Proceedings of the 2017 Winter Simulation Conference
W. K. V. Chan, A. D'Ambrogio, G. Zacharewicz, N. Mustafee, G. Wainer, and E. Page, eds.

A WORK-STEALING BASED DYNAMIC LOAD BALANCING ALGORITHM FOR CONSERVATIVE PARALLEL DISCRETE EVENT SIMULATION

Tang Wenjie
Yao Yiping
Zhu Feng
Li Tianlin

College of Information System and Management
National University of Defense Technology
109th Deya Road
Changsha, Hunan, CHINA

Song Xiao

School of Automation
Beihang University
37th Xueyuan Road
Beijing, CHINA

ABSTRACT

In the past few years, we have witnessed an increased interest in using multithreading PDES on multicore platforms. The work-stealing scheme, which towards to general multithread computing, can be utilized in PDES to achieve load balance straightly. However, to the best of our knowledge, the work-stealing scheme has only served as a competitor, instead of a cooperator, to other load balancing algorithm. In this paper, we propose a work-stealing based dynamic load balancing algorithm (WS-DLB) with the aim of combining their advantages. It adaptively rebalances the LPs distribution based on a priori estimation, and uses a greedy lock-free work-stealing scheme to eliminate bias at runtime. In addition, these two schemes are well adapted to enhance each other. We analyze the performance characteristics of the proposed algorithm by means of a synthetic benchmark. Experiments demonstrate that our WS-DLB algorithm achieves better performance.

1 INTRODUCTION

Discrete event simulation is utilized extensively to study complex systems, such as transportation systems, biological systems, and military systems. With the growing complexity of models, it is difficult to run applications using sequential simulators. By exploiting the inherent parallelism among simulation entities, Parallel Discrete Event Simulation (PDES) can substantially improve the performance and capacity, and thereby enabling simulate larger applications and more detailed models in shorter time. Similar to other parallel and distributed applications, one of the greatest challenges in PDES is load imbalance. Because the pace of the simulation is limited by the slowest processing element, the distribution of the workload has a significant impact on the performance. Various algorithms have been proposed to solve the load imbalance for PDES. Most of them built a load estimator to predict the future workload, and map the LPs into different groups with close workload, either statically or dynamically. The static approach assigns logical processes (LP) to processors, and the LPs will stay at the same processor during their lifetime. In contrast, the dynamic approach, can rebuild groups to rebalance the workload during the runtime by migrating LPs.

On the other hand, multicore architectures have quickly spread to all computing domains in the past few years. We also have witnessed an increased interest in moving PDES to multicore platforms, such as (Vitali, Pellegrini, and Quaglia 2012; Wang et al. 2014; Chen et al. 2011; Tang and Yao 2013). In these works, the multithreading have become the mainstream for PDES on multicore platforms. Moreover, thanks to the shared memory address, migrating LPs and coordinating threads are much more convenient and cause lower overhead, which provides a better circumstance for load balancing. As a special

application, classic load balancing strategies (Andrews 1999) for general multithreading computation can also be used in PDES straightly. Among these methods, the work-stealing strategy (Blumofe and Leiserson 1994) is proven to be a good algorithm which has acted as the kernel scheduling algorithm in many threading libraries, such as TBB, Cilk and .NET Task Parallel Library. Some researchers have previously introduced this scheme into PDES to enable dynamic load balancing.

However, to the best of our knowledge, the work-stealing scheme has only served solely as one of the load balancing algorithms (Cai, Letertre, and Turner 1997; Vee and Hsu 2000). In other words, it acts as a competitor, instead of a cooperator, to other load balancing algorithms. In fact, the scheme can be used as an assistant to enhance other load balancing algorithm. A load balancing algorithm, which uses the data of the past to predict the future, will inevitably make some error. In this case, the work-stealing scheme can be used to adjust the bias at runtime. On the other hand, the work stealing scheme also introduces some more overhead than normal execution, such as thread conflicts and cache penalties. The load balancing algorithm can exploit more application information to achieve a relatively balanced workload distribution, so that the number of stealing can be reduced and thus improving the performance.

Hence, we propose a work-stealing based dynamic load balancing algorithm (WS-DLB) to combine their advantages. The algorithm adaptively rebalances the LPs distribution based on the estimated workload, and uses a greedy lock-free work-stealing scheme to eliminate the bias. Our proposed WS-DLB algorithm is composed of three modules: performance monitor, load balance manager and scheduler. During simulation, the performance monitor records the time that an LP utilized for model computation. Once a new rebalance is needed, the load balance manager collects the data from all performance monitors, and calculates a near-optimal partition to map the LPs into different groups. Normally, the scheduler chooses the LPs from its own group using the largest workload first rule, which can improve the probability of successful stealing under the same workload distribution. When there is no LP left, it uses a greedy lock-free scheme to randomly steal LPs from other groups to make the thread keep on working. We discuss the performance characteristics of the proposed algorithm by means of a synthetic benchmark. Experiments show that our WS-DLB algorithm achieves better performance.

The remainder of this article is organized as follows. In Section 2 we discuss the background and related works. The description of the WS-DLB algorithm in provided in Section 3. Section 4 is devoted to the experimental study.

2 BACKGROUND AND RELATED WORKS

Recently, some researchers have designed multi-threaded PDES engines to improve PDES performance on multicores, such as (Vitali, Pellegrini, and Quaglia 2012; Wang et al. 2014; Chen et al. 2011; Lin et al. 2016). These works mainly use the optimistic time management algorithm to run the simulation. However, programming based on optimistic algorithm is extremely hard due to the necessity to write reversing (or state saving) and commit methods. Moreover, in most experiences of the authors' researches, only the libraries, instead of source code, can be provided by domain experts due to the intelligence protection. Hence, building applications on real problems remains great challenges even though some tools can generate optimistic methods from the sequential source code automatically. In addition, conservative algorithm can be further categorized into two classes, namely synchronous and asynchronous protocols. The asynchronous conservative algorithms rely on static topology which cannot fit our need in many applications. Take these reasons into account, we focus on the synchronous conservative PDES on multicore systems in this work.

