Building High Performance Threaded Applications using Libraries
or
Why you don't need a parallel compiler

Jim Cownie
Principal Engineer

Outline

Overview of Intel®Threading Building Blocks (Intel® TBB)

Problems that Intel® TBB addresses

Origin of Intel® TBB

Parallel Algorithm Templates

BREAK

How it works

Synchronization

Concurrent Containers

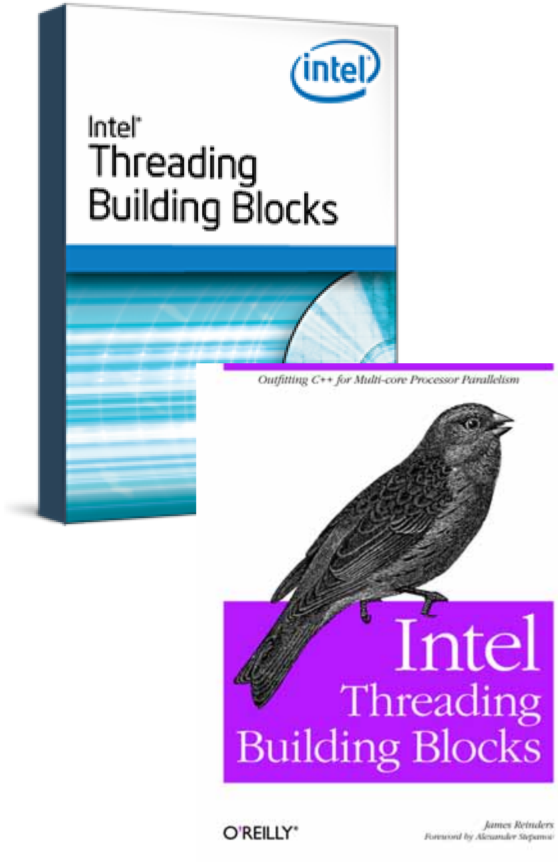
Miscellenea

When to use native threads, OpenMP, TBB

Quick overview of TBB sources

Overview

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



- Move the level at which you program above 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

Outline

Overview of Intel® Threading Building Blocks (Intel® TBB)

Problems that Intel® TBB addresses

Origin of Intel® TBB

Parallel Algorithm Templates

BREAK

How it works

Synchronization

Concurrent Containers

Miscellenea

When to use native threads, OpenMP, TBB

Quick overview of TBB sources

Problems exploiting parallelism

Gaining performance from multi-core requires parallel programming

Even a simple "parallel for" is tricky for a non-expert to write well with explicit 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

```
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);
}
```

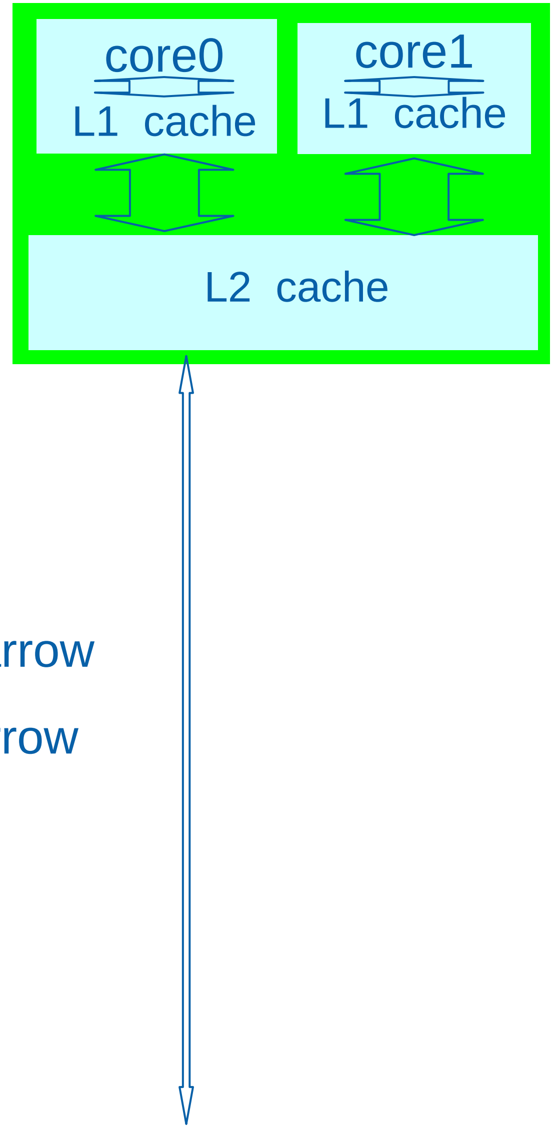
parallel goto

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

parallel comefrom

unscalable too!

