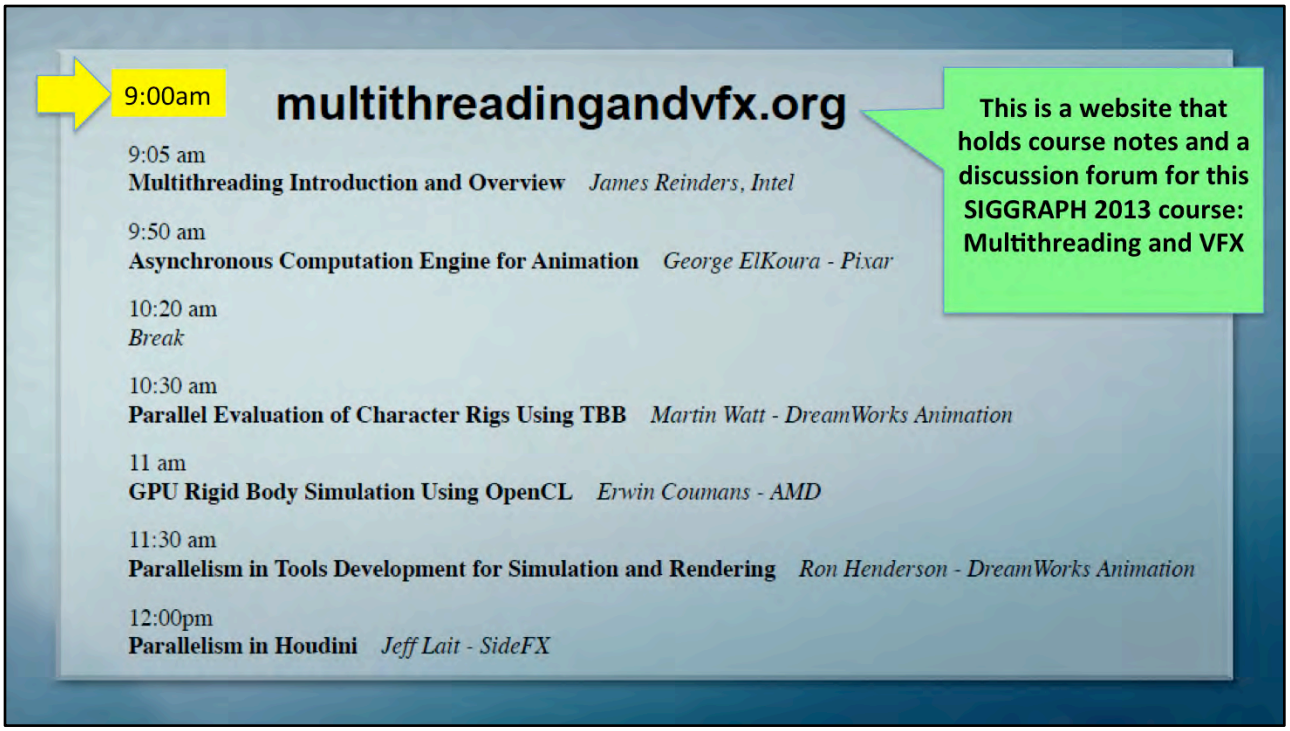


COURSE SCHEDULE
9 am Introduction
9:05 am Multithreading Introduction and Overview, Reinders
9:50 am Asynchronous Computation Engine for Animation, ElKoura
10:20 am Break
10:30 am Parallel Evaluation of Character Rigs Using TBB, Watt
11 am GPU Rigid Body Simulation Using OpenCL, Coumans
11:30 am Parallelism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

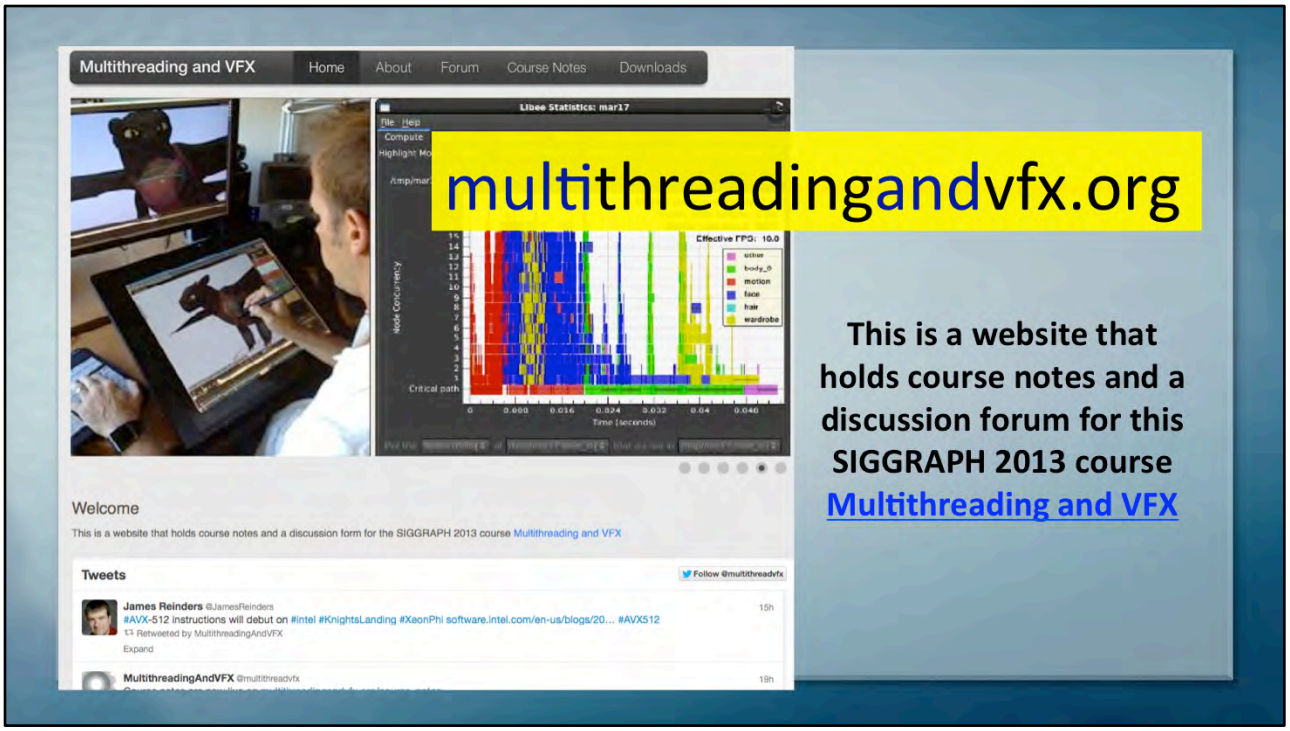
We are emerging from an era, of early adopters, to see parallel programming expand its reach and importance. James will discuss the rising role of parallelism and four key factors that are helping speed its growth: better tools, better programming models, significantly more hardware parallelism, and better educated programmers.

Topics include:
Parallel models: what makes a good abstraction?
- **Hardware agnostic, performance, "scale forward" (preserve investments)**
- **Reliable and predictable**
- **Effective use of scarce resource: us**

Examples of serious language issues:
- **Serial traps**
- **C++ futures**

The realities of vectorization (data parallelism)
- **how programming languages are the real failure point not compilers**
- **effective solutions**
- **future possibilities**

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



multithreadingandvfx.org

This is a website that holds course notes and a discussion forum for this SIGGRAPH 2013 course [Multithreading and VFX]

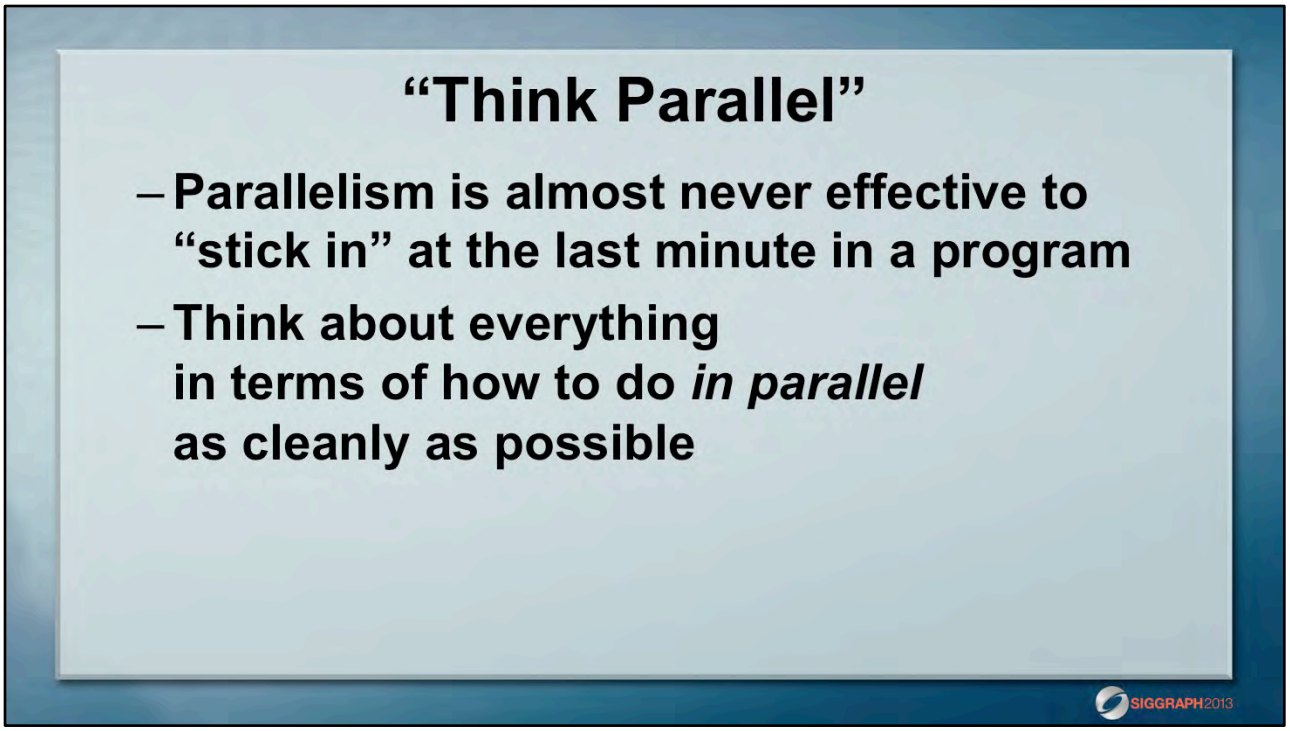
Multithreading and VFX http://s2013.siggraph.org/attendees/courses/events/multithreading-and-vfx

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.; Instructors: James Reinders, Intel Corporation; George ElKoura, Pixar Animation Studios; Erwin Coumans, Advanced Micro Devices, Inc.; Ron Henderson, DreamWorks Animation; Martin Watt, DreamWorks Animation; Jeff Lait, Side Effects Software Inc.

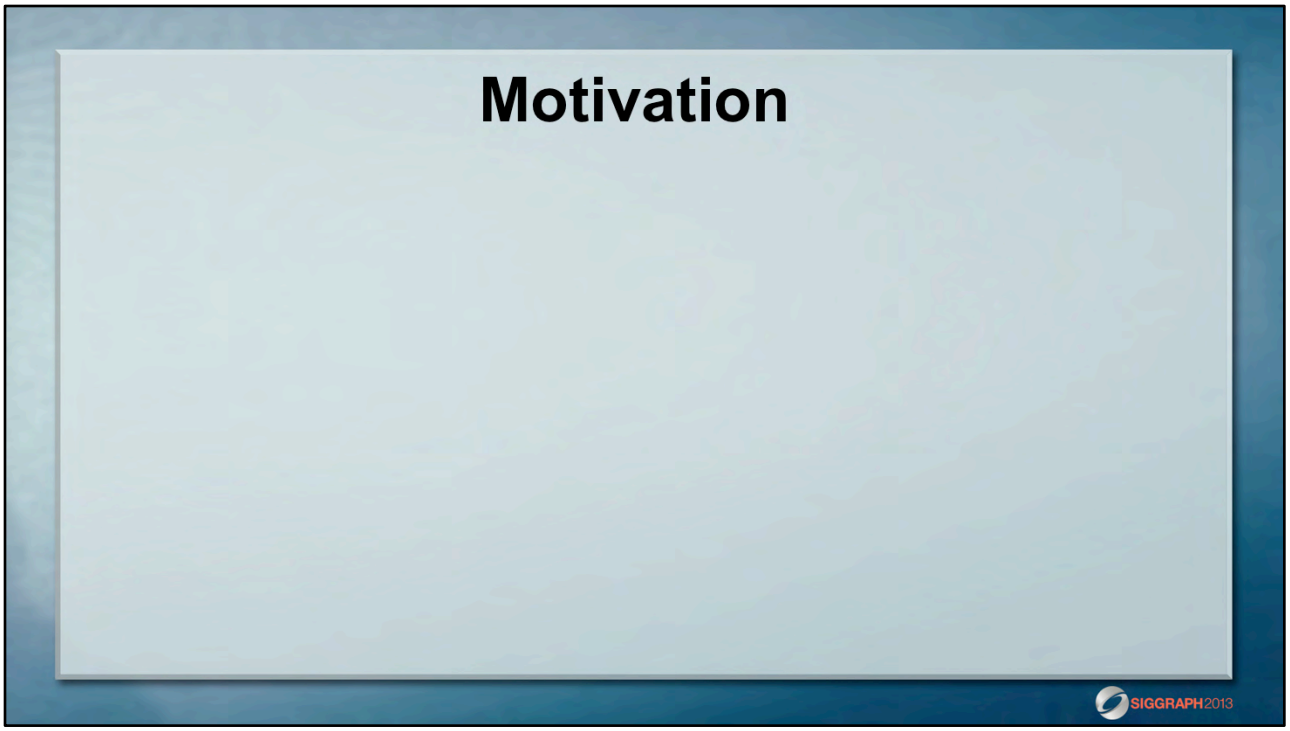
James Reinders, Director and Parallel Programming Evangelist and Senior Engineer, Intel
James is involved in multiple efforts at Intel to bring parallel programming models to the software industry—including models for the Intel® Many Integrated Core (Intel® MIC) architecture. He joined Intel Corporation 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 has authored technical books, including *VTune™ Performance Analyzer Esssentials* (Intel Press, 2005), *Intel® Threading Building Blocks* (O'Reilly Media, 2007), *Structured Parallel Programming* (Morgan Kaufmann, 2012), and *Intel® Xeon Phi™ Coprocessor High Performance Programming* (Morgan Kaufmann, 2013).



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.



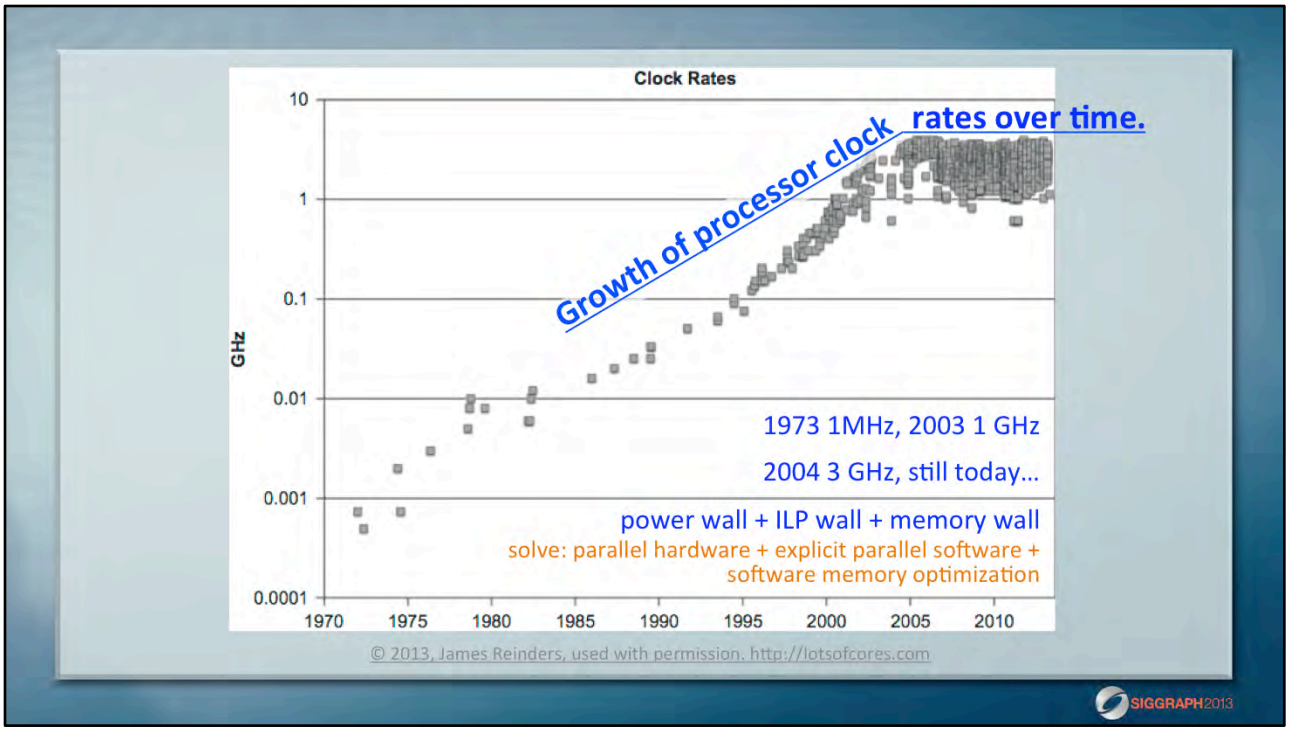
All computers are now parallel. Specifically, all modern computers support parallelism in hardware through at least one parallel feature including vector instructions, multithreaded cores, multicore processors, multiple processors, graphics engines, and parallel co-processors. This statement does not 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.

