Estimation of Job Execution Time in MapReduce Framework over GPU clusters

Yang Hung, Sheng-Tzong Cheng
Computer science and Information Engineering
National Cheng Kung University
Tainan, Taiwan
e-mail: stcheng@mail.ncku.edu.tw

Chia-Mei Chen
Information Management
National Sun Yat-Sen University
Kaohsuing, Taiwan
e-mail: cmchen@mail.nsysu.edu.tw

***Abstract*—The development of Graphic Processing Unit (GPU) makes it possible to put hundreds of cores in one processor. It starts a new direction of high-speed computing. In addition, in a Cloud computing environment, MapReduce over GPU clusters can execute graphics processing or general-purpose applications even much faster. It is crucial to manage the resources and parameters on GPU devices. In this paper, we study the execution time of MapReduce tasks over GPU clusters. We use Stochastic Petri Net to analyze the influence of GPU computing and develop SPN-GC model. The model defines formulas of every stage's execution time and estimates the execution time under different input data size. Our experimental result presents the comparison between the estimated execution time and actual values under different input data size. The error range is found out to be within 10%. This paper can be a useful reference when a developer is tuning the program.**

Keywords-GPU; MapReduce; Stochastic Petri Net; Estimation of execution time.

I. Introduction

As data is growing at an incredible speed, it is not easy to handle a huge amount of data to make timely decisions. The scale and variety of big data now challenges traditional computing paradigms. That is why Cloud Computing could come to help.

MapReduce is being considered as a programming model for large-scale parallel processing and an associated implementation for processing and generating large data sets. The benefits of this model include efficient resource utilization, improved performance, and ease-of-use via automatic resource scheduling, allocation, and data management. In addition, Graphic Processing Unit (GPU) was originally designed for graphics processing. But now, because of its parallel computing ability, GPU is more popular for scientific and engineering application.

Many algorithms and applications can be speed up by MapReduce, with the power of GPU computing [5]. To develop MapReduce over GPU clusters [6], programmers have to give thought to some performance-related issues. When running a MapReduce job, programmers cannot obtain any information about how the performance of jobs will be under their testing environment. In the past, programs were tuned by running a series of configurations based of past experiences and then waiting for jobs to complete for several times. If jobs' performance results can be estimated, programmers will be able to shorten their working hours by finding out the program's behavior in advance under a particular hardware specification and node configuration.

In Section II, Stochastic Petri Net [2] for MapReduce on GPU clusters model (SPN-GC) is developed to describe the detailed operations of MapReduce framework over GPU clusters [7]. SPN-GC estimates the execution time of MapReduce jobs with given parameters and returns the estimated results to programmers as a reference. In Section III, we validate the SPN-GC by running experiments. In Section IV, we conclude that programmers can use the proposed model to estimate and tune the programs in less time and with better performance.

II. SPN-GC: Modeling MapReduce over GPU clusters

SPN-GC is divided into nine phases. Each phase demonstrates a specific operation in the MapReduce framework and is represented by the directed arcs along with the transitions from places to next places. Figure 4 illustrates a SPN-GC with three map functions and two reduce functions.

1) Load user program/Split input: When user starts an application, a job is assigned to Hadoop jobtracker and initialized. Jobtracker of the master node copies user program to each tasktracker on worker nodes. According to the user program and information, the jobtracker breaks input data into splits.

2) Read input: Every worker node runs its own tasktracker as a task manager. Tasktracker first reads input splits as input data of map function. Hadoop Disctributed File System (HDFS) always distributes input data over all nodes. In general, the map function will process the part of input splits that is stored on local disks for optimization. If the input split is on a remote server, a network transmission is initialized and the received data are stored, either temporarily in memory or on the disk if the data are too large, until the map function is completed.

3) Map function M/Read into GPU device: Map function of the user application takes over Compute Unified Device Architecture (CUDA) kernel function is initialized because CPU instructs the process to GPU, and data are copied from main memory to GPU device memory. Let the number of map tasks to be M. Note that map functions might not complete at the same time because every work node has its own processing speed and different delay time.

4) GPU computing: CUDA performs data high-speed parallel computing on GPU at this phase. Every stream processor is assigned to a thread. By Nvidia, 32 threads is called a warp. Threads read and write data from shared memory on GPU device at the same speed of cache.

5) GPU device to host: After CUDA finishes the work, data are copied from GPU device to host memory for the next process.

