Session:

Threading for Performance with Intel® Threading Building Blocks
Objectives

At the end of this session, you will be able to:

• List the different components of the Intel Threading Building Blocks library

• Code parallel applications using TBB
Agenda

Intel® Threading Building Blocks background

Parallel Algorithms
  parallelInvoke
  parallelFor
  parallelReduce
  parallelSort example
  Task Scheduler

Concurrent Containers

Scalable Memory Allocation

Low-level Synchronization Primitives
TBB Background
### Intel® Threading Building Blocks (TBB) - Key Features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
</table>
| **High-level Abstraction** | • specify *tasks* instead of threads  
• Library maps your logical tasks onto physical threads, efficiently using cache and balancing load  
• Full support for *nested parallelism* |
| **Scalable**   | • *scalable data parallel* programming  
• Loop parallelism tasks are more scalable than a fixed number of separate tasks |
| **Portable**   | • Portable across Linux*, Mac OS*, Windows*, and Solaris*  
• Open source and licensed versions available |
| **Compatible** | • Can be used in concert with native threads and OpenMP |
Limitations

Intel® TBB is not intended for

- I/O bound processing

- Hard real-time processing
  - TBB unfair scheduler may pre-empt real-time task
  - Underlying Threading runtime is not validated for hard-time (true of most threading runtime libraries)

Would make interesting Research area
Components of TBB (version 2.2)

**Generic Parallel Algorithms**
- `parallel_for`, `parallel_for_each`*
- `parallel_reduce`
- `parallel_scan`
- `parallel_do`
- `pipeline`
- `parallel_sort`
- `parallel_invoke`*

**Synchronization Primitives**
- `atomic`, `mutex`, `recursive_mutex`
- `spin_mutex`, `spin_rw_mutex`
- `queuing_mutex`, `queuing_rw_mutex`
- `null_mutex`*, `null_rw_mutex`*

**Concurrent Containers**
- `concurrent_hash_map`
- `concurrent_queue`
- `concurrent_vector`

**Memory Allocation**
- `tbb_allocator`
- `cache_aligned_allocator`
- `scalable_allocator`

**Task scheduler**
- `task_group`*
- `task`
- `task_scheduler_init`
- `task_scheduler_observer`

*Intel® TBB 2.2 features available in latest 2.1 open source release.
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
TBB Parallel Algorithms
Generic Programming

Best known example is C++ STL
Enables distribution of broadly-useful high-quality algorithms and data structures
Write best possible algorithm with fewest constraints
  • Do not force particular data structure on user
  • Classic example: STL std::sort
Instantiate algorithm to specific situation
  • C++ template instantiation, partial specialization, and inlining make resulting code efficient

Standard Template Library, overall, is not thread-safe
The compiler creates the needed versions

```cpp
template <typename T> T max (T x, T y) {
    if (x < y) return y;
    return x;
}

int main() {
    int i = max(20, 5);
    double f = max(2.5, 5.2);
    MyClass m = max(MyClass("foo"), MyClass("bar"));
    return 0;
}
```

**WARNING: This code does not compile!**

T must define a copy constructor and a destructor

T must define operator<
// function template
#include <iostream>
using namespace std;

//template <class T> T GetMax (T a, T b)
{ T result;
  result = (a>b)? a : b; return (result);
}

int main ()
{ int i=5, j=6, k; long l=10, m=5, n;
  k=GetMax<int>(i,j);
  n=GetMax<long>(l,m);
  cout << k << endl;
  cout << n << endl;
  return 0;
}
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
- **parallel_invoke**: used to run several functions in parallel
- **Parallel_do**: similar to parallel_while, but preferred
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.
Some programming notes
C++0x Lambda Expressions

Part of C++0x draft - implemented in Intel® C++ Compiler 11.0.

