Proceedings of the 2002 Winter Simulation Conference
E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes, eds.

TECHNIQUES TO ENHANCE PERFORMANCE OF AN EXISTING AVIATION SIMULATION

David Carnes
Frederick Wieland

The MITRE Corporation
7515 Colshire Drive
McLean, VA 22102-7508, U.S.A.

ABSTRACT

Facing a need to run large scenarios on aviation models more quickly than the one to two days currently required, the MITRE Corporation undertook an effort to reduce the execution time of one such simulation. Time and cost constraints prohibited a major rewrite of the almost one million existing lines of code, so only solutions requiring minimal changes to the code base were considered. This paper describes the approaches taken to increase the speed of the original sequential simulation by employing more efficient algorithms and parallel processing technology. Specifically, an implementation of a new technique for parallel proximity detection provided an 80% reduction in the time spent checking for conflicts. In addition, implementation of a thread pool that enables the movement of multiple aircraft in parallel resulted in a 10%-15% reduction in the overall execution time of the simulation. In this paper we report on the design of these techniques and how they were implemented in the simulation.

1 INTRODUCTION

The MITRE Corporation's Center for Advanced Aviation Systems Development (CAASD) is streamlining a sequential air traffic control simulation containing nearly one million lines of code. The main constraint in speeding up this simulation is to minimize changes to its architecture.

One of its main bottleneck algorithms is determining whether and when two aircraft are within a specific distance. If the model detects a conflict between aircraft pairs, then the model can, using built-in conflict resolution strategies, vector the aircraft away from one another. When conflict detection is enabled by the analyst, it can consume up to 50% of the total run time in some scenarios. Updating aircraft state and position data dominates the run time when conflict detection is not enabled.

Another major bottleneck in the simulation is aircraft navigation, or movement. During the movement cycle, the current position of each aircraft is updated. The simulation also maintains a list of future aircraft positions for use with conflict detection and resolution that is also updated at this time. All sectors and airports that the aircraft may interact with during its flight are notified of these changes. Those sectors and airports may then reschedule the arrival of other aircraft that are affected by the original update in order to optimize flow. This can consume up to 70% of the simulation run time when conflict detection and resolution are not enabled.

Regarding proximity detection, most of the existing algorithms require extensive changes to the system, and cannot be used. The main concern of any proximity detection algorithm is efficient filtering of pairs of moving objects. We have devised a more efficient filter to reduce the number of aircraft that need to be checked during each cycle. This method also allows us to parallelize the remaining conflict checks in a manner that requires virtually no change to the underlying software architecture.

Our method is grid-based, employing a quad tree data structure that is normally found in the context of computer graphics (Samet 1990), and in conflict-free route planning in the field of robotics (Hamada and Hori 1996). The geometric calculations determining proximity should only be performed on pairs most likely to be in conflict. In the existing simulation, these filters are applied using bounding boxes around the aircraft's intended flight plan. We have replaced that process with a quad tree based algorithm that geographically filters the flight pairs in a manner unrelated to the aviation logic or flight dynamics in the simulation.

Regarding the movement algorithm, the story for parallelization resembles that of the conflict detection algorithm. In fact, much of the implementation of the quad tree parallelization was reused for this work. In general, the system works by queuing requests to update a specific aircraft, and separately allowing a set of worker threads to work independently on each aircraft.

2 PRIOR WORK

Parallel proximity detection has been extensively studied and reported in open literature. The available algorithms fall into two different categories. The first category encompasses the so-called grid based algorithms, which rely on dividing a simulated space into cells that "tile" the space. The grid based algorithms require sophisticated protocols for keeping track of objects that are near grid boundaries and for controlling the hand off of the objects as they move. The second category of algorithms encompass region based algorithms, which define a region in which an object declares an interest. The region based algorithms require common definitions of regions across objects sharing the same region, and require sophisticated geometric calculations in the simulation engine that, if done in full generality, can add much overhead to the system. Recently, hybrid algorithms that dynamically size the grids or combine the region and grid based approaches have emerged (Boukerche, Roy, and Thomas 2000).

