Journal of Computational Physics **179,** 539–556 (2002)
doi:10.1006/jcph.2002.7073

A Multiple-Heaps Algorithm for Parallel Simulation of Collision Systems[1]

Mo Mu

Department of Mathematics, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong
E-mail: mu@math.ust.hk

Received July 19, 2001; revised March 18, 2002

We consider the parallel simulation of collision systems. It has wide application, such as in hard-sphere molecular dynamics simulation for gas dynamics and crystals, as well as in studying molecular collision dynamics of chemical reactions. With detailed analysis, proper data structures are designed so that the central computational task is formulated as a consecutive search for the minimum in the collision time space of $O(N^2)$ entries, with multiple updates on $O(N)$ entries in the same space per collision step. The abstraction and formulation enable us to incorporate efficient techniques in computer science into this application, which leads to a heap-based sequential algorithm of $O(N \log N)$ time in one typical collision step, where N is the number of particles of the simulated collision system. A parallel algorithm of multiple heaps with a diagonal-oriented mapping is then proposed. We show that the parallel algorithm is load balanced and the parallel time per collision step is $O((N/P)\log(N^2/P) + \log P)$, where P is the number of processors. The parallel algorithm uses two levels of partitioning independently, one in the particle-based physical space and the other in the collision time space. An exchange-shift communication algorithm is presented to bridge the two different partitioning schemes. Besides collision system simulation, the parallel multiple heaps algorithm may find applications in many other computing areas where a heap-based priority queue needs to be maintained, such as in fast level-set methods. © 2002 Elsevier Science (USA)

Key Words: parallel computing; collision system; hard-sphere molecular dynamics simulation; heap; parallelism; load balance.

1. INTRODUCTION

We consider the parallel simulation of collision systems. It has wide application, such as in hard-sphere molecular dynamics simulation for gas dynamics [4, 5] and crystals

[1] This work was supported in part by Hong Kong RGC Competitive Earmarked Research Grant HKUST593/94E.

AP

0021-9991/02 $35.00
© 2002 Elsevier Science (USA)
All rights reserved.

[1, 3, 8, 22], as well as in studying molecular collision dynamics of chemical reactions [7]. For the illustration of general characteristics and technical issues, we present the problem in the context of a hard-sphere molecular dynamics simulation.

Molecular dynamics simulation is classified as soft-sphere and hard-sphere types in terms of different models of molecular systems. Efficient algorithms have been developed for soft-sphere molecular dynamics simulation based on the multipole technique [11]. The multipole technique is used in the potential energy evaluation that is the major computational part in soft-sphere molecular dynamics simulation. In contrast, in a hard-sphere molecular dynamics simulation, the most time-consuming computation in a typical collision step is to identify at each simulation step a particle pair with the least *collision time*. This pair of particles will collide with one another after the computed collision time and the corresponding velocities of the colliding particles will change due to the collision. Thus the pairwise collision time status of the entire system will be updated, and a new search will be conducted for the next colliding pair. With detailed analysis, proper data structures are designed so that the central computational task is formulated as a consecutive search for the minimum in the collision time space of $O(N^2)$ entries, with multiple updates on $O(N)$ entries in the same space per collision step. The abstraction and formulation enable us to incorporate efficient techniques in computer science into this application, which leads to a heap-based sequential algorithm of $O(N \log N)$ time in one typical collision step, where N is the number of particles of the simulated collision system. In practice, parallel computing is essentially necessary for a realistic molecular dynamics simulation. Based on the sequential min-heap algorithm, we further propose a parallel algorithm with multiple heaps and a diagonal-oriented mapping. The algorithm is highly parallel and load balanced. The parallel time per collision step is $O((N/P)\log(N^2/P) + \log P)$, where P is the number of processors. The parallel algorithm uses two levels of partitioning independently: one in the particle-based physical space and the other in the collision time space. An exchange-shift communication algorithm is presented to bridge the two different partitioning schemes.

Ideal gas in a container is a typical example of a hard-sphere collision system [15]. The earliest work of using computer simulation of molecular collisions to study gas dynamics can be tracked back to decades ago in [4, 5]. Another exciting application example of hard-sphere molecular dynamics simulation is the study of crystals, where harsh repulsive forces determine the structural properties of a simple liquid and attractive forces are in a sense of secondary importance [8]. Besides applications of hard-sphere molecular dynamics simulation, collision systems also appear in the study of molecular collision dynamics of chemical reactions [7]. Although our parallel multiple heaps algorithm is presented in the context of collision systems, it may also find application in many other computing areas where a heap-based priority queue needs to be maintained. For example, in fast level-set methods [19], a heap is used to quickly locate the lowest point to start with for constructing the static level-set surface. One can build such a surface piece by piece in parallel, with certain synchronization along the boundaries, by using our multiple-heaps technique.

2. PROBLEM SETTING AND A SEQUENTIAL MIN-HEAP ALGORITHM

In this section, we analyze the computational aspects of collision systems in the context of hard-sphere molecular dynamics simulation. For the physical aspects and applications, we refer readers to [3–5, 8, 12]. With detailed analysis, we design proper data structures

so that the central computational task is formulated as consecutive minimum searches with multiple updates per collision step. The formulation will enable us to devise efficient algorithms using suitable computer science techniques.

Consider a hard-sphere molecular system of N particles, each with its position and velocity assigned initially. For simplicity, we assume that all the particles are of the same size and that particle collisions are elastic and between two bodies. We also assume that attractive forces among particles are negligible. We do not specifically address boundary effects because a collision between a particle and the boundary can be treated by viewing the boundary as a virtual particle, which makes the computation uniform for both particle–particle and particle–boundary collision types.

Let $\mathbf{r}_i(t)$ and $\mathbf{v}_i(t)$ be the position and velocity vectors of particle p_i at time t. Given a status $PosVel(t) = \{(\mathbf{r}_i(t), \mathbf{v}_i(t)) \mid i = 1, 2, \ldots, N\}$, by Newton's law all particles will move straight without changing velocity until a collision happens between a pair of particles called the *colliding pair*. For each particle pair (p_i, p_j), $i, j = 1, \ldots, N, i \neq j$, define $\delta t_{ij}(t)$ as the *collision time* such that p_i and p_j will collide with one another at time $t + \delta t_{ij}(t)$. It is easy to calculate $\delta t_{ij}(t)$ from $(\mathbf{r}_i(t), \mathbf{v}_i(t))$, $(\mathbf{r}_j(t), \mathbf{v}_j(t))$ and the particle size as follows [12]. For simplicity, we omit the time variable t when there is no confusion in the context. Let $\mathbf{v}_{ij} \equiv \mathbf{v}_i - \mathbf{v}_j$ and $\mathbf{r}_{ij} \equiv \mathbf{r}_i - \mathbf{r}_j$ be the relative velocity and position between particles p_i and p_j. If

$$\mathbf{v}_{ij} \cdot \mathbf{r}_{ij} \geq 0, \tag{2.1}$$

p_i and p_j are moving away from one another, or moving in the same direction and with the same velocity so that they will never collide with one another. In this case, we define $\delta t_{ij} = +\infty$ for the particle pair. Otherwise, they are approaching one another, and then we test the condition

$$\Delta \equiv (\mathbf{v}_{ij} \cdot \mathbf{r}_{ij})^2 - |\mathbf{v}_{ij}|^2(|\mathbf{r}_{ij}|^2 - \sigma^2) < 0, \tag{2.2}$$

where σ is the sphere diameter. If (2.2) is true, p_i and p_j will pass by each other without collision so that we also define $\delta t_{ij} = +\infty$. Otherwise, a collision will happen between p_i and p_j after time

$$\delta t_{ij} = \frac{-\mathbf{v}_{ij} \cdot \mathbf{r}_{ij} - \sqrt{\Delta}}{|\mathbf{v}_{ij}|^2}. \tag{2.3}$$

A geometric argument can also be applied for the particle–boundary collision. Notice that $\delta t_{ij} = \delta t_{ji}$ so only the upper triangular part of a two-dimensional table is needed for representing the collision time data while the diagonal part may be used for the particle-boundary collision times. We denote the collision time data set by $ColTime(t)$.