A lot of research has been done to achieve load balance, including static and dynamic methods. Static approaches cannot handle the workload variation, and thus are usually employed with specific applications, such as (Boukerche and Fabbri 2000; Lemeire et al. 2004). Solutions for dynamic load balancing have been studied for both conservative and optimistic algorithms. Generally, the dynamic load balancing algorithms use a monitoring scheme to detect load imbalance, and make dynamic adjustment to improve the performance of simulation. These approaches differ in metrics of detecting load imbalance,

and balancing schemes. As we focus on conservative synchronization here, we refer to (Meraji, Zhang, and Tropper 2010) and (Carothers and Fujimoto 2000) for dynamic load balancing with optimistic synchronization.

(Boukerche and Das 1997) propose a dynamic load balancing algorithm that uses the notion of CPU-queue length as the load metric and a process migration mechanism as the balancing scheme. (Xiao et al. 1999) propose a critical channel traversing algorithm for dynamic load balancing on shared-memory multiprocessors, which let each processor to obtain a cluster of LPs to process from a centralized queue. It can be seen as a work-sharing strategy that utilized in PDES. Both the above algorithms are based on the CMB (Chandy-Misra-Bryant) protocol, so they attempt not only to balance the load for a distributed memory system, but also to minimize the number of null messages sent between LPs. There are also some work oriented to the synchronous protocol. (Vee and Hsu 2000) present several new locality-preserving load balancing mechanisms for synchronous simulations on shared-memory multiprocessors. (Cai, Letertre, and Turner 1997; Turner 1998) present a novel approach to parallel discrete event simulation based on the Cilk model of multithreaded computation. The simulation executes in cycles, where each cycle contains a divide and conquer computation. In essential, these works use the work-stealing scheme to achieve load balance. In our work, the work-stealing scheme is only a component of the proposed algorithm and some efforts have been taken to let it cooperates with other components.

3 ALGORITHM DISCRIPTION

3.1 Framework Overview

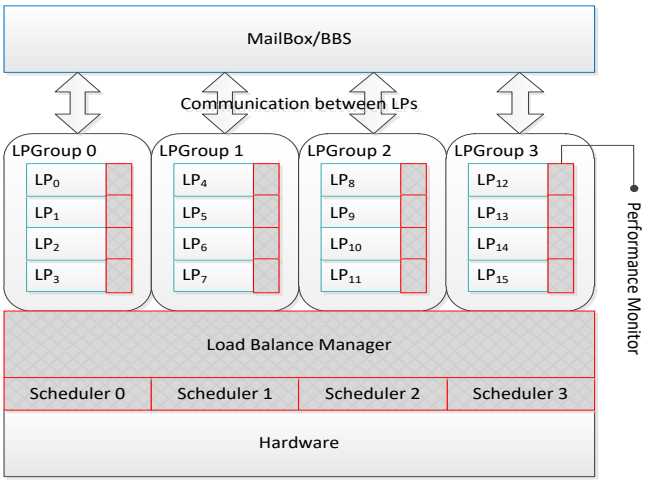


Figure 1: The architectural overview of WS-DLB algorithm on multicore platform.

The architectural overview of WS-DLB is illustrated in Figure 1. In WS-DLB, logical processes are the basic unit to adjust workload among threads. They are assigned into different groups (LPGroup), and each LPGroup is mapped onto a thread. LPs communicate with each other through exchanging peer-to-peer messages or using publish/subscribe. The mailbox is utilized to store peer-to-peer messages for each LP, and the BBS is used to store interest messages for publisher/subscriber. These communication utility are implemented by concurrent data structures so they can provide service to all LPs efficiently.

The WS-DBL algorithm consists of three modules: performance monitor, load balance manager and scheduler. The performance monitor is a lightweight timer which is transparent to the simulation application. It is inserted into each LP and collects the time that a LP consumed to do model computation. The load balance manager plays a role in rebalancing the workload among threads adaptively. Once imbalance is detected, the load balance manager gathers the data from all performance monitors, and

regroups the LPs according to the estimated workload. In this work, an exponential weighted moving average is adopted to forecast the workload trend based on the preceding data. Based on the information, the load balance manager is able to calculate a near-optimal partition to map the LPs on their suitable groups. The scheduler is responsible to select an LP and make it move forward. Normally, it chooses the LPs from its own LPGroup using the largest workload first (LWF) rule, which can help improve the probability of successful stealing under the same workload distribution. When there is no LP left in its own LPGroup, it uses a greedy lock-free algorithm to randomly steal LPs from other groups to adjust workload at runtime. In addition, a relative value is used for the load balance manager to detect imbalance. Explicitly, a threshold is counted as the number of stealing in a time window after a rebalancing. Once the number of stealing in a time window exceeds the threshold significantly, it means the current partition is not well enough and a new rebalancing must be triggered. The behind rationale is load imbalance can only be minimized rather than be annihilated. For example, if one LP consumes more than 90% computations, the load imbalance always exists wherever we distribute the LPs. The rebalancing is uesless on such situation. In general, the work-stealing can help to adjust the workload imbalance and provide a way to compute relative imbalance. The rebalancing uses application information to enable a relatively balanced scenario to reduce number of stealing.

