Multithreading for Visual Effects

SIGGRAPH 2015 Course Notes

James Reinders – Intel

Jeff Lait – Side Effects Software

Erwin Coumans – Google

George ElKoura – Pixar

Martin Watt – DreamWorks Animation

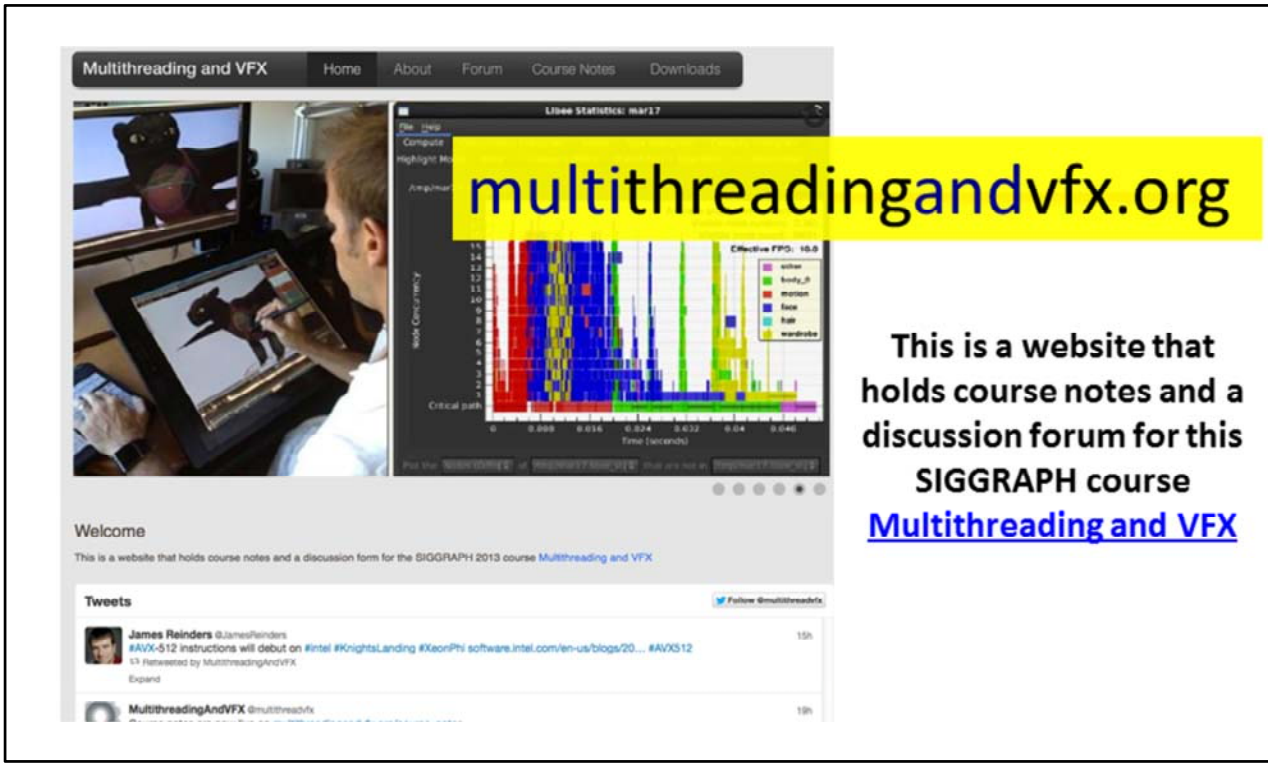
SIGGRAPH 2015 Courses, August 09 – 13, 2015, Los Angeles, CA.
ACM 978-1-4503-3634-5/15/08.

Table of Contents

Overview

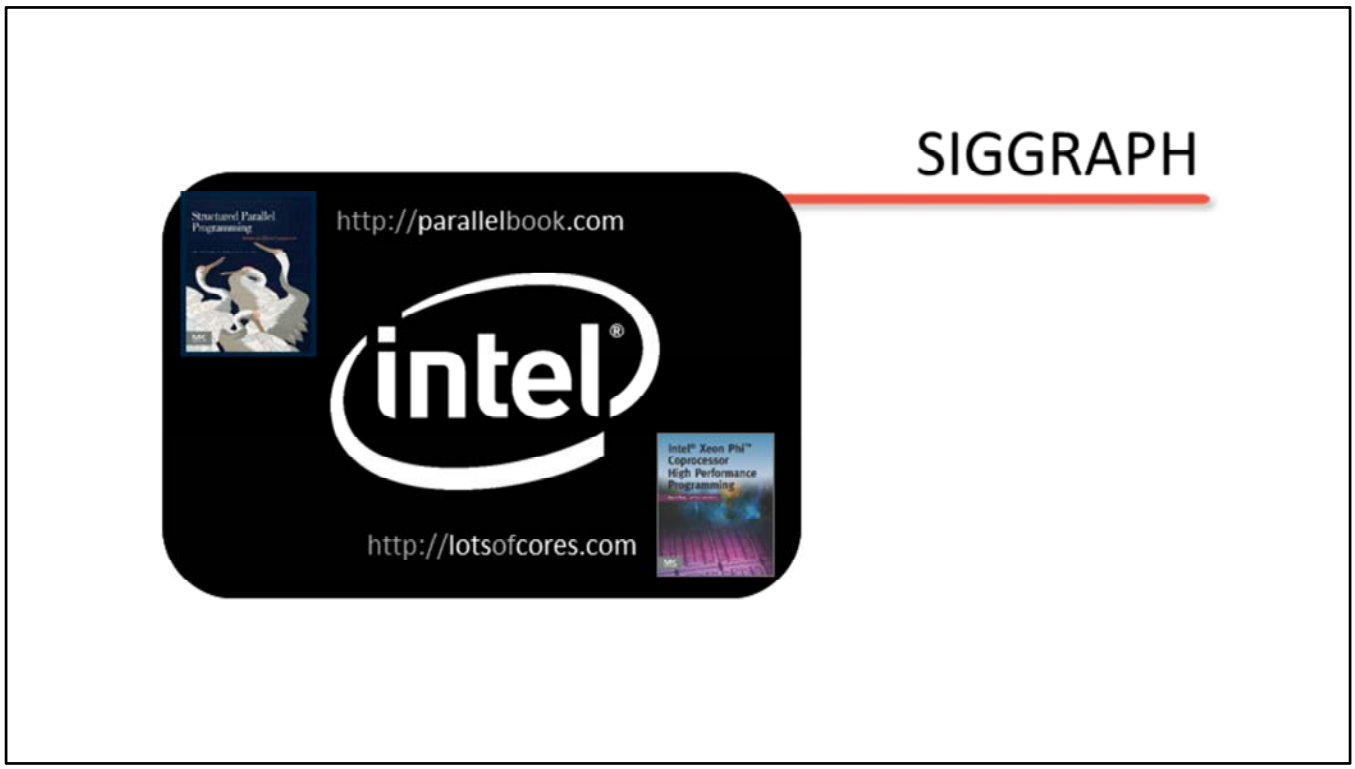
These are the course notes for the SIGGRAPH 2015 Course: Multithreading for Visual Effects. We present a variety of battle-tested approaches to multithreading in the visual effects industry.

Up to date information is available on our website: www.multithreadingandvfx.org/ and you can get the latest updates by following us on twitter @multithreadvfx.



multithreadingandvfx.org

This is a website that holds course notes and a discussion forum for this SIGGRAPH course Multithreading and VFX



Multithreading and VFX

Parallelism is important to many aspects of visual effects. In this course, experts in several key areas present their specific experiences in applying parallelism to their domain of expertise. The problem domains are very diverse, and so are the solutions employed, including specific threading methodologies. This allows the audience to gain a wide understanding of various approaches to multithreading and compare different techniques, which provides a broad context of state-of-the-art approaches to implementing parallelism and helps them decide which technologies and approaches to adopt for their own future projects. The presenters describe both successes and pitfalls, the challenges and difficulties they encountered, and the approaches they adopted to resolve these issues.

James Reinders, Director and Parallel Programming Evangelist and Senior Engineer, Intel
Parallel Programming Evangelist. James is involved in multiple engineering, research and educational efforts to increase use of parallel programming throughout the industry. He joined Intel Corporation in 1989, and has contributed to numerous projects including the world's first TeraFLOP/s supercomputer (ASCI Red) and the world's first TeraFLOP/s microprocessor (Intel® Xeon Phi™ coprocessor). James been an author on numerous technical books, including VTune™ Performance Analyzer Essentials (Intel Press, 2005), Intel® Threading Building Blocks (O'Reilly Media, 2007), Structured Parallel Programming (Morgan Kaufmann, 2012), Intel® Xeon Phi™ Coprocessor High Performance Programming (Morgan Kaufmann, 2013), and Multithreading for Visual Effects (A K Peters/CRC Press, 2014), High Performance Parallelism Pearls (Morgan Kaufmann, 2015) and High Performance Parallelism Pearls Volume 2 (Morgan Kaufmann, 2016). James is currently working on an update to the 2013 book on Intel Xeon Phi which will address the new Knight Landing product.

"Think Parallel"

- Parallelism is almost never effective to "stick in" at the last minute in a program
- Think about everything in terms of how to do *in parallel* as cleanly as possible

Parallelism is almost never something you can "stick in" at the last minute in a program
You are best off if you think about everything in terms of how to do in parallel as cleanly as possible

Of course – Usually… we ARE adding parallelism to existing programs.
Solutions like TBB, are designed with this in mind… with features like "sequential consistency" to feel at home in an program with sequential origins.

Just keep in mind that PARALLELISM needs serious consideration to be most effective… and generally is very limited if it is only "bolted on" as an after thought – or in a manner that is excessively limited in the amount of code and/or data structures which can be modified.

Motivation

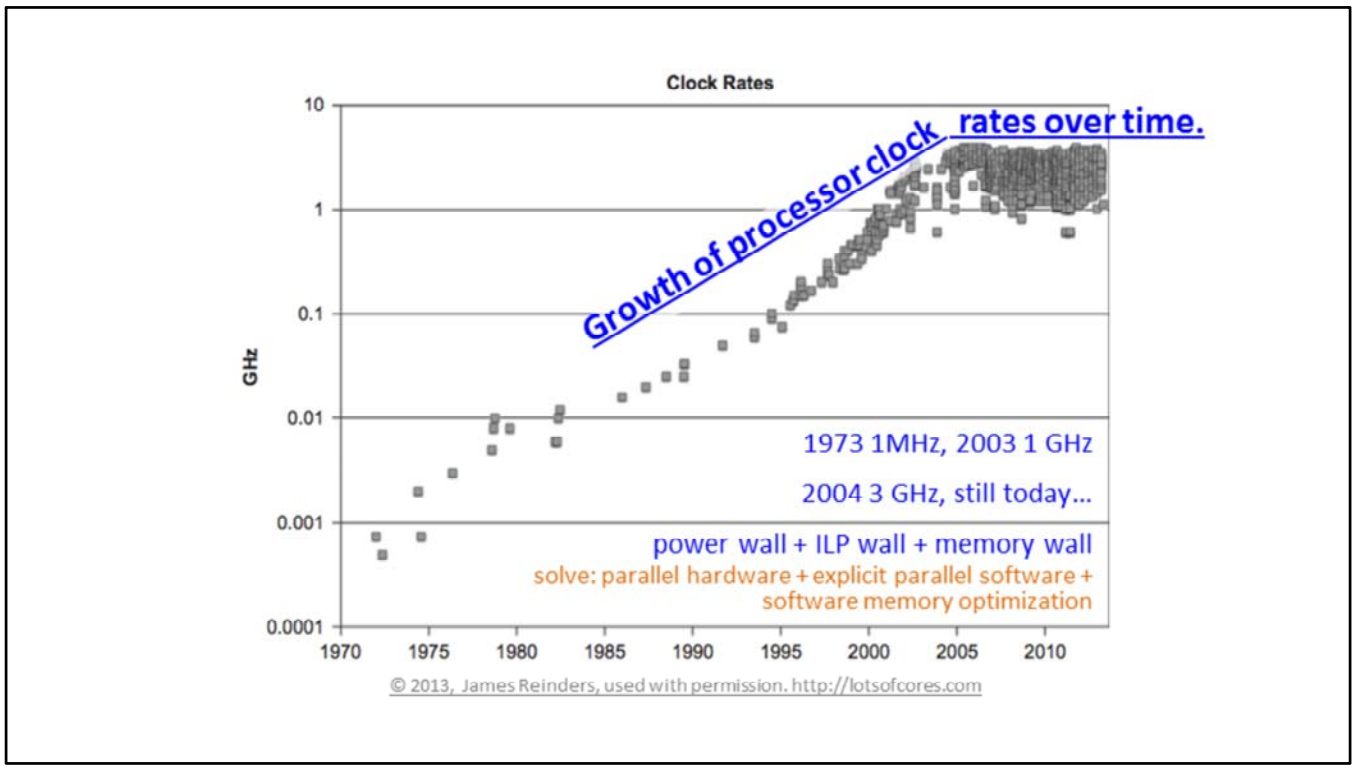
All computers are now parallel. Specifically, all modern computers support parallelism in hardware through at least one parallel feature including vector instructions, multithreaded cores, multicore processors, multiple processors, graphics engines, and parallel co-processors. This statement does not apply only to supercomputers. Even the lowest-power modern processors, such as those found in phones, support many of these features.
It is also necessary to use explicit parallel programming to get the most out of such processors.

Automatic approaches that attempt to parallelize serial code simply cannot deal with the fundamental shifts in algorithm structure required for effective parallelization.

The goal of a programmer in a modern computing environment is not just to take advantage of processors with two or four cores. Instead, it must be to write scalable applications that can take advantage of any amount of parallel hardware: all four cores on a quad-core processor, all eight cores on octo-core processors, thirty-two cores in a multiprocessor machine, more than fifty cores on new many-core processors, and beyond.

As we will see, the quest for scaling requires attention to many factors, including the minimization of data movement, and sequential bottlenecks (including locking), and other forms of overhead.

Parallel computers have been around for a long time, but several recent trends have led to increased parallelism at the level of individual, mainstream personal computers. Taking advantage of parallel hardware now generally requires explicit parallel programming.



Growth of processor clock rates over time.

This diagram shows the growth of processor clock rates over time (log scale); this diagram shows a dramatic halt by 2005 due to the power wall, although current processors are available over a diverse range of clock frequencies.

Until 2004, there was also a rise in the switching speed of transistors, which translated into an increase in the performance of microprocessors through a steady rise in the rate at which their circuits could be clocked. Actually, this rise in clock rate was also partially due to architectural changes such as instruction pipelining, which is one way to automatically take advantage of instruction-level parallelism.

An increase in clock rate, if the instruction set remains the same (as has mostly been the case for the Intel architecture), translates roughly into an increase in the rate at which instructions are completed and therefore an increase in computational performance.

Actually, many of the increases in processor complexity have also been to increase performance, even on single core processors, so the actual increase in performance has been greater than this.

Therefore, many people have confused “Moore’s Law” with a trend (true until 2004) that had performance (of single program threads) doubling about every two years. As we see on the next page, Moore’s Law continues. Claims that “Moore’s Law” some how ended or slowed in 2004 are based on the confusion of Moore’s Law with the related trend on performance rising.

From 1973 to 2003, clock rates increased by three orders of magnitude (1000x), from about 1MHz in 1973 to 1GHz in 2003. However, as is clear from this graph, clock rates have now ceased to grow, and are now generally in the 3GHz range. In 2005, three factors converged to limit the growth in performance of single cores, and shift new processor designs to the use of multiple cores.

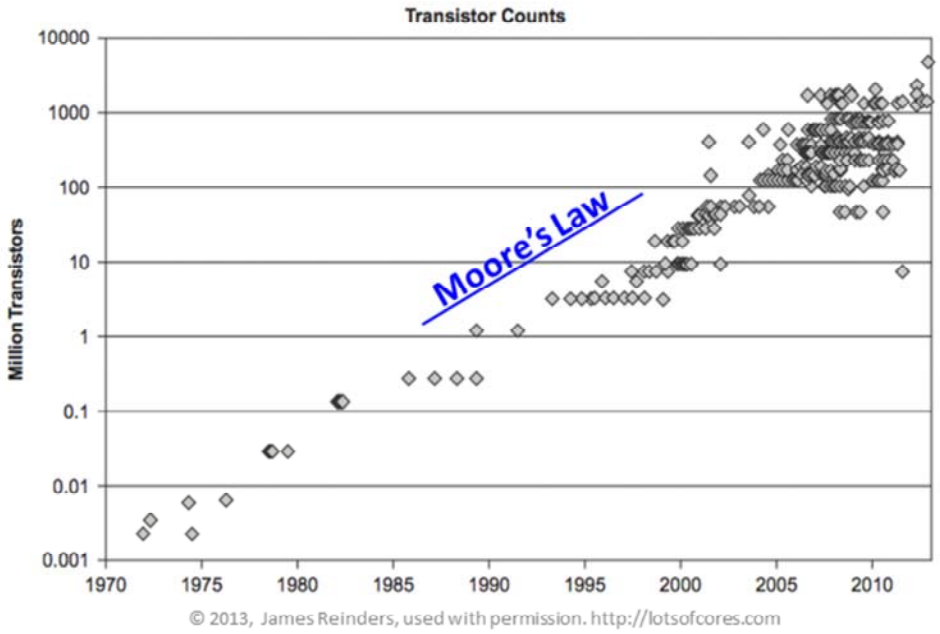
These are known as the ``three walls'':
- The Power Wall: Unacceptable growth in power usage with clock rate.
- The Instruction Level Program (ILP) Wall: Limits to available instruction level parallelism.
- The Memory Wall: A growing discrepancy of processor speeds relative to memory speeds.

In order to achieve increasing performance over time for each new processor generation, we cannot depend on rising clock rates, due to the power wall.
We also cannot depend on automatic mechanisms to find (more) parallelism in naive serial code, due to the ILP wall.
To achieve higher performance, we now must to write explicitly parallel programs. This is especially true if you want to see performance scale over time on new processors.

And finally, when you write these parallel programs, the memory wall means that you also have to seriously consider communication and memory access costs, and may even have to use additional parallelism to hide latency.
Instead of using the growing number of transistors predicted by Moore's Law for ways to maintain the ``serial processor illusion,'' architects of modern processor designs now provide multiple mechanisms for explicit parallelism.

However, you must use them, and use them well, in order to achieve performance that will continue to scale over time.

[NOTE: I made many versions of graphs based on Intel, non-Intel processors, x86-only and non-x86 architectures… and regardless of the selection the trends were the same. However the more variety I included, the harder it was to see because of the multiple lines that were essentially created for the various vendors or architectures even thought the trends for each line would be the same. Therefore, I settled on showing graphs made only of Intel x86 devices which had the continuity of same architecture and same vendor while also stretching over many decades. Regardless of vendor or architecture, I found the trends and conclusions to be the same. This is true of all the graphs I used in my books, and this tutorial.]



Moore's Law continues!

The trend on this graph, which is on a logarithmic scale, demonstrates exponential growth in the total number of transistors in a processor from 1970 to the present.

In 1965, Gordon Moore observed that the number of transistors that could be integrated on silicon chips were doubling about every two years, an observation that has become known as Moore's Law.

Consider this figure, which shows a plot of transistor counts for Intel microprocessors. Two rough data points at the extremes of this chart are 0.001 million transistors in 1971 and 1000 million transistors in 2011. This gives an average slope of 6 orders of magnitude over 40 years, a rate of 0.15 orders of magnitude every year. This is actually about 1.41x per year, or 1.995x every two years.

The data shows that Moore's original prediction of 2x per year has been amazingly accurate. While I only give data for Intel processors, processors from other vendors have shown similar trends.

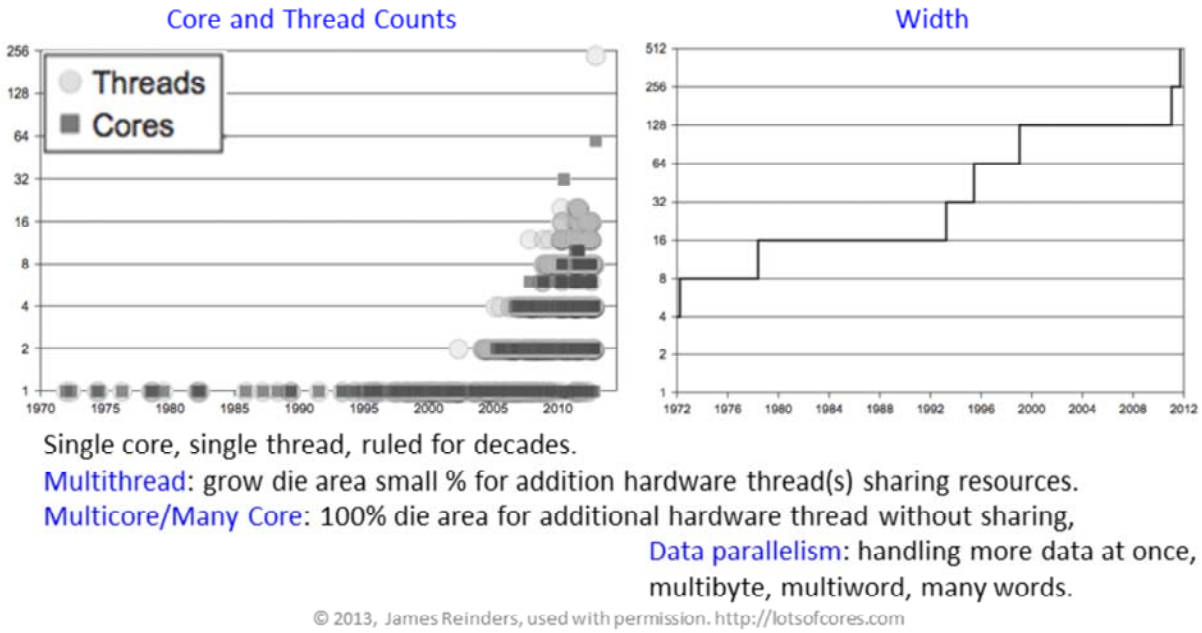
This exponential growth has created opportunities for more and more complex designs for microprocessors. The resulting trend in hardware is clear: more and more parallelism at a hardware level will become available for any application that is written to utilize it.

I also believe it greatly favors programmers over time – as the increased complexity of design will allow for innovations to increase programmability. I really do not see a future where hardware complexity can only address performance "at all costs" and ignore programmer productivity (and be a successful product commercially).

These trends affect designs of processors, GPUs and co-processors… it is not unique to processors!

The ``free lunch' of automatically faster  serial applications through faster microprocessors has ended. The ``new free lunch'' requires scalable parallel programming.

The good news is that if you design a program for scalable parallelism, it will continue to scale as processors with more parallelism become available.

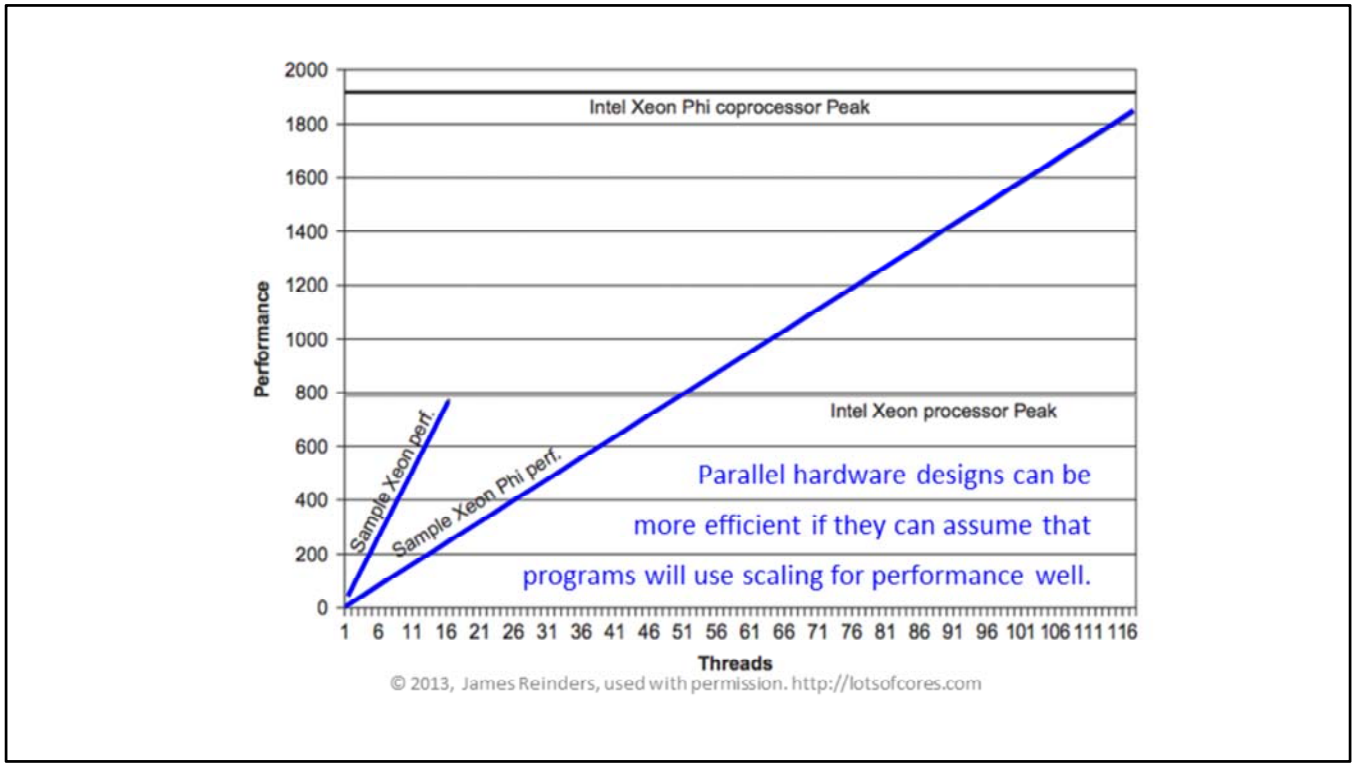


Cores and hardware threads per processor.

The number of cores and hardware threads per processor was one until around 2004, when growth in hardware threads emerged as the trend instead of growth in clock rate.
The concept of multiple hardware threads per core started earlier, in the case of Intel with hyper-threading in the early part of this century, prior to multicore taking off. Intel uses one, two or four hardware threads per core in Intel products.

Multicore and Many Core devices use independent cores that are essentially duplicates of each other, hooked together, to get work done with separate software threads running on each core. [Note: The definition of multicore and many core are the same… a device with more than one core. Intel has used Many Core to refer to "lots more than multicore"… which currently means multicore have 1-16 cores today, many core start at 57 cores/device.]

Multithreaded cores reuse the same core design to execute more than one software thread at a time efficiently. Unlike multicore/many core, which duplicate the design 100% to add support for an additional software thread, multithreading adds single digit (maybe 3%) die area to get support for an additional software thread. The 100% duplication in the case of multicore/many core can yield 100% performance boost with a second thread, while multithreading oftens adds only 5 to 20% performance boost from an additional thread. The added efficiency of multithreading is obvious when the added die area is considered.



The importance of scaling

When hardware can "assume" parallel programming at the expense of serial performance, a number of wonderful things happen in hardware design that give more compute density and power efficiency. However, the downside is a heavier reliance on scaling. This is true for GPUs and many core devices (shown is a many core device: Intel® Xeon Phi ™ coprocessor).

Task

- Key thing to know:
  - Program in TASKS *not* THREADS.

http://tinyurl.com/threadsYUCK

The "more solved" world of task parallelism
why task parallelism doesn't matter without data parallelism

If this topic interests you…
I highly recommend a paper titled "The problem with threads"
By Edward A. Lee, University of California Berkeley,
Published 05-01-2006
http://tinyurl.com/threadsYUCK

Task

- Key thing to know:
  - Program in TASKS *not* THREADS.

This means:

- Programmer's job: identify (lots) of tasks to do
- Runtime's job: map tasks onto threads at runtime

http://tinyurl.com/threadsYUCK

The "more solved" world of task parallelism
why task parallelism doesn't matter without data parallelism

If this topic interests you…
I highly recommend a paper titled "The problem with threads"
By Edward A. Lee, University of California Berkeley,
Published 05-01-2006
http://tinyurl.com/threadsYUCK

James' BIG 3 REASONS to avoid programming to specific hardware mechanisms

1. Portability is impaired severely when coding "close to the hardware"
2. Nested parallelism is IMPORTANT, and nearly impossible to manage well using "mandatory parallelism" methods such as threads
3. Other parallelism mechanisms (e.g., vectorization) needs to be considered and balanced. One machine may use threads, another may use vectors, another may use both. A hard coded program is "stuck."

"Close to hardware" may feel good, but understand what you give up!

Nesting is something that turns out to be VERY important… we really didn't understand that when we designed OpenMP in the mid-1990s. That mistake remains the single biggest problem (unsolvable without a fresh start) with OpenMP. Live and learn!

Vectorization – remains too much of a black art… I see promising ideas (including Cilk Plus) – but what we need to do TODAY (I'll show later in the talk) is still far more difficult than it should be for wide adoption…

Course Notes ONLY material

What makes a good abstraction?

- Hardware agnostic
- Performance
- "Scale forward" (preserve investments)
- Reliable and predictable
- Effective use of scarce resource: us

Parallel models: what makes a good abstraction?

Hardware agnostic, performance, "scale forward" (preserve investments)
Reliable and predictable
Effective use of scarce resource: us

Parallel Programming

- No widely used (popular) programming language was designed for expressing parallel programming.
  - Not Fortran, C, C++, Java, C#, Perl, Python, Ruby
- This creates many challenges
- Fundamental question of all programming languages: *level of abstraction*

Note: Java does have things that acknowledge that threading may be used… YES, I know that… but Java is not designed to make expressing parallelism easy… it simply came before we all knew how important this was going to be for everyone.

Course Notes ONLY material

Parallel Programming Models

- Sequential Semantics?
- A fundamental design choice;
- most abstract solutions being "retrofit" into existing programming languages tend to choose to preserve sequential semantics.
- TEST: if ignoring the parallel keywords/directives/calls would result in equivalent functionality (expected to be slower), then I'd expect that sequential semantics are at play.

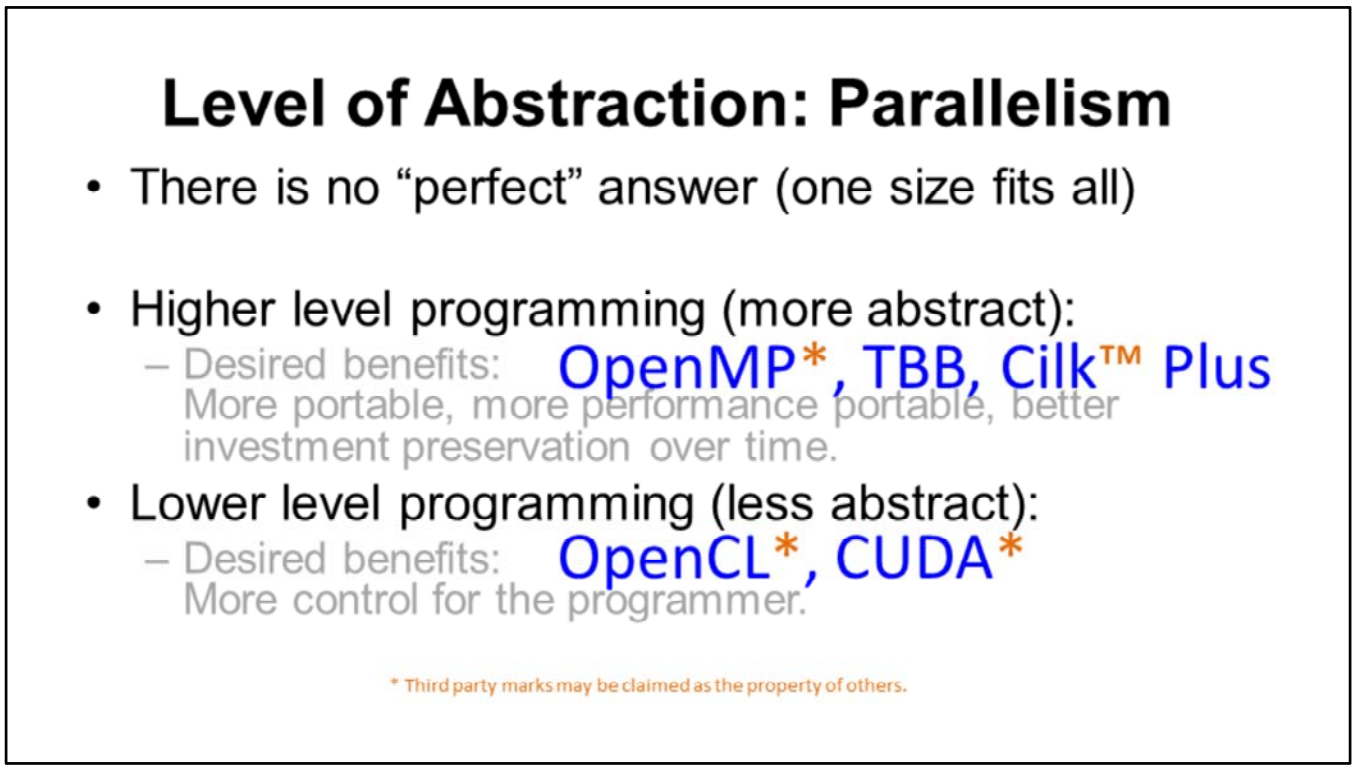
Programming Model Ideal Goals

- Performance:
  - achievable, scalable, predictable, tunable, portable
- Productivity:
  - expressive, composable, debugable, maintainable
- Portability
  - functionality & performance across operating systems, compilers, targets

Level of Abstraction: Parallelism

- There is no “perfect” answer (one size fits all)

- Higher level programming (more abstract):
  - Desired benefits:
    More portable, more performance portable, better investment preservation over time.
- Lower level programming (less abstract):
  - Desired benefits:
    More control for the programmer.



