Multicore Days 2008
Stockholm, September 11

Parallel Programming with Intel® Threading Building Blocks

Software and Solutions Group (SSG),
Developer Products Division

Alexey Kukanov

Outline

What is TBB?

Task based parallelism

High-level blocks

Other functionality (primarily for reference)

Future directions

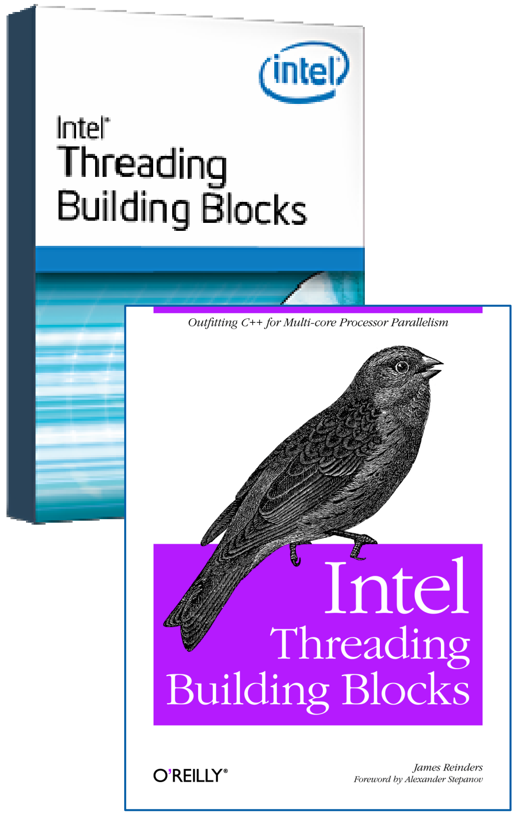
What Is TBB?

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

Not a new language or extension; works with off-the-shelf C++ compilers.

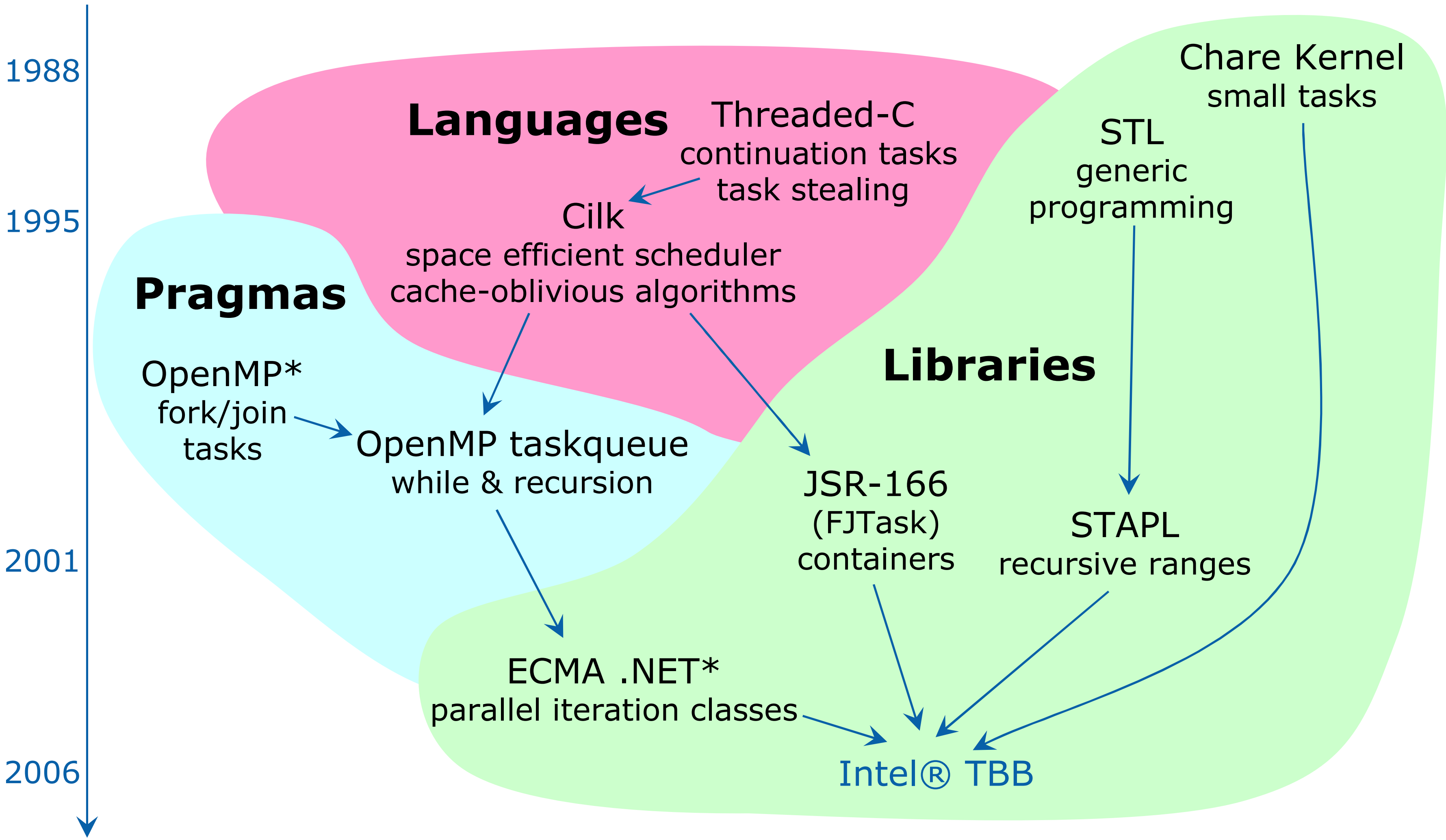
Proven to be portable to new compilers, operating systems, and architectures.

