A Task-level Adaptive MapReduce Framework for Real-time Streaming Data in Healthcare Applications

Fan Zhang

Kavli Institute for Astrophysics and Space Research

Massachusetts Institute of Technology

Cambridge, MA 02139, USA

Email: f_zhang@mit.edu

Junwei Cao

Research Institute of Information Technology

Tsinghua University

Beijing, China, 100084

Email: jcao@tsinghua.edu.cn

Samee U. Khan

Department of Electrical and Computer Engineering

North Dakota State University

Fargo, ND 58108-6050, USA

Email: samee.khan@ndsu.edu

Keqin Li

Department of Computer Science

State University of New York

New Paltz, New York 12561, USA

Email: lik@newpaltz.edu

Kai Hwang

Department of Electrical Engineering and Computer Science

University of Southern California

Los Angeles, CA 90089, USA

Email: kaihwang@usc.edu

Abstract

Healthcare scientific applications, such as body area network, require of deploying hundreds of interconnected sensors to monitor the health status of a host. One of the biggest challenges is the streaming data collected by all those sensors, which needs to be processed in real time. Follow-up data analysis would normally involve moving the collected big data to a cloud data center for status reporting and record tracking purpose. Therefore, an efficient cloud platform with very elastic scaling capacity is needed to support such kind of real time streaming data applications. The current cloud platform either lacks of such a module to process streaming data, or scales in regards to coarse-grained compute nodes.

In this paper, we propose a task-level adaptive MapReduce framework. This framework extends the generic MapReduce architecture by designing each Map and Reduce task as a consistent running loop daemon. The beauty of this new framework is the scaling capability being designed at the Map and Task level, rather than being scaled from the compute-node level. This strategy is capable of not only scaling up and down in real time, but also leading to effective use of compute resources in cloud data center. As a first step towards implementing this framework in real cloud, we developed a simulator that captures workload strength, and provisions the amount of Map and Reduce tasks just in need and in realtime.

To further enhance the framework, we applied two streaming data workload prediction methods, smoothing and Kalman filter, to estimate the unknown workload characteristics. We see 63.1% performance improvement by using the Kalman filter method to predict the workload. We also use real streaming data workload trace to test the framework. Experimental results show that this framework schedules the Map and Reduce tasks very efficiently, as the streaming data changes its arrival rate.

Key words: Adaptive Mapreduce; Big data; Healthcare scientific applications; Kalman filter; Parallel Processing

1. INTRODUCTION

Healthcare science has been consistently pushed forward by the advent of big-data technology. Healthcare scientific applications usually involve streaming input data generated by a large number of distributed sensors. Such data are further sent to the state-of-art big-data frameworks and platforms to process. For example, the Body Area Network [1] that is widely recognized as a medium to access, monitor, and evaluate the real-time health status of a person, has long been notorious for its computing intensiveness to process Gigabytes of data [2][3] in real-time. Such data are collected from well-configured sensors to sample the real-time signals of body temperature, blood pressure, respiratory and heart rate, chest sound, and cardiovascular status, to name a few among others.

To process stream big-data in real-time, traditional parallelized processing frameworks, such as Hadoop MapReduce [4], Pregel [5], and Graphlab [6][7], are structurally constrained and functionally limited. The major difficulty lies in their designs are primarily contrived to access and process the static input data. No built-in iterative module can be used when the input data arrives in a stream flow. Moreover, the existing frameworks are unable to handle the scenarios when the streaming input datasets are from various sources and have different arrival rates. Healthcare scientific applications vary the data acquisition frequency when the behavior of the person changes. For example, the data collected when a person is sleeping can be far less than the data collected when the person is running or swimming.

To process the highly dynamic, fluctuating, and varying datasets, cloud computing, with elasticity as one of its major advantages, provides a perfect match as a potential and scalable solution. One of the most successful commercial stories that leverage the scalability of a cloud platform is the Amazon Elastic MapReduce (EMR) framework [8]. The EMR scales up and scales out when workload varies. However, a major challenge is that the EMR scales the compute nodes as the golden-rule principle. We have proven in our previous study [9] that the MapReduce faces a large-scale limitation issue, that is, Mapreduce applications stop to promise the desirable scalability when the cluster size is very large and the Map and Reduce tasks cannot utilize the compute resources. Therefore, the scalability would find itself more tractable when one the MapReduce application scales at a task level — increasing or reducing the Map and Reduce task number when a variation of the workload is predicted.

To date, it is difficult to find such fine-grained scalable tools to process stream data, even though there are many commercial products that have been designed to serve the purpose. These products will be reviewed in the next section. Different design strategies as they are, none of them, to the best of our knowledge, scales to process tasks in such a fined grained manner. Moreover, most of these products lack a built-in support for processing data on a scalable cloud data center, and lack

a solid mathematical simulator to identify the optimal configuration option when varied workload arrives.

Towards that end, we propose a full-fledged MapReduce framework that is tailored to process streaming data in healthcare scientific applications. The framework goes beyond the traditional Hadoop MapReduce design, while also providing a much generic framework to cover a wider group of MapReduce applications. The generic MapReduce framework is built upon the widely used Hadoop MapReduce but is also quite different. The implementation of Hadoop MapReduce is hard coded while a generic MapReduce framework can be more flexible. For example, Hadoop MapReduce requires each Map and Reduce task be implemented as a separate Java Virtual Machine (JVM) while in generic MapReduce, one task can be specified as a JVM, a local process or a compute node among others. In other words, generic MapReduce takes the MapReduce design logic, that is multiple Map tasks connect to multiple reduce tasks, but can be implemented in a more flexible manner. The purpose of using this generic MapReduce is to ease our stream data modeling process, which is by no means being supported in Hadoop MapReduce at all. The major contribution of this paper is summarized below.

(1) We propose a task-level adaptive MapReduce framework to process streaming data in healthcare scientific applications. This framework extends the traditional Hadoop MapReduce framework and specifically addresses the varied arrival rate of big-data splits. The framework is designed to scale in heterogeneous cloud platforms by applying four scaling theorems and scaling corollaries. It is now fully developed and delivered as a standalone simulator, which implements all the theorems and corollaries.

(2) Stream data scientific applications need to estimate the real-time data arrival rate and plan for the computing resources accordingly. In this paper, we propose two workload prediction methods and compared their benefits and performance in real-life healthcare scientific applications.

(3) Real streaming data trace are used to justify the applicability of the framework. We report the experimental results by showing the real time Map and Reduce tasks number variation, which matches perfectly with the variation of the streaming data arrival rate.

This paper is organized as follows. In the first section we introduce the motivation, significance, and challenges of healthcare scientific applications and their direct requirement of processing stream-style big-data. The second section investigates the related work, and as a sneak peek overview we also give the core idea of our unique approach. A real healthcare application case is discussed in the third section. Following that, we identify the methodological details of the task-level adaptive framework in the fourth section. Two methods to predict the streaming data workload are proposed in the fourth section. We introduce our experimental settings and report the results in the fifth section. In the final section, we conclude the work and summarize a few directions to extend the work.

2. Related Work and our Unique Approach

There is an escalating interest on leveraging the state-of-art big-data platform to process stream data in real-time. In this section, we investigate previous publications in this area. Thereafter, we briefly describe our unique approach to show the advantage among other solutions.

2.1 Related Work

Health Information System [10] was originated and further extended from the hospital information system [11] that addresses what is called the health informatics issues. The major challenge involves the shift from paper-based to computer-based, and further to the Internet-based data storage processing. Patients, healthcare consumers, and professionals are more involved into a collaboration phase from a traditional in- or out-patient medication, to a widely acceptable online on-demand treatment. Such a shift requires a significantly powerful interconnection compute network and highly scalable compute nodes for both computing and big-data processing.

MapReduce is a simple programming model for developing distributed data intensive application in cloud platforms. Ever since Google initially proposed it on a cluster of commodity machines, there have been many follow-up projects. For instance, Hadoop [12] is a Mapreduce framework developed by Apache, and Phonix [13] is another framework designed for shared memory architecture by Stanford University. Pregel [5] is a message-based programming model to work on real-life applications that can be distributed as an interdependent graph. It uses vertex, messages, and multiple iterations to provide a completely new programming mechanism. GraphLab [6], [7] is proposed to deal with scalable algorithms in data mining and machine learning that run on multicore clusters.

The above-mentioned tools have a wide impact on the big-data community and have been extensively used in real-life applications. Along those lines, other research efforts addressing streaming data have been proposed. Nova [14], due to its support for stateful incremental processing leveraging Pig Latin [15], deals with continuous arrival of streaming data. Incoop [16] is proposed as an incremental computation to improve the performance of the MR framework. Simple Scalable Streaming System (S4) [17], introduced by Yahoo!, is universally used, distributed, and scalable streaming data processing system. As one of its major competitor, Twitter is using Storm [18] that has also gained momentum in real-time data analytics, online machine learning, distributed remote procedure call, ETL (Extract, Transform and Load) processing, etc. Other companies, such as Facebook, Linkedin and Cloudrera, are also developing tools for real-time data processing, such as Scribe [19], Kafka [20], and Flume [21]. Even though the programming languages are different, they all provide highly efficient and scalable

structure to collect and analyze real-time log files. Complex Event Processing systems (CEP) are also gaining interest recently. Popular CEP systems include StreamBase [22], HStreaming [23], and Esper&NEsper [24]. Essentially the CEP systems are primary used in processing inter-arrival messages and events.