Yes, I like to encourage using the "high levels" of abstraction…

But my key message is really: KNOW the pros and cons before you choose… choose what is right for you!

Advancing C and C++

Examples of serious language issues:

- **Serial traps**
- **C++ futures**

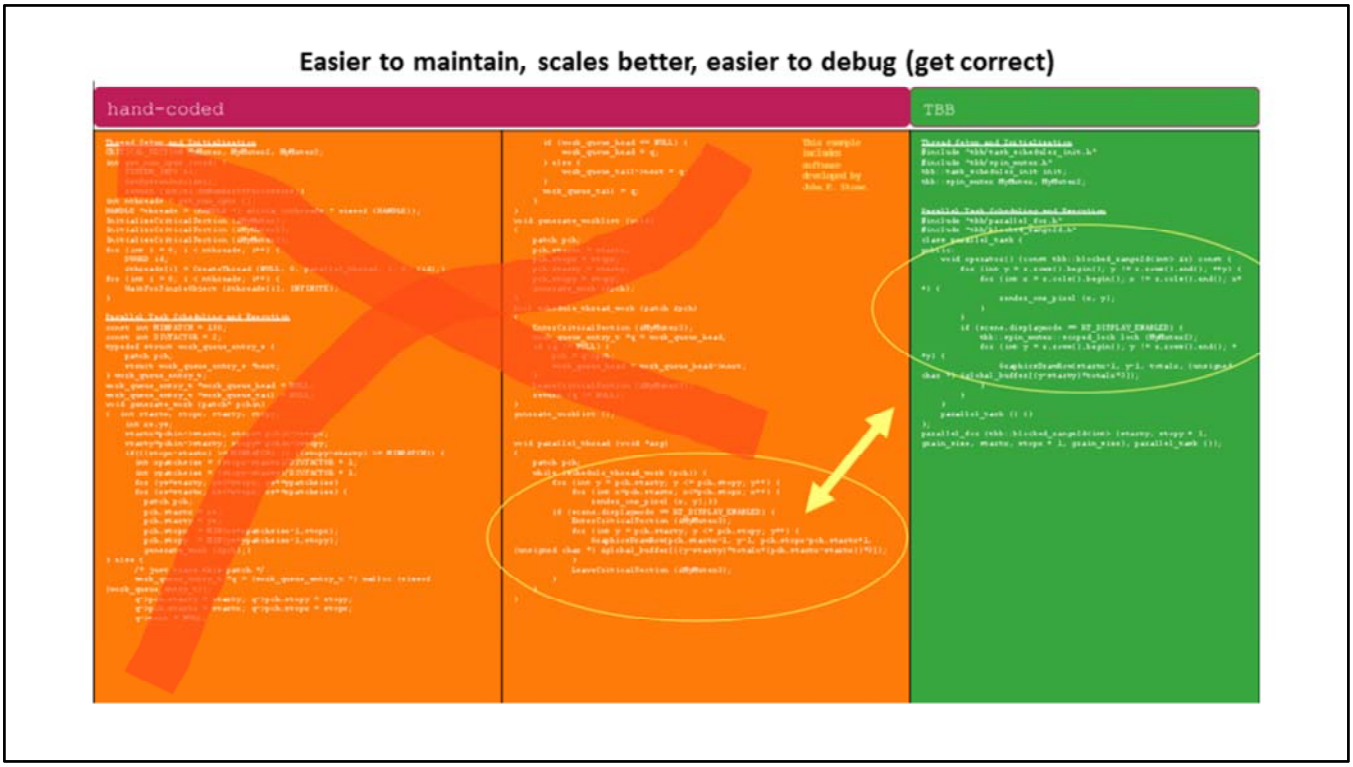
www.threadingbuildingblocks.org

- ✓ Most popular C++ abstraction
- ✓ Windows*
- ✓ Linux*
- ✓ Mac OS* X
- ✓ Xbox 360
- ✓ Solaris*
- ✓ FreeBSD*
- ✓ Intel processors
- ✓ AMD processors
- ✓ SPARC processors
- ✓ IBM processors
- ✓ open source
- ✓ standard committee submissions

The most used method to parallelize C++ programs

* Other names and brands may be claimed as the property of others.

TBB is a very successful open source project – one of the most rewarding projects I've ever work on!

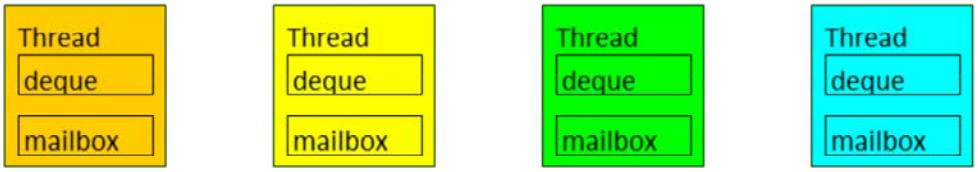


This code simply shows that without the TBB abstraction… the orange code has to set up threads and locks explicitly, including figuring out how many cores are available to run on… TBB "just does it"

Intel introduced Intel® Threading Building Blocks for C++ developers in 2006

- TBB will abstract scalable threading for developers of Windows, Linux and Mac OS applications
  - Java and .NET abstracted programming for C++, C, Fortran, Assembly language programmers
  - Threading Building Blocks abstracts low level threading for parallel programmers
- The developer writes high level / task level code that they're familiar with, the Threading Building Blocks detect the number of processor cores and scale the application

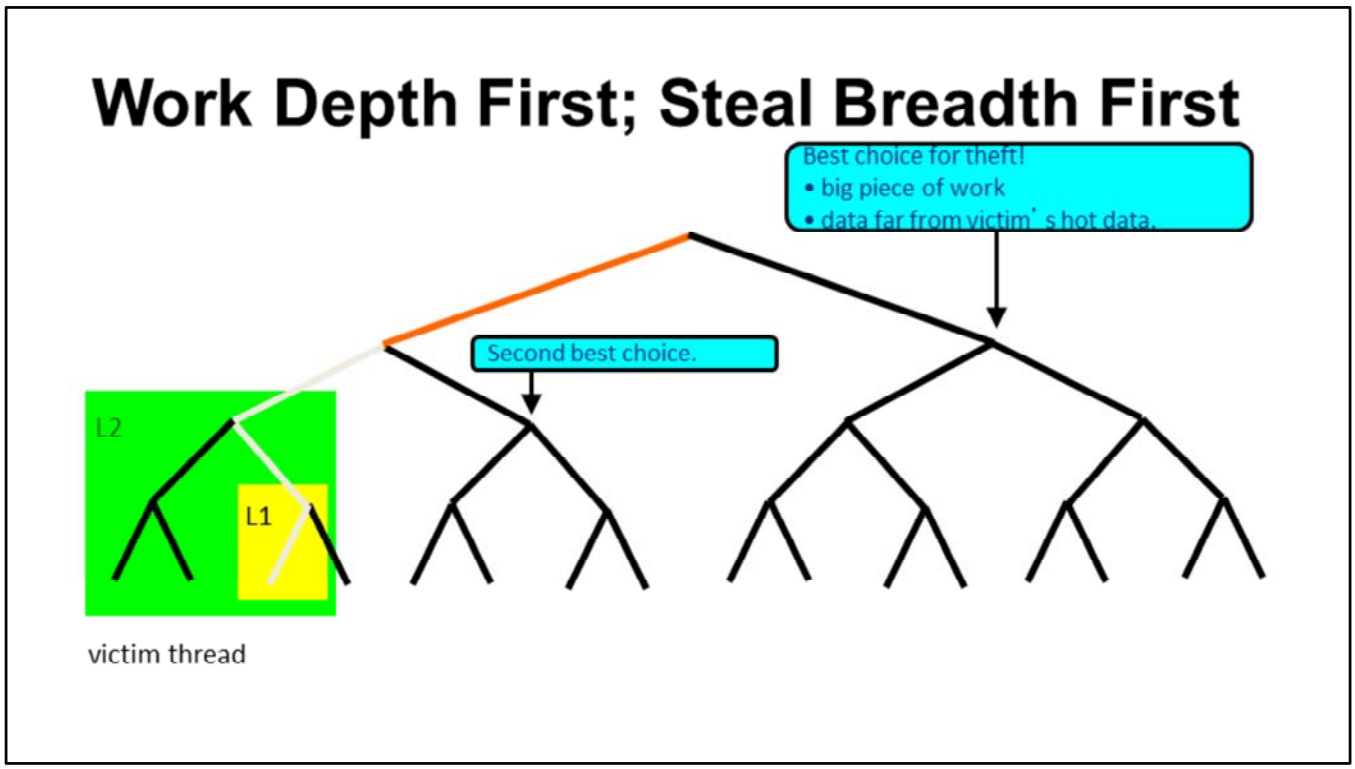
Work Stealing



0. Do explicitly specified task	**Override**
1. Take youngest task from my deque	**Locality**
2. Steal task advertised in mailbox	**Cache Affinity**
3. Steal oldest task from random victim	**Load balance**

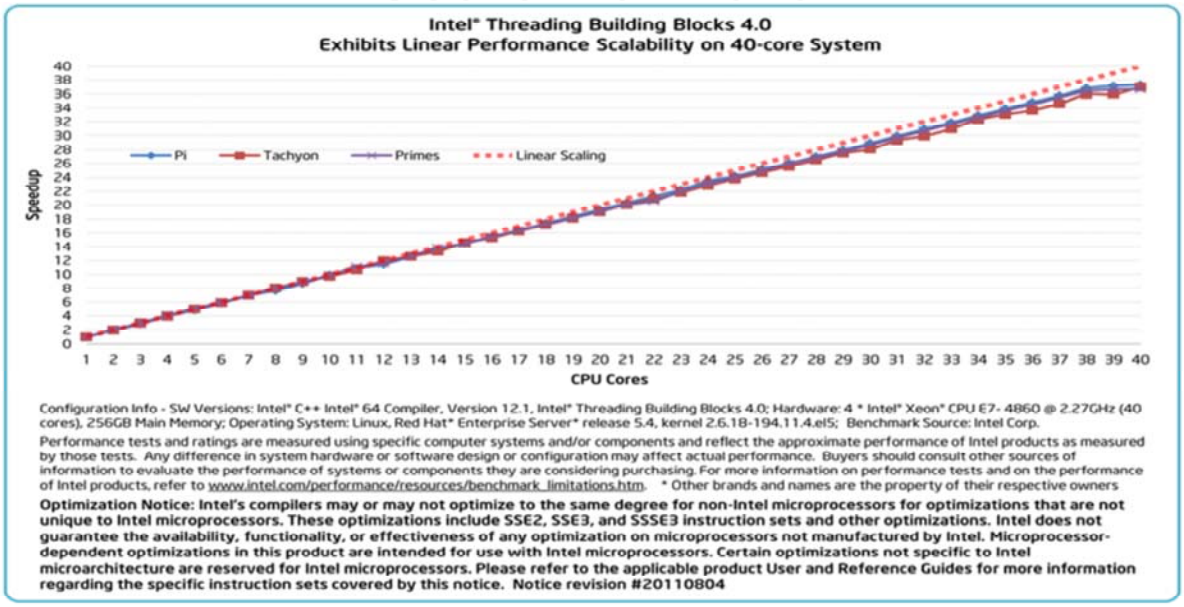
Work stealing is an important concept (invented by the Cilk project at MIT in the mid-1990s)… used my other schedulers… TBB made it very accessible to many people.

The concept is simple: distribute the queue of work to be done so that there is no global bottleneck involved in the distribution of tasks to the worker threads. With careful design, the cache behavior of this is excellent.

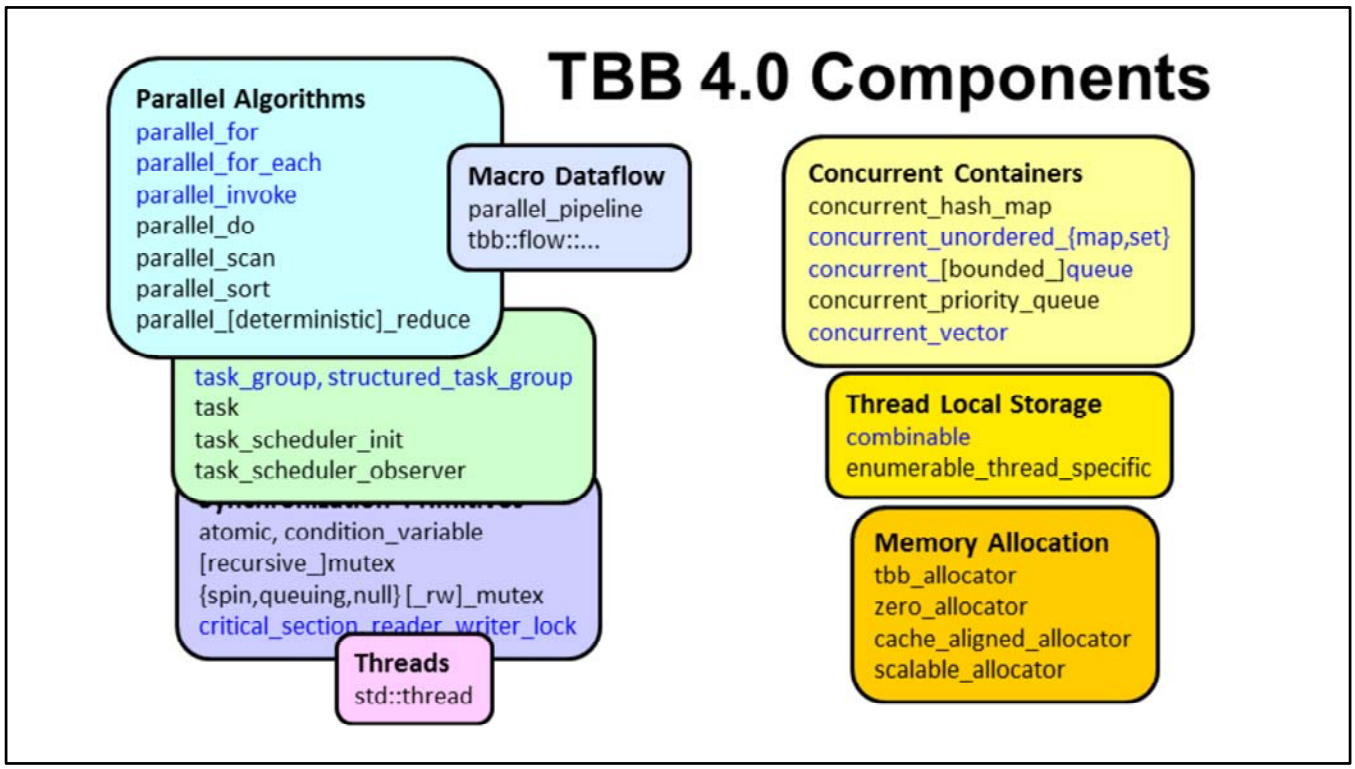


Good for on the fly load balancing *and* cache reuse

Scale Forward



This just proves that the overhead of using the runtime is virtually nothing… it doesn't promise you algorithm can scale… it does indicate that if your algorithm can scale, that TBB can stay out of the way and let it scale!



Some components are useful for other models. The containers, thread-local storage, and memory allocators, and synchronization primitives work with native threads too.
tbb::flow::... is a large subsystem targeting macro data flow.

Item in blue are common subset shared with Microsoft PPL.

tbb::parallel_for

- Has several forms.

Execute *functor*(*i*) for all $i \in$ [*lower*,*upper*)

```
parallel_for( lower, upper, functor );
```

Execute *functor*(*i*) for all $i \in$ {*lower*,*lower+stride*,*lower+2*stride*,...}

```
parallel_for( lower, upper, stride, functor );
```

Execute *functor*(*subrange*) for all *subrange* in *range*

```
parallel_for( range, functor );
```

Different way to get the same functionality.

For 2D.. The “lower, upper” or “lower, upper, stride” may seem easiest to understand… the “range” version is critical in higher dimensions.

Optional *partitioner* Argument

Recurse all the way down *range*.

```
tbb::parallel_for( range, functor, tbb::simple_partitioner() );
```

Choose recursion depth heuristically.

```
tbb::parallel_for( range, functor, tbb::auto_partitioner() );
```

Replay with cache optimization.

```
tbb::parallel_for( range, functor, affinity_partitioner );
```

auto_partitioner is the default now

without lambda, code has to go in a class

```
class ApplyABC {
public:
    float *a;
    float *b;
    float *c;
    ApplyABC(float *a_,float *b_,float *c_):a(a_), b(b_), c(c_) {}
    void operator()(const blocked_range<size_t>& r) const {
        for(size_t i=r.begin(); i!=r.end(); ++i)
            a[i] = b[i] + c[i];
    }
};

void ParallelFoo( float* a, float *b, float *c, int n ) {
    parallel_for(blocked_range<size_t>(0,n,10000),ApplyABC(a,b,c) );
```

doing with lambdas support is more natural

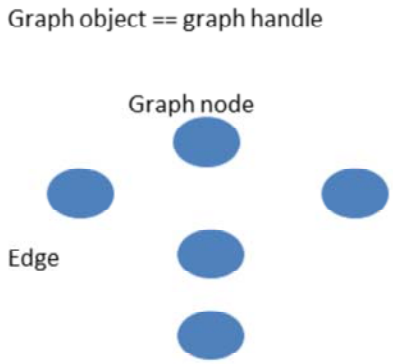
```
void ParallelApplyFoo(size_t n, int x) {
    parallel_for( blocked_range<size_t>(0,n,1000),
       [&]( const blocked_range<size_t>& r ) -> void
     {
         for( size_t i=r.begin(); i!=r.end(); ++i )
            a[i] = b[i] + c[i];
     });
}
```

I REALLY like TBB better now that C++11 offers this syntax!!!

Intel® TBB Class Graph: Components

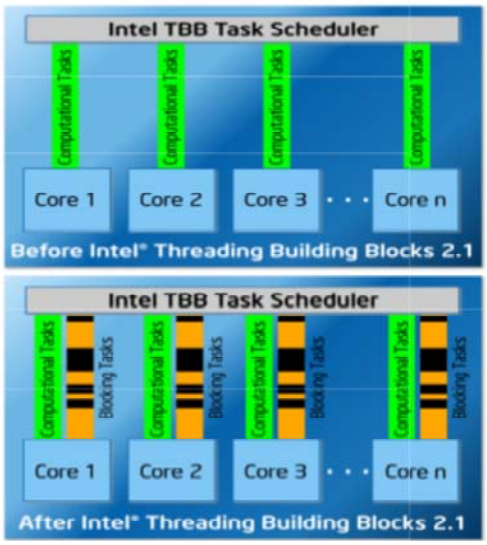
New Feature as of TBB 4.0 Release (2011)

- Graph object
  - Contains a pointer to the root task
  - Owns tasks created on behalf of the graph
  - Users can wait for the completion of all tasks of the graph
- Graph nodes
  - Implement *sender* and/or *receiver* interfaces
  - Nodes manage messages and/or execute function objects
- Edges
  - Connect predecessors to successors



A really great new feature in TBB – great for many problems: explicitly relating tasks based on dependencies… easy to do things like event-based programming, and much more!

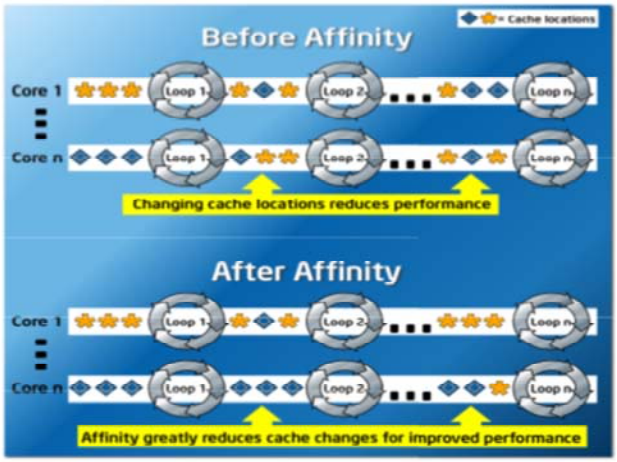
tbb_thread



Motivating the need here is the type of programming issue: blocking tasks – and how that challenges “program in tasks instead of threads”

When the NUMBER of SOFTWARE threads SHOULD be more than the number of PROCESSOR threads

affinity partitioner



Darn caches and memory – NOT going away – talk about how that challenges “program in tasks not threads”

Course Notes ONLY material

Mutex Summary

- "Scalable": does no worse than serializing
- Fair: preserves first-come first-serve (guarantees no starvation)
- Reentrant: thread can hold multiple locks on the same mutex
- Wait method: how threads wait for the lock

	"Scalable"	Reentrant	Fair	Long Wait	Size
mutex	OS dependent	No	OS dependent	Block	≥3 words
recursive_mutex	OS dependent	Yes	OS dependent	Block	≥3 words
spin_mutex	No	No	No	Yield	1 byte
queuing_mutex	Yes	No	Yes	Yield	1 word
spin_rw_mutex	No	No	No	Yield	1 word
queuing_rw_mutex	Yes	No	Yes	Yield	1 word
null_mutex	-	Yes	Yes	-	empty
null_rw_mutex	-	Yes	Yes	-	empty

Lots of portable lock options… code once, compile on lots of architectures and OSs

Course Notes ONLY material

Intel® Cilk™ Plus

Intel® Cilk™ Plus

Is a project at Intel to explore NEXT GEN tasking and vectorization solutions, with an emphasis on proving new approaches and then driving industry wide adoption (multiple compilers, and standards organizations)

Course Notes ONLY material

Scale Efficiently
Intel® Cilk™ Plus, three keywords to go parallel

```
cilk_for (int i=0; i<n; ++i) {
  foo(a[i]);
}
```

Course Notes ONLY material

Scale Efficiently
Intel® Cilk™ Plus, three keywords to go parallel

```
int fib(int n)
{
    if (n <= 2)
        return n;
    else {
        int x,y;
        x = fib(n-1);
        y = fib(n-2);
        return x+y;
    }
}
```

```
int fib(int n)
{
    if (n <= 2)
        return n;
    else {
        int x,y;
        x = cilk_spawn fib(n-1);
        y = fib(n-2);
        cilk_sync;
        return x+y;
    }
}
```

TBB

threadingbuildingblocks.org

- Reasons it might matter:
  - Everywhere (ports, open source)
  - Forever (commercial and open source)
- Problems:
  - C++ not C

Cilk™ Plus

cilkplus.org

- Reasons it might matter:
  - Space/time guarantees
  - C++ and C
  - Keywords bring compiler into the “know”
  - “Parent stealing”
  - *Vectorization help too (array notations, elem. func, SIMD)*
- Problems:
  - Requires compiler changes
  - Not feature rich

OpenMP*

openmp.org

- Reasons it might matter:
  - Everywhere (all major compilers)
  - Solutions for Tasking, vectorization, offload
- Problems:
  - C and Fortran, not so much C++
  - Not composable
  - Not always in-sync with language standards

OpenCL*

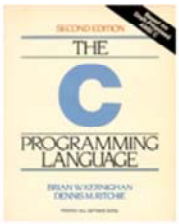
khronos.org/opencl

- Reasons it might matter:
  - Explicit heterogeneous controls
  - Everywhere (ports)
  - Non-proprietary
  - Underpinning for tools and libraries
- Problems:
  - Low level
  - Not performance portable
  - C moving to C++ only

Choice is Good

- Our favorite programming languages were NOT DESIGNED for parallelism.
- They need HELP.
- Multiple approaches and options are NEEDED.

- C
  - early key features "register" keyword out of use
  - "volatile" fading in usage
  - added: stronger typing (ANSI C, 1989)
  - C11
  - OpenMP* (1996)
  - Cilk™ Plus (2010)



- C++
  - Objected oriented
  - Intel® Threading Building Blocks (2006)
  - C++11
  - Cilk™ Plus (2010)

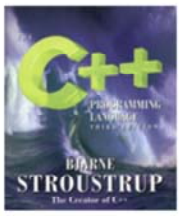


Photo: Wikimedia Commons (http://commons.wikimedia.org)

C++11 (some applies to C11 also)

Core language runtime performance enhancements
- Rvalue references and move constructors
- Generalized constant expressions
- Modification to the definition of plain old data

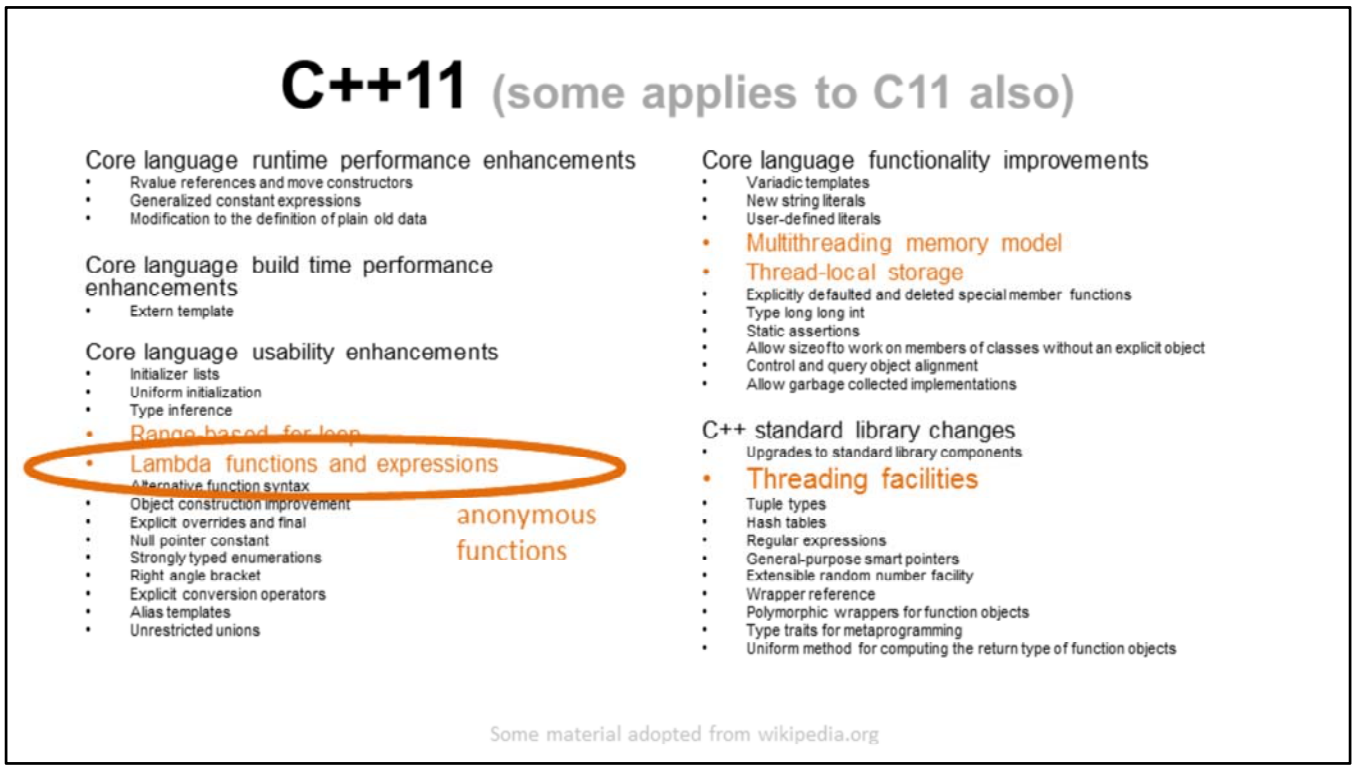
Core language build time performance enhancements
- Extern template

Core language usability enhancements
- Initializer lists
- Uniform initialization
- Type inference
- Range-based for-loop
- Lambda functions and expressions
- Alternative function syntax
- Object construction improvement
- Explicit overrides and final
- Null pointer constant
- Strongly typed enumerations
- Right angle bracket
- Explicit conversion operators
- Alias templates
- Unrestricted unions

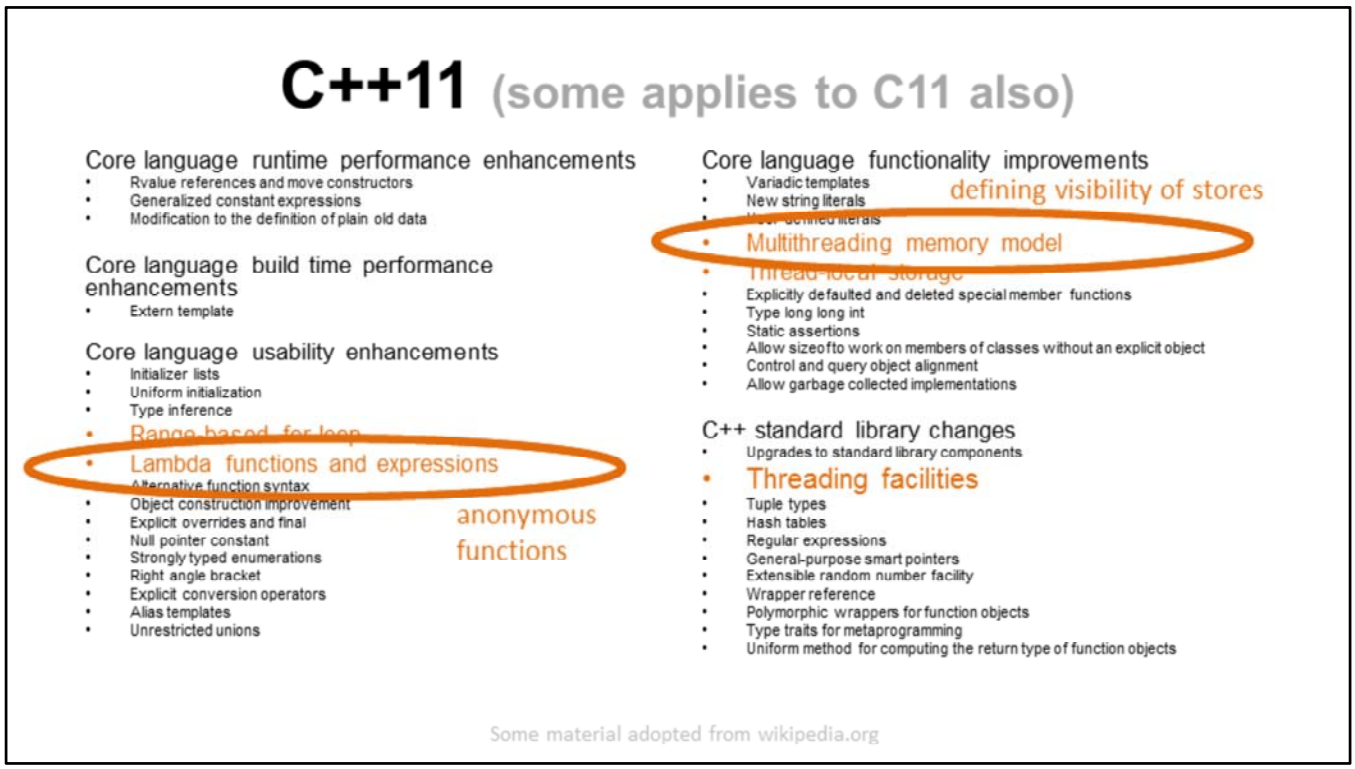
Core language functionality improvements
- Variadic templates
- New string literals
- User-defined literals
- Multithreading memory model
- Thread-local storage
- Explicitly defaulted and deleted special member functions
- Type long long int
- Static assertions
- Allow sizeof to work on members of classes without an explicit object
- Control and query object alignment
- Allow garbage collected implementations

