JTED: Parallel Discrete-Event Simulation in Java

James Cowie

Cooperating Systems Corporation
Deering, NH 03244

JTED is a prototype of a Java-based framework for discrete event simulation, currently demonstrating parallel speedup on shared-memory multiprocessor platforms. JTED specifies a set of base classes for building simulations; the modeler defines a particular simulation by creating derived classes, filling in the "hot spots" of the framework with active event-processing code, and registering instances of the derived classes with the running framework. The JTED modeler is freed from some of the routine bookkeeping chores common to most discrete-event simulations, including event delivery, timestamp generation, process scheduling, and logical clock management. The framework also provides standard hooks for integrating graphical tools with a running simulation. Most importantly, the framework's semantic restrictions allow user models to be parallelized, with various degrees of transparency, without the intervention of a compiler or a special-purpose modeling language.

JTED attempts to replicate (in Java) the advances in high-level modeling achieved by the TeD system from Georgia Tech [2, 3], which used a custom little language compiled to C++ and the Georgia Time Warp simulation kernel[1]. The design metaphor common to TeD and JTED models is that of nested **Entities**, connected by **Channels** over which **Processes** receive and dispatch **Events**, as described in figure 1.

Demonstrated Benefits of Java

Anonymous Inner Classes Java's support for *anonymous inner classes* has proven especially useful when defining processes. In C++, a process and its associated entity would have no special relationship, other than mutual reference exchange. Process code must explicitly qualify references to the entity's state (for example, channel endpoints) by reference to the entity. The JTED framework allows a given process to be associated with exactly one entity, though a given entity may have

JTED.Entity	Container for processes, channel endpoints, other entities
JTED.inChannel	Channel interface for receiving events from other entities
JTED.outChannel	Channel interface for sending events to other entities
JTED.Process	'Hot spot' for modeler's event processing and event generation code
JTED.Scheduler	Virtual processor: one logical clock to which Entities are aligned
JTED.Event	Quantum of information exchanged by entities over channels

Figure 1: JTED Framework Base Classes

multiple processes; this rules out ugly alternative schemes based on multiple inheritance. In Java, the user can define each process naturally as an inner class of its containing entity.

Reflection Java's reflection facilities allow JTED to construct a partner data structure for the entity/process/channel graph, known as a *schematic diagram.* A schematic enumerates all the fields that are publicly reachable from a given Object, by repeated calls to `getClass().getDeclaredFields()`, with table lookup to identify and break cycles of references. The schematic for JTED's root entity, for example, enumerates all the component entities, channels, and processes that are accessible from the root, and gives them unique names within the simulation. This schematic can then be used as the basis for representing the attributes and structure of model components in interactive tools.

Shared Memory Parallelism Our first instinct was to map each user process to a single Java thread, but the overhead of thread suspension/resumption on each wait() quickly dominated running time. We identified a subclass of JTED processes (the **inlinable processes**) that execute in zero logical time, waiting only once at the end of their execution. The `action()` methods of these processes are executed on the scheduler's thread, avoiding threading overhead.

Each entity is *aligned* to a given JTED scheduler; entities that are aligned to the same scheduler can perform direct state access on each other, rather than paying the overhead of exchanging many small query events. Each scheduler runs in one concurrent Java thread, which maps to one system (native) thread, thus hiding the mechanics of mapping logical concurrency to physical processors from the modeler.

Visualization and Control Figure 2 shows a screen dump from the JTED console prototype, showing an event being passed from one process to another as logical time advances. Process space runs top-to-bottom, and logical time runs left-to-right. The "stoplight" control in the bottom toolbar allows the user to start, stop, pause, and resume the running simulation in order to understand its structure and identify performance bottlenecks.

Example: 2-D Forwarding Mesh

We constructed a simple two-dimensional toroidal mesh (closed north-south and east-west) of forwarding nodes, in order to understand the sources of concurrency in a nontrivial JTED model.

Messages are generated by local bursty sources ($p_{on} = 0.05$, $p_{off} = 0.20$) colocated at each switching node in the mesh. These sources have a 250ns cycle time.

Forwarding nodes route the messages east-west until they reach their destination longitude, and then south-north until they reach their destination latitude. To achieve this effect in the JTED model, two processes in each node receive and buffer incoming messages from the west and south; a third process routes received messages to the north or east, unless they are addressed to the local node.

Each forwarding node has a 64-message FIFO, with a 400ns service time. There are microsecond transmission delays on each east-west and north-south link (e.g., 650 feet of coaxial cable). The simulation clock has nanosecond resolution. Messages are injected into the toroidal mesh for 100 microseconds, followed by another 100 microseconds of time for the messages to drain from the network.

Figure 2: JTED console application screen shot. Process space runs top-to-bottom, and logical time runs left-to-right.