Different from these research and commercial products, our work goes beyond a programming model framework, but also serves as a simulator to help users identify how their compute resources can be effectively used. Secondly, the framework is still based on a generic MapReduce, but not entirely a Hadoop MapReduce framework. We do not intend to design a completely new framework, but we aim to extend a widely acceptable model to allow it to seamlessly process streaming data. Our work may aid the programmers to manipulate the streaming data applications to process such kinds of flow data in a more scalable fashion.

2.2 Our Unique Approach

In a nutshell, our approach implements each Map and Reduce task as a running daemon. Instead of processing static data in Hadoop Distributed File System (HDFS) as traditional Hadoop does, the new Map tasks repeatedly fetch stream data that have been cached in the HDFS, process the data and push the intermediate key-value pairs to the corresponding Reduce tasks. These Reduce tasks, similar to the Map tasks, are also implemented as running daemons. These daemons repeatedly pull the corresponding data partitions from the entire Map task output, digest them, and push the output to a local cache. In this way, each Reduce task has to save the intermediate status of all the output key-value pairs when the application is still running.

For example, an enhanced WordCount application requires obtaining a real-time count of each word when the input dataset needs to be updated and edited. Multiple users add/remove/update words, sentences and files to HDFS as data streams. To implement such a framework, Map tasks are stateless, meaning that they just simply process the corresponding input data and produce output without having to worry about previous data that they have processed. However, the Reduce tasks must be implemented in a stateful way. This means that each of the Reduce task has to save the real-time count of each word and adaptively add or reduce the count whenever there is a change in the HDFS.

The essence of our approach, as we can see from the analysis, lies in the seamless connection to the MapReduce implementation. Users can develop data streaming applications in all the way they developed traditional MapReduce applications. The only difference is that they must define such a daemon, instructing the Map and Reduce tasks how to process the stream data. Secondly, our approach can be implemented to scale the Map and Reduce tasks separately. Traditional MapReduce scaling compute nodes usually leads to low compute resource usage when the active running tasks cannot utilize these compute nodes effectively.

An example is given below to illustrate the task-level adaptive MapReduce application that calculates the real-time occurrence of each word in a set of documents. These documents are consistently updated by multiple people, and therefore the statistics of each word count differ from time to time.

The Map tasks below are continuously fed by input data stream, and enter into a loop that would not stop until the data stream update ends. For each of the loop, all the words are extracted and emitted key value pairs as tradition WordCount does. The Reduce tasks, also being launched in a loop, are fed continuously by the intermediate data produced by all of the Map tasks. The only difference here is the result, which needs to be fetched from HDFS. Because for each result that has been calculated, it needs repeatedly updating. Therefore, Reduce tasks should be able to not only write data back to the HDFS, but also retrieve data back from HDFS for updating.

```
map(String key, String value):
// key: document name in a stream data
// value: document contents
While(MoreDataStream)
    value = GetCachedStreamData();
    for each word w in value:
        EmitIntermediate(w, "1");


reduce(String key, Iterator values):
// key: a word
// values: a list of counts
While(MoreIntermediateData)
    int result = getResultFromHDFS;
    for each v in values:
        result += ParseInt(v);
        Emit(AsString(result));
```

3. Problem Formulation of a Real-life Healthcare Application

In Figure 1, we illustrate a case study of the body area network as a real-life healthcare application. Health status regarding the respiration, breath, cardiovascular, insulin, blood, glucose and body temperature, which are consistently collected by sensors deployed all over the human body. This is what we call wearable computing, which sends data repeatedly to a mobile phone via local network. The sampling frequency is determined by the capacity of the sensors as well as the processing rate of the mobile device.

Because most of the mobile devices nowadays are equipped with advanced processing unit and large memory space, data can be continuously transferred to the mobile devices and even processed locally. Therefore, the various sources of input data need to be locally analyzed before moving to the remote data center. The data center has information on various disease symptoms and the corresponding value threshold in regards to the insulin pump and glucose level, etc. The purpose of the follow-up data transferring is to compare the collected data with those in the database, and quickly alert the user the potential symptom he/she is supposed to see, and provide a smart medical suggestion in real-time.

As a typical case study of wearable computing, the data-sampling rate can be as large as Gigabyte per minute. The higher the sampling rate is, the better the real-time medical treatment can be achieved. Therefore, a strong requirement is posed on the computing device as well as the backend data center. The data center may need to serve thousands of persons with millions of sensors. Real-time data stream needs to be processed to minimize the response time to all users.

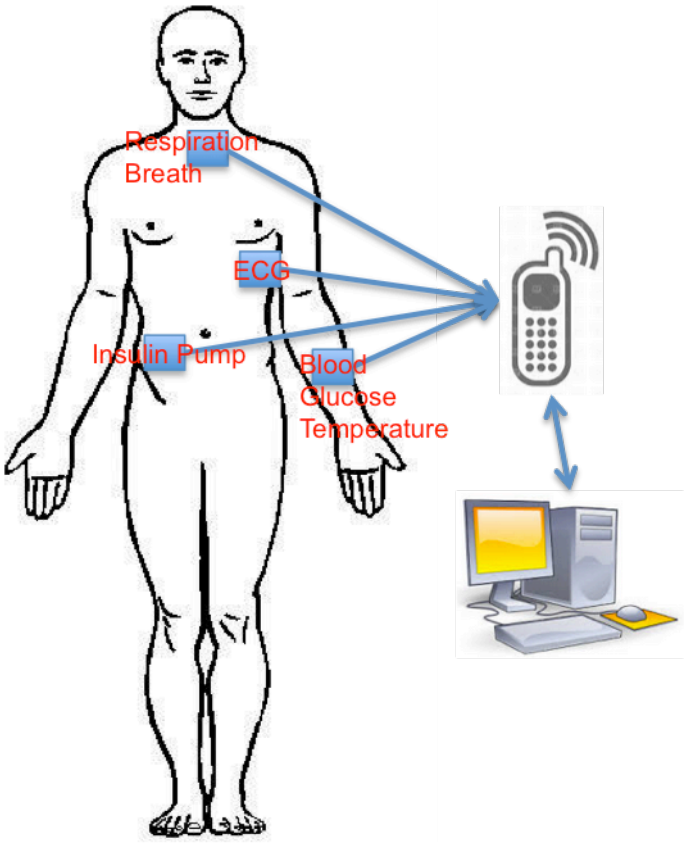


Figure 1. As a case study of the body area network, data streams collected from various sensors are pushed to a mobile device and backend data center for real-time medical treatment

4. TASK-LEVEL ADAPTIVE MAPREDUCE FRAMEWORK

In this section, we brief an overview of the Hadoop MapReduce framework as a start. Thereafter, the task-level provisioning framework is introduced in the subsequent text.

4.1 Preliminary of Hadoop MapReduce Framework

One skeleton of the standard Hadoop MapReduce framework is given in Figure 2. There are *4* parallel Map tasks and *3* parallel Reduce tasks, respectively. Because the total number of the Map tasks normally equals to the number of the input data splits, there are four data splits as well. Each Map task performs a user-defined Map function on the input data that resides in the HDFS and generates the intermediate key-value pair data. These intermediate data are organized on the *partition* basis. Each of the partition consists of certain key-value data pairs, whose keys can be classified into one group. The simplest classification method is a hash function. Under such a hash function, data partition belongs to the same group are shuffled across all the compute nodes and merged together. There are three data partitions shown in the figure. These merged data

partitions, as indicated by three different framed rectangular boxes, are consumed by three Reduce tasks separately. The output data generated by all the Reduce tasks are written back to the HDFS.

Each Map task resides in a Map slot of a compute node. Normally each compute node has two Map slots. The slot number per node can be adjusted in a MapReduce configuration file. The total number of the Map slots determines the degree of parallelism that indicates the total number of Map tasks that can be concurrently launched. For the Reduce task and Reduce slot, it is the same procedure. The whole Hadoop MapReduce workflow is controlled in a JobTracker located in the main computer node, or what is called the NameNode. The Map and Reduce tasks are launched at the TaskNodes, and each task corresponds to one TaskTracker to communicate with the JobTracker. The communication includes heat-beat message to report the progress and status of the current task. If detecting a task failure or task straggler, the JobTracker will reschedule the TaskTracker on another Task slot.