Algorithm 1. The Simulation Execution of the scheduler *i*

G_i: the i-th LPGroup

```
1.  WHILE  simulation in progress
2.    LBTS ← Synchronization()
3.    IF  need to rebalance = true && is_master_thread
4.        The load balance manager redistribute all LPs into each group
5.    IF a rebalance has began
6.       wait until the rebalance procedure completes
7.    lp←ChooseMaxLoadLP(Gi)
8.    IF  lp!=NULL
9.      advance(lp, LBTS), goto line 6
10.   ELSE
11.     steal a LP from other LPGroup, and advance the LP
12.     repeat line 10 until all LP has advanced in this cycle
13.   ENDIF
14. ENDWHILE
```

As mentioned before, the synchronous conservative protocol is studied in this work. As shown in Algorithm 1, the lowest bound on time-stamp (LBTS) of all LPs is computed after Synchronization (line 2). In this step, messages, which are generated in the previous cycle, are retrieved from the mailbox/BBS and be inserted into the target LP's event list. Once load imbalance has been detected, the master thread will trigger the load balance manager to assign LPs into different group (line 3-4). Other threads have to wait until the rebalance procedure finishes. After that, the scheduler chooses the LPs with the maximum estimated workload from its own LPGroup and make it advance to the LBTS. The procedure will be iterated until there is no LP left (line 6-8). Then, the scheduler will steal LPs from other LPGroups so that the thread can keep on executing useful work instead of idle waiting. The scheduler will repeat this procedure until all LPs are processed in this cycle (line 10-11). What needs illustration is that the scheduler only possesses the LP temporally, and the stole LP is still stored in the original LPGroup before the next rebalancing. How to rebalance the workload and how to efficiently steal work are described in detail in the following sections.

3.2 Rebalance LPs Distribution

Due to the dynamic nature of simulation, the workload of each LP is unknown in advance. Here, we utilize an exponential weighted moving average to predict the future workload, as shown in formula (1). The rational of the estimator is from the temporal and spatial locality of simulation entities.

$$\mathrm{load}(\mathrm{lp},\mathrm{w}) = \sum_{\mathrm{i}=0}^{\mathrm{w}-1} \lambda^{\mathrm{w}-\mathrm{i}-1}\mathrm{T}_{\mathrm{i}}(\mathrm{lp}) \tag{1}$$

Here, $\mathrm{T}_{\mathrm{i}}(\mathrm{lp})$ refers to the wall clock time for the LP in the i-th control interval, $\lambda \in [0,1]$ is a decay factor to remove the impact of the far past. In addition, observing the fact that $\mathrm{load}(\mathrm{lp},\mathrm{w}) = \lambda\mathrm{load}(\mathrm{lp},\mathrm{w}-1) + \mathrm{T}_{\mathrm{w}-1}(\mathrm{lp})$, the estimator can be easily calculated, and introduces negligible computational and memory overhead.

Now, we are given n LPs, lp_1, lp_2, …, lp_n, where each LP lp_k has an associated nonnegative workload of load_k. We are also given m identical cores, $core_1$, $core_2$, …, $core_m$. Any LP can run on any core. The goal is to find a partition to distribute LPs into different LPGroups so that the makespan (the total time that elapses from the beginning to the end) is minimum. In essential, it is a classic job shop scheduling problem, which has been proved to be NP hardness. Find an optimal solution will lead to unacceptable complexity and runtime overhead. Hence, we use an approximation approach to find a near-optimal solution, as shown in Algorithm 2. All LPs are sorted increasingly in LPLoadList ordered by the estimated workload. The rebalancing procedure will traverse the list, and put the next LP into the LPGroup with minimum workload. The rebalancing algorithm has a polynomial time complexity. It can be further proved to be a 2-approximation algorithm (Cormen et al. 2009).

Algorithm 2. Procedure of Rebalancing LPs Distribution

LPLoadList: an increasingly sorted list of all LPs, orderd by the predicted workload
LoadHeapofLPGroup: a priority queue of LPGroups, the key value is the sum of workload of all LPs in the group

```
1.  WHILE  LPLoadList contains item
2.    lp ←LPLoadList.pop()
3.    lpgroup ←LoadHeapofLPGroup.pop();
4.    lpgroup.append(lp)
5.    LoadHeapofLPGroup.insert(lpgroup)
6.  ENDWHILE
```

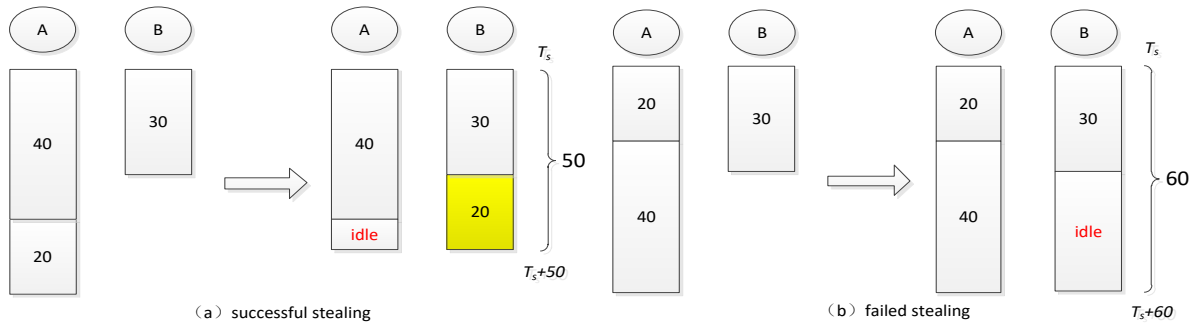


Figure 2: An example to show why scheduling sequence affects the work-stealing under the same workload distribution

Though the rebalancing algorithm behaves well in theory, many factors will affect the final result, such as biased workload estimation and un-optimal partition. Hence, it is still needed to use the work-stealing to adjust the imbalance. The scheduler uses the largest workload first rule to choose LP to advance. The scheme is necessary in order to improve the probability of successful stealing under the same workload distribution. Fig.2 uses an example to demonstrate the fact. At timestamp T_s, there are two