Other variations are possible. Tan, Zhang, and Ayani (2000) present a hybrid algorithm that combines the best approached of both the grid and region based filtering in the context of the DOD's HLA. Steinman and Wieland (1994) describe a grid-based algorithm where the grid boundaries are extended and overlapped to help minimize problems along grid boundaries. A later version of the algorithm (Steinman et.al. 1999) describes a spherical virtual space tiled with grid cells of approximately equal size. Within each cell, a 3-way tree is produced to act as an index into the x, y, and z positions of an object. Thus the three-way tree serves as a fast indexing mechanism for position.

Quad trees themselves are not new data structures. Prior work that used quad trees for parallelization primarily focused upon efficiently scheduling ready processes to available hardware; Srisawat and Alexandridis (2000) present a good example. Quad trees have also been used as a basis for communication topologies in parallel processors, such as quad tree based hypercube communication interconnects (Omri 2000). Collision detection of moving objects using quad trees, and their three dimensional extension to oct trees, has also been studied (Kitamura et.al. 1994) (Tzafestas and Coifffet 1996). Basch, Guibas, and Zhang (1997) describe the problem of using trees in their generality for geometric proximity problems. An overview of the general problem of conflict detection and resolution in the context of aviation simulation is presented by Vink and Kauppinen (1997).

Automatic parallelization of sequential simulations also has a long history of study. Some researchers have focused upon changing the underlying simulation engine to add constructs that allow modelers to exploit parallelism; a good example of this can be found in Nicol and Heidelberger (1995) and Nicol and Heidelberger (1996). Others have focused on the tasks required to explicitly parallelize a sequential simulation model (Bajaj, Bagrodia, and Meyer 1999). A good introduction to the concept of thread pools can be found in Nichols, Buttlar, and Farrell (1996).

The work described above differs from previous studies in a number of respects. First, the quad tree approach uses a quad tree as the basis for spatially decomposing the virtual world into unequal-sized grid cells. The basis for sizing is to maintain a high degree of filtering for potential conflicts. Furthermore, the sizes of the cells dynamically change with time as the simulation evolves, and as the spatial density of objects change. Finally, load balancing in both the conflict detection and movement phases is provided by queuing the work to be performed for processing by a separate set of "worker" threads.

Finer grained locking may also be a valid approach to reducing lock contention, but size of the simulation makes this approach difficult. That approach also conflicts with the goal of leaving the existing sequential simulation model and architecture undisturbed. The overhead and benefit of this approach would be difficult to judge until it is attempted.

3 CONFLICT DETECTION IN THE SIMULATION

We began by examining the largest bottleneck in the simulation, conflict detection. A conflict occurs when the geometric distance between any two aircraft in the simulation falls below the specified lateral and vertical separation thresholds. These thresholds are set by the analyst, and can vary by the air traffic control sector in which the aircraft is currently flying.

The simulation commences execution with the earliest scheduled event provided by the scenario's itineraries. It then steps through time by discrete intervals. The analyst can vary this interval but it is typically one to six seconds. Conflict detection begins at the time an aircraft is introduced in the simulation and becomes active. At that time, the *flight envelope* is computed for that aircraft, and the path of a ghost aircraft is projected along the predicted flight track one half hour into the future. The *flight envelope* is a box that encloses all waypoints from its present position to its destination, including requirements for lateral separation. That flight envelope is compared to the flight envelope of every other active aircraft in the simulation. If two flight envelopes overlap, it is possible that pair of aircraft may come into conflict at some time, and the pair is recorded in a candidate pairs list for more detailed investigation. During each cycle the list of such aircraft pairs is examined by comparing the recorded flight paths of the ghost aircraft to see if they actually come into conflict.

At the end of each interval, the simulation updates the current flight envelope for each aircraft to reflect its new current position. As a flight progresses towards its destination, its flight envelope will shrink, reducing the possibil-

ity of conflict with another aircraft. Existing aircraft pairs whose flight envelopes no longer intersect are removed from the candidate pairs list, as they will not approach each other closely enough to ever be in conflict. This reduces the amount of conflict checking computation to be done.

In effect, the process just described acts as a filter to reduce the number of computationally expensive conflict checks that need to be performed. The calculations that determine flight envelope intersection occupy a substantial part of the simulation run time. Several attempts were made to reduce this time, including an attempt in which the bounding boxes were precomputed at each leg of the flight plan and compared to current aircraft positions to determine potential conflicts. These earlier optimizations had only a minor effect on reducing run time.