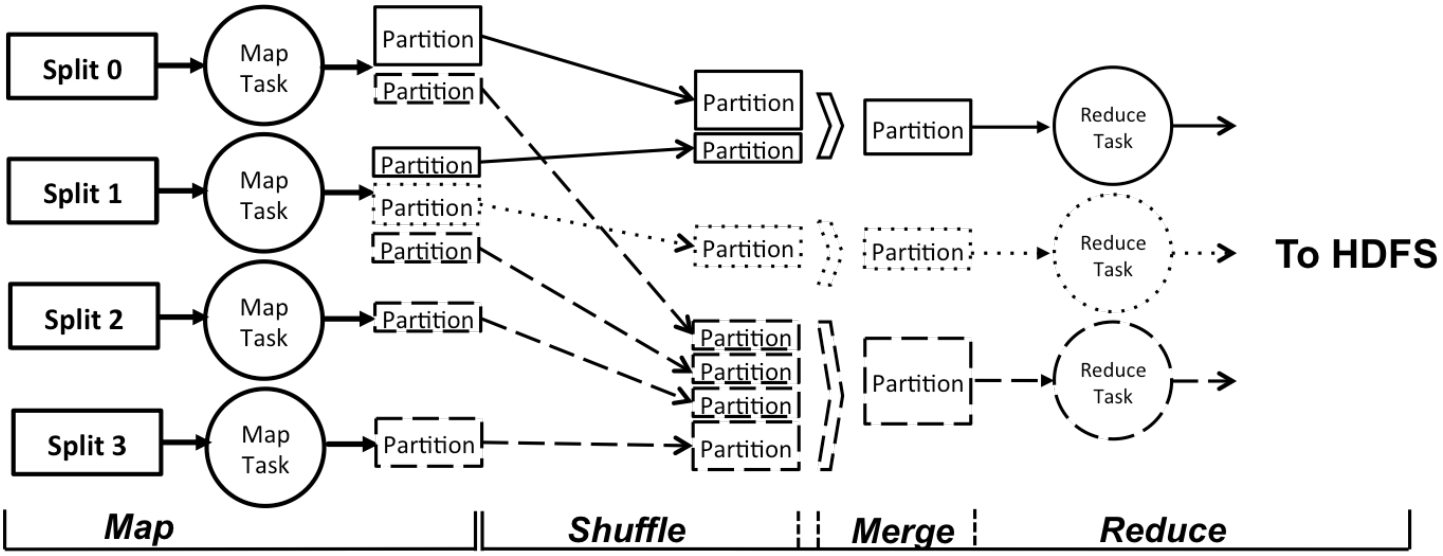


Figure 2. A MapReduce framework splits the input file into 4 segments, and each segment corresponds to one Map task. Map Tasks output data partitions, which are further shuffled to the corresponding Reduce tasks. There are *3* reduce tasks which generate *3* separate outputs.

As we can see from Figure 2, the Hadoop MapReduce is essentially a scheduling framework that processes data that can be sliced into different splits. Each Map task works on its own input data split without having to interact with other Map tasks at all. The Hadoop MapReduce framework can only be applied to process input data that have already existed. However, real-life scenarios of the state-of-art big-data applications that typically require the input data be provisioned in streaming and be processed in real-time. Therefore, an enhanced MapReduce framework is required to cater for such a need. That is the motivation behind out design of the task-level adaptive MapReduce framework.

4.2 Task-level Adaptive MapReduce Framework

An adaptive MapReduce framework is proposed to process the streaming data in real-time. A significant challenge here is how to address the incoming data streams with the varied arrival rate. Real-life application scenarios include workloads of various features. Some of the workloads show a typical pattern of periodical and unpredictable spikes, while others are more stable and predictable. There are four technical issues that we should consider when designing the adaptive framework.

First, the framework should be horizontally and vertically scalable to process a mixture of varied workloads. In other words, the compute nodes must be either scalable in terms of the total available number, but also in terms of a variety of the compute node types. For some Hadoop MapReduce applications, simply increasing the number of compute nodes is not necessarily sufficient. Certain kinds of workloads require large CPU-core instances while others need large-memory instances. In a nutshell, scaling in a heterogonous system is one of the primary principles.

Second, scaling the number of the Map and Reduce tasks should align with the scaling of the cluster size. It is actually the concurrent Map and Reduce task number that determines the performance of the scheduling framework. We must avoid scaling cluster size without efficiently using them. Therefore, there are lots of problems to consider. For example, as the number of the Reduce task increases, the hash function that maps a particular Map output partition data to a Reduce function changes. Take modular operation as a hash function as an example. Increasing the Reduce count from r to r' leads to *key mod r* to *key mod r'* as the corresponding new hash function.

Third, we also need to consider the heterogeneity of the processing capabilities of different Map tasks. Some of the Map tasks are scheduled on a slow compute node while others process data on much faster counterparts. An appropriate load balancing mechanism can further improve the rescheduling philosophy implemented in the traditional Hadoop MapReduce. The purpose is to coordinate the progress of the entire task without leading to skew task execution time.

Fourth, the optimal runtime Map and Reduce task count should be specified. Traditionally, the initial Map task count depends on the input dataset size and the HDFS block size. The Reduce task count is determined by the hash function. The new framework requires a redesign of the Map and Reduce Task scheduling policy by considering the input data arrival rate instead of their sizes instead.

To serve the purpose, we demonstrate the task-level adaptive MapReduce framework as shown in Figure 3 below. Two Map tasks are selected as representatives to show the Map stage. Each Map task, different from the Map task of the traditional Hadoop MapReduce, defines a loop function as shown in the self-pointed arrow. Map tasks are launched as special runtime daemons to repeatedly processing the incoming data. As shown in the figure, each of the Map produces two continual batches

of output data partitions. Similarly, the Reduce tasks are also scheduled in such a loop-like daemon that continuously process their corresponding intermediate data produced by all the Map tasks.

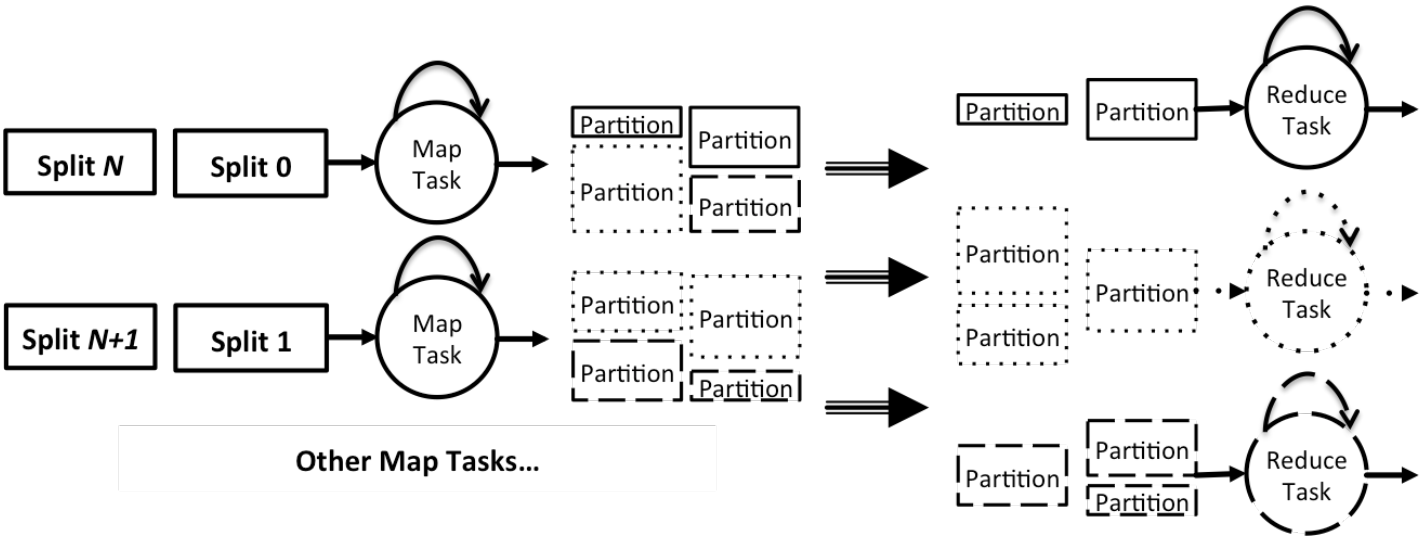


Figure 3. A demonstration of task-level adaptive MapReduce framework which processes streaming data. Each Map and Reduce task has a non-stop running daemon function which continuously processes the input data.

In this novel task-level adaptive MapReduce framework, the JobTrackers need to be redesigned to maintain a pool of TaskTrackers, and the TaskTracker count subject to change as the workload changes.

There are two ways to feed data split streams to the Map tasks. A proactive strategy caches streaming data locally first and pushes them every fixing period of time, for example every one minute. As an alternative option, data splits can also be pushed in a reactive way. In other words, a cache size is defined in HDFS before the input data starts to move in, whenever the cache usage hits a ratio, say 85%, the data splits begin to be pushed to the Map tasks.

