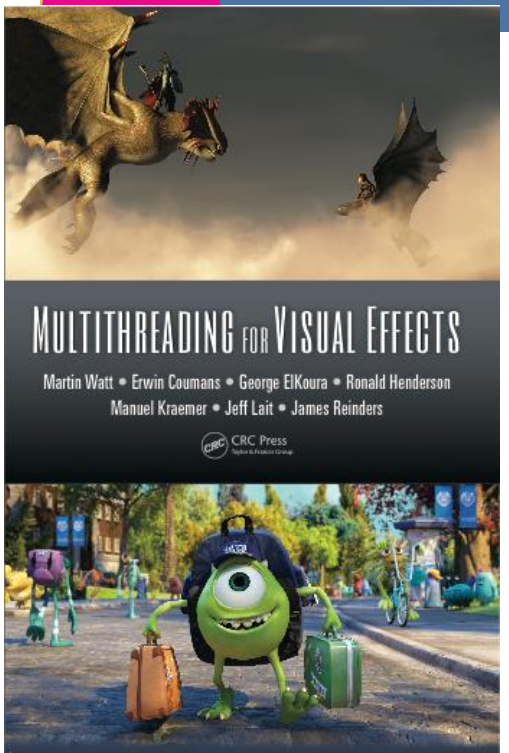
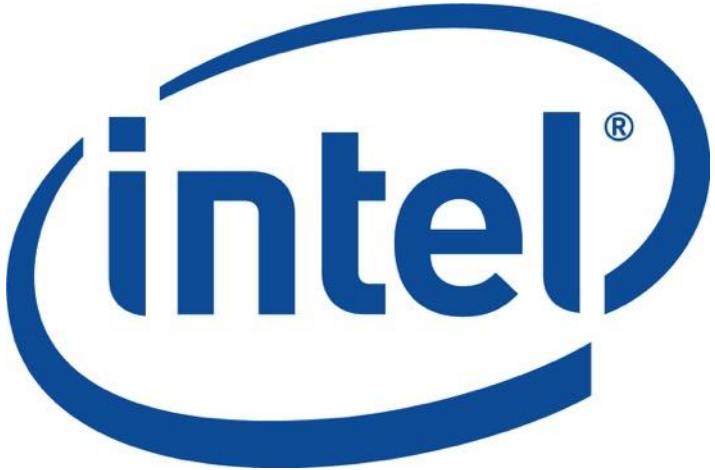


Multithreading for Visual Effects

9:00am-12:15pm

August 12, 2015


SIGGRAPH2015
Xroads of Discovery



SIGGRAPH2015
Xroads of Discovery

The 42nd International Conference and Exhibition
on Computer Graphics and Interactive Techniques

Multithreading for Visual Effects

Multithreading Introduction and Overview

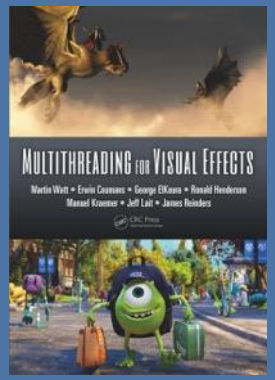
James Reinders
Intel

Agenda

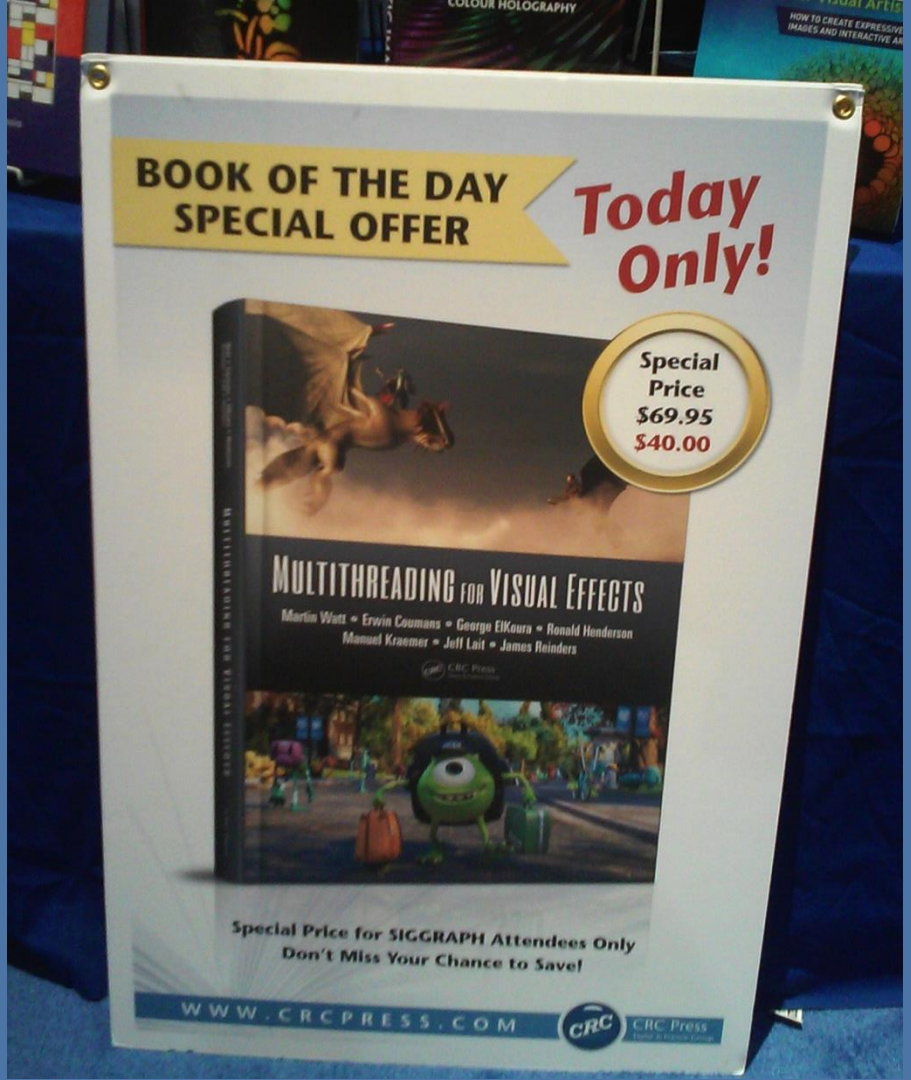
9:00am	Start	
9:00am	Introduction	James Reinders, Intel
9:05am	Multithreading Introduction and Overview	James Reinders, Intel
9:45am	Parallelism in Houdini - practical lessons learned	Jeff Lait, Side Effects Software
10:30am	Break	
10:45am	GPU Rigid Body Simulation Using OpenCL	Erwin Coumans, Google
11:10am	Asynchronous Computation Engine for Animation	George ElKoura, Pixar
11:40am	Parallel Evaluation of Character Rigs Using TBB	Martin Watt, Dreamworks Animation
12:15pm	Done	

@multithreadvfx

www.multithreadingandvfx.org



“Multithreading for Visual Effects”

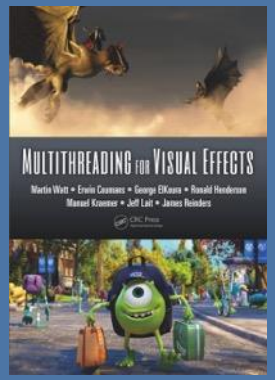


Taylor & Francis

Booth 528

@multithreadvfx

www.multithreadingandvfx.org



“Multithreading for Visual Effects”

