Proceedings of the World Congress on Engineering and Computer Science 2007
WCECS 2007, October 24-26, 2007, San Francisco, USA

Synchronization on Speculative Parallelization of Many-Particle Collision Simulation[*]

Hung-Chi Su[†] Hai Jiang[‡] Bin Zhang[§]

Abstract—**High performance particle simulations are on demand. As more and more computers switch from the traditional uniprocessor architectures to multi-core and multiprocessor parallel architectures, many computer simulations will be migrated to parallel environments to shorten the execution time of simulation. A recently developed speculative parallelization for many-particle collision simulation shows a performance gain. However, as multiple execution threads exist, overheads of common shared resources and data are unavoidable. To reduce these overheads, this paper proposes a new synchronization scheme that aims at improving the simulation efficiency by relaxing the requirements for locks when executing events by multiple threads. Our experiments indicate the performance gains of using this new synchronization scheme for speculative parallelization.**

Keywords: speculative execution, parallel simulation, particle collision

1 Introduction

The behavior of particle systems can be analyzed and simulated by studying interactions among moving particles. Such method has been adopted in atomic physics [1, 2], chemistry [3, 4], and many other academic research fields [5, 6, 7]. A system of partons (quarks and gluons) in the Quark-Gluon Plasma produced in relativistic heavy ion collisions is a typical many-particle system. The algorithm used in the parton cascade model aims to solve Boltzmann equations involving large numbers of particles [1]. Similar to many other simulation applications, this realistic simulation process is computationally intensive. When the number of moving particles in the system is large, it is common for computers to run for days or months to obtain the results. As long as the running time of simulation is concerned, scalable simulation systems are on demand for these so-called *Grand Challenge Problems.*

Since more and more computers switch from the traditional uniprocessor architectures to multi-core and multiprocessor parallel architectures, many computer simulations have been migrated to parallel environments to utilize extra system resources and shorten the execution time [8, 9]. However, there are many restrictions and overheads in the process of parallelizing many-particle collision simulations. The main restriction is the strong serialization characteristic of particle-collision events that prohibits running multiple events simultaneously. To deal with this problem, a speculative parallelization of many-particle collision simulation has been proposed so that more system resources from multi-core and multiple CPUs can be employed to simulate future events aggressively [8]. Based on this speculative parallelization, more flexible speculative parallelization models (dynamic master and hybrid speculative models) have been proposed [9] to improve the simulation efficiency by reducing the idle time in the original speculative parallelization.

As multiple execution threads exist, overheads in sharing common resources and data are unavoidable. The main contribution of this paper is to reduce these kinds of overheads. As a result, we propose a new synchronization scheme that aims to improve simulation efficiency by relaxing the requirements for blocks when executing events by multiple threads.

The paper is organized as follows. Section 2 describes background information. Section 3 illustrates how a normal speculative parallelization works and its data structure. Section 4 analyzes all the possible critical sections on shared objects and then proposes new synchronization scheme. Section 5 provides performance analysis and simulation results to illustrate the performance gains acquired from our new synchronization scheme. The paper ends with some concluding remarks.

2 Preliminaries and Relevant Work

Most particle systems, including particle collision systems [1] and molecular dynamics [3, 4], adopt the more efficient event-driven approach that displaces all the particles over a series of predicted collision events so that simulation time can advance to the next event time directly instead

[*]Manuscript received July 22, 2007. This research was partly supported by the U.S. National Science Foundation under grant No. PHY-0554930.

[†]H. C. Su is with the Department of Computer Science, Arkansas State University, Jonesboro, Arkansas 72401, USA Email:suh@cs.astate.edu

[‡]H. Jiang is with the Department of Computer Science, Arkansas State University, Jonesboro, Arkansas 72401, USA Email:hjiang@cs.astate.edu

[§]B. Zhang is with the Department of Chemistry & Physics, Arkansas State University, Jonesboro, Arkansas 72401, USA Email:bzhang@astate.edu of crawling through all time periods in between as in the time-driven approach. Cell structure [3, 10, 11, 12] is brought into the simulation system to improve the efficiency of selecting the next collision event based on the observation that a future collision for a particle will most likely occur in its neighborhood. Consequently, a new kind of event (boundary crossing event) is introduced to allocate particles to cells.

