Intel® Threading Building Blocks

Software and Services Group
Intel Corporation

Agenda

Introduction

Intel® Threading Building Blocks overview

Tasks concept

Generic parallel algorithms

Task-based Programming

Performance Tuning

Parallel pipeline

Concurrent Containers

Scalable memory allocator

Synchronization Primitives

Parallel models comparison

Summary

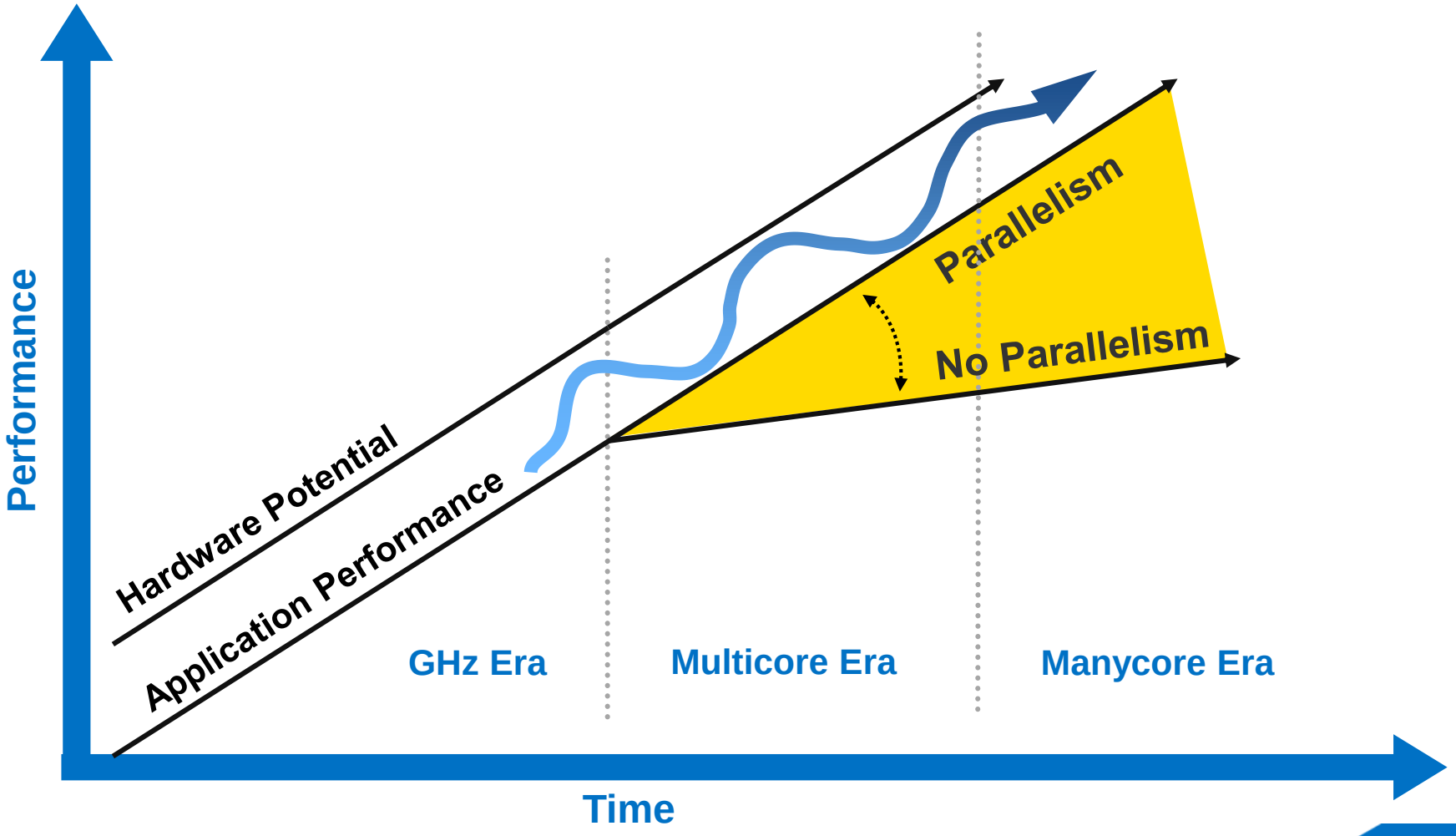
Agenda

Introduction

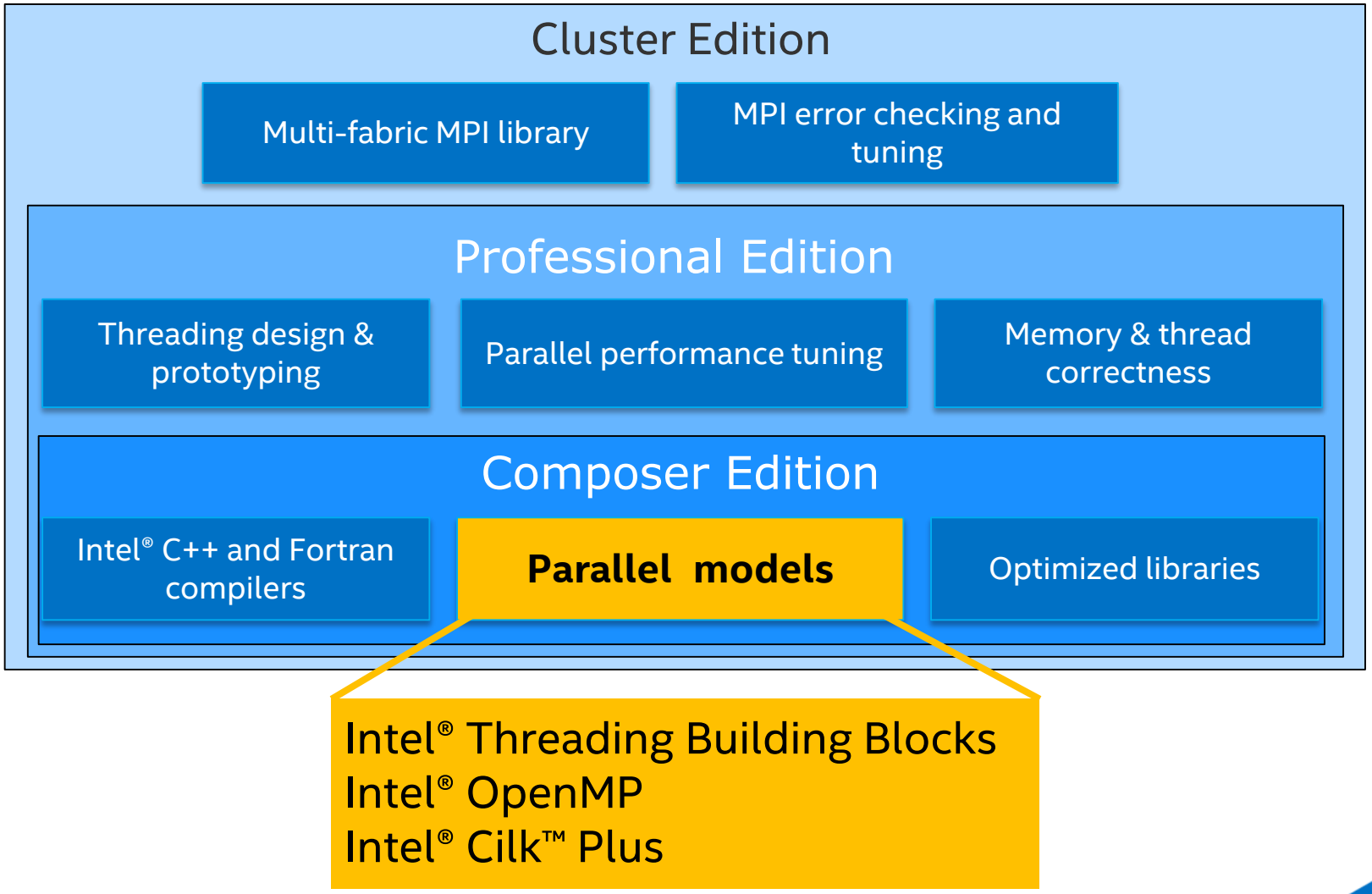
- Intel® Threading Building Blocks overview
- Tasks concept
- Generic parallel algorithms
- Task-based Programming
- Performance Tuning
- Parallel pipeline
- Concurrent Containers
- Scalable memory allocator
- Synchronization Primitives
- Parallel models comparison
- Summary

Parallel challenges

"Parallel hardware needs parallel programming"



Intel® Parallel Studio XE 2015



A key ingredient in Intel® Parallel Studio XE

Try it for 30 days free: http://intel.ly/perf-tools

	Intel® Parallel Studio XE Composer Edition[1]	Intel® Parallel Studio XE Professional Edition[1]	Intel® Parallel Studio XE Cluster Edition
Intel® C++ Compiler	√	√	√
Intel® Fortran Compiler	√	√	√
Intel® Threading Building Blocks (C++ only)	√	√	√
Intel® Integrated Performance Primitives (C++ only)	√	√	√
Intel® Math Kernel Library	√	√	√
Intel® Cilk™ Plus (C++ only)	√	√	√
Intel® OpenMP*	√	√	√
Rogue Wave IMSL* Library[2] (Fortran only)	Add-on	Add-on	Bundled and Add-on
Intel® Advisor XE		√	√
Intel® Inspector XE		√	√
Intel® VTune™ Amplifier XE[4]		√	√
Intel® MPI Library[4]			√
Intel® Trace Analyzer and Collector			√
Operating System (Development Environment)	Windows* (Visual Studio*) Linux* (GNU) OS X*[3] (XCode*)	Windows (Visual Studio) Linux (GNU)	Windows (Visual Studio) Linux (GNU)

Notes:
[1] Available with a single language (C++ or Fortran) or both languages.
[2] Available as an add-on to any Windows Fortran* suite or bundled with a version of the Composer Edition.
[3] Available as single language suites on OS X.
[4] Available bundled in a suite or standalone

Agenda

Introduction

Intel® Threading Building Blocks overview

Tasks concept

Generic parallel algorithms

Task-based Programming

Performance Tuning

Parallel pipeline

Concurrent Containers

Scalable memory allocator

Synchronization Primitives

Parallel models comparison

Summary

Intel® Threading Building Blocks (Intel® TBB)

What

- Widely used C++ template library for task parallelism.
- Features
- Parallel algorithms and data structures.
- Threads and synchronization primitives.
- Scalable memory allocation and task scheduling.

Also available as open source at threadingbuildingblocks.org

https://software.intel.com/intel-tbb

Benefit

- Rich feature set for general purpose parallelism.
- Available as an open source and a commercial license.
- Supports C++, Windows*, Linux*, OS X*, other OS's.
- Commercial support for Intel® Atom™, Core™, Xeon® processors, and for Intel® Xeon Phi™ coprocessors

Simplify Parallelism with a Scalable Parallel Model

Didn't we solve the Threading problem in the 1990s?

Pthreads standard: IEEE 1003.1c-1995
OpenMP standard: 1997

Yes, **but**…

- How to split up work? How to keep caches hot?
- How to balance load between threads?
- What about nested parallelism (call chain)?

Programming with threads is HARD

- Atomicity, ordering, and/vs. scalability
- Data races, dead locks, etc.