4.3 Adaptive Input Data Split Feeding

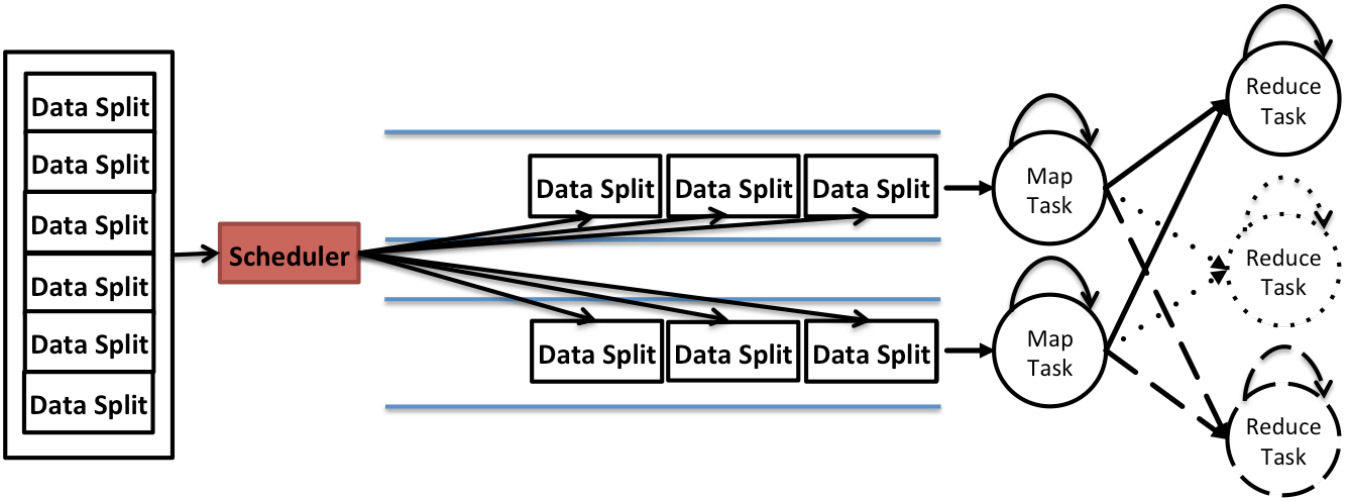
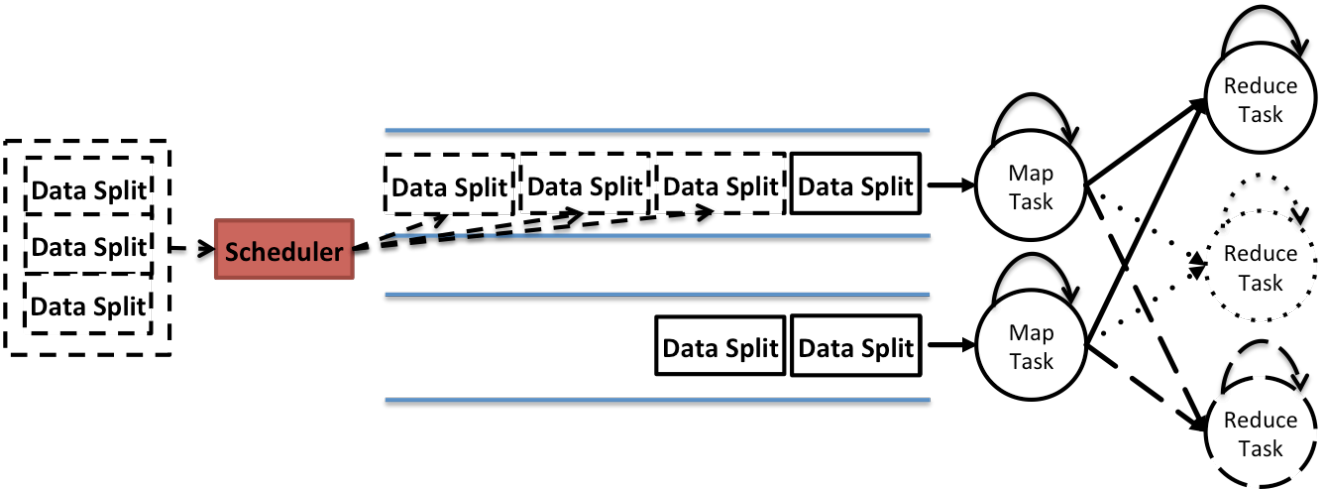


Figure 4(a). Initially six data splits arrive. Without knowing the compute capacity of each Map task, scheduler divides the workload evenly between the two queues, each one having three data splits.



4(b). After being aware of the processing capacity of each Map task, the scheduler sends three data splits to Map task one which shows twice the processing capacity at the consecutive scheduling period.

Figure 4. A demonstration of the adaptive input data split feeding.

The adaptive MapReduce framework starts from a novel runtime scheduler that feeds different Map tasks with different number of data splits. In Figure 4(a), we show a study case of the adaptive input data split feeding. As a start, six splits of input data arrive. The scheduler, without knowing the processing capability of each Map task, distributes the data splits evenly to the two Map tasks, which results to each Map task having three data splits. Suppose the first Map task is executed on a faster compute node and has processed two splits of the input data while the second Map task has processed only one. Being

aware of such a skewed processing capability, the scheduler sends the newly arrival three data splits adaptively to balance the workload in Figure 4(b). This leads to Map task one has four data splits while Map task two has two, and the total execution time of the Map stage is minimized.

In this case, processing the three newly arrival data splits doesn't result in an increase of Map task count, but trigger the scheduler to dispatch them fairly to all the Map tasks. Scheduler caches the input data locally in HDFS and regularly sends them to different Map tasks. The time interval is also adaptively determined by the arrival rate of the data splits.

To refresh readers' memory and ease difficulty in understanding all the mathematics below, we plot table one below, which summarizes all the symbols and explain their meanings.

Table 1. Symbols, notations and abbreviations with brief introduction

Notation	Brief Definitions with Representative Units or Probabilities
m	The total number of available Map tasks
Q	The total number of data splits that can be accommodated in each Map task
n_0	The total number of data splits arrives at the start time
t	The scheduling period, denoting the data feeding frequency from the scheduler to all Map tasks
$dMapTaskN(j)$	The number of data splits that remained in the queue of Map task j at time t_i
$dMapTaskN'(j)$	The number of data splits that remained in the queue of Map task j at time t_{i+1}
$eMapTaskC(j)$	The estimated data processing capacity for Map task j
$addedDataSplit(j)$	The number of data splits that needs to be added to Map task j after new stream data arrives
TT	Estimated finish time of all the Map tasks
α	Upper bound percentage threshold used when Map task number above $\alpha*Q$ in a queue, Map tasks are over provisioned
β	Lower bound percentage threshold used when Map task number below $\beta*Q$ in a queue, Map tasks are under provisioned

Suppose there are m Map tasks and the task queue length of each Map task equals to Q. In the Figure 4, we set m equal to 2 while Q equal to 4. Suppose a start input data includes n_0 data splits. As long as n_0 be less than $m*Q$, each Map task gets n_0/m data splits.

The scheduling period, namely the time interval between two data feeding periods is t. In other words, every t units (seconds or minutes) of time, scheduler feeds one batch of the cached data into the Map queue. Suppose at time t_i, the data splits count of each Map task queue equals to [$dMapTaskN(0)$, $dMapTaskN(1)$, …, $dMapTaskN(m-1)$] after the newly arrived data splits have been pushed into the queues. After time t at t_{i+1}, the remaining task count becomes [$dMapTaskN'(0)$, $dMapTaskN'(1)$, …, $dMapTaskN'(m-1)$]. The estimated processing capacity of each Map task is estimated as [($dMapTaskN'(0)$ - $dMapTaskN(0)$)/t, ($dMapTaskN'(1)$ – $dMapTaskN(1)$)/t ,…, ($dMapTaskN'(m-1)$ – $dMapTaskN(m-1)$)/t].

Suppose n_{i+1} data splits arrive at time t_{i+1}, a scheduling algorithm, which effectively distributes all these data splits to the

Map tasks, becomes our first concern (Theorem 1). Second, a mechanism determines if the number of Map task needs to change or not is also needed (Corollary 1). The change can be either increasing or decreasing the number. Third, if the change is needed, how many Map tasks need to be added or removed (Theorem 2)? In this section, we address the scenario that the Map task number doesn't need to change. Therefore, we answer the first and second question. In the next section where adaptive Map task number is discussed, we explore the solution of the third question.

Theorem 1: **Condition**: Suppose there are m_i Map tasks being actively used at time $t+1$. As a new stage, N_{i+1} new data splits arrive. For any Map task j, $dMapTaskN(j)$ data splits are in its queue. Its estimated data processing capacity is $eMapTaskC(j)$.

Conclusion: The new data split count to be added to its queue is represented by:

$$addedDataSplit(j) = eMapTaskC(j) * (N_{i+1} + SUM(dMapTaskN(:))) / SUM(eMapTaskC(:)) - dMapTaskN(j) \quad (1)$$

$SUM(dMapTaskN(:))$ denotes the total number of data splits across all the queues. $SUM(eMapTaskC(:))$ denotes the aggregated processing capacity of all the Map tasks.