To simulate all the events one by one, a sequential program for the event-driven approach repeats the following three essential steps for every event until the end of the simulation:

1. finding the new globally next event and advancing the simulation time,
2. processing the new globally next event by updating the states of the particles involved in this event, and
3. computing the next event time for particles involved in Step 2 or affected by this event.

In Step 1, we need to find out the overall next earliest event in the simulation system. A tree-like data structure such as min-heap and complete binary trees can help this step with the time complexity $O(\log N)$ [5, 7, 13, 10] where N is the total event number. The idea is to maintain a global queue mapped to the tree-like data structure for all possible events according to their timestamps. In Step 2, the actual event will be simulated. This step updates the states of the involved particles, including formation time, position, and momentum, etc. Then, the system goes on Step 3 where only the local information of the affected particles need to be updated. Such particles include the ones conducting the event and other affected particles in neighboring cells. After finishes these three steps, system will loop back to Step 1.

The difficulty in parallelizing the many-particle simulations is that collision events exhibit strong serialization characteristics. That is, even if more computers are available, only one event can be simulated and it might invalidate future events. To deal with this problem, speculative parallelization has been proposed [8, 9] so that multiple events can be processed simultaneously. Only one of them is the globally current event and its simulation will be correct for sure. Others are simply processed in advance and need to be verified later. On multiprocessor machines, multi-threading strategy is employed for these events. To share common resources, some threads might be blocked and such idleness slows down the overall simulation. Hence, further fine-grained scheduling strategies, such as optimized locking mechanism, are on demand.

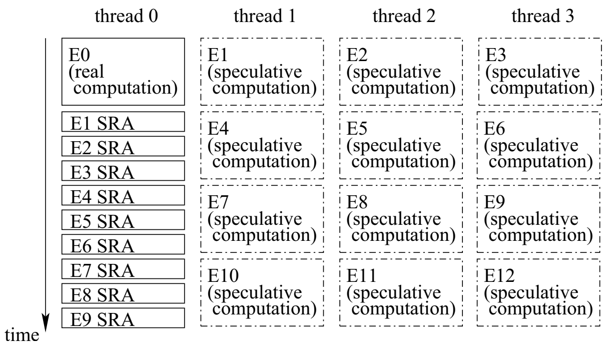
thread 0
thread 1
thread 2
thread 3
E0 (real computation)
E1 (speculative computation)
E2 (speculative computation)
E3 (speculative computation)
E1 SRA
E2 SRA
E3 SRA
E4 SRA
E5 SRA
E6 SRA
E7 SRA
E8 SRA
E9 SRA
E4 (speculative computation)
E5 (speculative computation)
E6 (speculative computation)
E7 (speculative computation)
E8 (speculative computation)
E9 (speculative computation)
E10 (speculative computation)
E11 (speculative computation)
E12 (speculative computation)
time

Figure 1: An ideal scenario of speculative parallelism

3 Speculative Parallelization

Due to the strong serialization among events, it is not appropriate to directly use normal parallelization to execute several events at same time. Speculative parallelization is, therefore, adopted to allow threads executing events simultaneously by ignoring any possible dependency temporarily. Basically, the globally earliest event will be handled first and this process is called *real computation*. At the same time, some future events can be speculatively simulated as well. Such early execution, called *speculative computation*, can take good advantage of extra computing resources (e.g., processors) for further parallelism and possible speedup.

3.1 Real vs. Speculative Computation and Simulation Models

Normally, when multithreading technique is applied, one thread (called master thread) is dedicated to all real computations, and other threads (called speculative threads) are dedicated to speculative computations. The master thread always gets the next earliest event from the global queue in the simulation. When the master thread obtains a next earliest event that has not been speculatively handled, the master thread will start the real computation for this event. On the other hand, when the obtained event has been handled by a speculative thread, the master thread needs to verify the validity of the speculative result for dependency violation. If no dependency violation exists (i.e., the speculation is correct), the master thread only needs to take the speculative result as the real one instead of executing the whole event (see Figure 1). This whole process is called Speculative Result Adoption (SRA), which is much faster than an event execution. Conversely, if a dependency violation is detected, the speculative result should be abandoned, and then this event will be simulated as normal by the master thread (Figure 2). The assumption here is that speculative computation always uses extra computing resources without slowing down the real computation, so the proposed strategy is only good on parallel computers.