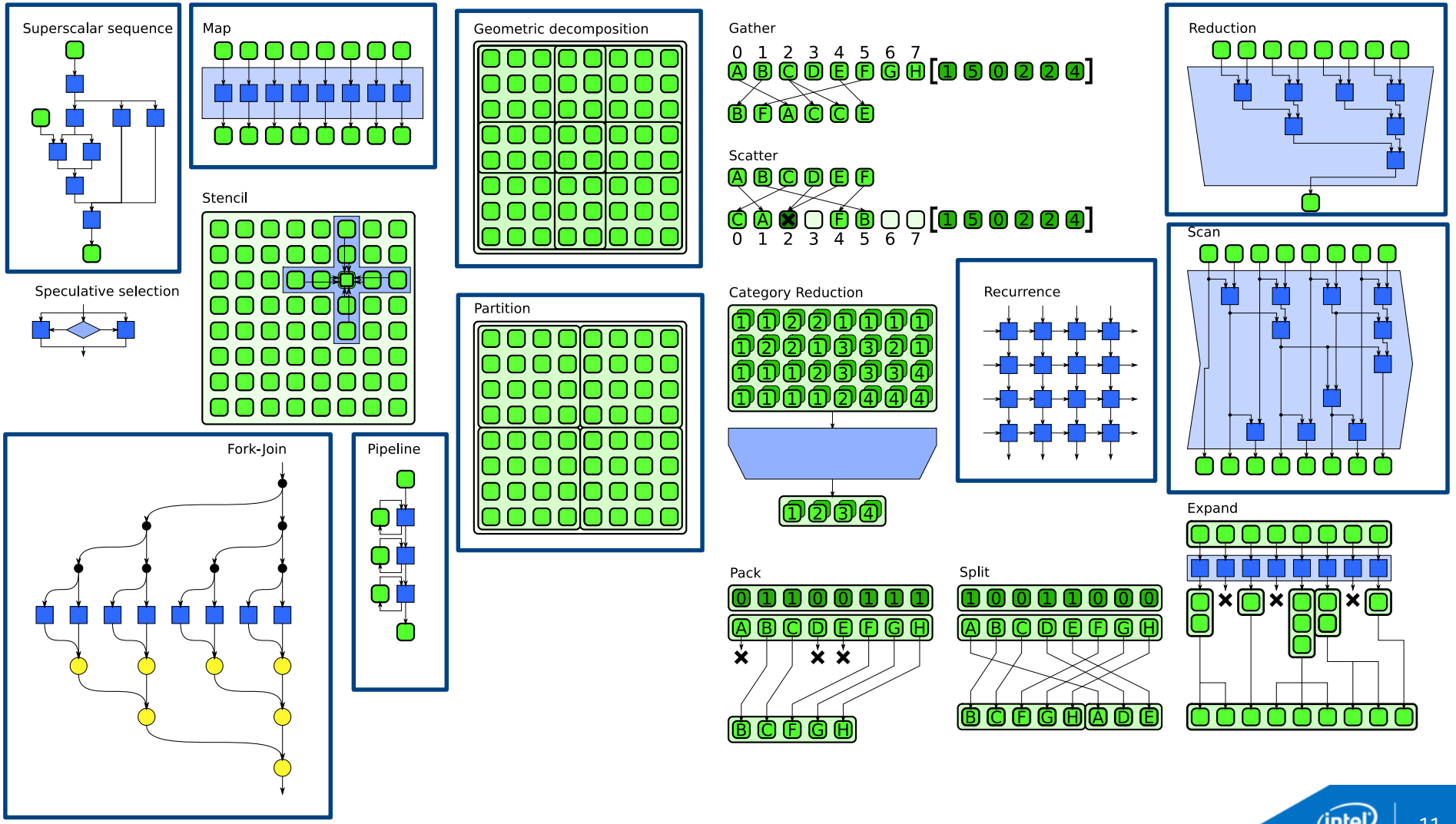
Design patterns

- Parallel pattern: commonly occurring combination of task distribution and data access
- A small number of patterns can support a wide range of applications

→ **Identify and use parallel patterns**
Examples: reduction, or pipeline

→ **TBB has primitives and algorithms for most common patterns** – don't reinvent a wheel

Parallel Patterns



Parallel algorithms and data structures

Threads and synchronization

Memory allocation and task scheduling

Rich Feature Set
for Parallelism

Generic Parallel Algorithms

Efficient scalable way to exploit the power of multi-core without having to start from scratch.

Flow Graph

A set of classes to express parallelism as a graph of compute dependencies and/or data flow

Concurrent Containers

Concurrent access, and a scalable alternative to containers that are externally locked for thread-safety

Synchronization Primitives

Atomic operations, a variety of mutexes with different properties, condition variables

Task Scheduler

Sophisticated work scheduling engine that empowers parallel algorithms and the flow graph

Timers and Exceptions

Thread-safe timers and exception classes

Threads

OS API wrappers

Thread Local Storage

Efficient implementation for unlimited number of thread-local variables

Memory Allocation

Scalable memory manager and false-sharing free allocators

Features and Functions List

Parallel algorithms and data structures

Threads and synchronization

Memory allocation and task scheduling

Generic Parallel Algorithms

- parallel_for
- parallel_reduce
- parallel_for_each
- parallel_do
- parallel_invoke
- parallel_sort
- parallel_deterministic_reduce
- parallel_scan
- parallel_pipeline
- pipeline

Flow Graph

- graph
- continue_node
- source_node
- function_node
- multifunction_node
- overwrite_node
- write_once_node
- limiter_node
- buffer_node
- queue_node
- priority_queue_node
- sequencer_node
- broadcast_node
- join_node
- split_node
- indexer_node

Concurrent Containers

- concurrent_unordered_map
- concurrent_unordered_multimap
- concurrent_unordered_set
- concurrent_unordered_multiset
- concurrent_hash_map
- concurrent_queue
- concurrent_bounded_queue
- concurrent_priority_queue
- concurrent_vector
- concurrent_lru_cache

Synchronization Primitives

- atomic
- mutex
- recursive_mutex
- spin_mutex
- spin_rw_mutex
- speculative_spin_mutex
- speculative_spin_rw_mutex
- queuing_mutex
- queuing_rw_mutex
- null_mutex
- null_rw_mutex
- reader_writer_lock
- critical_section
- condition_variable
- aggregator (preview)

Task Scheduler

- task
- task_group
- structured_task_group
- task_group_context
- task_scheduler_init
- task_scheduler_observer
- task_arena

Exceptions

- tbb_exception
- captured_exception
- movable_exception

Threads & timers

Thread

tick_count

Thread Local Storage

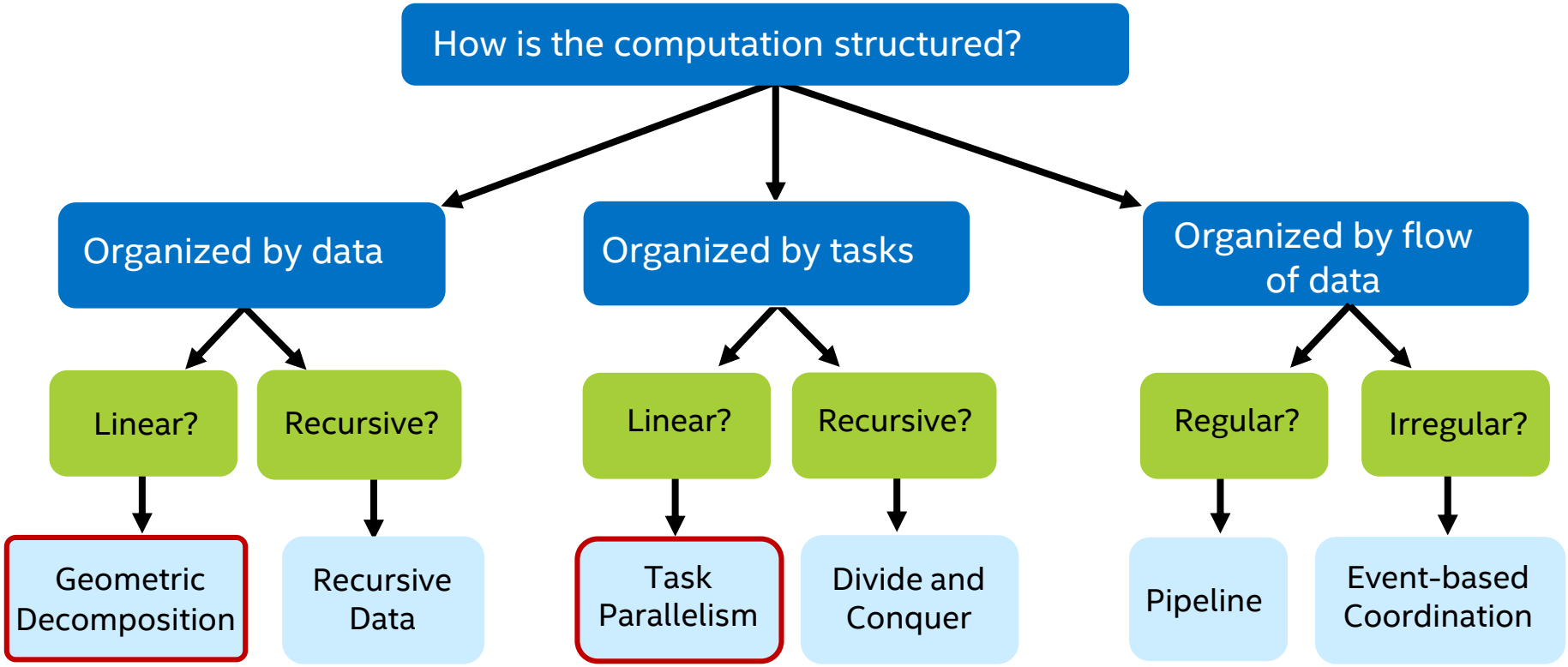
- combinable
- enumerable_thread_specific

Memory Allocation

- tbb_allocator
- scalable_allocator
- cache_aligned_allocator
- zero_allocator
- aligned_space
- memory_pool (preview)