Agenda

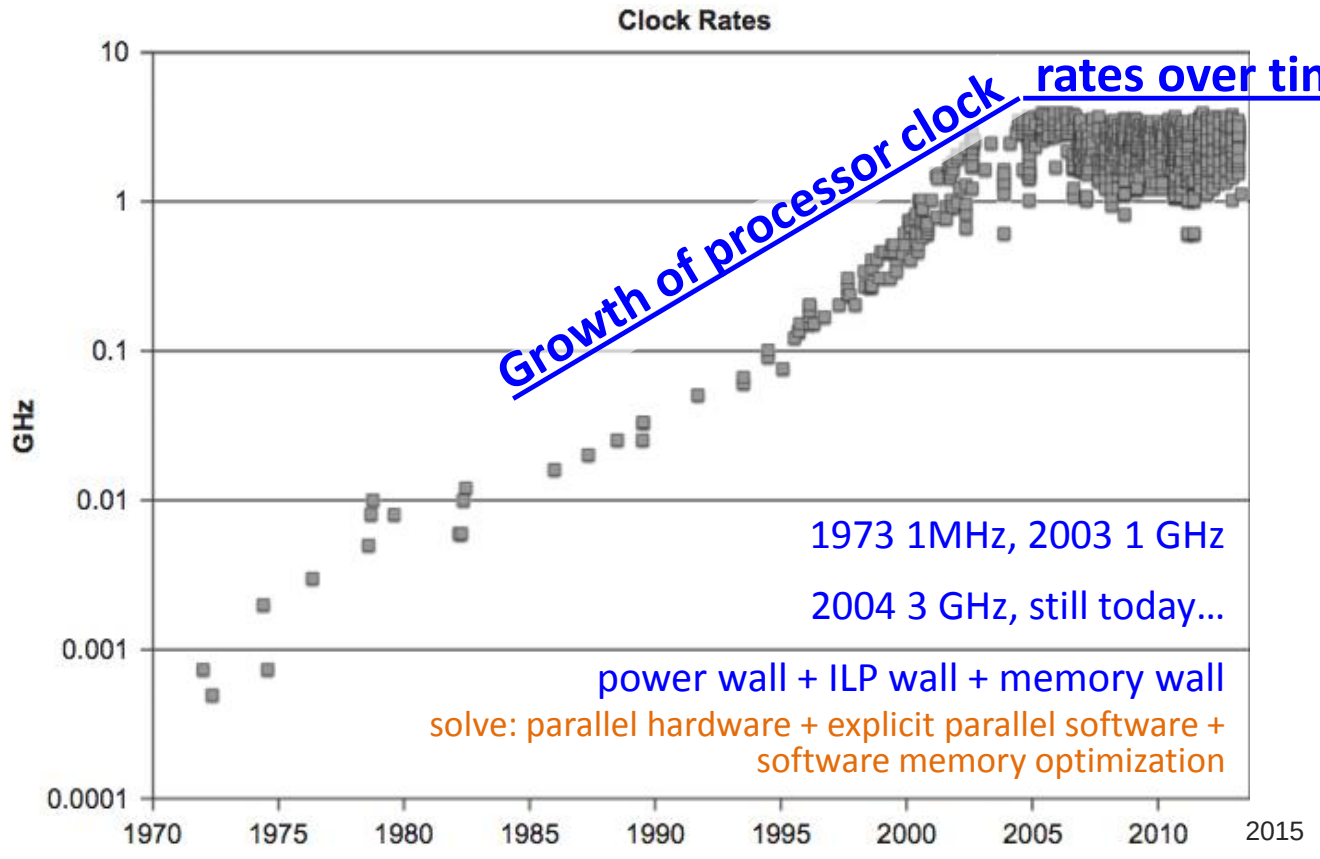
9:00am	Start	
9:00am	Introduction	James Reinders, Intel
9:05am	Multithreading Introduction and Overview	James Reinders, Intel
9:45am	Parallelism in Houdini - practical lessons learned	Jeff Lait, Side Effects Software
10:30am	Break	
10:45am	GPU Rigid Body Simulation Using OpenCL	Erwin Coumans, Google
11:10am	Asynchronous Computation Engine for Animation	George ElKoura, Pixar
11:40am	Parallel Evaluation of Character Rigs Using TBB	Martin Watt, Dreamworks Animation
12:15pm	Done	

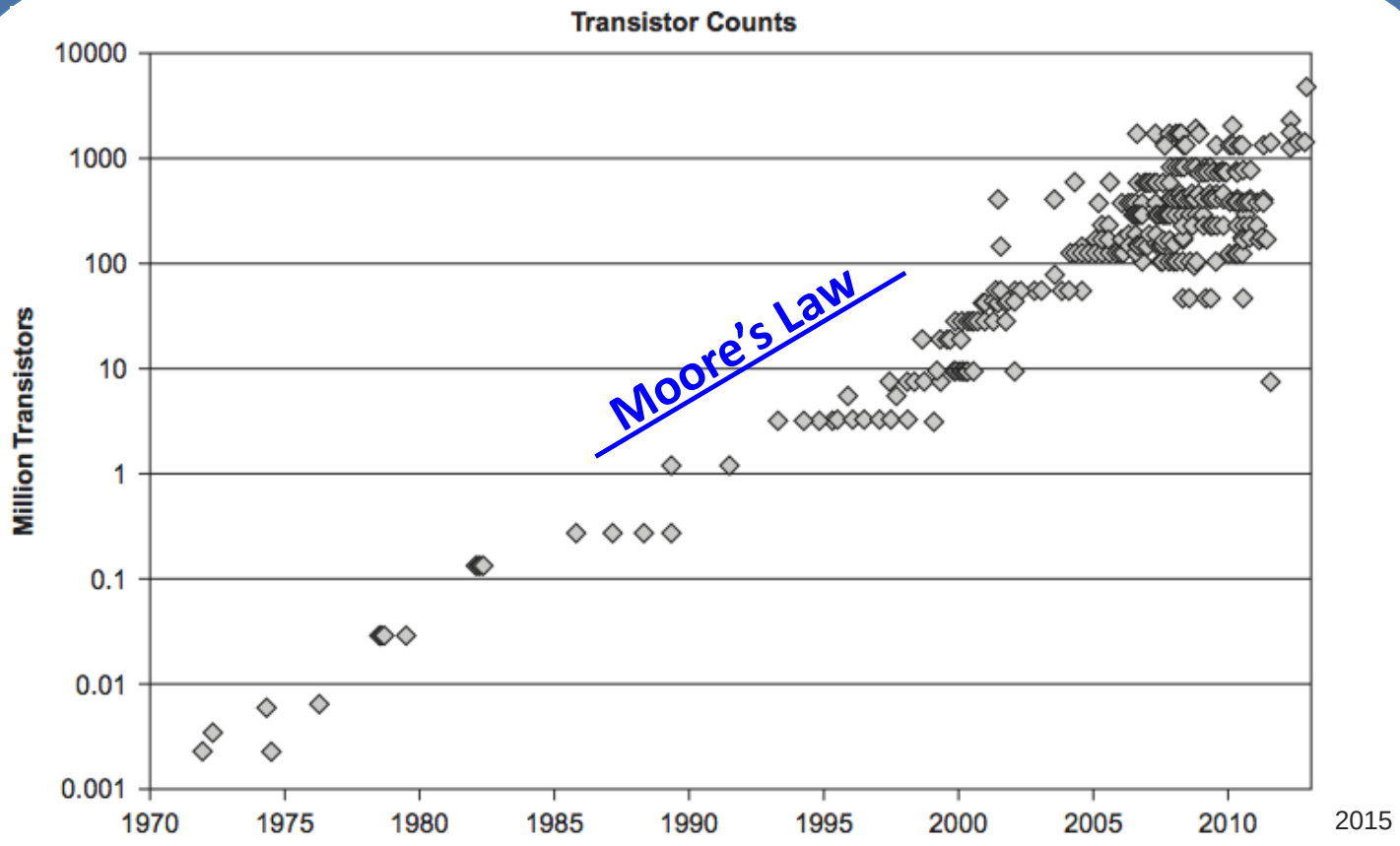
“Think Parallel”

