COMP 422, Lecture 10: Intel Thread Building Blocks (contd)

Vivek Sarkar

Department of Computer Science
Rice University

vsarkar@rice.edu

Recap of Lecture 9: OpenMP 3.0 tasks Example: postorder tree traversal

```
void postorder(node *p) {
    if (p->left)
      #pragma omp task
        postorder(p->left);
    if (p->right)
      #pragma omp task
        postorder(p->right);
#pragma omp taskwait // wait for child tasks
  process(p->data);
}
```

- **Parent task suspended until children tasks complete**

TBB Parallel Algorithm example: parallel_for

red = provided by TBB

```
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() );
}
```

Task

Pattern

Iteration space

Automatic grain size

Acknowledgments for today's lecture

- **Thread Building Blocks, Arch Robinson, HPCC 2007 tutorial**
  - —**http://www.tlc2.uh.edu/hpcc07/Schedule/tbBlocks**
- **"Threading for Performance with Intel Thread Building Blocks: Thinking Parallel", Victoria Gromova**
  - —**http://softwaredispatch.intel.com/?lid=1861&t=1**
- **Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism",** James Reinders, O'Reily, **First Edition, July 2007**
  - —**http://www.oreilly.com/catalog/9780596514808/**

Outline

- Parallel Algorithm Templates
- How TBB works
- Concurrent Containers
- Synchronization
- Summary: comparison of pthreads, OpenMP, TBB

Matrix Multiply: Serial Version

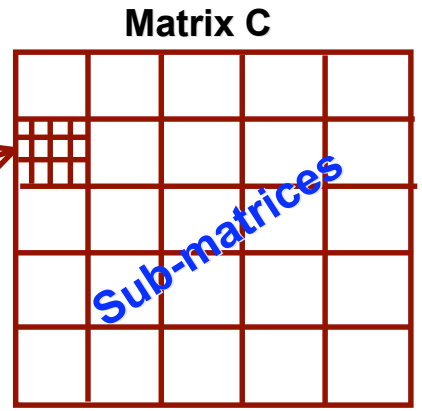
```
void SerialMatrixMultiply( float c[M][N], float a[M][L], float b[L][N] )
{
   for( size_t i=0; i<M; ++i ) {
      for( size_t j=0; j<N; ++j ) {
         float sum = 0;
         for( size_t k=0; k<L; ++k )
            sum += a[i][k]*b[k][j];
         c[i][j] = sum;
      }
   }
}
```

Matrix Multiply: 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) );
```

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) {}
};
```



Parallel_reduce

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

- **Requirements for parallel_reduce Body**

Body::Body(const Body&, split)	Splitting constructor
Body::~Body()	Destructor
void Body::operator() (Range& *subrange*);	Accumulate results from *subrange*
void Body::join(Body& *rhs*);	Merge result of *rhs* into the result of this.

- **Reuses Range concept from parallel_for**

Serial Example

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

Parallel Version (1 of 2)

green = provided by TBB

```
class MinIndexBody {
    const float *const my_a;
public:
    float value_of_min;
    long index_of_min;
    ... // Details on next slide
    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;
}
```

Parallel Version (2 of 2)

```
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;
      }
   }
   ...
};
```

accumulate result

split

join

Parallel Algorithm Templates: parallel_scan

- Interface is similar to parallel_for and parallel_reduce.
- Computes a parallel prefix for associative operation ⊕

Input	a	b	c	d
Output	a	a⊕b	a⊕b⊕c	a⊕b⊕c⊕d

Body::Body(const Body&, split)	Splitting constructor
Body::~Body()	Destructor
void Body::operator() (Range& *subrange, pre_scan_tag*);	Computes reduction
void Body::operator() (Range& *subrange, final_scan_tag*);	Computes reduction and updates result
void Body::reverse_join(Body& *rhs*);	Merge result of *rhs* into the result of this (this is right operand)

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!

