# Hadoop YARN in CSP Interpretation

Hanfei Lin, East China Normal University, China
Email:10122510319@ecnu.edu.cn

***Abstract*—YARN MapReduce system is the engine for the current popular Apache Hadoop©. It is "a framework for job scheduling and cluster resource management" [1] and its concurrent complexity deserves further studies. This paper adopts CSP language to model real YARN system and PAT software to verify properties of the model. Issues like Deadlock, Non-termination and Divergence-free are includedV. This paper serves as an attempt to apply process algebra on the current MapReduce frameworks and will trigger more works in this direction.**

***Index Terms*—Hadoop, YARN, Process Algebra, CSP, PAT**

## I. Introduction

YARN system, abbreviation of *Yet Another Resource Negotiator*, is the second generation of MapReduce engine for Apache Hadoop©. MapReduce is a programming model that processes and generates large data sets with a parallel, distribute algorithm on a cluster[3]. It is remarkable to fill the Moore's Law not longer only with the hardware manufacture evolution, but more and more depend on the software design optimism, especially the concurrent and distributive system architecture, since the slow down of the hardware technology improvements. Meanwhile the industry have been demanding high performance process on bigger data than ever before, which is the main driving force for technology like Hadoop.

The spirit of Hadoop can be dated back to 2004, when Google first published the paper introducing their MapReduce idea to the world. Later in February 2006, researchers in Google separated *Nutch Distributive File System*(NDFS) and *MapReduce* from Nutch and founded the project named Hadoop. Apache foundation then took over the support and in April 2008 Hadoop set up the world record to become the fastest $TB$ level data sorting system. From then on, Hadoop earned its reputation as the mainstream in enterprise deployment system[2].

Process algebra has long been brought to tackle with concurrent systems in the development of computer science. And sure it should be adopted to facilitate the infrastructure of large-scale computation clusters and their software framework. However relative studies that apply process algebra in studies to MapReduce systems are still in lack. To meet this demand, this paper make an attempt to apply *Communicating Sequential Processes*(CSP) language, one of the dialects in process algebra, to model the YARN system. The model is also verified by *Process Analysis Toolkit*(PAT).

Structure of this paper follows: The Background section II introduces the basic architecture of the real YARN system and also includes notations to CSP language and PAT software. Fundamental assumptions to the model are made in the section Model III. A series of simplifications are clarified which shapes the YARN system discussed in this paper. The PAT programming are explained with details in section Implementation IV. The corresponding verification results are presented in section Verification V. Finally, conclusions and expectations to the future works are drawn in section VI.

## II. Background

### A. Overview of YARN

The real YARN system is shown in Fig.1, which consists Nodes of ResourceManager and NodeManagers. As a convention, the word "Node" is reduced and names of ResourceManager and NodeManager simply refer to the corresponding computation node. ApplicationMaster and tasks are running on NodeManagers. This figure displays states of applications in YARN system. A MapReduce process can be divided into map task and reduce task and they can run on different computation nodes. As shown, MR Application Master and Map task run on the leftest node, while reduce task the second node and other tasks the rest. ApplicationMaster require resources from ResourceManager through heartbeat mechanism. The ResourceManager then allocates resources through scheduler.

