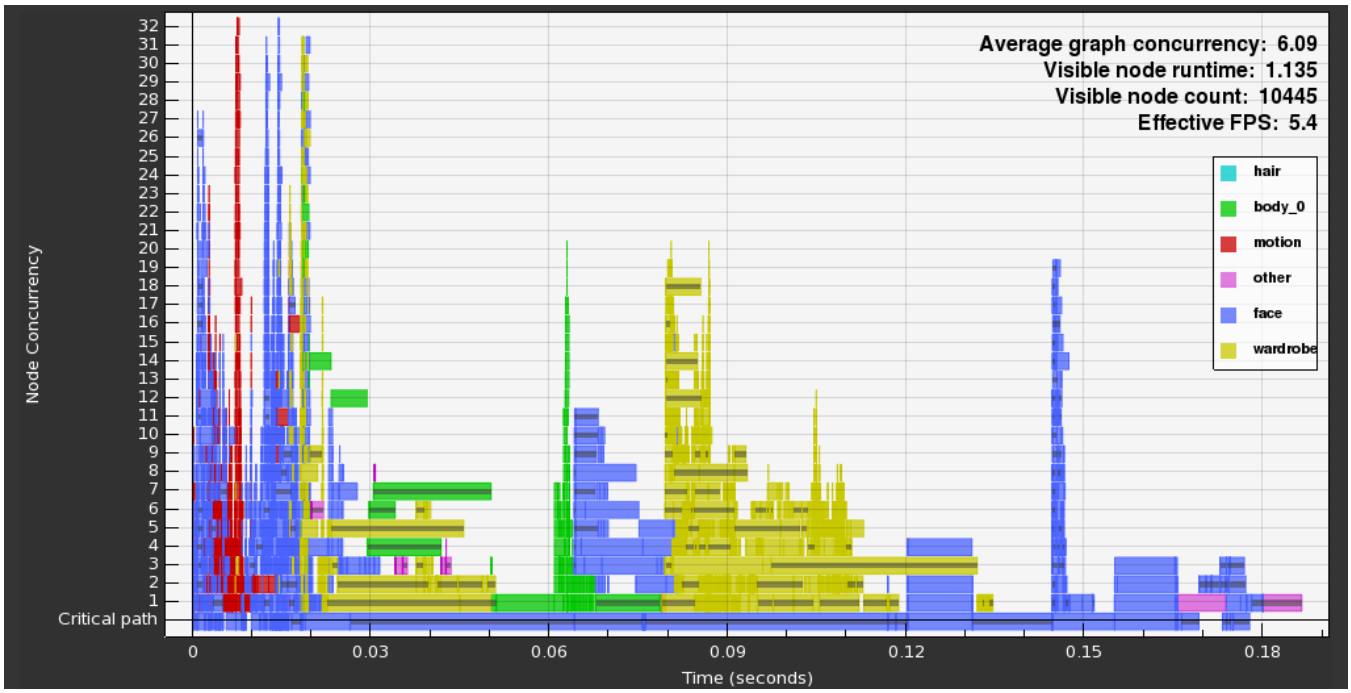
Parallel evaluation of character rigs using TBB

Martin Watt
Dreamworks Animation



Visualization of parallel evaluation of a single frame of animation for a hero character on a 32 core machine. The vertical axis shows concurrent nodes in flight for various parts of the character. Horizontal axis is time. Colors indicate different character components, as indicated in the legend. Each block is a single node in the graph. Note that parallelism within nodes is not shown here, so the true scalability is better than it appears. Nodes with internal parallelism are displayed with a horizontal bar through them.

Introduction

Computer-generated characters are central to an animated feature film and need to deliver appealing, believable on-screen performances. As such, character rigs continue to expand in complexity (for example higher fidelity skin, clothing and hair). This leads to growing system demands as animators wish to interact with complex rigs at interactive frame rates. Since single threaded CPU performance is no longer increasing at previous rates, we must find alternative means to improve rig performance. This paper focuses on multithreading our animation software and, in particular, our character setups to take advantage of multicore CPU architectures.

We have developed a new dependency graph (DG) evaluation engine called LibEE for our next

generation in-house animation tool, Premo. LibEE is designed from the ground up to be very heavily multithreaded, and is thus able to deliver high performance characters for use on computer-animated feature films.

A heavily multithreaded graph requires not just that individual expensive nodes in the graph are internally threaded, but that multiple nodes in the graph can evaluate concurrently. This raises numerous concerns. For example, how do we ensure the cores are used most effectively, how do we handle non-threadsafe code, and how do we provide profiling tools for production to ensure character graphs are optimized for parallel evaluation?

This document discusses the motivation, design choices and implementation of the graph engine itself, developer considerations for writing high performance threaded code, including ways to optimize scalability and find and avoid common threading problems, and finally covers important production adoption considerations.

Motivation

Our current generation in-house animation tool has been in use for many years at DreamWorks Animation. With each animated film, our filmmakers raise the bar in terms of the characters' on-screen performances and therefore in terms of the computational expense required in the character rigs. More expensive evaluation has partially been addressed over the years by taking advantage of the 'free lunch' associated with the increased performance delivered by each generation of processors. However for the past few years, processor architectures have been constrained by basic physics and can no longer provide significant increases in core clock speed nor from increasing instructions per clock. Scalable CPU performance improvements are now being delivered by offering CPUs with multiple cores.

Given the single threaded nature of our current generation tool, we were not able to take advantage of multicore architectures. We had reached the point where the execution performance of our animation tool was no longer accelerating due to hardware gains, while show demands, of course, continue to increase with each film. In short, our appetite was pushing us beyond the limit of what we could deliver in our existing animation environment, and we had to change.

To retrofit multithreading into the core architecture of our existing animation tool would have been extremely difficult. Instead we embarked on a studio initiative to write a new animation tool from the ground up. A key component of this new system is a highly-scalable, multithreaded graph evaluation engine called LibEE.

The primary benefits of our system are:

- We provide a framework to deliver significant graph level parallelism for complex character rigs which can run concurrently with internal node threading. Our graph engine is optimized for the specific requirements of character animation. In contrast, typical animation applications today achieve character rig threading scalability primarily through internal node concurrency.

- Our system scales to meet the demands of feature film production. We provide details for many of the challenges encountered and solutions we delivered.

- We have built a multithreaded graph visualization tool to enable Character TDs to optimize character setups for maximum parallelism and speed. We found this tool to be an essential enabler for the creation of scalable character rigs.

We first present the graph engine architecture and design choices for character animation. We then explore the implementation details including the threading engine, graph evaluation, thread safety, and implementation challenges. We finish with a discussion of production considerations and results.

Engine Architecture

The core engine for our animation tool is a dependency graph (DG), a commonly used approach for animation systems, also implemented in commercial packages such as Maya. The basic building block of the graph is a node, which is a standalone compute unit that takes in data via one or more input attributes, and produces data via one or more output attributes. A dependency graph is constructed by binding output attributes on individual nodes to input attributes on other nodes. The dependency graph is evaluated to obtain the required outputs for display, usually geometry, when driven by changing inputs, typically animation curves. Although the principle is well known, we have made a number of design choices given our target workload of animation posing and playback, as described in the next section.

Specific requirements for character animation

Our goal is to deliver a high performance evaluation engine specifically for character animation, not a general multithreading evaluation system. As a result, our design is driven by the unique requirements of a feature animation pipeline. Below are listed some characteristics of character animation evaluation which we used to optimize our specific implementation.

No graph editing

We developed an evaluation-only dependency graph for the animation environment which does not need to account for editing the characters. This enables us to take advantage of a graph

topology that is relatively fixed. We do not need to build an extensive editing infrastructure, and we can strip out from the graph any functionality that would be required to easily modify the topology. This reduces the size of nodes and connections, leading to better cache locality behavior. We can also exploit topology coherence that we would not be able to do in an editable graph environment.