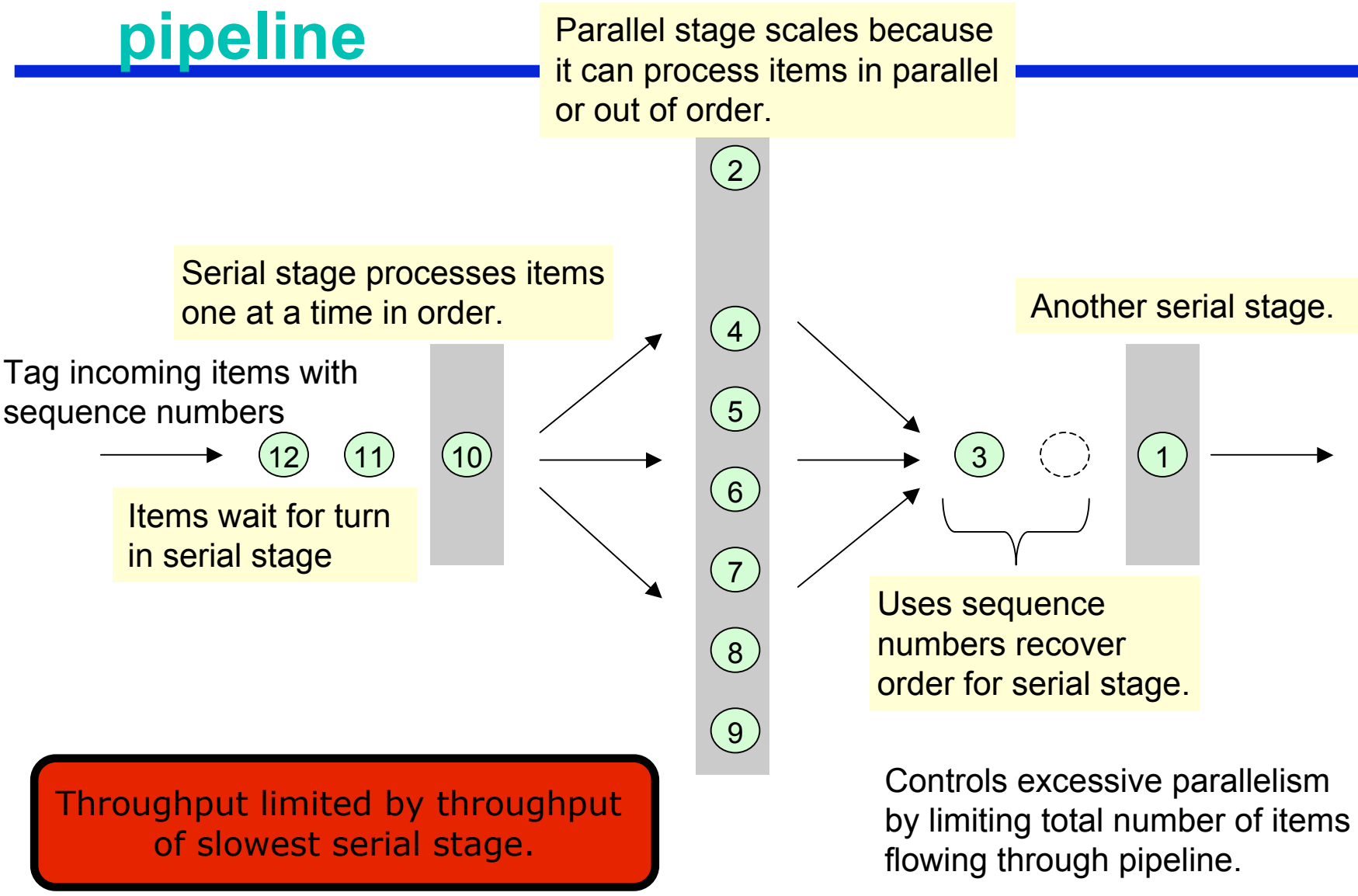
Parallel Algorithm Templates : parallel_while

