Intel® Threading Building Blocks

Scalable Programming for Multi-core

Software and Solutions Group (SSG),
Developer Products Division

Arch D. Robison
Principal Engineer
Overview

Intel® Threading Building Blocks (Intel® TBB) is a C++ library that simplifies threading for performance

- Move the level at which you program from threads to tasks
- Let the run-time library worry about how many threads to use, scheduling, cache etc.
- Committed to:
  - compiler independence
  - processor independence
  - OS independence
- GPL license allows use on many platforms; commercial license allows use in products
Problem

Gaining performance from multi-core requires parallel programming

Even a simple “parallel for” is tricky for a non-expert to write well with threads.

Two aspects to parallel programming

• Correctness: avoiding race conditions and deadlock

• Performance: efficient use of resources
  – Hardware threads (match parallelism to hardware threads)
  – Memory space (choose right evaluation order)
  – Memory bandwidth (reuse cache)
Threads are Unstructured

```c
pthread_t id[N_THREAD];
for(int i=0; i<N_THREAD; i++) {
    int ret = pthread_create(&id[i], NULL, thread_routine, (void*)i);
    assert(ret==0);
}
```

```c
for( int i = 0; i < N_THREAD; i++ ) {
    int ret = pthread_join(id[i], NULL);
    assert(ret==0);
}
```

unscalable too!
Off chip memory is slow.

Cache behavior matters!

Latency $\propto$ length of arrow
Bandwidth $\propto$ width of arrow

Main Memory

(Would be 16x4 feet if 2 GB were drawn to paper scale)
Sieve of Eratosthenes for Finding Primes

Start with odd integers

3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39

Strike out odd multiples of 3

3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39

Strike out odd multiples of 5

3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39

Strike out odd multiples of 7

3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39
Running Out of Cache

Strike out odd multiples of k

Current multiple of k

cache

Each pass through array has to reload from cache!
Optimizing for Cache Is Critical

Optimizing for cache can beat small-scale parallelism

Serial Sieve of Eratosthenes

Time (sec, log scale)

n (log scale)

Plain

Restructured for Cache

7x improvement!

Intel® TBB product has example version that is restructured and parallel.
One More Problem: Nested Parallelism

Software components are built from smaller components

If each turtle specifies threads...
Effect of Oversubscription

Text filter on 4-socket 8-thread machine with dynamic load balancing

- Speedup peaked at 4 threads
- Cache sharing
- Oversubscription

**Graph Details:**
- X-axis: Logical Threads (0 to 32)
- Y-axis: Speedup (0 to 3)
- Lines:
  - Blue: Best Run
  - Black: Geomean
  - Orange: Worst Run

**Note:**
- Other brands and names are the property of their respective owners.
- Copyright © 2006, Intel Corporation. All rights reserved.
- Prices and availability subject to change without notice.
Summary of Problem

Gaining performance from multi-core requires parallel programming

Two aspects to parallel programming

• Correctness: avoiding race conditions and deadlock

• Performance: efficient use of resources
  – Hardware threads (match parallelism to hardware threads)
  – Memory space (choose right evaluation order)
  – Memory bandwidth (reuse cache)
Three Approaches for Improvement

New language
• Cilk, NESL, Fortress, ...
• Clean, simple
• Very difficult to get acceptance in field

Language extensions / pragmas
• OpenMP
• Easier to get acceptance, but require special compiler

Library
• POOMA, Hood, ...
• Works in existing environment
• Somewhat awkward
  – Syntactic boilerplate
  – Cannot rely on advanced compiler transforms for performance
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
Key Features of Intel® Threading Building Blocks

You specify *task patterns* instead of threads
- Library maps user-defined logical tasks onto physical threads, efficiently using cache and balancing load
- Full support for *nested parallelism*

Targets threading for *robust performance*
- Designed to provide portable scalable performance for computationally intense portions of shrink-wrapped applications.

*Compatible* with other threading packages
- Designed for CPU bound computation, not I/O bound or real-time.
- Library can be used in concert with other threading packages such as native threads and OpenMP.

Emphasizes *scalable, data parallel* programming
- Solutions based on functional decomposition usually do not scale.
Components

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

Concurrent Containers
- concurrent_hash_map
- concurrent_queue
- concurrent_vector

Task scheduler

Low-Level Synchronization Primitives
- atomic
- mutex
- spin_mutex
- queuing_mutex
- spin_rwlock
- queuing_rwlock

Memory Allocation
- cache_aligned_allocator
- scalable_allocator

Timing
- tick_count
Serial Example

static void SerialApplyFoo( float a[], size_t n ) {
  for( size_t i=0; i!=n; ++i )
    Foo(a[i]);
}