LPGroups in the simulation, and each block denotes the processing time of an LP advancement. Both part (a) and (b) have the same LPs distribution and thus the same makespan. In part (a), scheduler B completes the advancement of LPs in its own group after timestamp T_s+30. It can steal an LP from LPGroup A because there are un-advanced LP left. However, scheduler B cannot steal in part (b) because the second LP in LPGroup A has begun its work. Finally, the part (a) can achieve less makespan than the part (b). The rationale behind this fact is: the LP is the basic unit for stealing in the proposed algorithm and it cannot be stolen once it begins processing events. When a stealing happens to an LPGroup, the largest workload first rule can make un-advanced LPs as many as possible. Therefore, the thief thread can steal a task successfully in higher probability. Furthermore, as shown in Algorithm 2, LPs are sorted decreasingly in the LPGroup after rebalancing, so the scheduler can simply get the first item from LPGroup.

3.3 Greedy Lock-free Stealing

The work-stealing scheme has been widely used in multithread programming. Generally, it acts roughly as follows: each thread holds a deque to store tasks. When a new task is generated (spawned), it will be put into the top of the deque. Threads will keep on getting tasks from the top of the deque. Hence, the most recently created tasks and, therefore, the hottest in the cache, are processed by the host thread. When there are no tasks in its own deque, the thread will randomly steal a task from the bottom of the other deque. However, we face a relatively less sophisticated environment than the general multithread programming, so that some optimization can be made to obtain a better performance. The main difference between the traditional work-stealing scheme and the proposed method includes: (a). Schedulers steals the LP with maximum workload from other LPGroups. (b). Stealing and coordination are implemented by atomic operations instead of locks.

Firstly, the workload of LP, though estimated, is known before stealing. The information can help us improve the efficiency of stealing. In other words, it helps us steal a specified LP, the LP with maximum workload in LPGroup, to make the load as balanced as possible. Let's inspect the stealing from a local view which includes only the *thief* thread and the *victim* thread. Assuming that $\mathrm{C_{victim}}$ denotes the completion time of the *victim* thread while $\mathrm{C_{thief}}$ denotes the completion time of the *thief* thread. At time $\mathrm{C_{thief}}$, the *thief* thread finishes its own job and tries to steal an LP from the *victim* thread. Among all LPs which have not advanced at that time, let mlp denotes the LP with maximum workload and alp denotes any other LP. Without loss of generality, mlp and alp are different LPs. Hence,

$$\mathrm{C_{victim}} - \mathrm{workload(mlp)} - \mathrm{workload(alp)} > \mathrm{C_{thief}} \tag{2}$$

if mlp is stolen, then the completion time of both threads is

$$\mathrm{T\_Complete_m} = \max\{\mathrm{C_{victim}} - \mathrm{workload(mlp)}, \mathrm{C_{thief}} + \mathrm{workload(mlp)}\}$$

if alp is stolen, then the completion time is

$$\mathrm{T\_Complete_a} = \max\{\mathrm{C_{victim}} - \mathrm{workload(alp)}, \mathrm{C_{thief}} + \mathrm{workload(alp)}\}$$

According to formula (2), $\mathrm{C_{thief}} + \mathrm{workload(mlp)} < \mathrm{C_{victim}} - \mathrm{workload(alp)}$.

According to the definition of mlp, $\mathrm{C_{victim}} - \mathrm{workload(mlp)} < \mathrm{C_{victim}} - \mathrm{workload(alp)}$.

$$\Rightarrow \mathrm{T\_Complete_m} < \mathrm{C_{victim}} - \mathrm{workload(alp)}$$

So, $\mathrm{T\_Complete_m} < \mathrm{T\_Complete_a}$.

In summary, stealing mlp is the best option from the view of the *thief* thread. And it is also the reason that we call the algorithm a greedy stealing. Furthermore, the stealing rule is consistent with the way that a scheduler chooses the LP from its own LPGroup. Therefore, it is also helpful for improving the probability of successfully stealing.

Secondly, each LPGroup is determined and will not change between two rebalancing. Thus, it is not necessary to use a flexible deque to store dynamic tasks. Instead, the LPGroup is implemented as a vector, and some atomic data are used to achieve a lock-free coordination between threads. Algorithm 3 shows the procedure of scheduler i choosing or stealing LP to advance. It can be decomposed into three stages,

namely *selfwork*, *stealing* and *reset*. In *selfwork* stage (Line 1-8), the scheduler keeps on getting LPs and makes it advance until there is no un-advanced LPs in its own LPGroup. The scheduler will add 1 to *n_finish_selfwork* at the end of the *selfwork* stage. In *stealing* stage (Line 9-18), the scheduler will randomly steal a task from other LPGroup. Atomic data, *index_ unadvance*$_i$, is used for coordination between host scheduler and thief scheduler in Group *i*. When *n_finish_selfwork* equals to the number of LPGroup, it means that all LPs in the simulation either have been selected by its host scheduler or stolen by *thief* schedulers. Hence, the scheduler can stop stealing and add 1 to *n_stop_stealing*. In *reset* stage (Line 19-25), the scheduler must wait until all schedulers finish their work in hand. When *n_stop_stealing* equals to the number of LPGroup, all LPs have been advanced in the current cycle. Hence, schedulers can move on to the next synchronization. In addition, the last scheduler need to set the shared atomic variables to 0 for the next cycle.