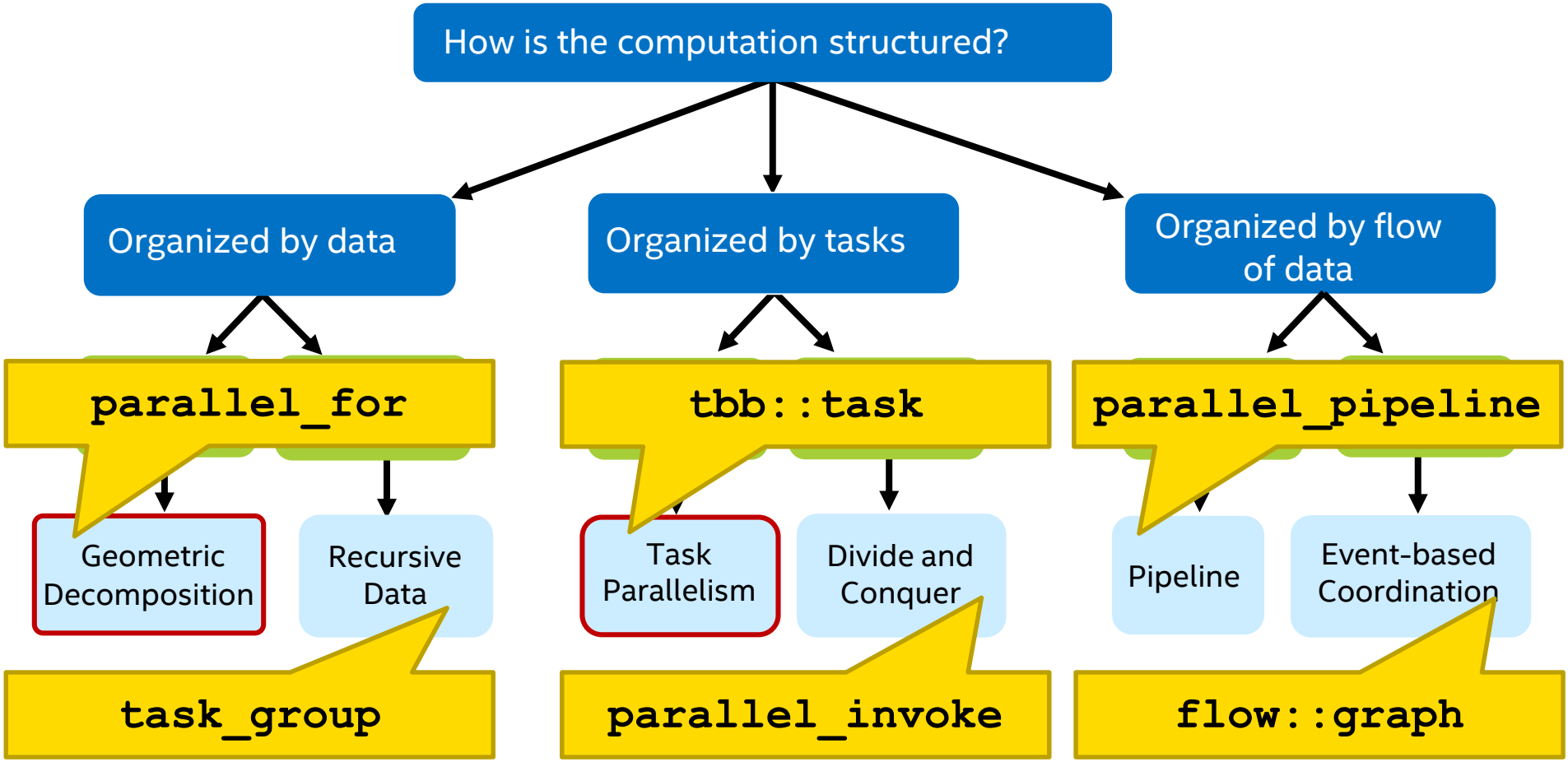
Algorithm Structure Design Space

Structure used to organize parallel computations



Algorithm Structure Design Space

Structure used to organize parallel computations



Agenda

Introduction

Intel® Threading Building Blocks overview

Tasks concept

Generic parallel algorithms

Task-based Programming

Performance Tuning

Parallel pipeline

Concurrent Containers

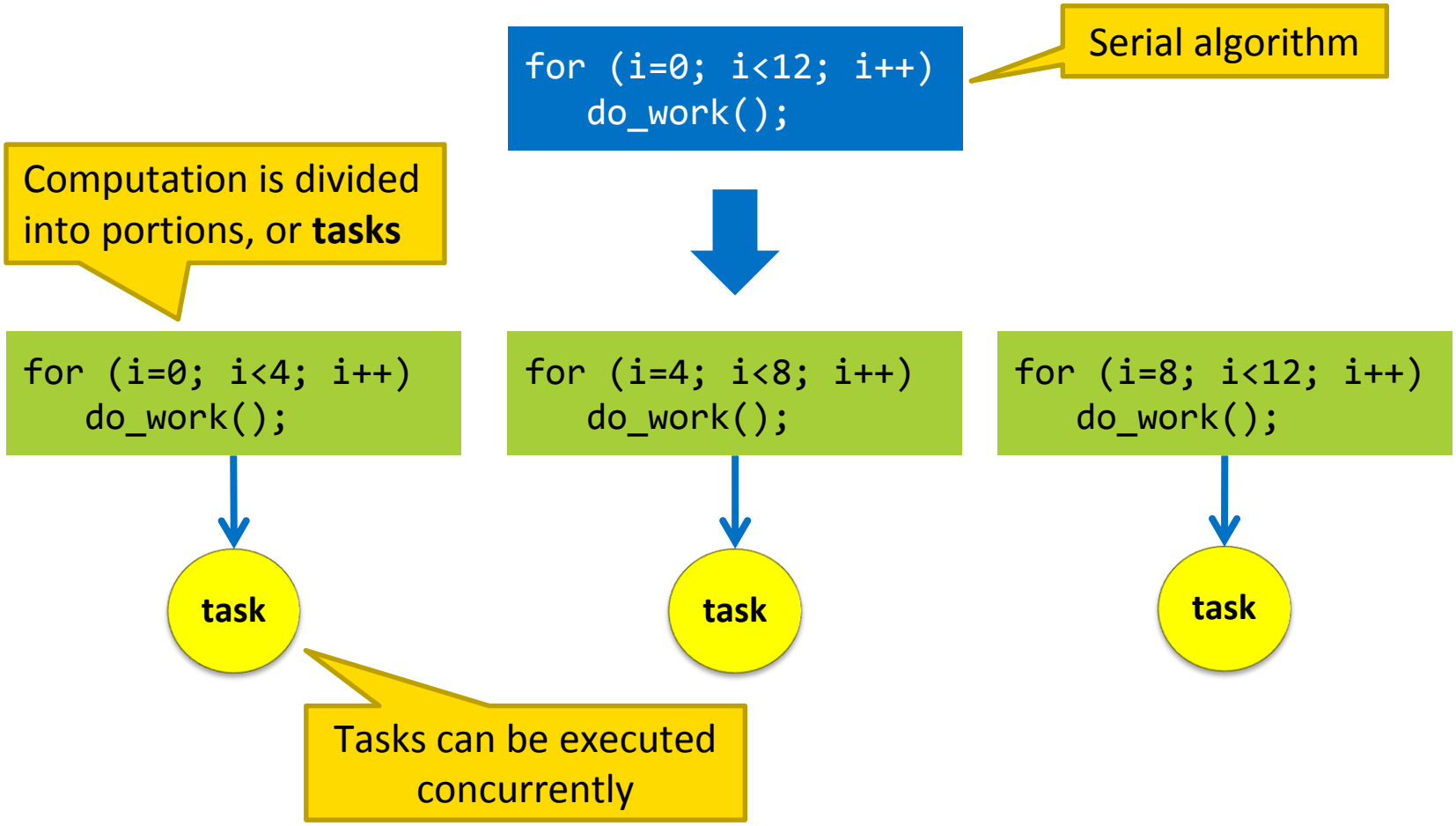
Scalable memory allocator

Synchronization Primitives

Parallel models comparison

Summary

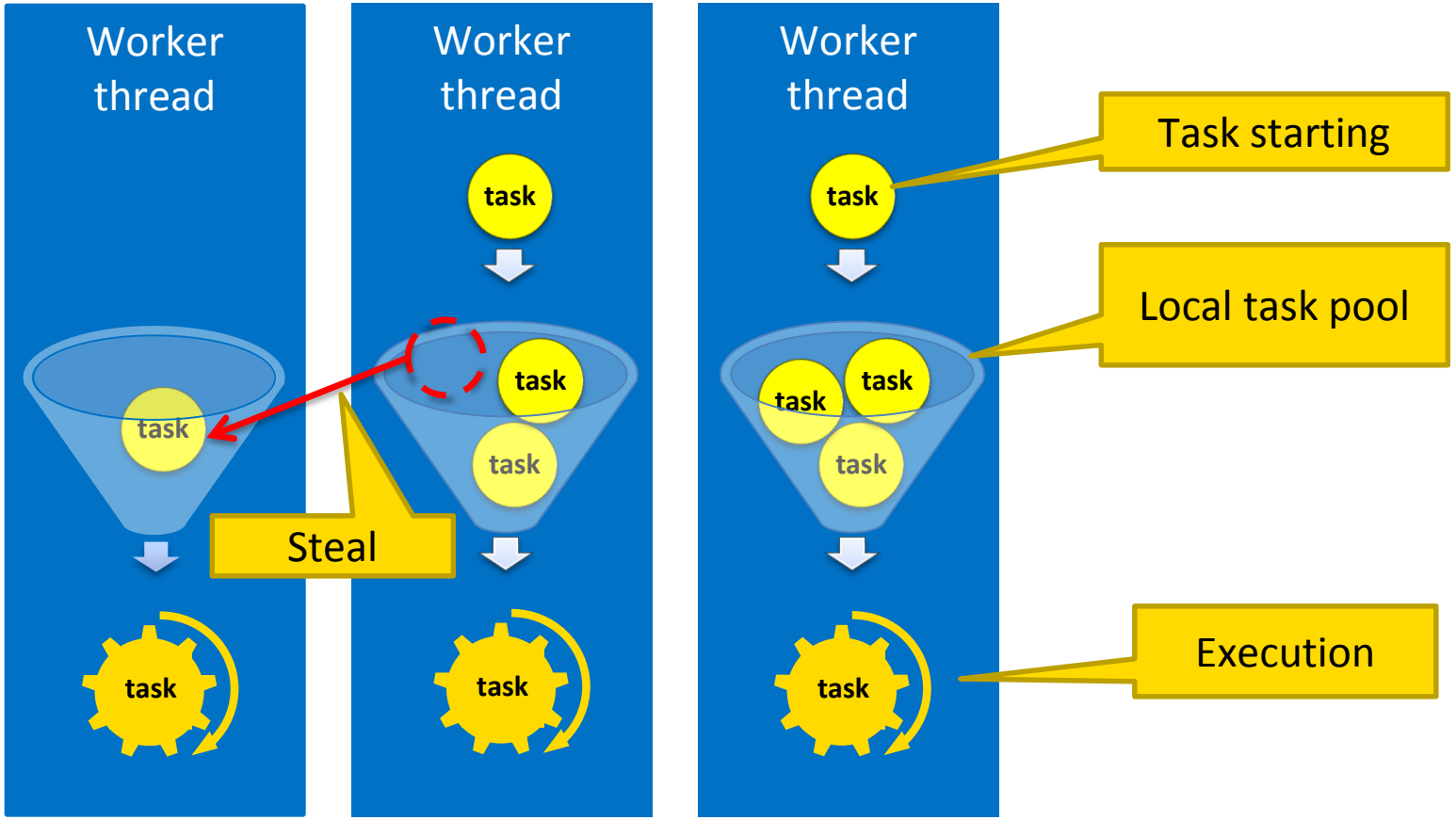
Tasks and parallel algorithms



Task Execution

Intel TBB runtime dynamically maps tasks to threads

Automatic load balance, not fair, lock-free whenever possible



Agenda

Introduction

Intel® Threading Building Blocks overview

Tasks concept

Generic parallel algorithms

Task-based Programming

Performance Tuning

Parallel pipeline

Concurrent Containers

Scalable memory allocator

Synchronization Primitives

Parallel models comparison

Summary

Generic parallel algorithms

Loop parallelization

parallel_for

parallel_reduce

- load balanced parallel execution
- fixed number of independent iterations

parallel_deterministic_reduce

- run-to-run reproducible results

parallel_scan

- computes parallel prefix

y[i] = y[i-1] op x[i]

Parallel sorting

parallel_sort

Computational graph

flow::graph

- Implements dependencies between tasks
- Pass messages between tasks

Parallel Algorithms for Streams

parallel_do

- Use for unstructured stream or pile of work
- Can add additional work to pile while running

parallel_for_each

- parallel_do without an additional work feeder

pipeline / parallel_pipeline

- Linear pipeline of stages
- Each stage can be parallel or serial in-order or serial out-of-order.
- Uses cache efficiently

Parallel function invocation

parallel_invoke

task_group

- Parallel execution of a number of user-specified functions

Let's start with simple example

```
#include <algorithm>
#define N 10

int main (){
    float a[N];
    // initialize array here…

    for (int i=0; i<N; i++ ){
            a[i] = sqrt(a[i]);
    }
    return 0;
}
```

and implement with std algorithm:

```
std::for_each( Iterator, Iterator, Func );
```

std::for_each

```
#include <vector>
#include <algorithm>
#define N 10

int main (){
   int a[N];
   // initialize array here…
   std::vector<float> array;

   for (int i=0; i<N; i++){
      array.push_back(a[i]);
   }

   std::for_each (array.begin(), array.end(),
      [=](float & elem) {
         sqrt(elem);
      });
   return 0;
}
```

A call to a template function **for_each<Iterator, Iterator, Func>**: with arguments
Iterator → array.begin
Iterator → array.end
Operation → Lambda or a functor

Interval

Loop body as C++ lambda expression

tbb::parallel_for_each

```
#include <vector>
#include <tbb/blocked_range.h>
#include <tbb/parallel_for_each.h>
#define N 10

int main (){
int a[N];
   // initialize array here…
   std::vector<float> array;

   for (int i=0; i<N; i++){
      array.push_back(a[i]);
   }

   tbb::parallel_for_each (array.begin(), array.end(),
      [=](float & elem) {
          sqrt(elem);
      });
   return 0;
}
```

parallel_for_each<Iterator, Iterator, Func>: created several tasks with lambda body invocation and the runtime executes them in parallel

Task: loop body as C++ lambda expression

A simple computational example

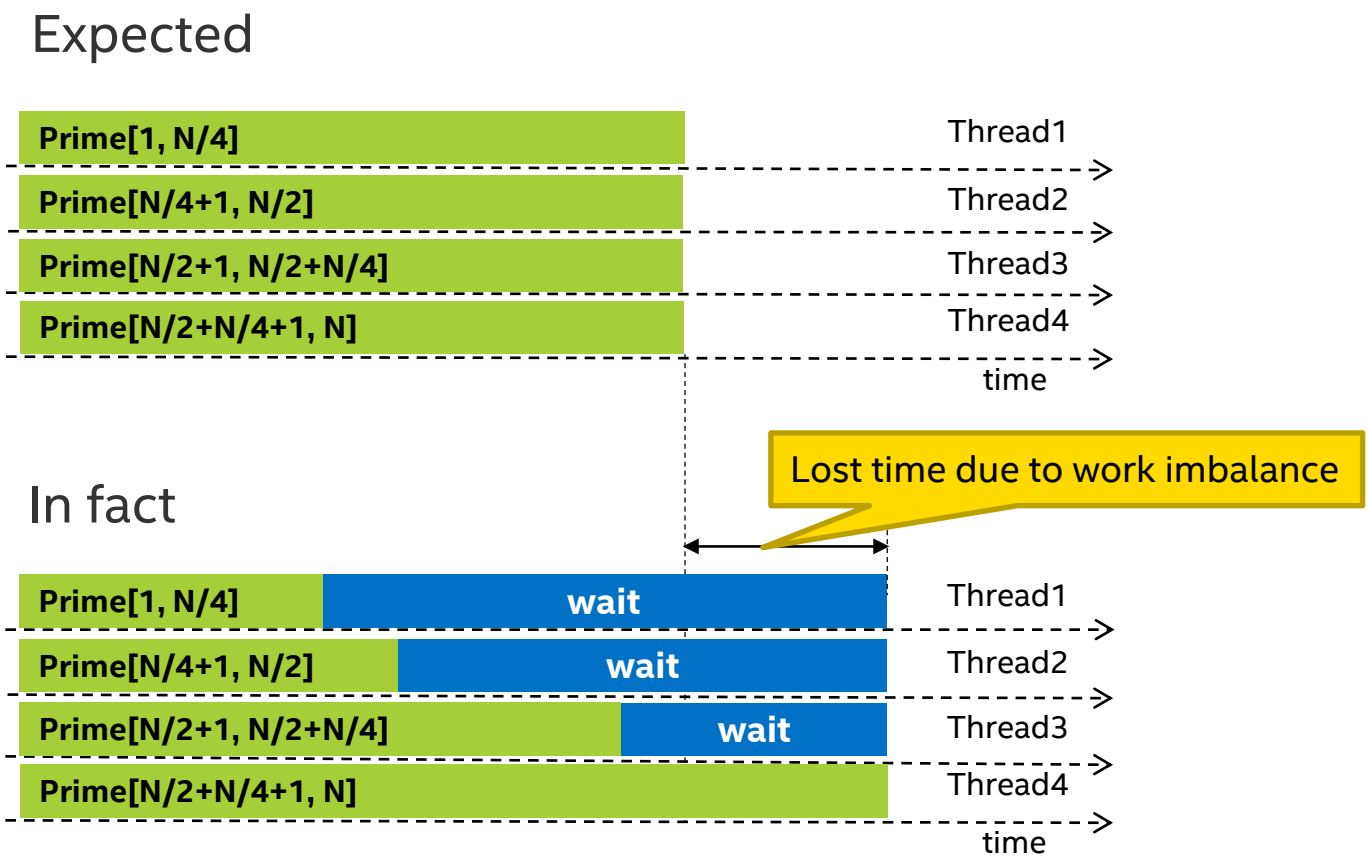
```
#include <algorithm>
#define N 10

inline int Prime(int & x) {
   int limit, factor = 3;
   limit = (long)(sqrtf((float)x)+0.5f);
   while( (factor <= limit) && (x % factor))
      factor ++;
   x = (factor > limit ? x : 0);
}

int main (){
   int a[N];
   // initialize array here…
   for (int i=0; i<N; i++ ){
      Prime(a[i]);
   }
   return 0;
}
```

Some non-trivial computation on an integer element

Applying the computation Prime() to each element of the array

Try to parallelize it, and then check work balance ☺

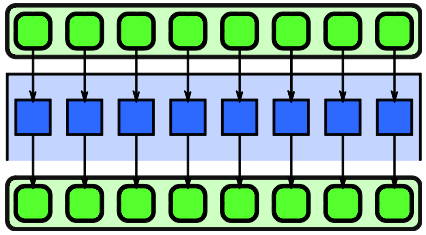
Parallel execution with equal range



Reality is even worse due to OS scheduling ☹

tbb::parallel_for

Map



Has several forms.