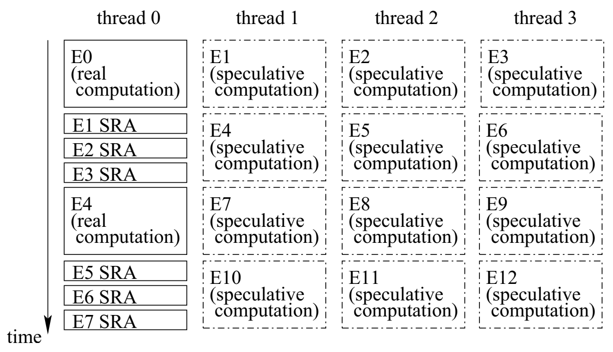
thread 0
thread 1
thread 2
thread 3
E0 (real computation)
E1 (speculative computation)
E2 (speculative computation)
E3 (speculative computation)
E1 SRA
E2 SRA
E3 SRA
E4 (speculative computation)
E5 (speculative computation)
E6 (speculative computation)
E4 (real computation)
E7 (speculative computation)
E8 (speculative computation)
E9 (speculative computation)
E5 SRA
E6 SRA
E7 SRA
E10 (speculative computation)
E11 (speculative computation)
E12 (speculative computation)
time

Figure 2: A scenario with an invalid result (E4) from speculation computation

When many-particle systems exhibit low dependencies among events, the master thread uses most of its time to accomplish SRAs whereas other threads are busy with speculative computations. With the implementation of pointer swapping, the overhead of SRA is much smaller than a regular event simulation. Hence certain performance gains can be expected.

Based on the above discussion, three different models for speculative parallelization have been accordingly developed [9] for improving computational efficiency. The first model is pure speculative model, and the other two are flexible simulation models, called dynamic master and hybrid speculative models. The dynamic master model lets the master thread act as a speculative thread when needed, but the hybrid speculative model may promote a speculative thread as the master thread temporarily. Breadth-first search on the heap by a speculative thread is applied to select an event for speculative computation in all three models, because it is simple and prevent the master thread from constantly waiting when the number of processor is limited [9]. Our new synchronization scheme will be implemented on these three models in order to evaluate its performances.

3.2 Data Structure

In this subsection, we discuss the objects (particle, cell, and heap) and their data structures in a simulation system for later analyses on synchronization.

3.2.1 Particle

A particle has two identical structured records saved. The first record (called *real record*) is to keep the current state of a particle (including its weight, position, momentum, etc.) and this particle's next event time (including next collision time and next boundary crossing time). The second record (called *speculative record*) is used to record this particle's speculative future state and next event time for this future state. After each event simulation (real computation), the real record of each participating particle needs to be updated at Step 2 for its new current state and at Step 3 for its new next event time, but the speculative record stays intact. Similarly, after each speculative computation, the speculative result will be saved in the speculative record for a possible adoption by the master thread without modifying the real record.

3.2.2 Cell

A cell records all its accommodating particles by using a linked list data structure. It also maintains the time (epoch) of the last event occurred in this cell to help the master thread to verify the validity of a speculative result before SRA. After each event simulation (real computation), a cell needs to remove a leaving particle from (or add an entering particle to) its linked list at a boundary crossing event, but it does not change the linked list at a collision event. Meanwhile, epoch needs to be increased by 1 to indicate a new change in this cell.

3.2.3 Heap

A heap is adopted in a simulation with smaller next event time of a particle as higher priority and is implemented in an array with each of elements containing a pointer to one unique particle. Thus, the particle that is pointed by the root of the heap will be involved in the globally next event. Accordingly, the master thread always gets the event pointed by the root of the heap while a speculative thread does breadth-first search for an unhandled event. Actually, a breadth-first search is a scan on the array from the second element (not the first element because the first element is the top of the heap and is always given to master thread). After each event (real computation) is executed by the master thread, the heap needs to be updated (by the master thread) for the removal of the old earliest event and the addition of newly generated events.

4 Synchronization

In a parallelized simulation system, all objects including particles, cells and heap become shared data that may cause race conditions. Applying regular locks in a simulation can enforce mutual exclusion to solve the race conditions, but it could result in long waiting threads. It could be even worse for being deadlocked when not carefully using locks on more than one object at the same time. In this section, we analyze the problems of using locks on an object to synchronize the conflicting threads, and then develop a better synchronization scheme for the speculative parallelization.

4.1 Assumption and Observation

An assumption is established for an operation on a variable of primitive data type: an operation is translated into three consecutive (atomic) instructions in machine language. These three consecutive instructions are: load (from a shared memory to a local CPU register), operate (on this local CPU register), and store (from this local CPU register to the shared memory). Normally, this assumption is true in a typical machine. As a result, when a thread reads a variable that is being executed in CPU, it gets the value of this variable either before or after the operation.