Few unique traversed paths through graph

Although the graph may contain a very large number of nodes, the number of unique evaluation paths traversed through the graph during a session is relatively limited. Typically there may be a hundred or so controls that an animator interacts with, and the output is typically a handful of geometric objects. Posing workflows involve manipulating the same sets of controls repeatedly for a series of graph recomputes. As a result, it becomes feasible to cache a task list of nodes that require evaluation for a given user-edit to the graph. Walking dependency graphs to propagate dirty state and track dependencies can be expensive, and is not highly parallelizable, so avoiding this step offers significant performance and scalability benefits.

Interactive posing

Our goal is to provide consistent interactivity within the animation environment. We have set a benchmark of at least 15 fps for executing a single full-fidelity character. Our graphs can contain thousands of nodes, all of which need to be scheduled and run in this time interval. This means we need a very low overhead scheduling system. Note that we do not expect to hit this benchmark when running a character-based simulation that must evaluate the graph multiple times through a range of frames.

Animation rigs have implicit parallelism

Typically characters have components that can be computed in parallel at multiple levels. As an example the limbs of a character can all be computed in parallel, and within each limb the fingers can similarly be evaluated concurrently. Such concurrency will vary from character to character. We have suggested that our filmmakers explore stories with herds of millipedes to demonstrate the full potential of our tool, but even with conventional biped characters we are able to extract significant levels of parallelism. A very important aspect of this however is that the rig must be created in a way that expresses this parallelism in the graph itself. This was a very challenging area, and we discuss it in more detail later.

Expensive nodes

Nodes such as deformers, tessellators and solvers are expensive, but can in many cases be threaded internally to operate on data components in parallel. We want to take advantage of this internal concurrency potential while also allowing such nodes to evaluate in parallel with other nodes in the graph. As a result we require a system that offers composability, i.e. nested threading is supported by making full use of the cores without oversubscribing the system.

No scripting languages in operators
One controversial restriction we place on node authors is that they cannot use any authoring language that is known to be low performance or inherently non-threadsafe or non-scalable. This means we do not allow authors to write nodes in our in-house scripting language, which is not threadsafe, nor do we allow Python which, while threadsafe, is not able to run truly concurrently due to the Global Interpreter Lock. Python is also significantly slower than C++. This limitation has caused some concern amongst Character TDs , who author nodes for specific show requirements, since scripted nodes are typically easier to write than their C++ equivalents. We discuss these issues later.

Mechanism to express non-threadsafety of nodes
Since all the nodes in the graph can potentially be run in parallel, we need a way to ensure we can handle potentially non-threadsafe nodes. A single non-threadsafe node can cause random unpredictable crashes. Managing non-threadsafe nodes is a very important issue, and we discuss it in more detail in later.

Threading Engine

We need to build or adopt a threading engine to manage the scheduling of our nodes. There are several upfront requirements for this engine:

- Our graphs can have up to 150k nodes for a single hero character, which we wish to evaluate at close to 24fps, so the engine needs to have very low per-node runtime overhead.

- The threading engine needs to deliver good scalability since our animator workstations has between 12 and 16 cores already, a number which will only increase in future, and we wanted to make effective use of these current and future hardware resources.

- We require a system that supports composability, i.e. nested threading, since we intend for some nodes to be internally threaded as well as the entire graph running in parallel, and want the two levels of threading to interoperate seamlessly.

We considered writing our own engine from the ground up, but that would be a very major effort in which did not have specific domain expertise, so we decided to focus our resources on the higher level architecture rather than devote effort to building the low-level components. We chose to adopt Intel's Threading Building Blocks (TBB) as our core threading library. This library offers composability, high performance and ease of use, as well as being industry proven, being used in commercial tools such as Maya and Houdini.

For the graph threading, we have a layer on top of TBB, also developed by Intel, called

Concurrent Collections (CnC) that manages task dependencies, ensures chains of nodes are run on the same core for maximum cache efficiency and provides controls over scheduling. Several Intel engineers on the CnC project worked with us to optimize this library for our specific requirements.

Since we have threading at the graph and node level, and node authors can potentially call any code in our studio, we needed to ensure that all this threaded code worked well together. As a result we made a studiowide decision to adopt TBB as our threading model globally for all new code we write. We also retrofitted existing code to use TBB where possible. This ensures node authors can safely invoke other studio code from their nodes and have the threading interact well with the graph level parallelism.

Graph evaluation mechanism

DGs are typically implemented as two pass systems. In the first pass, input attributes to the graph are modified and marked dirty. This dirty state propagates through the graph based on static dirty rules which define the dependency relationship between attributes within a node. Then a second pass occurs for evaluation, where the application requests certain outputs from the graph, such as geometry. If the node that outputs the geometry has the relevant attribute marked dirty, it will re-compute itself. If the node has inputs that are dirty it will recursively re-evaluate those input nodes, potentially triggering large sections of the graph to re-compute.

The second evaluation step is a challenge for threading since recursive node evaluation limits potential scalability. We have chosen to modify this evaluation model. In the evaluation pass we traverse the graph upstream to decide what to compute but, rather than computing the nodes right away, we instead add them to a 'task list' along with the dependencies between the tasks. Then we add a third pass where those tasks are evaluated by the core engine, extracting all potential parallelism given the dependencies in the list.

Although this approach provides good concurrency, it does mean that we can potentially over-compute, since some nodes may only know the required inputs once they are inside their compute routine. In our case we will compute all inputs that could potentially affect the given dirty output given the static dirty rules, even if the inputs are not actually required based on a dynamic analysis of the node state during evaluation. This can be a problem for 'switch' nodes that can select between for example low and high resolution geometry based on an index control. To address this problem, we do an additional one or more passes over the graph to evaluate the inputs to such nodes. If the switch index changes, we prune out the unwanted code path before doing our main evaluation pass. Since switch indices change rarely during an animation session, the pruning pass has minimal overhead in graph evaluation.

Node threadsafety

Ideally all nodes in the graph would be fully threadsafe and we could simply allow the scheduling engine to assign tasks to cores as it sees fit. In practice we always need to allow for the possibility of non-threadsafe nodes, as in the following examples:

- An author wishes to prototype an operator for testing and does not want to worry about making the code threadsafe right away.
- The author is not sure if the code they are calling is threadsafe, and wishes to err on the side of caution until it can be fully validated.
- The code calls a library or methods known for sure to be non-threadsafe.

For these cases we provided a mechanism where the author can declare that the node is not fully threadsafe, and the evaluation mechanism can take appropriate precautions in the way it evaluates the node. The following potential levels of threadsafety are possible:

Reentrant
The node can be evaluated concurrently with any other node in the graph and also the same instance of the node can be evaluated by more than one thread concurrently.

Threadsafe
The node can be evaluated concurrently with any other node in the graph but the same instance of the node cannot be evaluated by more than one thread concurrently.

Type Unsafe
The node cannot be evaluated with other instances of the same node type (e.g. a node that works with internal static data).

Group Unsafe
The node cannot be evaluated with any of a group of node types (i.e. nodes that deal with same global static data or third party closed source libraries).

Globally Unsafe
The node cannot evaluate concurrently with any other node in the graph (i.e. calls unknown code, or user is just being very cautious).

In practice the only categories we have needed in production are Threadsafe, Type Unsafe and Globally Unsafe. Our design does not allow the same instance to be evaluated concurrently, so the Reentrant category is not required, and the Group Unsafe category was considered too

difficult to maintain. For the latter case we simply default to Globally Unsafe.

If the scheduler sees a node is in a category other than Threadsafe it will ensure that node is run in such a way that it will not encounter potential problems by using an appropriate lock. A Type Unsafe node will not run concurrently with another node of the same type by using a lock specific to that node type, and a Globally Unsafe node will not run concurrently with any other node by using a global lock. Of course these latter states can severely limit graph scaling. Our goal is to have as few nodes as possible that are not threadsafe. In production rigs we have been able so far to be able to run characters where almost every node is marked as Threadsafe with just one or two exceptions over which we have no control. One example of such a node is an fbx reader node, since the fbx library itself is written by a third party and is not yet (as of 2013) threadsafe. Thus we mark it as Type Unsafe since it cannot run concurrently with other fbx reader nodes, but can run concurrently with other nodes in the graph.

