Scalable parallel simulator for vehicular collision detection

Ilan Grinberg and Yair Wiseman*

Computer Science Department,
Bar-Ilan University,
Ramat-Gan 52900, Israel
E-mail: ilan_grin@hotmail.com
E-mail: wiseman@cs.biu.ac.il
*Corresponding author

Abstract: Several simulations for parallel vehicular collision detection have been suggested during the last years. Such a simulator saves the need to physically cause the collision. The algorithms usually greatly depend on the parallel infrastructure and this dependency often causes non-scalability of performance. The dependency also harms the portability of the simulation. This paper suggests a scalable and portable parallel algorithm for a vehicular collision detection simulation that fits both clusters and MPI machines. This paper explains how this simulator was designed and implemented in a large transportation company.

Keywords: collision detection; vehicular simulation; bounding volumes.

Reference to this paper should be made as follows: Grinberg, I. and Wiseman, Y. (2013) 'Scalable parallel simulator for vehicular collision detection', *Int. J. Vehicle Systems Modelling and Testing*, Vol. 8, No. 2, pp.119–144.

Biographical notes: Ilan Grinberg was a Team Leader of 3D simulation development in IDF. He is now with Lucidlogix Technologies Ltd.

Yair Wiseman obtained his PhD from Bar-Ilan University and completed two post-doc – one at the Hebrew University of Jerusalem and one at Georgia Institute of Technology. His research interests include vehicular systems, intelligent transportation systems, process scheduling, hardware-software codesign, memory management, and real-time operating systems.

1 Introduction

There are many vehicular simulation software tools in the civilian and military markets for many purposes (Jimenez et al., 2001), for example, crash detection simulation (Brown et al., 2000), embedded intelligent vehicle system (Feng et al., 2009), vehicle survival performance (Wasmund, 2001), sensor calibration optimisation (Kiokes et al., 2009), survivability and lethality analysis (Yong et al., 2008). All of these tools are based on three-dimensional shapes that made up of elementary polygons. Such simulation systems are designed to illustrate the real world; hence, they require high accuracy. High accuracy is obtained by using ten thousands to millions of polygons (Curless and Levoy, 1996). Handling so many polygons obviously requires enormous computation resources.

The main computation in such simulators focuses in basic functions of computational geometry like:

- lines intersections
- line-polygon intersections
- polygon-polygon intersections
- collision detection on Gantt chart
- projections
- transformations
- axis switches and many more.

These functions are part of any standard graphic engine that is used as a framework for any three-dimensional simulator or any computer game (Cohen et al., 1995). Some of these computations consume many resources and an acute problem can occur if the number of the geometrical elements in a given space is too high.

Methodologies in this area raise several issues like what the optimal way to implement these functions in any computational environment is. Any special hardware like graphics accelerator card, graphic card with dual processor, multi-processor computer or computer cluster can significantly influence the implementation of these functions.

To accommodate the many requirements of the computational geometry functions (e.g., polygon-polygon intersections in a space where each polygon has its own velocity and acceleration) there will be a need for:

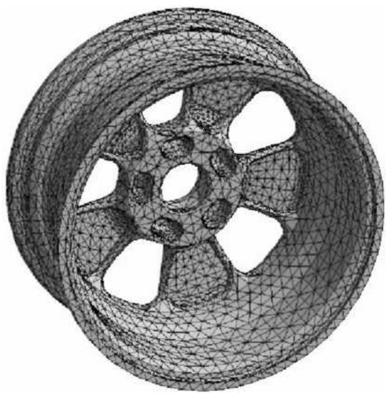
- clever algorithms that can reduce the complexity of the function
- utilisation of as many as possible of processors, i.e., parallel or distributed computation.

In this paper, we present a parallel algorithm for vehicular collision detection simulation. The suggested algorithm is based on the locality principle and the load balance principle. The algorithm is suitable for both complex and simple geometry models with no dependency on the parallel environment and the architecture of the machines.

2 Bounding volumes

One of the most common methods for efficiently implementing computational geometry functions is constructing a smart simulation model for each geometry shape consists of basic polygons. Figure 1 is an example for such a simulation model.

Figure 1 Simulation model of a geometry shape consists of basic polygons



Such a simulation model for the geometry shapes enables an execution of geometry functions on the simulation model in a much more efficient manner than executing the functions on each polygon that the geometry shape consists of. This simulation model is well known as spatial data structures (van der Bergen, 2004a).

Spatial data structures are used in two ways. The first way is reducing the number of intersection checks of static and dynamic objects in a given space. For n objects, there will be $O(n^2)$ potential objects that may be intersected. This number is obviously very high and significantly reducing the number of intersection checks obtained by spatial data structures is quite important.

The second way is reducing the number of intersection checks of pairs of primitive polygons in intersection detection of two complex objects or an intersection of a primitive object and a complex object. In this scheme, the spatial data structures are created in the preprocessing step and remain static, assuming the simulated objects are rigid.

Spatial data structures are employed for space partitioning (Ghanem et al., 2004) and bounding volumes (Trumbore, 1992). Space partitioning is a sub-partitioning of a space to convex regions called cells. Each cell maintains a list of objects that it contains. Using this structure, many pairs of objects can be easily sifted out.

Bounding volume is a split of an object set into consistent subsets and computing for each one of the subsets tight bounding volume so when the intersections of the subsets are checked, these subsets will be straightforwardly sifted out by finding which bounding volumes are not overlapping.

Some research have been conducted on tactics of representing bounding volumes like bounding spheres (James and Pai, 2004), k-DOPs – discrete orientation polytopes (Klosowski et al., 1998), oriented bounding boxes (OBB) (Gottschalk, 2000), axis aligned bounding boxes (AABB) (van den Bergen, 1998) and hierarchical spherical distance fields (Funfzig et al., 2006).

We will introduce the most common one, the AABB tactic. This tactic represents the bounding volume as minimum and maximum values of the geometric model over each one of the axes. This is how a bounding volume is created. Although the AABB representation consumes more storage than the 'bounding sphere' tactic and an intersection of two bounding spheres uses less than two AABB objects, AABB objects can more tightly enclose the original model than bounding spheres can, which will result in less intersection calculations.

Constructing a bounding volume in AABB technique is quite fast (van der Bergen, 2004b). You need to run through each element of the basic elements contained in the bounding volume and project it on each of the axes. What is left is to find the min/max values for each axis.

3 Bounding volume hierarchies

Bounding volume hierarchies are a tree data structure, whose leaves are constructed from the basic elements of the geometry. Each node's leaves are contained inside the bounding box it represents. Sibling nodes can overlap by their representing bounding volumes.

The advantages of using bounding box hierarchies are fast query for intersection testing and linear storage usage with respect to the number of elements constructing the geometry. The major drawback of using this technique is the time it takes to construct the representing tree of the geometry and the updates it inquires when using non-rigid solids. This is why the use of bounding volume hierarchies is common with rigid geometries – its representing tree is generated only once as a pre-processing step.

The test for collision between two geometric models is done recursively for each two nodes taken from each of the geometries trees, starting with the roots.

The total cost of collision detection between two geometric models, which are represented with bounding volume hierarchies, is expressed in the following equation (Kiokes et al., 2009):

$$T_{total} = N_b \cdot C_b + N_p \cdot C_p$$

where

T_{total}	the total time to test intersection between the two models
N_b	number of bounding volumes pairs which are tested for intersection
C_b	the cost of intersection test between pairs of bounding volumes
N_p	number of primitive polygons pairs which are tested for intersection
C_p	the cost of intersection test between pairs of primitive polygons.

The parameters that are affected by the type of bounding volume are N_b, N_p, and C_b. A tight-fitting bounding volume type, such as OBB, results in a low N_b and N_p, but has a relatively high C_b, whereas an AABB will result in more tests being performed, but the value of C_b will be lower.

4 Related works

The potential of sequential algorithms is somehow limited and the parallelising becomes an essential enhancement if the algorithm has to run quickly. Some works have been done to put into practice parallel collision detection. We survey the approaches of these works and explain what has led us to our approach.