Off-Chip Memory Is Slow : so cache behavior matters…



Latency ∝ length of arrow

Bandwidth ∝ width of arrow

memory

Locality Matters! Consider 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

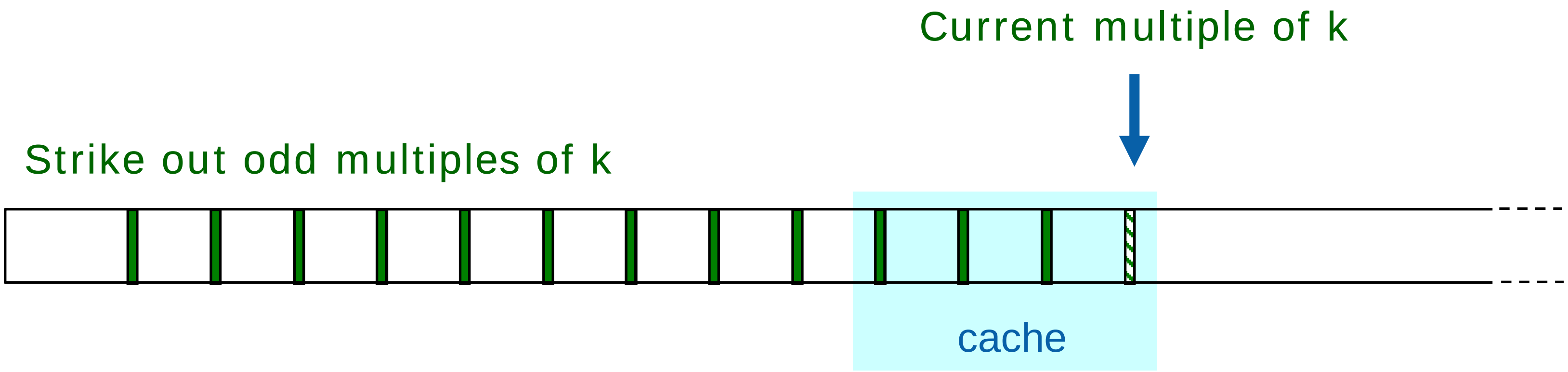
Strike out odd multiples of 5

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

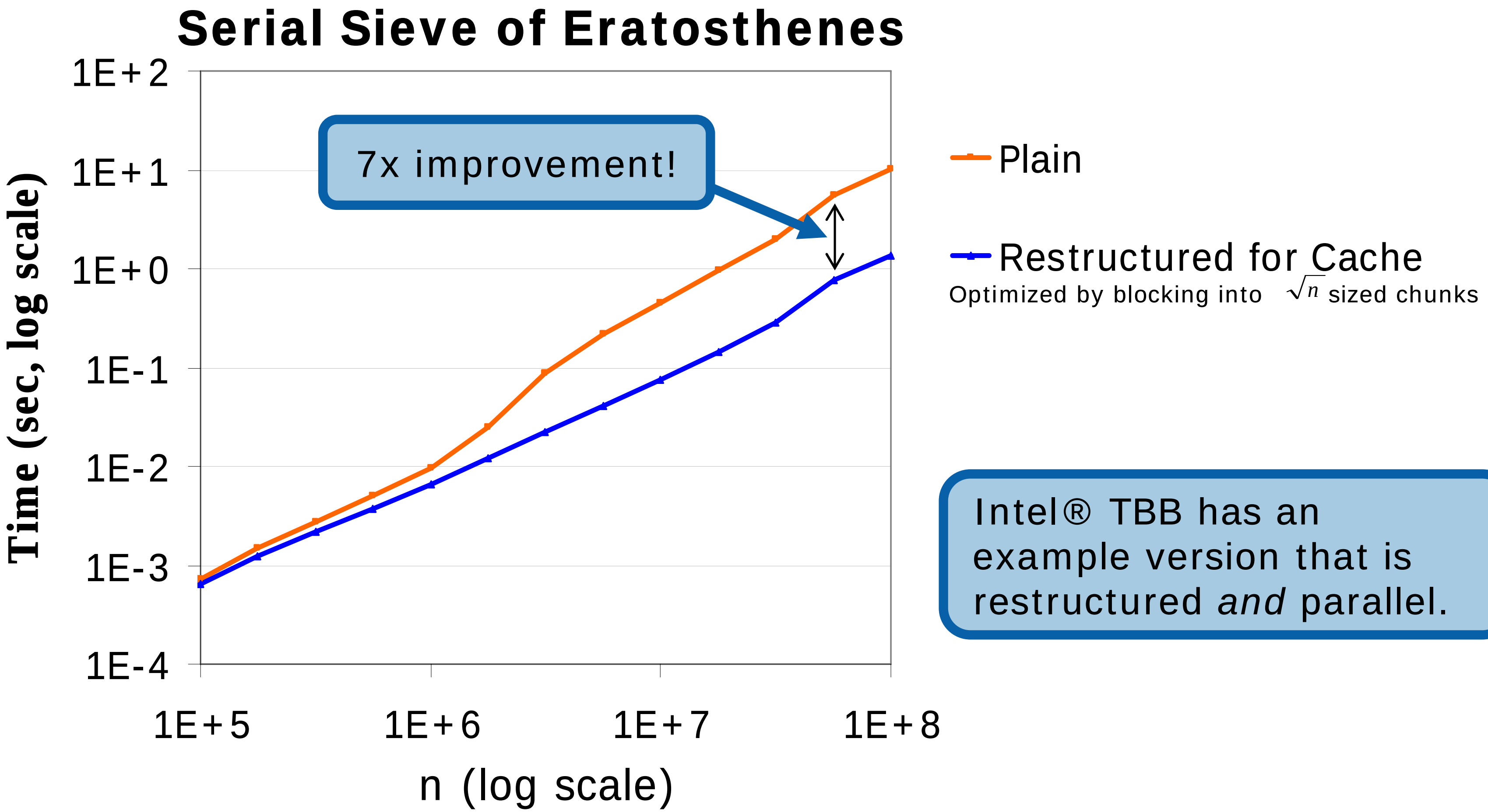
Running Out of Cache



Each pass through array has to reload the cache!

Optimizing for Cache Is Critical

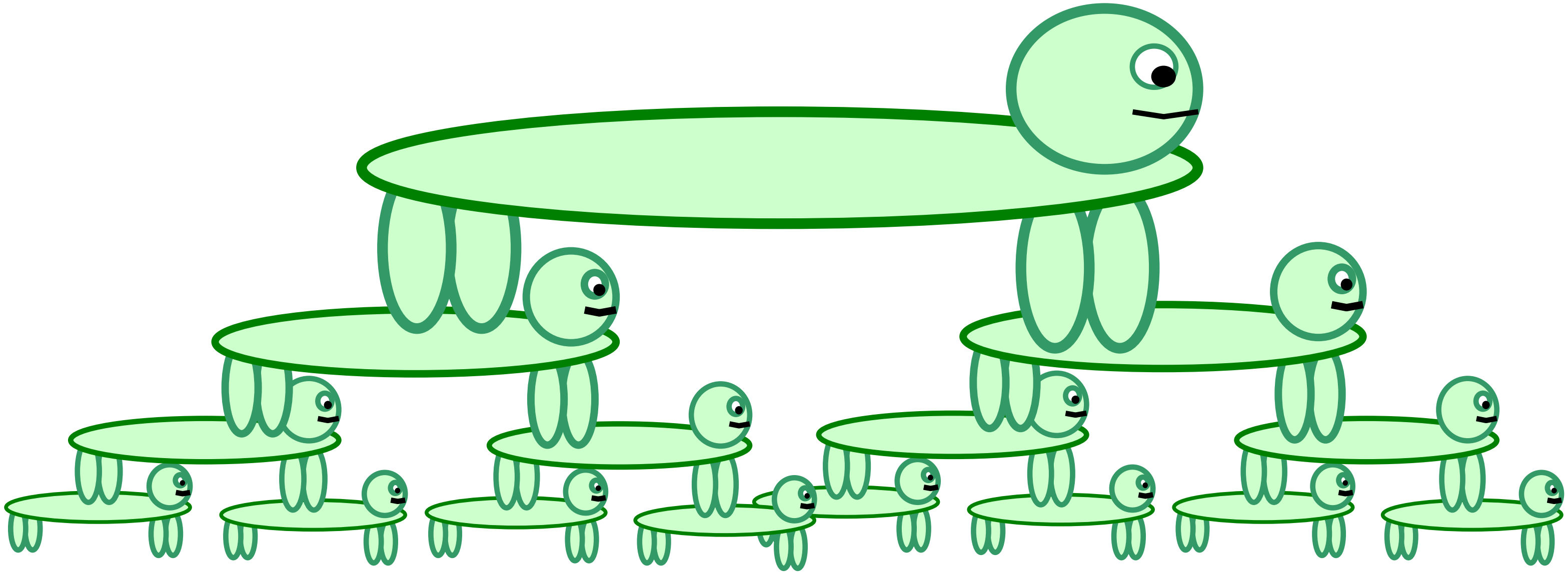
Optimizing for cache can beat small-scale parallelism



Intel® TBB has an example version that is restructured *and* parallel.