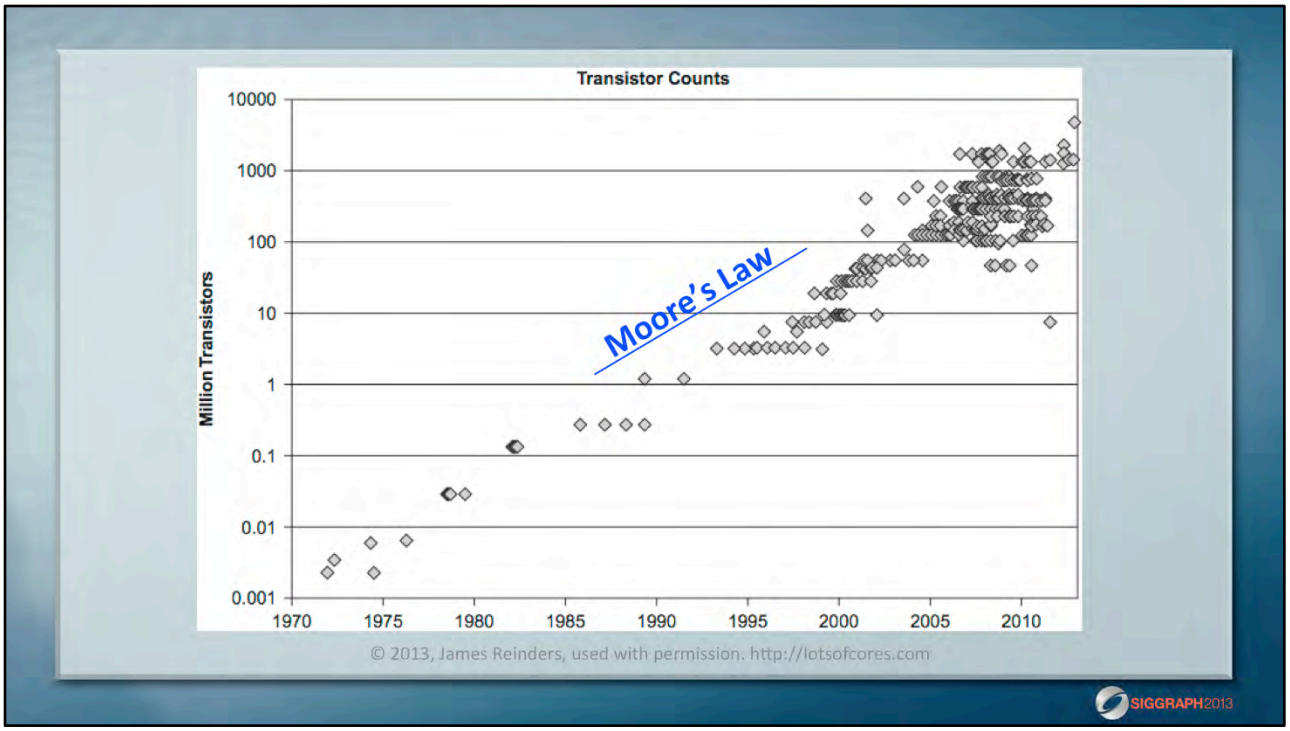
Until 2004, there was also a rise in the switching speed of transistors, which translated into an increase in the performance of microprocessors through a steady rise in the rate at which their circuits could be clocked. Actually, this rise in clock rate was also partially due to architectural changes 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.



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.

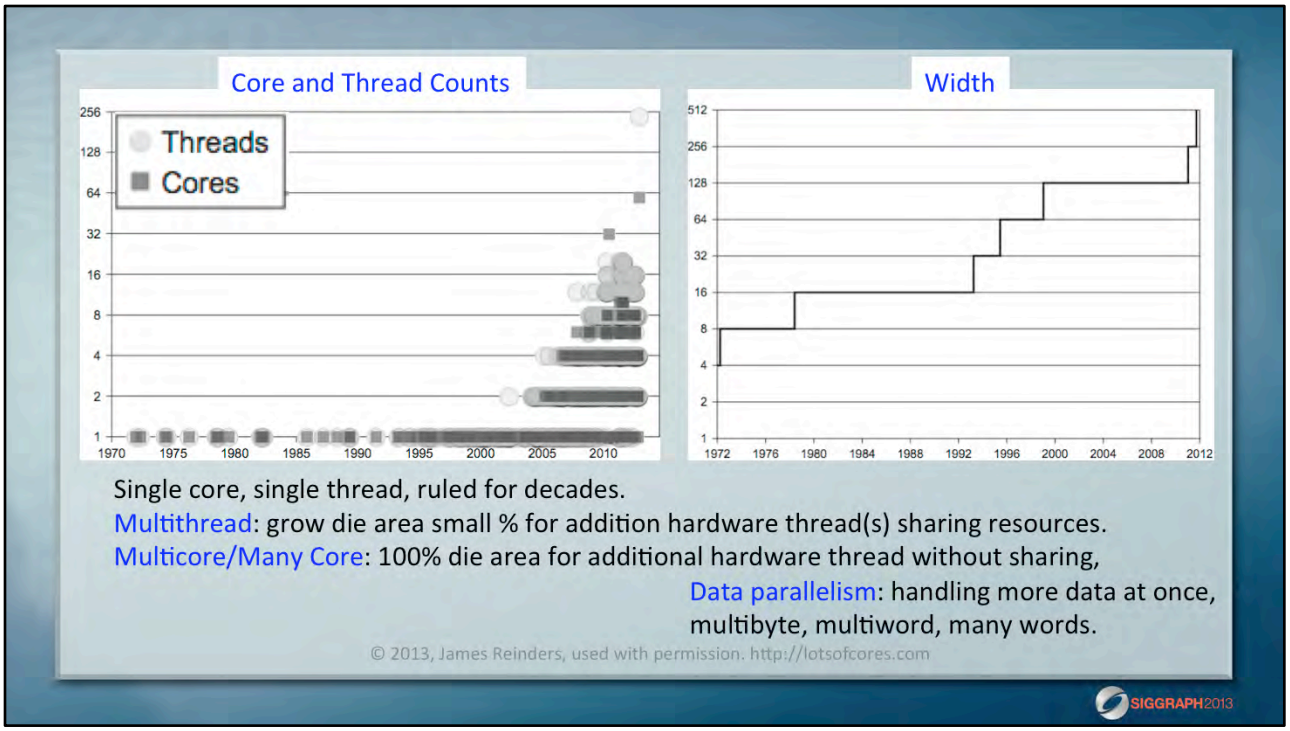
The data shows that Moore's original prediction of 2x per year has been amazingly accurate. While I only give data for Intel processors, processors from other vendors have shown similar trends.

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.

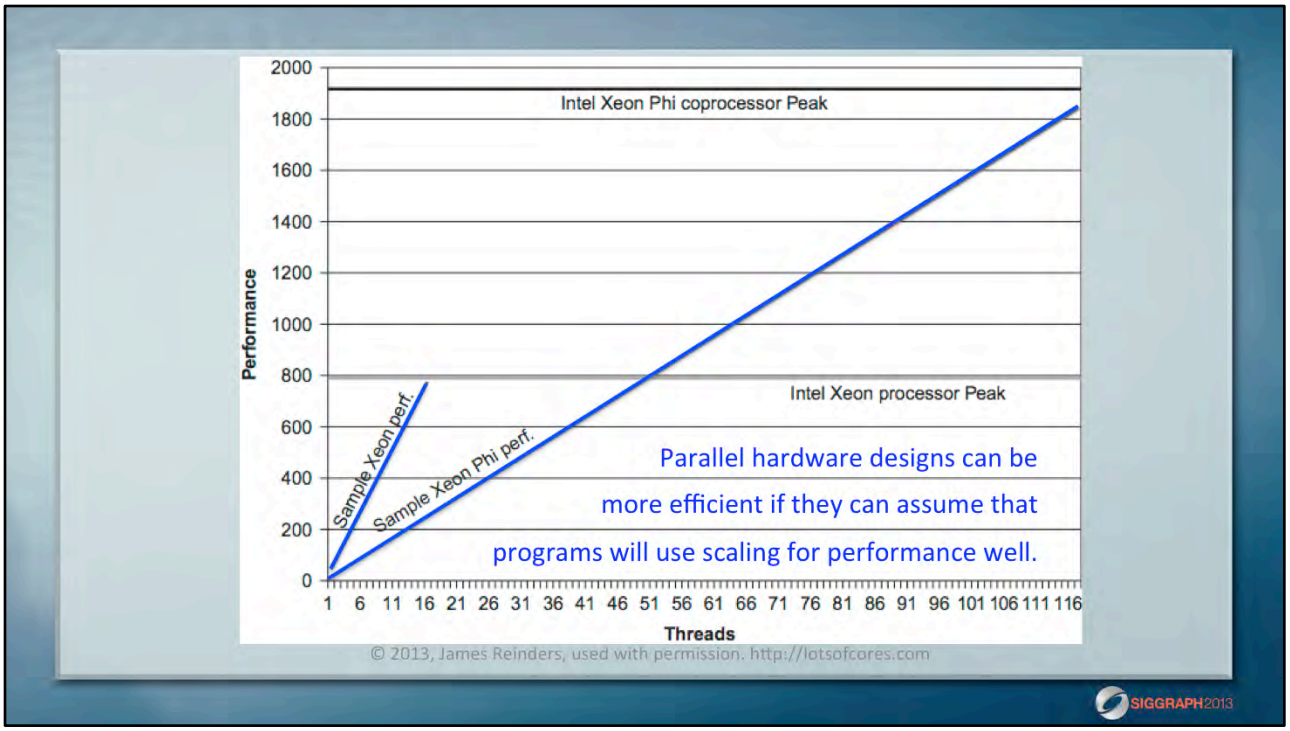


Cores and hardware threads per processor.

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.

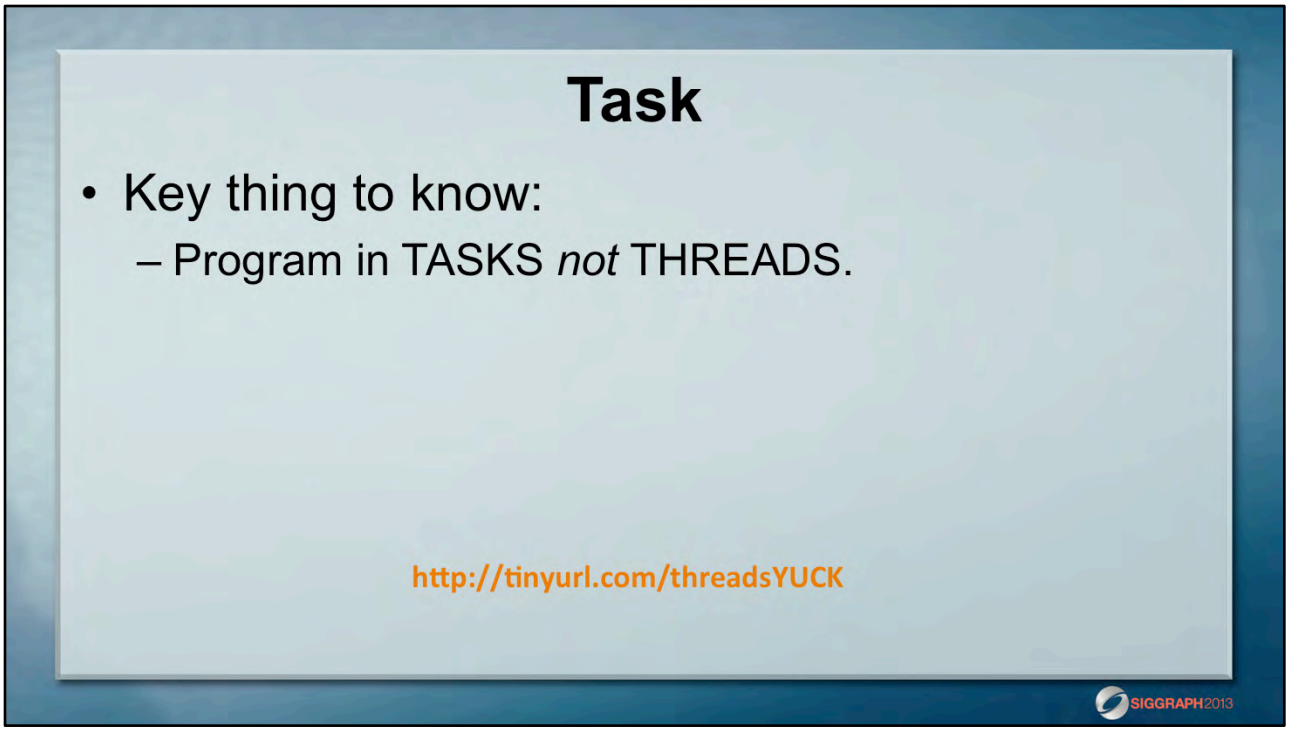
Multicore and Many Core devices use independent cores that are essentially duplicates of each other, hooked together, to get work done with separate software threads running on each core. [Note: The definition of multicore and many core are the same… a device with more than one core. Intel has used Many 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.



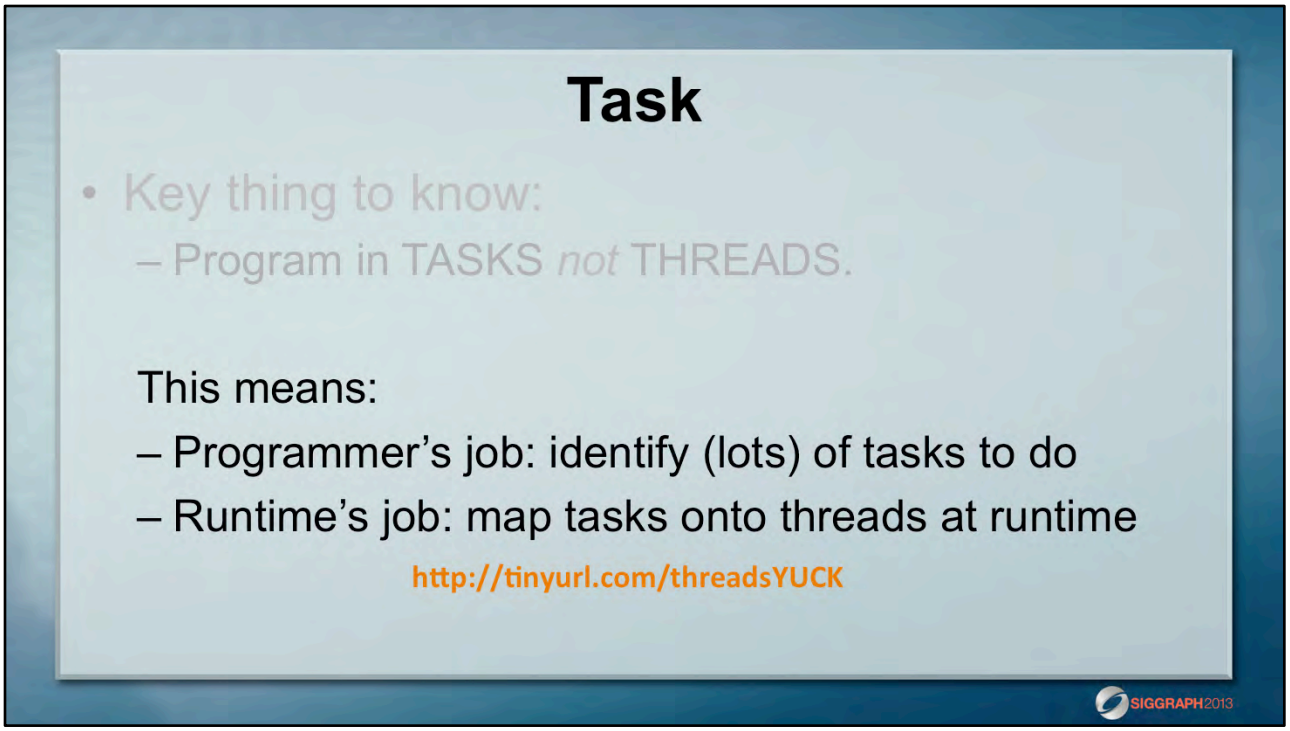
The importance of scaling

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).



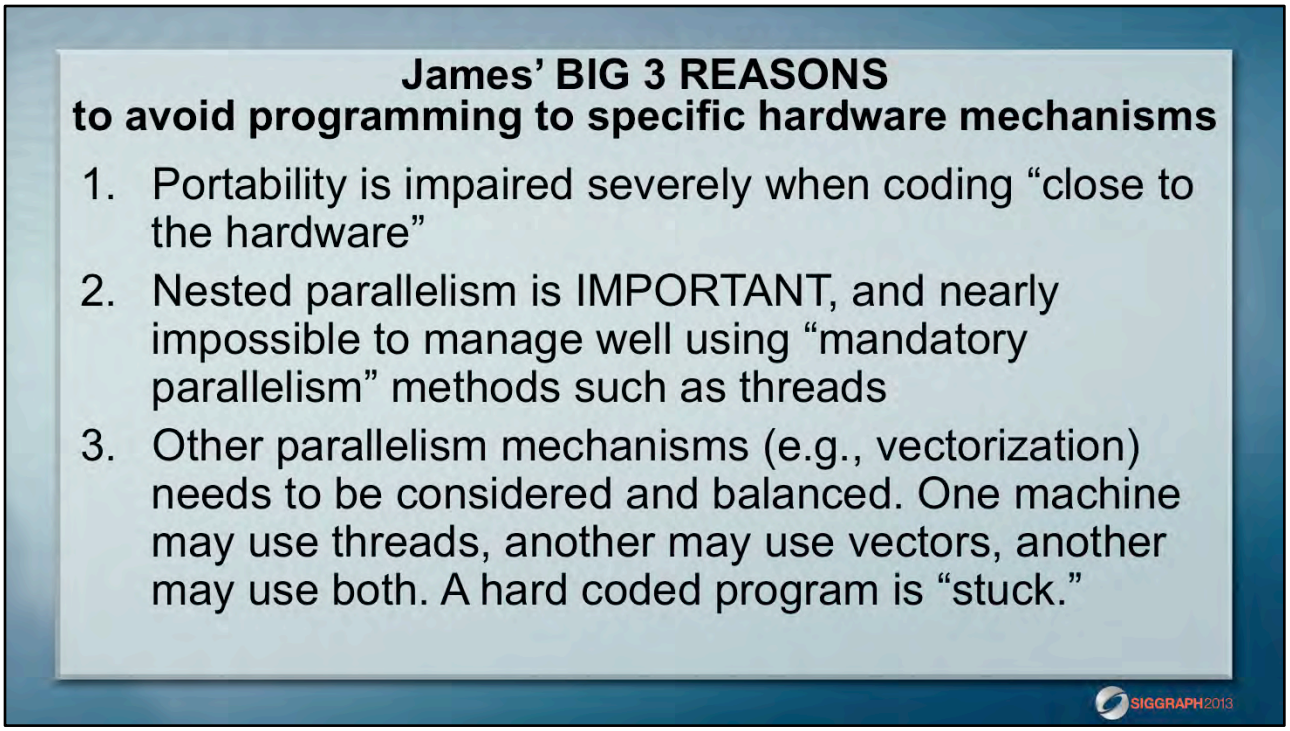
The "more solved" world of task parallelism
why task parallelism doesn't matter without data parallelism

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



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

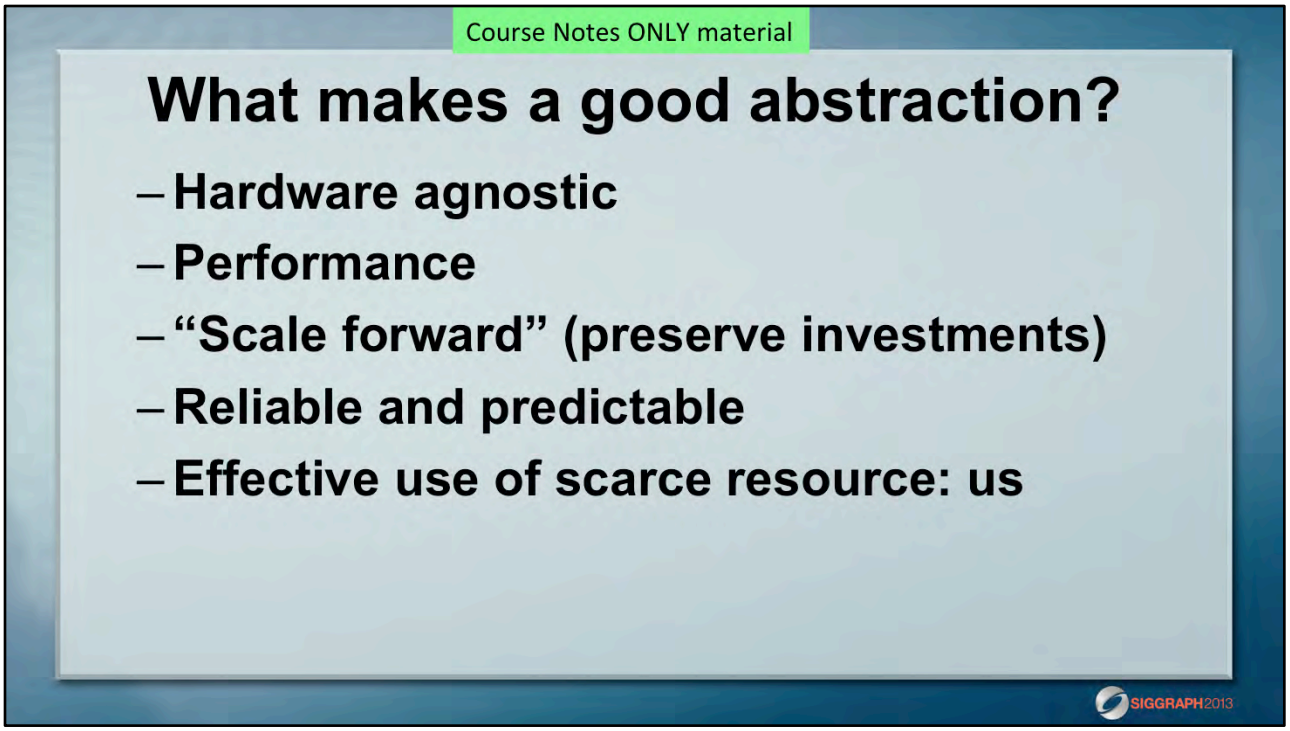
If this topic interests you…
I highly recommend a paper titled "The problem with threads"
By Edward A. Lee, University of California Berkeley,
Published 05-01-2006
http://tinyurl.com/threadsYUCK



