Cent. Eur. J. Comp. Sci. • 2(2) • 2012 • 118-127
DOI: 10.2478/s13537-012-0012-x

VERSITA

Central European Journal of **Computer Science**

Using parallelization to improve the efficiency of an automated taxi route generation algorithm

Research Article

Abe Hall[1*], Michael Gabilondo[1†], Damian Dechev[1,2‡§]

1 *University of Central Florida*
4000 Central Florida Blvd. Orlando, Florida, 32816, United States

2 *Scalable and Secure Systems R&D Department*
Sandia National Laboratories, Livermore, CA 94551, United States

Received 05 January 2012; accepted 23 May 2012

Abstract: As part of the Federal Aviation Administration's (FAA) Next Generation Air Transportation System (NextGen) concept, surface support tools that generate taxi routes and monitor pilot conformance against those routes have been designed and implemented by Mosaic ATM and tested in simulations conducted by The Mitre Corporation's Center for Advanced Aviation System Development(CAASD). The purpose of these tools is to increase the overall safety of the airport's surface by detecting aircraft movement that is not in conformance with the taxi route assigned to that aircraft. Additionally, the tools aim to increase the overall efficiency of airport operations by ensuring that aircraft taxi in compliance with their assigned routes. One of the keys to providing a reliable conformance monitoring system is to produce reliable taxi routes against which to monitor compliance. The tools provided by Mosaic ATM generate these taxi routes via a set of predefined routes commonly used at an airport. In the simulations conducted by CAASD, it was found that the routes provided were found to be reliable and trustworthy. In addition to the predefined routes, Mosaic ATM provided an ad hoc route capability. This capability uses an algorithm that finds a route based on the taxiways assigned by a user through the ad hoc route tool. However, in the simulations conducted by CAASD, this tool was not used extensively by the users. In this paper, we describe our efforts to verify the correctness of the ad hoc taxi route generation algorithm as well as our efforts to increase the speed of the algorithm by implementing a lock-free parallelized version.

Keywords: surface automation • taxi routing • multi-core processing • lock-free

* *E-mail: abe.hall@knights.ucf.edu*
† *E-mail: mgabilon@knights.ucf.edu*
‡ *E-mail: dechev@eecs.ucf.edu (Corresponding author)*
§ *Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.*

1. Introduction

The main purpose of the automated taxiroute generation tools is to provide a route that can then be used to monitor the conformance of an aircraft's surface movement. The goal of this is to improve the safety of surface operations at an airport. By providing a proper route, and then by monitoring conformance against it, controllers can be alerted as soon as a plane is out of conformance on the surface. This will allow the controller ample time to notice the error and take corrective measures. This will hopefully reduce the likelihood of an on surface collision or a take off accident that could occur otherwise. In tools already developed and provided by Mosaic ATM, their predefined taxi routes were found to be accurate and reliable. The algorithm used by their ad hoc tool was not evaluated [3, 7]. In this paper, we verify the correctness of the ad-hoc tool, we implement a parallelized version of that algorithm that achieves a speed-up over the sequential algorithm, and finally we confirm the accuracy of the routes the parallelized algorithm generates.
For our experiments, we used DFW (Dallas Fort-Worth) as the airport, since the simulations mentioned previously were also conducted for DFW. Thus, we already have a set of predefined routes that we may use to test the correctness of our parallelized implementation. The surface of the airport is modelled as a network of nodes and edges, representing the actual physical taxiways upon which an aircraft can travel. This, in essence, reduces the problem of generating the route to a specialized version of a shortest path problem. It is interesting to note that although DFW is fairly large by airport standards, the network of nodes and edges representing its surface is relatively small by network optimization standards, only 1820 nodes. The size of the network limits the potential for performance gains, although we are still able to gain a significant speed up with our algorithm.
It is also important to note that generating a taxiroute differs from finding the shortest route from the start to the end point in some key aspects. Firstly, in a typical shortest route algorithm, we are concerned only in finding the best cost path from point A to point B. However, when generating a taxiroute, we are generating a path that will be then be translated to a series of taxiways upon which an aircraft will be expected to taxi. Therefore, we must take into consideration the fact that some routes may not be possible because aircraft have a limited turn radius. For example, let us suppose that a plane is currently taxiing north on a taxiway A. The plane's target destination lies south of its current location. In a shortest route algorithm, we might very well find that the quickest route is simply go south down A from the current location and then from there go on to the destination. However, a plane cannot just make a 180 degree turn in the middle of the taxiway. Therefore, the algorithm must take into consideration both the current heading of the aircraft and its turn radius to generate a route that would send the plane along A until it comes to a taxiway it can turn onto, eventually turning the plane south.
Secondly, we must find a route that sticks to the routes assigned by the controller. If a controller assigns a route of A to B to B7 (hereafter denoted as A.B.B7) then the route generated should follow taxiway A to taxiway B to taxiway B7. This might not be the shortest route. Additionally, the flight should not leave taxiway A to go onto another taxiway just because it might have a shorter route to B.