6) Sort/ Spill: The map function finishes an input pair and has to deal with the outputs from the map function. The outputs are called intermediate files in the MapReduce process.

7) Transmit/ Shuffle: After the first map task is finished, nodes of map function start to transmit intermediate files to nodes of reduce functions either locally or to remote nodes. Intermediate files are processed and sorted into the final output file for the use of the reduce tasks later. The final output file is separated into R partitions by a collector, and each partition is transmitted to the corresponding worker node that handles reduce tasks.

8) Sort/ Merge: All the data for the reduce function are pre-processing. Partitions are collected and sorted as in the map phase. After partitions are downloaded, sorted, and merged concurrently, a temporary file is prepared as the input data for reduce function.

9) Reduce function R: The reduce function is defined by the application requirements. According to the number of reduce tasks set by the user, R reduce functions should be executed in parallel, although all reduce functions may not start and end together because of varying processing speeds. All phases finish when the reduce tasks are completed.

A. Model Formulation

Let M to be the number of actual map tasks that is determined by the split size in Hadoop, and R to be the number of reduce tasks that can be configured directly in user program. The SPN-GC model can be defined as a marked stochastic Petri Net that is a 6-tuple (P, T, I, O, M_0, L), where $P = \{p_1, p_2, \dots, p_{n_p}\}$ is a finite set of n_p places, and

$$n_p = 5 * M + 3 * R + 2. \tag{1}$$

$T = \{t_1, t_2, \dots, t_{n_t}\}$ is a finite set of n_t transitions, where

$$n_t = 5 * M + 2 * R + 2. \tag{2}$$

$I \subseteq \{ P \times T \}$ is a set of input arcs (flow relation), $O \subseteq \{ T \times P \}$ is a set of output arcs (flow relation), and $M_0 = \{m_1, m_2, \dots, m_{n_p}\}$ is the set of initial markings where the generic entry m_i is the number of tokens in place p_i, $\mathrm{L} = \{\lambda_1, \lambda_2, \dots, \lambda_{n_t}\}$ is an array of firing rates where λ_j is the firing rate associated with each transition t_j. In SPN, each transition is associated with a random variable with exponential distribution that indicates the delay from the enabling to the firing of the transition.

There are many selections for firing rate which produces the elapsed time at each stage. In this paper, delay on timed transitions takes exponential distribution to describe the occurrence of events as a Poisson process. The exponential probability density function is defined in (3), where λ is the rate parameter of the distribution,

$$f_X(x) = \lambda e^{-\lambda x},\ x \geq 0. \tag{3}$$

The mean or the expected value of an exponentially distributed random variable X for a timed transition is given by (4), and can also be represented as the mean delay time in the set T on each timed transition, T_i .

$$E[X] = 1/\lambda. \tag{4}$$

After computing the mean delay time of each transition, the inverse can be obtained as a set of firing rates $L = \{\lambda_1, \lambda_2, \dots, \lambda_{n_t}\}$ where $\lambda_i = 1/T_i$, $\mathrm{i} = 1,2, \dots, n_t$, and the random time delay can be generated following an exponential distribution.

A big difference between MapReduce application and CUDA application is in the splitting of the input data. Programmers can select how to cut their input data in different sizes or in different ways depending on purpose for the input data. In Hadoop, programmers can set the block size by configuration. When the job is initialed, input of every task will fit the block size based on configuration. Different block size may produce unequal map task stages and result in various performance. It is similar in CUDA that programmers also have to adjust the input size of threads for their specific purpose or algorithm.

B. Model Analysis

The number of map tasks, *M*, is related to the number of CPU cores on each server which runs tasktracker. According to the configuration of Hadoop, the number of map tasks, *M*, is split by the size of blocks, as (5) shows below.

$$M = \lceil \text{Input data size } (D_{input})/\text{split size}(D_{split}) \rceil. \tag{5}$$

Data in HDFS is stored on a data node in which a tasktracker resides. Depending on data locations, the speed of accessing data locally or to a remote data node might be different. In our work, a random variable takes into account the ratio of the number of replications to the number of data nodes. For each map task *j*, a random rate of map input on local disk, A_j, is defined as below.

$$A_j = \frac{No.of\ replications}{No.of\ data\ nodes}, \text{ where } j = 1,2, \dots, M;\ 0 \leq A_j \leq 1.$$

In order to estimate the total execution time for a MapReduce job over GPU clusters, we need to derive the execution time spent in each phase. Starting from phase 1 till phase 9, the estimated execution in each phase is studied.

