Threading for Performance with Intel® Threading Building Blocks
Acknowledgments for today’s lecture

• Thread Building Blocks, Arch Robinson, HPCC 2007 tutorial
  —http://www.tlc2.uh.edu/hpcc07/Schedule/tbBlocks

• “Threading for Performance with Intel Thread Building Blocks: Thinking Parallel”, Victoria Gromova
  —http://softwaredispatch.intel.com/?lid=1861&t=1

  —http://www.oreilly.com/catalog/9780596514808/
Summary of Intel® Threading Building Blocks

• It is a library
• You specify task patterns, not threads
• Targets threading for robust performance
• Does well with nested parallelism
• Compatible with other threading packages
• Emphasizes scalable, data parallel programming
• Generic programming enables distribution of broadly-useful high-quality algorithms and data structures.
• Available in GPL-ed version, as well as commercially licensed.
You specify *tasks* (what can run concurrently) instead of threads

- Library maps your logical tasks onto physical threads, efficiently using cache and balancing load
- Full support for *nested parallelism*

Targets threading for *scalable performance*

- Portable across Linux*, Mac OS*, Windows*, and Solaris*

Emphasizes *scalable data parallel* programming

- Loop parallelism tasks are more scalable than a fixed number of separate tasks

*Compatible* with other threading packages

- Can be used in concert with native threads and OpenMP

*Open source* and licensed versions available
Family Tree

Languages
- Threaded-C
- Continuation tasks
- Task stealing
- Cilk
- Space efficient scheduler
- Cache-oblivious algorithms

Pragmas
- OpenMP*
- Fork/join tasks
- OpenMP taskqueue
- While & recursion

Libraries
- STAPL
- Recursive ranges
- Intel® TBB
- JSR-166 (FJTask) containers
- ECMA .NET*
- Parallel iteration classes

Chare Kernel
- Small tasks

*Other names and brands may be claimed as the property of others
Components of TBB (version 2.1)

- **Parallel algorithms**
  - parallel_for (improved)
  - parallel_reduce (improved)
  - parallel_do (new)
  - pipeline (improved)
  - parallel_sort
  - parallel_scan

- **Concurrent containers**
  - concurrent_hash_map
  - concurrent_queue
  - concurrent_vector (all improved)

- **Task scheduler**
  - With new functionality

- **Synchronization primitives**
  - atomic operations
  - various flavors of mutexes (improved)

- **Memory allocators**
  - tbb_allocator (new), cache_aligned_allocator, scalable_allocator

- **Utilities**
  - tick_count
  - tbb_thread (new)
Tasks are light-weight entities at user-level

- Intel® TBB parallel algorithms map tasks onto threads automatically
- Task scheduler manages the thread pool
  - Scheduler is *unfair* to favor tasks that have been most recent in the cache
- Oversubscription and undersubscription of core resources is prevented by task-stealing technique of TBB scheduler
Generic Parallel Algorithms
Task scheduler powers high level parallel patterns that are pre-packaged, tested, and tuned for scalability

- **parallel_for**: load-balanced parallel execution of loop iterations where iterations are independent
- **parallel_reduce**: load-balanced parallel execution of independent loop iterations that perform reduction (e.g. summation of array elements)
- **parallel_while**: load-balanced parallel execution of independent loop iterations with unknown or dynamically changing bounds (e.g. applying function to the element of linked list)
- **parallel_scan**: template function that computes parallel prefix
- **pipeline**: data-flow pipeline pattern
- **parallel_sort**: parallel sort
Grain Size

OpenMP has similar parameter

Part of `parallel_for`, not underlying task scheduler

- Grain size exists to amortize overhead, not balance load
- Units of granularity are loop iterations

Typically only need to get it right within an order of magnitude
Tuning Grain Size

- Tune by examining single-processor performance
- When in doubt, err on the side of making it a little too large, so that performance is not hurt when only one core is available.
The parallel_for Template

```cpp
template <typename Range, typename Body>
void parallel_for(const Range& range, const Body &body);
```

Requires definition of:

- A range type to iterate over
  - Must define a copy constructor and a destructor
  - Defines `is_empty()`
  - Defines `is_divisible()`
  - Defines a splitting constructor, `R(R &r, split)`
- A body type that operates on the range (or a subrange)
  - Must define a copy constructor and a destructor
  - Defines `operator()`
Body is Generic

Requirements for `parallel_for` Body

- `Body::Body(const Body&)`  
  Copy constructor
- `Body::~Body()`  
  Destructor
- `void Body::operator() (Range& subrange) const`  
  Apply the body to `subrange`.