4 QUAD TREE ALGORITHM

The quad tree software library was designed with two objectives to decrease run time. The first is to provide a more efficient geographical filter to further reduce the number of candidate aircraft pairs that need detailed examination. The second objective is to provide a mechanism to perform conflict detection on multiple aircraft pairs in parallel.

The quad tree is a hierarchical structure that provides spatial decomposition of data such as points, lines, or polygons. In this case, aircraft positions are treated as points in three-dimensional space. The quad tree allows us to recursively divide that space into smaller regions. In this manner, aircraft separated by a significant distance are removed from conflict comparison in a less computationally intensive manner. The basic quad tree algorithm defines a rectangular region that encompasses the entire region in which the simulation takes place. That area then can be divided into four equal quadrants. This process can recurse indefinitely for each quadrant. An extension of the quad tree concept, oct trees, can take into account all three dimensions.

This technique is most useful if we limit how far this space is divided. We would like to limit the recursive process so that we obtain the smallest area where it is likely that all the aircraft contained within are in conflict with one another. If no node's x or y dimension falls below the smallest minimum lateral separation distance the that the analyst has specified for the entire simulation, then all aircraft in that area will be in conflict, unless they have sufficient vertical separation or are located in the extreme opposite corners of the region.

It is possible, however, that two aircraft may be quite close to one another and yet be placed into two adjacent, but separate nodes in the quad tree. In order to account for this possibility, we introduce the use of exterior regions for each node. Until now, no leaf nodes overlapped one another in space, and the interior regions described so far will continue to posses this trait. Exterior regions are areas that are within the largest minimum lateral separation that the analyst has specified for this scenario from the borders of the interior region. Each leaf node possess two lists of aircraft – one for aircraft present in the interior region corresponding to that node, and one for aircraft present in the exterior region for that node. Note that a given aircraft can only exist on one interior region, but can be present in multiple exterior regions.

The root node encompasses the entire space in which it is possible that an aircraft will occupy in a given scenario. No aircraft should ever be added into the exterior list of the root node; if this occurs, then the size of the root node is not sufficiently large and should be extended to include the offending object. An object is inserted into the quad tree by passing it to the insert method of the root node. The root node first checks to determine whether the object falls within its extended boundary, and if so, it will pass the object on to its four children. If no children exist, and the division of this node would not result in a node smaller than the minimum size established, then the children of the node will be created. If no children exist, and the boundary conditions for the quad tree are met, then that node is also a leaf node of the tree; otherwise the dividing process will repeat until the boundary conditions are met. Note that only parts of the tree that contain aircraft are split in this manner. The efficiency of the quad tree comes from the fact that large areas without aircraft in them can be quickly skipped, while areas that contain aircraft are small and thus remove a vast majority of aircraft from conflict consideration. If another aircraft is in the same node, then it is highly likely that it is close enough to be in conflict.

If the node is a leaf node, and the object falls within its inner rectangle, then the object is added to its interior list. If the object is not contained in the inner rectangle, then it is added to the exterior list. No aircraft is ever passed to a node if it does not fall within the area defined by its exterior node.

Typically, every six time steps a conflict detection cycle occurs. At this time, a new quad tree is created consisting of an empty root node with no children. The list of all active aircraft is traversed, and all aircraft actually in the air are inserted into the quad tree. Then the tree is traversed to generate candidate aircraft pairs for conflict detection. Each leaf node traversed will generate a candidate aircraft pair by comparing each aircraft in its interior list with all the aircraft subsequent to it on that list. This prevents each aircraft pair from being nominated twice for checking, as an aircraft can only exist in one interior list in the entire tree. Then each aircraft on the interior list is paired with the aircraft contained in the exterior list, and nominated for conflict detection. In order to prevent duplicate checking in this case, each aircraft can be given a unique number upon insertion into the quad tree. Only if the first aircraft's number is greater than the second one's should the pair be nominated for conflict detection. If they are not, then when the node containing the second aircraft

on its the interior list is encountered, the pair will be nominated. Each unique pair of aircraft so identified are the parameters for the existing conflict detection algorithm.