- Linux* OS: -std=c++0x
- Windows* OS: /Qstd:c++0x

Creates functor that captures local variables.

```cpp
float Example() {
    float a[4] = {1, 3, 9, 27};
    float sum = 0;
    ForEach( 0, 4, [&]( int i ) {sum += a[i];} );
    return sum;
}
```

- `&` introduces lambda expression that constructs instance of functor.
- Compiler automatically defines custom functor type tailored to capture `sum` and `a`.
- Parameter list and body for `functor::operator()`
This **lambda** function

\[
\text{[](int x) \{return x\%3==0;\}}
\]

Is the equivalent of the expression **unique()**, where **unique** is a secret identifier generated by compiler:

```
class unique
{
    public:
        bool operator() (int x) const {return x\%3==0;}
};
```
Lambda Syntax

\[ \text{[capture\_mode]} \ (\text{formal\_parameters}) \rightarrow \text{return\_type} \ \{\text{body}\} \]

- \[&\] \Rightarrow \text{by-reference}
- \[=\] \Rightarrow \text{by-value}
- [] \Rightarrow \text{no capture}

Can omit if there are no parameters and return type is implicit.
Can omit if return type is void or code is “return expr;”

**Examples**

\[&\](float x) \ { \text{sum}+=x; } \]

\[[]\]{\text{return rand();} }\]

\[&\]{\text{return } *p++; }\]

\[=[]\](float x) \ { \text{return a*x+b; } }\]

\[=[]\](float x, float y)->float \{ 
  \text{if(x<y) return x;}
  \text{else return y;}
\}
Compiler creates a unique *anonymous* functor type for each lambda expression. Therefore lambda expressions are generally useful only as arguments to template functions or with C++0x auto keyword.

```cpp
template<typename F>
void Eval( const F& f ) {
    f();
}

void Example1() {
    Eval( []{printf("Hello, world\n");} );
}

void Example2() {
    auto f = []{printf("Hello, world\n");} ;
    f();
}
```

Template deduces functor’s type instead of specifying it.

Expression `[]{...}` has anonymous type.

Compiler deduces type of `f` from right side expression. Implemented in Intel® C++ Compiler 11.0.
Some miscellaneous programming notes ...

- Some programmers prefer to use individual headers
  e.g. `#include "tbb/parallel_reduce"
  #include "tbb/parallel_for"
  Rather than
  `# include "tbb/tbb.h"

- Scheduler is automatically initialised by including headers.
  - Can be initialised manually with `task_scheduler_init`
  - Control construction & destruction
  - Specify num of threads
  - Specify stack size

- Avoid using `parallel_while` – use `parallel_do` instead

- Define `TBB_USE_THREADING_TOOLS` when using
  - Intel® Parallel Inspector, Intel® Parallel Amplifier,
  - Intel® Thread Checker or Intel® Thread Profiler
Parallel Invoke
Parallel_invoke

- Simple way to run several functions in parallel

**Function pointer method**

```cpp
void func1() {std::cout << “Func1!”;}  
void func2() {std::cout << “Func2!”;}  
void func3() {std::cout << “Func3!”;}  
parallel_invoke (func1(),func2(),func3());
```

**Using Lambda functions**

```cpp
parallel_invoke ( [] {std::cout << “Func1!”;},  
                [] {std::cout << “Func2!”;},  
                [] {std::cout << “Func3!”;});
```

**void function no return**

Max 10 functions
Using Objects

class MyClass {
    char * Name;
public:
    void operator() const {std::cout << Name;}
    MyClass (char * pName):Name(pName) {};
};

MyClass C1("Func1!");
MyClass C2("Func2!");
MyClass C3("Func3!");

parallel_invoke (C1, C2, C3);
Activity 1: Using parallel_invoke

Take and existing serial program

Convert it to use parallel_invoke

- C Functions
- Lambda functions
- Function Objects
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
- A body type that operates on the range (or a subrange)
Range – essential member items

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
Two choices for implementing code

- **Body Objects**
  - If using a compiler that does not support C++0x
  - More complicated than Lambda Functions
    - A distinct class
    - Must define a copy constructor and a destructor
    - Defines `operator()`

- **Lambda Functions**
  - C++0x standard
  - Much easier than Body Objects
  - Compiler does all the hard work
  - Aka ‘Anonymous functions’
Body – essential member items

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
An Example using parallel_for

Original code: Independent iterations and fixed/known bounds

```c
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

Add headers & namespace

```cpp
#include "tbb/tbb.h"
using namespace tbb;

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);
    parallel_change_array(A, N);
    return 0;
}
```

- Include Library Headers
- Scheduler automatically initialized
- Use namespace

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

Original change_array