Phase 1 is for loading program and splitting input data. Therefore, T_{phase1}, the estimated execution time of phase 1, is derived in (6). After loading a program, SPN-GC estimates the time of data uploading to HDFS and splitting into blocks.

$$T_{Phase1} = T_{load} + T_{upload}. \quad (6)$$

Phase 2 is for reading input from local disk and remote disk via network. Map-worker node download split from the input split locations. Every split data must be read from disk of HDFS and transmitted to map-worker node and then written into host memory. Therefore, T_{Phase2} is the maximal downloading time of all the nodes that do the map tasks.

$$T_{Phase2} = \max_{1\le j\le m} \{\frac{Dsplit}{Disk\_r}(1-A_j) + \frac{Dsplit}{Dnetwork}(1-A_j) + \frac{Dsplit}{Disk\_w}(1-A_j)\} \quad (7)$$

Phase 3 is about Mapping to GPU. Tasktrackers start map functions and copy input data into GPU devices. The execution time of map function can be estimated by the rate of data split size, CPU capability, and the time to copy data into GPU memory.

$$T_{Phase3} = \frac{Dsplit}{Disk\_r} + \frac{Dsplit}{Mem\_r} + \frac{CPU_{test}}{CPU_i} * T_m + \frac{Dgpu\_block}{GPU\_Mem\_w} \quad (8)$$

Phase 4 accounts for GPU memory read and GPU computing. Input split data is being read into GPU device cache from GPU device memory. The execution time of GPU computing can be estimated by the rate of GPU block size and GPU capability.

$$T_{Phase4} = \max_{1\le i\le n}(\frac{Dgpu\_block}{GPUmem\_r} + \frac{D_{gpu\_block}}{D_{test}} * \frac{GPUtest}{GPU_i} * T_{m\_GPU}),$$
$$\text{where } n = D_{split} / D_{gpu\_block}. \quad (9)$$

Phase 5 is for GPU device to host. After GPU computing, output data is copied from GPU device memory to Host memory. Key-value pair is ready to sort.

$$T_{Phase5} = \frac{Dsplit}{GPU\_Mem\_r} + \frac{Dsplit}{Disk\_w} + \frac{Dsplit}{Mem\_w}. \quad (10)$$

Phase 6 is Spill/ Merge. Spills generated either by the metadata buffer or by the sort buffer could reach a specific limit that can be evaluated. The metadata size is 16 bytes per key-value record, while $D_{metadata}$ is the metadata size for all records in the map tasks, which can be evaluated as $16 * D_{input}/(D_{test} * M) * Map_{spiller\_records}$ bytes. In addition, $D_{metabuffer}$ is equal to (sort.mb * sort.record.percent * sort.spill.percent).

D_{data} is all key-value data size of the map task. D_{data} is equal to $(D_{input}/D_{test})) * (D_{map}/M) - D_{metadata}$.

$D_{databuffer}$ is equal to [sort.mb*(1-sort.record.percent)* sort.spill.percent]. While, D_{spill} is equal to $(D_{metadata} + D_{data})$, meaning that all data sizes must be processed in this phase.

No. of spill$(= N_{spill}) = \max(\left\lceil\frac{D_{metadata}}{D_{metabuffer}}\right\rceil, \left\lceil\frac{D_{data}}{D_{databuffer}}\right\rceil)$.

No. of merges $(= N_{merge}) = \left\lceil\frac{N_{spill}}{sort.factor}\right\rceil$.

$$T_{Phase6} = N_{spill} * \left(\frac{D_{spill}}{MR_i} + \frac{D_{spill}}{HW_i}\right) + N_{merge} * D_{spill} * (\frac{1}{HR_i} + \frac{1}{HW_i} + \frac{1}{MR_i} + \frac{1}{MW_i}) \quad (11)$$

Phase 7 is used to transmit and shuffle. The product of $\frac{D_{input}}{D_{test*R}}$ and $D_{mapoutput}$ equals the estimated map output size serving to be the input data of each reduce-worker node. The input data of reduce-worker nodes can be stored on a local disk (the reduce-worker node also acts as a map-worker node) or on remote map-worker nodes that must transmit data through the network.