As potential conflict pairs are nominated above, they are placed in a statically sized circular queue to avoid the overhead of repeated allocation and deallocation of a dynamic list. A specified number of worker threads are created during simulation initialization, usually one per processor, and then block until work (the aircraft parameters generated above) arrive in the queue. When work is inserted into the queue, the threads are signaled that the queue is no longer empty. The threads then wake and take the next aircraft pair off the queue and invoke the existing conflict detection function. The work queue is locked in order to prevent race conditions in obtaining work, and insures that no two threads process the same work unit. The main thread of execution continues in parallel with this activity until all work units (aircraft pairs) have been generated from the traversal of the quad tree. At this point, the main thread of execution blocks until all work in the queue has been processed. Should the circular work queue fill up, then the main thread will block until the worker threads have opened up space in the circular queue by processing some work units. The use of a queue as an intermediary between the traversal of the quad tree and the invocation of the conflict detection allows a form of load balancing between the threads by preventing worker threads from becoming idle while work still is in the queue.

If the user only enables conflict detection, then the current position of the aircraft is used to determine where the aircraft will be inserted into the quad tree. The simulation discussed here also offers the user a chance to perform automatic conflict resolution. In that case, the position of the ghost aircraft previously mentioned is used as the position of an aircraft in the quad tree in order to implement look-ahead capability to the simulation. This provides the simulation with the opportunity to detect conflicts early enough to invoke automatic conflict avoidance.

5 QUAD TREE PERFORMANCE

Two test scenarios were selected for use with this project. The first scenario, referred to here as the one-fifth scenario, contains approximately 7,000 flights lasting for one-fifth of a day. The second scenario, or whole day scenario, contains approximately 34,000 flights and models an entire day of traffic. Both scenarios are airspace-intensive and are a good stress test for the simulation.

The work described here was performed on a four processor, 550 MHz Pentium III Xeon computer using version 2.6 of Solaris for Intel. Profiling the code revealed that the two greatest consumers of time in these scenarios were conflict detection between aircraft and movement of those aircraft. When conflict detection is enabled in the simulation, it can consume almost 50% of the simulation time in large simulations. Updates to the position and state of the aircraft in the simulation consumes the majority of the remaining time.

Figures 1 and 2 show the performance of the algorithm for the one-fifth and whole day scenario. In both figures, the upper dashed line represents the performance of the simulation with the original conflict detection algorithm enabled. The lower dashed line represents the simulation performance with conflict detection disabled. The difference between the two lines is the actual processing time the simulation consumes while performing conflict detection; this time does not vary against the *x* axis because it reflects the performance of the original sequential simulation.

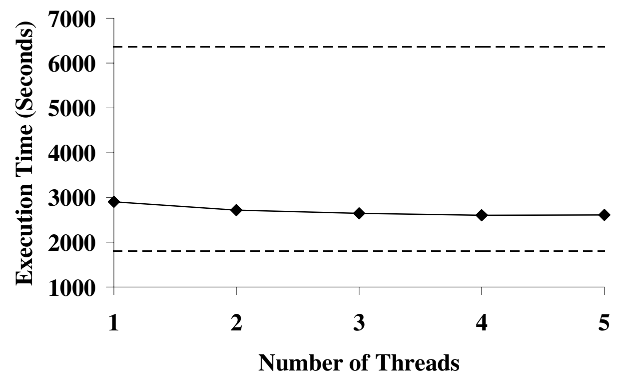

Figure 1: Performance for the one-fifth scenario. The upper dashed line is the run time of the baseline case with conflict detection enabled; the bottom dashed line is the baseline with conflict detection disabled.

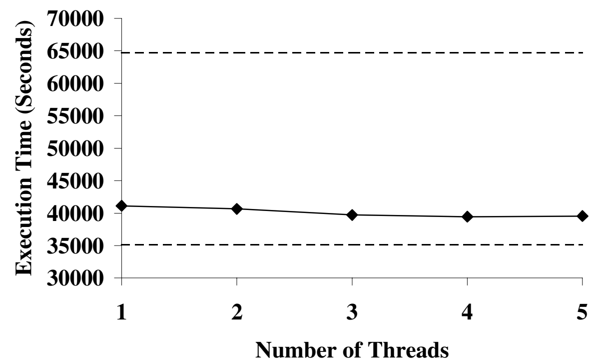

Figure 2: Performance for the whole day scenario. The upper dashed line is the run time of the baseline case with conflict detection enabled; the bottom dashed line is the baseline with conflict detection disabled.