Code threadsafety

The studio has a large existing codebase dating back 20 years, and significant parts of the codebase are not threadsafe. Since node authors can potentially call anything in our studio codebase, pulling in non-threadsafe code is a concern. In addition, any new code can introduce threadsafety issues. One challenge is that, given the graph level parallelism, every author needs awareness of threading even if they are not explicitly writing threaded code themselves. This is a real challenge given the varying level of expertise among node authors and given that node development spans both R&D and production departments.

When we started this project, this was probably the largest area of concern we had. There was a very real risk that the project might fail because of the level of threadsafety required and potential for a continuous stream of hard-to-find threading bugs and regressions. As a result we proactively set up a significant effort to make and keep our code clean, as discussed in some of the items below.

API layer
We wrote an API for our studio code that provided one layer of insulation from direct access to our studio code. This is partly to ensure users do not get hold of raw memory since the design of the graph requires the graph itself to do memory management. However we can also use this API layer to block access to classes and methods that are known to be non-threadsafe. We endeavour to keep this API complete enough that developers will not need to bypass it and obtain raw access to external data structures, although in some cases that is still required.

Unit tests
We require rigorous unit tests to be written for nodes. We also provide an automated wrapper testing framework that takes regular unit tests and automatically runs those tests in parallel. Our

goal is to continuously run stress tests to flush out intermittent threading bugs. Over time the number of failures in these tests has dropped, although it has not reached zero. Instead it has reached a low level where 'background noise' eg hardware problems, resource issues eg full disks, cause as many crashes as threading bugs. Because threading bugs are almost by definition hard to reproduce, remaining bugs are the ones that happen very rarely and so are the most difficult ones to track down. It is highly likely that threading bugs still persist in the code, and new node authoring means there is likely to be introduction of new threading problems, but they are now at a level that crashes from threading are fewer than crashes from other parts of the application, and are not a major productivity drain.

Threading checker tools

We have used tools to check for race conditions, for example Intel's Parallel Inspector. This tool can be very useful in catching threading problems. The main challenge is the number of false positives reported by this tool when running large workloads, which can make tracking down the actual bug very challenging.

Compiler flags

There are some very useful compiler settings with the Intel compiler that can warn for access to static variables, which is a common source of race conditions. The warnings, with example code, are as follows:

```
static int x=0;
if(x>1) {}; // warning #1710: reference to statically allocated variable
x = 1;      // warning #1711: assignment to statically allocated variable
int* p=&x;  // warning #1712: address taken of statically allocated variable
```

1712 currently triggers in a lot of system-level code that cannot be avoided. 1710 triggers for stream access like cout calls, which are also relatively common in our code. As a result, we choose to use only the 1711 warning, which is also the most useful of these warnings. Example:

```
int x = 0;
int main()
{
    x++;
}

>icc -c -ww1711 test0.cc
test.cc(8): warning #1711: assignment to statically allocated variable "x"
      x++;
      ^
```

We enable this warning in our build environment, and add macros that allow a user to disable the warning if it is considered harmless. So:

```
#define DWA_START_THREADSAFE_STATIC_WRITE     __pragma(warning(disable:1711))
#define DWA_FINISH_THREADSAFE_STATIC_WRITE    __pragma(warning(default:1711))
```

```
#define DWA_THREADSAFE_STATIC_WRITE(CODE)     __pragma(warning(disable:1711)); CODE; \
__pragma(warning(default:1711))
```

and example usage:

```
static atomic<int> x = 0;
DWA_THREADSAFE_STATIC_WRITE(x = 1; ) // threadsafe since x is atomic variable
```

or:

```
static atomic<int> x = 0;
DWA_START_THREADSAFE_STATIC_WRITE
x = 1; // safe since x is atomic variable
DWA_FINISH_THREADSAFE_STATIC_WRITE
```

Ultimately we hope to turn this warning into a global error setting so any new usage will cause builds to fail. However we are not yet at the point where existing warnings have been fixed, so we enable this warning-as-error on a per-library basis as libraries are cleaned up, rather than globally.

Note that this flag is only available with the Intel compiler, which we use for our production releases. We also build with gcc as a validation build, and for that compiler we redefine the above macros to be no-ops. We found this flag useful enough to recommend using the Intel compiler just for this purpose, even if the final executables were not released built with this compiler.

The Kill Switch

The one saving grace for threading bugs is that they can be 'fixed' by disabling threading in the application. If we can narrow down the problem to a specific node we can switch that node to be Unsafe until the problem is tracked down. In the extreme case the user has a runtime option to simply globally disable threading in the graph. This will of course severely affect performance, but the animator will at least be able to continue working. Such a control is also useful to determine if problems in the application are threading-related. If a bug persists after threading is disabled, it is clearly not a threading bug.

We would highly recommend all parallel applications feature a global ‘kill’ switch that allows all threading to be disabled, if for no other reason than a self defense mechanism for multithreading authors. As any intrepid multithreading developer will know, as soon as threading is introduced into an application, all bugs instantly become threading bugs. Having the global kill switch allows developer to fend off such attacks, as in the following (slightly) fictional account:

Developer 1: *My code is crashing. I suspect it may be your multithreaded code that may be to blame, since my code appears to be flawless.*
Developer 2: *Did you test by turning off all threading in the app using the kill switch we told everyone about?*

Developer 1: *No. Let me try that.*
[pause]
Developer 1: *The problem is still there. I guess it is not a threading bug after all.*

Code scalability

Authoring parallel loops

The standard TBB parallel for loop is somewhat complex to author. Newer versions of TBB support a simpler interface using lambda functions, however the syntax of a lambda can still be intimidating, particularly to Character TDs who we want to encourage to write parallel code. We adopted some simple macros to allow easier threading of parallel loops. An example macro is shown below;

```
#include <tbb/parallel_for.h>
#define DWA_PARALLEL_FOR_BEGIN(VARTYPE, VARIABLE, MINVAR, MAXVAR, STEPSIZE)     \
    tbb::parallel_for(MINVAR, MAXVAR, STEPSIZE, [&] (VARTYPE VARIABLE)

#define DWA_PARALLEL_FOR_END                                                    \
    );
```

Then the developer can express their parallel loop as follows:

```
    // transform all positions in array in parallel by xform
    DWA_PARALLEL_FOR_BEGIN(unsigned int, vertIdx, 0u, numVertices, 1u) {
        outPositions[vertIdx] = positions[vertIdx] * xform;
    }
    DWA_PARALLEL_FOR_END
```

One benefit of using a macro like this is that we can simply disable threading by redefining the macro to implement a regular for loop. Another benefit is that we can define alternative behavior, eg record timing information as shown below by adding code to the macro to have each parallel loop print out a report showing the execution time, and line number, as shown below,.

```
#define DWA_PARALLEL_FOR_BEGIN_TIMING(VARTYPE, VARIABLE, MINVAR, MAXVAR, STEPSIZE)  \
    int tbb_loop_trip_count = (MAXVAR - MINVAR) / STEPSIZE;                         \
    START_TBB_TIMER                                                                 \
    tbb::parallel_for(MINVAR, MAXVAR, STEPSIZE, [&] (VARTYPE VARIABLE)

#define DWA_PARALLEL_FOR_END_TIMING                                                 \
    );                                                                              \
    STOP_TBB_TIMER(__FILE__, __LINE__, tbb_loop_trip_count)
```

We can use code like this to decide on a threshold above which parallelism is worthwhile.

We have other versions of this macro that support the TBB grain size concept, which is the smallest piece into which the problem range will be decomposed. Allowing the problem to be

subdivided too finely can cause excessive work for the scheduler, so we choose a reasonable smallest unit that still allows significant concurrency while maintaining a reasonable amount of work. We can use the timer code to decide on a good grain size to choose for each loop.