Will parallelize by dividing iteration space of i into chunks
Parallel Version

class ApplyFoo {
  float *const my_a;
public:
  ApplyFoo( float *a ) : my_a(a) {} 
  void operator()( const blocked_range<size_t>& range ) const {
    float *a = my_a;
    for( int i= range.begin(); i!=range.end(); ++i )
      Foo(a[i]);
  }
};

void ParallelApplyFoo(float a[], size_t n ) {
  parallel_for( blocked_range<int>( 0, n ),
                ApplyFoo(a),
                auto_partitioner() );
}
template <typename Range, typename Body, typename Partitioner>
void parallel_for(const Range& range,
                  const Body& body,
                  const Partitioner& partitioner);

Requirements for Body

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

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>Method</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()</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>

Library provides **blocked_range** and **blocked_range2d**

You can define your own ranges

Partitioner calls splitting constructor to spread tasks over range

Puzzle: Write parallel quicksort using `parallel_for`, without recursion! (One solution is in the TBB book)
How this works on `blocked_range2d`

Split range...

.. recursively...

...until $\leq$ grainsize.

tasks available to thieves
Partitioning the work

Like OpenMP, Intel TBB “chunks” ranges to amortize overhead

Chunking is handled by a partitioner object

• TBB currently offers two:
  - `auto_partitioner` heuristically picks the grain size
    - `parallel_for( blocked_range<int>(1, N), Body(), auto_partitioner());`
  - `simple_partitioner` takes a manual grain size
    - `parallel_for( blocked_range<int>(1, N, grain_size), Body());`
Grain Size

OpenMP has similar parameter

Part of `parallel_for`, not underlying task scheduler

- Grain size exists to amortize overhead, not balance load.
- Units are iterations

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

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.
- See if auto_partitioner works for your case.

too fine ⇒ scheduling overhead dominates

too coarse ⇒ lose potential parallelism

⇒ scheduling overhead dominates

⇒ lose potential parallelism
Matrix Multiply: Serial Version

void SerialMatrixMultiply( float c[M][N], float a[M][L], float b[L][N] )
{
   for( size_t i=0; i<M; ++i ) {
      for( size_t j=0; j<N; ++j ) {
         float sum = 0;
         for( size_t k=0; k<L; ++k )
            sum += a[i][k]*b[k][j];
         c[i][j] = sum;
      }
   }
}
Matrix Multiply Body for parallel_for

class MatrixMultiplyBody2D {
    float (*my_a)[L], (*my_b)[N], (*my_c)[N];
public:
    void operator()( const blocked_range2d<size_t>& r ) const {
        float (*a)[L] = my_a; // a,b,c used in example to emphasize
        float (*b)[N] = my_b; // commonality with serial code
        float (*c)[N] = my_c;
        for( size_t i=r.rows().begin(); i!=r.rows().end(); ++i )
            for( size_t j=r.cols().begin(); j!=r.cols().end(); ++j ) {
                float sum = 0;
                for( size_t k=0; k<L; ++k )
                    sum += a[i][k]*b[k][j];
                c[i][j] = sum;
            }
    }
};

MatrixMultiplyBody2D( float c[M][N], float a[M][L], float b[L][N] ) :
    my_a(a), my_b(b), my_c(c) {}
Matrix Multiply: parallel_for

#include “tbb/task_scheduler_init.h”
#include “tbb/parallel_for.h”
#include “tbb/blocked_range2d.h”

// Initialize task scheduler
tbb::task_scheduler_init tbb_init;

// Do the multiplication on submatrices of size ≈ 32x32
tbb::parallel_for ( blocked_range2d<size_t>(0, N, 32, 0, N, 32),
                     MatrixMultiplyBody2D(c,a,b) );
Future Direction

Currently, there is no affinity between separate invocations of parallel_for.

- E.g., subrange [10..20) might run on different threads each time.
- This can hurt performance of some idioms.
  - Iterative relaxation, where each relaxation is parallel.
  - Time stepping simulations on grids, where each step is parallel.
  - Shared outer-level cache lessens impact 😊

A fix being researched is something like:

```cpp
affinity_partitioner ap; // Not yet in TBB
for(;;) {
    parallel_for( body, range, ap )
}
```
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
template <typename Range, typename Body, typename Partitioner>
void parallel_reduce(const Range& range,
    const Body& body,
    const Partitioner& partitioner);

Requirements for parallel_reduce Body

<table>
<thead>
<tr>
<th>Requirement</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);</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
Lazy Parallelism

**If spare thread is available**

Body(..., split) → operator(...) → operator(...) → join()

**If no spare thread**

operator(...) → operator(...) → operator(...)
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;
}
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;
}
class MinIndexBody {
    const float *const my_a;
public:
    float value_of_min;
    long index_of_min;
    void operator()( const blocked_range<size_t>& r ) {
        const float* a = my_a;
        int end = r.end();
        for( size_t i=r.begin(); i!=end; ++i ) {
            float value = a[i];
            if( value<value_of_min ) {
                value_of_min = value;
                index_of_min = i;
            }
        }
    }
    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<x.value_of_min ) {
            value_of_min = y.value_of_min;
            index_of_min = y.index_of_min;
        }
    }
};
Parallel Algorithm Templates

Intel® TBB also provides

  parallel_scan
  parallel_sort
  parallel_while

We’re not going to cover them in as much detail, since they’re similar to what you have already seen, and the details are all in the Reference manual and Intel® TBB book if you need them.