2. Previous work

As described above, various work has already been done in the field of automated taxi route generation and taxi conformance monitoring. A brief description of that work, as well as related multi-core processing work, follows.

2.1. Shortest forward path algorithm

The first algorithm used by Mosaic ATM was a shortest forward path algorithm. It differed from the common Dijkstra's shortest path algorithm in that it used both the current heading of the aircraft and the turn radius of the craft to ensure that the route generated would be one that the plane could follow. This eliminated the possibility of a route requiring an aircraft to make a 180 degree turn on a taxiway. An analysis of the correctness of this algorithm found that at SDF (Standiford Field, Louisville),the route predicted was correct 56% of the time. Here it was discovered that the shortest forward path algorithm was especially ineffectual in cases where two taxiways ran parallel to each other [7].

2.2. Predefined routes and the required taxiway algorithm

In order to increase the accuracy of the taxiroute generation, Mosaic ATM worked with controllers to create a set of commonly used taxi routes at DFW. These routes were then predefined and loaded into the system. The routes were assigned to taxiing aircraft based on the spot and runway assignments for the flight. In simulations conducted at CAASD, it found that the predefined routes were correct in nearly all cases and that the confidence rating given by the controller users in the simulation was also high, ranking at 8.58 out of 10.0. The ad hoc taxi route assignment tool was not used extensively during the simulations and therefore no conclusions about the required taxiway algorithm were offered at that time [7].

In this paper, we implement a parallelized version of the ad hoc taxi route assignment algorithm, hereafter referred to as the sequential required taxiway algorithm. Because no analysis of this algorithm exists, we first prove the correctness of the sequential algorithm. Then we improve the speed of this algorithm so that it can be a legitimate option for use in real time. One of the drawbacks for this algorithm is that it is slow, and that is one of the main reasons that predetermined routes are being used for the real time taxiway assignments in the Mosaic ATM software. However, the disadvantage of using predetermined routes is that it is a time consuming process to develop these routes as the routes themselves are defined by identifying the required vertices for each route. So someone must manually take a given route (given in human readable form, e.g., taxiways A.B.B7) and convert that to a series of required vertices. This also makes the routes dependent upon the adaptation work for the airport that the system is using. If the definitions of the vertices change in the adaptation work, they could potentially affect the predetermined route definitions as well. This means that every time the adaptation work changes, the predetermined routes would need to be reverified.

Our goal is to speed up the sequential required taxiway algorithm (which takes a human readable route as an input) enough that it can become viable option to use in the software system; the predetermined routes can then be defined as a series of taxiways instead of a series of vertices. This would save time in the development of these routes and also prevent any direct dependencies on the adaptation work done for the airport.

2.3. Parallel data structures

In our implementation, we use a Lock Free FIFO queue to hold the route jobs to be executed by the different threads. The queue we implement is described in chapter 10 of [4] . We also use the Lock Free Wait Free Hash Map implementation first described in [1] and downloadable at [2].

2.4. Parallel shortest paths algorithms

During part of its processing, our algorithm must solve instances of the shortest path problem between a set of start nodes and a set of end nodes. In our approach, we use a modified Dijkstra's algorithm for solving the single-node to single-node shortest paths problem, which we run in parallel for all combinations of start nodes in A and end nodes in B. We store the results in a shared map to track the current shortest path/distance for any given node from the start node. In other words, we run parallel copies of a sequential node-to-node shortest paths algorithm to solve the shortest paths problem for a set of start nodes to a set of end nodes.