$$T_{Phase7} = \frac{D_{input}}{D_{test*R}} * D_{mapoutput} * (\frac{A_j}{HR_i} + \frac{1-A_j}{Network_i} + \frac{1-A_j}{HW_i}) \quad (12)$$

Phase 8: Sort/ Merge. The data size of reduce input is expressed as $D_{shuffle}$, which is $\frac{D_{input}}{D_{test}} * D_{mapoutput} * \frac{Mtest}{R}$. In shuffling, the downloaded map output is buffered in memory first. When memory buffer is filled at a certain level of usage, the data are written to the disk, as specified in spilling. The time to merge all reduce input data and sort them can be estimated as disk read-write and memory read-write on $D_{shuffle}$ data size. Therefore,

$$T_{Phase8} = D_{shuffle} * \left(\frac{1}{MR_i} + \frac{1}{MW_i}\right) + \left\lceil\frac{D_{shuffle}}{Heap_{red} * shuffle.buffer.percent * shuffle.merge.percent}\right\rceil * D_{shuffle} * (\frac{1}{HR_i} + \frac{1}{HW_i} + \frac{1}{MR_i} + \frac{1}{MW_i}) \quad (13)$$

The final phase, Phase 9 is about Reduce. In this phase, programs may use CUDA for GPU computing. Data in memory must be read first, then the elapsed time of reduce function is estimated by multiplying T_r with the rate of $D_{shuffle}$ and the test map output-data size, $D_{mapoutput}$. After read and reduce, the output of the reduce function is written into HDFS. Hence,

$$T_{Phase9} = D_{shuffle} * \frac{1}{MR_i} + \max_{1\le i\le n}(\frac{Dgpu\_block}{GPU\_Mem\_w} + \frac{D_{gpu\_block}}{D_{test}} * \frac{GPUtest}{GPU_i} * T_{m\_GPU} + \frac{Dshuffle}{GPU\_Mem\_r}) + \frac{D_{shuffle}}{D_{mapoutput}} * \frac{CPU_{test}}{CPU_i} * T_r + \frac{D_{input}}{D_{test*R}} * D_{red} * \left(\frac{1}{HW_i} + \frac{1}{MR_i}\right) \quad (14)$$

C. Notations and Default Setting

Major notations used in this paper are summarized in Table I. Default values are suggested as well. Detailed description of each parameter could be found in [9]. System and application related notations used in this paper are summarized in Tables I and II, respectively. Default values are suggested as well. Detailed description of each parameter could be found in [9].

TABLE I. SYSTEM NOTATIONS AND SETTINGS

Job Phase		
Parameter	*Notation*	*Default*
Input split size	D_{split}	64 MB
No. of referred map tasks	M_ref	4
No. of reduce tasks	R	1
mapred.tasktracker.map.tasks.maximum	Max_map	4

Map Phase		
Parameter	*Default*	*Description*
sort.mb	100 (MB)	The amount of buffer space to use when sorting streams.
sort.spill.percent	0.8	The amount of sort buffer used before spilled to disk.
sort.record.percent	0.05	The amount of metadata buffer used in spilling.
sort.factor	10	The number of map output partitions to merge at a time.
Reduce Phase		
Parameter	*Default*	*Description*
Max Heap size of reduce task ($Heap_{red}$)	1024 (MB)	Max heap size that can be used by reduce task.
parallel_copies	5	The number of map output partitions to merge at a time.
shuffle.buffer.percent	0.7	The amount of buffer space to use when sorting streams.
shuffle.merge.percent	0.66	The amount of the sort buffer used before data spill to disk.

III. VALIDATION OF SPN-GC

Platform-Independent Petri Net Editor 2(PIPE2) v4.3.0 [3] is an open-source tool that supports the design and analysis of Stochastic Petri Net models. PIPE2 uses the “xml” format that is easy to describe the form of a Petri Net. The SPN-GC is validated to conform the regulations of Stochastic Petri Net by PIPE2. Our SPN-GC simulator is constructed using Java and is expected to be released as a package in PIPE2 soon.

A. *Simulation Settings*

In our experimental environment, we run Hadoop-1.2.1 as MapReduce framework [1] of four worker nodes with one physical server each. Each physical server is equipped with GPU of model NVIDIA Tesla C2050 [4]. The details of hardware specification and software version can be found in [9].