`parallel_for` partitions original range into subranges, and deals out subranges to worker threads in a way that:

- Balances load
- Uses cache efficiently
- Scales
Range is Generic

Requirements for `parallel_for` Range

- `R::R (const R&)`  
  Copy constructor

- `R::~R()`  
  Destructor

- `bool R::is_empty() const`  
  True if range is empty

- `bool R::is_divisible() const`  
  True if range can be partitioned

- `R::R (R& r, split)`  
  Splitting constructor; splits `r` into two subranges

Library provides predefined ranges
- `blocked_range` and `blocked_range2d`

You can define your own ranges
How splitting works on `blocked_range2d`

tasks available to be scheduled to other threads (thieves)
How splitting works on `blocked_range2d`

Tasks available to be scheduled to other threads (thieves)
How splitting works on `blocked_range2d`

tasks available to be scheduled to other threads (thieves)
How splitting works on blocked_range2d

tasks available to be scheduled to other threads (thieves)
const int N = 100000;

void change_array(float array, int M) {
    for (int i = 0; i < M; i++){
        array[i] *= 2;
    }
}

int main (){
    float A[N];
    initialize_array(A);
    change_array(A, N);
    return 0;
}
An Example using parallel_for

Include and initialize the library
An Example using parallel_for

Include and initialize the library

```cpp
#include "tbb/task_scheduler_init.h"
#include "tbb/blocked_range.h"
#include "tbb/parallel_for.h"

using namespace tbb;

int main (){
    task_scheduler_init init;
    float A[N];
    initialize_array(A);
    parallel_change_array(A, N);
    return 0;
}
```

Include Library Headers

Use namespace

Initialize scheduler

blue = original code
green = provided by TBB
red = boilerplate for library
An Example using parallel_for

Include and initialize the library

```cpp
#include "tbb/task_scheduler_init.h"
#include "tbb/blocked_range.h"
#include "tbb/parallel_for.h"

using namespace tbb;

int main(){
    task_scheduler_init init;
    float A[N];
    initialize_array(A);
    parallel_change_array(A, N);
    return 0;
}
```

blue = original code  
green = provided by TBB  
red = boilerplate for library
Use the `parallel_for` pattern

- blue = original code
- green = provided by TBB
- red = boilerplate for library
An Example using `parallel_for`

Use the `parallel_for` pattern

```cpp
class ChangeArrayBody {
    float *array;

public:
    ChangeArrayBody (float *a): array(a) {}  
    void operator()( const blocked_range<int>& r ) const{
        for (int i = r.begin(); i != r.end(); i++){
            array[i] *= 2;
        }
    }
};

void parallel_change_array(float *array, int M) {
    parallel_for (blocked_range<int>(0, M, IdealGrainSize),
                  ChangeArrayBody(array));
}
```

blue = original code  
green = provided by TBB  
red = boilerplate for library
An Example using parallel_for

Use the **parallel_for** pattern

```cpp
class ChangeArrayBody {
    float *array;
public:
    ChangeArrayBody (float *a): array(a) {}  
    void operator()( const blocked_range<int>& r ) const{
        for (int i = r.begin(); i != r.end(); i++){
            array[i] *= 2;
        }
    }
};

void parallel_change_array(float *array, int M) {
    parallel_for (blocked_range<int>(0, M, IdealGrainSize),
                  ChangeArrayBody(array));
}
```

**Define Task**

**Use Pattern**

**Establish grain size**
The parallel_reduce Template

```cpp
template <typename Range, typename Body>
void parallel_reduce (const Range& range, Body &body);
```

Requirements for parallel_reduce Body

- `Body::Body( const Body&, split )` - Splitting constructor
- `Body::~Body()` - Destructor
- `void Body::operator() (Range& subrange) const` - Accumulate results from `subrange`
- `void Body::join( Body& rhs );` - Merge result of `rhs` into the result of this.