The solid line in both figures represents the measured processing time of the simulation when utilizing the quad tree algorithm. The *x* axis depicts the number of worker threads available to process the conflict pairs. For the one-fifth scenario, the single threaded quad tree algorithm reduces the run time to 46% of the original run time (a

speedup factor of 2.8), and the whole day scenario reduces run time to 64% of the original (a speedup factor of 1.56). The time savings reflects the more efficient filtering provided by the quad tree algorithm as compared to the original envelope based technique. Because of the improved filtering, time was saved because fewer non-conflicting aircraft pairs were passed to the conflict detection code as possible conflicts.

The improvement due to parallel processing of conflict pairs seems less impressive in figures 1 and 2 in part because of the large vertical scale. Figures 3 and 4 better illustrate the parallel speedup obtained. In these figures, run time as reduced via parallelization is displayed as a percentage of the theoretical maximum speedup possible. The results show that we are obtaining up to 30% of the maximum possible speedup available to us. It is encouraging that the larger scenario has slightly greater parallel efficiency (27.8% versus 26.9%), indicating that this algorithm scales well and the overhead for this technique does not markedly increase relative to the problem set size, even in a simulation that was not designed from the ground up with parallel processing in mind.

Overall, we see a speedup factor of 2.4 for the one-fifth scenario and 1.6 for the whole-day scenario. Comparing our total run time reduction to the maximum we could have achieved, the one-fifth scenario achieved 75% of the maximum speedup while the whole day scenario achieved nearly 80% of the maximum.

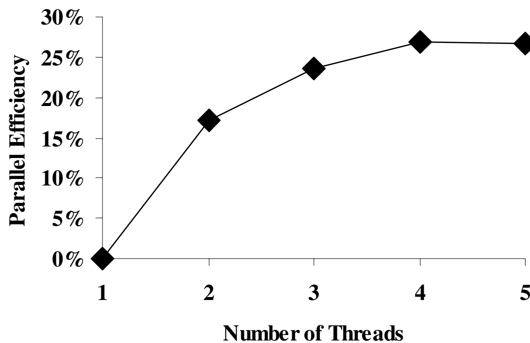

Figure 3: Parallel Efficiency for the One-Fifth Scenario

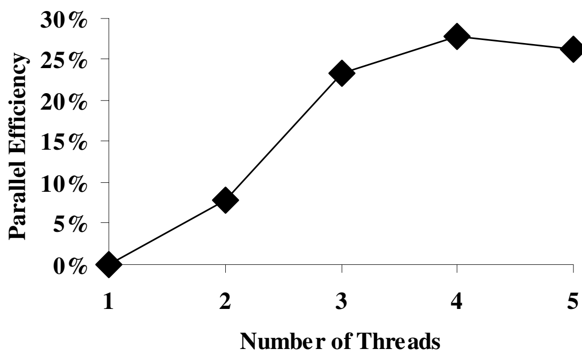

Figure 4: Parallel Efficiency for the Whole Day Scenario

6 AIRCRAFT MOVEMENT IN THE SIMULATION

In the context of the simulation, aircraft movement is the calculation of an updated aircraft position and state as a function of time. The simulation maintains a list of all aircraft that are active in the simulation at a given instant. This list is traversed and each aircraft is processed one at a time. Each aircraft is not independent of the others, however. The simulation takes into account the location of aircraft around the current one, expected congestion in the airspace and airports in its flight plan, as well as other factors, when performing aircraft movement.

Unfortunately, much of the information described above is shared among all the aircraft in the simulation. Updating an aircraft position necessarily updates the information about the scenario environment, so that the next plane to be updated reflects the most recent information. Race conditions can arise when a single data item, not directly related to the aircraft itself, is needed to process multiple aircraft. Only when all aircraft have been updated, the simulation clock can advance to the next time step.

