Resisting Skew-accumulation for Time-stepped Applications in the Cloud via Exploiting Parallelism

Yu Zhang, Xiaofei Liao, *Member, IEEE,* Hai Jin, *Senior Member, IEEE,* and Geyong Min

Abstract—Time-stepped applications are pervasive in scientific computing domain but perform poorly in the cloud because these applications execute in discrete time-step or tick and use logical synchronization barriers at tick boundaries to ensure correctness. As a result, the accumulated computational skew and communication skew that were unsolved in each tick can slow down time-stepped applications significantly. However, the existing solutions have focused only on the skew in each tick and thus cannot resist the accumulation of skew. To fill in this gap, an efficient approach to resisting the accumulation of skew is proposed in this paper via fully exploiting parallelism among ticks. This new approach allows the user to decompose much computational part (also called asynchronous part) of the processing for an object, into several asynchronous sub-processes which are dependent on one data object. Each sub-process from different ticks can then proceed in advance using the idle time whenever the needed data object is available, redressing the negative effects caused by accumulated unsolved computational and communication skew. To efficiently support such an approach, a data-centric programming model and also a runtime system, namely *AsyTick*, coupled with an ad hoc scheduler are developed. Experimental results show that the proposed approach can improve the performance of time-stepped applications over a state-of-the-art computational skew-resistant approach up to 2.53 times.

Index Terms—Time-stepped applications, computational skew, communication skew, asynchronous execution, parallelism

1 Introduction

Many important scientific applications are organized into logical time steps or ticks. Examples of such time-stepped applications include behavioral simulations [1], [2], [3], [4], and n-body problem [5], [6], [7], [8] and also are pervasive in graph algorithms [9], [10], [11], [12], scientific computing [13], [14] and so on. Recently, these applications have again attracted much attentions because they are becoming instrumental in characterizing physical, ecological, and societal systems. For example, transportation simulations are used to predict road condition and help with transportation engineering. This proves to be very helpful in mitigating traffic congestion, which costed \$12.1 billions in 2011, and produced 56 billion pounds of carbon dioxide (CO_2) pollution [15].

Currently, in order to better understand real-world phenomena, the data sets needed to be processed by time-stepped applications become even larger scale than before. To redress this challenge, the Cloud [16] is evolving as a new platform to support such large-scale applications for its inexpensive cost and highly scalability.

However, these applications perform poorly in the Cloud due to the use of logical synchronization barriers at tick boundaries, despite the proposition of parallel dataflow frameworks including MapReduce [17], Dryad [18]. Although these applications are typically highly data parallel within each tick, the end of each tick is a logical barrier to ensure correctness. The computation in the next tick can only proceed after the end of the previous tick. As a result, the completion time of straggler in each tick and the time to send its results for the next tick dominate the execution efficiency of these applications. Unsolved computational skew and communication skew in each tick are accumulated and thus slow down these applications.

The current solutions [5], [19], [20], [21], [22] primarily aim to resist computational skew among tasks via partitioning data in consideration of their characteristics and then resist computational skew among workers via migrating the straggler or redistributing tasks. On the other hand, other solutions aim at the communication skew caused by network jitter [23], [24], [25] via fine grained synchronization and computational replication. However, for time-stepped applications subject to these approaches, unsolved communication skew caused by network jitter or unsolved computational skew in each tick is still accumulated with the increase of ticks because of barriers.

• *Yu Zhang, Xiaofei Liao and Hai Jin are with Service Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China.*
E-mail: zhang_yu9068@163.com; {hjin, xfliao}@hust.edu.cn

• *G. Min is with the Department of Computing, School of Computing, Informatics and Media, University of Bradford, Bradford BD7 1DP, U.K.*
E-mail: g.min@brad.ac.uk

In reality, for some time-stepped applications, there is amount of parallelism existing among ticks, especially for the applications with localized effect property. In these applications, much computational part, or called asynchronous part, of the processing for data objects in each tick can be asynchronously executed in advance, whenever its needed data object is available. Furthermore, for some time-stepped applications with localized effect property, such as behavioral simulation, a data object's value can only affect a few other data objects within a single tick and its value may only affect some data objects after several ticks. In other words, the process of an object in subsequent ticks may only be dependent on a few objects' information at current tick. Therefore, for these applications, sub-processes from numerous different ticks can be asynchronously executed in a parallelism way, whenever its needed data object is available. Then the idle time caused by unsolved skew can be used to execute these asynchronous parts, relieving the negative effects of skew-accumulation caused by barriers.

In this paper, we propose a data-centric programming model to exploit these asynchronous parts in each tick, resisting the accumulation of skew. It provides users several interfaces to make them easily express what parts can be asynchronously executed and also ensure the correctness of applications at the same time. Then the asynchronous part of the processing for an object is decomposed into several sub-processes. Whenever a data object needed by such a sub-process is available, this sub-process of subsequent tick can be executed in advance via utilizing the idle time of workers caused by unsolved skew in current tick, resisting the accumulation of negative effects caused by skew in each tick. When the information of all its needed data objects is available and processed, the synchronous part of the processing for this object will be executed and immediately outputs its current state information to asynchronous sub-process of related objects for the next tick to process, ensuring the correctness.

Experimental results show that the computational imbalance degree and communication imbalance degree of current solutions are even up to 3.3 and 0.61, respectively. The ratio of asynchronous part is more than 85.7%, and our approach can improve the performance of time-stepped applications with a state-of-the-art approach up to 2.53 times via resisting the accumulation of its unsolved skew.

The major contributions of this paper include:

1) An efficient approach is proposed for time-stepped applications to exploit asynchronous parts and achieve more parallelism, redressing the negative effects caused by accumulated unsolved computational and communication skew.
2) We propose a data-centric programming model allowing users to easily express asynchronous parts and also implement a runtime system coupled with an ad hoc scheduler for time-stepped applications to support the efficient execution of asynchronous parts.
3) This study reveals quantatively how much unsolved computational skew and communication skew exist in time-stepped applications subject to the current solutions in the Cloud. Then it presents the ratio of asynchronous parts in each benchmarks and also shows how much unsolved skew can be eliminated by our proposed approach, followed by thorough performance comparison against state-of-the-art solutions.

The rest of this paper is organized as follows: Section 2 presents a survey of the related work. Section 3 illustrates the motivation, main ideas and challenges of our approach. The implementation details of this approach are described in Section 4, followed by the thorough analysis of experimental results in Section 5. Finally, the paper is concluded in Section 6.

2 Related work

Logical barriers that are employed to ensure the correctness of time-stepped applications make their performance suffer from computational skew and communication skew. Currently, many solutions have been proposed to eliminate computational skew and communication skew respectively.