There has been previous work in the parallelization of the Single Source Shortest Path (SSSP) problem and the All Pairs Shortest Path (APSP) problem [5, 6, 8]. For example, in [6], an algorithm to solve the SSSP for large graphs is proposed using a multi-threaded asynchronous visitor queue; it is similar to Dijkstra's algorithm in that it visits the node with shortest distance from the start node. That algorithm uses multiple "vertex visitor" threads, which may proceed in parallel if there are multiple shortest-paths pathways that can be independently traversed. In [8], a parallel algorithm for solving SSSP for real-road networks was proposed; that algorithm partitions the graph and solves the SSSP problem locally for the partitions, and then the individual solutions are combined. The work [5] provides a survey of parallel algorithms for solving the SSSP problem.

3. Methodology

3.1. Verification of the algorithms

First, we verify the accuracy of the original sequential required taxiway algorithm. Since the predefined routes had previously been found to be accurate, rather than formally prove the algorithm correct, we prove its correctness by comparing the results of the algorithm against the results of the predefined routes algorithm.
Towards this end, we create an input file that assigns each one of the 246 predefined routes supplied by Mosaic ATM to a different flight and then save the node by node routes for each flight. We then use those node by node routes to come up with the human readable form of each route (e.g., A.B.B7). Then, we run the sequential required taxiway algorithm with the human readable routes as input. We compare the output of the sequential required taxiway algorithm with the output of the predefined routes algorithm and find them to match 100%; thus, we conclude that the sequential required taxiway algorithm is correct.
To test the correctness of our lock-free parallelized implementation, we compare the routes it outputs with the correct routes that are outputted by the sequential algorithm. In all cases, the parallel algorithm routes results matched up with the sequential algorithm routes. This provides strong evidence that our algorithm is also correct.

3.2. The sequential required taxiway algorithm

The input to the sequential required taxiway algorithm is the graph G, the start node A, the end node B, and the taxiway constraints $T_1, T_2, ..., T_n$. A taxiway T_i is a subgraph of G with edges labeled T_i. The output is the shortest path that obeys the taxiway constraints $T_1, T_2, ..., T_n$; i.e., the aircraft will begin at A, take taxiway T_1, then T_2, until T_n and finally it will reach B. The algorithm will find the shortest path that obeys these constraints, but this may not not correspond to the shortest path between A and B. When the aircraft "takes taxiway T_i", it means it will try to stay on the particular taxiway T_i as long as possible (i.e., it will not venture off onto other taxiways); this is done by scaling the edge weights of taxiway T_i by some factor. The algorithm proceeds in phases as follows; there are $k+1$ phases if there are k taxiway constraints.
In our example we will use the taxiway constraints A.B.B7. In the first phase, depicted in Figure 1, the algorithm computes the shortest path from the start node to every node in taxiway A. The shortest path algorithm is based on Dijkstra's single source shortest path algorithm. While computing the shortest path during this phase, the weights of the edges in taxiway A are temporarily scaled down by some factor.
At the end of the first phase, the algorithm has computed the shortest path (and corresponding shortest distance) from the start node to each node in A; these values are stored in distance[n] and predecessor[n] mappings, which are used by the shortest path algorithm to keep track of the shortest distance and path found so far from the start node to some node n. Those shortest paths and distances are carried over into each succeeding phase, so that each phase begins with the output of the previous phase.
In the second phase, depicted in Figure 2, the algorithm computes the shortest path from each node in A to each node in B7 while scaling the edge weights of B7 down by some factor. In the third phase, the algorithm computes the shortest path from every node in B to every node in B7. Finally, in the last phase, the algorithm computes the shortest path from each node in B7 to the end node, depicted in Figure 3.

3.3. The parallelized required taxiway algorithm

The parallel required taxiway algorithm works similarly to the sequential algorithm. For example, suppose the input is some start node, some end node, and taxiway constraints A.B.B7. The parallel algorithm begins by computing the shortest path from the start node to every node in A, then from every node in A to every node in B, then from every node in B to every node in B7, and finally from every node in B7 to the end node, all the while scaling down the edge weights of the destination taxiway for the first three phases. Also, the outputted shortest paths and distances of each phase serve as input to the next phase, in the same way as described in the sequential required taxiway algorithm. The difference between the sequential and parallel algorithms lies in how the shortest path is computed between a set of start nodes and a set of end nodes; e.g., how the algorithm computes the shortest path from every node in A to every node in B is different in the two algorithms.