Similarly, lock mechanisms ensure data consistency when any two conflicting threads compete accessing an object. That is, they ensure a thread obtaining data either before or after data locked (then released) by others. However, they do not guarantee that data obtained by threads are in the order of event time. For example, a speculative thread is working on a future event (speculative computation) in advance, and it needs to access the neighboring particles while one of which is being updated by the master thread. This speculative thread will collect up-to-date information of that particle if it accesses the particle after the master thread issued lock and unlock on that particle. Otherwise, it collects obsolete data, which will be re-computed when the master thread obtains and verifies this obsolete result. However, a speculative thread will not collect inconsistent data such that part of the data is before the update and part of the data after the update.

4.2 Locks on an Object

In this subsection, we examine objects and also check how lock and unlock enforces mutual exclusion and the derived problems. In addition, whether removing the lock and unlock to avoid blocking threads is beneficial to the simulation system will be investigated as well.

4.2.1 Heap

In a simulation system, a heap may be updated by a master thread and read by many speculative threads at the same time. Intuitively, the access to the heap should be in a critical section to enforce mutual exclusion between the master and speculative threads. However, while enforcing mutual exclusion, using lock and unlock on the heap may lose the concurrency greatly.

After observing the selection of next event in a speculative thread, we relax the requirement of locks for a thread to access the heap. As stated earlier, the selection for next speculative computation by a speculative thread is a scan on the heap array (breadth-first search), and the chosen particle is not necessarily selected in the order of next event time. Thus, without using locks and unlocks, the speculative thread can still get a unique particle (pointer) to execute speculative computation, although the position of this particle (pointer) in the heap may be changed (by the master thread) after this pointer is selected. In short, removing the lock and unlock to avoid blocking threads will be beneficial to the simulation system.

4.2.2 Particle

Basically, locks and unlocks might be necessary for two different kinds of accesses to a particle. In the first kind of access, no two threads are allowed to do either real or speculation computation on the same particle at the same time during the whole simulation, so a lock is necessary to inform others when a particle is being executed. Note that this lock does not block any other threads from reading the locked particle's information. However, if a particle is involved in the globally next event and it is currently locked (held) by a speculative thread, this lock will block the master thread, which causes the master thread to wait till this lock is released. Two speculative parallelization models have been proposed to prevent the master thread from suffering such waiting time [9].

The second kind of access to a particle happens when speculative threads check particles in the neighboring cells for future collisions at Step 3. The information of the neighboring particles might be changed by the real computation. Thus, locks are necessary to ensure no access conflict between the master and speculative threads, and a priority lock is applied to the master thread to avoid starvation. All these locks can still block the conflicting threads, even though waiting time may be short. Worst of all, they may bring in deadlock (discussed in Section 4.2.3) when threads need to lock cells. To avoid such problems, we will relax the lock requirements in our new lock scheme in Section 4.3.

4.2.3 Cell

A cell records its accommodating particles and the time (epoch) of the last occurring event in it. This information can be updated only by the master thread (at Steps 2 and 3), but it can be read by all threads (at Step 3). Normally, the master thread issues a write-lock (priority lock) on a particle and a write-lock (priority-lock) or two (if boundary crossing) on this particle's resided cell(s) before updates this particle, but a speculative thread issues a read-lock on a particle and a read-lock on this particle's resided cell. By doing so, two problems may occur, which are long waiting threads and deadlock. For example, the master thread will not be granted for a lock on a cell if that cell is locked by a speculative thread (or more speculative threads). In other words, the master thread has

```
1. if (invalid or no speculative result)
2.    speculative computation;
3. swap real and speculative records;
```

Figure 3: Algorithm for real computation and SRA.

```
1. SpeculativeRecord.epoch = RealRecord.epoch + 1;
2. copy data to SpeculativeRecord from RealRecord
       except the value of its epoch;
3. speculative computation for simulation;
4. speculative computation for next event time;
5. SpeculativeRecord.epoch++;
```

Figure 4: Algorithm for speculative computation

to wait till the speculative threads finish all the computations in this resided cell. The waiting time could be long especially when there are many particles in this cell.

