INTEL® THREADING BUILDING BLOCKS OVERVIEW

Alex Katranov

Intel Software and Services Group

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
http://threadingbuildingblocks.org

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

Benefits

- Rich feature set for general purpose parallelism.
- Available as an open source (Apache 2.0) and a commercial license.
- Supports C++, Windows*, Linux*, OS X*, other (non-HPC) OS's like Android*.
- Commercial support for Intel® Atom™, Core™, Xeon® processors, and for Intel® Xeon Phi™ coprocessors

Simplify Parallelism with a Scalable Parallel Model

Fundamental Philosophical Difference between Intel® TBB and “classic” threading models

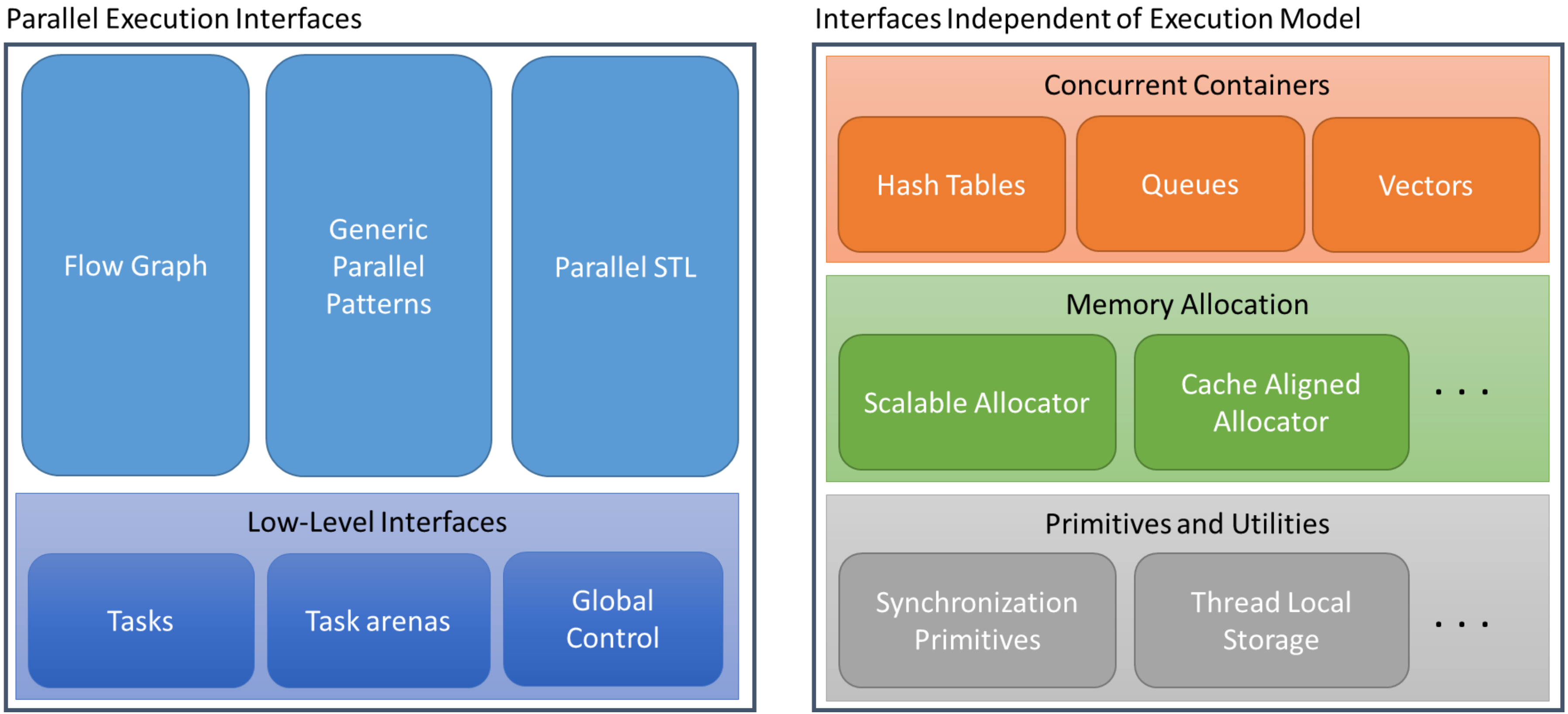
Classic threading models (OpenMP*, pthreads) describe **the implementation**

- Threads are the fundamental concern
- You know how many threads you have
- You can find out which thread is executing
- You have to work out how to map work onto threads

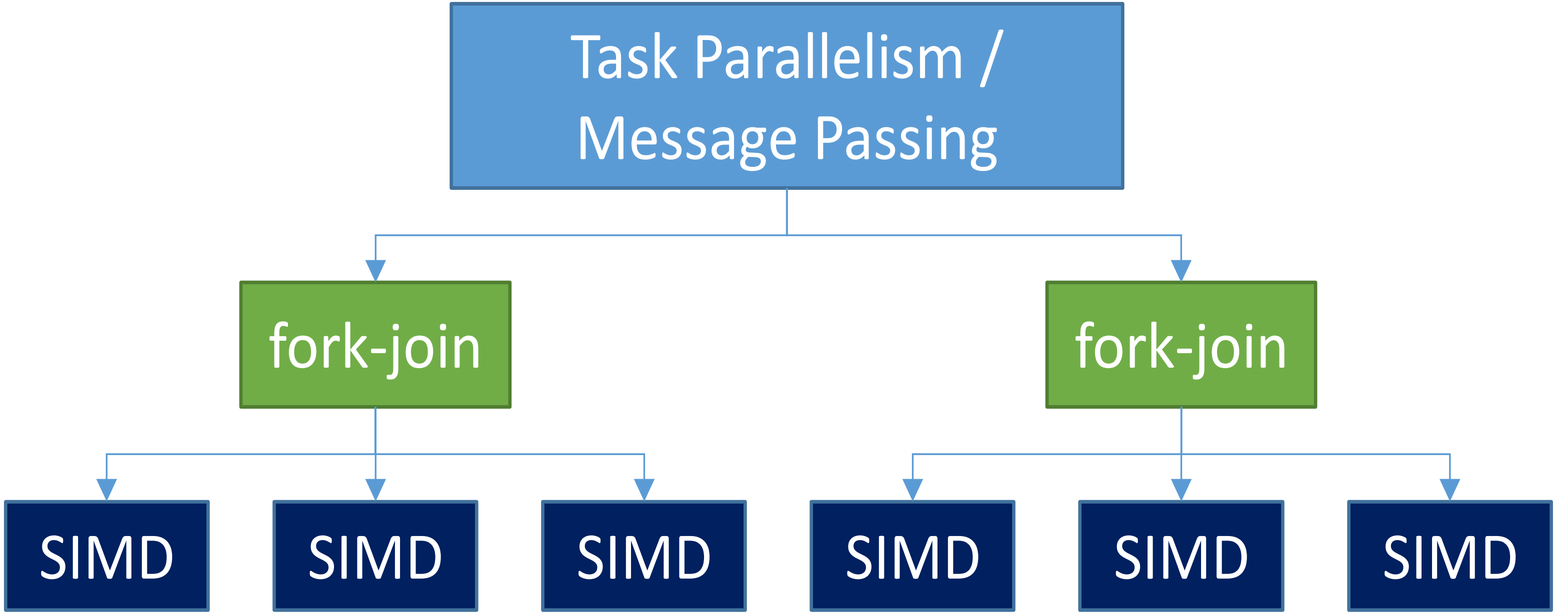
Intel® Threading Building Blocks describes **the algorithm**