GPL license allows use on many platforms; commercial license allows use in products.



http://threadingbuildingblocks.org

Family Tree



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

TBB History

August, 2004

- the TBB project started at Intel.

June, 2006 – Intel® TBB 1.0

- Intel's New Parallel Programming Model announced.

April, 2007 – Intel® TBB 1.1

- OS coverage, bug fixes & small improvements.

July, 2007 – Intel® TBB 2.0

- TBB announced as Open Source Software.

July, 2008 – Intel® TBB 2.1

- Offers much enriched functionality & enables new uses.
- Many features & improvements started as discussions with community and customers.

Intel® TBB 2.1 Components

Parallel Algorithms
parallel_for
parallel_reduce
parallel_scan
parallel_do
pipeline
parallel_sort

Concurrent Containers
concurrent_hash_map
concurrent_queue
concurrent_vector

Task scheduler
task
task_scheduler_init
task_scheduler_observer

Memory Allocation
tbb_allocator
cache_aligned_allocator
scalable_allocator

Explicit Threading
tbb_thread

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

Miscellanea/Support
tick_count, task_group_context
blocked ranges, partitioners

Shift from Serial to Parallel

It's all about managing dependences

- Find things that can be done (*almost*) independently.
- Analyze communication (dependences).
- Eliminate or organize dependences to exploit parallelism.

Allow parallelism, not mandate it

- Excessive concurrency has its problems.
- Mandatory parallelism is not composable.
- Good to have sequential execution e.g. for debugging.
- Also important for backward scaling.

Design for Scalability

Parallel slack

- Want potential parallelism to exceed HW parallelism
- Important for load balancing and forward scaling
- Functional decomposition does not scale

Data locality

- Memory latency varies (cache hierarchy, NUMA)
- Compute on data that is near, not far
- Avoid cache misses and sharing

Task Based Parallelism

Can be as fine-grain as necessary

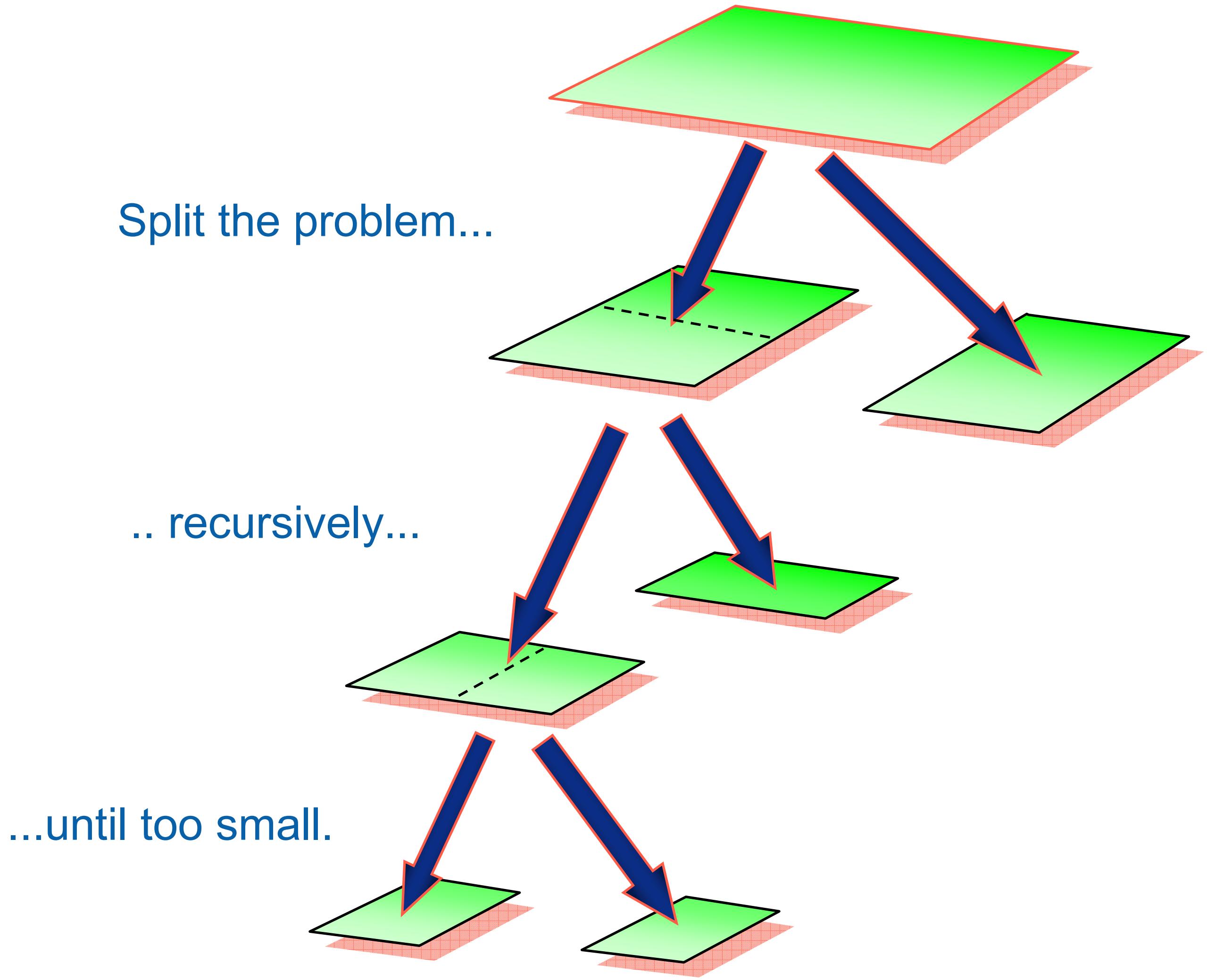
Focus on the work, not workers

Parallelism is optional

Data decomposition naturally provides parallel slack

Allows exploiting data locality

Recursive Decomposition



Practical Task Based Programming with TBB

TBB allows you to program in terms of task objects.