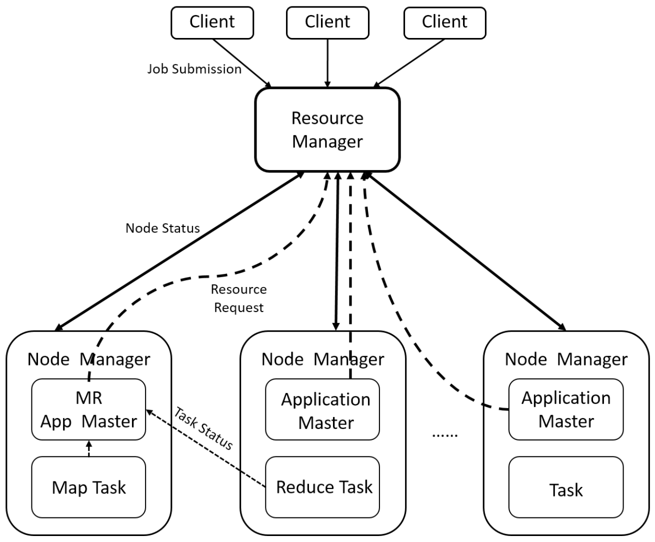


Fig. 1: Structure for Real YARN System

The workflow of a Hadoop job on YARN system mainly consists the following six steps[4]:

1) Client node submits a task to the YARN system.

2) The task is initialized by obtaining a container and being assigned an ApplicationMaster.
3) The ApplicationMaster requires resources from ResourceManager for all the map and reduce jobs of the task.
4) The task starts to execute.
5) The status of the execution is reported to ApplicationMaster, who also send heartbeat message to ResourceManager.
6) ApplicationMaster finish the execution of the task and send message to ResourceManager who release the corresponding resource for other use.

### B. CSP Model and PAT

CSP is a formal language for describing patterns of interaction in concurrent systems[5]. The YARN system was modeled using CSP. The relative notations are listed below:

P and Q are processes, a is event, e is the collection of events, E is tunnel[6].

- $a \rightarrow P$: a then P
- $P||Q$: P in parallel with Q
- $P\Box Q$: P choice Q
- $P \sqcap Q$: P or Q (non-deterministic)
- $P|||Q$: P interleave Q
- $P;Q$: P followed by Q
- $P\backslash e$: P without e (hiding)
- $E!a$: on channel E output value of a
- $E?a$: on channel E input to a

PAT project was first initialized in School of Computing, National University of Singapore in July 2007[7]. It provides a build-in CSP module implementation, which offering features of simulation and visualization. PAT will generate the results to assertions as long as users provide them in the coding. These could be done by the inner algorithms of PAT that check all possible states of the model. For the sake of convenience and reliability, verification in this paper was produced by PAT.

## III. Model

### A. Model Overview

Model to the whole system appears to be the concurrency of the Client Node and YARN resource distribution system. Illustration to the model is displayed in Fig.2. The Formula to describe such model turns out as following:

$$\begin{aligned} Model =_{df} (&YARNSystemConfig; \\ &(Client||YARNSystem)\backslash client) \end{aligned} \quad (1)$$

The YarnSytemConfig process serves to configure the system at the beginning of simulation. The configurations include setting status of each ApplicationMaster to OFF, initiating required resource for each application, allocating resource to each Compute Node and assigning expected Compute Nodes to each ApplicationMaster.Further more, client means the set of events in Client process.

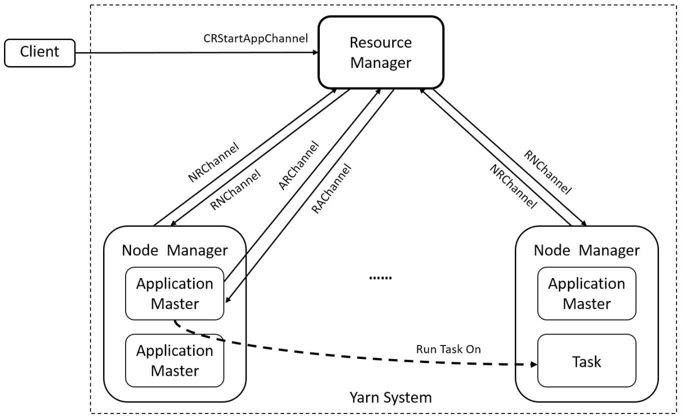


Fig. 2: Structure for YARN System Model

To simplify modelling process, it is assumed that all resources are unified to only one format, memory, actually the same as the YARN system in the real situation. The real YARN system takes the resource unit as Container and each Container packaged all required resource to run a task. In this model, this hierarchy is out of discussion, thus the real Container representation can be simplified to a resource unit. Therefore this treatment only concerns with resources directly related to the execution of the Application.

