Proceedings of the 2008 Winter Simulation Conference
S. J. Mason, R. R. Hill, L. Mönch, O. Rose, T. Jefferson, J. W. Fowler eds.

DEFERRED VS. IMMEDIATE MODIFICATION OF SIMULATION STATE IN A PARALLEL DISCRETE EVENT SIMULATOR USING THREADED WORKER POOLS

David W. Mutschler

Bldg 2185, Suite 2160-B4
22347 Cedar Point Rd
Naval Air Systems Command (NAVAIR)
Patuxent River, MD 20670, USA

ABSTRACT

The Joint Integrated Mission Model (JIMM) is a real-time legacy battlefield simulator employed in detailed analyses and virtual exercises. To leverage more processors to improve real-time execution, a worker pool of threads optimistically executes events in parallel but avoids cascading rollback by executing only one future event per simulated object. Safeguards for maintenance of simulation state are programmed explicitly and either deferred or immediate modification of state variables could be employed in case of event rollback. In the beginning of the main parallelization effort, deferred modification was used where simulation state is updated only when the event can be completed safely. However, after successful implementation, it was determined to be impractical. Later, all safeguard programming employed immediate modification where original state is restored in case of rollback. This paper discusses these techniques for parallel execution of events in JIMM from initial efforts through later code maintenance.

1 JOINT INTEGRATED MISSION MODEL

The Joint Integrated Mission Model (JIMM) is a general-purpose (Nalepka, Gump, & Kurker 2001) real-time discrete event simulator primarily used for forces modeling and simulation. It is employed as the main threat environment for Naval Air Systems Command (NAVAIR) Air Combat Environment Test & Evaluation Facility (ACETEF) test and training exercises (Mutschler 2007a). It was also employed for requirements generation by the Joint Strike Fighter (JSF) program office. Other uses include directed energy weapon modeling (Mutschler 2007b), weather modeling (Kelly et al. 2004), communication modeling (Chapman and Mutschler 2006), radar modeling (Worsham 2002), modeling of swarms of intelligence automata (Niland et al. 2005), air defense systems (Duquette, Nalepka, and Luczak 2004), and human behavior modeling (Hoagland et al. 2001), (Long et al. 2006).

1.1 Parallelization

JIMM is a legacy model with roots extending as far back as 1968 (JMMO 2008). It has its own simulation language to allow complex interactions as well as a graphical user interface for quick scenario development (Mutschler 2005a). However, despite its already efficient operation, there was still a desire to leverage multiple processors to execute larger and more complex test scenarios in ACETEF and other facilities while still meeting real-time deadlines. Hence, modifying JIMM to employ parallel processing was approved and funded by the Common High Performance Computer (HPC) Software Support Initiative as project #7 of Forces Modeling and Simulation (FMS #7) Computational Technology Area (CTA).

1.2 Using Worker Pools

JIMM is a real-time legacy model and is expected to operate in both single processor and multi-processor (high performance computing) environments. Therefore, performance in serial operation could not be severely impacted by the availability of parallel operation. In addition, shared memory symmetric multiprocessors (SMPs) were very common in ACETEF and other test facilities.

Hence, threads were chosen as the means for parallelization. They permitted separate thread processing for I/O and event execution. This would improve both serial and parallel operation. Threads also have very low overhead and can be used in the SMP environment.

There were other reasons for using threads. First, JIMM is a product currently in use where full releases are usually provided to the user community two or more times per year. Use of threads for I/O processing could be provided to the community earlier as a more immediate benefit of the parallel operation. Also, there was a significant fear of cancellation of the parallelization effort. Providing

results earlier would reduce loss should that cancellation occur.

JIMM is also used in real-time environments where steady simulation progress is the major requirement. This was thought to preclude approaches where rollback of simulation state could cause intermediate delays in simulator output. In addition, events in JIMM are computationally small and there was a major concern with communication overhead, even in a shared memory threaded processing environment. Lastly, there was a desire for totally repeatable operation to facilitate analysis and allow utilization of the extensive test programs already available (Gibson and Chapman 2001).