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

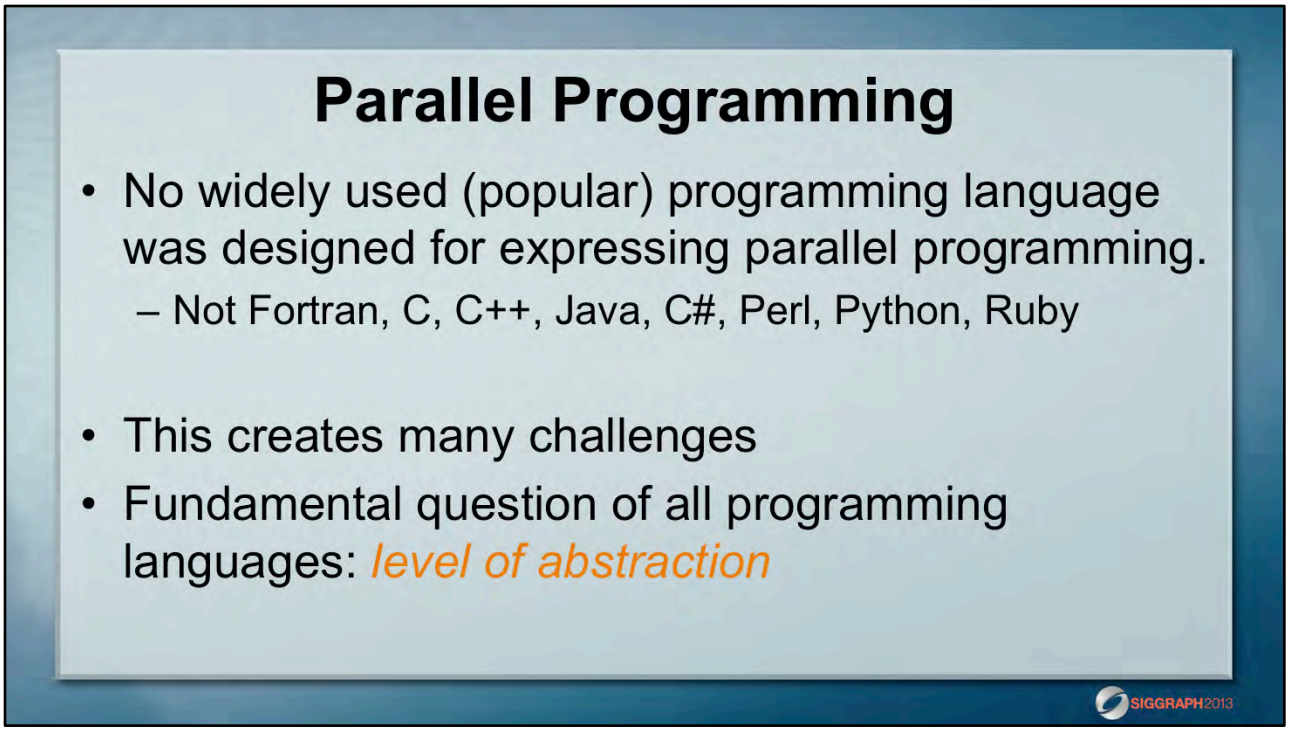
Nesting is something that turns out to be VERY important… we really didn't understand that when we designed OpenMP in the mid-1990s. That mistake remains the single biggest problem (unsolvable without a fresh start) with OpenMP. Live and learn!

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…



Parallel models: what makes a good abstraction?

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



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

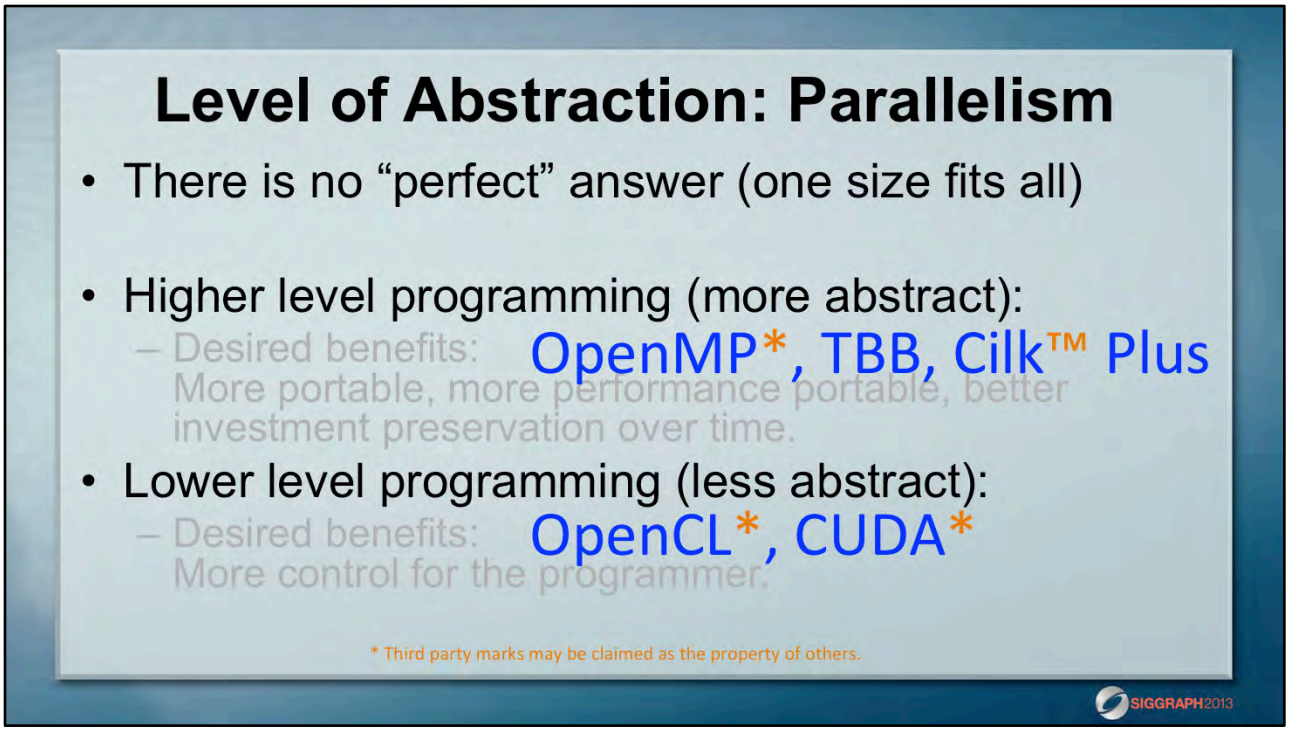
- Sequential Semantics?
- A fundamental design choice;
- most abstract solutions being “retrofit” into existing programming languages tend to choose to preserve sequential semantics.
- TEST: if ignoring the parallel keywords/directives/ calls would result in equivalent functionality (expected to be slower), 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

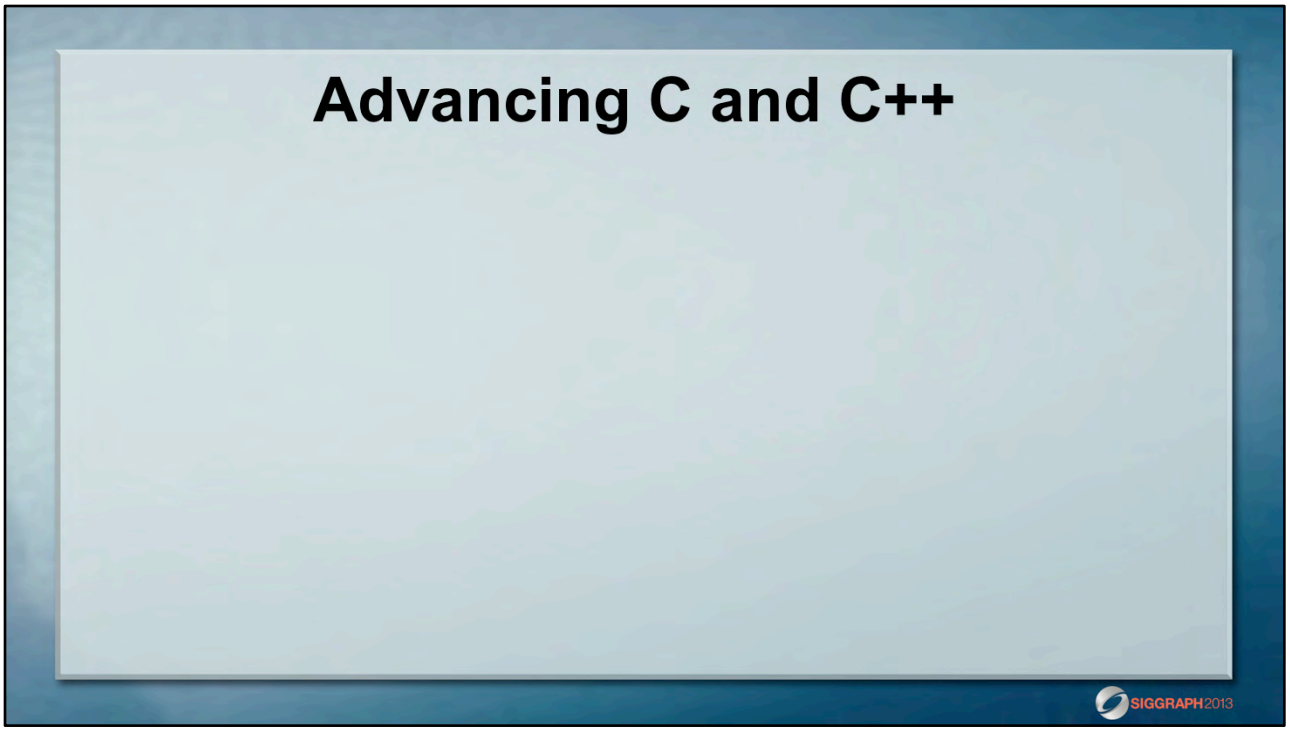
Level of Abstraction: Parallelism

- 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.



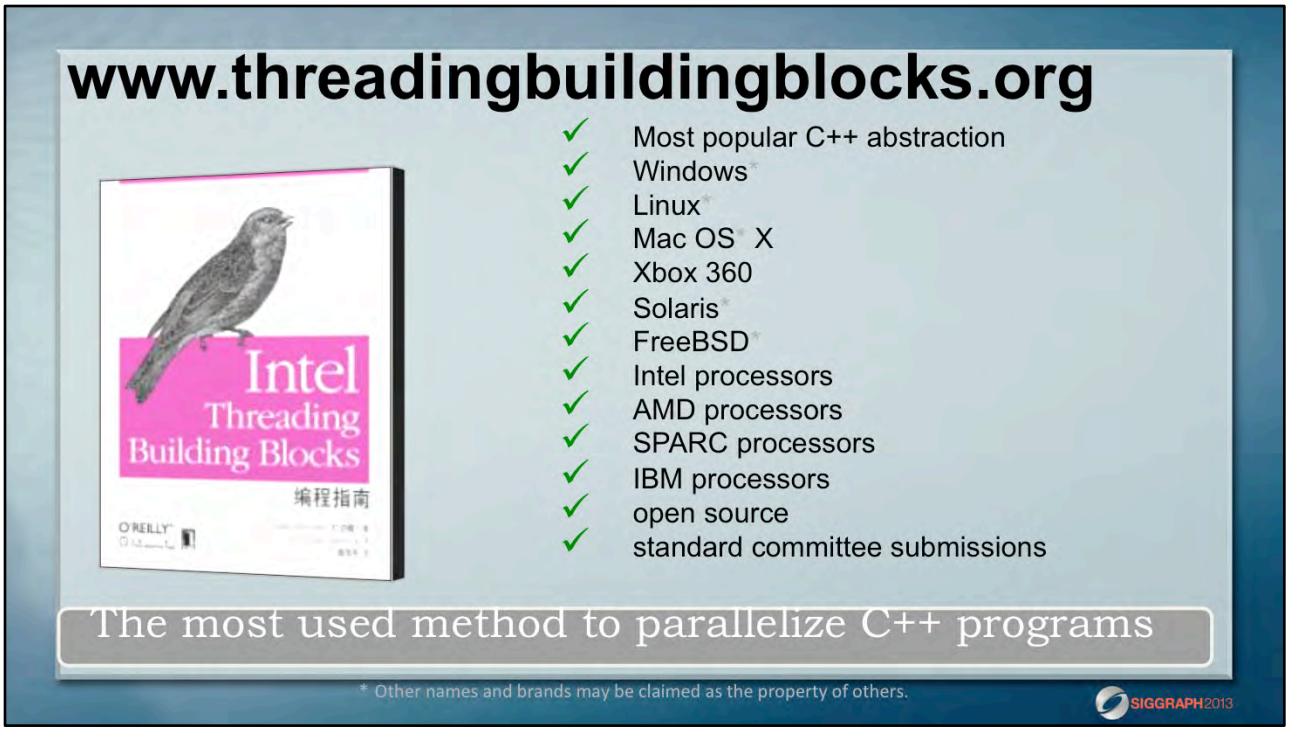
Yes, I like to encourage using the "high levels" of abstraction…

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

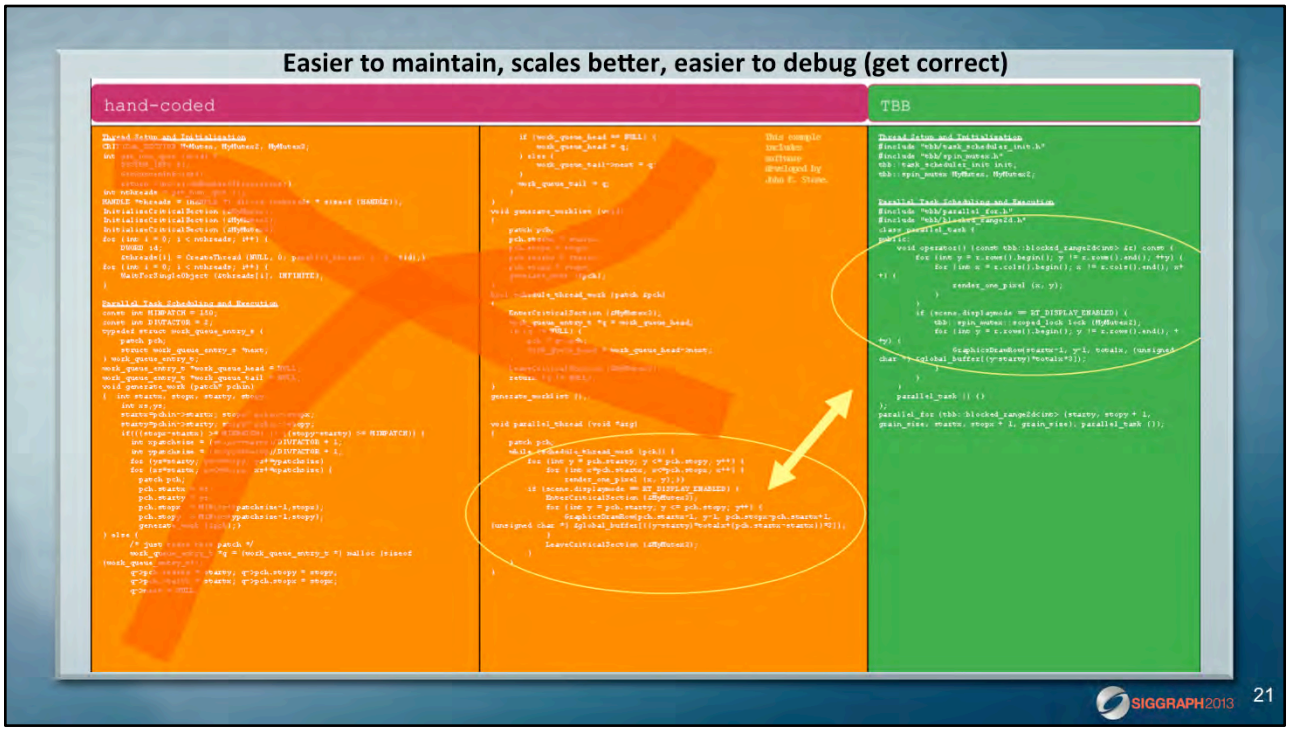


Examples of serious language issues:

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



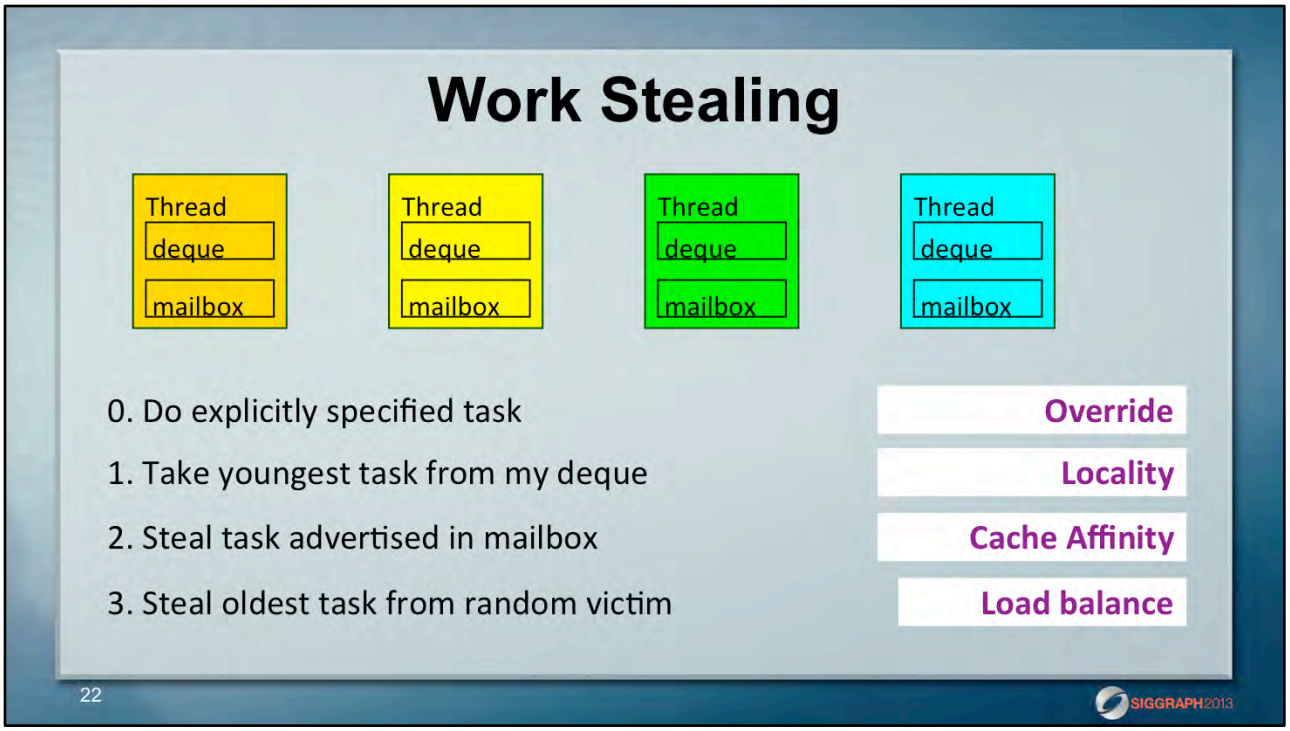
TBB is a very successful open source project – one of the most rewarding projects I've ever work on!



