Proceedings of the 2003 Winter Simulation Conference
S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, eds.

TOTAL AIRPORT AND AIRSPACE MODEL (TAAM) PARALLELIZATION COMBINING SEQUENTIAL AND PARALLEL ALGORITHMS FOR PERFORMANCE ENHANCEMENT

Neera Sood
Frederick Wieland

Center for Advanced Aviation System Development
The MITRE Corporation
7515 Colshire Drive
McLean, VA 22102-7508, U.S.A.

ABSTRACT

This paper describes how to achieve a desired speedup by careful selection of appropriate algorithms for parallelization. Our target simulation is the Total Airport and Airspace Model (TAAM), a worldwide standard for aviation analysis. TAAM is designed as a sequential program, and we have increased its speed by incorporating multi-threaded algorithms with minimal changes to the underlying simulation architecture. Our method was to identify algorithms that are bottlenecks in the computation and that can be executed concurrently, producing a hybrid sequential and parallel simulation. Our results show a performance gain that varied between 14% and 33%.

1 INTRODUCTION

TAAM is a fast-time gate-to-gate simulator of airspace and airports. It is a worldwide standard for air traffic simulation, yet its use is limited because of its long run time. TAAM is designed as a sequential program. We have incorporated parallel processing by adding multi-threading to increase its speed. Converting sequential algorithms to parallel algorithms is difficult. One needs to take into consideration processor architecture, software and data integrity. Algorithms which are suitable for single-processor computers are not always appropriate for parallel architectures. Through multi-threading, we parallelized and integrated two sequential algorithms in TAAM, the conflict detection algorithm and the aircraft navigation/movement algorithm. Parallelizing these two parts of the TAAM source code proved beneficial, resulting in a 14% to 33% improvement in speed over the baseline implementation.

As mentioned above, combining various parallel algorithms is a difficult task. In TAAM, the difficulty was caused by the fact that each algorithm ultimately changes the data structures used by flights in TAAM. On the one hand, the aim is to achieve as much independent, parallel computation as possible in order to maximize the speed at which computations occur. On the other hand, we must ensure that the parallel computations maintain data integrity and do not write over each other's data, which argues for serializing the computations. This trade-off between parallelism for speedup reasons vs. serialization required for verification purposes produces a hybrid system—neither entirely parallel nor entirely serial—whose performance justifies these tradeoffs (Wieland, F., D. Carnes, P. Wang, 2001).

The approach taken and the motivation behind this work are as follows. In many TAAM scenarios, conflict detection comprises almost 50% of the total run time. It is the slowest running algorithm. The second source of TAAM bottlenecks involves the aircraft navigation algorithm. We have incorporated parallel processing in both algorithms to increase its speed. The first step in creating a parallel program is ensuring that it has been properly parallelized. This means:

- Enough of the program has been parallelized to allow the program to attain the desired speedup.
- The workload is distributed evenly among the central processing units (CPUs).

The integration, at first, posed some challenges and resulted in many deadlock situations. However, these problems were finally resolved and some good performance results were achieved as a result of this study.

2 ALGORITHM SELECTION CRITERIA

When selecting a particular function, code section or algorithm for parallelization in a sequential program, one must be able to determine the following:

- Does it contain any inherent parallelism that can be exploited?
- Is the program amenable to either data or functional parallelism or both?

- Can the parts/algorithms identified for parallelization be executed concurrently?

Movement and conflict detection processing in TAAM are both time consuming. Therefore, they make good candidates for parallelization. Both algorithms use data parallelism and follow the Single Program Multiple Data (SPMD) paradigm (Quinn 1994).

The Parallel Conflict Detection algorithm was designed with two objectives in mind. As the quad tree data structure is naturally suited for parallelization, the first objective was to provide an efficient geographical filter to minimize the number of candidate aircraft pairs that require further checking. The second objective was to provide a mechanism to perform conflict detection on multiple aircraft pairs in parallel.

The Movement algorithm was designed to parallelize the processing of ground-based and airborne aircraft, which comprises most of the work in the *movement* function which is the second main source of TAAM simulation.