Algorithm 3. Procedure of scheduler *i* choosing or stealing LP to advance

```
Gi: the i-th LPGroup
index_unadvancei: an atomic int type, denote the index of the first un-advanced LP in LPGroup i [init: 0]
n_finish_selfwork: an atomic int type, denote number of schedulers that finish its own work [init: 0]
n_stop_stealing: an atomic int type, denote number of schedulers that stop stealing [init: 0]
n_end_wait: an atomic int type, denote sequence of a scheduler know that all schedulers have finished the stealing stage
[init: 0]
1.  FOR  i ← 0  to  Gi.size()
2.    current_index ← index_ unadvancei++        // get the id of the next unadvanced LP in Gi
3.    IF  current_index ≥ Gi.size()
4.        BREAK;
5.    lp ← Gi.getlp[current_index]
6.    advance(lp, LBTS)
7.  ENDFOR
8.  n_fsw ← ++n_finish_ selfwork                 // add 1 to n_finish_ selfwork and return the new value to n_fsw

9.  WHILE  n_fsw < Num_LPGroups
10.   randomly choose a victim LPGroup j
11.   steal_index ← index_ unadvancej++          // get the id of the next unadvanced LP in Gj
12.   IF  steal_index < Gj.size()
13.       lp ← Gj.getlp[steal_index]
14.       advance(lp, LBTS)
15.   ENDIF
16.   n_fsw ← n_finish_ selfwork
17. ENDWHILE
18. n_ss ← ++n_stop_stealing                     // add 1 to n_stop_ stealing return the new value to n_ss

19. WHILE  n_ss < Num_LPGroups
20.   n_ss ← n_stop_stealing
21.   CONTINUE;
22. n_ew ← ++n_end_wait
23. IF n_ew = Num_LPGroups                       // The last thread will reset the shared vairable
24.   n_stop_stealing ← 0, n_finish_ selfwork_stage ← 0, n_end_wait ← 0
25. ENDIF
26. NextSynchronization …
```

Figure 3 uses a series of snapshots to show some critical scenarios of the procedure. The initial LPs distribution is shown in t = 0, and the value associated with LP denotes how much time are needed to advance the LP. At this time, all schedulers select the item from their own LPGroups. When t = 55, scheduler 1 finishes all tasks and tries to steal a LP from LPGroup 0. Since the index_unadvance$_0$ = 2, scheduler 1 steals LP$_2$, and change the index_unadvance$_0$ to 3. Therefore, scheduler 0 will choose LP$_3$ to advance later and the final makespan is 63. For comparison, if the LP$_3$ is stolen, the final makespan is 66.

When t = 60, all schedulers have finished their jobs in *selfwork* stage. However, the LP_2 has been stolen and is being processed by the scheduler 1. Instead of entering to the next synchronization, the scheduler 0 has to wait until the scheduler 1 completes its tasks in hand (t = 63). Till then, all schedulers can go to the next synchronization according to Algorithm 3.

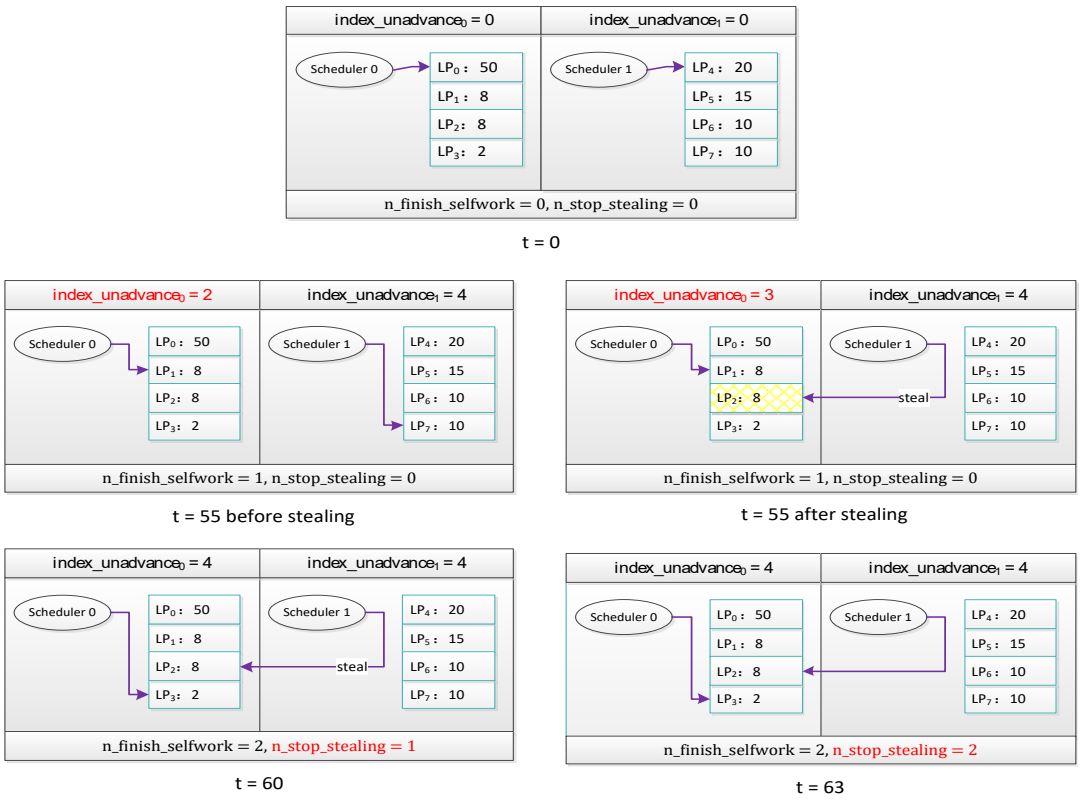


Figure 3: an example to show the procedure of the work-stealing scheme

4 EXPERIMENT

This section presents an initial evaluation of the proposed WS-DLB algorithm. We discuss the performance properties of the proposed algorithm using a parameterized benchmark, La-pdes. The computing platform is a Server providing two Intel Xeon E5-2650 8-core CPU with 32 GB of main memory. The simulation framework runs on a 64 bit version of CentOS 6.5 and g++ in version 4.4.7. Furthermore, each data point shows the mean and the 99% confidence intervals computed over 50 independent repetitions.

