Parallel Algorithms for Real-time Colliding Face Detection

Yoshifumi KITAMURA†, Andrew SMITH†, Haruo TAKEMURA‡ and Fumio KISHINO†

† ATR Communication Systems Research Laboratories
2-2 Hikaridai, Seika-cho, Soraku-gun, Kyoto, 619-02, Japan
<kitamura, kishino>@atr-sw.atr.co.jp

‡ Nara Institute of Science and Technology
8916-5 Takayama-cho, Ikoma-shi, Nara 630-01, Japan
takemura@is.aist-nara.ac.jp

Abstract

We propose parallel algorithms for detecting collisions among 3-D objects in real-time. First, a basic algorithm of serial version is described. It can detect potential collisions among multiple objects with arbitrary motion (translation and rotation) in three-dimensional (3-D) space. The algorithm can be used without modification for both convex and concave objects represented as polyhedra. This algorithm is efficient, simple to implement, and does not require any memory intensive auxiliary data structure to be precomputed and updated. Then, two parallel algorithms are proposed for MIMD multi-processors having a shared-memory; one uses a static and the other uses a dynamic method for proper load balancing. Experimental results demonstrate the performance of the proposed collision detection methods.

1 Introduction

Collision detection is one of the central problems in many application fields. For example, in order to enhance the reality of a virtual environment, it is necessary to avoid inter-penetration among objects, and to accurately simulate several kinds of physical phenomena. Other applications such as motion monitoring of mobile robots have the same requirement. For this purpose, we have to detect the "collisions" among polyhedral objects in the environment. In addition, we have to determine more precisely the "colliding pair of faces" and their normal vectors to accurately simulate the physical phenomena. Unfortunately, however, this is clearly an expensive proposition for a typical environment, which includes multiple objects with complicated shapes. It is also difficult to detect collisions accurately in real-time [1, 2].

The basic method of collision detection among polyhedral objects is to perform static intersection tests at discrete time instants by testing all combinations of faces and edges for the presence of an edge of one object piercing the face of another object [3]. However, this requires much computation as the number of objects with complicated shapes increases. Moreover, if the time interval between tests is not short enough, collisions within the time intervals may be missed. Especially for such applications as object manipulation in a virtual environment [4] and motion simulation of a mobile robot, it is essential not to miss the collisions among objects and it is further desirable to detect potential collisions before they occurs. For this purpose, in addition to the static interference test at discrete time instants, we have to interpolate object positions during the time interval and predict future object motion. This requires much computation; therefore, it has been impossible to accurately detect potential collisions in a typical environment in real-time. Although there is much literature devoted to solving these problems, most algorithms have limitations on object shape or environment. No method has been proposed to accurately detect collisions in a typical environment in real-time.

On the other hand, parallel computers have recently become more inexpensive recently, and are attracting wide interest. Even some types of conventional workstations have dozens of high speed processors and are provided with useful parallel libraries [5]. However, no report exists on a colliding face detection method that uses multiple processors.

This paper proposes parallel algorithms for detecting collisions among 3-D objects in real-time. First, the basic algorithm that efficiently detects potential collisions is described. Then, two parallel algorithms are proposed for MIMD multi-processors having a shared-memory; one uses a static and the other uses a dynamic method for proper load balancing. Finally, experimental results that demonstrate the performance of the proposed methods are given with discussion on the method's features.

2 Detection of Colliding Faces

Polyhedral shape representation is one of the most common shape representations. Basic methods for detecting collisions between polyhedral objects are described along with some efficient approaches.

IEEE International Workshop on
Robot and Human Communication
0-7803-2904-x/95 $4.00