Solutions for computational skew: Numerous approaches have been proposed to balance computational skew. For example, PowerGraph [21] tried to tackle the challenges of highly skewed power-law graphs for distributed graph applications, via partitioning and processing edges for each vertex over machines. SkewReduce [5] aimed at the scientific applications, where different partitions take vastly different amount of time to run even if their input datasets have the same size. SkewReduce proposed to determine how to partition the input data with user defined cost function to minimize the impact of computational skew for these applications.

When some nodes in the cluster become idle, SkewTune [20] identified the task with the greatest expected remaining processing time for MapReduce, then proactively repartitioned unprocessed input data of this straggling task to fully utilize the nodes in the cluster. Pearce *et al.* [26] proposed a load model for load balance algorithms based on application elements and their interactions to guide the selection of load balance algorithms.

Lifflander *et al.* [22] proposed persistence-based load balancers to redistribute the work to be performed in a given iteration based on measured performance profiles from previous iterations. They also presented retentive work stealing for applications with significant load imbalance within a phase, or applications with workloads that cannot be easily profiled. Ananthanarayanan *et al.* [19] proposed a mantri system, which classifies the reason of the skew for MapReduce task into three classes. Then this system monitors and analyses these reasons in advance, and takes related actions to reduce the negative effects caused by these reasons respectively via restarting, migrating and replication.

Solutions for communication skew: Significant efforts have been made in the HPC community to optimize communication for time-stepped applications [27], [28], [29]. However, these optimization techniques were developed using a model of fixed and unavoidable latency for sending a message across a dedicated network, but not for the unstable and unpredictable latency that characterizes the Cloud. Asynchronous communication primitives facilitate communication hiding, and many bulk synchronous applications use these primitives to overlap computation and communication. These optimizations work best when communication latency is uniform and predictable, and it is difficult in practice to characterize their effectiveness [30].

Some studies [31] proposed to avoid communication at the expense of performing some redundant computation. While communicating less often certainly helps, this technique alone cannot deal with latency spikes. Furthermore, it can only be applied to applications whose computational logic can be formulated as a sparse linear algebra problem. This specialization significantly impairs the productivity of scientists who want to develop new applications without regard of which optimizations to use for communication.

Zou *et al.* [23] tried to tolerate network jitter via fine grained synchronization and computational replication, which replaces global barriers with local synchronization, and also introduces the idea of computational replication to get needed data and completes the current tick in the absence of incoming messages. However, it still needs much synchronization between iterations and causes serious accumulation of skew.

Speculative execution: For the straggler problem in BSP model, some works [32], [33] propose to support it based on speculative execution. However, they either need to give up the computation which is based on stale value or just are best-effort approaches, in which its results may have error. For example, FastTrack [32] needs to give up the computation which is based on stale value. Thus, for large-scale algorithms, it may cause significant runtime overhead. The best-effort approaches [33] are calculating approximate results and are only suitable to the algorithms accepting approximate results. Unfortunately, the approximately results are useless for time-stepped applications, such as behavioral simulation. Our proposed approach is not based on speculative execution. The results of our sub-processes are bound to be used for the calculation of the final results of a data object, although it is only a partial result.

Compared with current solutions: In summary, the current solutions only focus on computational skew and communication skew in each tick. On one hand, they partition data in consideration of their characteristics to resist skew among tasks and migrate the straggler or redistributing tasks to resist skew among workers. On the other hand, they resist communication skew via fine grained synchronization, computational replication and so on. However, for time-stepped applications with these approaches, unsolved computational skew or unsolved communication skew in each tick will be accumulated with the increase of ticks because of barriers. To address this problem, this paper proposes a data-centric programming paradigm for time-stepped applications to exploit asynchronous parts and get more parallelism, resisting the accumulation of skew.

3 The Proposed Approach

This section firstly describes the benefits of extracting and asynchronously executing asynchronous part to resist the accumulation of skew via an example. Other time-stepped applications also have the similar property and thus can get such benefits in the same way. We then present the main ideas and challenges of the proposed method. The goal of this paper is to enable scientists to easily express and efficiently execute those asynchronous parts for each tick, resisting the accumulation of negative effects caused by unsolved skew in each tick.

3.1 Motivation

In this part, we take fish school simulation as an example to show the benefits of extracting and asynchronously executing the asynchronous part with regard to resisting skew-accumulation.

Fish school simulation is employed by Couzin *et al.* [1] to study information transfer in schools of fish. Within a tick, each fish agent inspects the current velocities of other visible fish to determine its new velocity for the next tick, where two parameters V (visibility) and R (reachability) determine how far a fish can see or move within a tick. In addition, informed individuals balance these social interactions with a preferred direction (e.g., a food source) to determine movement.

The processing of a tick is data parallel as in Algorithm 1. Each worker executes the tick logic for each fish agent independently, calculating its new state round by round, where the processing of a fish requires access to the state of all neighbor fish within its context C, which is a set of fish in a scope specified by parameter V. Note that Algorithm 1 just describes the fish school simulation with a fish, to simplify the description of benefits. In practice, objects are processed in blocks.

However, as described in Algorithm 1, the processing for each fish can be executed only when the

Algorithm 1 The fish school simulation algorithm

```
Require: Fish $f$, State $C$
Ensure: $f$ //Information of fish $f$ for the next tick.
1: for each fish $g$ in context $C$ do//where $g$ is visible to $f$
2:     ...//Computes influence of $g$ to $f$.
3: end for
4: ...//Computes the preferred direction for fish $f$.
5: return $f$
```

Algorithm 2 Asynchronous fish school simulation algorithm

```
1: #define Begin $Ad$
2: #define End($f$) if(Barrier($f$)) Jump //If a fish in context of fish $f$ is not processed, it jumps to the address $Ad$ described as in Line 1.
Require: Fish $f$
Ensure: $f$ //Information of fish $f$ for the next tick.
3: Begin
4: Fish $g \leftarrow$ Get($f$)
5: ...//Computes the partial contribution value PartialState$_{new}$ of $g$ to $f$.
6: Update($f$, PartialState$_{new}$)
7: End($f$)
8: ...//Computes the preferred direction for fish $f$.
9: Set($f$, $value$)
10: //Spreads the new calculated information of fish $f$ to related fish for the processing of the next tick.
11: Diffuse($f$).
```

information of all fish at the previous tick is available. If the information of a fish is unavailable because of computational skew or communication skew in current tick, the execution of this function in subsequent ticks needs to wait, and the processing of fish f for subsequent ticks can not proceed. Then the execution of subsequent ticks is delayed. Consequently, the negative effect of skew caused in each tick is accumulated.