“Think Parallel”

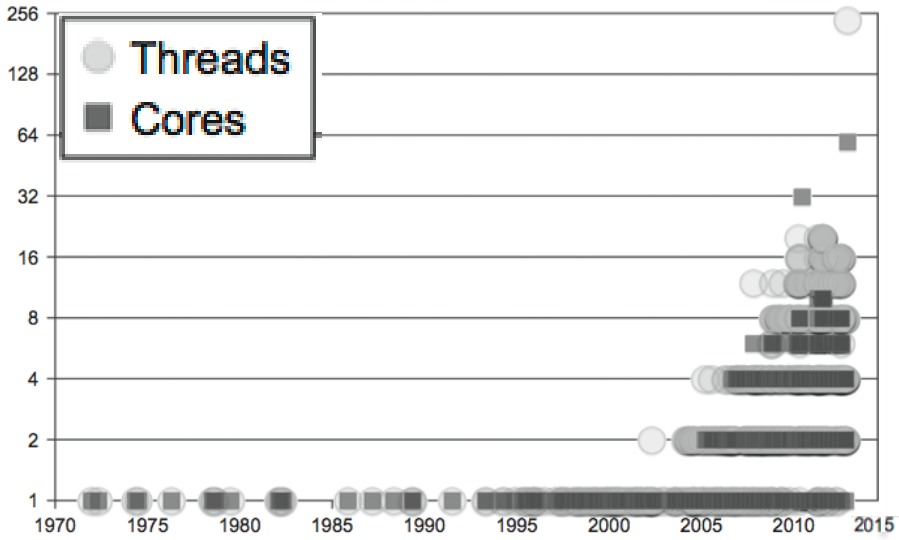
- Parallelism is almost never effective to “stick in” at the last minute in a program
- Think about everything in terms of how to do *in parallel* as cleanly as possible

Motivation

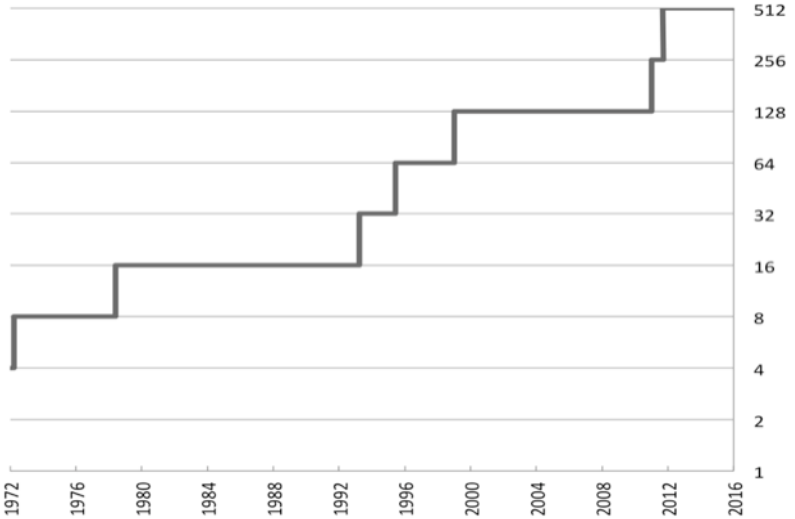




Core and Thread Counts



Width

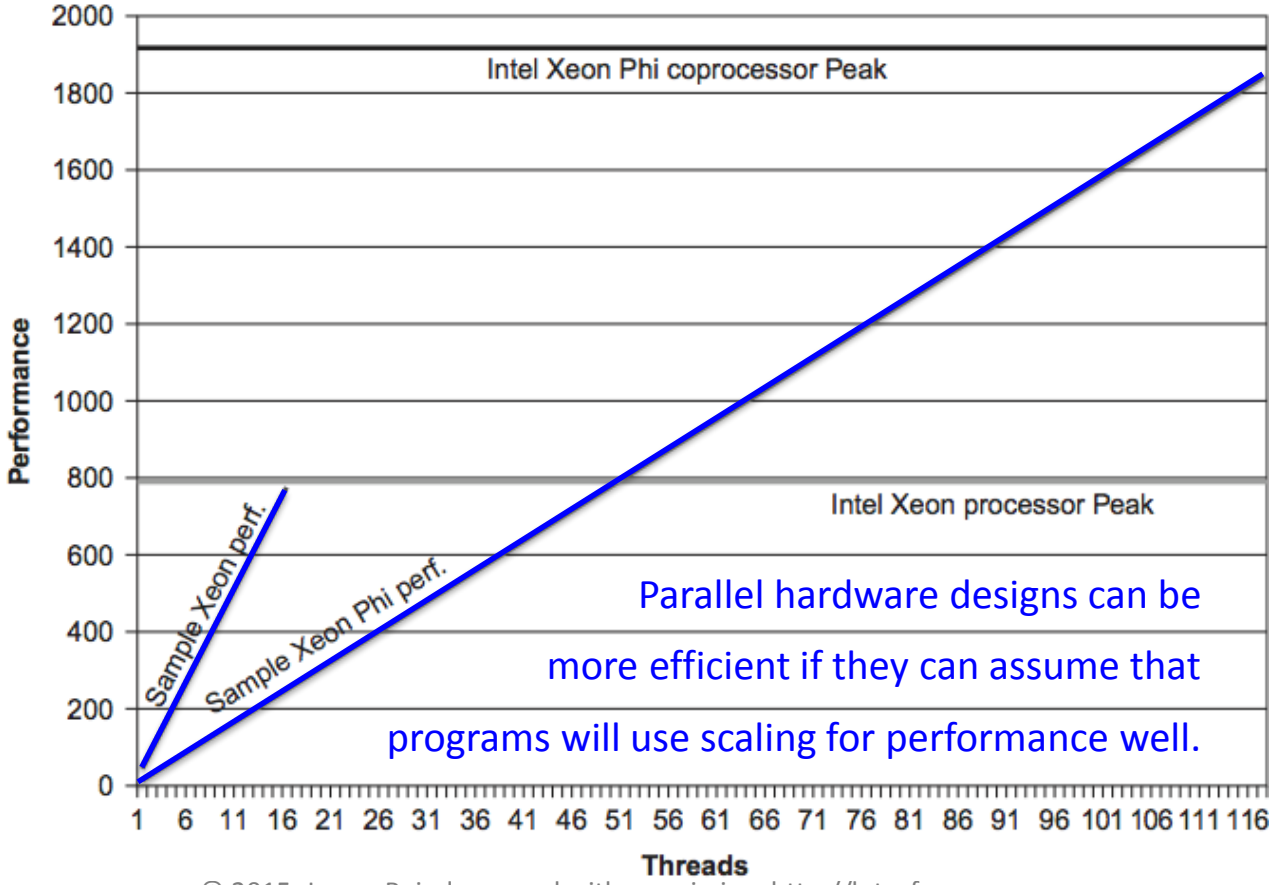


Single core, single thread, ruled for decades.