3 THE SEQUENTIAL CONFLICT DETECTION ALGORITHM

Conflict Detection processing in TAAM entails the processing of the current flight envelope for each aircraft. This design consists of a bounding box containing all the waypoints in an aircraft's flight plan. Two flights are candidates for conflict detection if their bounding boxes overlap. This algorithm is quite inefficient as it evaluates more conflict pairs than its quad tree equivalent. For example, let us consider the case of two flight plans of a pair of aircraft, one from Seattle to New York and the other from San Francisco to Washington D.C. If the bounding boxes of these two aircraft intersect with each other, the Sequential Conflict Detection algorithm will evaluate the two aircraft even though they may not be in conflict.

4 PARALLEL CONFLICT DETECTION ALGORITHM DESIGN

The Parallel Conflict Detection algorithm is based on the principle of recursive decomposition of a rectangular space, based on an array data structure. The number of times that the decomposition process is applied can either be fixed or is governed by the properties of the input data. In parallel TAAM, the basic parallel strategy is based on the SPMD model. The simulated world is divided into different regions of unequal geographic extent but roughly equal aircraft density. Conflict detection is performed in each region, simultaneously (in parallel), using a multiprocessor computer system. In the tree hierarchical structure, the root node corresponds to the entire bounded space, a rectangle. Each child of a node represents a quadrant (labeled in order NW, NE, SW, SE) of the region represented by that node. A boundary area, equal to the size of the maximum conflict distance, is created around the edge of the parent node. The conflict distance is the largest distance in the simulation (since it can vary by sector) that two planes can be from one another and still be in conflict with one another. The reason for using the quad tree is the need to reduce the amount of space necessary to store data through the use of aggregation of homogeneous blocks (Samet 1990). Each child node has an external boundary region as well as an internal boundary region. This configuration leads to the reduction of the execution time of a number of operations. Conflict detection is performed only on objects near enough to the space contained by node boundaries to require it. For this reason this algorithm is able to overcome the problem of evaluating aircraft that are not in conflict. For more details about the Sequential Conflict Detection and the Parallel Conflict Detection algorithms, refer to (Wieland, F., D. Carnes, G. Schultz, 2001).

5 MOVEMENT PARALLELIZATION

Movement in parallel TAAM implies high volumes of air traffic movements and aircraft movement control strategies. A movement control strategy defines the logic used in controlling the aircraft. Each strategy or combination can have different effects on aircraft movement and can therefore be used to resolve different situations. Such strategies include organizing taxi paths for aircraft movement between gates and runways, aircraft navigation (ground or airborne), scheduling dynamic sectors (changing and reassigning of sectors), setting in trail separation delays, recalculating Estimated Time of Arrival (ETA) for aircraft and adjusting flight plans. These comprise the main operations of TAAM and are one of the sources of TAAM bottlenecks. The *movement* procedure within the TAAM source code is scheduled to run once per time step. Time steps are the speed at which the simulation runs and are user-settable. Most often they vary between one and six simulated seconds.

6 MOVEMENT ALGORITHM DESIGN

The Parallel Movement algorithm in parallel Taam follows the SPMD parallel model, by dividing the number of airports (and hence the number of aircraft assigned to a particular airport) among the threads. Parallelization is achieved by multithreading and queuing the calls to a function/method called *proc_acft(),* which handles the bulk of the work during movement (Wieland, F., D. Carnes, 2002). Mutual exclusion is achieved by assigning different airports to different threads, through the use of circular queues and by locking the list data structure either by departure airport for flights on the ground or by arrival airport for airborne flights. The idea is that each thread will process the data/aircraft from its airport's circular queue. This type of processing was achieved by implementing

airport-based threading. A given airport is assigned to a thread, such that all flights requiring a lock on that airport are always assigned to the same thread. Different threads acquire locks on different airports, so that parallelism is maintained. All flights requiring a lock on a given airport are serially processed by the same thread, but flights requiring locks on a different airport may be processed by a different thread, in parallel (Wieland, F., D. Carnes, 2002).

The strategy of multithreading and queuing the call to the function/method *proc_acft()* is similar to the one used for conflict detection in the quad tree parallel implementation.