In reality, the mobility of a fish is limited and may only affect some fish's behavior after several ticks. In other words, the process of a fish in subsequent ticks may only be dependent on a few fish's information at current tick. Therefore, we can consider whether this accumulation of unsolved skew can be redressed by exploiting the idle time of workers to asynchronously execute some computation of its subsequent ticks in advance. For example, the code from Line 1 to Line 3 of Algorithm 1 can be asynchronously executed as in Algorithm 2. In this way, the asynchronous part (from Line 3 to Line 7) at subsequent ticks can be executed in advance whenever the information of a fish in context C is available. Moreover, as shown in Section 5, its asynchronous part is up to 85.7%, showing the fact that much computation in subsequent ticks can be executed in advance, compensating the negative effects of unsolved skew in current tick and resisting the accumulation of skew.

3.2 Main ideas

As discussed in Section 3.1, due to logical barriers, the performance of time-stepped applications suffers from the accumulation of computational skew and communication skew unsolved by current skew-resistant approaches. Fortunately, this problem can be tackled via executing asynchronous part of subsequent ticks in advance as discussed in Section 3.1.

Consequently, we propose an efficient approach for time-stepped applications to express asynchronous parts and fully exploit its inherent parallelism, redressing the negative effects caused by accumulated unsolved computational and communication skew. This approach then allows the user to decompose the asynchronous part of the processing for an object into several asynchronous sub-processes. The idle time caused by unsolved skew in each tick can be employed to execute asynchronous sub-processes at subsequent ticks, whenever the required data object is available.

In this way, the negative effects of unsolved computational skew or communication skew in the current tick can be redressed via executing asynchronous part of its subsequent ticks in advance. When the information of all data objects required for an object is available, the remaining synchronous part (e.g. Line 8 to Line 11 as in Algorithm 2) will be executed for this object and immediately output its current state information to asynchronous sub-processes of related objects for the next tick, ensuring the correctness.

3.3 Challenges and solutions

As shown in the above discussion, to ensure the correctness of our approach, synchronous part of objects can be executed only when the information of all the needed objects is available. In this section, we show how to make an object efficiently know when its synchronous part can be executed.

Although an object does not know exactly what objects are needed by it, fortunately, it can easily know the objects of which partition are needed through location and given application parameters. Take the fish school simulation as the example, a fish can easily know these partitions via the parameters R and V, because the scope where a fish can move or see is limited, and a fish only interacts with fish in this scope. So, our approach uses the completion condition of these partitions to determine whether the synchronous part of an object can be executed. Note that in our approach all objects are range-partitioned across nodes just to leverage the better data locality and reduce communication cost, because each object only interacts with the objects in its context C, which is a set of objects specified by users.

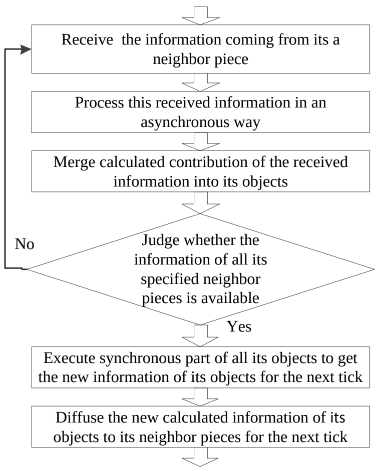

Receive the information coming from its a neighbor piece
Process this received information in an asynchronous way
Merge calculated contribution of the received information into its objects
No
Judge whether the information of all its specified neighbor pieces is available
Yes
Execute synchronous part of all its objects to get the new information of its objects for the next tick
Diffuse the new calculated information of its objects to its neighbor pieces for the next tick

Fig. 1. Execution progress overview of a tick for a piece.

However, in this way, an object may wait for the end of the processing for many unnecessary objects in its neighbor partitions. Moreover, objects even in the same partition also need an individual notification respectively, inducing strict synchronization and much communication cost.

To increase parallelism and reduce communication cost for the above method, we further divide the partition into several pieces for each worker, and make them synchronize with piece as unit. That is to say, it divides the partition processed in each worker into several pieces. When all objects in a piece are processed, this piece notifies all pieces that are dependent on it. Then all asynchronous parts of objects in these notified pieces can be executed in advance. Note that the piece itself is also taken as an exotic piece and notifies itself to calculate its own contribution to its objects for the next tick. After all needed pieces are processed for a piece, all synchronous parts of those objects in this piece can be executed, and output the results to related pieces for the next tick to process as above discussed again. The execution progress overview of a tick is described in Fig. 1.

Take the fish school simulation as the example, a partition for a worker is further divided into many pieces. Each piece tries to get information of its neighbor pieces, and then processes related asynchronous parts of all its fish together whenever a needed piece is available. Note that the information of fish in each piece is always available to this piece itself for processing of the current tick. After the information of all its needed pieces is available and processed, synchronous part of all its fish is processed together and outputs the information of all its fish to its neighbor pieces for the next tick to process.

4 Implementation

This section presents the implementation details of the approach discussed above. It mainly presents a data-centric programming model and also develops an efficient runtime system to make the asynchronous part of time-stepped applications easily expressed and efficiently executed.

TABLE 1
Programming interface

Interface
Get(piece P**)** Get the object information of a piece needed by piece P.
Update(object O, $state$**)** Merge the new calculated partial state into object O. Its results are stored in an intermediate value list, described in the next part.
Diffuse(piece P**)** Spread the information of objects in piece P to its neighbor pieces for the next tick.
Barrier(piece P**)** Check whether the information of all pieces needed by piece P are available.
Set(object O_1, **object** O_2**)** Set the value of O_1 with the value of O_2.
Termination() Check whether the user specified termination condition is met after each tick.
Divide(space S**)** Divide all objects in simulation space S into several sized partitions. Then all these partitions are automatically assigned to different workers to process.
Partition(partition P**)** It is used to further divide partition P into several equal sized pieces for each worker.

4.1 Programming model

This part describes the data-centric programming model that is used to easily express and extract the asynchronous part.

In each tick, the processing of each piece, in reality, can be mainly abstracted as follows: 1) it asynchronously receives and processes the information coming from its neighbor pieces to get the contribution of the received information to the final results of this piece, then merges this calculated contribution into its objects; 2) Judge whether all the needed information of its neighbor pieces is available; 3) When all the information is available, it executes synchronous parts of all its objects to get the new information of its objects for the next tick, and then diffuses its new final information to its neighbor pieces for the next tick to process. Otherwise, it goes back to Step 1 again for processing of the remained neighbor pieces in the future.