Execute *functor*(*i*) for all *i* ∈ [*lower*,*upper*)

parallel_for(*lower, upper, functor*);

Execute *functor*(*i*) for all *i* ∈ {*lower*,*lower*+*stride*,*lower*+2**stride*,...}

parallel_for(*lower, upper, stride, functor*);

Execute *functor*(*subrange*) for all *subrange* in *range*

parallel_for(*range, functor*);

tbb::parallel_for

```
#include <tbb/blocked_range.h>
#include <tbb/parallel_for.h>
#define N 10

inline int Prime(int & x) {
   int limit, factor = 3;
   limit = (long)(sqrtf((float)x)+0.5f);
   while( (factor <= limit) && (x % factor))
      factor ++;
   x = (factor > limit ? x : 0);
}

int main (){
   int a[N];
   // initialize array here…
   tbb::parallel_for (0, N, 1,
      [&](int i){
         Prime (a[i]);
      });
   return 0;
}
```

A call to a template function **parallel_for** (lower, upper, stride, functor)

Task: loop body as C++ lambda expression

tbb::parallel_for

```
#include <tbb/blocked_range.h>
#include <tbb/parallel_for.h>
#define N 10

inline int Prime(int & x) {
   int limit, factor = 3;
   limit = (long)(sqrtf((float)x)+0.5f);
   while( (factor <= limit) && (x % factor))
      factor ++;
   x = (factor > limit ? x : 0);
}
int main (){
   int a[N];
   // initialize array here…
   tbb::parallel_for (tbb::blocked_range<size_t>(0,N,1),
      [&](const tbb::blocked_range<size_t>& r){
         for (int i=r.begin(); i!=r.end(); i++ ){
            Prime (a[i]);
         }
      });
   return 0;
}
```

A call to a template function **parallel_for** (range, functor)

blocked_range – TBB template representing 1D iteration space

tbb::parallel_for

```
#include <tbb/blocked_range.h>
#include <tbb/parallel_for.h>
#define N 10

class ChangeArray{
    int* array;
public:
    ChangeArray (int* a): array(a) {}
    void operator()(const tbb::blocked_range<size_t>& r ) const{
        for (int i=r.begin(); i!=r.end(); i++ ){
            Prime (array[i]);
        }
    }
};

int main (){
    int a[N];
    // initialize array here…
    tbb::parallel_for (tbb::blocked_range<size_t>(0,N,1), ChangeArray(a));
    return 0;
}
```

```
inline int Prime(int & x) {
    int limit, factor = 3;
    limit = (long)(sqrtf((float)x)+0.5f);
    while( (factor <= limit) && (x % factor))
        factor ++;
    x = (factor > limit ? x : 0);
}
```

Functor

The main work is done inside **operator()**
operator() body is the **task** that is invocated multiple times in parallel

A call to a template function
parallel_for (range, functor)

Loop body as functor

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

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 partitions original range into subranges, and deals out subranges to worker threads in way that:

- Balances load
- Uses cache efficiently
- Scales

Range concept

Represents a recursively divisible set of values

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 models:

blocked_range models a one-dimensional range

blocked_range2d models a two-dimensional range

blocked_range3d models a three-dimensional range

You can define your own ranges

Grain Size concept

blocked_range specifies **grainsize** of type size_t

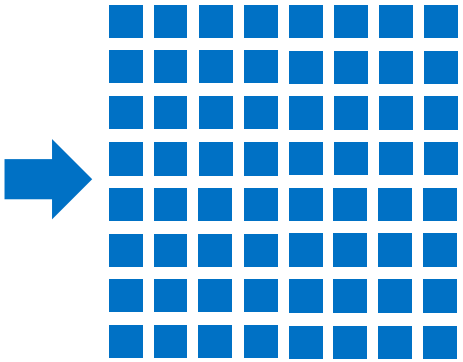
Part of parallel_for, not underlying task scheduler

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

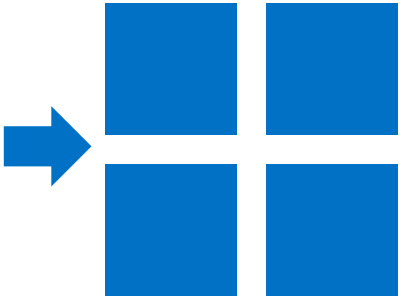
Typically only have to get it right within an order of magnitude

- If too fine, scheduling overhead dominates
- If too coarse, lose some potential parallelism
- When in doubt, err on the side of making it too large

Too fine scheduling overhead dominates



Too coarse lose potential parallelism



Partitioner – optional argument

Recurse all the way down *range*

```
tbb::parallel_for( range, functor, tbb::simple_partitioner() );
```

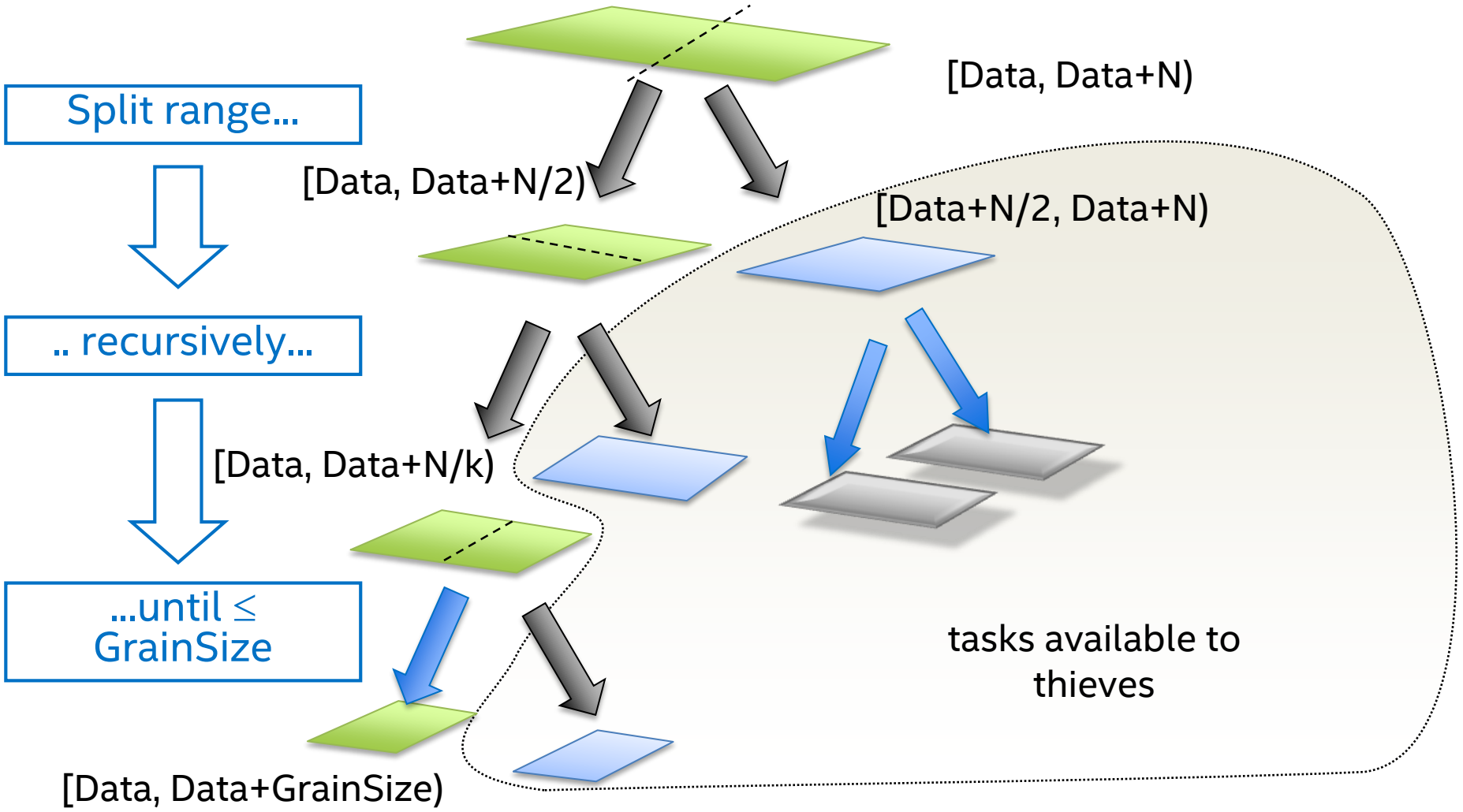
Uses heuristics to dynamically adjust recursion depth

```
tbb::parallel_for( range, functor, tbb::auto_partitioner() );
```

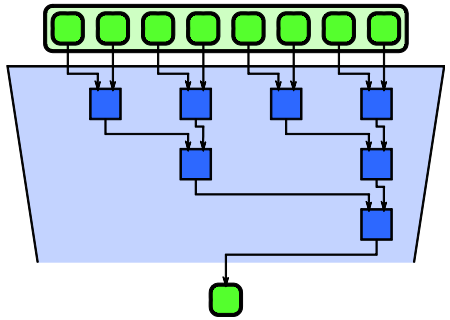
Replay with cache optimization

```
tbb::parallel_for( range, functor, affinity_partitioner() );
```

Recursive parallelism



Reduction pattern



Using thread local data:

```
enumerable_thread_specific<T> globalVar;
localCopy = globalVar.local();
```

Using special template function:

Lambda-friendly: with identity value and separate functors

```
parallel_reduce( range, identity, functor, combine );
```

"Simpler" form that requires a special body class

```
parallel_reduce( range, body );
```

Example: Thread-local Storage (TLS)

```
#include "tbb/enumerable_thread_specific.h"
#include "tbb/parallel_for.h"

float array[] = { 1, 2, 3, 4, 5, 6, 7, 8 };
tbb::enumerable_thread_specific<float> tls(0.f);

int main (){
   tbb::parallel_for(0, 8, [&](int i) {
      float& accum = tls.local();
      accum += array[i];
   });
   float sum = 0;

   for (auto i = tls.begin(); i < tls.end(); ++i) {
      sum += *i;
   }
}
```

Thread local container

Local copy

Serial reduction

parallel_reduce: the imperative form

```
#include "tbb/blocked_range.h"
#include "tbb/parallel_reduce.h"

struct Sum {
    float value;
    Sum() : value(0) {}
    Sum( Sum& s, split ) {value = 0;}
    void operator()( const tbb::blocked_range<float*>& r ) {
        float temp = value;
        for( float* a=r.begin(); a!=r.end(); ++a ) {
            temp += *a;
        }
        value = temp;
    }
    void join( Sum& rhs ) {value += rhs.value;}
};

int ParallelSum ( float array[], size_t n ) {
   Sum total;
   tbb::parallel_reduce(tbb::blocked_range<float*>( array, array+n ), total );
   return total.value;
}
```

Setting initial value

Splitting constructor

Summing up elements in sub-range

Combine results from all bodies

Body class object

parallel_reduce: the functional form

```
#include <numeric>
#include <functional>
#include "tbb/blocked_range.h"
#include "tbb/parallel_reduce.h"
using namespace tbb;

float ParallelSum( float array[], size_t n ) {
    return parallel_reduce(
        blocked_range<float*>( array, array+n ),
        0.f,
        [](const blocked_range<float*>& r, float value)->float {
            return std::accumulate(r.begin(),r.end(),value);
        },
        std::plus<float>()
    );
}
```

Identity value

Initial value for reducing sub-range r

Reduce sub-range

ComCombining sub-range results

Determinism of reduction

For non-associative operations, parallel_reduce does not guarantee deterministic results

- Re-association of operations done differently
- Depends on the number of threads, the partitioner used, and on which ranges are stolen.