Reuses Range concept from parallel_for
static long num_steps=100000;
double step, pi;

void main(int argc, char* argv[])
{
    int i;
    double x, sum = 0.0;

    step = 1.0/(double) num_steps;
    for (i=0; i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n", pi);
}
```c
#include "tbb/parallel_reduce.h"
#include "tbb/task_scheduler_init.h"
#include "tbb/blocked_range.h"

using namespace tbb;

int main(int argc, char* argv[]) {
    double pi;
    double width = 1./(double)num_steps;
    MyPi step((double *const)&width);

    parallel_reduce(blocked_range<size_t>(0,num_steps), step);
    pi = step.sum*width;

    printf("The value of PI is \%15.12f\n",pi);
    return 0;
}
```
class MyPi {
    double *const my_step;
public:
    double sum;
    void operator()( const blocked_range<size_t>& r ) {
        double step = *my_step;
        double x;
        for (size_t i=r.begin(); i!=r.end(); ++i)
        {
            x = (i + .5)*step;
            sum = sum + 4.0/(1. + x*x);
        }
    }
    MyPi(MyPi& x, split) : my_step(x.my_step), sum(0) {}
    void join(const MyPi& y) {sum += y.sum;}
    MyPi(double *const step) : my_step(step), sum(0) {}
};
Parallel pipeline

• Linear pipeline of stages
  — You specify maximum number of items that can be in flight
  — Handle arbitrary DAG by mapping onto linear pipeline

• Each stage can be serial or parallel
  — Serial stage processes one item at a time, in order.
  — Parallel stage can process multiple items at a time, out of order.

• Uses cache efficiently
  — Each worker thread carries an item through as many stages as possible
  — Biases towards finishing old items before tackling new ones
Parallel pipeline

Parallel stage scales because it can process items in parallel or out of order.

Serial stage processes items one at a time in order.

Tag incoming items with sequence numbers

Items wait for turn in serial stage

Throughput limited by throughput of slowest serial stage.

Uses sequence numbers recover order for serial stage.

Controls excessive parallelism by limiting total number of items flowing through pipeline.
Task Based Approach

Intel® TBB provides C++ constructs that allow you to express parallel solutions in terms of task objects
- Task scheduler manages thread pool
- Task scheduler avoids common performance problems of programming with threads

<table>
<thead>
<tr>
<th>Problem</th>
<th>Intel® TBB Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oversubscription</td>
<td>One scheduler thread per hardware thread</td>
</tr>
<tr>
<td>Fair scheduling</td>
<td>Non-preemptive unfair scheduling</td>
</tr>
<tr>
<td>High overhead</td>
<td>Programmer specifies tasks, not threads</td>
</tr>
<tr>
<td>Load imbalance</td>
<td>Work-stealing balances load</td>
</tr>
</tbody>
</table>
Example: Naive Fibonacci Calculation

Recursion typically used to calculate Fibonacci number

Widely used as toy benchmark

- Easy to code
- Has unbalanced task graph

```c
long SerialFib( long n ) {
    if( n<2 )
        return n;
    else
        return SerialFib(n-1) + SerialFib(n-2);
}
```
Example: Naive Fibonacci Calculation

Can envision Fibonacci computation as a task graph
Fibonacci - Task Spawning Solution