C++ standard library changes
- Upgrades to standard library components
- Threading facilities
- Tuple types
- Hash tables
- Regular expressions
- General-purpose smart pointers
- Extensible random number facility
- Wrapper reference
- Polymorphic wrappers for function objects
- Type traits for metaprogramming
- Uniform method for computing the return type of function objects

Some material adopted from wikipedia.org

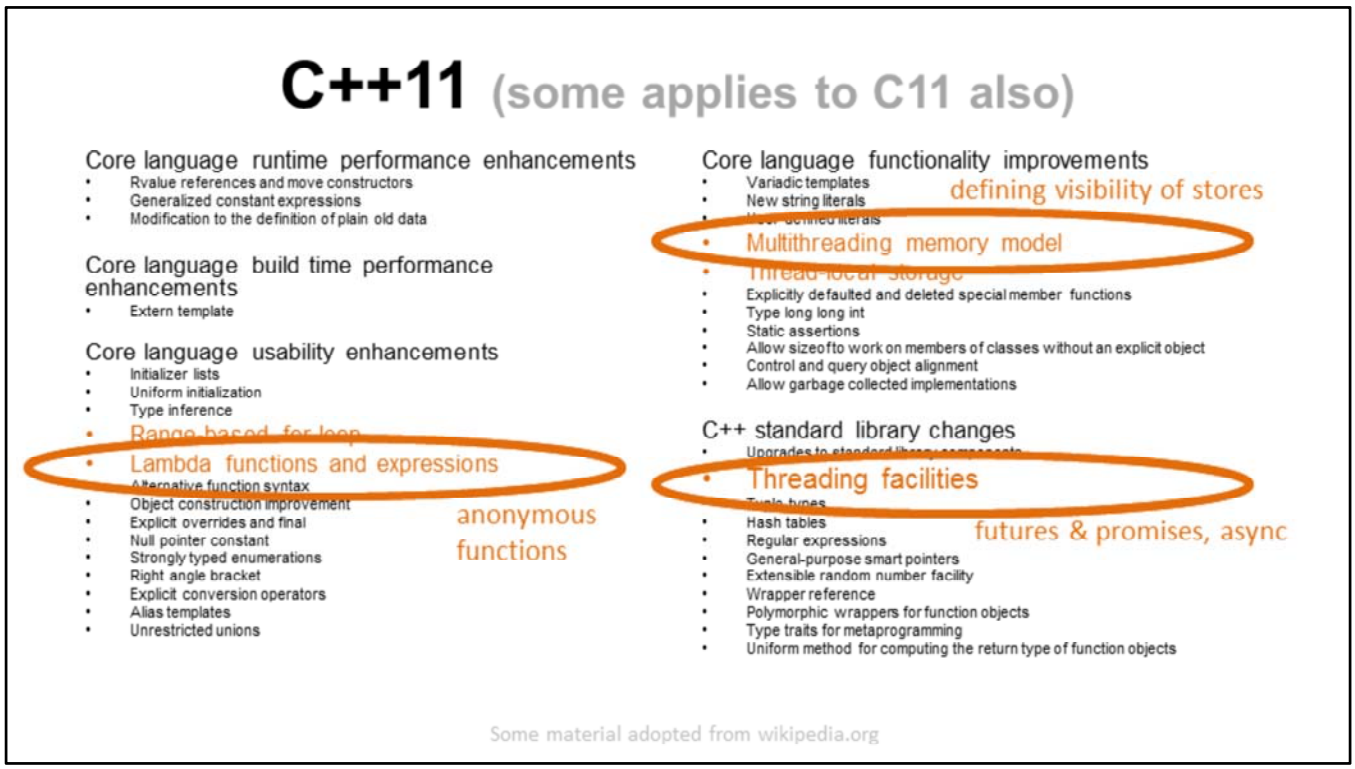


Nothing is more "exciting" than lambda functions. I'm a big fan. They really help make some constructs in parallel programming easier to read – including using TBB.



Maybe nothing is more important in C++11 than getting a specified memory model for multithreaded C++ programs.

This alone makes it shocking that C++ parallel programs ever worked reliably. The reality is this: most compilers behaved better than required, in order to make this better than it could have been. The standard gives us a blueprint for everyone to follow to tidy things up and keep them that way.



C++11 addresses key fundamental issues REQUIRED to make C++ a solid foundation for parallel programming.

When you get to know the things C++11 fixes… you might wonder why parallel programs work in C++ (or any language) at all.

C++11 makes C++ ready for serious parallel programming, but does not add the abstractions for effective abstract parallel programming.

Those mechanisms are being (sometime hotly) debated for C++17 and C++22.

Course Notes ONLY material

What about futures & promises?

future : *think of as a* consumer end of a 1-element produce/consumer queue

- A future can be created only from an existing promise object.
- Producer computes the value: calls set_value() on the promise.
- Consumer needs the future value: it calls get() on the future.
- Consumer blocks waiting on the producer if producer has not yet set_value().
- Futures can be used via the async() member function.

```
double foo(double arg);   // consider normal function

    // You can execute foo(x) asynchronously by calling
std::future<double>  result = std::async(foo, x);
...
double  val = result.get();
```

It is breathtaking that people continue to try to scale using futures 20 years after the practice was shown to be a dead-end.

Course Notes ONLY material

What about futures & promises?

The problems with the future/async model are both linguistic and performance-related.

The key flaw is that the whole notion of scalability with using futures was soundly refuted in the seminal 1993 paper:

Space-efficient scheduling of multithreaded computations by Blumofe and Leiserson.

This is the paper that motivated the development of Cilk in the first place.

It is breathtaking that people continue to try to scale using futures 20 years after the practice was shown to be a dead-end.

Course Notes ONLY material

What about futures & promises?

The linguistic problems are more subtle.

The following two statements that do roughly the same thing:

```
std::future<double> result = std::async(foo, x);
double result = cilk_spawn foo(x);
```

The first statement looks like a call to async().
The second statement looks like a call to foo().

This is an aesthetic distinction, but nevertheless important.

Course Notes ONLY material

What about futures & promises?

Semantically, consider the following:

```
std::string s("hello");
int bar(const std::string& s);

std::future<int> result = std::async(bar, s + " world");
```

The above statement is intended to pass "hello world" to bar and run it asynchronously.

This means that bar may get a reference to a temporary string that has already been destroyed by the time it is used.

Course Notes ONLY material

What about futures & promises?

Semantically, consider the following:

```
std::string s("hello");
int bar(const std::string& s);

std::future<int> result = std::async(bar, s + " world");
```

The above statement is intended to pass "hello world" to bar and run it asynchronously.

The problem is that s + " world" is a *temporary* object that gets destroyed as soon as the statement completes.

This means that bar may get a reference to a temporary string that has already been destroyed by the time it is used.

Course Notes ONLY material

What about futures & promises?

```
std::string s("hello");
int bar(const std::string& s);

std::future<int> result = std::async(bar, s + " world");
```

Boosters of std::async will counter that all you need is to add a lambda:

```
std::future<int> result = std::async([&]{ bar(s + " world"); });
```

Without the lambda - it is a *race condition* that should **not** exist in a linguistically sound parallel construct, but it is pretty much unavoidable in a library-only specification.

While this would work (in this case), it has two problems: 1) It is ugly as sin and 2) the user can do this only if the user recognizes the bug in the first place. While this is really just another way to create race conditions, I consider it to be a particularly subtle one with a particularly ugly work-around. It is also a race condition that should not exist in a linguistically sound parallel construct, but it is pretty much unavoidable in a library-only specification.

Vectorization: Who, What, Why, When

The realities of vectorization (data parallelism)

- **how programming languages are the real failure point not compilers**
- **effective solutions**
- **future possibilities**

COURSE SCHEDULE
9 am Introduction
9:05 am Multithreading Introduction and Overview, Reinders
9:50 am Asynchronous Computation Engine for Animation, ElKoura
10:20 am Break
10:30 am Parallel Evaluation of Character Rigs Using TBB, Watt
11 am GPU Rigid Body Simulation Using OpenCL, Coumans
11:30 am Parallelism in Tools Development for Simulation and Rendering, Henderson
Noon Parallelism in Houdini, Lait

A moment of clarity

task parallelism
doesn't matter
without data parallelism

(James' observation about what
Amdahl and Gustafson were ultimately pointing out)

vector data operations: data operations done in parallel

```
void v_add (float *c,
            float *a,
            float *b)
{
    for (int i=0; i<= MAX; i++)
       c[i]=a[i]+b[i];
}
```

My simple vector add

vector data operations: data operations done in parallel

```
void v_add (float *c,
            float *a,
            float *b)
{
    for (int i=0; i<= MAX; i++)
       c[i]=a[i]+b[i];
}
```

Loop:

1. LOAD a[i] -> Ra
2. LOAD b[i] -> Rb
3. ADD Ra, Rb -> Rc
4. STORE Rc -> c[i]
5. ADD i + 1 -> i

My simple serial “assembly” code to perform the loop body

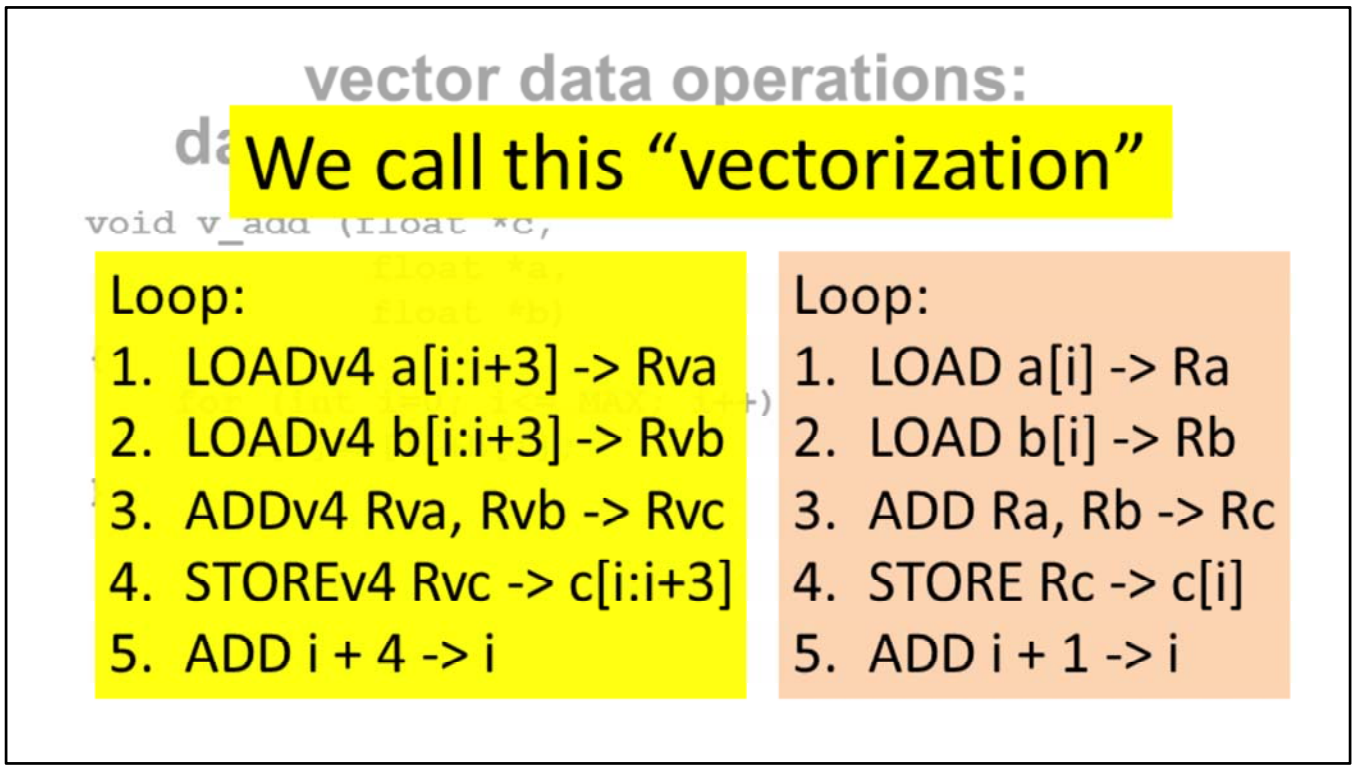
vector data operations: data operations done in parallel

```
void v_add (float *c,
```

Loop:
1. LOADv4 a[i:i+3] -> Rva
2. LOADv4 b[i:i+3] -> Rvb
3. ADDv4 Rva, Rvb -> Rvc
4. STOREv4 Rvc -> c[i:i+3]
5. ADD i + 4 -> i

Loop:
1. LOAD a[i] -> Ra
2. LOAD b[i] -> Rb
3. ADD Ra, Rb -> Rc
4. STORE Rc -> c[i]
5. ADD i + 1 -> i

My simple parallel (four at a time vector instructions) “assembly” code to perform the loop body



This is what we mean by “vectorization” in modern microprocessors: the transformation of the instructions used to ones that do multiple operations at a time.

vector data operations: data operations done in parallel

```
void v_add (float *c, float *a, float *b)
{
     for (int i=0; i<= MAX; i++)
        c[i]=a[i]+b[i];
}
```

PROBLEM:

This LOOP is NOT LEGAL to (automatically) VECTORIZE in C / C++ (without more information).

- Arrays not really in the language
- Pointers are, evil pointers!

We'll see why later.

vectorization solutions

1. auto-vectorization (let the compiler do it *when legal*)
   - sequential languages and practices gets in the way
2. give the compiler hints
   - C99 "restrict" keyword (implied in FORTRAN since 1956)
   - Traditional pragmas like "#pragma IVDEP"
   - Cilk Plus #pragma SIMD
   - OpenMP 4.0 #pragma OMP SIMD
3. code differently
   - SIMD instruction intrinsics
   - Cilk Plus array notations
   - Cilk Plus __declspec (vector)
   - OpenMP 4.0 #pragma OMP declare SIMD
   - OpenCL / CUDA kernel functions

vectorization solutions

1. auto-vectorization (let the compiler do it *when legal*)
   - sequential languages and practices gets in the way
2. give the compiler hints
   - C99 "restrict" keyword (implied in FORTRAN since 1956)
   - Traditional pragmas like "#pragma IVDEP"
   - Cilk Plus #pragma SIMD
   - OpenMP 4.0 #pragma OMP SIMD
3. code differently
   - SIMD instruction intrinsics
   - Cilk Plus array notations
   - Cilk Plus __declspec (vector)
   - OpenMP 4.0 #pragma OMP declare SIMD
   - OpenCL / CUDA kernel functions

In all cases, studying "vectorization reports" can become a way of life.

vec_report is one of the most popular switches on the Intel compilers with our users – very helpful in the battle against inadaquate programming languages

C99 “restrict” keyword

```
void v_add (float *restrict c,
            float *restrict a,
            float *restrict b)
{
    for (int i=0; i<= MAX; i++)
       c[i]=a[i]+b[i];
}
```

2. give the compiler hints
   - C99 “restrict” keyword
   - Traditional pragmas like “#pragma IVDEP”
   - Cilk Plus #pragma SIMD
   - OpenMP 4.0 #pragma OMP SIMD
3. code differently
   - SIMD instruction intrinsics
   - Cilk Plus array notations
   - Cilk Plus __declspec(vector)
   - OpenMP 4.0 #pragma OMP declare SIMD
   - OpenCL / CUDA kernel functions

The C99 standard (not part of any C++ standard) introduces the restrict keyword for use in pointer declarations.
It is a declaration of intent given by the programmer to the compiler. It declares the intent that during lifetime of the pointer, only it or a value directly derived from it (such as pointer + 1) will be used to access the object to which it points. In other words, no aliasing to another pointer exists that will be used. This limits the effects of pointer aliasing, aiding optimizations such as vectorization. If the declaration of intent is not followed and the object is accessed by an independent pointer, this may result in undefined behavior.
The use of the restrict keyword in C, in principle, allows C to achieve the same performance as the same program written in Fortran in a straight forward manner.
A key use is in parameters to functions. In Fortran, aliasing of parameters has always been forbidden (original specification was in 1956).
In C programming, aliasing of parameters is allowed but should be discouraged because of the increased difficulty in understanding such a program and the resulting increase in opportunity for programming mistakes.

IVDEP

```
void v_add (float *c,
            float *a,
            float *b)
{
#pragma IVDEP
    for (int i=0; i<= MAX; i++)
       c[i]=a[i]+b[i];
}
```

2. give the compiler hints
   - C99 "restrict" keyword
   - Traditional pragmas like "#pragma IVDEP"
   - Cilk Plus #pragma SIMD
   - OpenMP 4.0 #pragma OMP SIMD
3. code differently
   - SIMD instruction intrinsics
   - Cilk Plus array notations
   - Cilk Plus __declspec(vector)
   - OpenMP 4.0 #pragma OMP declare SIMD
   - OpenCL / CUDA kernel functions

#pragma IVDEP is a non-standard but widely supported compiler hint

It tells a compiler that it can ignore "implied loop carried dependences."
In plain English, it says that loop iterations can be assumed to be independent if the compiler had doubts.
All the compilers I know of, will only ignore the "maybe" issues and not any explicit (real) dependences that are found.

The net effect is that the compiler will find the loop acceptable to vectorize.

SIMD

```
void v_add (float *c,
            float *a,
            float *b)
{
#pragma SIMD
    for (int i=0; i<= MAX; i++)
       c[i]=a[i]+b[i];
}
```

2. give the compiler hints
   - C99 "restrict" keyword
   - Traditional pragmas like "#pragma IVDEP"
   - Cilk Plus #pragma SIMD
   - OpenMP 4.0 #pragma OMP SIMD
3. code differently
   - SIMD instruction intrinsics
   - Cilk Plus array notations
   - Cilk Plus __declspec(vector)
   - OpenMP 4.0 #pragma OMP declare SIMD
   - OpenCL / CUDA kernel functions

#pragma SIMD is a non-standard compiler hint, part of Intel's Cilk Plus project which is gaining in popularity (multiple compiler implementations) – http://cilkplus.org

It tells a compiler that it can ignore "implied loop carried dependences."
In plain English, it says that loop iterations can be assumed to be independent if the compiler had doubts.
All the compilers I know of, will only ignore the "maybe" issues and not any explicit (real) dependences that are found.

The net effect is that the compiler will find the loop acceptable to vectorize.

OMP SIMD

```
void v_add (float *c,
            float *a,
            float *b)
{
#pragma OMP SIMD
    for (int i=0; i<= MAX; i++)
       c[i]=a[i]+b[i];
}
```

2. give the compiler hints
   - C99 "restrict" keyword
   - Traditional pragmas like "#pragma IVDEP"
   - Cilk Plus #pragma SIMD
   - OpenMP 4.0 #pragma OMP SIMD
3. code differently
   - SIMD instruction intrinsics
   - Cilk Plus array notations
   - Cilk Plus __declspec(vector)
   - OpenMP 4.0 #pragma OMP declare SIMD
   - OpenCL / CUDA kernel functions

The #pragma SIMD from Cilk Plus was accepted as a feature in the upcoming OpenMP 4.0. The meaning is the same as #pragma SIMD.
OpenMP is widely implemented (by most compilers).
OpenMP 4.0 (available from http://openmp.org)
http://openmp.org

SIMD instruction intrinsics

```
void v_add (float *c,
            float *a,
            float *b)
{
    __m128* pSrc1 = (__m128*) a;
    __m128* pSrc2 = (__m128*) b;
    __m128* pDest = (__m128*) c;
    for (int i=0; i<= MAX/4; i++)
        *pDest++ = _mm_add_ps(*pSrc1++, *pSrc2++);
}
```

2. give the compiler hints
   - C99 "restrict" keyword
   - Traditional pragmas like "#pragma IVDEP"
   - Cilk Plus #pragma SIMD
   - OpenMP 4.0 #pragma OMP SIMD
3. code differently
   - SIMD instruction intrinsics
   - Cilk Plus array notations
   - OpenMP 4.0 #pragma OMP declare SIMD
   - OpenCL / CUDA kernel functions

SIMD instruction intrinsics offer "pseudo-functions" for each major SIMD instruction on a one-to-one basis.

SIMD instruction intrinsics let the compiler do much of the work while allowing for very tight control.

Unfortunately, SIME instruction intrinsics are tied tightly to the SIMD instructions and hence the width.
SSE intrinsics are written for 128-bit wide SIMD, whereas AVX is for 256-bit wide SIMD, and AVX-512 intrinsics are for 512-bit wide SIMD.
This either forces rewrites to move between vendors or to future width increases, which is an unacceptable expectation.

I suggest avoiding SIMD intrinsics.

array operations

```
void v_add (float *c,
            float *a,
            float *b)
{
        c[0:MAX]=a[0:MAX]+b[0:MAX];
}
```

Challenge: long vector slices can cause cache issues; fix is to keep vector slices short.

2. give the compiler hints
   - C99 "restrict" keyword
   - Traditional pragmas like "#pragma IVDEP"
   - Cilk Plus #pragma SIMD
   - OpenMP 4.0 #pragma OMP SIMD
3. code differently
   - SIMD instruction intrinsics
   - Cilk Plus array notations
   - Cilk Plus __declspec(vector)
   - OpenMP 4.0 #pragma OMP declare SIMD
   - OpenCL / CUDA kernel functions

Similar to Fortran90 arrays

See http://http://cilkplus.org/tutorial-array-notation

Intel Cilk Plus includes extensions to C and C++ that allows for parallel operations on arrays. The intent is to allow users to express high-level vector parallel array operations. This helps the compiler to effectively vectorize the code. Array notation can be used for both static and dynamic arrays. In addition, it can be used inside conditions for "if" and "switch" statements. The extension has parallel semantics that are well suited for per-element operations that have no implied ordering and are intended to execute in data-parallel instructions.
A new operator **[:]** delineates an array section:
array-expression[lower-bound : length : stride]
Length is used instead of upper bound as C and C++ arrays are declared with a given length.
The three elements of the array notation can be any integer expression. The user is responsible for keeping the section within the bounds of the array.
Each argument to [:] may be omitted if the default value is sufficient.

The default lower-bound is 0.

The default length is the length of the array. If the length of the array is not known, length must be specified.

The default stride is 1. If the stride is defaulted, the second ":" may be omitted.

So using *array-expression[:]* denotes an array section that is the entire array of known length and stride 1.
Array notation allows for multi-dimensional array sections as multiple array section operators can be used for a single array.
array-expression[:][:] denotes a two dimensional array
Two new terms are introduced for array sections:
The *rank* is the number of array sections used on a single array. A rank zero expression is a scalar expression.
The *shape* is the length of each array section.
Within a statement all expressions must have the same rank and shape or must be rank zero. A rank zero expression will be evaluated once and broadcast for each element in the array section.
Array notation also provides a set of built-in functions to provide reductions, shifts, and rotations of array

sections.
Note that it is undefined behavior if there is overlap between the right hand side array expression and the left hand side unless it is exact overlap.

__declspec(vector)

```
__declspec(vector)
void v_add (float *c,
            float *a,
            float *b)
{
for (int i=0; i<= MAX; i++)
       c[i]=a[i]+b[i];
}
```

2. give the compiler hints
   - C99 "restrict" keyword
   - Traditional pragmas like "#pragma IVDEP"
   - Cilk Plus #pragma SIMD
   - OpenMP 4.0 #pragma OMP SIMD
3. code differently
   - SIMD instruction intrinsics
   - Cilk Plus array notations
   - Cilk Plus __declspec(vector)
   - OpenMP 4.0 #pragma OMP declare SIMD
   - OpenCL / CUDA kernel functions

Elemental (kernel) function – a scalar function that get used in a vector form with this hint to the compiler!

#pragma OMP declare SIMD

```
#pragma OMP declare SIMD
void v_add (float *c,
            float *a,
            float *b)
{
for (int i=0; i<= MAX; i++)
       c[i]=a[i]+b[i];
}
```

2. give the compiler hints
   - C99 "restrict" keyword
   - Traditional pragmas like "#pragma IVDEP"
   - Cilk Plus #pragma SIMD
   - OpenMP 4.0 #pragma OMP SIMD
3. code differently
   - SIMD instruction intrinsics
   - Cilk Plus array notations
   - Cilk Plus __declspec(vector)
   - OpenMP 4.0 #pragma OMP declare SIMD
   - OpenCL / CUDA kernel functions

Elemental (kernel) function – a scalar function that get used in a vector form with this hint to the compiler!

kernel

```
kernel void v_add (global const float *c,
                   global const float *a,
                   global const float *b)
{
      int id = get_global_id(0);
      c[id]=a[id]+b[id];
}
```

2. give the compiler hints
   - C99 "restrict" keyword
   - Traditional pragmas like "#pragma IVDEP"
   - Cilk Plus #pragma SIMD
   - OpenMP 4.0 #pragma OMP SIMD
3. code differently
   - SIMD instruction intrinsics
   - Cilk Plus array notations
   - Cilk Plus __declspec(vector)
   - OpenMP 4.0 #pragma OMP declare SIMD
   - OpenCL / CUDA kernel functions

Elemental (kernel) function – a scalar function that get used in a vector form with OpenCL construct

vector data operations: data operations done in parallel

```
void v_add (float *c,
            float *a,
            float *b)
{
    for (int i=0; i<= MAX; i++)
       c[i]=a[i]+b[i];
}
```

```
float a[MAX];
a[0] = 1;
v_add(a+1,a,a);
```

Call sequence loads the array with powers of 2.
This legal call sequence – CANNOT be vectorized (without getting different answers)

vector data operations: data operations done in parallel

```
void v_add (float *c,
            float *a,
            float *b)
{
    for (int i=0; i<= MAX; i++)
       c[i]=a[i]+b[i];
}
```

```
float a[MAX];
a[0] = 1;
v_add(a+1,a,a);
```

Legal call sequence – CANNOT be vectorized
(without getting different answers)

Serial Trap?

Call sequence loads the array with powers of 2.
This legal call sequence – CANNOT be vectorized (without getting different answers)

Memory

optimizing the sharing & movement of data

is *very often* more important than

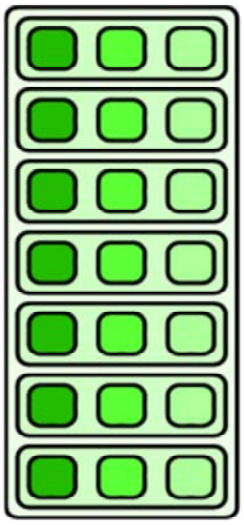
optimizing calculations

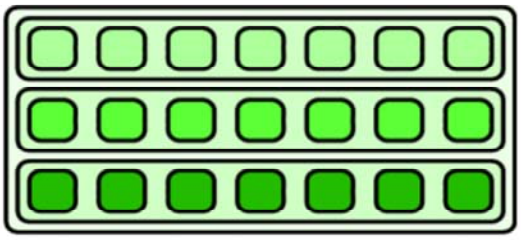
TREND: "very often" heads toward "always"

Memory concerns are HUGE… I wish I had an easy solution to offer.

I think this may be the single most important challenge to help with through tooling in the future!

Data Layout: AoS vs. SoA





Array of structures (AoS) tends to cause cache alignment problems, and is hard to vectorize.

Structure of arrays (SoA) can be easily aligned to cache boundaries and is vectorizable.

SoA is great for independently used arrays and when vectorization is desired

AofS – while often “avoided” can better for some programs because of cache behavior – VERY program dependent

When in doubt – SoA is more likely to be the better

Data Layout: Alignment

Array of Structures (AoS), padding at end.

Array of Structures (AoS), padding after each structure.

Structure of Arrays (SoA), padding at end.

Structure of Arrays (SoA), padding after each component.

sharing

- decompose data to minimize sharing between tasks (threads)
- beware: false sharing (it's very real)

False sharing is a term used to describe when two or more data elements reside in the same cache line, but are used by different threads or cores… this creates a condition where the cache line is SHARED between cores which can cause thrash (if both are writing to the same cache line – even if different parts of the cache line)…

Avoid by padding data structures

Avoid by using thread aware memory allocation (such as the memory allocator included within TBB)

Think Parallel

- Parallelism is almost never something you can "stick in" at the last minute in a program
- You are best off if you think about everything in terms of how to do in parallel as cleanly as possible

"Thinking parallel"

the benefits of teaching parallel programming without teaching computer architecture, gasp!

why computer architecture matters

Think Parallel

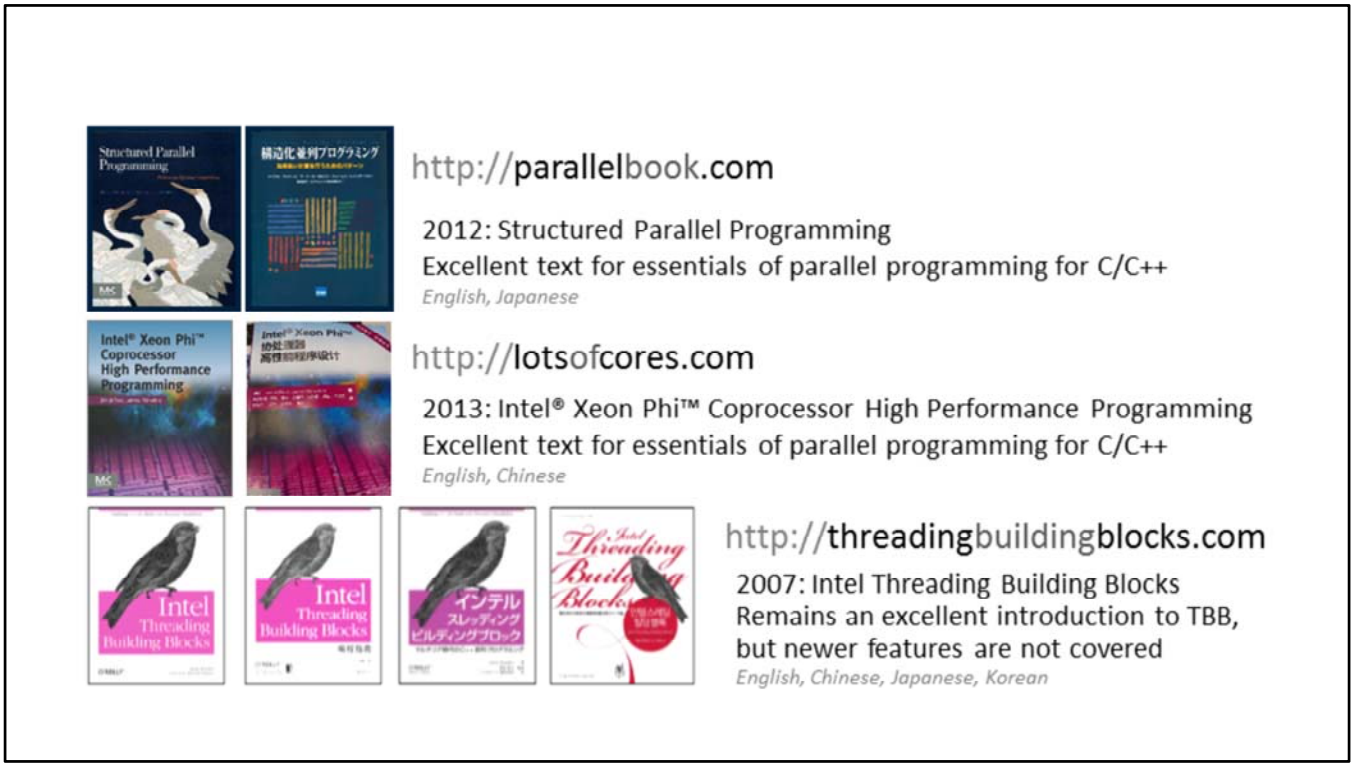
- Abstract parallelism is best – can be taught without learning computer architecture
- Contradicting the desire to ignore computer architecture as a driver for programming…

  Computer architecture basics matter – most of all: movement of data needs to be minimalized

"Thinking parallel"

the benefits of teaching parallel programming without teaching computer architecture, gasp!