We timed model executions for a variety of mesh sizes and shapes, ranging from four nodes (2x2) to 1024 nodes (64x16), block-partitioned by columns across from one to eight schedulers (logical processes). In the largest meshes tested, a total of 4096 simulation processes were concurrently active (three per forwarding node, plus one per source, across 1024 nodes and 1024 sources).

Conservative synchronization techniques are applied on each channel that crosses a scheduler boundary, so that the recipient's progress past time t can be held back until the potential for incoming events of timestamp less than or equal to t has been eliminated by contract on the channel. JTED uses a "classical" window-based scheme, creating periodic barrier synchronizations among all schedulers based on the minimum lookahead off all cross-scheduler channels.

We compared parallel versions of the JTED model using Java's green threads and native threads. In the one-processor tests, all of the forwarding mesh nodes and event sources remained aligned to a single scheduler (logical process). In the multiprocessor tests, each block of columns of mesh nodes and their associated event sources were aligned to one scheduler; timings were gathered for different degrees of potential parallelism, from one-way to eight-way.

This was not an embarassingly parallel partition in any sense, since many mesh edges connect forwarding nodes located in neighboring columns that cross block partition boundaries. These so-called "cross-scheduler" edges determine the necessary synchronization conditions; their available lookahead (determined by the speed-of-light transmission delay) bounds the progress of any scheduler with respect to its western neighbor.

Forwarding nodes and bursty sources within each scheduler (i.e., block of mesh columns) were simulated within one Java thread. In the native-threads tests, each Java thread was in turn mapped to one lightweight process, running on one processor of a moderately loaded 14-processor Sun Ultrasparc, running Solaris 2.6, JDK1.2 beta 2, and JTED 0.4.0.

The results are summarized in figure 3. Using native threads, reasonable speedup was evident for all mesh sizes. As the number of running Scheduler threads begins to exceed the number of idle processors, synchronization overhead increases, and speedup ultimately grinds to a halt. Scarcity of logical-time parallelism demands careful consideration of scheduler set size and entity alignment. The performance knee kicks in sooner than the physical processor count would seem to predict. [1]

Conclusion JTED demonstrates that putting Java to work in discrete-event simulation requires the promise of good development tools and transparent access to parallel hardware. Java's unique features (security, portability, serialization, built-in multithreading, and reflection) have the potential to outweigh C++'s raw performance advantages in development of high-level modeling environments for parallel discrete event simulation.

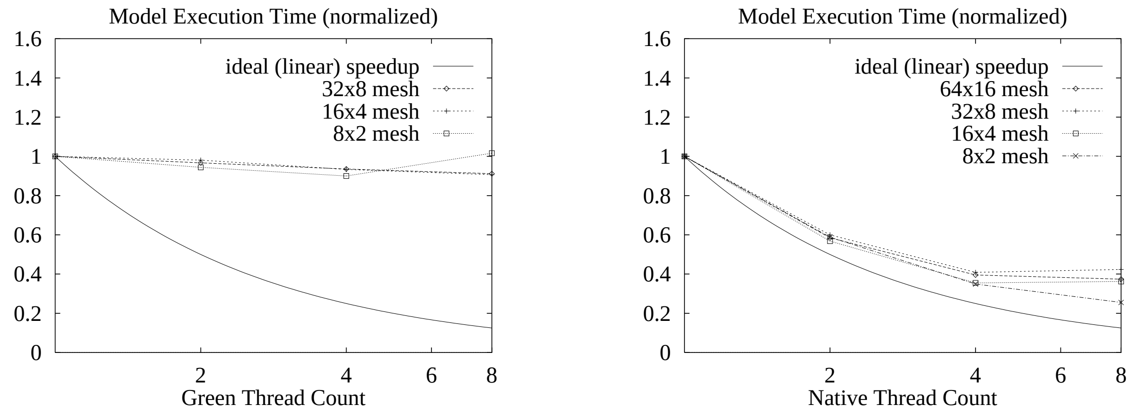

Figure 3: Model execution time (normalized to one thread) for various sizes of the 2-D toroidal forwarding mesh, block-partitioned over one to eight green and native threads, on a moderately loaded 14-processor Sun Ultrasparc.

References

[1] S. Das, R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. GTW: A Time Warp system for shared memory multiprocessors. In *1994 Winter Simulation Conference Proceedings*, pages 1332–1339, December 1994.

[2] K. S. Perumalla and R. M. Fujimoto. A c++ instance of ted. Technical report, College of Computing, Georgia Institute of Technology, 1996.

[3] K. S. Perumalla, A. T. Ogielski, and R. M. Fujimoto. Metated – a meta language for modeling telecommunications networks. Technical report, College of Computing, Georgia Institute of Technology, 1996.

[1] While the test machine has 14 physical processors, in practice the load average on the machine hovers around 2.00 due to daemon processes, activity by other users, and so forth.