- **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



Outline

- Parallel Algorithm Templates
- How TBB works
- Concurrent Containers
- Synchronization
- Summary: comparison of pthreads, OpenMP, TBB

How TBB works

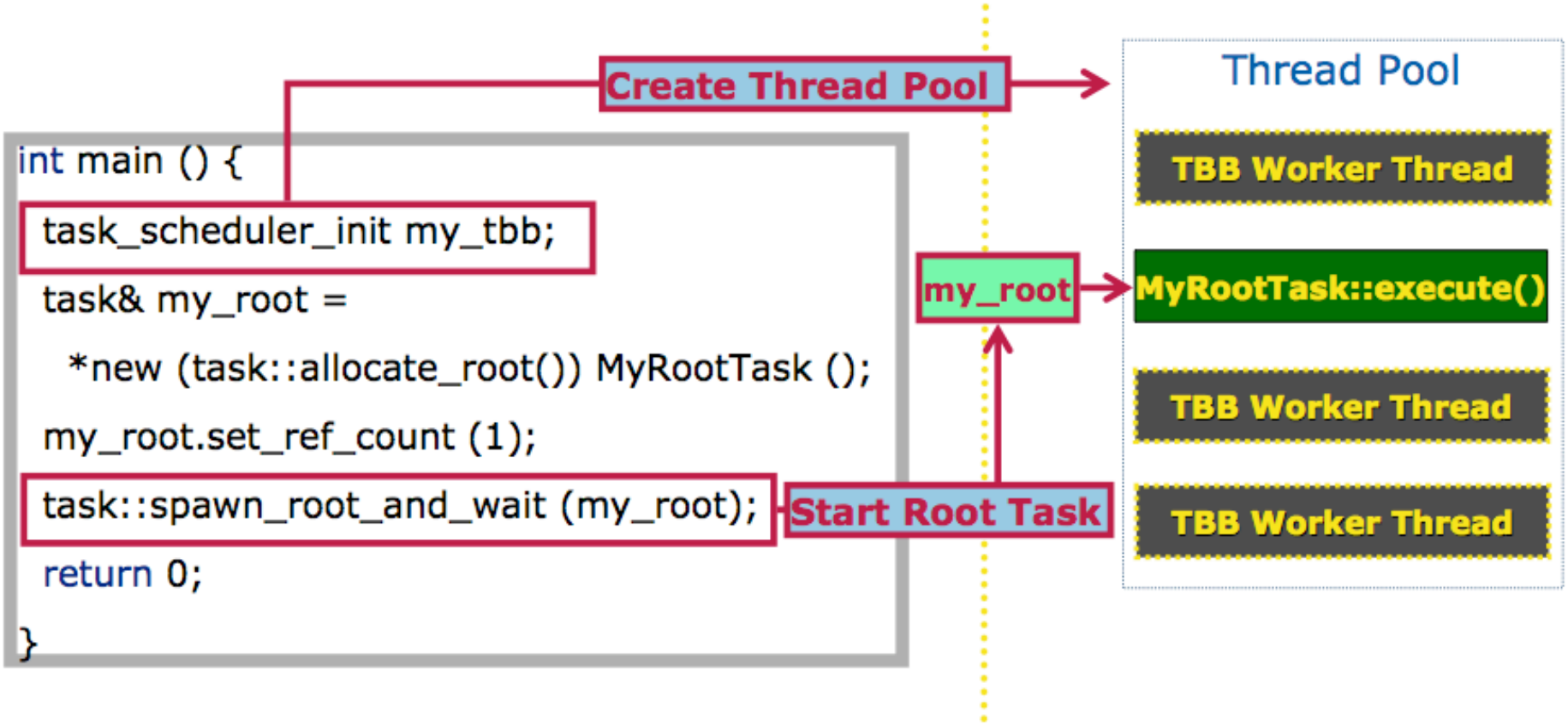
- **Task Scheduler**
- **Recursive partitioning to generate work as required**
- **Task stealing to keep threads busy**

task Scheduler

- **The engine that drives the high-level templates**
- **Exposed so that you can write your own algorithms**
- **Designed for high performance – not general purpose**