Client process models the Client Nodes, which in fact is not part of the YARN resource distribution system and it's actions should be hidden. However, client nodes plays a significant role in the whole system by communicating with ResourceManager and sending requests such as starting or closing certain Application. In this model, all specific behaviors for each Client Node are abstracted as a process which randomly send requests to ResourceManager. In order to verify deadlock possibility, the model also assumes Clients will not actively kill an Application. An application will only be terminated by obtaining all its necessary resource and execute successfully. Client process can send request of starting an ApplicationMaster to ResourceManager through the channel of CRStartAppChannel. Parameter i indicates the expected node by this ApplicationMaster. Each ApplicationMaster has a unique ID that is represented by the other parameter j. Formal expression related to Client follows:

$$\begin{aligned} Client =_{df} (&\Box_{i\in 0...NodeNumber-1} \\ &(\Box_{j\in 0...AppNumber-1} \\ &StartApp_{i,j})) \end{aligned} \quad (2)$$

YarnSystem process imitates the YARN resource distribution system, which is the concurrency of a ResourceManager, several NodeManagers and ApplicationMasters. In real YARN system, each ApplicationMaster will also manage serval tasks which is abridged as it's a Two-Tier Allocaition. But this model only discuss the resource allocation to the level of ApplicationMaster. In Yarn System, each node has a NodeManager that manages global resource allocation. Each job runs on the node belongs to an ApplicationMaster that manage the job's life cycle. The jobs are submissions from the Client to the ApplicationMaster. The ResourceManager and the NodeManager together form the data-computation framework [8]. Notation i and j represents correspondingly the i-th NodeManager and j-th ApplicationMaster. Formal expression for YarnSystem follows:

$$\begin{aligned} YARNSystem =_{df}\; & (ResourceManager|| \\ & (|||_{i\in 0\ldots NodeNumber-1} NodeManager_i)|| \\ & (|||_{j\in 0\ldots AppNumber-1} AppMaster_j)) \end{aligned} \tag{3}$$

Further more, the following six channels are introduced in the model:

- CRStartAppChannel: channel from Client to YarnSyterm carrying message to start an Application
- RNChannel[NodeID]: channel from ResourceManager to NodeManager carrying message to start, pause or request resource from a compute node
- NRChannel[NodeID]: channel from NodeManager to ResourceManager carrying heartbeat that report the current available resource
- RAChannel[AppID]: channel from ResourceManager to ApplicationMaster carrying message to start or pause latter
- ARChannel[AppID]: channel from ApplicationMaster to ResourceManager carrying heartbeat that request or gain resource
- ScheduleBuffer: buffer for ResourceManager to perform FIFO allocations

## B. ResourceManager

ResourceManager is the master and schedule center of the whole YARN cluster which manages the global allocation of compute resources to application. A ResourceManager consists of a Scheduler and an ApplicationMaster. The Scheduler is responsible for resource allocation, which no longer concerns about application states compared with the previous ResourceManagerV1.0. As a substitute, the ApplicationMaster take charge of this. ApplicationsManager can communicate with each ApplicationMaster to monitor states for different applications. When an application submits a logout message, the ApplicationMaster will release corresponding resources.