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"

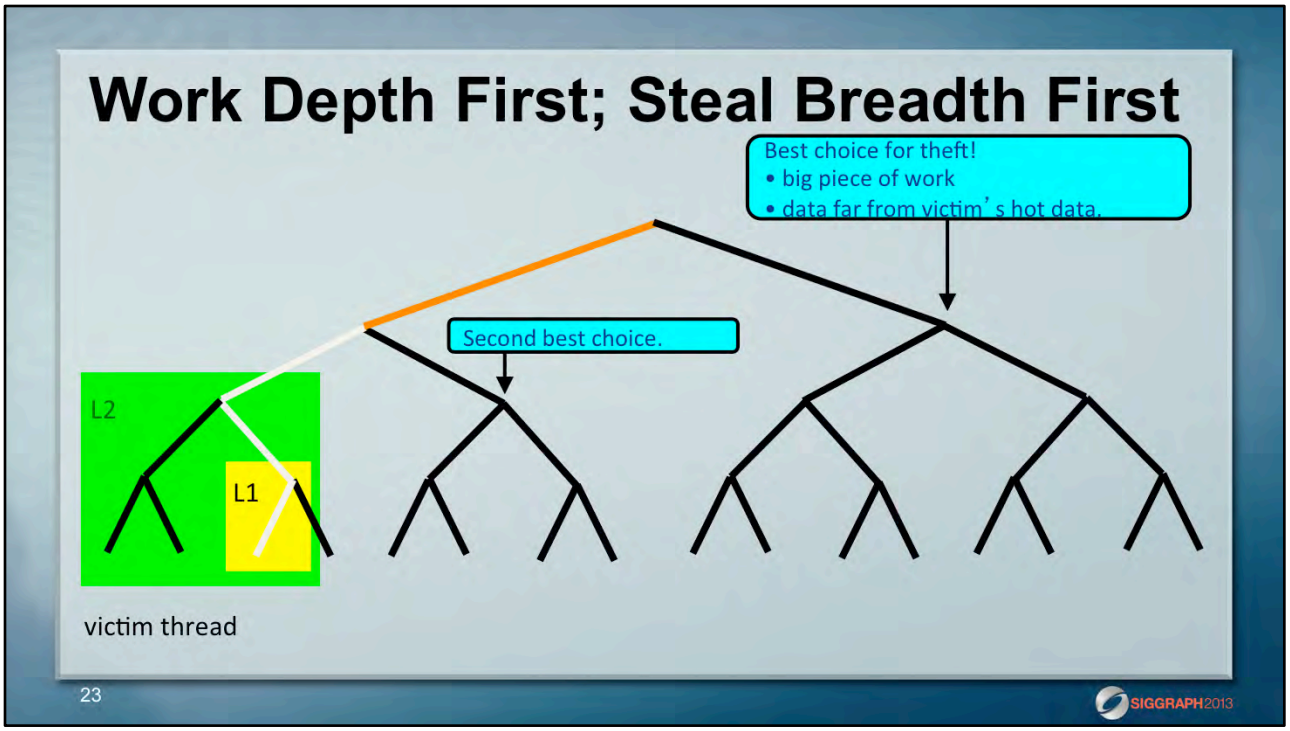
Intel introduced Intel® Threading Building Blocks for C++ developers in 2006

- 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

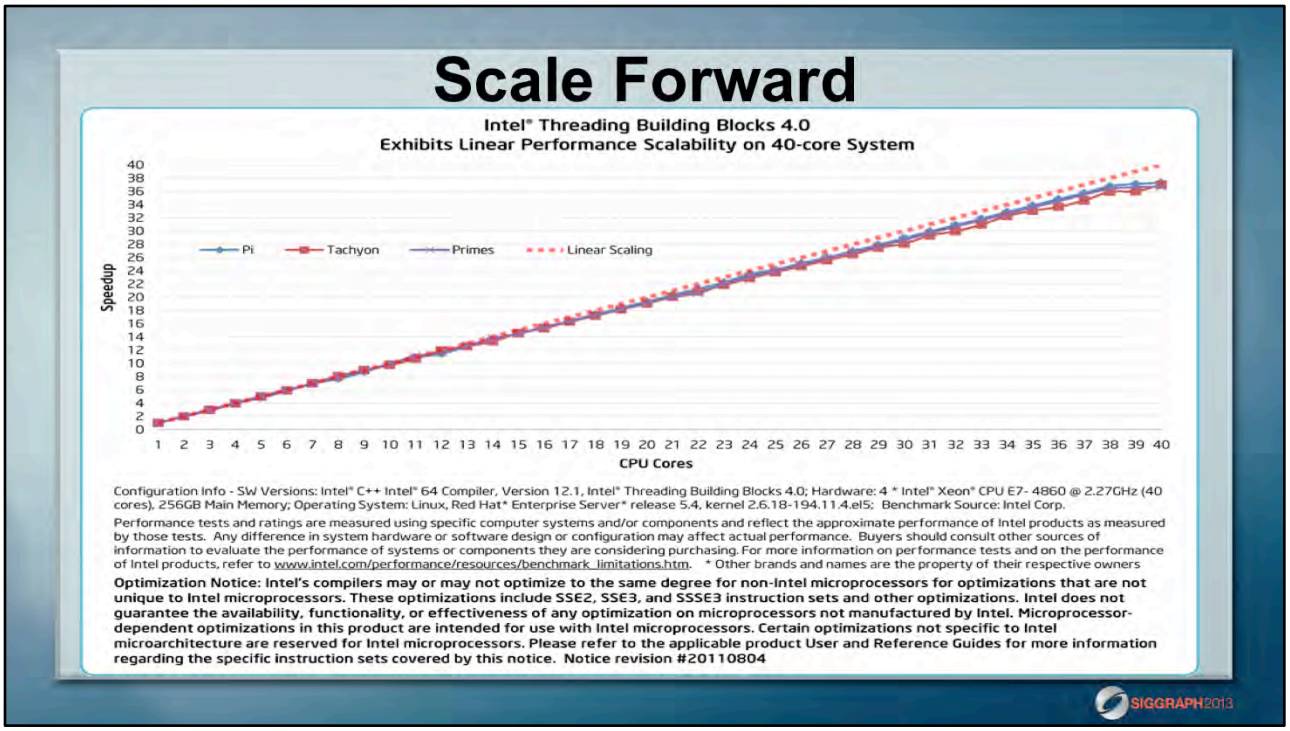


Work stealing is an important concept (invented by the Cilk project at MIT in the mid-1990s)… used my other schedulers… TBB made it very accessible to many people.

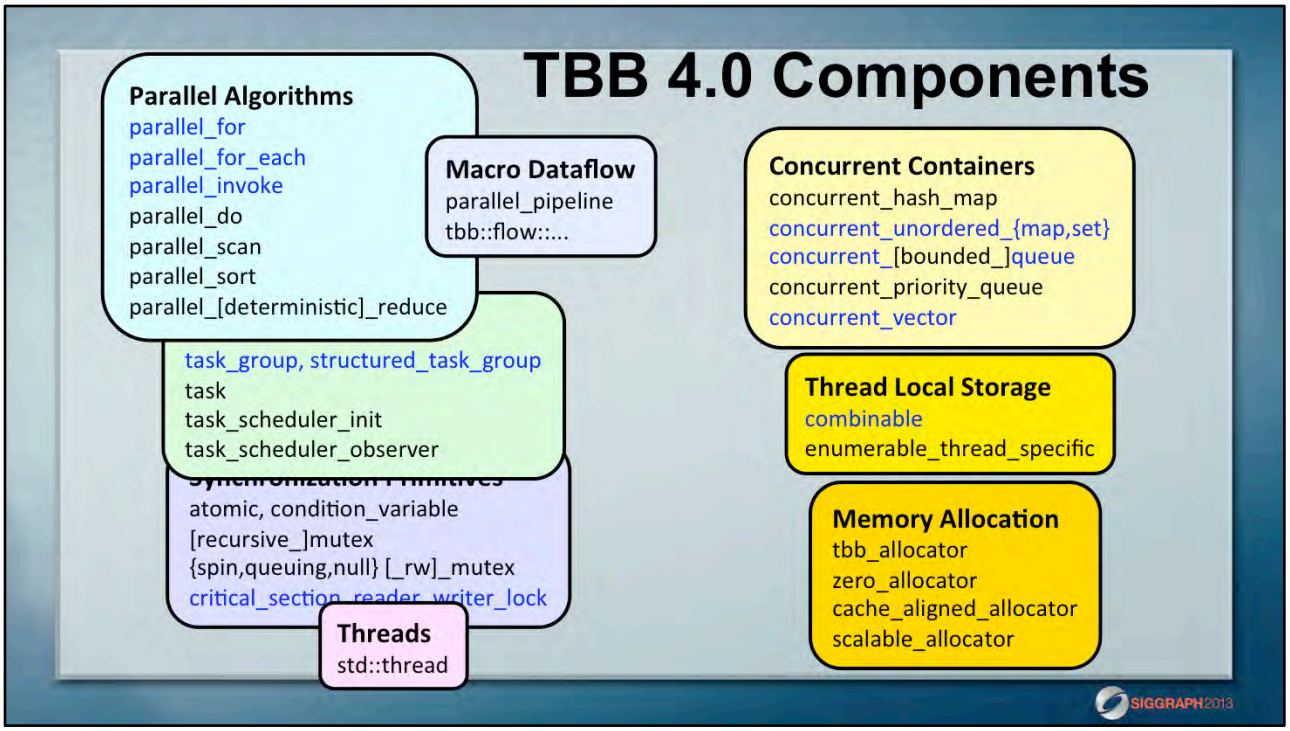
The concept is simple: distribute the queue of work to be done so that there is no global bottleneck involved in the distribution of tasks to the worker threads. With careful design, the cache behavior of this is excellent.



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

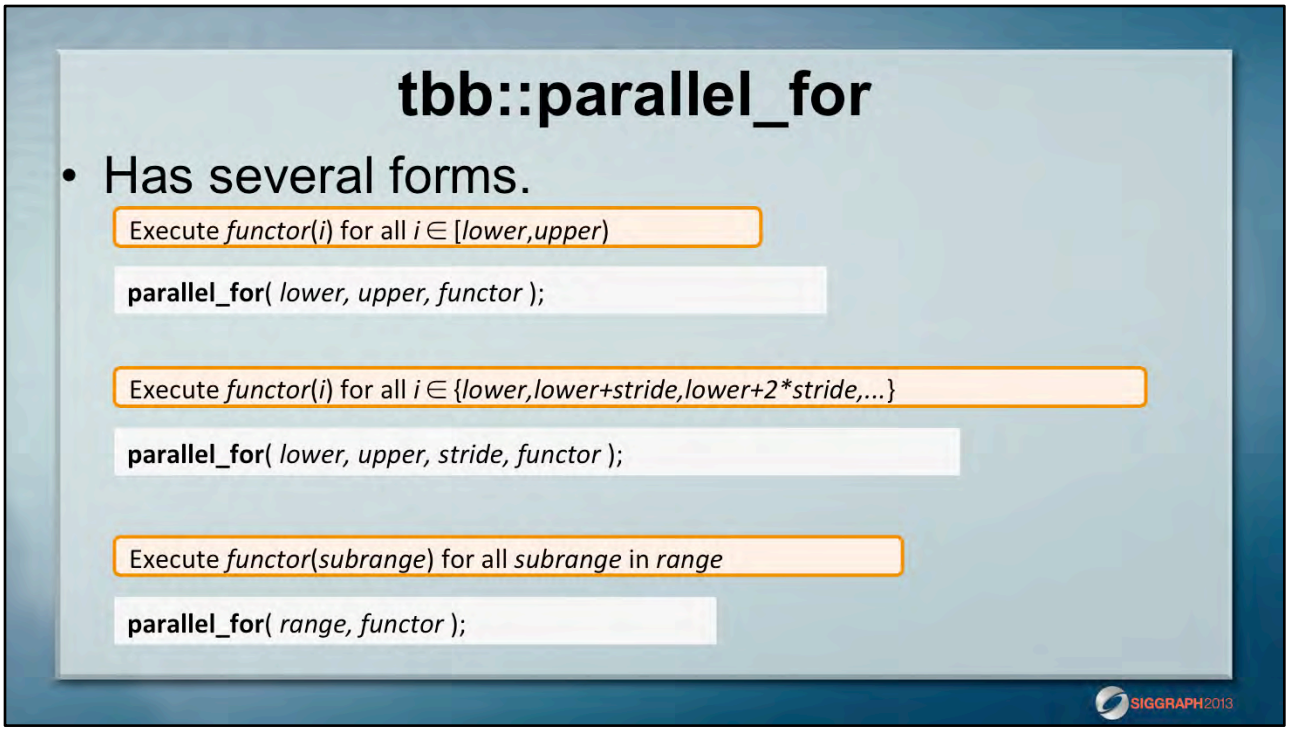


This just proves that the overhead of using the runtime is virtually nothing… it doesn't promise you algorithm can scale… it does indicate that if your algorithm can scale, that TBB can stay out of the way and let it scale!



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.



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) );
```

putting code in a class is annoying

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];
      });
}
```

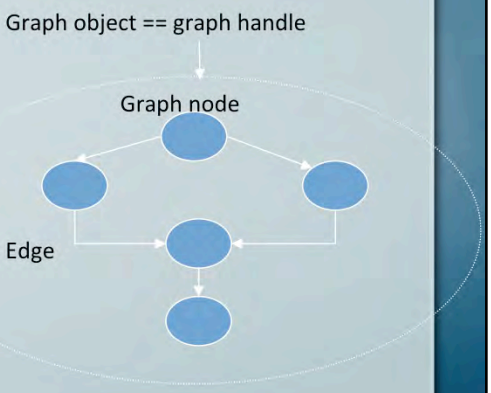
much easier to teach/read with lambdas

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

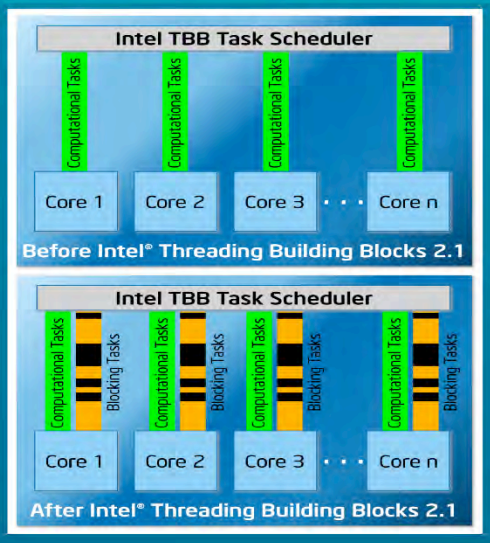


SIGGRAPH2013

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

- Make # of software threads higher than # of hardware threads
- Blocking tasks support
- Task scheduler now permits blocking tasks
- Needed for GUI, AI, I/O, and network events
- Doesn't interfere with computational work



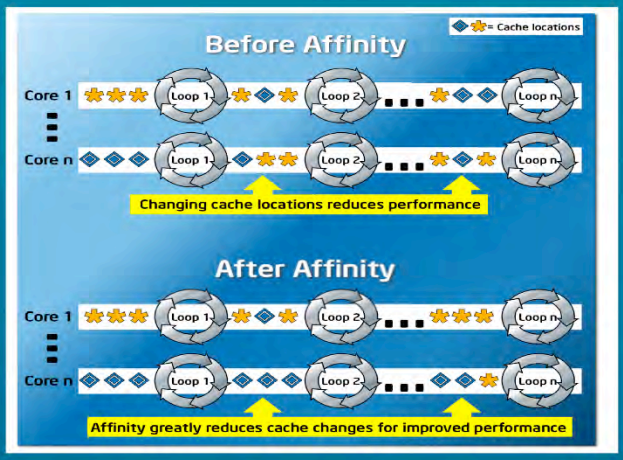
Motivating the need here is the type of programming issue: blocking tasks – and how that challenges "program in tasks instead of threads"

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

affinity partitioner

Portable affinity mechanism gets more performance from chained parallel operations.

More efficient cache use.