2.1 Basic Methods

The basic method of interference detection among polyhedral objects is to perform static intersection tests at discrete time instants by testing all combinations of faces and edges for the presence of an edge of one object piercing the face of another object [3]. However, some applications require flawless detection of collisions before they occur. For example, in motion simulation of a mobile robot in a 2-D or 3-D environment, the robot must judge whether it will collide with obstacles if it maintains the current motion until the next time instant once again it recognizes its position. Here, the position and orientation of the robot at discrete time instants are assumed to be known. Other applications, such as object manipulation in a virtual environment [4], have the same requirement. For this purpose, in addition to the static interference test at discrete time instants, we have to interpolate the object position between the time instants and predict future object motion to completely detect potential collisions. As an example, there are methods in which the colliding point and time can be derived analytically by using mathematical expressions to solve the equations that represent trajectories of vertices and edges. In order to simplify the process of solving these equations, some restrictions are added to the motion of vertices and edges. For example, [3] assumed that one of the objects stands still while the other object moves straight or rotates around an axis, [6] assumed that the motions of objects are translations with constant velocities or rotations with constant angular velocities, and [7] assumed that the motions of objects are represented by cubic functions of time. An extension of these approaches is a sweeping approach,which computes the volume swept out by object motion and tests whether these swept volumes intersect with other swept volumes [8]. However, the generation of sweeping volumes and intersection tests among them require much computation, so it is difficult to use this approach in an environment involving complicated shapes or motions.

2.2 Efficient Collision Detection Approaches

Much research on collision detection for polyhedra aims to drastically reduce the number of edge-face pairs that need to be checked for intersection. A common first step in many collision detection routines is an approximate bounding region (usually an axis-aligned box or a sphere) overlap test to quickly eliminate many objects as not interfering. An extension of this idea is to use a hierarchy of bounding regions to quickly localize collision regions [2] and methods that use octrees or voxel sets [9–15]. However, these methods have limited usefulness because objects are essentially limited to being rigid; this is because when an object deforms its auxiliary data structures must be recomputed, and this is usually an expensive operation. In addition to being expensive to recompute, storing auxiliary data structures for each object can occupy considerable memory. This limits the number of objects for which such algorithms can be effectively used.

Another method for collision detection involves keeping track of the distance between each pair of objects in the world; if the distance between a pair goes below some small threshold then the pair has collided [16–18]. However, these algorithms have limitations because they require objects to be rigid and convex.

A fast algorithm using a recursive spatial subdivision technique for typical (i.e. the environment can contain both convex and concave objects), deformable polyhedral objects undergoing arbitrary motion is proposed [19]. However, because collision is tested at discrete time instants, if an object moves faster relative to the time interval, collisions within the time interval may be missed.

2.3 Real-time Detection of Potential Colliding Faces

Suppose position and orientation of each object at discrete time instants are known and collisions are tested at the same discrete time instants. We suggest a simple method to detect potential collisions among objects with continuoes motion without missing actual collisions: if a volume swept by each face during a unit time interval interferes with another volume, then these faces collide. Because this method requires much computation in generating sweeping volumes and testing interference among them, it has been difficult to detect colliding faces of complicated objects in real-time. However, as these processes are repetitions of relatively simple calculations, speedup can be achieved by using efficient algorithms. In addition, further speedup of these processes can be easily achieved by using a data parallel algorithm.

Generally, such systems as a mobile robot recognizes their own position and orientation at some discrete time intervals based on the processing time of sensor information. Similarly, in other applications of direct object manipulation in a virtual environment [4], object position and orientation are usually measured at discrete time instants. For example, it is possible to measure them several times per second if an image processing technique is used, and tens of measurements are available by a magnetic sensor which is widely used in the latter application as the position and orientation sensor. In this paper, we assume that object position and orientation are obtained almost ten times every second and set up a standard of real-time performance by detecting collisions safely within this time interval (100 milliseconds). This standard agrees with the cycle time of a perceptual processor for human perceptual ability [20] and is recognized as an average time to provide a user with a natural impression of motion if the user interface incorporates visual feedback with computer graphics.

3 Algorithm for Colliding Face Detection

This section describes a basic serial algorithm for colliding face detection. Parallelization is described in the next section.

3.1 Assumptions

Every object in the world is modeled as a polyhedron (boundary representation). Objects can be concave or convex. Objects can perform motion that is not predetermined; object motion can be both translation and rotation. Objects can be deformed during motion. The position and orientation of each object are given every Δt at discrete time instants $(\cdots, t_{i-1}, t_i, t_{i+1}, \cdots)$. If an object deforms, each vertex point of the object is supposed to be known. Here we assume that the speeds of moving objects are sufficiently slow relative to the sampling intervals, i.e. $|\vec{p}_{object_i}(t_i) - \vec{p}_{object_i}(t_{i-1})| \ll L/2^l$ where $\vec{p}_{object_i}(t)$ is the position vector of objects numbering i at time t, L is the side length of the entire workspace, and l is the depth of the face octree.

