst. 2: APGAS$_{stat}$: Code for Pi

Table VI reports the LOC metric for all codes. For a fair comparison, codes were styled in the same way, according to the Google Java Style Guide [30].

As the table shows, Spark always has the lowest LOC value, while APGAS$_{dyn}$ ranks second, APGAS$_{stat}$ third, and PCJ fourth. PCJ and both APGAS variants need more lines than Spark, because storing and reducing the result has to be implemented explicitly. However, since APGAS$_{dyn}$ offers some support for this, it ranks second.

For example, Spark's Pi code in Listing 4 only needs one call of `reduce()` in line 24 to accumulate all distributed results. In contrast, the PCJ code in Listing 1 reduces the results manually (lines 48–59, excluding the output in line 58), and defines the partial results as shared variables (lines 1, 2, 7, 11, 12, 13 and 47). In the APGAS$_{stat}$ code in Listing 2, each task adds its result to the overall result on place 0 (lines 7, 21–23 and 29). In the APGAS$_{dyn}$ code in Listing 3, each tasks merges its result into the local worker result (lines 9 and 15). After all tasks have been processed, the overall result is computed by reduction on place 0 (line 18).

When developing the benchmarks, APGAS$_{dyn}$ was felt to be most productive. Its use was intuitive, simple and efficient. In particular, the locality-flexible tasks simplified the implementation, because there was no need to think about load balancing. Moreover, APGAS$_{dyn}$ provides several handy constructs for storing and reducing results, see above. Our impressions were confirmed by both metrics. Personally, we felt that the NLC metric better reflects our subjective impressions of programming difficulty and time consumption. In the subjective comparison, we ranked APGAS$_{stat}$ second,

```
1  long points = 1L << Integer.parseInt(args[0]);
2  int tasksPerWorker = Integer.parseInt(args[1]);
3  int allWorkers = localWorkers() * places().size();
4  int numTasks = allWorkers * tasksPerWorker;
5  long pointsPerTask = points / numTasks;
6
7  finishAsyncAny(() -> {
8   staticAsyncAny(() -> {
9    long tmpCount = 0;
10   for (long j = 0; j < pointsPerTask; ++j) {
11    double x = 2 * randomDouble() - 1.0;
12    double y = 2 * randomDouble() - 1.0;
13    tmpCount += (x * x + y * y <= 1) ? 1 : 0;
14   }
15   mergeAsyncAny(tmpCount, PLUSLONG);
16  }, numTasks);
17 });
18 long count = reduceAsyncAnyLong(PLUSLONG);
19 println("Pi is roughly " + 4.0 * count / points);
```

Lst. 3: APGAS$_{dyn}$: Code for Pi

```
1  long points = 1L << Integer.parseInt(args[0]);
2  int totalWorker = Integer.parseInt(args[1]);
3  int tasksPerWorker = Integer.parseInt(args[2]);
4  int totalTasks = totalWorker * tasksPerWorker;
5  long points = 1L << n;
6  long pointsPerTask = points / numTasks;
7
8  SparkConf sparkConf = new SC().setAppName("Pi");
9  JavaSparkContext jsc = new JSC(sparkConf);
10
11 List<Int> list = new ArrayList<>(totalTasks);
12 for (int i = 0; i < totalTasks; i++) list.add(i);
13
14 JavaRDD<Int> rdd = jsc.parallelize(list, totalTasks);
15
16 long count = rdd.map(integer -> {
17  long tmpCount = 0;
18  for (long i = 0; i < pointsPerTask; ++i) {
19   double x = 2 * randomDouble() - 1.0;
20   double y = 2 * randomDouble() - 1.0;
21   tmpCount += (x * x + y * y <= 1) ? 1 : 0;
22  }
23  return tmpCount;
24 }).reduce((int1, int2) -> int1 + int2);
25 println("Pi is roughly " + 4.0 * count / points);
```

Lst. 4: Spark: Code for Pi

because task distribution and result reduction had to be implemented by hand. Aside from that, APGAS$_{stat}$ was just as easy to understand and use as APGAS$_{dyn}$.

The described differences between APGAS$_{stat}$ and APGAS$_{dyn}$ become clear in Listings 2 and 3. In the APGAS$_{dyn}$ code, all tasks are spawned by a single construct in line 8. Note that the number of tasks is defined in the same call, but in line 16. In contrast, the APGAS$_{stat}$ code manually distributes the tasks evenly over all places, see lines 11–13.

Spark required more time than APGAS to get familiar with, and the algorithms had to be adapted to the MapReduce scheme. Still, the resulting source codes are short and easy-to-understand. The code in Listing 4 creates a task list in lines 11 and 12, and distributes it evenly in line 14. However, the list itself is not really needed, but only used to distribute consecutive numbers in line 14. This feels cumbersome, but is the easiest and officially recommended way.