Solution: **parallel_deterministic_reduce**

- Uses deterministic reduction tree.
- Generates deterministic result even for floating-point.
  - But different from serial execution
- Partitioners are disallowed
- Specification of grainsize is highly recommended.

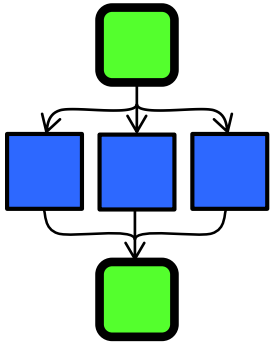
parallel_reduce: the functional form

```
sum =  tbb::parallel_deterministic_reduce (
    tbb::blocked_range<int>(0,n,10000),
    0.f,
    [&](tbb::blocked_range<int> r, T s) -> float
    {
        for( int i=r.begin(); i!=r.end(); ++i )
            s +=  a[i];
        return s;
    },
    std::plus<T>()
);
```

Grainsize

parallel_deterministic_reduce(*range*, *identity*, *functor*, *combine*);

Fork-Join pattern



Useful for *n*-way fork when *n* is a small constant.

parallel_invoke($functor_1$, $functor_2$, ...);

Useful for *n*-way fork when *n* is large or a run-time value.
Allows asynchronous execution.

```
task_group g;
...
g.run( functor1 );
...
g.run( functor2 );
...
g.wait();
```

Typical use cases for fork-join

Functional parallelism

- Easy to understand and use
- Does not scale on its own

Recursive divide-and-conquer problems

- Scales together with the problem
- Sometimes, can be alternatively expressed with parallel_for/_reduce and custom ranges

Example: Parallel Quicksort

```
template<typename I>
void parallel_qsort(I begin, I end){
   typedef typename std::iterator_traits<I>::value_type T;

   if (begin != end) {
      const I pivot = end - 1;
      const I middle = std::partition(begin, pivot,
         std::bind2nd(std::less<T>(), *pivot));
      std::swap(*pivot, *middle);

   tbb::parallel_invoke(
     parallel_qsort(begin, middle),
     parallel_qsort(middle + 1, end));
   }
}
```

parallel_invoke (const Func0&, const Func1&);

* Pure Quicksort (e.g., tail recursion is not avoided)

Example: tbb::parallel_sort

```
#include "tbb/parallel_sort.h“
#include <math.h>

const int N = 100000;
float a[N];
float b[N];

// initialize a and b here

parallel_sort(a, a + N);
parallel_sort(b, b + N, std::greater<float>());
```

parallel_sort (iterator, iterator);
parallel_sort (iterator, iterator, const Compare&);

complexity of *O(N log (N))*

Agenda

Introduction

Intel® Threading Building Blocks overview

Tasks concept

Generic parallel algorithms

Task-based Programming

Performance Tuning

Parallel pipeline

Concurrent Containers

Scalable memory allocator

Synchronization Primitives

Parallel models comparison

Summary

Task Scheduler

Terminology

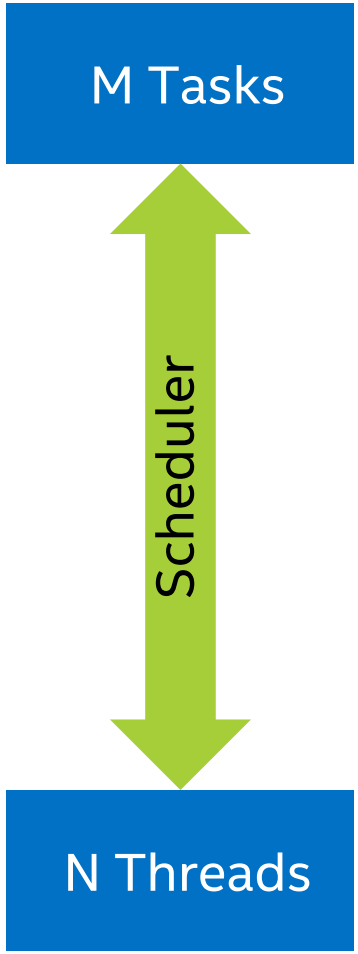
- **Thread** refers to a physical thread*
- **Task** refers to a piece of work

Scheduler

- Maps tasks to threads (M:N relation)
- Balances resource consumption and parallelism
- Runtime-dynamic and lock-free
- Essential component of Intel® TBB

Task queuing

1. LIFO: thread-local queue (spawned tasks)
2. ~FIFO: shared global queue (enqueued tasks)
3. ~FIFO: Task-stealing (random foreign queue)



* Regardless whether SMT is used or not.

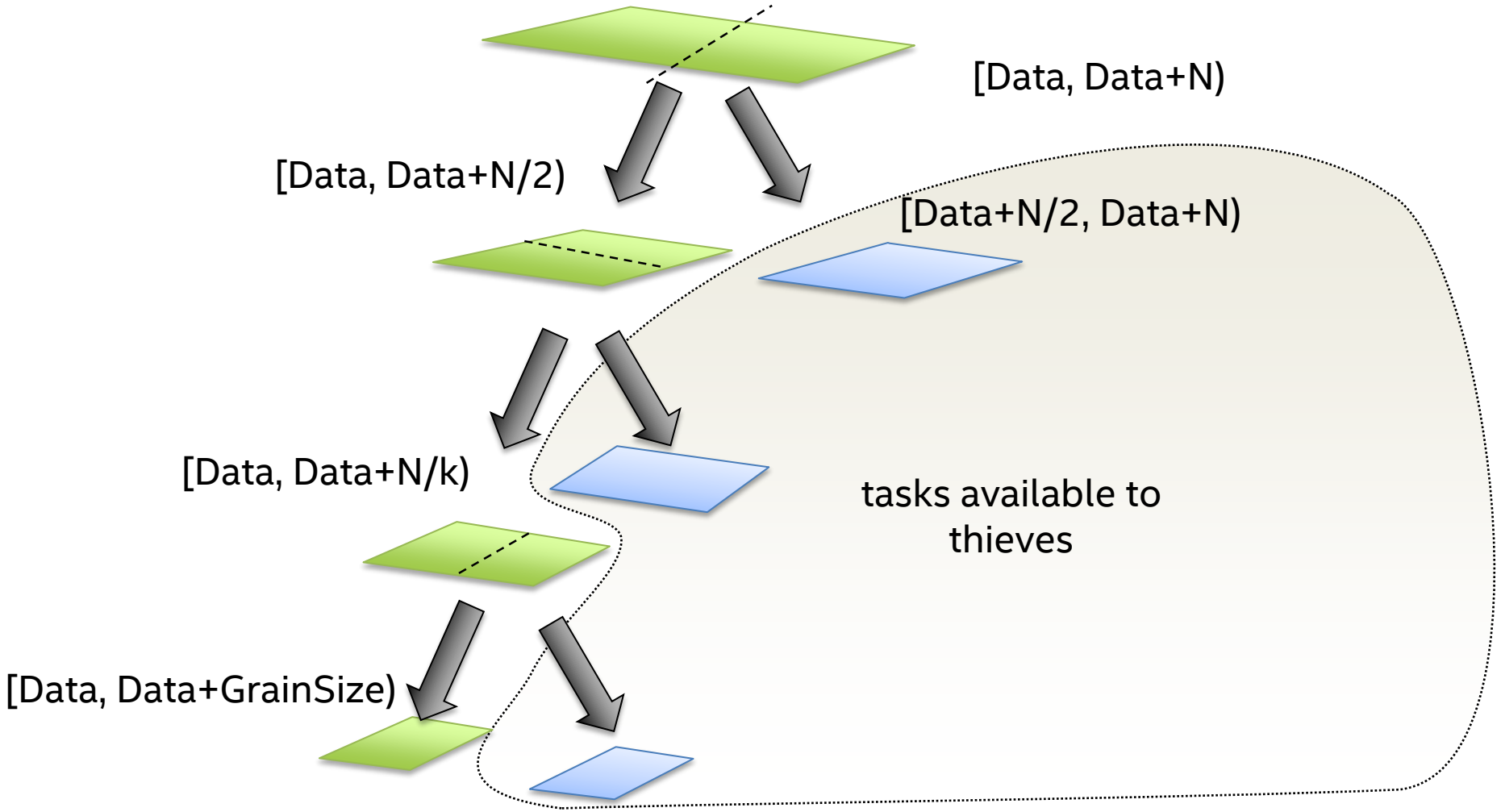
Work Stealing Task Scheduler:

how does it work?

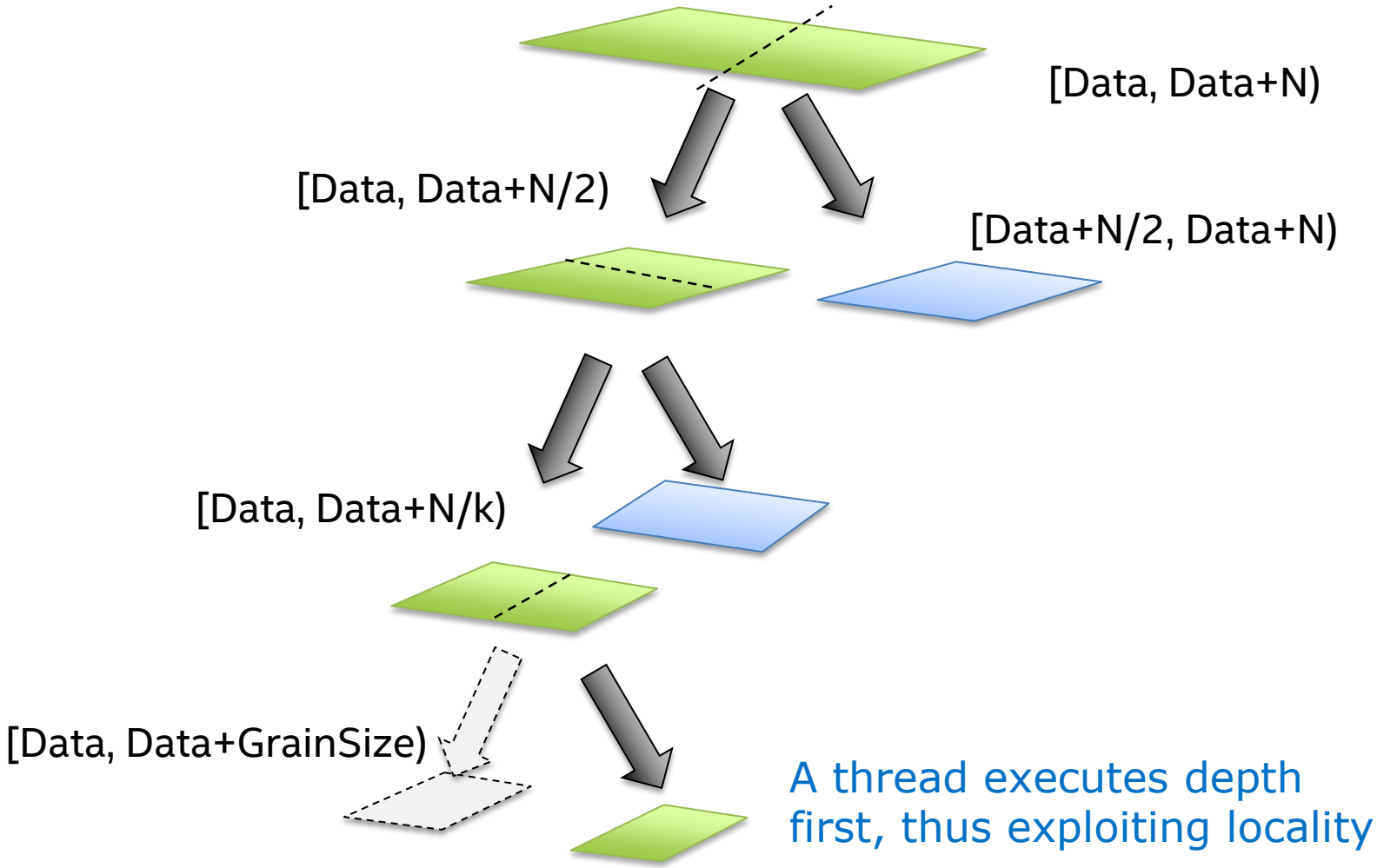
Each thread has a deque of tasks

- Newly created tasks are pushed onto the front
- When looking for tasks the thread pops from the front
- If it has no work
  - Pick a random victim
  - Attempt to steal a task from the back of their deque