Problem	Solution
Oversubscription	**One TBB thread per hardware thread**
Fair scheduling	**Non-preemptive unfair scheduling**
High overhead	**Programmer specifies tasks, not threads**
Load imbalance	**Work-stealing balances load**
Scalability	**Specify tasks and how to create them, rather than threads**

Managing Tasks: Example



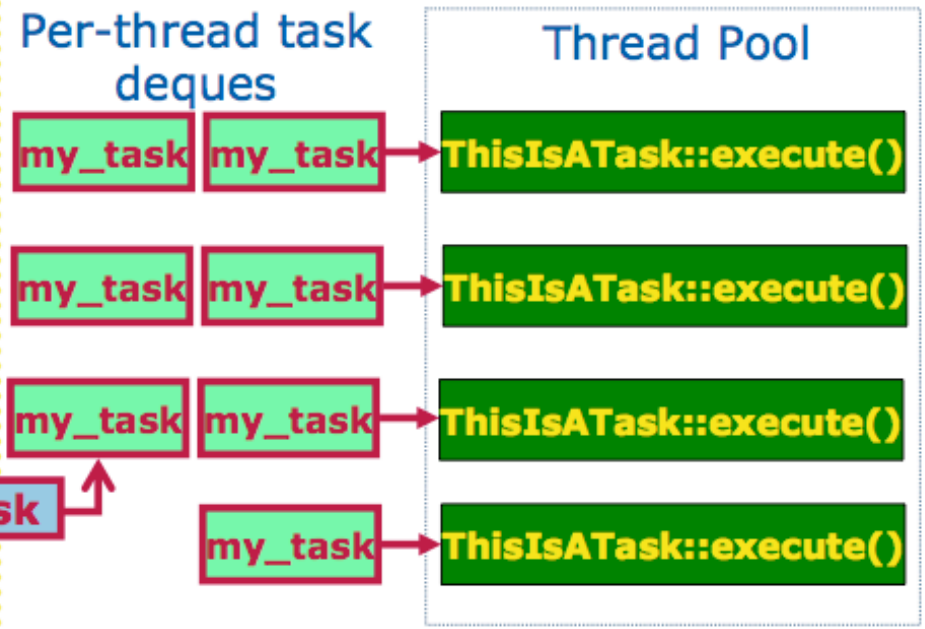
Managing Tasks: Example (contd)

Task-based App

```
class MyRootTask: public task {
public:
  task* execute () {
   for (int i=0; i<N; i++) {
     task& my_task = *new
      (task::allocate_additional_child_of
        (*this)) ThisIsATask ();
     spawn (my_task);
   }
   wait_for_all ();
   return NULL;
  }
};
```

