NEWT - A Resilient BSP Framework for Iterative Algorithms on Hadoop YARN

Ilja Kromonov, Pelle Jakovits, Satish Narayana Srirama
Institute of Computer Science, University of Tartu, J. Liivi 2, Tartu, Estonia
{kromon, jakovits, srirama}@ut.ee

***Abstract*—The importance of fault tolerance for parallel computing is ever increasing. The mean time between failures (MTBF) is predicted to decrease significantly for future highly parallel systems. At the same time, the current trend to use commodity hardware to reduce the cost of clusters puts pressure on users to ensure fault tolerance of their applications. Cloud-based resources are one of the environments where the latter holds true. When it comes to embarrassingly parallel data-intensive algorithms, MapReduce has gone a long way in ensuring users can easily utilize these resources without the fear of losing work. However, this does not apply to iterative communication-intensive algorithms common in the scientific computing domain. In this work we propose a new programming model inspired by Bulk Synchronous Parallel (BSP), for creating a new fault tolerant distributed computing framework. We strive to retain the advantages that MapReduce provides, yet efficiently support a larger assortment of algorithms, such as the aforementioned iterative ones. The model adopts an approach similar to continuation passing for implementing parallel algorithms and facilitates fault tolerance inherent in the BSP program structure. Based on the model we created a distributed computing framework - NEWT, which we describe and use to validate the approach.**

***Keywords*—Bulk Synchronous Parallel, fault tolerance, cloud computing, iterative algorithms, Hadoop YARN**

I. INTRODUCTION

In recent years cloud-based platforms have emerged as alternatives to supercomputers and grids for high performance computing needs. With the illusion of infinite resources, cloud computing allows one to loan computation time on demand with a flexible pay-as-you-use billing model. However, applications are placed in an environment associated with a high risk of hardware failure. This is further amplified in private cloud setups where use of commodity equipment lessens the infrastructure cost.

For these reasons, Hadoop MapReduce [1] framework has found widespread use in the cloud-based distributed computing field. It provides fault tolerance by replicating both data and computation in an attempt to guarantee that the started applications produce a result. Originally introduced by Google in 2004 [2], MapReduce excels at solving data-heavy embarrassingly parallel problems, however, it has trouble with more sophisticated algorithms [3], as the model was simply not designed to support them. Furthermore, even MapReduce implementations that are aimed at iterative computation, such as Twister MapReduce [4], have trouble with most scientific computing problems, with one of the main reasons being that, by design, MapReduce processes are stateless. The stateless nature of a process implies that no state information is associated with the given process at any time, ensuring that any part of input data is eligible for any of the available processes without affecting the outcome. This concept ensures that failure of one of the nodes does not affect the sequential consistency of the program and is at the core of the MapReduce fault tolerance mechanism.

When MapReduce does not suffice, a common alternative is to use Message Passing Interface (MPI) - an established standard, which throughout the years has become the de facto way of writing parallel programs. While allowing for a large degree of flexibility in implementing synchronization between processes, MPI code tends to be error prone and difficult to debug and maintain. The ability to introduce various low-level optimizations comes with the danger of encountering notorious deadlocks and race conditions. As of MPI version 3, fault tolerance is still not part of the MPI standard, so any developments in this direction are left to specific implementations. Most of the time these highly specialized solutions make the programmer do extra work to ensure his application is fault tolerant, or require additional cluster infrastructure, such as dedicated checkpoint servers. In practice, the effort to maintain these implementations is lacking, and they usually end up falling behind the most recent MPI standard. For example, OpenMPI has had a transparent checkpoint/restart system as part of its implementation of MPI for some time, however, since version 1.7, it's no longer maintained and thus is not part of the most current OpenMPI packages [5], [6].

This leads us to believe that the Bulk Synchronous Parallel (BSP) model can be an ideal basis for a parallel computing framework. It can have all MapReduce advantages while providing a message passing paradigm similar to MPI without many of the issues involved. In this work we propose a BSP-inspired programming model which enables transparent stateful fault-tolerance for programs that follow this model. We used this model to develop a distributed computing framework and implemented a number of typical iterative scientific computing algorithms on it to validate the approach. This paper expands upon the ideas presented in [7] by including a comprehensive state-of-the-art survey, a more thorough and larger scale performance evaluation and checkpointing overhead comparison as well as detailed examples of adapting

978-1-4799-5313-4/14/$31.00

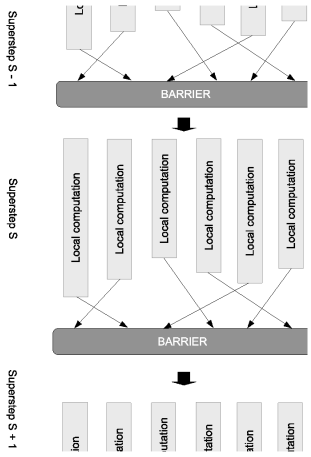

Figure 1. Illustration of the BSP model

algorithms.

Section II describes the BSP model and its advantages, followed by a description of current state of the art in frameworks based on this model in section III. Section IV describes the issues we see with currently employed approaches and our proposed solution is described in section V. Section VI provides examples of adapting algorithms to our model, followed by validating the proposal in section VII through a set of experiments. Finally, section VIII provides a summary and outlines the future work directions.