Denote $\Delta t = \min\{\delta t_{ij}(t) \mid \delta t_{ij}(t) \in ColTime(t)\}$. Starting from the current time t, the system will have no collision until time $t + \Delta t$. Over this time period, each particle moves at a constant velocity and the position update is given by

$$\mathbf{r}_i(t + \Delta t) = \mathbf{r}_i(t) + \Delta t * \mathbf{v}_i(t), \quad i = 1, \ldots, N. \tag{2.4}$$

At time $t + \Delta t$, the pair (p_i, p_j) corresponding to the minimum collision time Δt collide with one another, which leads to a velocity change for the colliding pair while all other

particles retain their precollision velocities. The corresponding postcollision velocities for p_i and p_j can be computed according to the formulae

$$\mathbf{v}_i(t + \Delta t) = \mathbf{v}_i(t) - \Delta \mathbf{v}_{ij}, \tag{2.5}$$

$$\mathbf{v}_j(t + \Delta t) = \mathbf{v}_j(t) + \Delta \mathbf{v}_{ij}, \tag{2.6}$$

where

$$\Delta \mathbf{v}_{ij} = [\mathbf{v}_{ij}(t) \cdot \mathbf{r}_{ij}(t)]\mathbf{r}_{ij}(t)/|\mathbf{r}_{ij}(t)|^2. \tag{2.7}$$

Notice that only two velocity entries have to be updated in *PosVel*(t) to obtain *PosVel*$(t + \Delta t)$ in one time-step advance. Furthermore, if a particle p_k is not involved in any collision in certain consecutive time advance steps over a total period ΔT time, then besides its velocity remaining unchanged, its position movement is also accumulative and can simply be calculated without intermediate updates by

$$\mathbf{r}_k(t + \Delta T) = \mathbf{r}_k(t) + \Delta T * \mathbf{v}_k(t). \tag{2.8}$$

More important, for all particle pairs not involved in a collision, the corresponding collision times are decreased by the same amount Δt. Namely,

$$\delta t_{kl}(t + \Delta t) = \delta t_{kl}(t) - \Delta t, \quad k \text{ and } l \neq i, j, \tag{2.9}$$

where p_i and p_j are the colliding particles. Therefore, the relative relationship among all the noncolliding particles in the least-collision-time search procedure is not affected by the collision either.

These observations allow us to design a more efficient, though slightly more sophisticated, data structure for *PosVel* and *ColTime*, where each entry in *PosVel* and *ColTime* stores the relative value with respect to the time when the corresponding particle (or particles) was involved in the latest collision. In other words, different entries usually refer to the data at different times instead of the current (wall clock) time t. Notice that in the least-collision-time search procedure, only the relative relationship of collision times matters. So in a collision update for *ColTime*, instead of decreasing the entry values for the $O(N^2)$ noncolliding pairs as in (2.9), we equivalently modify the entry values for the other $O(N)$ pairs by the same increment Δt. Namely,

$$\delta t_{kl}(t + \Delta t) = \overline{\delta t_{kl}}(t + \Delta t) + \Delta t, \quad k \text{ or } l = i, j, \tag{2.10}$$

where $\overline{\delta t_{kl}}$ is calculated according to (2.3). Notice that these entries need to be updated anyway due to the corresponding velocity change. With this data structure, an entry needs to be updated only when the corresponding particle (or particles) is involved in a collision. To retrieve the corresponding data at time t for an entry in *PosVel* and *ColTime* easily and efficiently, we only need to introduce two auxiliary data structures: one recording the history of collision times $\{\Delta t_m\}$, and the other marking for each particle the reference collision time m. Despite the actual implementation, to simplify the presentation we still assume in the rest of the paper that all entries in *ColTime* hold the collision time data referring to the same time t although they are actually retrieved from the relative values

through indirect addressing and simple manipulation. On the other hand, from the above discussion we can essentially assume that if p_i and p_j collide with one another, only those entries $\{\delta t_{kl} | k \text{ or } l = i, j\}$ in *ColTime* have to be updated. For the sake of intuition, we also use an $N \times N$ two-dimensional array as the virtual data structure for *ColTime* although only the upper triangular part that is physically mapped to a smaller two-dimensional array in the implementation is needed.

Now we are in a position to formulate the problem in an abstract form. In a hard-sphere molecular dynamics simulation, a typical collision step is as follows. The first and central computational task is to identify the next colliding pair. This is determined by a minimum search in *ColTime*:

$$\Delta t = \min\{\delta t_{ij} \mid i, j = 1, \ldots, N, i \leq j\}. \tag{2.11}$$

Once Δt and the corresponding colliding pair (p_i, p_j) are identified, the related entries in *PosVel* are updated for the particle movement with only $O(1)$ time. Finally the entries in *ColTime*, as circled in Fig. 1 (for simplicity we omit δ in δt_{ij}), are updated using (2.10) with $O(N)$ time due to the velocity change caused by the collision between p_i and p_j. We denote this subset of *ColTime* by $\mathbf{Q_{ij}}$ and call it the *updating set*. The simulation is then ready to advance to the next step. In a practical simulation, observable macroscopic properties such as instantaneous temperature and pressure are computed at certain time intervals. These quantities are defined in statistical mechanics as certain averages in terms of the microscopic position and velocity over all particles in the simulated system at the corresponding time. Such a computation is of $O(N)$ time and is needed only once for a large number of collision steps depending on the relaxation time of the system. Therefore, the main concern is to devise efficient algorithms for the consecutive minimum search problem (2.11) with multiple updates (2.10) per collision step. We consider the sequential case in this section and the parallel computation in the next section.