Parallelism is expressed explicitly via TBB constructs.

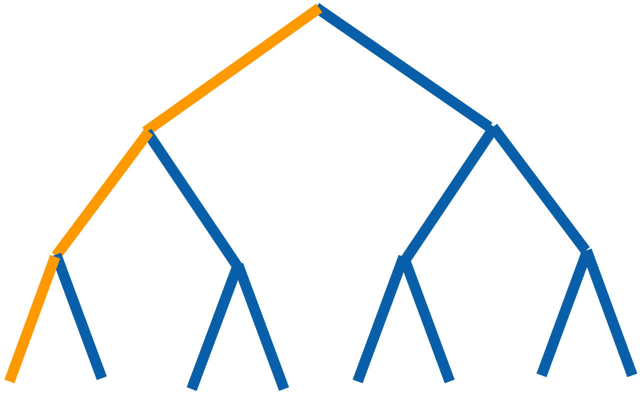
- No magic bullets, and no free lunch
- Trust the programmer

Task scheduler maps user-defined logical tasks onto physical threads.

- One SW thread per HW thread
- Work stealing balances load
- Data locality is controlled implicitly and explicitly
- Works well with nested parallelism

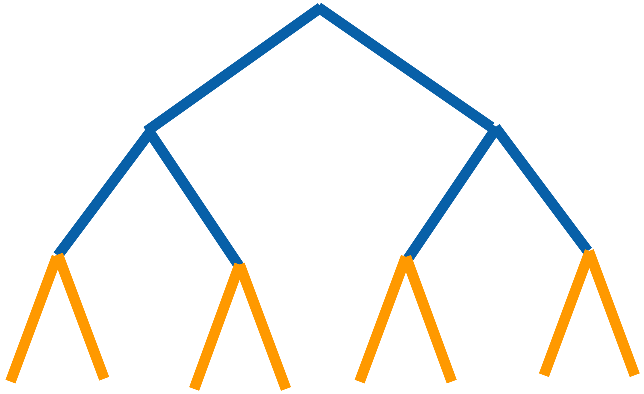
Two Possible Task Execution Orders

Depth First
(stack)



Small space
Excellent cache locality
No parallelism

Breadth First
(queue)