- You don’t describe threads or know how many there are
- You do describe the parts of your code that can run in parallel
  - An algorithmic concept, not an implementation one
- The runtime chooses
  - How much of the available parallelism to use
  - How to map the work onto the hardware resources available to it at any instant
  - You can provide more tuning information, but don’t have to

Rich Feature Set for Parallelism



Applications often contain multiple levels of parallelism



Intel TBB helps to develop composable levels

Rich Feature Set for Parallelism

Parallel Execution Interfaces

- Flow Graph
- Generic Parallel Patterns
- Parallel STL

Low-Level Interfaces

- Tasks
- Task arenas
- Global Control

Interfaces Independent of Execution Model

Concurrent Containers

- Hash Tables
- Queues
- Vectors

Memory Allocation

- Scalable Allocator
- Cache Aligned Allocator
- . . .

Primitives and Utilities

- Synchronization Primitives
- Thread Local Storage
- . . .

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

Parallel Algorithms for Sequences and 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

parallel_for generic form

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

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

- Balances load
- Uses cache efficiently
- Scales

Library provides range classes:

- blocked_range models a one-dimensional range
- blocked_range2d models a two-dimensional range
- blocked_range3d models a three-dimensional range

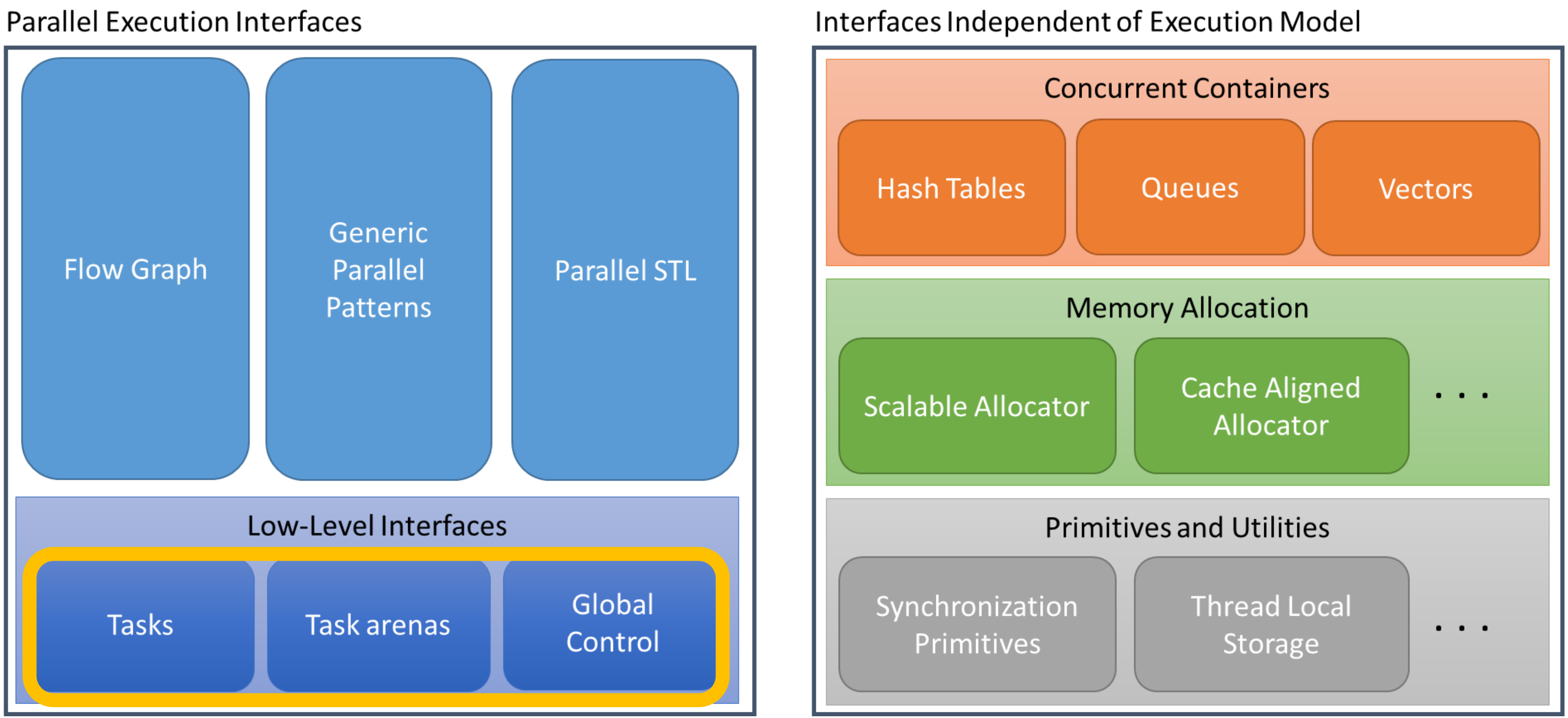
parallel_for simple form for 1D loops

```
template <typename Index, typename Body>
void parallel_for (Index lower, Index upper, const Body &body);
```

Example:

```
void sum(const int* in, const int* in2,
         std::size_t size, int* out)
{
  tbb::parallel_for(std::size_t(0), size,
                    [=](std::size_t i) {
                      out[i] = in[i] + in2[i];
                    } );
}
```

Rich Feature Set for Parallelism



How does it work? What is a Task Scheduler?

Terminology

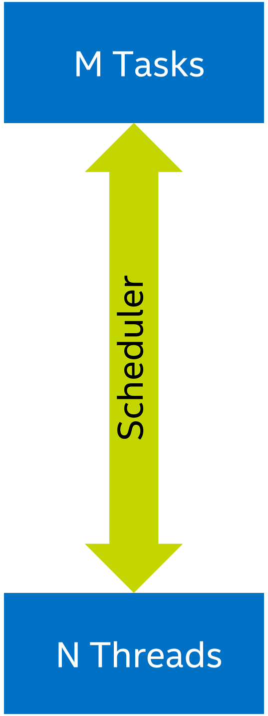
- Thread refers to a physical thread (`logicalCPU` in Linux-speak)
- Task refers to a piece of work