Problem (2.11) itself is straightforward if it is considered as a concrete minimum search procedure without other collision steps and entry updates. A brute-force approach is to

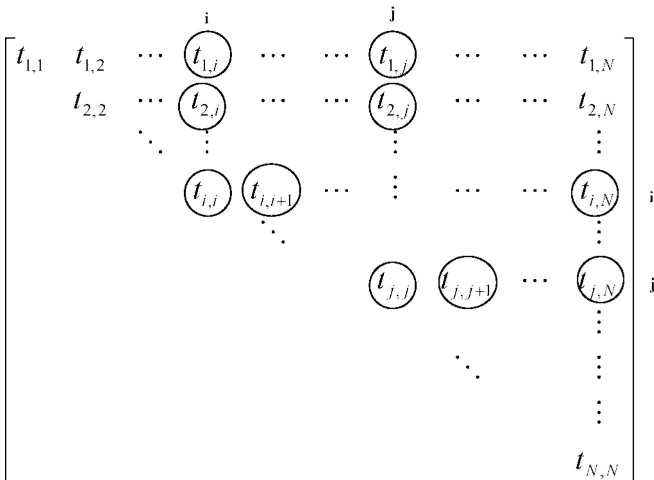

FIG. 1. The table (upper triangular) representation of *ColTime*. The updating subset $\mathbf{Q_{ij}}$ consists of the circled entries to be updated due to a collision between particles p_i and p_j.

use a table data structure *ColTime.Table* to represent *ColTime*. It is easy to update an arbitrary entry in *ColTime.Table* due to the simplicity of the table data structure. The total update cost is $O(N)$ per step, as seen from Fig. 1 and (2.10) because only the entries in the updating set $\mathbf{Q_{ij}}$ need to be updated if p_i and p_j collide with one another. However, since the data are not sorted in *ColTime.Table*, the minimum cost for searching Δt defined by (2.11) is $O(N^2)$. Thus the total cost is $O(N^2)$ in a typical collision step. There is a comprehensive survey in [12, 21] (also see references cited therein) on techniques for improving the computational efficiency. One is the so-called *single-event* (or *soonest-to-occur event*) approach, where the data in *ColTime* are partially sorted with respect to each particle. Namely, for each row in *ColTime*, only the minimum is stored in a *Time List* data structure of size N. Now, the global minimum search for Δt is done with *Time List* in $O(N)$. It is also very memory effective, which is important for serial computers, for *ColTime* is not explicitly stored. However, the difficulty is moved to the update procedure for *TimeList* after each collision. Since any entry in the list might be affected by a collision, an outer loop over the N entries is necessary to check for possible updates. Once an entry is identified for updating a particle, an inner loop over all the other particles is needed to search for the new minimum and to update the entry because *ColTime* is not stored. Therefore, the theoretical worst-case time complexity is still $O(N^2)$ for this part due to the double loops (see, for example, subroutine **HSUPDT** on p. 424 in [12]), although some or many of the inner loops may not be carried out in practice, depending on different collision steps, applications, as well as models used. The practical performance and the average factor in front of N are discussed in [21] for several chain dynamics applications of polymeric fluids using the Rapaport model [18]. Another technique to avoid the inner loop search over all particles is to keep a *Neighbor List* for each particle based on the assumption that only particles in close proximity are destined to collide. This leads to more complexity and uncertainties: First, an optimum geometric range must be determined in order to define the neighbor list; second, the collision time depends on not only the distance but also on the relative velocity between each pair of particles; third, neighbor list expiration is checked periodically (i.e., every 100 collisions for some applications [21]); and fourth, neighbor list construction/renewal is expensive even using efficient data structures, such as link lists. These factors are again problem dependent. Fine-tuning the code for each particular application is a challenging task. The other major approach is the so-called *multiple-events* (or *soon-to-occur events*) method. Instead of storing only the soonest-to-occur event for each particle, multiple soon-to-occur events are stored for each particle in an unsorted order, which makes the timetable update (i.e., addition and deletion of an entry) easier with the expense of more memory. The efficiency for searching Δt in this timetable depends on the product of N and the number of events N_e, which again is problem dependent. On the other hand, with a fixed N_e, the smaller N_e is, the more expensive the timetable is to maintain (i.e., when the soon-to-occur events set becomes empty or full for a particle), and the extreme case is the single-event algorithm where $N_e = 1$. If N_e is allowed to vary dynamically, say by implementing the multiple events by a linked list for each particle, then searching would become more expensive. Experiments in [21] show that the single-event algorithm is approximately 75% faster than the multiple-events algorithm for molecular fluids but yields equivalent performance for atomic fluids. More important, these algorithms are designed in the sequential setting and the optimization techniques employed make the algorithms very complicated and difficult for parallel computing.

It is known that there are various sorting and searching techniques available in computer science. We propose a min-heap algorithm suitable for this application by implementing the collision time space as a priority queue. The algorithm has the optimal complexity $O(N \log N)$ per collision step. Recall that in a priority queue, the element to be deleted is the one with the lowest (or highest) priority. At any time, an element with arbitrary priority can be inserted into the queue. Heap is an important data structure. It finds application when a priority queue is to be maintained. A *min heap* is defined as a *complete binary tree* that is also a *min tree* in which the key value in each tree node is no larger than the key values in its children (if any). For a priority queue application, two standard operations on a heap are *insertion* and *deletion*. Both of them can be implemented by $O(\log M)$ algorithms for a heap of M nodes [13]. Specifically, the insertion algorithm starts with appending the new node to the tail of the complete binary tree and then pops it up along the path toward the root until the property of a min tree is satisfied, namely until the key value of the floating node is larger than that of its parent node. Similarly, the deletion algorithm starts by replacing the root, the node to be actually deleted, with the node at the tail position of the heap and then deletes the tail node to have a complete binary tree of one fewer node, and then it makes the new root node sink down to the children until the min tree property is satisfied, namely until the key value of the sinking node is no larger than that of its children. Another operation on a heap changes the key of a particular node without affecting the tree structure. It is often called *decrease key* and *increase key* in the literature, and we will call it *update key* in our context. This operation is used, for example, in the sparse graph version of Dijstra's algorithm, also called Johnson's algorithm [2, 14]. A similar $O(\log M)$ algorithm is also available by combining the techniques in both insertion and deletion operations. Notice that an update action does not change the tree structure of a heap. So only node swapping is needed in order to maintain the min tree property after such a key-value update. Therefore, the main idea in the *update-key* algorithm is to determine the moving direction of the given *updating node* in such a node-swapping procedure, which can be done by comparing the new key value of the updating node with those of its parent and child nodes. Once we know that the updating node needs to move up or down, the corresponding node-swapping procedure in *insertion* or *deletion* can then be applied similarly. The implementation of these algorithms is described in Section 4.

Applying this technique to our case, we represent the collision time space by a min heap, instead of a table. Since the data are so sorted in a min heap, it requires only $O(1)$ time for identifying the colliding pair from the root node by using collision time as the sorting key and adding the corresponding particle indices to the node data field. A collision between particles p_i and p_j results in a sequence of key updates for all entries in the updating set $\mathbf{Q_{ij}}$. There are $O(N)$ *update-key* operations applied to the min heap of $O(N^2)$ nodes in each typical collision step, so the total time is $O(N \log N)$, where the $\log N$ factor in the worst-case upper bound comes from the height of the heap. Similar to the single-event approach where the inner loops may be skipped for some of the particles in the time-list updating procedure, many of the *update-key* operations do not need to go through a complete path from the root to a leaf node. Furthermore, in the following parallel algorithm, in addition to the distribution of the *update-key* operations to many processors, the tree height for each local heap is also smaller due to the fact that the global tree is equally distributed among all the processors, which will lead to a very efficient parallel algorithm.

3. A PARALLEL ALGORITHM OF MULTIPLE HEAPS

This section studies parallel simulation of collision systems. We first discuss the difficulties and conflicts in the trade-off of efficiency versus parallelism as well as computation versus communication for the parallel setting. Once these are well analyzed and understood, it becomes clear how to find a suitable way to solve the problem, which leads to a parallel algorithm of optimal parallel efficiency. The parallel implementation and performance studies are also presented as the parallel algorithm is developed.