Large space
Poor cache locality
Maximum parallelism

Work Stealing

Each thread maintains an (approximate) deque of tasks

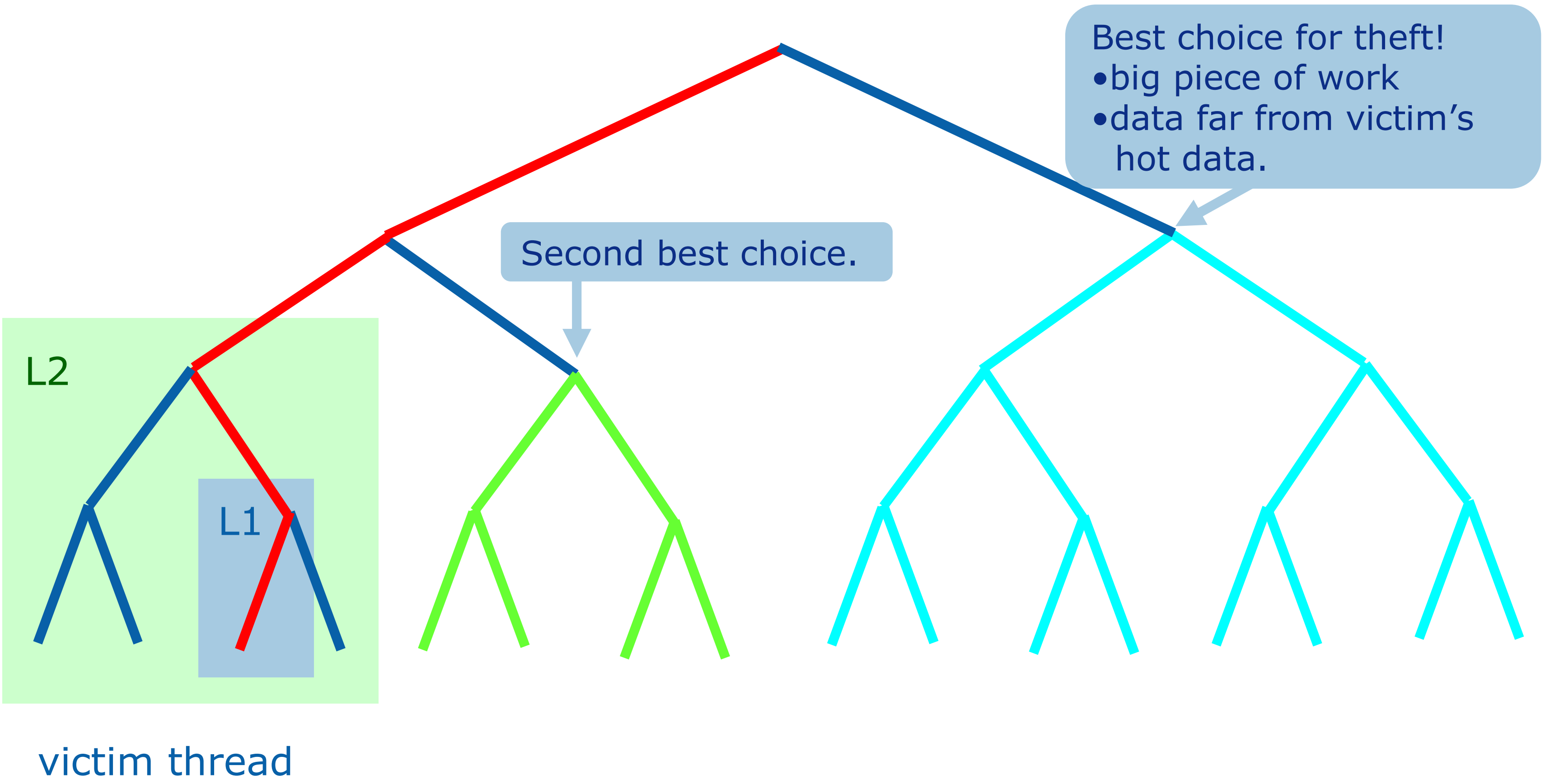
A thread performs depth-first execution

- 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.

Work Depth First; Steal Breadth First



Initializing TBB

Create *task_scheduler_init* object in a thread that uses TBB.

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

Constructor specifies thread pool size (as *automatic*, *explicit* or *deferred) and t*hread stack size.

```
task_scheduler_init  init( task_scheduler_init::automatic, my_stack_size);
```

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

Parallel Algorithms

Loop parallelization

- parallel_for
- parallel_reduce
- parallel_scan

Algorithms for Streams

- parallel_do
- pipeline

Sorting

- parallel_sort

Parallel Algorithms

Classic parallel programming

- Let non-expert get scalable parallel speedup on shared-memory multi-core processor.
- Common simple patterns
- Coarse-grain (typically $\geq 10^4$ instructions per serial chunk)

Implemented on top of work-stealing scheduler

- Algorithms designed to be easy to use in practical way
- Scheduler designed for efficiency

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
  - E.g., std::sort
  - tbb::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

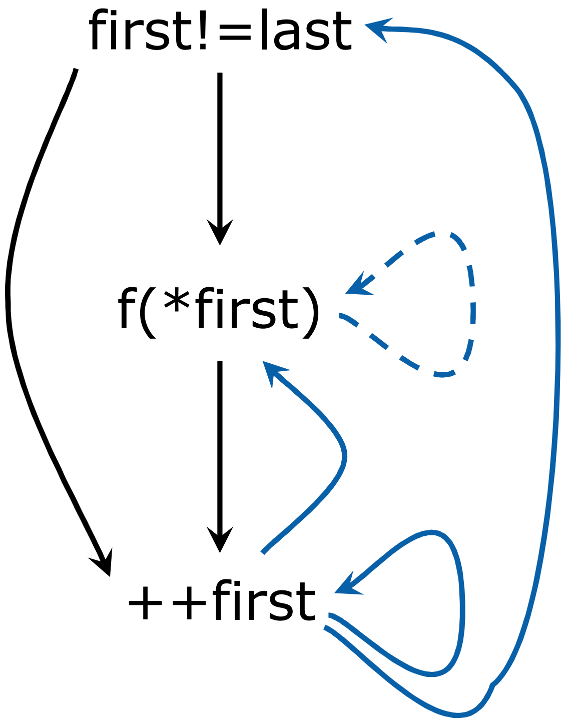
STL

algorithm $\xleftrightarrow{\text{iterator}}$ container

```
template<class InputIter, class Func>
Func for_each( InputIter first, InputIter last, Func f ) {
    while( first!=last ) {
        f(*first);
        ++first;
    }
    return f;
}
```

STL = Serial Template Library?

Dependence graph (loop carried dependences in blue)



Generic Serial Programming

Generalization of pointer bumping

4 of 5 iterator categories are inherently serial

$$
\begin{tikzcd}
\text{Output} \arrow[dr] & & & \\
& \text{Forward} \arrow[r] & \text{Bidirectional} \arrow[r] & \text{RandomAccess} \\
\text{Input} \arrow[ur] & & &
\end{tikzcd}
$$

Often Depends on Coordinated Bumping

```
template <class InputIter1, class InputIter2, class T>
T inner_product(InputIter first1, InputIter1 last1,
                        InputIter2 first2, T init)
{
    while( first1!=last1 ) {
        init = init + *first1 * *first2;
        ++first1;
        ++first2;
    }
    return init;
}
```

Need Richer Topology for Parallelism

Some choices

- Random access iterators
- Random access indices
- Recursively divisible ranges
  - Scale invariant
  - Subsumes random access iterators/indices
  - Not limited to one dimensional spaces
  - Good fit for divide and conquer
  - Maps to work-stealing

Analogy

Serial

algorithm $\xleftrightarrow{\text{iterator}}$ container

for(*init*; *termination-condition; next*)

Parallel

algorithm $\xleftrightarrow{\text{range}}$ container

recurse(*init*; *leaf-condition; split*)

