Task-based Data Parallel Programming

Asaf Yaffe
Developer Products Division
Software and Services Group

May, 2009

Agenda

• Overview
• Data Parallel Algorithms
• Tasks and Scheduling
• Synchronization and Concurrent Containers
• Summary
Multi-Core is Mainstream

• Multi-core is everywhere: desktops, laptops, and netbooks (soon)
• Gaining performance from multi-core requires parallel programming
  • The “Free Lunch” is over (!?). Serial code does not scale automatically
• Parallel programming is used for:
  • Reducing or hiding latency
  • Increasing throughput

Aspects of Parallel Programming

• Correctness: avoiding race conditions and deadlocks
• Performance: efficient use of resources
  • Multiple cores – match parallelism to hardware threads
  • Memory – choose right evaluation order, use cache effectively
• Programming with threads is difficult
  • Not automatically scalable (load balance has to be managed manually)
  • API is low level and OS-specific. Hard to come up with a clean and portable design
Parallel Decomposition

• Task-parallel: decompose a program into tasks which can be run in parallel
  + Aligns with the way we think about a program
  - Does not scale
  - Becomes complex when tasks depend on each other

• Data-parallel: decompose the problem space into chunks which can be processed in parallel
  + Parallelism is not mandatory. Easier to debug
  + Scales very well
  - Hard to program with the native threading API
  - Hard to utilize all cores effectively (load balance)

Need a high-level, portable way to express parallelism

Intel® Threading Building Blocks
Extend C++ for parallelism

Highlights
• A C++ runtime library that uses familiar task patterns, not threads
• Appropriately scales to the number of cores available
• The thread library API is portable across Linux, Windows, or Mac OS platforms. Open Source community extended support to FreeBSD, IA Solaris and XBox 360
• Run-time library decides on the optimal number of threads, tasks granularity and performance oriented scheduling
  • Automatic load balancing through tasks stealing
  • Cache efficiency and memory reusing
• Committed to:
  • Compiler independence
  • Processor independence
  • OS independence

Both a GPL and commercial license are available.
http://threadingbuildingblocks.org
Data Parallel Algorithms

Generic Parallel Algorithms

• Classic parallel programming
  – Let non-expert get scalable parallel speedup on shared-memory multi-core processor
  – Common, simple patterns
  – Coarse-grain (typically >= $10^4$ instructions per serial chunk)

• Implemented on top of work-stealing scheduler
  – Algorithms designed to be easy to use in practical ways
  – Scheduler designed for efficiency
Data Parallel Decomposition

- Serial decomposition
  + Scales with the number of hardware threads
  + Good cache locality
  - May have load-balance issues
  - May not be adequate for multi-dimensional spaces (matrices, cubes, etc...)
  - Does not fit nested parallelism

- Recursive decomposition
  + Scales with the number of hardware threads
  + Not limited to one-dimensional spaces
  + Good fit for “divide and conquer”
  + Maps to work-stealing
  - Data locality and cache issues
Recursive Decomposition

- [Data, Data+N]
- [Data, Data+N/2]
- [Data/N/2, Data+N]
- [Data, Data+GrainSize]
- [Data, Data+GrainSize/k]

Tasks available to thieves

Practical Matters

- Efficient serial snippets are the bricks of parallel programming
- Generic algorithms are the mortar!

- Concerns
  - Grain size
  - Blocking
  - Locality and affinity
**Grain Size**

- too fine ⇒ scheduling overhead dominates
- too coarse ⇒ lose potential parallelism

Tune by examining single-processor performance
- Typically adjust to lose 5%-10% of performance for grainsize=$\infty$
- 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.

**Blocking**

- Programmer’s responsibility
- Blocking calls reduce concurrency
  - Unless there’s more work to schedule
- Blocking is likely to cause a context switch
  - “lost” CPU cycles
- Synchronization activities can block
  - Many synchronization options. Need to choose the most efficient
Data Locality and Task-to-thread Affinity

Example – Game Parallelization

A typical game parallelization with native OS threads

- Render
- Physics
- Update several AI units
- Update several AI units
- ...
- Update several AI units
- Update several AI units
- Particles
Example – Game Parallelization

Task-based parallelization – data locality issues

Example – Game Parallelization

Effect of task-to-thread affinity – improved data locality
Generic Parallel Algorithms in TBB