One More Problem: Nested Parallelism

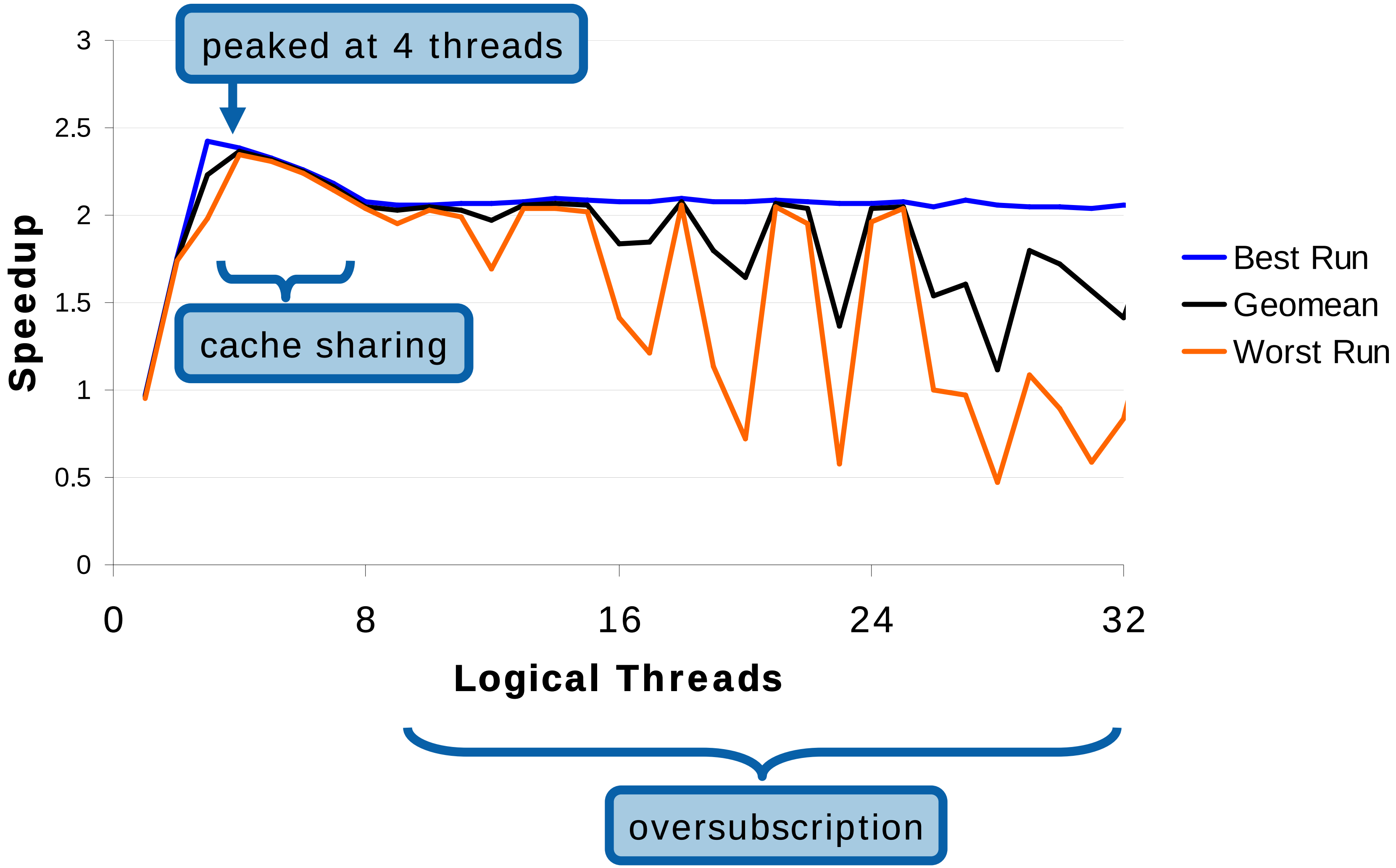
Software components are built from smaller components



If each turtle specifies threads...

Effect of Oversubscription

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



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)

Outline

Overview of Intel®Threading Building Blocks (Intel® TBB)

Problems that Intel® TBB addresses

Origin of Intel® TBB

Parallel Algorithm Templates

BREAK

How it works

Synchronization

Concurrent Containers

Miscellenea

When to use native threads, OpenMP, TBB

Quick overview of TBB sources

Three Approaches for Improvement

New language

- Cilk, NESL, Fortress, ...
- Clean, conceptually simple
- **But** very difficult to get widespread acceptance

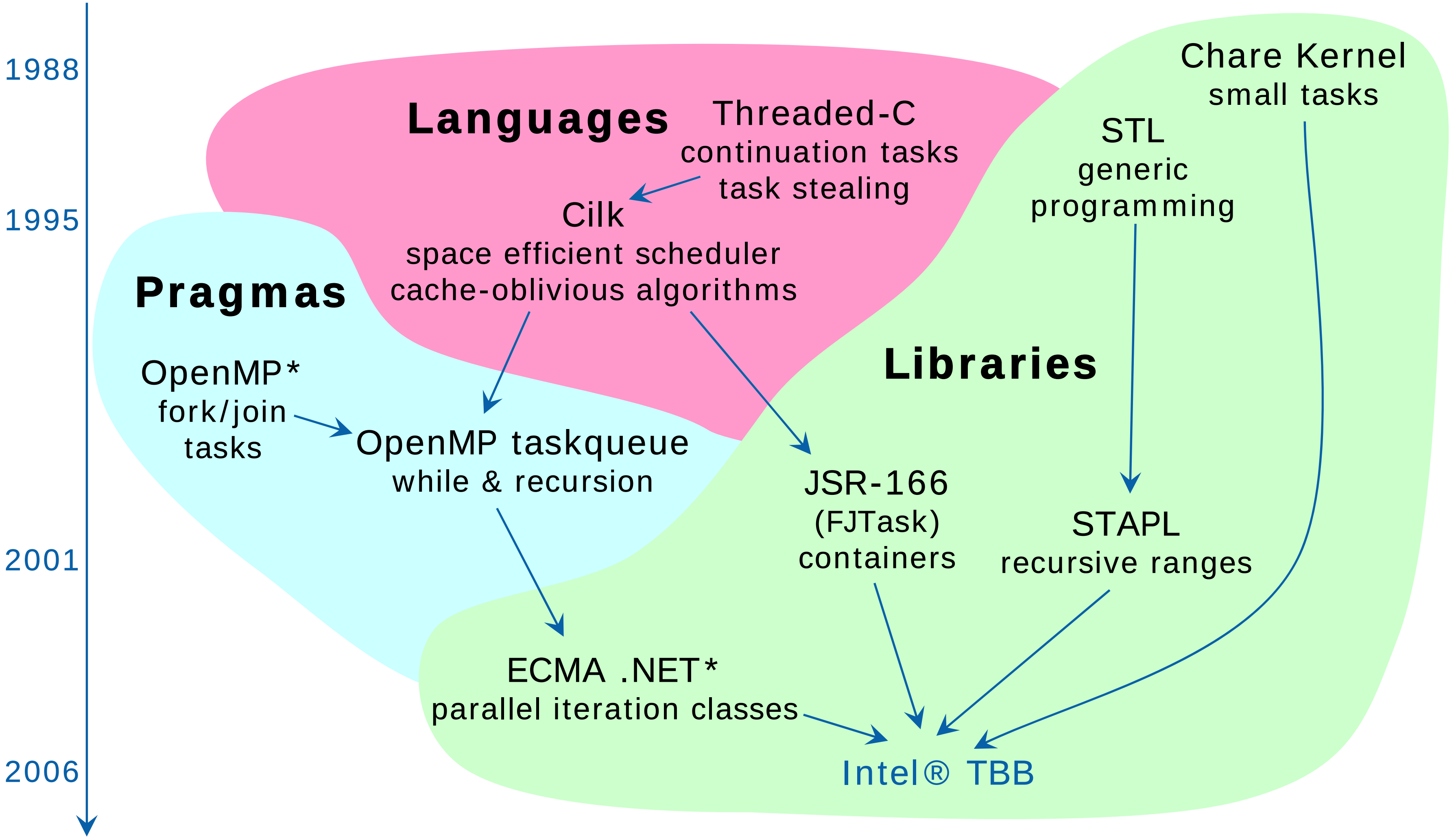
Language extensions / pragmas

- OpenMP, HPF
- Easier to get acceptance
- **But** still require a special compiler or pre-processor

Library

- POOMA, Hood, MPI, ...
- Works in existing environment, no new compiler needed
- **But** Somewhat awkward
  - Syntactic boilerplate
  - Cannot rely on advanced compiler transforms for performance

Family Tree



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

Generic Programming

Best known example is C++ Standard Template Library

Enables distribution of broadly-useful high-quality algorithms and data structures

Write best possible algorithm in most general way

- Does not force particular data structure on user (like std::sort)
  - parallel_for does not require specific type of iteration space, but only that it have signatures for recursive splitting

Instantiate algorithm to specific situation

- C++ template instantiation, partial specialization, and inlining make resulting code efficient
- E.g., parallel loop templates use only one virtual function

I assume you all saw Bjarne Stroustrup yesterday, so know **all** about this!

Key Features of Intel® Threading Building Blocks

You specify *task patterns* instead of threads (focus on the work, not the workers)

- 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.