The worse case is the possible deadlocks if all the locks are not carefully issued and arranged. A simple solution to this deadlock problem is that the master thread issues the write-lock(s) on the cell(s), not on the particle when updating the particle and cell states, and speculative threads issue read-lock on the cell (and this cell only), not on the particle while they checking neighboring particles for the (speculatively) next collision event. This simple solution can avoid deadlock because a thread can issue one lock only (except the master thread that issues two for a boundary crossing event). However, this coarse-grain synchronization may decrease concurrency and increase threads' waiting time. A constantly long waiting master thread can degrade the performance of a simulation since the master thread is on the critical path of simulation.

4.3 New Scheme for Synchronization

In this section, a new scheme is developed to synchronize the master and speculative threads without creating deadlock and long waiting threads when threads are competing for the same objects. Based on the observation in Section 4.1 that the results of speculative computations are not always valid because the lock mechanisms do not guarantee the executions of speculative computations in their event time order, our new synchronization scheme relaxes the requirements for locks with the guarantee of obtaining consistent information of an object. That is, the scheme ensures the information consistency when a thread retrieves the data of an object.

As discussed in Section 4.2.1, there is no need for locks when threads accessing the heap, we will show the new scheme for synchronization on particles and cells only. Since the master thread is the only one thread to update the information about an object, it always obtains consistent information of an object. Regarding a particle, the master thread is the only one allowed to update the real record of a particle. For the master thread to conduct

```
1. copy pointer of the particle's RealRecord to RecordPtr;
2. RecordEpoch = RecordPtr->epoch;
3. if (RecordEpoch is an odd number)
4.    return -1; //inconsistent data could occur
5. read all the data for computation;
6. if (RecordEpoch == RecordPtr->epoch)
7.    return 0; // has obtained consistent data
8. else  return -1; // inconsistent data
```

Figure 5: Algorithm to read data of a particle.

an event simulation (at Steps 2 and 3), it does either real computation or SRA (Figure 3). When the master thread does the real computation, it can first do the speculative computation (Figure 4), and then SRA, which swaps the pointers of real and speculative records. Consequently, the original real record becomes the current speculative record, and the original speculative record that has the new state of this particle becomes the current real record. Thus, when a thread reads the information of a particle, it can copy the pointer of the real record to its local variable, and then retrieves the data consistently via this local variable even if the master thread is working on this particle simultaneously. There could be a problem, even though rarely occurs, if the particle is updated too frequently and too fast while a speculative thread is reading it. To overcome this problem, a thread can increase particle's epoch twice when it conducts speculative computation, one before and one after any computation (Lines 1 and 5 in Figure 4). For a thread to read a particle, it needs to verify the epoch being even number and unchanged after it finishes reading the data (Lines 3 and 6 in Figure 5).

In addition to the information of a particle, we have to enforce the consistency of the retrieved information for a cell. Similar to the particle access, we use the same scheme to verify the consistency of the obtained data for a cell object by increasing the resided cell's epoch twice when the master thread executes an event simulation. However, there is a special case caused by a boundary crossing event: a leaving particle is added to the linked list of its new resided cell while it is being read by a speculative thread. Consequently, the speculative thread will jump to other cell's linked list and will miss all the rest particles in its original linked list because the speculative thread follows the obtained particle's link to get another one. Under such circumstance, the speculative thread should check the resided cell after finishes accessing a particle, although it is fine (but possibly wasting time) to consider the particles in other cells for speculative computation. Figure 6 gives the algorithm for the master thread to update a particle in a cell and Figure 7 is for a thread to read the particles in a cell.

Similar to the access to a particle, there could be a problem that a cell is changed by the master thread so frequently and so fast to cause the speculative result invalid and make the speculative thread redoing the accesses. A

```
1.  Cell = the particle's resided cell;
2.  Cell.epoch++;
3.  particle's real computation; //Figure 3
4.  if (boundary crossing event) {
5.      remove this particle from Cell.LinkedList;
6.      Cell.epoch++;
7.      Cell = the new cell for this particle;
8.      Cell.epoch++;
9.      add particle to Cell.LinkedList;
10.     Cell.epoch++;
11. }
12. else  Cell.epoch ++;
```

Figure 6: Algorithm to update a particle in a cell.

```
1.  localCellEpoch = Cell.epoch;
2.  if (localCellEpoch is an odd number)
3.      return -1; //inconsistent data could occur
4.  for (ptr=Cell.LinkedList.firstParticle; ptr != NULL;
            ptr = ptr->nextParticle) {
5.      access data pointed by ptr; //Figure 5
6.      if (pointed particle by ptr is in other cell)
7.          go to Line 1; // redo it
8.  }
9.  if (localCellEpoch == Cell.epoch)
10.     return 0; // has obtained consistent data
11. else  return -1; // inconsistent data
```

Figure 7: Algorithm to read particles in a cell.

solution to this problem is to set up a limit number of redoing accesses, because the master thread will redo it anyway.