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

Course Notes ONLY material

Mutex Summary

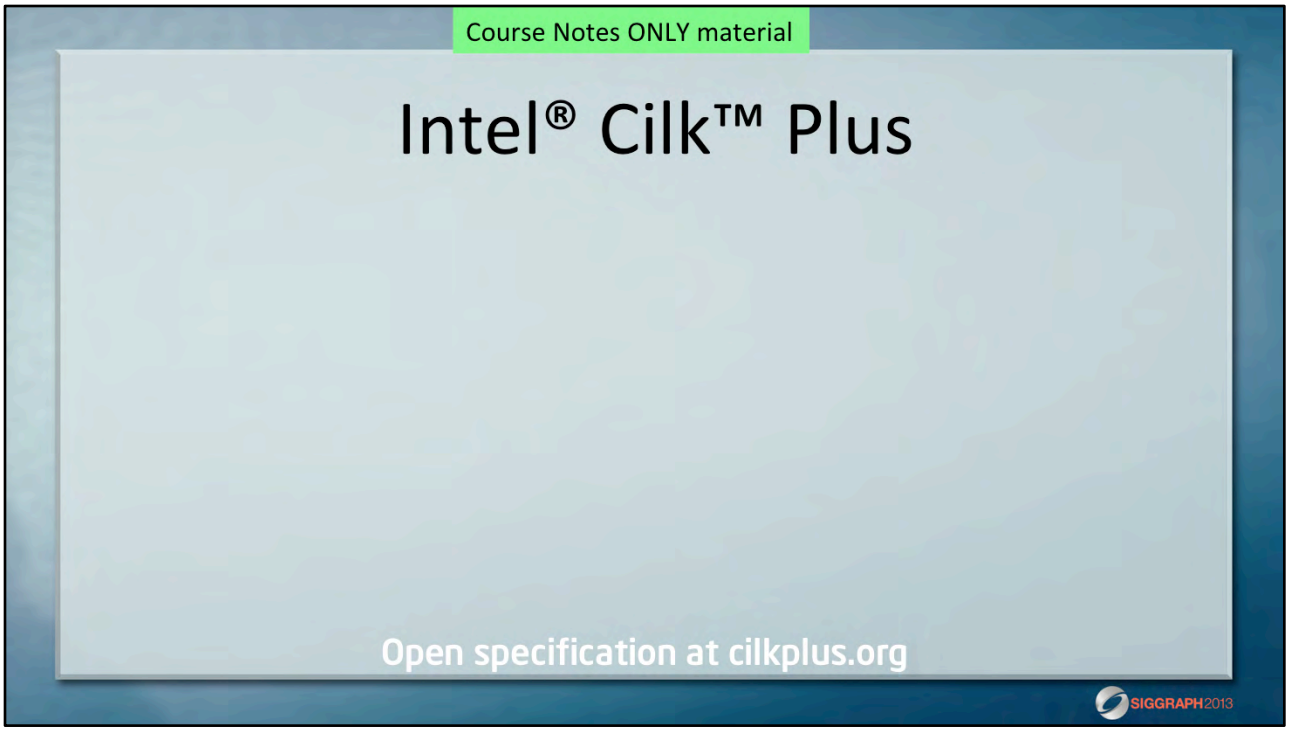
- "Scalable": does no worse than serializing
- Fair: preserves first-come first-serve (guarantees no starvation)
- Reentrant: thread can hold multiple locks on the same mutex
- Wait method: how threads wait for the lock

	"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

33

SIGGRAPH2013

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



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]);
}
```

Open specification at cilkplus.org

SIGGRAPH2013

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;
    }
}
```

Open specification at cilkplus.org

SIGGRAPH2013

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

SIGGRAPH2013

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: WikimediaCommons (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

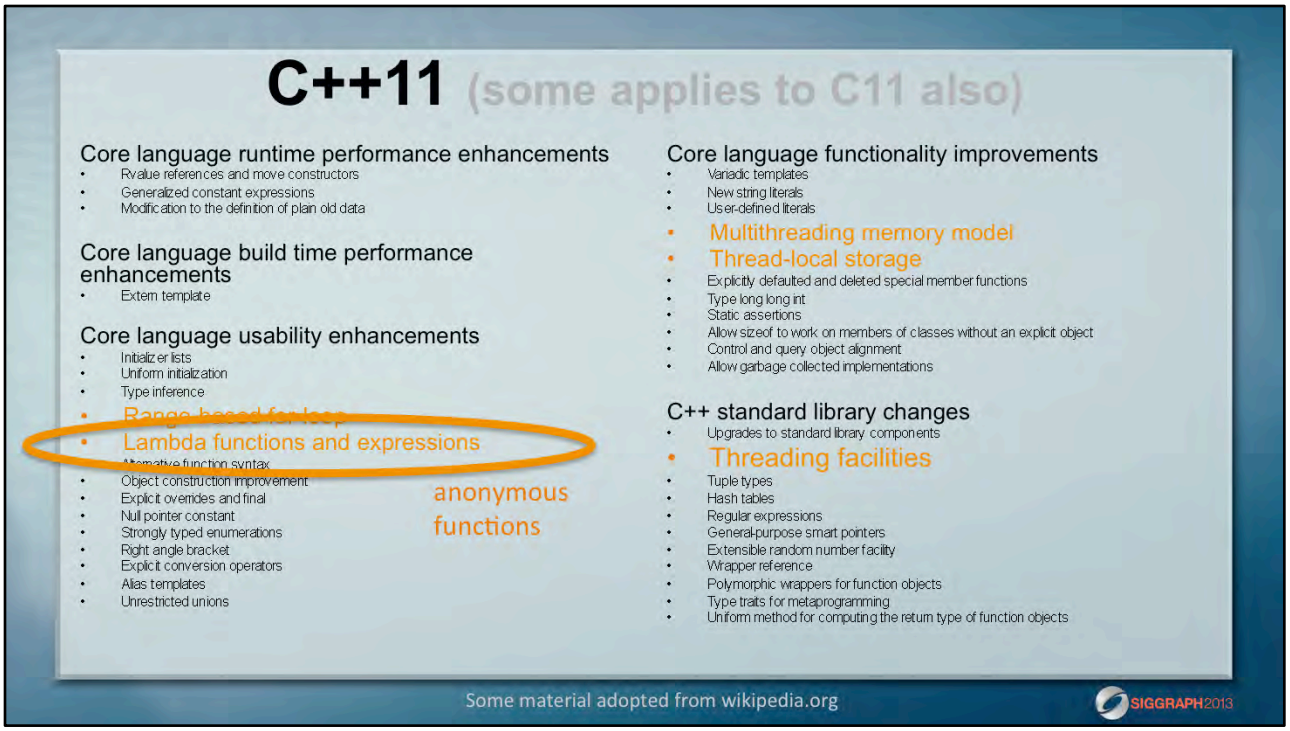
Core language build time performance enhancements
- Extern template

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

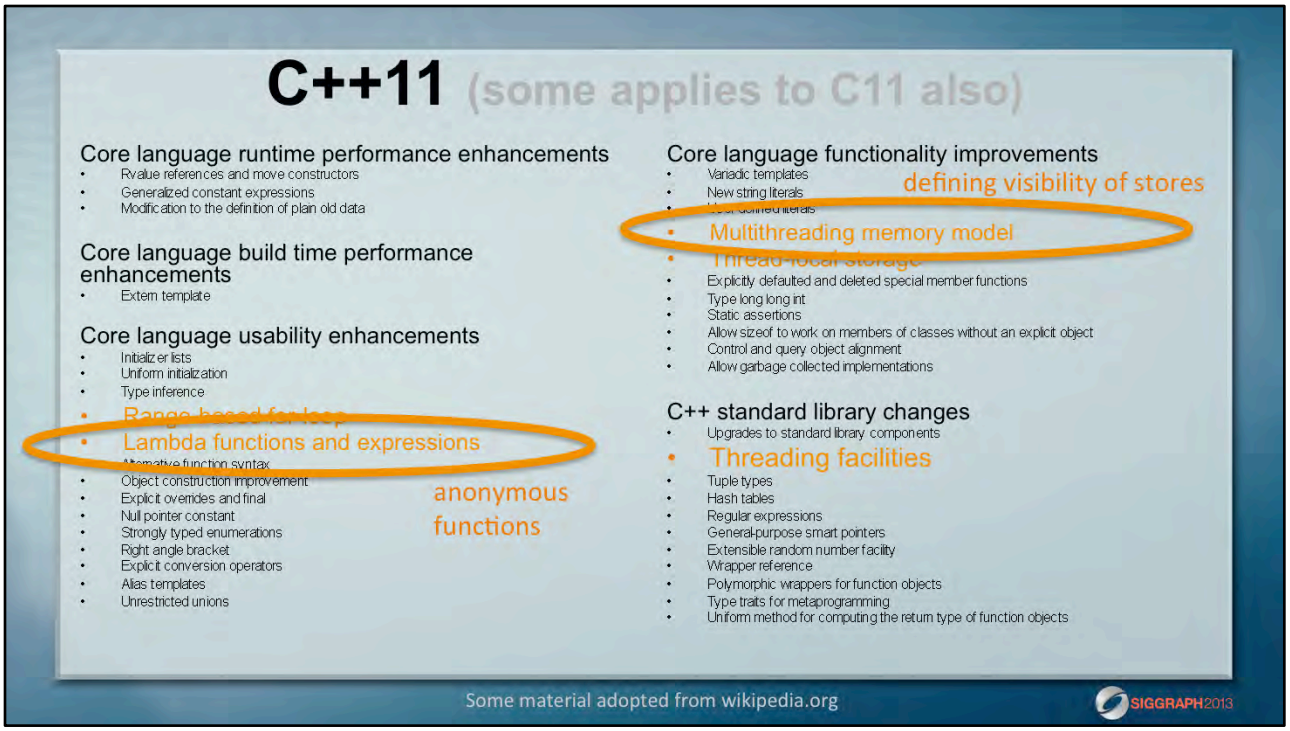
Core language functionality improvements
- Variadic templates
- New string literals
- User-defined literals
- Multithreading memory model
- Thread-local storage
- Explicitly defaulted and deleted special member functions
- Type long long int
- Static assertions
- Allow sizeof to work on members of classes 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

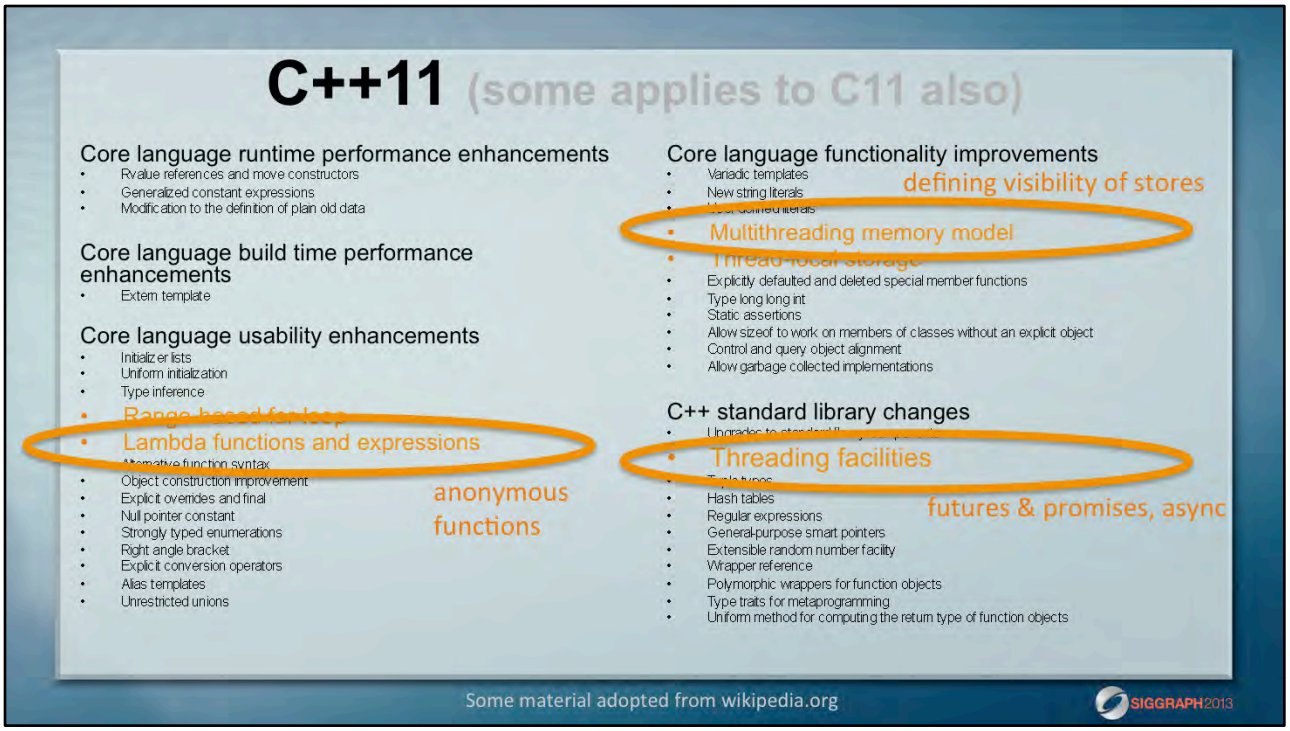


Nothing is more "exciting" than lambda functions. I'm a big fan. They really help make some constructs in parallel programming easier to read – including using TBB.



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();
```

SIGGRAPH2013

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.

SIGGRAPH2013

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:

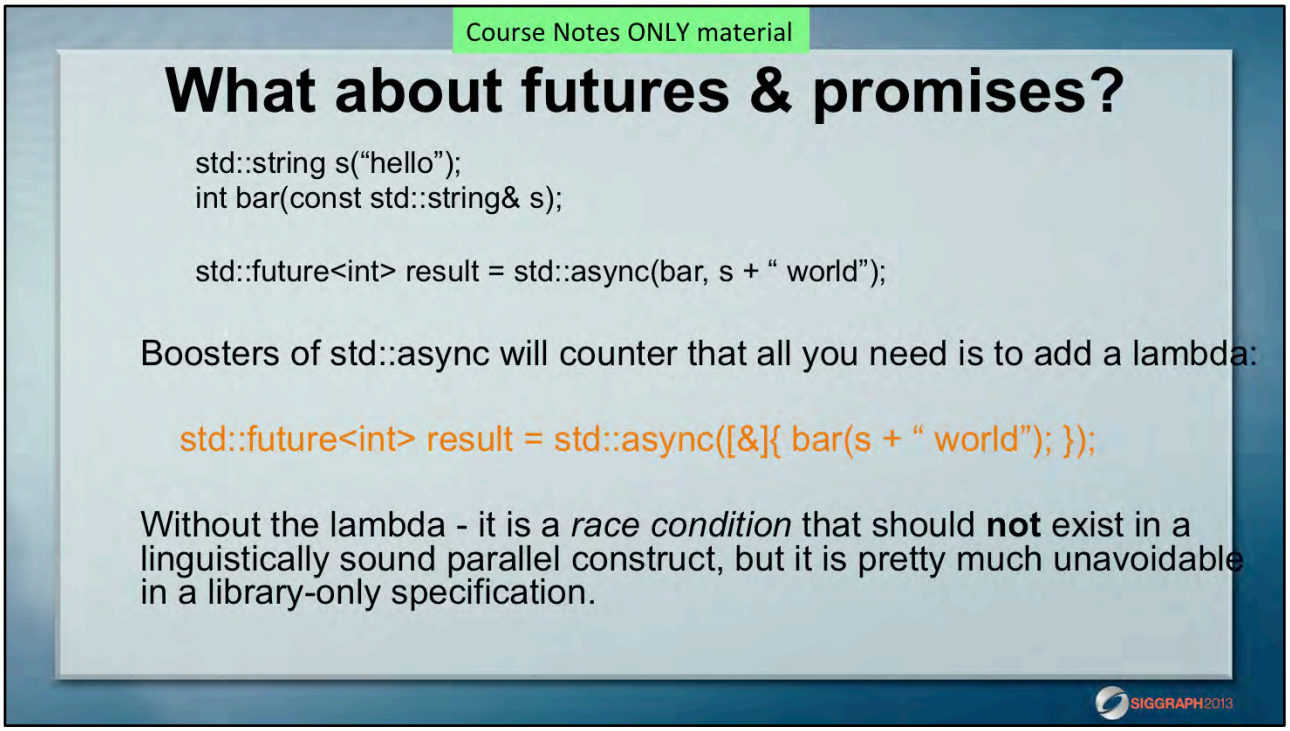
```
std::string s("hello");
int bar(const std::string& s);

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

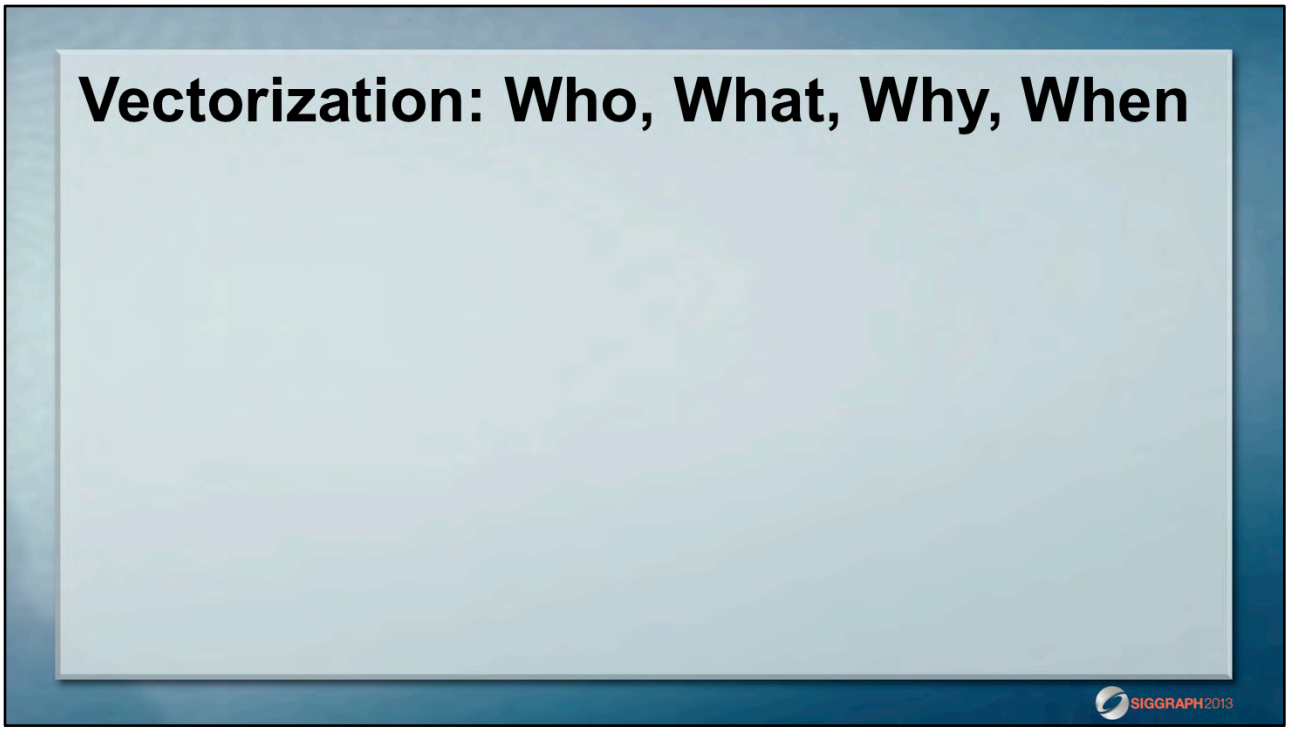
The above statement is intended to pass "hello world" to bar and run it asynchronously.

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.



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.