Use TBB tasks to thread creation and execution of task graph

Create new root task
  - Allocate task object
  - Construct task

Spawn (execute) task, wait for completion

```cpp
long ParallelFib( long n ) {
    long sum;
    FibTask& a = *new(Task::allocate_root()) FibTask(n,&sum);
    Task::spawn_root_and_wait(a);
    return sum;
}
```
class FibTask: public task {
public:
    const long n;
    long* const sum;
FibTask(long n_, long* sum_):
    n(n_), sum(sum_)
{}
task* execute() {  // Overrides virtual function task::execute
    if( n<CutOff ) {
        *sum = SerialFib(n);
    } else {
        long x, y;
        FibTask& a = *new(allocate_child()) FibTask(n-1,&x);
        FibTask& b = *new(allocate_child()) FibTask(n-2,&y);
        set_ref_count(3);  // 3 = 2 children + 1 for wait
        spawn(b);
        spawn_and_wait_for_all(a);
        *sum = x+y;
    }
    return NULL;
}
};
class FibTask: public task {
public:
    const long n;
    long* const sum;
    FibTask(long n_, long* sum_) :
        n(n_), sum(sum_)
    {}
    task* execute() { // Overrides virtual function task::execute
        if( n<CutOff ) {
            *sum = SerialFib(n);
        } else {
            long x, y;
            FibTask& a = *new(allocate_child()) FibTask(n-1,&x);
            FibTask& b = *new(allocate_child()) FibTask(n-2,&y);
            set_ref_count(3); // 3 = 2 children + 1 for wait
            spawn(b);
            spawn_and_wait_for_all(a);
            *sum = x+y;
        }
        return NULL;
    }
};
class FibTask: public task {
public:
    const long n;
    long* const sum;
    FibTask(long n_, long* sum_) :
        n(n_), sum(sum_)
    {}
    task* execute() {     // Overrides virtual function task::execute
        if (n<CutOff) {
            *sum = SerialFib(n);
        } else {
            long x, y;
            FibTask& a = *new(allochild()) FibTask(n-1,&x);
            FibTask& b = *new(allochild()) FibTask(n-2,&y);
            set_ref_count(3); // 3 = 2 children + 1 for wait
            spawn(b);
            spawn_and_wait_for_all(a);
            *sum = x+y;
        }
        return NULL;
    }
};
The execute method does the computation of a task.

class FibTask: public task {
public:
    const long n;
    long* const sum;
    FibTask(long n_, long* sum_): 
        n(n_), sum(sum_)
    {} 
task* execute() { // Overrides virtual function task::execute
        if (n<CutOff) {
            *sum = SerialFib(n);
        } else {
            long x, y;
            FibTask& a = *new(allocate_child()) FibTask(n-1,&x);
            FibTask& b = *new(allocate_child()) FibTask(n-2,&y);
            set_ref_count(3); // 3 = 2 children + 1 for wait
            spawn(b);
            spawn_and_wait_for_all(a);
            *sum = x+y;
        } 
        return NULL;
    }
};
Fibonacci - Task Spawning Solution

class FibTask: public task {
public:
    const long n;
    long* const sum;
    FibTask(long n_, long* sum_) :
        n(n_), sum(sum_)
    {}
    task* execute() { // Overrides virtual function task::execute
        if( n<CutOff ) {
            *sum = SerialFib(n);
        } else {
            long x, y;
            FibTask& a = *new(allocate_child()) FibTask(n-1,&x);
            FibTask& b = *new(allocate_child()) FibTask(n-2,&y);
            set_ref_count(3); // 3 = 2 children + 1 for wait
            spawn(b);
            spawn_and_wait_for_all(a);
            *sum = x+y;
        }
        return NULL;
    }
};

Create new child tasks to compute \((n-1)\)th and \((n-2)\)th Fibonacci numbers
class FibTask: public task {
public:
    const long n;
    long* const sum;
    FibTask(long n_, long* sum_) :
        n(n_), sum(sum_)
    {}
    task* execute() {     // Overrides virtual function task::execute
        if( n<CutOff ) {
            *sum = SerialFib(n);
        } else {
            long x, y;
            FibTask& a = *new(allocate_child()) FibTask(n-1,&x);
            FibTask& b = *new(allocate_child()) FibTask(n-2,&y);
            set_ref_count(3); // 3 = 2 children + 1 for wait
            Spawn(b);
            spawn_and_wait_for_all(a);
            *sum = x+y;
        }
        return NULL;
    }
};

Reference count is used to know when spawned tasks have completed
Set before spawning any children
class FibTask: public task {
    public:
        const long n;
        long* const sum;
        FibTask(long n_, long* sum_) :
            n(n_), sum(sum_)
        {}
        task* execute() { // Overrides virtual function task::execute
            if( n<CutOff ) {
                *sum = SerialFib(n);
            } else {
                long x, y;
                FibTask& a = *new(allocate_child()) FibTask(n-1,&x);
                FibTask& b = *new(allocate_child()) FibTask(n-2,&y);
                set_ref_count(3); // 3 = 2 children + 1 for wait
                spawn(b);
                spawn_and_wait_for_all(a);
                *sum = x+y;
            }
            return NULL;
        }
};
class FibTask: public task {
public:
    const long n;
    long* const sum;
    FibTask(long n_, long* sum_) :
        n(n_), sum(sum_) {
    }

    task* execute() { // Overrides virtual function task::execute
        if( n<CutOff ) {
            *sum = SerialFib(n);
        } else {
            long x, y;
            FibTask& a = *new(allocate_child()) FibTask(n-1,&x);
            FibTask& b = *new(allocate_child()) FibTask(n-2,&y);
            set_ref_count(3); // 3 = 2 children + 1 for wait
            spawn(b);
            spawn_and_wait_for_all(a);
            *sum = x+y;
        }
        return NULL;
    }
};