This scheme relaxes all the locks at the Steps 2 and 3, so a thread will not be blocked by others when accessing the information of an object. To guarantee the data consistency, a speculative thread might possibly (but rarely) redo accessing objects, which is a small price to pay for the new synchronization scheme. However, the benefit of this scheme is much greater than the cost.

5 Empirical Results and Analysis

In this section, the experiment results are summarized and compared between two schemes of synchronization (the simple solution in Section 4.2.3 and our newly proposed scheme in Section 4.3). For the purpose of evaluating the performances of these two schemes, a variety of experiments have been conducted on the three speculative parallelization models (pure speculative, dynamic master, hybrid speculative models) with a simplified homogeneous system under fixed system space (10 units), fixed number of cells ($14 \times 14 \times 14$), fixed cut-off sized particles (0.5 unit), and fixed momentum magnitude of particles. All the experiments in this study have been performed on a Sun Server E440 with four CPUs (1.28 GHz each) and 16GB-main-memory for 100 seconds in the simulated systems.

Since there are only four CPUs in this machine, additional threads will not acquire any performance gain, because the machine can serve up to four threads at one time, and all other unserved threads have to be idle. On the contrary, the additional threads will slow down the simulation because of the high overhead of context switch.

To compare the performances between these two schemes on the three speculative parallelization models, we run programs for two, three and four threads, respectively. The simple solution scheme in these figures are denoted by "with cell locks" to indicate its using only cell locks to access neighboring particles, our newly proposed scheme is denoted by "without cell locks" to indicate no lock is used to access neighboring particles. Figures 8, 9 and 10 illustrate the simulation performances of these two schemes for all three models. All the experiments on these three speculative parallelization models suggest that our proposed scheme substantially outperforms the simple solution. In short, it is very beneficial to relax the requirements for locks when updating a particle's state by a master thread and when accessing neighboring particles by any threads in a simulation system.

Number of Particles	Pure Speculative Model *with/without* Cell Locks		Dynamic Master Model *with/without* Cell Locks		Hybrid Speculative Model *with/without* Cell Locks	
	With	Without	With	Without	With	Without
1,000	7.54	5.85	6.07	4.93	6.12	5.08
2,000	27.68	22.72	21.58	18.06	21.70	18.47
3,000	67.12	56.72	50.08	43.00	50.35	43.57
4,000	123.02	107.26	93.90	82.65	94.02	82.77
5,000	214.18	190.63	156.58	140.17	157.02	139.70
6,000	332.43	301.57	240.06	218.54	240.95	215.78
7,000	475.40	438.03	345.75	319.65	347.25	312.61
8,000	639.01	596.98	476.26	445.01	477.50	432.57
9,000	870.72	821.99	634.32	598.65	636.22	579.27

Figure 8: Performance comparisons when using 2 threads

Number of Particles	Pure Speculative Model *with/without* Cell Locks		Dynamic Master Model *with/without* Cell Locks		Hybrid Speculative Model *with/without* Cell Locks	
	With	Without	With	Without	With	Without
1,000	4.63	3.95	4.45	3.95	4.59	4.23
2,000	18.01	14.46	15.50	12.80	15.93	13.30
3,000	40.77	34.07	35.30	29.93	36.25	30.96
4,000	77.15	66.52	65.74	57.08	67.04	58.36
5,000	130.51	115.39	108.59	96.38	110.68	97.68
6,000	201.95	182.04	165.65	149.47	168.24	150.07
7,000	291.28	267.03	237.78	217.81	241.52	216.93
8,000	399.55	371.65	326.70	303.08	330.93	299.57
9,000	536.80	504.95	433.79	406.65	439.63	399.92