3.2 Approximate Interference Detection Using Bounding Boxes (step1)

At each time instant, axis-aligned bounding boxes are computed for all objects, and all pairs of objects are compared for overlap of their bounding boxes. For each pair of objects whose bounding boxes overlap, the intersection between the two bounding boxes is determined (called an overlap region) and put into a list of overlap regions for each of the two objects. The overlap regions are utilized in the next step.

3.3 Determination of Faces Intersecting Overlap Regions (step 2)

For every object with a list of overlap regions, all faces of the object are compared for intersection with the overlap regions. Once a face of an object is determined as intersecting with at least one overlap region, it is placed in a face check list for the object. If there are face check lists for two or more objects, these proceed to the next stage.

3.4 Face Octree Spatial Subdivision Stage (step 3)

A face octree is built to a user-specified resolution for the remaining faces starting from the world cube of side length L as its root. To minimize computation, only as much of the face octree as is necessary for collision detection is built; in particular, a parent node is subdivided into its eight children only if it contains faces from two or more objects, and only the faces found to intersect the parent node are tested for intersection with the children nodes. Also, there is no condensation of the face octree (i.e., eight black child nodes are not erased and replaced by their single black parent node). If there are voxels in the face octree, then in each voxel there are faces from two or more objects. For each voxel, all possible pairs of faces, where the faces are from different objects, are determined and put into a face pair checklist. However, a face pair is only included if it was not previously put there by examination of another voxel. The face pair checklist then passes to the next stage. Note that it is not necessary to allocate memory and actually build a face octree; faces can simply be checked for intersection with the standard cubes of an octree and checked recursively for lower-level cubes (thus requiring no memory, beyond the small amount used by the stack during recursion, for storing octrees). Also note that a face octree is built for only a very small portion of the aggregate faces; the previous stage eliminates most faces as not interfering.

3.5 Face Pair Collision Check (step 4)

The faces identified above are checked for collisions. At any time instant t_i, the possibility of a collision between t_i and t_{i+1} is tested by considering the volume expected to be swept by each face during the interval $[t_i, t_{i+1}]$ (Figure 1). This is how collisions between discrete time instants are avoided. To be conservative, collision is assumed if these volumes intersect, even though this type of intersection is a necessary, but not sufficient, condition for the occurrence of collisions.

For each moving face A, the method computes the convex hulls $V_A^{t_i}$ of a set of vertex points of A^{t_i} (i.e. $a_0^{t_i}$, $a_1^{t_i}$, $a_2^{t_i}$,...) and $A^{t_{i+1}}$ (i.e. $a_0^{t_{i+1}}$, $a_1^{t_{i+1}}$, $a_2^{t_{i+1}}$,...) (chapter 3 in [21]) expected to be swept by face A during the interval $[t_i, t_{i+1}]$. For each face B^{t_i} with which intersection of A^{t_i} is to be tested during the interval $[t_i, t_{i+1}]$, the convex hulls $V_B^{t_i}$ of a set of vertex points of B^{t_i} and $B^{t_{i+1}}$ are also computed. Here, faces A and B at time $t = t_i$ are specified by A^{t_i} and B^{t_i}, respectively.

Next, the intersection between $V_A^{t_i}$ and $V_B^{t_i}$ is tested. An intersection is detected by testing whether one of the following positional relationships of all combinations of faces and edges exists: both endpoints of an edge lie on the same side of the plane containing the face (Edge 1), an edge intersects the outside of the face plane (Edge 2), or an edge intersects the inside of the face plane (Edge 3). An intersection is detected in the case of Edge 3.

This identifies all pairs of faces that are expected to collide in the time interval $[t_i, t_{i+1}]$ by testing for collisions between faces in the face pair checklist. This method is not efficient when the number of vertices of each face is large. In this case, a more efficient method (such as the Muller-Preparata method in chapter 7 of [21]) might be useful to test for intersection of convex polyhedra. Figure 1 shows the simplest case (triangles).