Furthermore, new events in JIMM can be scheduled for the same simulation time as their parent events. This was thought to preclude conservative parallelization approaches where a minimum non-zero simulation time between an event and successive events affecting the same simulation object is required. This left optimistic approaches where events occurring in future simulation time are calculated assuming little chance that their inputs would change (Fujimoto 1999). However, to ensure steady progress, significant rollback common to some optimistic approaches had to be avoided. Hence, future processing of parallel events was limited to one future event per simulation object.

For these reasons as well, a general worker pool approach (also known as "scatter and gather (SAG)" or "single process multiple data (SPMD)") was used instead an approach where different threads execute events simultaneously and communicate simulation state through messages. First, the worker pool approach is applicable within the SMP environment. Second, overall simulation state is saved at a single processing point once an event is finished processing. Total ordering of output, regardless of the number of processors can then be achieved permitting ready use of available test capability. There is no communication overhead. Lastly, the worker pool also ensures that one event currently being processed is the earliest event and would never be rolled back, thereby ensuring minimum forward progress.

Synchronization overhead was still a major concern. However, events in JIMM are well structured and affected simulation objects are identified before event execution. This would simplify detection for the need for rollback after event execution. Also, JIMM scenarios are characterized by a large number of simulated entities. This would alleviate overhead from rollback processing as well as the limit from processing only one future event per simulation object. Lastly, significant work could be put into reducing processing bottlenecks.

1.3 Parallel Operation

The general architecture divides thread execution into three parts: upper and lower critical regions that are protected by a single common mutual exclusion operation ("mutex") so that only one thread can execute within them and the last part where multiple threads can execute events in parallel (Mutschler 2005b).

After simulation start and until simulation end, threads simultaneously execute in a loop. After simulation start, the threads set a single mutex to ensure safe serial operation and enter the upper critical region (UCR) where they obtain events. They exit the UCR, unset the mutex, and process the events in parallel. After processing is finished, the threads again set the mutex and enter the lower critical region (LCR) where the safety of final event processing is determined. New events are queued. Lastly, output and state maintenance are arranged for later processing in parallel before executing a different event. The threads then enter the UCR without unsetting or resetting the mutex and the loop begins anew.

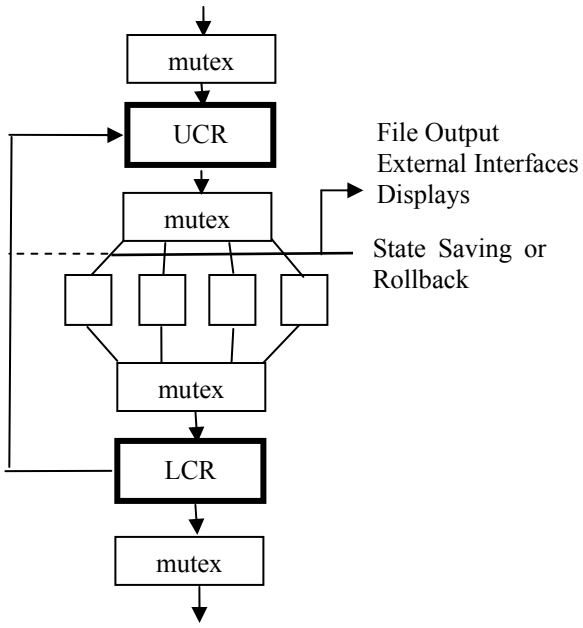

Figure 1: JIMM Architecture with Four Threads

In the UCR, events are obtained from a common priority queue ordered by simulation time. If simulation times are equal, then a unique event integer identifier that is incremented and assigned when events are initial queued is used to resolve order. A Least Global Virtual Time (LGVT) is defined as the simulation time and event identifier of the earliest ordered event in the queue, currently being executed, or already executed but awaiting final processing in the LCR because earlier events still exist.