```c++
#include "tbb/tbb.h"
using namespace tbb;
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);
    parallel_change_array(A, N);
    return 0;
}
```
An Example using parallel_for - Body Object

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
An Example using parallel_for - Lambda Func

Using Lambda functions

```c
void parallel_change_array(float *array, int M) {
    parallel_for (blocked_range <int>(0, M, IdealGrainSize),
                  [=](const blocked_range<int> &r){
        for (int i = r.begin(); i != r.end(); i++) {
            array[i] *= 2;
        }
    });
}
```

blue = original code
green = provided by TBB
red = boilerplate for library
How splitting works on **blocked\_range2d**

- tasks available to be scheduled to other threads (thieves)
Activity 2: Matrix Multiplication

Convert serial matrix multiplication application into parallel application using parallel_for

- Triple-nested loops

Hint: Stages of Development

1. Either
   - Create a new class
     - Constructor
     - Add Body to
   - Or Use a Lambda Function

2. Add Parallel_for to Parallel_MxM
Parallel Reduce
Reduction is ...

Two versions in TBB

- Functional form
  - For use with Lambda expressions

- Imperative form
  - Minimize copying of data
The parallel_reduce - function

return = parallel_reduce (range, identity, func, reduction);

Reduces by applying \textit{func} to subranges in \textit{range} using the binary operator \textit{reduction}. Returns the result of the reduction.

<table>
<thead>
<tr>
<th>Range</th>
<th>Range that will be split</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identity</td>
<td>Init value for reduction variable</td>
</tr>
<tr>
<td>Func</td>
<td>Does the calculations</td>
</tr>
<tr>
<td>Reduction</td>
<td>Merge two results.</td>
</tr>
</tbody>
</table>
The parallel_reduce Template

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

Requirements for parallel_reduce Body

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Body::Body( const Body&amp;, split )</td>
<td>Splitting constructor</td>
</tr>
<tr>
<td>Body::~Body()</td>
<td>Destructor</td>
</tr>
<tr>
<td>void Body::operator() (Range&amp; subrange) const</td>
<td>Accumulate results from subrange</td>
</tr>
<tr>
<td>void Body::join( Body&amp; rhs );</td>
<td>Merge result of rhs into the result of this.</td>
</tr>
</tbody>
</table>

Reuses Range concept from parallel_for
Serial Example

// Find index of smallest element in a[0...n-1]
long SerialMinIndex ( const float a[], size_t n )
{
    float value_of_min = FLT_MAX;
    long index_of_min = -1;
    for( size_t i=0; i<n; ++i ) {
        float value = a[i];
        if( value<value_of_min ) {
            value_of_min = value;
            index_of_min = i;
        }
    }
    return index_of_min;
}
Serial Example – what I would like ...

