RABIT: A Reliable Allreduce and Broadcast Interface

Tianqi Chen      Ignacio Cano      Tianyi Zhou

Department of Computer Science & Engineering
University of Washington

Abstract

Allreduce is an abstraction commonly used for solving machine learning problems. It is an operation where every node starts with a local value and ends up with an aggregate global result. MPI provides an Allreduce implementation. Though it has been widely adopted, it is somewhat limited; it lacks fault tolerance and cannot run easily on existing systems.

In this work, we propose RABIT\(^1\), an Allreduce library suitable for distributed machine learning algorithms that overcomes the aforementioned drawbacks; it is fault-tolerant and can easily run on top of existing systems. We compare RABIT with existing solutions and show that it performs competitively.

1 Introduction

Given the increase of data on the web, it becomes necessary to distribute both data and computation among different nodes in order to do efficient processing.

Several abstractions have been proposed. MapReduce [2] has been widely adopted. Though is good for solving data parallel problems, it is not ideal for many machine learning tasks. Compared to data processing programs, machine learning programs usually consume more resources, they often demand allocation of temporal results and caching. Since a machine learning process is generally iterative, it is desirable to make it persistent across iterations, and perform the resource allocation just once. MapReduce abstraction does not provide such property. Figure 1 shows how an iterative MapReduce computation looks like, there is a new process for every iteration, and re-allocation of resources is required.

An abstraction that overcomes such limitations is Allreduce. Allreduce avoids the unnecessary map phases, reallocation of memory and disk reads-writes between iterations, by allowing programmers to easily maintain an in-memory representation of the model, which is replicated to every node. Figure 2 illustrates the typical workflow of an iterative computation using Allreduce, where a single map phase takes place, followed by one or more Allreduce stages. The program persists across iterations, and it re-uses the resources allocated previously. It is therefore a more convenient abstraction for solving many large scale machine learning problems.

Despite Allreduce’s advantages, traditional implementations such as OpenMPI, are non fault-tolerant, i.e. if one node fails in the middle of a computation, its portion of the data and current state are lost. Further, programs that use MPI implementations cannot be easily used with other platforms [3]. This kind of integrations are highly desirable; the final system would be able to take advantage of both worlds. For example, by combining an Allreduce implementation with Mapreduce, the system can rely on the latter to move data and restart nodes, while the former

\(^1\)https://github.com/tqchen/rabit
In this work, we propose RABIT, a C++ Allreduce fault-tolerant library, suitable for solving large scale machine learning problems, that can be easily combined with existing platforms.

### 2 Allreduce

In Allreduce settings, nodes are organized in a tree structure. Each node holds a portion of the data and computes some values on it. Those values are passed up the tree and aggregated, until a global aggregate value is calculated in the root node (reduce). The global value is then passed down to all other nodes (broadcast).

Figure 3 shows an example of an Allreduce sum operation. The leaf nodes pass data to their parents (interior nodes). Such interior nodes compute an intermediate aggregate and pass the value to the root, which in turn computes the final aggregate and then passes back the result to every node in the cluster.

Several machine learning problems fit into this abstraction, where every node works on some portion of the data, computes some local statistics (e.g., local gradients) and then invokes the Allreduce primitive to get a global aggregated result (e.g., global gradient) in order to further proceed with the computations.

### 3 RABIT

We propose RABIT, a reliable Allreduce library that can handle node failures and resume computation from virtual checkpoints. The following list is not exhaustive, but attempts to capture what we think are RABIT’s key contributions:

- **Portability:** Rabit is a portable library instead of a framework, a program only needs to link the library to run. This enables effortless integrations with third-party systems.
- **Flexibility in programming:** Programs can call Rabbit functions in any order, as opposed to frameworks where callbacks are offered and called at specific points in time.
- **Programs persistence:** Programs continue running over all iterations, instead of being restarted on each one of them.
- **Fault tolerance:** Rabbit programs can recover their state (e.g., machine learning model) using synchronous function calls.
- **MPI compatible:** Code that uses Rabbit API also compiles with existing MPI compilers, i.e., users can use MPI Allreduce with no code modification.

#### 3.1 Fault tolerance

RABIT’s key design principle is fault tolerance. Implementing a fault-tolerant version of Allreduce is not a trivial task, there is no much prior work in the area. The closest example to our work is proposed by Agarwal et al. [1], in which they introduce a tree-shape communication infrastructure that efficiently accumulates and broadcasts values to every node involved in a computation. Nevertheless, their implementation of Allreduce is not fault-tolerant during program execution.

RABIT leaves the restart of failed nodes to external systems. We provide fault tolerance based on two main protocols, a consensus and a routing protocol. In order to recover a failed node, 5 steps need to be performed:

1. Pause every node until the failed node is fully recovered.
2. Detect the model version we need to recover by obtaining the minimum operation number. This is done using the Consensus Protocol explained in section 3.2.2.
3. Transfer the model to the failed node using the Routing Protocol explained in section 3.2.2.
4. Resume the execution of the failed node using the received model.
5. Resume the execution of the other nodes as soon as the failed node catches up.
Figure 4: Fault Tolerance

Figure 4 illustrates the 5 steps. It further shows virtual checkpoints, which are points in time where it is safe to remove previous models and where operations numbers are increased. They are virtual because they do not write models to disk, they simply act as synchronization barriers for the group of nodes.

3.2 Recovery

We build our recovery module based on two primitives using non-blocking I/O. The two primitives are Allreduce and Message Passing. Based on these primitives, we propose a Consensus and a Routing protocol to do recovery.

