Parallel evaluation of character rigs using TBB and vectorization

Martin Watt
DreamWorks Animation
Overview

Threading
Parallel graph design

Vectorizerization
Applied to animation graphs

Hardware challenges
eg CPU Power modes, Hyperthreading, Turbo Boost, Memory, NUMA

Lessons learned - all of the above
Motivation

Existing animation tool (EMO) unthreaded so not getting faster.

Complexity of characters continues to increase

Too difficult to retrofit existing tools with threading

Need to build new tools and new engine for scalability
Goals

Much faster evaluation - interactive manipulation of character rigs

Immersive workflows, a fluid artist experience

High quality display of full resolution characters and environments
Character workload

Motion system (bones)
  100s to 1000s of relatively lightweight nodes

Deformation system (skin, muscle)
  10s of heavier nodes
  ~1 million vertices on meshes

Effects (Cloth, hair, fur)
  Very expensive nodes
Hero Rig dependency graph
(150k nodes)
Character animation features

Character graph has natural parallelism
  eg limbs usually independent

Some heavy nodes (eg deformers/solvers)
  Use internal threading, so require nested threading support

Require real time (>10fps for thousands of nodes)
  Scheduling and graph traversal overhead needs to be very low
Evaluation mechanism

Dependency Graph

Traditional DG has two pass evaluation
- Dirty propagation
- Pull evaluation (recursive, hard to parallelize)

Libee has three pass evaluation
- Dirty propagation
- Upstream traversal to determine dirty nodes to recompute, tag dependencies
- Final pass evaluating nodes concurrently
Evaluating graphs efficiently

Graph is evaluated each frame
  Typically thousands of nodes and edges
  Need to evaluate in tens of milliseconds

Multithreading can help performance
  Nodes can be executed in parallel
  Question of the best approach to use
Low level threading

Could potentially use native pthreads
  Create graph partitions
  Manually assign partitions to threads

However, graph partitioning is a hard problem
  Suitable partitions may take longer to compute than frame time
  Want solution that can quickly adapt to graph changes

Also, pthreads are not good for nested parallelism
  Some nodes might exploit parallelism internally

Would like the threading at different levels to work together
Higher level threading

OpenMP does not handle nesting well

Threading Building Blocks (TBB) solves our issues
  - Supports nested parallelism
  - Reasonably low overhead

Each thread has a task pool
  - Idle threads steal work
  - Provides load balancing
Serial evaluation

Evaluate only nodes that are necessary
Artist may not modify all controls
Not all outputs may be visible
Parallel evaluation

List of nodes is no longer sufficient
  Parallelism removes ordering guarantees
  Must track dependencies between nodes

A node with ready inputs is enabled
  Corresponds to spawning of TBB task
  The task will be executed at some point

Tasks also spawned for internal parallelism
Chaining optimization

One task per node creates high scheduling overhead

Over 75% of nodes are in chains

Group serial chains of nodes into single TBB task
  Reduces threading scheduling overhead
  Improves cache locality
Rig evaluation ordering

Need to order dependencies to enable parallelism

Riggers need to become thread-aware

Took us way too long to realize this

Finally created tool for riggers to show graph eval
Parallel view (Hiccup, HTTYD2)
Critical path most important
Others less important
Focus limited resources on critical path

Don't start here even if most costly node in graph
Reordering evaluation
Manual parallel evaluation
(Almost) free clothing!

Critical path does slow down a little as other concurrent tasks are added.
Multiple characters
Most benefits at graph level

None

Node

Graph
Threading challenges

R&D and artists can author operators

Engine is new, but code running in nodes is often old

Operators can call into arbitrary studio code

Code can be running concurrently even if not threaded
  Need to think about threading even if not ‘threading’
  Parallelism exists at higher level than algorithm
Approach

Clean up studio code
  Significant effort

Review
  Adopt review process for new nodes to validate for threadsafety

Validation
  Parallel unit tests
  Code review
  Compiler flags
  Thread checking tools
Additional restrictions imposed

No scripting languages, eg Python, in-house scripting languages
  Too slow, not threaded
  Concern for riggers, easier to use script-based nodes

Nodes cannot directly access other nodes
  For example expression nodes that query other node attributes disallowed

No loops in the graph
Keeping code threadsafe

Parallel unit tests run 24/7
Intermittent failures show possible threading bugs
TBB locks - be careful

tbb::spin_mutex mutex;
{
    tbb::spin_mutex::scoped_lock(mutex);
    [non-threadsafe code]
}

What is wrong here?
TBB locks - be careful

tbb::spin_mutex mutex;
{
    tbb::spin_mutex::scoped_lock(mutex);
    [non-threadsafe code]
}

No named object to persist!
Lock destructs at semicolon, code is unprotected
TBB locks - be careful

tbb::spin_mutex mutex;
{
    tbb::spin_mutex::scoped_lock myLock(mutex);
    [non-threadsafe code]
}
TBB locks - be careful

This happened at DreamWorks multiple times. Even to extremely experienced engineers.
TBB locks - be careful

How did we find it?

Enabled stricter compiler warnings:

```bash
> icpc -c main.c // before - no warnings
> icpc -w3 -c main.c // enable strict warnings -w3
```

remark #3280: declaration hides variable "mutex" (declared at line 3)

```cpp
tbb::spin_mutex::scoped_lock(mutex);
```

Different warning but does point to the problem, somewhat luckily
Thread local storage

C++11 keyword

Useful when state persists beyond scope of method, eg in legacy C code.

Less needed in C++ - use class scope
Thread local storage problems

Limited in size. Can run out.