Proof: The scheduling target is to make sure all the tasks of the Map queues be finished almost at the same time, and let that task time be an unknown value TT. For any Map task j, $TT = (dMapTaskN(j) + addedDataSplit(j))/ eMapTaskC(j), j \in [0, m_i-1]$. Note that $\Sigma\ dMapTaskN(j) = N_{i+1}$. Solving a total of m_i-1 equations leads to the proof of theorem 1. **Q.E.D**

Corollary 1: Let Q be the queue length of each Map task, namely the total number of data splits that can be accommodated in one Map task queue. Other conditions are the same as in the Theorem 1. Then new Map tasks need to be added if $\exists\ j \in [0, m_i-1], dMapTaskN(j)) + addedDataSplit(j) > \alpha*Q$. Similarly, Map task number needs to be reduced if $\forall j \in [0, m_i-1], dMapTaskN(:) + addedDataSplit(j) < \beta*Q$. Symbol $\alpha \in [0, 1]$ is a preset threshold to determine how full the Map task queues are allowed. Similarly, $\beta \in [0, 1]$ is preset to determine how empty the Map task queues are allowed.

Proof: If $\exists\ j \in [0, m_i-1], dMapTaskN(j)) + addedDataSplit(j) > \alpha*Q$, this means the Map task number of one Map task queue will be above threshold if the new data splits were added. It automatically triggers a Map task bumping request to the scheduler. Similarly, if $\forall j \in [0, m_i-1], dMapTaskN(j) + addedDataSplit(j) < \beta*Q$ holds, this indicates the Map task count of each Map queue is less than a preset value, which means sufficient resources have been provided. A request is therefore sent out to reduce the Map task count. **Q.E.D**

In a nutshell, the purpose of designing such an adaptive scheduler is to leverage the processing capability of all the Map tasks and balance the start time of all the Reduce tasks.

4.4 Adaptive Map Task Provisioning

In the previous section, we focus on discussing the Map task provisioning condition, that is, when Map task needs to be

updated. A direct and natural extension along that line requires answering a provisioning mechanism — how many Map tasks need to be added or reduced to process the new stream data splits. If adding Map task is required, how to distribute the stream data splits across all the Map tasks, including the new added ones. On the contrary, if reducing Map task is required, how to distribute the stream data splits, as well as the data splits in the queues that are supposed to remove, to all the remaining Map task queues.

Theorem 2: Given the condition in Theorem 1 and $\exists\, j \in [0, m_i\text{-}1],\ dMapTaskN(j)) + addedDataSplit(j) > \alpha^*Q$, the number of the new Map tasks that is needed is given below:

$$\lfloor (N_{i+1} - \alpha Q\ SUM(eMapTaskC(:))/\ eMapTaskC(j^*) + SUM(dMapTaskN(j))) * eMapTaskC(j^*) / \alpha Q \rfloor + 1 \qquad (2)$$

For the Map task j, the new data splits count added to its queue equals to the formula below when $j \in [0, m_i\text{-}1]$.

$$\alpha Q\ eMapTaskC(j)/\ eMapTaskC(j^*) - dMapTaskN(j)) \qquad (3)$$

Suppose the default estimated computing capacity of each new Map task is *eMapTaskC*. For the new added Map task, each is allocated an initial number of data splits in their queues. The data split number is given below:

$$\alpha Q\ eMapTaskC\ /\ eMapTaskC(j^*) \qquad (4)$$

Proof: Suppose Map task j^* has the maximum computing capacity across all the Map tasks: $eMapTaskC(j^*) > eMapTaskC(j)$ for all $j \in [0, m_i\text{-}1]$. Then the maximum data splits count allowed to be added to its queue equals to $\alpha Q - dMapTaskN(j^*))$. Proportionally compared, the maximum data split count of the j^{th} Map task queue equals to $\alpha Q\ eMapTaskC(j)/\ eMapTaskC(j^*) - dMapTaskN(j))$ and Equation (3) is proven. Therefore, aggregating all the data splits that are allocated to Map task j equals to $\alpha Q\ SUM(eMapTaskC(:))/\ eMapTaskC(j^*) - SUM(dMapTaskN(:))$. Since we assume that all the Map tasks can be finished within $\alpha Q/\ eMapTaskC(j^*)$, then given the default processing capacity of all the new Map tasks for the remaining data splits, the needed Map tasks count is calculated by dividing the remaining data split count over the expected Map task finish time and Equation (2) is therefore proven. Equation (4) is calculated by multiplying the predicted Map task execution time with the default processing capacity of each Map task. **Q.E.D**

Theorem 3: Given the condition in Theorem 1 and $\forall\, j \in [0, m_i\text{-}1],\ dMapTaskN(j) + addedDataSplit(j) < \beta^*Q$ and Suppose $dMapTaskN(0) > dMapTaskN(1) > \ldots > dMapTaskN(m_i\text{-}1)$, the Map Task set {*MapTask_0*, *MapTask_1*, …, *MapTask_k*} needs to be removed if: $\forall\, j \in [k, m_i\text{-}1],\ dMapTaskN(j) + addedDataSplit(j, k) < \beta^*Q$ and $\exists\, j \in [k+1, m_i\text{-}1],\ dMapTaskN(j)) + addedDataSplit(j, k+1) > \alpha^*Q$. After removing the Map tasks, the remaining Map task j ($j \in [k+1, m_i\text{-}1]$) adds data split count: $addedDataSplit(j, k+1)$ A more general term is defined as follows.

$$addedDataSplit(j, p) = eMapTaskC(j) * (N_{i+1} + SUM(dMapTaskN(:))) / SUM(eMapTaskC(p: m_i\text{-}1)) - dMapTaskN(j) \qquad (5)$$

Proof: A descending order of the remaining data splits leads to removing the Map task starting from the slowing one. The slower one Map task is, the more data split count that stay in its queue. We start to remove *MapTask_0* and add its queued data splits to N_{i+1}. Reallocating the total N_{i+1}+ *dMapTaskN(0)* data splits to the remaining m_i-*1* Map tasks. If the data split count of each these remaining Map task still lower than β^*Q, the procedure moves on. This procedure stops until when at least there is one Map task has its queued data split count larger than α^*Q. **Q.E.D**

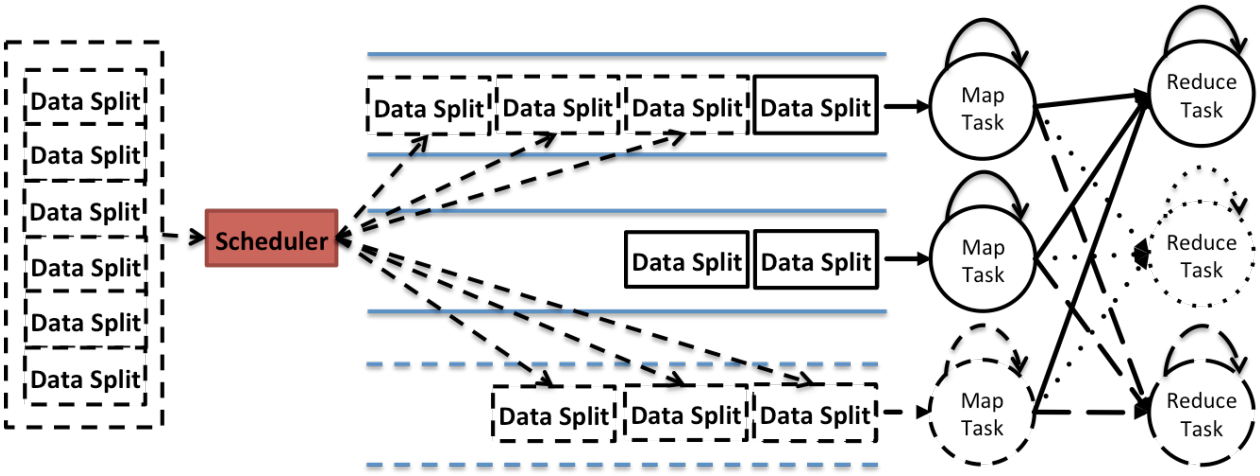


Figure 5. Demonstrates of adding an adaptive Map task. Continued from the previous example, if the input data split count is six, the scheduler adaptively launches one Map task instead of feeding all the data splits to the queues. The other three data splits are moved to the newly added Map task.

4.5 Adaptive Reduce Task Provisioning

Adaptively provision the Reduce tasks is far less straightforward than provisioning the Map tasks. Since Hadoop MapReduce is a framework primarily designed to scale the Map stage by involving embarrassingly parallel Map tasks, the Reduce tasks require network resource and an *m* to *r* data shuffling stage. In Figure 6, we identify a scaling scenario of adding a new Reduce task to the original three tasks. The new added Reduce task should have no impact on saving the network usage since all the intermediate data still have to be moved among all the compute nodes. The only difference is the degree of parallelism in the Reduce stage, that each Reduce task can process less data partitions as well as move less output data back to HDFS.

As aforementioned, the Map tasks can be added incrementally one by one. However, this doesn't necessarily guarantee best scheduling performance if Reduce tasks are added in a same strategy. This is because there is no strict demand that one input data split should go to a particular Map task. The Reduce tasks, however, only accepts their partition data in need. Adding one Reduce task would inevitable change the hash function, which accordingly leads to the partition changed.