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

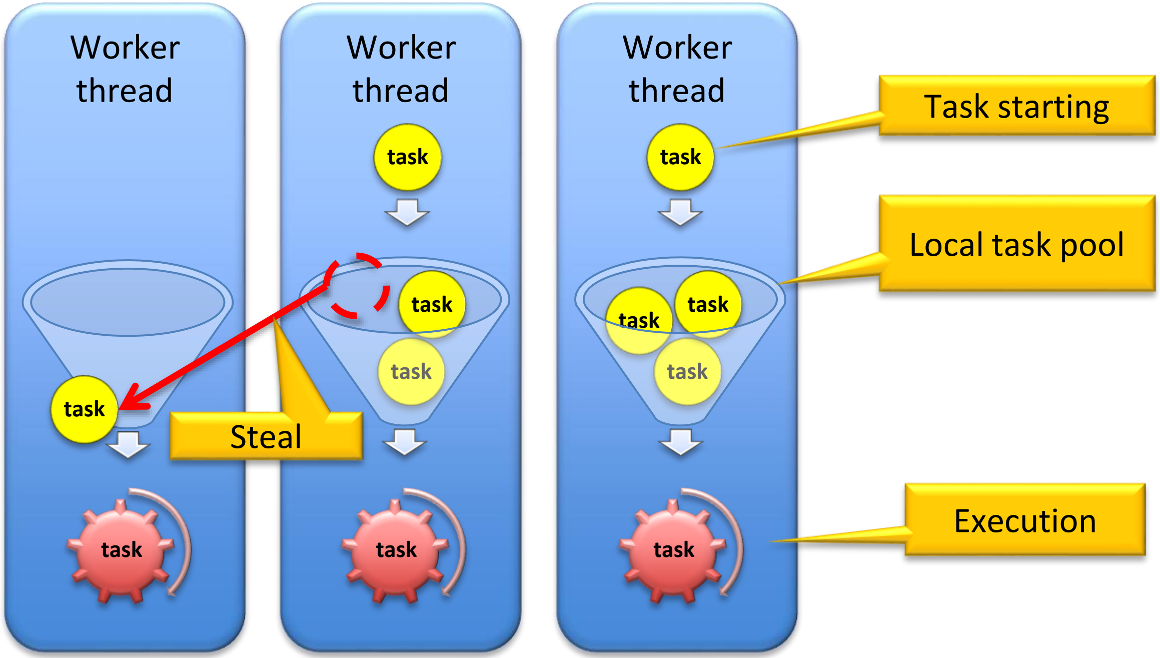
- LIFO: thread-local queue (spawned tasks)
- ~FIFO: shared global queue (enqueued tasks)
- ~FIFO: Task-stealing (random foreign queue)



Task Execution in Intel® TBB (simplified)

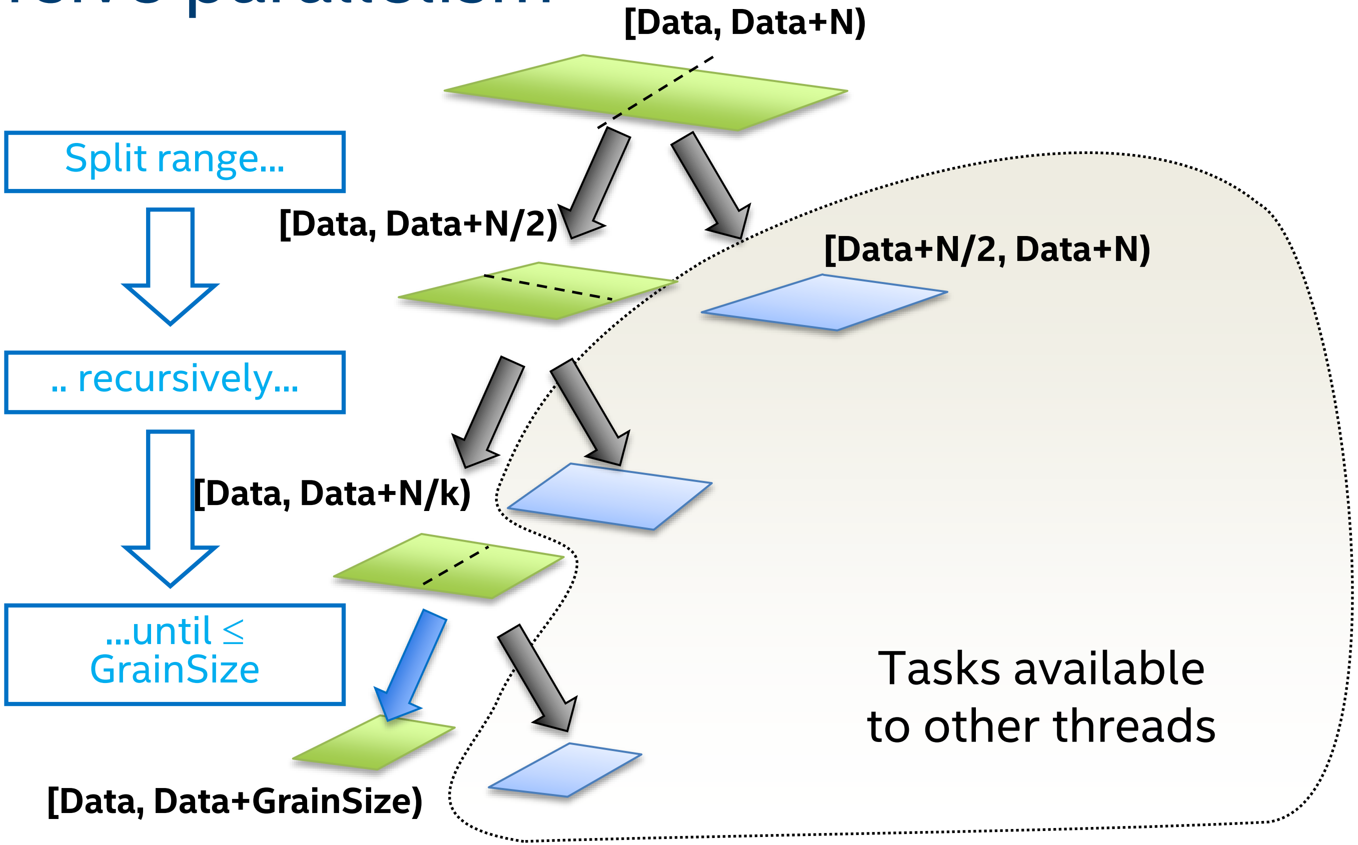
The Intel® TBB runtime dynamically maps tasks to threads