Multithread: grow die area small % for addition hardware thread(s) sharing resources.

Multicore/Many Core: 100% die area for additional hardware thread without sharing,

Data parallelism: handling more data at once, multibyte, multiword, many words.



Task

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

http://tinyurl.com/threadsYUCK

Task

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

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

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

1. Portability is impaired severely when coding "close to the hardware"
2. Nested parallelism is IMPORTANT
   (nearly impossible to manage well using "mandatory parallelism" methods such as threads)
3. Other parallelism mechanisms (e.g., vectorization) needs to be considered and balanced.

Course Notes ONLY material

What makes a good abstraction?

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

Parallel Programming

- No widely used (popular) programming language was designed for expressing parallel programming.
  - Not Fortran, C, C++, Java, C#, Perl, Python, Ruby

- This creates many challenges
- Fundamental question of all programming languages: *level of abstraction*

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.

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.

TBB, OpenMP*

- Lower level programming (less abstract):
  - Desired benefits:
    More control for the programmer.

OpenCL*, CUDA*

Advancing C and C++

www.threadingbuildingblocks.org

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

The most used method to parallelize C++ programs

Easier to maintain, scales better, easier to debug (get correct)

hand-coded

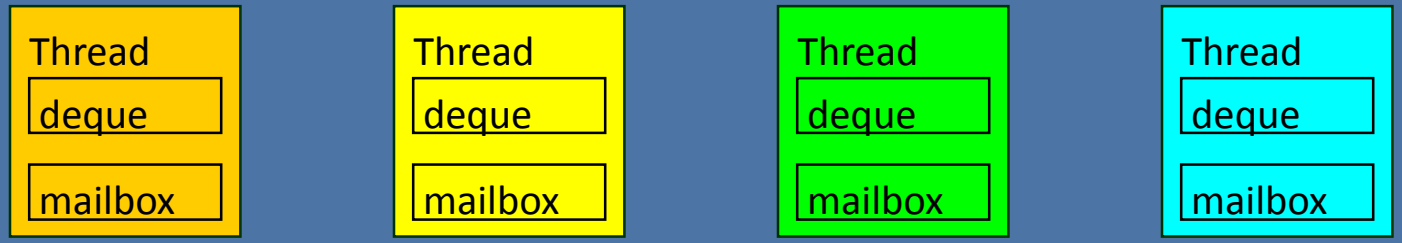
```
Thread Setup and Initialization
CRITICAL_SECTION MyMutex, MyMutex2, MyMutex3;
int get_num_cpus (void) {
    SYSTEM_INFO si;
    GetSystemInfo(&si);
    return (int)si.dwNumberOfProcessors;}
int nthreads = get_num_cpus ();
HANDLE *threads = (HANDLE *) alloca (nthreads * sizeof (HANDLE));
InitializeCriticalSection (&MyMutex);
InitializeCriticalSection (&MyMutex2);
InitializeCriticalSection (&MyMutex3);
for (int i = 0; i < nthreads; i++) {
    DWORD id;
    &threads[i] = CreateThread (NULL, 0, parallel_thread, i, 0, &id);}
for (int i = 0; i < nthreads; i++) {
    WaitForSingleObject (&threads[i], INFINITE);
}

Parallel Task Scheduling and Execution
const int MINPATCH = 150;
const int DIVFACTOR = 2;
typedef struct work_queue_entry_s {
    patch pch;
    struct work_queue_entry_s *next;
} work_queue_entry_t;
work_queue_entry_t *work_queue_head = NULL;
work_queue_entry_t *work_queue_tail = NULL;
void generate_work (patch* pchin)
{   int startx, stopx, starty, stopy;
    int xs,ys;
    startx=pchin->startx; stopx= pchin->stopx;
    starty=pchin->starty; stopy= pchin->stopy;
    if(((stopx-startx) >=MINPATCH) || ((stopy-starty) >= MINPATCH)) {
      int xpatchsize = (stopx-startx)/DIVFACTOR + 1;
      int ypatchsize = (stopy-starty)/DIVFACTOR + 1;
      for (ys=starty; ys<=stopy; ys+=ypatchsize)
      for (xs=startx; xs<=stopx; xs+=xpatchsize) {
        patch pch;
        pch.startx  = xs;
        pch.starty  = ys;
        pch.stopx   = MIN(xs+xpatchsize-1,stopx);
        pch.stopy   = MIN(ys+ypatchsize-1,stopy);
        generate_work (&pch);}
} else {
      /* just trace this patch */
      work_queue_entry_t *q = (work_queue_entry_t *) malloc (sizeof
(work_queue_entry_t));
      q->pch.starty = starty; q->pch.stopy = stopy;
      q->pch.startx = startx; q->pch.stopx = stopx;
      q->next = NULL;
      if (work_queue_head == NULL) {
        work_queue_head = q;
      } else {
        work_queue_tail->next = q;
      }
      work_queue_tail = q;
    }
}
void generate_worklist (void)
{
    patch pch;
    pch.startx = startx;
    pch.stopx = stopx;
    pch.starty = starty;
    pch.stopy = stopy;
    generate_work (&pch);
}
bool schedule_thread_work (patch &pch)
{
    EnterCriticalSection (&MyMutex3);
    work_queue_entry_t *q = work_queue_head;
    if (q != NULL) {
      pch = q->pch;
      work_queue_head = work_queue_head->next;
    }
    LeaveCriticalSection (&MyMutex3);
    return (q != NULL);
}
generate_worklist (),


void parallel_thread (void *arg)
{
    patch pch;
    while (schedule_thread_work (pch)) {
        for (int y = pch.starty; y <= pch.stopy; y++) {
            for (int x=pch.startx; x<=pch.stopx; x++) {
                render_one_pixel (x, y);})
        if (scene.displaymode == RT_DISPLAY_ENABLED) {
            EnterCriticalSection (&MyMutex3);
            for (int y = pch.starty; y <= pch.stopy; y++) {
                GraphicsDrawRow(pch.startx-1, y-1, pch.stopx-pch.startx+1,
(unsigned char *) &global_buffer[((y-starty)*totalx+(pch.startx-startx))*3]);
            }
            LeaveCriticalSection (&MyMutex3);
        }
    }
}
```

This example includes software developed by John E. Stone.

