David Dantowitz, sole developer
david@dantowitz.com
201-532-3053

A large, complex project in C, written from scratch, was a new synchronous parallel event-driven multi-threaded simulator for shared memory multiprocessors. It was developed with a highly optimized message transport system that minimized cross processor memory traffic and the use of synchronization primitives.

The goal was to study and develop a discrete-event simulation engine with maximal efficiency for single processor and shared memory parallel computers. The simulator permits a broad range of systems to be modeled without attending to the fact that you may or may not be running on parallel hardware.

Accomplishments

In designing and implementing a complete parallel simulation engine from scratch and running it on three different operating systems and platforms (one a parallel super computer), I learned a great deal. A high level look at some of the tasks included the need to design high performance, scalable parallel data structures, analyzing how data is shared by processors in parallel, and debugging the kernel, a complex multi-threaded parallel application.

On a parallel processor it is preferred that the OS has the ability to allocate dynamic memory three ways:

a) locally (non-shared),
b) shared, location not important
c) shared memory located on processor *i*,
where *i* is specified at the time memory is requested

Performance was optimized in two areas:
a) Cross-processor memory traffic
b) Synchronization primitives

Cross-processor memory traffic can't be avoided when objects are running on different processors, but the overhead can be reduced by optimizing the code that creates, schedules, and moves messages between objects. By developing a specialized transport mechanism, I was able to streamline message handling to a very high degree.

Synchronization primitives are a key source of overhead in synchronous parallel processing:

i) for each clock tick in the simulation the processors must wait for others in the simulation group

ii) synchronization primitives are used to control access to shared memory resources used by the simulation engine

With locked-step, synchronous simulation, the overhead due to (i) is unavoidable unless conservative or optimistic simulation methods are used for clock updates. At least one *all* processor synchronization primitive is required for each step of the simulation to keep the processors in lock step.

Synchronization is also required in access methods (ii) to keep data structures consistent when shared asynchronously by multiple processors. The need for this type of synchronization was removed by creating specialized data access methods to minimize synchronization requirements. In essence, time and the knowledge of the locations and types of data access were used as the synchronization primitive.

To minimize synchronization primitives the goal was set to use a single all-CPU barrier per cycle of simulation. This set in motion the need to develop lock or latch-free data structures for communication between CPUs for delivery of messages. Once processes are assigned to CPUs, and the paths of communication between processes declared, those paths are aggregated to dedicated CPU to CPU message queues. Each message has a source, destination, delivery time and a chunk of data. Each CPU has a message processor that manages these queues.

Because each queue is written only by one CPU and read one other CPU, and reads and writes take place at different locations in the structure, there is no need for additional synchronization. The data structures are self-synchronized by time. The Consumer CPU handles messages to be delivered at the current simulation time. The producer adds messages at future times. This boils down to one reader and one writer at distinct locations in the data structure.

When work for a simulation time step finishes, each CPU determines the next time step for the simulation from its point of view. This information is shared so that all CPUs are in agreement as to the next cycle time. This is accomplished with a modified butterfly barrier where the minimal cycle time is shared between CPUs as they reach the barrier. As an aside, this facilitates a minor optimization by allowing a simulation to skip ahead cycles where no messages are delivered across all CPUs.

The Details

$T_{current}$ and TickTock begin at 0. $T_{current}$ monotonically increases and represents the current simulation clock. TickTock alternates between 0 and 1 on each cycle of the simulator and is key to synchronization. The pointer to the queue of messages from CPU A to B at $T_{current}$ is located at Queue[A,B,TickTock]. This is only accessed (read) by CPU B. Meanwhile, CPU A is free to update the future messages queue, located at Queue[A,B,(TickTock xor 1)] at will, knowing that CPU B may not read it until the next simulation cycle. So, for N CPUs, Queue is a 3 dimensional matrix NxNx2 in size.

As messages are created by CPU A for CPU B, for each CPU A there are N threaded trees in RAM local to CPU A, one for each CPU B. Each tree node maintains a simulation cycle time for messages in the queue to be delivered from A to B and a pointer to the first and last message at that delivery time (there are N queues as well from A to B). As the tree is threaded, it provides O(1) access for the next delivery time, as the lowest time value is the head of the threaded tree from CPU A to CPU B. As most simulations go, the bulk of messages are delivered in the next or near next simulation cycle time, which means you can cache the last message added before searching the tree. Removing nodes from the tree is straight forward as they are removed in ascending order, and is thus the left most node.

When a CPU A is done consuming and producing messages its message processor determines the minimum next simulation cycle from its point of view by finding the minimal message to be delivered (read the head of each of the N threaded trees). It then stores the queue head pointer for each CPU B in Queue[A, B, (TickTock xor 1)] for those at that are the minimum and waits at the barrier.

When the barrier is complete, $T_{current}$ is set to the new simulation cycle time and TickTock is set to TickTock xor 1. Then, each CPU can read messages in its respective queues and deliver them to the processes it is simulating.

Note that CPUs stop delivering messages when the next message in the queue is NULL or has a deliver time in the future. Reading the next message pointer in a queue element does not require any form of synchronization. It is either NULL, or a message to be delivered now or in the future.

To further improve performance across CPUs, shared RAM use is minimized, as it is more costly than local RAM. By definition each message crossing from one CPU to another incurs cross processor penalties, so this is unavoidable in many cases. One can partition the processes needing connections to reside on the same CPU and minimize this cost, but as the primary use for the simulation engine was multi-stage connection networks it wasn't considered. This problem is similar to the "partitioning problem" of chip element layout and minimizing cross chip connections. It could be addressed if simulating systems that benefitted from optimized distribution of processes across CPUs.

An additional performance gain was achieved with an internal dynamic memory management library. This enabled the engine to create and re-use pools of shared memory as each initial request for shared memory had a major set up cost from the OS Kernel.

The use of self-synchronized or time-synchronized data structures, a single barrier per cycle with information dissemination, intelligent use of local vs. cross CPU shared RAM and a custom memory manager created measured performance gains.

Instrumentation

To study performance and related statistics, the kernel is heavily instrumented to measure delays due to synchronization, message delivery, thread use, and more (instrumentation is enabled optionally at compile time). The kernel also has three levels of debugging settings, which aid in porting the kernel and tracing and debugging simulations.

Status

Results from parallel runs of existing simulations were excellent and demonstrated significant gains from running on multiple CPUs.