4 Parallelization

For the method described in section 3, steps 1 and 2 take such a small percentage of the computation time that it is best to do them serially (i.e. the parallelization overhead will probably be excessive). Thus, this method's parallelization is focused on steps 3 and 4. Especially since step 4 must test every pair of faces in the face pair checklist, this step requires much computation. However, as this process involves repetitions of relatively simple calculations and does not require any memory intensive auxiliary data structures, speedup can be easily achieved by using a data parallel algorithm. In this section, we describe two parallelization

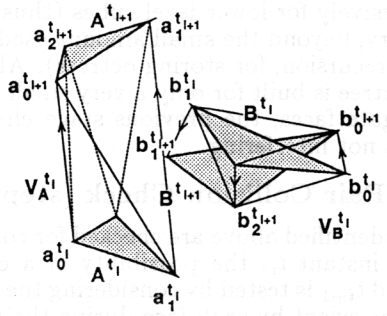

Figure 1: Collision detection between moving faces identified by octree representation as potentially colliding.

strategies that prove useful for enhancing the performance of the collision detection algorithm.

4.1 Parallelization by using a Shared-Memory

The basic idea in realizing the method described in section 3 as a data parallel algorithm is to parallelize step 4. The output of step 3 will be a list of possibly colliding face pairs. Step 4 is then easily parallelized by dividing up the possibly colliding face pairs equally among the available processors; each processor then runs the same code to determine whether the face pairs assigned to it collide. In order to execute this method efficiently, face pairs must be assigned equally to achieve a flat rate of load distribution, and data concerning face pairs or results of collision tests must be communicated between specified processors at high speed. There are basically two communication methods. The first using a shared-memory, which is accessible from all processors equally. In the second method, each processor has local memory and communicates with other processors by passing messages on the communication network. In the former case, because data is read and written through a bus, it is said to take a longer transfer time between processors. However, the computer we use has suitable architecture for parallelized implementation using the shared-memory model of interprocessor communication; in this model, processors communicate by modifying variables that are accessible by all processors. This system is the Silicon Graphics Onyx/Reality Engine with 24 150 MHz R4400 RISC processors. The memory architecture of this machine is shared-memory, where each processor has a 16 Kbyte instruction cache, a 16 Kbyte data cache, and a 1 Mbyte secondary unified instruction/data cache; the main (shared) memory has a 512 Mbyte size, and is 4-way interleaved. Parallelization was effected by using the Sequent compatible parallel programming primitive library [5] on IRIX 5.2. To effect parallelization with library, the *m_fork* function is used to create multiple copies of a function and to start them running on multiple processors; each processor then identifies itself with its ID using the *m_get_myid* function and performs unique computation based on this ID. In addition, the system function *sysmp* [22] was used to schedule the processes to always run on the same processor; this was done to take advantage of cache affinity (i.e. a process quickly fills up its cache with needed data—if the process is rescheduled to a new processor, it has to refill the cache of the new processor, which requires time-consuming main memory accesses). In addition to being faster than rescheduling, this technique also allowed the programs to run more smoothly (i.e. there were not wild variations in computation time at each step of the simulation).

In the following subsections, we assume use of a MIMD (Multiple Instruction stream/Multiple Data stream) type architecture with about ten multiple processors. We describe two parallelization algorithms useful for enhancing the performance of the collision detection algorithm.

4.2 Parallel Algorithm based on Static Load Distribution

The simplest parallelization method follows the Single Program, Multiple Data (SPMD) [23] abstract model of parallel computation in parallelizing only step 4; in SPMD, the processors all run the exact same program, but on different data. The method is as follows. Perform steps 1, 2 and 3 serially (i.e. using just one processor). The output of step 3 will be a list of possibly interfering face pairs (i.e. face pairs for which both faces intersected the same voxel). Step 4 is then easily parallelized by dividing up the possibly interfering face pairs equally among the available processors; each processor then runs the same code to determine whether the face pairs assigned to it interfere. In other words, if there are N processors then processor i will receive face pairs $i, i+N, i+2N, \ldots$. In this case, because the face pairs are assigned to the processors staticly, good performance is expected if computation time on each processor is equal.