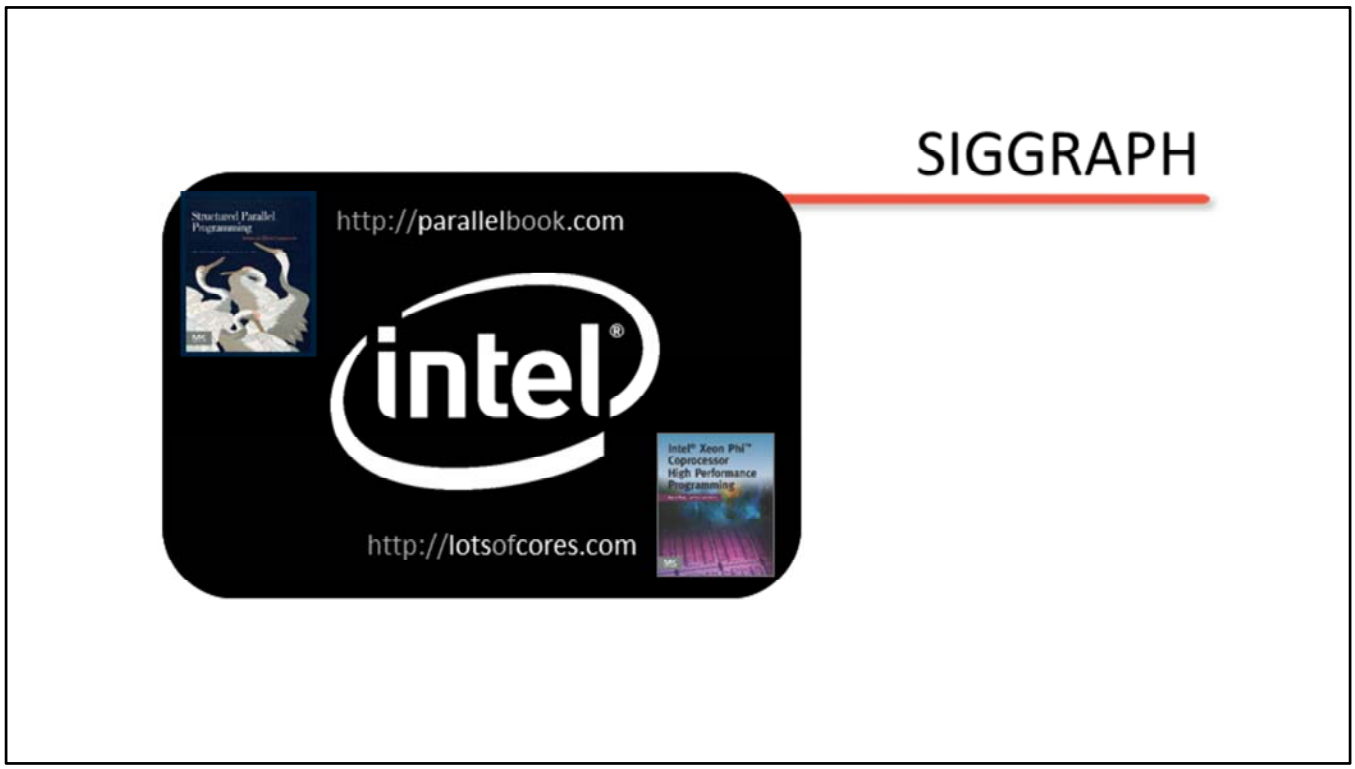
why computer architecture matters



Some comments about books I've worked on for parallel programming:

"Structured Parallel Programming" is designed to teach the key concepts for parallel programming for C/C++ without teaching it via computer architecture. In other words, we teach parallel programming as a programming skill – and show code, and discuss the standard solutions (like map, reduce, stencils, etc.) to solve parallel programming problems. Published in July 2012, it is very timely. A Japanese translation was published in 2013.

James Reinders, Director and Parallel Programming Evangelist and Senior Engineer, Intel
Parallel Programming Evangelist. James is involved in multiple engineering, research and educational efforts to increase use of parallel programming throughout the industry. He joined Intel Corporation in 1989, and has contributed to numerous projects including the world's first TeraFLOP/s supercomputer (ASCI Red) and the world's first TeraFLOP/s microprocessor (Intel® Xeon Phi™ coprocessor). James been an author on numerous technical books, including VTune™ Performance Analyzer Essentials (Intel Press, 2005), Intel® Threading Building Blocks (O'Reilly Media, 2007), Structured Parallel Programming (Morgan Kaufmann, 2012), Intel® Xeon Phi™ Coprocessor High Performance Programming (Morgan Kaufmann, 2013), and Multithreading for Visual Effects (A K Peters/CRC Press, 2014), High Performance Parallelism Pearls (Morgan Kaufmann, 2015) and High Performance Parallelism Pearls Volume 2 (Morgan Kaufmann, 2016). James is currently working on an update to the 2013 book on Intel Xeon Phi which will address the new Knight Landing product.



Multithreading and VFX

Parallelism is important to many aspects of visual effects. In this course, experts in several key areas present their specific experiences in applying parallelism to their domain of expertise. The problem domains are very diverse, and so are the solutions employed, including specific threading methodologies. This allows the audience to gain a wide understanding of various approaches to multithreading and compare different techniques, which provides a broad context of state-of-the-art approaches to implementing parallelism and helps them decide which technologies and approaches to adopt for their own future projects. The presenters describe both successes and pitfalls, the challenges and difficulties they encountered, and the approaches they adopted to resolve these issues.

The course begins with an overview of the current state of parallel programming, followed by five presentations on various domains and threading approaches. Domains include rigging, animation, dynamics, simulation, and rendering for film and games, as well as a threading implementation for a full-scale commercial application that covers all of these areas. Topics include CPU and GPU programming, threading, vectorization, tools, debugging techniques, and optimization and performance-profiling approaches.

Level: Intermediate; Prerequisites: Software development background. Parallel programming experience is not required. Intended Audience: Software developers or technical artists interested in improving performance of their applications though application of parallel-programming techniques.

James Reinders, Director and Parallel Programming Evangelist and Senior Engineer, Intel
Parallel Programming Evangelist. James is involved in multiple engineering, research and educational efforts to increase use of parallel programming throughout the industry. He joined Intel Corporation in 1989, and has contributed to numerous projects including the world's first TeraFLOP/s supercomputer (ASCI Red) and the world's first TeraFLOP/s microprocessor (Intel® Xeon Phi™ coprocessor). James been an author on numerous technical books, including VTune™ Performance Analyzer Essentials (Intel Press, 2005), Intel® Threading Building Blocks (O'Reilly Media, 2007), Structured Parallel Programming (Morgan Kaufmann, 2012), Intel® Xeon Phi™ Coprocessor High Performance Programming (Morgan Kaufmann, 2013), and Multithreading for Visual Effects (A K Peters/CRC Press, 2014), High Performance Parallelism Pearls (Morgan Kaufmann, 2015) and High Performance Parallelism Pearls Volume 2 (Morgan Kaufmann, 2016). James is

currently working on an update to the 2013 book on Intel Xeon Phi which will address the new Knight Landing product.

Legal Disclaimer & Optimization Notice

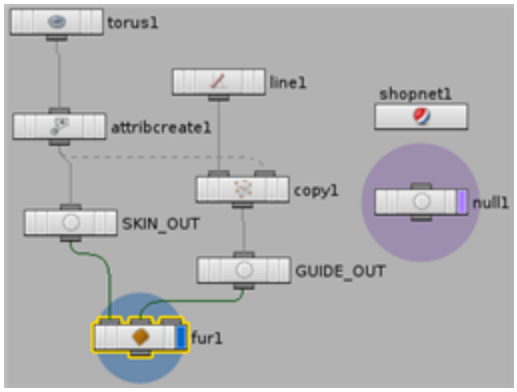
Multithreading Houdini

Definitions

What is Houdini?

The goal of this course is to share our experiences and pitfalls in the on-going process of making Houdini run in a multithreaded manner. While the attempt will be made to make the examples as context-agnostic as possible, I will no doubt fall into some specialized jargon and make some assumptions about your familiarity with the package. The following can be skipped by those already familiar with Houdini.

Houdini refers to the flagship package of Side Effects Software, http://www.sidefx.com, which is a complete 3d animation and effects package. It is well known for its extremely procedural approach to art creation. The driving vision of the package is to create the tools that let artists express themselves using this new visual medium. What really separates computer generated art from other types of art is proceduralism: computers excel at repeating rote tasks. Unfortunately, the act of instructing computers is considered a highly technical task, and often seen as divorced from the act of creating art. Procedural art tools are an attempt to bridge that gap, to ensure that the full power of computers can be harnessed by artists.



Central to the Houdini interface is the network editor. The network consists of many nodes (also called operators or OPs) which are wired together. Logically the data flows down this graph from the top to the bottom. These network editors can consist of many different contexts to reflect the different types of data that can be processed. Of particular interest to this discussion are two contexts: one for manipulating geometry (Surface Operators, or SOPs) and one for setting up simulations (Dynamic Operators, or DOPs).

This is a good time for a quick glossary:

OPs, Nodes: The vertices of the network graph. Each one has a type, which determines what computation it represents. Each one also has a page of parameters (based on the type) which act as extra

inputs to its computation.

Cook: All processing is called cooking. To cooking a node is to run the operation its type represents on its inputs.

Parameters: Each node has an interface defined by name/value/type tuples. Each of these tuples forms a parameter to the node. For example, a Box node may have a parameter to define the size of the box. Its name may be "size", type "vector", and value "1, 1, 1". Other packages use the term "attributes" for a similar concept, which is a bit confusing as attributes refers to something quite different in Houdini.

Geometry: A big bag of primitives. Spheres, polygons, and NURBS can all be jumbled together, along with arbitrary named and typed attributes on the points, primitives, or vertices thereof.

Attributes: Geometry can define extra named data that is attached to all points, vertices, or primitives. Position, colour, normal, and texture UVs are all examples of attributes. Note that all points in a geometry have to have the same set of attributes.

In the picture of the network each of the squares is a SOP node. Each SOP node represents a verb to act on geometry. Each wire represents a path to pass geometry data along. The *torus1* node is a generator – it will just create a polygonal torus according to parameters on the node. The *attribcreate1* node is a filter – it copies the input geometry and manipulates it. In this case, it adds a user defined attribute.

What is Mantra?

In addition to Houdini, Side Effects Software also produces a renderer known as Mantra. This production-proven renderer supports micropolygon, raytraced, and PBR approaches to solving the rendering equation.

What is VEX?

VEX is a shading language similar to the Renderman Shading Language (RSL). It is a software interpreted language so provides the flexibility of scripting. However, it also has an implicitly SIMD evaluation approach that amortizes interpretation overhead.

VEX has moved well beyond just shading, however. It is used as the workhorse for simulation and geometry processing.

Often it is compared to hardware shading languages, such as GLSL, OpenCL, CUDA, or Cg. This is a bit misleading. If they were truly similar, multithreading VEX would be trivial. And moving it onto the GPU would likewise be a manageable task. We will talk more about the difficulties VEX poses later.

Challenges in Multithreading Houdini

It is Old

Houdini 1.0 was released in 1996. Some code, however is even older, dating to the original PRISMS from which Houdini spawned. We thus have a large, mature, codebase to work with. While we have always been tangentially aware of multithreading, we had been able to rely on increasing clock frequencies rather than multiple cores to gain performance. While the assumptions of this period came to a halt in 2007 with the release of the Core Duo architecture, it still leaves us with a lot of inertia.

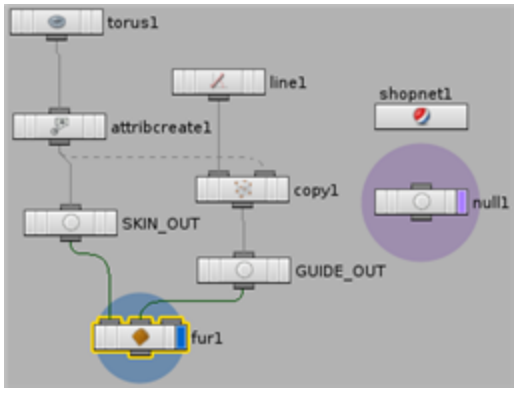
So, what sort of holes have we dug for ourselves over the decades?

It is Interrelated

Every part of Houdini can talk to every other part of Houdini. You can have a geometry operator that creates curves based upon the results of a composite operator. You can have a simulation which is triggered by an event in a different simulation. And the parameters to these operators themselves can introspect the values of other parts of the scene – the effective network topology can thus shift depending on the data passing through the network.

This is also not the exception, found only in esoteric examples. It is a very deep part of how problems are solved with Houdini.

Cooking is Bottom-Up



The natural way to view this network is to imagine the geometry created by *torus1* flowing down through the graph. This is not, however, how it is actually computed. Instead, all computation is done on-demand. When a request is made for the geometry of *fur1*, the *fur1* operator will make its own requests for the geometry of *SKIN_OUT* and *GUIDE_OUT*. Each node will also cache its generated data, avoiding the diamond dependency to trigger two evaluations of *attribcreate1*.

This, unfortunately, results in a sort of uncertainty principle in the network. To know what data is required to compute an operator requires actually computing the operator!

For example, one common operation is the *switch* node. At run time the *switch* node picks one of its

inputs based on a parameter. That parameter, however, may use introspection. For example, consider this expression:

npoints("/obj/geo1/reference") == 0

This expression will ask the node at the path */obj/geo1/reference* to evaluate. Then, it will return 1 if that node has zero points in it (ie, is empty) and 0 if it has some points. The result? The *switch* operator will pick its first input if */obj/geo1/reference* has geometry and the second input if it is empty. The dependency graph of the *switch* node has, through user interaction, become entwined with the evaluation of another arbitrary node. It is not enough to say the *switch* depends on */obj/geo1/reference* (though it certainly does), but the dependencies of *switch* depend on it as well!

Multithreading Houdini

Despite being faced with a huge code base of tightly coupled systems, the state of silicon forces our hand. We have to multithread Houdini. This isn't a one-time process, but a continuing effort. What I present here is an attempt at distilling some guiding principles we've learned over the years.

Methodology

Incremental Changes

Continuous integration is a cornerstone of our development process. Daily working builds that pass all regression tests are a goal throughout a release. This can seem like straightjacket when contemplating significant reforms to the system. But we have found that a large number of small corrections over a long time can result in significant architectural changes.

There is a wonderful feeling in making a one-line change to implement something that, at one point a few years back, seemed to require a complete system re-write.

I don't think there is a simple correct answer to the re-write vs re-factor question. But I will express skepticism if the reason for the re-write is because it is believed that you cannot re-factor. You can.

Death to Globals

Static, my new favorite keyword!

Of course, we all know not to use this. But if you do have old code, you probably do have global variables and static locals.

Finding Globals

You can try grepping your source for statics, but this will not find all of them. It will also falsely report constant statics and static functions, both of which are safe. In Linux you can do:

```
nm libGEO.so | c++filt | grep -i '^[0-9a-F]* [bcdgs]'
```

This will report all the writable variables in the given library. There are still a lot of false positives: compiler generated variables and your own legitimate statics.

```
nm libGEO.so | c++filt | grep -i '^[0-9a-F]* [bcds]' | grep -v
'const::[a-zA-Z]*$' | grep -v '::ignore$' | grep -v openvdb::
| grep -v 'vtable for' | grep -v 'typeinfo for' | grep -v 'VTT
for' | grep -v std::__ioinit | grep -v 'guard variable for'
```

Reasons for Statics

Before one removes something, one needs to know why it was put there. From combing over a large number of statics I found some general reasons why they are used:

Because I Could:

```
float
foo(float x, float y)
{
	static float a;

	a = y * 2;
	return a * x;
}
```

Seriously. I wish this was just one or two circumstances. Was the author trying to save stack space?

Fallback Results:

```
float
foo(float x, float y)
{
	static float lastgood;

	if (!validvalues(x, y))
		return lastgood;

	lastgood = x / y;
	return lastgood;
}


void
program()
{
	float a = foo(3, 4);
```

```
	float b = foo(5, 0);
}
```

I tended to find this pattern in code such as knot insertion. The fix is to force the caller to explicitly track the fallback values:

```
float
foo(float x, float y, float &lastgood)
{
	if (!validvalues(x, y))
		return lastgood;
	lastgood = x / y;
	return lastgood;
}


void
program()
{
	float lastgood = 0;
	float a = foo(3, 4, lastgood);
	float b = foo(5, 0, lastgood);
}
```

A problem can arise if *foo* is called from significantly different parts of the program. In practice I never found this to be a problem. I only found a single case where there wasn't a direct coupling of the invocations that needed the fall-back value. The rgb to hsv and hsv to rgb colour converters tried to correctly cache the hue for de-saturated colours. The idea was that rgb_to_hsv(hsv_to_rgb(0.5, 0, 0)) would properly preserve the hue. However, this pattern fails even if there is no multithreading: consider if you just broke the conversion into two passes across multiple elements!

Co-Functions:

Two different functions may want to share some state, but hide that from the interface seen by the caller. Usually this seems to be a result of pre-reference C code and/or laziness. While this could be solved by judicious use of thread-local storage, I recommend saving that as a last resort. It is much better if you can fix the functions to make their interdependency explicit.

One also wants to fix it in an easy way – you shouldn't suffer because someone else was lazy. (Of course, sometimes a revision-control search reveals the awful truth: that someone else **was** you).

An easy mechanistic solution to this, and to most of the parameter problems, is to make a simple *struct*.

```
float knotspace, knotaccuracy, tolerance;

float setup(float a)
{
```

```
	knotspace = a;
	tolerance = 0.01;
	knotaccuracy = a / tolerance;
}

float apply(float a)
{
	return a * knotspace;
}
```

becomes:

```
struct ApplyParms
{
	float knotspace, knotaccuracy, tolerance;
};


float setup(float a, ApplyParms &parms)
{
	parms.knotspace = a;
	parms.tolerance = 0.01;
	parms.knotaccuracy = a / parms.tolerance;
}

float apply(float a, ApplyParms &parms)
{
	// I wonder why tolerance and accuracy exist?
	return a * parms.knotspace;
}
```

It is then straightforward to build *ApplyParms* on the stack before the first invocation of *setup* and add it to all the *apply* calls.

Callbacks Without void *:

The right thing to do is to change the callback to have a *void* * escape clause. However, this can really, really, mess with your API forcing considerable cascading changes. Thus, sadly, all too often I'd recommend instead just punting and converting your globals to thread local storage.

Callbacks With void *:

While a callback with a void * is no functor, it should be sufficient to let you write code without globals. But still you may find some. The problem is that a very common pattern is to pass *this* into the *void* *. And, then one is tempted to put any extra parameters as members of the class. This pattern seems to work fine until *this* is constant.

```
static float glb_parm;

static void callback(void *vdata)
{
	const MyClass *data = (const MyClass *)vdata;

	data->callback(glb_parm);
}

void program(const MyClass *data, float parm)
{
	glb_parm = parm;


	runalgorithm(callback, data);
}
```

Thankfully the transformation is simple: realize you don't have to pass the class as void *! You can just create another ad hoc struct, build it on the stack, and pass that in.

```
struct CallbackParms
{
	float parm;
	const MyClass *data;
}

static void callback(void *vdata)
{
	CallbackParms *parms = (CallbackParms *)vdata;
	parms->data->callback(parms->parm);
}

void program(const MyClass *data, float parm)
{
	CallbackParms parms;

	parms.parm = parm;
	parms.data = data;


	runalgorithm(callback, &parms);
}
```

Returning const char *:

The history of C++ can be seen as a history of trying to get strings to work in C++. It is thus not unusual to see:

```
const char *
createname(int id)
{
	static char buf[100];
	sprintf(buf, "name_%d", id);
	return buf;
}
```

This is wrong for many reasons. Successive calls will change previous results, for example. But practicality often forces your hand to leave the interface as it is. In this case, thread local storage is the best answer.

Alloc Optimization:

A common premature optimization is for innermost functions to build their own cached allocation:

```
void
munge_array(float *data, int len)
{
	static float *temp;
	static int templen;

	if (!temp || templen < len)
	{
		if (temp) free(temp);
		temp = (float *) malloc(len * sizeof(float));
		temple = len;
	}


	// Munge data using temp for temporary…
}
```

The first thing to identify is if this optimization is even needed. While allocation is expensive, it isn't that expensive! I usually found in code like this the cost of allocation was not an issue.

If it is a problem, that is what *alloca* can be used for. *alloca* allocates variable data on the stack. Unfortunately, your stack is limited, so you will have to switch between *alloca* and *malloc* depending on size. The theory is that if you are allocating something substantially large, your time will be dominated by whatever you are doing with that thing, not by the allocation call.

With Houdini we also have a *UT_StackBuffer* class which handles this by having its own default buffersize array as member data. If the requested size exceeds this, it allocates, otherwise returns its own member data.

```
void
munge_array(float *data, int len)
```

```
{
	UT_StackBuffer<float> temp(len);

	// Munge data using temp.array() for temporary…
}
```

Lookup Tables:

A lookup table in a function is usually static to avoid it being constructed on the stack with each call. In this case just make it const. Note that this requires two const with strings as you have to make both the table and the strings const.

```
static const char *const namelist[] = { "name1", "name2", 0 };
```

Focus on Stupidly Parallel

There are two distinct regimes that need optimization

First, is interactive performance. This occurs as an artist prototypes a shot. Evaluation times need to be fast. Geometric complexity is kept low. Amdahl's law comes into effect and you find yourself needing everything parallelized before you gain significant performance.

The second is behavior at scale. This occurs when an artist scales their prototype with production data. Usually, this follows the "The reward for good work is more work" principle. The faster you process the data, the bigger the data set you will be given.

The time to compute a frame can be broken down into two components. The first is the cost to determine what operations need to be done to what data. This is the cost of walking the network graph and evaluating the parameters. The second is the cost of actually performing the operations. The first cost is constant regardless of data size, and the second is hopefully linear in data size.

As we began to multithread Houdini, we focused on the behavior at scale. While improving interactivity is a very worthy goal, it is also considerably more difficult. Further, at the end of the day, the shot still needs to scale.

UT_ThreadedAlgorithm

If you look at the cause of global variables, you will notice one common motif. Laziness. Or maybe, less pejoratively, "expedience". It is very common for developers facing real pressures to trade off code quality for development time. This is why the stupidly parallel should be stupidly easy to parallelize.

Fortunately, this has grown a lot easier over the years. Tools like TBB certainly make things easier: tbb::parallel_for is preferable to building your own dedicated thread pool. However, it is still a general purpose tool. It is worth the time to build a wrapper which matches the data sets you have and the coding style you have. For example, we have wrapped parallel_for in several ways:

UTparallelFor: A generic implementation that lets you specify an explicit grain size and subscription ratio. Has early exit code to inline the body for small tasks.

UTparallelForLightItems: A wrapper of the generic version that sets a grain size of 1024.

UTparallelForHeavyItems: Another wrapper with a grain size of 1.

UTserialFor: A single threaded version.

Having a *UTserialFor* construct is essential. This lets you trivially swap between threaded and non-threaded implementations of an algorithm, very useful for quickly determining if a bug is due to multithreading. And if it is, which algorithm is responsible.

Because we started our parallel push before TBB was released, we did build another approach for the quick transformation of functions from serial to parallel. A common coding convention in Houdini is to use the member functions of a class to process the data of the class. Thus, the *UT_Vector* class that represents arbitrarily dimensioned floating point vectors has a *addScaledVec* method which will, as its name suggests, add another vector while scaling it.

The serial code, roughly, was:

```
class UT_Vector
{
	void addScaledVec(float s, const UT_Vector &v);
};

void
UT_Vector::addScaledVec(float s, const UT_Vector &v)
{
	exint	i, end;

	end = length();
	for (i = 0; i < end; i++)
		myVector[i] = s * v.myVector[i];
}
```

We were faced with scores of functions like this that we wished to multithread. To do this, we defined a set of macros that would, given a function signature, build the appropriate functors to invoke the function. One would just have to provide a Partial version of the function that knows how to complete the computation for a single thread. The transformed code is:

```
class UT_Vector
{
	bool shouldMultiThread() const
	{ return length() > 5000; }

	THREADED_METHOD2(UT_Vector, shouldMultiThread(),
```

```
                        addScaledVec,
                        float, s,
                        const UT_Vector &, v);
        void addScaledVecPartial(float s, const UT_Vector &v,
                        const UT_JobInfo &info);
};

void
UT_Vector::addScaledVecPartial(float s, const UT_Vector &v,
                                const UT_JobInfo &info)
{
        exint        i, end;
        info.divideWork(length(), i, end);
        for ( ; i < end; i++)
                myVector[i] = s * v.myVector[i];
}
```

The macro makes two new functions, *addScaledVec* and *addScaledVecNoThread*. The latter acts like the *UTserialFor* in providing a simple way to turn off threading from the caller. Being able to disable threading when invoking is also very useful when your function is only sometimes threadsafe. If you can determine at runtime if the threadsafe conditions are met, you can invoke the parallel version.

Note the use of the *shouldMultiThread* function. Any valid bit of C code can be used in the macro at that point, but usually a data structure has a consistent break-even point making it easier just to move it to a function.

This is a thread-based approach, not a task based approach. The *addScaledVecPartial* is ideally called once for each thread and it is expected in its body to figure out the partitioning. The *UT_JobInfo* assists with this by providing the number of jobs actually run and which job this invocation represents. Further, it also provides a shared lock to be used for synchronization between the threads.

A big problem with the sort of partitioner used in this example is that there is no room for load balancing. If one thread encounters a particularly sticky bit of computation the entire system will wait for it. To ameliorate this, the *UT_JobInfo* also contains an atomic integer that can be used to implement a simple task-stealing system. When processing volumes, for example, this integer is used to control which 16^3 block each thread processes.

One advantage of this thread-based approach is that you don't have to pass all of your algorithm state into the tasks. Often there is common code outside of the loop – since there is an upper bound on the number of threads invoked, you can leave this common code inside the *addScaleVecPartial* method. As your data set scales, the contribution of this will remain constant and hence diminish. I have found one virtue of this is that it makes it a lot easier to port algorithms that did not plan on being parallel. Often the interface into the function is clean, but the number of parameters going into the for loop are considerable.

It is a dangerous trap, however, to let this go too far.

VEX

Our VEX language was one of the first things to be parallelized in Houdini. Its SIMD nature meant that splitting across cores was – for simple code – easy. One of our strategies is thus to simply write as many algorithms as possible in VEX rather than in C++. As an added bonus, both VEX and its node representation VOPs are integrated directly in Houdini. Development can be done in a live session.

Patterns

Errors to Avoid

Reentrancy

Always be reentrant. Do not use non-reentrant locks. It is a question of when, not if, that someone innocuously wraps your locked function in another locked function.

```
class foo
{
public:
	void interface1()
	{
		UT_Lock::Scope scope(ourlock);
		// Do computation.
	}
	void interface2()
	{
		UT_Lock::Scope scope(ourlock);
		// Reuse interface1
		interface1()
	}
};
```

One could build coding rules to avoid this. But what is the dead-locker guilty of? Trying to factor code! We want people to factor more code, not be punished for it.

If your concern about reentrant locks is performance, then do not be concerned. You already gave up on performance when you added the lock.

Sleep

Use condition variables to sync threads. Do not just wait.

Performance Traps

Never Lock

When I received formal instruction in concurrent programming, the course consisted of solving twisted multithreaded problems through the use of locks and semaphores. In practice, if you find yourself building such a system, you probably have already lost.

First, and probably most importantly, it is ridiculously difficult to get right. With production code, it is also not sufficient to make code that you can get right. You have to make code that everyone else who touches it will also get it right. Just like "clever" algorithms that create a twisted maze of code should be avoided, so should "clever" threading patterns.

Even if you do get the locking right, however, you face another serious problem. Performance.

When writing CPU-based multi-threaded code it is tempting to build a pristine mental model of how your code will execute. You can imagine your six-core chip running each of your six threads in, if not synchronicity, at least somewhat together. It does not help a lot of work on parallel algorithms was done on dedicated hardware where this assumption isn't unreasonable. On a desktop, however, things are different. You have no way of knowing how many "real" CPUs you have, or how many are "hyper threads". You have no way of knowing if threads will stay on the same physical CPU from timeslice to timeslice. You are not the only process requesting timeslices – the music player, the web browser cycling a forgotten flash animation, the other copy of your application that was backgrounded waiting a completion of its task. While it can be argued that there are ways to answer some of these questions, I would contend that it is best not to. Instead, your application must be tolerant of this environment and thread efficiently within it.

Of course, high level locks are essential. Never lock is a guideline for how you approach the part of your algorithm you expect to actually scale. A heavily locked algorithm will allow you to peg your CPU monitor to 100%, but a close inspection will show 50% of that being spent in the kernel. I color the kernel times in my CPU monitor red, as opposed to blue for user computation, so these occurrences show up clearly.

Atomic are Slow

One way to take some locks out of your code is to replace them with atomic operations. Keep in mind, however, this is just moving the locking effect to hardware. It certainly is faster and preferable to using OS locks, but you should treat it as an uncached memory operation.

```
class FOO_Array
{
public:
	shared_ptr<float *> getData() const
	{ return myData; }
private:
	shared_ptr<float  *> myData;
};
```

```
float
sum_array(const FOO_Array &foo)
{
       float total = 0;
       for (int i = 0; i < foo.length(); i++)
              total += foo.getData()[i];
}
```

shared_ptr is a powerful new tool in C++. It is often used as a way to abdicate responsibility for tracking ownership of data. The *FOO_Array* doesn't have to worry if the caller destroys it after fetching it's data – the fetched data will still be valid. This, however, is not without a cost. Because *shared_ptr* is threadsafe, it needs to do some form of atomic operation to track if it is still unique or not. If we were to convert *sum_array* into a threaded invocation we'll be facing the worst-case contention for that atomic structure as every addition is copying the *shared_ptr* and hence changing the atomic!

Watch your Grain Size

Never fork without checking grain size. The caller should be forced to explicitly think about it.

It can be very easy to accidentally make things very slow by adding a multithreading code path. At the time one probably is testing with a large dataset, so can miss out on how things behave with simple datasets. This problem can show up at scale, since some times scaling up a situation involves processing a small dataset millions of times.

The attributes on a piece of geometry are independent of each other. When duplicating a piece of geometry, each point attribute can thus be duplicated independently. This is a straightforward place to do a bit of multithreading: *parallel_for* across the attributes. However, it is quite common to have a simple object have scores of attributes. Each attribute has the data only for a dozen points, but we still pay the huge overhead of creating and dispatching tasks to duplicate each attribute. The threshold of when to split up this task can't be determined by just looking at the number of attributes on the geometry, it also has to take into account the number of points.

Practical Tips

Command Line Control

All standalone applications should have command-line control of their maximum thread usage.

We would encourage the use of *–j*, where j stands for jobs. This is inspired by *make* and provides a consistent way for end users to create scripts to limit your thread usage.

A good selfish reason to do this is for debugging. When a troublesome file shows up, you can easily run both with and without threading to determine the locus of the fault.