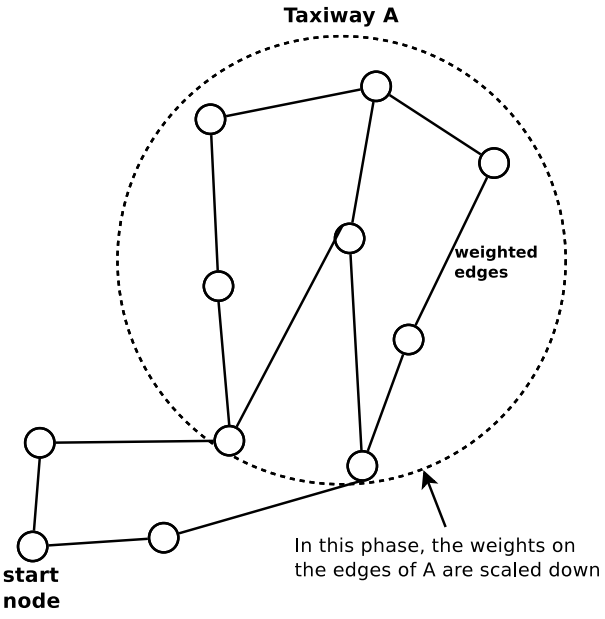

Figure 1. Phase I.

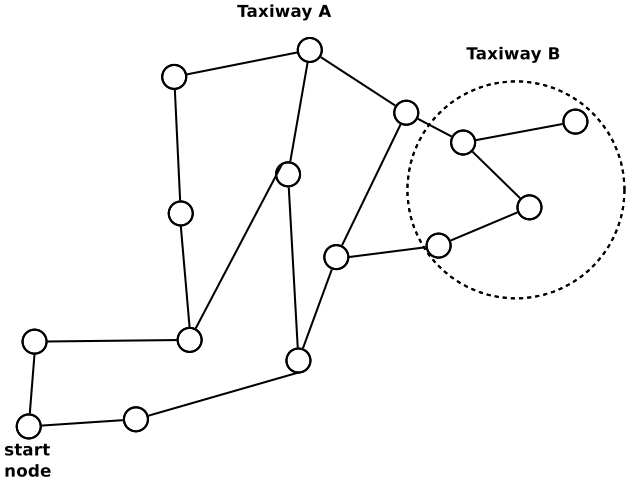

Figure 2. Phase II.

We noted that the sequential algorithm used a modified Dijkstra's single source shortest path algorithm to compute the shortest path between a set of start nodes and a set of end nodes; this was done by running an instance of modified Dijsktra's on each start node. Each instance of modified Dijkstra's gives us the shortest path from that particular start node to each end node. Thus, by running modified Dijkstra's on every start node, we get the shortest path from each start node to each end node.

In our parallel implementation, we do not compute the shortest path from one start node to every end node in one run of modified Dijkstra's; instead, we use a modified node-to-node Dijkstra's. A single run of node-to-node Dijkstra's in our parallel implementation finds the shortest path between one start node and one end node. In practice, this is implemented by modifying Dijkstra's to terminate early once it reaches the target end node. The parallel implementation runs instances of node-to-node Dijkstra's in parallel while computing the shortest path between a set of start nodes and

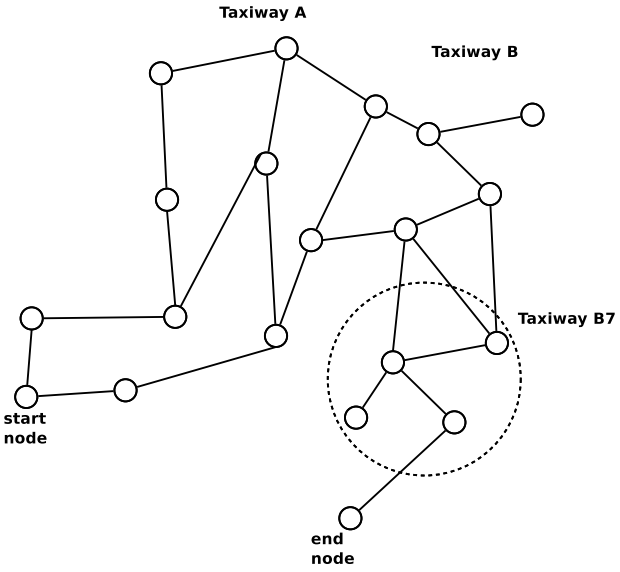

Figure 3. Phase III and Termination.

a set of end nodes; if there are n start nodes and m end nodes, then there may be at most mn instances of node-to-node Dijkstra's running in parallel.