Notice that procedure *update key* is essentially sequential and difficult for parallelization. However, we observe that it is a single-node updating procedure, while in a typical collision step the complete task is to update a total of $2N$ such heap nodes. Namely, procedure *update key* is invoked repeatedly, corresponding to all entries in the updating set $\mathbf{Q_{ij}}$, which suggests certain potential parallelism. For instance, if two updating nodes belong to different subtrees the corresponding min-heap updates within the subtrees can be carried out independently. But synchronization is still needed when the subtrees merge together, and after that the parallelism is lost again. In addition, identification and management of these subtrees with many updating nodes would considerably increase the complexity and difficulty of the parallel procedure. Related work on parallel heap algorithms can be found in [6, 9, 10, 17], where concurrent operations on a single heap are allowed to a certain extent by a window-lock technique.

Another difficulty of parallelism is in the mismatch between partitioning particles and parallelizing the minimum search procedure. It is natural to partition the whole set of N particles into P disjoint subgroups $\{G_k\}$ and assign each subgroup G_k to a processor P_k, where P is the number of processors used in the parallel computation. With this *physical space partitioning*, it is perfect to parallelize all the particle-based computation, such as updating *PosVel* and calculating macroscopic physical properties by statistics. The minimum search procedure, however, is carried out in the pairwise collision time space. Although a local search for the local particles in each G_k could be carried out independently in processor P_k, the assembled data set does not sample the whole collision time space *ColTime*, only the part corresponding to certain diagonal blocks of *ColTime.Table*. The rest of the search for the global minimum would require substantial communication.

We propose a parallel algorithm for simulating collision systems that adopts two levels of partitioning: one in the physical space and the other in the collision time space. With the physical space partitioning as described above, the position–velocity data set *PosVel* is also partitioned into P subgroups $\{PosVel_k\}$ and distributed to the processors correspondingly. This allows full parallelism for all the particle-based computation. We now describe the *collision time space partitioning* that also allows high parallelism for simultaneously updating all the $2N$ entries in the updating set $\mathbf{Q_{ij}}$ among the $O(N^2)$ entries in the collision time set *ColTime* and locating the next colliding pair in $O((N/P)\log(N^2/P) + \log P)$ time. Finally, we should address the connection between the two partitionings, where the main difficulty is in communication.

From the previous discussions, the particle-based partitioning suffers from intensive communication for the global minimum search over the collision time space. On the other hand, the sequential min-heap algorithm offers satisfactory efficiency but is not suitable for parallelism. Thus, in addition to the physical space partitioning we propose also to partition the collision time set *ColTime* into P subgroups $\{ColTime_k\}$ and distribute them to the P processors with each subgroup $ColTime_k$ residing in P_k, and the global minimum

search task can then be split into two subtasks. First, each processor P_k conducts a search procedure over $ColTime_k$ to find its local minimum Δt^k. Then the global minimum is given by $\Delta t_{ij} = \min\{\Delta t^k \mid k = 0, 1, \ldots, P-1\}$, which can be easily implemented by a standard global operation in $O(\log P)$ time that is available on any distributed memory parallel computer. The local minimum search step is fully parallel, and each local minimum problem can be efficiently solved using the sequential min-heap algorithm presented in the previous section, where a local min heap for $ColTime_k$ is maintained by processor P_k. In other words, instead of maintaining a single heap for the entire collision time space we actually use P-distributed min heaps over all the processors. The parallel algorithm has only one synchronization point at the *global min* step. This solves the parallelization problem perfectly.

Next, the load balance, or equivalently the parallel efficiency, depends on two factors: the distribution of the collision time data set *ColTime* and the distribution of the updating data set $\{\mathbf{Q_{ij}}\}$ among the processor. Let M_k be the number of entries in $ColTime_k$, i.e., the number of nodes of the local min heap $Heap_k$ in processor P_k. The load balance for both memory and heap updating would require a balanced partitioning of the node set *ColTime* so that $M_k = O(N^2/P)$. Notice that the data set *ColTime* corresponds to the upper triangular part of the two-dimensional array *ColTime.Table*, as shown in Fig. 1. Intuitively it is easier to uniformly partition a rectangular array into P subsets, say by a column (or row)-oriented wrapping (or block) partitioning. So we first map the upper triangular part of *ColTime.Table* to a virtual rectangular array. This can be done in different ways. For example, as shown in Fig. 2, in a *column-oriented virtual mapping* we can merge the columns of the upper triangular part of

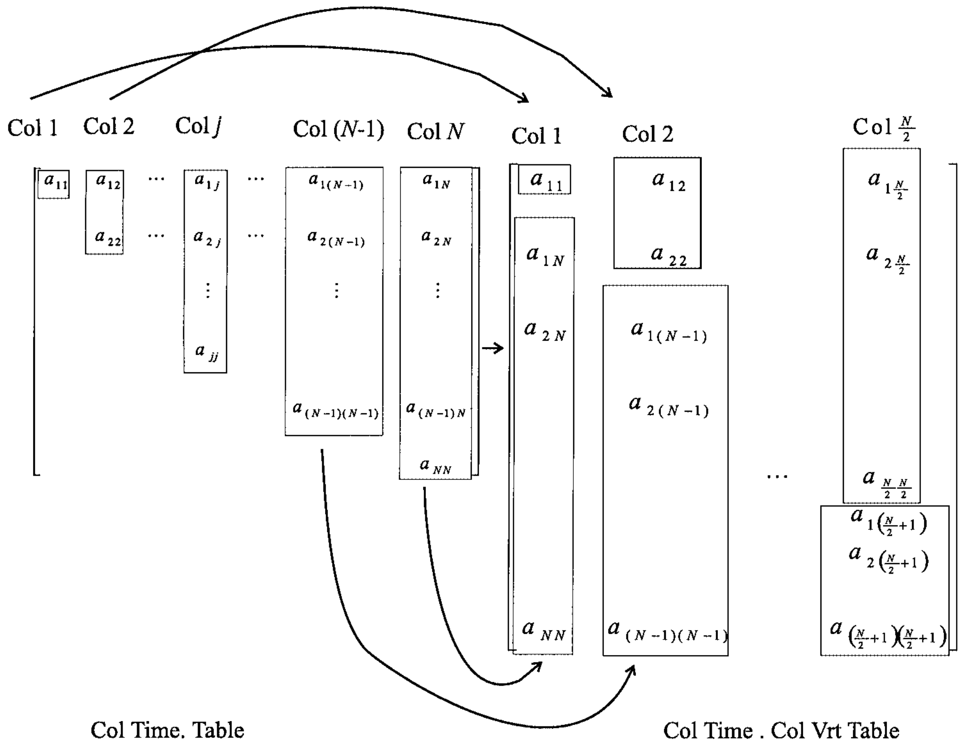

FIG. 2. The column-oriented virtual mapping of *ColTime.Table* (the upper triangular part) to *ColTime.-ColVrt. Table*.

ColTime.Table into pairs in the order of $(col1, colN)$, $(col2, col(N-1))$, and so forth, to obtain an $(N+1) \times (N/2)$ rectangular matrix represented by a two-dimensional array data structure *ColTime.ColVrt.Table.* For simplicity, we assume here that N is even. When N is odd, we simply leave half of the last column empty and unused in *ColTime.ColVrt.Table*. Similarly, one can have a row-oriented mapping. Then a uniform stripe partitioning can be applied to the virtual rectangular data structure straightforwardly. With this uniform partitioning of *ColTime*, we have $M_k = N(N+1)/2P$ for $k = 0, 1, \ldots, P-1$. Therefore, the update time is $O(\log M_k) = O(\log(N^2/2P))$ per updating node in each processor.