TBB

```
Thread Setup and Initialization
#include "tbb/task_scheduler_init.h"
#include "tbb/spin_mutex.h"
tbb::task_scheduler_init init;
tbb::spin_mutex MyMutex, MyMutex2;


Parallel Task Scheduling and Execution
#include "tbb/parallel_for.h"
#include "tbb/blocked_range2d.h"
class parallel_task {
public:
    void operator() (const tbb::blocked_range2d<int> &r) const {
        for (int y = r.rows().begin(); y != r.rows().end(); ++y) {
            for (int x = r.cols().begin(); x != r.cols().end(); x++
) {
                render_one_pixel (x, y);
            }
        }
        if (scene.displaymode == RT_DISPLAY_ENABLED) {
            tbb::spin_mutex::scoped_lock lock (MyMutex2);
            for (int y = r.rows().begin(); y != r.rows().end(); +
+y) {
                GraphicsDrawRow(startx-1, y-1, totalx, (unsigned
char *) &global_buffer[(y-starty)*totalx*3]);
            }
        }
    }
    parallel_task () {}
};
parallel_for (tbb::blocked_range2d<int> (starty, stopy + 1,
grain_size, startx, stopx + 1, grain_size), parallel_task ());
```

Work Stealing



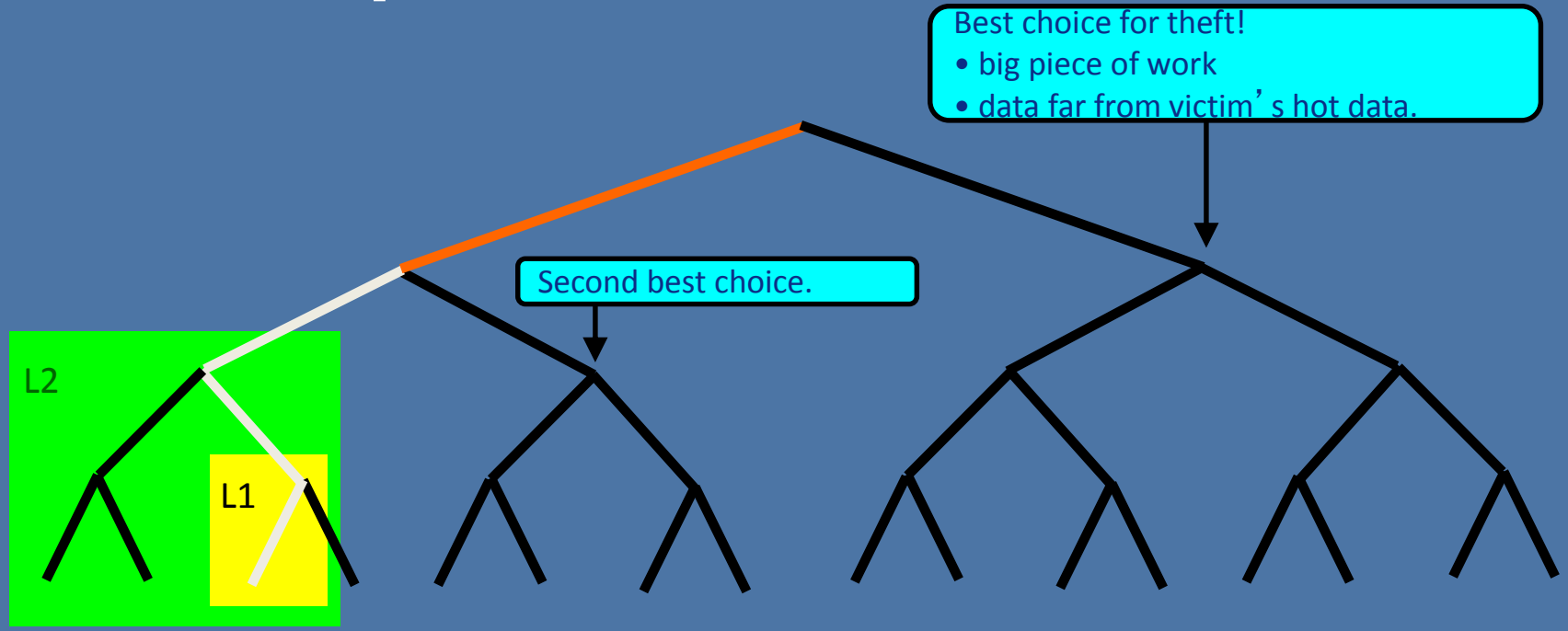
1. Take youngest task from my deque
2. Steal task advertised in mailbox
3. Steal oldest task from random victim

Locality

Cache Affinity

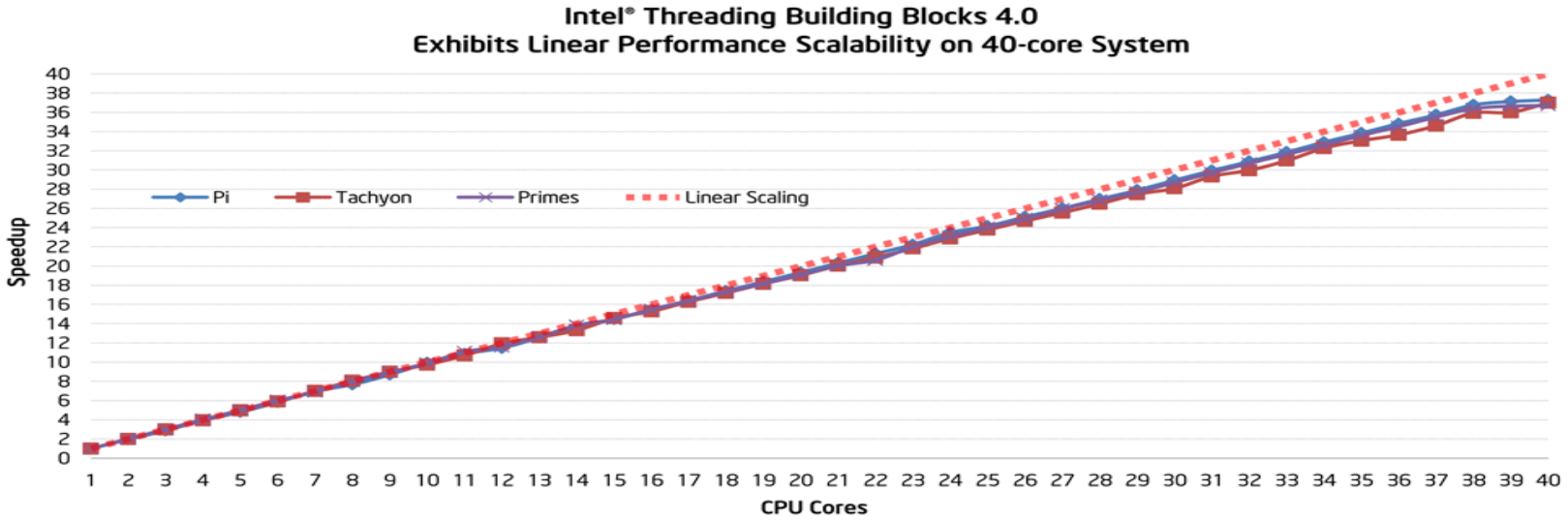
Load balance

Work Depth First; Steal Breadth First



victim thread

Scale Forward



Configuration Info - SW Versions: Intel® C++ Intel® 64 Compiler, Version 12.1, Intel® Threading Building Blocks 4.0; Hardware: 4 * Intel® Xeon® CPU E7- 4860 @ 2.27GHz (40 cores), 256GB Main Memory; Operating System: Linux, Red Hat* Enterprise Server* release 5.4, kernel 2.6.18-194.11.4.el5; Benchmark Source: Intel Corp.

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, refer to www.intel.com/performance/resources/benchmark_limitations.htm. * Other brands and names are the property of their respective owners

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

TBB Components

Parallel algorithms and data structures

Threads and synchronization

Memory allocation and task scheduling

Generic Parallel Algorithms

Efficient scalable way to exploit the power of multi-core without having to start from scratch.

