Intel® Threading Building Blocks

Michael Wrinn, Intel

ParLab Boot Camp • August 16, 2010
Family Tree

Languages
- Cilk
  - space efficient scheduler
  - cache-oblivious algorithms
- Threaded-C
  - continuation tasks
  - task stealing

Languages
- OpenMP*
  - fork/join tasks
- OpenMP taskqueue
  - while & recursion
- JSR-166 (FJTask)
  - containers

Libraries
- STL
  - generic programming
- STAPL
  - recursive ranges

Pragmas
- OpenMP*
  - fork/join tasks
- ECMA .NET*
  - parallel iteration classes

Intel® TBB 1.0
- Intel® TBB 3.0
- Microsoft® PPL

*Other names and brands may be claimed as the property of others
Key Features

Intel® Threading Building Blocks (TBB)

- It is a *template library* intended to ease parallel programming for C++ developers
  - Relies on generic programming to deliver high performance parallel algorithms with broad applicability

- It provides a *high-level abstraction* for parallelism
  - Shifts focus from workers (threads) to the work (you specify tasks patterns instead of threads)
  - Hides low level details of thread management (maps your logical tasks onto physical threads)
  - Library maps your logical tasks onto physical threads, efficiently using cache and balancing load
  - Full support for nested parallelism

- It facilitates scalable performance
  - Strives for efficient use of cache, and balances load
  - Portable across Linux*, Mac OS*, Windows*, and Solaris*
  - Emphasizes scalable data parallel programming bLoop parallelism tasks are more scalable than a fixed number of separate tasks

- Can be used in concert with other packages such as native threads and OpenMP
- Open source and licensed versions available
Limitations

- TBB is not intended for
  - I/O bound processing
  - Real-time processing

- General limitations
  - Direct use only from C++
  - Distributed memory not supported (target is desktop)
  - Requires more work than sprinkling in pragmas
**Intel® Threading Building Blocks**

- **Generic Parallel Algorithms**
  Efficient scalable way to exploit the power of multiple cores without having to start from scratch

- **Concurrent Containers**
  Common idioms for concurrent access - a scalable alternative to serial container with a lock around it

- **Thread Local Storage**
  Scalable implementation of thread-local data that supports infinite number of TLS

- **Task scheduler**
  The engine that empowers parallel algorithms that employs task-stealing to maximize concurrency

- **Synchronization Primitives**
  User-level and OS wrappers for mutual exclusion, ranging from atomic operations to several flavors of mutexes and condition variables

- **Miscellaneous**
  Thread-safe timers

- **Threads**
  OS API wrappers

- **Memory Allocation**
  Per-thread scalable memory manager and false-sharing free allocators
Tasks are light-weight entities at user-level

- 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 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.

```
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;
}
```

T must define a copy constructor and a destructor.

T must define operator<.
Generic Parallel Algorithms

Loop parallelization

- **parallel_for** and **parallel_reduce**  Load balanced parallel execution of fixed number of independent loop iterations
- **parallel_scan** Template function that computes parallel prefix ($y[i] = y[i-1] \text{ op } x[i]$)

Parallel Algorithms for Streams

- **parallel_do** Use for unstructured stream or pile of work; Can add additional work to pile while running
- **parallel_for_each**  parallel_do without an additional work feeder
- **pipeline / parallel_pipeline**
  - Linear pipeline of stages - you specify maximum number of items that can be in flight
  - Each stage can be parallel or serial in-order or serial out-of-order. Stage (filter) can also be thread-bound
  - Uses cache efficiently: Each worker thread flies an item through as many stages as possible; Biases towards finishing old items before tackling new ones

- **parallel_invoke**  Parallel execution of a number of user-specified functions
- **parallel_sort**  Comparison sort with an average time complexity $O(N \text{ Log}(N))$; When worker threads are available parallel_sort creates subtasks that may be executed concurrently
The parallel_for Template

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Body::Body(const Body&amp;)</td>
<td>Copy constructor</td>
</tr>
<tr>
<td>Body::~Body()</td>
<td>Destructor</td>
</tr>
<tr>
<td>void Body::operator() (Range&amp; subrange) const</td>
<td>Apply the body to subrange.</td>
</tr>
</tbody>
</table>