A practical reason is for speed. Multithreading is less efficient than single threading. If memory

resources permit, it is usually more efficient in terms of throughput to single thread your program. The assumption underlying this counter-intuitive result is that there is not just one job to do. On a render farm with thousands of frames to process it often would be faster to run six frames at once, each single threaded, than try to balance one frame across six cores. Naturally, there are exceptions and trade-offs, as one balances memory use, network bandwidth, and artist turn-around time. By providing a command line thread control you make it very straightforward for users to adjust your programs behavior to what they have found works best for their farm and their shots.

Constant Memory vs Cores

When faced with a many tasks that want to write to the same object, there are several approaches you can take.

Lock the Writes: Each write to the structure can acquire a write lock. However, locks are slow, so this is only viable for the most lightweight writes.

Theadsafe Write Pattern: You can arrange the task break down so no two tasks will write to the same part of the object. For example, our *UT_VoxelArray* is broken into 16^3 tiles. It is not thread safe for two threads to write to the same tile, however, it is safe for them to write to different tiles. Thus, by ensuring the tasks break up along tile boundaries, we can ensure the ensuing writes are all safe.

Copy and Merge: Each task can make its own copy of the object. It can then write to its copy safely. A post-process can reduce the copies back into a single version.

This last process is the focus of this discussion. It commonly shows up in scatter-gather problems. A canonical case is the stamping of points into a volume. Because accumulating points is symmetric, we can make an empty volume for each task. That task can stamp its points into it to get a partially filled volume. These volumes can then be composited together to produce the result.

One obvious efficiency is that we do not need a volume per task, we only need a volume for each thread. With the thread-based parallelism of *UT_ThreadedAlgorithm* this is straightforward – we know the prologue of each invocation is only run once per thread, so we can allocate the volumes there. Even with a purely task-based system this is still possible. One can make a thread-local variable to store the volume. Each task will use this to create and fill. After completion, you can iterate over all the thread specific values of this variable to merge and clear out the volumes.

This approach seems simple and straightforward. It also will seem to work in a lot of cases. However, it has a large pitfall looming in front of it. What happens when we run this algorithm on a four-socket, ten-core, machine with hyperthreading? With 80 threads active you may see an 80x peak in your memory usage!

The solution is to ensure sparsity in your replicated data. Either what you replicate has to be constant in your data-set size, or it has to be sparse enough to not scale with the number of cores after the data size is factored out. For example, because of tiling, an empty volume does not take the same space as a full

volume. So if each thread were assigned a subset of tiles, and instead of only iterating over a subset of particles for each thread, we iterated over all particles, we could ensure our number of live tiles does not grow. Of course, this also has a significant cost of having to read all of the points through all threads – perhaps a bucketing pass ahead of time might help. Also, if your tiled volume allows independent writes to independent tiles, you don't need to make the copies at all as this has been then changed into the threadsafe write pattern.

Memory Allocation

Traditional memory allocators lock. As such, *malloc* and *new* are not things you want to see in the innermost loops of your threaded algorithms. Of course, this is a rather general principle that applies to non-threaded code as well, so it usually is not a problem.

But what if you really need to allocate memory? What do you do when *alloca* does not suffice? Traditionally the answer was to write your own small object allocator. More recently, we've seen things like *tbb::scalable_malloc* provide ready-made solutions for highly contentious allocations. Unfortunately, with memory allocation we are not just facing the threat of slow performance. Memory fragmentation is a very serious problem that can easily halve your effective working set.

Thankfully there is an easy solution: use *malloc* and *new* as normal, but link against *jemalloc*. *jemalloc* replaces your standard allocator with one that does all the proper tricks of small object lists and thread local caches, but it does it in a way which aggressively avoids fragmentation.

Copy on Write

Ownership is Important

A greatly misunderstood feature of C++ is its lack of garbage collection. This is derided as a foolish decision based on antiquated notions of efficiency. Memory leaks and dead pointers are said to abound in C++ code, leading to crashing programs and inefficient software. While there is a lot of truth to these objections, there is a silver lining to this cloud. Without a garbage collector to use as a safety-net, C++ programming idioms have developed to ensure clear ownership of objects are tracked. Techniques like RAII solve many of the pitfalls of manual memory management without losing this key advantage.

Having a clear understanding of object ownership and lifetimes solves a lot of problems. A common example is disk files – who should write to them and when they should be closed is straightforward in an ownership based model. I also contend that this sort of thinking can help solve multithreading problems in a way that minimizes locks.

This does require one to avoid Java-style ownership, where every reference to an object is an owner of that object. In particular, the *shared_ptr* device, while incredibly useful, should be kept to a minimum. Consider again our simple array class:

```
class FOO_Array
```

```
{
public:
	shared_ptr<float *> getData() const
	{ return myData; }
private:
	shared_ptr<float  *> myData;
};
```

We saw earlier how this can result in problems with multithreading; but this is also an important conceptual problem. Who owns myData? Why does someone who wants to inspect myData have to acquire ownership? Usually the argument is made that the caller doesn't know the lifetime of the *FOO_Array*. However – it does! It has must already hold a reference to the enclosing *FOO_Array* or it wouldn't be able to invoke the *getData* function. It is only if it is planning keeping the returned pointer beyond the *FOO_Array*'s guaranteed lifetime that it would require a *shared_ptr*. But, in this case, we are conceptually caching the result of the call, so having to explicitly signal this by gaining ownership is not surprising.

```
class FOO_Array
{
public:
	float *getData() const
	{ return myData.get(); }

	shared_ptr<float *> copyData() const
	{ return myData; }
private:
	shared_ptr<float  *> myData;
};
```

Here we have made this transformation explicit: the caller invokes *copyData* if they want to maintain the data beyond *FOO_Array*'s lifetime, and *getData* if they merely want to inspect it locally.

Reader/Writer Locks

The most common pattern one encounters when attempting to write stupidly-parallel code is the reader/writer problem. You have a data structure which you want to allow many threads to read from in parallel. But you also potentially want many threads to write to it. Obviously, if you write to it while threads are reading, you will end up reading inconsistent state. Similarly, if the structure has caches, you may not even be able to read from it simultaneously. This creates a simple to describe, but hard to implement, hierarchy of locks. Read locks are acquired whenever a thread wishes to read, and write locks when they want to write. Read locks can prevent the write lock from acquiring (ensuring information doesn't change underneath them) and can have a different lock semantic (for example, allowing many readers at once).

As a concrete example, consider a single frame of geometry. This geometry will contain, among other

things, a large array of point positions. We would like to be able to read these point positions from many threads safely. But we also would like to be able to update them in a deformer, and do so in a way that won't cause any reader to see an inconsistent state. Most importantly, we want to do this in a lock-free manner.

Const Correctness

A lot of this course is focused on how to get things working with old code which often has done things the wrong way. Thus, it is a bit of a cheat to bring this up. However, in my defense, Houdini is old code and yet it is const-correct.

Const is one of my favorite keywords in C++. It exemplifies what the language does right: allow you to build contracts that the compiler will enforce. Our main motivation for const correctness was driven by the belief that compilers would be able to use this information to better optimize the code. I am still unsure if this has ever occurred, but the rewards we reaped in code maintenance easily justified the continued use of this practice.

With single threaded code the const keyword is a contract to the caller that the invoked code is side-effect free. This greatly simplifies understanding the code. Having the compiler enforce it is essential. While there are *const_cast*, *mutable*, and *C-casts* to worry about, in practice these are remarkably rare exceptions. Instead, most lazy/harried programmers will just opt to drop the const altogether rather than use one of these workarounds. This has the beneficial effect that code with the const keyword can be trusted to be const, because it usually was added in a bottom-up fashion by someone carefully performing code-hygiene, not in a rush by someone trying to meet a deadline.

As the use of multithreading spreads through our code base, the const keyword becomes an essential tool. Not only can it be used to imply the code is side-effect free, it can also be used to imply the code supports many-readers. Again, care must be taken due to *mutable* caches or global variables, but it allows the easy validation of large swathes of code. Further, by ensuring const structures are sent to the multiple threads, you can have the compiler enforce that you don't accidentally call an unsafe function.

Sole Ownership is a Writer Lock

Many readers, in the absence of writers, can be implemented without locks by ensuring all the functions used by the readers are threadsafe. Using the const keyword, you can get the compiler to help validate this requirement. But what happens when someone wants to write to that data?

A reader/writer model usually has a way to keep track of the active readers. This is essential to detect if it is safe to start writing. We want to avoid that, however, since we want reads to be lock-free. Our solution is to cheat and redefine our problem. By solving a slightly less general problem, we can have an efficient solution that avoids any locking on the part of the readers.

When designing a multithreaded algorithm writing to a shared data structure, there are two types of readers to worry about. The first are the reads my own algorithm will generate. I can reason about these

and create solutions for my planned read pattern. I don't need special reader/writer locks, I instead just need safe access patterns. The second are the reads that other algorithms running concurrently may generate. This is the scary case that I cannot reason about or predict.

How then do we detect if there are any external readers? If we aren't tracking individual read requests, we can only detect if there is the possibility of external readers. For an algorithm external to us to read from a data structure, it must have a way to point to or reference that data structure. Our concept of data ownership now provides a solution: for someone else to be able to unexpectedly read from the structure, someone else must have ownership of the structure. After all, if they have not acquired ownership, they have no guarantee of the lifetime of the object, so shouldn't be reading from it anyways!

Our write-lock problem is hence simplified. Before we can write to a potentially shared data structure, we have to first ensure we have sole-ownership. Provided we are the only owner, we know no other system can gain ownership – after all, we have the only reference! Provided we've ensured all caches properly "own" the object, we have no fear of surprisingly increasing our ownership count because there can be no references to our object outside of our algorithm.

So what is to be done if we want to write to an object and discover its ownership is shared? We simply copy it. In almost all of our use cases, the share is due to a cache, in which case a copy is the right thing to do in any case – it is unexpected for the cached version to be updated. Even if we do want to update the shared object, however, it is still semantically correct to work on a copy. This means other threads will see the old version until the new version is posted, but this is almost always advantageous since we have eliminated risks of inconsistent states mid-algorithm. Further, we can always imagine that all the would-be readers happened to stall until the algorithm was complete, so this instant posting is something the overall system should be able to handle.

This pattern is equivalent to Copy On Write, which is often used as a way to share memory. We have found, however, it is an effective way to manage multithreaded access to shared structures. Again, let us look at the *FOO_Array* built in this fashion.

```
class FOO_Array
{
public:
	const float *readData() const
	{ return myData.get(); }

	float *writeData()
	{ makeUnique(); return myData.get(); }

	shared_ptr<float *> copyData() const
	{ return myData; }

private:
	void makeUnique()
	{
```

```
            if (myData.unique()) return;

            shared_ptr<float *> copy(new float*[size];);
            memcpy(copy.get(), myData.get(),
                       sizeof(float)*size);
            myData = copy;
        }


        shared_ptr<float  *> myData;
};
```

First, note that we've made the *readData* function const correct. It returns a *const float* * making it more difficult for people to accidentally write to shared data if they were given a *const FOO_Array*. If we do want to write to the data inside the *FOO_Array*, we have to go through the non-const *writeData*. It guards all access with *makeUnique* invocation to ensure the caller is the only owner of the underlying data. Our claim is that after the *makeUnique* call we will be the only owner of the underlying data.

It is important to note that the uniqueness is not guaranteed by the code. A malicious caller can easily stash a pointer to the *FOO_Array* else-thread and call *copyData* to violate this assumption. However, it is guaranteed provided the ownership model is respected. This is the same sort of contract that already exists to avoid memory leaks, and the same sort of coding practices can be used to ensure there are no surprising behaviours.

While our ownership contract ensures there can be no surprising increases to the *unique* count of *myData*, it says nothing about surprising decreases. As such, after the *unique* call and until the assignment to *myData* it is possible another thread will decide it is done with its copy of the data and leave this as the only copy. In this case, however, the only penalty is an extra copy being made. Further, the extra copy is something that, but for the timing of threads, may have been required anyways! As a result, the actual *unique* invocation can be very weak. Since it may be invoked a lot, it is useful to make it no stronger than a volatile.

Failure Mode of This System

Using Copy on Write to solve the reader/writer problem is not without its own pitfalls. As expected from a system that requires a contract with the programmer, it is quite possible for the contract to be violated and things to fail. One should thus, of course, ensure this is as transparent as possible to end users.

The main problem we found ourselves running into with this approach is when we were too liberal in providing ownership. It is tempting with *shared_ptr* to fall into a Java-style model of programming where everything is owned by everyone. Not only does this result in a lot of unnecessary atomic operations, but with copy on write it can result in writes disappearing into the ether.

For example, consider this multithreaded code:

```
void applyPartial(FOO_Array foo, RANGE partialrange)
{
       float *dst = foo.writeData();

       for (i in partialrange)
       {
              dst[i] *= 2;
       }
}

FOO_Array bar;

invoke_parallel(bar, applyPartial);
```

Here we have treated *FOO_Array* as a lightweight container so have passed it by value to our threaded tasks. This would work with the original definition of a *shared_ptr* reference to *myData*, but now that we are using copy on write the pass-by-value means that the caller will see an unchanged *bar*. Each of the tasks will instead build their own copy of the array, write to that, and then delete the copy.

While that example can be solved by proper use of references, there are some more nasty situations that can develop if some care is not taken.

```
void apply(float *dst, const float *src)
{
     for (i = 0; i < size; i++)
     {
          dst[i] = src[i/2];
     }
}

void process(FOO_Array &foo)
{
     const float *src = foo.readData();
     float *dst = foo.writeData();

     apply(dst, src);
};
```

This contrived example of pointer aliasing has a few problems. First, whether *src* == *dst* depends on the share count of the incoming *foo* reference. If *foo* is already unique, the *readData* and *writeData* will report the same pointer and we will get the expected aliasing. However, if it were shared, *writeData* will cause *dst* to have a duplicate copy, leaving us with two independent blocks of memory. This is not the most serious problem, however. Consider if *foo* was shared during the invocation, but the other copy was released after the *readData* and before the *writeData*. After the *writeData* completes its copy it will free the original data as it now is unshared, leaving *src* pointing to freed memory.

Using copy on write to solve the reader/writer problem is not a silver bullet. It definitely does require some additional care and code hygiene. However, these are not too much more onerous than the requirements already posed by lacking garbage collection, so the techniques should be familiar and accessible.

War Stories

isMainThread

Due to the single-threaded ancestry of Houdini, we use the same thread for drawing the UI as we use for computation. During long computations a callback is invoked to see if the operation should be interrupted. This callback also draws to the screen to update the user of the progress.

Open GL, however, does not appreciate it if two different threads write to the same Open GL context. When we started multithreading we started to get random crashes when one of the task threads decided to update the screen. Our solution is to register one thread as the primary, main, thread. All calls into drawing then can be gated with a query of *UT_Thread::isMainThread* to verify it is the correct thread making the call.

Task Locks

A reentrant lock is a lock that allows the same thread to acquire it multiple times. We have found, however, that there is a further generalization of a reentrant lock.

While to the OS all of the threads in a program are equal, they are not the same semantically. One example would be the main thread that has special permissions for writing to Open GL. Another is a pool of worker threads working on the same task – these threads have a much tighter relationship with each other than to other threads in the system.

One thread often idling in Houdini is the Python thread. This thread runs a Python interpreter. It is able to query the geometry in a network. However, if the geometry is being updated by SOP cooking, it is important that it blocks until the SOP cooking is complete. We have a lock around SOP cooking that prevents two separate threads from triggering cooking. This works well with a main thread and a python thread. But what happens when we have a thread pool performing the cooking?

A SOP operator can be defined in the VEX programming language. Many functions in VEX allow users to query the geometry in the scene – an action which will trigger cooking. We can't build this dependency ahead of time because it is a result of arbitrary computation within the VEX code. When we do trigger the geometry cook we will want to ensure only the worker thread that started this computation can proceed – other worker threads should block to avoid double-processing. However, the thread that first encounters the dependency may not be the same thread that grabbed the original SOP cooking lock! If this happens, it will deadlock.

Our solution was to create a higher-level reentrant lock, a task lock. Whenever we assigned worker

threads to tasks they will acquire that task's id. This id can then be used to allow the thread into a task lock. When the python thread tries to acquire the lock, it has a different task id so is blocked. When one of the worker threads first tries to acquire it, it can be allowed through and a second internal lock used to ensure only a single worker thread is performing the task at once.

This worked well when we used a fixed worker-pool model for multithreading. Each job that was threaded would be split into a fixed number of threads from a static threadpool. So, if the cooking invoked by the VEX operation itself triggered a multithreaded code path, the effect would be single threaded execution as all of the other worker threads would be stalled on the cook lock.

We then switched to using TBB's task scheduler. This has the advantage of properly handling multiple multithreaded executions as it abstracts away the handling of the thread pool. Unfortunately, it also exposed a potential deadlock in our system.

If cooking a VEX SOP triggers the cooking of a second VEX SOP, that second VEX SOP will put all of its tasks onto the TBB scheduler. Most of the time the scheduler is empty at this point – the standard VEX procedure is to produce one task per worker thread, so the first invocation will have N-1 tasks blocked and the Nth task processing the queue of N new tasks. However, what happens if the second VEX SOP now invokes a third? The third SOP will again queue N tasks onto the scheduler. Provided these are all processed before we return to the tasks from the second SOP, everything will work. However, TBB is unaware of this dependency so may process one of the second SOPs tasks before the new ones are complete. This task will then try to cook the third VEX SOP and deadlock on acquiring the cook lock.

The right solution for this is something we can't do in TBB. (Or maybe we can now, I'd love to be corrected!) And that is yielding.

Yielding

Mantra, our rendering solution, is also multithreaded. But surely this is a more trivial example? Divide the screen into buckets, make each bucket a task, and let TBB handle the load balancing for you!

This would work but for one problem: shared acceleration structures. Again, this is a delayed dependency problem. We don't know when we start the render which acceleration structures are needed. They are often built on demand when a ray deigns to intersect their area of influence. Because they have significant memory footprints, we don't want to duplicate them per thread. Our usual answer was to just put a lock around the creation of them.

Building these structures can be slow, however. We'd see frames freeze at a single core for minutes before forking into full-CPU utilization. Clearly we want to multithread the generation of these structures!

Our first approach was to use TBB tasks. This, however, fails for the similar issues that the SOP/VEX problem encounters. The thread whose bucket that first hits the acceleration structure can process the new tasks just fine. But the other threads who run into this structure have a tough choice facing them.

They want to block and wait for the structure to build. But then we'd have at most single threaded performance as all the other buckets would soon block on this thread and not release their computation to the construction algorithm. Even worse, if the thread that is building the tree should happen to pick-up one of the bucket tasks rather than its tree building task, it could find itself deadlocking.

We could cancel the bucket's computation. That bucket could be re-enqueued with the proper dependency. We would lose all of the computation that had been done so far, however, an unattractive prospect.

Instead, what we'd prefer to do would be to still block on the building lock, but then yield our worker thread back into a new thread pool owned by that building lock.

Unfortunately, we have not yet built a way to do that. Our solution is thus a bit more primitive.

Since scheduling the rendering of buckets is a rather straightforward task, we chose to take that out of the control of TBB. We built an explicit pool of worker threads to be our main bucket-thread pool. These threads can then invoke TBB's task scheduler and use an additional set of threads for any such tasks. This oversubscription is not as bad as it sounds as in the usual case the bucket threads will quickly all block if the TBB threads are active building something.

Ray Intersect

The *GU_RayIntersect* structure in Houdini is one of those gnarly old pieces of code that one love's to hate. Even when it doesn't work, it doesn't work in peculiar ways that some other code no doubt depends on. So, when we wish to make it threadsafe, we suffer the usual painful constraints of having to retain the original behavior.

Internally it stores a spatially partitioned tree of intersection primitives. A primitive, however, can end up in multiple nodes of the tree. When the spatial partitioning splits a primitive, the primitive is added to both halves. To avoid double intersecting the same primitive, each ray is given a unique serial number. Then each primitive stores its own number tracking the serial number of the last ray that hit it. Obviously, if two threads tried to send two different rays into this structure at the same time, chaos would ensue.

We could add thread local storage to each primitive to store a serial number per thread. Accessing this is not cheap, however, and in the case of our 80 thread machine, its memory requirements can also add up.

Another approach is to have the caller provide an array to store the primitive serial numbers. This requires the primitives to know their primitive number in this list. As it happens, we have a convenient structure that stores the hit information, *GU_RayInfo* which could be transparently extended to store a serial number list.

We would fall afoul of this code, however:

```
GU_RayIntersect inter;
```

```
for (int i = 0; i < 10000; i++)
{
       GU_RayInfo   info;
       inter.sendRay(info, …);
}
```

Because the *GU_RayInfo* used to be a light structure people may have put it in inner loops. Initializing the serial list is O(N) in the number of primitives, unacceptable when *sendRay* is supposed to be O(lgN).

Our solution was to add a thread-local cached serial list to the *GU_RayIntersect* structure itself. The *GU_RayInfo* still has a pointer to a serial list, but it is a pointer to one owned by the *GU_RayIntersect* call and just acts as a cache to both avoid hitting thread local storage and to pass the list down to the innermost intersection routines. Unlike storing thread-local storage on the primitives, our extra memory for the thread-local copies of the serial list is only used when we actually use multiple threads.

GPU rigid body simulation using OpenCL

Erwin Coumans, http://bulletphysics.org

Introduction

This document discusses our efforts to rewrite our rigid body simulator, Bullet 3.x, to make it suitable for many-core systems, such as GPUs, using OpenCL. Although OpenCL is thought, most of it can be applied to projects using other GPU compute languages, such as NVIDIA CUDA and Microsoft DX11 Direct Compute.

Bullet is physics simulation software, in particular for rigid body dynamics and collision detection in. The project started around 2003, as in-house physics engine for a Playstation 2 game. In 2005 it was made publically available as open source under a liberal zlib license. Since then it is being used by game developers, movie studios and 3d modelers and authoring tools such as Maya, Blender, Cinema 4D etc.

Before the rewrite, we have been optimizing and refactoring Bullet 2.x for multi-code, and we'll briefly touch on those efforts. Previously we have worked on simplified GPU rigid body simulation, such as [Harada 2010] and [Coumans 2009]. Our recent GPU rigid body dynamics work has approached the same quality compared to the CPU version.

The Bullet 3.x rigid body and collision detection pipeline runs 100% on the GPU using OpenCL. On a high-end desktop GPU it can simulate 100 thousand rigid bodies in real-time. The source code is available as open source at http://github.com/erwincoumans/bullet3 . Appendix B shows how to build and use the project on Windows, Linux and Mac OSX.

Bullet 2.x Refactoring

Bullet 2.x is written in modular C++ and its API was initially designed to be flexible and extendible, rather than optimized for speed. The API allows the user to derive his own classes and to select or replace individual modules that are used for the simulation. A lot of refactoring work has been done to optimize its single-threaded performance, without changing the API and data structures.

- Use very efficient acceleration structures to avoid doing expensive computations
- Incrementally update data structures instead of computing from scratch
- Pre-compute and cache data so that results that can be reused
- Optimize the inner loops using SSE and align data along cache lines
- Reduce the amount of dynamic memory (de)allocations, for example using memory pools

We ported Bullet 2.x to Playstation 3 Cell SPUs. This required some refactoring and we re-used some of this effort towards a basic multithreaded version that was cross-platform using pthreads and Win32 Threads.

Bullet 3.x Full Rewrite

It became clear that the Bullet 2.x API, data structures and algorithms didn't scale well towards massive parallel multi-threading. The following sections explain some of the issues we ran into. To be future proof, we started to invest in GPGPU technology, with the expectation that this effort would also help towards CPU multi-threading with a larger number of cores.

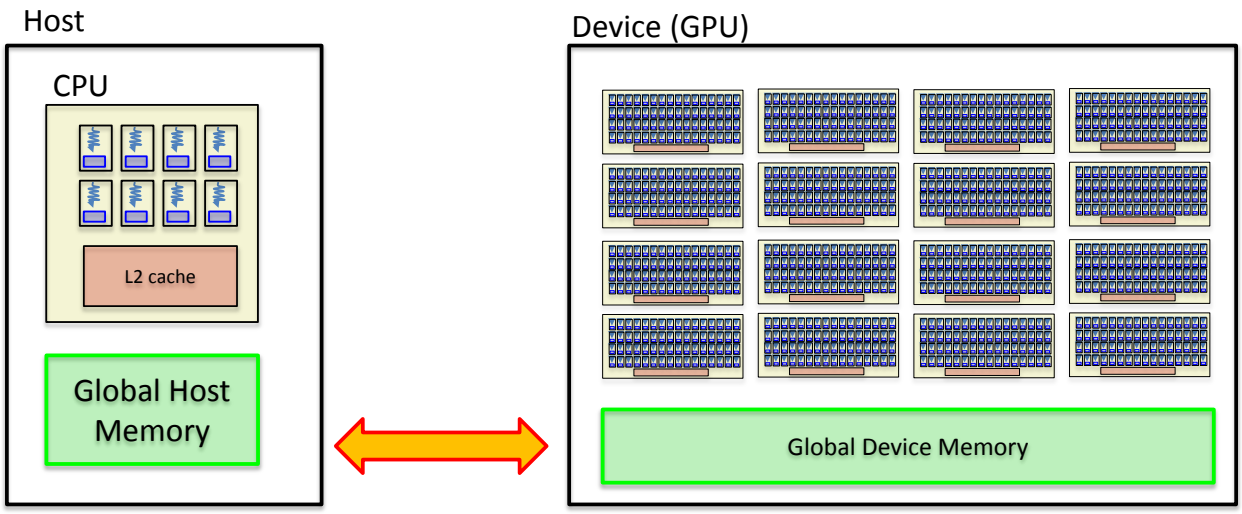
Optimizing for GPU requires more changes to code and data, in comparison to CPU multithreading. This document goes into details of this on-going work on Bullet 3.x.

Getting started with OpenCL

You can use OpenCL to parallelize a program, so that it runs multi-threaded on many cores. Although OpenCL seems primarily suitable for GPUs, an OpenCL program can be compiled so it executes multi-threaded on a CPU target. The performance of an OpenCL program running on CPU is competitive with other solutions, such as pthreads or Intel Thread Building Blocks. There are even implementations that allow to run OpenCL programs distributed over a network using MPI. So OpenCL is a very flexible and cross-platform solution.

OpenCL terminology

If we target a GPU Device to execute our OpenCL code, we still need a Host such as the CPU for initialization of the device memory and to start the execution of code on the device.



The OpenCL code that runs on the Device is called a Kernel. OpenCL kernel code looks very similar to regular C code. Such kernel code needs to be compiled using a special compiler, that is usually provided by the Device vendor. This is similar to graphics shader compilation, such as GLSL, HLSL and Cg.

To get access to OpenCL we need at minimum the OpenCL header files, and some way to link against the OpenCL implementation. Various vendors such as AMD, Intel, NVIDIA and Apple provide an OpenCL software development kit, which provides those header files and a library to link against. As an alternative, we also added the option to dynamically load the OpenCL dynamic library and import its symbols at run-time. This way, the program can continue to run, even if OpenCL is not installed.

Our first OpenCL kernel

Let's start with a very simple example that shows the conversion of some simple code fragment into OpenCL kernel.

```
typedef struct
{
        float4 m_pos;
        float4 m_linVel;
} Body;

void   integrateTransforms (Body* bodies, int nodeID, float timeStep)
{
        for (int nodeID = 0;nodeID<numBodies;nodeID++)
        {
                if( bodies[nodeID].m invMass != 0.f)
                {
                        bodies[nodeID].m pos +=  bodies[nodeID].m linVel * timeStep;
                }
        }
}
```

When we convert this code into an OpenCL kernel it looks like this:

```
__kernel void   integrateTransformsKernel( __global Body* bodies,const int numNodes, float
timeStep)
{
        int nodeID = get global id(0);
        if( nodeID < numNodes && (bodies[nodeID].m_invMass != 0.f))
        {
                bodies[nodeID].m_pos +=  bodies[nodeID].m_linVel * timeStep;
        }
}
```

We need to write some host code in order to execute this OpenCL kernel. Here are typical steps for this host code:

1. Initialize OpenCL context and choose the target device
2. Compile your OpenCL kernel program
3. Create/allocate OpenCL device memory buffers
4. Copy the Host memory buffer to the device buffer
5. Execute the OpenCL kernel
6. Copy the results back to Host memory

The OpenCL API is very low-level, so we created a simple wrapper to match our coding style and to make it easier to use. This was a good learning experience. We also added additional features in the wrapper, you can check out the OpenCL Tips and Tricks section for more information. This wrapper doesn't hide the OpenCL API, so at any stage we can use plain OpenCL.

Porting existing code to OpenCL

We ran into many issues that prevented our code to run on GPU at all, let alone efficiently. This section shares some experiences, even though the solutions might be obvious.

Replace C++ by C

OpenCL is close to plain C, but Bullet 2.x is written in C++. This means that most of the code needs to be rewritten to use C and structures, instead of C++ and classes with inheritance etc. This conversion was easy, although it took time. It helps that in Bullet 2.x we avoid many C++ features: no exception handling, no run-time type information (RTTI), no STL and very limited use of template classes.

Share CPU and GPU code

In the process of porting, it is easiest to first implement a CPU version of an OpenCL kernel, and then port this code to an OpenCL kernel. Once this is done, it is useful to validate the OpenCL kernel with the CPU reference version, making sure that the output is the same.

In Bullet 2.x a lot of the algorithms involve a basic linear algebra math library for operations on 3d vectors, quaternions and matrices. OpenCL has some built-in support for this, with the float4 data type and operators such as the dot, cross product on float4 and many others. It helps to refactor your CPU math library so that the same code can run on the GPU: it allows development and debugging of the same implementation on the CPU. For compatibility we added some typedefs, global operators and access to the scalar operators .x .y .z and .w of the b3Vector3 on CPU, which resembles a float4.

Easy GPU<->CPU data synchronization

In Bullet 2.x we mainly use a resizable container template, similar to the STL std::vector. The actual name is b3AlignedObjectArray, but for simplicity we just call it std::vector here. During porting to OpenCL we found that it is really useful to have a container that keeps the data on the GPU. This container should make it easy to synchronization between the CPU and GPU data. So we designed the btOpenCLArray<> template.

```
template <typename T>
class b3OpenCLArray
{
        size_t  m_size;
        size_t  m_capacity;
        cl_mem  m_clBuffer;


. . .


        inline bool push_back(const T& _Val,bool waitForCompletion=true);
        void copyToHost(std::vector& destArray, bool waitForCompletion=true) const;
        void copyFromHost(const b3AlignedObjectArray<T>& srcArray, bool waitForCompletion=true)

}
```

This simplifies the host code a lot. Here is some example use.

```
std::vector<b3RigidBody> cpuBodies;
b3OpenCLArray<b3RigidBody> gpuBodies(clContext,clQueue);
gpuBodies.copyFromHost(cpuBodies);


. . .


gpuBodies.copyToHost(cpuBodies);
```

Move data to contiguous memory

In the Bullet 2.x API, the user is responsible for allocating objects, such as collision shapes, rigid bodies and rigid constraints. Users can create objects on the stack or on the heap, using their own memory allocator. You can even derive their own sub class, changing the so that the object size. This means that objects are not store in contiguous memory, which makes it hard or even impossible to transfer to the GPU.
The easiest way to solve this is to change the API so that objects creation and removal is handled internally by the system. This is a big change in the API and one of many reasons for a full rewrite.