After the event is obtained from the event queue, it is checked for possible collisions with events that have not

yet finished processing. Event collision occurs when the event examined could read or write data within a simulation object involved with a later event (in simulation time) that has been or is currently being processed. Processing of the later event hence might no longer be valid because it employed data that might have changed.

Should a collision occur, then if we are considering the later event, it is queued in a structure on the executing event and rescheduled once that event completes. Otherwise, this event is placed on a different list on the executing event and the executing event is marked for rollback. Once complete, the events are ordered and placed back on the queue.

If no collision would occur, the event is placed on a different priority queue for events being executed or events that finished executing and are awaiting final processing in the LCR. This common queue simplifies LGVT calculation.

A thread would execute an event in parallel once the mutex for the critical region is unset. Changes to simulation state are maintained on an ordered list associated with the event. Output to external interfaces is also kept on a separate ordered list associated with the event.

The thread then waits upon the mutex and once that is satisfied, enters the lower critical region (LCR). If the simulation of the processed event is equal to LGVT, then it is processed immediately. Otherwise, the most recent event on the queue of executed or executing events is then examined to determine if it has finished executing and if its event time would be the LGVT. If so, then it is processed.

Processing of an event with LGVT in the LCR involves several other actions. First, new events generated by the processed events are queued in the order generated and event identifiers assigned. LGVT is checked and updated when the new event would be the earliest unprocessed event in the simulation. Checks for collision and subsequent rollback are also made.

In addition, event output is also arranged to be sent in proper order outside the simulation before the thread processes its next event. State saving (moving forward) or rollback is also determined to be executed for each simulation object before its processing of its next associated event. Event output and simulation object rollbacks are handled in the main body and outside the LCR to decrease the impact of the critical regions as processing bottlenecks.

After processing an event, the queue of executing and executed events is repeatedly examined and the earliest event processed until the earliest event time is no longer the LGVT.

1.4 Optimizations

A number of optimizations were made to limit the impact of the UCR and LCR bottleneck. First, these regions were coded to be highly efficient. Aforementioned optimizations include the following:

- Update or rollback of simulation state is determined within the LCR but processed outside the critical regions before the next pertinent event for the affected simulation object.
- Submission of output to external displays and files is ordered in the LCR but processed before processing of a thread's next event.
- Use of a common queue for events currently executing or awaiting final processing to simplify LVGT processing.

Other optimizations include the following:

- A common memory pool already employed by JIMM was coded for parallel operation (Mutschler 2006).
- Input from external sources is also handled outside the critical regions.
- Each event is assigned a main simulation object and then ordered by time on a list associated with that object. This list is referenced as a single structure on the main simulation queue based on the time of its earliest event. This avoids collisions of events with the same main simulation object.
- The simulator can be adjusted to obtain more than one event in the UCR and then execute them one after the other. This can reduce synchronization overhead due to mutex operation at the cost of potential parallel execution.
- A separate method is used for serial execution of events as opposed to event execution in parallel. This reduces overhead from parallel operation capability when it is not necessary and there no benefit from parallel operation would be achieved.

2 IMPLEMENTATION OF STATE SAVING

JIMM is implemented in the C++ programming language and makes extensive use of object-oriented programming constructs such as derived classes that inherit data and function methods from associated base classes.

Execution of events in parallel was implemented in the last phases of the parallelization effort. Associated with these phases was the saving of the state of objects associated with events in the case of rollback.

State saving data is implemented as a doubly-linked list of objects associated with a single simulation object. The objects have a base class known as a "result". Consequently, the list is known as a "result list". The base class includes pointers for the list as well as specification of two function methods: "Apply()" where the simulation state of the object is changed, and "Discard()" where the simulation object is returned to its original state. The derived

classes of the result base class contain pertinent data as well as implementations of the Apply() and Discard() functions.

Events act upon simulation objects. At the start of event, the result list for each of the simulation objects should be empty. When the event is processed, results are appended to the result list as needed. When an event is determined to be safe (e.g. has the earliest LGVT) or rolled back (e.g. an event collision has occurred), the determination is noted in a bit stored within the simulation object. The results are later processed before the simulation object is accessed in a different event.