Flow Graph

A set of classes to express parallelism as a graph of compute dependencies and/or data flow

Concurrent Containers

Concurrent access, and a scalable alternative to serial containers with external locking

Synchronization Primitives

Atomic operations, a variety of mutexes with different properties, condition variables

Task Scheduler

Sophisticated work scheduling engine that empowers parallel algorithms and the flow graph

Thread Local Storage

Unlimited number of thread-local variables

Threads

OS API wrappers

Miscellaneous

Thread-safe timers and exception classes

Memory Allocation

Scalable memory manager and false-sharing free allocators

Course Notes ONLY material

tbb::parallel_for

Has several forms.

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

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

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

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

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

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

Course Notes ONLY material

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 );
```

Course Notes ONLY material

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

Course Notes ONLY material

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];
      });
}
```

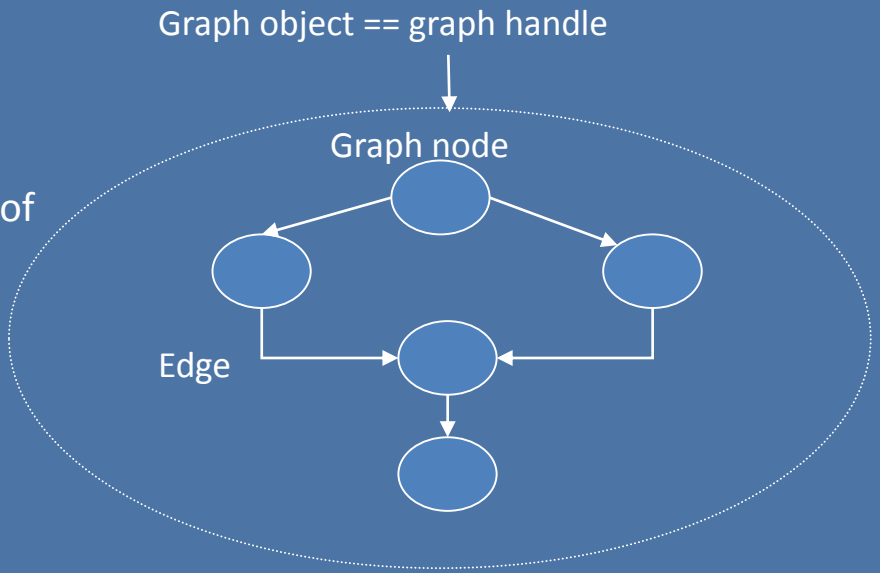
much easier to teach/read with lambdas

Course Notes ONLY material

Intel® TBB Class Graph: Components

Interesting "new" aspect since TBB 4.0 Release (2011)

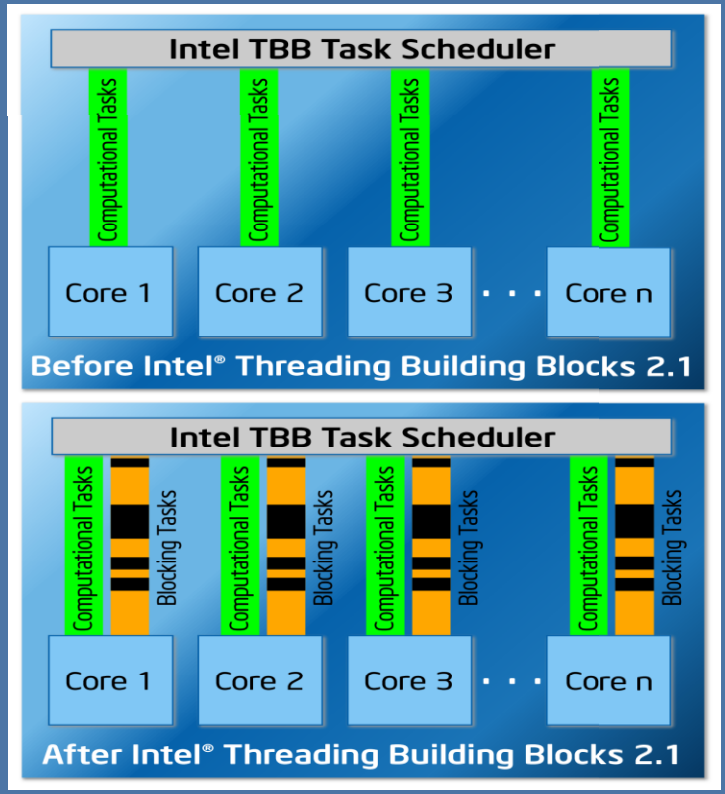
- Graph object
  - Contains a pointer to the root task
  - Owns tasks created on behalf of the graph
  - Users can wait for the completion of all tasks of the graph
- Graph nodes
  - Implement *sender* and/or *receiver* interfaces
  - Nodes manage messages and/or execute function objects
- Edges
  -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



Course Notes ONLY material

tbb_thread

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

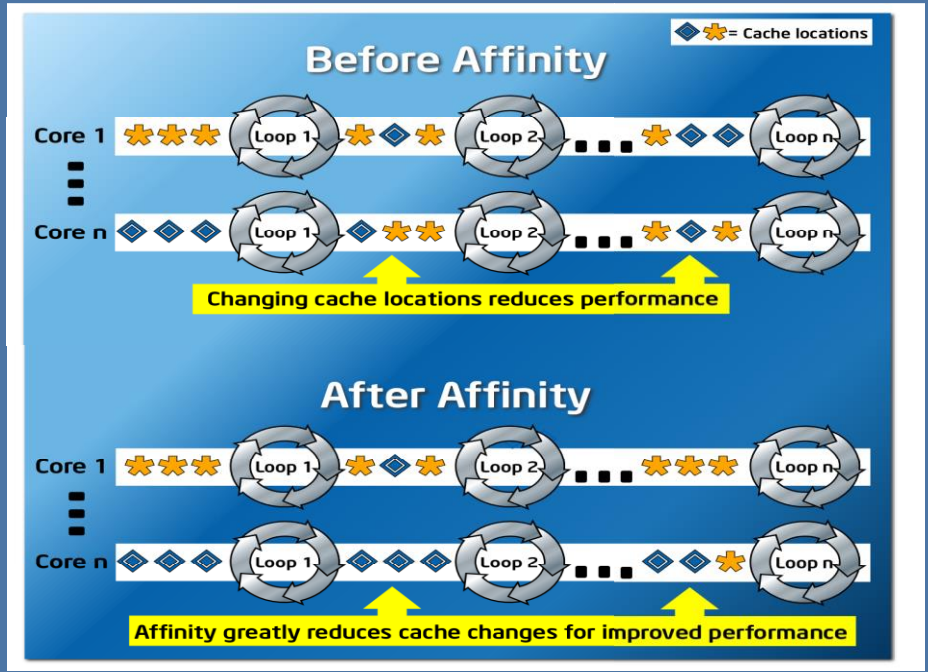


Course Notes ONLY material

affinity partitioner

Portable affinity mechanism gets more performance from chained parallel operations.

More efficient cache use.



