A Case Study of Load Distribution in Parallel View Frustum Culling and Collision Detection

Ulf Assarsson[1] and Per Stenström[2]

[1] ABB Robotics, Drakegatan 6, SE-412 50 Göteborg, Sweden
uffe@ce.chalmers.se,
[2] Department of Computer Engineering Chalmers University of Technology SE-412 96, Göteborg, Sweden
pers@ce.chalmers.se

Abstract. When parallelizing hierarchical view frustum culling and collision detection, the low computation cost per node and the fact that the traversal path through the tree structure is not known à priori make the classical load-balance versus communication tradeoff very challenging. In this paper, a comparative performance evaluation of a number of load distribution strategies is conducted. We show that several strategies suffer from a too high an orchestration overhead to provide any meaningful speedup. However, by applying some straightforward tricks to get rid of most of the locking needed, it is possible to achieve interesting speedups. For our industrially related test scenes, we get about a four-fold speedup on eight processors for view frustum culling and three times speedup for collision detection.

1 Introduction

View frustum culling (VFC) and collision detection are two very common components of real time computer graphics applications. VFC aims at reducing the computational complexity of a succeeding rendering pass by extracting the graphics objects that are in the view frustum. For hierarchical VFC, a hierarchy is built up as a tree structure from the bounding volume of each object. Each node in the tree has a bounding volume enclosing a part of the scene. The tree is traversed from the root in a depth-first manner, and if a bounding volume is found to be outside the frustum during the traversal, the contents of that subtree can be culled from rendering. The typically low computation cost makes the load distribution in a parallel implementation extremely challenging.

In this paper we evaluate the effectiveness of a set of load distribution strategies on parallel implementations of hierarchical view frustum culling with scenes from an industrial application. We also examine the capability of the most promising scheme applied on collision detection. For VFC we use axis aligned bounding box (AABB) trees [8], while for collision detection we use both AABB- and oriented bounding box (OBB) trees [4].

The load distribution schemes we select are a global task queue, and a number of distributed task queue schemes well-known from the literature. We evaluate the speedup of the parallel implementations using these strategies on a 13-node Sun Enterprise shared-memory multiprocessor and on a dual PentiumIII 500 MHz personal computer.

We find that while some of the schemes were expected to provide a reasonable speedup, they performed inferior owing to the high communication and synchronization cost. Our results show that due to the low computation cost per node compared to the distribution cost, only the more sophisticated lock-free scheme provides interesting speedup numbers. By considering a number of optimizations – especially by getting rid of the synchronizations – we managed to get promising results, even for highly unbalanced industrial scenes. For our scenes, we achieve a speedup of around four on eight processors for view frustum culling and about three on seven processors for collision detection with real test cases from an industrial case study.

2 Experimental Set-Up

The code for testing a bounding volume against the view frustum is the one of a previously proposed optimized algorithm [1]. This implements many optimizations such as caching of previous computations, implying little computation cost per node in many cases. Other optimizations include plane-coherency, octant, and translation and rotation coherency tests (see [1] for details).

We use three trees that are the hierarchical scene graph representations of three 3D models - all of real environments and all used in industrial applications. The three highly unbalanced trees used in the tests are: a car factory shop floor in $3,932$ graph nodes, a factory shop floor in $1,137$ graph nodes and a factory cell in 254 graph nodes. We refer to them as the *large model*, the *medium model*, and the *small model,* respectively.

The camera–or view frustum–used in the view frustum culling computations is moved along one specific path for each model, each sampled from a user walk through in the model. The presented traversal times and speedups are the average times and average speedups of all traversals during the walk through.

The experiments are carried out on a Sun Enterprise 4000 shared-memory multiprocessor. This machine is equipped with 14 UltraSPARC-II CPUs running at 248 MHz. Each CPU is attached to a 16-Kbyte L1 data cache and a 1-Mbyte L2 cache, both using a line size of 32 bytes. The locks used have been implemented using the SPARC-instruction `ldstub` which loads a byte followed by a store that sets all bits in that byte atomically. We only show results for up to 13 processors. One processor is left for the operating system to avoid the perturbation it would cause when it is invoked every millisecond.

