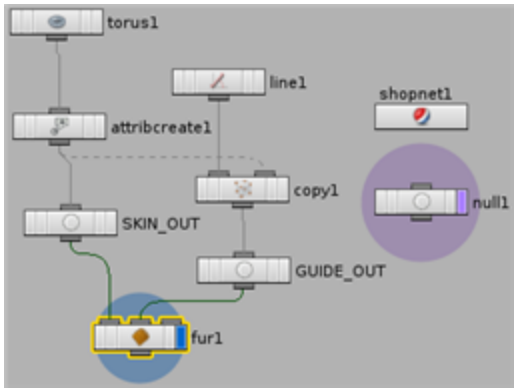
Multithreading Houdini

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.

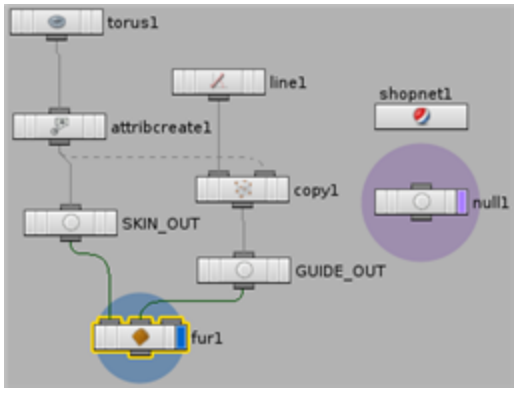
So, what sort of holes have we dug for ourselves over the decades?

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.