Unfortunately TBB does not yet have a version of the parallel_for lambda function that works with grain size, so right now it is somewhat cumbersome to code this case.

Overthreading

Any new discovery initially leads to overuse before a happy medium is found. We encountered the same with multithreading. Once developers learned about it, some were keen to use it wherever possible, and we found that even code like the following was in some cases being expressed as a parallel loop:

```
for (int i=0; i<4; i++)  x[i] = 0;
```

Clearly in this case the overhead of threading the loop vastly outweighs the benefits achieved. The rule of thumb is that invoking a parallel region costs 10k clock cycles, so unless the work in the loop is significantly greater than this, the benefits of threading are not there. It can be hard to apply that threshold to real code - we recommend timing the loop with and without threading at different trip counts to decide if it is worthwhile to parallelize it.

Threading fatigue

The initial excitement of a large speedup from parallelizing a loop can quickly pall when the inevitable threading bugs are encountered and production deadlines loom, as the following email from the Character TD trenches attests to:

“This whole foray into DWA_PARALLEL_FOR has been a pretty miserable experience. There's just nowhere near enough people or docs in support of it, and it's just a bit too bloody complex a task when I can't stay concentrated on it due to production demands.”
- Anon CTD.

Clearly if we are to ask developers to author parallel code, we need to provide and maintain a support system for them.

Indirect code optimization benefits

One factor that is often encountered when threading is applied to code, is that the serial version of the code is also optimized, as the code is cleaned up and reorganized for threading. We found this to be true in our case as well, as for example:

For the record, our final results are approximately 50x speed up for the complex case over the pre-optimization version. I wish I could say that was all or mostly parallelization, but the truth is, it was about 1/3 parallelization and about 2/3 basic cleanup work.
- Same Anon CTD.

Thread-friendly memory allocators

There are a number of thread-friendly allocators which hold per-thread stack memory so that malloc calls do not go to the global heap (and therefore are blocking calls). We have chosen to use TBB's allocator as we have found it delivers very good performance for our needs, giving us a 20% performance boost over regular malloc. Note that there is no requirement to use the TBB allocator in conjunction with TBB, nor do the versions need to be in sync. We are using TBB 4 with the TBB allocator from TBB 3, since we find we get better performance for our specific needs with the older allocator.

Note that a thread-aware allocator will in general consume more memory than regular malloc as it holds onto large blocks of memory per thread to avoid going to the global heap where possible. We find the TBB allocator increases our memory footprint by around 20% over usage of regular malloc, a tradeoff that we consider acceptable.

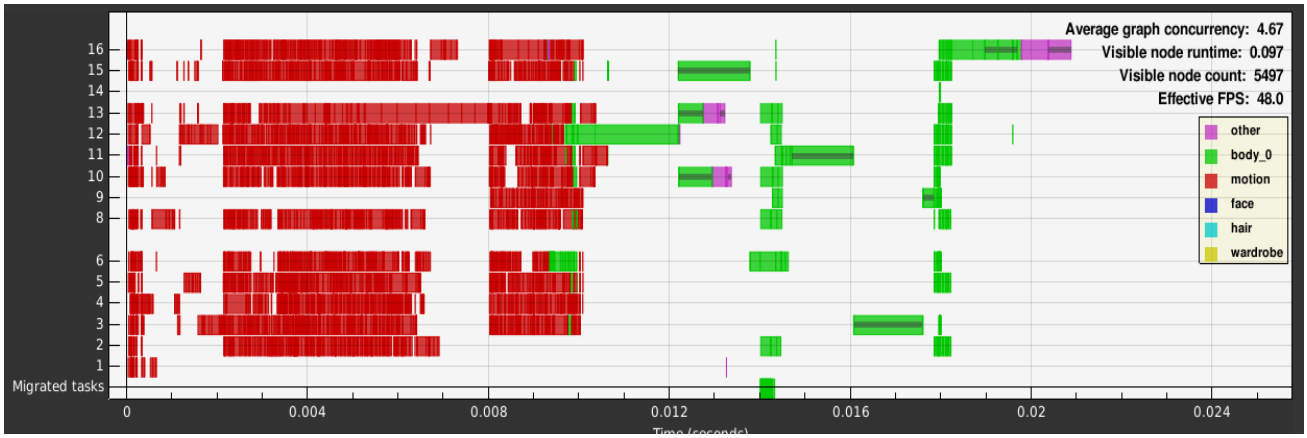
Tasks and nodes

In general the scheduling system treats each node as a separate task to be scheduled by the TBB task scheduler. This ensures we get maximum possible scaling by extracting all the potential parallelism from the graph.

Cache reuse - chains of nodes

One exception to the above is that the scheduling system treats chains of nodes (nodes with a single input and output, all connected together) as a single task for scheduling purposes, since there is no potential scalability benefit to treating each node as a separate task for the scheduler. This reduces the number of tasks, reducing scheduling overhead. Since TBB tasks are assigned to cores, this also has the benefit of ensuring chains of nodes all run on the same core, improving cache reuse. Note that it is possible for the OS to migrate a task between cores during evaluation. In practice we found that tasks are short enough that this very rarely happens. In a graph with 10k nodes, we find at most one or two tasks that are moved to a different processor core to the one they start on.

We investigated treating 'hammocks' - where there is a fan-out from one node to several chains and then a fan-in again to a single node - as a special case, but found the relative benefit to be far smaller than for chains, and the management complexity significantly larger, so did not pursue this further.



Tasks plotted against processor Id. Note that chains of nodes compute on the same processor, maximizing cache reuse. At 0.014s there is one node on the bottom row, indicating it started on one processor and was migrated by the OS to another processor before completion. This is bad for cache reuse, but note that it is just one node that showed this behavior out of a total of over 5000 nodes. Note also that no tasks we allocated to processor 7 because another long running process was occupying that core when this run was performed.

Hardware considerations

Power modes

The workloads in parallel graph execution are very spiky *(see xosview screenshot, or TV cores view).* This can mean individual cores can be switching between busy and idle states very frequently on sub-millisecond intervals. We found the power saving modes in Sandy Bridge processors to switch too slowly into high performance mode for our needs, which meant that the processor was often in a low clock speed mode for a significant portion of the time it was doing heavy compute. Because of this we chose to enable performance rather than power-saving modes on our systems to keep the processors running at their full clock speeds all the time. This provided a 20% performance boost over power saving mode, with the obvious downside of greater power consumption. It is to be hoped that future revisions of processor hardware can enable faster transitions between power saving and full speed modes. (Note that enabling full performance mode is not necessarily trivial as there are OS as well as BIOS settings that need to be coordinated for maximum performance to be achieved.)

NUMA

The workstations we use are dual socket systems, therefore NUMA is an area of concern. Currently we do not actively allocate memory on the banks nearest to the cores doing the work, as TBB can assign work to arbitrary cores. We have not yet found this to be a significant performance issue, although it is something we are monitoring closely. Experiments with 4 socket systems have shown this becomes a real issue on such topologies, since memory can be multiple hops away from the cores where the compute is being done. However at this time we are using only 2 socket systems, for which the overhead has mostly not been a major issue.