7 MOVEMENT PARALLELIZATION

The goal of this project phase is to speed up the simulation's execution time by performing the processing of these aircraft in parallel for a given time step. The approach taken parallelizes the calls that perform the movement and state updates for each aircraft. This was accomplished by extending the work queue described above to record requests to update each aircraft. Then multiple worker threads would take the parameters from this queue and perform the updates in parallel.

Unfortunately, the simulation architecture does not permit updating each aircraft independently. The simulation must take into account the environment in which the aircraft is flying. An example might be an aircraft that needs to slow down in order to avoid a congested sector ahead. Likewise, in order for the model to accurately simulate the approach to an airport, that airport needs an updated prediction of the time in which aircraft will reach it, in order to more efficiently schedule all incoming traffic. Since this data is used by all aircraft approaching a given airport, any parallelization will require simultaneous access and updates to this shared data. To mitigate these race conditions, a means to regulate this interaction needed to be devised, however care was exercised so as not to add excessive overhead when implementing this protection. Every airport has a lock added to protect airport data, including information about all aircraft that are on the ground at each airport. Locks were also needed to protect

memory allocation, the list library, fuel burn and cost accumulations, sectors, and waypoints. Some global variables, such as the time step, that were temporarily changed during an aircraft update, were "deglobalized" so that each thread had its own copy while executing.

8 MOVEMENT PERFORMANCE

Many movement cycles only involved updating one or fewer aircraft. These cycles present no opportunity for parallel processing, since synchronization must occur at the end of each movement cycle and that prevents us from processing the next aircraft in parallel with the current one. However, measurements made by inserting timing code utilizing the Solaris high-resolution clock showed that most of the time spent in the movement cycle was still available for parallelization.

The initial implementation of the thread pool parallelization spent a large time waiting to acquire the airport lock. Before an aircraft can be processed, the lock for the airport that may be affected must be obtained in order to prevent corruption of this data. However, not only did the large delays encountered while waiting to obtain locks take time that could be spent processing aircraft in parallel, waiting for these locks serialized the processing and robbed the simulation of the benefit of parallelism.

In order to prevent this loss of parallelism, we assigned the processing of all aircraft that needed a particular airport lock to the same thread. For the vast majority of its flight, an aircraft only needs to access data at its arrival airport. Before it takes off, it predominately uses data from its departure airport. However we could not remove the airport lock at this time even though predominately one thread would be handling aircraft needing the data at a given airport. At several points during a flight, such as when it lifts off, an aircraft needs to remove itself from the departure airport data structures and update its information at the arrival airport.

Grouping processing involving an specific airport to a particular thread greatly reduced the waiting for the airport lock, in one typical case the time each thread spent waiting to acquire the airport lock dropped from an average of 294 seconds per thread to 52 seconds per thread. However, a few airports have a disproportionate amount of activity at them (such as JFK Airport in New York), and this unbalances the distribution of work units. Certain threads must handle significantly greater amounts of work, and some threads are left idle while those threads finish handling their flights. This effect can be mitigated, but not eliminated, by insuring that the busiest airports are distributed across the threads and that one thread is not processing most of the busiest airports. The particular airports that require such attention will differ from scenario to scenario. We took no specific steps to insure that one thread would not be excessively unbalanced for the result presented here. An analysis of the execution of these scenarios revealed, however, that the load was already distributed fairly evenly.

9 CONCLUSION

We have shown that it is possible to speed up an existing sequential simulation via a combination of more efficient algorithms as well as parallel processing techniques. We have also demonstrated that this can be accomplished with minimal changes to the underlying simulation architecture. The quad tree algorithm provides a means to more efficiently prevent unnecessary conflict checking, and provides a basis for further speed gains through the use of parallel processing.

The key to parallel speedup is to call the sequential function in separate threads, provided that the work that it does is independent of the work being done by other threads. In the simulation discussed here, it was important to identify functions that could be geographically separated. This allowed parallel evaluation of the operations, while minimizing the amount of locking that was needed to ensure a correct result. Improper or absent locking can lead to system instabilities and crashes. Excessive locking leads to serialization of the system and little, if any, reduction in execution time. Striking the proper balance can result in notable speed increases.