The speed of memory is measured by using the “dmidecode” command on Linux. The system measures the elapsed time when creating 100 of 1000 bytes blocks to read in and read out. The speed of disk is measured in the same way. GPU device information can be queried from NVIDIA system management interface.

Three GPU computing benchmarks are studied in our experiments: converting side-by-side (SbS) video to depth video, matrix multiplication, and K-mean clustering. Due to the paper length, here we present the experimental results of 3-D video case and K-means only. Interested readers are recommended to study further [8][9].

For 3-D video conversion, the program is to transfer side-by-side video into depth video based on tremendous graphic processing. Every frame in the SbS video is composed of two almost-identical pictures except a little different angle of camera view. The depth video can be played on a 3-D monitor and delivers the 3-D visual effect. The data set was collected from Internet and the file format is any SbS video of .mp4 file.

TABLE II. APPLICATION RELATED NOTATIONS

Parameter	**Notation**
Input data size of the estimated job	D_{input}
Input data size of the test small job	D_{test}
Program loading and data split time of the test small job	T_{load}
Exec. time of map function in the test small job	T_m
Exec. time of reduce function in the test small job	T_r
Size after executing map function of the test small job	D_{map}
Size after executing reduce function of the test small job	D_{red}
Size of map output which is equal to reduce shuffle bytes	$D_{mapoutput}$
Data size of GPU block bytes	D_{gpu_block}

We use these SbS videos to compare the accuracy between the actual job execution time and the estimated execution time by SPN-GC. Since most of the GPU computing applications are not utilizing the MapReduce framework yet, we need to port the program written in CUDA to MapReduce framework. This justifies our major contribution in the area. To collect data, ten sets of different data size are fed in to each program to test the performance under different input configurations.

The second experiment is about K-means clustering, which is a method of vector quantization, originally from signal processing. A popular cluster analysis in data mining, K-means clustering aims to partition *n* observations into *k* clusters in which each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster. The problem is known to be computationally difficult (i.e., NP-hard). In our experiment, the observations are generated by random variables using time values as seed.

Our SPN-GC simulator is following the nine-phase execution and calculating the mean delay time to estimate the mean elapsed time in each phase and return the total estimated execution time. Exponential distributed random variables are used. Every run of simulation was performed 2000 times. The average of all simulated values is returned as the estimated execution time for the program.

B. *Experimental Results*

Figure 1 shows the results of converting SbS videos into depth videos. The execution time of actual test and estimated time by SPN-GC for input video length from 30, 60, 90, 120, 150, 180, 210, 240, 270, 300 seconds are shown respectively. We can see that the average of estimated execution time and actual test values were very close.

Figure 2 illustrates the results of the K-Means execution time of actual test and estimated execution time by SPN-GC. The input data size ranges from 0.5, 1, 2, 3, 4, 5, 6, to 7 millions. Every block reads 20 thousands as input. The application randomly chooses 10 points as the centers, then partition *n* observations into 10 clusters with the nearest mean to the cluster. We can see that the estimated execution time of each input data is very close to the actual test value.

To evaluate the significance of SPN-GC model, we compare the time taken by SPN-GC and by actually executing the test job, Time Cost Ratio is calculated to reflect how much time saving is obtained by SPN-GC estimation. SPN-GC is proved to be able to save lots of time in cluster selection or performance tuning.

$$\text{Time Cost Ratio} = \frac{time\ spent\ by\ SPN{-}GC}{Actual\ job\ execution\ time} * 100\%$$

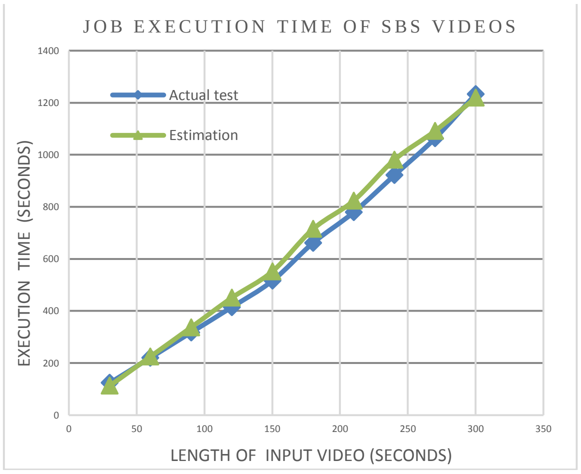

Figure 1. Job Execution Time of Side-by-side Videos.