- **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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>R::R(const R&amp;)</code></td>
<td>Copy constructor</td>
</tr>
<tr>
<td><code>R::~R()</code></td>
<td>Destructor</td>
</tr>
<tr>
<td><code>bool R::is_empty() const</code></td>
<td>True if range is empty</td>
</tr>
<tr>
<td><code>bool R::is_divisible() const</code></td>
<td>True if range can be partitioned</td>
</tr>
<tr>
<td><code>R::R(R&amp; r, split)</code></td>
<td>Splitting constructor; splits <code>r</code> into two subranges</td>
</tr>
</tbody>
</table>

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

- You can define your own ranges
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;
}
Include and initialize the library

```c++
#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 (3 of 4)

- Use the `parallel_for` algorithm

```cpp
class ChangeArrayBody {
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),
                ChangeArrayBody(array), auto_partitioner());
}
```

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

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),  
        ChangeArrayBody(array),  
        auto_partitioner());
}
Parallel algorithm usage example

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

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

void ChangeArrayParallel (int* a, int n )
{
  parallel_for (blocked_range<int>(0, n), ChangeArrayBody(a));
}

int main (){
  int A[N];
  // initialize array here…
  ChangeArrayParallel (A, N);
  return 0;
}
```

ChangeArrayBody class defines a for-loop body for parallel_for

blocked_range – TBB template representing 1D iteration space

As usual with C++ function objects the main work is done inside operator()

A call to a template function parallel_for<Range, Body>:
  with arguments
  Range → blocked_range
  Body → ChangeArray

```cpp
```
C++0x Lambda Expression Support

parallel_for example will transform into:

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

void ChangeArrayParallel (int* a, int n )
{
    parallel_for (0, n, 1,
        [=](int i) {
            Foo (a[i]);
        });
}

int main (){
    int A[N];
    // initialize array here…
    ChangeArrayParallel (A, N);
    return 0;
}
```

parallel_for has an overload that takes start, stop and step argument and constructs blocked_range internally

Capture variables by value from surrounding scope to completely mimic the non-lambda implementation. Note that [&] could be used to capture variables by reference.

Using lambda expressions implement MyBody::operator() right inside the call to parallel_for().
Example using parallel_for (Game of Life)

1 of 4

Independent iterations and fixed/known bounds

```c
enum State {DEAD, ALIVE};  // cell status
typedef State **Grid;

void NextGen(Grid oldMap, Grid newMap)
{
    int row, col;
    int ncount;
    State current;

    for (row = 1; row <= MAXROW; row++)
        for (col = 1; col <= MAXCOL; col++) {
            current = oldMap[row][col];
            ncount = NeighborCount(oldMap, row, col);
            newMap[row][col] = CellStatus(current, ncount);
        }
}
```
Example using `parallel_for` (Game of Life)

2 of 4

```c++
#include “tbb/task_scheduler_init.h”
#include “tbb/blocked_range.h”
#include “tbb/parallel_for.h”

using namespace tbb;

typedef enum state {DEAD, ALIVE} State;    // cell status

typedef State Grid[MAXROW+2][MAXCOL+2];

... 

int main (int argc, char *argv[]) 
{
    task_scheduler_init init;
    ... 
}
```
class CompNextGen {
    Grid oldMap;
    Grid newMap;

public:
    CompNextGen (Grid omap, Grid nmap) :
        oldMap(omap), newMap(nmap) {}

    void operator()( const blocked_range<int>& r ) const {
        for (int row = r.begin(); row < r.end(); row++)
            for (int col = 1; col <= maxcol; col++) {
                State current = oldMap[row][col];
                int ncount = NeighborCount(oldMap, row, col);
                newMap[row][col] = CellStatus(current, ncount);
            }
    }
};

#define Task class

#define operator() method is copied from original code
Example using lambdas (Game of Life)

- Lambdas are part C++0X proposal
- Can be used in TBB if compiler supports lambdas