We needed a little more training time for PCJ than for

Spark. This was related to the fact that, even for simple problems, more constructs are needed. Both the PCJ and APGAS$_{stat}$ codes explicitly take care of task distribution. However, more effort was required for that in PCJ than in APGAS, compare Listing 1 lines 28–34 and Listing 2 lines 10–13, respectively. Moreover, in PCJ, replicating values such as `tasksPerPlace` requires much programming effort, see Listing 1 lines 1, 2, 8, 11, 12, 15, 31, 32 and 33. In contrast, the other systems automatically copy final variables into lambdas for remote reading, see e. g. Listing 4 lines 6 and 18. Even after some time, we found the syntax and use of shared variables in PCJ difficult and error-prone.

Spark and both APGAS variants provide automatic intra-node work balancing, but PCJ does not. PCJ programmers may manually implement it, see Listing 1 line 40.

For testing our programs, we first installed all libraries on local workstations. These installations did not cause any trouble, although the Spark installation was by far the most complicated. When deploying the libraries on a typical HPC cluster with Slurm as workload manager, our experiences varied. Writing a submit-script for Spark programs, which starts Spark in standalone mode and sets all environment variables correctly, was quite time consuming and challenging. In contrast, both PCJ and APGAS offer launchers, which we could use without much effort.

V. Related Work

The gap between HPC and big data systems has received much attention in recent years. For example, Asaadi, Khaldi and Chapman [31] give a survey of MPI, OpenMP, OpenSHMEM, Spark and Hadoop. They discuss different system characteristics and performance. These authors conclude that a new programming model should be developed that combines the best of both worlds.

Several researchers have combined Spark with typical HPC systems. For example, Spark+MPI [1] exchanges serialized data between Spark and an existing MPI library via a shared memory file system. Since a data exchange requires several seconds, the system is only useful for long-running Spark computations.

In contrast, Alchemist (Spark $\leftrightarrow$ MPI) [8] uses sockets for the data transfer between Spark and MPI. All data must be stored twice: in a Spark RDD, and in a distributed matrix on the MPI side. Still, using Alchemist in Spark programs significantly improves the performance.

SWAT [7] combines Spark with accelerated tasks. Users can still write their Spark programs in Java, but SWAT generates OpenCL code from the JVM bytecode at runtime. The generated code is then executed on GPUs. The authors report a speedup by a factor of 3.24 on six machines.

Previous work by Bała, Nowicki et al. [20] [11] compared PCJ to Apache Hadoop. These authors report that PCJ is easier to use than Hadoop, and PCJ programs are 5 to 500 times faster. Moreover, PCJ was observed to perform better than MPI with Java bindings, but up to three times worse than MPI with C bindings.

Suter, Tardieu and Milthorpe compared the Scala version of APGAS to Akka [32], which is an actor-based concurrency library [23]. These authors conclude that APGAS and Akka are similar in both program complexity and performance.

In previous own work [33], we developed a cooperative GLB implementation in APGAS$_{stat}$. Recall that GLB [15] is the global load balancing framework, from which the APGAS$_{dyn}$ work stealing algorithm was taken. That cooperative GLB implementation outperformed the official X10 implementation of GLB (when compiled with Java) by up to 27%.

HabaneroUPC++ [34] is another asynchronous library implementation of the PGAS model. It allows direct global memory accesses, but has no support for fault tolerance and elasticity. A more detailed comparison of APGAS and HabaneroUPC++ was conducted by Scherbaum [35].

VI. Conclusions

This paper has compared the big data library Spark and the HPC libraries PCJ and APGAS. For APGAS, we included both the original version and an own extension by locality-flexible tasks. The comparison was based on Java implementations of three benchmarks, which were partly taken from HPC and the big data domain, respectively. All implementations were conducted by the same author, who had no previous experience with any of the systems. Furthermore, we took care to implement the same algorithms.

On one hand, we evaluated productivity, based on personal impressions and objective metrics. The extended APGAS variant turned out best, closely followed by the original APGAS variant and Spark. The extended APGAS variant was most intuitive to use, required the lowest number of different library constructs, and its code was by only a few lines longer than that of the Spark variant.

On the other hand, we carried out performance measurements with up to 144 workers. They showed the extended APGAS variant as a clear winner. All APGAS programs scaled well. With 144 workers, their execution time was by up to 28.88% less than that of the PCJ programs, and by up to 56.81% less than that of the Spark programs.

Overall, our results suggest that the extended APGAS library may be a good candidate for programming both HPC and big data applications with the same system.

Future work should compare the performance of APGAS$_{dyn}$ with MPI-based systems, include more benchmarks, and run them on a larger number of nodes.