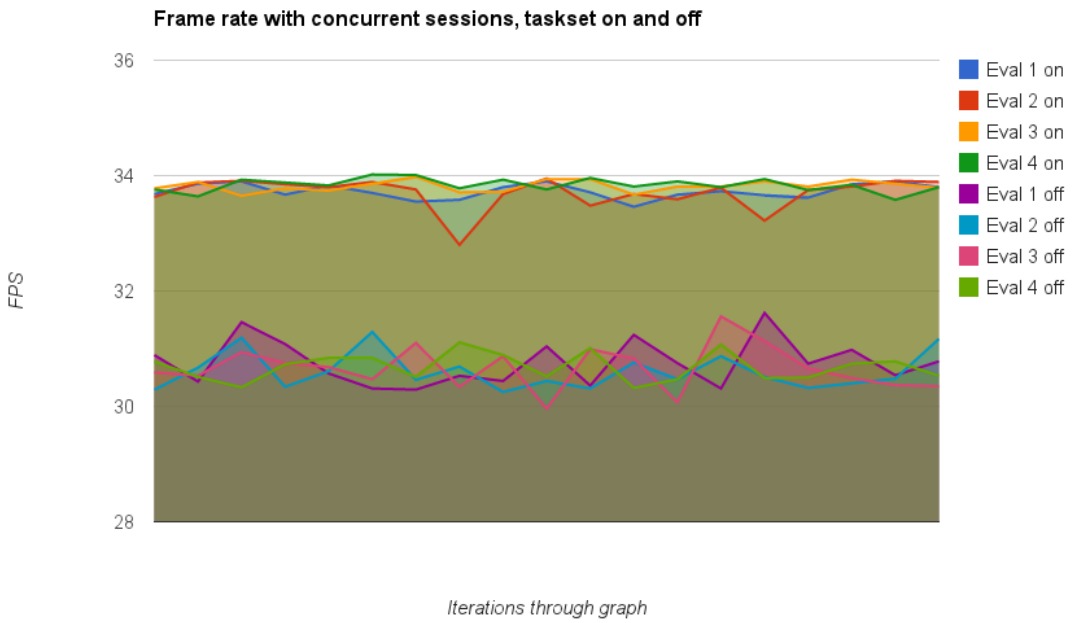
Other processes running on system

Since the application is an animation tool which is primarily running live rather than in batch mode, and which has the users primary attention, we have found that users do not often run multiple concurrent animation sessions. However it is common for artists to be running other applications on their systems, for example a web browser playing cute cat videos. This can make heavy use of one or two cores. Since TBB assumes it has the full machine at its disposal, this could potentially lead to choppy behavior as the machine cores are oversubscribed.

In our testing so far, we have found it is other applications that experience lower performance when the animation system is running, eg streaming video in a browser can become choppy. In this case we may just be lucky, and we have added a control over the total thread count being used in the application by TBB, so we can in principle dial back the number of threads used by the graph if the machine is being overloaded. As core counts increase, the relative loss of one or two cores becomes less and less significant.

If we wish to limit the number of cores being used, we have found it useful to set CPU affinity to

bind the process to particular cores. Below is a figure showing a graph running with 4 threads with and without CPU affinity set (via the taskset command on Linux).



Evaluation of graph on 4 cores with and without CPU affinity set. The bottom set of four runs show the frame rate over time without affinity set, the top four runs show the performance with affinity set. Note that not only if overall frame rate ~10% higher with affinity set, the frame rate is also much more consistent between frames, leading to a better animator experience.

Production Considerations - Character scalability

The above relates primarily to the R&D work of building a parallel engine. Equally important are the production considerations that relate to the use of this engine. Integrating the new dependency graph engine into a feature animation production requires a number of significant changes to the character rigging process, namely:

- The character rigs need to be re-architected to integrate into the new animation tool and the engine.
- All custom nodes have to be written in C++ instead of in scripting languages like Python.
- Character systems need to be optimized for parallel computation.

We discuss the challenges and efforts surrounding these three production changes in the following sections.

Character Systems Restructure

We realized that our underlying character systems needed to be re-architected to work in the new interactive animation environment built on top of the graph engine. Most of the graph nodes used to construct our character setups had to be re-written to be threadsafe and to integrate with the engine's interfaces. In revamping our character systems, many existing nodes were discarded and new ones with different requirements were written in their place. Given that our code base has been written over a span of many years, such a rewrite is a major undertaking.

C++ Custom Node Development - No more scripted nodes

Character TDs often write custom graph nodes to handle unique rigging challenges. Typically custom nodes were written in our proprietary scripting language or in Python for ease of authoring. These nodes are relatively simple to write and integrate nicely into our production codebase. As mentioned earlier, the graph evaluation engine requires all nodes to be compiled C++ code for optimum performance. Furthermore, these nodes are strongly desired to be threadsafe. This transition has been difficult for production, although clearly necessary.

To help streamline node authoring, we implemented a high-level API layer for our studio codebase. This provides node authors with a safe and consistent interface to our studio code libraries. In this API we can ensure we expose only threadsafe code, or provide access to unsafe methods with appropriate locking encoded within the API itself. In some cases the API may add unacceptable performance overhead, so we do allow direct access to code on the understanding that the user accepts the risks involved. Wrapping all studio code is unfeasible, but we are attempting to maintain and extend this interface based on the code users wish to access.

We have provided training programs for Character TDs to transition from writing script-based nodes to building C++ equivalents. We have also developed a training curriculum to spread knowledge of writing threaded and threadsafe code.

Optimizing for maximum parallelism

One of the interesting transitions required, which was in hindsight obvious but not greatly considered initially, was that Character TDs had to become familiar with multithreading not just at a coding level, but also when it comes to building the rigs themselves. Dependencies between

nodes in the graph need to be expressed in a way that allows as much of the graph as possible to run in parallel. No matter how well the engine itself scales, it can only extract from a graph the parallelism that the graph authors, ie the Character TDs, have put into the system.

Previously, to optimize character setups, Character TDs used profiling tools to identify bottlenecks in individual graph nodes and then worked to address performance issues in these nodes. This approach is still necessary, but is no longer sufficient with a highly multithreaded graph since the ordering and dependency between nodes becomes a very large factor in the overall performance. There were new concepts that needed to be considered, in particular the concept of the critical path.

The critical path is the most expensive serial chain of nodes in the graph. The overall runtime of the graph is limited by the critical path runtime. A goal for optimization therefore is to try to reduce the cost of the critical path, either by removing nodes from the path or by optimizing nodes along it. A corollary to this is that there is less benefit to optimizing nodes that are not on the critical path since, to first approximation, those nodes do not directly affect the graph runtime (although of course those nodes do consume compute resources that might otherwise be used to speed evaluation of nodes along the critical path).

Since Character TDs have limited time to spend on optimization, it is important that this time be used wisely. There was significant frustration early on that optimization efforts did not seem to yield the expected speedups. This turned out to be due to optimizations being applied to expensive nodes that were not on the critical path. As a result, we found the need to write a tool which would provide information on the graph performance characteristics, including the critical path, for the Character TDs.