So, the processing of each piece needs Get() operation to asynchronously get the information of its a neighbors piece. Then it processes this received information and calculates the partial contribution of this received information to objects in its piece. Later, it uses Update() operation to asynchronously merge the new calculated partial contribution into objects in its piece. Finally, Diffuse() is used to spread the new information of this piece to its neighbor pieces for the next tick.

```
Asynchronous execution()
{
    Begin
    get() //Gets the information of its a neighbor piece
    ··· //Processes the received information
   update() // Merges the calculated value into its objects
    End //The code between Begin and End is the asynchronous part
   ··· // Executes the remaining synchronous part of all its objects
   // Diffuses the new information of all its objects for the next tick
   Diffuse()
}
```

Fig. 2. A general example of employing the proposed programming model to process a piece in a tick.

Those functions that require application developers to instantiate are summarized in Table 1. Fig. 2 gives a general example to employ such a programming model. The asynchronous part is abstracted by the block between macro $Begin$ and End. The Barrier() in macro End is used to determine whether the code needs to go back to $Begin$. When the information of needed pieces is not all available, macro End will make the executing code go back to $Begin$, to process the information of the unprocessed pieces that will be received in the future. Otherwise, the code following End of this piece can be executed. In reality, the code following macro End is the code that cannot be asynchronously executed or called synchronous parts.

Now, we show how to employ such a programming model to express the above discussed fish school simulation. The details are described as in Algorithm 3. From this algorithm, we can observe that the code from Line 3 to Line 11 is the asynchronous part. It can be executed whenever its needed information is available. Moreover, the asynchronous part of the next tick can be executed in advance via employing the idle time caused by unsolved skew in the current tick. However, because the remained code following Line 11 needs that the information of all fish is available, the negative effects caused by this synchronous part still exist. Fortunately, the asynchronous part occupies most of the total execution time of each tick as discussed in the following section. Consequently, our approach can efficiently resist the accumulation of unsolved skew in each tick.

4.2 AsyTick

To efficiently support the above discussed programming model and the execution of asynchronous part, a running system, namely AsyTick, is presented in this part. AsyTick also provides an efficient scheduler based on the characteristics of time-stepped applications in order to further reduce the negative effects caused by the accumulation of unsolved skew.

4.2.1 Architecture

AsyTick framework contains a master and multiple workers. Its architecture is presented as in Fig. 3. The master coordinates the workers and monitors the status of workers, each of which contains and schedules several user tasks (user code) to process different pieces, respectively. Moreover, these user tasks are executed in threads and communicate with others via passing messages. Those tasks in the same worker communicate with each other through logical messages, which is simulated by the shared memory of this worker.

Logical messages are used to communicate so as to make the execution of tasks message-driven and enable an object to know whether all objects on which it depends on have been processed. This is because it is different from periodical information pulling that causes much delay even if the information is already available, and also is unlike arbitrarily information pulling that induces frequent communication and much overhead. In order to reduce its communication overhead, AsyTick also aggregates those messages that are sent to the same node.

AsyTick works as follows. At the beginning, each worker loads a subset of data objects into memory, and makes them shared by its tasks. These data objects are grouped into units of pieces, according to their locality. That is to say, all data objects in the same piece are stored in the same piece element. All pieces in the same worker are maintained in a local in-memory key-value store, namely *state table*.

Algorithm 3 Refined asynchronous fish school simulation algorithm

```
 1: #define Begin $Ad$
 2: #define End($G$) if(Barrier($G$)) Jump //If a piece in
    context of group $G$ is not completed, it jumps to
    the address $Ad$ described as in line 1.
Require: FishGroup $G$ //$G$ contains the information
    of all fish in a piece.
Ensure: FishGroup $G$
 3: Begin
 4: FishGroup $G_{get}$ ← Get($G$)
 5: for each fish $g$ in $G_{get}$ do
 6:     for each fish $f$ in $G$, and $g$ is visible to $f$ do
 7:         ...//Computes the partial contribution
    value of $g$ to fish $f$.
 8:         Update($f$, PartialState$_{new}$)
 9:     end for
10: end for
11: End($G$)
12: for each fish $f$ in $G$ do
13:     ...//Computes the preferred direction for fish
    $f$.
14:     Set($f$, $value$)
15: end for
16: //Spreads the information of each fish in $G$ to
    related pieces for the processing of the next tick.
17: Diffuse($G$)
```

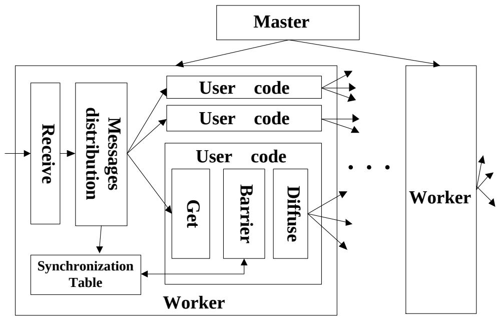


Fig. 3. AsyTick Architecture Overview.

Each entry of the state table corresponds to a piece indexed by its key. And each table entry contains three fields. The first field stores the key value j of a piece; the second field stores its tick number; the third field stores a value list of contained data objects for this tick and a related intermediate value list for each object. This intermediate value list contains several values needed by the synchronous parts of each object. Note that these intermediate values are calculated by those asynchronous sub-processes, and will finally be used to calculate the new value of each object for the next tick by the synchronous parts.

Whenever a message is received by a worker, it picks up a piece of data objects recorded in this message, uses *Messages distribution* Module to redistribute the information of these data objects to related user tasks and then immediately trigger the execution of these tasks. When a task is triggered, it immediately processes its received data objects and merges calculated contribution of these data objects' information into the intermediate value list of related data objects. Later, the function Barrier() in macro End judges whether all needed pieces are available for the piece processed by this task.

To know when all needed pieces are available for a piece, the synchronization table is employed to record which pieces are already available. All this information can be gained according to the message received by this worker. When a message is received by a worker, its *Messages distribution* Module makes the synchronization table record the key value of all those pieces contained in this message. Then, these records are employed by Barrier() in each user task to determine whether all its needed pieces are processed.

After all needed pieces are available for a piece, the synchronous part of all objects in this piece can be executed. It calculates the new value of objects for the next tick based on the value recorded in the intermediate value list. At the same time, for some applications, some data objects also need to be inserted into related piece for the next tick. This is because some objects of a piece may move to another piece at the new tick for some applications, *e.g.* fish school simulation. Then the process of this piece for the current tick ends.

TABLE 2
The summary of notations

Notation	Description
P	A piece containing many objects.
Pri(P)	The priority of piece P.
$D(P)$	The shortest distance of piece P to one boundary of the partition containing it. This value is calculated once only, when runtime initially divides partition into pieces for each worker.
$N(P)$	The tick number of piece P.

4.2.2 Scheduler

AsyTick also provides an ad hoc scheduler to efficiently support the execution of asynchronous part and to further reduce the negative effects caused by accumulation of skew, based on the characteristics of time-stepped applications.