II. BSP MODEL

In 1990 [8] Valiant argued that a bridging model between software and hardware has to be introduced to properly utilize existing computing resources for parallel computation. BSP model was his proposed candidate for streamlining the move of sequential computation to the parallel infrastructure. A BSP based program consists of a series of supersteps as illustrated on figure 1, each divided into three stages:

- Concurrent computation (using only local data)
- Communication
- Barrier synchronization

One of its main advantages over MPI is the elimination of race conditions and deadlocks by avoiding circular data dependencies. The resulting program structure simplifies obtaining an overview of the implemented algorithm's granularity and estimating the expected parallel runtime and performance. While BSP may not have been widely adopted for its initial purpose, it has inspired new programming models and is the basis for several parallel programming libraries for which we give an overview in the following section.

III. STATE OF THE ART

This section provides the related work describing the existing BSP solutions in the context of fault tolerance and their suitability for long running computations. Additionally, table I provides an overview of these implementations and compares them to the most widely used solutions that follow other distributed computing models. The table also includes the proposed solution, which will be described in more detail in the following sections.

A. BSPlib

BSPlib is a specification for a programming library standard inspired by the BSP model [9]. Indeed, it was the first derivative of Valiant's efforts. Its goal was to define low-level communication and synchronization primitives that would allow the creation of parallel programs that conform to the ideas expressed in the BSP model, while keeping the API as simple and clear as possible.

Implementations which follow its specification such as PUB [10] or Oxford BSP toolset [11] are not optimized for modern architectures, furthermore, Oxford BSP toolset will not even compile out-of-the-box on 64-bit systems, which are prevalent in today's high performance computing world (for example, the largest Amazon EC2 instance types are restricted to 64-bit). A more recent implementation exists in BSPonMPI [12], however, as with MPI, the task of implementing fault-tolerance lies entirely on the shoulders of the user.

B. Pregel

Pregel [13] is a BSP-inspired parallel programming library that has since grown into a fully fledged parallel programming model. It was developed by Google engineers to address MapReduce's inability of handling iterative graph processing algorithms efficiently. In order to use Pregel for solving real-world problems, one must first express them in terms of vertices, edges and operations to be performed on them.

Fault tolerance in Pregel is achieved by storing the state of the program to persistent storage at the start of a superstep (known as checkpointing). Google's proprietary implementation of the model is now used widely in-house to solve various problems that can be represented as a graph. Pregel has sparked tremendous interest from the parallel programming community and a number of Pregel-like solutions have been developed since its unveiling. Currently, two of the most promising open-source of projects of this kind are Apache Giraph [14] and Stanford GPS [15].

Pregel, and its open source counterparts are specifically designed for graph processing applications. While it would not be impossible to adapt other types (like scientific computing) of distributed applications to these frameworks by restructuring them, it would require extra effort from the user and may decrease the parallel efficiency of the result.

C. Hama

HAMA [16] was originally envisioned as a framework that provides Hadoop-compatible interfaces to different computation engines, including MapReduce and Pregel. It has been

TABLE I
COMPARISON BETWEEN EXISTING AND THE PROPOSED APPROACH

	Progr. model	Applications	Fault tolerance	Data distribution	General comments
MPI	N/A	Any	Not part of MPI standard or its major implementations. Existing transparent solutions store entire task address space.	Explicit	Does not follow any specific parallel programming model, left up to user to implement.
BSPlib [9]	BSP	Iterative applications	Generally same as MPI.	Explicit	Messages are not received until global barrier has been completed. Difficult to get working on newer hardware.
Hadoop MapReduce [2]	MapReduce	Embarrassingly parallel, data processing	All input and output data is replicated. Restarts failed map and reduce tasks. Task state is not preserved.	Uses Hadoop Distributed File System (HDFS). Large data sets automatically split among mappers.	Programs consist of 'map' and 'reduce' functions. Speculatively executes slowest tasks on finished nodes. No real support for iterative execution. Long job configure time ($\sim$17 sec)
Pregel [13]	BSP	Graph processing	Checkpoints and recovers intermediary vertex values/message queues and continues execution from the last checkpoint.	Each process is associated with a vertex value	Program consists of a single function, which is continuously executed on each vertex. Proprietary.
Apache Giraph [14]	BSP, Pregel	Graph processing	Checkpoints and recovers intermediary vertex values/message queues and continues execution from the last checkpoint.	Uses HDFS for distributing data.	Program consists of a single function, which is continuously executed on each vertex. It extends Pregel by introducing master computations, sharded aggregators and edge-oriented input.
Stanford GPS [15]	BSP, Pregel	Graph processing	Checkpoints and recovers intermediary vertex values/message queues and continues execution from the last checkpoint.	Uses HDFS for distributing data.	Extends Pregel API with global computations instead of just vertex centric ones. Includes optimizations to repartitioning graphs and reducing network I/O.
Hama [16]	BSP	Any	N/A	Uses HDFS, data distribution similar to Hadoop MapReduce.	Manually placed barriers when synchronization is needed, similar to BSPlib
NEWT	BSP	Any	Checkpoints and recovers intermediary state/message queues and continues execution from the last checkpoint.	Uses HDFS. Creates a process for each input file. (Alternative approaches under consideration)	Programs consist of an arbitrary number of labeled functions with user-defined transitions

one of the most actively developed (with occasional spikes of activity, as well as periods of stagnation) open-source frameworks based on the BSP model, and has gone through countless iterations, dropping support for MapReduce along the way. Currently, HAMA retains the Pregel part of the API (the 'HAMA Graph' package), but also provides a different approach to writing programs under the BSP model. This alternative interface is more akin to BSPlib than Pregel, and gives the programmer much more control over how the each superstep is defined by allowing the manual specification of synchronization points and not requiring the solved problem to be represented as a graph.