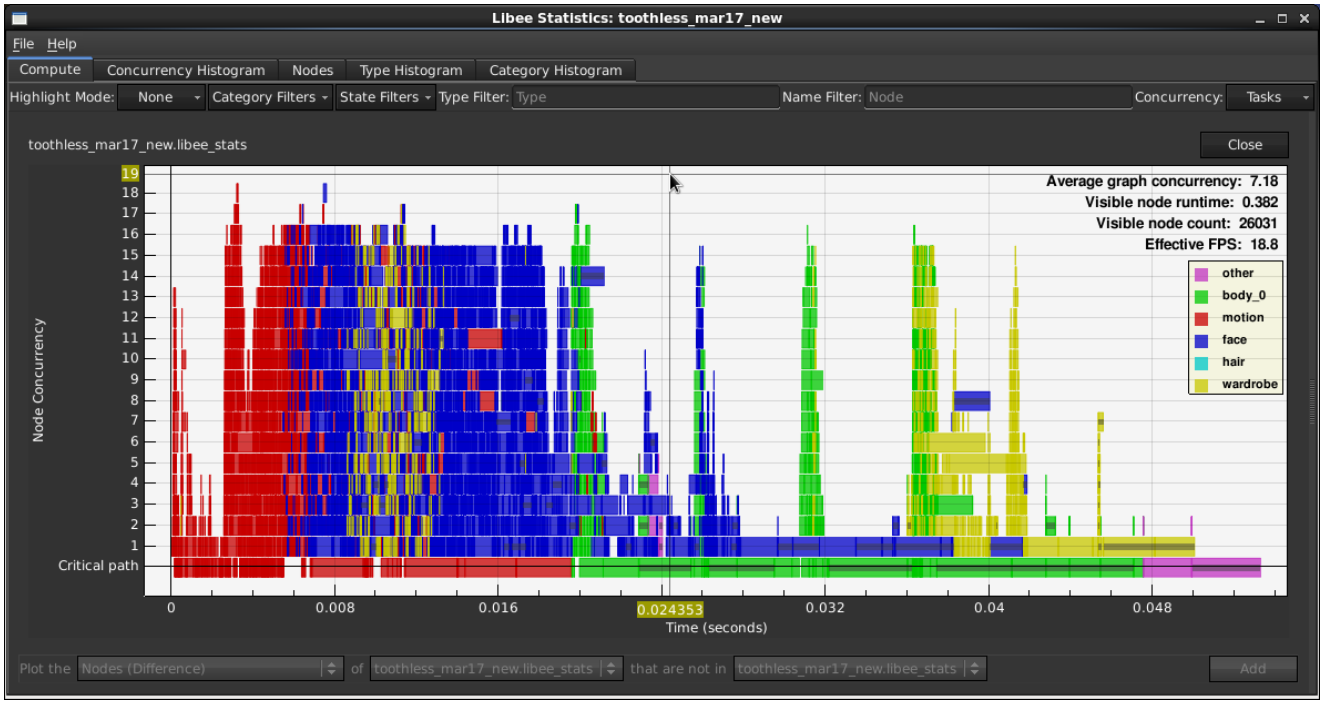
This is something that is obvious in hindsight, but which we did not anticipate, so the tool was developed relatively late, at a time when many of the characters for the first production show had already been developed and were no longer open to significant changes. This meant that characters on the first show were not as fully optimized as they could have been, but also that we expect future shows to have more highly optimized characters. Expressing parallelism in rigs is a new and very important skill for Character TDs, one that we expect them to become increasingly skilled at over the years.

Threading Visualizer tool

In this section we describe the tool that was built to show the parallelism in the graphs. We discuss this tool in some depth because this was and continues to be an extremely important part of the optimization process (and also because it allows us to show some pretty pictures which have been sadly lacking up to this point).

Building a graph that allows the engine to extract maximum scalability from the workload is a

new challenge for production artists. We have implemented profiling tools that allow Character TDs to visualize data flow and node evaluation in the graph. Artists can identify which components of their character are running in parallel, where graph serialization bottlenecks occur, and where unexpected dependencies exist between parts of the rig. The tool highlights nodes on the critical path , which determines the overall best possible runtime of the graph.



The threading visualization tool enables artists to investigate bottlenecks within the graph. The tool highlights different character rig components in different colors (i.e. body, face, wardrobe, and hair) and provides overall graph concurrency statistics. Note that the average concurrency only represents graph parallelism and does not include node parallelism. The bottom row of nodes are the nodes on the critical path.

The average concurrency metric is a very useful value as it gives a simple easily understood metric to indicate the parallelism in a rig. When comparing similar characters we expect similar levels of concurrency, and outliers attract special attention to detect potential problems in the rig that limit scalability.

Over time we are learning how to build scalable characters in this new environment, but this is an ongoing process. Here are some of the strategies we have developed to optimize multithreaded character rigs:

- Focus primarily on nodes along the critical path.
- Identify expensive nodes that are bottlenecks and internally optimize these nodes as well

as move their execution to a more parallel location within the graph.

- Identify a section of the graph that is serial and work to parallelize this area of the rig.

- Identify groups of nodes that are used repeatedly in the graph and rewrite them as a single custom node. This reduces the overall graph complexity and therefore minimizes thread scheduling overhead.

We are able to collect before/after statistics and compare them in the visualizer to give immediate feedback to the author on the benefits of their optimizations to the overall character runtime profile, as in the example below.

Mode to compare two profiles to check benefits of optimizations

We have utilized a combination of the above described optimization approaches to successfully improve performance of our character setups. Below are some real world production examples that highlight our profiling process in action.

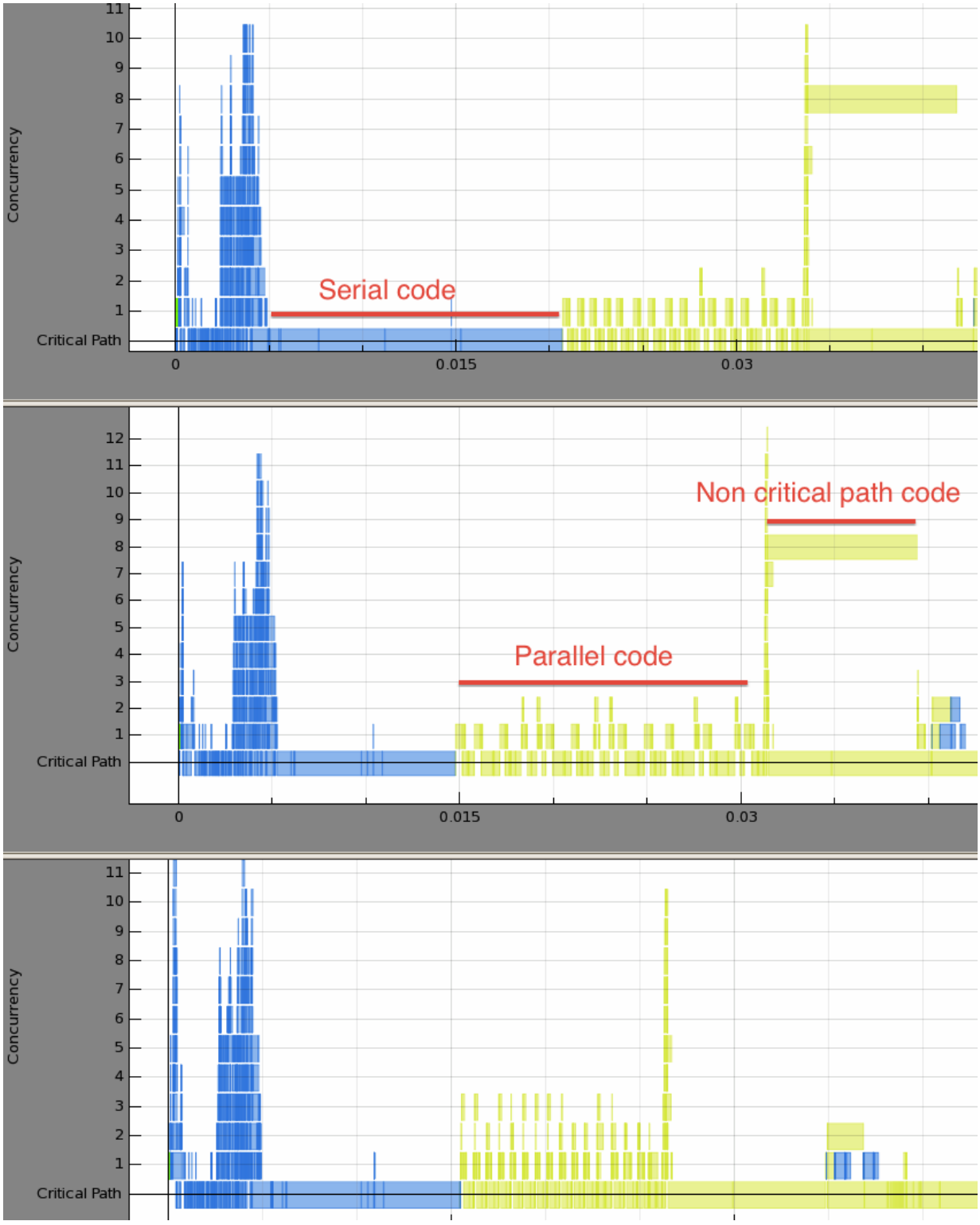
Case Study: Quadruped Claws

In this case we show the graph visualizations used to identify and fix bottlenecks within a

quadruped character. We first identify a problematic long chain of serial nodes along the critical path in blue. This represents the character's motion system for 12 claws (three claws per foot). Upon closer inspection, a node that concatenates joint hierarchies is used extensively in this chain but is not efficiently coded.