In AsyTick, each worker owns a scheduler, which schedules the asynchronous part processing order of all its pieces. It realizes this scheduling via assigning the processing order of pieces that are waited by asynchronous parts. If a worker needs the asynchronous part of a piece to be executed, it just needs to make the information of its needed piece available via *Messages distribution* Module. Then it can wake up the thread, which is executing the asynchronous part of this piece. The details of priority definition are described as follows.

Firstly, we present what factors should be in the consideration of the priority definition. Because time-stepped applications have strong data locality, each object only interacts with its nearby objects. The processing of each object in time-stepped applications always only needs the information of its neighbor objects. Then it can parallelize the processing of objects inside this partition and more objects in its neighbor partitions for the next tick via firstly processing the piece that is the nearest one to boundaries of this partition. Then more workers can process objects of their own partition in parallel, further redressing skew accumulation. Consequently, the scheduler can employ the distance between the centre of piece and boundaries of partition as a factor of the priority.

Because some objects of different ticks may need to be processed on the same worker, the scheduler should firstly schedule the objects of the earlier ticks, and make the partition of the earlier ticks completed quicker, reducing the waiting time of the processing for its subsequent ticks.

Now, we present how to define the priority for each piece. From the above discussion, we can find that the priority for each piece is mainly related to the following two factors: 1) its distance to the boundaries of the partition containing it; 2) its tick number. Consequently, we can define the priority Pri(P) of piece P as follows, and all notations are summarized in Table 2.

$$\begin{cases} \mathsf{Pri}(\mathcal{P}) & = & \alpha \times \mathcal{D}(\mathcal{P}) + \beta \times \mathsf{N}(\mathcal{P}) \\ \alpha + \beta & = & 1 \end{cases}, \quad (1)$$

where α and β are constant value to adjust the proportion of $D(P)$ and $N(P)$ in Equation (1). Note that because $D(P)$ of some time-stepped applications, such as PageRank, is difficult to be calculated, it is selected as a part of priority for scheduler, thus α may be assigned with the value of 0. Obviously, the value $N(P)$ is more important than $D(P)$, and β should be set larger than α. In this way, it can firstly process the piece with the earliest tick. If there are several pieces of the same tick in a partition processed by a worker, this worker tries to firstly process the piece that is the nearest one to boundaries of the partition owning this piece.

5 Experimental results and Evaluation

In this section, we first investigate how much unsolved computational skew and communication skew still exist in time-stepped applications subject to the current solutions, and then show how much such unsolved skew can be eliminated by the proposed AsyTick. Finally, the performance of AsyTick is evaluated thoroughly, followed by the investigation of the impact of data size and the number of pieces on its performance.

Platform and benchmarks: The hardware platform used in our experiments is a Cluster with 256 cores residing on 16 nodes, while the network interconnection is a 2 Gigabit Ethernet. Each node is a 2-way octuple-core with Intel(R) Xeon(R) CPU E5-2670 at 2.60 GHz CPUs and 64 GB memory. Each node has 16 cores and thus a maximum of 16 workers are spawned for each node to run applications. In order to evaluate the system performance, four typical benchmarks described as follows are implemented:

1) *Fish school simulation (*FS*).* It is a typical example of behavioral simulation applications, which model complex systems of individual, intelligent agents, such as transportation networks and animal swarms, to better understand real-world phenomena. Its details are described in Section 3.1.
2) *PageRank.* It is a popular algorithm initially proposed for ranking web pages. Later on, this algorithm has found a wide range of applications, such as link prediction. During each time-step, each worker updates the ranking score for a page based on ranking score of other pages being linked to it.
3) *N-body.* It simulates the dynamics of a set of interactional particles over many discrete time-steps, where particles apart further than a threshold $distance(r)$ are assumed to have no effects on each other. During each time-step, each worker updates a particle's velocity and position based on its current velocity and the positions of other particles within r distance away.
4) *Jacobi method.* It is a popular scientific computing algorithm to solve a large system of linear equations and eigenvalue problems. In this algorithm, each diagonal element is solved with an approximate value, which then used for the next iteration until it converges.

TABLE 3
Data sets summary

Benchmarks	Data Sets
Fish school simulation	# Fish: 5 billions
PageRank	# Nodes: 4,847,571; # Edges: 68,993,773
N-body	# Points: 500 millions
Jacobi method	# Rows: 50,000; # Columns: 50,000

The data sets used for these algorithms are described in Table 3. The data sets for PageRank algorithm are real graphs data downloaded from website [34].

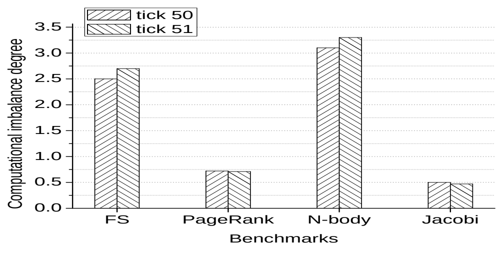
Performance metrics: The performance evaluation mainly uses the following metrics.

1) *Computational imbalance degree* λ_L: The computational imbalance degree
$$\lambda_L = L_{max}/L_{avg} - 1 \quad (2)$$
is adopted to evaluate the computational skew, where L_{max} is the maximum computation time on any processor and L_{avg} is the mean computation time over all processors.
2) *Communication imbalance degree* λ_C: The communication imbalance degree
$$\lambda_C = C_{max}/C_{avg} - 1 \quad (3)$$
is used to evaluate the communication skew, where C_{max} and C_{avg} are the maximal and mean communication time wasted on diffusing the result of current tick to the next tick, respectively.

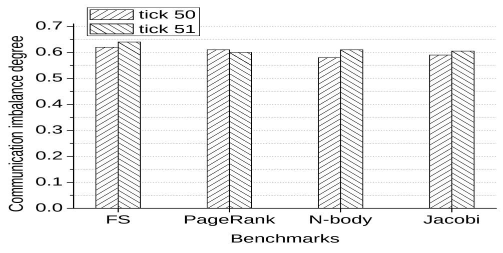
The schemes to be compared: To demonstrate the efficiency of our approach, we realize three other systems in order to compare their performance.

1) *OriTick*, which is modified by Piccolo [35] and does not employ any method to resist skew.
2) *ComTick*, leveraging persistence-based load balancer [22] on OriTick.
3) *AsyTick-PLB*, leveraging persistence-based load balancer on AsyTick.

Persistence-based load balancer is state-of-the-art balancer for iterative applications. It redistributes tasks based on the performance profiled from previous ticks. The difference between AsyTick-PLB and ComTick is whether or not adopting our approach. Note that our approach is proposed to resist the accumulation of



(a) Computational skew for ComTick



(b) Communication skew for ComTick

Fig. 4. Computational and communication skew for ComTick.