Despite being in development for years, no fault tolerance mechanisms have been implemented and any future developments in that direction are hard to assess due to a lack of available up-to-date information online.

IV. ISSUES WITH CURRENT SOLUTIONS

It is clear that the existing BSP solutions either do not provide the fault tolerance required by long running applications or are designed for solving specific types of problems, such as graph processing and thus require extra effort for adapting other kind of applications to them.

In addition to the BSP implementations mentioned in the previous section, one may argue that MapReduce also follows the Bulk Synchronous Parallel model (albeit in a very restricted manner), as it is defined by two supersteps: aptly named 'map' and 'reduce', with communication from the first superstep to the next one being followed by a global barrier. The distinguishing feature is that 'mappers' and 'reducers' are spawned separately, such that the number of active processes changes from one superstep to the other. This forces all intermediate data to be written to the distributed file system, with no state being kept in memory between the two supersteps. This distinction is one of the main issues preventing MapReduce to accommodate most algorithms (most notably iterative ones). The second issue is the stateless nature of MapReduce processes, and it surfaces when attempts are made to allow the model to accommodate iterative computations.

The fault-tolerance mechanism of MapReduce depends on rescheduling processes using the input of the failed process, since that is the only state restored during fault recovery, algorithms that hold a large amount of state information, while only small portions of it are needed to be transmitted to other processes, suffer a significant overhead from being implemented using MapReduce. An example framework which achieves iterative fault-tolerant computation, using the MapReduce model, is HaLoop [17]. The aforementioned issue also applies to HaLoop, as it uses the underlying distributed file system to store the output of each iteration, which can be later used in the fault recovery procedure.

An alternative approach to transparent fault-tolerance is taken by Pregel. In Pregel, each vertex is associated with a value and a set of edges, which can change from superstep to superstep. These values are written to resilient storage at configurable intervals and can be used to restore the state of the whole computation when one or more nodes fail. Additionally, Pregel allows for messages to be transmitted between vertices and thus all incoming messages of that superstep also need

to be stored. This approach seems reasonable for many types of problems (granted it is assumed for Pregel that the state of the vector is relatively small), but the restriction of defining all supersteps of a program in a single function forces the programmer to implement algorithms as a state machine, and synchronize the current state with the master. This is a rather restrictive approach, which can get cumbersome for more complex iterative algorithms. One solution to this issue is the use of Domain Specific Languages, such as Green-Marl [18] proposed by authors of Stanford GPS. However, given the goal of Pregel, these languages are very specific to graph processing problems and thus not very suitable for general purpose programming.

To combat these issues we propose a parallel programming model (not unlike MapReduce) which enables transparent stateful fault-tolerance and has better support for general-purpose iterative algorithms than currently existing solutions.

V. PROPOSED SOLUTION

The proposed solution to the aforementioned problems with current fault-tolerant parallel programming frameworks is a Bulk Synchronous Parallel and functional programming inspired model, that would allow a framework to provide many of the advantages of MapReduce, such as fault-tolerance, yet still efficiently support a much wider range of algorithm types.

The goals of the proposed solution are the following:

- Provide automatic fault recovery.
- Retain the program state after fault recovery.
- Provide a convenient programming interface.
- Support (iterative) scientific computing applications.

Before formally defining the model, the first thing to note is that in more complex iterative programs, each iteration may consist of more than one distinct BSP superstep. To accommodate the continuation of the recovered program at the correct stage of the iteration, without storing the entire address space, it makes sense to write it as a finite state machine (FSM). In the resulting FSM each such stage is equivalent to one of the states. This leads us to view programs under the BSP model as suitable for an abstract computer, consisting of:

- Memory, containing mutable state and message queues
- Mapping of labels to instructions, where each instruction corresponds to computation done at one of the supersteps
- Function pointer or state register, holding the label of the next instruction to be executed
- Communicator - allows messages to be sent and received

The given description is similar to a counter machine (apart from a communicator), and indeed we are simply applying the same principles to a higher level of abstraction. The instructions, in this case, are user-defined functions and the message queues hold incoming and outgoing messages, to adhere to principles outlined in the BSP model regarding communication and barrier synchronization. Writing a program under this model would then be similar to using continuation-passing style, known from functional programming.

Using high-level imperative programming concepts, the following pseudo code emulates the inner workings of the described abstract machine:

$state \leftarrow initialState$
$next \leftarrow initialLabel$
while $true$ **do**
  $next \leftarrow execute(next, state, comm)$
  $barrier(comm)$
  if $next == none$ **then**
    $break$
  end if
end while

The $execute$ call runs the function defined by label $next$ and returns the label of the next function in the sequence. The mutation of state and sending/receiving of messages (through communicator $comm$) is achieved as a side-effect of these functions. There is a need for communication primitives that cover the semantics of sending and receiving messages. These primitives are made accessible through the communicator. The $barrier$ initiates communication and synchronizes all machines as per the BSP model.