Automatic load balance, lock-free whenever possible, unfair

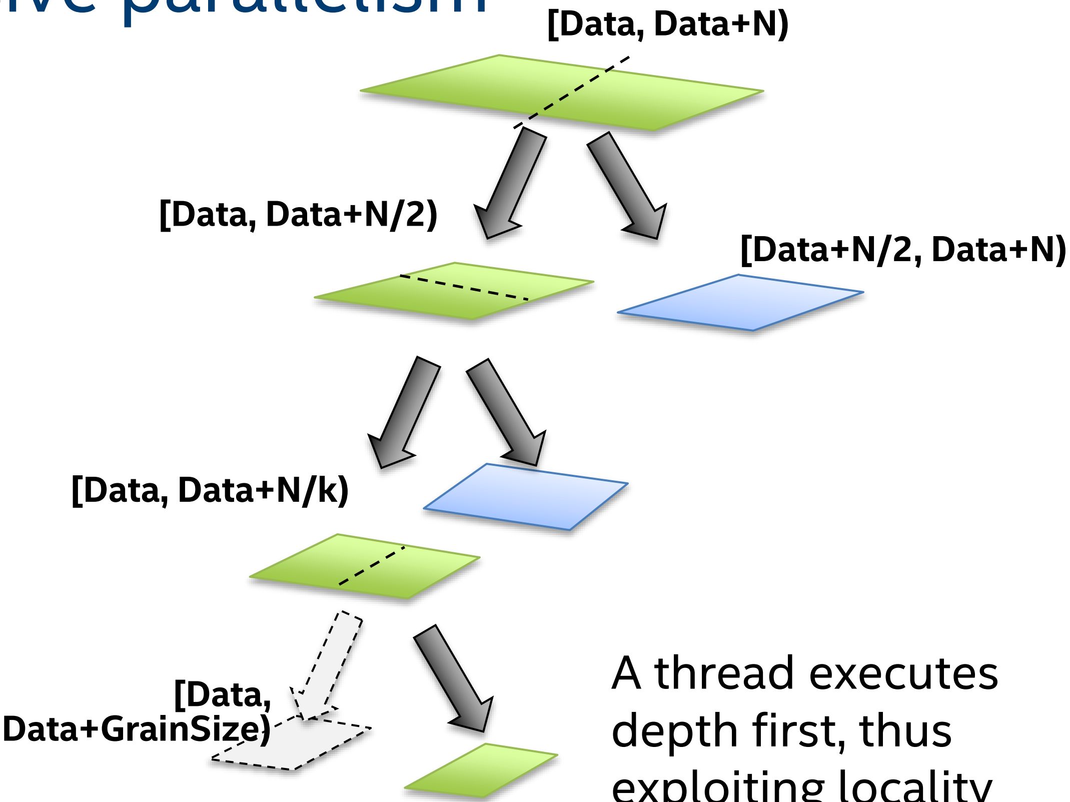


Abstract version of the scheduler

Recursive parallelism

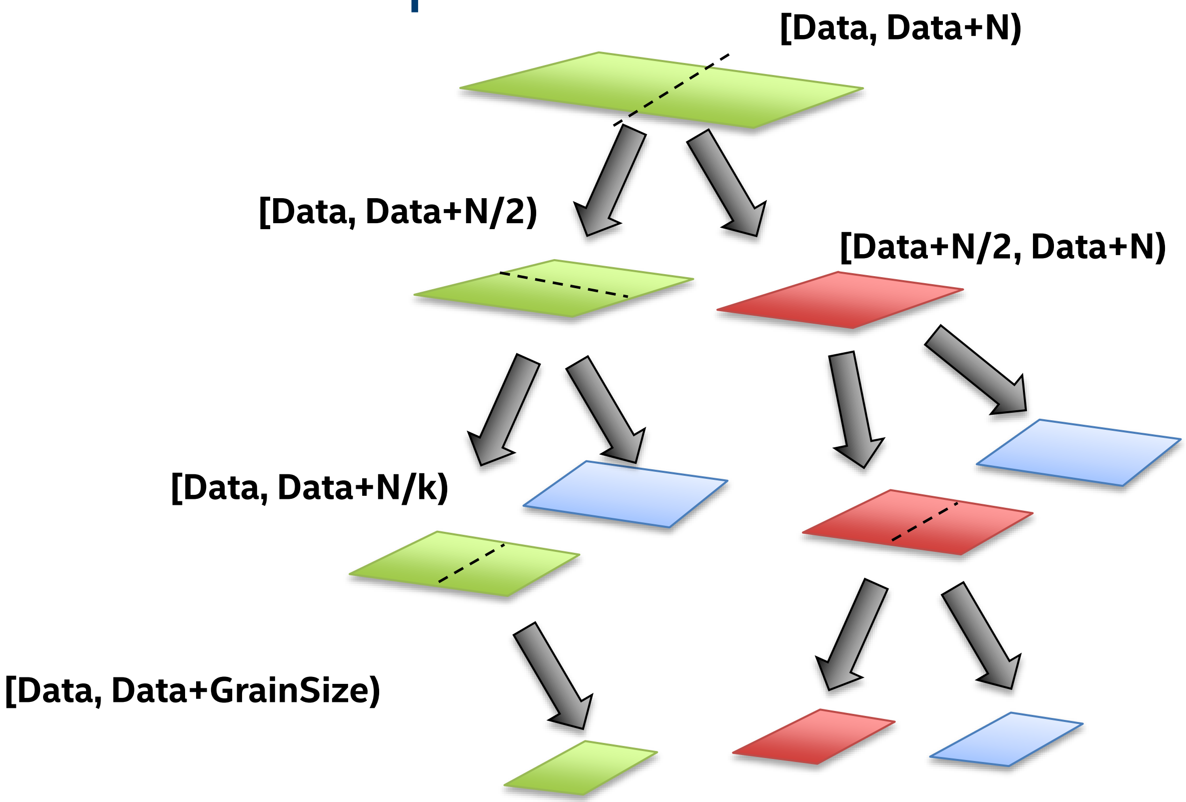


Recursive parallelism



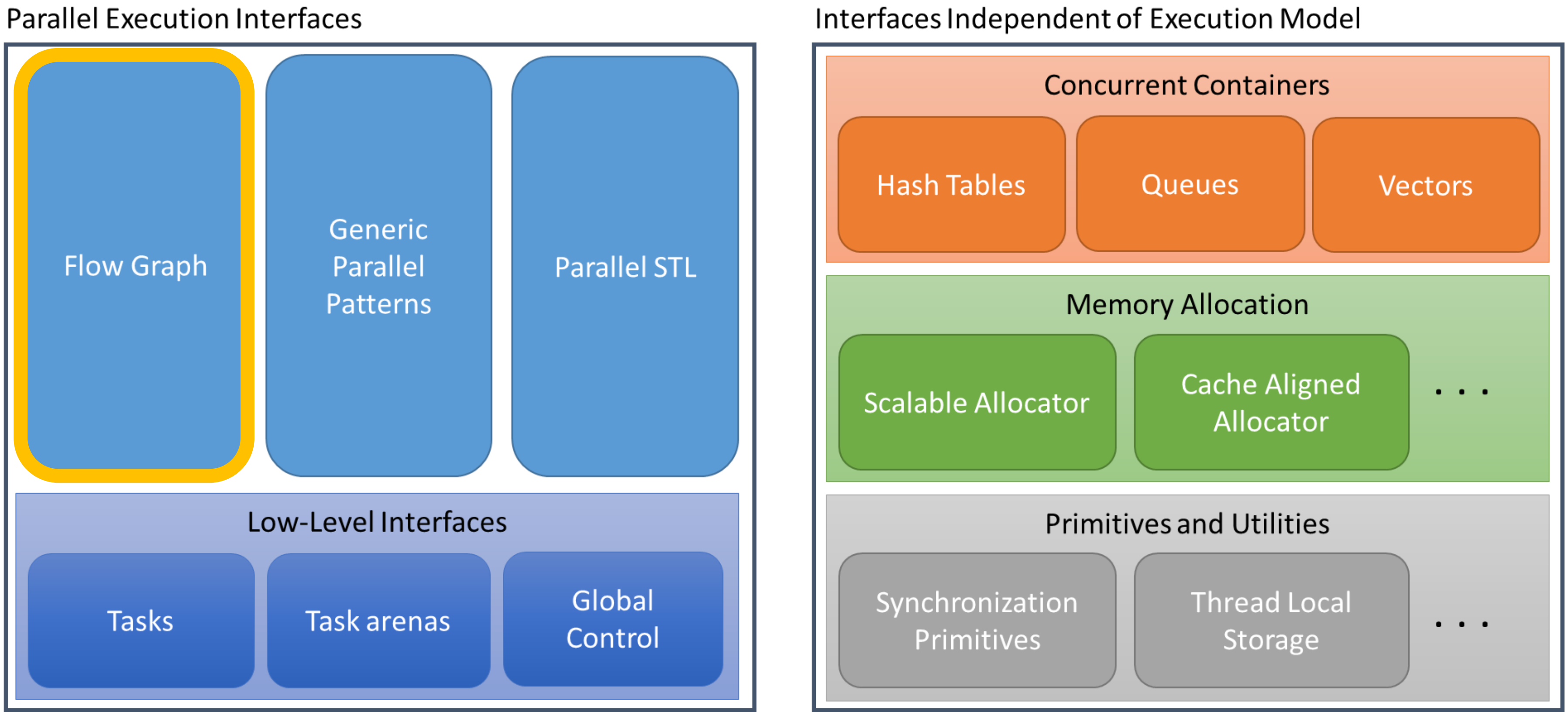
A thread executes depth first, thus exploiting locality

Recursive parallelism



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

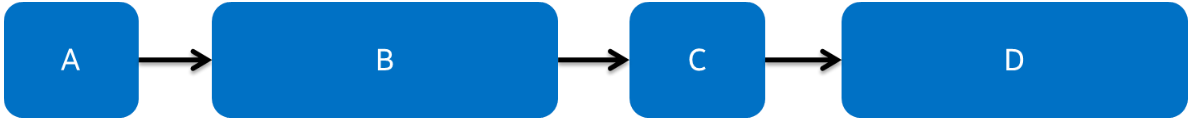
Rich Feature Set for Parallelism



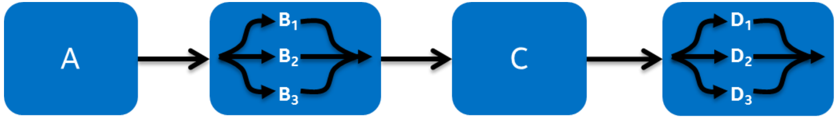
Motivation for data flow and graph-parallelism

```
x = A();
y = B(x);
z = C(x);
D(y, z);