In Figueiredo and Fernando (2004), the authors employ AABB to represent the geometry models that are used for collision detection. The algorithm constructs for each model a hierarchy of three levels of bounding volumes. The principle is not to construct a huge tree containing leaves with only one bounding volume of a single polygon. However, complex geometry shapes are likely to have leaves with many primitive polygons. Because of such leaves, the execution time is much larger than a full hierarchy of bounding volumes due to the many checks of intersections of polygon pairs.

The algorithm of Figueiredo and Fernando (2004) is dedicated for SMP machines and cannot work on computer clusters. In SMP machines, the RAM is in the vicinity of all the processors; hence, the locality principle is not kept. If the algorithm is used on a computer cluster and the geometry shapes are complex, a bottleneck will be occur when the geometry shapes are loaded in the beginning of any computation unit.

The algorithm of Lawlor and Kale (2002) suggests a parallel version for space partitioning-based collision detection. The algorithm is scalable and keeps the locality principle by making any voxel a separate process. However, this algorithm does not employ bounding volumes hierarchy. Rather, it employs bounding volumes in a constant size that has been set in advance. This feature will drastically harm the performance of the algorithm if the geometry shapes have non-homogenous density in the polygon prefix.

In addition, some polygons have empty bounding volumes that cause unequal balance on the nodes in the cluster. In order to balance the load the algorithm utilises the parallel infrastructure. Actually, this indicates that no effort is taken to balance the load. If such a case occurs, the parallel infrastructure will be supposed to resolve the unequal balance. This tactic creates an overhead – the solution for unbalance nodes is a migration of processes from one node to another. These migrations may harm the performance of the algorithm.

The cost of constructing the voxels' data structures in Lawlor and Kale (2002) is quite low; hence, rigid objects that require frequent updates of the data structure can benefit from this feature.

The splitting of the space into a large number of voxels enables a node that hosts many voxels to efficiently manage voxels that need the processor and voxels that need the communication line, by the operating system. Obviously, this will not be correct in the beginning of a new simulation when there is no data for any voxel and all the voxels call one node at the same time and create a bottleneck.

In Assarsson and Stenstr (2001), several versions of parallel algorithms for collision detection are presented. The algorithms keep the locality principle and keep the load balance of the nodes. The algorithms are aimed at collision detections of animations (roughly some dozens of frames per seconds) for simple geometry shapes (less than 4,000 primitive shapes). The algorithms are also aimed at just SMP architectures with shared memory.

The load balancing in Assarsson and Stenstr (2001) is static and is done in the beginning before the intersection check step. In addition, the algorithm assumes that the processors are homogenous. Static load balancing reduces the communication overhead; hence is more suitable for real-time simulation. The authors also suggest several techniques to reduce this overhead by using a common queue for several processors.

There are also works that are only aimed at grid environments like Lawlor (2001). This work suggests a simulation that cannot work properly on SMP machines.

In this paper, we suggest a dynamic load balancing. This may have a slightly higher overhead, but it will be able to handle heterogeneous processors and will be able to manage better computer clusters.

5 Scalable collision detection

The new parallel simulation that is suggested in this paper includes several advantages over the known parallel collision detection simulations. The main idea of the suggested simulation is keeping the scalability principle while not abandoning the locality principle and the load balancing of the system (Grinberg and Wiseman, 2007).

We can use one of the known algorithms for bounding volumes hierarchy for checks of an intersection of two models or a collision. Let us define the smallest 'work unit' as one operation (like a collision detection) on a complex geometry model or one operation between two complex geometry models. Indeed, a finer split into smaller unit could have been done like the author of van den Bergen (1998) suggest; however, the cost of execution of one 'work unit' that we suggest will be still very small, even if the geometry model is very complex. Experiments show that a split of geometry models into too many smallest units can produce too much overhead.

Let us define 'processing unit' as one process that gets some parts of the collision detection procedure and returns the results to the master process. Any process in the parallel system can migrate from one processor to another processor in the same SMP or migrate from one node to another node in the same cluster, if this is the policy of the parallel infrastructure.

The algorithm uses the vector space technique to find similarity of scenarios ('work units') and machines ('processing units') in a similar way to queries in document sets in the information retrieval field.

Let us assume that we have two geometry models consisting of basic polygons, n different scenarios where the models are placed in various places and various orientations, and there is a collision in each scenario.

The work is defined as finding k_1, k_2, k_n intersection points of two objects in the n different scenarios where k_i is the number of the intersection points in scenario i. In such a case, the finding of one single collision will be denoted as one 'work unit'.

5.1 The simulation algorithm

Let us denote np as the maximal processors in our machine.

- Create np children that will be the 'processing units'.
- Create a queue of 'processing units' in an arbitrary order.
- Construct the bounding volumes hierarchy of the two geometry models by one of the known models that have been cited above. The data structure can be saved along with the geometry information so there will be no need to reconstruct the hierarchy many times. The bounding volumes hierarchy trees represent the geometry models and any leaf in any tree contains one basic polygon. The indices are put in nodes of no more than level d in each tree from left to right as can be seen in Figure 2.

- Create a list of scenarios containing for each scenario, the scenario index and the bounding volume vector, i.e., for each scenario (‘work unit’), the bounding volumes (the black nodes in Figure 2) that are a part of the current check will be put in the list.
- Let us denote the bounding volume vector as BB. The value of a BBi that is not intersected in the given scenario will be 0. The value of a BBj that is intersected in the given scenario will be the number of the primitive polygons that the bounding volume bounds. An example for such a data structure can be seen in Table 1 – the first table is the bounding volume vectors that are intersected in given scenarios for the geometry model that is depicted in Figure 2. The second table depicts bounding volume vectors for the same scenarios but for the geometry colliding with the first model.
- Create a list of ‘processing units’. Each ‘processing unit’ will contain a vector in the same length as the vectors in the ‘work unit’ list. In the beginning, these vectors are zeroed.
- Allocate *q* scenarios for each ‘processing unit’ in this way:
- For each ‘work unit’ in the processing queue, the *q* free scenarios that are most similar to the ‘work unit’ vector will be selected. The most similar scenarios can be chosen by the well-known VS tactic (Lawlor and Kale, 2002), i.e., a scalar multiplication of the scenario vector and the ‘processing unit’ vector.
- These *q* scenarios will be allocated for these *q* ‘processing units’ and will be removed from the ‘work unit’ list.
- The ‘processing unit’ vector will be update by a switch of 0 to 1 for any bounding volume that was added by the new *q* scenarios.
- Any ‘processing unit’ finds the collision points of the two geometry models for any scenario that was allocated for this specific ‘processing unit’. If the ‘processing unit’ needs more information on the primitive polygons and it does not have the information, the ‘processing unit’ will call the parent process and will get this information from it. The ‘processing unit’ will cache this information for a possible future use.
- When a ‘processing unit’ finishes its jobs, the ‘processing unit’ will call the parent process and will return the results about the intersections that were found in each scenario. The parent process will add the ‘processing unit’ to the free ‘processing unit’ queue.
- This procedure will be repeated until the ‘work unit’ list is empty.

Figure 2 depicts an example of a node indexing getting to level 4 in the tree. Each of the filled squares leaves represent one single polygon. The white nodes represent internal nodes whereas the black nodes are the nodes that will be indexed and will be put in the geometry vector. The number within the node indicate the number of the polygons within the volume that the node bounds. The number beside the node is the index of the node in the vector.