The given generic program structure allows for the state, label of the next stage and incoming message queue to be stored into persistent storage (such as a distributed file system) between invocations of $execute$, for later recovery in case of machine failure. This recovery is then seamlessly achieved by using the stored data on the rescheduled task, instead of the initial one, and, in the same fashion, read from the checkpoint by the remaining processes, when they are notified of failure elsewhere in the network by a coordinator, which has to detect the failure state in the first place. It has to be noted, that the processes that did not fail do not need to be restarted and can complete the recovery by simply replacing the current state with an earlier one and then continue the execution.

A program under this model has to define the state and a mapping of labels to functions, which describe the program flow. The return value of each of these functions is explicitly the label of the next function to be executed. It is possible to use this model to generalize any MapReduce program, since such programs can be represented with stages labeled 'map' and 'reduce' with a null state, sending messages at the end of 'map' to machines associated with specific keys, with these messages becoming available for processing at the 'reduce' stage. However, it is not restricted to problems that can be summarized in only two stages. For instance, one can model iterative programs with an arbitrary number of iterations, by having a function return its own label as long as more iterations need to be computed, and since the tasks remain in memory and are not rescheduled for each stage, the overhead is minimal. The frequency of checkpointing can then be configured by the user, based on the estimated running time and the stability of the underlying cluster hardware.

To validate this approach, we created a proof-of-concept prototype implementing the given model. The following subsection gives an overview of the used technologies.

A. Technology Overview

Successful fault-tolerance of processes depends on a scheduling mechanism and a resilient storage environment, where checkpoints can be stored in a reliable manner and retrieved in case of machine or network failures. There are existing solutions that can be used for this purpose. To take advantage of the ongoing development of Apache Hadoop we chose the following established and continuously supported software components:

- YARN (Yet Another Resource Negotiator) - separates the Hadoop framework components, allowing for models other than MapReduce to utilize cluster resources. [19]
- HDFS (Hadoop Distributed File System) - provides a highly fault-tolerant distributed file system. Designed to be run on commodity hardware. [20]

In context of the chosen technologies, a coordinator for the framework is implemented as a YARN ApplicationMaster and the main program loop, as described in the previous section, runs inside YARN containers[a]. The final requirement is message passing, which is initially implemented using Apache MINA [21]. The reason of not using MPI directly is that while there are (at the time of writing this article) developments in making MPI a first-class citizen on Hadoop YARN cluster [22], there are no working implementations yet. However, we expect it to be available in the near future.

B. API

The user mainly interacts with the framework through two interfaces - *BSPState* and *Stage<BSPState>*. The job is configured by providing a state class, which is an extension of *BSPState* and contains all program state that will need to be stored in checkpoints. The program flow is defined by providing instances of *Stage<BSPState>* (and their labels) that implement it's *execute* method. The signature of the method is:

- *String* execute(*BSPComm* comm, *BSPState* state)

In the actual implementation *BSPState* is replaced by it's extension as provided by the generic parameter of *Stage* (which should be the user's state class).

The *BSPComm* class exposes message passing functionality to the user. The currently available functions for sending messages are:

- send(*Writable* message, *int* pid)
- send(*Writable* message, *int* pid, *String* tag)
- sendAll(*Writable* message)
- sendAll(*Writable* message, *String* tag)

[a]Constrained environments for launching user's applications

Messages are subclasses of the Hadoop *Writable* interface. Message passing is race-free and messages can be safely retreived from the receive queue in the same order they were sent in. The following functions enable the user to do so:

- move()
- move(*int* pid)
- getReducedValue(*String* tag, *ReduceOp<Writable>* op)

The *move* function retrieves the received messages from all processes (sorted by process ID in ascending order) and in case *pid* is specified it retrieves a message from the process corresponding to that process ID. The *getReducedValue* method can make use of the optional *tag* parameter that can be specified when sending messages. It mimics the functionality of the *MPI_Reduce* collective operation by applying the *reduce* method of *ReduceOp* to a sequence of messages (those that were tagged with the specified string) and outputs a single message. Common reduction operations such as sum-of-floats or maximum-of-integers are provided by the framework.

The program functions are defined as Java 1.7 closures, the goal is to move to the more concise Java 1.8 closure syntax in the future.

VI. EXAMPLES

To illustrate the usability of this approach we implemented two scientific computing applications - conjugate gradient method (CG) [23] and k-medoids clustering algorithms as examples of how one would adapt algorithms to this model. In previous work we have used these particular algorithms to compare a number of BSP and MPI implementations [24] to each other and this enables us to directly compare the efficiency of the created prototype to previous results.

The algorithm descriptions detail the structural adaptations necessary to implement them with NEWT. For how these implementations might look in actual code refer to the simple pi estimator NEWT code in Figure 2.