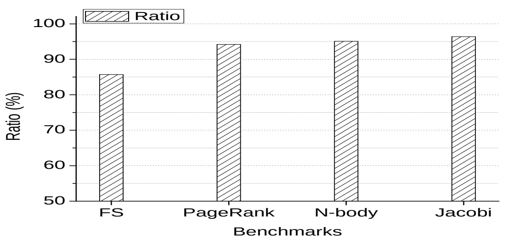


Fig. 5. Ratio of asynchronous part in a tick.

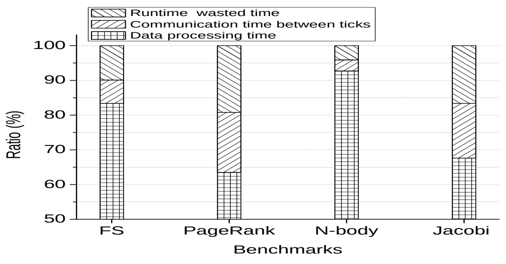


Fig. 6. Execution time breakdown of AsyTick.

skew, and also can work together with all the current skew-resist solutions. So, we just test its performance with the persistence-based load balancer.

5.1 Computational skew and communication skew for current solutions

To show the necessity of resisting the accumulation of skew, we firstly test the condition of computational skew and communication skew for the above four benchmarks with ComTick.

Fig. 4(a) and Fig. 4(b) respectively show the computational and communication skew situation for successive two ticks of the above four benchmarks with ComTick. From these two figures, we can observe that there is much unsolved computational and communication skew in each tick. The computational imbalance degree of n-body benchmark with ComTick is even up to 3.3 at tick 51. The communication imbalance degree of n-body benchmark with ComTick is also up to 0.61 at tick 51 for network jitter. They are two reasons may induce so much high computational skew and serious network jitter. First, it is difficult to profile the workloads of some time-stepped applications, such as behavioral simulations. A distinguishing feature of behavioral simulations is their frequent and high-volume group migration, the phenomenon in which simulated objects traverse domains in groups at massive scale in each tick. This results in continual and significant load imbalance among tasks, which are difficult to be profiled in advance. Second, many factors, such as multi-tenancy, imbalance service utilization, hardware variability, imbalance among tasks of the same applications, all may result in serious network jitter and significant variation of the load imbalance statuses in the cloud.

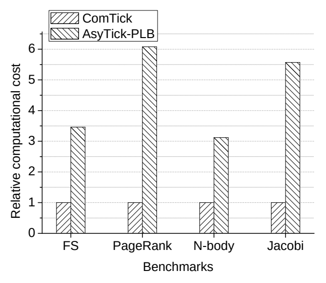
Moreover, from Fig. 4(a), we can also observe that the computational imbalance degree even increases with ticks for ComTick. Take the n-body benchmark as an example, the computational imbalance degree of ComTick is only 3.1 at tick 50, yet becomes 3.3 at tick 51. Based on the detailed analysis, we find that this is because simulated objects in n-body benchmark frequently migrate in the simulated space in massive volume. The frequent and massive load change makes ComTick invalid for such applications like n-body.

5.2 Ratio of asynchronous part and runtime overhead

In this part, we firstly show the ratio of asynchronous part execution time against the total execution time of a tick. Later, the runtime breakdown of our approach is presented, followed by the computational and communication overhead of our approach.

Fig. 5 shows the execution time ratio of asynchronous part in a tick against the total execution time of this tick on the above four benchmarks. Note that this execution time does not include the time to transfer intermediate results between ticks. From Fig. 5, we can observe that the ratio of asynchronous part on the above four benchmarks are all more than 85.7%. For the Jacobi method, the ratio of asynchronous part is even up to 96.4%, which means that much idle time caused by unsolved skew of previous ticks can be employed to asynchronously execute these asynchronous parts in advance. Then the negative effects caused by the accumulation of unsolved skew can be redressed.

Fig. 6 shows the execution time breakdown of AsyTick. We can observe from this figure that the maximum time wasted by our approach only occupies 19.2% of the total execution time. For n-body


(a) Relative computational overhead

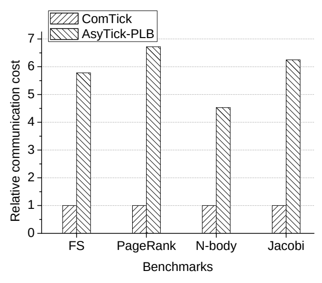

(b) Relative communication overhead

Fig. 7. Runtime overhead of AsyTick-PLB against ComTick.

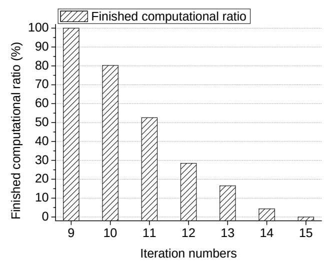


Fig. 8. The completion situation of subsequent ticks for the FS.

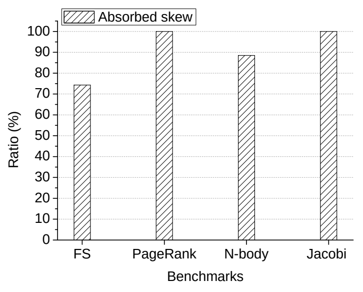


Fig. 9. The ratio of ComTick's unsolved skew absorbed by AsyTick-PLB on each benchmark.

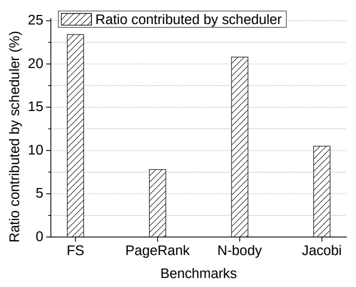


Fig. 10. The impact of scheduling algorithm to the skew-absorbing ability of AsyTick.

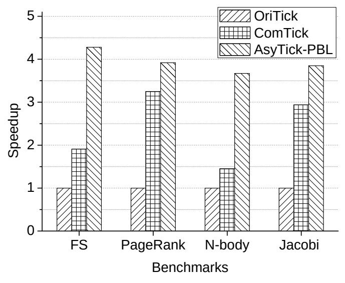


Fig. 11. The speedup of AsyTick-PLB and ComTick against OriTick on each benchmark.

benchmark, the time wasted in our approach even occupies only 4.1% of the total execution time of a tick. Moreover, the time to process data objects and the time to communicate intermediate results between ticks are up to 92.7% and 3.2%, respectively.