3 Evaluation of Load-Distribution Schemes

In this section, we consider the effectiveness of load distribution strategies that seem adequate for the dynamic behavior of our workload. As a reference, we use the classical global task queue scheme which we consider first.

3.1 Global Task Queue

In this approach, each processor removes and add tasks (tree-nodes) using a global task queue. The virtue is good load balance while the overhead associated with orchestrating the global task queue is known to be high.

Results from the experiments of parallel VFC are presented in Figure 1. Figure 1.a-1.c show the average speedup, and Figure 1.e-1.f show the average execution time for VFC of one frame.

For the global task queue, the maximum number of processors that can provide speed-up, before the global task queue becomes the bottle-neck, is limited to the total time for processing a node divided by the time for accessing the global queue (*node_cost/ access_cost*). We see that we get a maximum speedup of only 1.5, with only three processors on the small model. Moreover, when we increase the number of processors, the speedup goes down owing to serialization effects, as expected.

3.2 The Global Counter Scheme

A more scalable strategy is to associate a local task queue with each processor. Each processor adds tasks to the local queue pointed to by a global counter that is incremented after each insertion by any processor and protected by a lock[1] according to [11]. In this way the load will be nearly optimally balanced if all processors can process nodes equally fast. The serialization of accesses to one single queue is replaced by the serialization of reading and incrementing the global counter, which is usually faster. However, the lock mechanism around the counter can potentially become a new bottleneck when we increase the number of processors. In addition, the locks that synchronize the accesses to the queue attached to each processor is another potential bottleneck.

As can be seen in Figure 1a, compared to the global task queue algorithm, the stagnation in speed-up which peaks at about 1.9, comes later – at more than eight processors instead of three, which is expected since incrementing a counter is quicker than inserting or removing a task (which in our implementation basically consists of changing an array index and reading the contents of the array element, i.e about twice the cost). The stagnation comes from the global lock which gives a high cost and introduces serialization.

3.3 The Hybrid Scheme

To further reduce the orchestration overhead and contention due to locking and shared memory access, we considered two optimizations of the global counter scheme. The resulting scheme is referred to as *hybrid.*

– **The skip-pointer tree optimization:** A common optimization in raytracing is to represent the tree in depth-first order in an array [16], with a skip

[1] For some processors it is possible to atomically read and increment a variable with just one or two assembler instructions instead of using a lock.

index for each node that points out the next element to access if the underlying sub-graph should be skipped during the traversal. Then a full tree traversal can be performed by simply accessing the array sequentially from start to end. Every subtree will be represented in the array as a consecutive chunk of elements, so instead of distributing a node (subtree), we send the start-index and the stop-index of the array. While it provides good cache-locality in the sequential single processor case, it can also give better locality in the parallel case.

– **Trading off larger tasks for less load balance:** This straightforward optimization uses the observation that at a certain depth, when the underlying subtree only contains a few nodes, it will be faster to process the nodes rather than distributing them, if the computation-cost is smaller than the distribution-cost [13].

Since the size of each subtree is not known beforehand, the heuristic we have tried is to distribute tasks at the node-level until a certain level after which the rest of the subtrees are considered as tasks. The first phase uses the global counter scheme according to Section 3.2, whereas the second phase serially executes the tree traversal algorithm with no further balancing of the load. Both phases use the skip pointer optimization and thus will enjoy the increased locality it provides. A counter keeps track of how many nodes that so far have been processed by the distribution algorithm. If a threshold number is exceeded, all processors finish the computations and distribution of children for the node it is currently working on, and enter the serial phase. We found empirically that a threshold of six times the number of processors gave the best performance for our models with a difference in load of less than 2% for the large model.

The skip-pointer tree optimization contributed with an overall speedup of $15 - 40\%$ compared to the global counter scheme. Despite the possibility to also trade between load balance and larger tasks, the total speedup for both optimizations together peaks at only 2.2 times (for 10 processors).