4.3 Parallel Algorithm based on Dynamic Load Distribution

In the first parallel algorithm based on static load distribution, proper load balancing cannot be achieved if computation time on each processor is not equal. Actually, there are face pairs which are determined as not colliding in an earlier stage of step 4, but all tests have to be performed when a pair collides. This is one source of bad load balancing. Therefore, the second algorithm is a model of parallel computation in parallelizing steps 3 and 4; this method dynamically assigns face pairs to the processors. It is based on the well-known parallel paradigm known as "producer-consumer" [5]. In this paradigm, one processor is the "producer" that produces the items the "consumers" grab and consume (i.e. do some computation on).

This second method works as follows. Perform steps 1 and 2 serially. Then, have one processor (producer) determine the possibly interfering face pairs (step 3); as soon as this processor finds a possibly interfering face pair, it puts it on a list accessible by all of the

processors. The other processors (consumers) go directly to step 4, grab the possibly colliding face pairs from the list (as they are added to the list by the first processor), and check whether they collide. After completing the list of possibly colliding face pairs, the producer becomes a consumer and helps in the checking. Here, multiple processors can access the shared-memory exclusively by using the *m_lock* system function. This method should be faster than the first static algorithm because it parallelizes both steps 3 and 4 by distributing the load dynamically.

5 Experimental Results

This section is concerned with the performance of the proposed collision detection algorithms and presents experimental results from a serial algorithm using one processor and two parallel algorithms using multiple processors. For performance evaluation, sphere-like objects approximated by differing numbers of triangular patches were used; spheres were selected for testing because of their orientation invariance. The basic experiment had two identical sphere objects start at different (non-penetrating) positions, moving toward each other (with both translation and rotation motion) until they collide. For collision detection between spheres, constraints such as the distance between centers against the sum of diameters, or among other parametric approaches might be useful for eliminating candidate faces. However, since such constraints are not always suitable for concave or complex objects, we did not use them.

5.1 Experiment with Serial Algorithm

Detection of collisions between two identical objects (spheres) represented by several kinds of polyhedral shapes, each having a different number of planar patches, was used for the single-processor serial algorithm described in section 3. Figure 2 shows for four of the experimental objects the computation time required at each processing cycle from $t = 1(cycle)$, when there is no collision, until $t = 72(cycle)$, when faces from two sphere objects are found to be colliding. In the last cycle, the spheres with 960 faces required 126 milliseconds to determine that 21 out of 186 checked pairs were colliding; the spheres with 3968 faces required 434 milliseconds to determine that 121 out of 1160 face pairs were colliding. Accordingly, the percentage of face pairs tested accurately was only 0.007 (%) and 0.02 (%) among all possible pairs for each case, respectively.

5.2 Experiment with Parallel Algorithm

To determine where parallelization should actually be directed, we measured the computation times of the four steps of the collision detection algorithm. The computation time required at the last stage for the spheres with 3968 faces was broken down into the time for each step of the algorithm (table 1). Similar percentages were obtained for the spheres with 960 faces each, and, in general, for the objects in other experiments. Thus, it is clear from these numbers

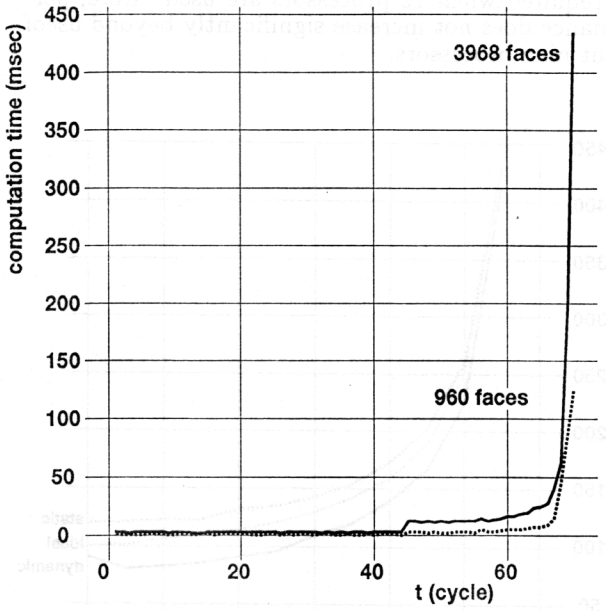

Figure 2: Computation time by single-processor serial algorithm for each processing cycle for two standarized objects

that steps 3 and 4 dominate the total computation and should be the main focus of parallelization.