Fibonacci - Task Spawning Solution
class FibTask: public task {
    public:
        const long n;
        long* const sum;

    FibTask(long n_, long* sum_) :
        n(n_), sum(sum_)
    {} 
    task* execute() {     // Overrides virtual function task::execute
        if( n<CutOff ) {
            *sum = SerialFib(n);
        } else {
            long x, y;
            FibTask& a = *new(allocation_child()) FibTask(n-1,&x);
            FibTask& b = *new(allocation_child()) FibTask(n-2,&y);
            set_ref_count(3); // 3 = 2 children + 1 for wait
            spawn(b);
            spawn_and_wait_for_all(a);
            *sum = x+y;
        }
        return NULL;
    }
};
Further Optimizations Enabled by Scheduler

Recycle tasks
- Avoid overhead of allocating/freeing Task
- Avoid copying data and rerunning constructors/destructors

Continuation passing
- Instead of blocking, parent specifies another Task that will continue its work when children are done.
- Further reduces stack space and enables bypassing scheduler

Bypassing scheduler
- Task can return pointer to next Task to execute
  - For example, parent returns pointer to its left child
  - See include/tbb/parallel_for.h for example
- Saves push/pop on deque (and locking/unlocking it)
Low-Level Synchronization Primitives
Parallel tasks must sometimes touch shared data
  • When data updates might overlap, use mutual exclusion to avoid race

High-level generic abstraction for HW atomic operations
  • Atomically protect update of single variable

Critical regions of code are protected by scoped locks
  • The range of the lock is determined by its lifetime (scope)
    – Can’t forget to unlock it due to “return” or “break” statement
  • **Leaving lock scope calls the destructor, making it exception safe**
  • Minimizing lock lifetime avoids possible contention
  • Several mutex behaviors are available
    – Spin vs. queued
      – “are we there yet” vs. “wake me when we get there”
    – Writer vs. reader/reader (supports multiple readers/single writer)
    – Scoped wrapper of native mutual exclusion function
Atomic Execution

atomic<T>
- T should be integral type or pointer type
- Full type-safe support for 8, 16, 32, and 64-bit integers

Operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x.fetch_and_store (y)</code></td>
<td>z = x, y = x, return z</td>
</tr>
<tr>
<td><code>x.fetch_and_add (y)</code></td>
<td>z = x, x += y, return z</td>
</tr>
<tr>
<td><code>x.compare_and_swap (y,p)</code></td>
<td>z = x, if (x==p) x=y; return z</td>
</tr>
</tbody>
</table>

```cpp
atomic <int> i;
...;
int z = i.fetch_and_add(2);
```
Mutex Concepts

Mutexes are C++ objects based on scoped locking pattern
Combined with locks, provide mutual exclusion

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M()</td>
<td>Construct unlocked mutex</td>
</tr>
<tr>
<td>~M()</td>
<td>Destroy unlocked mutex</td>
</tr>
<tr>
<td>typename M::scoped_lock</td>
<td>Corresponding scoped_lock type</td>
</tr>
<tr>
<td>M::scoped_lock()</td>
<td>Construct lock w/out acquiring a mutex</td>
</tr>
<tr>
<td>M::scoped_lock(M&amp;)</td>
<td>Construct lock and acquire lock on mutex</td>
</tr>
<tr>
<td>M::~scoped_lock()</td>
<td>Release lock if acquired</td>
</tr>
<tr>
<td>M::scoped_lock::acquire(M&amp;)</td>
<td>Acquire lock on mutex</td>
</tr>
<tr>
<td>M::scoped_lock::release()</td>
<td>Release lock</td>
</tr>
</tbody>
</table>
Mutex Flavors

**spin_mutex**
- Non-reentrant, unfair, spins in the user space
- VERY FAST in lightly contended situations; use if you need to protect very few instructions