A. Conjugate Gradient

CG is a rather complex iterative algorithm for solving sparse systems of linear equations and is outlined in figure 3. Each algorithm iteration needs to be split into several supersteps due to requirements of synchronizing a global dot product on two occasions and synchronizing overlapping portions of vectors between neighbors for matrix-vector multiplication. The algorithm is implemented in the typical 'single program multiple data' (SPMD) fashion, with state of size N being split among p processes as evenly as possible. The segregation into stages is represented in figure 5 and is as follows:

- Init - initializes all needed state variables and checks how close the initial guess for x is to the actual solution
- Start of Loop - defines the beginning of the loop (containing operations up until the first dot product), to serve as a reference point in flow control, at the end messages

```
public static void main(String[] args) throws IOException {
    Configuration conf = NEWTConfiguration.createDefault();
    final int NUM_PROCS = 8;
    Input input = new NullInput(NUM_PROCS); //no input needed
    conf.set("newt.output.dir", "/out");
    NEWTJobMaster<NullState> am =  //no state to keep track of
            new NEWTJobMaster<NullState>(conf, NullState.class, input);
    am.addStage("map", new Stage<NullState>() {
        @Override public String execute(BSPComm bsp, NullState state) {
            double sum = 0.0; int samples = 1000;
            Random random = new Random();
            for (int i = 0; i < samples; i++)
                sum += Math.sqrt(1 - Math.pow(random.nextDouble(),2));
            bsp.send(0, new DoubleWritable(sum/samples), "pi/4");
            return "reduce";
        }
    });
    am.addStage("reduce", new Stage<NullState>() {
        @Override public String execute(BSPComm bsp, NullState state) {
            if (bsp.pid() == 0) {
                DoubleWritable result =
                        bsp.getReducedValue("pi/4", Reducer.DOUBLE_SUM);
                result.set(4*result.get()/NUM_PROCS);
                addToOutput("result", result, bsp); //write output to HDFS
            }
            return Stage.STAGE_END;
        }
    });
    am.run();
}
```

Figure 2. Simple pi estimator implemented with NEWT

Input: b, A and initial guess for x
Output: better approximation for x
$r \leftarrow b - Ax$
$err \leftarrow error(r)$
while $err < treshold$ and $iter < max$ **do**
  $z \leftarrow r$
  $\sigma_{old} \leftarrow \sigma$
  $\sigma \leftarrow z \cdot r$
  $p \leftarrow z + \frac{\sigma}{\sigma_{old}} p$
  $q \leftarrow Ap$
  $\gamma \leftarrow \frac{u}{p \cdot q}$
  $x \leftarrow x + \gamma p$
  $r \leftarrow r - \gamma q$
  $err \leftarrow error(r)$
end while
return x

Figure 3. Conjugate gradient method of approximating solution to $Ax = b$.

need to be sent to all processes, containing the partial dot product and error value

- Check Ending Condition - completes the computation of dot product and current error value, the latter being used to decide whether the algorithm should be finishing or the main loop should continue, returning the label of one of the two possible stages
- Continue Loop - prepares the vector p for the subsequent matrix-vector multiplication, sending the required overlapping portions of this vector to neighboring processes
- Do MatVec - receives the overlapping vector pieces and completes matrix-vector multiplication, then computes another partial dot product, sending the result to all processes
- End of Loop - completes the computation of dot product from the received partial ones and computes the new error value, then returns the label of the 'Start of Loop' stage
- Stop - the stage that is executed when the error gets below the required margin or the maximum iteration count is exceeded in the 'Check Ending Condition' stage, completes the computation and writes the output to disk

```
public static class BeginLoop extends Stage<CGState> {
    @Override
    public String execute(BSPComm bsp, CGState state) {
        //calculations
        bsp.sendAll(new DoubleWritable(state.u), "sigma");
        bsp.sendAll(new DoubleWritable(state.error), "error");
        return "checkCondition";
    }
}

public static class CheckCondition extends Stage<CGState> {
    @Override
    public String execute(BSPComm bsp, CGState state) {
        state.u = bsp.getReducedValue("sigma", Reducer.DOUBLE_SUM).get();
        state.error = bsp.getReducedValue("error", Reducer.DOUBLE_MAX).get();
        if (state.error > TOLERANCE && state.it <= MAX_IT) {
            //calculations
            return "doMatVec";
        } else {
            //calculations
            return Stage.STAGE_END;
        }
    }
}
```

Figure 4. Synchronization of CG state in the NEWT program

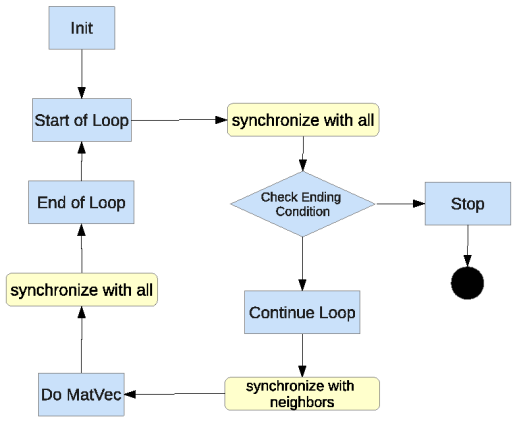

Figure 5. Structure of CG under the proposed model

The structure is similar to programs written under common message passing paradigms (such as BSPlib or MPI), with the distinction that the code is split into several functions, akin to a MapReduce program. figure 4 shows an excerpt from the implementation of CG, showing only the synchronization of state between stages.