ACKNOWLEDGMENTS

We would like to thank Dr. Paul Wang for his assistance in performing some of the performance measurements. His help is greatly appreciated. We would like to thank The MITRE Corporation Center for Advanced Aviation Systems Development for sponsoring this work. The contents of this material reflect the views of the authors. Neither the Federal Aviation Administration nor the Department of Defense makes any warranty or guarantee or promise, express or implied, concerning the content or accuracy of the views expressed herein.

REFERENCES

Bajaj, L., R. Bagrodia, and R. Meyer. 1999. Case Study: Parallelizing a Sequential Simulation Model. *13th Workshop on Parallel and Distributed Simulation*, 29-36. IEEE Computer Society Press.

Basch, J., L. J. Guibas, and L. Zhang. 1997. Proximity Problems on Moving Points. *Computational Geometry 97*, 344-351. ACM

Boukerche, A., A. Roy, and N. Thomas. 2000. Dynamic Grid-Based Multicast Group Assignment in Data Distribution Management. In *Proceedings of the 4th International Workshop in Distributed Simulation and Real-Time Applications*, 47-54. IEEE Computer Society Press.

Hamada, K. and Y. Hori. 1996. Octree-Based Approach to Real-time Collision-free Path Planning for Robot Manipulator. IEEE.

Kitamura, Y., H. Takemura, N. Ahuja, and F. Kishino. 1994. Efficient Collision Detection Among Objects in Arbitrary Motion Using Multiple Shape Representations. IEEE.

Nichols, B., D. Buttlar, and J. P. Farrell. 1996. Pthreads Programming, 98-107. Sebastopol, CA: O'Reilly & Associates

Nicol, D. M. and P. Heidelberger. 1995. On Extending Parallelism to Serial Simulators. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, 60-67. IEEE Computer Society Press.

Nicol, D., P Heidelberger. 1996. On Extending More Parallelism to Serial Simulators. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, 202-205. IEEE Computer Society Press.

Omri, M. 2000. Routing in Quad Tree-Hypercube Networks. *2000 ACM Symposium on Applied Computing, volume 2*, 677-681.

Samet H. 1990. Applications of Spatial Data Structures. Addison-Wesley.

Srisawat, J. and N. A. Alexandridis. 2000. A New 'Quad-Tree Based' Sub-System Allocation Technique for Mesh-connected Parallel Machines. *2nd ACM International Conference on Multimedia*, 279-286. Association of Computing Machinery.

Steinman J. S. and F. Wieland. 1994. Parallel Proximity Detection and the Distribution List Algorithm. *1994 Workshop on Parallel and Distributed Simulation*, 3-11, IEEE Computer Society Press.

Steinman, J. S., T. Tran, J. Burckhardt, and J. Brutocao. 1999. Logically Correct Data Distribution Management in SPEEDES. *1999 Fall SIW Conference,* Paper 99F-SIW-067.

Tan, G., Y. Zhang and R. Ayani. 2000. A Hybrid Approach to Data Distribution Management. *4th IEEE International Workshop in Distributed Simulation and Real-Time Applications*, 55-61. IEEE Computer Society Press.

Tzafestas, C. and P. Coiffet. 1996. Real-Time Collision Detection using Spherical Octrees: Virtual Reality Application. *5th IEEE International Workshop on Robot and Human Communication*, IEEE.

Vink, A., and S. Kauppinen. 1997. Medium Term Conflict Detection in EATCHIP Phase III. *16th AIAA/IEEE Digital Avionics Systems Conference*.

AUTHOR BIOGRAPHIES

DAVID CARNES is a Senior Software Systems Engineer at The MITRE Corporation. He joined MITRE after two years building flight simulators for Lockheed Martin. He has a B.S. in Computer Science from Towson State University (1997). His is a member of the ACM, and his interests include simulation and high performance computing, including distributed and parallel computing. His email address is <dcarnes@mitre.org>.

FREDERICK WIELAND is an employee of The MITRE Corporation. He holds a BS in Astronomy from Caltech, an MS in Information Systems from The Claremont Graduate School, and a PhD in Information Technology/Operations Research from George Mason University. His research interest is in modeling and simulation techniques, and has published in the parallel simulation community, in physics-based simulation, in military modeling communities, and in aviation modeling and simulation. His email address is <fwieland@mitre.org>.