**queuing_mutex**
- Non-reentrant, fair, spins in the user space
- Use Queuing_Mutex when scalability and fairness are important

**queuing_rw_mutex**
- Non-reentrant, fair, spins in the user space

**spin_rw_mutex**
- Non-reentrant, fair, spins in the user space
- Use ReaderWriterMutex to allow non-blocking read for multiple threads
#include "tbb/spin_mutex.h"
Node* FreeList;
typedef spin_mutex FreelistMutexType;
FreelistMutexType FreelistMutex;

Node* AllocateNode (){  
    Node* n;  
    {  
        FreelistMutexType::scoped_lock mylock(FreelistMutex);  
        n = FreeList;  
        if ( n ) FreeList = n->next;  
    }  
    if ( !n ) n = new Node();  
    return n;  
}

void FreeNode( Node* n ) {  
    FreelistMutexType::scoped_lock mylock(FreelistMutex);  
    n->next = FreeList;  
    FreeList = n;  
}
Generic Highly Concurrent Containers
Concurrent Containers

TBB Library provides highly concurrent containers

- STL containers are not concurrency-friendly: attempt to modify them concurrently can corrupt container
- Standard practice is to wrap a lock around STL containers
  - Turns container into serial bottleneck

Library provides fine-grained locking or lockless implementations

- Worse single-thread performance, but better scalability.
- Can be used with the library, OpenMP, or native threads.
Concurrency-Friendly Interfaces

Some STL interfaces are inherently not concurrency-friendly

For example, suppose two threads each execute:

```cpp
extern std::queue q;
if(!q.empty()) {
    item=q.front();
    q.pop();
}
```

At this instant, another thread might pop last element.

Solution: `concurrent_queue` has `pop_if_present`
**Concurrent Queue Container**

`concurrent_queue<T>`

- **Preserves local FIFO order**
  - If thread pushes and another thread pops two values, they come out in the same order that they went in
- **Method** `push(const T&)` places copy of item on back of queue
- **Two kinds of pops**
  - Blocking – `pop(T&)`
  - non-blocking – `pop_if_present(T&)`
- **Method** `size()` returns signed integer
  - If `size()` returns `–n`, it means `n` pops await corresponding pushes
- **Method** `empty()` returns `size() == 0`
  - Difference between pushes and pops
  - May return true if queue is empty, but there are pending `pop()`
Concurrent Queue Container Example

Simple example to enqueue and print integers

Constructor for queue

Push items onto queue
- Pop item off
- Print item

```cpp
#include "tbb/concurrent_queue.h"
#include <stdio.h>
using namespace tbb;

int main ()
{
    concurrent_queue<int> queue;
    int j;

    for (int i = 0; i < 10; i++)
        queue.push(i);

    while (!queue.empty()) {
        queue.pop(&j);
        printf("from queue: %d\n", j);
    }
    return 0;
}
```
**Concurrent Vector Container**

**concurrent_vector**<T>

- Dynamically growable array of T
  - Method `grow_by(size_type delta)` appends *delta* elements to end of vector
  - Method `grow_to_at_least(size_type n)` adds elements until vector has at least *n* elements
  - Method `size()` returns the number of elements in the vector
  - Method `empty()` returns `size() == 0`
- Never moves elements until cleared
  - Can concurrently access and grow
  - Method `clear()` is not thread-safe with respect to access/resizing
Concurrent Vector Container Example

```c++
void Append( concurrent_vector<char>& V, const char* string )
{
    size_type n = strlen(string)+1;
    memcpy( &V[V.grow_by(n)], string, n+1 );
}
```

Append a string to the array of characters held in concurrent_vector
Grow the vector to accommodate new string
  • grow_by() returns old size of vector (first index of new element)
Copy string into vector
Concurrent Hash Table Container

concurrent_hash_map<Key, T, HashCompare>

- Maps Key to element of type T
- You define class HashCompare with two methods
  - hash() maps Key to hashcode of type size_t
  - equal() returns true if two Keys are equal
- Enables concurrent find(), insert(), and erase() operations
  - find() and insert() set “smart pointer” that acts as lock on item
    - accessor grants read-write access
    - const_accessor grants read-only access
  - lock released when smart pointer is destroyed
User-defined method `hash()` takes a string as a key and maps to an integer