Relaxed Sequential Semantics

TBB emphasizes *relaxed sequential* semantics

- Parallelism as accelerator, not mandatory for correctness.

Examples of mandatory parallelism

- Producer-consumer relationship with bounded buffer
- MPI programs with cyclic message passing

Evils of mandatory parallelism

- Understanding is harder (no sequential approximation)
- Debugging is complex (must debug the whole)
- Serial efficiency is hurt (context switching required)
- Throttling parallelism is tricky (cannot throttle to 1)
- Nested parallelism is inefficient (all turtles must run!)

Scalability

How the performance improves as we add hardware threads.

Ideally you want Performance $\propto$ Number of hardware threads, but Amdahl tells us this is hard.

Generally prepared to accept Performance $\propto \sqrt{\text{Number of threads}}$

Impediments to scalability

- Any code which executes once for each thread (e.g. a loop starting threads)
- Coding for a fixed number of threads (can't exploit extra hardware; oversubscribes less hardware)
- Contention for shared data (locks cause serialization)

TBB approach

- Create tasks recursively (for a tree this is logarithmic in number of tasks)
- Deal with tasks not threads. Let the runtime (which knows about the hardware on which it is running) deal with threads.
- Try to use recursive tasks to avoid the need for locks.
  - Provide efficient atomic operations and locks if you really need them.

Intel® TBB Components

Generic Parallel Algorithms

parallel_for
parallel_while
parallel_reduce
pipeline
parallel_sort
parallel_scan

Concurrent Containers

concurrent_hash_map
concurrent_queue
concurrent_vector

Task scheduler

Synchronization Primitives

atomic, spin_mutex, spin_rw_mutex,
queuing_mutex, queuing_rw_mutex, mutex

Memory Allocation

cache_aligned_allocator
scalable_allocator

Outline

Overview of Intel®Threading Building Blocks (Intel® TBB)

Problems that Intel® TBB addresses

Origin of Intel® TBB

Parallel Algorithm Templates

BREAK

How it works

Synchronization

Concurrent Containers

Miscellenea

When to use native threads, OpenMP, TBB

Quick overview of TBB sources

Parallel Algorithm Templates : parallel_for

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

Requirements for parallel_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

R::R (const R&)	Copy constructor
R::~R()	Destructor
bool R::empty() const	True if range is empty
bool R::is_divisible() const	True if range can be partitioned
R::R (R& r, split)	Split r into two subranges

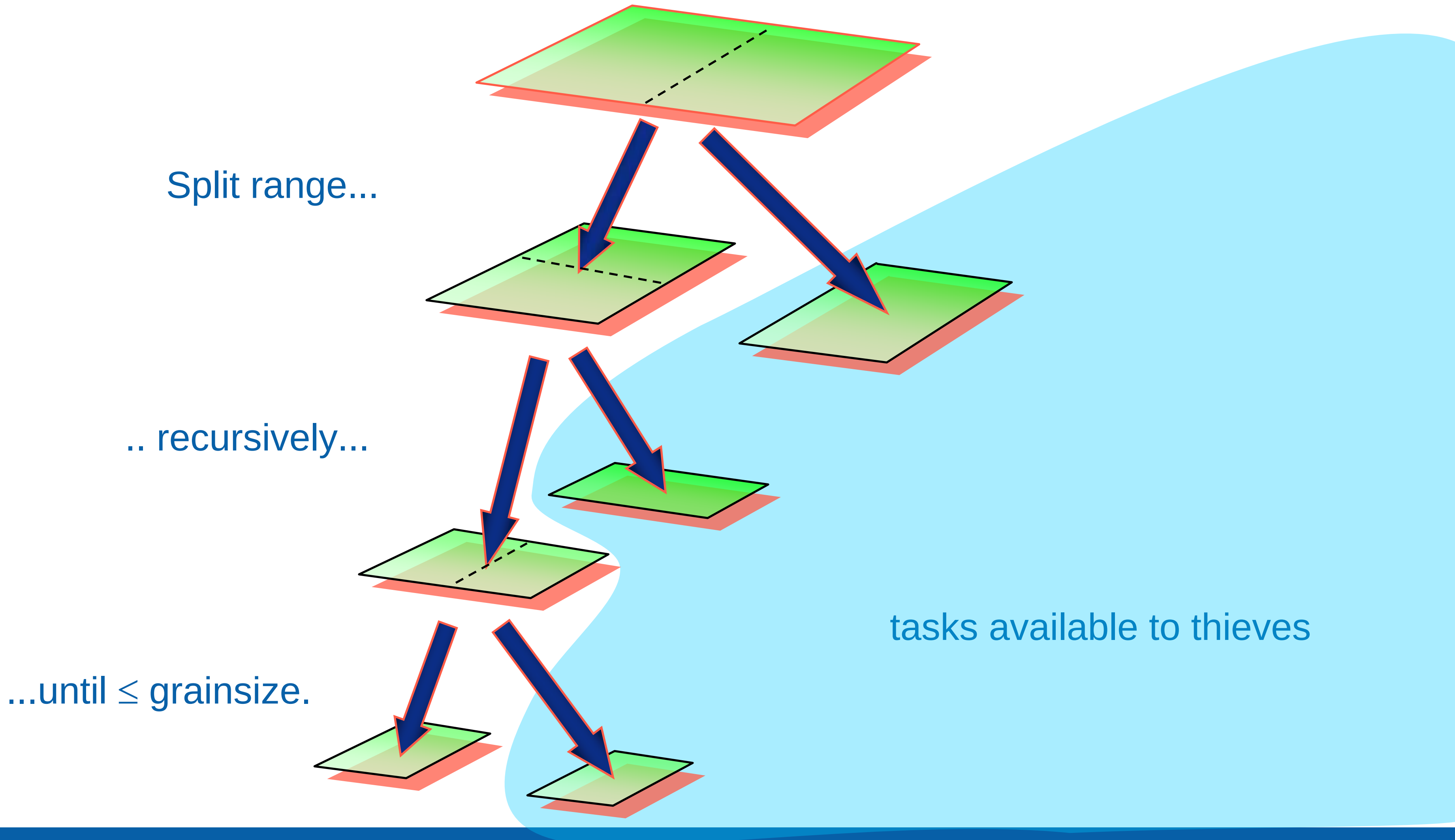
Library provides blocked_range and blocked_range2d

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 which you'll get at the end of the tutorial)

How this works on blocked_range2d



Partitioning the work

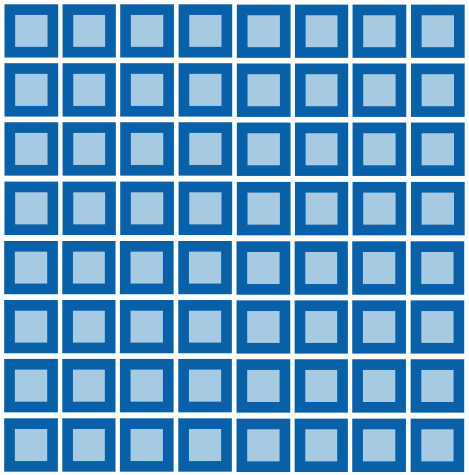
Like OpenMP, Intel TBB “chunks” ranges to amortize overhead

Chunking is handled by a partitioner object

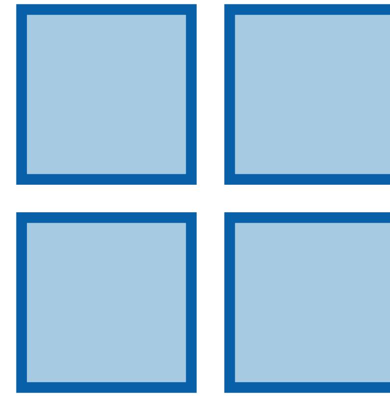
- TBB currently offers two: the auto_partitioner() and the simple_partitioner()
  - Normally use the auto_partitioner
    - parallel_for(blocked_range<int>(1, N), Body() , auto_partitioner());
  - Simple_partitioner reserved for special cases using a manual grain size
    - parallel_for(blocked_range<int>(1, N, grain_size), Body());