B. PAM k-medoids Clustering

Partitioning Around Medoids (PAM) is an iterative clustering method that divides a set of observations (2D points in our case) into k clusters based on the pairwise distances between them. The algorithm consists of the following steps:

- Randomly select k objects as the starting medoids
- Associate each object to the closest medoid, forming k different clusters

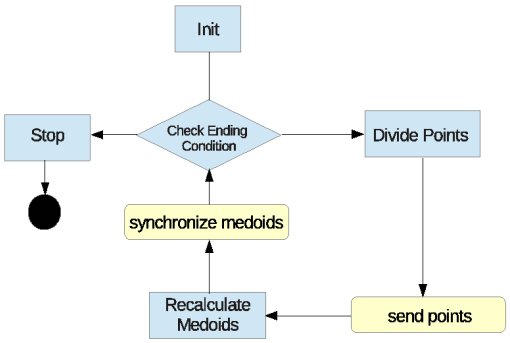

Figure 6. Structure of PAM under the proposed model

- For each cluster, recalculate its medoid m:
  - For each object o in the cluster with medoid m, swap m and o and compute the cost of the cluster
  - Select the object o with the lowest cost as the new medoid for this cluster
- Stop once medoids no longer change

This procedure of dividing objects between clusters and recalculating the medoids is repeated until the medoids no longer change positions between the clusters, at which point the clustering has become stable and is concluded. In the parallel implementation each process handles k/p clusters, where p is the number of parallel processes, this means the medoids need to be synchronized after they are recalculated and each process has to send points that are closer to medoids it does not handle to the corresponding process.

The segregation into stages of the algorithm is rather straightforward and follows the algorithm's description very closely, as is displayed in figure 6:

- Init - initializes all needed state variables and randomly selects the initial medoids in each cluster
- Check Ending Condition - calculates the difference between the previous iteration's medoids and the current one, if the medoids did not change then move to 'Stop', otherwise proceed to 'Divide Points'
- Divide Points - finds to which medoid each point is closest to, and transmits points that belong to clusters handled by other processes
- Recalculate Medoids - finds new medoids in each cluster and sends the new medoids to each other process
- Stop - the stage that is executed when medoids are determined to no longer change in the 'Check Ending Condition' stage, completes the computation and writes the output to disk

These examples demonstrate when the serial programs need to be split into multiple functions. The most obvious one is the synchronization requirement, such as the computation of the global dot product after the 'Do MatVec' stage for CG. The second case stems from branching statements, such as the 'Check Ending Condition' stage, which can continue execution in two different directions. The last case in the given example is seen in the CG 'Start of Loop' stage, which serves as a reference point for code, which should be executed repeatedly, serving as a base for a loop structure in this case.

One simple optimization is to combine stages that are part of a loop, but do not require any communication between them, as the repeated intermediary barriers would cause unnecessary overhead. For example, the PAM 'Check Ending Condition' and 'Divide Points' would be merged in this fashion.

VII. EXPERIMENTS

To validate our approach we have run a series of trials on the prototype using the implementations of algorithms described in the previous section. We compared the prototype to parallel versions of these algorithms that used BSPonMPI - a BSPlib implementation that uses MPI for communication, which we determined in previous work [24] to perform as good as MPI for the given algorithms. The test cluster composed of 16 Amazon's m3.xlarge (Standard Extra Large) instances, each with 15 GB of memory and 8 EC2 Compute Units (4 virtual cores), running Ubuntu Server 12.04 with Hadoop YARN 2.2.0 installed. We conducted two types of trials: one to measure scalability and the other to assess overhead from creating state checkpoints and recovery in case of failure.

A. Measuring scalability

In the scalability trials, each of the algorithms was given input of size that was kept constant (a sparse system of 125000000 linear equations for CG and 250000 points across 64 clusters for PAM). Only the number of processes p was increased for consecutive trials.

TABLE II
RUNTIME COMPARISON BETWEEN NEWT AND BSPONMPI (SECONDS).

	conjugate gradient			k-medoids clustering	
p	NEWT	BSPonMPI	p	NEWT	BSPonMPI
1	4476	4616	1	1889	1873
2	2225	2415	2	1248	1172
4	1245	1221	4	646	601
8	697	689	8	339	330
16	350	346	16	203	185
32	227	219	32	150	153
64	240	207	64	122	151

The results in table II for NEWT include a $\sim$14 second overhead that is induced by YARN for initialization and allocation of process containers.

The scaling of the coarse-grained parallel algorithm (PAM) is slightly better than the BSPonMPI implementation when looking at the 4-core and higher results. It suggest that structuring the algorithm according to the model does not impose a significant overhead.

