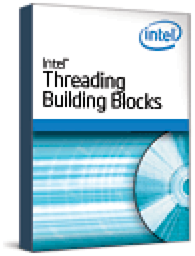
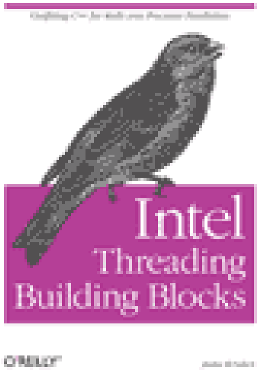
Threading Building Blocks

Martin Kruliš

Introduction

- http://threadingbuildingblocks.org/
  - Commercial and open source versions
- J. Reinders - Intel® Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism – O'REILLY 2007

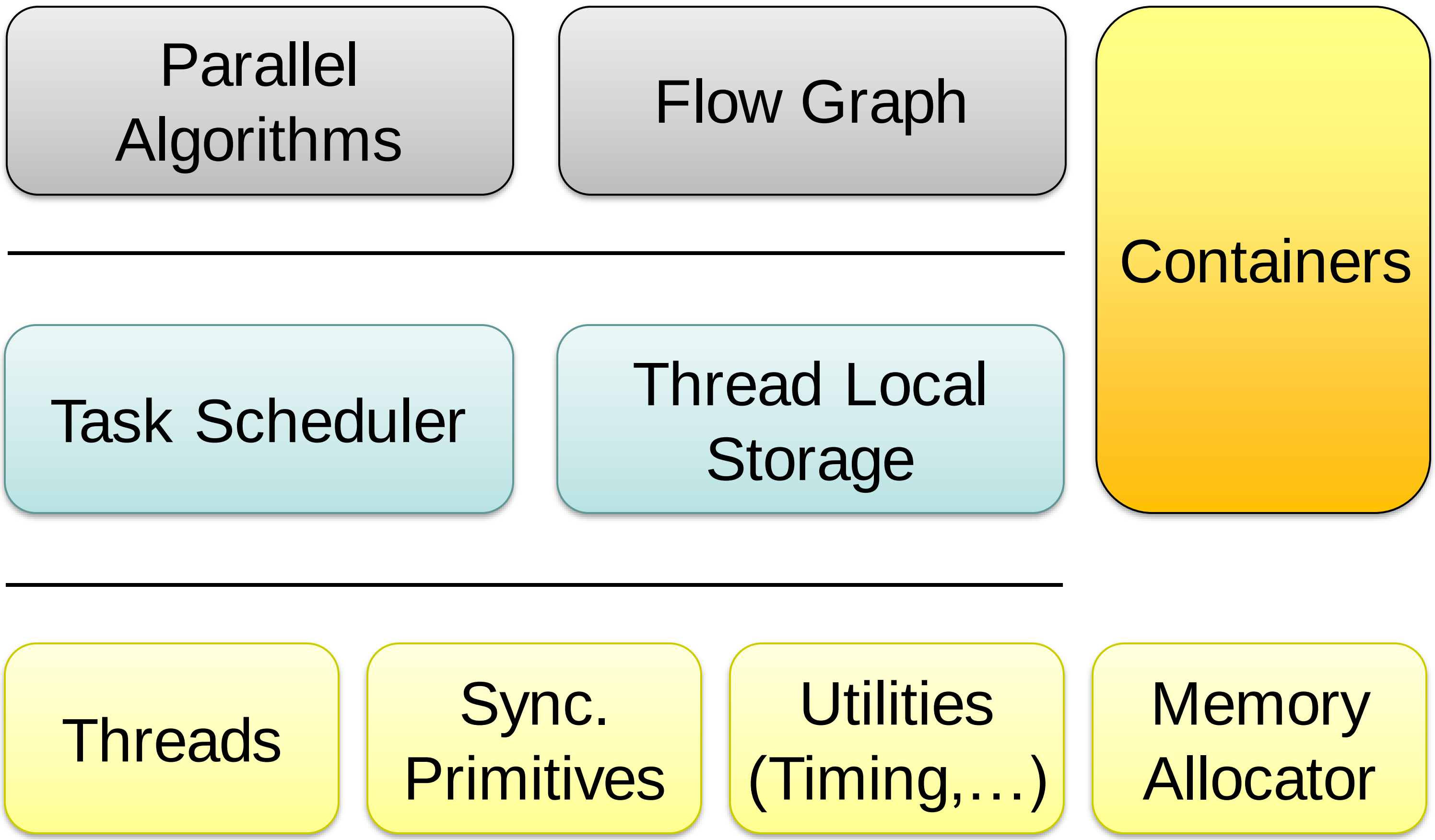




Introduction

- C++ library
  - Namespace tbb
- Templates
- Compatible with other threading libraries (pthreads, OpenMP,…)
- Works with tasks, not threads

Contents of the Library



Initialization

- automatic since TBB 2.2

```
#include <tbb/task_scheduler_init.h>
using namespace tbb;
int main()
{
  task_scheduler_init init;
}
```

Optionally, the number of threads may be specified

Parallel algorithms

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

parallel_for (type 1)

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

compare to

```
template <class InIter, class UnaryFunc>
UnaryFunc for_each(InIter first, InIter
  last, UnaryFunc f);
```

Range vs. Iterator

Splittable Concept

- A splittable object has the following constructor: `X::X(X& x, Split)`
- Unlike copy constructor, the first argument is not constant
- Divides (splits) the first argument into two parts
  - one is stored back into the first argument
  - other is stored in the newly constructed object
- Applies to both Range and Body
  - splitting of a range into two parts (first part into argument, second part into newly created object)
  - splitting body to two instances executable in parallel

Body

- Action (functor) to be executed in parallel

```
void Body::operator()(Range& range) const
```

- Unlike STL functor, the argument is a range
  - your job is to iterate the range

Range

- bool R::empty() const
- bool R::is_divisible() const
- R::R(R& r, split)
- split should create two parts of similar size
  - recommendation, not requirement

```
struct TrivialIntegerRange {
  int lower, upper;