Fig. 7(a) presents the relative computational overhead of AsyTick-PLB against ComTick. It can be seen that the relative computational overhead of AsyTick-PLB against ComTick is up to 6.08. Fortunately, as shown in Fig. 6, the time wasted in our approach is negligible against the total execution time. Moreover, as the discussion below, the benefits gained from resisting the accumulation of unsolved skew are much more than the costed runtime overhead.

Fig. 7(b) presents the relative communication overhead of AsyTick-PLB against ComTick. Although AsyTick-PLB needs the intermediate results of the current tick to be immediately transferred to the related worker and makes this worker process them for the next tick, we can observe that the communication cost of AsyTick-PLB is less than 6.72 times higher than the communication overhead of ComTick. This is because AsyTick-PLB can gather and compress many messages. Note that we just test the computational overhead and communication overhead of AsyTick-PLB, not including the storage overhead. This is because AsyTick-PLB can immediately process intermediate results whenever receiving it. Then there is no necessity to record it. So, the storage cost of AsyTick-PLB is almost the same as ComTick, although AsyTick-PLB needs a synchronization table.

5.3 Performance Comparison

In this part, we firstly test how much computation can be executed in advance and then evaluate how much unsolved skew of ComTick can be absorbed by our approach. After that the performance of AsyTick-PLB, ComTick and OriTick is presented for comparison, followed by the scalability evaluation of our approach.

Fig. 8 shows the completion situation of subsequent ticks for the FS benchmark when its ninth iteration has a significant load imbalance issue. From this figure, we can observe that the completion ratios of its successive five iterations are 80.2%, 52.6%, 28.4%, 16.5% and 4.3%, respectively. It means that there is amount of parallelism among ticks can be exploited by our approach, since the localized mobility property of fish school simulation algorithm.

Fig. 9 shows how much unsolved skew of ComTick can be eliminated by AsyTick-PLB on the above four benchmarks. In reality, it shows the ratio of spared time in successive ticks via AsyTick-PLB against the extra execution time caused by unsolved computational and communication skew in a tick of ComTick. From Fig. 9, we can observe that AsyTick-PLB can absorb 74.3% and 88.5% unsolved skew of FS and n-body benchmarks with ComTick respectively, and even absorb all unsolved skew of PageRank and jacobi method with ComTick.

The impact of scheduling algorithm on the skew-absorbing ability of AsyTick is shown in Fig. 10. To investigate this impact, we make the computational imbalance degree of benchmarks very high, until the

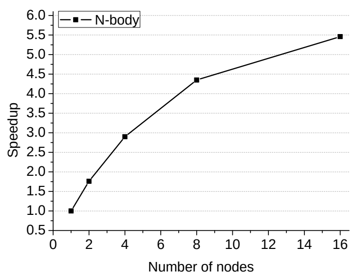


Fig. 12. The scalability of AsyTick on n-body benchmark.

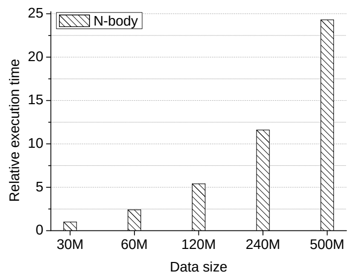


Fig. 13. The impact of data size on AsyTick above benchmark n-body.

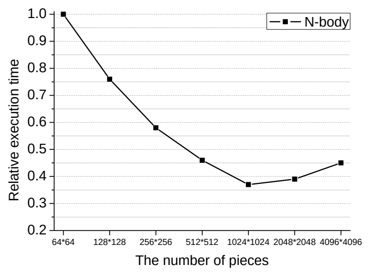


Fig. 14. The impact of the number of pieces on AsyTick above n-body.

skew of these benchmarks cannot be all absorbed by AsyTick. Then we test the results via turning-on and turning-off scheduler of AsyTick. In this way, it can present how much ratio of all absorbed skew is contributed by the scheduling algorithm of AsyTick. As shown in Fig. 10, we can observe that 23.4% of all absorbed skew is contributed by the scheduling algorithm of AsyTick on FS benchmark. It can also be seen that both the ratio of PageRank and jacobi method are less than FS and n-body, because the scheduler for PageRank and jacobi method only takes tick number in consideration of the priority definition.

Fig. 11 shows the performance of AsyTick-PLB and ComTick against OriTick. As shown in the figure, for the n-body benchmark, the speedup of ComTick against OriTick is only 1.45. Yet, the speedup of AsyTick-PLB against OriTick can achieve up to 3.67 for the n-body benchmark. In other words, our approach can improve the performance of persistence-based load balancer up to 2.53 times. This is because AsyTick-PLB can execute much asynchronous part of subsequent ticks in advance via exploiting the idle time caused by the unsolved skew still existing in each tick. Then AsyTick-PLB can resist the accumulation of the skew unsolved by ComTick in each tick.

Finally, Fig. 12 shows the scalability of AsyTick on n-body. From this figure, we can observe that our solution has a good scalability owing to the low computational skew and low communication skew with low runtime overhead.

5.4 Impact of system parameters

In this part, we examine the impact of data size and the number of pieces on the performance of AsyTick using the above n-body benchmark. Fig. 13 reveals that the execution time of n-body is almost linear with the data size for AsyTick. In other words, the data size has very little impact on the performance of AsyTick.

Fig. 14 reveals the impact of the number of pieces on the performance of AsyTick. From this figure, we can observe that the execution time of n-body decreases as the number of pieces for AsyTick increases. Yet, when the number of pieces is more than 1024×1024, the completion time is even increasing because more pieces carry more runtime overhead, although it benefits more from resisting the accumulation of skew, showing the tradeoff between the benefits gained from our approach and its cost runtime overhead.

6 Conclusion

Time-stepped applications need logical synchronization to ensure its correctness. But the logical synchronization between ticks makes the negative effects of unsolved skew accumulated in each tick. However, the current approaches can not resist the accumulation of skew. This paper reveals that much computational part of the processing for objects in each tick can be asynchronously executed in advance via utilizing the idle time of workers caused by unsolved skew in the current tick. An efficient approach is then proposed to exploit these asynchronous parts and achieve more parallelism, redressing the negative effects caused by accumulated unsolved computational and communication skew. Furthermore, we develop a data-centric programming model allowing users to easily express asynchronous parts and also implement a runtime system coupled with an ad hoc scheduler for time-stepped applications to support the efficient execution of asynchronous parts. Experimental results show that our approach can improve the performance of time-stepped applications compared to a state-of-the-art computational skew-resistant approach up to 2.53 times. The experiments also demonstrate that our approach can work well together with current skew-resist solutions.

Now, we have provided an approach to efficiently support the execution of time-stepped applications which can be decomposed. In the future work, we will discuss which time-stepped applications can be decomposed and how to correctly decompose these applications. At the same time, we will also incorporate our approach into other skew-resistant approaches and analyse their performance together to show its efficiency, and also employ it to more time-stepped applications to demonstrate its feasibility.