```

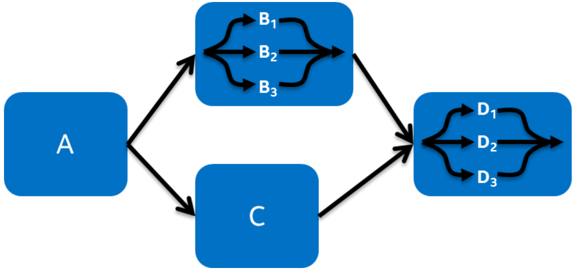
Serial implementation (perhaps vectorized)



Loop-parallel implementation ("Classic OpenMP*")



Loop- and graph-parallel implementation



Intel® TBB flow graph

Users express dependencies between computational nodes

- The runtime extracts the implicit parallelism
- Schedules computations using Intel TBB

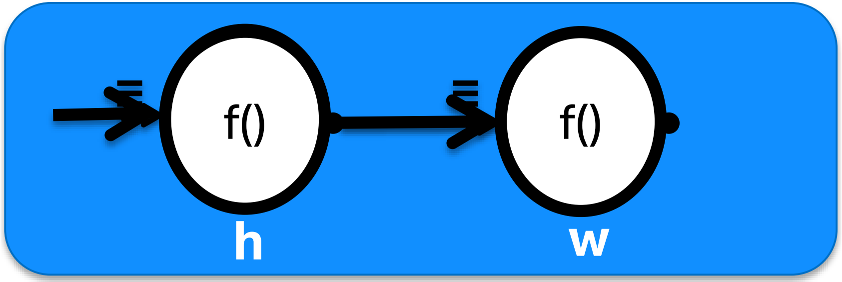
Use cases

- Streaming of images, frames, financial data etc…
- Reacting to GUI events
- Expressing dependencies in computations to enable parallelism
- Offloading computations to other devices (“accelerators”)