OR algorithm $\xleftrightarrow{\text{range}}$ indices $\xrightarrow{\text{indexing}}$ container

Serial Example

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

Will parallelize by partitioning iteration space into chunks

Parallel Version

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

Loop body as function object

```
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( size_t i=range.begin(); i!=range.end(); ++i )
         Foo(a[i]);
   }
};

void ParallelApplyFoo( float a[], size_t n ) {
   parallel_for( blocked_range<size_t>( 0, n ),
                 ApplyFoo(a),
                 auto_partitioner());
}
```

Iteration space

Parallel algorithm

Partitioning hint

With C++ 200x Lambda Expression

```
void ParallelApplyFoo(float a[], size_t n ) {
   parallel_for( blocked_range<size_t>( 0, n ),
                 [=](const blocked_range<size_t>& range) {
                    for( int i= range.begin(); i!=range.end(); ++i )
                       Foo(a[i]);
                 },
                 auto_partitioner() );
}
```

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

Requirements for Body B

B::B(const B&)	Make a copy
B::~B()	Destroy the copy
void B::operator() (Range& *subrange*) const	Process *subrange*.

parallel_for distributes subranges to worker threads

parallel_for does **not** interpret meaning of range

Range is Generic

Requirements for **Range R**

R::R (const R&)	Make a copy
R::~R()	Destroy the copy
bool R::empty() const	Is range empty?
bool R::is_divisible() const	Can range be split?
R::R (R& r, split)	Split r into two subranges

Library provides blocked_range, blocked_range2d, blocked_range3d

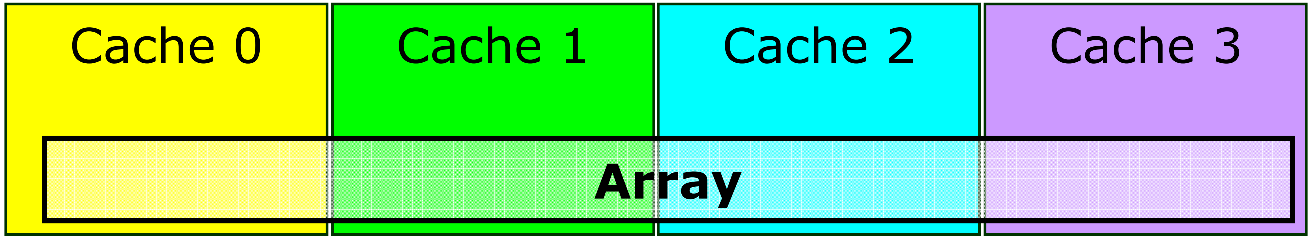
Programmer can define new kinds of ranges

Do not have to be dimensional!

Iteration↔Thread Affinity

Big win for serial repetition of a parallel loop.

- Numerical relaxation methods
- Time-stepping marches



(Simple model of separate cache per thread)

```
affinity_partitioner ap;
...
for( t=0; ...; t++ )
    parallel_for(range, body, ap);
```

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

Requirements for parallel_reduce Body B

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

Operation not necessarily commutative

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), mib, auto_partitioner() );
    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;
      }
   }
   ...
};
```

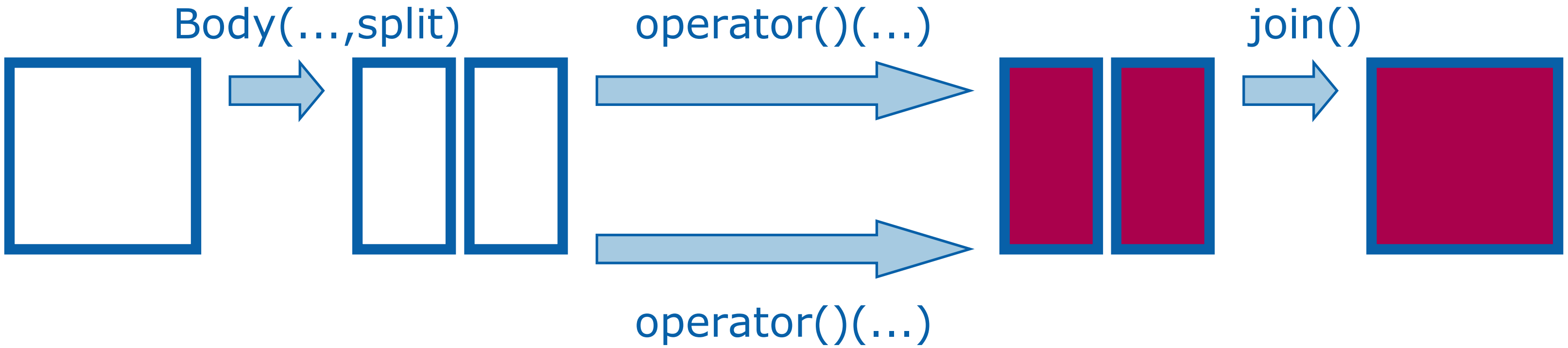
accumulate result

split

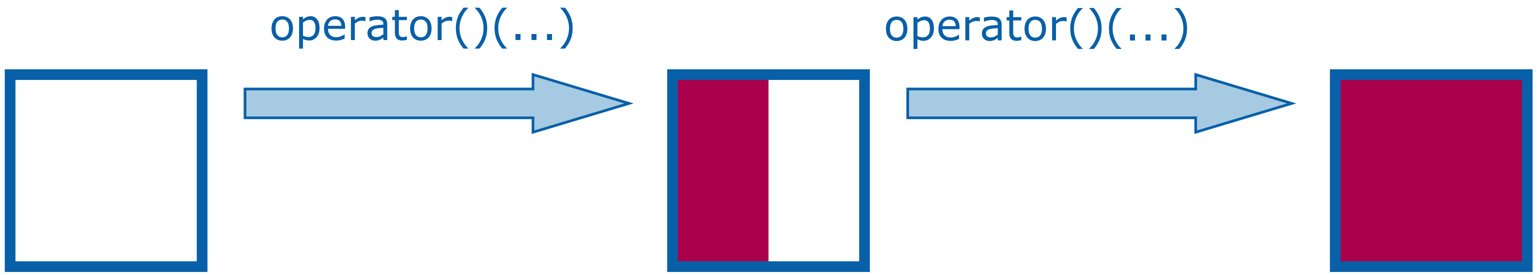
join

Lazy Parallelism in parallel_reduce

If a spare thread is available



If no spare thread is available



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

Requirements for parallel_scan Body B

B::B(B&, split)	Splitting constructor
B::~B()	Destructor
void B::operator() (Range& *subrange,* pre_scan_tag);	Accumulate partial summary.
void B::operator()(Range& *subrange,* final_scan_tag);	Compute final result
void B::reverse_join(B& *lhs*);	Merge summary of *lhs* into this.

Reuses Range concept from parallel_for

Remarks

Brick is efficient serial code

parallel_scan free to optimize evaluation order

- 1 pass for serial execution
- ≤2 passes for parallel execution

STL solution requires four passes for parallel execution

1. generate boolean vector that marks insertion points

2-3. partial_sum to compute destinations

4. copy and “correct” string.