Since current focuses are on level of the whole YARN cluster, there's no need to separate ResourceManager into Scheduler and ApplicationMaster in this model. ResourceManager will receive messages from client which contain requests to start applications. Meanwhile, ResourceManager listens to heartbeat message from all ApplicationMaster. For simplicity sake, crashes of Nodes are neglected. Therefore, it's no need to simulate the behavior of the periodic heartbeat from NodeManager and in this model ResourceManager can monitor each node status at any moment. Also ResourceManager is responsible for scheduling function. Such ResourceManager can be described with the following equation:

$$\begin{aligned} ResourceManager =_{df}\; & \\ & (GetStartCommandFromClient|| \\ & (\sqcap_{i\in 0\ldots NodeNumber-1} \\ & GetHeartBeatFromAppMaster_i)|| \\ & Schedule; \\ & ResourceManager) \end{aligned} \tag{4}$$

The following are processes from above and corresponding formulas:

- GetStartCommandFromClient process fetches necessary information for starting an application from CRStartAppChannel. It then sends this to the corresponding node through RNChannel. Later the node will allocate a container to ApplicationMaster.

$$\begin{aligned} GetStartCommandFromClient =\; & \\ & CRCStartAppChannel?startmsg \\ & \rightarrow ((RNChannel_m!n \rightarrow Skip) \\ & \lhd (NRAvailable[m] > 0\&\& \\ & AppMasterStatus[n] == OFF) \rhd Skip) \end{aligned} \tag{5}$$

- GetHeartBeatFromAppMaster process handles heartbeat information from each ApplicationMaster. Due to the adoption of FIFO allocation strategy, a buffer to store resources received by GetHeartBeatFromAppMaster is necessary, which is introduced as the channel of ScheduleBuffer.

$$\begin{aligned} GetHeartBeatFromAppMaster_i =\; & \\ & ARChannel_i?heartbear \\ & !i.r.h \rightarrow Skip \end{aligned} \tag{6}$$

- Schedule process responsible for resources scheduling while ScheduleBuffer receives three input: i for the requiring application ID, r for the requiring amount, h for the demanding node. As soon as the current schedule finishes, ResourceManager will release the resource-allocation-complete message to the corresponding ApplicationMaster.

$$\begin{aligned} Schedule =\; & ScheduleBuffer?i.r.h \rightarrow schedule \\ & \rightarrow RAChannel_i!scheduleack \rightarrow Skip \end{aligned} \tag{7}$$

Whenever NodeManager receive the switch-on message of an ApplicationMaster from a RNChannel, it will run this ApplicationMaster by initializing it and allocating the resources. Parameter i stands for the ID of the NodeManager and n for ID of the ApplicationMaster.

## C. NodeManager

NodeManager is an agent framework that installed to each machine in real situation which is responsible for launching applications' containers, monitoring resource usage (CPU, memory, disk, network) and reporting to the Scheduler.[9] The model assumed all Nodes can start working from the beginning and are free of crashes, therefore, that it reduce the descriptions of log-in and log-off for each Node. Once a NodeManager receive message to support a ApplicationMaster, it will allocate one Container resource and run this ApplicationMaster. The following is the formal expression for the ApplicationMaster:

$$\begin{aligned} NodeManager_i &= RNChannel_i?n \\ &\to run_{i,n} \to NodeManager_i \end{aligned} \tag{8}$$

When each NodeManager receive the message to start an ApplicationMaster, it will execute the action "run" by initializing the ApplicationMaster and allocating corresponding resources. Parameter i represents the ID of the NodeManager while n the ID of ApplicationMaster.

### D. ApplicationMaster

An ApplicationMaster coordinates applications and manages their states. Applications are assigned to their ApplicationMaster. Once an application is submitted by client, the ResourceManager will ask the corresponding Node to run an ApplicationMaster, and allocate a container for it. Periodic heartbeats are sent to ResourceManager during the execution of ApplicationMaster, which contains information to required resource amount, expected Node and application priority. ResourceManager receives these heartbeats and allocates resources. YARN system adopt asynchronous resource distribution mechanism, in other words, which ResourceManager will temporarily stores resources in a buffer after its allocations, instead of directly sending resources to applications. Also even if a Node can not afford the whole resources for an application, ResourceManager will still allocate the available partial to this application, rather than waiting for collecting enough resources. For the sake of simplification, all applications are assumed to have the same priority and resources are allocated under FIFO strategy. Formal expression for ApplicationMaster follows:

$$\begin{aligned} AppMaster_j =_{df} & (((InitAppMaster_j \\ & \lhd (AppHoldResource_j >= ResourceForEachApp) \rhd \\ & UpdateNeededResource_j) \\ & \lhd (AppMasterStatus_j == ON) \rhd Skip); \\ & AppMaster_j) \end{aligned} \tag{9}$$