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

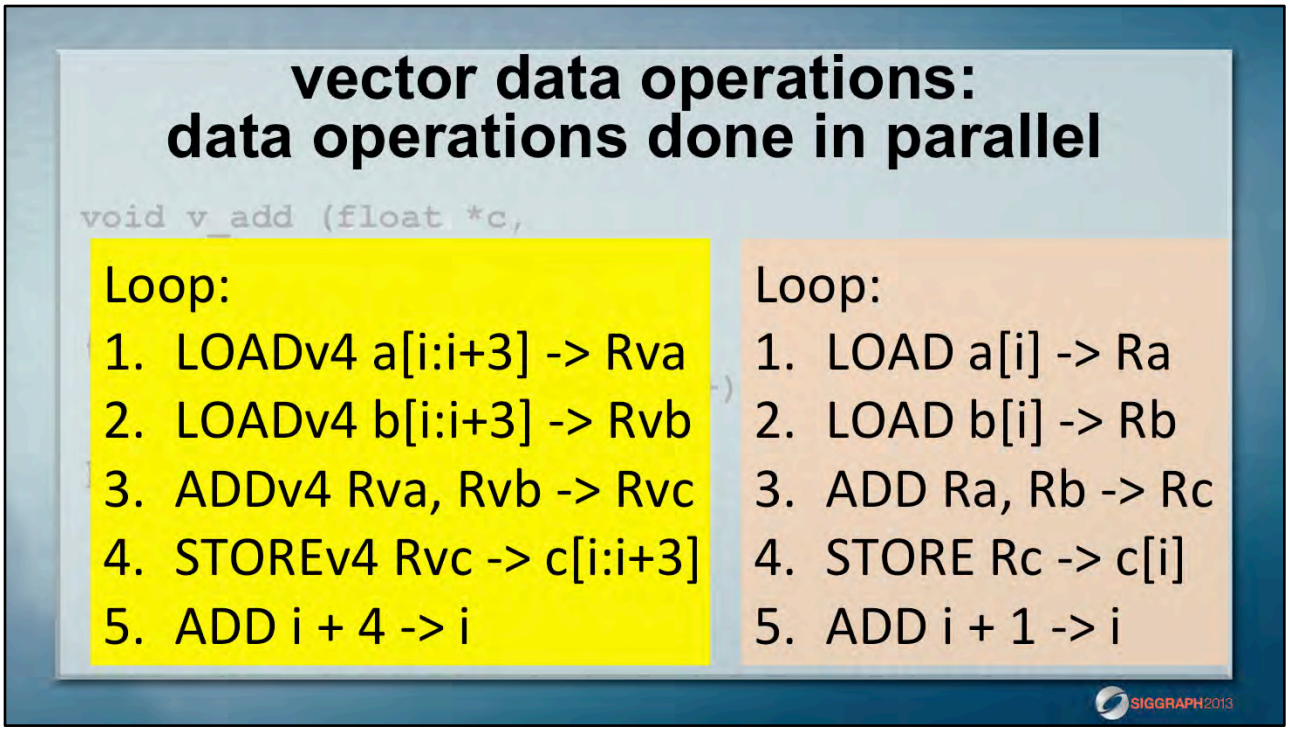
vector data operations: data operations done in parallel

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

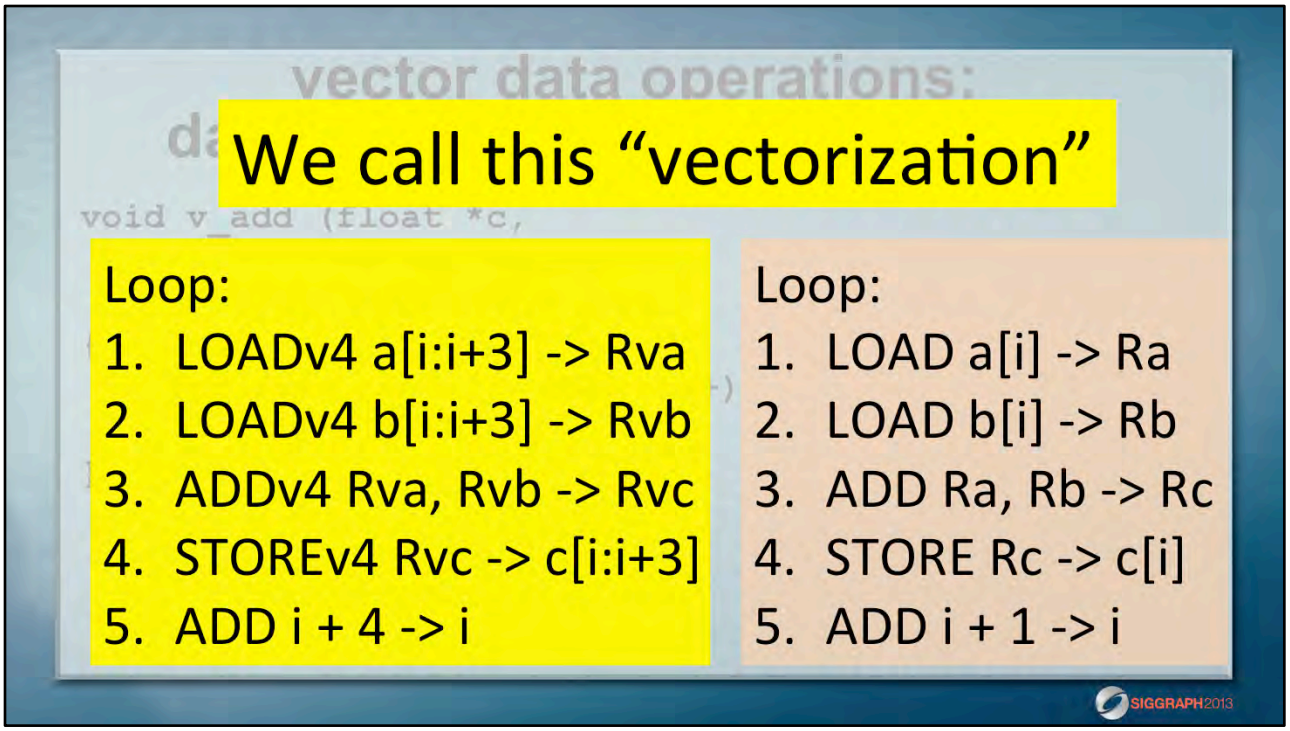
Loop:

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



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).

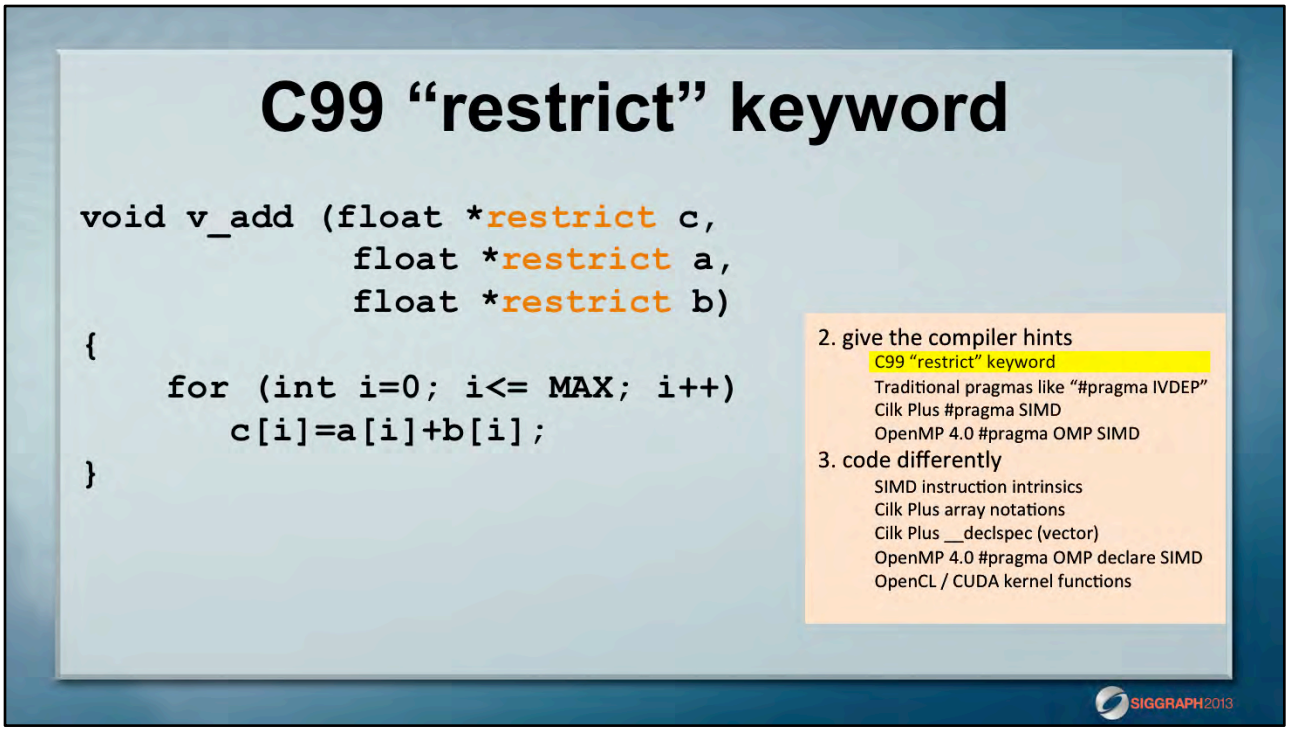
- Arrays not really in the language
- Pointers are, evil pointers!

We'll see why later.

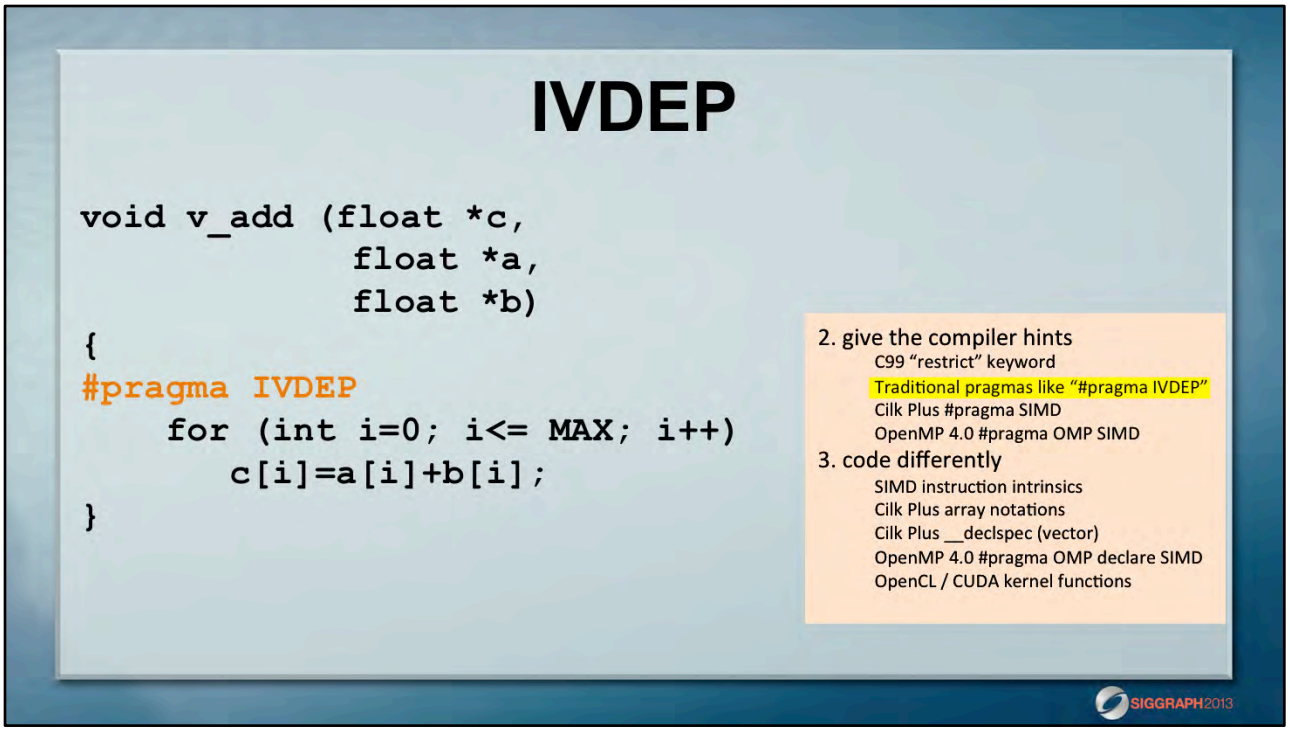
vectorization solutions

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

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



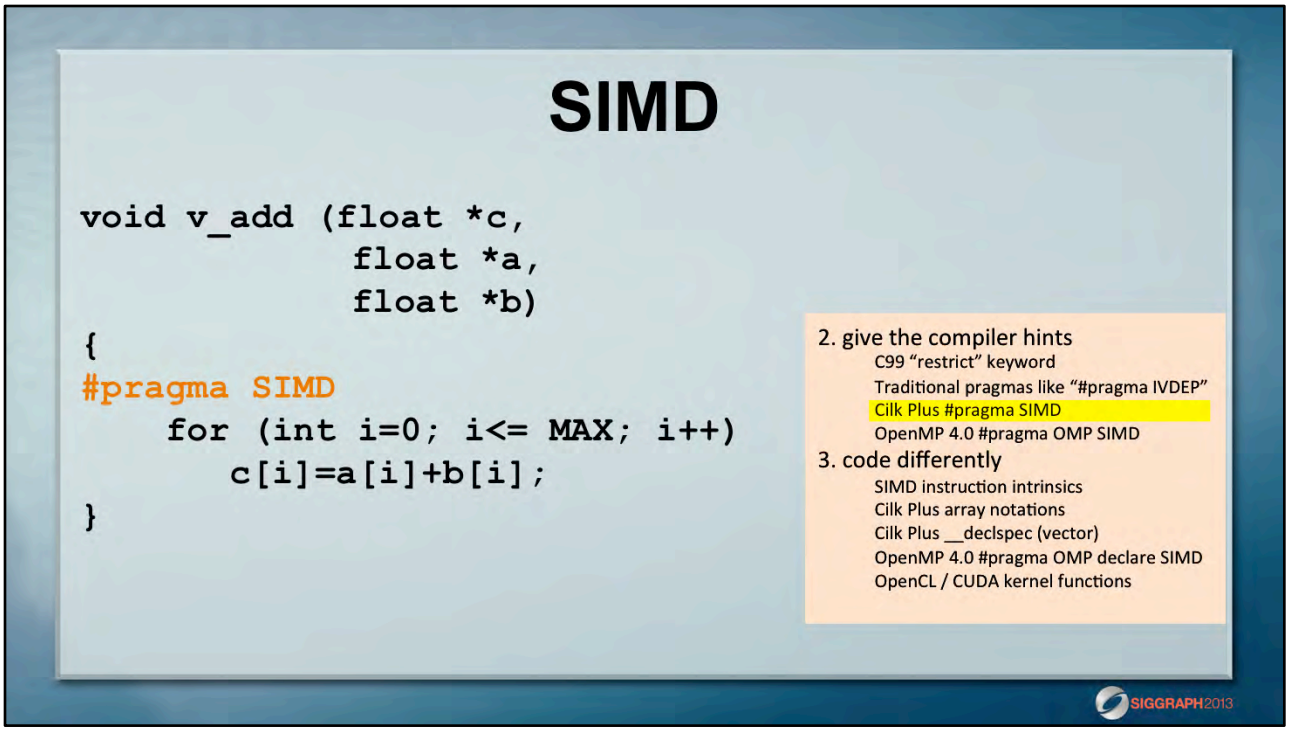
The C99 standard (not part of any C++ standard) introduces the restrict keyword for use in pointer declarations.
It is a declaration of intent given by the programmer to the compiler. It declares the intent that during lifetime of the pointer, only it or a value directly derived from it (such as pointer + 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.



#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.

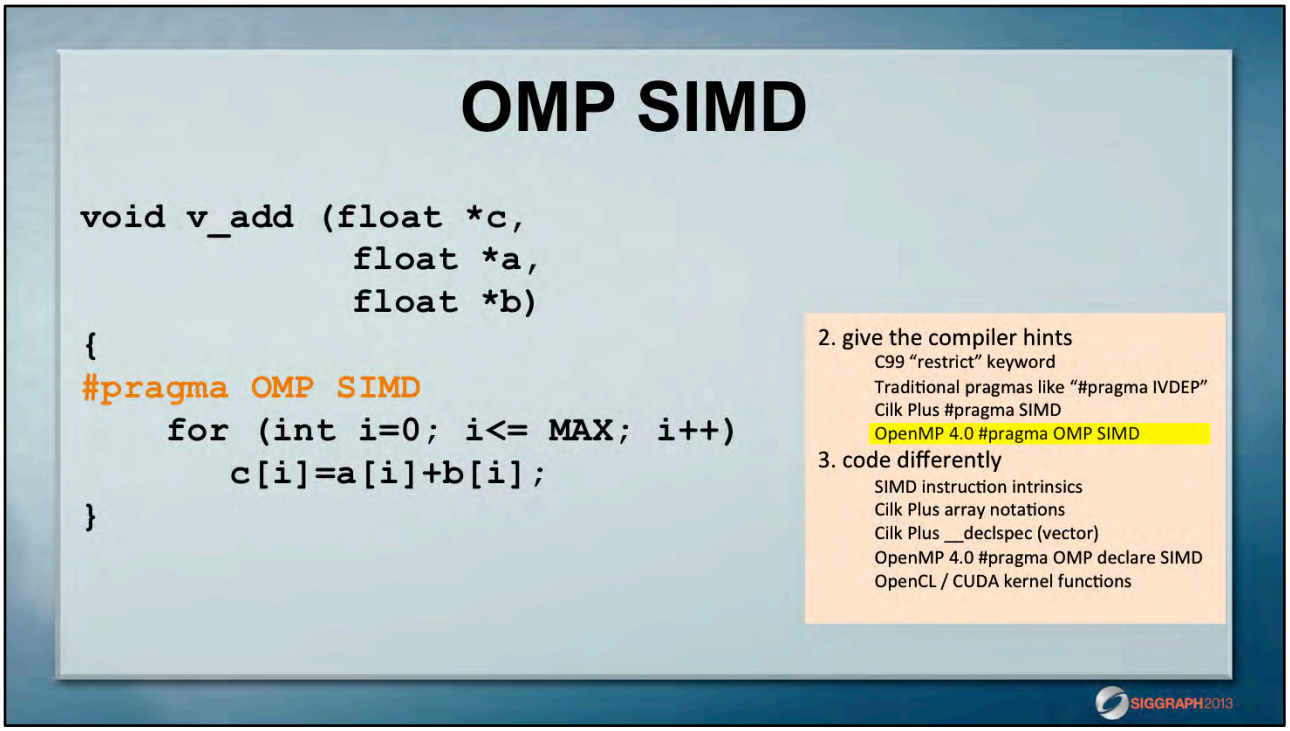
The net effect is that the compiler will find the loop acceptable to vectorize.



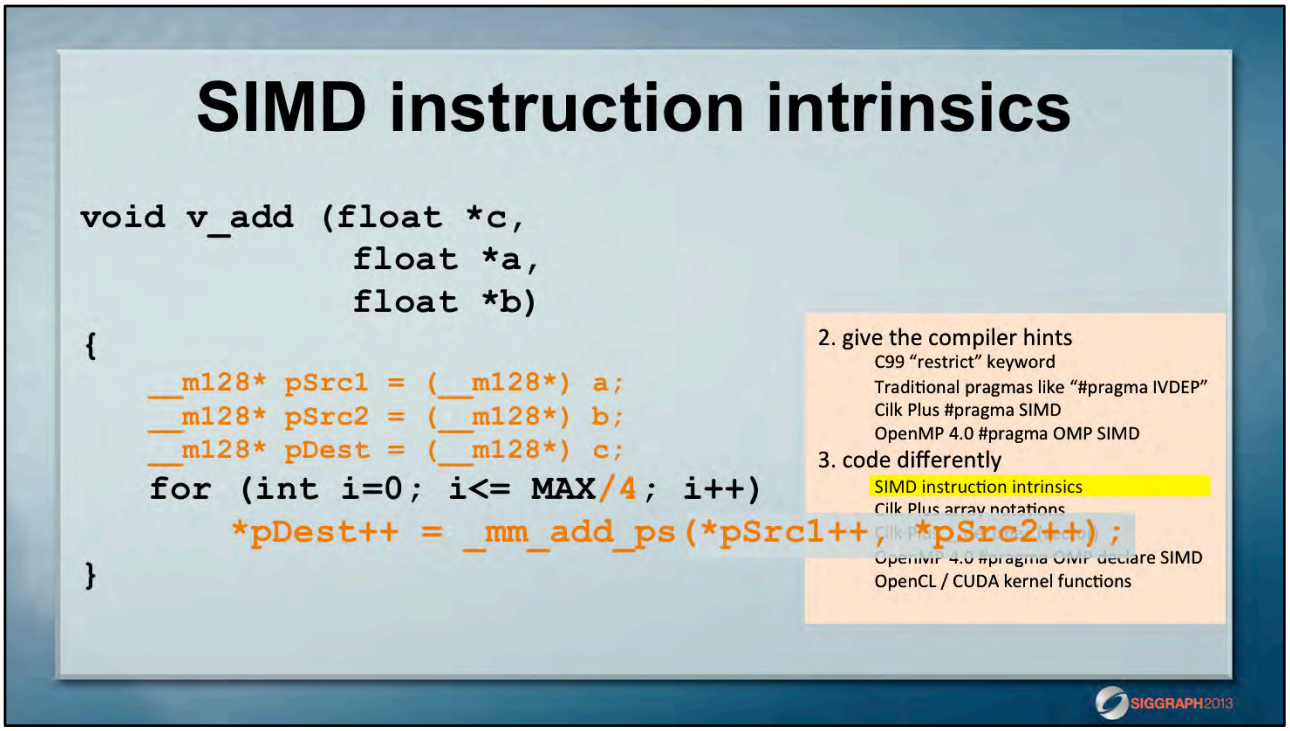
#pragma SIMD is a non-standard compiler hint, part of Intel's Cilk Plus project which is gaining in popularity (multiple compiler implementations) – http://cilkplus.org

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.