Limited in number of libraries allowed (16)
dlopen: cannot load any more object with static TLS

Have custom Linux glibc patch from RH to increase limit!
(Bumped up in RHEL7.x)
https://bugzilla.redhat.com/show_bug.cgi?id=1124987

TLS not really recommended. More of a band aid.
Memory allocators

Need a thread-friendly allocator, unless/until remove mallocs

Using TBB’s memory allocator, tbb_malloc
Performs well.

Occasional pathological behavior with incrementally growing reallocs.

Slightly better than jemalloc for Premo, but YMMV
Can be useful, but sometimes a lot of false positives
Vectorization

Threading offers largest initial gains
  First version of LibEE incorporates threading

Next look at vectorization
  Heavy compute nodes, eg deformers
Vectorization - starting point

Deformers <50% of runtime (green nodes above)
Amdahl-like law applies here too. Limited benefits for this example
Rig optimization procedure

Want to vectorize deformers
   Only ~30% of eval time
   Skeleton takes >50% of time

So... optimize skeleton first
   Optimize the non-vectorized part first
Motion system optimization first

Optimization:
Transform Building Blocks (XBB)

Avoid zero multiplies (very common with matrices)

Link at end of this slide deck

No threading
No vectorization
1.6x faster
Programming SIMD

Assembly
Really hard

Intrinsics
Hard, locked in to specific vector instruction set

Compiler autovectorizer
Safer
Portable
Maintainable
Tricky to guarantee vectorization
SIMD Building Blocks (SBB) is a C++ template library for vectorization offering:

- Containers, accessors, kernels, engines
- Optimizes code so compiler can auto-vectorize
- Handles data transformation & alignment
- Engine handles iteration
- Works with any C++11 compiler
- Enable transparent vectorization + threading (single threaded, TBB, OpenMP) execution of kernels.

“TBB for Vectorization”

Link at end of this slide deck
SIMD Building Blocks (SBB)

Generate efficient SIMD code by encapsulating in-memory data layout of objects isolating it from kernels.

Allows containers to transparently use an SOA data layout.

Provides methodologies that avoid common pitfalls to generating efficient SIMD code.
SBB compiler requirements

Need some C++11

No proprietary Intel compiler features are required

Other compilers may not generate vectorized code
Vectorization speedups
Overall speedup

Production Rig:
  Deformer: 4x faster
  Overall Rig: 10% faster
Two types of scaling

Amdahl’s Law
   Same problem in smaller time

Gustafson’s observation
   Bigger problem in same time

We are usually on the second of these
Complexity always increases

Shrek’s Law
Increase workload

Can evaluate twice the mesh resolution in the same time
Overall speedup

Production Rig:
  Deformer: 4x faster
  Overall Rig: 10% faster

Scaled up production rig (2x resolution)
  Deformer: 4x faster
  Overall Rig: Same speed
Hardware issues

CPU Power modes
Hyperthreading
Turbo Boost
Memory Wall
Thread Affinity
NUMA
More cores vs more clock?
Power saving vs performance mode

Found a big difference (~20%)

Problem is workloads are very bursty
CPU takes time to clock up
Other system activity

TBB assumes it has all cores available

Often one or more cores used for other purposes

With higher thread counts, ok to give up 1 or 2 threads
Hyperthreading

One physical core
acts like

Two logical cores

Image © Intel
Hyperthreading: load balancing

More work than cores

Workload uses all logical cores

Benefits from HT
Hyperthreading: load balancing

More cores than work

Two tasks can run on same core while other cores are idle

Hard to avoid as tasks come and go

Inefficient system usage
Hyperthreading vs workload

Walk cycle with varying thread count and Hyperthreading

Frames per second

HT off
HT on

1 thread 2 threads 3 threads 4 threads 5 threads 6 threads 7 threads 8 threads
Turbo Boost

Fewer active cores
->
Faster clock speed

Careful when doing scalability tests

Image © Intel
Apparent scaling: 75% of ideal
Real scaling: 85% of ideal
The memory wall

Example from DreamWorks hair system...
Threading Scalability

- Great scaling to 2 threads!
- Some brick wall hit for thread 3 and 4
- Looking at multiple runs, scalability peaks at 2.7x for 4 threads
Observed Bandwidth using SNB IMC events on 50K hairs (working set 213MB) Serial code
Thread affinity

Affinity faster, more consistent.
More important with more cores, NUMA
NUMA: 4 socket system

NUMA effects much more severe, taskset more critical
High thread count dropoff easier to hit
Dual socket Haswell
The villain: QPI snoop (!)

The QPI Snoop Configuration setting will control how cache snoops are handled.

When using the “Early Snoop” option the snoops will be sent by the caching agents; this will provide better cache latency for processors when the snoop traffic is low.

The “Home Snoop” option will cause the snoops to be sent from the home agent; this provides optimal memory bandwidth balanced across local and remote memory access.
Moral

Need to check machine configurations carefully

Things keep getting more complicated

More variables to optimize within power constraints

Don’t assume BIOS is set up optimally for your needs
Limits of scalability
Performance with newer CPUs

More cores (16->20->28). Lower clock (3.1->3.0->2.6GHz)
Haswell 28 vs 36 cores, 1 rig
Haswell 28 vs 36 cores, 8 rigs
Haswell 28 vs 36 cores

Do you take 10% more clock or 20% more cores?

 Depends on workload scalability
Other Siggraph 2015 presentations

XBB (Transform Building Blocks)
http://www.slideshare.net/IntelSoftware/dreamworks-animation-51882186

SBB (SIMD Building Blocks)
http://www.slideshare.net/IntelSoftware/dreamwork-animation-dwa
THANKS!

@multithreadvfx

www.multithreadingandvfx.org

“Multithreading for Visual Effects”