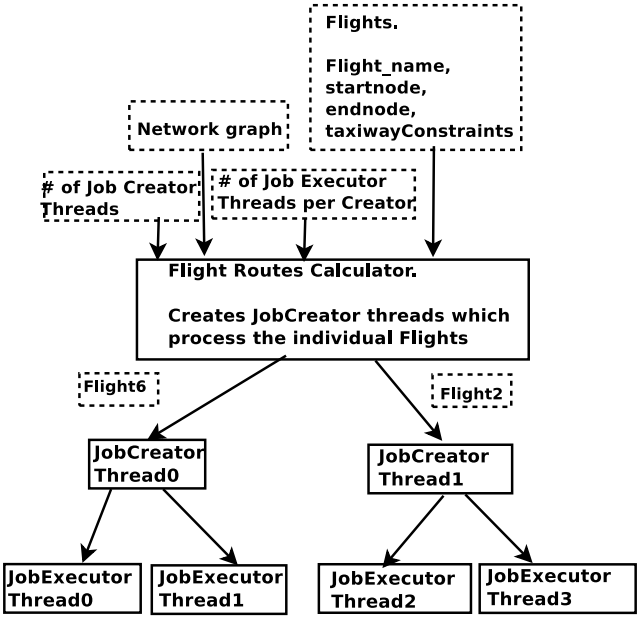

Figure 4. The architecture of the parallel algorithm.

The architecture of our parallel implementation is shown in Figure 4. The main program is represented in the figure by the Flight Routes Calculator. The input to the program is the Flights file, the Network graph file, the number of job creator threads, and the number of job executor threads per creator. The Flights file specifies the inputs to our algorithm; it is a list of (FlightName, startNode, endNode, taxiwayConstraints) tuples. Each Flight corresponds to one run of our required taxiway algorithm.

At startup, the Flight Routes Calculator creates the specified number of job creator threads. The Flight Routes Calculator reads in the Flights and enqueues them in a shared queue. These Flights are dequeued and processed by the job creator threads. A job creator thread is responsible for processing one Flight at a time. At startup, a job creator thread creates the specified number of job executor threads per creator; we will show how the executor threads run node-to-node Dijkstra's for the creator thread.
For a particular input Flight, a job creator thread is first responsible for computing the shortest path from the start node to each node in the first taxiway, with the edge weights of the first taxiway scaled down. If there are n nodes in the first taxiway, then there are n shortest path jobs. Each of these shortest path jobs is enqueued in a shared queue that is maintained by the job creator. The job executors repeatedly dequeue jobs from the shared queue and run node-to-node Dijkstra's on the specified start node and end node. After each run of node-to-node Dijkstra's, a better path may have been found to the end node than was previously found by any other threads; the shortest path and distance to each end node is stored in a shared map maintained by the job executor (implementation by Cliff Click). This shared map serves the purpose of the distance[n] and predecessor[n] described in the sequential required taxiway algorithm. The job executor is finished with the first phase when the shared queue of node-to-node shortest path jobs is empty; it detects this by using a shared counter, which the executor threads increment when they are finished with a job. When the value of the shared counter equals the total number of enqueued shortest path jobs for a particular phase, then the algorithm can move to the next phase.
In the second phase, the job creator is responsible for computing the shortest path from every node in the first taxiway to every node in the second taxiway with the edges of the second taxiway scaled down. If there are l nodes in the first taxiway and k nodes in the second taxiway, then the executor will enqueue lk shortest path jobs, one for each start node, end node pair. It will wait for the jobs to finish, before moving on to subsequent phases. In the last phase, the algorithm computes the shortest path from each node in the last taxiway to the end node.

3.4. Object pooling

We also take into consideration that there are a lot of objects being built during the execution of our parallel algorithm. The concern is that the creation of numerous objects and the subsequent garbage collection that occurs when those objects are no longer referenced will slow down the algorithm. In order to minimize these effects, we implement an object pool for the shortest path job objects that the job creator threads construct. We chose these objects since each creator thread constructs many of them (one for each node-to-node shortest path job), and after each job is completed, it is no longer referenced and is therefore a candidate for garbage collection. After each phase of the algorithm completes, we reuse those shortest path objects. By using an object pool, we avoid constructing as many new objects, as each creator thread always has the pool of objects ready for reuse after the first route.