7 THREAD DESIGN CONSIDERATIONS

We used The Portable Operating System Interface for UNIX (POSIX) threads for our multi-threading parallelization design. There are different parallel programming paradigms and no single scheme fits all designs. We considered two parallel programming paradigms, Single Program Multiple Data (SPMD) and Multiple Instruction Multiple Data (MIMD) (Quinn 1994) for the two algorithms and selected the SPMD model.

Although the basic thread design used for Movement Parallelization is the same as that of the Conflict Detection Parallelization because both are SPMD and exploit data parallelism, there is a difference between the two designs. For the Movement Parallelization, the thread design uses thread-specific data in order to manage the threads and their address spaces better. Unlike processes, all threads within a process share the same address space. This means that variables may be read and written by all threads within the process. Such an action can and did lead to race conditions. Using circular queues together with threads helped avoid deadlock conditions between threads when accessing shared memory areas/global variables. In order to enable each thread to have its own value for a private variable and locate it, it was necessary to use the thread-specific data storage functionality provided by POSIX threads. Thread-specific data allows each thread to have a separate copy of a variable. This data structure is similar to an array of thread-specific data values, which is indexed by a common "key" value. A common key is created for all threads in the same process, but each thread can associate its own independent value with that shared key.

8 INTEGRATION OF CONFLICT DETECTION PARALLELIZATION WITH MOVEMENT PARALLELIZATION

The integration of Conflict Detection Parallelization with the Movement Parallelization first resulted in race conditions and caused the application to malfunction. The malfunctioning was caused by the call, via the parallel Conflict Detection callback function, to a function called *time_step()*, which is part of the thread-specific data for the Movement threads. This call resulted in a deadlock situation. The callback function, *proximity()* executes in a different process and is not synchronized with the Movement worker threads. We considered three schemes to avoid this deadlock and finally selected the third scheme.

In order to make the application thread safe, we first considered incorporating the Conflict Detection worker threads into the thread pool which is created for the Movement Parallelization, thus following the MIMD programming paradigm. According to this paradigm, each thread can execute a separate stream of instructions on its own data (Quinn 1994). We thought that this model would enable better thread management. After conducting a few experiments, this idea was soon abandoned because of load balancing and synchronization overhead. We also realized that it would not let us achieve the goal of the concurrent execution of the two parallel algorithms in two different processes. The two parallelizations are currently set up so that the Conflict Detection worker threads run, in a different process, in parallel with the Movement worker threads, thereby increasing throughput. If we combined them with the Movement worker threads we would lose this gain in speed. In order to avoid this race condition, we then considered modifying the design of the Conflict Detection parallelization by modifying the Conflict Detection logic according to the Movement "threadpool" package through the incorporation of the thread-specific data storage functionality. This design change would eliminate the need for the callback function and the Conflict Detection worker threads would still be able to run in parallel with the Movement worker threads. This experiment, however, led to more deadlock situations, which could only be avoided by a significant change to the overall TAAM design. Therefore it was decided to leave the Conflict Detection parallelization logic in tact. The deadlock problem with the callback function was finally resolved through the simple use of parameters.

9 PERFORMANCE RESULTS ANALYSIS

Tables 1 through 3 and Figures 1 through 4 show the performance results and the speedup, loss in speed and improvement statistics of what we call the 6Ctr scenario, with the number of aircraft equal to 6747. These tests were run on a four processor, 450 MHz Pentium III Xeon SMP with 2 gigabytes of RAM, using the Sun Solaris operating system. The time step was set to six seconds with Conflict Detection on.

These results are for a maximum of four worker threads for the Movement Parallelization and a maximum of four worker threads for the Conflict Detection parallelization. Seven timings are compared:

- Baseline.
- Parallel Conflict Detection (alone).
- Movement (alone).

- Four thread combinations of the Conflict Detection and Movement Parallelizations.

10 EFFICACY OF THE TWO ALGORITHMS

Tables 1 and 2 show the results of the two algorithms, Parallel Movement with Sequential Conflict Detection and Parallel Conflict Detection with Sequential Movement. For the 6Ctr Scenario, with the number of aircraft equal to 6747, the parallel Movement algorithm, when combined with the Sequential Conflict Detection (Table 1), does not show any speedup over the baseline. For the 6Ctr Scenario, with the number of aircraft equal to 6747, the Parallel Conflict Detection algorithm (Table 2), even when not combined with any other parallelization, resulted in a maximum improvement of 18.44% over the baseline, when run with four threads.