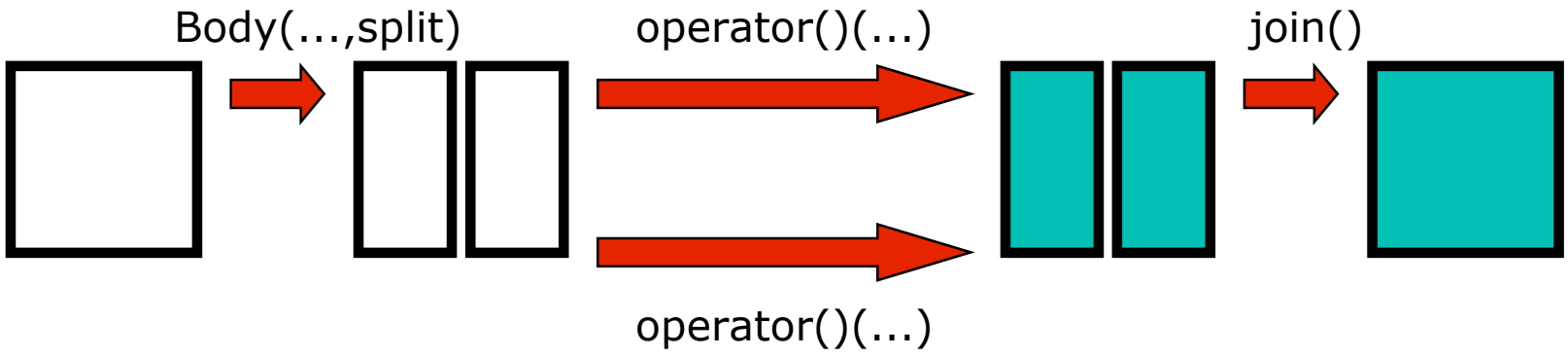
Submit Task

Task Scheduler

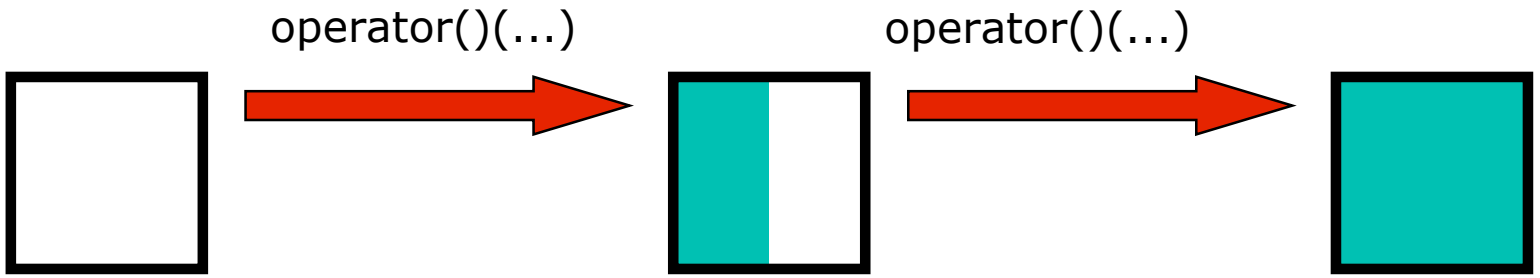


Lazy Parallelism in parallel_reduce

If a spare thread is available

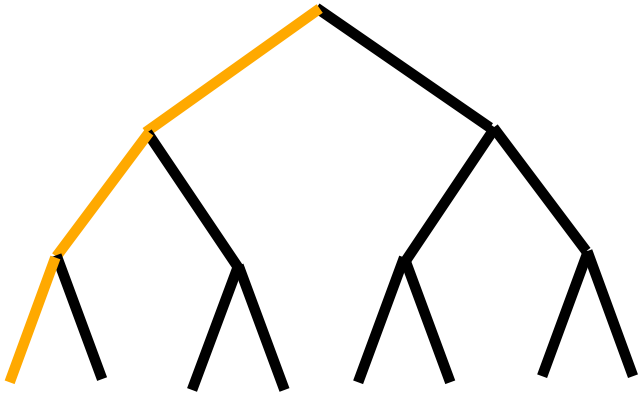


If no spare thread is available



Two Possible Execution Orders

Depth First Task Order (stack)

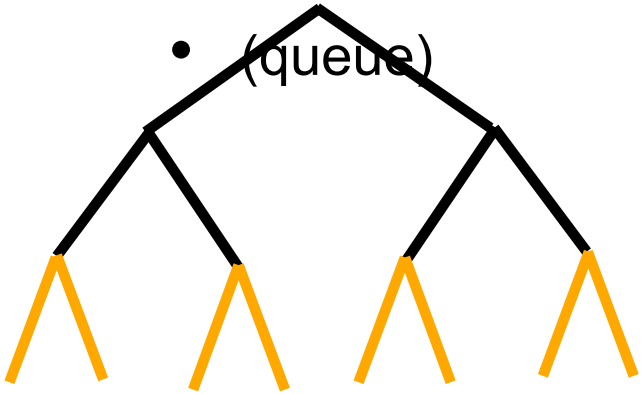


Small space

Excellent cache locality

No parallelism

- Breadth First
- Task Order
- (queue)