```c
// Find index of smallest element in a[0...n-1]
long SerialMinIndex ( const float a[], size_t n )
{
    float value_of_min = FLT_MAX;
    long index_of_min = -1;
    for( size_t i=0; i<n; ++i ) {
        float value = a[i];
        if( value<value_of_min ) {
            value_of_min = value;
            index_of_min = i;
        }
    }
    return index_of_min;
}
```
#include "tbb/tbb.h"
using namespace tbb;

long SerialMinIndex ( const float a[], size_t n ) {
    parallel_reduce(blocked_range<int>(0,n),
        0L, 
        [&](blocked_range<int>& r, long index_of_min)->long{
            float value_of_min = a[index_of_min];
            for(int i = r.begin(); i < r.end(); ++i){
                float value = a[i];
                if( value < value_of_min ) {
                    value_of_min = value;index_of_min = i;}
            }
            return index_of_min;
        },
        [&](long l, long r)->long { return a[l]<a[r] ? l : r;}
    );
}

Iteration space
Return type
Variable to hold intermediate results
Identity element - Initializes both TYPE and VALUE
Reduction functor

blue = original code
green = provided by TBB
red = boilerplate for library
class MinIndexBody {
    const float *const my_a;

public:
    float value_of_min;
    long index_of_min;

    MinIndexBody ( const float a[] ) :
        my_a(a),
        value_of_min(FLT_MAX),
        index_of_min(-1)
    {
    }

};

// Find index of smallest element in a[0...n-1]
long ParallelMinIndex ( const float a[], size_t n ) {
    MinIndexBody mib(a);
    parallel_reduce(blocked_range<size_t>(0,n,GrainSize), mib);
    return mib.index_of_min;
}

On next slides:
- operator()
- Split constructor
- join

blue = original code
green = provided by TBB
red = boilerplate for library
class MinIndexBody {
    const float *const my_a;
public:
    ....
    void operator()( const blocked_range<size_t>& r ) {
        const float* a = my_a;
        for( size_t i = r.begin(); i != r.end; ++i ) {
            float value = a[i];
            if( value < value_of_min ) {
                value_of_min = value;
                index_of_min = i;
            }
        }
    }
    ....
}
class MinIndexBody {
    const float *const my_a;
public:
    . . .
    MinIndexBody( MinIndexBody& x, split ) :
        my_a(x.my_a),
        value_of_min(FLT_MAX),
        index_of_min(-1)
    {}
    void join( const MinIndexBody& y ) {
        if( y.value_of_min < value_of_min ) {
            value_of_min = y.value_of_min;
            index_of_min = y.index_of_min;
        }
    }
    . . .
}
Activity 3: parallel_reduce Lab

Numerical Integration code to compute Pi
Parallel Sort Example (with work stealing)
Quicksort – Step 1

| 32 44 9 26 31 57 3 19 55 29 27 1 20 5 42 62 25 51 49 15 54 6 18 |
| 48 10 2 60 41 14 47 24 36 37 52 22 34 35 11 28 8 13 43 53 23 61 |
| 38 56 16 59 17 50 7 21 45 4 39 33 40 58 12 30 0 46 63 |

Thread 1 starts with the initial data

```
tbb::parallel_sort(color, color+64);
```
Quicksort – Step 2

THREAD 1

11 0 9 26 31 30 3 19 12 29 27 1 20 5 33 4 25 21 7
15 17 6 18 16 10 2 23 13 14 8 24 36 32 28 22 34 35

Thread 1 partitions/splits its data

THREAD 3

52 47 41 43 53 60 61 38 56 48 59 54 50
49 51 45 62 39 42 40 58 55 57 44 46 63

THREAD 2

37

THREAD 4

37

Copyright © 2008, Intel Corporation. All rights reserved.
Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States or other countries. * Other brands and names are the property of their respective owners.
QuickSort – Step 2

Thread 2 gets work by stealing from Thread 1
Quicksort – Step 3

Thread 1 partitions/splits its data

Thread 2 partitions/splits its data
**Quicksort – Step 3**

**Thread 1**

1 10 2 6 4 5 3

**Thread 3**

11 0 9 26 31 30 3 19 12 29 27 1 20 5 33 4 25 21 7 15 17 6 18 16 10 2 23 13 14 8 24 36 32 28 22 23 34 35

**Thread 2**

37 52 47 41 43 53 60 61 38 56 48 59 54 50 49 51 45 62 59 56 61 58 55 57 44 46 63

**Thread 4**

Thread 3 gets work by stealing from Thread 1

Thread 4 gets work by stealing from Thread 2
Quicksort – Step 4

**THREAD 1**

11 0 9 26 31 15 17 6 18 16 10 2 23 13 14 8 24 36 32 28 22 34 35

**THREAD 2**

52 47 41 43 53 60 61 38 56 48 59 54 50 49 51 45 62 39 42 40 58 55 57 44 46 63

**THREAD 3**

1 0 2 6 4 5 3 7 12 29 27 19 20 30 33 31 25 21 11 15 17 26 18 16 10 9 23 13 14 8 24 36 32 28 22 34 35

37

**THREAD 4**

45 47 41 43 46 44 40 38 42 48 39 49 50 52 51 54 62 59 56 61 58 55 57 60 53 63

Thread 1 sorts the rest of its data

Thread 3 partitions/splits its data

Thread 2 sorts the rest of its data

Thread 4 sorts the rest of its data
Quicksort – Step 5

Thread 1 gets more work by stealing from Thread 3

Thread 3 sorts the rest of its data

Thread 1 sorts the rest of its data

Thread 2 sorts the rest of its data

Thread 4 sorts the rest of its data
QuickSort – Step 6

Thread 1 partitions/splits its data
Quicksort – Step 6

Thread 1 sorts the rest of its data

Thread 2 gets more work by stealing from Thread 1
Quicksort – Step 7

Thread 2 sorts the rest of its data.

DONE
The Task Scheduler
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

### Problem | Intel® TBB Approach
---|---
Oversubscription | One scheduler thread per hardware thread
Fair scheduling | Non-preemptive unfair scheduling
High overhead | Programmer specifies tasks, not threads
Load imbalance | Work-stealing balances load
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);
}
```
Can envision Fibonacci computation as a task graph

```plaintext
SerialFib(4)
  SerialFib(3) SerialFib(2)
    SerialFib(2) SerialFib(1)
      SerialFib(1) SerialFib(0)
    SerialFib(2) SerialFib(1)
      SerialFib(1) SerialFib(0)
    SerialFib(3)
      SerialFib(2) SerialFib(1)
        SerialFib(1) SerialFib(0)
    SerialFib(2)
      SerialFib(1) SerialFib(0)
        SerialFib(1) SerialFib(0)
```