Taking a closer look at the above column (or similarly row)-oriented virtual mapping, we notice that the processors holding columns (or rows) i and j in *ColTime.Table* are responsible for updating many more nodes than others. This implies that these processors have to execute the local min-heap updates many more times than do others, although all local heaps are of the same size. So this factor also causes severe load imbalance in computation. Let J_k be the number of updating nodes in the updating set $\mathbf{Q_{ij}}$ that are assigned to processor P_k. Then the total updating time per simulation step for processor P_k is $O(J_k \log(N^2/2P))$. The corresponding parallel time is $O(\max_k\{J_k\} \log(N^2/2P))$. Therefore, to minimize the parallel time, it is also necessary to have a uniform distribution of the updating set $\mathbf{Q_{ij}}$ among the processors such that $J_k = O(2N/P)$ for $k = 0, 1, \ldots, P-1$. This would lead to the load balance in both memory and computation and maximizes the parallelism. The optimal parallel time is then $O((N/P)\log(N^2/2P))$ for the local minimization step. Also notice that the updating set $\mathbf{Q_{ij}}$ changes dynamically from one collision to another according to the collision pair, and the collision process is random. This prompts us to use a *diagonal-oriented virtual mapping*, also known as Bruno–Capello mapping, which ensures that every row as well as every column is evenly distributed across all processors. Specifically, as shown in Fig. 3, we order the diagonals in the upper triangular part of *ColTime.Table* starting

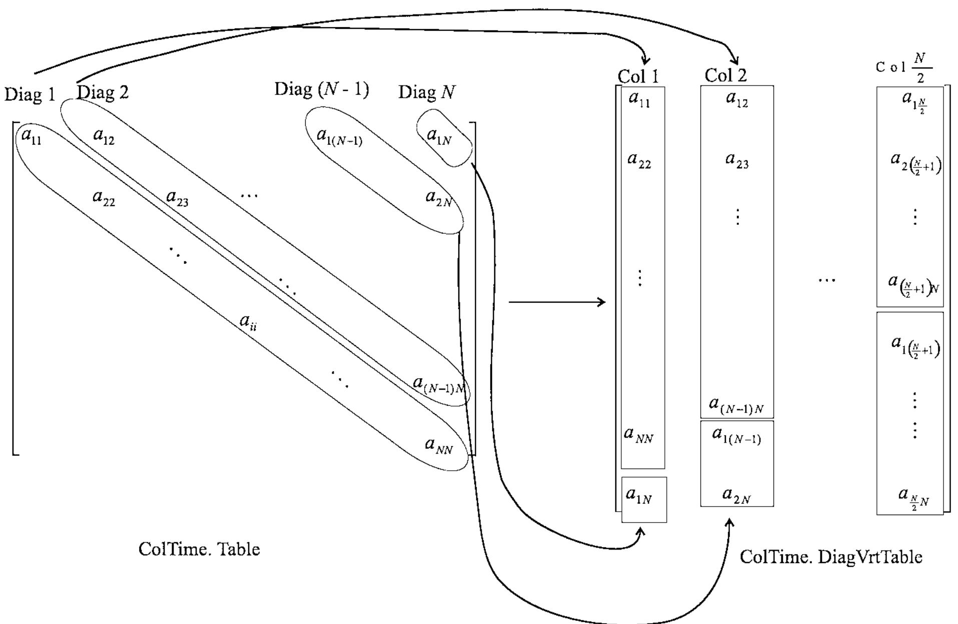

FIG. 3. The diagonal-oriented virtual mapping of *ColTime.Table* (the upper triangular part) to *ColTime.-DiagVrt.Table*.

from the main diagonal toward the upper right corner and merge the diagonals in pair on the order of $(diag1, diagN)$, $(diag2, diag(N-1))$, and so forth, to form another $(N+1) \times (N/2)$ rectangular matrix represented by *ColTime.DiagVrt.Table*. It is clear that this virtual mapping also leads to a uniform multiple-heap partitioning when combined with a uniform partitioning for *ColTime.DiagVrt.Table*, like the previous column (or row)-oriented mapping. However, each diagonal only contains at most four updating nodes, those circled entries of $\mathbf{Q_{ij}}$, as shown in Fig. 1. Therefore, if the columns (diagonal pairs) in the virtual rectangular matrix *ColTime.DiagVrt.Table* are assigned to P processors in wrapping, it is guaranteed that each local heap contains at most $O(N/P)$ updating nodes from the updating set $\mathbf{Q_{ij}}$. Namely, each processor executes local min-heap updates $O(N/P)$ times at most. Thus the parallel time to determine the global minimum is $O((N/P)\log(N^2/P) + \log P)$ per simulation step, where the second term corresponds to the global operation to compute the global minimum from the local minima. It is seen that the parallel time of our algorithm is of the optimal order.

Finally, let us discuss the computation of the updating set $\mathbf{Q_{ij}}$ and the connection between the two levels of partitioning in the physical space and the collision time space, where the major difficulty is in communication. Recall that the entries in the updating set $\mathbf{Q_{ij}}$ are δt_{kl}, where $k, l = i$ or j. δt_{kl} is determined by $(\mathbf{r}_k, \mathbf{v}_k)$ and $(\mathbf{r}_l, \mathbf{v}_l)$. On the other hand, *SolVel* is partitioned based on particles. First we make $(\mathbf{r}_i, \mathbf{v}_i)$ and $(\mathbf{r}_j, \mathbf{v}_j)$ available to all processors, which can be done by the standard global communication procedure *one-to-all broadcast* in $O(\log P)$ time. All processors are then able to independently compute the updated values of certain entries in the updating set $\mathbf{Q_{ij}}$. For example, processor P_q assigned particle p_l can compute the updated values for δt_{li} and δt_{lj} using its local data $(\mathbf{r}_l, \mathbf{v}_l)$ and the data $(\mathbf{r}_i, \mathbf{v}_i)$ and $(\mathbf{r}_j, \mathbf{v}_j)$ received from the corresponding broadcast operations. Notice, however, that *ColTime* is partitioned independently. So δt_{li} and δt_{lj} are generally not assigned to the same processor P_q with the collision time space partitioning, but to other processors, say P_r and P_s, which implies that communication is needed for processor P_q to send δt_{li} to processor P_r and δt_{lj} to processor P_s. Also, each source processor P_q is assigned many particles and all processors should participate in the *send-receive* communication. So care has to be taken for proper coupling to occur between the two levels of partitioning in different spaces and for efficient communication of the updating data set $\mathbf{Q_{ij}}$.