```cpp
void NextGen(Grid oldMap, Grid newMap)
{
    parallel_for (blocked_range<int>(1, maxrow+1),
        [&](const blocked_range<int>& r){
            for (int row = r.begin(); row < r.end(); row++)
                for (int col = 1; col <= MAXCOL; col++) {
                    State current = oldMap[row][col];
                    int ncount = NeighborCount(oldMap, row, col);
                    newMap[row][col] = CellStatus(current, ncount);
                }
        }
    );
}

auto_partitioner() is default
```

Use `&` to capture variables by reference
Task Scheduler

- Task scheduler is the engine driving Intel® Threading Building Blocks
  - Manages thread pool, hiding complexity of native thread management
  - Maps logical tasks to threads
- Parallel algorithms are based on task scheduler interface
- Task scheduler is designed to address common performance issues of parallel programming with native 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>
Two Execution Orders

Depth First
(stack)

- Small space
- Excellent cache locality
- No parallelism

Breadth First
(queue)

- Large space
- Poor cache locality
- Maximum parallelism
Work Depth First; Steal Breadth First

Best choice for theft!
• big piece of work
• data far from victim’s hot data.

Second best choice.

victim thread
How splitting works on `blocked_range2d`

tasks available to be scheduled to other threads (thieves)
Another example: Quicksort – Step 1

**THREAD 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 6 39 33 40 58 12 30 0 46 63

Thread 1 starts with the initial data
Quicksort – Step 2

Thread 1 partitions/splits its data
Quicksort – Step 2

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

**Thread 1**
- Partitions/splits its data
- Numbers: 32, 44, 9, 26, 31, 57, 3, 19, 55, 29, 27, 1, 20, 5, 42, 25, 51, 49, 15, 54, 6, 18, 48, 10, 2, 60, 41, 14, 47, 24, 36, 37, 52, 22, 34, 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 2**
- Partitions/splits its data
- Numbers: 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

Number 37 is the pivot point for partitioning.
Quicksort – Step 3

Thread 3 gets work by stealing from Thread 1

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

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 gets more work by stealing from Thread 3
Thread 1 partitions/splits its data

Quicksort – Step 6

Thread 1
1 0 2 6
4 5 3

Thread 2
52 47 41 43
53 60 61
38 56 48 59
40 58 55
57 44 46 63

Thread 3
11 8 14 13
9 10 16 12
17 15

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 splits its data
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 2 splits its data
52 47 41 43 53 60 61 38 56 48 59 40 58 55 57 44 46 63

Thread 3 splits its data
11 8 14 13 9 10 16 12 17 15

Thread 4 splits its data
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

Quicksort – Step 6

Thread 1 partitions/splits its data

Thread 2 splits its data

Thread 3 splits its data

Thread 4 splits its data

Quicksort – Step 6

Thread 1 partitions/splits its data

Thread 2 splits its data

Thread 3 splits its data

Thread 4 splits its data

Quicksort – Step 6

Thread 1 partitions/splits its data

Thread 2 splits its data

Thread 3 splits its data

Thread 4 splits its data
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 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>
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 += 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
#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);
    task_scheduler_init init;

    parallel_reduce(blocked_range<size_t>(0,num_steps), step, auto_partitioner() );

    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 += 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) {}
};
int main(int argc, char* argv[]) {
    double pi;
    double width = 1./(double)num_steps;

    pi = parallel_reduce(
        blocked_range<size_t>(0, num_steps),
        double(0),
        [&]( blocked_range<size_t>& r, double current_sum ) -> double {
            for (size_t i = r.begin(); i < r.end(); ++i) {
                double x = (i+0.5)*step;
                current_sum += 4.0/(1.0 + x*x);
            }
            return current_sum;
        },
        [](double s1, double s2) {
            return s1+s2; // "joins" two accumulated values
        });

    pi = pi *= step;
    printf("The value of PI is %15.12f\n",pi);
    return 0;
}
Task cancellation avoids unneeded work

There is a whole class of application that can benefit from ability to cancel work early
Task cancellation example