Figure 2 Example of a node indexing

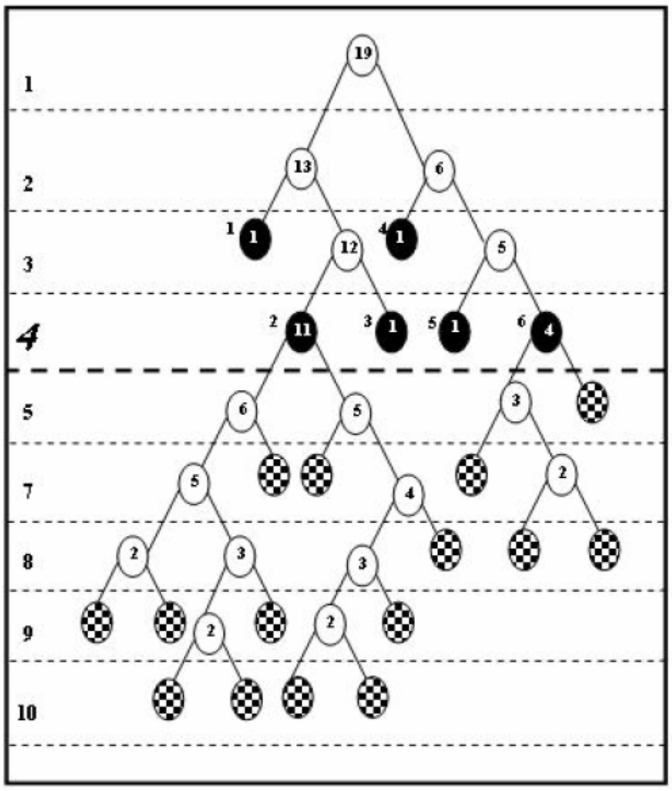


Table 1 Example of a 'work unit' list

Scenario	*BB1*	*BB2*	*BB3*	*BB4*	*BB5*	*BB6*
1	0	0		0	1	4
2	0	11		1	1	0
3	1	11		1	0	4
4	0	0		0	0	0
5	0	11		1	1	0
6	…	…	…	…	…	…

Scenario	*BB1*	*BB2*	*BB3*	*BB4*
1	2	0	14	6
2	2	1	14	6
3	0	0	14	6
4	0	1	0	0
5	0	1	0	6
6	…	…	…	…

5.2 The simulation analysis

It can be seen that the 'work unit' splitting mechanism enables the simulation to keep the locality principle, because the splitting of a job that contains information on parts of the geometry model is similar to the information that the 'processing unit' has worked on it. In this way, the overhead of transferring the geometry models to all the machines in the cluster is prevented. If the geometry models are very complex (this is very common in many simulation tools) and the communication line speed is the common 1 Gb/sec, the execution time can be improved significantly.

The simulation keeps the load balancing in the 'processing units' by managing a dynamic queue and allocation of a small work portion in each iteration for each 'processing unit'. In this way, an optimal computation time will be obtained even if the simulation is executed on a complex parallel infrastructure.

The simulation allocates the needed memory for each work portion in each 'processing unit' and in that way the simulation saves unnecessary memory allocations in other machines in the cluster.

The simulation is generic and it can be independently implemented on any operating system, middleware, hardware or framework. The simulation is fully portable and can be used in any environment.

There is no harm for the simulation performance in any geometry models. The simulation can handle flawlessly geometry of different sizes or shapes.

6 Implementation

Our collision detection algorithm was implemented based on a wide background including several fields, like: computer graphics, computational geometry, object-oriented programming and distributed programming. Our implementation can be divided into two categories: a serial infrastructure and a parallel infrastructure. The serial infrastructure includes the following elements:

- basic mathematical and geometrical operations of matrices and vectors (Foley, 1995)
- primitives intersection algorithms, like point, line, segment, plane and triangle (Eberly, 2001)
- 3D interactive viewer implemented in openGL (Woo et al., 1999)
- loading geometry information from a file (Taubin et al., 1998)
- generating a bounding box tree (Eberly, 2001)
- intersections between two bounding box trees (Eberly, 2001).

The implementation of the serial infrastructure was implemented according to the specified references except of the triangle-splitting algorithm in the segment that generates a bounding box tree. The new splitting algorithm will be described in the following section.

The parallel infrastructure includes the following elements:

- data structures for parallel bounding box trees
- a particular client-server model
- a ‘matchmaker’ object
- a parallel algorithm for an intersection between two bounding box trees.

6.1 Bounding box tree generation

Given a triangular mesh consisting of a vertices collection and a connectivity list, the basic methodology to constructing bounding box tree is recursive.

The generation process can be divided into three major steps:

1. Generating a bounding box for the set of remained triangles.
2. Splitting the set of triangles into two submeshs.
3. Running the recursive process on the two new split submeshs.

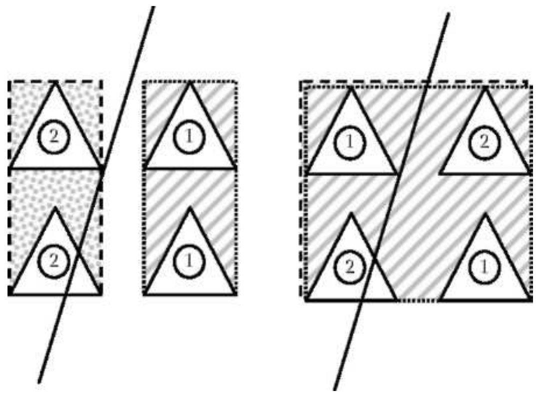
The two submeshs of triangles that have been created in each iteration of the recursion represent the child nodes of the triangles’ initial group node that contains the two submeshes. If a submesh contains at least two triangles, then the process will be rerun on that submesh.

The generation of the bounding box algorithm and the triangles splitting algorithm have a significant effect on the bounding box tree generation algorithm and its performance in a range of operations. We use ‘fitting points with Gaussian distribution’ to generate the bounding box as described in Eberly (2001).

The motivation for splitting the triangles into two submeshs is creating bounding boxes with a minimal volume for the submeshes. This split typically produces a better regional distinction that helps the parallel collision detection algorithm to build distinctive clients and fewer nodes will be needed in each scenario of collision detections.

Figure 3 depicts an example of four triangles split in two different ways. The number inside each triangle represents the submesh it belongs to after the splitting has been done. This figure shows that a hierarchical intersection testing with a specific segment may produce fewer triangles intersection tests in the left chart because of a smaller volume of the bounding boxes.

Figure 3 Example of triangle split



This attribute led us to use the splitting algorithm described below.

6.2 Triangles splitting algorithm

Given a triangular mesh with a corresponding bounding box, the mesh can be split into two submeshes. We define a variable that contains the minimum sum of the relative segments projected onto one of the axes of the box. This variable has been initially set to max-float.

1. For each of the box axes:
   - A positive direction for the axis is chosen.
   - For each triangle, the maximal valued vertex on the projected axis is found.
   - The triangles are sorted by their maximal vertex value.
   - For each triangle from the minimum to the maximum:
     - a The triangle is labelled as a ‘splitting triangle’, indicates that the first submesh will take in the triangles from the minimal to the splitting triangles and the second submesh will take in the rest.
     - b The sum of the relative segments of the two submeshes is calculated. A relative segment is the length of the projection of a submesh onto the box axis divided by the original mesh projection length.
     - c If the relative segments sum is less than the ‘minimum relative segments sum’ variable then:
       1. The ‘minimum relative segments sum’ variable is set to the new relative segments sum.
       2. The current axis is labelled as the splitting axis and the current triangle index is labelled as the splitting index.
       3. The triangles are split according to the latest splitting axis and the latest splitting triangle index. The algorithm motivation is ensuring that there is a minimum overlapping between the two submeshes’ bounding boxes.

Figure 4 and Figure 5 show an example of two different splitting indices. Splitting at triangle index 4 (Figure 4) will cause a bigger overlapping between the two divided segments than splitting in triangle index 2 (Figure 5). This is explains why the algorithm will choose triangle index 2 to be the splitting triangle.

6.3 Data structures for parallel bounding box trees

Our suggested algorithm has an unusual implementation of bounding box trees. This is the reason why we need a different implementation of bounding box tree data structure. We call this data structure, parallel OBB tree. We will refer bounding box trees as OBB trees for the reason that AABB is a case of OBB and our tree data structure construction applies for both AABB and OBB trees. Not all the nodes are known by the clients.