Table 1 : Movement Parallelization Speedup

Worker Threads	Execution Time (Seconds)	Speedup Over Baseline
Baseline	4349.6	--
1 Worker Thread	4919.7	0.88
2 Worker Threads	4411.1	0.98
3 Worker Threads	5384.9	0.8
4 Worker Threads	5123.0	0.85

Table 2: Conflict Detection Parallelization

Number of Conflict Detection Worker Threads	Execution Time (Seconds)	Speedup Over Baseline	Improvement
Baseline	4349.6	--	--
1	3836.4	1.13	11.79%
2	3750.3	1.16	13.77%
3	3753.5	1.15	13.7%
4	3547.1	1.22	18.44%

Combining the two parallel algorithms, Movement and Conflict Detection, lead to the best performance results. Figures 1, 2 and Table 3 show that for the 6Ctr scenario, the Conflict Detection and the Movement Parallelization, combined, show much more improvement over the previous results, with a maximum improvement of almost 33% over the baseline.

11 MOVEMENT (ALONE) AND PARALLEL CONFLICT DETECTION WITH MOVEMENT - DISCREPANCY ANALYSIS

Our results in Figure 1 indicate a large discrepancy between the execution times of the Movement Parallelization alone (Series 2) and the Movement Parallelization com-

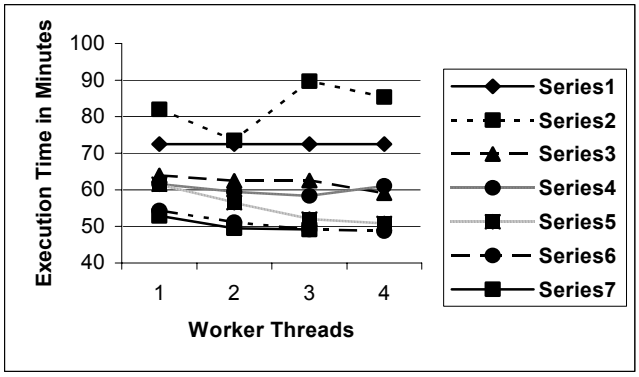

Figure 1: TAAM Parallelization Results - 6Ctr Scenario

Figure 1 Legend:
Series 1 – Baseline
Series 2 – Movement Parallelization with Sequential Conflict Detection
Series 3 – Conflict Detection Parallelization
Series 4 – Movement Parallelization with Conflict Detection Parallelization – Movement Worker Thread = 1
Series 5 – Movement Parallelization with Conflict Detection Parallelization – Movement Worker Threads = 2
Series 6 – Movement Parallelization with Conflict Detection Parallelization – Movement Worker Threads = 3
Series 7 – Movement Parallelization with Conflict Detection Parallelization – Movement Worker Threads = 4

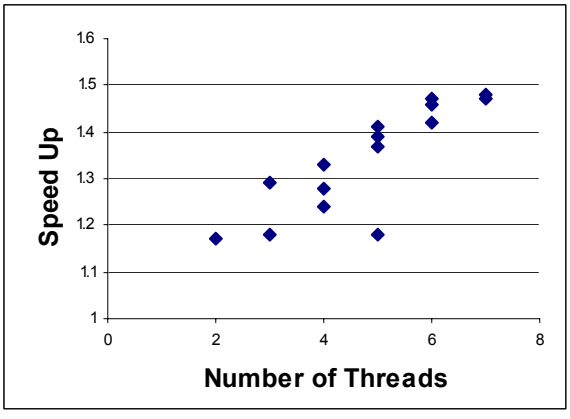

Figure 2: Conflict Detection and Movement Parallelization

bined with Conflict Detection Parallelization runs (Series 4, 5, 6 and 7). When we first noticed this difference in the two execution times, we attributed it to the thread load imbalance in the Movement algorithm.