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

Course Notes ONLY material

Intel® Cilk™ Plus

Open specification at cilkplus.org

Course Notes ONLY material

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

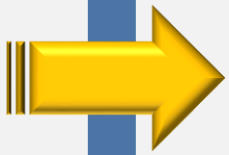
```
cilk_for (int i=0; i<n; ++i) {
  foo(a[i]);

}
```

Open specification at cilkplus.org

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

TBB

threadingbuildingblocks.org

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

Cilk™ Plus

cilkplus.org

- Reasons it might matter:
  - Space/time guarantees
  - Performance Portable
  - 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
  - Only in Intel and gcc compilers (and some in Clang/LLVM)
  - Standards adoption still “future”

OpenMP*

openmp.org

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

OpenCL*

khronos.org/opencl

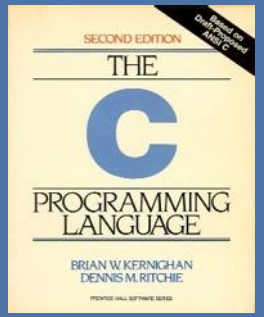
- Reasons it might matter:
  - Explicit heterogeneous controls
  - Everywhere (ports)
  - Non-proprietary
  - Underpinning for tools and libraries
- Problems:
  - Low level
  - Not performance portable

Choice is Good

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

Course Notes ONLY material

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



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

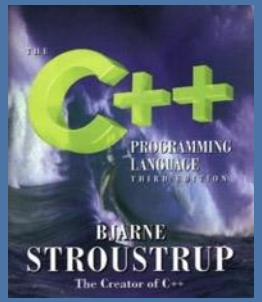


Photo: Wikimedia Commons (http://commons.wikimedia.org)

Course Notes ONLY material

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

Course Notes ONLY material

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

anonymous functions

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

Course Notes ONLY material

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

anonymous functions

Core language functionality improvements

- Variadic templates
- New string literals
- User-defined literals
- Multithreading memory model
- Thread-local storage

defining visibility of stores

- 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

Course Notes ONLY material

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

anonymous functions

Core language functionality improvements

- Variadic templates
- New string literals
- User-defined literals

defining visibility of stores

- 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

futures & promises, async

- 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

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();
```

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.

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().

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.

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.

Course Notes ONLY material

What about futures & promises?

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

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

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

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

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

Vectorization: Who, What, Why, When

A moment of clarity

task parallelism

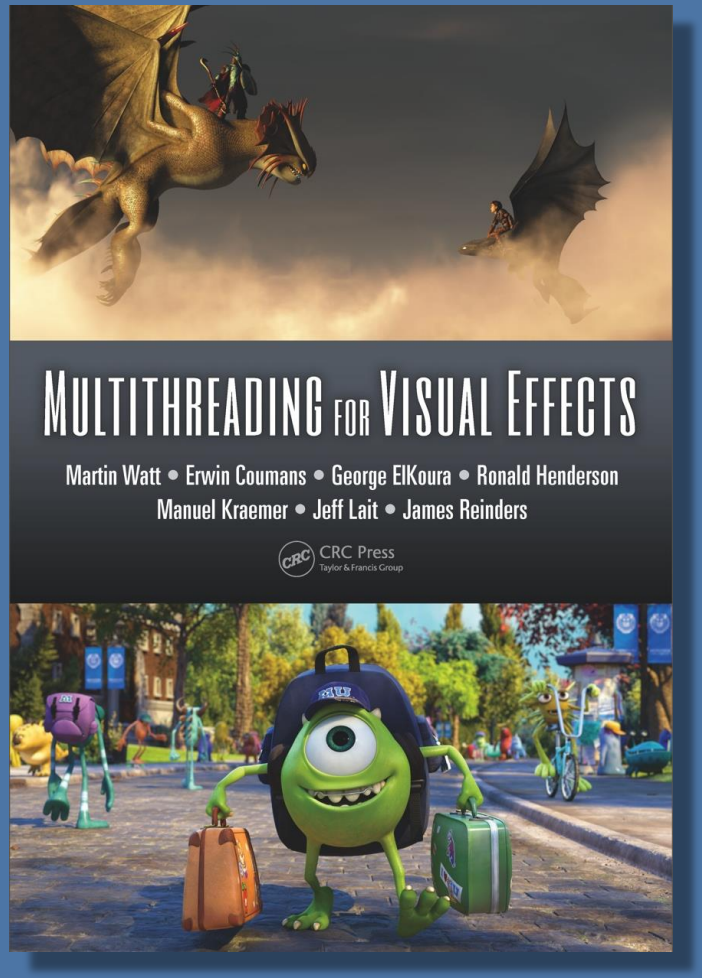
doesn’t matter

without data parallelism

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

Parallel first

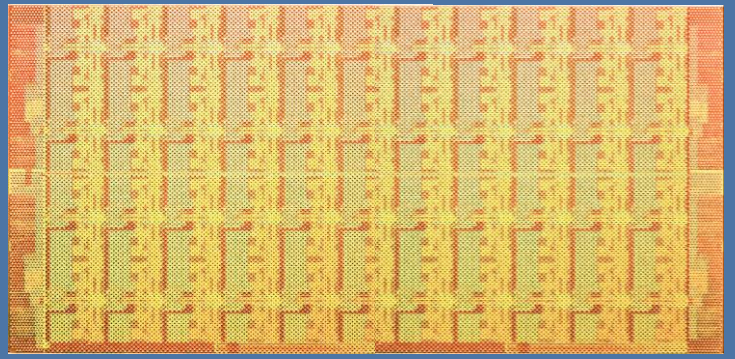
Vectorize second

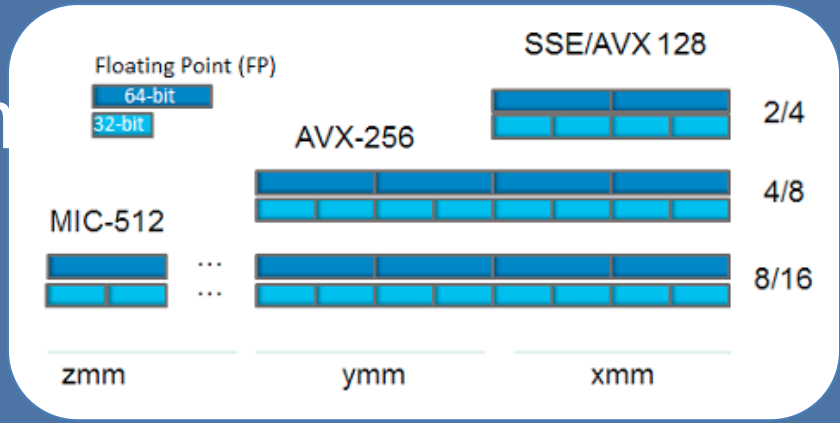


Multithreading is more powerful than vectorization – by simple math:

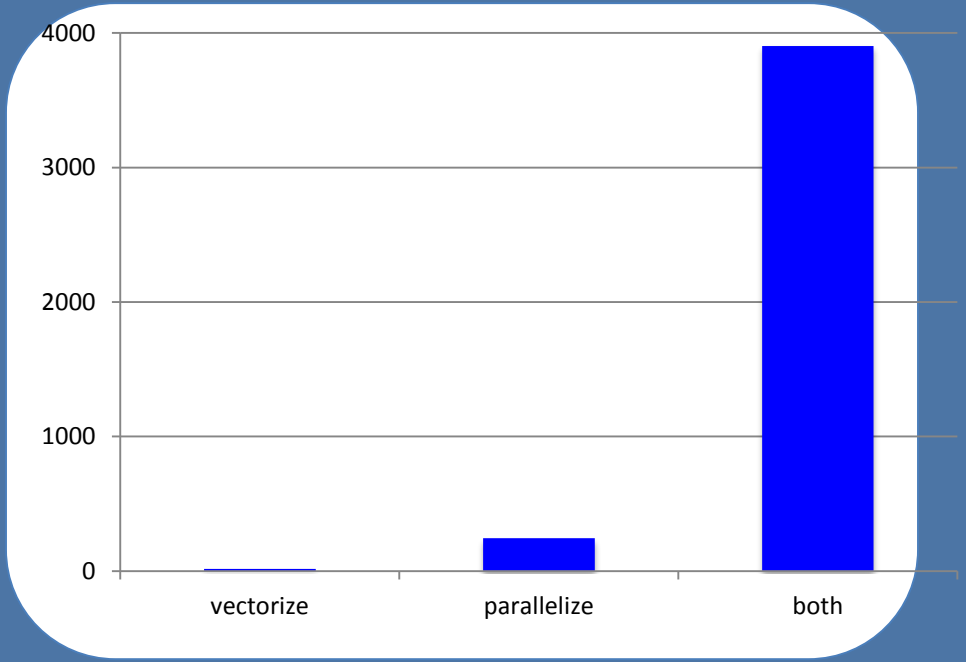
16 way from vectorization

244 way from thread parallelism





16 x 244 = 3904



Multithreading

Vectorization

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];
}
```