Figure 4 Group of triangles sorted from left to right on the projecting axis with splitting index 4

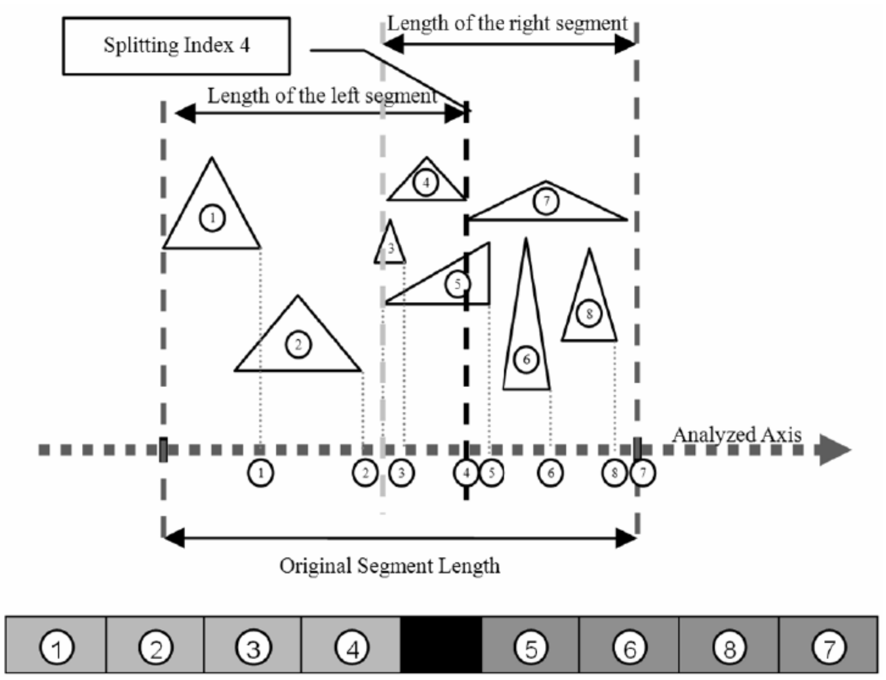
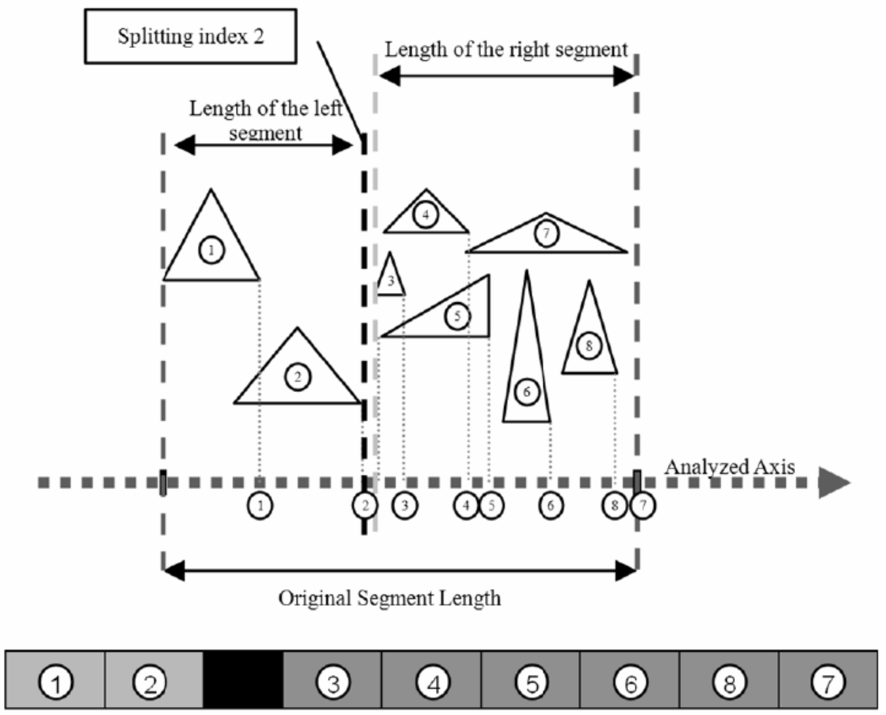


Figure 5 Group of triangles sorted from left to right on the projecting axis with splitting index 2



The parallel OBB tree inherits its properties from a standard OBB tree object. Additional members and methods provide the parallel OBB tree parallel attributes (Figure 6). The client holds a list of pointers to the nodes, which contain the bounding volume vector of the parallel OBB tree. The client initialises the pointers in the list to null at the beginning of the analysis. Each time the client needs to analyse a collision detection scenario, the client will set all its relevant pointers from the list to the new distributed root subtrees, which play a part in the scenario. These subtrees are received from the master process.

Figure 6 Parallel OBB tree data structure

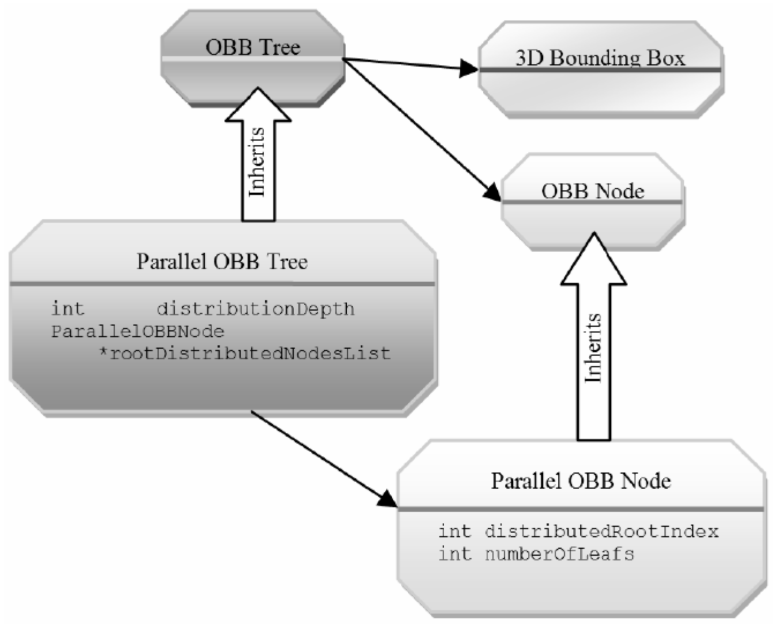
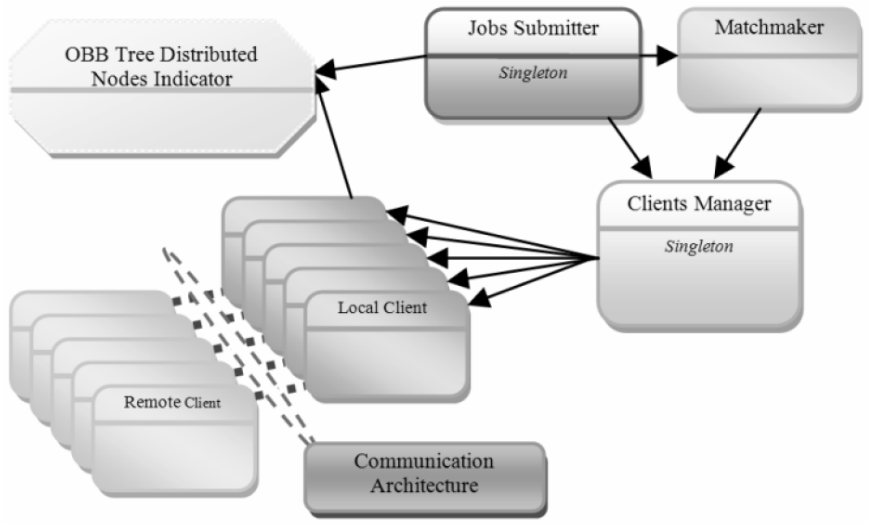


Figure 7 Data structure of client-server model



6.4 Client-server model

To facilitate the algorithm implementation, we developed a dedicated client-server model. The model was designed using dedicated objects, which had a specific role in the work process. Figure 7 shows the client-model data structure and its internal relations. Here in below, we describe the role of each object.

6.4.1 Client manager

There is only one instance of this object in the model. The client manager object represents the server in the client-server model. This object is in charge of the local-clients, remote-clients and the communication infrastructure that connects them. Some of the client manager main tasks are:

- initiate all the relevant objects that are under control of this object
- destroy these objects if needed
- synchronise the clients in the relevant stages
- collecting the results from the clients
- collecting the status and the performance from the clients
- exporting functions that take effect on all the clients to other objects.

6.4.2 Remote client

Runs as a distinct process. It communicates only with its representing local client at the main process. The local client is in charge of the collision detection scenarios computation. These scenarios are sent from the main process. The remote client keeps a list of all the distributed roots of each OBB tree that plays a part in the collision detection scenarios. Before each scenario is analysed, the remote client updates its new distributed root nodes that play a part in the scenario.