For example, suppose the key set of the whole dataset is [0, 1, 2, …, 8]. There are three Reduce tasks R1, R2 and R3 as shown in Figure 6. The hash function is a simple modular operation, e.g. key mod 3, in this case. Therefore, R1 gets partition data whose keys are [0, 3, 6]; R2 gets partition data whose keys are [1, 4, 7]; R3 gets partition data whose keys are [2, 5, 8]. Adding one Reduce task leads to the keys are [0, 4, 8] for R1, [1, 5] for R2, [2, 6] for R3 and [3, 7] for R4. In all, there are six keys that are either moved to R4 or being exchanged among inside R1, R2, and R3. Similar conclusion applies to the case that five Reduce tasks are used. However, if the Reduce task number doubled to 6, then [0, 6] will be for R1; [1, 7], [2, 8], [3], [4], [5] are keys for R2 to R6 respectively. Then there are data associated with only three keys, [3], [4] and [5], that needs to moved.

In such a case, a workaround would replace the hash function with an enumerated list of the keys as a lookup table. For each intermediate key-value pair needs to be shuffled, the corresponding Reduce task number is searched through the list. For example, the list can be like this [R1, 0, 1, 2], [R2, 3, 4, 5], [R3, 6, 7, 8]. If a new Reduce task R4 is added, we can simply create a new entry as [R4, 2, 8], and remove the keys [2] and [8] from their corresponding list.

The downside of the workaround approach can be easily identified. The search operation might involve I/O data accessing, which is far less efficient than calculating the hash function. We can put the mapping list in memory instead if the total number of the keys are not very large.

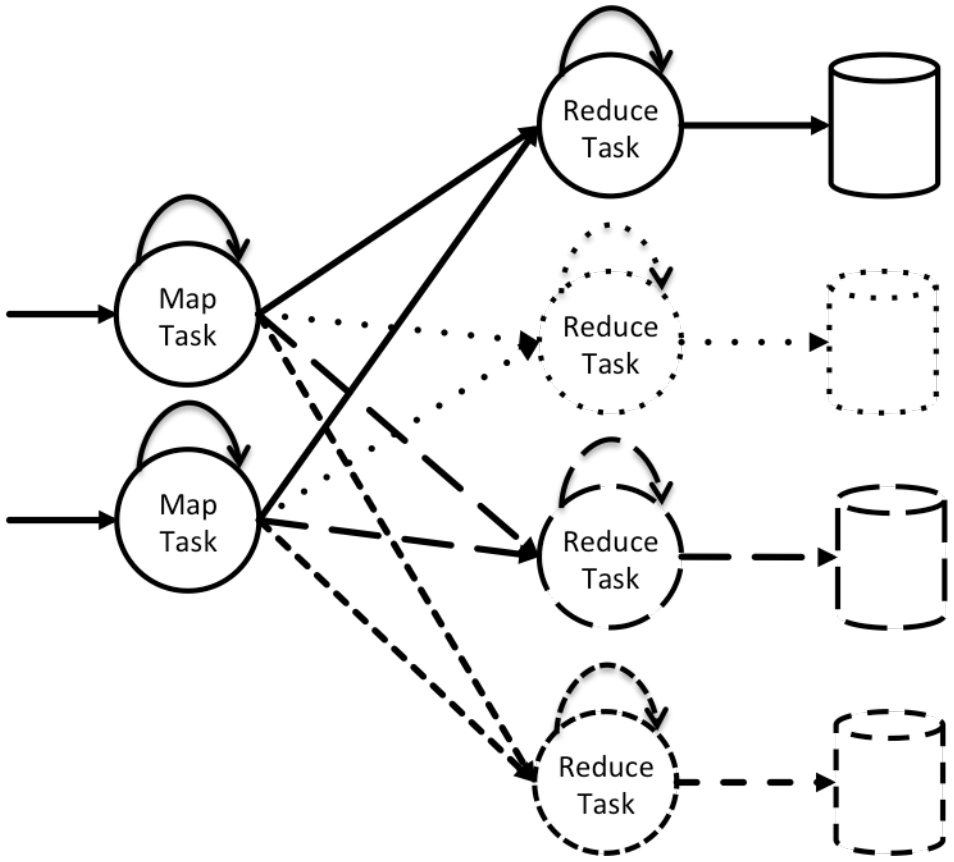


Figure 6. A graphical illustration shows one parallel Reduce task being added. This added Reduce task brings no benefit in the data shuffling stage but results to a reduced data volume to be processed/outputted for each Reduce task.

5. EXPERIMENTAL STUDIES

In this section, we first propose two methods for stream data workload prediction. After that, we show our experimental results of the prediction performance of the methods and the makespan of using these methods. Last, we report our task-level adaptive experimental results in terms of the Map and Reduce count in runtime when workload changes.

5.1 Workload Prediction Methods

For stream data applications, adaptive MapReduce task provisioning strategy should align with the workload variation. However, workloads are normally unknown in advance. In this section, we investigate two widely used prediction methods first and compare them prediction performance using real workload in the next section.

Stochastic control, or learning-based control method, is a dynamic control strategy to predict workload characteristics. There are numerous filters that can be applied. For example, smooth filter, or what we normally call as smoothing technique, predicts real-time workload by averaging the workload of a previous time span. The basic assumption here is that workload behaves reactively and not subject to significant variation in a short period of time. The average of the past one period would best represents the future workload.

There are many further improvements on the smoothing technique. For example, weighted smoothing gives higher weights to more recent workload than those that are old. The assumption here is that more recent workload would show higher impact on the real-time workload than older ones. Other prediction methods include AR method, which applies polynomial functions to approximate the workload. Among others, we want to bring forward the Kalman filter [25], also named as linear quadratic estimation, which is also widely used in workload prediction. Kalman filter works on a series of historical data stream of noise, updates and predicts future trend with statistically optimal estimations.

5.2 Experimental Settings

Simulations are carried out by using SimEvent [26], which is a software toolkit included in Matlab. The Scheduler, Map and Reduce tasks are emulated as a queuing network model, with each node being a queue. We assume that Map or Reduce tasks are running at different service capacity, thus the queue length varies during the simulation process. This justifies our needs for the task-level adaptive scheduling solution.

We apply both the Kalman Filter and the Smooth Filter to predict the workload. Comparisons are made in terms of the workload prediction accuracy as well as the makespan when applying the three approaches. Our workloads are generated based on case studies of body area network data trace [27][28]. The workload fluctuation amplitude is based on the web trace from the 1998 Soccer World Cup site [29]. This workload trace shows the average arrival rate during each minute over sixty-minute duration as shown in Figure 7(a), 7(d) and 7(g). We carefully choose three typical stream data workload types: small, moderate and heavy, for the simulation purpose. Light workload typically generates 20-60 data splits per minute. Moderate workload generates 30-150 data splits per minute while heavy workload generates 160-1180 data splits per minute.