Recursive parallelism

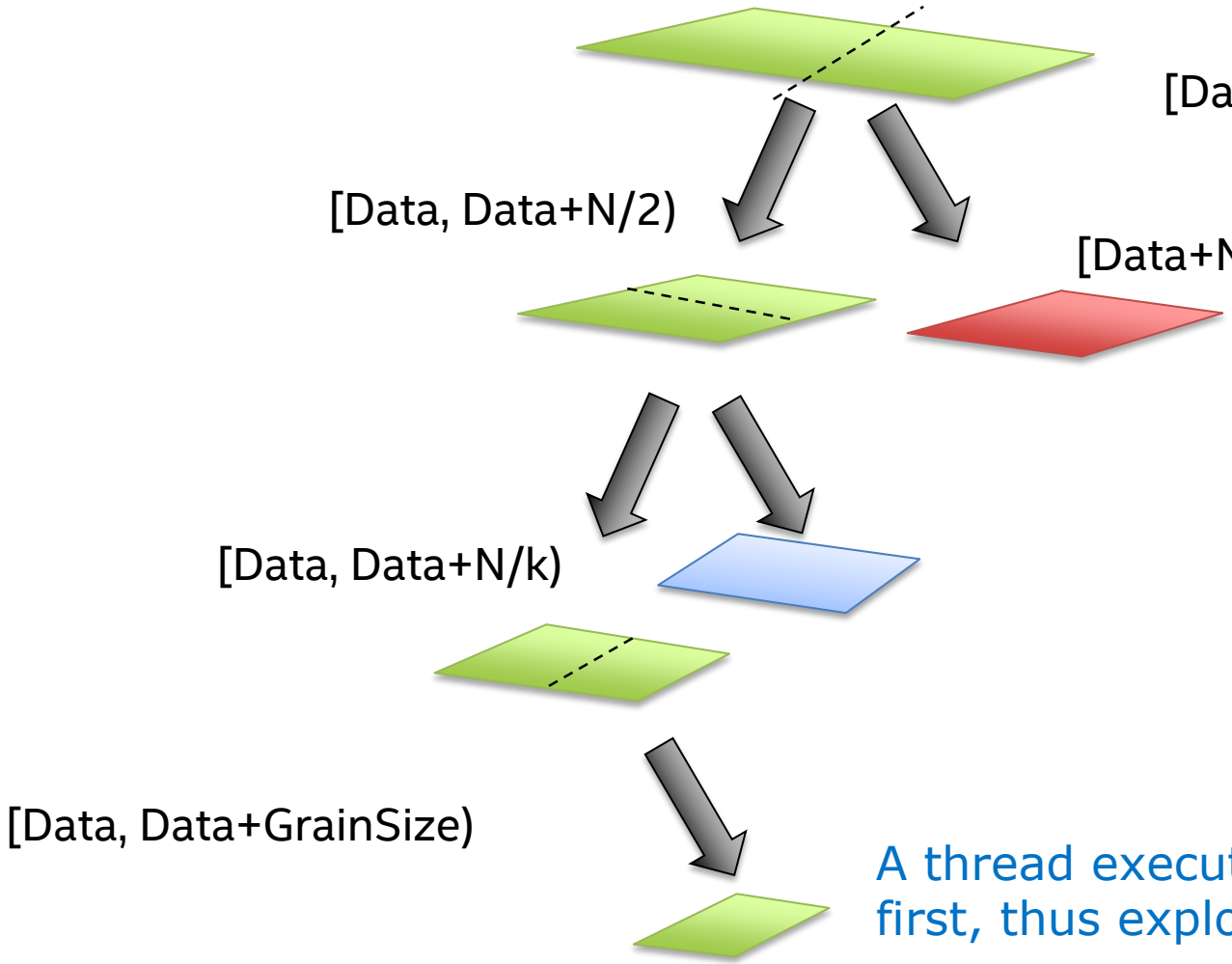


Recursive parallelism



A thread executes depth first, thus exploiting locality

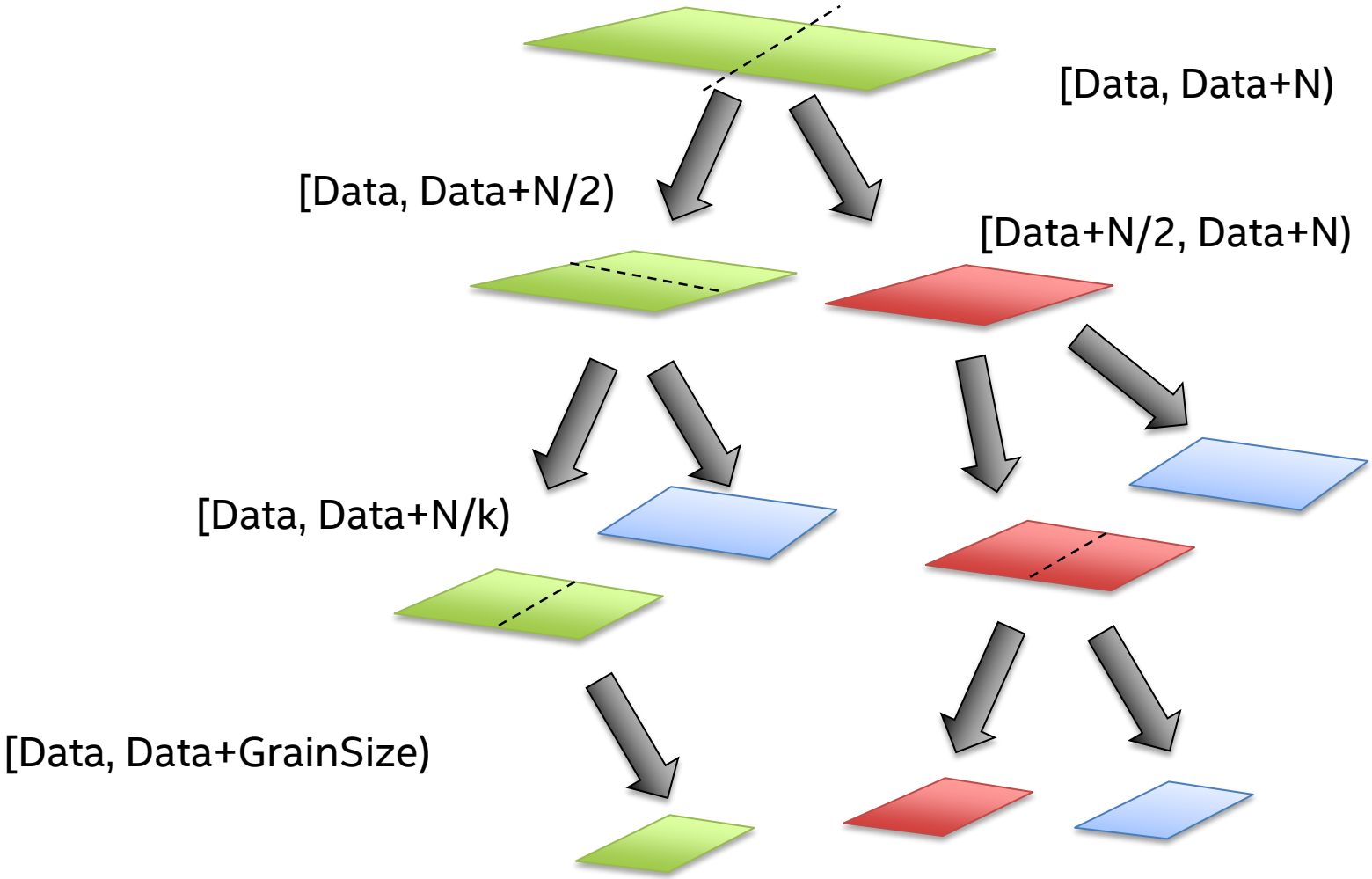
Recursive parallelism



Other worker threads steal work breadth first, taking older, larger pieces of work

A thread executes depth first, thus exploiting locality

Recursive parallelism



Implement the Intel® TBB task interface to create your own tasks

```
#include <tbb/task.h>

class my_task: public tbb::task {
public:
  tbb::task* execute() {
     WORK();
     return 0;
  }
};

my_task*  t = new (tbb::task::allocate_root())
        my_task(args);
```

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

Note: please also consider tbb::parallel_invoke or tbb::task_group to just spawn a series of functions to run in parallel.

Spawn vs. Enqueue

Spawn task to emphasize locality (cache-oblivious)

- Nested parallelism tends to depth-first execution
- Similar to Intel Cilk Plus task-stealing algorithm
- Used for heavy work, and light contention

Enqueue task to emphasize fairness

- Nested parallelism tends to breadth-first execution
- Only used with clear reason

Agenda

Introduction

Intel® Threading Building Blocks overview

Tasks concept

Generic parallel algorithms

Task-based Programming

Performance Tuning

Parallel pipeline

Concurrent Containers

Scalable memory allocator

Synchronization Primitives

Parallel models comparison

Summary

Performance Tuning

1. Tune the grain size (argument of blocked range)
   - Usually not beneficial*
   - Consider auto-tuning
2. Control affinity
   - Use affinity_partitioner with parallel_for & parallel_reduce
     - Specifies an “affinity domain”
     - Stores state to use as a hint by subsequent parallel algorithms
   - Use task_scheduler_observer to adjust thread affinity (pinning)
3. Interleave memory allocation (NUMA)
   - Linux: numactl –i all …

* Conceptually, an explicitly set grain size would also disable a runtime-dynamic adjustment.

Example: Grain Size (Auto-chunking)

```
#include <tbb/parallel_for.h>
#include <tbb/blocked_range.h>

int grainsize=16;

void sum(int* result, const int* a, const int* b, std::size_t size){
   tbb::parallel_for(tbb::blocked_range<std::size_t>(0, size, grainsize),
      [=](const tbb::blocked_range<std::size_t>& r) {
         for (std::size_t i = r.begin(); i != r.end(); ++i) {
            result[i] = a[i] + b[i];
         }
      }
   );
}
```

grainsize limits minimum chunk size. But TBB scheduler may choose bigger chunks.

Example: Grain Size (Max. Control)

```
#include <tbb/parallel_for.h>
#include <tbb/blocked_range.h>

tbb::simple_partitioner partitioner;

int grainsize=16;

void sum(int* result, const int* a, const int* b, std::size_t size){
   tbb::parallel_for(tbb::blocked_range<std::size_t>(0, size, grainsize),
      [=](const tbb::blocked_range<std::size_t>& r) {
         for (std::size_t i = r.begin(); i != r.end(); ++i) {
            result[i] = a[i] + b[i];
         }
      },
      partitioner
   );
}
```

Simple partitioner makes TBB scheduler to divide chunks until grainsize limit is reached.

Example: Affinity Partitioner

Use affinity partitioner when:

- The computation does a few operations per data access.
- The data acted upon by the loop fits into cache.
- The loop, or a similar loop, is re-executed over the same data.
- There are more than two hardware threads available.

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

void ParallelApplyFoo( float a[], size_t n ) {
    static tbb::affinity_partitioner partitioner;
    tbb::parallel_for(tbb::blocked_range<size_t>(0,n),
        ApplyFoo(a), partitioner);
}

void TimeStepFoo( float a[], size_t n, int steps ) {
    for( int t=0; t<steps; ++t )
        ParallelApplyFoo( a, n );
}
```

Tuning?

- Grain size tuning
  - Usually not necessary with auto/affinity partitioner
    - Keep in mind: developer's system != target system (usually)
  - Recommended for deterministic algorithms that rely on simple_partitioner e.g., `parallel_deterministic_reduce`
- Affinitization
  - Threads (core and SMT): not recommended beyond the use of affinity partitioner (observer method tends to hardcode strategy e.g., "compact", round-robin, etc.)
  - Socket: recommended for heterogeneous programs (MPI)
    - Pin each process to a socket (via e.g., I_MPI_* env. var.)

Agenda

Introduction

Intel® Threading Building Blocks overview

Tasks concept

Generic parallel algorithms

Task-based Programming

Performance Tuning

Parallel pipeline

Concurrent Containers

Scalable memory allocator

Synchronization Primitives

Parallel models comparison

Summary

Parallel Pipeline

Linear pipeline of stages

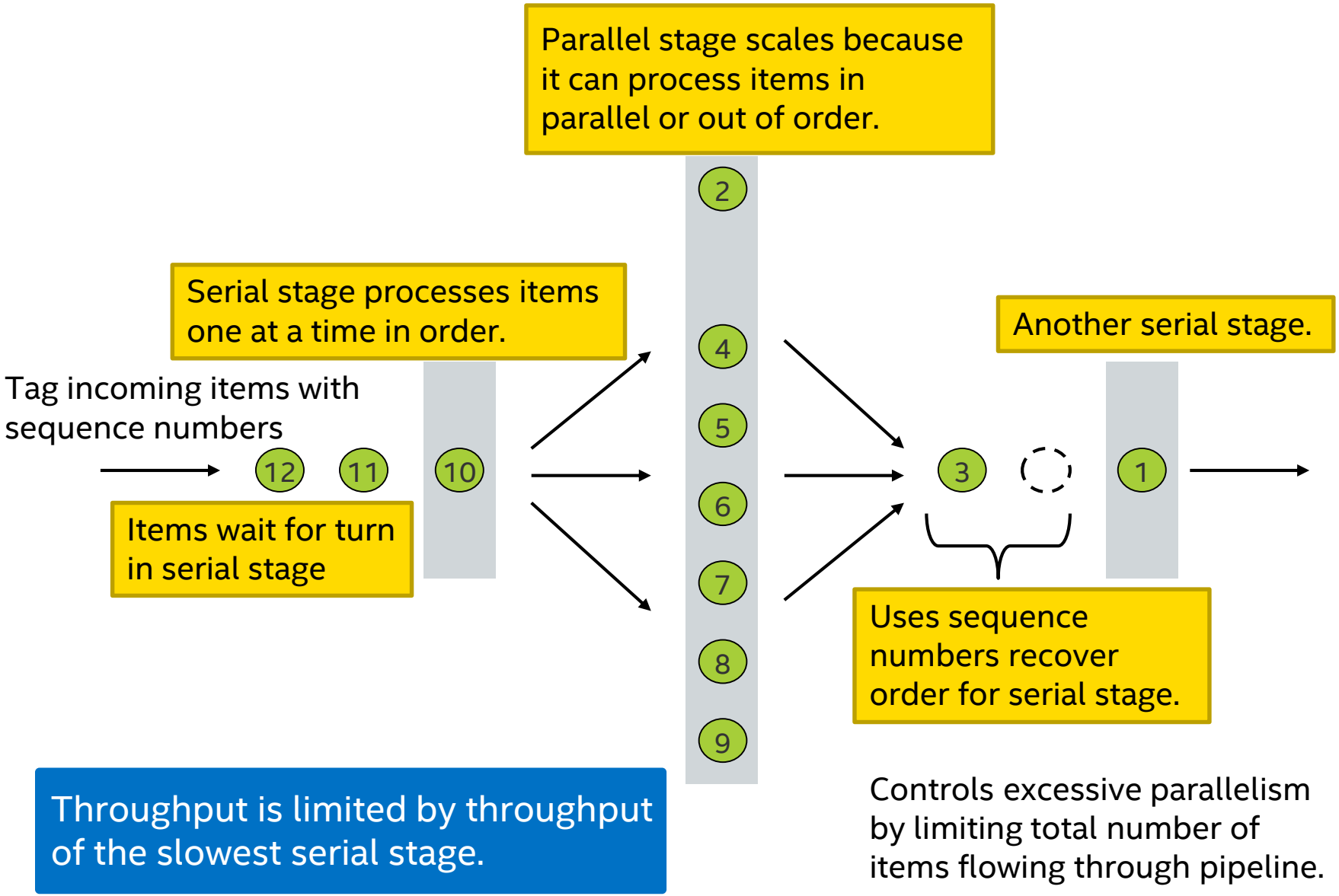
- You specify maximum number of items that can be in flight
- Handle arbitrary DAG by mapping onto linear pipeline

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.

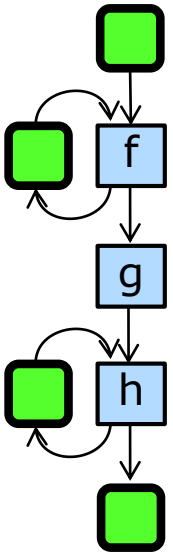
Uses cache efficiently

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



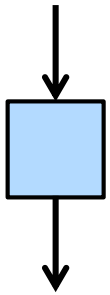
Pipeline Pattern