If an ApplicationMaster has gained all required resources, i.e. AppHoldResource is greater than ResourceForEachApp, then it is assumed this Application has finished. Otherwise it will wait for the complete resource, which is represented by UpdateNeededResource action.

## IV. Implementation

### A. Parameters

PAT implementation of the previous model requires assumptions of the input parameters. This came as a trade-off between the real situation description and computation limitation. If better computation had been provided, more desirable results were within reach. The following is the definition part of the implementation code.

```
#define NodeNumber 2;
#define AppNumber 2;
#define ResourceOnEachNode 3;
#define ResourceForEachApp 2;
#define ResourceForStartApp 1;
#define OFF 0;
#define ON 1;
#define FIFOBuffer 30;
```

The YARN system has 2 available Nodes and totally 2 applications will run on the system. ResourceOnEachNode stands for the resources on each node which is 3 containers. ResourceForEachApp is the amount of resources required by each application to run which is 2. ResourceForStartNode is the resource to switch on each ApplicationMaster which is 1. ON and OFF represent the states for ApplicationMaster. FIFOBuffer is for the volume of FIFO allocation strategy. Also, tables used to store information during the simulation are listed as following:

```
AppResourceNode[AppNumber];
AppResource[AppNumber];
NRAvailable[NodeNumber];
ResourceForApp[AppNumber];
NodeStatus[NodeNumber];
AppMasterStatus[AppNumber];
AppOnNode[AppNumber];
AppHoldResource[AppNumber];
```

### B. Processes

The CSP model to YARN system includes two processes, the first one is Schedule. The corresponding codes follows:

```
Schedule() = ScheduleBuffer?i.r.h
-> schedule{
  if ( NRAvailable[h] >= r )
  {
    NRAvailable[h] = NRAvailable[h]-r;
    ResourceForApp[i] = r;
  }
  else
  {
     ResourceForApp[i] = NRAvailable[h];
     NRAvailable[h] = 0;
  }
} -> RAChannel[i]!1 -> Skip();
```

Schedule process simulates the allocations by ResourceManager, which read information of i(the ApplicationMaster ID), r(the required resources amount) and h(the Node ID). Schedule process first compares the available resource(NRAvailable[h]) to the amount of requirements(r). If the former is greater than latter, it will allocate the resource to the ApplicationMaster[i] in the way that first store the resources into the buffer and let the ApplicationMaster to fetch at the next heartbeat message. On the other situation, it will allocate all resources on this Node to the ApplicationMaster the same way as above.

The AppMaster is the other process where parameter j stands for the ApplicationMaster ID. If the state of ApplicationMaster j(AppMasterStatus[j]) is ON, then it will periodically send heartbeat through ARChannel meanwhile fetch resources from ResourceForApp[j]. Once ApplicationMaster j has gained all the required resource and finished it execution, it will release all the occupied resources. Otherwise, it will always stay in the resource pool and collects resources.

```
AppMaster(j) = atomic{
[AppMasterStatus[j] == ON ](ARChannel[j]
  ! AppResource[j].AppResourceNode[j]
  -> apprun.j{
  AppHoldResource[j] =
    ResourceForApp[j] + AppHoldResource[j];
  ResourceForApp[j] = 0;
  if ( AppHoldResource[j]
    >= ResourceForEachApp )
  {
    var node = AppOnNode[j];
    NRAvailable[node] =
      NRAvailable[node] + 1;
    var nodee = AppResourceNode[j];
    NRAvailable[nodee] =
      ResourceForEachApp - 1;
    AppMasterStatus[j] = OFF;
    AppResourceNode[j] = j;
    AppOnNode[j] = -1;
    ResourceForApp[j] = 0;
    AppResource[j] = ResourceForEachApp;
    AppHoldResource[j] = 0;
  }
else
  {
    AppResource[j] =
    ResourceForEachApp - AppHoldResource[j];
  }
} -> AppMaster(j))};
```

## V. VERIFICATION

Assertion were used in the PAT program and the result was shown in Fig.3.