5.3 Experimental Results

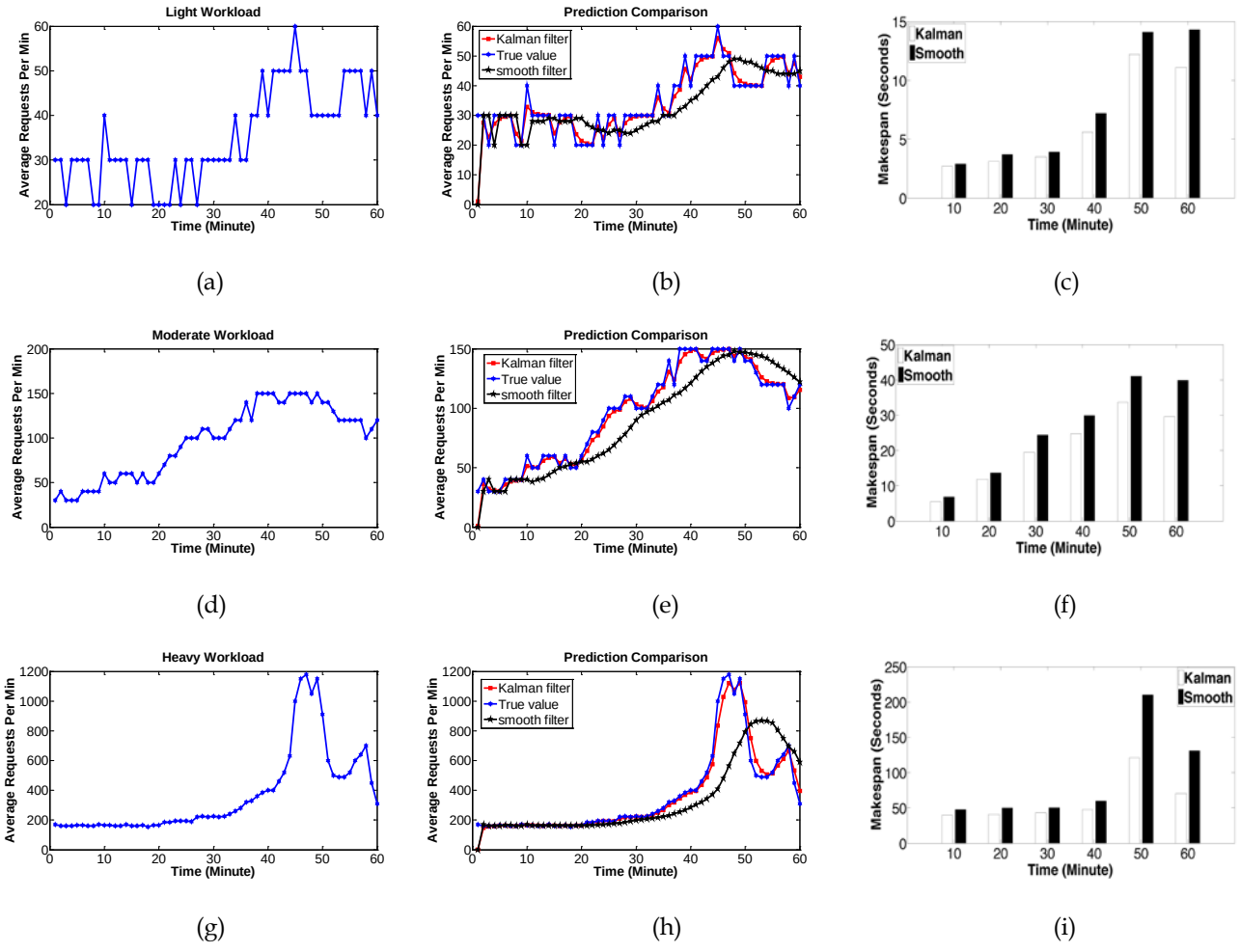
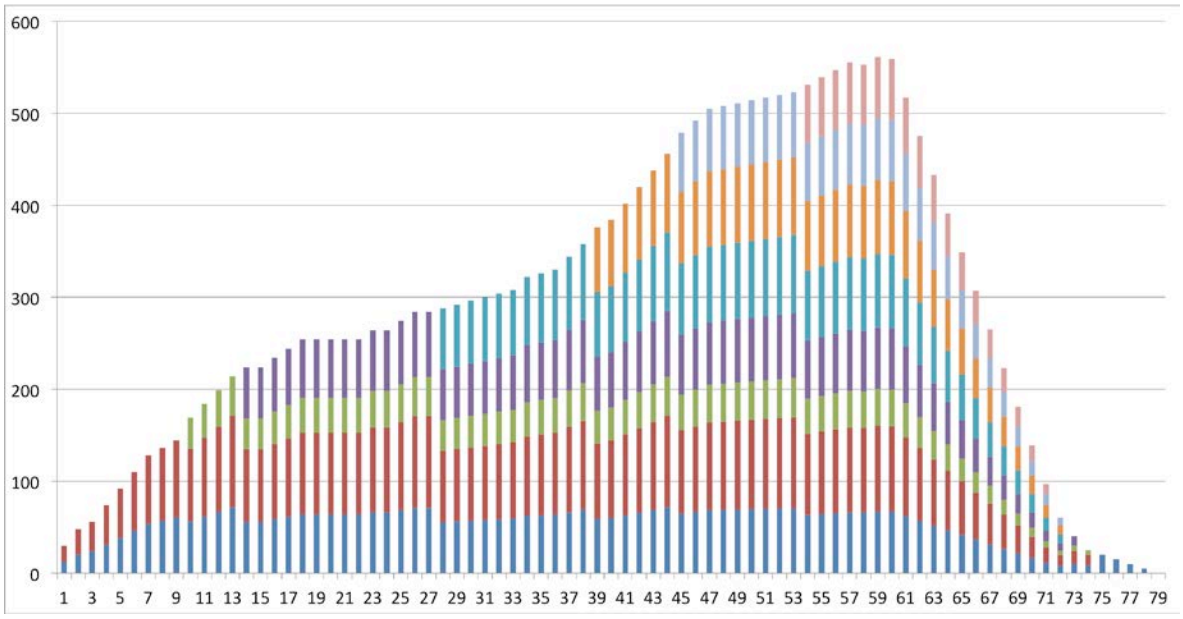


Figure 7. Comparison under three types of workloads. Figure (a)(d)(g) are light, moderate and heavy workload respectively; Figure (b)(e)(h) demonstrate the workload prediction accuracy of the two method: Smooth filter and Kalman filter; Figure (c)(f)(i) report the makespan of using the two prediction methods. Kalman filter based workload prediction performs better than the Smooth filter based prediction method.

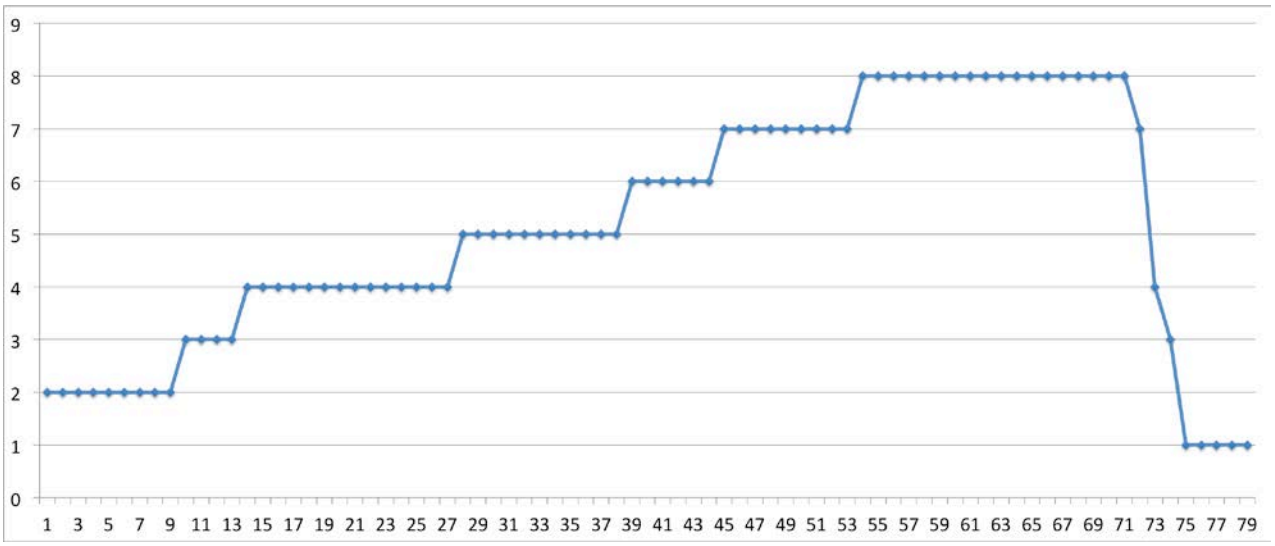
As we can see from Figure 7 (b)(e)(h), the Kalman filter shows up to 19.97% prediction error compared to 50% of the Smooth filter method in the light workload case. In the moderate workload case, the prediction errors of the two methods are 14.1% and 35% respectively. In the heavy workload case, these values are 27.2% for the Kalman filter prediction compared with 90.3% for the Smooth filter prediction. Comparing and subtracting the prediction error of the two methods over all the workload cases, the maximum margin is in the heavy workload case, which is typically 63.1% less by using the Kalman filter method.

From Figure (c)(f)(i), we can see that under the three types of workloads, the Kalman based workload prediction based method outperforms the smooth filter based method over up to 28%, 34% and 85%. All these results indicate that a good prediction method only gives a satisfactory estimation of the workload trend, but also improves the scheduling performance.



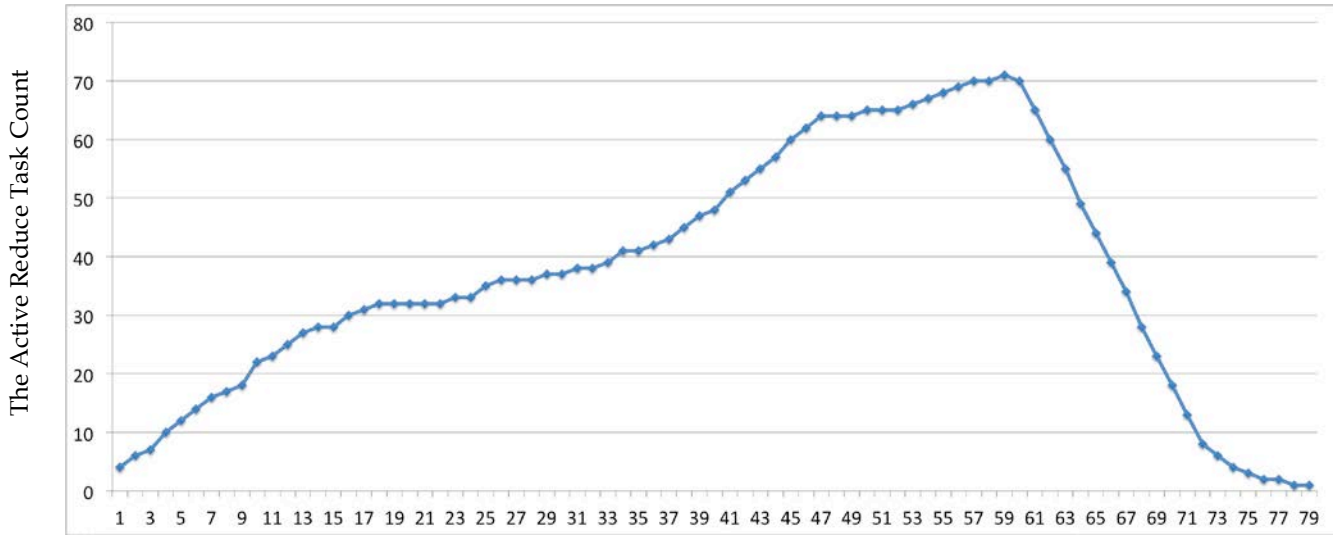
(a) Data split count for each Map task queue in each minute

The Active Map Task Count



Time(Minute)

(b) Map task count in each minute



Time(Minute)

(c) Reduce task count in each minute

Figure 8. Demonstration of the Map task queue, Map task number and Reduce task number in each minute of the light workload case. The Map and Reduce count adaptively follow the workload trend firmly.