We also made measurements showing that if the cost of the VFC computations at each node were virtually zero, we would get a huge slowdown using more than one processor. The reason is the high distribution cost compared to the cost of the serial traversal of the skip-pointer tree. Skipping the distribution phase, resulting in a serial single processor algorithm, would actually have been optimal for this case.

The schemes used so far suffer from too much overhead, especially concering lock accesses. This motivated us to seek for a lock-free approach which we study in the next section.

4 A Lock-free Scheme

The Lock-Free scheme distributes the load without requiring locks or any synchronization. The way we adapted the original scheme to avoid locking is as follows.

Each processor has one local-queue and some in-queues. A processor removes tasks from its local-queue and its in-queues, and adds new tasks to its local queue and dedicated in-queues of neighboring processors. The in-queues are created such that one processor can insert tasks at one end of the queue and another processor can remove tasks from the other end of the queue, without any need for synchronization between the two. There is one dedicated in-queue for each sender/receiver pair. We use a ring buffer with two indices to point out the start and the end of the buffer.

The `insert()` method only needs to affect the start-index, and the `remove()` method only needs to affect the end-index. It is easy to assure that the `insert()` and `remove()` operations never can access the same memory location simultaneously.

The `remove()` operation needs to check if the queue is empty before allowing removal of a task, and because the `insert()` operation always inserts a task into the array before incrementing the end-index, computing `end - start` will always give a safe result. The same safe situation holds for the `insert()` method, when checking if the queue has room for more elements before inserting a task. The array simulates a ring and the indices will wrap around to the first element after passing the last element of the array, but this is easy to adjust for.

Since we want to avoid locks completely, we only allow a processor to either insert or remove jobs from an in-queue - not both. The opposite could be interesting to try, since there are ways to implement this such that the locks, with a high probability, seldom will be used [3].

4.1 Topology

In order to easily change the number of processor connections in the topology, we first order the processors virtually in a ring, where each processor distributes tasks to its successor's in-queue. When increasing the connectivity and wanting every processor to send tasks to n receivers, with p processors in the ring, we add connections to every $(\frac{p}{n} + 1)$:th successor. When inserting a connection between two processors, we assign an in-queue for the receiver and let the sender send tasks to this queue. Figure 1.h) shows an example of 6 processors, each distributing to 3 receivers.

Load Balancing For Adaptive Contracting within neighborhood (ACWN), the least loaded nearest neighbor is always selected as the receiver of a newly generated job. It is known that local averaging strategies generally outperforms methods such as the randomized allocation and the ACWN algorithm significantly in large scale system [17]. Since our shared memory system is a so called one-port communication system (i.e at most one neighbor can receive a message in a communication step) with one central data bus, we use the Local Averaging Dimension Exchange (LADE) policy. Generally it is better than the diffusion method (LADF) on such a system [18]. In LADF, load balancing is done with all neighbors, while in LADE load balancing is only done with one of the neighbors, or one at a time with the new load-balance successively considered.

Our approach is to use a sender-induced rather than a receiver-induced load distribution strategy. An advantage of the receiver-induced approach is that tasks are only distributed on demand which potentially reduces the overall cost of distribution. A disadvantage, however, is that processors may sit idle to wait for tasks to be available which may waste computing cycles. We briefly tried some receiver-induced approaches for the lock-based schemes, but they were inferior to the sender-induced, and thus we decided to try the sender-induced policy first for the lock-free schemes. Cilk-5 [3] is a parallel development system that uses the other approach (see section 6).