In the case of the fine-grained CG algorithm the scaling of NEWT is also better initially (2-16 cores) but starts to decline afterwards (32 and 64 cores). This is because the

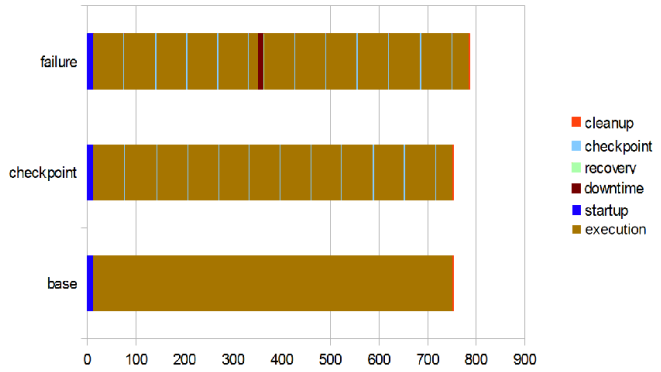

Figure 7. PAM performance after enabling checkpointing and on node failure

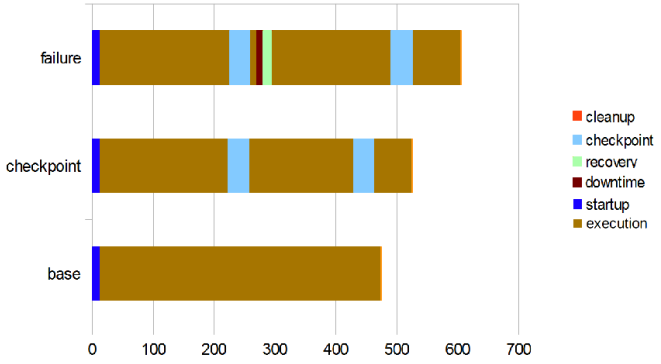

Figure 8. CG performance after enabling checkpointing and on node failure

communication part of the runtime starts to significantly outweigh the computation part, resulting in a slightly worse result than BSPonMPI, which leaves room for optimizing the implementation of the barrier synchronization.

The sequential version (on 1 node) of CG structured according to the NEWT's model consistently outperformed the sequential MPI implementations, the concrete cause of this is under investigation. We hypothesize that it is related to the JVM being able to optimize the code better when certain algorithms are structured according to NEWT's model.

B. Measuring overhead

To measure the overhead imposed by storing checkpoints in HDFS we enabled periodic checkpointing. The figures 7 and 8 show timelines for three kinds of executions of the implemented algorithms:

- without checkpointing
- with checkpointing enabled
- with checkpointing and recovery after a failure

To simulate failure in the cluster, one of the Amazon nodes was shutdown around halfway of the program's runtime. An additional node was added to the cluster, such that recovered processes may be started on it when one of the nodes goes down. The algorithms were executed on 16 Amazon m1.medium instances, each with 3.75 GB of memory and 1 virtual core.

From figure 7 it can be seen that the addition of checkpoints (every minute) for PAM had a negligible effect on performance, since the state kept by the algorithm is very small in relation to the amount of computation. The checkpoints consisted of floating point coordinates for the points belonging to clusters that were handled by any given process and any incoming messages, totaling under a megabyte on average (much less than the entire address space of the program). The time it took to write such checkpoints to HDFS was approximately 30 milliseconds. When node failure occurred in the cluster, there was a period of downtime of around 10 seconds, which included the time it took for YARN to recognize the failure and allocate a replacement container. Following the period of downtime, there was a brief session of recovery, which consisted of reading the state of every process from the checkpoints and the reestablishment of socket connections between them, this also took under a second.

When the CG algorithm was distributed among 16 processes (Figure 8), it required to store over 400 megabytes of data to HDFS during checkpointing by each process. Due to the size of checkpoints, the frequency of their creation was reduced to three and a half minutes. Storing these checkpoints took over 30 seconds and reading them from HDFS in case of failure took approximately half the time. Table III shows average times for the different types of overhead the framework imposes for the two tested algorithms.

C. Comparison to OpenMPI Transparent Checkpoint/Restart

We attempted to apply the OpenMPI 1.5 transparent checkpoint/restart (CR) mechanism to the BSPonMPI program. The memory snapshots of the conjugate gradient algorithm done by BLCR 0.8.5 [25] were around 470 MB in size, due to all of the program's memory pages being checkpointed, including those containing state which was not required to restart the NEWT processes, such as the immuTable sparse matrix (generated on initialization). The average time to write these checkpoints to local storage was the same (∼35 seconds) as time required for creating NEWT checkpoints in HDFS. When checkpoints were written directly to a mounted network file system (NFS) share (located on an additional m1.large Amazon instance) the time to finalize checkpoint creation exceeded 450 seconds. The PAM checkpoints created by BLCR were around 20 MB in size - significantly larger than NEWT checkpoints (for the same reason as the CG ones). The average time to write these to local storage was 3.3 seconds and directly to the mounted NFS share around 14.5 seconds, both significantly longer than then 0.1 second interruption to create NEWT checkpoints in HDFS. These results tell us that in comparison to setting up a Hadoop cluster, significant time and resource investment is needed for creating a high throughput redundant storage system that makes the classic checkpoint/restart utilities work efficiently, such as setting up a IBM GPFS [26] cluster

instead of a single NFS server. This also shows that using Hadoop allows NEWT to efficiently make use of commodity infrastructure without much overhead. Due to the unsTable nature of OpenMPI CR on the test infrastructure and the fact that continuation of processes needs to be performed manually, the restart functionality was not compared.