A single scenario analysis process has three major steps:

1 Getting as an input pairs of the distributed root nodes, which collide in the scenario.

2 Computing a standard collision detection of OBB trees for each pair of nodes.

3 Buffering the results in the remote client's result buffer.

When the result buffer is filled to capacity, it is transmitted back to the local client.

6.4.3 Local client

Represents a remote client, which was connected through a communication channel by the client manager. The communication uses a dedicated protocol which is described below. The information that the local client keeps about the remote client is the total number of scenarios that the remote client is currently working on, the list of allocated distributed root nodes and several statistical performance details.

The local client is in charge of transmitting the collision detection scenarios to the remote client from other components. Single transmission includes several scenarios with the purpose of minimising the amount of the requests from the main process to the client process. This explains why the local clients keep a ‘scenarios buffer’. When a buffer is filled to capacity, all the scenarios will be sent to the remote client as a collected task. The buffer will be considered as full when it contains *q* scenarios or when its size is more than *V kb* (Wiseman et al., 2008). In the experimental results section we examine the influence of various values of *q* on the performance. *V* was set according to the memory of the transmitting machines dividing by the number of clients. The scenarios and the geometry data in the buffer are saved in a binary stream buffer with the aim of saving time during the transmission.

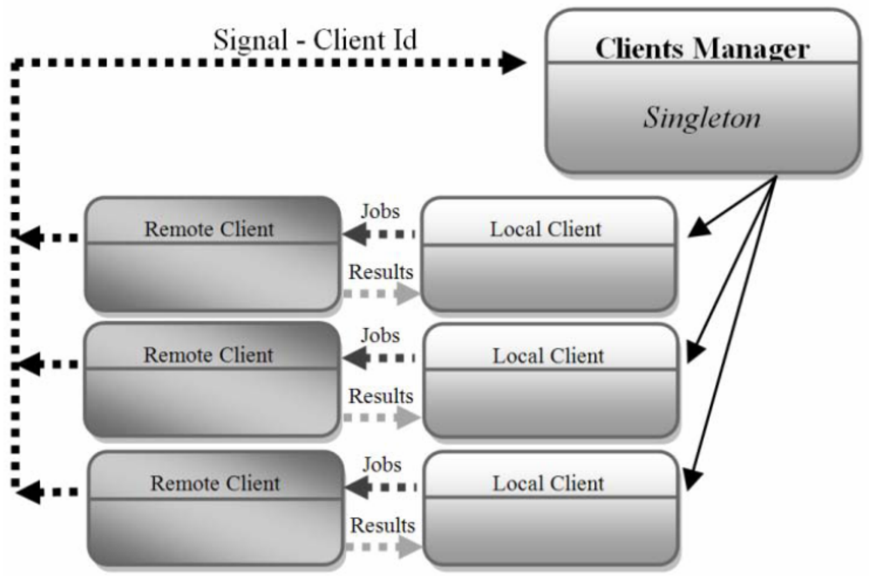
After a buffer is filled to capacity, the local client will label itself as ‘claimed’. This label prevents the client from getting new scenarios. The client will be labelled as ‘unclaimed’ just after the remote client will transmit the last task’s results.

6.4.4 Communication architecture

Our application was designed to run on top of the Mosix operating system. Mosix enables the developer to transparently refer to its computer cluster as one computer. The Mosix operating systems is responsible for a smart migration of the local processes to other machines in the cluster. This is the reason why the communication architecture in this application is based on the standard IPC, pipes.

The communication layer contains two communication channels as can be seen in Figure 8.

Figure 8 Communication architecture



Note: The dashed lines represent the different communication channels.

A channel from the local client to the remote client includes several message types:

- adding a new bounding box tree
- removing an existing bounding box tree
- transmitting a new collision detection scenario
- informing about an end of task
- terminating the client.

A channel from the remote client to the local client is used for scenario results transmission. There is another communication channel that connects all the remote clients together through one channel to the client manager. The main process samples this channel every predefined amount of scenarios transmitted, for new results. This channel is non-blocking so as to enables the main process to keep running whenever there is no result has been transmitted. Once a remote client has a result to transmit, it will signal the main process with a message containing the client id. The main process will get the signal and will call the related local client so as to inform it to collect its results.

6.4.5 OBB tree distributed nodes indicator

This object sends other objects information about the distributed root nodes of the parallel OBB tree. This object contains a pointer to a particular OBB tree and an array of integers in size of the distributed root nodes number. In our implementation, this object is used by the local client to indicate which distributed root nodes were transmitted to the remote client. This object is also used by the client transmitter to indicate which root nodes play a part in the scenario.

6.4.6 Jobs submitter

This object is in charge of the connection between the application that creates the jobs (scenarios of collision detections) and the client that receive it. Job transmission includes the two colliding OBB trees after being placed in one space according to the same coordinate system. The jobs submitter computes the colliding distributed root nodes of the OBB trees and creates an OBB tree distributed nodes indicator for each of the OBB trees to specify which nodes participate in the collision. Each distributed root node participating in the collision detection scenario is set in the integer array of the indicator object to the number of leafs it contains. A distributed root node that does not participate in the collision detection will be set to zero as can be seen in Table 1.

After generating the indicators object, the jobs submitter retrieves from the client manager a list of unclaimed clients. After that it calls the matchmaker object with the aim of finding the best-matched client to the current scenario. In the end of this process, the jobs submitter transmits the best-matched client of the current scenario for analysis.

6.5 The 'matchmaker' object

This object is in charge of the matchmaking of a collision detection scenario to the client with the most similar geometry parts. The matchmaking gets the two indicators objects that represent the scenario and a list of potential clients to be matched.

As mentioned in Section 5.1., for each client the matchmaker sums the multiplication of the integer array of the scenario indicator object with the integer array of the client's indicator object of each participating OBB trees. The client with the highest score will be the best-matched client.

6.6 Intersection between two parallel OBB trees

To facilitate the parallel collision detection simulation implemented in this paper, adjustments should be made to the standard collision detection algorithm. Thus, the parallel collision detection implemented here is split into two entities – the client and the server. The server makes pre-analysis and computes pairs of colliding distributed root nodes of the parallel OBB trees. The client continues the computation of the collision detection by computing each of the pre-computed distributed root pairs so as to obtain the exact triangles collision detection.

7 Experimental results

With the intention of showing the contribution of the algorithm suggested in this paper, we examined the algorithm using several parameters. The uniqueness of the algorithm is based on the matchmaker mechanism supplement that was added to the process of the client selection. Therefore, the comparisons tests that are shown in this section will focus on the differences between the 'matchmaker' algorithm suggested in this paper to other algorithms of client selection.

All the experimental results were conducted on the same setting consists of an heterogenic computer cluster with five computers of two dual cpus, Intel Xeon 2.8 GHz dual core and 3 GB RAM, four computers of two dual cpus, Xeon 2.4 GHz single core, and 2 GB RAM, altogether 28 cores in the cluster. The cluster was interconnected with lGbps LAN network. Linux Kernel 2.6 with Mosix 2.0 was installed on all the computers.

From the application point of view, Mosix has a non-deterministic migration decisions depending on the load of the network and the machines in the cluster. Therefore, running a test with the same parameters twice can produce unequal results. In order to minimise the error effect that this feature can generate, each result in the current section represents an average of three samples using the same parameters.

We selected two different geometries for our tests. The first one represents a heavy vehicle model that is represented by approximately 300,000 triangles and its serialised parallel OBB tree consumes approximately 62 MB. This geometry is confidential; hence we will plainly name it as 'heave vehicle'. The second geometry represents a Jaguar car model (shown in Figure 9) from http://www.3dcontentcentral.com with approximately 90,000 triangles and its serialised parallel OBB tree consumes approximately 19 MB.

With the aim of testing the collision detection system, we should run several thousands of collision scenarios. There are numerous possibilities of forming a collision scenario. There are 6 degrees of freedom for positioning geometry in space, three axial and three rotational. In our tests, we statically put one geometry at the centre of the coordinate system and we randomly transformed the other geometry around it in all its degrees of freedom. The location of the second geometry was selected with the purpose of having a small to mid collision penetration within the first geometry, because clearly big collision penetrations are not the common case in simulations and applications.