```
tbb::parallel_pipeline (
    ntoken,
    make_filter<void,T>(
        filter::serial_in_order,
        [&]( flow_control & fc ) -> T{
            T item = f();
            if( !item ) fc.stop();
            return item;
        }
    ) &
    make_filter<T,U>(
        filter::parallel,
        g
    ) &
    make_filter<U,void>(
        filter:: serial_in_order,
        h
    )
);
```



Serial vs. Parallel Stages

Parallel stage is functional transform.

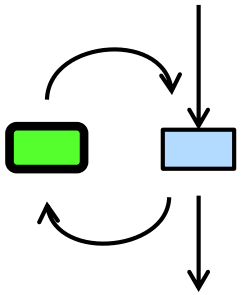


<from,to>

```
make_filter<X,Y>(
    filter::parallel,
    []( X x ) -> Y {
        Y y = foo(x);
        return y;
    }
)
```

You must ensure that parallel invocations of **foo** are safe.

Serial stage has associated state.

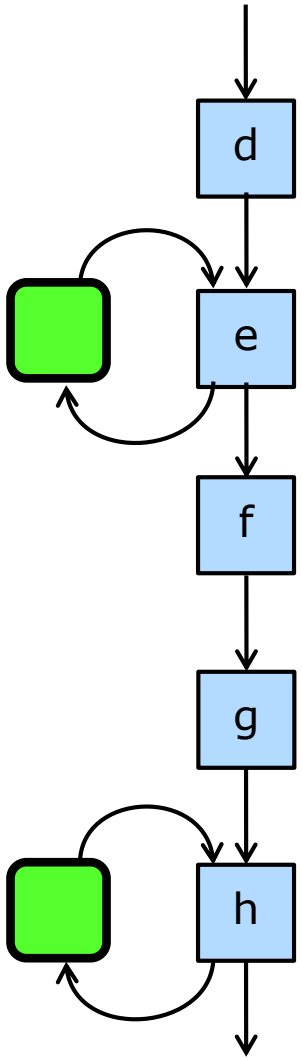


```
make_filter<X,Y>(
    filter::serial_in_order,
    [&]( X x ) -> Y {
        extern int count;
        ++count;
        Y y = bar(x);
        return y;
    }
)
```

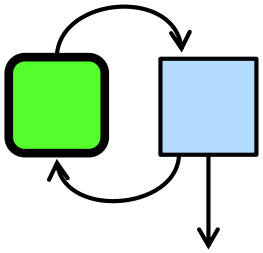
In-Order vs. Out-of-Order Serial Stages

Each in-order stage receives values in the order the previous in-order stage returns them.

Out-of-order stages process data serially, but the order is not deterministic.



Special Rules for First & Last Stage



"from" type is **void**

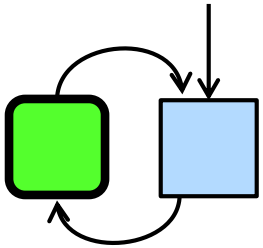
```
make_filter<void,Y>(
    filter::serial_in_order,
    [&]( flow_control& fc ) -> Y {
        Y y;
        cin >> y;
        if( cin.fail() ) fc.stop();
        return y;
    }
)
```

First stage receives special **flow_control** argument.

All filter types allowed

The return value is passed further through the pipeline, but ignored after a call to **flow_control::stop()**

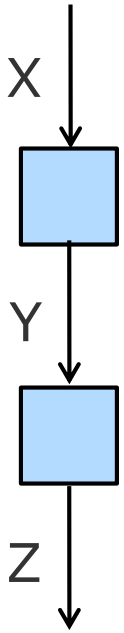
"to" type is **void**



```
make_filter<X,void>(
    filter::serial_in_order,
    [&]( X x ) {
        cout << x;
    }
)
```

Composing Stages

- Compose stages with **operator&**




```
make_filter<X,Y>(
    ...
)
&
make_filter<Y,Z>(
    ...
)
```

types must match

Type Algebra

make_filter<*T*,*U*>(*mode*,*functor*) → filter_t<*T*,*U*>

filter_t<*T*,*U*> & filter_t<*U*,*V*> → filter_t<T,V>

Running the Pipeline

```
parallel_pipeline(
    size_t ntoken, const filter_t<void,void>& filter );
```

Attempts to use cache efficiently

- A thread tries to proceed through as many stages as possible
- Biases towards finishing old items before tackling new ones

Functional decomposition is usually not scalable.
Parallel stages [may] make parallel_pipeline scalable.
Still, throughput limited by that of the slowest serial stage.

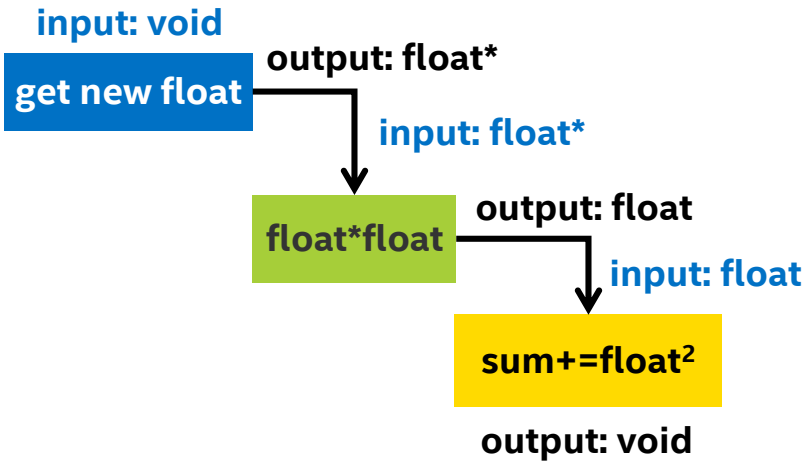
parallel_pipeline

```
float RootMeanSquare( float* first, float* last ) {
    float sum=0;
    parallel_pipeline( /*max_number_of_tokens=*/16,
      make_filter<void,float*>(
          filter::serial,
          [&](flow_control& fc)->float*{
              if( first<last ) {
                  return first++;
              } else {
                  fc.stop(); // stop processing
                  return NULL;
              }
          }
      ) &
      make_filter<float*,float>(
          filter::parallel,
          [](float* p) ->float* {return (*p)*(*p);}
      ) &
      make_filter<float,void>(
          filter::serial,
          [&sum](float x) {sum+=x;}
      )
    );
    // sum=first2+(first+1)2 + … +(last-1)2
    // computed in parallel
    return sqrt(sum);
}
```

Call function tbb::**parallel_pipeline** to run pipeline stages (filters)

Create pipeline stage object tbb::**make_filter**<InputDataType, OutputDataType>(mode, body)

Pipeline stage mode can be **serial**, **parallel**, **serial_in_order**, or serial_out_of_order



Agenda

Introduction

Intel® Threading Building Blocks overview

Tasks concept

Generic parallel algorithms

Task-based Programming

Performance Tuning

Parallel pipeline

Concurrent Containers

Scalable memory allocator

Synchronization Primitives

Parallel models comparison

Summary

Concurrent Containers

- Several STL-alike containers (similar concepts)
  - Simultaneously invoke certain methods of same container
  - More performance compared to external protection (fine grained locks or lock-free implementation)
- Thread-safety
  - Usually only concurrent reads
  - Concurrent mutual operations (Intel® TBB)
- Can be mixed with OpenMP*, or native threads

Concurrent Containers Key Features

concurrent_hash_map < Key, T , Hasher, Allocator >

- Models hash table of std::pair <const *Key*,*T*> elements
- Maps *Key* to element of type *T*
- User defines *Hasher* to specify how keys are hashed and compared
- Defaults: Allocator=tbb::tbb_allocator

concurrent_unordered_map < Key, T, Hasher, Equality, Allocator>

- Permits concurrent traversal and insertion (no concurrent erasure)
- Requires no visible locking, looks similar to STL interfaces
- Defaults: Hasher=tbb::tbb_hash, Equality=std::equal_to, Allocator=tbb::tbb_allocator
- Other unordered containers: multimap, set, multiset

concurrent_vector < T, Allocator >

- Dynamically growable array of T: grow_by and grow_to_at_least
- cache_aligned_allocator is a default allocator

concurrent_queue < T, Allocator >

- For single threaded run concurrent_queue supports regular “first-in-first-out” ordering
- If one thread pushes two values and the other thread pops those two values they will come out in the order as they were pushed
- cache_aligned_allocator is a default allocator

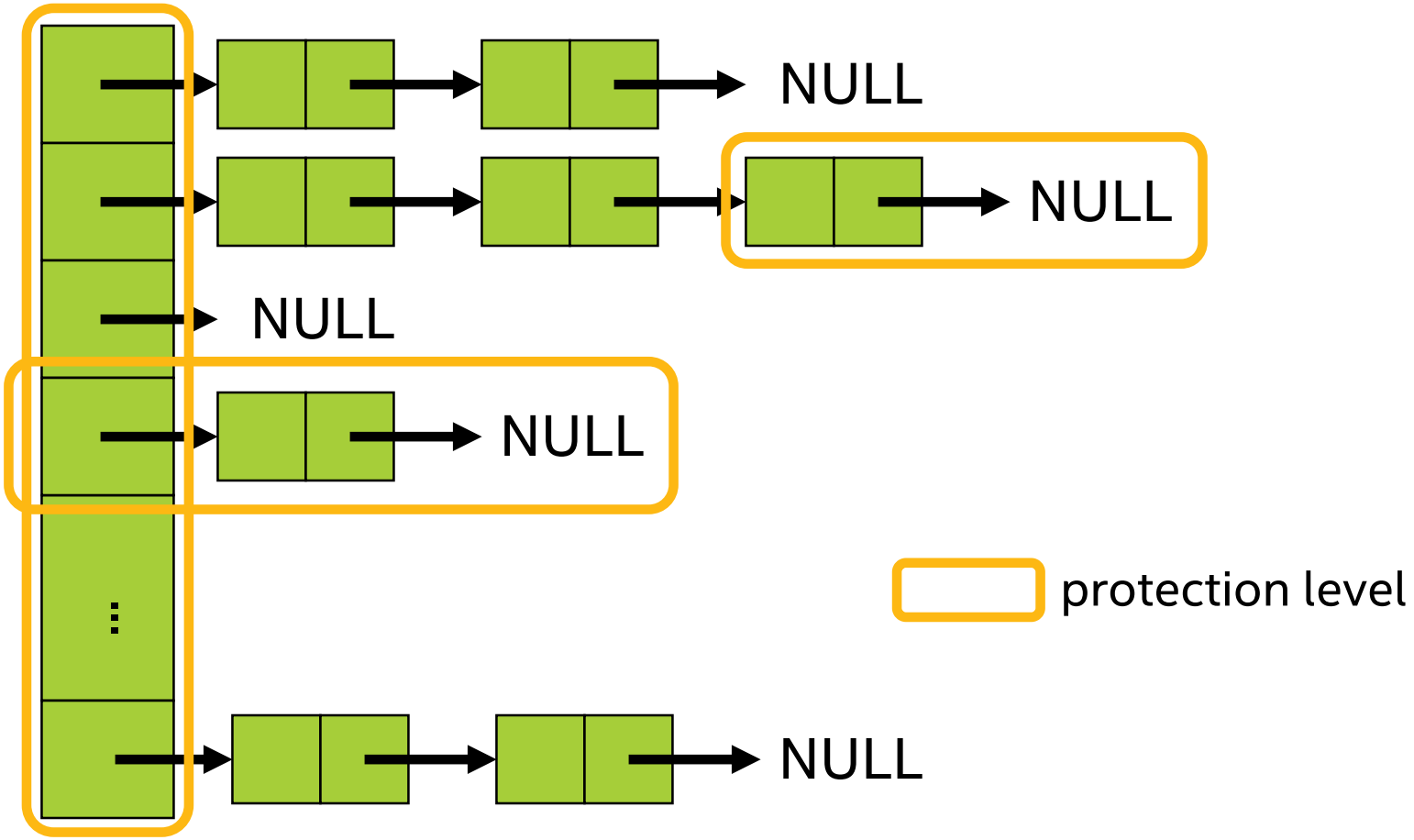
concurrent_bounded_queue <T, Allocator >

- Similar to concurrent_queue with a difference that it allows specifying capacity. Once the capacity is reached ‘push’ will wait until other elements will be popped before it can continue.

concurrent_priority_queue <T, Compare, Allocator >

- Permits to concurrently push and pop items; pops in user-specified priority order.
- Defaults: Compare=std::less, Allocator=tbb::cache_aligned_allocator

Example: Concurrent Hash Table



Example: concurrent_hash_map