- 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 \( y[i] = y[i-1] \oplus x[i] \)

- Parallel Algorithms for Streams
  - `parallel_do`
    - Use for unstructured stream or pile of work
    - Can add additional work to pile while running
  - `pipeline`
    - Linear pipeline of stages - you specify maximum number of items that can be in flight
    - Each stage can be serial or parallel
    - 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 Sort
  - `parallel_sort`
    - Comparison sort with an average time complexity \( O(N \log(N)) \)
    - When worker threads are available `parallel_sort` creates subtasks that may be executed concurrently

Example – Serial Version

```java
static void ChangeArraySerial( int a[], int n ) {
    for( int i=0; i<n; ++i )
        Foo(a[i]);    // Foo() operates on a single array element
}
```

Will parallelize by dividing iteration space of \( i \) into chunks
Example - Parallel Version

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

class ChangeArray
{
private:
array;

public:

ChangeArray (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), ChangeArray(a), auto_partitioner());
}

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

ChangeArray 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

Template:

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

- Requirements for Body

<table>
<thead>
<tr>
<th>Member Function</th>
<th>Requirement</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 schedules tasks to operate in parallel on subranges of the original, using available threads so that:

- Loads are balanced across the available processors
- Available cache is used efficiently
- Adding more processors improves performance of existing code (without recompilation!)
Range is Generic

• Requirements for parallel_for Range

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

• Partitioner calls splitting constructor to spread tasks over range
• Library provides blocked_range and blocked_range2d
• You can define your own ranges
• Puzzle: Write parallel quicksort using parallel_for, without recursion! (One solution is in the TBB book)

Partitioning the work

• Like OpenMP, TBB “chunks” ranges to amortize overhead
• Chunking is handled by a partitioner object
  TBB currently offers:

  > simple_partitioner takes a manual grain size
    ✓ parallel_for( blocked_range<int>(0, N, grain_size), Body() );

  > auto_partitioner heuristically adjusts grain size
    ✓ parallel_for( blocked_range<int>(0, N), Body() ,
                   auto_partitioner () );

  > affinity_partitioner tries to “replay” previous invocation of an algorithm
    ✓ Rationale and an example will follow
More on affinity_partitioner

```c
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.

Tasks and Scheduling
Task Scheduler

- Task scheduler is the engine that drives task-based programming
  - Manages thread pool hiding complexity of native threads
  - 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>Performance-biased Scheduler 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>

What is a Task

- A piece of code representing some work
  - Examples: a function, a C++ Functor, a C++ class with an “execute” method, an unnamed (Lambda) function, etc...
- CPU-bound
- Ideally uses > $10^4$ CPU cycles
Example – a TBB Task

```cpp
#include “tbb/task_scheduler_init.h”
#include “tbb/task.h”

using namespace tbb;

class ThisIsATask: public task {
public:
    task* execute () {
        WORK ();
        return NULL;
    }
};
```

- Derive from `tbb::task` class
- Implement `execute()` member function
- Create and spawn root task and your tasks
- Wait for tasks to finish

Task Dependencies

- Task dependencies define execution order
- Usually implemented as “trees”
- The “root” represents the “entry point” of execution
- Task synchronization options:
  - Explicit: a task can wait for its dependent (children) tasks to finish
  - Implicit: using continuation passing (see later)
Task Tree Example

The “wait_for_all()” call may not necessarily block the calling thread! **It blocks the task however.** A worker thread can keep stealing tasks while waiting.

Optimization: Continuation Passing

Continuation task(cont):
- Waits for its children
- Executes when all are done
- Its parent task can continue
Example: Naive Fibonacci Calculation

- Really simple (dumb?) way to calculate Fibonacci number
- But 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);
}
```

```c++
long ParallelFib( long n ) {
    long sum;
    FibTask& a = *new(tbb::task::allocate_root()) FibTask(n, &sum);
    tbb::task::spawn_root_and_wait(a);
    return sum;
}
```

```c++
class FibTask : public tbb::task {
public:
    const long n;
    long* const sum;
    FibTask( long n_, long* sum_ ) :
        n(n_), sum(sum_){}
    tbb::task* execute() { // Overrides virtual method of tbb::task
        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;
    }
};
```