### A. Deadlock

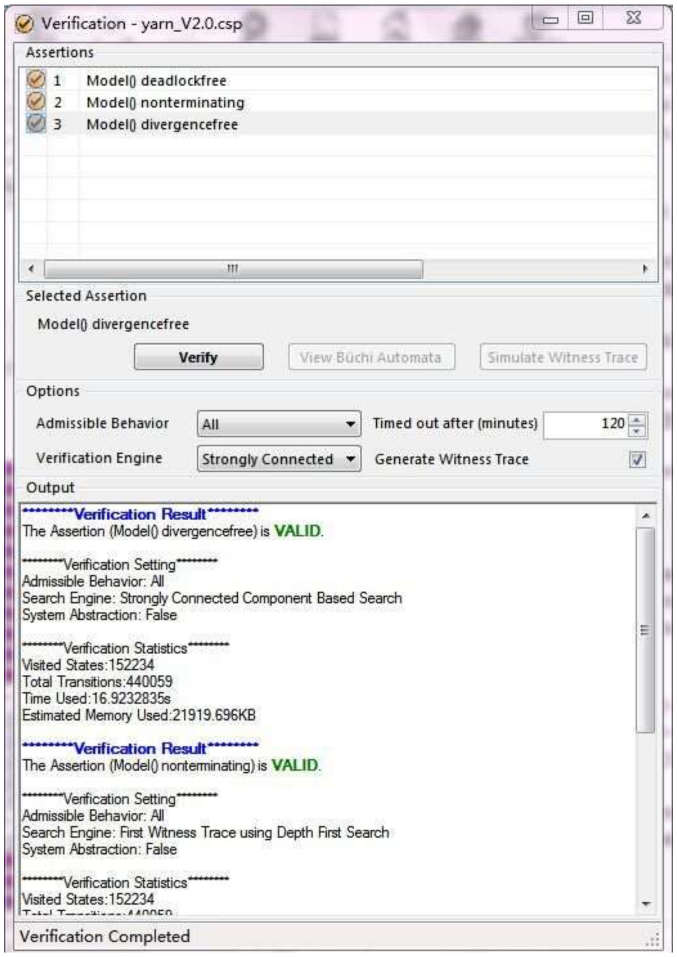
According to simulation, the YARN model is free of deadlock.

### B. Non-termination

According to simulation, the YARN model does not have terminating state.

### C. Divergence-free

According to simulation, the YARN model does not perform internal transitions forever without engaging any useful events. In other words, it's divergence-free.


Verification - yarn_V2.0.csp

Assertions

| | | |
|---|---|---|
| ✓ | 1 | Model() deadlockfree |
| ✓ | 2 | Model() nonterminating |
| ✓ | 3 | Model() divergencefree |

Selected Assertion

Model() divergencefree

Options

Admissible Behavior: All

Timed out after (minutes): 120

Verification Engine: Strongly Connected

Generate Witness Trace ✓

Output

```
********Verification Result********
The Assertion (Model() divergencefree) is VALID.

********Verification Setting********
Admissible Behavior: All
Search Engine: Strongly Connected Component Based Search
System Abstraction: False

********Verification Statistics********
Visited States:152234
Total Transitions:440059
Time Used:16.9232835s
Estimated Memory Used:21919.696KB

********Verification Result********
The Assertion (Model() nonterminating) is VALID.

********Verification Setting********
Admissible Behavior: All
Search Engine: First Witness Trace using Depth First Search
System Abstraction: False

********Verification Statistics********
Visited States:152234
```