```
struct wordsCompare {
    bool equal (const string& w1, const string& w2) const {
        return w1 == w2;
    }
    size_t hash(const string& w) const {
        size_t h = 0; for (const char* s = w.c_str();
        return h;
    }
};

void ParallelWordsCounting(const text_t& text) {

    parallel_for( blocked_range<size_t>( 0, text.size() ),
        [&text]( const blocked_range<int> &r ) {
            for(int i = r.begin(); i < r.end(); ++i) {
                concurrent_hash_map<string, int, wordsCompare>::accessor acc;
                wordCounters.insert(acc, text[i]);
                acc->second++;
            }
        });
}
```

User-defined “wordsCompare” class needs to implement functions for comparing two keys and a hashing function

An element of a concurrent_hash_map can be accessed by creating an “accessor” object, which is somewhat like a smart pointer implementing the necessary data access synchronization

Agenda

Introduction

Intel® Threading Building Blocks overview

Tasks concept

Generic parallel algorithms

Task-based Programming

Performance Tuning

Parallel pipeline

Concurrent Containers

Scalable memory allocator

Synchronization Primitives

Parallel models comparison

Summary

Why yet another memory allocator?

- Memory allocation can be (and often still is) a bottleneck in concurrent/parallel programs
- Thread-friendly, scalable allocators are known to be important for many real-world application
- If memory allocation is bottleneck, changing allocator can be fast and efficient solution

Using the allocator

- Shipped as a separate library: tbbmalloc
- Convenient interfaces:
  - Substitution for malloc/realloc/free etc. calls (C and C++)
  - Allocator classes to use with STL and Intel TBB containers (C++)
  - Dynamic replacement of standard memory allocation routines for the whole program (C and C++)
  - Preview feature: Special classes for memory pools (C++)
- Used internally by the main Intel TBB library
  - "If available", which means: found in the same directory

C interface (malloc-like)

Allocation Routine	Deallocation Routine	Matches
scalable_malloc	scalable_free	C standard library
scalable_calloc		
scalable_realloc		
scalable_posix_memalign		POSIX*
scalable_aligned_malloc	scalable_aligned_free	Microsoft* C run-time library
scalable_aligned_realloc		

Additional routines

size_t **scalable_msize**(void* ptr) Get usable size of the memory block pointed to by *ptr*	
int **scalable_allocation_mode**(int mode, intptr_t value) Adjust behavior of the memory allocator	
TBBMALLOC_USE_HUGE_PAGES	Use huge pages if supported by OS
TBBMALLOC_SET_SOFT_HEAP_LIMIT	Set a threshold for memory use
int **scalable_allocation_command**(int command, void *) Command the scalable memory allocator to perform an action	
TBBMALLOC_CLEAN_ALL_BUFFERS	Clean internal memory buffers
TBBMALLOC_CLEAN_THREAD_BUFFERS	Same but only for the calling thread

C++ allocator classes

Standard-compliant API with special features

Use the Intel TBB memory allocator (tbbmalloc)

container< T, **scalable_allocator<T>** > c;

Use tbbmalloc DLL if available, otherwise fall back to std::malloc

container< T, **tbb_allocator<T>** > c;

In addition to above, prevents false sharing between allocated objects

container< T, **cache_aligned_allocator<T>** > c;

```
typedef tbb::scalable_allocator <double> Alloc;
    std::vector<double,Alloc> buffer(1024);
```

Automatic malloc replacement

It is possible to automatically replace calls to memory allocating functionality with corresponding Intel® TBB function calls

This is done by the use of malloc_proxy libraries:

- Linux and OSX: libtbbmalloc_proxy.so.2 and libtbbmalloc_proxy_debug.so.2
- Windows: tbbmalloc_proxy.dll and tbbmalloc_debug_proxy.dll

Example: Huge Memory Pages

Service function takes precedence over environment variable (TBB_MALLOC_USE_HUGE_PAGES).

```
#if (4 <= TBB_VERSION_MAJOR && 2 <= TBB_VERSION_MINOR)
   // 0: disable, 1: enable
   int status = scalable_allocation_mode (TBBMALLOC_USE_HUGE_PAGES, 1);
#endif
```

Agenda

Introduction

Intel® Threading Building Blocks overview

Tasks concept

Generic parallel algorithms

Task-based Programming

Performance Tuning

Parallel pipeline

Concurrent Containers

Scalable memory allocator

Synchronization Primitives

Parallel models comparison

Summary

Synchronization Primitives

Atomic operations

- High-level abstractions

Exception-safe locks

- **spin_mutex** is VERY FAST in lightly contended situations; use it if you need to protect very few instructions
- Use **queuing_mutex** when scalability and fairness are important
- Use reader-writer (***_rw_mutex**) variants to allow non-blocking read for multiple threads
- Use **recursive_mutex** when your algorithm requires that one thread can re-acquire a lock. All locks should be released by one thread for another one to get a lock.

Properties

Sync. Primitive	Scalable	Fair	Reentrant	Sleeps
mutex	OS dependent	OS dependent	No	Yes
spin_mutex	No	No	No	No
queuing_mutex	Yes	Yes	No	No
spin_rw_mutex	No	No	No	No
queuing_rw_mutex	Yes	Yes	No	No
recursive_mutex	OS dependent	OS dependent	Yes	Yes

Example: scoped_lock

```
tbb::parallel_for(0, data_size,
   [=](int i) {
      data_item = compute(i);
      { // critical section scope
         spin_mutex::scoped_lock raii(lock);
         serial_container.insert(data_item);
      }
   }
);
```

If an exception occurs within the protected code block the destructor will automatically release the lock if it was acquired, avoiding a dead-lock

Example: Atomic Operation

```
#include <tbb/blocked_range.h>
#include <tbb/parallel_for.h>

tbb::atomic <int> sum;

int main () {
    int a[n];
    // initialize array here…

    tbb::parallel_for (0, n, 1,
        [=](int i) {
            Foo (a[i]);
            sum += a[i];
        });
    return 0;
}
```

This operation is performed atomically

Agenda

Introduction

Intel® Threading Building Blocks overview

Tasks concept

Generic parallel algorithms

Task-based Programming

Performance Tuning

Parallel pipeline

Concurrent Containers

Scalable memory allocator

Synchronization Primitives

Parallel models comparison

Summary

When to use:
Native threads, OpenMP, TBB

	Intel® TBB	Intel® Cilk™ Plus	OpenMP	C++11 threads	Native threads
Data decomposition support	Yes	Yes	Yes	No	No
Task level parallelism	Yes	Yes	Yes	No	No
Loop based parallelism	Yes	Yes	Yes	No	No
Complex parallel patterns	Yes	No	No	No	No
Scalable nested parallelism	Yes	Yes	No	No	No
Efficient load balancing	Yes	Yes	Yes	No	No
Affinity support	No *	No	Yes	No	Yes
Equal-size (static) work distribution	No	No	Yes	No	No
Concurrent data structures	Yes	No	No	No	No
Scalable memory allocator	Yes	No	No	No	No
Portable atomic operations	Yes	No	Yes	Yes	No
Synchronization (spinning/blocking)	Yes/Yes	No/No	Yes/No*	No*/Yes	Yes/Yes
Compiler support is not required	Yes	No	No	No	Yes
Vector parallelism	No	Yes	Yes	No	No
Co-processors or accelerators	Yes	Yes	Yes	No	No
Cross OS support	Yes	Yes	Yes	Yes	No
C / C++ / Fortran	C++	C/C++	C/C++/ Fortran	C++	C/C++/ Fortran

*Available via design pattern

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

Intel® TBB in the Computing Spectrum

Cloud/Data Centers

Clusters

Coprocessors Workstations

Servers

Notebooks

Desktops

Tablets

Netbooks

Embedded

Smartphones

MPI

OpenMP

Intel® TBB

Agenda

Introduction

Intel® Threading Building Blocks overview

Tasks concept

Generic parallel algorithms

Task-based Programming

Performance Tuning

Parallel pipeline

Concurrent Containers

Scalable memory allocator

Synchronization Primitives

Parallel models comparison

Summary

What's New

Intel® Threading Building Blocks 4.3

Feature	Benefits
Memory Allocator Improvements	Improved tbbmalloc – increases performance and scalability for threaded applications
Improved Intel® TSX Support	Applications that use read-write locks can take additional advantage of Intel TSX via tbb::speculative_spin_rw_mutex
Improved C++ 11 support	Improved compatibility with C++ 11 standard.
Tasks arenas	Improved control over workload isolation and the degree of concurrency with new class tbb::task_arena
Latest and Future Intel® Architecture Support	Supports[1] latest Intel® Architecture. Future proof with the next generation.

New Flow Graph Designer tool (currently in Alpha)

Visualize graph execution flow that allows better understanding of how Intel® TBB flow graph works.

Available today on http://whatif.intel.com

[1]See Intel® TBB release notes for hardware support matrix

Resources

Intel® Threading Building Blocks (Intel® TBB)

The Open-Source Community Site:
www.threadingbuildingblocks.org

Commercial product page:

https://software.intel.com/en-us/intel-tbb

Commercially available with Intel® Parallel Studio XE 2015:
https://software.intel.com/en-us/intel-parallel-studio-xe

Licensing questions

http://www.threadingbuildingblocks.org/licensing

A Good Parallel Programming Model

Easy to use

Produces software that is safe

Makes my code go faster and scale

Co-exists with other programs

Thank you

Legal Disclaimer & Optimization Notice

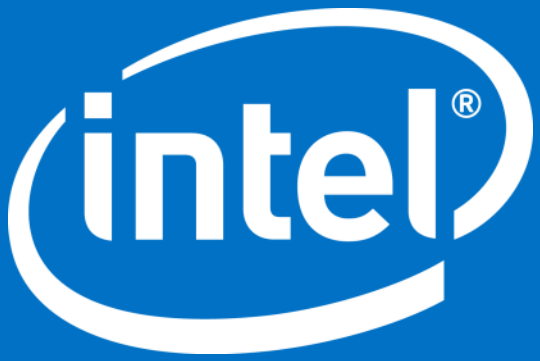
INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804


intel®

Backup

Example: Task Scheduler Observer

```
class ThreadPinner: public
  tbb::task_scheduler_observer {
public:
  void on_scheduler_entry(bool /**/) {
    HANDLE hCurThread = GetCurrentThread();
    SetThreadAffinityMask(hCurThread,
      compute_mask(hCurThread));
  }
  void on_scheduler_exit(bool /**/) {}
private:
  DWORD_PTR compute_mask(HANDLE hThread) {
    return (DWORD_PTR)1;
  }
};
```

```
int main(int argc, char* argv[])
{
  task_scheduler_init init;
  ThreadPinner tp;
  tp.observe();

  parallel_for(blocked_range<size_t>(0, N),
    [](blocked_range<size_t> &r) {
      for (size_t i = r.begin();
        i < r.end(); ++i) do_work(i);
    }, auto_partitioner());

    return 0;
}
```

* Note: the above code is specific for Windows *. However, it's similar when using Pthreads.