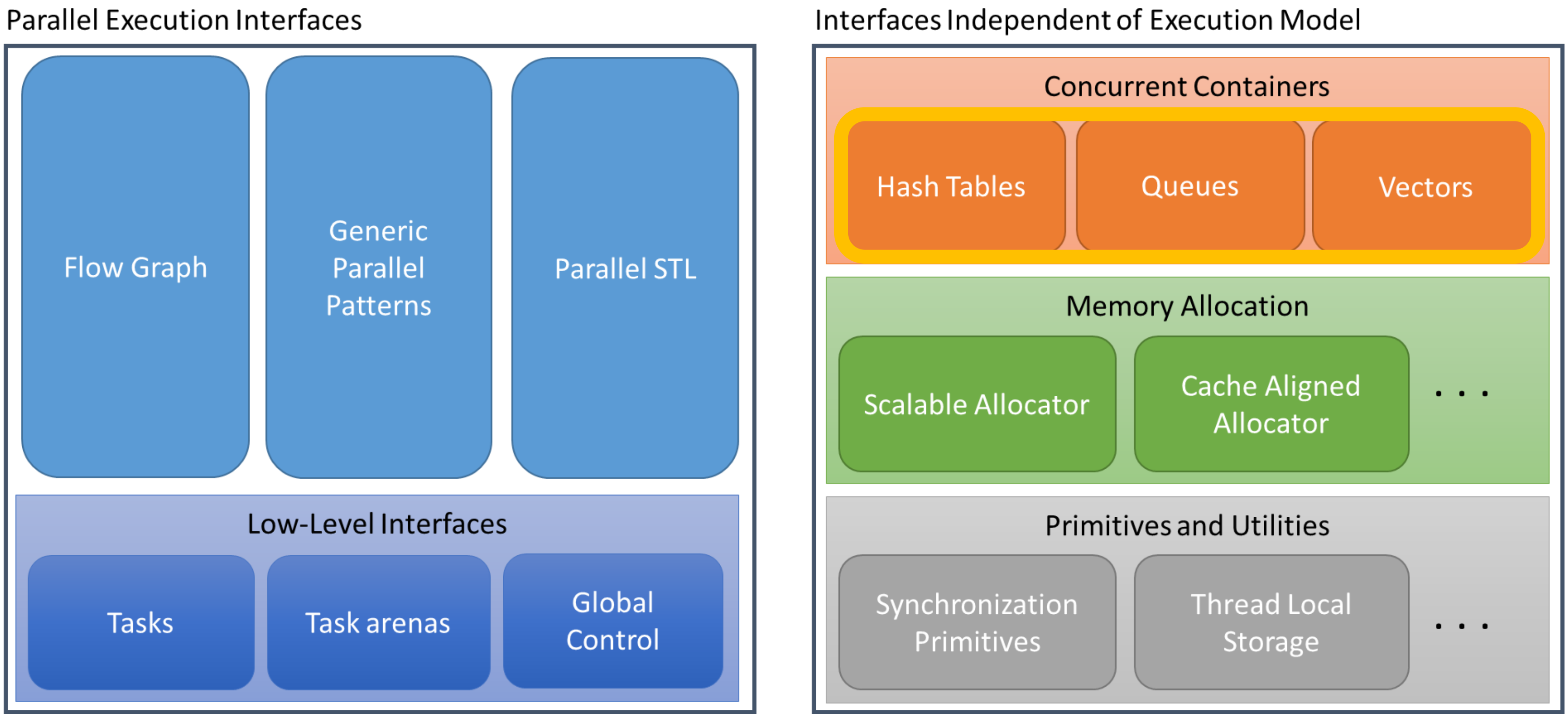
Hello World Example

Users create nodes and edges, interact with the graph and wait for it to complete

```
tbb::flow::graph g;
tbb::flow::continue_node< tbb::flow::continue_msg >
  h( g, []( const continue_msg & ) { std::cout << "Hello "; } );
tbb::flow::continue_node< tbb::flow::continue_msg >
  w( g, []( const continue_msg & ) { std::cout << "World\n"; } );
tbb::flow::make_edge( h, w );
h.try_put(continue_msg());
g.wait_for_all();
```



Rich Feature Set for Parallelism



Concurrent Containers

Several STL-like containers

- Similar concepts, partially compatible API

Better thread safety guarantees

- Basic C++ guarantee: safe concurrent reads, i.e. const methods
- Intel TBB guarantee: some modifying methods can be invoked concurrently
- Data modifications might require additional protection

Better performance compared to external lock protection

- Intel TBB uses fine grained locks or lock-free implementation

Can be mixed with OpenMP*, C++ or native threads, …

- A simple way to start using Intel TBB

Concurrent Containers

Associative tables

- concurrent_hash_map
- concurrent_unordered_[multi]map, concurrent_unordered_[multi]set

Queues

- concurrent_queue, concurrent_bounded_queue
- concurrent_priority_queue

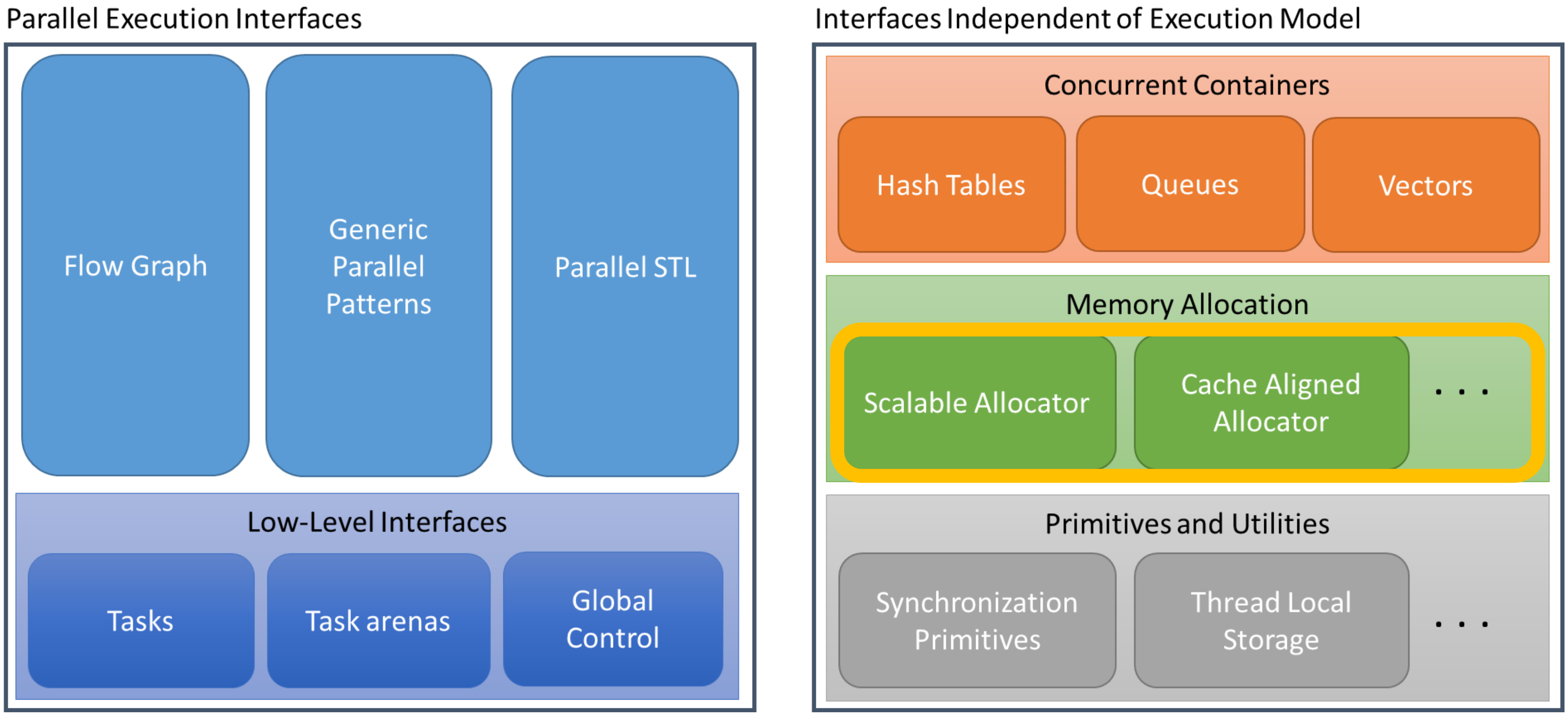
Random access

- concurrent_vector

Thread-local data storage

- enumerable_thread_specific, combinable

Rich Feature Set for Parallelism



Why yet another memory allocator?