Figure 9 Jaguar car model

The number of scenarios chosen for each test depended on the geometry size. With the aim of emphasising the matchmaker's contribution to the simulation, it is important to limit the number of scenarios. At the first step of the computation, when the computer cluster spends a lot of time on acquiring the geometry data but it is required to quickly analyse scenarios, the matchmaker is mostly needed. The number limit of the scenarios was chosen in a trial and error way – for each test we seek out the limit that caused a performance drop comparing to the other algorithms.

The matchmaker algorithm described in this paper is based on finding out the client with the most similar geometry parts to the given collision detection scenario. With the purpose of testing this algorithm, we need to compare different matching strategies between a client and a scenario. We compared the following algorithms:

- *Best match* – The suggested algorithm of this paper.
- *Random match* – For each scenario, a random unclaimed client will be picked. This algorithm represents a strategy that actually does not find a connection between clients and scenarios.
- *Lowest match* – For each scenario, the unclaimed client with the geometry that less resembles the scenario is selected.
- *Best match-load* – One can argue that if a client is loaded with many scenarios, this client should not be preferred to analyse the next scenario over a less loaded client even if the less loaded client's geometry is less similar (Wiseman and Feitelson, 2003). A client load is calculated by dividing the buffered scenarios in the client by its buffer maximal size. We took into consideration both the load and the geometry similarity to the client so as to obtain the best performance.

In the tests performed in the following sections, the following parameters have been used:

- Collisions between 'heavy vehicle' to Jaguar scenarios and collisions between two Jaguars scenarios.

- Scenarios buffer size (q) has been set to 20, which is approximately the optimal value (as has been tested in Section 7.4).
- The fixed geometry has been distributed at depth 12 for the 'heavy vehicle' and at depth 10 for the Jaguar, which is approximately the optimal value (as has been tested in Section 7.3). The unfixed geometry has always been the Jaguar model in all tests and it has been distributed at depth 6.

7.1 Serial test analysis

With the aim of estimating the performance boost of each matchmaker algorithm, we first have to transmit its serial performance on a single CPU core. Table 2 and Table 3 show this data.

Table 2 The serial performance data of the 'heavy vehicle' collisions with the Jaguar

Amount of scenarios [percents of the total amount of scenarios]	*Run time [sec]*		
	Computer 1	*Computer 2*	*Weighted time*
749 [20%]	281	452	*330*
1,498 [40%]	597	953	*698*
2,246 [60%]	1,313	2,091	*1,535*
2,995 [80%]	1,961	3,127	*2,294*
3,744 [100%]	2,679	4,268	*3,133*

Table 3 The serial performance results for collisions of one Jaguar with another Jaguar

Amount of scenarios [percents of the total amount of scenarios]	*Run time [sec]*		
	Computer 1	*Computer 2*	*Weighted time*
545 [20%]	326	532	*385*
1,089 [40%]	642	1,045	*757*
1,634 [60%]	969	1,580	*1,144*
2,178 [80%]	1,308	2,132	*1,544*
2,723 [100%]	1,618	2,636	*1,909*

The label 'computer 1' represents an execution on a single core of the Xeon 2.8 GHz dual core. The label 'computer 2' represents an execution on a single core of the Xeon 2.4 GHz single core. The weighted time is calculated in respect to the number of cores in each computer in the cluster. The weighted time was calculated using the following formula:

$$\frac{n_1 \cdot t_1 + n_2 \cdot t_2}{n_1 + n_2}$$

where

n_k number of cores in the cluster of computer k

t_k run time of a single core of computer k.

7.1.1 Performance bound

Notionally, we can compute the maximum performance boost that can be gained in the cluster using the following formula:

$$n_1 + n_2 \cdot \frac{t_1}{t_2}$$

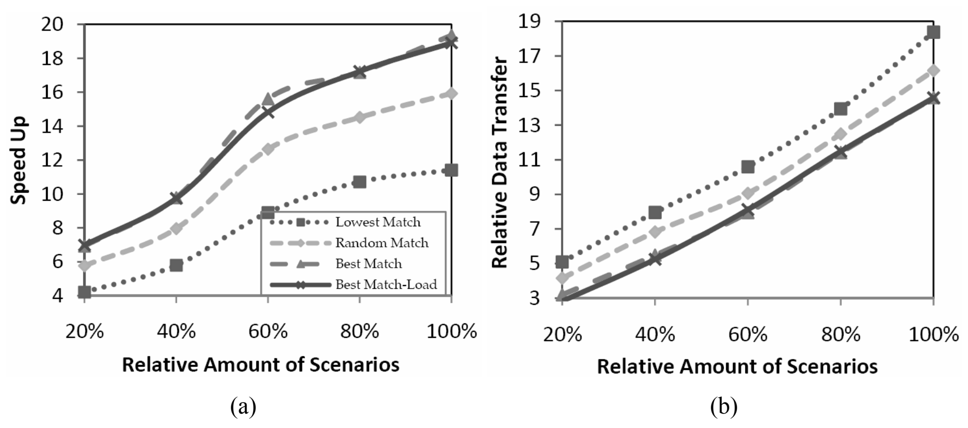
The maximal performance boost of our cluster is 25 times faster than a single core of computer 1.

7.2 Amount of scenarios influence analysis

The main data transmission overhead occurs at the early stages of the work, when all the clients have small portions of the geometries parts. At this stage, each new starting job consumes more time due to many geometry parts transmission. Testing the influence of the number of scenarios from the early stage to the saturation stage, when there is a little data transmission overhead for each job submission, is significant.

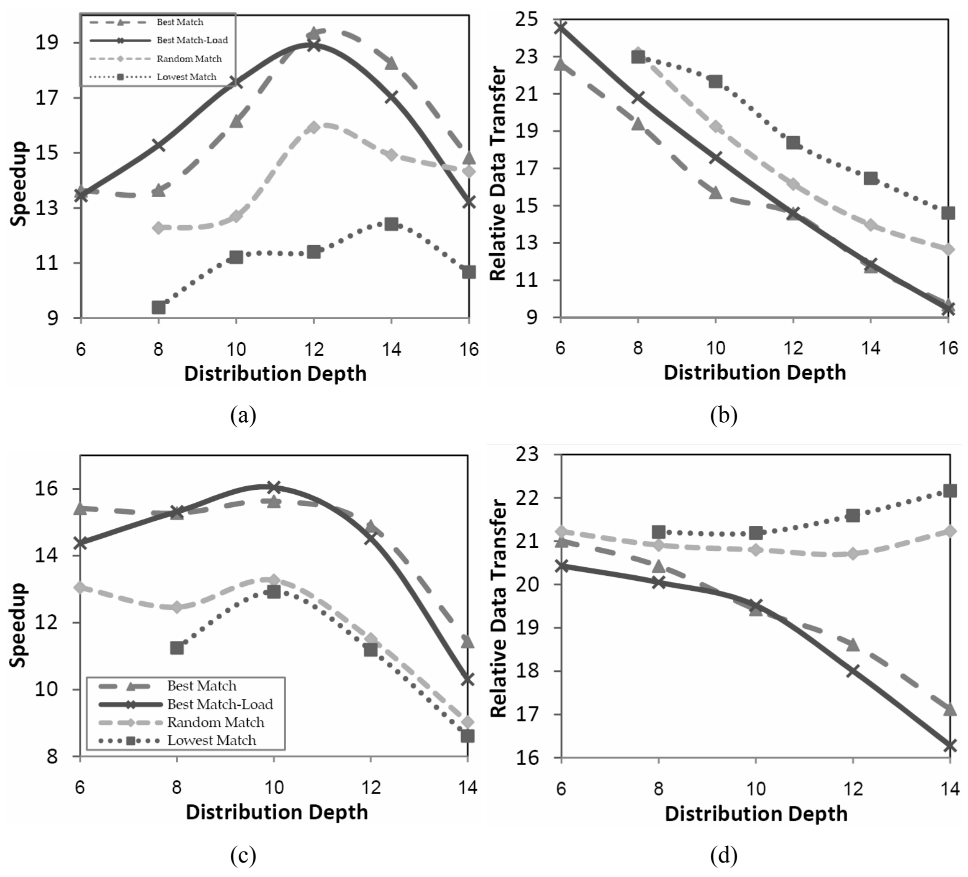
In Figure 10, we tested this influence from two different aspects, speedup and relative data transfer. The speedup is calculated by dividing the weighted serial time by the parallel time measured for each test. The relative data transfer is calculated by dividing the total amount of data transmitted to the remote clients by the entire serial size of the two test geometries, as is mentioned at the beginning of the experimental results section.