Example: Naive Fibonacci Calculation
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 {
            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**

Derived from TBB task class

The `execute` method does the computation of a task

Create new child tasks to compute \((n-1)\)th and \((n-2)\)th Fibonacci numbers

Spawn task; return immediately

Spawn task; block until all children have completed execution

Can be scheduled at any time

Reference count is used to know when spawned tasks have completed

E_child(): FibTask(n-1,&x);
E_child(): FibTask(n-2,&y);
Children + 1 for wait
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)
Activity 4: Task Scheduler Interface for Recursive Algorithms

Develop code to launch Intel TBB tasks to build and traverse a binary tree.
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.
Concurrence-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();
}
```

Solution: `concurrent_queue` has `pop_if_present`

At this instant, another thread might pop last element.
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

#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;
}

Constructor for queue

Push items onto queue

While more things on queue
    • Pop item off
    • Print item
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_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;
    }
};
```
Concurrent Hash Table Container Example Key Insert

typedef concurrent_hash_map<string,int,MyHashCompare> myHash;
myHash table;
string newstring;
int place = 0;
...
while (getNextString(&newString)) {
    myHash::accessor a;
    if (table.insert( a, newString )) // new string inserted
        a->second = ++place;
}

If insert() returns true, new string insertion
- Value is key’s place within sequence of strings from getNextString()
Otherwise, string has been previously seen
If `find()` returns `true`, key was found within hash table
Activity 5: Concurrent Container Lab

Use a hash table to keep track of the number of string occurrences.
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);
    ...
}
```

Use TBB scalable allocator for STL basic_string class

Use TBB scalable allocator to allocate 24 integers
Activity 6: Memory Allocation Comparison

Do scalable memory exercise that first uses “new” and then asks user to replace with TBB scalable allocator.
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)
- 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/writer (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>Method</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>

```
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_rwlock_mutex
- Non-reentrant, fair, spins in the user space

spin_rwlock_mutex
- Non-reentrant, fair, spins in the user space
- Use ReaderWriterMutex to allow non-blocking read for multiple threads

mutex
- Wrapper for OS sync: CRITICAL_SECTION for Windows*, pthread_mutex on Linux*
Reader-Writer Lock Example

```cpp
#include "tbb/spin_rwlock_mutex.h"
using namespace tbb;

spin_rwlock_mutex MyMutex;

int foo (){
    /* Construction of 'lock' acquires 'MyMutex' */
    spin_rwlock_mutex::scoped_lock lock (MyMutex, /*is_writer*/ false);

    read_shared_data (data);

    if (!lock.upgrade_to_writer ()) {
        /* lock was released to upgrade;
           may be unsafe to access data, recheck status before use */
    }
    else {
        /* lock was not released; no other writer was given access */
    }

    return 0;
    /* Destructor of 'lock' releases 'MyMutex' */
}
```
One last question...

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
**Review**

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

- **Know when to select Intel® Threading Building Blocks, the OpenMP API or Explicit Threading**