  bool empty() const { return lower == upper; }
  bool is_divisible() const { return upper > lower+1; }
  TrivialIntegerRange(TrivialIntegerRange& r, split) {
    int m = (r.lower+r.upper) / 2;
    lower = m; upper = r.upper; r.upper = m;
  }
};
```

blocked_range<Value>

```
template<typename Value> class blocked_range;
```

- Value needs to support:
  - copy constructor, destructor
  - < (Value, Value)
  - + (Value i, size_t k) – k-th value after I
  - - (Value, Value) – returns size_t – distance
- Grainsize
  - minimal number of elements in a range
- Funkce `begin(), end()`
  - minimal and maximal value – half-closed interval

parallel_for (type 1) – Again

- Splits the range until it cannot be split further
  - `Range::is_divisible()`
- Creates a copy of body for each subrange
- Executes bodies over ranges

- Actual splitting and execution is done in parallel in a more sophisticated manner
  - Better use of CPU caches

blocked_range - Grainsize

- How to determine proper grainsize
  - depends on the actual algorithm
  - one call ~ at least 10.000 to 100.000 instructions
  1. set the grainsize to ~10.000
  2. run on one processor
  3. halve the grainsize
- Repeat steps 2 and 3 and observe slowdown
  - the performance decreases with growing overhead (1 core)
  - optimal grainsize is considered for 5-10% slowdown
- No exact way to get the best value

Partitioners

- Other solution to grainsize problem
- Third (optional) parameter to parallel_for
- Range may not be split to the smallest parts
  - simple_partitioner (default) – split to the smallest parts
  - auto_partitioner – enough splits for load balancing, may not provide optimal results
  - affinity_partitioner – same as auto_partitioner but better cache affinity

parallel_for (type 2)

```
template<typename Index, typename Func>
void parallel_for(Index first, Index last,
  Index step, const Func& f);
```

```
template<typename Index, typename Func>
void parallel_for(Index first, Index last,
  const Func& f);
```