For now just remember that they exist.
Parallel Algorithm Templates: \texttt{parallel\_scan}

Interface is similar to \texttt{parallel\_for} and \texttt{parallel\_reduce}.

Computes a parallel prefix for associative operation $\oplus$

<table>
<thead>
<tr>
<th>Input</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>a</td>
<td>a$\oplus$b</td>
<td>a$\oplus$b$\oplus$c</td>
<td>a$\oplus$b$\oplus$c$\oplus$d</td>
</tr>
</tbody>
</table>
Parallel Algorithm Templates : parallel_sort

A parallel quicksort with $O(n \log n)$ serial complexity.

- Implemented via parallel_for
- If hardware is available can approach $O(n)$ runtime.

In general, parallel quicksort outperforms parallel mergesort on small shared-memory machines.

- Mergesort is theoretically more scalable...
- ...but Quicksort has smaller cache footprint.

Cache is important!
Parallel Algorithm Templates: parallel_while

Allows you to exploit parallelism where loop bounds are not known, e.g. do something in parallel on each element in a list.

• Can add work from inside the body (which allows it to become scalable)

• It’s a class, not a function, and requires two user-defined objects
  – An ItemStream to generate the objects on which to work
  – A loop Body that acts on the objects, and perhaps adds more objects.
Parallel pipeline

Linear pipeline of stages

- You specify maximum number of items that can be in flight
- Handle arbitrary DAG by mapping onto linear pipeline

Each stage can be serial or parallel

- Serial stage processes one item at a time, in order.
- Parallel stage can process multiple items at a time, out of order.

Uses cache efficiently

- Each worker thread carries an item through as many stages as possible
- Biases towards finishing old items before tackling new ones
Parallel pipeline

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

Serial stage processes items one at a time in order.

Tag incoming items with sequence numbers

Items wait for turn in serial stage

Uses sequence numbers recover order for serial stage.

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

Serial pipeline

Throughput limited by throughput of slowest serial stage.

Another serial stage.
Concurrent Containers

Library provides highly concurrent containers

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

Library provides fine-grained locking or lockless implementations

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

Some STL interfaces are inherently not concurrency-friendly

For example, suppose two threads each execute:

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

At this instant, another thread might pop last element.

Solution: `concurrent_queue` has `pop_if_present`
concurrent_queue<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
• non-blocking

Method size() returns signed integer
• If size() returns \(-n\), it means \(n\) pops await corresponding pushes.
**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**

```cpp
// Append sequence [begin,end) to x in thread-safe way.
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_hash<Key,T,HashCompare>

Associative table 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
Example: map strings to integers

// 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;
StringTable MyTable;

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

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

Concurrent Containers
- concurrent_hash_map
- concurrent_queue
- concurrent_vector

Low-Level Synchronization Primitives
- atomic
- mutex
- spin_mutex
- queuing_mutex
- spin_rw_mutex
- queuing_rw_mutex

Task scheduler

Memory Allocation
- cache_aligned_allocator
- scalable_allocator

Timing
- tick_count
Example: Naive Fibonacci Calculation

Really 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);
}
```
long ParallelFib( long n ) {
    long sum;
    FibTask& a = *new(Task::allocate_root()) FibTask(n,&sum);
    Task::spawn_root_and_wait(a);
    return sum;
}

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

atomic<T> provides atomic operations on primitive machine types.

- fetch_and_add, fetch_and_increment, fetch_and_decrement
- compare_and_swap, fetch_and_store.

- Can also specify memory access semantics (acquire, release, full-fence)

Use atomic (locked) machine instructions if available, so efficient.

Useful primitives for building lock-free algorithms.

Portable - no need to roll your own assembler code
Example: Reference Counting

```c
struct Foo {
    atomic<int> refcount;
};

void RemoveRef( Foo& p ) {
    --p. refcount;
    if( p. refcount ==0 ) delete &p;
}

void RemoveRef(Foo& p ) {
    if( --p. refcount ==0 ) delete &p;  // WRONG! (Has race condition)
}
```

```c
void RemoveRef(Foo& p ) {
    if( --p. refcount ==0 ) delete &p;  // Right
}
```
Timing

Problem
- Accessing a reliable, high resolution, thread independent, real time clock is non-portable and complicated.