TABLE III
AVERAGE OVERHEAD TIMES (SECONDS) IMPOSED BY NEWT

	CG	PAM
startup	12	12
checkpoint	34.75	0.1
downtime	11	12
recovery	15	0.3
cleanup	2	2

VIII. CONCLUSION AND FUTURE WORK

We summarized the state of the current BSP-based distributed computing solutions and identified several problems with their approaches. To counter these drawbacks we presented a BSP-inspired parallel programming model that enables transparent stateful fault tolerance through checkpointing. To validate the usefulness of the proposed model, we created a distributed computed framework, called NEWT.

NEWT supports a larger range of applications than the current BSP implementations and utilizes Hadoop YARN to perform automatic checkpoint/restart of programs. We implemented two very different computation kernels on the framework and showed that it performs adequately for coarse-grained algorithms. However, our results also show that the current barrier synchronization implementation could still be optimized for better support of very fine-grained algorithms. We compared the current NEWT implementation's checkpointing time requirements to BLCR's and determined that writing NEWT checkpoints to HDFS is as fast as writing BLCR checkpoints to local storage.

Future work also includes identifying common communication patterns and providing directives for executing them through the API, as well as providing an implicit way of handling input and output data. More advanced topics include implementing optional intelligent checkpointing strategies, such as hierarchical checkpoint/restart. The framework is already open-source and available online [27].

ACKNOWLEDGMENT

This work is supported by European Regional Development Fund through EXCS, AWS in Education Grant, IT Academy, Estonian Science Foundation grant PUT360 and Target Funding SF0180008s12.

REFERENCES

[1] Apache Software Foundation. (2014, March) Apache hadoop mapreduce. [Online]. Available: http://hadoop.apache.org/docs/current/

[2] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[3] S. N. Srirama, P. Jakovits, and E. Vainikko, "Adapting scientific computing problems to clouds using mapreduce," *Future Gener. Comput. Syst.*, vol. 28, no. 1, pp. 184–192, Jan. 2012.

[4] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative mapreduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10, 2010.

[5] Indiana University. (2014, March) Openmpi fault tolerance. [Online]. Available: http://www.open-mpi.org/faq/?category=ft

[6] Open Systems Laboratory. (2014, March) Ompi transparent checkpoint/restart. [Online]. Available: http://osl.iu.edu/research/ft/ompi-cr/

[7] I. Kromonov, P. Jakovits, and S. N. Srirama, "Newt - a fault tolerant bsp framework on hadoop yarn," in *Proceedings of the 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing*, ser. UCC '13, 2013, pp. 309–310.

[8] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990.

[9] J. M. D. Hill, B. Mccoll, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling, "Bsplib: The bsp programming library," 1998.

[10] O. Bonorden, B. Juurlink, I. V. Otte, and I. Rieping, "The paderborn university bsp (pub) library - design, implementation and performance," in *In Proc. of 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP*. Society Press, 1999, pp. 99–104.

[11] J. Hill. (2014, March) The oxford bsp toolset. [Online]. Available: http://www. bsp-worldwide.org/implmnts/oxtool/

[12] W. J. Suijlen. (2014, March) Bsponmpi. [Online]. Available: http://bsponmpi.sourceforge.net/

[13] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, ser. SIGMOD '10, 2010.

[14] Apache Software Foundation. (2014, March) Apache giraph. [Online]. Available: http://incubator.apache.org/giraph/

[15] S. Salihoglu and J. Widom, "Gps: A graph processing system," in *Scientific and Statistical Database Management*. Stanford InfoLab, July 2013. [Online]. Available: http://ilpubs.stanford.edu:8090/1039/

[16] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng, "Hama: An efficient matrix computation with the mapreduce framework," in *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, ser. CLOUDCOM '10, 2010, pp. 721–726.

[17] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: efficient iterative data processing on large clusters," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 285–296, Sep. 2010.

[18] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, "Green-marl: a dsl for easy and efficient graph analysis," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII, 2012.

[19] Apache Software Foundation. (2014, March) Apache hadoop yarn. [Online]. Available: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

[20] ——. (2014, March) Apache hadoop hdfs. [Online]. Available: http://hadoop.apache.org/docs/r2.0.5-alpha/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

[21] ——. (2014, March) Apache mina. [Online]. Available: http://mina.apache.org/

[22] (2014, March) Hamster: Hadoop and mpi on the same cluster. [Online]. Available: https://issues.apache.org/jira/browse/MAPREDUCE-2911

[23] J. R. Shewchuk, "An introduction to the conjugate gradient method without the agonizing pain," Tech. Rep., 1994.

[24] P. Jakovits, S. Srirama, and I. Kromonov, "Viability of the bulk synchronous parallel model for science on cloud," in *High Performance Computing & Simulation, 2013. HPCS '13. International Conference on*, 2013, (In print).

[25] Berkeley Future Technologies Group. (2014, march) Berkeley lab checkpoint/restart (blcr). [Online]. Available: http://crd.lbl.gov/groups-depts/ftg/projects/current-projects/BLCR/

[26] IBM Corporation. (2014, march) General parallel file system. [Online]. Available: http://www-03.ibm.com/systems/software/gpfs/

[27] (2013, Nov.) NEWT bitbucket repository. [Online]. Available: https://bitbucket.org/mobilecloudlab/newt