For illustration, we only consider the subset $\mathbf{Q_i}$ of $\mathbf{Q_{ij}}$, consisting of the entries in column i (i.e., $\delta t_{li}, l = 1, 2, \ldots, i$) and row i (i.e., $\delta t_{il}, l = i+1, i+2, \ldots, N$). It is similar for the other subset, $\mathbf{Q_j}$, corresponding to column j and row j, and we have $\mathbf{Q_{ij}} = \mathbf{Q_i} \cup \mathbf{Q_i}$. Because of the symmetry $\delta t_{il} = \delta t_{li}$, the data set $\mathbf{Q_i}$ exactly corresponds to the whole row i of the matrix *ColTime.Table*. Now assume that the particles are assigned to processors by wrapping; namely particle p_l is assigned to processor $P_{mod(l-1,P)}, l = 1, 2, \ldots, N$. Therefore, δt_{il} is computed by $P_{mod(l-1,P)}, l = 1, 2, \ldots, N$ after the broadcast for $(\mathbf{r}_i, \mathbf{v}_i)$. On the other hand, for the partitioning of the collision time space, assume that the columns of the virtual rectangular matrix *ColTime.DiagVrt.Table* are also assigned to processors by wrapping. Then according to the original data structure *ColTime.Table*, we observe for the data set $\mathbf{Q_i}$ that starting with the diagonal entry δt_{ii} assigned to processor P_0, the left-side entries are assigned to processors by wrapping leftward (i.e., $\delta t_{i,(i-l)} \mapsto P_{mod(l,P)}, l = 0, 1, \ldots, i-1$) and the right-side entries are assigned to processors by wrapping rightward (i.e., $\delta t_{i,(i+l)} \mapsto P_{mod(l,P)}, l = 0, 1, \ldots, N-i$), respectively. To illustrate the communication pattern, we show in Table I the correspondence between the entries in the data set $\mathbf{Q_i}$ and the processor locations before and after the communication for the case of $N = 11$, $P = 4$, and $i = 8$.

TABLE I
The Correspondence between the Entries in the Data Set $\mathbf{Q_i}$ and the Processor Locations before and after the Communication for the Case of N = 11, P = 4, and i = 8

Source	P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3	P_0	P_1	P_2
Row 8	δt_{81}	δt_{82}	δt_{83}	δt_{84}	δt_{85}	δt_{86}	δt_{87}	δt_{88}	δt_{89}	δt_{810}	δt_{811}
Destination	P_3	P_2	P_1	P_0	P_3	P_2	P_1	P_0	P_1	P_2	P_3

In addition, we assume that the P processors are configured as a clockwise ring that can be embedded into almost all the existing parallel internet-work architectures. Denote the diagonal index $s = mod(i-1, P)$. So the diagonal entry $\delta t_{i,i}$ is computed by processor P_s according to the partitioning in the physical space but is assigned to processor P_0 according to the partitioning in the collision time space. As seen in Table I, the communication for the data set $\mathbf{Q_i}$ can be organized into two types. First, for the entries on the right side of the diagonal one inclusive, $\delta t_{i,i+l}$ is computed by processor $P_{mod(s+l,P)}$ and needs to be sent to processor $P_{mod(l,P)}$, for $l = 0, 1, \ldots, N-i$. In other words, each destination processor P_d needs to receive from its partner source processor $P_{mod(d+s,P)}$, if $s \neq 0$, a pack of messages

$$M_d = \left\{\delta t_{i,i+(d+kP)}, k = 0, 1, \ldots, K_d\right\}, \tag{3.1}$$

where K_d is the largest integer such that $i + (d + K_d P) \leq N$, for $d = 0, 1, \ldots, P-1$. This can be implemented as a shift operation along the ring, with each processor independently sending a piece of message to a partner processor at the same distance. The shift is clockwise if $s > P/2$ or counterclockwise otherwise in order to take a shorter amount of time. The shift is unnecessary if $s = 0$. Since $N \gg P$ in practice, with the wormhole routing technology as used in most of today's interconnection networks, the parallel time for this communication is $O(t_s + t_w(N-i)/P)$, where t_s is the startup time and t_w is the per-word transfer time. Second, for the entries on the left side of the diagonal one exclusive, $\delta t_{i,i-l}$ is computed by processor $P_{mod(s-l,P)}$ and needs to be sent to processor $P_{mod(l,P)}$, for $l = 1, 2, \ldots, i-1$. Therefore, each destination processor P_d needs to receive from its partner source processor $P_{mod(d-s,P)}$, if necessary, a pack of messages

$$M_d = \left\{\delta t_{i,i-(d+kP)}, k = 0, 1, \ldots, K_d\right\}, \tag{3.2}$$

where K_d is the largest integer such that $i - (d + K_d P) \geq 1$, for $d = 0, 1, \ldots, P-1$. Notice that in this communication, the source and destination processors appear in pairs, as seen in Table I. That is, if processor P_s sends a piece of message to processor P_d, then processor P_d should also send another piece of message to processor P_s. Thus this part of the communication can be implemented as a parallel pairwise exchange procedure. The pairing of the $P/2$ pairs is determined by the index s. Similarly, the parallel communication time is $O(t_s + t_w i/P)$. Overall, in terms of the number of messages passed, the above *exchange-shift* algorithm for communicating the data set $\mathbf{Q_{ij}}$ is of $O(1)$ time.

4. NUMERICAL RESULTS

In this section, we discuss the implementation details and report on the numerical results.

The parallel method presented in this paper has been implemented and tested [20] on Intel's Paragon, which is a distributed memory and message-passing parallel computer. For the reader's convenience, the full code is also made available for downloading on the web site *http://www.math.ust.hk/˜mamu/MultipleHeap*. The code can be easily ported to other parallel computers and clusters by implementing the communication part in Algorithm 4.2 using a machine-independent virtual platform such as PVM or MPI, as discussed later. We provide technical details for the key elements of the implementation.

There are two levels of partitioning in the method, as discussed in Section 3. One is the physical space partitioning that uniformly maps N particles to P processors. We use the wrapping-around mapping in the implementation, where the relation between the global index of a particle i and the corresponding processor ID $pid(i)$ is given by

$$pid(i) = mod(i, P). \tag{4.1}$$

All the physical variables and their computations directly associated with particles, such as position, velocity, and macroscopic properties, are mapped to processors according to this partitioning.

The other is the partitioning applied to the collision time space. The diagonal-oriented virtual mapping, as illustrated in Fig. 3, from the collision time for a particle pair (i, j) with $i \leq j$ to the processor ID $pid(i, j)$ is implemented by

$$pid(i, j) = \begin{cases} mod(j - i, P), & \text{if } j - i < N/2; \\ mod(N - (j - i) - 1, P), & \text{otherwise.} \end{cases} \tag{4.2}$$

This uniformly partitions the collision time data into P subsets, with each being implemented as a local heap in each processor, and guarantees the load balance for updating the collision time space after each collision.

The local min heap in the node program for each processor can in principle be implemented by any data structure suitable for a binary tree, where we implement a tree node as an object with pointers to its parent and two children, a key value for the collision time, and two indices (RowId ColId) for the corresponding particle pair. *insertion* and *deletion* are two standard textbook algorithms, and their program details can be found in [13]. *update key* is less trivial, we provide its pseudocode for reader's convenience .