Large space

Poor cache locality

Maximum parallelism

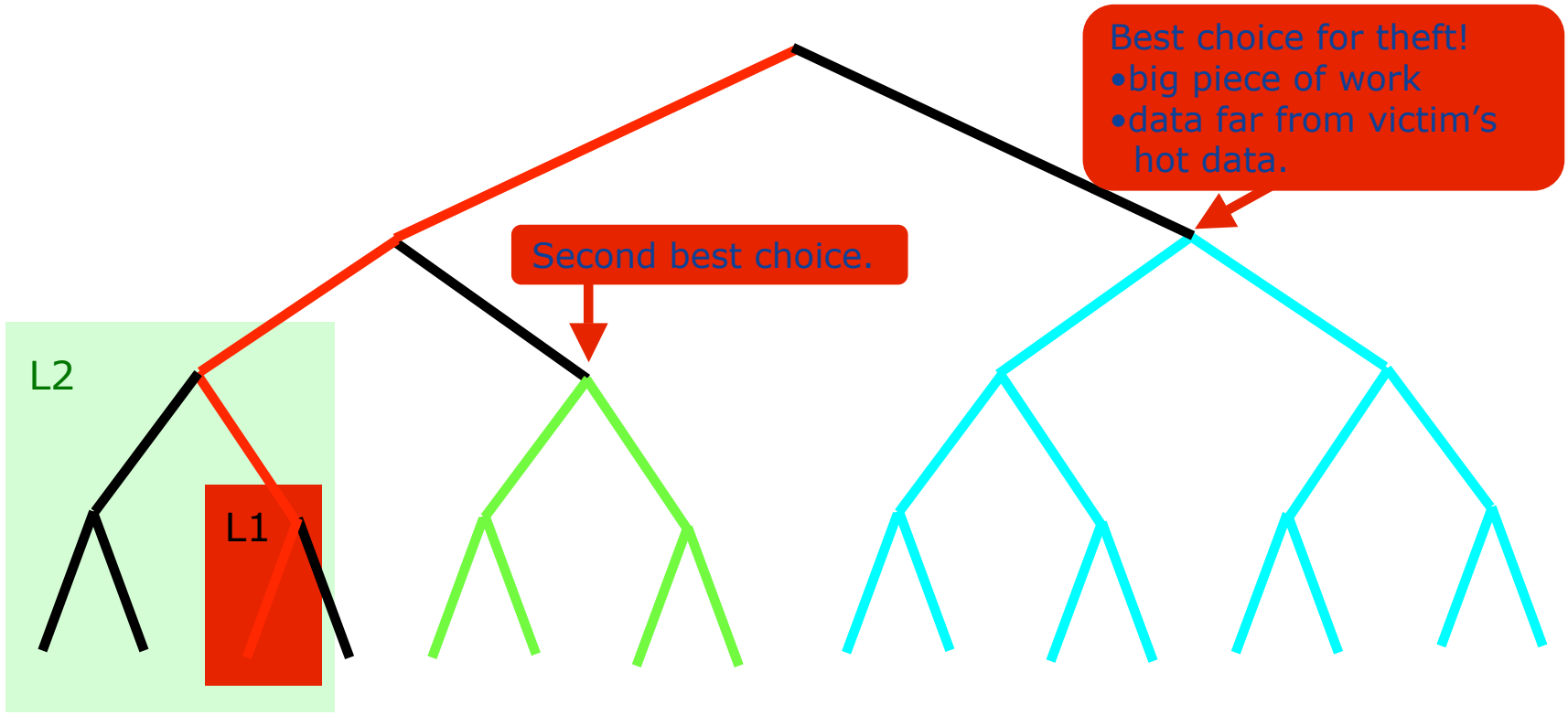
Work Stealing

- **Each thread maintains an (approximate) deque of tasks**
  - **—Similar to Cilk & Hood**
  - **A thread performs depth-first execution**
  - **—Uses own deque as a stack**
  - **—Low space and good locality**
  - **If thread runs out of work**
  - **—Steal task, treat victim's deque as queue**
  - **—Stolen task tends to be big, and distant from victim's current effort.**