Tuning 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=∞
- 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.
- To begin with use the auto_partitioner

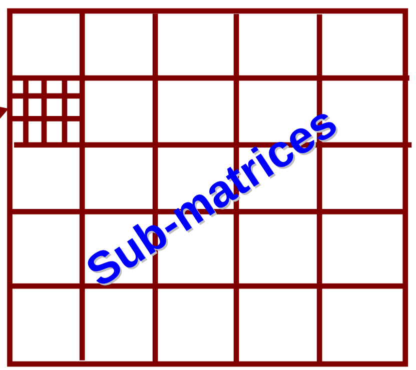
parallel_for
Matrix Multiply: Serial Version

```
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 Body

```
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 C



Matrix Multiply: parallel_for

```
#include "tbb/task_scheduler_init.h"
#include "tbb/parallel_for.h"
#include "tbb/blocked_range2d.h"


tbb::task_scheduler_init tbb_init(); // Initialize task scheduler
tbb::parallel_for (/*Range*/ blocked_range2d<size_t>(0, N, 32, 0, N, 32),
                   /*Body*/ MatrixMultiplyBody2D(c,a,b) );
```

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

Requirements for parallel_reduce Body

Body::Body(const Body&, split)	Splitting constructor
Body::~Body()	Destructor
void Body::operator() (Range& *subrange*) const	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)

blue = original code
red = provided by TBB
black = boilerplate for library

```
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;
}
```

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

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 Intel®TBB book if you need them.

For now just remember that they exist.

Parallel Algorithm Templates : parallel_scan

Computes a parallel prefix

Interface is similar to parallel_for and parallel_reduce.

Parallel Algorithm Templates : parallel_sort

An unstable comparison sort with O(n *log* n) serial complexity.

If hardware is available can approach O(n) runtime.

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
  - The loop body

Parallel pipeline

Linear pipeline of stages

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

Each stage can be *serial* or *parallel*

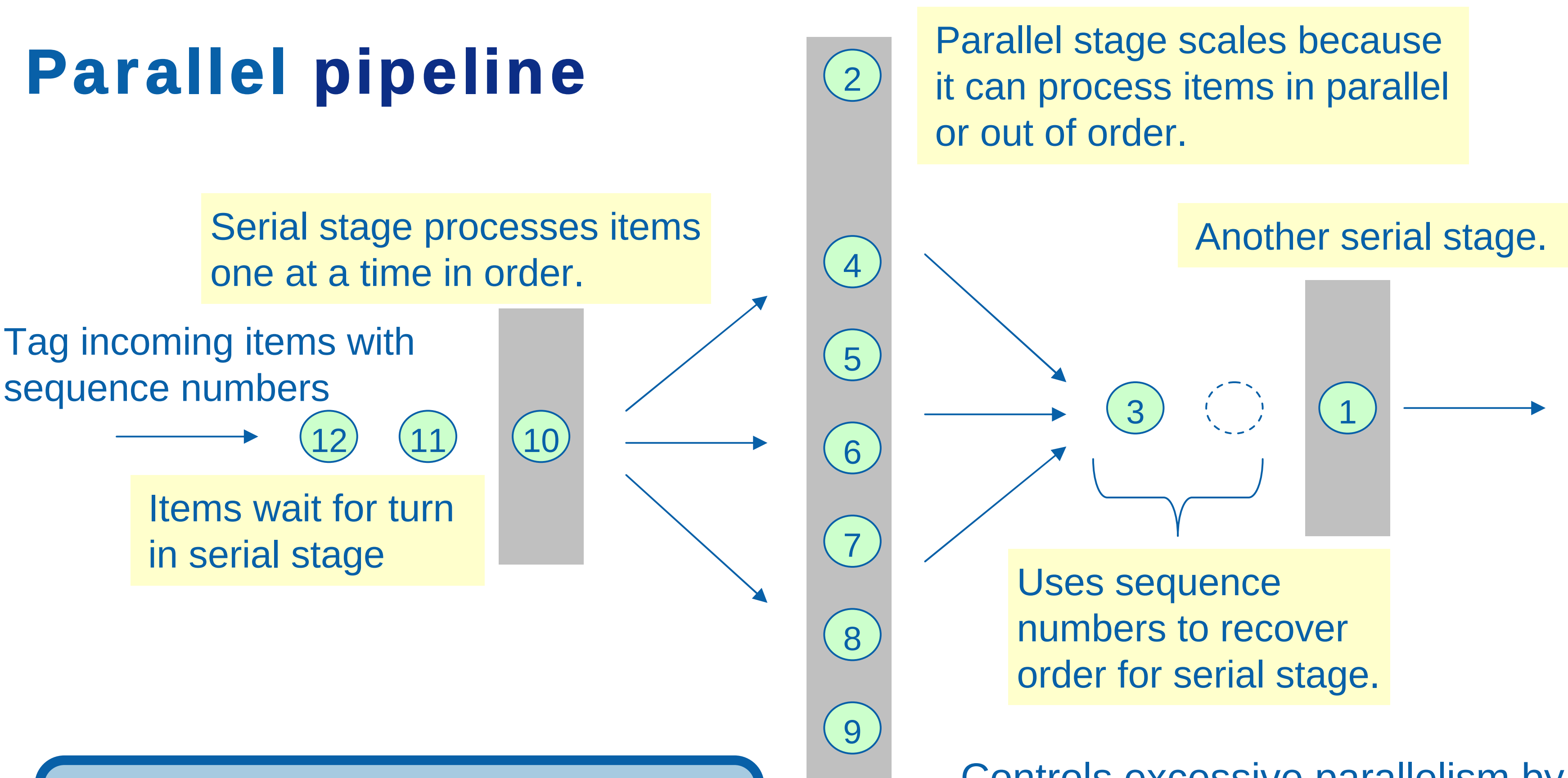
- Serial stage processes one item at a time, in order.
- Parallel stage can process multiple items at a time, out of order.

Uses cache efficiently

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

BUT functional decomposition is normally not scalable, so if you can, find another way.