- Index must be an integral type
- Semantics:

  ```
  for (auto i=first; i<last; i+=step) f(i);
  ```

parallel_reduce<Range,Body>

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

- Similar to parallel for, but returns a result
- Example: sum of all values
- New requirement for Body

```
void Body::join(Body& rhs);
```

  - join two results into one

parallel_reduce<Range,Body>

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

- One body may process more ranges (seq.)
- `operator()` and `join()` can run parallel with splitting constructor (but not each other)
  - does not cause problems in most cases
- Noncomutative operatation $\oplus$
  - `left.join(right)` should produce left $\oplus$ right

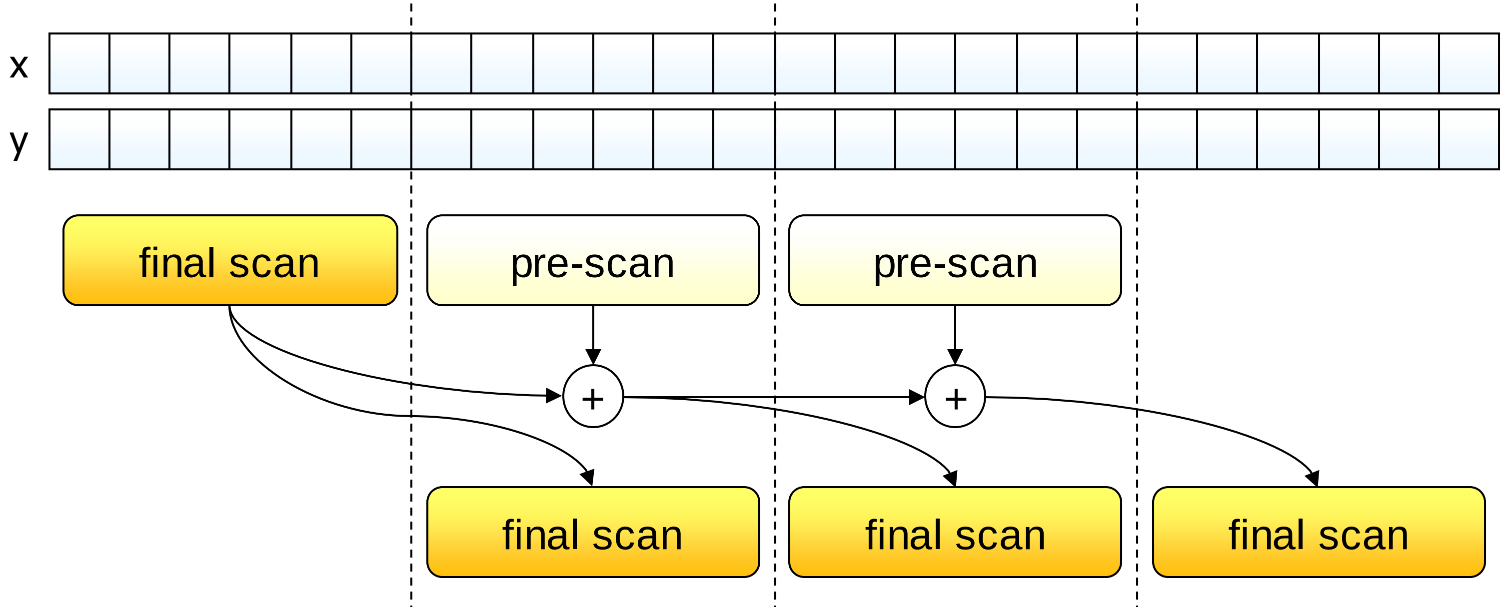
parallel_reduce (type 2)

```
template<typename Range, typename Value,
  typename Func, typename Reduction>
void parallel_reduce(const Range& range,
  const Value& identity, const Func& f,
  const Reduction& reduction);
```

- Identity is the left identity element for left operator of Func::operator()
- Value Func::operator() (const Range& range, const Value& x) accumulates subrange to the result starting with value x
- Value Reduction::operator() (const Value& x, const Value& y)
- Lambda-friendly, more data copying

parallel_scan<Range,Body>

- $y_0 = id_{\oplus} \oplus x_0$, $y_i = y_{i-1} \oplus x_i$
- $\oplus$ needs to be associative



parallel_do

```
template<typename InIter, typename Body>
void parallel_do(InIter first, InIter last,
  Body body);
```