Figure 9: Performance comparisons when using 3 threads

Number of Particles	Pure Speculative Model *with/without* Cell Locks		Dynamic Master Model *with/without* Cell Locks		Hybrid Speculative Model *with/without* Cell Locks	
	With	Without	With	Without	With	Without
1,000	4.77	4.66	4.79	4.62	4.97	4.83
2,000	12.85	13.05	12.89	12.96	13.3	13.98
3,000	30.10	25.35	28.69	24.38	29.72	25.33
4,000	56.89	49.44	53.02	46.37	54.69	47.74
5,000	95.44	84.79	87.15	77.58	89.65	79.42
6,000	147.35	133.27	131.82	119.74	135.83	121.20
7,000	212.71	195.38	189.58	174.19	193.63	174.37
8,000	292.85	273.30	258.37	239.10	264.10	239.38
9,000	391.24	369.33	343.65	323.05	350.16	318.96

Figure 10: Performance comparisons when using 4 threads

6 Conclusions

As more and more computers switch from the traditional uniprocessor architectures to multi-core and multiprocessor parallel architectures, many computer simulations will be migrated to parallel environments to shorten the execution time of simulation. Recently developed speculative parallelization for many-particle collision simulation shows remarkable performance gain. However, as multiple execution threads exist, overheads of common shared resources and data are unavoidable. To improve the simulation efficiency, this paper proposes a new synchronization scheme by relaxing the requirements for locks when executing events by multiple threads.

For evaluating the performances of these two schemes, a variety of experiments have been conducted on the three speculative parallelization models (pure speculative, dynamic master, hybrid speculative models). From the experiment results, the performances of our proposed synchronization scheme substantially outperforms the simple solution (in Section 4.2.3). In short, it is very beneficial to relax the requirements for locks when updating a particle's state by a master thread and when accessing neighboring particles by any threads in a simulation.

The future work includes further optimization of master thread and speculative computation: the focus will be on time-consuming operations including collision calculation and the scanning neighboring cells, as well as distributing the computation over cluster machines with or without multi-core CPUs.

References

[1] B. Zhang, "ZPC 1.0.1: a parton cascade for ultra-relativistic heavy ion collisions," *Computer Physics Communications*, vol. 109, pp. 193–206, 1998.

[2] M. Feix, A. K. Hartmann, R. Kree, J. Muñoz-García, and R. Cuerno, "Influence of collision cascade statistics on pattern formation of ion-sputtered surfaces," *Physical Review B*, vol. 71, no. 12, p. 125407, 2005.

[3] B. J. Alder and T. E. Wainwright, "Studies in molecular dynamics. I. general method," *Journal of Chemical Physics*, vol. 31, no. 2, pp. 459–466, 1959.

[4] M. Allen, D. Frenkel, and J. Talbot, "Molecular dynamics simulation using hard particles," *Comput. Phys. Rep*, vol. 9, pp. 301–353, 1989.

[5] B. D. Lubachevsky, "How to simulate billiards and similar systems," *Journal of Computational Physics*, vol. 94, no. 2, pp. 255–283, 1991.

[6] R. Szeliski and D. Tonnesen, "Surface modeling with oriented particle systems," in *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pp. 185–194, 1992.

[7] D.-J. Kim, L. J. Guibas, and S. Y. Shin, "Fast collision detection among multiple moving spheres," *IEEE Transactions on Visualization and Computer Graphics*, vol. 4, no. 3, pp. 230–242, 1998.

[8] R. Li, H. Jiang, H. C. Su, J. Jenness, and B. Zhang, "Speculative parallelization of many-particle collision simulations," in *Proceedings of the 2007 International Conference on Parallel and Distributed Processing Techniques and Applications*, pp. 128–136, 2007.

[9] H. C. Su, H. Jiang, and B. Zhang, "Flexible speculative parallelization of many-particle collision simulations." Submitted for publication, 2007.

[10] H. Sigurgeirsson, A. Stuart, and W.-L. Wan, "Algorithms for particle-field simulations with collisions," *Journal of Computational Physics*, vol. 172, no. 2, pp. 766–807, 2002.

[11] H. C. Su, H. Jiang, and B. Zhang, "An empirical study on many-particle collision algorithms," in *Proceedings of the 22nd International Conference on Computers and Their Applications*, pp. 12–17, 2007.

[12] H. Jiang, H. C. Su, and B. Zhang, "Toward optimizing particle-simulation systems," in *Proceedings of International Conference on Computational Science*, pp. 286–293, 2007.

[13] M. Isobe, "Simple and efficient algorithm for large scale modecular dynamics simulation in hard disk system," *International Journal of Modern Physics*, vol. C, no. 10, pp. 1281–1293, 1999.