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 was released in draft form in November 2012, with a revised (and completed) draft in early 2013 (available from http://openmp.org)
http://openmp.org

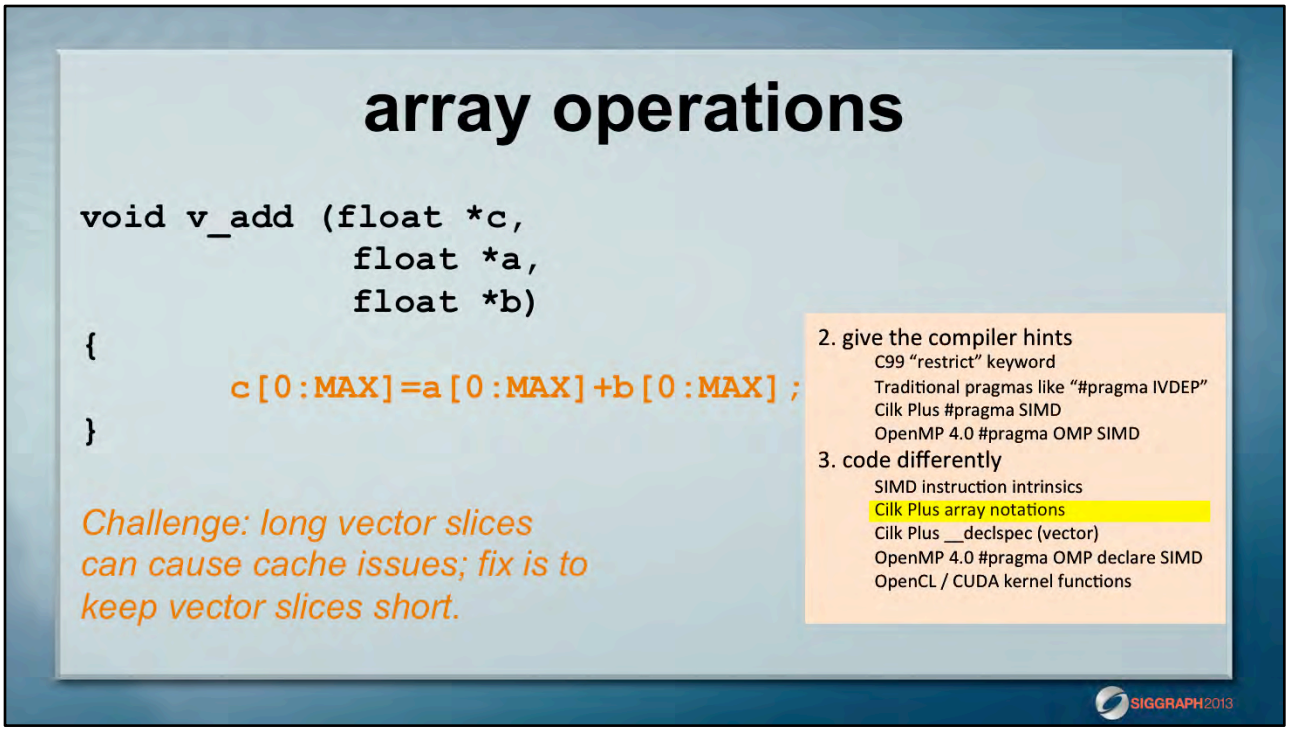


SIMD instruction intrinsics offer "pseudo-functions" for each major SIMD instruction on a one-to-one basis.

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.



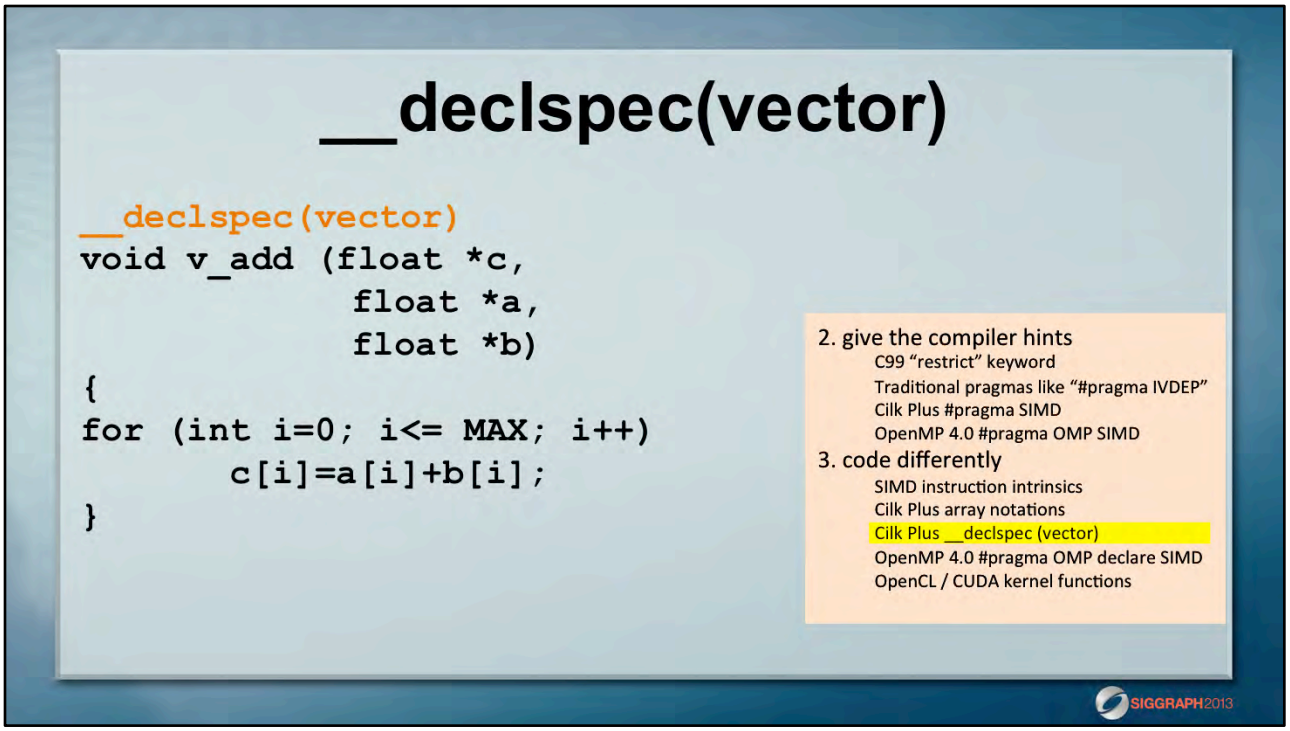
Similar to Fortran90 arrays

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

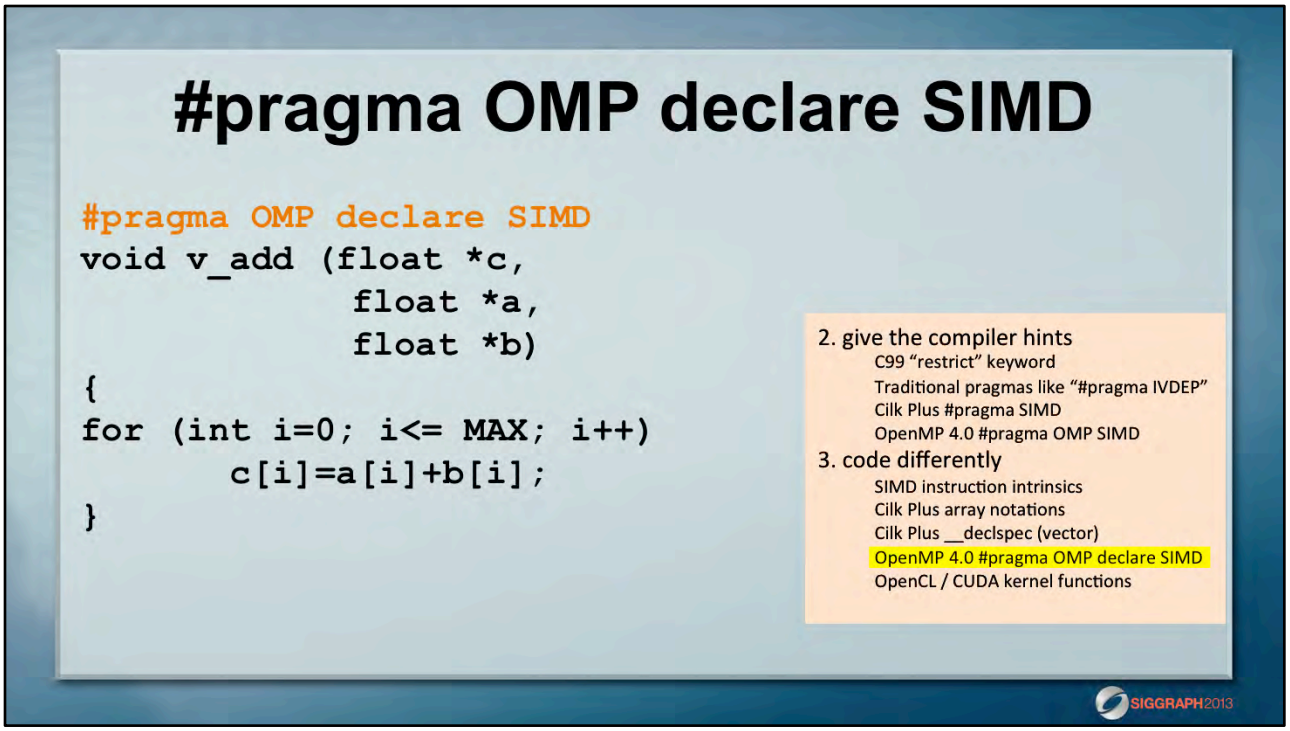
Intel Cilk Plus includes extensions to C and C++ that allows for parallel operations on arrays. The intent is to allow users to express high-level vector parallel array operations. This helps the compiler to effectively vectorize the code. Array notation can be used for both static and dynamic arrays. 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.

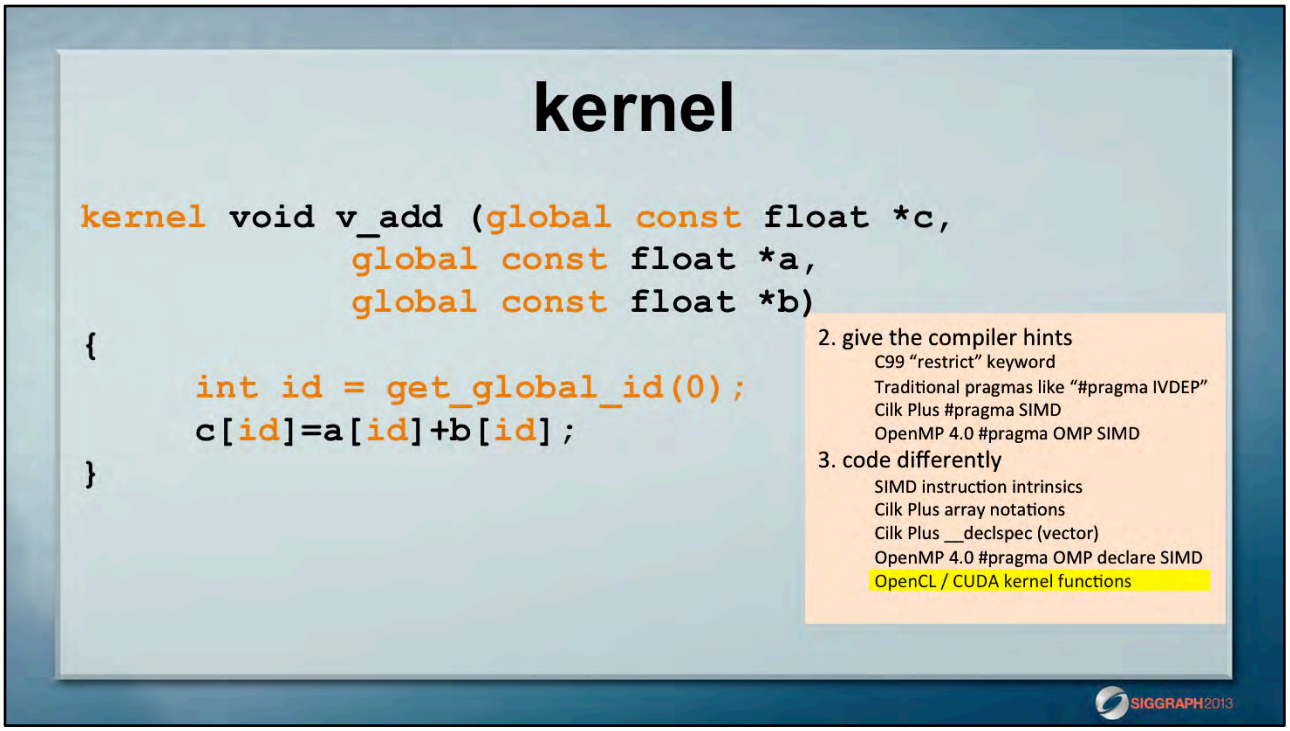
So using *array-expression[:]* denotes an array section that is the entire array of known length and stride 1.
Array notation allows for multi-dimensional array sections as multiple array section operators can be used for a single array.
array-expression[:][:] denotes a two dimensional array
Two new 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.



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



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



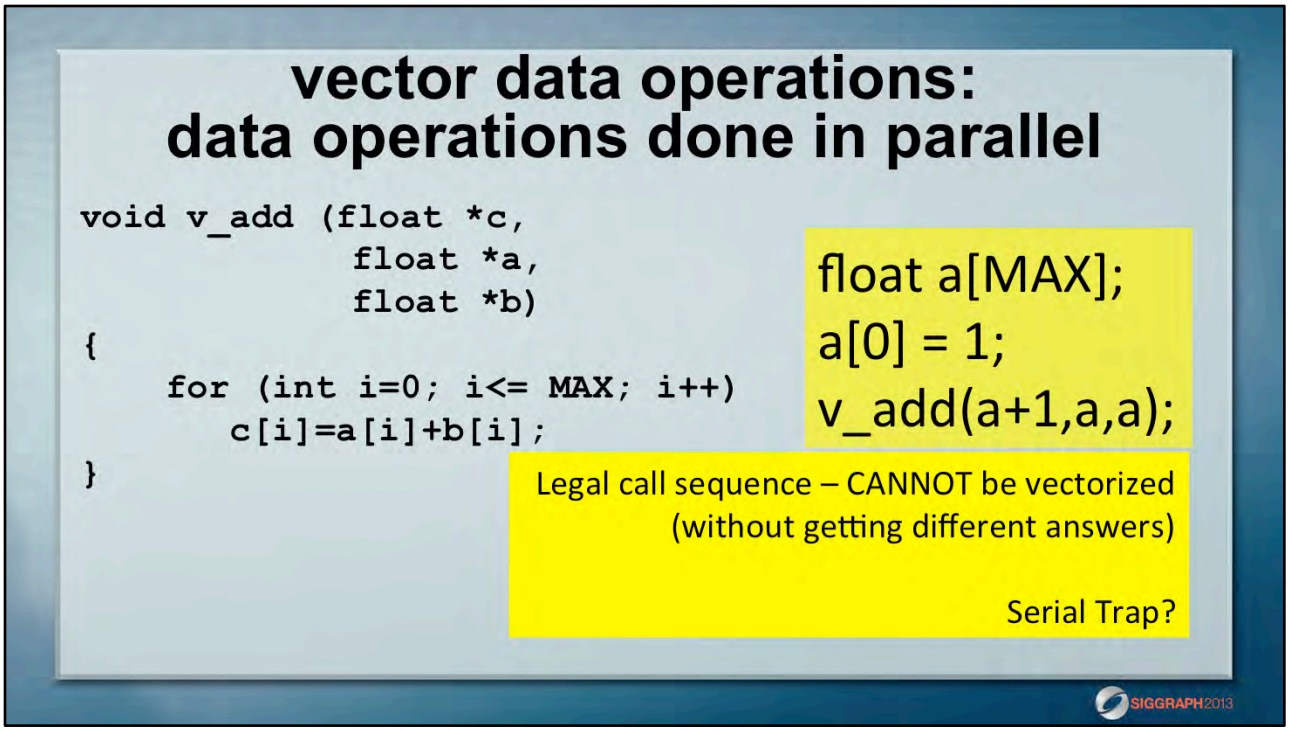
Elemental (kernel) function – a scalar function that get used in a vector form with OpenCL construct