The second figure represents the rig after R&D optimized the hierarchy concatenation code. Notice that we have shortened the serial path in blue but have not changed the graph structure or parallelism. Next we look at the claw's deformation system, identified by the yellow section of the diagram. We find that while the graph shows a small degree of parallelism, the claws are still executing in serial, one after the other. Character TDs rewired the graph so that each claw deforms independently, and then they are merged together as a last step. Character TDs separately optimized an expensive node not on the critical path.

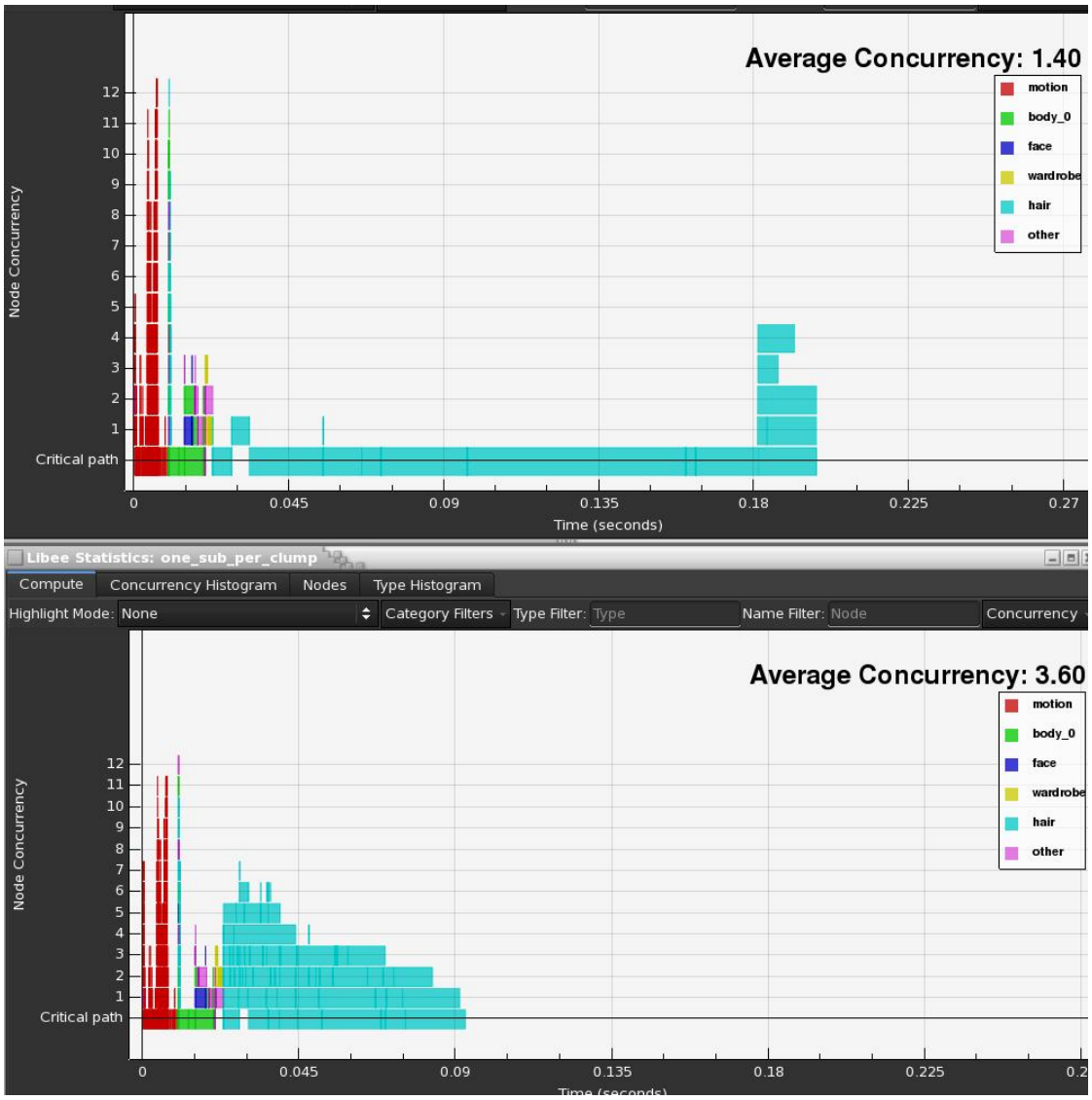
The third figure shows the results of these two character graph optimizations. The claw deformation code path in yellow has shrunk. However note that the second optimization, to the expensive node not on the critical path, did not significantly affect the overall runtime. This demonstrates the importance of focusing efforts on the critical path of the graph.



Example showing the process for optimizing a quadruped character's claws. Note that the overall time for evaluation of the frame is reduced at each stage - the above images just focus on the first part of the frame.

Case study: hair solver

In this example the Character TDs took an initial serial chain of nodes that implemented the hair system and changed to dependencies to allow different parts of the hair system to run in parallel. The same amount of work is performed, but in a much shorter overall runtime.

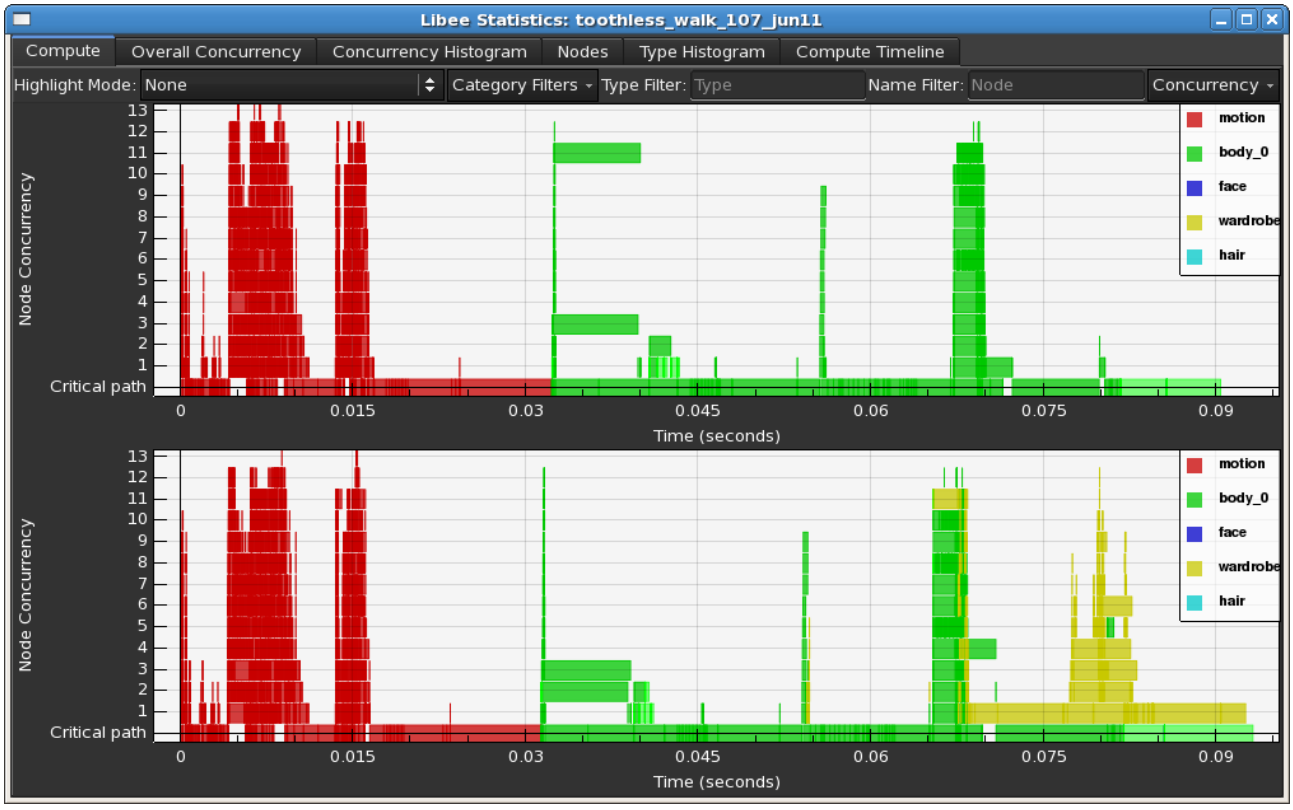


Top profile shows initial hair system implementation, bottom shows same workload with dependencies expressed between nodes in a way that allows more parallel execution

Case Study: The road to Damascus: free clothing!