Throttles parallelism to keep hardware busy without excessive space consumption.

Works well with nested parallelism

Work Depth First; Steal Breadth First



victim thread

Example: Naive Fibonacci Calculation

- **Really dumb way to calculate Fibonacci number**
- **But widely used as toy benchmark**
  - —**Easy to code**
  - —**Has unbalanced task graph**

```
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)

Outline

- Parallel Algorithm Templates
- How TBB works
- Concurrent Containers
- Synchronization
- Summary: comparison of pthreads, OpenMP, TBB

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:

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

At this instant, another thread might pop last element.

Solution: tbb::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
  - —pop (blocking)
  - —pop_if_present (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

```
// 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;
}
```

Multiple threads can insert and update entries concurrently.

accessor object acts as a smart pointer and a writer lock: no need for explicit locking.

Example: Naive Fibonacci Calculation

- **Really dumb way to calculate Fibonacci number**
- **But widely used as toy benchmark**
  - —**Easy to code**
  - —**Has unbalanced task graph**

```
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)

Outline

- Parallel Algorithm Templates
- How TBB works
- Concurrent Containers
- Synchronization
- Summary: comparison of pthreads, OpenMP, TBB

Synchronization Primitives

- Parallel tasks must sometimes touch shared data
  - —When data updates might overlap, use mutual exclusion to avoid races

- All TBB mutual exclusion regions are protected by scoped locks
  - —The range of the lock is determined by its lifetime (lexical scope)
  - —Leaving lock scope calls the destructor,
    - making it exception safe
    - you don't have to remember to release the lock on every exit path
  - —Minimizing lock lifetime avoids possible contention

- Several mutex behaviors are available
  - —Spin vs. queued ("fair" vs. "unfair")
    - spin_mutex, queuing_mutex
  - —Multiple reader/ single writer)
    - spin_rw_mutex, queuing_rw_mutex
  - —Scoped wrapper of native mutual exclusion function
    - mutex (Windows*: CRITICAL_SECTION, Linux*: pthreads mutex)

Example: spin_rw_mutex promotion

```
Set mySet;
spin_rw_mutex MyMutex;

void AddToSet ( Item x ){
    spin_rw_mutex::scoped_lock lock (MyMutex, /*is_writer*/ false);
    if( !mySet.contains(x) ) {
        if ( lock.upgrade_to_writer () || !mySet.contains(x) ) {
            mySet.add(x);
    }
    // Destructor of 'lock' releases 'MyMutex'
}
```

- **Exceptions occurring within the locked code range automatically release the lock (*lock* passes out of scope), avoiding deadlock**
- **Any reader lock may be upgraded to writer lock; upgrade_to_writer() fails if the lock had to be released before it can be locked for writing**

Thread-safe Table: Naïve Implementation

```
port_t& gateway::add_new_mapping (ip_t& ip, port_t& port, port_t& new_port) {
    port_number* mapped_port = new port_number (port);
    ip_address*  addr = new ip_address (ip, new_port);
    pthread_mutex_lock (my_global_mutex); // Protect access to std::map
    mapped_ports_table::iterator a;
    if ((a = mapped_ports.find (new_port)) == mapped_ports.end())
        mapped_ports[new_port] = mapped_port;
    else { // Re-map found port to packetAppPayload port
        delete a->second;
        a->second = mapped_port;
    }
    mapped_ports[port] = addr;
    pthread_mutex_unlock (my_global_mutex); // Release lock
    return new_port;
}
```