Table 1: Computation time for each processing step

Processing step	computation time (%)
step 1	0.5
step 2	2.4
step 3	13.2
step 4	83.9

5.2.1 Experiment with Parallel Algorithm based on Static Load Distribution

The implementation of the first parallel method was done by having the serial stage (i.e. steps 1, 2, and 3) write the possibly colliding face pairs to an array accessible by all of the processors. Then, in the parallel stage, the face pairs are distributed evenly among the processors, and each processor checks for collision of its face pairs. This implementation provided fairly good speedups, and the computation time (at the last cycle, when faces were found to be colliding) versus the number of processors can be seen in figures 3 and 4 respectively (graphs identified by "static" in both figures). For the spheres with 3968 faces, 430 milliseconds are required to detect colliding faces by one processor; however, performance increases as the number of processors increase, so that 120 milliseconds

are required when 12 processors are used. Here, performance does not increase significantly beyond useof about nine processors.

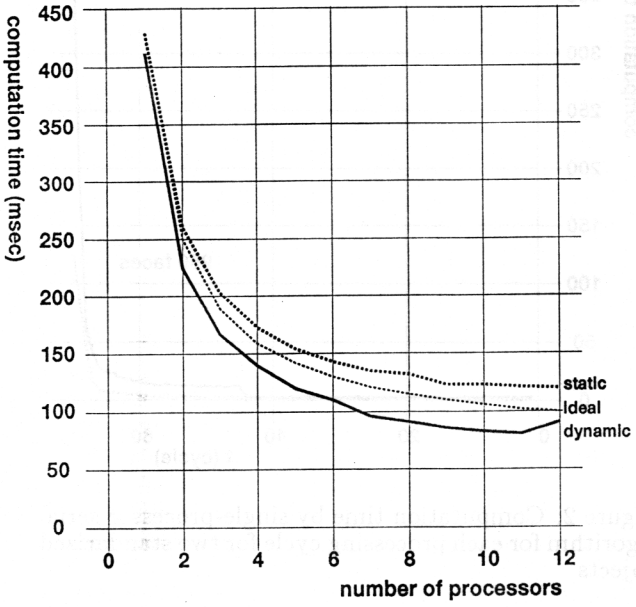

Figure 3: Computation time by parallel algorithm in the last cycle of collision detection for two standarized objects (3968 faces) against the number of processors

5.2.2 Experiment with Parallel Algorithm based on Dynamic Load Distribution

As expected, the implementation of the second parallel method gave better results. In this implementation, after completing steps 1 and 2 serially, one processor finds the possibly interfering face pairs (step 3) and writes them to an array accessible by all of the processors. The other processors go directly to step 4 and wait for this array to fill up. These other processors grab face pairs as they are added by the first processor and check them for interference (thus, the distribution of face pairs to processors is dynamic). The first processor, after creating the list of possibly intersecting face pairs, then goes on to step 4 and helps the other processors finish checking for intersection of the face pairs. This implementation provided very good speedups and the computation time (in the last cycle, when faces were found to be colliding) versus the number of processors can be seen in figures 3 and 4, respectively (graphs identified by "dynamic" in both figures). 80 milliseconds were required to detect colliding faces for the spheres with 3968 faces when using 11 processors. On the other hand, 27 milliseconds were needed for the spheres with 960 faces when using eight processors. However, performance does not increase significantly when more than about nine processors are used (in fact, it decreases).