User-defined method `equal()` returns `true` if two strings are equal

```cpp
struct MyHashCompare {
    static size_t hash( const string& x ) {
        size_t h = 0;
        for( const char* s = x.c_str(); *s; s++ )
            h = (h*157)^*s;
        return h;
    }
    static bool equal( const string& x, const string& y ) {
        return strcmp(x, y) == 0;
    }
};
```
If `insert()` returns `true`, new string insertion
- Value is key’s place within sequence of strings from `getNextString()`
Otherwise, string has been previously seen
Concurrent Hash Table Container Example Key Find

```cpp
myHash  table;
string s1, s2;
int p1, p2;
...
{
    myHash::const_accessor a;  // read_lock
    myHash::const_accessor b;
    if (table.find(a,s1) && table.find(b,s2)) {  // find strings
        p1 = a->second; p2 = b->second;
        if (p1 < p2)
            printf("%s came before %s\n",s1,s2);
        else
            printf("%s came before %s\n",s2,s1);
    }
    else printf("One or both strings not seen before\n");
}
```

If `find()` returns `true`, key was found within hash table
Scalable Memory Allocation
Serial memory allocation can easily become a bottleneck in multithreaded applications
- Threads require mutual exclusion into shared heap

False sharing - threads accessing the same cache line
- Even accessing distinct locations, cache line can ping-pong

Intel® Threading Building Blocks offers two choices for scalable memory allocation
- Similar to the STL template class `std::allocator`
- `scalable_allocator`
  - Offers scalability, but not protection from false sharing
  - Memory is returned to each thread from a separate pool
- `cache_aligned_allocator`
  - Offers both scalability and false sharing protection
Methods for `scalable_allocator`

```cpp
#include "tbb/scalable_allocator.h"

template<typename T> class scalable_allocator;

Scalable versions of malloc, free, realloc, calloc

- void *scalable_malloc( size_t size );
- void scalable_free( void *ptr );
- void *scalable_realloc( void *ptr, size_t size );
- void *scalable_calloc( size_t nobj, size_t size );

STL allocator functionality

- T* A::allocate( size_type n, void* hint=0 )
  - Allocate space for n values
- void A::deallocate( T* p, size_t n )
  - Deallocate n values from p
- void A::construct( T* p, const T& value )
- void A::destroy( T* p )
```
Scalable Allocators Example

```cpp
#include "tbb/scalable_allocator.h"
typedef char _Elem;
typedef std::basic_string<_Elem,
    std::char_traits<_Elem>,
    tbb::scalable_allocator<_Elem>> MyString;

int *p;
MyString str1 = "qwertyuiopasdfghjkl";
MyString str2 = "asdfghjklasdfghjkl";
p = tbb::scalable_allocator<int>().allocate(24);
```
#include "tbb/scalable_allocator.h"
typedef char _Elem;
typedef std::basic_string<_Elem, std::char_traits<_Elem>,
    tbb::scalable_allocator<_Elem>> MyString;

Use TBB scalable allocator for STL basic_string class
Scalable Allocators Example

```cpp
#include "tbb/scalable_allocator.h"
typedef char _Elem;
typedef std::basic_string<_Elem,
    std::char_traits<_Elem>,
    tbb::scalable_allocator<_Elem>> MyString;
...
{
    ...
    int *p;
    MyString str1 = "qwertyuiopasdfghjkl";
    MyString str2 = "asdfghjklasdfghjkl";
    p = tbb::scalable_allocator<int>().allocate(24);
    ...
}
```

Use TBB scalable allocator to allocate 24 integers
Wrapup

Thursday, December 1,
One last question...
One last question...

How do I know how many threads are available?
How do I know how many threads are available?

Do not ask!

- Not even the scheduler knows how many threads really are available
  - There may be other processes running on the machine
- Routine may be nested inside other parallel routines

Focus on dividing your program into tasks of sufficient size

- Task should be big enough to amortize scheduler overhead
- Choose decompositions with good depth-first cache locality and potential breadth-first parallelism

Let the scheduler do the mapping
**Intel® Threading Building Blocks**

is a parallel programming model for C++ applications

- Used for computationally intense code
- A focus on data parallel programming
- Uses generic programming

- Intel® Threading Building Blocks provides
  - Generic parallel algorithms
  - Highly concurrent containers
  - Low-level synchronization primitives
  - A task scheduler that can be used directly