Replace pointers by indices

In Bullet 2.x, our data structures contained pointers from one object to the other. For example, a rigid body has a pointer to a collision shape. A rigid constraint has a pointer to two rigid bodies, and so on. Those pointers are only valid in CPU Host memory and cannot be used on the GPU. This means that data created on the CPU cannot be used on the GPU. This issue can be solved, by replacing pointers by indices.
Generally it may be better to rethink the data structures. For Bullet 3.x the data structures are different, and there is not always a one-to-one mapping between new Bullet 3.x and Bullet 2.x types.

```
struct btTransform
{
	btMatrix3x3	m_basis;
	btVector3	m_position;
};
class btRigidBody : public btCollisionObject
{
	btMatrix3x	m_inverseInertiaWorld;
	btVector3	m_linearVelocity;
	btVector3	m_angularVelocity;
	btScalar	m_mass;
	. . .
};
class btCollisionObject
{
	btTransform	m_worldTransform;
	btCollisionShape* m_collisionShape;
	. . .
};
```

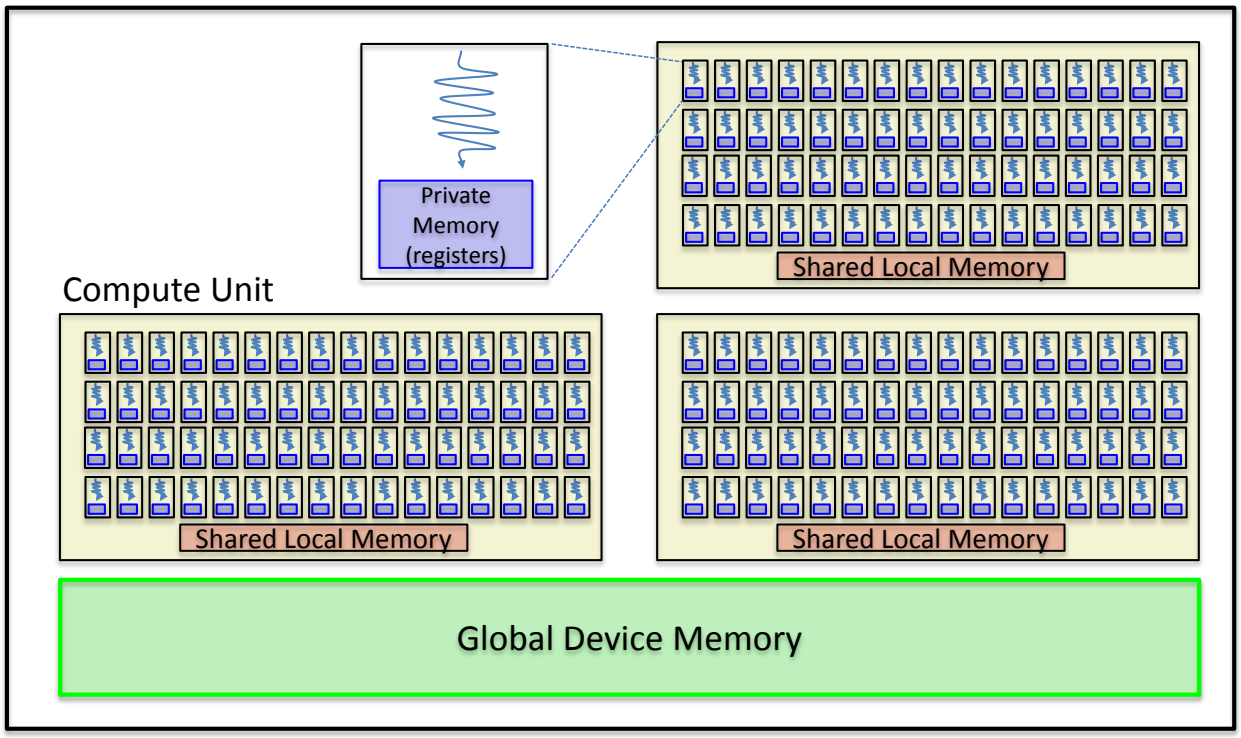
In our current Bullet 3.x we don't derive the rigid body from the collision object, and we moved some data from one class/struct to the other and created some new structures.

```
struct b3RigidBody
{
	b3Vector3	m_position;
	b3Quaternion	m_orientation;
	int		m_collidableIndex;
	. . .
};
struct b3Collidable
{
	int m_shapeType;
	int m_shapeIndex;
};
```

The basic *integrateTransforms* example in the previous section is embarrassingly parallel: there are no data dependencies between the bodies. Many of the algorithms in Bullet 2.x have some data dependencies so they cannot be trivially parallelized. The following section briefly covers GPU and OpenCL. After that, we discuss how we made effective use of the GPU many-cores and the GPU memory hierarchy in our rigid body simulation work.

Exploiting GPU hardware

A high-end desktop GPU has thousands of cores that can execute in parallel, so you need to make effort to keep all those cores busy. Those cores are grouped into Compute Units with typically 32 or 64 cores each. The cores inside a single Compute Unit execute the same kernel code in lock-step: they are all executing the same instruction, like a wide SIMD unit. The work that is performed by executing a kernel on a single core is called a Work Item in OpenCL. To make sure that multiple work items are executed on the same Compute Unit, you can group them into a Work Group. The hardware is free to map Work Groups to Compute Units, and this makes OpenCL scalable: if you add more Compute Units, the same program will run faster. The drawback is that there is no synchronization between Compute Units, so you need to design your algorithm around this. The host can wait until all Work Groups are finished, before starting new work.



Dealing with branchy code/thread divergence

Because all work items in a Compute Units execute in lock step, this means that code that has a lot of conditional statements can become very slow and inefficient.

```
  kernel void   branchyKernel (. . .)
{
        if (conditionA)
        {
                        someCodeA(. . .);
        } else
        {
                        someCodeNotA(. . .);
        }
}
```

If not all the work items in a Compute Unit have the same value for 'conditionalA' then they have to wait for eachother to finish executing 'someCodeA' and 'someCodeB'.

On the other hand, if all work items in the Compute Unit have the same value for *conditionA*, then only one of the two 'someCode*' sections will be executed, and there is no performance penalty for the if-statement.

Sort the input

If we know the *conditionalA* before executing the OpenCL kernel, we can sort the work items based on this *conditionalA*. This way, it is more likely that all the work items with a similar *conditionalA* will be processed in the same Compute Unit. In Bullet 3 we could use two parallel radix sorts on the overlapping pair array, based on each collision shape type (shapeTypeA and shapeTypeB) in a pair:

```
__kernel void   primitiveContactsKernel(__global int2 pairs, _global b3RigidBody* rigidBodies,
                                        global b3Collidable* collidables, const int numPairs)
{
        int nodeID = get global id(0);
        if (nodeID>=numBodies)
                return;
        int bodyIndexA = pairs[i].x;
        int bodyIndexB = pairs[i].y;
        int collidableIndexA = rigidBodies[bodyIndexA].m collidableIdx;
        int collidableIndexB = rigidBodies[bodyIndexB].m_collidableIdx;
        int shapeTypeA = collidables[collidableIndexA].m_shapeType;
        int shapeTypeB = collidables[collidableIndexB].m_shapeType;
        if (shapeTypeA== SHAPE PLANE && shapeTypeB==SHAPE SPHERE)
                return contactPlaneSphere(. . .);
        if (shapeTypeA== SHAPE SPHERE && shapeTypeB==SHAPE SPHERE)
                return contactSphereSphere(. . .);
        . . .
}
```

Breakup into pipeline stages

In some algorithms, we only know the value of the conditional, during the kernel execution. One example is the following algorithm, computing the contacts between a pair of convex collision shapes:

```
bool hasSeparatingAxis = findSeparatingAxis(objectA,objectB)
if (hasSeparatingAxis)
{
        clipContacts(objectA,objectB);
}
```

In this case, we can break up the algorithm into 2 stages, and first execute the 'findSeparatingAxis' stage for all objects. Then we execute the 'clipContacts' stage, only for the objects that have a separating axis.

The output of the first stage could be some array of boolean values. We would like to discard all the work items that have a negative value. Such stream compaction can be done using a parallel prefix scan. Essentially this shrinks the array and only leaves the positive elements. Then we can use this array as input for the next stage of the pipeline.

Use Parallel Primitives

When implementing software for GPU, several patterns or parallel primitives are very common:

- Sorting
- Counting, bound search
- Parallel sum, prefix scan

It is important to have an efficient GPU implementation of those to use as a building block.

Use Local Memory

Most GPUs have a memory hierarchy including

- Global GPU memory that can be accessed by all Compute Units
- Local shared memory, that can be accessed by all threads within one Compute Unit
- Private memory that can only be accessed by a single thread/Work Item

Local shared memory is usually at least one order or magnitude faster than global GPU memory. The use of local shared memory is best when there is data that can be shared by several Work Items in a Work Group. Local shared memory can be useful for read-only input data, but also to improve performance of writing the output data.

Barrier synchronization

In order to make use of local shared memory, we also need to make sure that the data is valid, before any of the Work Items starts accessing the memory. For this we can use a barrier. A very common usage is as follow:

- Copy the data from global GPU memory to local shared memory
- Add a barrier so that all threads are waiting for the data to be valid
- Perform some computation using the local shared memory

Atomics

Atomic operations can be useful in many cases. One way we use atomic operations is to emulate a global append buffer. Any Work Item in any Work Group might want to append some data to a global array. We can create an integer index in global GPU memory. Each thread can use an *atomic_add* operation to append a number of items to the buffer.

GPU rigid body simulation

A lot of work can go into picking or designing an algorithm that is suitable for GPU. In this section we will go more in detail how we implemented each stage of the rigid body pipeline to GPU.

Rigid body introduction

A rigid body is an object that has some properties that generally don't change over time, such as mass and inertia properties and a collision shape representation. A rigid body also has some state variables such as position and orientation and linear and angular velocity that change over time, usually due to some forces such as gravity, collisions or other constraint forces. A 3d rigid body has 6 degrees of freedom for its motion: translation along each of the 3 primary axis (x,y,z) and rotation among those 3 axis.

It is a physics engine task to update the state of rigid bodies according to the Newton laws. At regular intervals it moves the objects and detects collisions between rigid bodies and makes sure that objects don't penetrate. Aside from such non-penetration constraints, there are other constraints that control the degrees of freedom between a pair of rigid bodies.

A rigid body pipeline consists of all consecutive stages of computation that are performed during a single simulation time step.

A collision shape describes the surface or volume of an object. There are many collision shape representations, for example convex shapes such as a sphere, box, cone, cylinder, capsule or a convex hull of some vertices/points. Another popular representation is a triangle mesh geometry. Multiple collision shapes can be grouped into a compound collision shape.

To accelerate collision detection between two collision shapes, we first test if the world space bounding volumes of the shapes overlap. The bounding volume we use is an Axis Aligned Bounding Box or AABB.

In addition to a single world space AABB, for complex shapes we can have an additional local space acceleration structure. We can use a bounding volume hierarchy or BVH such as an AABB tree for this.

Aside from updating the state of all rigid bodies in the world, a physics engine can also provide collision queries and ray intersection queries against the collision shapes of the rigid bodies. For example you can query the closest points or the penetration depth between two objects.

The rigid body pipeline

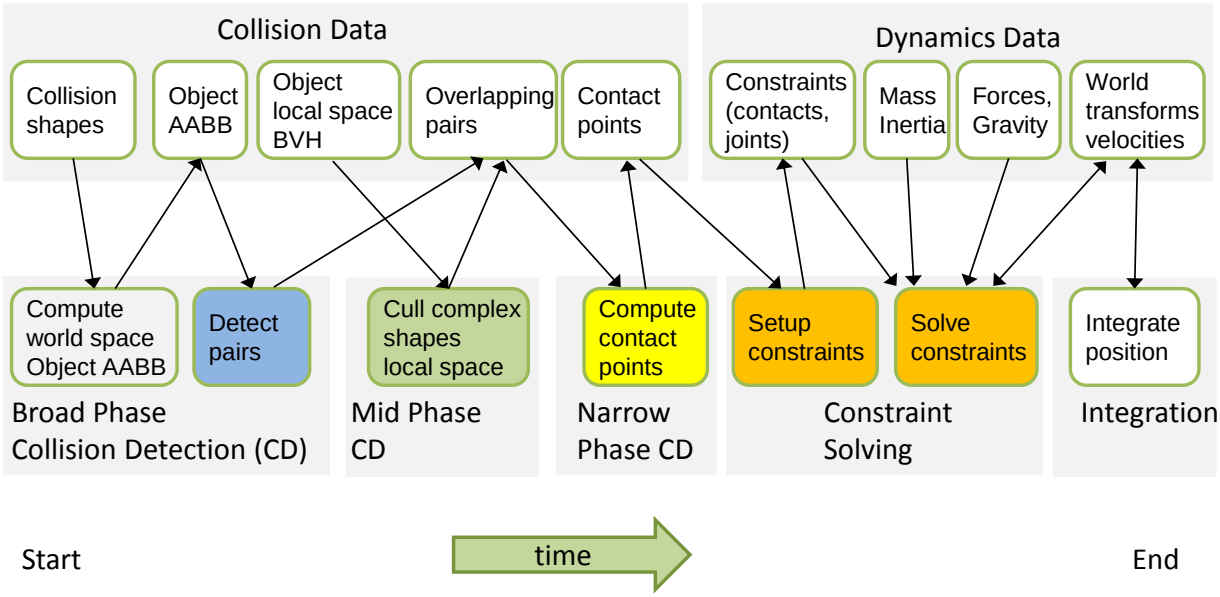
The Rigid Body Pipeline 1 diagram shows the data structures and computation stages of a simplified rigid body pipeline. In a nutshell, we detect potential overlapping pairs in the Detect Pairs stage. Given *n* rigid bodies, this step will reduce the expected time complexity from *O(n*n)* to *O(n)*.

Once we have the potential overlapping pairs, we compute the contact points in the Narrow Phase step. For a pair of overlapping spheres, this step is trivial, but for a pair of convex hull meshes it becomes more complicated. If we deal with large triangle meshes, we usually add an additional culling step, known as the Mid Phase collision detection.

The contact points are converted into contact constraints, so that objects will no penetrate at the end of the simulation step. The contact constraints are satisfied together with non-contact constraints in the Constraint Solving step. We solve all the constraints together, because satisfying one constraint might violate another constraint, so this is a global problem.

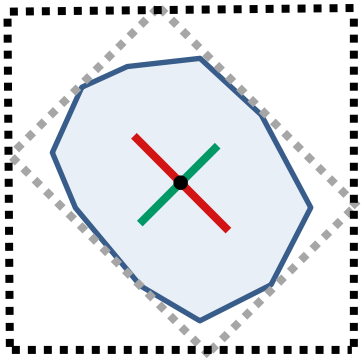
The output of the constraint solving step is updated velocities. Those are used in the Integrate Position step, so update the position and orientation of the rigid bodies.

Rigid Body Pipeline 1



Computing the object AABBs

We need to compute the axis aligned bounding box, AABB, for each object. This is an embarrassingly parallel operation so we won't go in detail. Normally, the collision shape is used to compute the world space AABB. One of the optimizations we made is to cache the local space AABB for the collision shape, the grey dotted box in this figure, and use this to recompute the world space AABB, the black dotted box.



You can find the kernel source code in *src/Bullet3OpenCL/RigidBody/kernels/updateAabbsKernel.cl*

GPU overlapping pair detection

Given all the object axis aligned bounding boxes, we need to find all the object pairs that have overlapping AABBs. The brute force algorithm would perform O(n^2) checks. The original CPU version looks like this:

```
void   computePairsKernelBruteForce (const btAabbCL* aabbs, volatile __global int2*
pairsOut,volatile  __global int* pairCount, int numObjects, int maxPairs)
{
   for (int i=0;i<numObjects;i++)
   {
        for (int j=i+1;j<numObjects;j++)
        {
                if (TestAabbAgainstAabb2GlobalGlobal(&aabbs[i],&aabbs[j]))
                {
                        int2 myPair;
                        myPair.x = aabbs[i].m_objectIndex;
                        myPair.y = aabbs[j].m objectIndex;
                        int curPair = *pairCount;
                        if (curPair<maxPairs)
                        {
                                pairsOut[curPair] = myPair; //flush to main memory
                                pairCount++;
                        }
                }
        }
   }
}
```

When transforming this to an OpenCL kernel, we could drop the outer for-loop, and let individual Work Items deal with each i. This would provide some amount of parallelism:

```
__kernel void   computePairsKernelBruteForceKernel ( __global const btAabbCL* aabbs, volatile
__global int2* pairsOut,volatile  __global int* pairCount, int numObjects, int maxPairs)
{
        int i = get_global_id(0);
        if (i>=numObjects)
                return;
        for (int j=i+1;j<numObjects;j++)
        {
                if (TestAabbAgainstAabb2GlobalGlobal(&aabbs[i],&aabbs[j]))
                {
                        int2 myPair;
                        myPair.x = aabbs[i].m objectIndex;
                        myPair.y = aabbs[j].m_objectIndex;
                        int curPair = atomic_inc (pairCount);
                        if (curPair<maxPairs)
                        {
                                pairsOut[curPair] = myPair; //flush to main memory
                        }
                }
        }
}
```

The global *pairCount* variable keeps track of the output to a global array of overlapping pairs. Each Work Item can use the OpenCL *atomic_inc* operation to increment this variable. The return value of *atomic_inc* is the old value, that we use to store the output. We also need to perform a range check against *maxPairs*, to make sure we don't write beyond the output array.

The use of atomic_inc on a global variable can reduce performance, when many Work Items try to increment the variable at the same time. We can optimize this in various ways. One way is to create a small temporary buffer in local shared memory, and perform local atomic_inc operations on this buffer. Only when the buffer is full, or when the kernel is completed, we can write the results into global

memory. Instead of writing a single result, we can use the atomic_add operation. This use of a global buffer is also known as an Append Buffer.

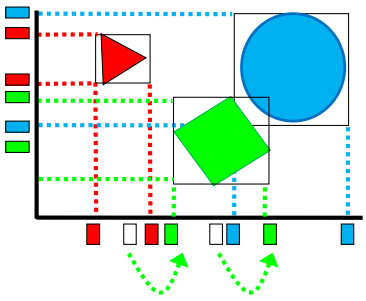
There are many ways to optimize the pair detection, and there are various factors that affect the performance:

- What percentage of the objects are moving?
- How fast do the objects move?
- Do we have large variation in object (bounding volume) sizes?
- Do we add and/or remove many objects?
- Do we want to re-use the acceleration structure for ray intersection queries and swept volume?

One GPU friendly acceleration structure is the uniform grid. This can be parallelized very well, but the drawback is that the collision shapes of rigid bodies can vary a lot in size, so there is not a suitable grid cell size that fits all collision shapes in general. One option is to use a mix of different acceleration structures, a uniform grid for small objects and particles, and a different acceleration structure for larger objects.

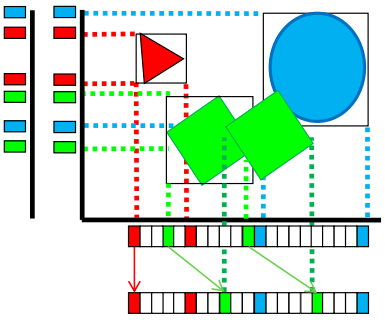
On the CPU, we have incremental algorithms that are unsuitable for GPU in their unmodified form. In Bullet 2.x we have the btDbvtBroadphase, based on a dynamic AABB trees, which are incrementally updated, instead of rebuild from scratch. The incremental updates have a lot of data dependencies and random memory access, so that we cannot easily perform them in parallel. This is still ongoing work.

Another pair search algorithm in Bullet 2.x is the 3-axis sweep and prune. Similar to the previous case, this acceleration structure is incrementally updated with a lot of data dependencies. In this case, however, we came up with some modifications that make it suitable for GPU.



In the original sequential CPU version, the AABB min- and max-coordinates for each axis are move to their new location. Pairs are added or removed during the swap operations that move the AABB coordinate from the old to the new position. The dependencies of data accesses, during this incremental sorting operation, prevent a parallel version.

We modified the incremental algorithm and perform a full sort on each of the 3 axes, with projected AABB begin and end coordinates. In addition, we keep track of the previous sorted arrays for each axis. Then, each object can perform read-only operations that mimic the 'swap' operations to add or remove overlapping pairs: for each object we traverse the original and new arrays from the original index to the new index to determine added or removed pairs:



We also have an implementation of a parallel 1-axis sweep and prune, which performs a full sort on a single axis. The axis with the best variance is computed using a parallel prefix sum. Both the 1-axis and 3-axis sweep and prune algorithm requires a parallel radix sort on floating point values. Our radix sort can only sort arrays of integers, so we need to convert the floating point values to integers first.

You can find the implementation in *src/Bullet3OpenCL/BroadphaseCollision/kernels/sapFast.cl*

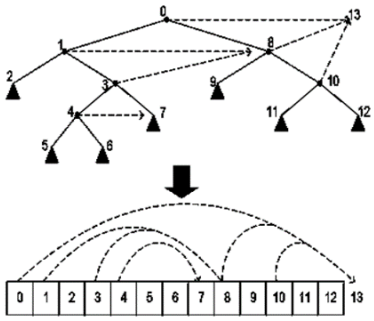
More details about the 1-axis GPU sweep and prune can be found in the paper "Real-time Collision Culling of a Million Bodies on Graphics Processing Units" [Liu 2010]

GPU local space BVH culling for complex shapes

Some of the overlapping pairs contain complex concave shapes that need additional culling before performing exact contact detection. This will effectively simplify overlapping pairs that contain concave triangle meshes and concave compound shapes to new overlapping pairs containing only convex shapes.

We can use a precomputed local space AABB tree to find overlapping child shapes. For a concave triangle mesh shape, the child nodes of the tree refer to individual triangles. For compound collision shapes, the child nodes in the tree refer to individual convex child collision shapes.

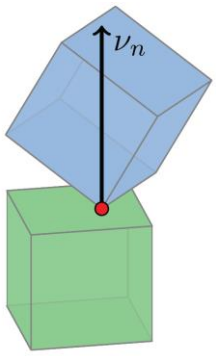
Among the GPU optimizations, we compressed the AABB nodes to 16 bytes using quantization. Additionally, we use a stackless tree traversal, which is very efficient on the GPU.



The implementation is in *src/Bullet3OpenCL/NarrowphaseCollision/kernels/bvhTraversal.cl*

GPU contact computation

Once we finish the pair detection, we have access to overlapping pairs of convex shapes. The algorithm that we use depends on each of the collision shape type in the overlapping pair.

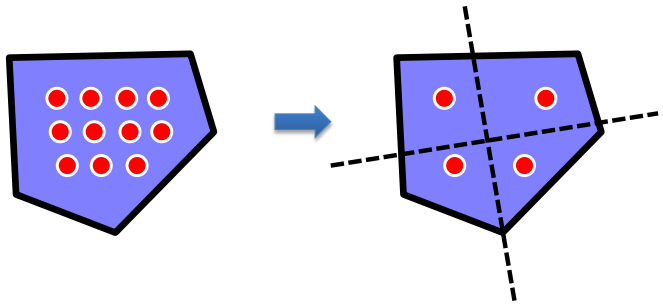


In Bullet 2.x we use the GJK and EPA algorithm to compute contacts between convex shapes, but those algorithms not trivially optimized for GPU.

For Bullet 3.x we compute contacts between convex polyhedral collision shapes using the separating axis test (SAT), contact clipping and contact reduction. The CPU version is very branchy and not suitable for GPU. The pseudo code looks like this:

```
void   computeContactsSAT(convexHullA,convexHullB,transformA,transformB)
{
        b3Vector3 satAxis;
        if (findSeparatingAxis(convexHullA,convexHullB,transformA,transformB))
        {
                if (clipHullHull(satAxis, convexHullA,convexHullB,transformA,transformB))
                {
                        findClippingFacesKernel(. . .)
                        clipFacesAndContact(. . .)
                        contactReduction(...)
                }
        }
}
```

We refactored the code into a collection of kernels that form a pipeline: first we compute the separating axis for all pairs. Once we have the results, we can discard all pairs that don't overlap, using stream compaction (using a prefix sum parallel primitive). The next stage is *clippHullHull* for all pairs in parallel, followed by the *clipFacesAndContactReductionKernel* stage. This way, we the main branches in the original algorithm become kernel executions between the different pipeline stages.

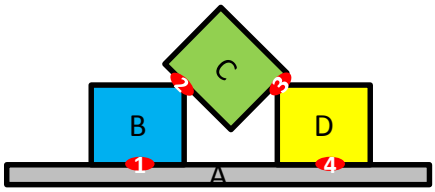


The contact reduction in Bullet 2.x was based on an incremental algorithm that can add or remove one point at a time, using a persistent contact point cache. In Bullet 3.x we compute the full set of contact points, and perform contact reduction on the resulting points.

We also refactored the data structures for convex shapes, faces, edges and vertices in a GPU friendly way.

GPU parallel contact solving

Solving pair-wise constraints means we have to update data for each of the bodies involved. This means that solving multiple constraints is not embarrasingly parallel: they might try to access the same bodies. In the following example, the constraints are numbered 1,2,3,4 and the bodies A,B,C,D.



Constraint 1 and constraint 2 both have read-write access to B, and this means those two constraints cannot be executed in parallel, without synchronization. In order to solve this, we can sort the constraints in independent batches, where the constraints in each batch don't have read-write access to the same bodies.

The pseudo code of the sequential batch sorting looks like this:

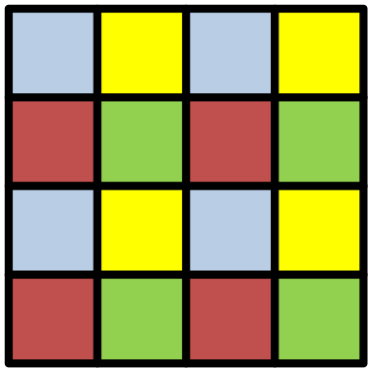
```
void   batchConstraints(constraints, int numConstraints)
{
        int batchIdx=0;
        while( numValidConstraints < numConstraints)
        {
                clear(bodiesUsed);
                for(int i=numValidConstraints; i<numConstraints; i++)
                {
                        int bodyA = constraints[i].m_bodyA;
                        int bodyB = constraints [i].m bodyB;
                        if (isAvailable(bodyA,bodyB,bodiesUsed))
                        {
                                markUsed(bodyA,bodyB,bodiesUsed);
                                constraint[i].m_batchId = batchIdx;//assign the batch index
                        }
                }
                batchIdx ++;
        }
}
```

If we want to move the batch generation onto GPU, there are several considerations. We can separate batch generation in two stages:

1. Local batching: batching within a single compute unit
2. Global batching: split the input so that it can be processed on different compute units

If we only deal with a single compute unit, we could process the batch generation within a single thread/work item. For a compute unit that has 64 threads, 63 threads would be idle, so that is a waste. If we use more then a single work item/thread to perform batch generation, we need to implement a parallel version of 'isAvailable' and 'markUsed'. This can be done using local shared memory, with a small buffer representing the bodies that marked as used. Threads can try to mark bodies as used. After this step, each thread can check if their marked bodies are not overwritten by other constraints. If not, then they succeeded in marking the bodies as used, and the constraint can be added to a batch. Due to lack of local shared memory, we have only limited storage for the marked-as-used bodies. Using a modulo operation, multiple bodies can map to the same 'marked-as-used' storage. A drawback is that some threads might fail to add constraints to a batch, so we need to execute the attempt of body reservation multiple times.

For the global batching among multiple compute units, we cannot rely on synchronization. A way to split the input for global batching is using spatial information: objects that are further away than the maximum object size cannot collide. We can divide space in cells, and solve non-neighboring cells in parallel: the blue cells can be processed in parallel, and the same for the yellow, red and green cells.



Another way to split the input is by creating groups of objects, and compute the pair interactions of the groups carefully to avoid conflicts. A simple example of this would be a grouping in bodies with an odd and even index. The pair search within the ODD group is independent from the EVEN group, so they can be performed in parallel on separate compute units. The pair search between a body of the ODD group and the EVEN group needs to be performed in a later batch. On high-end GPUs we have tens of compute units, so we need to create enough separate groups. Bodies can be added to the same group, if they have the same lower n-bits, creating 2^n groups. Given 2^n groups, we can create a static interaction table to determine what groups can be solved in parallel, similar to the spatial grouping. For more information see "A parallel constraint solver for a rigid body simulation" by Takahiro Harada [Harada 2011].

GPU parallel joint solving

Once the constraints are sorted in independent batches, the constraint solver can solve all constraints in a batch in parallel. The global batching also enables multiple compute units to process constraint rows in parallel. We just have to make sure that we finish all constraints in one batch, before we proceed solving the constraints in the next batch. If we dispatch each batch from the host, the batch execution is synchronized, so the order is guaranteed. Typically we use 4 to 10 PGS iterations, and for each iteration we need to execute the n local batches sequentially. If we have 20 local batches, this would require 200 kernel enqueue (clEnqueueNDRangeKernel) commands. The overhead of the clEnqueueNDRangeKernel can become a bottleneck. This can be avoided by letting the compute unit manage the synchronization of the local batches, using local atomic operations. Each work item keeps on solving constraints, as long as the batch index is the same as the current batch index in local shared memory.
Here is an OpenCL code snippet:

```
while (ldsCurBatch < maxBatch)
{
        for(; idx<end; )
        {
                if (gConstraints[idx].m_batchIdx == ldsCurBatch)
                {
                        solveContactConstraint( gBodies, gShapes, &gConstraints[idx] );
                        idx+=64;
                } else
                {
                        break;
                }
        }
        barrier(CLK_LOCAL_MEM_FENCE);

        if( lIdx == 0 )
        {
                ldsCurBatch++;
        }
        barrier(CLK_LOCAL_MEM_FENCE);
}
```

We also implemented a parallel GPU Jacobi solver [Tonge 2012] but the convergence was not as good as the Projected Gauss Seidel algorithm.

GPU deterministic simulation

The work items and compute units in a GPU are executed in parallel, and the order in which work items are executed can be different each time. This non-determinism, or lack of consistency, can affect the results. For instance, if the pair search appends pairs using an *atomic_inc* operation, there is no guarantee that pairs are inserted in the same order.

If we have a different order of overlapping pairs, and contact points, we may also have a different order of contact constraints. The Projected Gauss Seidel algorithm produces different results, if the constraint rows are solved in a different order. If we want the same results each time we run the simulation (on the same hardware/compiler) we need to make sure that the order is always the same.

We can sort the overlapping pairs, or contact points, using a parallel radix sort. In a similar way, we need to sort the output that is generated during parallel tree traversals for complex concave collision shapes.

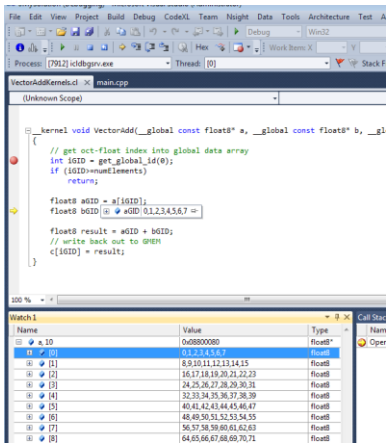
Debugging and Performance Profiling

Debug on the CPU

Debugging OpenCL kernels is much harder than debugging regular CPU code. So the best way is to keep a duplicate implementation running on CPU, and do the debugging and development there first.

Intel OpenCL debugger

Intel provides an OpenCL debugger that allows us to step through the OpenCL kernels, when using a CPU OpenCL Device. Some of our kernels don't run on CPU though, because of minimum Work Group size requirements. Here is a screenshot:



printf debugging

If the CPU version is working all fine, but the GPU version doesn't, we need other tools. One of them is *printf* debugging, from within an OpenCL kernel. Not all OpenCL implementations support *printf* debugging, it depends on the version and the vendor.

Debug buffers

Another way to debug the OpenCL kernels is to add additional kernel buffers, especially for debugging. This way, we can inspect any data, after the kernel finishes execution.