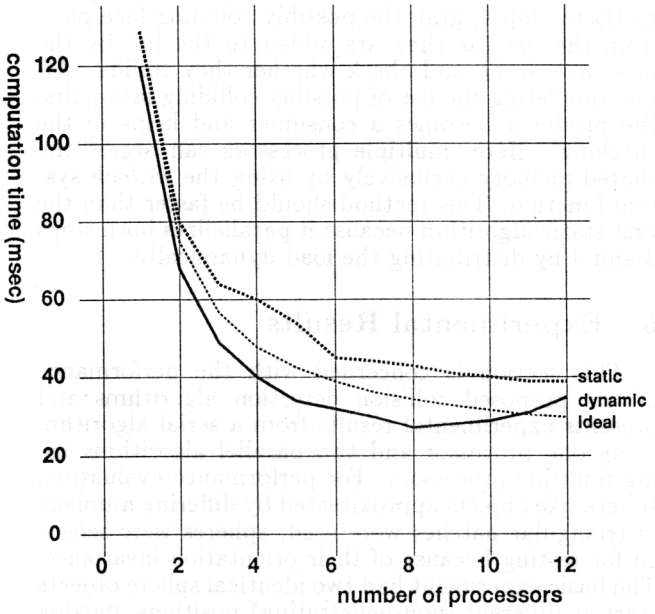

Figure 4: Computation time by parallel algorithm in the last cycle of collision detection for two standarized objects (960 faces) against the number of processors

6 Discussion

6.1 Discussion on Communication

Both static and dynamic load distribution algorithms transfer data from one processor to aother through shared-memory. Here, when more and more processors are used, they all compete for access to the shared bus (which only one processor can access at a time), and this creates considerable overhead. Suppose a face pair checklist has P face pairs after step 3 (described in 3.4) and p pairs among P are colliding. With the static load distribution algorithm, processors access the shared-memory a total of $P+p$ times in the parallel stage because the face pair checklist is already complete. On the other hand, shared-memory is accessed $2P+p$ times with the dynamic load distribution algorithm under the same conditions. Therefore, the exclusion procedure requires more computation in the dynamic load distribution algorithm because the number of times that processors access the shared-memory increases. Actually, in the last stage of collision detection, which requires maximum computation, 121 face pairs out of 1160 were colliding according to the experiment described in the previous section. In this case, processors accessed the shared-memory 2441 and 1281 times for the dynamic and static load distribution algorithms respectively.

Time required for each communication might be a problem. In the proposed parallel algorithm using communication through the shared memory, data is transferred to/from the shared-memory through the bus. Therefore, the time required for the communication is considered the rate-determining step. Commu-

nications in the proposed parallel algorithm consist of data transfer (such as face pairs) and system functions (such as process of execution). Because the computer used in this experiment has high speed RISC processors, data processing was faster than transfer speed. There seems to be a limit to the performance that can be achieved through parallelizm. In our experiments, we noticed that for all of the data sets in figures 3 and 4 this limit was about eight to ten processors. This is not due to the algorithm, but is a general problem with shared-memory parallel architectures.

6.2 Discussion on Effeciency of Parallelization

Both parallel algorithms described in section 4 have the abstract model of parallel computation in parallelizing step 4. In this subsection, expected efficiency of parallelization is discussed for the static load distribution algorithm under ideal conditions without overhead of communication between processors. For the experimental result with the serial algorithm described in 5.1, the percentage required for each step is shown in table 1. Suppose we have N processors and step 4 (subsection 3.5) is parallelized by the static load distribution method described in 4.2. In this case, the computation time required for step 4 becomes $1/N$, while computation times for the other steps (step 1 to 3) do not change because overhead of communication is neglected here. Therefore, the expected total computation time T_N is given by

$$T_N = t_1 + t_2 + t_3 + \frac{t_4}{N}$$

where t_i is the computation time of step i on a single processor. In figures 3 and 4, the expected computation time T_Ns for the spheres with 960 and 3968 faces are shown against the number of processors N (identified by "ideal"). These graphs are considered to show the theoretical performance limit that can be expected if the static load distribution algorithm is used. Actually, overhead of communication is included in the measured computation time, and this overhead is considered to become greater as the number of processors increase. Figures 3 and 4 closely match the results of these theoretical expectations.

7 Summary and Conclusions

We proposed parallel algorithms for detecting collisions among 3-D objects in real-time. First, the basic algorithm that efficiently detects potential collisions in a typical environment was described. Then two parallel algorithms were proposed for MIMD multiprocessors having a shared-memory; one used a static and the other used a dynamic method for proper load balancing. Experimental results showed that 80 milliseconds are required to detect colliding faces among two identical standerized objects having 3968 faces. Through investigation of the performance of the proposed parallel algorithms, we found a limit to performance that can be achieved through parallelism. This limit is due to communication between processors through the shared-memory being a rate-determining step. Future work will involve parallelization by communication using message passing and implementation on other architectures having local distributed memories.