Naïve Implementation Problems

1. Global lock blocks the entire table. Multiple threads will wait on this lock even if they access different parts of the container
2. Some methods just need to read from the container (e.g., looking for assigned port associations). One reading thread will block other readers while it holds the mutex
3. If method has several "return" statements, developer must remember to unlock the mutex at every exit point
4. If protected code throws an exception, developer must remember to unlock mutex when handling the exception

- *2, 3, and 4 can be resolved by using tbb::spin_rw_mutex instead of pthread_mutex_t*

Thread-safe Table: tbb::spin_rw_mutex

```
#include "tbb/spin_rw_mutex.h“
tbb::spin_rw_mutex my_rw_mutex;
class port_number : public address {
    port_t port;
    port_number (port_t& _port) : port(_port) {}
public:
     bool get_ip_address (mapped_ports_table& mapped_ports, ip_t& addr) {
          // Constructor of tbb:scoped_lock acquires reader lock
          tbb::spin_rw_mutex::scoped_lock lock (my_rw_lock, /*is_writer*/ false);
          mapped_ports_table::iterator a;
          if ((a = mapped_ports.find (port)) != mapped_ports.end()) {
              return a->second->get_ip_address (mapped_ports, addr);
              // Destructor releases “lock”
          }
          return false; // Reader lock automatically released
      }
};
```

Nicer, but there is a still better way to do this…

Synchronization: Atomic

- **A better (smaller, more efficient) solution for our threadcount problem…**

```
atomic<int> tasksDone;

void doneTask(void)
{
   tasksDone++;
}
```

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**
- **Useful primitives for building lock-free algorithms.**
- Portable **- no need to roll your own assembler code**

Example: Reference Counting

```
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)

Right

Outline

- Parallel Algorithm Templates
- How TBB works
- Concurrent Containers
- Synchronization
- Summary: comparison of pthreads, OpenMP, TBB

Comparison table

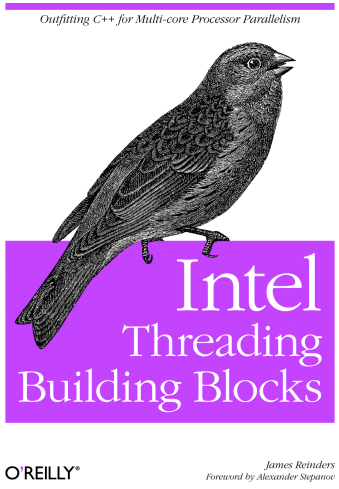
	Native Threads	OpenMP	TBB
Does not require special compiler	**Yes**	**No**	**Yes**
Portable (e.g. Windows* <-> Linux*/Unix*)	**No**	**Yes**	**Yes**
Supports loop based parallelism	**No**	**Yes**	**Yes**
Supports nested parallelism	**No**	**Maybe**	**Yes**
Supports task parallelism	**No**	**Coming soon**	**Yes**
Provides locks, critical sections	**Yes**	**Yes**	**Yes**
Provide portable atomic operations	**No**	**Yes**	**Yes**
Supports C, Fortran	**Yes**	**Yes**	**No**
Provides parallel data structures	**No**	**No**	**Yes**

Intel® Threading Building Blocks and OpenMP Both Have Niches

- **Use OpenMP if...**
  - —Code is C, Fortran, (or C++ that looks like C)
  - —Parallelism is primarily for bounded loops over built-in types
  - —Minimal syntactic changes are desired
- **Use Intel® Threading Building Blocks if..**
  - —Must use a compiler without OpenMP support
  - —Have highly object-oriented or templated C++ code
  - —Need concurrent data structures
  - —Need to go beyond loop-based parallelism
  - —Make heavy use of C++ user-defined types

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 **Open Source**
  - http://www.intel.com/software/products/tbb **Commercial**

- 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