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

vector data operations: data operations done in parallel

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

Loop:

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

Loop:

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

vector data operations:

We call this “vectorization”

Loop:

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

Loop:

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

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!

vectorization solutions

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

vectorization solutions

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

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

C99 “restrict” keyword

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

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

IVDEP

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

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

simd

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

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

omp simd

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

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

Course Notes ONLY material

simd instruction intrinsics

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

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

Course Notes ONLY material

array operations

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

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

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

Course Notes ONLY material

__declspec(vector)

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

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

Course Notes ONLY material

#pragma omp declare simd

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

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

Course Notes ONLY material

kernel

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

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

Many choices… I recommend:

omp simd

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

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

Course Notes ONLY material

Memory

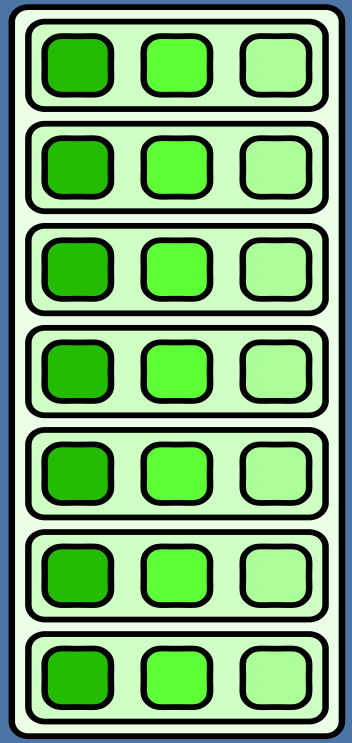
optimizing the sharing & movement of data

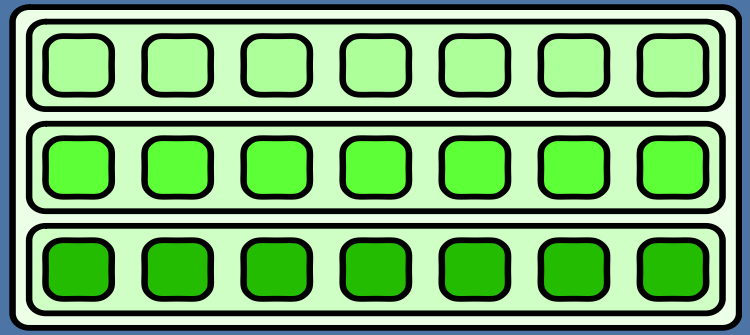
is *very often* more important than

optimizing calculations

TREND: “very often” heads toward “always”

Data Layout: AoS vs. SoA





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

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

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.

Course Notes ONLY material

sharing

decompose data to minimize sharing between tasks (threads)

beware: false sharing (it’s very real)

Think Parallel

- Parallelism does NOT work well if "added" at the last minute in a program

- think about *every thing* "how to do in parallel cleanly"

Course Notes ONLY material

Think Parallel

- Abstract parallelism is best – can be taught without learning computer architecture

- Contradicting the desire to ignore computer architecture as a driver for programming…

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

http://www.multithreadingandvfx.org/ *- downloads include material from today*

2014: Multithreading for Visual Effects

Expansion of material covered in this tutorial plus some extras.

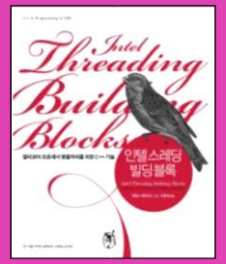
English

http://threadingbuildingblocks.com

2007: Intel Threading Building Blocks

Remains an excellent introduction to TBB; newer features are not covered

English, Chinese, Japanese, Korean



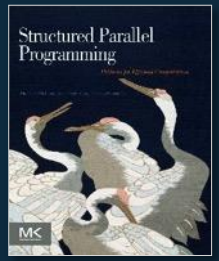
http://parallelbook.com

website has free teaching materials

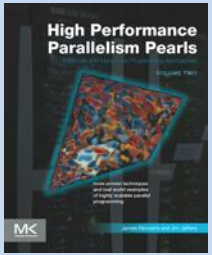
2012: Structured Parallel Programming

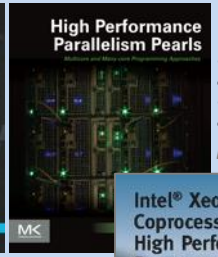
Excellent text for essentials of parallel programming for C/C++

English, Japanese



http://lotsofcores.com



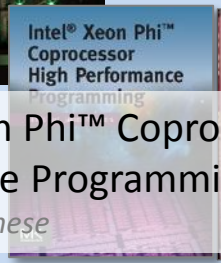


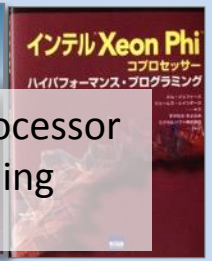
High Performance Parallelism Pearls

2015 …Pearls Volume Two

2014 …Pearls Volume One

English





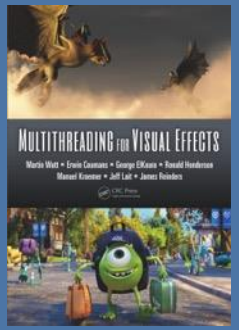
2013: Intel® Xeon Phi™ Coprocessor High Performance Programming

English, Japanese, Chinese

Thank you!

@multithreadvfx

www.multithreadingandvfx.org



“Multithreading for Visual Effects”

Agenda

9:00am	Start	
9:00am	Introduction	James Reinders, Intel
9:05am	Multithreading Introduction and Overview	James Reinders, Intel
9:45am	Parallelism in Houdini - practical lessons learned	Jeff Lait, Side Effects Software
10:30am	Break	
10:45am	GPU Rigid Body Simulation Using OpenCL	Erwin Coumans, Google
11:10am	Asynchronous Computation Engine for Animation	George ElKoura, Pixar
11:40am	Parallel Evaluation of Character Rigs Using TBB	Martin Watt, Dreamworks Animation
12:15pm	Done	

Legal Disclaimer & Optimization Notice

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