Verification Completed

Fig. 3: Snapshot for PAT software

## VI. CONCLUSION AND FUTURE WORKS

This paper verify the Deadlock, Nontermination and Divergence properties for Hadoop YARN MapReduce system by simulating *resource allocation progress* and *heartbeat mechanism* while simplifying the other features. FIFO allocation strategy serves as the YARN system model scheduler and incremental placement, ApplicationMaster periodic heartbeat mechanism also discussed in this paper. The model is constructed using CSP formalism and results are based on the simulation of PAT software.

Attempts were made to increase Nodes and ApplicationMasters but were aborted due to the memory shortage. One explanation would be the defects in the real YARN system. The YARN system applies *pessimistic locking strategy* in the resource allocation: Scheduler (in ResourceManager) will only release resources from an Application when it finishes and returns. This can be viewed as a global lock and simulation will have to maintain the hungry Nodes(those who hold their resource and require for more in order to finish), therefore consumes large amount of memory when the Nodes and ApplicationMasters increase.

The current simplifications in the code implementation can be properly eased in the future which points out some further works like modifying the allocation strategy and simulations with better computation environment. As below, five possible directions are listed out for discussions:

1) To discuss the Two-Tier situation. The current study only consider the first tier allocation by ResourceManager to ApplicationMaster, while the second tier from ApplicationMaster to each job may as well be discussed.
2) To assume different allocation strategies. The real YARN system includes not only FIFO, but Fair Schedule and Capacity Schedule Strategy as well[10] that can be a topic in the future.
3) To specify resource types. In the future, YARN system will also introduce CPU as resources aside the current only memory. The corresponding effect can be studied using CSP and PAT.
4) To include the NodeManager heartbeat. It is assumed no crash in this paper, which omits the possibilities of machine crashes in the real situation. Therefore the accident capacity of YARN system based on the heartbeat from NodeManagers and its concurrency features can be further explored. Another interesting mechanism is the blacklist maintain by ApplicationMaster. If tasks keep failing on one Node, this Node will be add to the blacklist. The criterion to judge the Nodes worth lots of discussion.
5) To add comparisons to other resource allocation framework. Borg© (Google), Mecos© (Twitter) and Corona© (Facebook) are similar resource allocation framework and could be studied in the same way. It will be interesting to look into the architecture, accident capacity, allocation ability among them.

## REFERENCES

[1] Apache. What Is Apache Hadoop?[EB/OL]. http://hadoop.apache.org/, 2011-02-14

[2] Tom W. Hadoop:The Definite Guide, Sec.1.4[EB/OL]. 2012

[3] Wikip. MapReduce[EB/OL]. https://en.wikipedia.org/wiki/MapReduce, 2015-08-18

[4] Tom W. Hadoop:The Definite Guide, Sec.6.1[EB/OL]. 2012

[5] Wikip. Communicating Sequential Processes[EB/OL]. https://en.wikipedia.org/wiki/Communicating$_sequential_processes$, 2015-08-10

[6] Hoare.C. Communicating Sequential Processes, Sec.1.4[EB/OL]. 2004

[7] PAT3.5. Process Analysis Toolkit 3.5 User Manual[EB/OL]. http://pat.comp.nus.edu.sg/wp-source/resources/OnlineHelp/htm/index.htm, 2012

[8] Arunc. The Next Generation of Apache Hadoop MapReduce[EB/OL]. https://developer.yahoo.com/blogs/hadoop/next-generation-apache-hadoop-mapreduce-3061.html, 2011-02-14

[9] Apache. Apache Hadoop NextGen MapReduce (YARN)[EB/OL]. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html, 2015-06-09

[10] Tom W. Hadoop:The Definite Guide, Sec.6.3[EB/OL]. 2012