The La-pdes benchmark application, which is developed by Los Alamos National Laboratory, aims to mimic the behavior of real discrete event simulation applications (Park et al, 2015). Roughly, the La-pdes contains some entities, each of which has a *SendHandler*, a *ReceiveHandler* and a local list data structure consisting of floating-point value elements. The *SendHandler* sends messages to other destination entities, and re-schedules itself for the next event. The *ReceiveHandler* receives an event and then calculating a specific number of floating point multiplications and additions using the elements of the list. The La-pdes benchmark contains a set of twelve parameters, which enables fine-grained control over the numbers of events, the number of entities, the distribution of events to sending and receiving entities and so on. In this paper, we only focus on the a subset that closely relative with the load balancing, as shown in Table 1. More details can be found in the reference paper (Park et al. 2016).

Table 1: Description of parameters for La-pdes experiment.

Parameter	Default	Value Range	Description
n_{ent}	1000	1, …, ∞	Number of entities
s_{sent}	1000	1, …, ∞	Average number of send events per entity
endTime	1000	1, ⋯, ∞	Duration of simulation
$p_{receive}$	0.0	[0, 1]	Entity E_i receives a fraction of $p_{receive}\times(1-p_{receive})^{i-1}$ of all the requested messages. $p_{receive}$ = 0: uniform distribution.
p_{send}	0.0	[0, 1]	Entity E_i sends a fraction of $p_{send}\times(1-p_{send})^{i-1}$ of all the requested messages. p_{send} = 0: uniform distribution.
p_{list}	0.0	[0, 1]	Parameter for geometric distribution of linear list sizes. Set to 0 for uniform distribution; Set to 1.0 to make entity 0 the only entity with a list.
ops_{ent}	10^4	1, ⋯, ∞	Average number of operations per handler per entity.
cores	8	1, … , 16	Number of cores to run simulation.

Three algorithms are utilized for comparison, namely WS-TBB, WS-DLB and DLB. The WS-TBB algorithm only uses the work-stealing scheme to achieve balance, and is built upon the Intel TBB library. The TBB provides the parallel template and supports the work-stealing scheme automatically. The proposed WS-DLB algorithm is built based on the boost threading library which exposes more controls to developers. The DLB algorithm only uses the rebalance scheme of the WS-DLB to rebuild LPGroups. Without number of work-stealings to indicate imbalance, the DLB algorithm has to trigger the rebalancing periodically. In addition, the Sequential algorithm is utilized to compute speedup.

(1) Performance comparison with different p_{list} settings

The first experiment is carried out to compare the simulation execution performance with different plist settings. Though all entities receive similar number of *SendHandler* and *ReceiveHandler* events, lower-indexed entities have longer list, thereby involving more computation. For example, when p_{list} = 0.1, E_0 needs to process 11% computation among all *ReceiveHandler* workload and E_1 needs to process 10% computation among all *ReceiveHandler* workload. Hence, the speedups achieved by two algorithms drop down when p_{list} increases, as shown in Figure 4. In some extreme case, the parallel algorithm is even worse than the sequential algorithm. It is also an evidence that load balancing is necessary for simulation. In comparison, the WS-DLB algorithm behaves better than the WS-TBB and the DLB algorithms. When imbalance is moderate (p_{list} = 0.01), both algorithms achieve similar results and relatively high speedup. When imbalance is serious, the WS-DLB algorithm is better than the WS-TBB algorithm by 50%-66%.

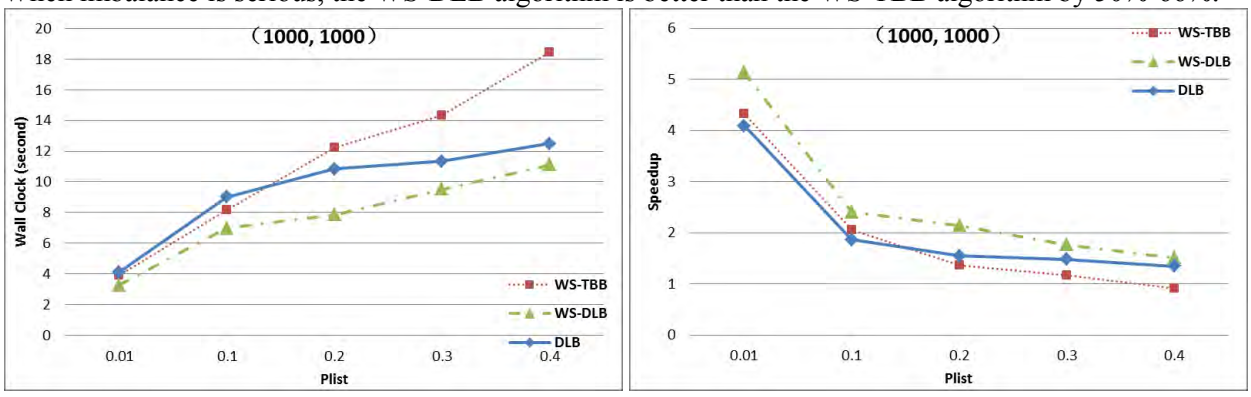


Figure 4: Performance Comparison of La-pdes using different p_{list}

(2) Performance comparison with different $p_{receive}$ settings

The second experiment is carried out to compare the simulation performance with different $p_{receive}$ settings, from 10^{-5} to 10^{-1}. Different from the first experiment, the value of $p_{receive}$ also affects the performance of sequential algorithm. When $p_{receive}$ is small, simulation entity needs compute $p_{receive} \times (1 - p_{receive})^{i-1}$ many times in order to find the destination entity. Hence, simulation with smaller $p_{receive}$ consumns more time even with the sequential algorithm. Moreover, the second experiment presents a more complicated scenario. $p_{receive}$ not only generates imbalance on event processing, but also generates imbalance on simulation management. Lower-indexed entities will receive larger shares of *ReceiveHandler* events and thereby consumes more computation. The results demonstrates the peak performance are achieved by both algorithms when $p_{receive} = 10^{-3}$, as shown in Figure 5. Since LP is the basic and indivisible unit to adjust workload, the performance improvement cannot be achieved when the workload cannot be divided equally. In comparison, the WS-DLB algorithm obtains better or comparable results than the WS-TBB and the DLB algorithms. When imbalance is moderate ($p_{receive}=10^{-5}$, 10^{-4}, 10^{-3}), the WS-DLB algorithm is better than the WS-TBB algorithm by 16%-36%. When imbalance is serious, all algorithms achieve similar results.