In Figure 8, we demonstrate the scheduling effect of the task-level adaptive MapReduce framework. Figure 8(a) illustrates the data split count of each Map task queue in each minute. Correspondingly, Figure 8(b) demonstrates the number of Map task that are actively using. These Map task numbers are calculated by Theorem 2 and 3. The number of the Reduced tasks for each minute is reported in Figure 8(c). The Reduce task counts are calculated by the total data partition for all the Reduce tasks over the processing capacity.

We can see from the figures that as the workload grows, the Map task count grows accordingly and each Map task has more data splits in its queue, so do the Reduce tasks. In Figure 8(a), the rising trend becomes less when hitting the 61th minute since no more follow-up stream data splits are coming. However, it is not until the 71th minute when the Map task number starts to reduce as shown in Figure 8(b). The reason is that the data splits in each Map task is accumulating in the previous 61 minutes. Until the 71th minute the data splits of each Map task queue are sufficiently short and the Theorem 3 starts to reduce the total Map tasks.

6. CONCLUSIONS AND FUTURE WORK

In this section, we first conclude our major contributions in this work and then suggest three directions to extend this work.

6.1 Conclusions

As more and more healthcare scientific applications need to process stream data, we have designed a task-level adaptive MapReduce framework and proposed a few scheduling mechanisms to determine the active count Map and Reduce tasks in runtime. To summarize the major contribution, we have three technical aspects as follows.

(1) **A task-level adaptive MapReduce scheme**. This novel scheme implements a daemon-based mechanism to design each Map and Reduce task. The purpose is to revise the current MapReduce framework to process stream data in real-life healthcare scientific applications. We demonstrate the detail of the scheme by using real MapReduce case studies to justify the applicability of this scheme.

(2) **Runtime Map and Reduce task calculation**: We have proposed three theorems and one corollary to investigate the runtime Map and Reduce task count. These methods are used to increase and/or decrease the runtime Map and Reduce tasks as the workload fluctuates.

(3) **Adaptive MapReduce simulator**: We release the adaptive MapReduce simulator online and not only can it be used as a software toolkit for the healthcare application workload purpose, but also can be applied to other stream data application scenarios of a much wider areas.

6.2 Future Work

We suggest extending this work in the following two directions:

(1) **Extend the work theoretically towards the heterogeneous cloud systems.** Currently the adaptive scheme in this paper scales Map and Reduce tasks only. Readers would equally be interested in the scalable framework if they have more

knowledge of how the task scaling aligns with the compute node scaling. When the compute nodes are heterogeneous, the problem is more challenging but also very interesting. We need to extend the theoretical part of this work along this line.

(2) **Implement the scheme in Hadoop MapReduce applications**. This paper primarily focuses on a generic MapReduce framework, which is out of the context of the widely used Hadoop MapReduce. To be able to get it deployed on a large group of application scenarios, we need to release such a Hadoop release to help users deploy their stream data scientific applications on both public and private cloud.

(3) **Building useful tools to serve for larger virtualized cloud platform**. The simulation analyzing the optimal number of the Map and Reduce tasks in our experiment should be packaged into software toolkits in order to make it available for larger virtualized cloud platforms. Our experimental software can be tailored and prototyped toward this end.

ACKNOWLEDGEMENT

This work was supported in part by the National Nature Science Foundation of China under grant No. 61233016, by the Ministry of Science and Technology of China under National 973 Basic Research Grants No. 2011CB302505, No. 2013CB228206 and Guangdong Innovation Team Grant 201001D0104726115. The work was also partially supported by an IBM Fellowship for Fan Zhang, and by the Intellectual Ventures endowment to Tsinghua University.

REFERENCES

[1] S. Ullah, H. Higgins, B. Braem, et al. A Comprehensive Survey of Wireless Body Area Networks. Journal of Medical Systems 36(3)(2010) 1065-1094

[2] M. Chen, S. Gonzalez, A. Vasilakos, et al. Body Area Networks: A Survey. ACM/Springer Mobile Networks and Applications. 16(2)(2011) 171-193.

[3] R. Schmidt, T. Norgall, J. Mörsdorf, et al. Body Area Network BAN--a key infrastructure element for patient-centered medical applications. Biomed Tech 47(1)(2002)365–8.

[4] J. Dean and S. Ghemawat, Mapreduce: Simplified Data Processing On Large Clusters, in: Proc. of 19th ACM symp. on Operating Systems Principles, OSDI 2004, pp. 137-150.

[5] G. Malewicz, M. H. Austern, A. J. C. Bik, et al. Pregel: A System for Large-Scale Graph Processing, in: Proc. of the 2008 ACM SIGMOD international conference on Management of data, SIGMOD 2010, pp. 135-146.

[6] Y. Low, J. Gonzalez, A. Kyrola, et al, GraphLab: A New Framework for Parallel Machine Learning, in: Proc. of the 26th Conference on Uncertainty in Artificial Intelligence, UAI 2010.

[7] Y. Low, J. Gonzalez, A. Kyrola, et al, Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud, Journal Proceedings of the VLDB Endowment, 5(8)(2012), pp. 716-727.

[8] http://aws.amazon.com/elasticmapreduce/

[9] F. Zhang, M. F. Sakr, Cluster-size Scaling and MapReduce Execution Times, in: Proc. of The International Conference on Cloud Computing and Science, CloudCom 2013.

[10] R. Haux, Health information systems–past, present, future, International Journal of Medical Informatics, 75(3-4)(2006), pp. 268-281.

[11] P. L. Reichertz, Hospital information systems—Past, present, future, International Journal of Medical Informatics, 75(3-4)(2006), pp. 282-299.

[12] http://hadoop.apache.org/

[13] J. Talbot, R. M. Yoo and C. Kozyrakis, Phoenix++: modular MapReduce for shared-memory systems, in: Proc. of the second international workshop on MapReduce and its applications, MapReduce 2011, pp. 9-16.

[14] O. Christopher, C. Greg and C. Laukik, et al, Nova: Continuous Pig/Hadoop Workfows, in: Proc. of the 2011 ACM SIGMOD international conference on Management of data, SIGMOD 2011, pp. 1081-1090.

[15] C. Olston, B. Reed, U. Srivastava, et al, Pig latin: a not-so-foreign language for data processing, in: Proc. of the 2008 ACM SIGMOD international conference on Management of data, SIGMOD 2008, pp. 1099-1110.

[16] P. Bhatotia, A. Wieder and R. Rodrigues, et al, Incoop: MapReduce for incremental computations, in: Proc. of the 2nd ACM Symposium on Cloud Computing, SoCC 2011.

[17] L. Neumeyer, B. Robbins and A. Nair, et al, S4: Distributed Stream Computing Platform, in: Proc. of the International Workshop on Knowledge Discovery Using Cloud and Distributed Computing Platforms, KDCloud 10, pp. 170-177.

[18] http://storm.incubator.apache.org/

[19] http://www.scribesoft.com/

[20] J. Kreps, N. Narkhede, J. Rao et al. Kafka: A Distributed Messaging System for Log Processing. in: Proc. of 6th International Workshop on Networking Meets Databases NetDB 2011.

[21] http://flume.apache.org/index.html

[22] http://www.streambase.com/

[23] http://www.hstreaming.com/

[24] http://esper.codehaus.org/

[25] R. E. Kalman, A new approach to linear filtering and prediction problems, Journal of Basic Engineering 82(1)(1960), pp. 35–45.

[26] http://www.mathworks.com/products/simevents/

[27] C. Otto, A. Milenković, C. Sanders and E. Jovanov, System architecture of a wireless body area sensor network for ubiquitous health monitoring, 1(4)(2005), pp. 307-326

[28] E. Jovanov, A. Milenkovic, C. Otto1 and P. C de Groen, A wireless body area network of intelligent motion sensors for computer assisted physical rehabilitation, Journal of NeuroEngineering and Rehabilitation, 2(6)(2005), pp. 1-10.

[29] M. Arlitt, T. Jin, Workload characterization of the 1998 World Cup Web Site (Tech. Rep. No. HPL-1999-35R1). Palo Alto, CA: HP Labs.