3.2.1 Primitives

- **Allreduce**: as mentioned in section 2, Allreduce is commonly used in a tree-shaped structure, and it allows different reduce operations, such as max, min, sum, etc.

  Figure 5 shows an example of an Allreduce min operation. The values are passed in a bottom-up approach. The green node on the left subtree receives \( \{1,1\} \) from its children. It has a 0 value, so it computes the minimum between \( \{1,1,0\} \), and forwards the result to its parent. The same process occurs on the right side of the tree. The root finally decides 0 is the min number and broadcasts the result back to every node. At this point, every node knows that 0 is the minimum value.

- **Message Passing**: nodes send messages to their neighbors to find out information about them. It can be used in many algorithms such as Shortest Path.

3.2.2 Protocols

- **Consensus Protocol**: The consensus protocol agrees on the model version to recover using an Allreduce min operation. In the example shown in Figure 5, the value of each node is the model version they store. The green node on the left subtree is recovering so it wants to find which model it needs in order to make progress. It does so with the Allreduce min operation. After the value 0 is broadcast-ed to every node, everyone knows that model 0 is the version to be used.
for recovery.

- **Routing:** Once the model version to recover is agreed among the nodes, the routing protocol executes. It finds the shortest path a failed node needs to use in order to retrieve the model. It uses two rounds of message passing. The first round computes the distance from a failed node to the nearest node that has the model. The second round sends requests through the shortest path until it reaches the node that owns the model, and retrieves it.

In the example shown in Figure 6, the green node on the left subtree is the one asking for the model. After the graph is traversed, the node knows the closest node where it can get the model from. Once this first round ends, the second round of message passing occurs, in which the green node ends up requesting the model to its right child (not shown in Figure 6).

### 3.3 Tracker

Though RABIT is inherently a P2P system, where all messages in the Data Plane are sent among peer nodes, the startup of the system and the reconnection of failed nodes is done by a small centralized process called the tracker.

On startup, every worker connects to the tracker in order to find its child and parent links. When a node fails and restarts, it also asks the tracker for its link information.

### 3.4 Toolkits

We intend to provide several machine learning algorithms out of the box as part of an ML Toolkit. At the time of this writing, the current implementation of RABIT (version 0.1) only contains the popular KMeans algorithm.

The pseudo code below shows a sample RABIT program. For further details refer to the project repository on GitHub.

```plaintext
Load (data)
gModel, lModel := nil
if (!LoadCheckpoint(gModel, lModel))
  Init(gModel, lModel)
for i to iterations:
  DoComputations(gModel, lModel)
  MoreComputations(lModel)
  Broadcast(lModel)
  Allreduce(gModel)
  Checkpoint(gModel, lModel)
  Save(gModel, lModel)
```

The program first loads the input data and checks for the latest checkpoint-ed models, both local and global. In case there are no models, we simply initialize them. In case the node failed in the middle of a computation and is restarting, the load of the checkpoint triggers the recovery protocols described in section 3.2.2, in order to either recover a previous checkpoint-ed model from other node or recover the result of lagged-behind Allreduce calls.

 Afterwards, the program performs some local computations to update the models. It can call Broadcast and Allreduce functions. It is worth mentioning that the library allows the programmer to perform as many Allreduce and Broadcast calls as the program logic requires, in this example we just show one call for each function. After Allreduce returns, a program can update its local model using the global state received and then perform a checkpoint. A checkpoint means the program finished an execution stage, therefore the model version is increased. Though the API supports checkpointing global and local models, the local model checkpoint is not currently implemented. The successful completion of the Checkpoint function guarantees the global model is the same in every node. This process is repeated for the number of iterations the user specifies or until convergence. Finally, the models can be flushed to disk.

### 4 Experiments

We compare RABIT against OpenMPI and Spark on a 32-node EC2 cluster.

We test RABIT and the state-of-art non fault-tolerant Allreduce implementation OpenMPI on different operations, using various array sizes (from 10K to 10M). In order to reduce the variance, we average the results computed over \( \frac{1003}{\text{len(array)}} \) repetitions, i.e. we perform 10 runs for an array size of 10M, 100 runs for an array size of 1M, and so on.

Figure 7 shows the results. We test the Broadcast function, and the Allreduce one using sum and max operations. The results show that our throughput is around two times slower than the highly optimized OpenMPI implementation, which we consider a strong evidence of RABIT’s effectiveness. OpenMPI has been around since 2004 and its current version is 1.8.3, while RABIT is still in its early stages.

Figure 8 shows the results of comparing our KMeans implementation against Spark MLlib on a BOW Wikipedia dataset of \( \approx 2\times 10^6 \) examples, with a vocabulary of \( \approx 500\times 10^3 \) words. Our KMeans implementation was triggered using Hadoop Streaming. With \( k = 5 \), RABIT is 6 times faster than Spark. On higher k’s, MLlib runs
out of memory as it uses a dense representation for centroids. We believe these promising results further demonstrate the huge potential of the library.

5 Conclusion

With the exponential increase of data on the web, it becomes critical to build systems that can process information efficiently in order to extract value out of it. Several abstractions have been proposed to address these requirements. In this project, we focus on the Allreduce abstraction, suitable for solving large scale machine learning problems. We propose an efficient and fault tolerant version that can be used together with existing big data analytics systems. We compare our solution with existing systems and the results evidence that RABIT is on the right track.

Acknowledgments

This work was supported by TerraSwarm, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the sponsors.

References