ALGORITHM 4.1. update a node in a min heap.

procedure *UpdateKey*(*CurrentNode* : TreePointer, *CollisionTime* : **real**);

{*CurrentNode* is a pointer to a node in a min heap. On input, it points to the node to be updated. *CollisionTime* is the corresponding new sorting key value}

begin

{Update the key value}

CurrentNode ↑ *.key* = *CollisionTime*;

{Record the input node}

NodeIn = *CurrentNode*;

{First, check if the key value of the updating node is less than that of its parent node, if any. If so, pop it up like in *insertion*}

```
    ParentNode = CurrentNode ↑ .Parent;
    while (ParentNode ≠ nil and CurrentNode ↑ .key < ParentNode ↑ .key) do
    begin
        {exchange CurrentNode with its parent by calling a procedure
        Swap. On return, CurrentNode points to its original parent before swapping.}
        Swap(CurrentNode, ParentNode);
        {Continue to pop the updating node up}
        ParentNode = CurrentNode ↑ .Parent;
    end;
    {If the updating node has been popped up, the heap update is done}
    if (NodeIn ≠ CurrentNode) exit;
    {Otherwise, make the updating node sink down like in deletion}
    LeftChildNode = CurrentNode ↑ .LeftChild;
    RightChildNode = CurrentNode ↑ .RightChild;
while (LeftChildNode ≠ nil or RightChildNode ≠ nil) do
begin
    {Select the swapping node from the two child nodes}
    if (LeftChildNode = nil) then
       SwappingNode = RightChildNode;
    elseif (RightChildNode = nil) then
         SwappingNode = LeftChildNode;
      else
        {There are two children. Select the swapping node with smaller key value}
        if (LeftChildNode ↑ .key < RightChildNode ↑ .key) then
          SwappingNode = LeftChildNode;
        else
          SwappingNode = RightChildNode;
        endif
      endif
    endif
      {Swap the updating node with the swapping node}
      Swap(CurrentNode, SwappingNode);
      {Continue sinking}
      LeftChildNode = CurrentNode ↑ .LeftChild;
      RightChildNode = CurrentNode ↑ .RightChild;
    end;
end;
```

Using a pointer *ColTimeHeap* to the root node of the local min-heap, the local colliding particle pair is identified by

$$i = ColTimeHeap \uparrow .RowId,$$
$$j = ColTimeHeap \uparrow .ColId.$$

For each entry $\delta t_{kl} \in \mathbf{Q_{ij}}$ the new collision time value is computed and

$$UpdateKey(HeapNode(k, l), \delta t_{kl})$$

is invoked by the corresponding processor, where $HeapNode(k, l)$ is the pointer to the min-heap node corresponding to the particle pair (p_k, p_l).

Based on the min heap, the multiple-heaps algorithm for parallel simulation of collision systems can be implemented by the following pseudocode.

ALGORITHM 4.2. A multiple heaps algorithm for parallel simulation of collision systems is as follows.

```
procedure MultipleHeaps;
{Multiple heaps algorithm for parallel simulation of collision systems}
begin
    {Obtain the multiprocessor configuration}
    P = NumNodes();
    k = MyNode();
    {Initialization}
    Initialize the local data structures PosVel_k, ColTime_k, Heap_k;
    for each collision step ColStep do
    {Execute collision}
    begin
        {Identify the local colliding pair from Heap_k}
        t_loc = ColTimeHeap ↑ .Key;
        i_loc = ColTimeHeap ↑ .RowId;
        j_loc = ColTimeHeap ↑ .ColId;
        {Identify the global colliding pair (p_i, p_j) and the corresponding
        collision time t_ij by the global – min procedure}
        GlobalMin(t_loc, i_loc, j_loc, t_ij, i, j);
        {Particle movement up to the position and velocity change caused
        by the collision between p_i and p_j.}
        Update PosVel_k;
        {Communication for (s_i, v_i) and (s_j, v_j) by broadcasting}
        Broadcasting (s_i, v_i);
        Broadcasting (s_j, v_j);
        {Compute the updating set Q_ij}
        Compute δt_il and δt_jl if I hold particle p_l for l = 1, 2, ..., N;
        {Communication for the updating set Q_ij by the exchange-shift procedure}
        Exchange(the left side of the diagonal entry of Q_i);
        Shift(the right side of the diagonal entry of Q_i);
        Exchange(the left side of the diagonal entry of Q_j);
        Shift(the right side of the diagonal entry of Q_j);
        {Local heap update}
        for each entry δt_qr ∈ Q_ij assigned to me do
            UpdateKey(HeapNode(q, r), δt_qr);
    end;
    {Compute macroscopic properties every NumStepMacro steps}
    begin
        if Mod(ColStep, NumStepMacro) = 0 then
        Compute macroscopic properties;
    end;
end;
```

We note that in this parallel implementation, only four communication procedures, **GlobalMin**, **Broadcast**, **Exchange**, and **Shift**, plus two primatives, **NumNodes** and **MyNode**, involve parallel programming. The shift and exchange procedures are easily implemented by using (3.1) and (3.2), respectively. Others can be found in a standard parallel library for any platform supporting distributed memory and message-passing parallel programming. Furthermore, besides the min-heap part for searching and updating the collision time in order to improve the computational efficiency, all the other computational components, such as initialization, position and velocity, and macroscopic properties, remain the same as in any standard hard-sphere molecular dynamics simulation system, and their parallel implementation is straightforward. This implies that the speedup is achieved only by exploiting parallelism without affecting the physical properties and versatility of the molecular dynamics simulation approach, as already demonstrated in the literature for a variety of applications [12, 20, 21].

We now report on the practical performance of our parallel method on Intel's Paragon. Our Paragon system has 140 compute nodes, with 128 nodes allocated for the computational purpose. Each compute node has an Intel i860 processor and 32 MB of memory. Under the scope of this paper, we are concerned with reducing of computational time by using parallel computing. As discussed earlier, the core and dominating computational part is the loop over all the collision steps, while other parts are much less time-consuming, not executed at every collision step, and physically unchanged. Therefore, we focus on the practical time reduction of the core loop of collision steps by using the multiple-heaps parallel algorithm. Consider the simulation of N particles in a closed domain with the reflecting boundary condition. Initially, the domain is separated into two regions by a barrier, where the temperatures and densities are different for the two regions, but each region is in its equilibrium state locally. For each region, the initial positions are assigned based on the FCC (faced-centered cubic) lattice, and the initial velocities are assigned based on the Maxwell distribution. At time 0, the barrier is removed, and the particles start to collide, as in a gas mixture application. We measure the parallel time (wall-clock time) on our Paragon after 1000 collisions by using the full configuration of 128 processors, with the number of particles N varying from several hundred to over 3000. As seen in Fig. 4, the parallel time is only limited to a few minutes for all the cases. For comparison, we also measure the time of the sequential code running on UltraSparc-I for the same simulation. Notice that ideally we should measure the time reduction by running the sequential code on a single processor of the same parallel machine. However, because Paragon's i860 processor has relatively slow speed and small memory due to the rapid development of hardware technology in the past a few years, it is not suitable for running a large simulation on such a single processor for the sequential purpose; it is suitable only for the parallel purpose. The UltraSparc-I machine (167 MHZ) we use has 196 MB and a theoretical peak performance of 333 MFLOPS, while the i860 processor has only 32 MB and a theoretical peak performance of 75 MFLOPS, as reported in the Netlib Benchmark. As seen in Fig. 4, the sequential time on UltraSparc-I exceeds 1500 s for a 3000-particle simulation. So the sequential simulation on Paragon would easily take several hours per 1000 steps because i860 is about four times slower than UltraSparc-I, while the parallel time is just under 5 min, which clearly demonstrates the substantial time reduction when using our parallel algorithm, and the gain rapidly increases as the number of particles increases. We refer the reader to [20] for more details about several physical simulations of up to 122,000 collisions with up to 4900 particles while running our parallel code on Paragon.