Acknowledgments

This work is supported by the Deutsche Forschungsgemeinschaft, under grant FO 1035/5-1.

References

[1] M. Anderson, S. Smith, N. Sundaram *et al.*, "Bridging the gap between HPC and big data frameworks," *Proc. of the VLDB Endowment*, vol. 10, no. 8, pp. 901–912, 2017.

[2] J. L. Reyes-Ortiz, L. Oneto, and D. Anguita, "Big data analytics in the cloud: Spark on Hadoop vs MPI/OpenMP on beowulf," *Procedia Computer Science*, vol. 53, pp. 121–130, 2015.

[3] T. White, *Hadoop: The Definitive Guide*, 1st ed. O'Reilly Media, 2009.

[4] M. Zaharia, R. S. Xin, P. Wendell *et al.*, "Apache Spark: A unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, Nov. 2016.

[5] M. Nowicki, M. Ryczkowska, L. Górski *et al.*, "PCJ – a Java library for heterogenous parallel computing," in *Proc. Int. Conf. on Software Engineering, Parallel and Distributed Systems (SEPADS)*. WSEAS Press, 2016, pp. 66–72.

[6] O. Tardieu, "The APGAS library: resilient parallel and distributed programming in Java 8," in *Proc. ACM SIGPLAN Workshop on X10*, 2015.

[7] M. Grossman and V. Sarkar, "SWAT: a programmable, in-memory, distributed, high-performance computing platform," in *Proc. Int. Symp. on High-Performance Parallel and Distributed Computing (HPDC)*. ACM, 2016.

[8] A. Gittens, K. Rothauge, S. Wang *et al.*, "Alchemist: An Apache Spark MPI interface," in *Concurrency and Computation: Practice and Experience*, 2018.

[9] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, p. 107, 2008.

[10] M. Zaharia, M. Chowdhury, T. Das *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. USENIX Conference on Networked Systems Design and Implementation*, 2012.

[11] P. Bała, L. Górski, and M. Nowicki, "Performance evaluation of parallel computing and big data processing with Java and PCJ library," in *Cray User Group*, 2018.

[12] Oracle, "Class ForkJoinPool," 2018. [Online]. Available: https://docs.oracle.com/javase/10/docs/api/java/util/concurrent/ForkJoinPool.html

[13] J. Posner and C. Fohry, "A combination of intra- and inter-place work stealing for the APGAS library," in *Parallel Processing and Applied Mathematics*. Springer, 2018, pp. 234–243.

[14] ——, "Hybrid work stealing of locality-flexible and cancelable tasks for the APGAS library," *The Journal of Supercomputing*, vol. 74, no. 4, pp. 1435–1448, 2018.

[15] W. Zhang, O. Tardieu, D. Grove *et al.*, "GLB lifeline-based global load balancing library in X10," in *Proc. ACM Workshop on Parallel Programming for Analytics Applications*, 2014.

[16] S. Olivier, J. Huan, J. Liu *et al.*, "UTS: An unbalanced tree search benchmark," in *Languages and Compilers for Parallel Computing*. Springer LNCS 4382, 2006, pp. 235–250.

[17] The Apache Software Foundation, "Apache Spark," 2018. [Online]. Available: https://github.com/apache/spark

[18] ——, "Apache Mesos ," 2018. [Online]. Available: http://mesos.apache.org

[19] P. Bała and M. Nowicki, "PCJ library repository," 2018. [Online]. Available: https://github.com/hpdcj/PCJ

[20] M. Nowicki, M. Ryczkowska, L. Górski *et al.*, "Big data analytics in Java with PCJ library: Performance comparison with Hadoop," in *Parallel Processing and Applied Mathematics*. Springer, 2018, pp. 318–327.

[21] M. Szynkiewicz and M. Nowicki, "Fault-tolerance mechanisms for the Java parallel codes implemented with the PCJ library," in *Parallel Processing and Applied Mathematics*. Springer, 2018, pp. 298–307.

[22] IBM, "X10 programming language library repository," https://github.com/x10-lang/x10, 2018.

[23] P. Suter, O. Tardieu, and J. Milthorpe, "Distributed programming in Scala with APGAS," in *Proc. SIGPLAN Symp. on Scala*. ACM, 2015, pp. 13–17.

[24] Hazelcast, "The leading open source in-memory data grid," 2018. [Online]. Available: http://hazelcast.org

[25] IBM, "The APGAS library for fault-tolerant distributed programming in Java 8." [Online]. Available: https://github.com/x10-lang/apgas

[26] J. Posner, "Extended APGAS library repository," 2018. [Online]. Available: https://github.com/posnerj/PLM-APGAS

[27] University of Kassel, "Scientific data processing," 2018. [Online]. Available: https://www.uni-kassel.de/its-handbuch/en/daten-dienste/wissenschaftliche-datenverarbeitung.html

[28] L. Tolstoy, *War and Peace*, 1952.