4. Results

We ran the sequential algorithm and parallel algorithm on two machines, a quad core machine and a dual core machine. The sequential algorithm took 7:38 on the quad core and 6:59 on the dual core. Table 1 shows the parallel and sequential timing results for the quad core, and Table 2 shows the results for the dual core. Increasing the number of Creator threads causes the algorithm to process more Flights in parallel. Increasing the number of executors parallelizes the processing of single Flights.
Note that that with one creator and one executor, the parallel implementations finish in seconds (0:50 and 0:45), while the sequential implementation takes much longer (7:38 and 6:59); this suggests that our parallel implementation, when run sequentially, is much faster than the original sequential implementation. We think that the object pool is probably one reason for this. Additionally, since the sequential algorithm could not be fully decoupled from the Mosaic ATM surface surveillance software, it is probable that some of the slowness of the sequential algorithm can be attributed to the fact that the sequential algorithm has to run within that environment, so some of the performance time is probably attributed to various processes that are running in the software that don't exist in our parallel algorithm implementation. Also, in the sequential algorithm, each shortest path job runs with its own result map. In the last stage of the sequential algorithm, the final route is calculated by looking for the best result in all the maps. In our parallel algorithm, we use a shared map to store best routes so each thread executing a job could store and access the results. We believe that

this helps speed up the algorithm because each thread has access to the results of the other threads, not only at the destination nodes, but also at every other intermediate node that has been previously calculated. Therefore, if another thread has already calculated a best cost at some node on the route, the next thread that comes along knows that cost, and can easily eliminate calculating any route to that node that would result in a higher cost. In other words, because some other thread has already calculated the best route to that intermediate node, the new thread is spared all the work of investigating any other route to that node that wouldn't be optimal.

Even in the scenario where we use only one creator and one executor, using this shared results map shows this kind of performance improvement since the results from every previous job are accessible as the executor evaluates the current job. Unfortunately, because of concerns about the proprietary nature of the Mosaic ATM software system, we were unable to attempt to build our algorithm ro run within the system. Because of this limitation and the differences described above, it is hard to fairly assess the performance gain due to the parallelization by comparing the execution time of the parallel implementation versus the execution time of the sequential implementation. However, we note that the parallelized algorithm does achieve speed-up over the one-executor and one-creator setup when adding more executor and creator threads, which suggests that the parallelization did have an overall positive.

Finally, we note that there is not a huge performance increase by running the algorithm with many executor threads, as compared to the larger performance increase by running the algorithm with many creator threads. In the dual-core results we show that running with 1 creator and multiple executors is about 4-5 seconds slower than the fastest time we were able to achieve by running with multiple creators. In the quad-core performance tests, we see that the difference drops down to 1-2 seconds.

		# Executors			
		1	**2**	**4**	**8**
# Creators	**1**	50	29	21	21
	2	31	21	20	21
	4	22	21	20	21
	8	22	22	21	21
	16	19	20	20	20
Sequential		7:38			

Table 1. The Intel(R) Core(TM) 2 quad CPU Q6600 @2.4 GHz results.

		# Executors			
		1	**2**	**4**	**8**
# Creators	**1**	45	28	28	29
	2	31	27	26	26
	4	27	25	25	25
	8	25	24	25	25
	16	24	24	24	25
Sequential		6:59			

Table 2. The Intel(R) core(Tm) 2 Duo CPU results.

Table 3 shows the results on the quad core machine before calls to the random function in Java were removed. We were using random to determine how long each thread would sleep to give up control to a different thread. But since random blocks, removing it and using a static amount of time vastly improved our performance. It is interesting to note that after removing the call to random, the number of threads we needed decreased. We determined the reason for this to be that the calls to random were causing the threads to sleep for a random value up to 1 full second. However, a job executor took, on average, 8-20 ms to finish a job. This meant that in a second, a single thread might complete only one job, while

having been actively processing data for only 0.8% of the available time. As a result, while using the random function, we were able to continue to see performance gains with an increasingly large number of threads simply because most of the threads were sleeping for most of the time.