```
template<typename Iterator, typename Body>
void parallel_do( Iterator first, Iterator last,
                  const Body& body );
```

- Exploit parallelism where loop bounds are not known, e.g. do something in parallel on each element in a list.
- Works with standard containers
- Can add more work from inside the body

```
void Body::operator()( Body::argument_type item,
                       tbb::parallel_do_feeder& feeder ) const
{
      <do some work>
      if( <another item produced> )
         feeder.add( <the new item> );
};
```

Parallel pipeline

Linear pipeline of stages

- You specify maximum number of items that can be in flight

Each stage can be serial or parallel

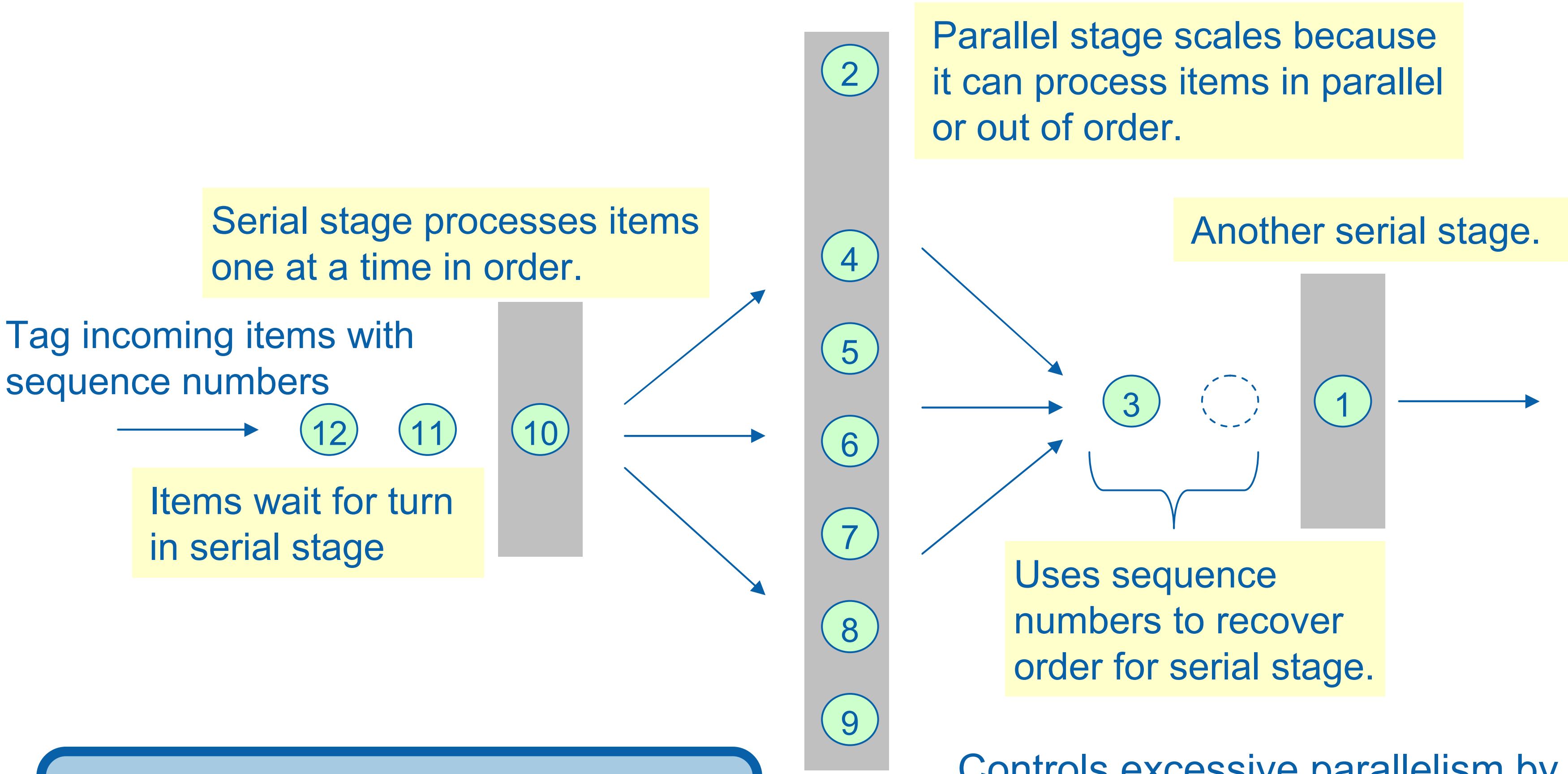
- Serial stage processes one item at a time, in order.
- Parallel stage can process multiple items at a time, out of order.

Uses cache efficiently

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

Functional decomposition is usually not scalable. It's the parallel stages that make tbb::pipeline scalable.

Parallel pipeline



Throughput limited by throughput of the slowest serial stage.

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

Summary of TBB Parallel Algorithms

Generic programming (not STL) is starting point

- C++ is language of choice for generic programming.
- Lambdas make it better

Explicit parallelism

- A little education goes a long way
- Programmer specifies logical parallelism
- Library maps parallelism to the machine

Three algorithms based on recursively divisible ranges

- parallel_for, parallel_reduce, parallel_scan

Grains of serial code provide the bricks

Not much new here – popularizing the classics!

Concurrent Containers

Intel® TBB provides concurrency-friendly containers

- STL containers are unsafe under concurrent operations
  - attempting concurrent modifications 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 for efficient, short term contention

- Worse single-thread performance, but better scalability.
- Can be used with TBB, OpenMP, or native threads.
- STL-compatible interfaces also provided, documented as not thread-safe

Concurrency-Friendly Interfaces

Some STL interfaces are inherently not concurrency-friendly

For example, suppose two threads share an STL queue:

```
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.
- No global guarantees

Two kinds of pops

- Blocking: pop()
- Non-blocking: pop_if_present()

Method size() returns *signed* integer

- If size() returns –*n*, it means *n* pops await corresponding pushes.

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

concurrent_vector<T>

Dynamically growable array of *T*

- grow_by(n)
- grow_to_at_least(n)

Elements are not moved when vector grows

- Can concurrently access and grow
- Some methods are **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_map<Key,T,HashCompare>

Associative table that maps a *Key* to an element of type *T*

- *HashCompare* is a class that specifies how keys are hashed and compared

Allows concurrent access for reads and updates

- bool insert(accessor &result, const Key &key) to add or edit
- bool find(accessor &result, const Key &key) to edit
- bool find(const_accessor &result, const Key &key) to look up
- bool erase(const Key &key) to remove

Lifetime of accessor object delimits extent of the access

Reader locks coexist – writer locks are exclusive

Platform-Independent Thread Wrapper

Implementation of the thread class recently-proposed to standardize.

Motivation: Many requests from community and customers

- Task-based parallelism is great, but what if I really need a thread?
- Why should I need to learn both TBB and pthreads or winthreads?

Allows explicit thread creation for:

- GUI, file I/O or network interface threads.
- Threads that need to wait on external events.
- Programs that previously needed to use both threads and Intel® TBB tasks

Makes threaded code more portable across platforms

- Easier to later migrate to ISO C++200x threads

WARNING: If you use threads, you may have all of the oversubscription problems that tasks shield you from.

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
- Uses the highest resolution wall-clock which is consistent between different threads.

Correctness Debugging of TBB programs

Debug single-threaded version first!

```
task_scheduler_init init(1);
```

Compile with macro TBB_DO_ASSERT=1 to enable checks in the header/inline code

Compile with TBB_DO_THREADING_TOOLS=1 to enable hooks for Intel's Threading Analysis tools

- Intel® Thread Checker can detect potential race conditions

Link with libtbb_debug.* to enable internal checking

Performance Debugging

Study scalability by using explicit thread count argument.

```
task_scheduler_init init(number_of_threads);
```

Compile with TBB_DO_ASSERT=0 to disable checks in the header/inline code.

Compile with TBB_DO_THREADING_TOOLS=1 to enable hooks for Intel's Threading Analysis tools.

- Intel® Thread Profiler can detect performance bottlenecks

Link with libtbb.* to get optimized library.

The tick_count class offers convenient timing services.

- uses the highest resolution wall clock consistent between different threads.

Future Direction – Lambda Friendly Interfaces

Example: parallel_reduce

- Current *body* argument encapsulates 3 pieces of information:
  - How to initialize processing for a subrange
  - How to process a leaf subrange
  - How to merge results
- Lambda friendly version (already available in latest updates!)
  - parallel_reduce(*range, init, body, reduction [, partitioner]);*

init: $\rightarrow$ Value

body: Range $\times$ Value $\rightarrow$ Value

reduction: Value $\times$ Value $\rightarrow$ Value

☹ Losing some in-place efficiency. Maybe rvalue references help?

Future Direction – Add STL Style Interfaces

☺ Familiar interface

☹ Often inefficient (blocking and fusion issues)

Examples

- parallel_for_each(*first*,*last*,*func*)

```
void ParallelApplyFoo(float a[], size_t n ) {
   parallel_for_each( a, a+n, [=](float x) {Foo(x);} );
}
```

Note: This particular example can be done in TBB 2.1 via parallel_do.