Figure 10 Amount of scenarios influence analysis on (a) speedup and (b) relative data transfer of 'heavy vehicle' with Jaguar collisions



It can be seen in Figure 11 that best match and best match-load algorithms give the best speedup. When the number of scenarios gets bigger, the ratio between the performances of these two algorithms and the other two algorithms will get bigger. At the final stage (100% of the scenarios), the speedup gets to 20 out of 25 comparing to lowest match, which gets only to speedup of 12.

Figure 11 Distribution depth analysis of (a) speedup and (b) relative data transfer of *'heavy vehicle' with Jaguar* collisions and of (c) speedup and (d) relative data transfer of *two Jaguars* collisions



As long as there are more scenarios the clients will collect more geometries portions. Best match algorithms utilise this way of collecting to reduce the data transmission overhead.

The best match algorithms save more transferred data, which causes less memory allocations at the clients, comparing to the two other algorithms.

It can be seen that best match-load algorithm does not give any significant performance improvement comparing to the standard best match algorithm.

7.3 Distribution depth analysis

Choosing the distribution depth of the OBB tree is not a trivial task. Choosing a small depth will cause the geometry being split into large overlapping bounding volumes, which will cause a retrieval of many node collision pairs. Choosing a large depth will cause the main process spending more time in analysing the first step of the collision detection and as a result may create a bottleneck.

We have examined the influence of various depths on the matchmaker algorithms. The results are shown in Figure 11. It can be noticed that best match algorithms give better performance, both in speedup and relative data transfer.

Figure 11 also shows that each geometry model has an optimal distribution depth, particularly, for ‘heavy vehicle’, the optimal distribution depth is 12 and for Jaguar, the optimal distribution depth is 10.

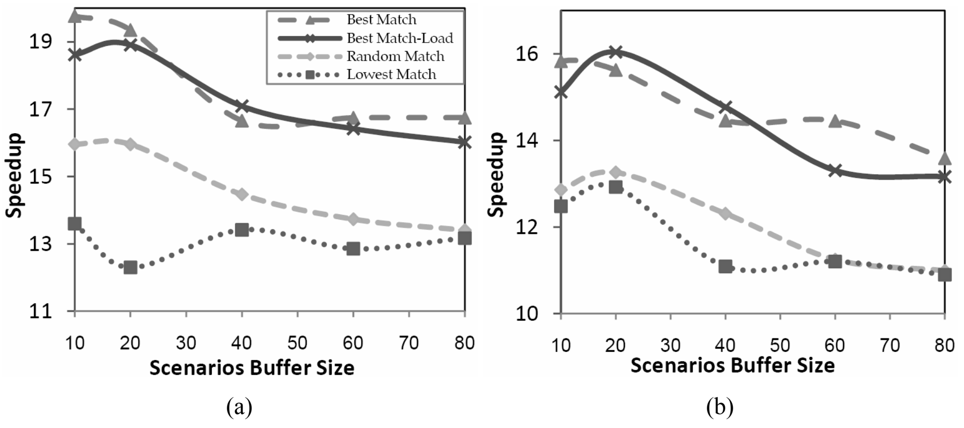
It can be concluded from Figure 11 that if the given geometry model is bigger, the relative performance of the best match algorithm will be better.

We can see in Figure 11 trimmed lines in the low distribution depth of lowest match and random match algorithms. This missing data is a result of a crash at the transmitting machine due to a lack of memory. At the initial stage, when the first jobs are transmitted, all the remote clients are still located as a process at the local (transmitting) machine waiting for being migrated to other machine in the cluster. When using a memory wasteful algorithm like lowest match or random match, the machine can quickly run out of memory and crash.

7.4 Scenarios’ buffer size analysis

The scenarios’ buffer size at the local client has a noticeable effect on the performance of the algorithms. The scenarios’ buffer gives the local client a possibility of collecting en masse several jobs and transmitting them to the remote client. As a result, the main process can devote more time to prepare tasks for other clients and reduce the number of network communications calls.

Figure 12 Scenarios’ buffer size analysis on (a) speedup of *‘heavy vehicle’ with Jaguar* collisions and (b) speedup of *two Jaguars* collisions



Choosing a small buffer size can cause an unbalanced state. One client can be constantly preferred over the others because this client will finish tasks fast and will ask for new tasks. Since this client has more parts of the geometry model than the others, it will be chosen time and again. On the contrary, choosing a big buffer can occupy the remote clients for too long time; as a result the remote clients can fail to take jobs that are very suitable for them.

Figure 12 shows the influence of several scenarios' buffer sizes on the matchmaker algorithms. We can see in Figure 12 that a buffer size 20 gives the best performance in most of the algorithms. Increasing the buffer size can cause a performance drop because the clients are occupied for too long time and consequently they lose scenarios that other less suitable clients will take.

Another reason that can cause the performance drop is the increasing transmission time that the main process takes to transmit its first tasks to the clients. The initial time of the process is crucial for decent performance because at this time period, the clients are idle and parallelism still does not take place.

7.5 Scalability analysis

The most significant advantage of the algorithm suggested in this paper is its ability to scale up its performance in respect to the size of the computer cluster. In algorithms where there is no relation between the scenarios and the clients, each added client generates more data transmission and consequently a performance drop. These algorithms do not scale up well; therefore the parallelisation will be unworthy.

In the following analysis, we present the comparison between the performances of each algorithm in respect to the amount of processing units in the cluster. As mentioned above, the cluster that was used for the tests contains two types of CPUs; one of the CPUs is almost two times faster than the other. We biased the number of processing units (can be referred as max speedup of the cluster) in respect to the faster CPU core. The relative speeds of the two different CPUs were calculated as a ratio of them and the serial results. In Table 4, we can see the cluster weights calculated for several cluster configurations.

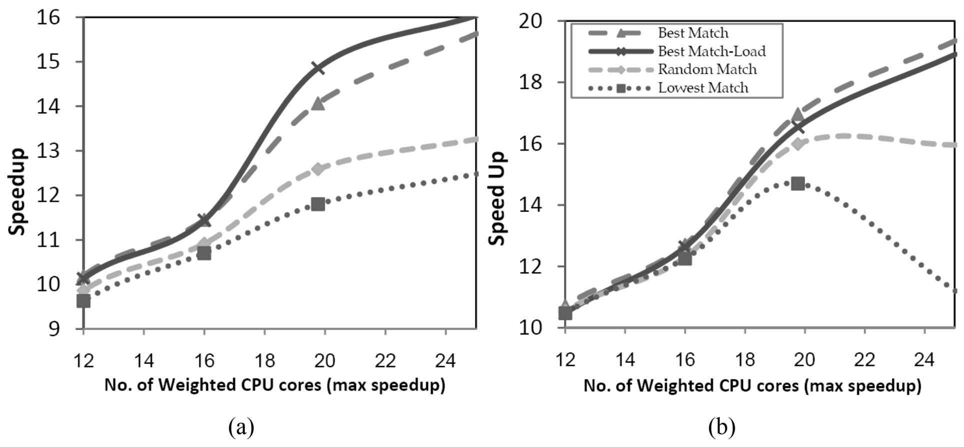
Table 4 Tested cluster configurations

No. of CPU cores		*Weighted cores*
Fast computer	*Slow computer*	
12	0	*12.0*
16	0	*16.0*
16	6	*19.8*
20	8	*25.0*

Figure 13 shows the influence of several cluster configurations on the matchmaker algorithms. It can be seen that best match algorithms are scalable to the number of processing units in the cluster. Lowest match and random match algorithms do not scale up well. We can also see in Figure 13(a) that when the tested geometries are big, it will be unworthy to use more than 20 weighted cores for the scenarios analysis for these two algorithms.