- Different Body than e.g. parallel_for
  - `operator()` is unary and must be reentrant
    - Optional 2nd argument is feeder, which is used to add more work that is process by subsequent calls (for better scalability)
  - no splitting
- Not suitable for trivial operations
  - less than 10.000 instructions in Body::operator()

parallel_for_each

- Like STL's for_each
  - Same semantics, except for parallel processing
  - Equivalent with parallel_do
    - But without feeder

pipeline

- Sequence of filters

```
class pipeline {
public:
  void add_filter(filter& f);
  void run(size_t max_num_of_live_tokens);
};
```

- Filters are serial or parallel
  - serial is either in-order or out-of-order
- Filters derive from class filter and override virtual `void* operator()(void *item);`
  - the first filter’s operator() is called repeatedly until it returns NULL
  - the results returned by the last filter are ignored

parallel_pipeline

- Strongly typed lambda-friendly pipeline

```
void parallel_pipeline(
  size_t max_number_of_live_tokens,
  const filter_t<void,void>& filter_chain);
```

- Filters
  - combined using `&` operator
  - same modes as original pipeline
  - input and output type (first and last are voids)
  - functor for the action

parallel_pipeline

```
float RootMeanSquare(float* first, float* last) {
    float sum=0;
    parallel_pipeline(/*max_number_of_live_token=*/16,
        make_filter<void,float*>(filter::serial,
            [&](flow_control& fc) -> float* {
                if (first < last) return first++;
                else { fc.stop(); return nullptr; }
            }
        ) &
        make_filter<float*,float>(filter::parallel,
            [](float* p){ return (*p)*(*p); }
        ) &
        make_filter<float,void>(filter::serial,
            [&](float x) { sum += x; }
        )
    );
    return sqrt(sum);
}
```

Flow graph

- Generalization of the pipeline
  - arbitrary oriented graph
  - nodes and explicit (dataflow) connections
- More complex rules for execution
  - e.g., no explicit “number of live tokens”
- Pre-defined set of node types
  - function, join, limiter,…
- See the documentation…

parallel_sort

```
template<typename RndAccIter>
void parallel_sort(RndAccIter begin, RndAccIter end);
template<typename RndAccIter, typename Compare>
void parallel_sort(RndAccIter begin, RndAccIter end,
   const Compare& comp);
```

- Parallel unstable sort
- Average time complexity of O(N log (N))
- Usage:

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

parallel_invoke

- Parallel invocation of 2 to 10 functors
  - May be used to invoke completely different tasks in a simple manner

```
tbb::parallel_invoke(
    myfunc1,
    []{ bar(2); },
    []{ bar(3); },
);
```

Containers

```
concurrent_unordered_map<Key,Val,Hash,Eq,Alloc>
concurrent_unordered_set<Key,Hash,Eq,Alloc>
concurrent_hash_map<Key,T,HashCompare,Alloc>
concurrent_queue<T,Alloc>
concurrent_bounded_queue<T,Alloc>
concurrent_priority_queue<T,Compare,Alloc>
concurrent_vector<T>
```

- Selected operations are thread-safe
- Can be used with “raw” threads, OpenMP,…
- Designed for high concurrency
  - fine grained locking, lock-free algorithms
- Higher overhead than STL (due to concurrency)

concurrent_vector

- Concurrent growth and access
- Careful with exceptions
  - Special rules for item constructors and destructors
- Existing elements are never moved
  - iterators and references not invalidated by growth
  - results in fragmentation – call `compact()`
- Copy construction and assignment are not thread-safe

concurrent_vector

- **`grow_by(size_type delta [, const T& t]);`**
  - adds **`delta`** elements (atomically) to the end of vector
  - returns original size of the vector … *k*
  - new elements have indexes [*k*,*k*+`delta`)
- **`size_t push_back(const T& value);`**
  - atomically ads copy of value to the end of the vector
  - returns index of the new value
- access
  - operator[], at, front, back – all const and non-const
  - can be called concurrently while the vector grows
  - may return reference that is currently being constructed**!**