vector data operations: data operations done in parallel

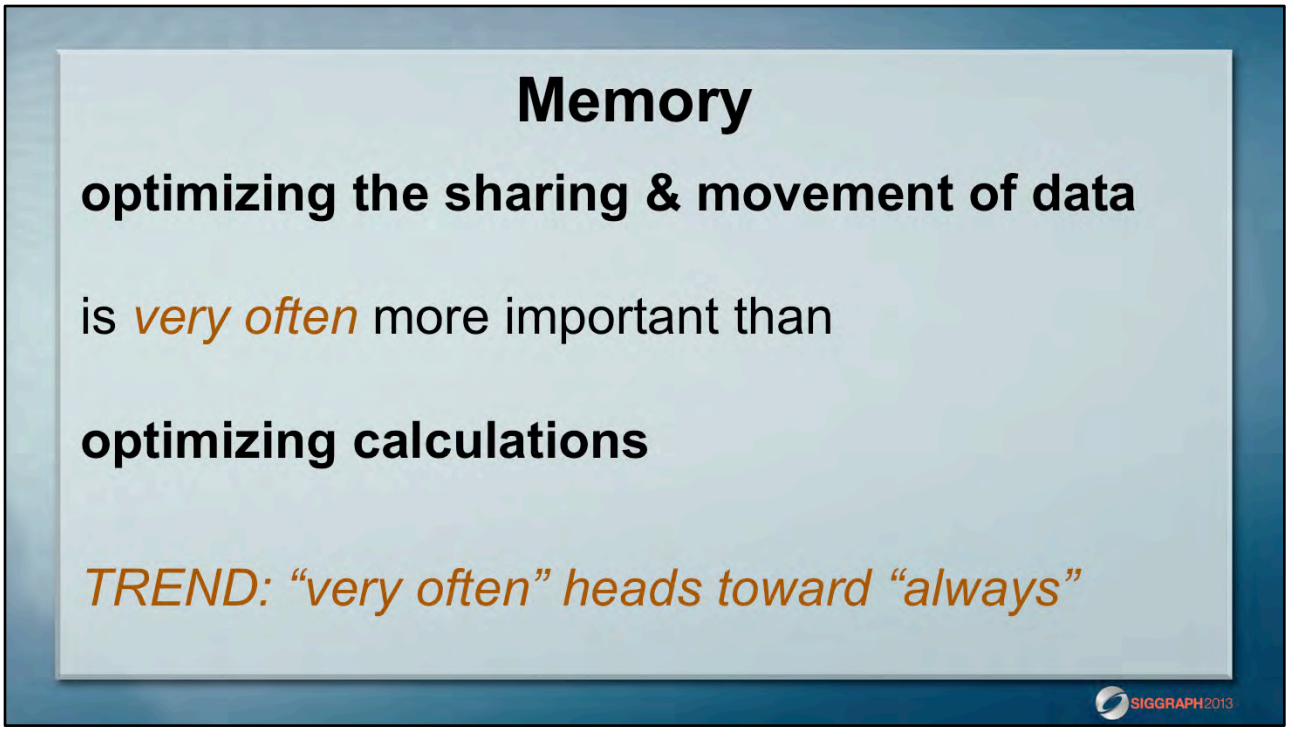
```
void v_add (float *c,
            float *a,
            float *b)
{
    for (int i=0; i<= MAX; i++)
       c[i]=a[i]+b[i];
}
```

```
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)

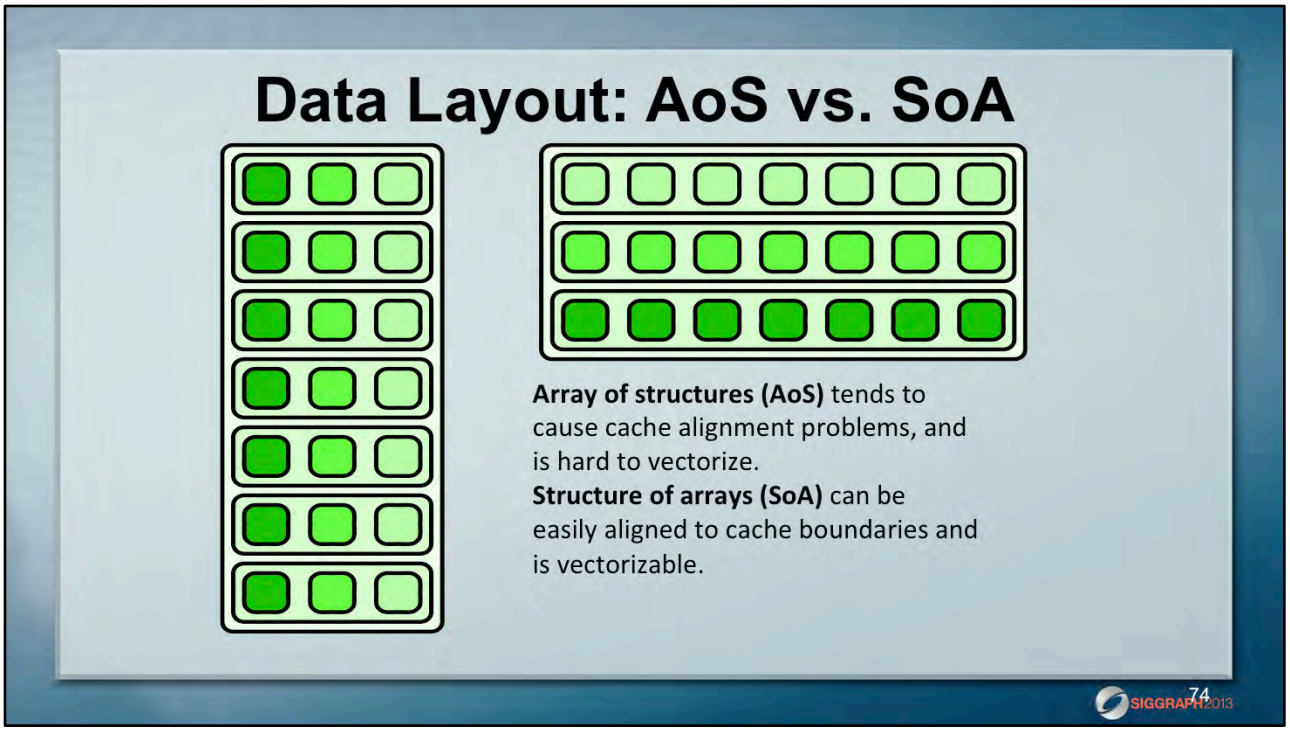


Call sequence loads the array with powers of 2.
This legal call sequence – CANNOT be vectorized (without getting different answers)



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!



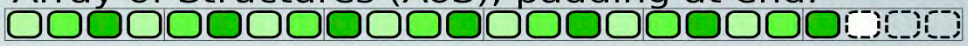
SoA is great for independently used arrays and when vectorization is desired

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

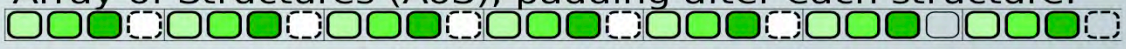
When in doubt – SoA is more likely to be the better

Data Layout: Alignment

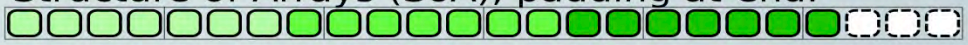
Array of Structures (AoS), padding at end.



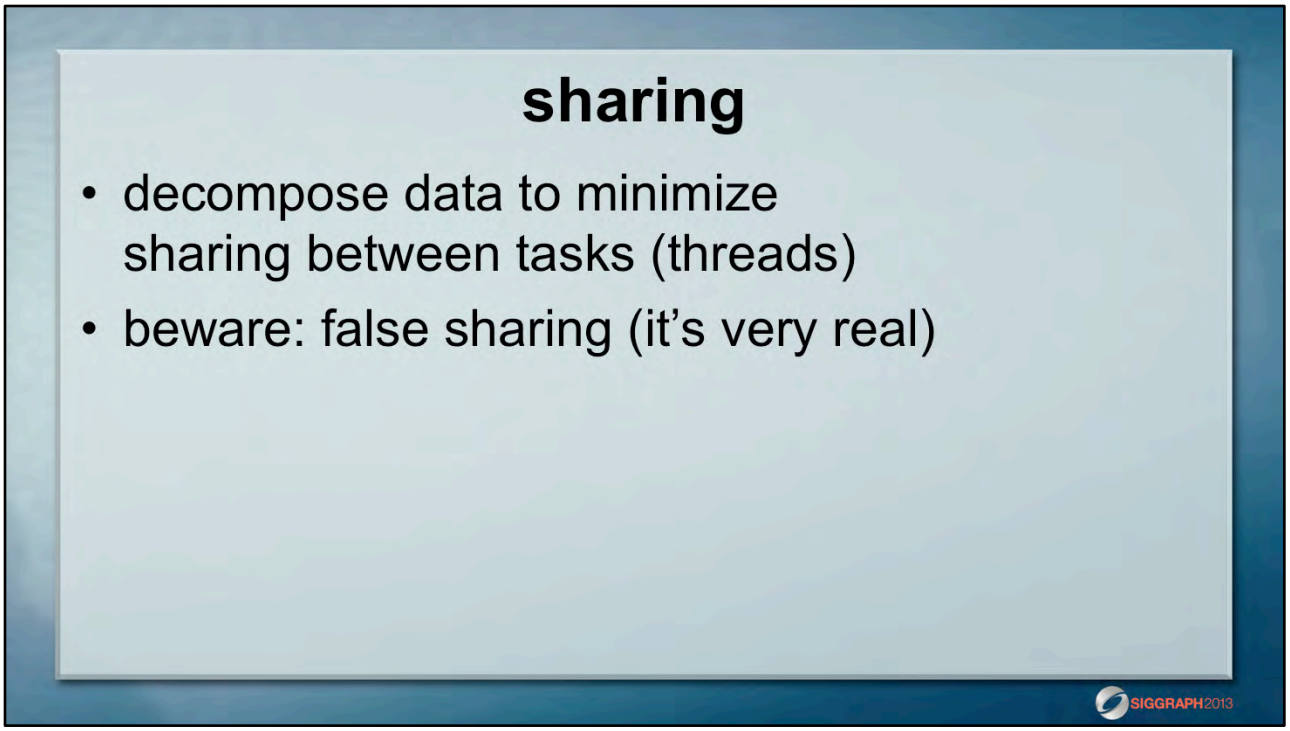
Array of Structures (AoS), padding after each structure.



Structure of Arrays (SoA), padding at end.



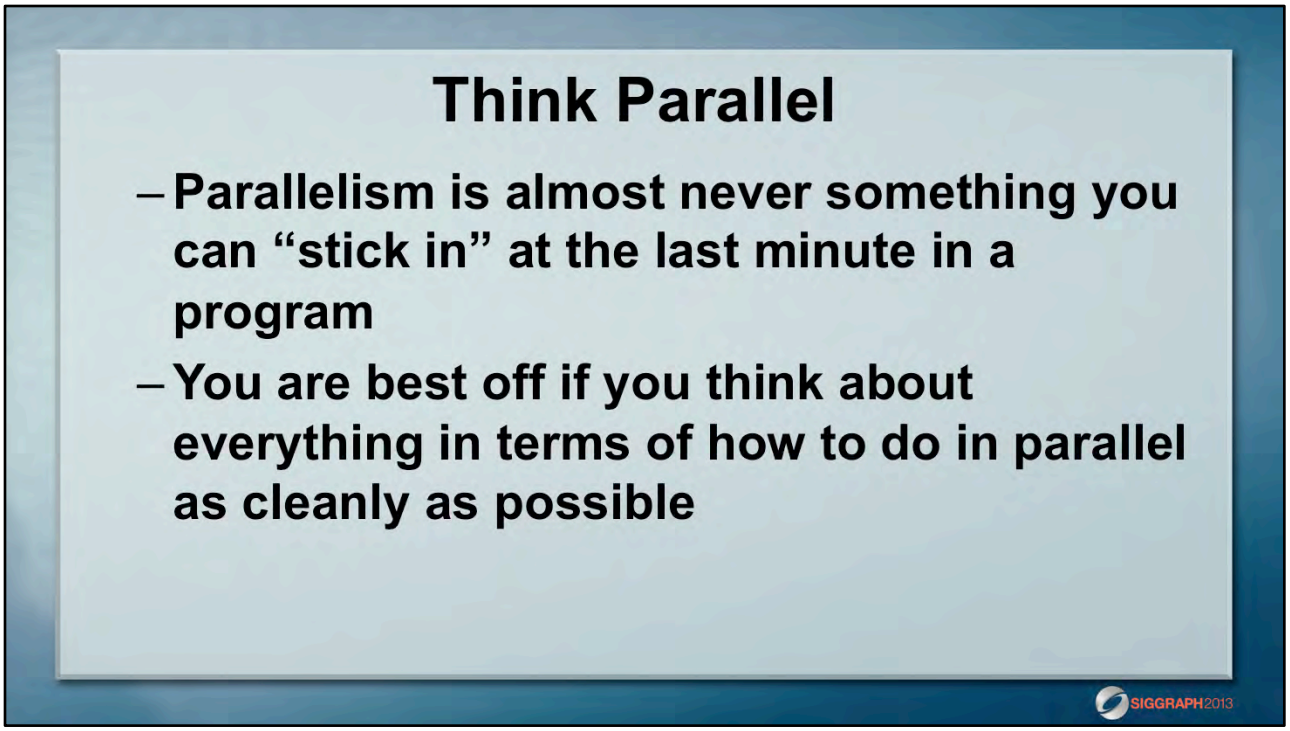
Structure of Arrays (SoA), padding after each component.



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)

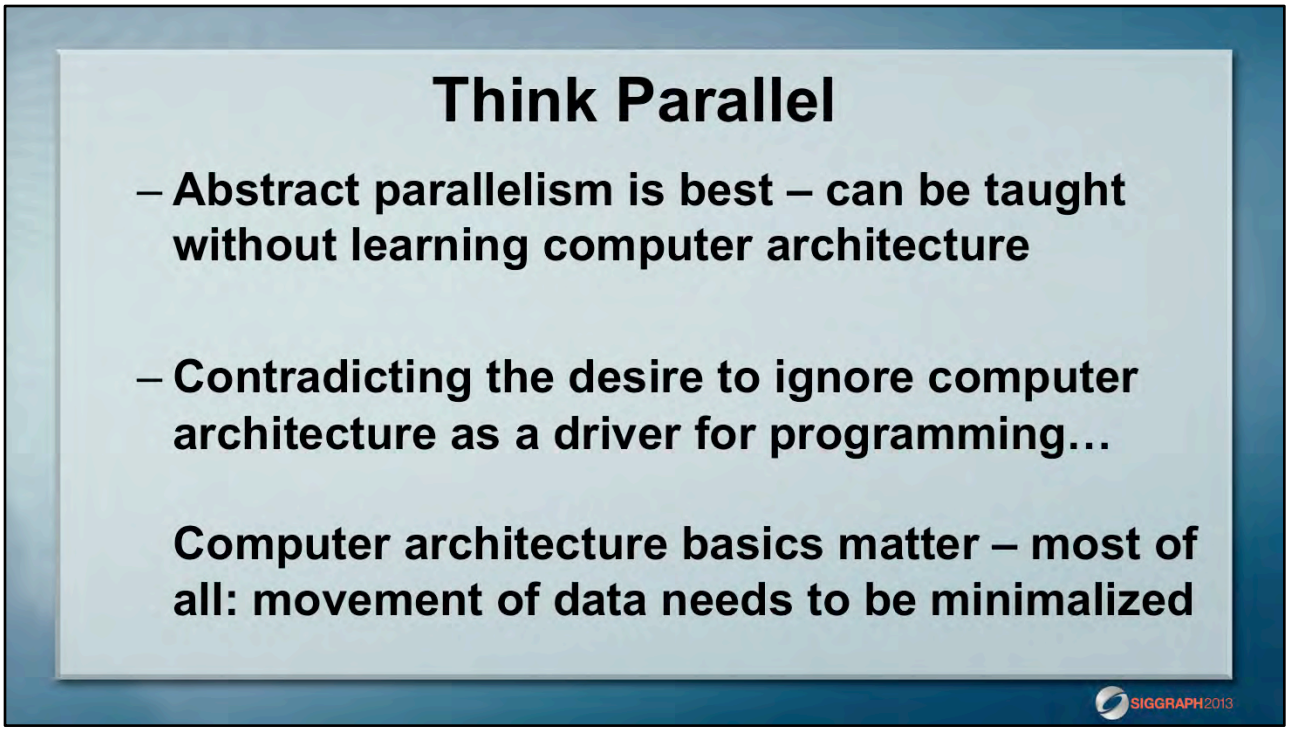


"Thinking parallel"

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

why computer architecture matters

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

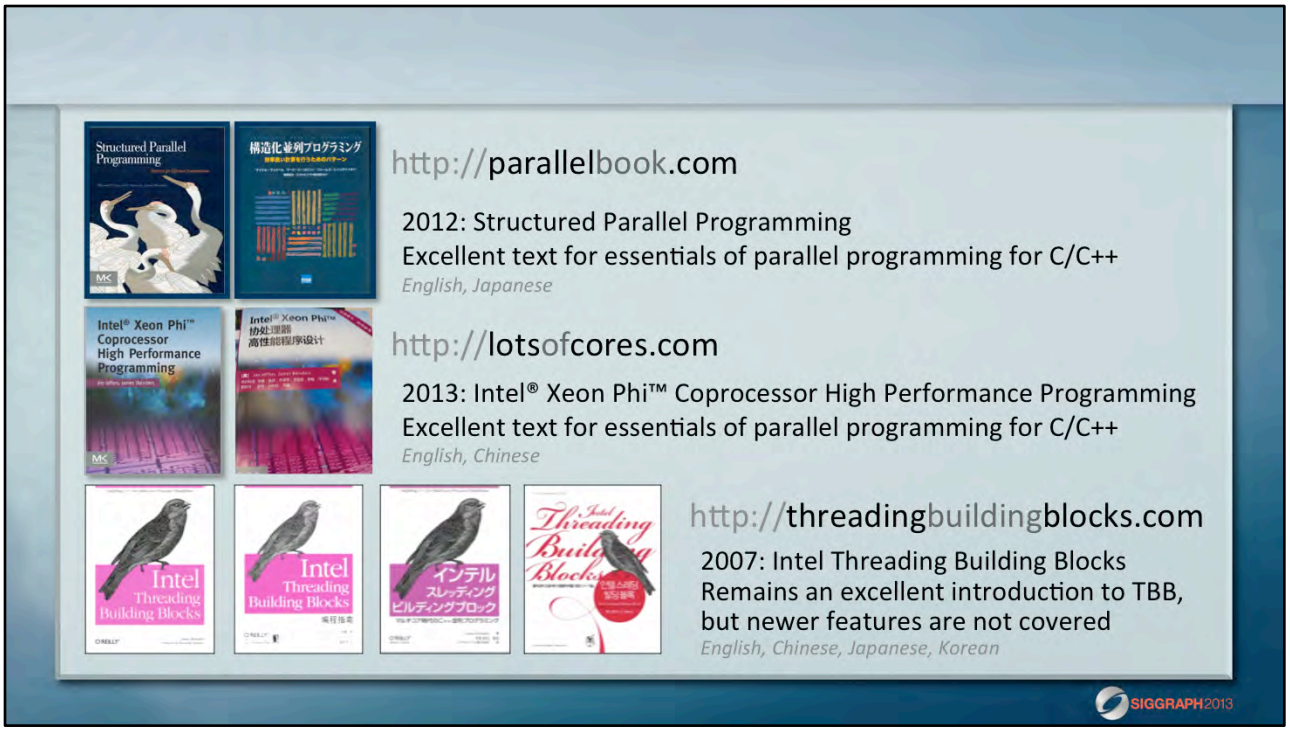


"Thinking parallel"

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

why computer architecture matters

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



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
James is involved in multiple efforts at Intel to bring parallel programming models to the software industry—including models for the Intel® Many Integrated Core (Intel® MIC) architecture. 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 has authored technical books, including *VTune™ Performance Analyzer Esssentials* (Intel Press, 2005), *Intel® Threading Building Blocks* (O'Reilly Media, 2007), *Structured Parallel Programming* (Morgan Kaufmann, 2012), and *Intel® Xeon Phi™ Coprocessor High Performance Programming* (Morgan Kaufmann, 2013).

http://s2013.siggraph.org/attendees/courses/events/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.

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


multithreadingandvfx.org
9:05 am
Multithreading Introduction and Overview James Reinders, Intel
9:50 am
Asynchronous Computation Engine for Animation George ElKoura - Pixar
10:20 am
Break
10:30 am
Parallel Evaluation of Character Rigs Using TBB Martin Watt - DreamWorks Animation
11 am
GPU Rigid Body Simulation Using OpenCL Erwin Coumans - AMD
11:30 am
Parallelism in Tools Development for Simulation and Rendering Ron Henderson - DreamWorks Animation
12:00pm
Parallelism in Houdini Jeff Lait - SideFX

Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference www.intel.com/software/products.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804