In the Parallel Conflict Detection algorithm, the data load is properly balanced among the threads. In the Movement algorithm the data load among the threads is not bal-

Table 3 : Conflict Detection and Movement Parallelization

Number of Conflict Detection Worker Threads	Number of Dynamic Worker Threads for Movement	Execution Time (Seconds)	Speedup Over Baseline	Improvement
Baseline	1	4349.6	--	
1	1	3699.4	1.17	14.9%
	2	3568.5	1.29	17.96%
	3	3497.8	1.24	19.58%
	4	3664.5	1.18	15.75%
2	1	3690.5	1.18	15.15%
	2	3393.3	1.28	21.98%
	3	3119.4	1.39	28.28%
	4	3050.9	1.42	29.85%
3	1	3258.9	1.33	25.07%
	2	3068.3	1.41	29.45%
	3	2951.4	1.47	32.14%
	4	2925.2	1.48	32.74%
4	1	3174.6	1.37	27.01%
	2	2972.7	1.46	31.65%
	3	2947.6	1.47	32.33%

anced, implying that some threads are working longer than others, as indicated by Figure 3. We attribute this imbalance to the overhead caused by assigning work to threads and the overhead of locking the queues. In the Parallel Movement algorithm, each call to the function *proc_acft()* takes a different amount of time to execute. Its execution time depends on the current state of a particular aircraft. For example, the processing of a ground-based aircraft, which requires significant computations of taxi paths, takes longer than the processing of an airborne aircraft in an under loaded sector (Wieland, F., D. Carnes, 2002).

On measuring the thread variance between the two runs, Movement Parallelization with the Sequential Conflict Detection algorithm and Movement Parallelization with the Parallel Conflict Detection algorithm, it was discovered that the thread variability between the two runs does not change. Figure 3 indicates that the variability between the thread load imbalance remains almost exactly the same.

While it is true that the overhead incurred by the locking of data structures in the Movement Parallelization minimizes parallelism, resulting in a net slowdown in execution time, it was not the cause of the discrepancy.

We speculate that the large discrepancy is caused by the Sequential Conflict Detection algorithm when Movement Parallelization is run with the Conflict Detection option turned on. The execution time of the Sequential Conflict Detection algorithm is double that of its parallel equivalent.

Figure 4 is a comparison of the Movement Parallelization when run with and without the Sequential Conflict Detection algorithm. This result is a clear indication of the inefficiency of the Sequential Conflict Detection algorithm.

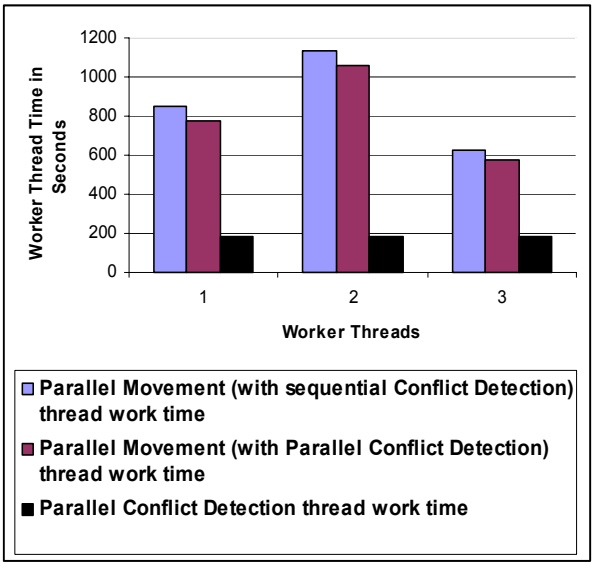

Figure 3: Worker Threads Load Balance Analysis

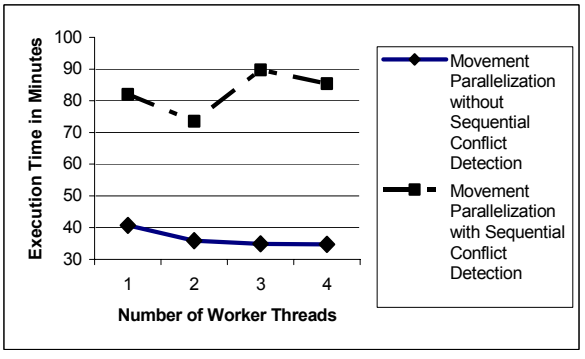

Figure 4: Movement Parallelization

Figure 4 shows that in spite of the thread load imbalance, the Parallel Movement algorithm has a more predictable parallel performance without the Sequential Conflict Detection. Its curve follows the same trend as the curves in Figure 1. These curves indicate that the thread model that gives the best run time is the "Many-to-few" thread model (Butenhof 1997), thus validating the fact that increasing the number of threads leads to an increase in speedup.