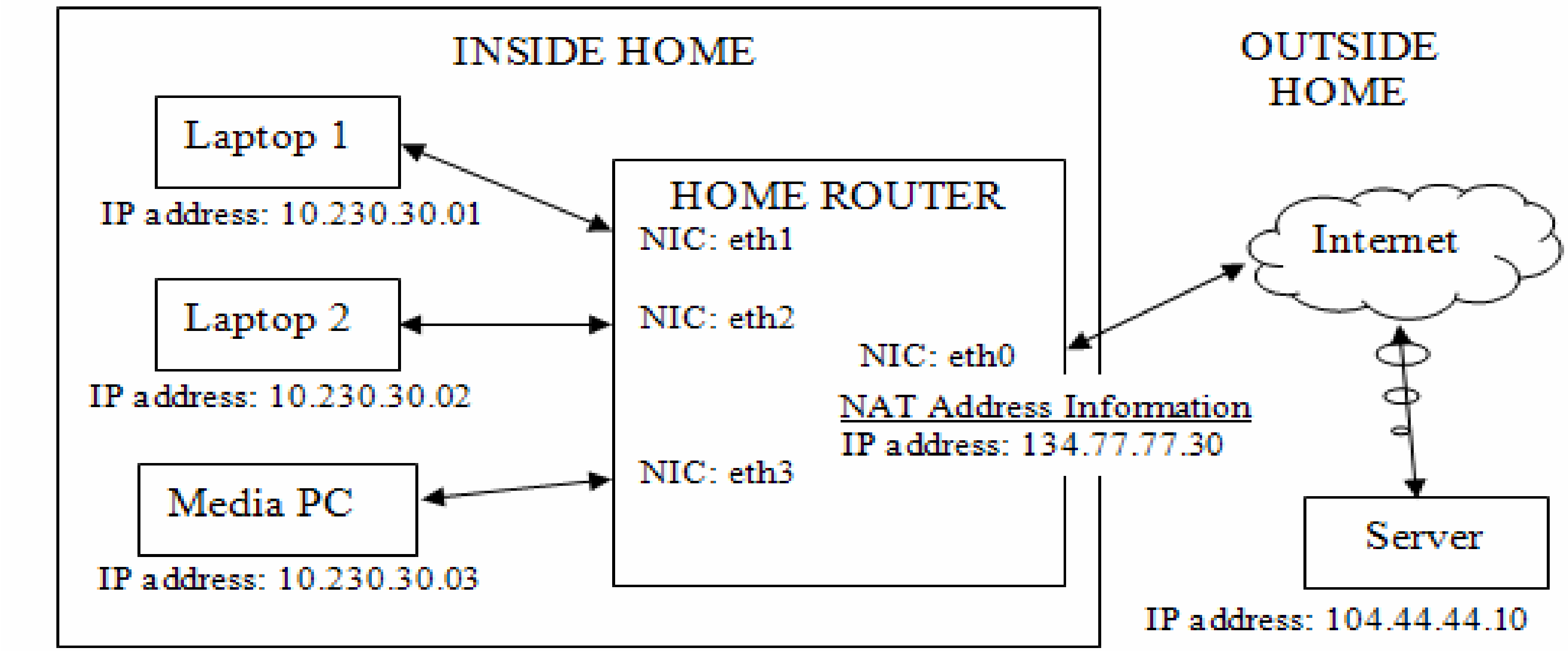
Parallel pipeline



Throughput limited by throughput of slowest serial stage.

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

pipeline example:
Local Network Router



(IP,NIC) and (Router Outgoing IP, Router Outgoing NIC) – network configuration file

Pipeline Components for Network Router

Recv Packet | NAT | ALG | FWD | Send Packet

Network Address Translation: Maps private network IP and application port number to router IP and router assigned port number

Application Level Gateway: Processes packets when peer address or port are embedded in app-level payloads (e.g. FTP PORT command)

Forwarding: Moves packets from one NIC to another using static network config table

TBB Pipeline Stage Example

```
class get_next_packet : public tbb::filter {
        istream& in_file;
public:
        get_next_packet (ifstream& file) : in_file (file), filter (/*is_serial?*/ true) {}
        void* operator() (void*) {
                packet_trace* packet = new packet_trace ();
                in_file >> *packet; // Read next packet from trace file
                if (packet->packetNic == empty) { // If no more packets
                        delete packet;
                        return NULL;
                }
                return packet; // This pointer will be passed to the next stage
        }
};
```

Router Pipeline

```
#include "tbb/pipeline.h"
#include "router_stages.h"

void run_router (void) {
  tbb::pipeline pipeline; // Create TBB pipeline

  get_next_packet receive_packet (in_file); // Create input stage
  pipeline.add_filter (receive_packet);       // Add input stage to pipeline

  translator network_address_translator (router_ip, router_nic, mapped_ports);// Create NAT stage
  pipeline.add_filter (network_address_translator); // Add NAT stage

  /* Create and add other stages to pipeline: ALG, FWD, Send */

  pipeline.run (number_of_live_items); // Run Router
  pipeline.clear ();
}
```

Outline

Overview of Intel® Threading Building Blocks (Intel® TBB)

Problems that Intel® TBB addresses

Origin of Intel® TBB

Parallel Algorithm Templates

BREAK

How it works

Synchronization

Concurrent Containers

Miscellenea

When to use native threads, OpenMP, TBB

Quick overview of TBB sources

intel®

Leap ahead™

Outline

Overview of Intel® Threading Building Blocks (Intel® TBB)

Problems that Intel® TBB addresses

Origin of Intel® TBB

Parallel Algorithm Templates

BREAK

How it works

Synchronization

Concurrent Containers

Miscellenea

When to use native threads, OpenMP, TBB

Quick overview of TBB sources

How it all 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

How recursive partitioning works on `blocked_range2d`

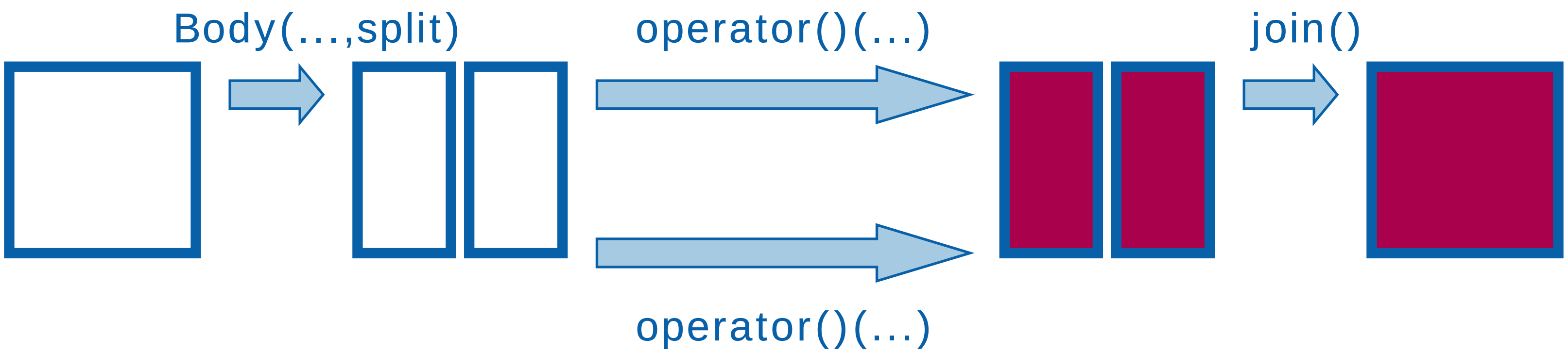
Split range...

.. recursively...

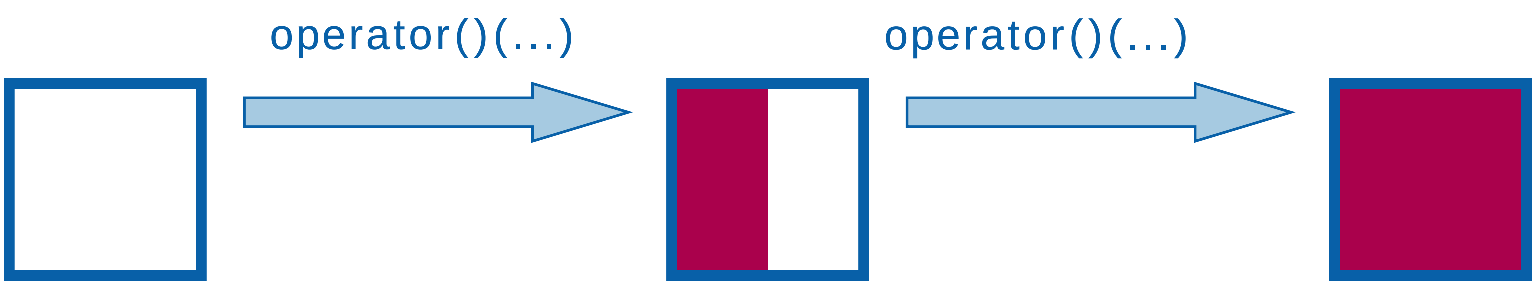
...until ≤ grainsize.