Run the root task
Task entry point
When the problem is small enough, run a serial computation
Allocate child tasks
Run children and wait for them to complete
Two Execution Orders

Depth First
(stack)

Small space
Excellent cache locality
No parallelism

Breadth First
(queue)

Large space
Poor cache locality
Maximum parallelism

Work Dept First; Steal Breadth First

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

Second best choice.

victim thread
Executing and stealing tasks in TBB

Executing and stealing tasks
Executing and stealing tasks

- Task A
  - Depth: 0
- Task B
  - Depth: 1
- Task G
  - Depth: 1
- Task F
  - Depth: 2

Threads:

- Thread 1:
  - Task B
- Thread 2:
  - Task G
- Thread 3:
  - Task F
- Thread 4:
  - Task G

Depth levels:

- d = 0
- d = 1
- d = 2
- d = 3
Executing and stealing tasks

Steal from the top
Execute from the bottom
### Comparing TBB with OpenMP

<table>
<thead>
<tr>
<th></th>
<th>TBB</th>
<th>OpenMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread pool</td>
<td>One per process (singleton)</td>
<td>Created at each #pragma</td>
</tr>
<tr>
<td>Language support</td>
<td>C++</td>
<td>C, C++, Fortran</td>
</tr>
<tr>
<td>Compiler support</td>
<td>C++ library.</td>
<td>Requires compiler support</td>
</tr>
<tr>
<td>Applicability</td>
<td>Can be introduced gradually when applicable</td>
<td>Usually apply for scientific code where you have a single large loop which is an obvious candidate for parallelism</td>
</tr>
<tr>
<td>Nested Parallelism</td>
<td>Built-in</td>
<td>Hard? Not supported at all (version &lt; 3.0)?</td>
</tr>
</tbody>
</table>
| Thread Balance   | Uses task stealing. The victim is unaware of stealing so it does not suffer. Stealing itself is implemented as an atomic operation and no locks. | There are some scheduling policies available:  
  - *Simple*: range is divided equally between threads. Balance depend on the data  
  - *Dynamic*: split range into more parts than threads, and keep serving threads which finished their work.  
  - *Guided*: start with large chunks and optimize on the way  

Both Dynamic and Guided policies are using a central job manager/queue for getting work. Therefore, on many-core systems, there is a contention on the job manager - accesses to this manager MUST be serialized. This problem does not exists in TBB which uses Task Stealing and has no central job manager.

### Synchronization and Concurrent Containers
Synchronization Primitives

- Synchronization is required to produce “correct” multi-threaded code in shared-memory architectures
  - Avoiding race conditions
- Each OS provides its own set of synchronization primitives
  - Atomic operations, mutexes, semaphores, events, etc...
- OS synchronization primitives usage can be tricky
  - Selecting the appropriate primitive
  - Understanding the overhead
  - Avoiding deadlocks
  - Requires some abstraction for writing OS-independent code

TBB Synchronization Primitives

<table>
<thead>
<tr>
<th></th>
<th>Scalable</th>
<th>Fair</th>
<th>Reentrant</th>
<th>Sleeps</th>
</tr>
</thead>
<tbody>
<tr>
<td>mutex</td>
<td>OS dependent</td>
<td>OS dependent</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>spin_mutex</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>queuing_mutex</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>spin_rw_mutex</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>queuing_rw_mutex</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>recursive_mutex</td>
<td>OS dependent</td>
<td>OS dependent</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
TBB Synchronization Primitives Features

• Atomic Operations.
  – High-level abstractions

• Exception-safe Locks
  – spin_mutex is VERY FAST in lightly contended situations; use it if you need to protect very few instructions
  – Use queuing_rw_mutex when scalability and fairness are important
  – Use recursive_mutex when your threading model requires that one thread can re-acquire a lock. All locks should be released by one thread for another one to get a lock.
  – Use reader-writer mutex to allow non-blocking read for multiple threads

Example: spin_rwlock_mutex

```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);
    ...  
    if (!lock.upgrade_to_writer ()) { ... }
    else { ... }
    return 0;
    // Destructor of 'lock' releases 'MyMutex'
}
```

• If exception occurs within the protected code block destructor will automatically release the lock if it’s acquired avoiding a dead-lock