A high degree of connections between processors in the virtual topology enables better load-balancing. Since the communication is the bottle-neck and the computation cost at each node in the tree is low, we need a simple/fast load-balancing scheme. Only sending newly generated jobs to each receiver and to the local queue in a round-robin fashion, was found to be insufficient to maintain good load-balance. We needed to consider the load difference between processors, which costs computation and communication. If a processor has more jobs than the receiver, it sends half the difference of the load. However, empirically we found that it was enough to even out the load balance this way with only one of the receivers and send blindly to the rest, to get similar load balance as the global task queue. We chose to consider the load balance difference only with the successor in the main ring. If n jobs are transferred in this step, we wait at least n traversed nodes before trying to load-balance carefully again, since load balancing is expensive and the successor probably will have work to do at least the corresponding time. In the final algorithm, after every processed node we distribute the newly generated jobs to the local-queue and the receiving processors in a round-robin fashion. If $n = 0$, where n is a variable set to the number of tasks sent to the successor last time and decreased after every traversed node, we also do the extra load-balancing with the successor. Every time we distribute jobs to the successor we may increment n.

If we have a topology with many connections for each processor, we potentially risk lowering cache-locality when we spread the jobs over many queues. In the shared memory system, the jobs are physically sent when the receiving processor reads its in-queues and the corresponding cache-blocks are transferred from the sending processor to the receiving. In order to minimize the number of cache-block reads, the receiving processor selects one in-queue for reading until it is empty, before selecting a new in-queue. We could also avoid using an in-queue for reading that does not fill up an entire cache-block, if there are others that do, but we did not implement this.

In general, a high number of connections between processors in the virtual topology seemed to be preferred (see tables at the side of Figure 1.a-1.c).

4.2 Experimental results

For this scheme, the speedup is substantially better for the large and medium model, with 4.3 and 3.1 times respectively. For the small model it is only 1.7, but this model provides poor speedup for all the schemes. Load balance is similar to what that of the other schemes.

It was found that the time for just traversing the trees in parallel, not doing any VFC-computations, was fairly constant independently of the number of processors used. This means that we can decompose the total execution time as:

$$time_{total} = time_{traversal} + time_{VFC} \tag{1}$$

where $time_{VFC}$ is the only term that enjoys speedup from the parallelism in VFC. This speedup, however, is basically optimal with respect to the possible parallelism provided by the traversed paths.

Depending on which parts of the scene-graph that are visible in a frame, the maximum of possible parallelism can vary, since there is a limited amount of parallel paths in the traversed graph. We found that if the whole tree is traversed, with each child selected for continued traversal disregarding the result of the VFC computations, the speedup peaks at 5.1, which is slightly higher than the average speedup. This indicates that the speedup is limited by the appearance of the scene graph. Since it represents a bounding box hierarchy, we cannot rearrange the graph without caution.

We also tested the Lock-Free scheme on a 2-processor PentiumIII 500MHz, with 256 Mb RAM, with a simpler load distribution policy that just keeps every 2:nd child and distributes the other to the other processor. The topology is a virtual ring of 2 nodes. With this approach we got 1.7, 1.5 and 1.3 times speedup for the large-, medium- and small model respectively. The load balance was practically perfect.

5 Collision Detection

Since the lock-free scheme was pretty successful in parallelizing VFC we tested it on hierarchical collision detection to see how it performs on this similar type of problem. We kept the same load balancing strategy. Collision detection is known as non-trivial to parallelize [14].

To find collision between two objects, their bounding box hierarchies are tested against each other for overlap. If any of the leaves between the two trees intersect, the objects are considered colliding. The algorithm starts with the root boxes of both trees. If intersection occurs, the algorithm continues recursively by testing the smallest of the two boxes (or the one that is not a leaf) against the children of the larger box respectively. If both boxes are leaves, a collision is found and the algorithm terminates. In this way a virtual graph is traversed.

A hierarchical AABB-tree of a small industry-robot with 102 nodes and a tree-depth of 11, was tested for intersection against the large model (a car factory). The robot was spatially placed such that the algorithm is forced to traverse deep down in both trees to verify that collision (in this case) not occurs.