```cpp
const int problemSize = N;

int main() {
    vector<int> intVec(problemSize);
    const int valToFind = K;
    int valIdx = -1;

    parallel_for( blocked_range<int>(0, problemSize),
                  [&](const blocked_range<int>& r) {
        for( int i = r.begin(); i < r.end(); ++i ){
            if ( intVec[i] == valToFind ) {
                tbb::task::self().cancel_group_execution();
            }
        }
    });

    return 0;
}
```

When the value is found the task cancels itself and all the other tasks in the same “group” (by default these are all of the tasks of the same algorithm)
Uncaught exceptions cancel task execution

```cpp
int main() {
    try {
        parallel_for( blocked_range<int>(0, N),
                      [&](const blocked_range<int>& r) {
                          ... for (int i = r.begin(); i != r.end(); ++i) {
                              if (data[i] == bad_value)
                                  throw std::logic_error("Bad value in list");
                          }
                      });
    } catch (tbb::captured_exception& e) {
        cout << e.name() << " with description: " << e.what() << endl;
    }
    return 0;
}
```

An exception thrown from inside the task does not need to be caught in the same task. It will cancel task group execution and can be caught from outside the algorithm.

A `tbb::captured_task` can be handled in the catch block.
Improving data locality with task-to-thread affinity
(using Destroy the Castle demo as an example)

- “Destroy the Castle” computes Artificial Intelligence, Particles and Physics for moving objects for each frame by executing corresponding TBB tasks.
- Localizing computations of each type on one particular thread, but at the same time being able to balance the load is very important.

**Traditional approach to game parallelization vs. the Intel® Threading Building Blocks approach**

- Traditionally the number of threads is hardwired in the code:
  - Better data locality
  - Virtually no scalability
- Intel TBB approach is to subdivide into smaller tasks and run in parallel on an optimal number of threads
  - “Free” scalability
  - Poor data locality

```cpp
affinity_partitioner partitioner;

for ( i = 0; i < N_iter; ++i ) {
    parallel_for( blocked_range<int>(0, num_bodies),
                  AIBody(...), partitioner );
}
```

Affinity partitioner maps tasks to threads and stores “affinity” information in between iterations. This is why it has to be declared outside the nesting for-loop.
Effect of task-to-thread affinity
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
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.
Concurrent Containers Key Features

**concurrent_hash_map <Key,T,Hasher,Allocator>**
- Models hash table of std::pair <const Key, T> elements
- Maps Key to element of type T
- User defines Hasher to specify how keys are hashed and compared
- Defaults: Allocator=tbb::tbb_allocator

**concurrent_unordered_map<Key,T,Hasher,Equality,Allocator>**
- Permits concurrent traversal and insertion (no concurrent erasure)
- Requires no visible locking, looks similar to STL interfaces
- Defaults: Hasher=tbb::tbb_hash, Equality=std::equal_to, Allocator=tbb::tbb_allocator

**concurrent_vector <T, Allocator>**
- Dynamically growable array of T: grow_by and grow_to_atleast
- cache_alignedAllocator is a default allocator

**concurrent_queue <T, Allocator>**
- For single threaded run concurrent_queue supports regular “first-in-first-out” ordering
- If one thread pushes two values and the other thread pops those two values they will come out in the order as they were pushed
- cache_alignedAllocator is a default allocator

**concurrent_bounded_queue <T, Allocator>**
- Similar to concurrent_queue with a difference that it allows specifying capacity. Once the capacity is reached ‘push’ will wait until other elements will be popped before it can continue.
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
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>'x' and 'x = '</th>
<th>read/write value of x</th>
</tr>
</thead>
<tbody>
<tr>
<td>x.fetch_and_store (y)</td>
<td>z = x, x = y, return z</td>
</tr>
<tr>
<td>x.fetch_and_add (y)</td>
<td>z = x, x += y, return z</td>
</tr>
<tr>
<td>x.compare_and_swap (y,p)</td>
<td>z = x, if (x==p) x=y; return z</td>
</tr>
</tbody>
</table>

```c
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;  
}
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