If the event is determined to be safe for final processing, then the result list is processed through execution of the "Apply()" function from earliest result to last result to ensure order of update. If the event must be rolled back, then the result list is processed from its last update to its earliest through execution of the "Discard()" function to ensure correct reverse order of state restoration.

In deferred modification, the original state is maintained in the simulation object and changes are usually (but not always) stored in the result class. The event is then coded to use the interim values and ensure that the initial state in the object is not altered. Processing of the result list via the Apply() function overwrites the initial state in the simulation object with the modified state. Processing of the Discard() function discards the changes but makes no modification to the original simulation object state, leaving it intact.

In immediate modification, the original value is stored within the result class. The data in the original data structures for events are then modified and used. When the Apply() function is executed, the original data is thrown away since saving it is no longer necessary. When the Discard() function is executed, the original state is restored.

Saving and restoration of simulation state is only handled by result classes and these must be used for each state change. This explicit update was assumed to be more efficient in terms of performance than employing classes that perform state saving automatically.

Results can be programmed for each variable type as well as structures or other collections of data. In the beginning, all the data for a simulation object in an event was handled by a derived result class instance. Specific code modules were denoted by a "dr_" prefix for "derived result".

3 EVOLUTION OF STATE SAVING

As a legacy model in current use, JIMM is updated and released two or more times per year. Most corrections and enhancements are achievable as single stage efforts provided in a single release. Larger efforts are divided into increments that correspond to the releases. Because work for parallel operation was a multi-year effort, it was divided into parts for incorporation into successive versions.

The early part of the parallelization effort from August 2000 to 2002 focused on use of threads for output, allowing multithreaded access to terrain and the memory pool, better organization of events via derived classes, organizing event output for external transmission after the event completed, and eliminating cases where simulation state was modified outside events (JMMO 2008).

Work on parallel execution of events started in late 2001. In retrospect however, specific development and integration of parallel execution of events had three major stages: initial development using deferred modification up to successful demonstration of parallelism starting in early 2002 and ending in 2003, rejection of deferred modification and completion of the development effort, and subsequent perfective maintenance and greater use of generic classes through 2007. Employment of result classes has evolved over these periods.

3.1 Initial Development

When the parallelization effort was initiated in 2000, the proposed preliminary architecture was examined. Checking of collisions in the UCR was not thought to be necessary and use of deferred modification of simulation objects would allow associated events to be executed in parallel safely. This assumption was later rejected since update of simulation state when one event was determined to be safe could still occur during processing in the main body of a different event with the same associated simulation object. However, the preference for deferred modification was established.

State saving using deferred modification was implemented and tested for several small events and shown to operate correctly. Developers initially considered the approach to be straightforward.

Generic routines for saving of basic types such as integers, doubles, pointers, and lists where implemented but were not used extensively. Instead, each event had an associated "result" class for each simulation object which handled all the state saving processing.

In subsequent development, use of deferred modification worked well. Unfortunately, as more complex events were considered, retaining and using the intermediate state instead of the original state required heroic programming efforts. Extensive structures containing intermediate data needed to be retained and passed from procedure to procedure. Coding became quite cumbersome especially as common procedures and classes were used in multiple events.

Even so, the methods worked for an initial implementation involving the major events associated with sensing, perception cognizance, and decision making. Near linear

speedup with up to twenty-four separate processors was obtained (Mutschler 2005b).

3.2 Rejection of Deferred Modification

Once the initial parallel version was complete, many events still needed to be coded for operation in parallel. This effort was completed by the JIMM Model Management Office (JMMO) in the following year as part of its code maintenance efforts.

In maintaining the code, several problems became apparent. First, event implementations could rarely mix deferred modification (a.k.a. "Apply-based") and immediate modification (a.k.a. "Discard-based") methods. An event had to be implemented either one way or the other. This was especially a cause of concern given the use of common procedure and classes and the now inherent utilization of state saving.