• Any reader lock may be upgraded to writer lock; upgrade_to_writer indicates whether the lock had to be released before it can upgrade
Concurrent Containers

• Programs store data in containers (vectors, lists, sets, hash tables, etc...)
• Parallel data processing requires correct concurrent access to containers
• Fine-grained locking or a lockless implementation is a key to parallel performance and scalability

Concurrent Containers – C++ STL

• STL containers are not concurrency-friendly: attempt to modify them concurrently can corrupt container
• STL containers are inherently not thread-safe
• Wrapping a lock around an STL container turns it into a serial bottleneck and still does not always guarantee thread safety
**TBB Concurrent Containers**

- 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_vector<T>**

- Dynamically growable array of T
  - grow_by(n)
  - grow_to_at_least(n)
- Never moves elements until cleared
  - Can concurrently access and grow
  - Method `clear()` is not thread-safe with respect to access/resizing

**Example**

// Append sequence [begin,end) to x in a thread-safe way.

```cpp
template<typename T>
void Append( concurrent_vector<T>& x, const T* begin, const T* end )
{
    std::copy(begin, end, x.begin() + x.grow_by(end-begin))
}
```
**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.
- Two kinds of pops
  - Blocking: `pop()`
  - Non-blocking: `pop_if_present()`
- Method `size()` returns signed integer
  - If `size()` returns \(-n\), it means \(n\) pops await corresponding pushes.
- BUT beware: a queue is cache unfriendly. A pipeline pattern might perform better...

**concurrent_hash_map<Key, T, HashCompare>**

- Associative table that maps a Key to an element of type T; holds pairs `std::pair<const Key, T>`
- `HashCompare` is a user-defined class that specifies how keys are hashed and compared
- Allows concurrent access for reads and updates
  - `bool insert(accessor &result, const Key &key)` to add or edit
  - `bool find(accessor &result, const Key &key)` to edit
  - `bool find(const_accessor &result, const Key &key)` to look up
  - `bool erase(const Key &key)` to remove
- Reader locks coexist – writer locks are exclusive
concurrent_hash_map Example

// Define hashing and comparison operations for the user type.
struct MyHashCompare {
    static long hash (const char* x) {
        long h = 0;
        for (const char* s = x; *s; s++)
            h = (h*157)^*s;
        return h;
    }

    static bool equal(const char* x, const char* y) {
        return strcmp(x,y) == 0;
    }
};

typedef concurrent_hash_map<const char*, int, MyHashCompare> StringTable;

void MyUpdateCount(const char* x) {
    StringTable MyTable;
    MyTable::accessor a;
    MyTable.insert(a, x);
    a->second += 1;
}

Summary

Multiple threads can insert and update Entries concurrently.

The accessor object acts as a smart pointer and as a writer lock.
No need for explicit locking.
Summary

• Parallel programming with low-level thread API is difficult.
• Concentrate on tasks, use scalable parallel algorithms for data decomposition
• A performance-biased scheduler is better for short, CPU-bound tasks (compared to a fair scheduler)
• Use scalable OS-independent synchronization primitives and concurrent containers

Supplementary Links

TBB Open Source Web Portal
www.threadingbuildingblocks.org

TBB Technical Articles:
"Demystify Scalable Parallelism with Intel Threading Building Block’s Generic Parallel Algorithms"
http://www.devx.com/articles/32925
"Enable Safe, Scalable Parallelism with Intel Threading Building Block’s Concurrent Containers"
http://www.devx.com/plus/Article/33334

Related Technologies:
Cilk
http://supertech.csail.mit.edu/cilk/
Microsoft Parallel Pattern Library (for native code)
Microsoft Task Parallel Library (for .NET)
Threading Building Blocks

Generic Parallel Algorithms
- parallel_for
- parallel_reduce
- parallel_do
- pipeline
- parallel_sort
- parallel_scan

Concurrent Containers
- concurrent_hash_map
- concurrent_queue
- concurrent_vector

Task scheduler
- task
- task_scheduler_init
- task_scheduler_observer

Miscellaneous
- tick_count

Synchronization Primitives
- atomic, mutex, recursive_mutex
- spin_mutex, spin_rw_mutex
- queuing_mutex, queuing_rw_mutex

Memory Allocation
- tbb_allocator; cache_aligned_allocator; scalable_allocator

Threads
- tbb_thread