References

[1] Pentland, Alex P. Computational complexity versus simulated environments. *Computer Graphics*, Vol. 24, No. 2, pp. 185–192, 1990.

[2] Hahn, James K. Realistic animation of rigid bodies. *Computer Graphics*, Vol. 22, No. 4, pp. 299–308, 1988.

[3] Boyse, John W. Interference detection among solids and surfaces. *Communications of the ACM*, Vol. 22, No. 1, pp. 3–9, 1979.

[4] Chanezon, A., Takemura, H., Kitamura, Y., and Kishino, F. A study of an operator assistant for virtual space. In *Virtual Reality Annual International Symposium*, pp. 492–498. IEEE, 1993.

[5] Barr E. Bauer, editor. *Practical Parallel Programing*. Academic Press, Inc., 1992.

[6] Canny, John. Collision detection for moving polyhedra. *IEEE Transactions on PAMI*, Vol. 8, No. 2, pp. 200–209, 1986.

[7] Kawabe, S., Okano, A., and Shimada, K. Collision detection among moving objects in simulation. *Robotics Research*, Vol. 4, pp. 489–496, 1988.

[8] Cameron, Stephen. Collision detection by four-dimensional intersection testing. *IEEE Transactions on Robotics and Automation*, Vol. 6, No. 3, pp. 291–302, 1990.

[9] Moore, M. and Wilhelms, J. Collision detection and response for computer animation. *Computer Graphics*, Vol. 22, No. 4, pp. 289–298, 1988.

[10] Turk, Greg. Interactive collision detection for molecular graphics. M.sc. thesis, Department of Computer Science, University of North Carolina at Chapel Hill, 1989.

[11] Zyda, M. J., Pratt, D. R., Osborne, W. D., and Monahan, J. G. NPSNET: Real-time collision detection and response. *The Journal of Visualization and Computer Animation*, Vol. 4, No. 1, pp. 13–24, 1993.

[12] Shaffer, C. A. and Herb, G. M. A real-time robot arm collision avoidance system. *IEEE Transactions on Robotics and Automation*, Vol. 8, No. 2, pp. 149–160, 1992.

[13] Hayward, V. Fast collision detection scheme by recursive decomposition of a manipulator workspace. In *International Conference on Robotics and Automation*, pp. 1044–1049. IEEE, 1986.

[14] Kitamura, Y., Takemura, H., and Kishino, F. Coarse-to-fine collision detection for real-time applications in virtual workspace. In *International Conference on Artificial Reality and Tele-Existence*, pp. 147–157, July 1994.

[15] Garcia-Alonso, A., Serrano, N., and Flaquer, J. Solving the collision detection problem. *Computer Graphics and Applications*, Vol. 14, No. 3, pp. 36–43, May 1994.

[16] Lin, M. C., Manocha, D., and Canny J. F. Fast contact determination in dynamic environments. In *International Conference on Robotics and Automation*, pp. 602–608. IEEE, 1994.

[17] Gilbert, G., Johnson, W., and Keerth, S. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal of Robotics and Automation*, Vol. 4, No. 2, pp. 193–203, 1988.

[18] Quinlan, Sean. Efficient distance computation between non-convex objects. In *International Conference on Robotics and Automation*, pp. 3324–3329. IEEE, 1994.

[19] Smith, A., Kitamura, Y., Takemura, H., and Kishino, F. A simple and efficient method for accurate collision detection among deformable polyhedral objects in arbitrary motion. In *Virtual Reality Annual International Symposium*, pp. 136–145. IEEE, March 1995.

[20] Card, S. K., Moran, T. P., and Newell, A. *The Psychology of Human-Computer Interaction.* Hillsdale, NJ: Lawrence Erlbaum Associates, 1983.

[21] Preparata, F. P. and Shamos, M. I. *Computational geometry, an introduction.* Springer-Verlag, 1988.

[22] Silicon Graphics, Inc., Mountain View, CA USA. *Silicon Graphics Online Documentation*, 1994.

[23] Alan H. Karp. Programming for parallelism. *IEEE Computer*, pp. 43–57, May 1987.