Testing two AABBs against each other for overlap is extremely fast and basically consists of just 6 compares, while testing two arbitrarily oriented bounding boxes (OBBs) costs about 200 flops in average [4]. OBBs, however, can be more tight fitting and are thus often preferred. We wanted to test both cases. In the OBB-case, for simplicity, the AABBs were treated as OBBs in the overlap-computation with the orientation incidentally coinciding with the x,y,x-axes.

We found that for collision detection as well as for VFC, the traversal time without collision computations was nearly independent of the number of processors used. Consequently, since AABBs are very fast to test for overlap, we only got very limited speedup - 30% with 4 processors. For OBBs, however, the speedup peaks at 3.2 as can be seen in Figure 1.d).

6 Related Work and Discussion

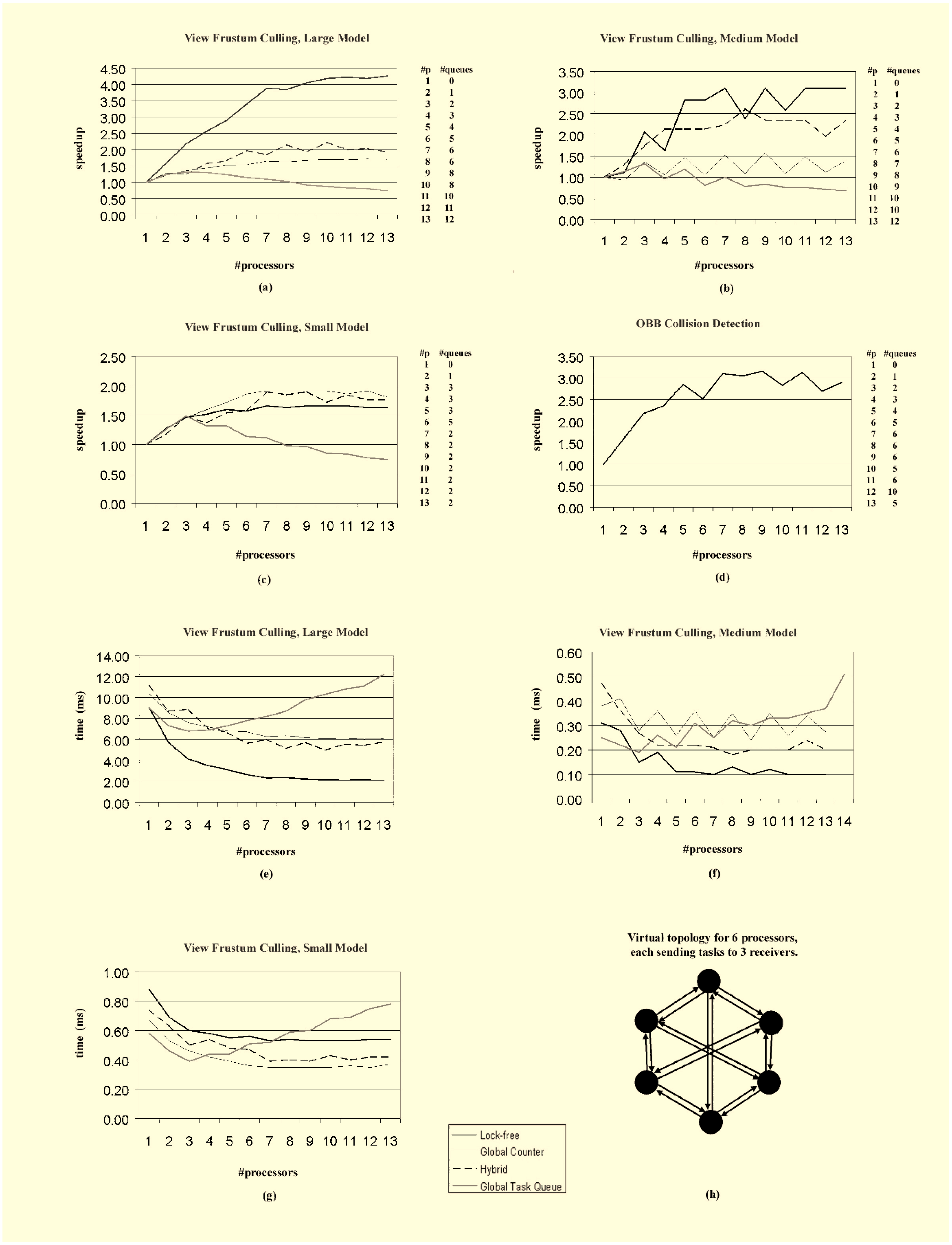
Several older parallel branch-and-bound techniques [2, 5, 7, 9, 10, 19] and depth-first search algorithms like backtracking [11–13] seem at a first glance to be applicable to the applications we have at hand. Our results indicate, however, that the load distribution strategies in these algorithms do not apply very well to tree traversals found in VFC and collision detection because of the low computation cost per node compared to the distribution cost.