Lazy Parallelism

If a spare thread is available

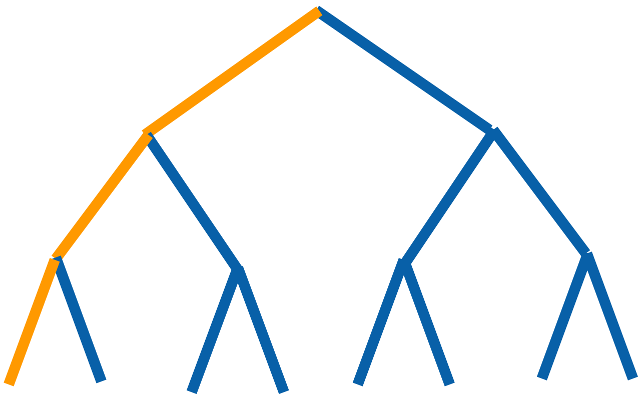


If no spare thread is available



Two Possible Execution Orders

Depth First Task Order (stack)

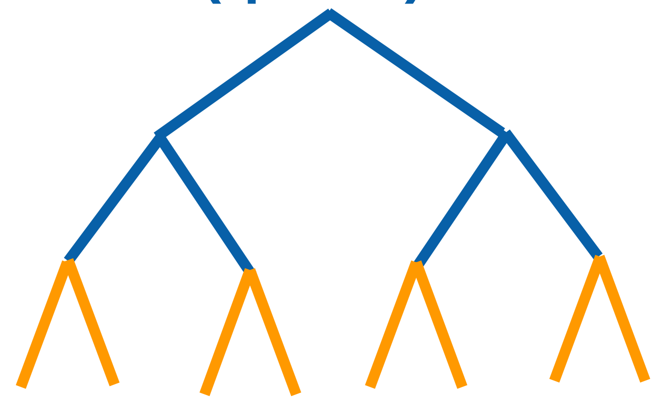


Small space

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

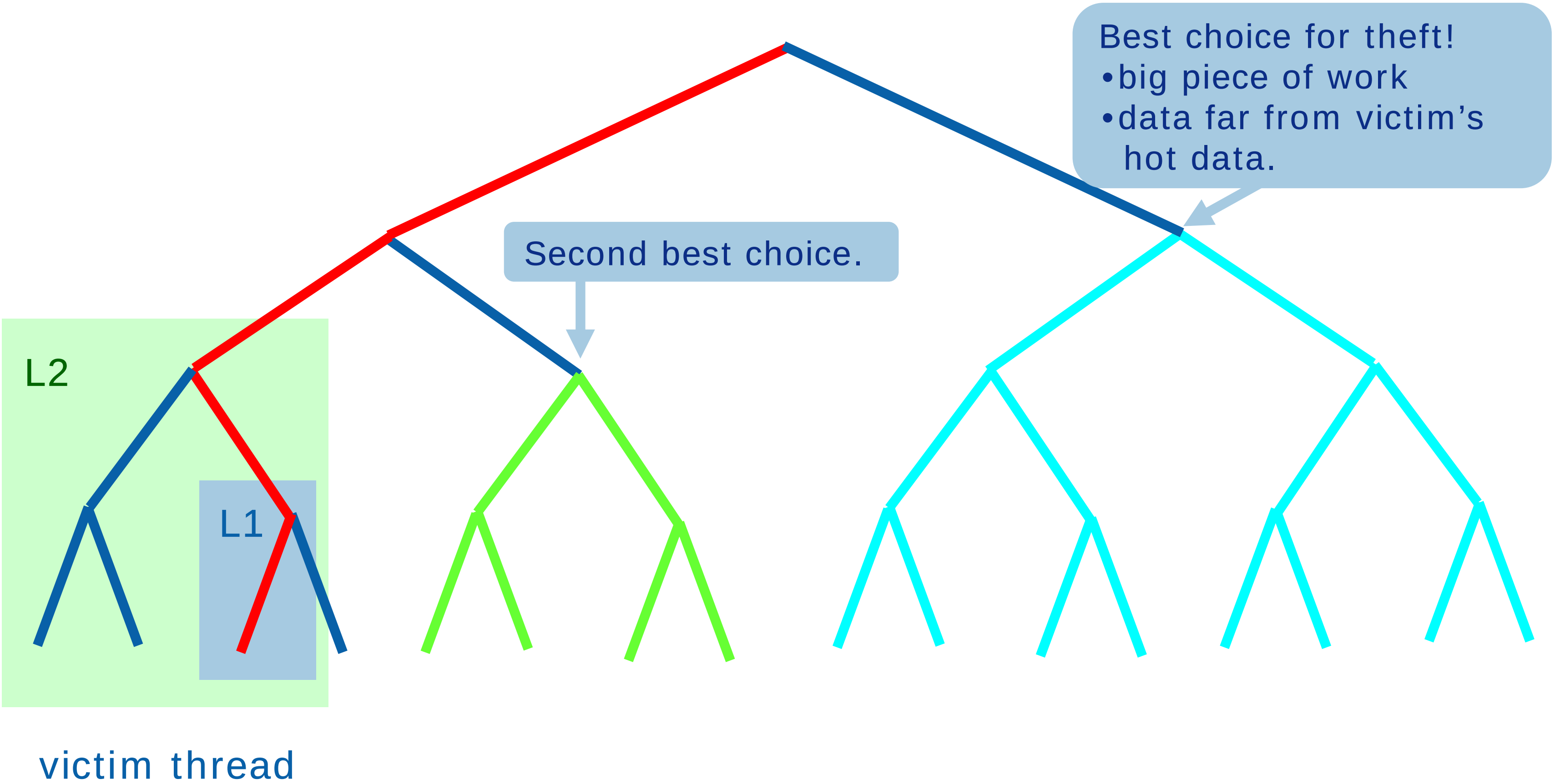
If thread runs out of work

- 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



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

Overview of Intel® Threading Building Blocks (Intel® TBB)

Problems that Intel® TBB addresses

Origin of Intel® TBB

Parallel Algorithm Templates

BREAK

How it works

Synchronization

Concurrent Containers

Miscellenea

When to use native threads, OpenMP, TBB

Quick overview of TBB sources

Synchronization

Why synchronize?

Consider simple code like this

```
int tasksDone = 0;
void doneTask(void)
{
   tasksDone++;
}
```

What will happen if we run it concurrently on two threads?

We **hope** that when they have finished tasksDone will be 2.

BUT that need not be so, even though we wrote "tasksDone++", there can be an interleaving like this

T1	T2
Reg = tasksDone;	Reg = tasksDone;
Reg++;	Reg++;
tasksDone = Reg;	
	tasksDone = Reg;

So, tasksDone == 1, because we lost an update...

Race condition

There were multiple, unsynchronized accesses to the same variable, and at least one of them was a write.

To fix the problem we have to introduce synchronization, either

1. Perform the operation atomically

Or

2. Execute it under control of a lock (or critical section).

```
int tasksDone = 0;

void doneTask(void)
{
   static class spin_mutex myMutex;
   class spin_mutex::scoped_lock(myMutex);
   tasksDone++;
}
```

Intel Thread Checker can **automatically** detect most such races.

But Beware: Race Freedom does not guarantee correctness

Consider our simple code written like this

```
int tasksDone;

void doneTask(void)
{
    int tmp;
    CRITICAL {
      tmp = tasksDone;
    }
    tmp ++;
    CRITICAL {
       tasksDone = tmp;
    }
}
```

CRITICAL here is assumed to provide a critical section for the next block. It is just for explanation, and not a genuine language feature…

There are no races… **BUT** the code is still broken the same way as before.