- parallel_accumulate(*first*,*last*,*identity*,*binaryOp*)
- parallel_partial_sum(*first*,*last*,*result*,*identity*,*binaryOp*)
- ?

Wish List

Divide and conquer template

Practical fusion of algorithms via concept axioms

Practical library-only solution for stencil algorithms

Even better support for I/O mixed with computations

More and better concurrent containers (incl. non-blocking)

Better cooperation with other Intel's parallel tools

Reap benefits provided by C++ 200x

Real time? Low power?

...

Summary of Intel® Threading Building Blocks

It is a *library*

You specify *task patterns*, not threads

Targets threading for *robust performance*

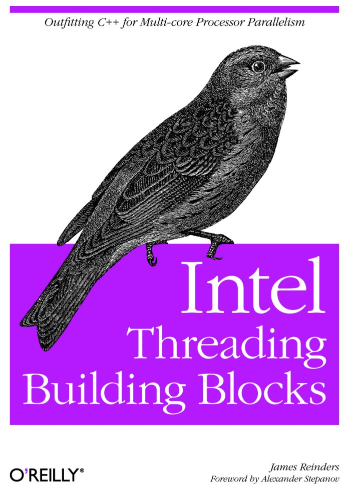
Does well with *nested parallelism*

Compatible with other threading packages

Emphasizes *scalable, data parallel* programming

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

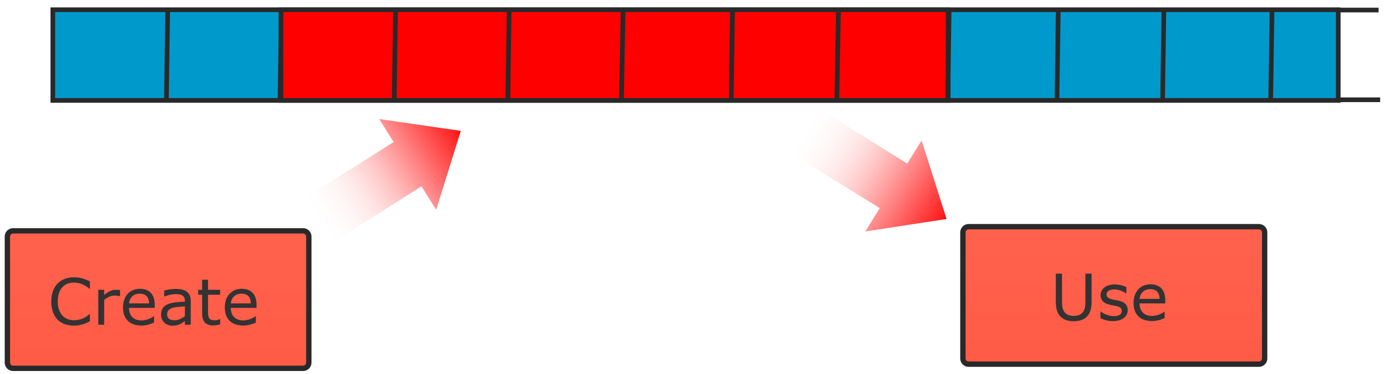
Available in open source version under GPL, as well as commercially licensed.



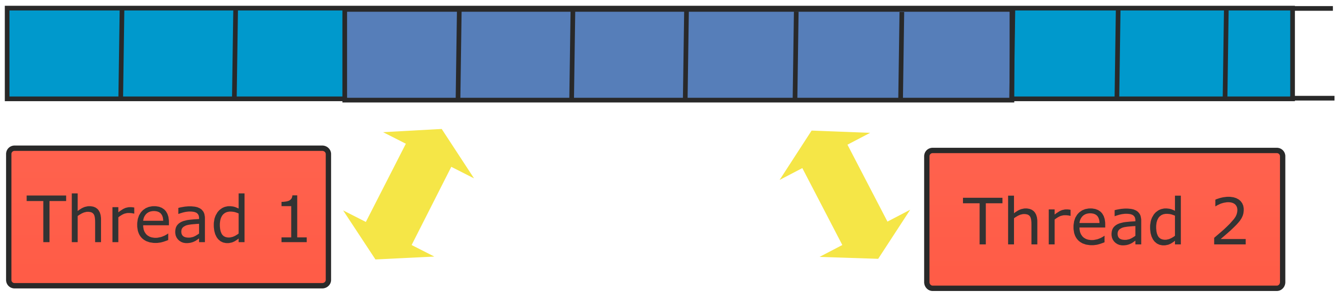
Backup

Cache efficiency

Working on data, which is hot in cache, is more efficient



Data eviction can introduce noticeable penalty



Key points about Intel® Threading Building Blocks

- It is a *template library* intended to ease parallel programming for C++ developers
  - Relies on generic programming to deliver high performance parallel algorithms with broad applicability
- It provides a *high-level abstraction for parallelism*
  - Shifts focus from workers (threads) to the work
  - Hides low level details of thread management
  - Fully supports nested parallelism
- It facilitates *scalable performance*
  - Designed for CPU bound computation
  - Strives for efficient use of cache, and balances load
  - Emphasizes data parallel programming as opposed to non-scalable functional decomposition
- It works across a variety of machines today, and *readies programs for tomorrow*
  - Also can be used in concert with other threading packages such as native threads and OpenMP.

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

Scalability

Ideally you want Performance $\propto$ Number of hardware threads

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 partial ordering of tasks to avoid the need for locks.
  - Provide efficient atomic operations and locks if you really need them.

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.