Acknowledgments

This paper is supported by China National Natural Science Foundation under grant No. 61272408, 61322210, National High-tech Research and Development Program of China (863 Program) under grant No.2012AA010905, CCCPC Youngth Talent Plan, Doctoral Fund of Ministry of Education of China under grant No. 20130142110048.

REFERENCES

[1] I. D. Couzin, J. Krause, N. R. Franks, and S. A. Levin, "Effective leadership and decision-making in animal groups on the move," *Nature*, vol. 433, no. 7025, pp. 513–516, 2005.

[2] G. Wang, M. V. Salles, B. Sowell, X. Wang, T. Cao, A. Demers, J. Gehrke, and W. White, "Behavioral simulations in mapreduce," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 952–963, 2010.

[3] V. Springel, S. D. White, A. Jenkins, C. S. Frenk, N. Yoshida, L. Gao, J. Navarro, R. Thacker, D. Croton, J. Helly *et al.*, "Simulations of the formation, evolution and clustering of galaxies and quasars," *nature*, vol. 435, no. 7042, pp. 629–636, 2005.

[4] "Biological modeling and simulation," http://zool33.uni-graz.at/schmickl/index.html, 2012.

[5] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skew-resistant parallel processing of feature-extracting scientific user-defined functions," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 75–86.

[6] Y. Kwon, D. Nunley, J. Gardner, M. Balazinska, B. Howe, and S. Loebman, "Scalable clustering algorithm for n-body simulations in a shared-nothing cluster," in *Scientific and Statistical Database Management*. Springer, 2010, pp. 132–150.

[7] J.-M. Alimi, V. Bouillot, Y. Rasera, V. Reverdy, P.-S. Corasaniti, I. Balmes, S. Requena, X. Delaruelle, and J.-N. Richet, "First-ever full observable universe simulation," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society Press, 2012, pp. 1–11.

[8] S. Byna, J. Chou, O. Rübel, H. Karimabadi, W. S. Daughton, V. Roytershteyn, E. Bethel, M. Howison, K.-J. Hsu, K.-W. Lin *et al.*, "Parallel i/o, analysis, and visualization of a trillion particle simulation," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society Press, 2012, pp. 1–12.

[9] U. Kang, C. E. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec, "Hadi: Mining radii of large graphs," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 5, no. 2, p. 8, 2011.

[10] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer networks and ISDN systems*, vol. 30, no. 1, pp. 107–117, 1998.

[11] D. Liben-Nowell and J. Kleinberg, "The link-prediction problem for social networks," *Journal of the American society for information science and technology*, vol. 58, no. 7, pp. 1019–1031, 2007.

[12] H. H. Song, T. W. Cho, V. Dave, Y. Zhang, and L. Qiu, "Scalable proximity estimation and link prediction in online social networks," in *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*. ACM, 2009, pp. 322–335.

[13] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst, *Templates for the solution of linear systems: building blocks for iterative methods*. Society for Industrial Mathematics, 1987, no. 43.

[14] G. M. Shroff, "A parallel algorithm for the eigenvalues and eigenvectors of a general complex matrix," *Numerische Mathematik*, vol. 58, no. 1, pp. 779–805, 1990.

[15] D. Schrank, B. Eisele, and T. Lomax, "Tti's 2012 urban mobility report," 2012.

[16] P. Jamshidi, A. Ahmad, and C. Pahl, "Cloud migration research: A systematic review," *IEEE Transactions on Cloud Computing*, vol. 99, no. PrePrints, p. 1, 2013.

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

[18] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, pp. 59–72, 2007.

[19] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using mantri," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. USENIX Association, 2010, pp. 1–16.

[20] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skewtune: Mitigating skew in mapreduce applications," in *Proceedings of the 2012 international conference on Management of Data*. ACM, 2012, pp. 25–36.

[21] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proc. of the 10th USENIX conference on Operating systems design and implementation (OSDI)*. USENIX Association, 2012, pp. 17–30.

[22] J. Lifflander, S. Krishnamoorthy, and L. V. Kale, "Work stealing and persistence-based load balancers for iterative overdecomposed applications," in *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*. ACM, 2012, pp. 137–148.

[23] T. Zou, G. Wang, M. V. Salles, D. Bindel, A. Demers, J. Gehrke, and W. White, "Making time-stepped applications tick in the cloud," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 2011, p. 20.

[24] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime measurements in the cloud: observing, analyzing, and reducing variance," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 460–471, 2010.

[25] G. Wang and T. E. Ng, "The impact of virtualization on network performance of amazon ec2 data center," in *INFOCOM, 2010 Proceedings IEEE*. IEEE, 2010, pp. 1–9.

[26] O. Pearce, T. Gamblin, B. R. de Supinski, M. Schulz, and N. M. Amato, "Quantifying the effectiveness of load balance algorithms," in *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 2012, pp. 185–194.

[27] R. Alpert and J. Philbin, "cbsp: Zero-cost synchronization in a modified bsp model," *NEC Research Institute, 4 Independence Way, Princeton NJ*, vol. 8540, 1997.

[28] O. Bonorden, B. Juurlink, I. Von Otte, and I. Rieping, "The paderborn university bsp (pub) library," *Parallel Computing*, vol. 29, no. 2, pp. 187–207, 2003.

[29] C. Jhon, "Efficient barrier synchronization mechanism for the bsp model on message-passing architectures," in *Proceedings of the 12th. International Parallel Processing Symposium on International Parallel Processing Symposium*. IEEE Computer Society, 1998, p. 255.

[30] J. C. Sancho, K. J. Barker, D. J. Kerbyson, and K. Davis, "Quantifying the potential benefit of overlapping communication and computation in large-scale scientific applications," in *SC 2006 Conference, Proceedings of the ACM/IEEE*. IEEE, 2006, pp. 17–17.

[31] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick, "Avoiding communication in sparse matrix computations," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1–12.

[32] K. Kelsey, T. Bai, C. Ding, and C. Zhang, "Fast track: A software system for speculative program optimization," in *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2009, pp. 157–168.

[33] J. Cipar, Q. Ho, J. K. Kim, S. Lee, G. R. Ganger, G. Gibson, K. Keeton, and E. Xing, "Solving the straggler problem with bounded staleness," in *Proceedings of the 14th USENIX conference on Hot Topics in Operating Systems*. USENIX Association, 2013, pp. 22–22.

[34] "Stanford dataset," http://snap.stanford.edu/data/, 2009.

[35] R. Power and J. Li, "Piccolo: building fast, distributed programs with partitioned tables," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI)*. USENIX Association, 2010, pp. 1–14.