concurrent_vector

- range (const and non-const)
  - used in parallel algorithms
- size()
  - number of elements, including those currently being constructed
- standard ISO C++ random access iterators

concurrent_queue

- push, try_pop
  - non-blocking
- No STL-like front and back
  - can’t be made thread-safe
  - unsafe_begin, unsafe_end, unsafe_size
- concurrent_bounded_queue
  - added later, originally concurrent_queue was bounded
  - limits the maximal number of elements in queue
  - push and pop are blocking
    - active lock used for blocking operations – should not wait long!
  - size() returns number of pushes minus number of pops

concurrent_queue

- Order of items partially preserved
  - if one threads pushes X before Y and another thread pops both values, then X is popped before Y
- Provides iterators
  - for debugging only, invalidated on modification
  - STL compliant iterator and const_iterator
- To be used wisely
  - maybe parallel_do or pipeline can be used as well
  - parallel algorithms perform better – avoid one bottleneck (the queue), better use caches, …
  - resist the temptation to implement producer-consumer (or similar) algorithms

concurrent_priority_queue

- Similar to STL priority_queue
  - With fixed underlying container
  - try_pop() instead of pop()
  - thread-unsafe empty() and size()

concurrent_unordered_map

- Similar to std::unordered_map
- Insertion and iteration are safe
  - Even combined
  - Insertion does not invalidate iterators
- Erasing is unsafe
- No visible locking
- Same analogy goes for unordered_set

concurrent_hash_map

```
template<typename Key, typename T,
  typename HashCompare, typename Alloc>
class concurrent_hash_map;
```

- Provides accessors
  - **`const_accessor`** and **`accessor`**
  - smart pointer with implicit lock (read or RW lock)
  - cannot be assigned or copy constructed
  - operators * and ->
  - explicit release of locks – call `release()`
    - if points “to nothing” – can be tested by `empty()`

concurrent_hash_map

- Concurrent operations
  - `find` (returns const or non-const accessor)
  - `insert` (returns const or non-const accessor)
  - `erase`
  - copy constructor
    - the copied table may have operations running on it
- `range(size_t grainsize)`
  - returns (constant or non-constant) range that can be used for parallel algorithms to iterate over contents of the map
  - cannot be run concurrently with other operations
- Forward iterators – `begin()`, `end()`

Memory Allocation

- Optional part of the library
  - C++ and STL compliant allocators
  - C functions (malloc and free)

```
template<typename T, size_t N>
  class aligned_space;
```

- variable that has size big enough to contain N elements of type T cache-aligned in memory
- `T* begin(), T* end()`
  - actual begin and end of the aligned data
- to be used as a local variable or field

Memory Allocation

- scalable_allocator
  - each thread has its own memory pool
    - no global lock, prevents false sharing of private memory
- cache_alligned_allocator
  - all functionality of scalable_allocator
  - aligns memory to cache lines
    - prevents false sharing of separate allocations
    - always pads data to cache line size (usually 128bytes) **!**
  - use only when it proves to be needed
- tbb_allocator
  - uses scalable_allocator if possible (TBB malloc is present) or the standard malloc/free

Explicit Synchronization

- Mutexes
  - mutex
  - recursive_mutex
  - spin_mutex
  - queuing_mutex
  - spin_rw_mutex
  - queuing_rw_mutex
- atomic<T>
  - provides atomic operations over a type
    - fetch_and_increment, compare_and_swap, operator++, operator+=, …

Mutexes

- Many variations
  - different implementation (OS / user space, …)
  - different semantics (simple/recursive, fair/unfair, plain/read-write)
- Scoped locking pattern
  - acquired lock is represented by an object
  - no explicit unlocking – less prone to errors
  - exception safe

Thread Local Storage

- Separate variable for each thread
  - lazily created (on demand)
- `combinable<T>`
  - reduction variable, each thread has a private copy
  - combined using a bin. functor or unary (for_each)
- `enumerable_thread_specific<T>`
  - container with element for each thread (lazy)
  - can access either local value or iterate all values
  - may be combined using a binary functor