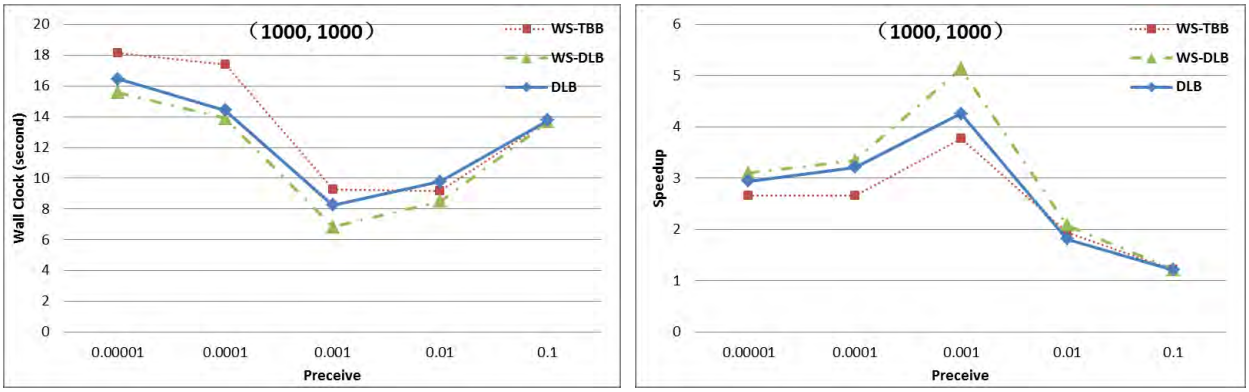
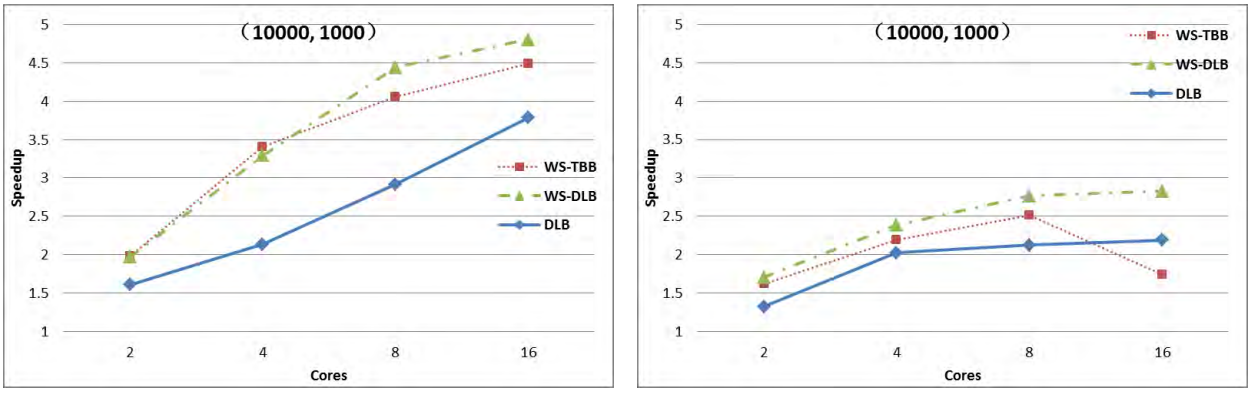


Figure 5: Performance Comparison of La-pdes using different $p_{receive}$

(3) Scalability

The third experiment is carried out to compare the strong scalability between WS-TBB and WS-DLB methods. As shown in Figure 6, the number of cores is varied from 2 to 16. Different from default setup, n_{ent} is set to 10000, p_{list} is set to 0.01 in part (a) and $p_{receive}$ is set to 0.001 in part (b). Different from the previous experiments, the DLB algorithm usually performs worst due to more overhead is needed for each rebalancing and the DLB algorithm cannot trigger the rebalancing adaptively. Hence, it is not a good choice to achieve load balance when application scale is large. Moreover, the scalability is affected by the load imbalance in both cases, so the linear speedup cannot be achieved. In comparison, the WS-DLB algorithm presents a better scalability. It always obtain better performance when more cores are used. In contrast, the WS-TBB algorithm meets a inflexion point in part (b). The performance dropped dramatically when 16 cores are used, even worse than the DLB algorithm.



(a) $p_{list} = 0.01$ (b) $p_{receive} = 0.001$

Figure 6: Performance Comparison of La-pdes using different cores

5 CONCLUSIONS AND FUTURE WORKS

In this paper, a work-stealing based dynamic load balancing algorithm (WS-DLB) is proposed to solve the load imbalance of the synchronous conservative protocol running on multicore platforms. The algorithm rebalances the LPs distribution based on estimated workload adaptively, and adjusts the assignment when imbalance occurs. Experiments show that the proposed algorithm can take advantages of load balancing and work-stealing, and thus achieve a better performance.

In our future work, we plan to consider the impact of communication and simulation housekeeping, in order to be adapted to more fine-grained applications.