[29] G. de Scudéry, *Artamène ou le Grand Cyrus*, 1654.

[30] Google, "Google Java style guide," 2018. [Online]. Available: https://google.github.io/styleguide/javaguide.html

[31] H. Asaadi, D. Khaldi, and B. Chapman, "A comparative survey of the HPC and big data paradigms: Analysis and experiments," in *IEEE Int. Conf. on Cluster Computing*, 2016, pp. 423–432.

[32] Lightbend, "Akka library repository," 2018. [Online]. Available: https://github.com/akka/akka

[33] J. Posner and C. Fohry, "Cooperation vs. coordination for lifeline-based global load balancing in APGAS," in *Proc. ACM SIGPLAN Workshop on X10*, 2016.

[34] R. University, "HabaneroUPC++ library repository," 2018. [Online]. Available: https://github.com/habanero-rice/habanero-upc

[35] J. Scherbaum, "Comparison of HabaneroUPC++- and APGAS library," bachelor's thesis, University of Kassel, Germany, 2016.

APPENDIX A
ARTIFACT DESCRIPTION APPENDIX: [COMPARISON OF THE HPC AND BIG DATA JAVA LIBRARIES SPARK, PCJ AND APGAS]

A. Abstract

This artifact description describes information needed to deploy the libraries on HPC clusters.

B. Description

1) Check-list:

- **Program: Java**
- **Compilation: javac 1.8 for Spark, javac 10 for PCJ and APGAS**
- **Run-time environment: Linux with Oracle Java**
- **Hardware: Any Hardware**
- **Output: Calculated results, elapsed running time**
- **Publicly available?: Yes**

2) How software can be obtained (if available): All libraries and benchmarks are available as github repository:

- Spark: https://github.com/apache/spark,
- PCJ: https://github.com/hpdcj/PCJ,
- APGAS: https://github.com/posnerj/PLM-APGAS,
- Developed Benchmarks:
  - https://github.com/posnerj/SC18-Spark-Benchmarks
  - https://github.com/posnerj/SC18-PCJ-Benchmarks
  - https://github.com/posnerj/SC18-APGAS-Benchmarks

3) Hardware dependencies: The software will run on any general purpose computer. However, experiments in this paper were conducted on a cluster with 12 homogeneous nodes [27]. Each node comprises two 6-core Intel Xeon E5-2643 v4 CPUs, and 256 GB of main memory. All nodes are interconnected with Infiniband.

4) Software dependencies: The software can be compiled and executed with Oracle Java 1.8 (Spark) and 10 (PCJ, APGAS), respectively. However, experiments in this paper were conducted with the following versions:

- CentOS 7.2,
- Slurm 17.11,
- GPFS 4.2.3-8,
- Java 1.8.0_172 for Spark,
- Java 10.0.1 for PCJ and APGAS,
- Spark 2.3.0,
- PCJ 5.0.6 in its latest available revision (May 29, 2018) [19],
- APGAS in its latest revision (August 14 2018), 2018) [26].
- Hazelcast 3.10 (only for APGAS),
- Gradle 4.7.

5) Datasets: Both novels used by WordCount are online available:

- Lev Tolstoy's *War and Peace* [28] (written in English, 3.3 MB): http://www.gutenberg.org/files/2600/2600-0.txt
- Georges des Scudéry's *Artamène ou le Grand Cyrus* [29] (written in French, 10 MB): http://www.artamene.org/telecharger.php

C. Installation and Local Execution

a) Spark:

```
git clone https://github.com/posnerj/SC18-Spark-Benchmarks
cd SC18-Spark-Benchmarks
./gradlew build
cd bin
java -Dspark.master="local[4]" -cp .:../spark/jars/\* Pi.Pi 4 20 64
```

b) PCJ:

```
git clone --recurse-submodules https://github.com/posnerj/SC18-PCJ-Benchmarks
cd SC18-PCJ-Benchmarks
./gradlew build
cd bin
echo "localhost\nlocalhost" > hostfile
java -cp .:../lib/\* Pi.Pi hostfile 20 64
```

c) APGAS:

```
git clone --recurse-submodules https://github.com/posnerj/SC18-APGAS-Benchmarks
cd SC18-APGAS-Benchmarks
./gradlew build
cd bin
java -cp .:../lib/\* -Dapgas.places=2 -Dapgas.threads=2 PiDynamic.PiDynamic 20 64
```

D. Experiment Workflow

The experiments are done by submitting the benchmarks via own Slurm scripts and the results are stored in text files.