Intel® Threading Building Blocks

Michael Wrinn, Intel

ParLab Boot Camp • August 19, 2009
Family Tree

Languages
- Cilk
- Threaded-C
- cache-oblivious algorithms
- task stealing

Pragmas
- OpenMP*
- fork/join tasks
- OpenMP taskqueue
- while & recursion

Libraries
- STL
- generic programming
- small tasks
- OpenMP taskqueue
- JSR-166 (FJTask) containers
- STAPL recursive ranges
- Intel® TBB 1.0
- ECMA .NET*
- parallel iteration classes
- Microsoft® PPL
- Intel® TBB 2.2

*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
Check Intel® TBB online

Active user forums, FAQs, technical blogs and TBB Developers Wiki

Several very important contributions were made by the OS community allowing TBB 2.1 to build and work on:

XBox* 360, Sun Solaris*, AIX*
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® TBB 2.2 Components

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

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

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

- Concurrency Containers
  - `concurrent_hash_map`
  - `concurrent_queue`
  - `concurrent_bounded_queue`
  - `concurrent_vector`

- Thread Local Storage
  - `combinable`
  - `enumerable_thread_specific`

- Memory Allocation
  - `tbb_allocator`
  - `zero_allocator`
  - `cache_aligned_allocator`
  - `scalable_allocator`

- Threads
  - `tbb_thread`
Task-based Programming

- 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;
}
TBB 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_do`: 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`: evaluates up to 10 functions, possibly in parallel and waits for all of them to finish.
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, \texttt{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><code>Body::Body(const Body&amp;)</code></td>
<td>Copy constructor</td>
</tr>
<tr>
<td><code>Body::~Body()</code></td>
<td>Destructor</td>
</tr>
<tr>
<td><code>void Body::operator() (Range&amp; subrange) const</code></td>
<td>Apply the body to <code>subrange</code>.</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 r into two subranges</td>
</tr>
</tbody>
</table>

- 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)
Quicksort – Step 1

Thread 1 starts with the initial data

[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 0 13 43 53 23 61 38 56 16 59 17 50 7 21 45 6 39 33 40 58 12 30 0 46 63]
Quicksort – Step 2

Thread 1 partitions/splits its data
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 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

Quicksort - Step 5
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
An Example using parallel_for (1 of 3)

- 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 (2 of 3)

- 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
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());
}
```

Define Task
Use algorithm
Use `auto_partitioner()`
An Example using parallel_for (3b of 3)

- Use the `parallel_for` algorithm

```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),
                  ChangeArrayBody(array),
                  auto_partitioner());
}
```
An Example using parallel_for with C++0x lambda functions

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

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

Use lambda function to implement MyBody::operator() inside the call to parallel_for().

Closer resemblance to sequential code
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><code>Body::Body( const Body&amp;, split )</code></td>
<td>Splitting constructor</td>
</tr>
<tr>
<td><code>Body::~Body()</code></td>
<td>Destructor</td>
</tr>
<tr>
<td><code>void Body::operator() (Range&amp; subrange) const</code></td>
<td>Accumulate results from <code>subrange</code></td>
</tr>
<tr>
<td><code>void Body::join( Body&amp; rhs );</code></td>
<td>Merge result of <code>rhs</code> 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;
}
parallel_reduce Example

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) {}    
};
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.
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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x.fetch_and_store (y)</code></td>
<td>(z = x, x = y,) 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
```c
#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
Lithe: Enabling Efficient Composition of Parallel Libraries

Heidi Pan, Benjamin Hindman, Krste Asanović

xoxo@mit.edu • {benh, krste}@eecs.berkeley.edu
Massachusetts Institute of Technology • UC Berkeley

ParLab Boot Camp • August 19, 2009
Real-World Parallel Composition Example

Sparse QR Factorization
(Tim Davis, Univ of Florida)

System Stack

Software Architecture
Out-of-the-Box Performance

Performance of SPQR on 16-core Machine

Time (sec)

Input Matrix

Out-of-the-Box

sequential
Out-of-the-Box Libraries Oversubscribe the Resources

[Diagram showing the oversubscription of resources with OTT, TBB, OpenMP, and virtualized kernel threads.]
MKL Quick Fix

Using Intel MKL with Threaded Applications
http://www.intel.com/support/performancetools/libraries/mkl/sb/CS-017177.htm

- If more than one thread calls Intel MKL and the function being called is threaded, it is important that threading in Intel MKL be turned off. Set **OMP_NUM_THREADS=1** in the environment.
Sequential MKL in SPQR
Sequential MKL Performance

Performance of SPQR on 16-core Machine

- Out-of-the-Box
- Sequential MKL

Input Matrix

- Time (sec)
- landmark
- deltaX
- ESOC
- Rucci

Input Matrix
SPQR Wants to Use Parallel MKL

No \textit{task-level} parallelism!

Want to exploit \textit{matrix-level} parallelism.
Share Resources Cooperatively

Tim Davis manually tunes libraries to effectively partition the resources.
Manually Tuned Performance

Performance of SPQR on 16-core Machine

- Out-of-the-Box
- Sequential MKL
- Manually Tuned

Bar charts showing time (sec) for different input matrices:
- landmark
- deltaX
- ESOC
- Rucci

Input Matrix
Manual Tuning Destroys Black Box Abstractions

Tim Davis

$\text{OMP\_NUM\_THREADS} = 4$

MKL
OpenMP
Manual Tuning Destroys Code Reuse and Modular Updates
Virtualized Threads are Bad

Different codes compete unproductively for resources.
Harts: Hardware Thread Contexts

- Represent real hw resources.
- Requested, not created.
- OS doesn’t manage harts for app.

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
<th>Core 2</th>
<th>Core 3</th>
<th>Core 4</th>
<th>Core 5</th>
<th>Core 6</th>
<th>Core 7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Diagram:
- SPQR
- TBB
- MKL
- OpenMP

Harts:
- Core 0
- Core 1
- Core 2
- Core 3
- Core 4
- Core 5
- Core 6
- Core 7
Hierarchical Cooperative Scheduling

- Column Elimination Tree
  - Hierarchical, Cooperative
  - TBB Sched

- Direct Control of Resources

- OS (Harts)

- Hardware

- LAPACK
- MKL
- OpenMP Sched
Analogous to function call ABI for enabling interoperable codes.

Mechanism for sharing harts, *not* policy.
SPQR with Lithe

call
req
enter
yield
enter
yield
ret

TBB_{Lithe}
MKL
OpenMP_{Lithe}

SPQR

Time
SPQR with Lithe
Performance of SPQR with Lithe

Input Matrix

- Out-of-the-Box
- Manually Tuned
- Lithe

Time (sec)
Lithe: Enabling Efficient Composition of Parallel Libraries
We would like to thank George Necula and the rest of Berkeley Par Lab for their feedback on this work.

Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). This work has also been in part supported by a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation. The authors also acknowledge the support of the Gigascale Systems Research Focus Center, one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program.