Another problem was that much of processing for events employing deferred modification was moved outside the event into the Apply() function of the result class. This made the code difficult to follow and understand. On the other hand, events employing deferred modification retained much of their original coding inside the event, thus making understandability and subsequent code maintenance much easier.

After long deliberation, the JMMO determined that even though many events were already implemented using deferred modification, it would no longer be used and that extensive effort would be undertaken to convert events implemented using deferred modification to immediate modification methods. This effort has been completed. However, instance of the use of deferred modification are still found and treated as low priority required software changes.

3.3 Using of Generic Classes

One of the by-products of the use of deferred modification was the generation of many instances of derived "result" classes specific to events and common procedures. The number of files became very cumbersome.

As events were converted (or modified to operate in parallel), it was noted that many functions of the result classes could be handled by several instances of more "generic" result classes that operated only on basic data types such as integers and pointers or on simple types such as lists. Simulation state saving would not be done in a single class instance but would be done with multiple smaller result class instances on the event's associated result list.

Because the generic result classes only dealt with simple types, their implementation was very efficient. Thus, the associated performance cost of employing them vice a single result class instance was negligible. Moreover, code module understandability improved significantly, thereby reducing overall maintenance costs.

The generic result classes were expanded slightly with instances to handle simple classes, structures, and other blocks of data. Otherwise, they were left unaltered.

Eventually, many of the larger event-specific result class instances were completely replaced. The JMMO then determined that all event or procedure specific result class instances would be replaced with generic results. As of November 2007, less than a dozen of these instances (from an initial count of more than one hundred fifty) remained.

The JIMM simulator is still in use. The release of version 3.2 was provided to the user community in June, 2008 (JMMO 2008). Maintenance of parallel execution of events continues.

4 CONCLUSION

This paper has described the use of threaded worker pools to execute events in parallel with a real-time legacy simulator known as the Joint Integrated Mission Model (JIMM). Methods for deferred modification and immediate modification of simulation state variables are discussed and shown to be different applications of a common state saving class known as "results".

During initial conversion to parallel operation, deferred modification techniques were employed successfully. However, as the code was later maintained, this approach was rejected and techniques using immediate modification were employed instead due to the need for increased understandability and simpler overall construction given complex code and use of common procedures.

Eventually, more elaborate derived result classes specific to events and common procedure were replaced with more generic result class instances that handled basic types, blocks, and lists.

ACKNOWLEDGEMENTS

JIMM was created and derived from previous models developed by Peter Lattimore. The initial implementation of the state saving ("result") structure was provided by William Brooks. The initial version was developed by Jon Anderson, William Brooks, Michael Chapman, Ralph Gibson, Doug Pickeral, Jon Smith, and others. Extensive enhancement and utilization of the generic classes was implemented by Blair Kitchen. JIMM is currently maintained by the JIMM Model Management Office (JMMO). Information about JIMM may be obtained by contacting the JIMM Model Manager at <jmmo@navy.mil>.

REFERENCES

Chapman, M. D., and D. W. Mutschler. 2006. Communication Modeling in the Joint Integrated Mission Model

(JIMM) and the Air Combat Environment Test & Evaluation Facility (ACETEF). *ITEA Modeling and Simulation Conference*, Las Cruces NM.

Duquette, M, J. Nalepka, & R. Luczak. 2004. The enhanced generic air defense system. *AIAA Modeling and Simulation Technologies Conference and Exhibit*. Providence RI. (AIAA-2004-4799)

Fujimoto, R. M. 1999. *Parallel and Distributed Simulation Systems*. John Wiley & Sons Inc. New York, N.Y.

Gibson, R. D., and M. D. Chapman. 2001. Legacy software testing – a current methodology. *The Eleventh Annual International Council On Systems Engineering (INCOSE)*. Melbourne, Australia.

Hoagland, D., E. Martin, and M. Anesgart. 2001. Representing Goal-Oriented Human Performance in Constructive Simulations: Validation of a Model Performing Complex Time-Critical-Target Missions. *Proceedings from the Spring 2001 Simulation Interoperability Workshop*. Simulation Interoperability Standards Organization. San Diego CA. Paper Number 01S-SIW-137.