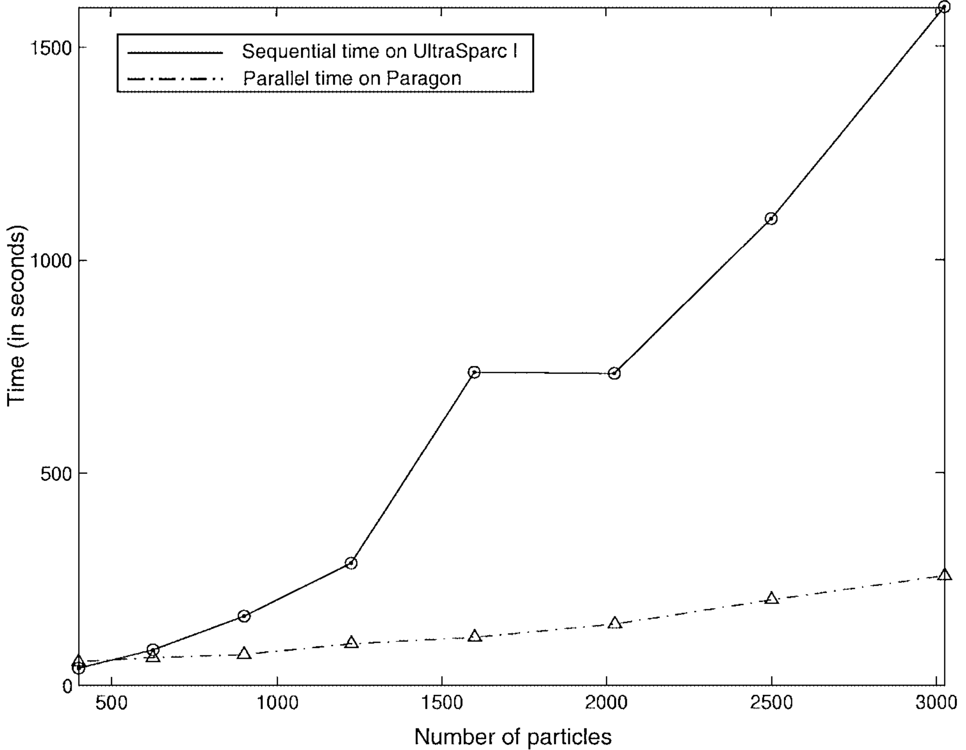

FIG. 4. The time performance of the multiple-heaps parallel algorithm on Paragon using 128 processors versus that of the sequential min-heap algorithm on UltraSparc-I for 1000 collisions with different numbers of particles. Notice that UltraSparc-I is about four times faster than a single Paragon processor.

5. CONCLUSIONS

In this paper, we analyze the computational issues in parallel simulation of collision systems and design proper data structures such that the central computational task can be formulated as a consecutive minimum search problem with multiple updates per collision step. It leads to a sequential min-heap algorithm for efficiently identifying colliding pairs and updating collision times. A parallel algorithm with multiple heaps and diagonal-oriented partitioning for the collision time space is also presented. The algorithm is highly parallel, load balanced, and of the optimal order of parallel time. The partitioning in the physical space is bridged with the partitioning in the collision time space by an efficient exchange-shift communication scheme for the updating set in the collision time space.

The parallel algorithm is in principle applicable to all collision systems although the technical issues are presented in the context of hard-sphere molecular dynamics simulation applications. It has been used in gas dynamics simulation based on the Boltzmann gas kinetic theory, which provides an alternative approach for studying the gas dynamics properties, especially when difficulties occur in both the macroscopic and microscopic PDE-based approaches. Besides molecular dynamics simulation, the parallel algorithm may also be applied to other computing areas where a priority queue is needed. The work may be extended to systems where the repulsion effect is present, or where collision may involve more than two particles. The latter case may lead to a more accurate model than the Boltzmann theory based on the bimolecular collision assumption.

ACKNOWLEDGMENTS

The author thanks the referees for the helpful and constructive comments and suggestions that substantially improved the presentation of the manuscript. The author also thanks C. H. Siu and S. H. Wong for their participation and help in the code development and data collection in this project, in particular the timing data used in Fig. 4.

REFERENCES

1. D. J. Adams, Chemical potential of hard-sphere fluids by Monte Carlo methods, *Mol. Phys.* **28**, 1241 (1974).
2. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*, (Addison–Wesley, Reading, MA, (1983).
3. B. J. Alder and T. E. Wainwright, Phase transition for a hard sphere system, *J. Chem. Phys.* **27**, 1208 (1957).
4. B. J. Alder and T. E. Wainwright, Studies in molecular dynamics. I. General method, *J. Chem. Phys.* **31**, 459 (1959).
5. B. J. Alder and T. E. Wainwright, Studies in molecular dynamics. II. Behavior of a small number of elastic spheres, *J. Chem. Phys.* **33**, 1439 (1960).
6. J. Biswas and J. C. Browne, Simultaneous update of priority structures, in *Proceedings of the International Conference on Parallel Processing*, p. 124 (Pennsylvania State University Press, Pennsylvania, PA, 1987).
7. J. M. Bowman, *Molecular Collision Dynamics*, Topics in Current Physics 33 (Springer-Verlag, Berlin/New York, 1983).
8. D. Frenkel and B. Smit, *Understanding Molecular Simulation—From Algorithms to Applications* (Academic Press, San Diego, 1996).
9. C. C. Ellis, Concurrent search and insertion in 2-3 trees *Acta Inf.* **14**, 63 (1980).
10. C. C. Ellis, Concurrent search and insertion in avl trees, *IEEE Trans. Comput. C* **29**(9), 811 (1980).
11. L. Greengard and V. Rokhlin, A fast algorithm for particle simulations, *J. Comput. Phys.* **73**, 325 (1987).
12. J. M. Haile, *Molecular Dynamics Simulation: Elementary Methods* (Wiley, New York, 1992).
13. E. Horowitz and S. Sahni, *Fundamentals of Data Structures in PASCAL*, 3rd ed. (Freeman, New York, 1990).
14. D. B. Johnson, Efficient algorithms for shortest paths in sparse networks, *J. ACM* **24**, 1 (1977).
15. M. N. Kogan, *Rarefied Gas Dynamics* (Plenum, New York, 1969).
16. V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing Design and Analysis of Algorithms* (Benjamin–Cummings, Redwood City, CA, 1994).
17. V. N. Rao and V. Kumar, Concurrent access of priority queues, *IEEE Trans. Comput. C* **37**(12), 1657 (1988).
18. D. Rapaport, Molecular dynamics simulation of polymer chains with excluded volume, *J. Phys. A.* **11**, L213 (1978).
19. J. A. Sethian, *Level Set Methods* (Cambridge Univ. Press, Cambridge, UK, 1966).
20. C. H. SIU and S. H. Wong, Molecular Dynamics of Hard Sphere Simulation, FYP-1999 (Dept of Math, Hong Kong University of Sci. and Tech, 1999).
21. S. Smith, C. Hall, and B. Freeman, Molecular dynamics for polymeric fluids using discontinuous potentials, *J. Comp. Phys.* **134**, 16 (1997).
22. W. W. Wood and J. D. Jacobson, Preliminary results from a recalculation of the Monte Carlo equation of state of hard-spheres, *J. Chem. Phys.* **27**, 1207 (1957).