Debugging a frozen system

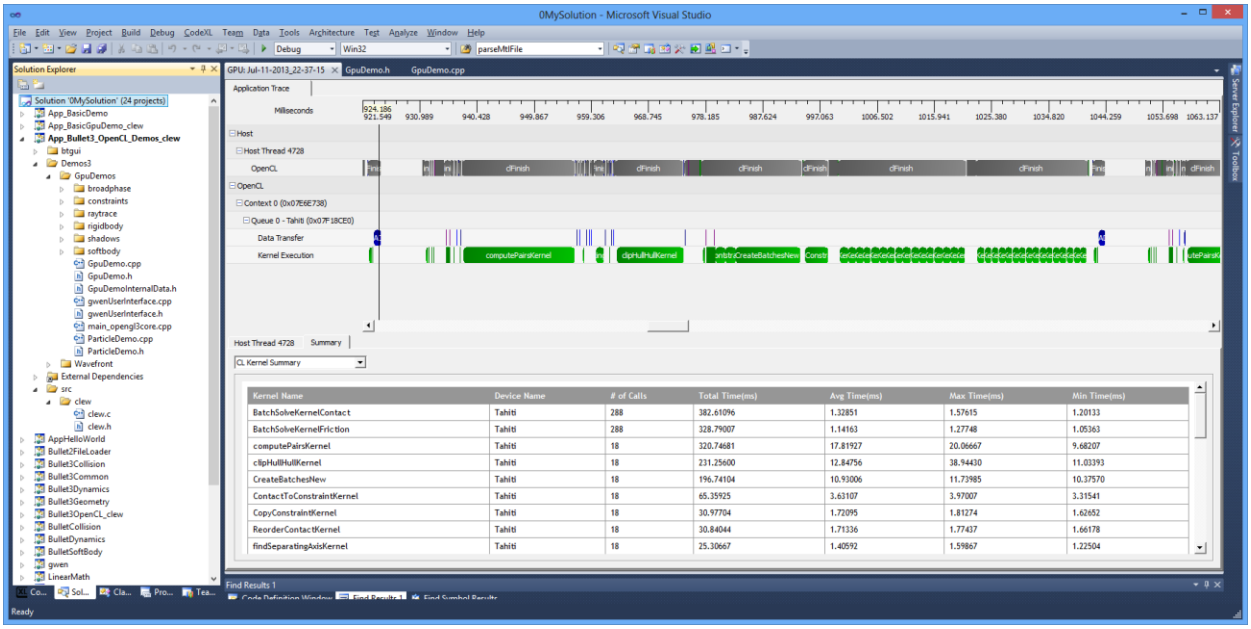
Sometime we encounter a system hang during testing of OpenCL kernels. In most cases on Windows PC, there is some mechanism that resets the GPU when this happens, so we can continue our work. In other cases, especially on Mac OSX, the entire operating system can freeze when an issue happens. We need to figure out what kernel causes the hang, and to do this, we add instrumentation that output some message to the console or to a file. This instrumentation can be added to an OpenCL wrapper, or it can be added to performance profiling zones.

Profile Zones

We add profile zones to our code on the CPU side, to get a course-grain hierarchical profile timings. For more detailed GPU performance statistics we use either CodeXL or NVIDIA NSight.

CodeXL Performance Profiler

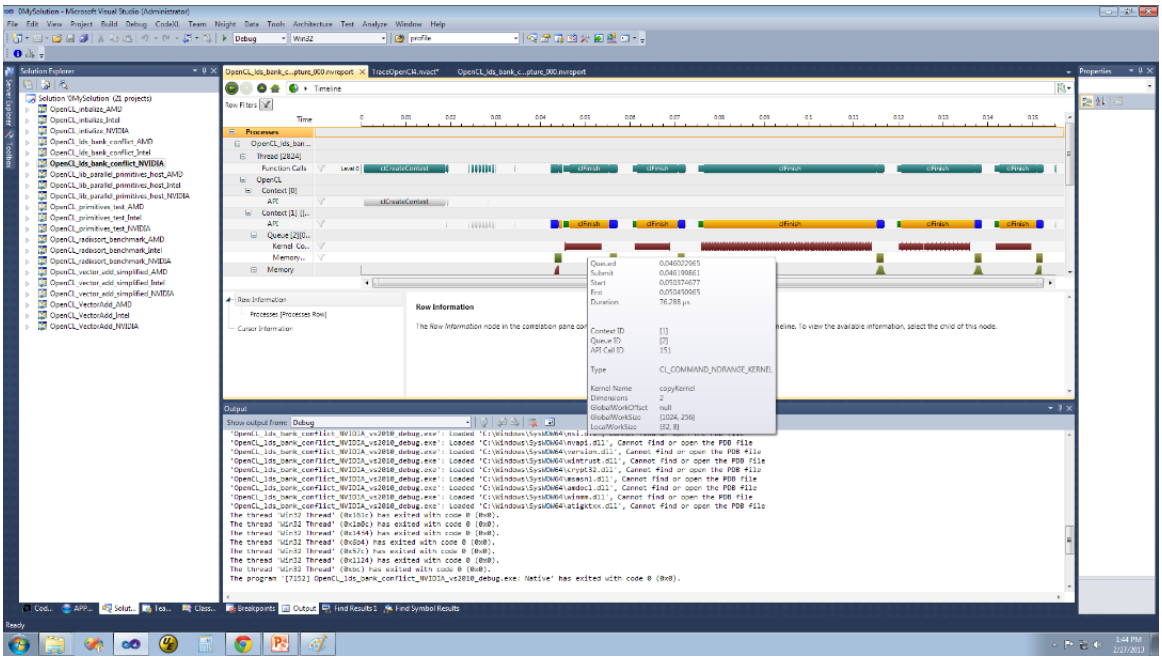
AMD CodeXL also contains a performance profiler that gives a lot of profile information:



Kernel Name	Device Name	# of Calls	Total Time(ms)	Avg Time(ms)	Max Time(ms)	Min Time(ms)
BatchSolveKernelContact	Tahiti	288	382.61096	1.32851	1.57615	1.20133
BatchSolveKernelFriction	Tahiti	288	328.79007	1.14163	1.27748	1.05363
computePairsKernel	Tahiti	18	320.74681	17.81927	20.06667	9.68207
clipHullHullKernel	Tahiti	18	231.25600	12.84756	38.94430	11.03393
CreateBatchesNew	Tahiti	18	196.74104	10.93006	11.73985	10.37570
ContactToConstraintKernel	Tahiti	18	65.35925	3.63107	3.97007	3.31541
CopyConstraintKernel	Tahiti	18	30.97704	1.72095	1.81274	1.62652
ReorderContactKernel	Tahiti	18	30.84044	1.71336	1.77437	1.66178
findSeparatingAxisKernel	Tahiti	18	25.30667	1.40592	1.59867	1.22504

NVIDIA NSIGHT Profiler

The NVIDIA NSight can also show performance profile for OpenCL kernels:



OpenCL Tips and Tricks

Here are a few practical tips from our experience with OpenCL:

Create your own OpenCL wrapper

OpenCL is a low level API and it can be cumbersome to use. With very little effort, you can create your own wrapper to make it much easier to use. There are several benefits of writing your own wrapper, rather than using an of-the-shell wrapper. First of all, you can make it fit very well with your own coding style. Secondly, it is a very good learning experience to learn the details of the low-level OpenCL API. Third, you can add extra features in your wrapper library. We implemented several of the other tips and tricks using our wrapper.

Dynamically load OpenCL

If you link against an OpenCL SDK from a vendor, your program will abort if the user doesn't have any OpenCL driver installed. You can deal with this in a better way so that the program can continue running, even if the OpenCL driver is missing. You can to load the OpenCL dynamic library at run-time and bind against the API dynamically using the clew library. You can download it from https://code.google.com/p/clew

Cache the precompiled OpenCL kernel binaries

The compilation of OpenCL kernels normally happens after you start running your program. This kernel compilation can take a lot of time, so if you have many kernels it is better to store the compiled kernels to disk, and load the binaries.

Keep a host implementation of your kernel

Debugging OpenCL kernels can be very hard and time consuming. We find it very useful to first implement a host implementation, and maintain it even after you get the OpenCL kernel up and running. It can be very hard to locate a bug in a program with many OpenCL kernels, and if you can enable/disable OpenCL kernels individually, it makes bugs easier to find and fix.

Unit test an OpenCL kernel

We added the option to serialize all the input and output data of an OpenCL kernel, so we can debug the kernel outside of our program. Sometimes it takes a lot of time and effort to reproduce a bug, and with this serialization effort, we can directly run a buggy kernel separately. This makes life much easier.

References

Tonge 2012, "Mass splitting for jitter-free parallel rigid body simulation", Richard Tonge et al., SIGGRAPH 2012, http://dl.acm.org/citation.cfm?id=2185601 and http://www.richardtonge.com

Harada 2011, "A parallel constraint solver for a rigid body simulation" by Takahiro Harada. SIGGRAPH Asia 2011 Sketches. http://dl.acm.org/citation.cfm?id=2077406

Liu 2010, "Real-time Collision Culling of a Million Bodies on Graphics Processing Units", SIGGRAPH Asia 2010, http://graphics.ewha.ac.kr/gSaP

Coumans 2009, "OpenCL Game Physics", Erwin Coumans, NVIDIA Game Technology Conference 2009, http://www.nvidia.com/content/GTC/documents/1077_GTC09.pdf

Harada 2007, "Real-Time Rigid Body Simulation on GPUs" Chapter 29 in the GPU Gems 3 book, http://http.developer.nvidia.com/GPUGems3/gpugems3_ch29.html

Appendix A: Bullet 3.x Source code

The source code of Bullet 3.x is available under a permissive zlib/BSD style license on Github at http://github.com/erwincoumans/bullet3.

Requirements

The code is tested on Windows 7/8, Linux and Mac OSX desktops with a recent high-end GPU such as an AMD Radon 7970 or AMD W9000, or an NVIDIA GTX 680. Most laptop GPUs are too slow and don't have enough memory for this project to be useful.

Building on Windows using Visual Studio

You can click on build3/vs2010.bat
Open Bullet3/build3/vs2010/0MySolution.sln

Building on Linux (or Mac OSX) using gcc

Open a terminal

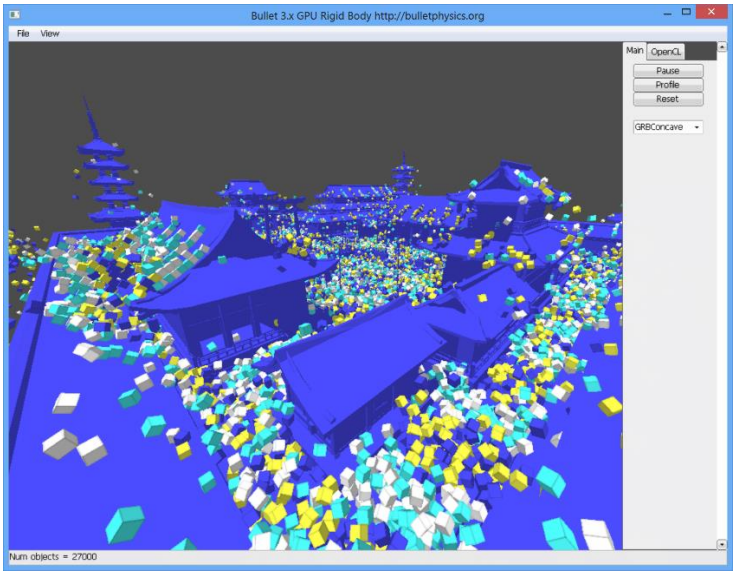
```
cd build3
./premake_linux64 gmake
cd gmake
make
```

Building on Max OSX using XCode

Click on build3/xcode.command and open build3/xcode4/0MySolution.xcodeproj

Usage

You can execute the demo application from the bin directory. It looks like this:



Benchmark

The demo includes a benchmark mode that export a comma separated file (for Excel)
bin/App_Bullet3_OpenCL_Demos_clew_vs2010 --benchmark
You can use the F1 key to create a screenshot and the Escape key will terminate the demo.

Feedback

Although the new Bullet 3.x OpenCL rigid body work is still work-in-progress, it can already be useful for VFX projects that need to simulate a large amount of bodies on a single desktop computer.

If you have any feedback about the software, please contact the author at erwin.coumans@gmail.com or visit the Bullet physics forums at http://bulletphysics.org

Presto Execution System: An Asynchronous Computation Engine for Animation

George ElKoura
Pixar Animation Studios

Introduction

We present an asynchronous computation engine suitable for use in an animation package. In particular, we describe how this engine is used in Pixar's proprietary animation system, Presto.

We will start by first describing what Presto is and concentrate on how it is used by two different disciplines. We will then dive into Presto's Execution System, its phases and the structures that make up its architecture. We then describe how several different multithreading strategies can be implemented using the architecture. We focus on Background Execution, a feature that allows users to keep working while soon-to-be-needed computations are performed in the background. We discuss the requirements and considerations from a user interaction perspective. We then end with some of the future work we'd like to pursue.

Presto

Presto is Pixar's proprietary, fully featured, animation package. Besides the main interactive application, Presto is built on top of a rich set of reusable libraries. The application supports integrated workflows for a variety of feature-film departments including rigging, layout, animation and simulation. It also provides built-in media playback and asset management tools.

For the purposes of this course, we will mainly discuss Presto's Execution System. We will use two common disciplines, rigging and animation, to illustrate how the system works. Much of what will be discussed applies to other disciplines as well. Though we will mainly talk about posing points, the execution engine itself is generic and is used for other kinds of computations as well.

One of the challenges in Presto is its integrated architecture. In a single session, the user may wish to animate or do some rigging or run a sim or all three without an explicit context switch. Some of these tasks do not lend themselves well to a multithreading environment, and yet must coexist seamlessly with all features of the application.

Rigging in Presto

Rigging is the process of modeling the behavior of the characters and props on a show. Riggers describe how, given a set of inputs, a character poses. Riggers use what is effectively a visual programming language to describe the structure and behavior of their models. In Presto, riggers do not directly author the execution structures. Instead, they visually author human-readable objects using higher-level concepts. These objects later get transformed into data structures that are designed specifically for efficient computation.

Network topology changes often during the course of regular rigging workflows. Geometry topology also changes during rigging, but is fixed once the rig is finalized. In other words, the rig, itself, does not modify the topology of the geometry.

Animation in Presto

Animation is responsible for bringing the characters to life. Animators supply input values to the rig in order to hit desired character poses.

The topology of the character's geometry does not change during the course of regular animation workflows. Network topology may change, but does so infrequently. Addition of post-rigging deformers, such as lattices, and adding new constraints are examples of animator workflows that do change the network topology.

Presto's Execution System

Presto's Execution System is a general-purpose computation engine. Given a set of inputs (e.g. animation splines) and an execution network (e.g. derived from a rig), the job of the execution system is to provide the computed result as quickly as possible. Common computations include posing points, calculating scalar fields, determining object visibility, and so on.

Much like other such systems, at its core, the execution system in Presto evaluates a data flow network. Presto's data flow network is vectorized, meaning that many uniquely identifiable elements may flow along a single connection. We don't need to get into the details of this aspect of the system to talk about multithreading, but it is worth noting as it will come up now and again.

In the following sections we'll explore the architecture of the execution system, how the parts fit together, and how they lead to a framework that is amenable to multithreaded computation.

Phases of Execution

The execution system is broken up into three major phases:

- Compilation
- Scheduling
- Evaluation

Each phase amortizes costs for the next phase. The phases are listed in the order that they run, also in the order of least-frequently run to most frequently run and from most to least expensive runtime costs.

Compilation

As mentioned above, riggers author scene objects using a rich, high level visual language. This allows riggers to be efficient at their main task. By abstracting away details of the execution system, we allow riggers to concentrate on building the rigs. However, the visual paradigms that allow for a fast rigging process may not always lend themselves to fast rig evaluation. Compilation is the phase of execution that converts the human-readable scene objects into optimized data structures that can be used for fast repeated evaluations of the rig (e.g., during animation).

The result of compilation is an execution network consisting of nodes and connections. While riggers deal in various concepts like connections, deformers, weight objects, and so on, once compilation is done, the execution system sees only one homogenous concept of execution nodes.

In addition to enabling fast evaluation, compilation provides a layer of abstraction that allows us to keep the assets separate from the data structures required by our system. Assets are time consuming and expensive to author, and we would like to have them be independent from the implementation of the execution system. That is to say, if we decide that we've been doing things all wrong in how the execution system's data structures are organized, we wouldn't have to modify a lot of assets to overhaul the system. Compilation also provides a convenient place to perform optimizations at the network level.

Full network compilation typically happens only when a character is first loaded in a session. Rigging workflows invoke an incremental recompilation code path that builds the network in pieces as the rig is developed. The results of compilation can be serialized, in which case full network compilation can be eliminated from most use cases.

Scheduling

Given a set of desired output values (e.g. the posed point positions of a character), which we call

a request, scheduling is responsible for performing the dependency analysis necessary to determine which nodes need to run, in order to satisfy the request. Scheduling serves to amortize dependency analysis costs that would otherwise have to be incurred during each network evaluation. The specifics of the analysis performed during scheduling is up to the implementation. For example, it may be beneficial for certain schemes that scheduling determine the partial ordering in which the nodes run, and for others, it might be more efficient that scheduling only determine what nodes need to run, and the ordering is left to the evaluation phase.

Scheduling is performed more often than compilation, but not as often as evaluation. Scheduling must be performed after network topology changes caused by incremental recompilation, and therefore occurs more often during rigging workflows, and rarely during animation workflows. Animation workflows may cause scheduling, for example, when adding a new constraint or creating new deformers.

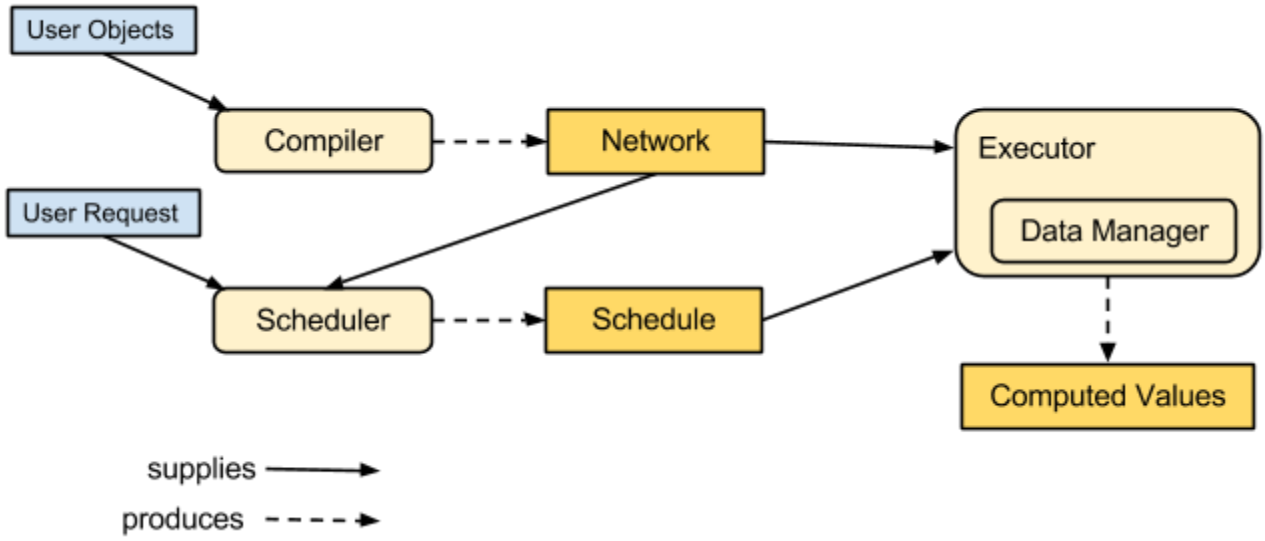
Evaluation

Evaluation is the most frequently run phase of execution. Its job is to run the nodes in the network as determined by the schedule in order to produce computed values for the requested outputs. Evaluation is run every time an input value changes, or a new output is requested, and the results are pulled on (e.g. to satisfy a viewer update).

Engine Architecture

The execution system consists of the following data structures:

- Network
- Schedulers
- Data Managers
- Executors



Network

The network is generated from user-authored scene objects and is a static representation of computations and the dependencies among them. A network is made up of nodes and connections. A node is made up of zero or more inputs and zero or more outputs (a leaf node with zero outputs is only used for tracking and broadcasting dependencies to external systems). Client code requests computed values by specifying node outputs.

This is a data flow network with added support for vectorization. *It is important to note that no stateful information is stored in the network, it embodies only the computation's static structure.*

Schedulers

Schedulers are responsible for the scheduling phase of execution and produce schedules that are held on to by the system and are used by the executors for repeated evaluations.

Schedulers can be specialized for experimenting with different kinds of multithreading schemes, or for providing more analysis that can be taken advantage of by executors.

Data Managers

Data managers are simply data containers for nodes in the network. They are used by the executors to store the computed data. Not storing data in the nodes themselves is an important architectural feature of our network. You would specialize a data manager only if you wanted to provide a faster implementation. There is no deep work going on in these objects.

Data managers store computed data as well as validity masks and other per-node or per-output data that is required by the executor.

Executors

Executors are the workhorses of the execution system. Executors orchestrate computation using the compiled network, a schedule and a data manager. They run the “inner loop” and need to run as efficiently as possible.

Executors can be arranged in a hierarchy, where the child is allowed to read (but not write) from the parent in order to avoid redundant computations. This feature is put to use in a few different ways in the system, but for the purposes of this course we will discuss how it is used for Background Execution a little later.

Executors can be specialized to supply different algorithms for running the network. You might want to specialize an executor to experiment with different multithreading techniques, for example, or to target a different platform. When we later discuss the different multithreading strategies, they are primarily implemented in different kinds of executors.

User Extensions

Since we have always intended for our system to be run in a multithreaded environment, we needed to carefully consider the responsibilities we impose on clients of the system. One of our goals was to make it as safe as possible for users to write plugin code without worrying about multithreading in most cases.

One of the problems that often complicates multithreading is having to call out to client code. Not knowing what the clients are going to do, and what resources they might acquire makes it difficult to create a resilient and robust system, let alone one with predictable performance characteristics. Our system is no different, but we do take a few extra precautions in order to minimize the chance that a user will accidentally shoot themselves in the foot. Dogged determination to shoot oneself in the foot, on the other hand, is a little more difficult to prevent. The structure of the system helps avoid common multithreading pitfalls for users in the following ways:

Dependencies Declared a Priori

The system is set up such that users declare ahead of time the inputs that their node computations will consume. They can make certain dependencies optional (meaning that if they are not available, they can still produce a reasonable answer), but they cannot add more

dependencies at run time. The static structure of the network is fixed and is built based on the dependencies that the user declares.

Client Callbacks Are Static Functions

All client callbacks of the execution system are expected to be static functions (i.e. not class methods), that are passed a single argument. They take the following form:

```
static void MyCallback(const Context &context) {
    …
}
```

The callbacks are disallowed from being methods on objects by structuring the code in such a way as to force the binding of the callback before the user can have access to any of the specific instances of the objects in question.

Clients must read their inputs and write to their outputs using only the API provided by the passed-in context. This structure discourages users from storing any state for a node that is not known to the execution system. Users are not intended to derive new node types.

Presto Singletons are Protected

Some of Presto's libraries provide API that allows users to access system functionality through singleton objects and registries. Some of these libraries are not thread safe and are not allowed to be called from user-code running inside of execution. As a safety measure for client-code, we detect and prevent the use of such singletons while running in this context. Users are of course still free to create their own singletons and access static data, but they must be responsible for the consequences of doing so unsafely.

Iterators

Access to large, vectorized, data (e.g. points) is provided through iterators that are easy to use and hide from the user the detail of where the memory is stored, or what subset of the computations their callback is dealing with. This allows us to modify memory allocation and access patterns, as well as modify our multithreading strategies, without changing client code.

And then there's Python...

Python is famously known to not play well within a multithreaded system. For this reason, we

initially disallowed the use of Python for writing execution system callbacks. However, there are some clear advantages to supporting Python:

1. Python allows for quicker prototyping and iteration.
2. Python is accessible to a wider range of users than C++.
3. Python has a rich set of packages (e.g. numpy) that users would like to leverage.

These benefits make it difficult to adopt alternatives (e.g. a different existing language, or a custom-written language).

Global Interpreter Lock

The Python interpreter is not threadsafe. It cannot be run from multiple threads simultaneously. Python provides a global lock, the Global Interpreter Lock (GIL), that clients can use to make sure that they don't enter the interpreter from multiple threads simultaneously[1]. A thread that needs to use Python must wait for its turn to use the interpreter.

Getting the locking right is tricky; it is easy to find yourself in classic deadlock situations. Consider the following user callback (though this code should be discouraged due to its use of locking to begin with, it nevertheless happens):

```
static void MyCallback(const Context &context) {
      Auto<Lock> lock(GetMyMutexFromContext(context));
      ...
      EvalMyPythonString(str); // A function that takes the GIL
      ...
}
```

Now consider the sequence of events in the threads which result in a deadlock:

MAIN THREAD	OTHER THREAD
Python command acquires GIL	Work started
Computation Requested	MyCallback runs and acquires myMutex
	MyCallback now waits for GIL
MyCallback runs and waits for myMutex	(waiting for GIL)

[1] http://www.dabeaz.com/python/UnderstandingGIL.pdf

One thing to note about this situation is that if, in the main thread, the call was made from C++, then there would be no need to hold the GIL in the main thread, and everything would be fine. If, however, it is called from Python, we get the hang. Moreover, neither subsystem knows about the other, the locks are taken in client code. The client code could be smarter about the order in which the locks are acquired, but that's not always a viable solution. In this case, the client is calling out to a function in a library, and may be unaware about it taking the GIL to begin with.

One solution in this case is that, in the main thread, we no longer need to be holding the GIL once we make a computation request in C++. Ideally, you would structure your bindings to always release the GIL upon re-entry to C++.

This is a good example of why global, system-wide, locks are a bad idea. Use lock hierarchies[2] to avoid the common deadlock patterns if you must have wide-reaching locks. Better still, prefer locks that have local, easy to reason about, scope if you must lock at all.

Performance

Anything running in Python is the only thing running in Python. This means that if your execution callbacks are all implemented in Python, you lose all the benefits of a multithreaded system.

However, Python can still be effective for writing the control logic and have the heavy lifting be performed in C++ (with the GIL released).

Memory Access Patterns

Although we tried to construct the system in as flexible a way as we could, a major guiding principle was to make sure that we paid attention to memory access patterns. How memory is accessed is extremely important for performance, and we wanted to make sure that our desire for a flexible system did not impose any detrimental patterns for memory access.

It's important that bulk data is processed in a way that is compatible with the processor's prefetcher[3]. Luckily, modern processors are clever and do a good job at prefetching memory, therefore, it is preferable to remain within the patterns that the processor can detect and pre-fetch. Hardware prefetchers typically work best with ascending access order. Though more complicated patterns can also be detected, such as descending and uniform strides, it's always best to check your hardware specifications. Arranging your data in a structure-of-arrays rather

---

[2] http://www.drdobbs.com/parallel/use-lock-hierarchies-to-avoid-deadlock/204801163

[3] http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf, section 2.1.5.4, page 2-23

than an array-of-structure often helps the prefetchers improve your application's performance.

Locality is important for performance. Most common multicore platforms available today use NUMA[4], so keeping memory and code access local to each core will improve the scalability of a multithreaded system. Memory allocators can help achieve this goal. Using an allocator written with this sort of scalability in mind, like `jemalloc`[5], is preferable to an unaware allocator.

It's important to always measure the performance of your system and monitor how changes to the code affect it. Modern hardware architectures are sufficiently complicated that intuition and educated guesses often fail to predict performance. Always test performance.

Flexibility to Experiment

We designed the system with two main unknowns: 1) we didn't know exactly what architecture it was going to run on, and 2) we didn't know how the user requirements were going to evolve. We didn't want to base our architecture on the current state of hardware and user desires. We attempted to build in as much flexibility as we could.

For example, we have written a variety of different executors to satisfy different computation models and multithreading strategies. If the target hardware architecture changes, we expect only to write a new executor, which is a relatively small piece of code. Schedulers are not often specialized, though they may be in order to add acceleration data for use with specialized executors.

Another example is if user requirements change such that the objects they deal with need to be re-thought or re-designed, much of the execution system can remain untouched. In this case, only compilation would need to be re-written. Though, admittedly, that is a lot of code.

Multithreading Strategies

Finding the right granularity for the tasks to run in parallel is an important aspect of getting the most performance from the hardware. Too many small tasks cause too much time to be spent in context switching and other thread management overhead. On the flip side, too many large tasks can lead to poor utilization.

One of the factors we have to keep in mind while choosing a granularity for our specific domain

---

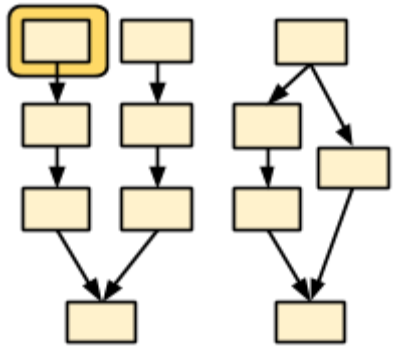
[4] http://software.intel.com/en-us/articles/optimizing-applications-for-numa

[5] https://www.facebook.com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919

is that we have a fairly small time budget for evaluating the network. We'd like to aim for running the rig and drawing the character at 24 fps. Even if we ignore the costs of drawing the character, that gives us less than 42 ms to run approximately 30,000 nodes (e.g. for a light character). We therefore have to choose a granularity for the units of work that is compatible with these target time budgets.

The architecture described above allows for easy experimentation with various multithreading strategies in order to find the right granularity for the greatest performance. In this section we'll explore only a few of the possibilities. Note that these strategies are not mutually exclusive and can be combined.

Per-Node Multithreading



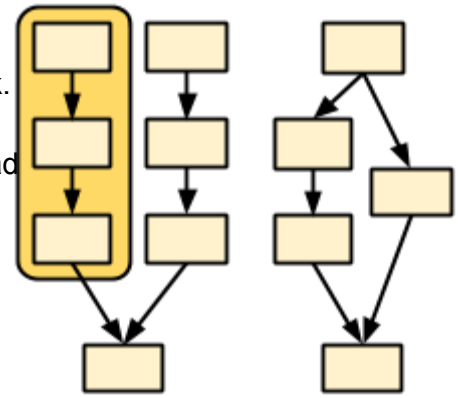
By per-node multithreading, we mean that each node independently runs its computation in a multithreaded fashion. In order to implement this scheme, nearly no specialization of the execution system data structures are needed. So long as the algorithm in the node is efficiently parallelizable, this might be a viable way to get a performance boost from multithreading.

A node doesn't typically need to synchronize or inform the system in any way that it intends to run its algorithm in multiple threads. It is also free to use any multithreading infrastructure that is deemed appropriate, for example Intel® TBB or OpenMP®, or the system's own implementation. The advantage of using the system's implementation is that it can coordinate the total number of threads in flight and can help avoid oversubscription.

In practice, the majority of our nodes run in a small amount of time and don't lend themselves to efficiently multithreading. While some particularly heavy nodes are multithreaded in this way, most of the callbacks in our system do not use this mechanism.

For most of the nodes in our system, per-node is too fine grained. We next look at a larger task unit.

Per-Branch Multithreading

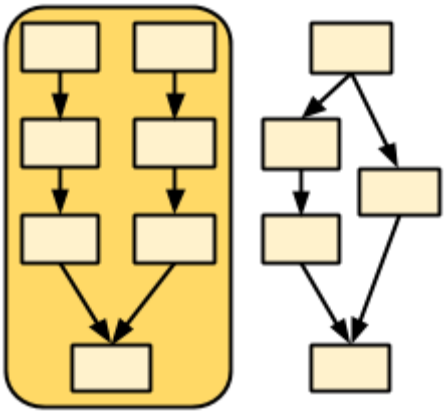


Another strategy is to launch a thread per branch in the network. Once the executor reaches a node that needs multiple inputs to be satisfied, each input branch can be launched in its own thread and can execute in parallel.

In order to implement this scheme, the executor would have to be specialized as well as the scheduler.

The amount of work that can be done in each thread here is much better, provided that your network has a lot of branches, each of which has a significant enough amount of work. This assumption does not always hold in our networks, and we find that often the largest amount of work is along a linear spine, and the branches are relatively cheap in comparison.

Per-Model Multithreading



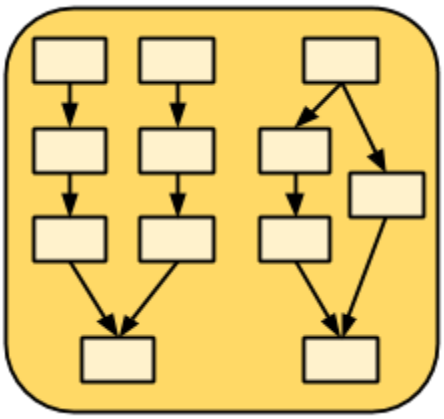
An even larger level of granularity is to run each model on its own thread. This is fairly straightforward to implement. The scheduler first finds all the disconnected subgraphs in the network, and the executor launches a thread per subgraph.