JIMM Model Management Office (JMMO). 2008. *JIMM 3.2 Users Guide Volume One*. Available via <jmmo@navy.mil>.

Kelly, M., S. Vick, J. Schloman, J., and F. Zawada. 2004. A Weather Service for Introducing Dynamic Attenuation Factors in the Joint Integrated Mission model (JIMM). *Proceedings from the Fall 2004 Simulation Interoperability Workshop*. Simulation Interoperability Standards Organization. 04F-SIW-107.

Long, G., D. W. Mutschler, and G. Lohman. 2006. Human Behavior Modeling in the Joint Integrated Mission Model. *Proceedings from the Fall 2006 Simulation Interoperability Workshop*. Simulation Interoperability Standards Organization. Orlando, FL.

Mutschler, D. W. 2007a. Employing Future Path Information to Improve Position Accuracy in Distributed Simulations. *Proceedings of the Fall 2007 Simulation Interoperability Workshop*. Simulation Interoperability Standards Organization. Orlando FL. (07F-SIW-033)

Mutschler, D. W. 2007b. Integration of Directed Energy Weapons Modeling in the Air Combat Environment Test & Evaluation Facility (ACETEF). *Directed Energy Professional Society (DEPS) Modeling & Simulation Conference*, Monterey CA.

Mutschler, D. W. 2006. Enhancement of Memory Pools Toward a Multi-Threaded Implementation of the Joint Integrated Mission Model (JIMM). In *Proceedings of the 2006 Winter Simulation Conference*, ed. L. F. Perrone, B. Lawson, J. Liu, F.P. Wieland, 856-862. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Mutschler, D. W. 2005a. Language-based Simulation, Flexibility and Development Speed in the Joint Integrated Mission Model. In *Proceedings of the 2005 Winter Simulation Conference*, ed. M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, 1190-1197. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Mutschler, D. W. 2005b. Parallelization of the Joint Integrated Mission Model (JIMM) Using Cautious Optimistic Control. *Proceedings of the 2005 Summer Computer Simulation Conference*. Cherry Hill NJ, Society for Modeling and Simulation International, 145-152.

Nalepka J., J. Gump, R. Kurker. 2001. JIMM: The next step for mission models. *2001 SPIE Aerospace/Defense Sensing and Controls Conference* (#2367), Orlando FL.

Niland, W., B. Skolnik, S. Rasmussen, K. Finley, and K. Allen. 2005. Enhancing a Collaborative UAV Mission Simulation Using JIMM and the HLA. *Proceedings of the Spring 2005 Simulation Interoperability Workshop*, Simulation Interoperability Standards Organization, San Diego CA.

Worsham, R. 2002. Northrop Grumman Radar Simulation (AVSIM). *Proceedings of the 2002 IEEE Radar Conference*, 176-186.

AUTHOR BIOGRAGHY

DAVID W. MUTSCHLER has been employed by the U.S. Navy in the Naval Air Systems Command (NAVAIR) since 1985. He obtained his doctorate in Computer and Information Science from Temple University in 1998. He worked on the Joint Integrated Mission Model (JIMM) and its predecessor, the Simulated Warfare Environment Generator (SWEG), from 1996 to 2007. He was the principle investigator for the JIMM parallelization effort from 2000 to 2003 and the JIMM Model Manager from July 2004 to Feb 2006. He is now the Government Software Integrated Product Team (IPT) Lead for the CH-53K Heavy Lift Replacement (HLR) rotorcraft. He is a member of the Association for Computing Machinery (ACM), its Special Interest Group in Simulation (ACM SIGSIM) and the Institute for Electrical and Electronics Engineers (IEEE) Computer Society (IEEE/CS). He is also an Associate Professor of Computer Science at the Florida Institute of Technology (FIT) University College – Patuxent River MD site. His e-mail address is <david.mutschler@navy.mil>.