REFERENCES

Andrews, Greg R. *Foundations of Parallel and Distributed Programming*. 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc, 1999.

Blumofe, R.D., and C.E. Leiserson. 1–29. "Scheduling Multithreaded Computations by Work Stealing." *Proceedings 35th Annual Symposium on Foundations of Computer Science*, 1994. doi:10.1109/SFCS.1994.365680.

Boukerche, Azzedine, and Sajal K Das. 20–28. "Dynamic Load Balancing Strategies for Conservative Parallel Simulations." *ACM SIGSIM Simulation Digest* 27 (1): 1997. doi:10.1145/268823.268897.

Boukerche, Azzedine, and Alessandro Fabbri. 1449–1457. "Partitioning Parallel Simulation of Wireless Networks." In *Proceedings of the 32Nd Conference on Winter Simulation*, 2000. WSC '00. San Diego, CA, USA: Society for Computer Simulation International. http://dl.acm.org/citation.cfm?id=510378.510591.

Cai, Wentong, Emmanuelle Letertre, and Stephen J Turner. 178–181. "Dag Consistent Parallel Simulation: A Predictable and Robust Conservative Algorithm." In *Proceedings 11th Workshop on Parallel and Distributed Simulation*, 1997. doi:10.1109/PADS.1997.594604.

Carothers, Christopher D., and Richard M. Fujimoto. 299–317. "Efficient Execution of Time Warp Programs on Heterogeneous, NOW Platforms." *IEEE Transactions on Parallel and Distributed Systems* 11 (3): 2000. doi:10.1109/71.841745.

Chen, Li-li, Ya-shuai Lu, Yi-ping Yao, Shao-liang Peng, and Ling-da Wu. 1–9. "A Well-Balanced Time Warp System on Multi-Core Environments." In *Proceedings of the 2011 IEEE Workshop on Principles of Advanced and Distributed Simulation*, 2011. PADS '11. Washington, DC, USA: IEEE Computer Society. doi:10.1109/PADS.2011.5936752.

Cormen, Thomas, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms-3rd Edition*. *Contemporary Sociology,* 2009. Vol. 25. doi:10.2307/2077150.

Lemeire, Jan, Bart Smets, Philippe Cara, and Erik Dirkx. 189–194. "Exploiting Symmetry for Partitioning Models in Parallel Discrete Event Simulation." In *Proceedings - Workshop on Parallel and Distributed Simulation*, 18: 2004. doi:10.1109/PADS.2004.1301300.

Lin, Zhongwei, Carl Tropper, Robert A McDougal, Mohammand Nazrul Ishlam Patoary, William W Lytton, Yiping Yao, and Michael L Hines. "Multithreaded Stochastic PDES for Reactions and Diffusions in Neurons." *ACM Trans. Model. Comput. Simul.* 27 (2), 2016. New York, NY, USA: ACM: 7:1--7:27. doi:10.1145/2987373.

Meraji, Sina, Wei Zhang, and Carl Tropper. 1368–1380. "On the Scalability and Dynamic Load-Balancing of Optimistic Gate Level Simulation." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 29 (9): 2010. doi:10.1109/TCAD.2010.2049044.

Park, Eunjung, Stephan Eidenbenz, Nandakishore Santhi, Guillaume Chapuis, and Bradley Settlemyer. 2836–2847. "Parameterized Benchmarking of Parallel Discrete Event Simulation Systems: Communication, Computation, and Memory." In *Proceedings - Winter Simulation Conference*, 2016–Febru. doi:10.1109/WSC.2015.7408388.

Tang, W., and Y. Yao. 1335–1354. "A GPU-Based Discrete Event Simulation Kernel." *Simulation* 89 (11): 2013. doi:10.1177/0037549713508839.

Turner, Stephen J. 395–409. "Models of Computation for Parallel Discrete Event Simulation." *Journal of Systems Architecture* 44 (6–7):1998. doi:10.1016/S1383-7621(97)00055-6.

Vee, Voon-Yee, and Wen-Jing Hsu. 131–138. "Locality-Preserving Load-Balancing Mechanisms for Synchronous Simulations on Shared-Memory Multiprocessors." In *Proceedings of the Fourteenth Workshop on Parallel and Distributed Simulation*, 2000. PADS '00. Washington, DC, USA: IEEE Computer Society. http://dl.acm.org/citation.cfm?id=336146.336175.

Vitali, Roberto, Alessandro Pellegrini, and Francesco Quaglia. 211-220. "Towards Symmetric Multi-Threaded Optimistic Simulation Kernels." In *Proceedings - 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation, PADS 2012*, 2012. doi:10.1109/PADS.2012.46.

Wang, Jingjing, Deepak Jagtap, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 1574–1584. "Parallel Discrete Event Simulation for Multi-Core Systems : Analysis and Optimization Communication Mechanism in the" 25 (6): 2014.

Xiao, Z, B Unger, R Simmonds, and J Cleary. 20–28. "Scheduling Critical Channels in Conservative Parallel Discrete Event Simulation." In *Proceedings of the Thirteenth Workshop on Parallel and Distributed Simulation*, 1999. PADS '99. Washington, DC, USA: IEEE Computer Society. http://dl.acm.org/citation.cfm?id=301429.301452.

AUTHOR BIOGRAPHIES

TIANLIN LI is at National University of Defense Technology with e-mail address: ltl@mail.ustc.edu.cn.
XIAO SONG is at Beihang University with e-mail address: s761127@sina.com.
WENJIE TANG is at National University of Defense Technology with e-mail address : tangwenjie@nudt.edu.cn.
YIPING YAO is at National University of Defense Technology with e-mail address: ypyao@nudt.edu.cn.
FENG ZHU is at National University of Defense Technology with e-mail address: zhufeng@nudt.edu.cn.)