- Memory allocation can be (and often is) a bottleneck in concurrent/parallel programs
- Thread-friendly, scalable allocators are known to be important for many real-world applications
- If memory allocation is bottleneck, changing the allocator can be an easy, non-intrusive, way to improve performance

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++) (can be achieved using `LD_PRELOAD` on some OSes)
- 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

`tbbmalloc` can be used without any of the rest of TBB

Rich Feature Set for Parallelism

Parallel Execution Interfaces

- Flow Graph
- Generic Parallel Patterns
- Parallel STL

Low-Level Interfaces

- Tasks
- Task arenas
- Global Control

Interfaces Independent of Execution Model

Concurrent Containers

- Hash Tables
- Queues
- Vectors

Memory Allocation

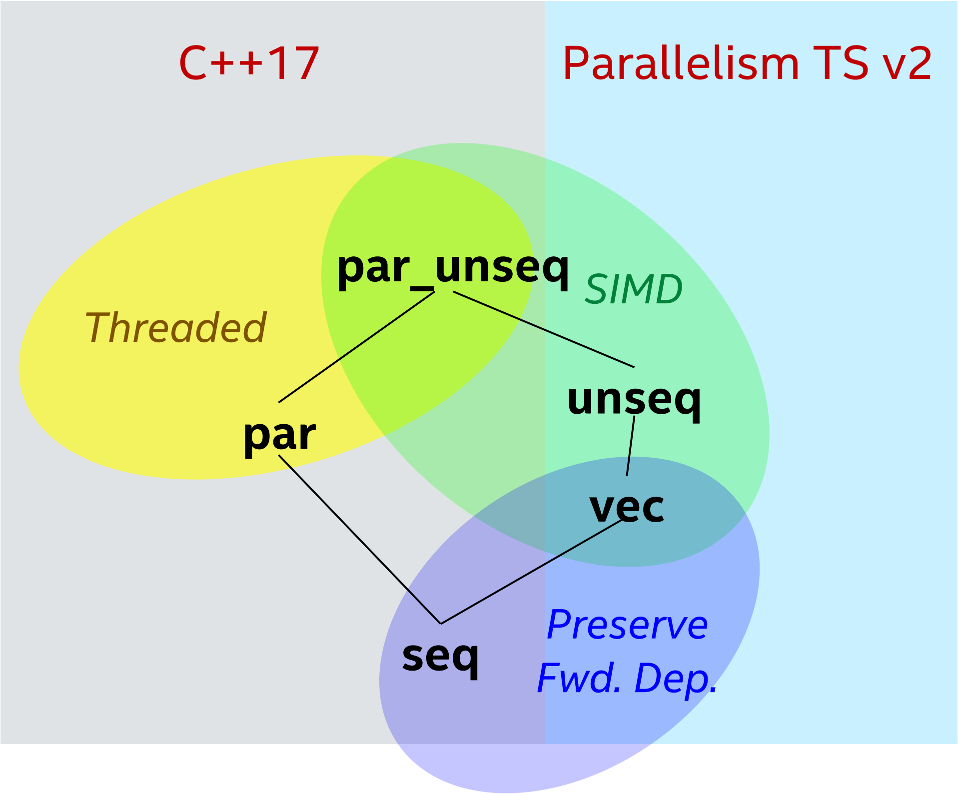
- Scalable Allocator
- Cache Aligned Allocator
- . . .

Primitives and Utilities

- Synchronization Primitives
- Thread Local Storage
- . . .

Parallel C++ Standard Template Library (STL)

- Extension of C++ Standard Template Library algorithms with the "execution policy" argument
- Support for parallel execution policies is approved for C++17
- Support for vectorization policies is being developed in Parallelism Technical Specification (TS) v2



Simple example

```
#include <algorithm>
#include <execution>

void increment( float *in, float *out, int N ) {
    using namespace std::execution;
    transform( par, in, in + N, out, []( float f ) {
        return f+1;
    }
}
```

The implementation of Parallel STL

Goal: provide first-class implementation for Intel® processors

- Scalable parallel execution
- Efficient vector execution
- Vector+parallel execution is a combination of above
- Relies on the standard library for sequential execution and not yet implemented other policies for an algorithm

C++ compiler prerequisites

- C++11 and OpenMP 4.0 vectorization (#pragma omp simd)

Parallel runtime

- The first version is based on Intel TBB. Other back-ends might be added later, based on customer demand.

FINAL WORDS

Conclusions

Choice of parallel model matters

Intel® TBB is a good choice for C++ codes

Even if you already have threaded code Intel TBB may have some components that can help you

- Better memory allocator
- Concurrent containers
- Low level locks, timers, …

Intel TBB is open-source, portable, and has commercial and non-commercial licenses

Try it http://www.threadingbuildingblocks.org/

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

BACKUP

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

Pthreads standard: IEEE 1003.1c-1995
OpenMP* 1.0 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.

Threads are too low level a model.

What Do We Mean by “Task” ?

A piece of work represented by a (lambda) function and its captured arguments that we can run in parallel with other tasks

Modern C++ uses lambda functions in the STL, e.g. `std::for_each`

```
std::vector<float> array;
// Replace each element in an array with its square root
std::for_each (array.begin(), array.end(),
   [=](float & elem) { elem = sqrt(elem);});
```

Intel® TBB also exploits them, e.g. parallelize the code above

```
std::vector<float> array;
// Replace each element in an array with its square root
tbb::parallel_for_each (array.begin(), array.end(),
   [=](float & elem) { elem = sqrt(elem);});
```

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

Task Scheduler

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

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)

Timers and Exceptions

- tbb_exception
- captured_exception
- movable_exception
- tick_count

Threads

Thread

Thread Local Storage

- combinable
- enumerable_thread_specific

Memory Allocation

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

Work Stealing Task Scheduler Implementation

The simple version:

Each thread has a deque of tasks

- Newly created tasks are pushed onto the front
- Other threads steal from the back

Allows local task creation and use to be lock-free (so fast)

When looking for tasks the thread pops from the front

- Task is likely still to be hot in the cache since it was the most recently pushed

If it has no work

- Pick a random victim
- Attempt to steal a task from the back of their deque

Stolen tasks are likely to be

- Large, so the cost of stealing is amortized over a lot of work
- Old, so cold in the cache

task_arena provides control of the number of threads used and work isolation

```
tbb::task_arena limited(2);
tbb::task_group tg;

limited.execute([&]{ // use at most 2 threads
    tg.run([]{ // run in task group
        tbb::parallel_for(1, N, unscalable_work());
    });
});

tbb::parallel_for(1, M, scalable_work());

limited.execute([&]{ tg.wait(); });
```

Use no more than 2 threads in this arena

task_group is used to submit a job and wait for it later

Run another job concurrently with the loop above
It will use the default number of threads

Put the wait for the task group inside execute()
This will wait only for the tasks that are in this task group.

global_control

Application-level control of resources

– Imposes high composability risks, and thus is highly discouraged to use in libraries

tbb::global_control (parameter, value)

where *parameter* could be:

max_allowed_parallelism

– Limits total number of worker threads that can be active in the library

thread_stack_size

– Sets stack size for the threads created by the library

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 (though `flow::graph` may be a better match now it exists!)

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.