In this paper we have focused on sender-induced schemes since this seemed most promising for the lock-based approaches. However, Cilk-5, which has been available for a short time, uses task-stealing in a way that looks promising. It requires the use of locks, but there are convincing arguments that they seldom will cause contention or significantly increased communication. Two of the main features of Cilk-5 is 1) that it compiles two versions of the code: one serial and one parallel, and can switch in run-time when load-balancing requests are issued, and 2) that load-balancing can occur efficiently through queues similar to those we use in our lock-free schemes.

Other related work that aims at reducing the orchestration overhead in tree traversals includes using prefetching techniques to tolerate communication latencies in the system. Karlsson et al. [6] studied how annotation of prefetch instructions can speed up tree traversals to tolerate the latency of cache misses. They especially considered the class of tree traversals where the traversal path is not known beforehand and obtained encouraging results. While they studied only sequential tree traversals it would be interesting to study the potential for parallel tree traversals.

7 Conclusion

In this paper we have presented a comparative evaluation of load distribution strategies based on a real application case study including two important computer graphics algorithms used in virtual reality. The low computation-to-communication ratio in these algorithms make load distribution particularly challenging. Based on some minor – but important – adaptations of well-known load distribution schemes in the literature, we managed to demonstrate reasonable speedups on a symmetric multiprocessor. Since multiprocessors of this scale are now being used in personal computers, and are seriously considered to



(a)

#p	#queues
1	0
2	1
3	2
4	3
5	4
6	5
7	6
8	6
9	8
10	8
11	10
12	11
13	12

(b)

#p	#queues
1	0
2	1
3	2
4	3
5	4
6	5
7	6
8	7
9	8
10	9
11	10
12	10
13	12

(c)

#p	#queues
1	0
2	1
3	3
4	3
5	3
6	5
7	2
8	2
9	2
10	2
11	2
12	2
13	2

(d)

#p	#queues
1	0
2	1
3	2
4	3
5	4
6	5
7	6
8	6
9	6
10	5
11	6
12	10
13	5

Fig. 1. (a-c) Speedup with 1 to 13 processors for the large, medium and small model. For the lock-free scheme, the figures are for the best topology, with the number of connections (in-queues) per processor marked at the side. (d) Speedup for collision detection with an OBB-algorithm with the lock-free scheme. The jaggedness comes from the difference in topology and number of optimal connections. (e-g) Corresponding execution time for the algorithms. (h) Virtual topology for 6 processors where each processor distributes load to 3 other processors. Note that depending on the camera position, a larger tree can be faster to traverse than a smaller. This is the case for the small vs. medium model, where the small offers more immerse navigation.

migrate to the chip-level, our results are indeed encouraging. They show that multiprocessors can be exploited for an emerging class of real-time computer graphics applications.

Acknowledgments

We would like to thank ABB Robotics Products, for the financial support of Ulf Assarsson's research. This research has also been supported by the Swedish Foundation of Strategic Research (SSF) financed ARTES/PAMP program, Sun Microsystems Inc, and by an equipment grant from the Swedish Council for the Planning and Coordination of Research (FRN) under contract 96238.