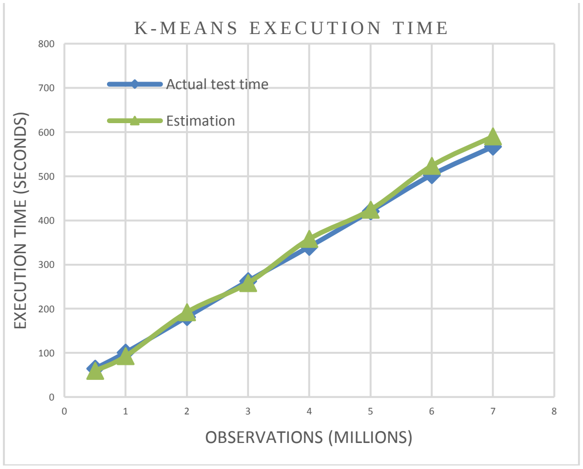

Figure 2. K-Means Execution Time.

Figure 3 draws the time cost ratio of SPN-GC to actual time cost for K-means clustering. The input range is from 0.5 to 8 millions of observations. We can see that, as the data size grows, the ratio drops to 0.7% approximately. It can be identified that SPN-GC would be able to perform an accurate estimation of the execution time for a job with a very small time cost. The benefit becomes more significant when the size of the problem increases.

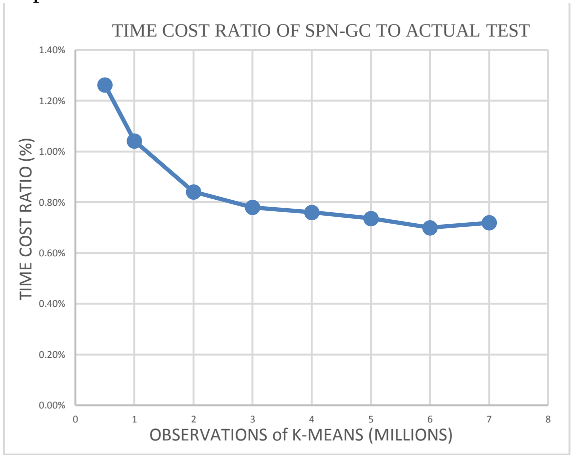

Figure 3. Ratio of SPN-GC Time Cost.

IV. CONCLUSIONS

In this paper, we develop the SPN-GC model to estimate the execution time of a job under the MapReduce framework over GPU clusters. Job execution time is an important performance indicator that provides crucial information for cluster evaluation. The considered environment is that input data are split into Hadoop block sizes and then spilt it again into the blocks for CUDA in GPU computing. There is also a problem that when GPU is computing graphic processing, every block gets different data and therefore each block has its own complexity. The data complexity and mean delay time are solved under the assumption of exponentially distributed random variables. In experimental results, SPN-GC is validated by PIPE2 and compared the estimation execution time with actual data test under three applications. The average error range of estimation execution time was found to be within 10%. SPN-GC can be a reference to evaluate GPU clusters performance.

ACKNOWLEDGMENT

This work is supported in part by Ministry of Science and Technology, Taiwan, R.O.C. under the grant number “102-2221-E-006-086-MY3” and by Ministry of Economic Affairs, Taiwan, R.O.C. under the grant number “103-EC-17-A-02-S1-201” respectively.

REFERENCES

[1] G. Wang, A. R. Butt, P. Pandey, and K. Gupta, “A Simulation Approach to Evaluating design decisions in MapReduce setups,” IEEE Symposium, Modeling, Analysis & Simulation of Computer and Telecommunication Systems, pp. 1-11, 2009.

[2] M. K. Molloy, “Performance Analysis Using Stochastic Petri Nets,” IEEE Transactions, Computers, vol. C-31, issue 9, pp. 913-917, 1982.

[3] N. J. Dingle, W. J. Knottenbelt, and T. Suto, “PIPE2: A Tool for the Performance Evaluation of Generalized Stochastic Petri Nets,” ACM SIGMETRICS, Performance Evaluation Review, vol. 36, issue 4, pp. 34–39, 2009.

[4] E. Lindholm, J. Nickolls ,S. Oberman, and J. Montrym, “NVIDIA Tesla: A Unified Graphics and Computing Architecture,” IEEE conference, pp. 39-55, 2008.