		# Executors				
		1	**2**	**4**	**8**	**16**
# Creators	**1**			10:34	9:53	9:24
	2		5:48	5:15	4:53	4:41
	4	3:33	2:53	2:33	2:29	2:26
	8	1:47	1:29	1:19	1:14	1:12
	16	0:55	0:45	0:40	0:42	0:41
	32	0:28	0:25	0:22	0:21	0:21
	64	0:18	0:16	0:15	0:15	0:15
	128	0:14	0:14	0:14	0:14	0:14
Sequential		7:38				

Table 3. The Intel(R) Core(TM) 2 quad CPU Q6600 @2.4 GHz results before calls to random were removed.

5. Conclusions

The goal of this work was to create a parallelized version of the sequential required taxiway algorithm and to gain performance speed-up while maintaining the accuracy of the sequential version; from our results, it appears that we have been successful. The parallel required taxiway algorithm did not produce a single incorrect route, and our final implementation ran much faster than the sequential algorithm. Although we could not do a direct fair comparison of performance because of the implementation differences between the sequential and parallel algorithms, it is apparent that with some of the optimizations we used in our parallel algorithm, the speed of the original Mosaic ATM algorithm can be increased. In addition to the performance gain resulting from the shared results hash map and the object pooling optimizations, our results also show that parallelization improves the performance of this algorithm. We also notice that the algorithm seems to scale poorly as we do not see any significant increase once we get beyond a handful of threads. We attribute this trend, in part, to the small size of the network. We found that the average time for a single shortest path job to complete ranged between 8 and 20 ms depending on the setup used. The more executors we had, the less average time was spent on each job by each executor. However, the more executors we used, the more time was spent on overhead. Our results show that it was not long before the overhead time cancelled out the advantages we were gaining from using more executors. We conclude that there is a possibility to use this algorithm over the predefined routes algorithm and the sequential required taxiway algorithm for conformance monitoring, but that the algorithm does not scale well.

The biggest potential issue that we see with this implementation is that there is no guarantee of progress if work is split at the flight level (i.e. running with multiple flight creator threads). When the algorithm runs with multiple job executor threads, one thread is always guaranteed to make progress. And since the route is calculated in phases, in the worst case, a thread that is constantly interrupted will finish when all the other jobs in the phase have been completed. Because of this, when splitting the work at the shortest job level, we are assured that the algorithm will return the route in a timely manner. However, when splitting the work at the flight level in a live system, requests for flight trajectories can come in continuously. It is, therefore, theoretically possible that work on a particular flight can be interrupted and pushed off indefinitely as new flight trajectory requests come in; therefore, we run the risk of not getting a trajectory for a flight in a timely manner. But as our results show, there is not a huge performance gain to be had by splitting work at the flight level rather than at the shortest path job level. Therefore, if this is a concern, then the algorithm can be configured to only split work at the shortest path job level and most of the benefits we have reported can still be realized but without the risk of having a delay in calculating the flight trajectory.

References

[1] C. Click. A lock-free hash table, In: 2007 JavaOne Conference, 2008
[2] C. Click, Highly scalable Java beta, November 2011, http://sourceforge.net/projects/high-scale-lib/
[3] P. Diffenderfer, C. Morgan, Surface conformance monitoring in the NextGen timeline, In: 9th USA/Europe Air Traffic Management Research and Development Seminar, 2011
[4] M. Herlihy, N. Shavit, The Art of Multiprocessor Programming (Morgan Kaufmann, March 2008), http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0123705916
[5] U. Meyer, Design and Analysis of Sequential and Parallel Single-Source Shortest-Paths Algorithms, PhD thesis, Universitat Saarlandes, Saarbrucken, 2002
[6] R. Pearce, M. Gokhale, N.M. Amato. Multithreaded asynchronous graph traversal for inmemory and semi-external memory, In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC'10, Washington, DC, USA, 2010, IEEE Computer Society, http://dx.doi.org/10.1109/SC.2010.34
[7] E. Stelzer, C. Morgan, K. McGarry, K. Klein, K. Kerns, Human-in-the-loop simulations of surface trajectory-based operations: An evaluation of taxi routing and surface conformance monitoring decision support tool capabilities, In: 9th USA/Europe Air Traffic Management Research and Development Seminar, 2011
[8] Y. Tang, Y. Zhang, H. Chen, A parallel shortest path algorithm based on graph-partitioning and iterative correcting, In: Proceedings of the 2008 10th IEEE International Conference on High Performance Computing and Communications, HPCC '08, Washington, DC, USA, 2008, IEEE Computer Society, http://dx.doi.org/10.1109/HPCC.2008.113