References

1. Ulf Assarsson and Tomas Möller, "Optimized View Frustum Culling Algorithms for Bounding Boxes", Journal of Graphics Tools, 5(1), Pages 9-22, 2000.
2. E. W. Felten, "Best-first Branch-and Bound on a Hypercube", Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, (Vol. 2), Pages 1500-1504, 1988.
3. Matteo Frigo, Charles E. Leiserson, and Keith H. Randall, "The Implementation of the Cilk-5 Multithreaded Language", ACM SIGPLAN Conference on Programming Language, 1998.
4. S. Gottschalk, M.C Lin, and D. Manocha, "OBBTree: A Hierarchical Structure for Rapid Interference Detection", Proc. of ACM Siggraph, Pages 171-180, 1996.
5. V. K. Janakiram, D. P. Agrawal, and R. Mehrotra, "A Randomized Parallel Branch-and-Bound Algorithm", in Proc. Int. Conf. Parallel Process., Pages 69-75., Aug. 1988.
6. M. Karlsson, F. Dahlgren, and P. Stenström, "A Prefetching Technique for Irregular Accesses to Linked Data Structures", Proc. of 6th Int. Symp. on High Performance Computer Architecture, Pages 206-217, Jan. 2000.
7. Richard M. Karp, Yanjun Zhang, "Randomized Parallel Algorithms for Backtrack Search and Branch-and-Bound Computation", Journal of the ACM, Volume 40, Pages 765-789, Issue 3, 1993.
8. Tomas Möller and Eric Haines, "Real-Time Rendering", A.K. Peters Ltd, ISBN 1-56881-101-2, 1999.
9. Roy P. Pargas and E. Daniels Wooster, "Branch-and-Bound Algorithms on a Hypercube", Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, (Vol. 2), Pages 1514 - 1519, 1988.
10. Michael J. Quinn, "Analysis and Implementation of Branch-and-Bound Algorithms on a Hypercube Multicomputer", IEEE Transactions on Computers, vol. C-39, Pages 384-387, no. 3, March, 1990.
11. V. Nageshwara Rao and Vipin Kumar, "Parallel Depth-First Search on Multiprocessors — Part I: Implementation; and Part II—analysis", International Journal of Parallel Programming, vol. 16, no. 6, 1987.
12. V. Nageshwara Rao, Vipin Kumar, "On the Efficiency of Parallel Backtracking", IEEE Transactions on Parallel and Distributed Systems, vol 4, no. 4, Pages 427–437, April, 1993.

13. A. Reinefeld, V. Schnecke, "Work-Load Balancing in Highly Parallel Depth-First Search", Proc. Scalable High Performance Computing Conf. SHPCC'94, IEEE Comp. Sc. Press, Pages 773-780, 1994.
14. Peter Rundberg, "An Optimized Collision Detection Algorithm", http://www.ce.chalmers.se/staff/biff/exjobb, 1998.
15. A. Saulsbury, F. Pong, and A. Novatzyk, "Missing the Memory Wall: The Case for Processor/Memory Integration" Proc. of 23rd Int. Symp. on Computer Architecture, Pages 90-101, June, 1996.
16. Brian Smits, "Efficiency Issues for Ray Tracing", A K Peters, Ltd, Journal of Graphics Tools, vol 3, no 2, Pages 1-14, 1999.
17. C. Xu, S. Tschoke, and B. Monien, "Performance Evaluation of Load Distribution Strategies in Parallel Branch and bound Computations", Proc. of the 7th IEEE Symposium of Parallel and Distributed Processing (SPDP95), Oct. 1995.
18. C. Xu and R. Lüling and B. Monien and F. Lau, "An analytical comparison of nearest neighbor algorithms for load balancing in parallel computers", Proceedings of 9th International Parallel Processing Symposium, 1995.
19. Myung K. Yang, Chita R. Das, "Evaluation of a Parallel Branch-and-Bound Algorithm on a Class of Multiprocessors", IEEE Transactions on Parallel and Distributed Systems, vol. 5, no. 1, January, 1994.