[5] Y. Guo, W. Liu, G. Voss, and W. Mueller-Wittig, “GCMR: A GPU Cluster-based MapReduce Framework for Large-scale Data Processing,” IEEE conference, High Performance Computing and Communications & Embedded and Ubiquitous Computing, pp. 580-586, 2013, doi:10.1109/tencon.2013.6719008.

[6] J. A. Stuart and J. D. Owens, “Multi-GPU MapReduce on GPU Clusters,” IEEE Intl. Symp. Parallel & Distributed Processing, pp.1068–1079, 2011,.

[7] H. Gao, J. Tang, and G. Wu, “A MapReduce Computing Framework Based on GPU Cluster,” IEEE Conference, High Performance Computing and Communications & Embedded and Ubiquitous Computing, pp. 1902-1907, 2013.

[8] H. Wang, “Using Petri Net to Estimate Job Execution Time in MapReduce Model,” master thesis, National Cheng Kung University, Taiwan, 2013.

[9] H. Yang, “Estimation of Job Execution Time in MapReduce on GPU Clusters,” master thesis, National Cheng Kung University, Taiwan, 2014.

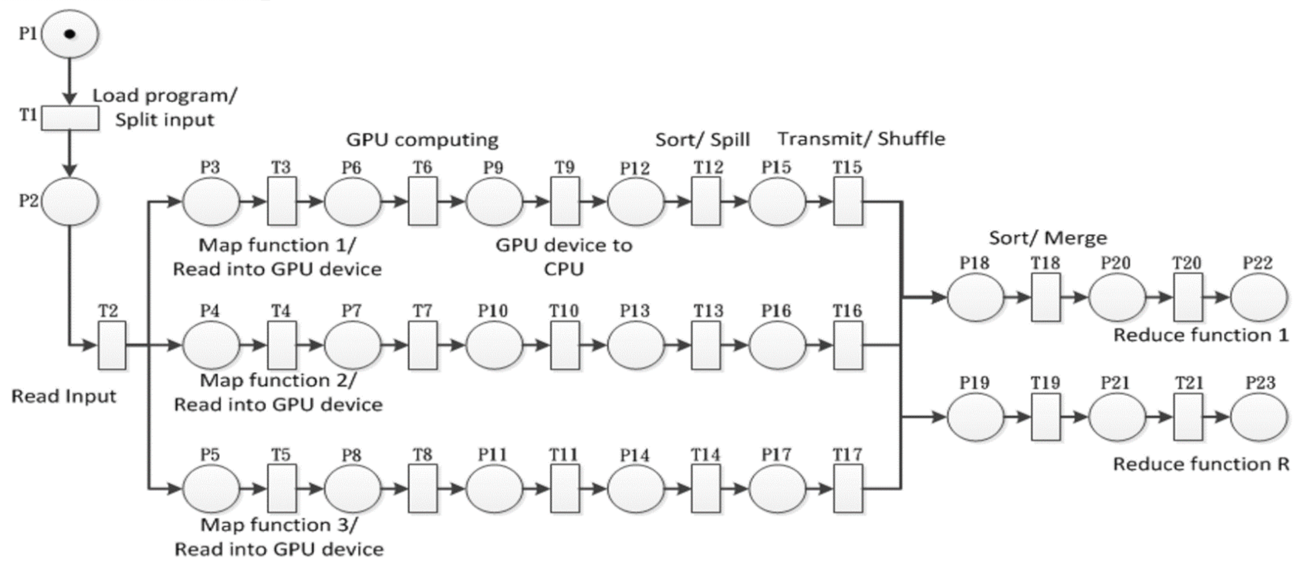
P1
T1
Load program/ Split input
P2
T2
Read Input
GPU computing
Sort/ Spill
Transmit/ Shuffle
P3 T3 P6 T6 P9 T9 P12 T12 P15 T15
Map function 1/ Read into GPU device
GPU device to CPU
P4 T4 P7 T7 P10 T10 P13 T13 P16 T16
Map function 2/ Read into GPU device
P5 T5 P8 T8 P11 T11 P14 T14 P17 T17
Map function 3/ Read into GPU device
Sort/ Merge
P18 T18 P20 T20 P22
Reduce function 1
P19 T19 P21 T21 P23
Reduce function R

Figure 4. Nine phases of SPN-GC (with M =3, R =2).