Thus the combined effect of load imbalance, concurrency control measures and inefficient sequential algorithms can reduce realized parallelism by a substantial amount.

12 CONCLUSION

We have shown that through careful selection of sequential algorithms, amenable to parallelization and the use of effective parallel algorithms, it is possible to successfully parallelize sequential simulations and achieve significant speedup over the baseline, Figures 1, 2 and Table 3.

Through the use of POSIX threads for multi-threading and parallelism, we were able to take advantage of the pthreads library. This library allocates a varying number of execution resources (CPUs) and dispatches them to the runnable threads. These execution resources are allocated and dispatched entirely in the user process space and do not require the creation of UNIX processes (Butenhof 1997), thus avoiding excessive communication with the kernel. Such a scheme makes multi-threading more efficient, increases speedup and facilitates parallelization.

One may argue that a speedup of 33% is not worth the effort of parallelizing a sequential application. With today's advance in technology and increase in CPU capacity, this speedup may be achieved in less than six months. However, one must bear in mind that the 33% increase is multiplied by whatever hardware advances occur. If hardware doubles the speed of TAAM, then with the parallel software the speed increase will be approximately 2.66 times (because the speed improvements will also have been doubled). Hardware gains do not obviate software gains; they multiply them.

Further work still needs to be performed on TAAM, to eliminate the thread load imbalance in the Movement Parallelization algorithm in order to achieve additional speedup. This will require the implementation of a more efficient locking mechanism. Parallelizing movement in TAAM was, however, difficult. This is because the movement function processes not just aircraft but also sectors, sector bodies, airports and in-trail separation delays, if required. In order to maximize the advantages of our approach it will be necessary to separate the different processing algorithms and identify the ones that can be further parallelized. A faster TAAM will allow airport clients worldwide to exploit its full potential.

ACKNOWLEDGMENTS

We gratefully acknowledge the assistance provided by Curt Holden and David Bodoh of The MITRE Corporation and the help provided by Preston Aviation Solutions. The contents of this material reflect the views of the authors. Neither the Federal Aviation Administration nor the Department of Transportation makes any warranty or guarantee, or promise, expressed or implied, concerning the content or accuracy of the views expressed herein.

REFERENCES

David R. Butenhof, 1997. *Programming with POSIX Threads.* Addison-Wesley.

Wieland, F., D. Carnes, P. Wang, 2001, *Parallelizing Conflict Detection in the Total Airport and Airspace Model (TAAM),* McLean Va: The MITRE Corporation.

Wieland, F., D. Carnes, G. Schultz, 2001, *Using Quad Trees for Parallelizing Conflict Detection in a Sequential Simulation.* PADS 2001 Proceedings.

Wieland, F., D. Carnes, 2002, *Parallelizing Movement in the Total Airport and Airspace Model (TAAM),* McLean Va: The MITRE Corporation.

Michael J. Quinn, 1994. *Parallel Computing Theory and Practice*. 2nd ed. New York: McGraw-Hill, Inc.

Hanen Samet, 1990. *Applications of Spatial Data Structures: Computer Graphics, Image Processing and GIS.* Massachusetts: Addison-Wesley.

AUTHOR BIOGRAPHIES

NEERA SOOD received her Post-Bachelor's Degree in Computer Science from Wayne State University and M.S. in Computer Science from George Mason University and is currently working with Air Traffic Simulation and Modeling. Her email is <nsood@mitre.org>.

FREDERICK WIELAND holds a PhD in information technology/applied probability theory from George Mason University. He is the developer of numerous simulations for the U.S. Department of Defense as well as the Federal Aviation Administration, including CTLS, DPAT, and others, and has done extensive research in the parallelization of large-scale simulations such as TAAM. He has been working in the simulation field for 20 years. He can be contacted by e-mail at <fwieland@mitre.org>.