When using fewer than 16 weighted cores, the performance is equal for all algorithms. This comes about because the geometry transmission to the clients comes to an end very quickly because of the small number of clients.

Figure 13 Processing units' scalability analysis on (a) speedup of *'heavy vehicle' with Jaguar* collisions and (b) speedup of *two Jaguars* collisions



8 Conclusions and future work

Given complex geometry models, the vehicle simulation can detect an intersection in an efficient execution time. The suggested simulation cuts down the initial overhead of testing a parallel collision between complicated geometries on a computer cluster. This overhead is cut down by minimising the dependency of data transfer growth and the number of processing units in the cluster. This is the reason why the suggested simulation scales up well in respect to the cluster size and the geometries size, whereas standard algorithms fail to scale up well.

Reducing the amount of clients' memory allocation is another benefit of the suggested simulation. This reduction gives the simulation the flexibility to be ported to any given parallel infrastructure.

In the future we would like to add an ability of transferring geometry data between clients with the intention of reducing the I/O load of the transmitting machine. When a client will need a geometry portion that another client has, the clients-manager will be able to transmit the missing portion to this client.

Another interesting addition we would like to integrate into this simulation is our work on the subject of compressing the transferred data on top of the communication channel (Wiseman et al., 2004). The compression of the different parts of the OBB tree can be done in the preprocessing phase and can reduce the overhead of the data transmission.

Acknowledgements

A preliminary version appears in Proceeding of IEEE Conference on Vehicular Electronics and Safety (IEEE ICVES-2010) Qingdao, ShanDong, China, 2010.

References

Assarsson, U. and Stenstr, P. (2001) 'A case study of load distribution in parallel view frustum culling and collision detection', *Proceedings of Euro-Par 2001 Parallel Processing – 7th International Euro-Par Conference*, Manchester, UK, 28–31 August 2001, pp.663–673.

Brown, S., Attaway, S., Plimpton, S. and Hendrickson, B. (2000) 'Parallel strategies for crash and impact simulations', *Computer Methods in Applied Mechanics and Engineering*, Vol. 184, Nos. 2–4, pp.375–390.

Cohen, J.D., Lin, M.C., Manocha, D. and Ponamgi, M. (1995) 'I-COLLIDE: an interactive and exact collision detection system for large-scale environments', *Proceedings of the 1995 Symposium on Interactive 3D Graphics (Monterey, California, USA, 9–12 April 1995), SI3D '95*, ACM Press, New York, NY, pp.189–end.

Curless, B. and Levoy, M. (1996) 'A volumetric method for building complex models from range images', *Proceedings of ACM Siggraph '96*, pp.303–312.

Eberly, D.H. (2001) *3D Game Engine Design: A Practical Approach to Real-time Computer Graphics*, s.l., pp.38–61, ISBN 1558605932, Morgan Kaufmann, San-Francisco, CA, USA.

Feng, L., Chu, L. and Zechang, S. (2009) 'Intelligent vehicle simulation and debugging environment based on physics engine', *Proc. International Asia Conference on Informatics in Control, Automation and Robotics, CAR '09*, Bangkok, pp.329–333.

Figueiredo, M. and Fernando, T. (2004) 'An efficient parallel collision detection algorithm for virtual prototype environments', *ICPADS'04*.

Foley, J.D. (1995) *Computer Graphics: Principles and Practice*, s.l., pp.213–280, ISBN 0201848406, Addison-Wesley, Boston, Massachusetts, USA.

Funfzig, C., Ullrich, T. and Fellner, D.W. (2006) 'Hierarchical spherical distance fields for collision detection', *IEEE Computer Graphics and Applications*, January–February, Vol. 26, No. 1, pp.64–74.

Ghanem, T.M., Shah, R., Mokbel, M.F., Aref, W.G. and Vitter, J.S. (2004) 'Bulk operations for space-partitioning trees', *Proceedings of the 20th Annual IEEE International Conference on Data Engineering (ICDE '04)*, March–April, Boston.

Gottschalk, S.A. (2000) 'Collision queries using oriented bounding boxes', Doctoral thesis, University of North Carolina at Chapel Hill.

Grinberg, I. and Wiseman, Y. (2007) 'Scalable parallel collision detection simulation', *Proc. Signal and Image Processing (SIP-2007)*, Honolulu, Hawaii, pp.380–385.

http://www.3dcontentcentral.com

James, D.L. and Pai, D.K. (2004) 'BD-tree: output-sensitive collision detection for reduced deformable models', *ACM Transactions on Graphics (TOG)*, Vol. 23, No. 3, pp.391–396.

Jimenez, P., Thomas, F. and Torras, C. (2001) '3D collision detection: a survey', *Computers and Graphics*, Vol. 25, No. 2, pp.269–285.

Kiokes, G., Amditis, A. and Uzunoglu, N.K. (2009) 'Simulation-based performance analysis and improvement of orthogonal frequency division multiplexing – 802.11p system for vehicular communications', *IET Intelligent Transport Systems*, Vol. 3, No. 4, pp.429–436.

Klosowski, J.T., Held, M., Mitchell, J.S.B., Sowizral, H. and Zikan, K. (1998) 'Efficient collision detection using bounding volume hierarchies of k-DOPs', *IEEE Transactions on Visualization and Computer Graphics*, January–March, Vol. 4, No. 1, pp.21–36.

Lawlor, O. (2001) 'A grid-based parallel collision detection algorithm', Master's thesis, March, University of Illinois at Urbana-Champaign.

Lawlor, O. and Kale, L. (2002) 'A voxel-based parallel collision detection algorithm', *Proceedings of the 16th International Conference on Supercomputing*.

Taubin, G., Horn, W.P., Lazarus, F. and Rossignac, J. (1998) 'Geometry coding and VRML', June, *Proceedings of the IEEE*, Vol. 86, No. 6, pp.1228–1243, ISSN: 0018-9219.

Trumbore, B. (1992) 'Rectangular bounding volumes for popular primitives', in Kirk, D. (Eds.): *Graphics Gems III*, pp.295–300, Academic Press.

van den Bergen, G. (1998) 'Efficient collision detection of complex deformable models using AABB trees', *J. Graph. Tools*, January, Vol. 2, No. 4, pp.1–13.

van der Bergen, G. (2004a) 'Collision detection in interactive 3D environments', Chapter 5 – Spatial Data Structures, pp.171–217, ISBN 155860801, Morgan Kaufmann Publishers Inc., San Fransisco, CA, USA.

van der Bergen, G. (2004b) 'Collision detection in interactive 3D environments', Chapter 4.3.4 – Support Mapping, pp.130–139, ISBN 155860801, Morgan Kaufmann Publishers Inc., San Fransisco, CA, USA.

Wasmund, L.T. (2001) 'New model to evaluate weapon effects and platform vulnerability: AJEM', *WSTIAC*, Vol. 2, No. 4, pp.1–3.

Wiseman, Y. and Feitelson, D.G. (2003) 'Paired gang scheduling', *IEEE Transactions on Parallel and Distributed Systems*, Vol. 14, No. 6, pp.581–592.

Wiseman, Y., Isaacson, J. and Lubovsky, E. (2008) 'Eliminating the threat of kernel stack overflows', *Proc. IEEE Conference on Information Reuse and Integration (IEEE IRI-2008)*, Las Vegas, Nevada, pp.116–121.

Wiseman, Y., Schwan, K. and Widener, P. (2004) 'Efficient end to end data exchange using configurable compression', *Proc. the 24th IEEE Conference on Distributed Computing Systems (ICDCS 2004)*, pp.228–235, Tokyo, Japan.

Woo, M., Neider, J. and Davis, T. (1999) 'OpenGL programming guide', *Third Edition: The Official Guide to Learning OpenGL*, No. 3, s.l., ISBN: 0-201-60458-2, Addison-Wesley.

Yong, W., Jing, Z. and Ping-bo, W. (2008) 'Dynamics simulation model of double united articulated container flat vehicle', *Journal of Traffic and Transportation Engineering*, Vol. 1, No. 3, pp.184–188.