This is a very good amount of work to do per-thread, and generally works well. It runs into a couple of hurdles.

The first is that many of the scenes in our movies are animated using a small number of characters loaded at any one time. Smaller, say, than the typical number of cores found in modern hardware. This means that some cores can sit idle, which is not desirable.

Secondly, characters are rarely independent. They often have constraints among them. These constraints limit the parallelism between them, decreasing the system's potential performance.

Per-Frame Multithreading



Up until now, the strategies we've discussed don't take advantage of any domain-specific knowledge. We could employ these techniques for the evaluation of any data flow network. If we consider that our software's primary goal is animation, we find a whole new dimension along which to parallelize: time.

By allowing the system to compute multiple frames concurrently, we can significantly boost the performance of very common workflows. This approach lets us design a system where thread synchronization is not necessary at all while the concurrent tasks are running. This approach also introduces its own set of challenges that are due to its asynchronous nature.

We call this feature of our system "Background Execution", and that is the multithreading strategy that we found most successful for us and that we will discuss in more detail.

Background Execution

Background execution is the ability of our system to schedule frames to be computed in asynchronous background threads. This feature allows users to continue working while frames are computed.

The system architecture allows for easy implementation of the feature in its most naive incarnation: grab the frames that you need to compute in parallel, create an executor for each thread that you have available, and start doling out one frame per task. This approach leads to several practical issues that need to be considered when embedding in an interactive application.

User Interaction

A typical use-case is that a user will change an attribute value and then scrub on the timeline to see the effect the change has had on the animation. We would like to use the Background Execution feature to make sure that the computations of neighbouring frames have completed by the time the user scrubs to them. So as soon as the user stops modifying the value, background threads are kicked off to compute the frames that have become invalid due to the attribute change operation. How these frames are scheduled is discussed below. By the time the user scrubs over to the new frames, the idea is that the system would get cache hits for the frames and be able to return much quicker than it would have otherwise.

While the background threads are actively computing, the user is free to modify the values again, possibly invalidating the existing computations. The user's work can't be delayed by what the system is doing in the background, and we'll discuss interruption policies below as well. The main takeaway is that it's unacceptable for the system to pause or hitch due to work that the user is not expecting: avoid performance surprises.

Along the same lines, providing users with feedback that work is happening in the background can inform them of what to expect in terms of performance. For example, a scheme where you see the animation fill in in the background, helps you predict that moving to a filled in frame will be faster than moving to a frame that has not yet been processed. The feedback is also valuable for those trying to debug the system. When things are working well, the user generally doesn't have to pay too much attention and can just work undisturbed.

Frame Scheduling

In order to effectively schedule the next frames to be computed, you would need to be able to accurately predict what the user will do next. For example, if the user wants to play the frames in order after making a modification, then you would want to schedule them in ascending order. If, on the other hand, the user will frame back and forth around the dirtied frame, then you will want to schedule the frames to bloom out from the dirtied frame. But we don't know ahead of time which the user will do. We would prefer not to make this an option to the user because the user is only in a marginally better position to make this decision. The choice is not necessarily known to the user a priori either.

Some of the possible approaches are:

1. Pick a scheme and stick to it (e.g. always ascending, or always bloom out).
2. Base the decision on how the frames were dirtied. For example, if they were dirtied in a spline editor due to a tangent change, then bloom out; if it was due a change in the knot value, then use the ascending order scheme. The editor in which the change originated can also be a clue.
3. Keep a record of the user frame changes and match the dirty frame schedule to the pattern observed in past frame changes.
4. Combine #2 and #3

Interruption

The user is free to modify the state of the application while the background tasks are in flight. In particular, they are free to modify the state in such a way as to make the currently scheduled tasks invalid. One of the major problems we therefore need to deal with is interruption. It is critical that during common animator workflows, for example, changing spline values, or playing back, that the background tasks not interfere with the foreground work. Any hitch or delay can

immediately be detected by the user and can annoy users or, worse, can cause RSI. These kinds of hitches are particularly irritating because users can't predict when they will occur or how long they will last.

We don't explicitly kill threads (e.g. through `pthread_cancel` or `pthread_kill`) because that is a problematic approach[6]. For starters, most of our code is not exception-safe. We also felt that tracking down resource cleanup problems that resulted from abrupt interruption of threads would be time-consuming and constant source of bugs. So we decided to avoid it altogether.

We also didn't want to burden clients with having to check for an interruption flag. That approach is problematic as well, as some clients may either perform the checks too often or not enough.

The system therefore completely controls interruption and checks for interruption after each node has completed. Since our nodes are typically fast, that granularity seems appropriate. Unfortunately, we occasionally have nodes that do take longer to run and waiting for those nodes to complete before interruption is unacceptable for some workflows. Although we may want to support a cooperative scheme in the future, our experience so far has been that it is always better to avoid the long running computation altogether. No one node can be allowed a very long runtime if we expect to stay within our 42 ms budget.

While animators are working, we cannot afford to wait for interruption at all. Animators can change values several dozens of times per second, and cannot be interrupted. Luckily, these common, high frequency tasks, do not change the topology of the network, and therefore do not impose a requirement for our system to actually wait for the background tasks to stop before changes are made. We take advantage of this fact by not interrupting at all during these workflows. Instead, we remove the remaining tasks and set a flag telling the threads to throw away their results once the next node is done computing. With this approach, animators are no longer bothered by what is going on in the background. As mentioned earlier, animator workflows that do not modify the network, such as adjusting spline values, adjusting tangents, inserting and deleting knots, and so on, do not require us to join with the background threads. On the other hand, tasks that do alter the network topology, for example, setting new constraints or adding new deformers, do require us to stop and wait for the background threads. For these less frequent tasks, we rely on node-level granularity of interruption to be sufficient.

Constant Data

Since we would like to avoid synchronization among threads while the computations are in flight, each thread gets to manage its own network data. We quickly notice that there could potentially be a lot of computations in common among the threads. Time-independent computations will all

[6] http://www.drdobbs.com/parallel/interrupt-politely/207100682

yield the same results in all threads. These computations also tend to be slow and memory intensive. They tend to be the kinds of computations that set up large projection caches or acceleration structures that don't change with time Launching all threads to compute this same data at the same time saturates the bus and decreases the system's throughput. We would like to avoid running computations redundantly, and yet maintain the lockless nature of the execution engine.

Our solution is to launch a single thread, which we call the starter thread. The starter thread's job is to compute all the constant data in the network and make it available to all the other threads. The starter thread uses a starter executor to compute the constant data, then schedules the background frames and kicks off all the concurrent threads. For each concurrent thread, we create a new executor that is allowed to read from the starter executor. This multiple-reader scenario is supported without needing to lock since the starter executor's data is read-only while the other frames are being computed. In addition to speeding up the background threads, this scheme also significantly reduces the amount of memory needed to run multiple threads and allows our memory consumption to scale much better with the number of available cores.

The starter thread allows the main thread to be free to respond to user interaction. It also runs infrequently: once at load time and only again when time-independent data is invalidated. We further decrease the amount of time this thread needs to execute by allowing it to also locklessly read constant data (i.e. data that can't be invalidated through the model's animation interface) that is computed only once per model.

Future Work

We're excited to keep experimenting with multi-core architectures and multithreading strategies.

The system, since the start, has been designed to support a strip-mining strategy for multithreading where all the requested vectorized elements are processed in parallel in equal chunks among the available number of cores. We haven't yet taken advantage of that approach but we would like to. This strategy would complement Background Execution and would be used for improving the speed of single frame evaluations. We feel that this approach has the potential to scale well with the ever increasing number of cores available at user desktops.

A small modification to our Background Execution infrastructure would allow us to perform predictive computations. In other words, while a user is continuously changing a value on an input, we can schedule computations for input values that the system predicts the user will ask for next.

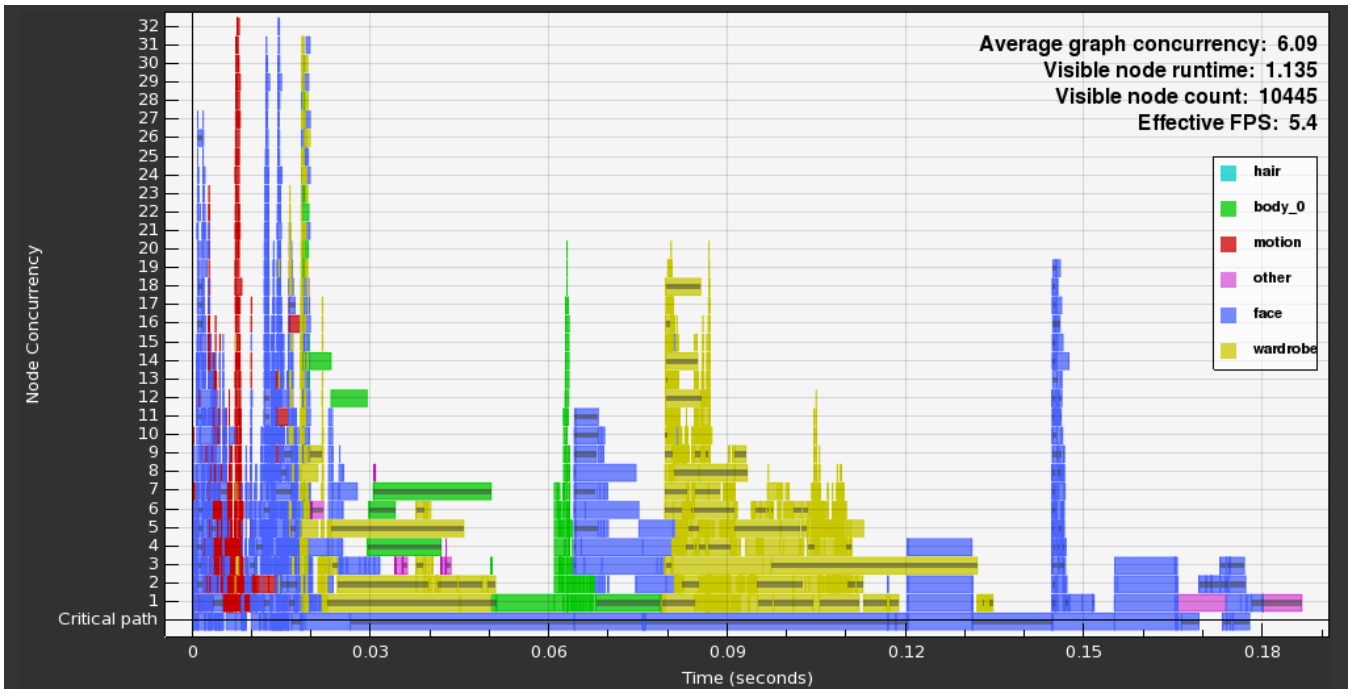
We'd like to experiment with writing a GPU Executor and a VRAM Data Manager, for example, to

run all or parts of our rigs on the GPU. Similarly, we'd like to explore the benefits of Intel's Xeon Phi™ coprocessor and see how we might adapt our architecture to get the best utilization on a very large number of cores.

We'd like to continue developing tools to help users introspect and profile their rigs. We'd also like to adopt or develop more and better tools to help us better identify performance bottlenecks and correctness issues.

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.

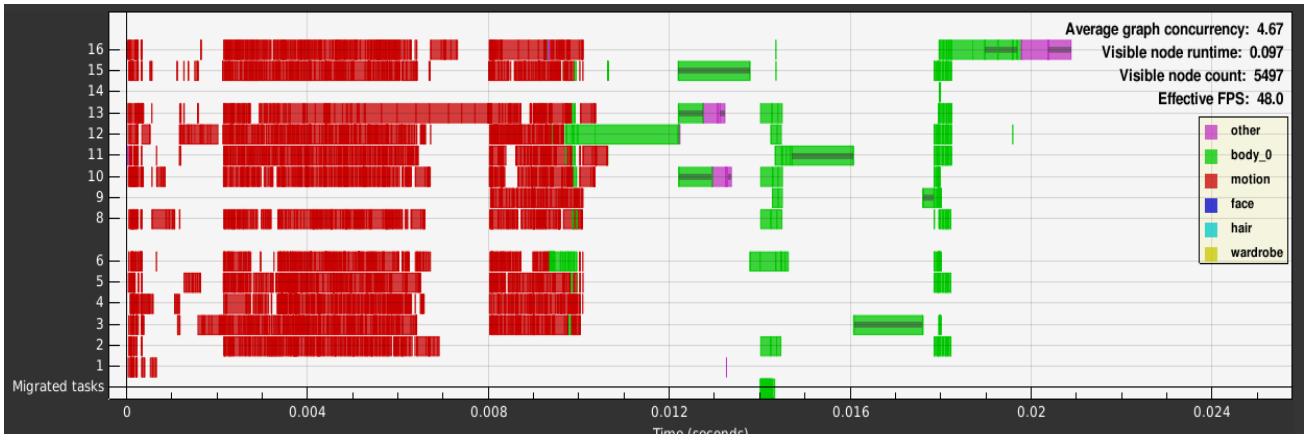
Tasks and nodes

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.

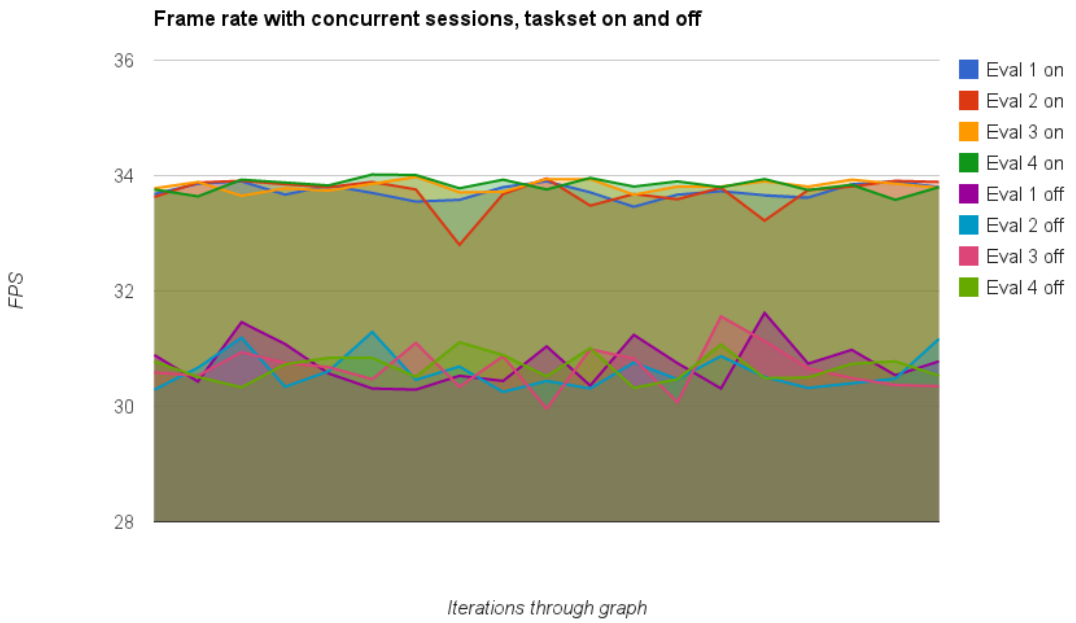
Other processes running on system

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.

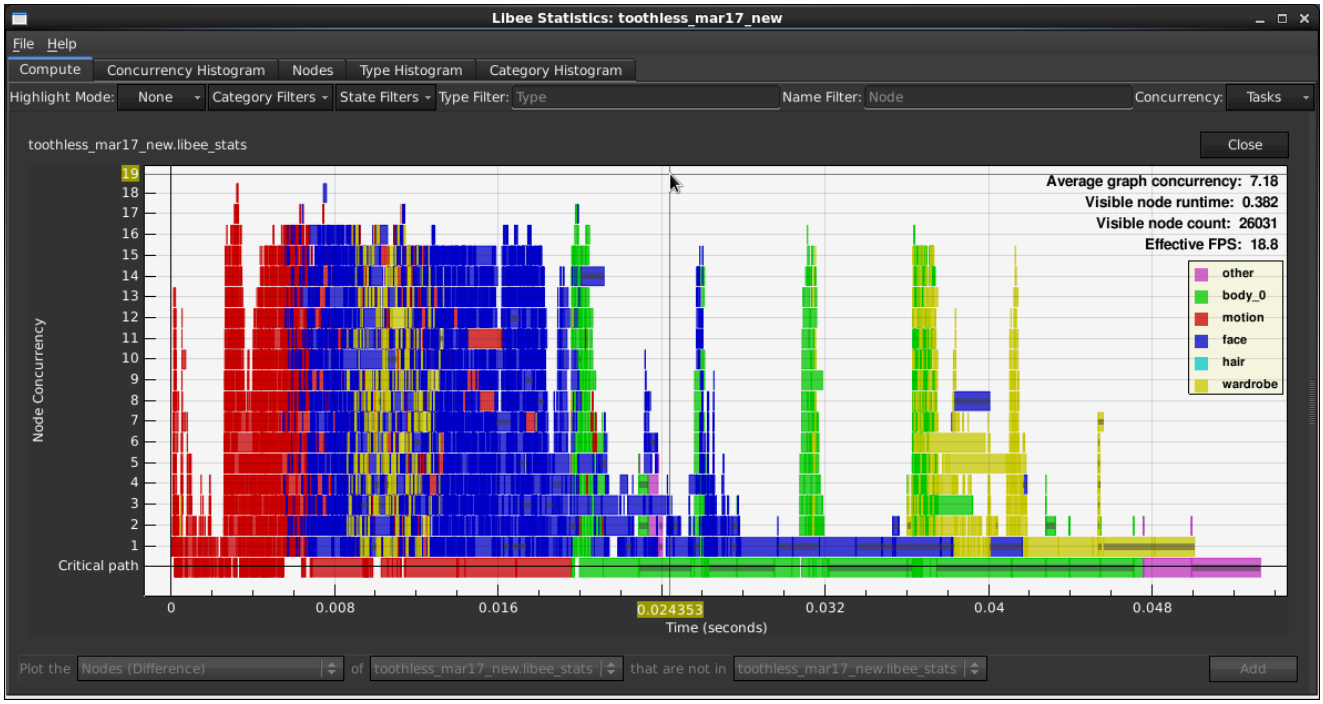
This is something that is obvious in hindsight, but which we did not anticipate, so the tool was developed relatively late, at a time when many of the characters for the first production show had already been developed and were no longer open to significant changes. This meant that characters on the 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.

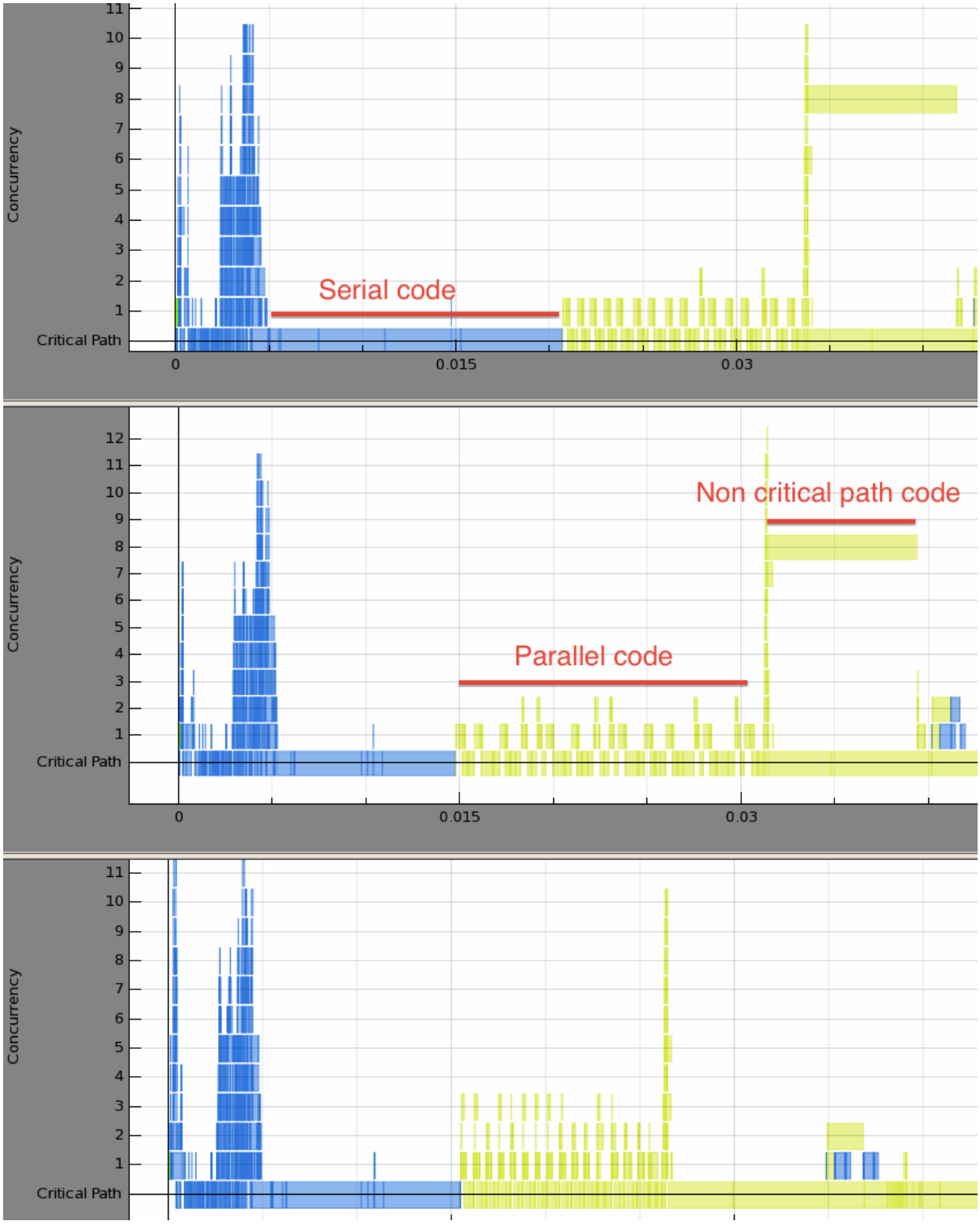
Case Study: Quadruped Claws

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.

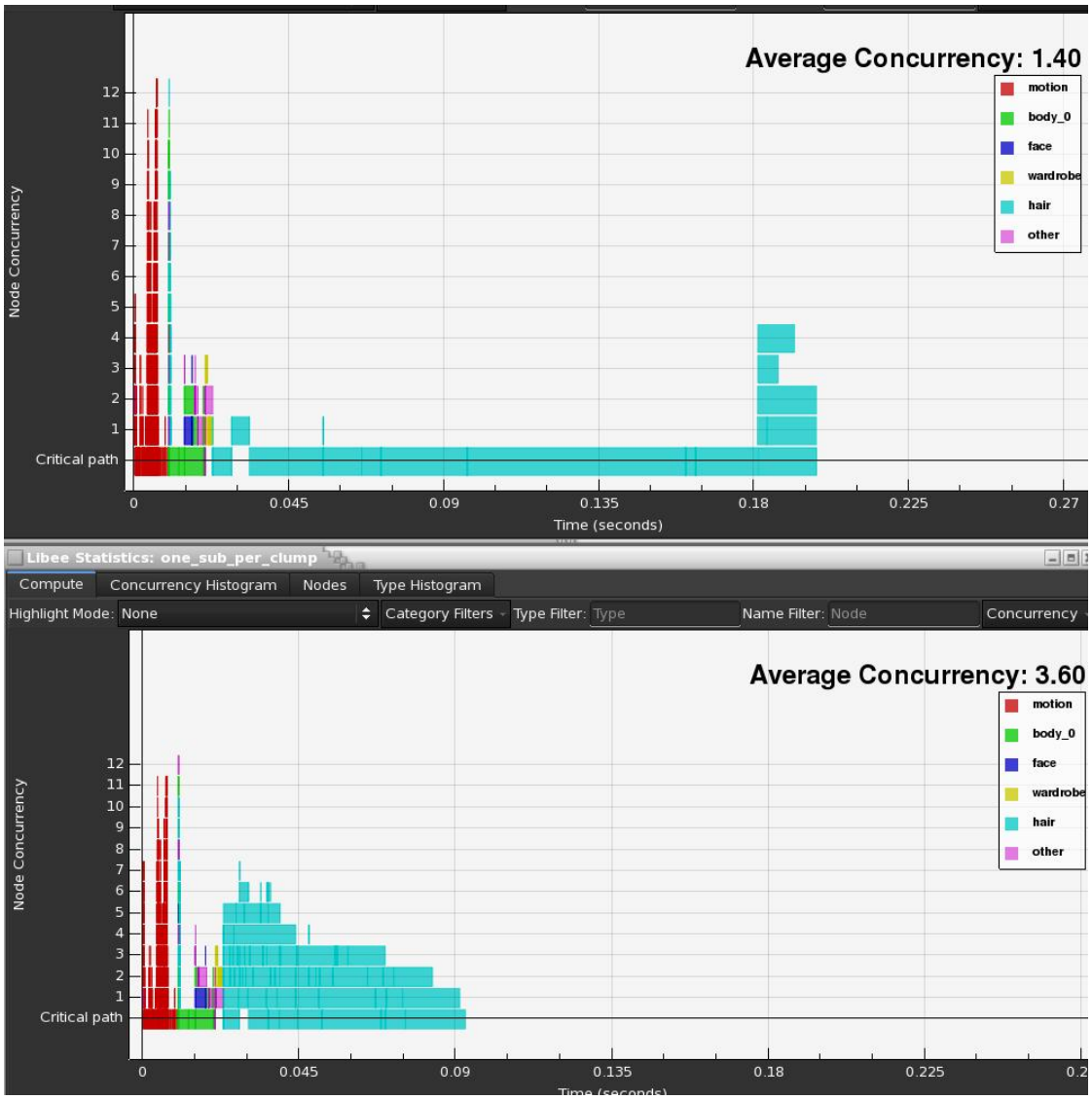
The third figure shows the results of these two character graph optimizations. The claw deformation code path in yellow has shrunk. However note that the second optimization, to the expensive node not on the critical path, did not significantly affect the overall runtime. This demonstrates the importance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

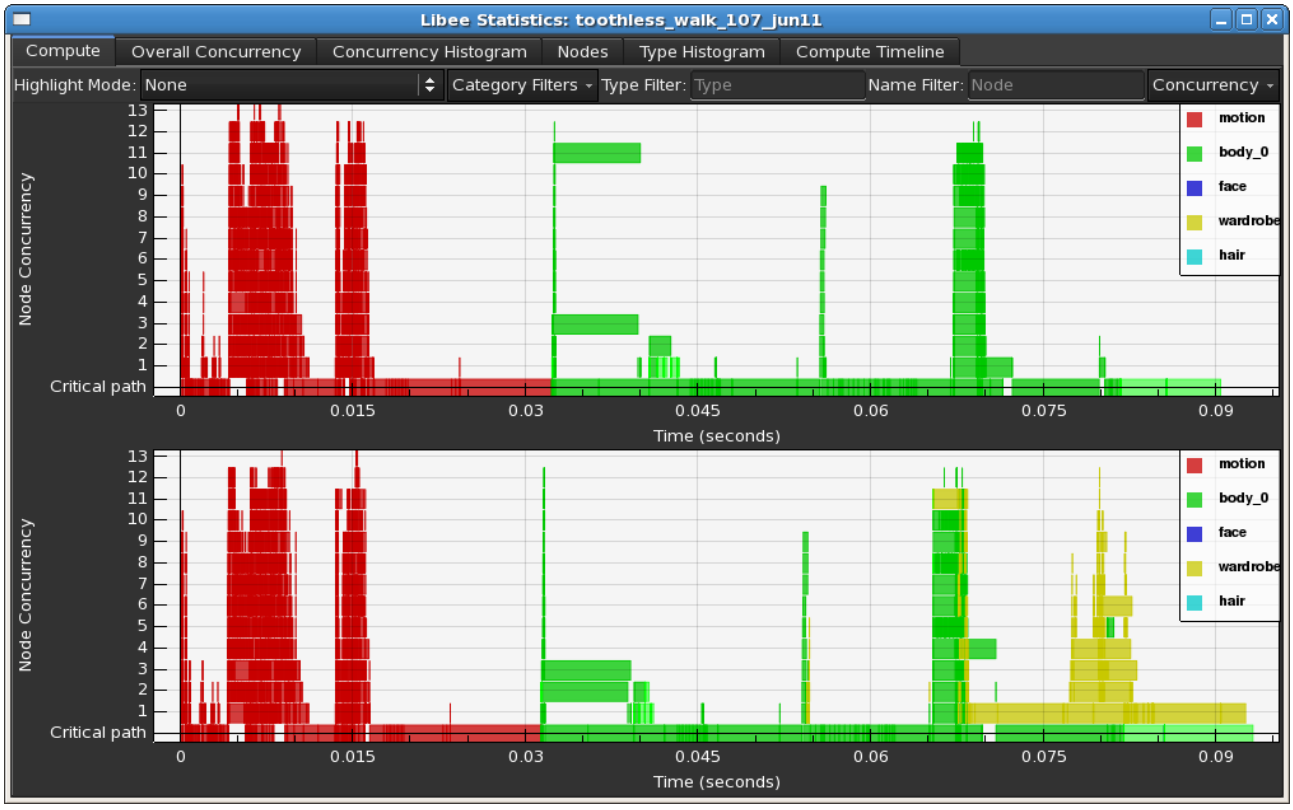


Top profile shows initial hair system implementation, bottom shows same workload with dependencies expressed between nodes in a way that allows more parallel execution

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

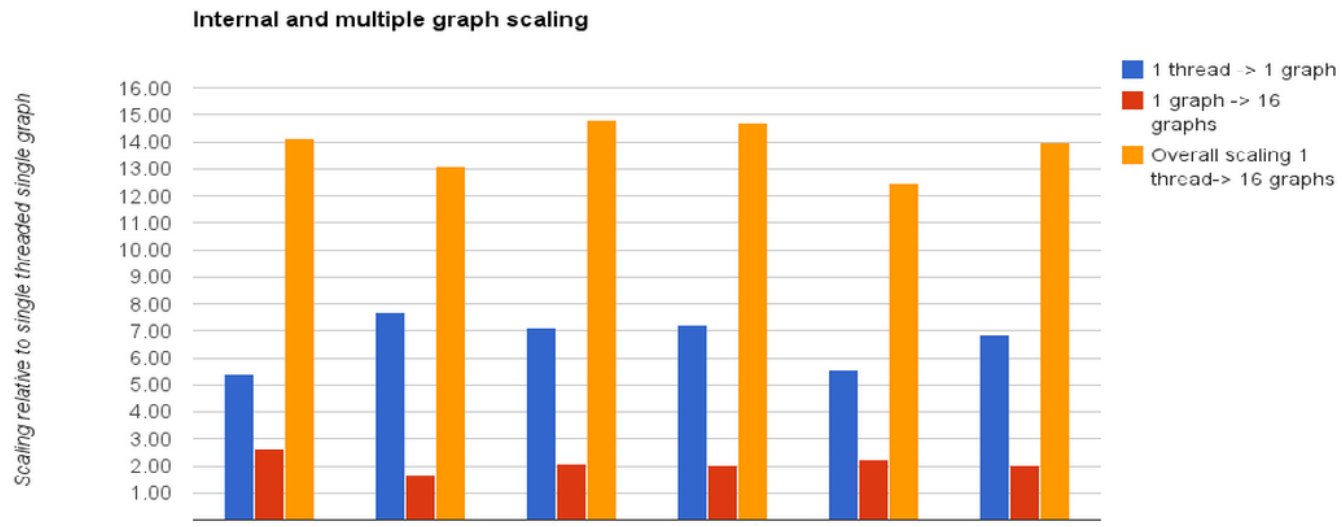
We spent significant effort in optimizing both the rigs and the nodes themselves to attempt to improve scalability. As indicated in the previous section, we are approaching 8x scaling on 16 core machines, which is a good result, but one has to ask if it is possible to do better. Core counts will only 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.