This graph shows the moment when for many of us the benefits of the threaded graph evaluation

system finally became dramatically real. The top figure shows the motion and deformation system for a character. The bottom graph shows the same character with the addition of rigged clothing. This clothing is an expensive part of the overall character, but because it was attached to the torso it was able to compute in parallel with the limbs of the character. Since the critical path ran through the character body rather than the clothing, this meant that effectively the clothing evaluation was free.



Top graph shows character with motion and deformation system, bottom graph shows addition of clothing. Note that the overall runtime increases only marginally although there is significant extra work being performed in the rig.

We have been happy to see that this tool is now often open on a Character TD workstation as they are working on their rigs, the ultimate compliment showing that they now consider it an invaluable part of their character development process.

Performance results

We are hitting performance benchmarks of 15-24 fps interactive posing of complete full-fidelity characters in the animation tool on HP Z820 16 core Sandy Bridge workstations with 3.1GHz clock speeds, which are our standard deployment systems for animators. Our fps benchmarks represent the execution of a single character rig without simulations running live (which would evaluate the character graph repeatedly through a range of frames).

One question is how much benefit we get from node threading versus graph threading. The following image shows the performance of a rig with either node or graph threading disabled, and shows that most of the performance benefits come from graph level threading rather than node level threading, proving the value of the threaded graph implementation over simply threading individual nodes.

Top graph runs with node and graph threading disabled. Second graph has node threading enabled. We see a 1.35x speedup for this case. Bottom graph has graph threading enabled. We see an additional 4.2x scaling in this case, for an overall 5.7x total speedup on a hero biped character setup on a 16 core machine.

The above results have since improved, and our hero characters are now showing overall scaling of 7-8x from a combination of node and graph level parallelism in most cases, with a couple of outliers still languishing at ~5.5x. We anticipate further improvements to graph level scaling on future productions as Character TDs continue to develop expertise in parallel optimizations.

Scalability limits

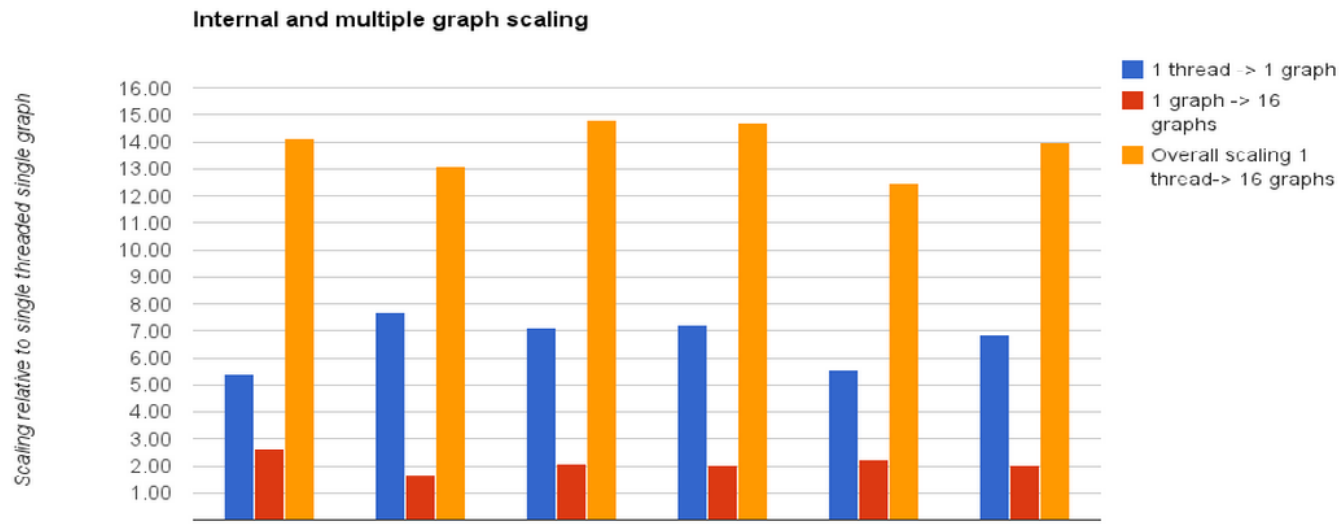
We spent significant effort in optimizing both the rigs and the nodes themselves to attempt to improve scalability. As indicated in the previous section, we are approaching 8x scaling on 16 core machines, which is a good result, but one has to ask if it is possible to do better. Core counts will only continue to increase, and a simple application of Amdahl's law tells us that 8x on a 16 core machine will only give us 10x on a 32 core machine.

We investigated possible hardware limitations, eg memory bandwidth, but this does not appear to be a factor with our rigs. Instead it appears that the scalability limits at this point are simply due to the amount of parallelism inherent in the characters themselves. We do expect to improve rig parallelism as Character TDs gain expertise, but fundamentally there is a limit to how much parallelism it is possible to achieve in a human or simple animal rig, and we appear to be close to those limits.

The figure at the beginning of this current document shows one way to improve scaling, which is to have multiple characters in a scene. This is a common occurrence in real shots, and we are finding that overall scaling is indeed improved as the complexity of the scene increases. Of course the overall runtime of the scene can still be slow, but at least the extra benefits of parallelism become more effective in such cases.

A second approach is to evaluate multiple graphs in parallel. For animation where frames are independent, we have tested workflows where we fire off frame N and N+1 as independent graph evaluations. What we find is that overall throughput increases significantly although, as expected, the latency for any particular frame to compute is increased as it has to compete against evaluation of other graphs. Nevertheless, if the goal is to process as many frames as possible in the shortest time, this approach shows considerable promise. Futhermore, the total scaling is very close to the machine limits, pushing 15x in some cases on a 16 core machine. This means not only are we using current hardware to its full extent, and are still not hitting memory bandwidth limits, but there is hope that we should be able to scale well to future machines with higher core counts.

An obvious downside to this approach is the increased memory consumption due to storage of multiple independent graph states, a significant problem with large complex rigs.



Multiple graph evaluation for six hero characters. The blue bar shows regular scaling from node and graph threading, the red bar shows the additional scaling from computing multiple graphs concurrently. Overall scaling is 12-15x on a 16 core machine.

Summary

With this new engine we have achieved well over an order of magnitude speedup in comparison to our existing in-house animation tool, and are also seeing performance significantly higher than we are able to achieve in third party commercial tools. This will allow us to dramatically increase the level of complexity and realism in our upcoming productions, while simultaneously delivering a fluidity of workflow for animators by giving them immediate real-time feedback even for very heavy production character rigs.

Implementing the parallel graph evaluation engine was a very significant effort (the project took 4 years) but the final results are proving to be very worthwhile to our animators, and we expect the tool to continue to scale in performance as rigs increase in complexity and hardware core counts rise.

One of the unanticipated requirements is that Character TDs need to develop significant new skills to be able to build and optimize character rigs in a multithreaded environment, which is a long term learning process, and providing high quality tools to enable them to do this is a critical requirement for success.