Solution
- The **tick_count** class offers convenient timing services.
  
  tick_count::now() returns current timestamp
  
  tick_count::interval_t::operator-(const tick_count &t1, const tick_count &t2)
  
  double tick_count::interval_t::seconds() converts intervals to seconds

  They use the highest resolution real time clock which is consistent between different threads.
A Non-feature: thread count

There is no function to let you discover the thread count.

You should not need to know...

• 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.

• Tasks 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.

Worry about your algorithm and the work it needs to do, not the way that happens.
Why Open Source?

Make threading ubiquitous!

• Offering an open source version makes it available to more developers and platforms quicker

Make parallel programming using generic programming techniques standard developer practice

Tap the ideas of the open source community to improve Intel® Threading Building Blocks

• Show us new ways to use it
• Show us how to improve it
A (Very) Quick Tour

Source library organized around 4 directories
- src – C++ source for Intel TBB, TBBmalloc and the unit tests
- include – the standard include files
- build – catchall for platform-specific build information
- examples – TBB sample code

Top level index.html offers help on building and porting

• Build prerequisites:
  - C++ compiler for target environment
  - GNU make
  - Bourne or BASH-compatible shell
  - Some architectures may require an assembler for low-level primitives
Correctness Debugging

Intel TBB offers facilities to assist debugging

- Debug single-threaded version first!
  
  \[
  \text{task_scheduler_init init}(1);
  \]

- Compile with macro \text{TBB\_DO\_ASSERT}=1 to enable checks in the header/inline code

- Compile with \text{TBB\_DO\_THREADING\_TOOLS}=1 to enable hooks for Intel Thread Analysis tools
  
  - Intel\textregistered\  Thread Checker can detect potential race conditions

- Link with \text{libtbb\_debug.}* to enable internal checking
Performance Debugging

- Study scalability by using explicit thread count argument
  ```c
  task_scheduler_init init(number_of_threads);
  ```
- Compile with macro `TBB_DO_ASSERT=0` to disable checks in the header/inline code
- Optionally compile with macro `TBB_DO_THREADING_TOOLS=1` to enable hooks for Intel Thread Analysis tools
  - Intel® Thread Profiler can detect bottlenecks
- Link with `libtbb.*` to get optimized library
- The `tick_count` class offers convenient timing services.
Task Scheduler

Intel TBB task interest is managed in the task_scheduler_init object

```cpp
#include "tbb/task_scheduler_init.h"
using namespace tbb;
int main() {
    task_scheduler_init init;
    ....
    return 0;
}
```

Thread pool construction also tied to the life of this object
- Nested construction is reference counted, low overhead
- Keep init object lifetime high in call tree to avoid pool reconstruction overhead
- Constructor specifies thread pool size `automatic, explicit` or `deferred`.
Summary of Intel® Threading Building Blocks

It is a library
You specify task patterns, not threads
Targets threading for robust performance
Does well with nested parallelism
Compatible with other threading packages
Emphasizes scalable, data parallel programming

Generic programming enables distribution of broadly-useful high-quality algorithms and data structures.

Available in GPL-ed version, as well as commercially licensed.
References

Intel® TBB: http://threadingbuildingblocks.org
http://www.intel.com/software/products/tbb

Cilk: http://supertech.csail.mit.edu/cilk

Parallel Pipeline: MacDonald, Szafron, and Schaeffer. “Rethinking the Pipeline as Object-Oriented States with Transformations”, Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'04).

STAPL: http://parasol.tamu.edu/stapl

Other Intel® Threading tools: Thread Profiler, Thread Checker
http://www.intel.com/software/products/threading
Supplementary Links

- **Open Source Web Site**
  - [http://threadingbuildingblocks.org](http://threadingbuildingblocks.org)
- **Commercial Product Web Page**
- **Dr. Dobb’s NetSeminar**
  - “Intel® Threading Building Blocks: Scalable Programming for Multi-Core”
- **Technical Articles:**
  - “Demystify Scalable Parallelism with Intel Threading Building Block’s Generic Parallel Algorithms”
    - [http://www.devx.com/cplus/Article/32935](http://www.devx.com/cplus/Article/32935)
  - “Enable Safe, Scalable Parallelism with Intel Threading Building Block's Concurrent Containers”
    - [http://www.devx.com/cplus/Article/33334](http://www.devx.com/cplus/Article/33334)
- **Industry Articles:**
  - Product Review: Intel Threading Building Blocks
    - [http://www.devx.com/go-parallel/Article/33270](http://www.devx.com/go-parallel/Article/33270)
  - “The Concurrency Revolution”, Herb Sutter, Dr. Dobb’s 1/19/2005