Synchronization has to be used to preserve program invariants, it’s not enough to lock every load and store.

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 ("are we there yet" vs. "wake me when we get there")
  - spin_mutex, queuing_mutex
- Writer vs. reader/writer (supports multiple reader/ single writer)
  - spin_rw_mutex, queuing_rw_mutex
- Scoped wrapper of native mutual exclusion function
  - mutex (Windows: CRITICAL_SECTION, Linux: pthreads mutex)

Synchronization Primitives

Mutex traits

- Scalable: does no worse than the serializing mutex access
- "Fair": preserves the order of thread requests; no starvation
- Reentrant: locks can stack; not supported for provided behaviors
- Wait method: how threads wait for the lock

	Scalable	"Fair"	Wait
mutex	OS dependent	OS dependent	Sleep
spin_mutex	No	No	Spin
queuing_mutex	Yes	Yes	Spin
spin_rw_mutex	No	No	Spin
queuing_rw_mutex	Yes	Yes	Spin

Mutexes, having a lifetime, can deadlock or convoy

- Avoid by reducing lock lifetime or using atomic operations
  - atomic<T> supports HW locking for read-modify-write cycles
  - useful for updating single native types like semaphores, ref counts

reader-writer lock lab:
Network Router: ALG stage

```
typedef std::map<port_t, address*> mapped_ports_table;
class gateway: public tbb::filter {
   void* operator() (void* item) {
      packet_trace* packet = static_cast<packet_trace*> (item);
      if (packet->packetDestPort==FTPcmdPort) { //Outbound packet sends FTP cmd
         // packetPayloadApp contains data port - save it in ports table
         add_new_mapping (packet->packetSrcIp, packet->packetPayloadApp,
                          packet->packetSrcPort);
         packet->packetSrcIp = outgoing_ip; // Change packet outgoing IP and port
         packet->packetPayloadApp = packet->packetSrcPort;
      }
      return packet; // Modified packet will be passed to the FWD stage
   }
};
```

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

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 effcient.

Useful primitives for building lock-free algorithms.

Portable no need to roll your own assembler code

Synchronization: Atomic

A better (smaller, more effcient) solution for our threadcount problem…

```
atomic<int> tasksDone;

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

Outline

Overview of Intel®Threading Building Blocks (Intel® TBB)

Problems that Intel® TBB addresses

Origin of Intel® TBB

Parallel Algorithm Templates

BREAK

How it works

Synchronization

Concurrent Containers

Miscellenea

When to use native threads, OpenMP, TBB

Quick overview of TBB sources

Concurrent Containers

Intel® TBB provides concurrent containers

- STL containers are <u>not safe</u> under concurrent operations
  - attempting multiple modifications concurrently could corrupt them
- Standard practice: wrap a lock around STL container accesses
  - Limits accessors to operating one at a time, killing scalability

TBB provides fine-grained locking and lockless operations where possible

- Worse single-thread performance, but better scalability.
- Can be used with TBB, 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_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_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.

BUT beware: a queue is cache unfriendly. A pipeline pattern might well perform better…

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.

Improving our Network Router

Remaining problem: Global lock blocks the entire STL map. Multiple threads will wait on this lock even if they access different parts of the container

TBB solution: tbb::concurrent_hash_map

- Fine-grained synchronization: threads accessing different parts of container do not block each other
- Reader or writer access: multiple threads can read data from the container concurrently or modify it exclusively

Network Router: tbb::concurrent_hash_map

```
bool port_number::get_ip_address (mapped_ports_table &mapped_ports, ip_t &addr)
{
        tbb::spin_rw_mutex::scoped_lock lock (my_rw_lock, /*is_writer*/ false);
        // Accessor constructor acquires reader lock
        tbb::concurrent_hash_map<port_t, address*, comparator>::const_accessor a;
        if (mapped_ports.find (a, port)) {
            return a->second->get_ip_address (mapped_ports, addr);
        }
    return false; // Accessor's destructor releases lock
}
```

Outline

Overview of Intel® Threading Building Blocks (Intel® TBB)

Problems that Intel® TBB addresses

Origin of Intel® TBB

Parallel Algorithm Templates

BREAK

How it works

Synchronization

Concurrent Containers

Miscellenea

When to use native threads, OpenMP, TBB

Quick overview of TBB sources

Scalable Memory Allocator

Problem

- Memory allocation is often a bottle-neck a concurrent environment
  - Thread allocation from a global heap requires global locks.

- Solution
  - Intel® Threading Building Blocks provides tested, tuned, scalable, per-thread memory allocation
  - Scalable memory allocator interface can be used…
    - As an *allocator* argument to STL template classes
    - As a replacement for malloc/realloc/free calls (C programs)
    - As a replacement for *new* and *delete* operators (C++ programs)

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 real time

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.

Outline

Overview of Intel® Threading Building Blocks (Intel® TBB)

Problems that Intel® TBB addresses

Origin of Intel® TBB

Parallel Algorithm Templates

BREAK

How it works

Synchronization

Concurrent Containers

Miscellenea

When to use native threads, OpenMP, TBB

Quick overview of TBB sources

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

Use Native Threads when

You already have a code written using them.

But, consider using TBB components

- Locks, atomics
- Data structures
- Scalable allocator

They provide performance and portability and can be introduced incrementally

Outline

Overview of Intel®Threading Building Blocks (Intel® TBB)

Problems that Intel® TBB addresses

Origin of Intel® TBB

Parallel Algorithm Templates

BREAK

How it works

Synchronization

Concurrent Containers

Miscellenea

When to use native threads, OpenMP, TBB

Quick overview of TBB sources

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

Summary of Intel® Threading Building Blocks

It is a *library*

You specify *task patterns*, not threads

Targets threading for *robust performance*

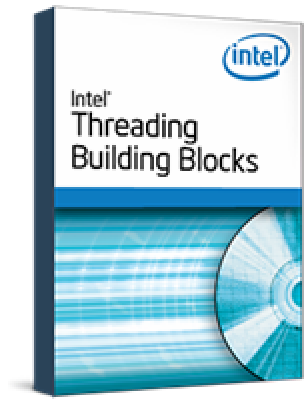
Does well with *nested parallelism*

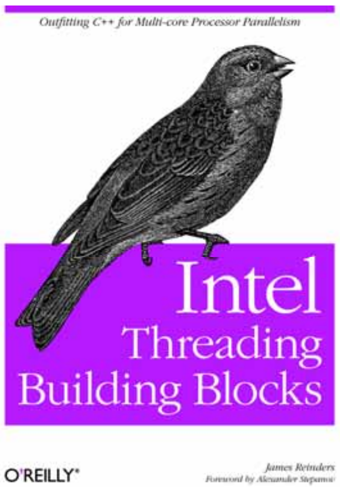
Compatible with other threading packages

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 (or buy Lawrence a beer and ask him all about it)

Other Intel® Threading tools: Thread Profiler, Thread Checker

http://www.intel.com/software/products/threading intel®

Leap ahead™

Supplementary Links

- **Open Source Web Site**
- http://threadingbuildingblocks.org
- **Commercial Product Web Page**
- http://www.intel.com/software/products/tbb
- **Dr. Dobb's NetSeminar**
- "Intel® Threading Building Blocks: Scalable Programming for Multi-Core"
- http://www.cmpnetseminars.com/TSG/?K=3TW6&Q=417
- **Technical Articles:**
- "Demystify Scalable Parallelism with Intel Threading Building Block's Generic Parallel Algorithms"
- 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
- **Industry Articles:**
- Product Review: Intel Threading Building Blocks
- http://www.devx.com/go-parallel/Article/33270
- "The Concurrency Revolution", Herb Sutter, Dr. Dobb's 1/19/2005
- http://www.ddj.com/dept/cpp/184401916