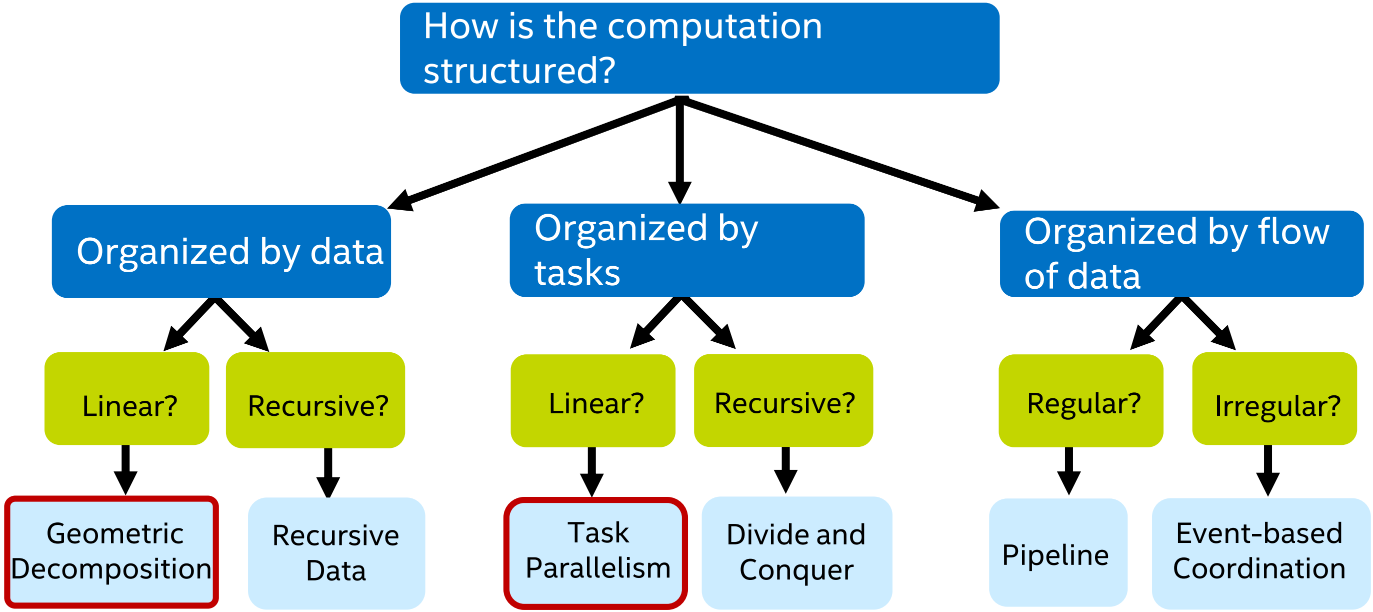
Uses cache efficiently

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

Improves on the naïve one thread/stage implementation

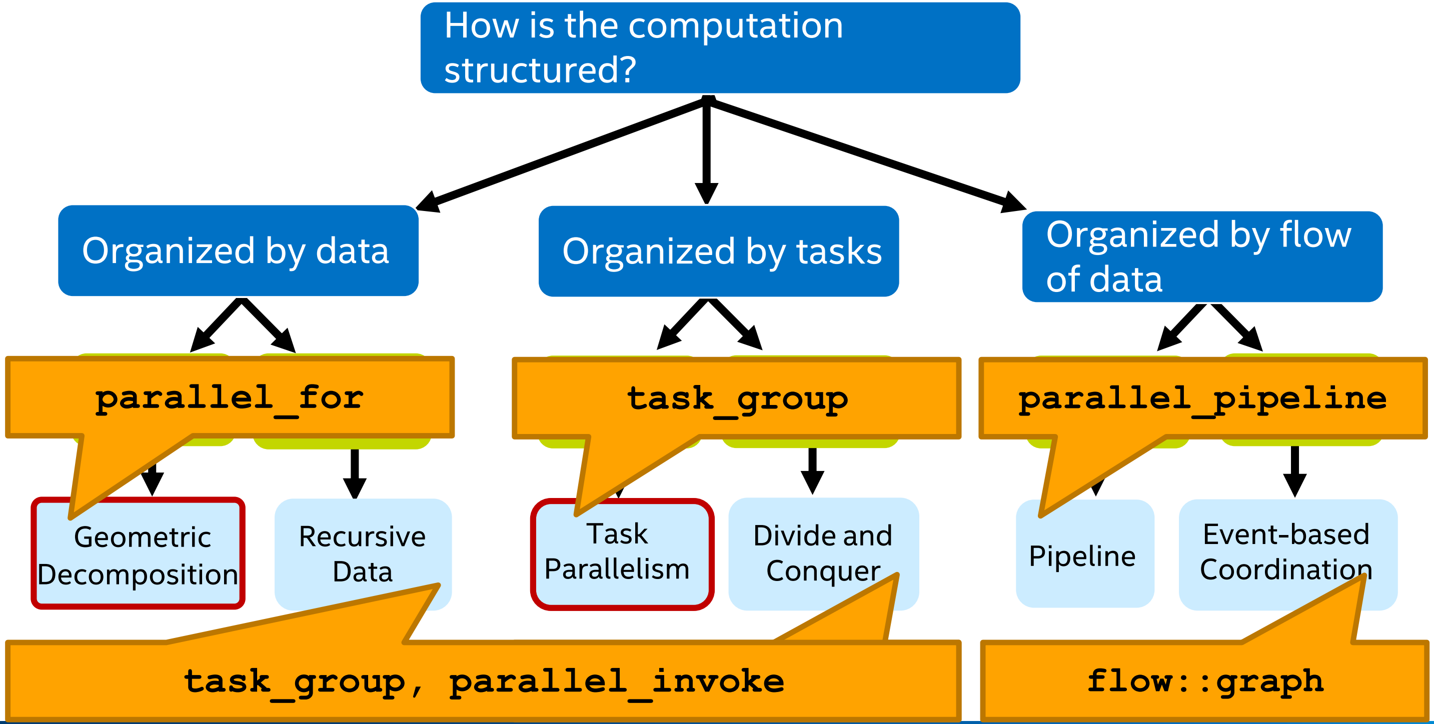
Algorithm Structure Design Space

Structure used to organize parallel computations



Algorithm Structure Design Space

Structure used to organize parallel computations



Evolution of STL

Standard Template Library | C++ Standard

C++ Extensions for Parallelism | Technical Specification

vec, unseq policies

indexed based loops

(deferred to separate presentation)

Intel proposals

Example Supported by Technical Specification

```
extern std::vector<float> x, y;
using namespace std::experimental::parallel;
auto f = [](auto a) {return a*a;};

// Sequential
transform(seq, x.begin(),  x.end(), y.begin(), f);

// Parallel
transform(par, x.begin(),  x.end(), y.begin(), f);

// Dynamically-selected policy
execution_policy e = seq;
if( x.size()>10000)
    e = par;
transform(e, x.begin(),  x.end(), y.begin(), f);
```

Legal Disclaimer & Optimization Notice

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

experience what's inside™