Timing

- Thread-safe timing routines
- Classes tick_count and tick_count::interval_t

```
tick_count t0 = tick_count::now();
... action being timed ...
tick_count t1 = tick_count::now();
printf("time for action = %g seconds\n",
   (t1-t0).seconds());
```

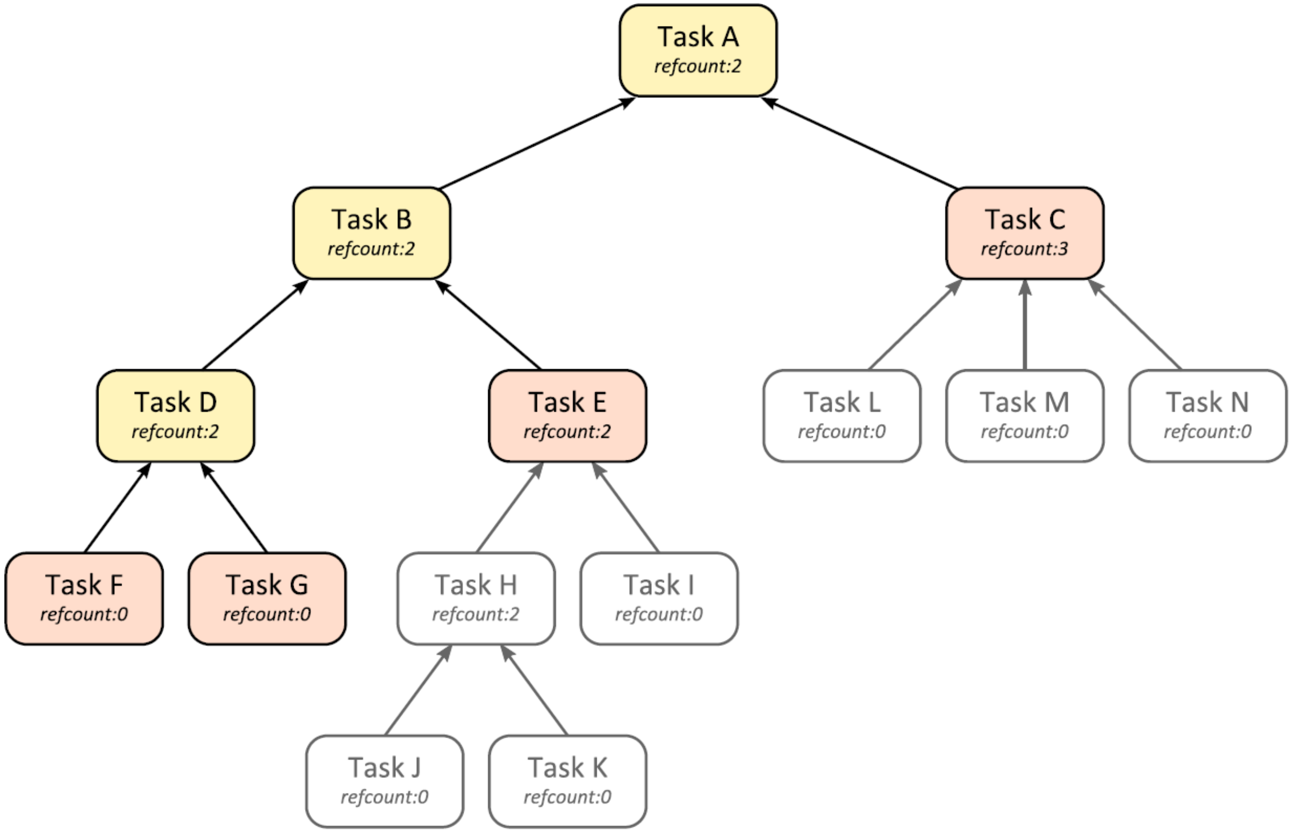
Task Scheduling

- Task scheduler assigns tasks to threads
- Maintains task graph
  - directed graph of dependent task
  - parent node can start only when all children have finished

```
class task {
public:
    task* execute();
};
```

Task Scheduling

- Example of a task graph

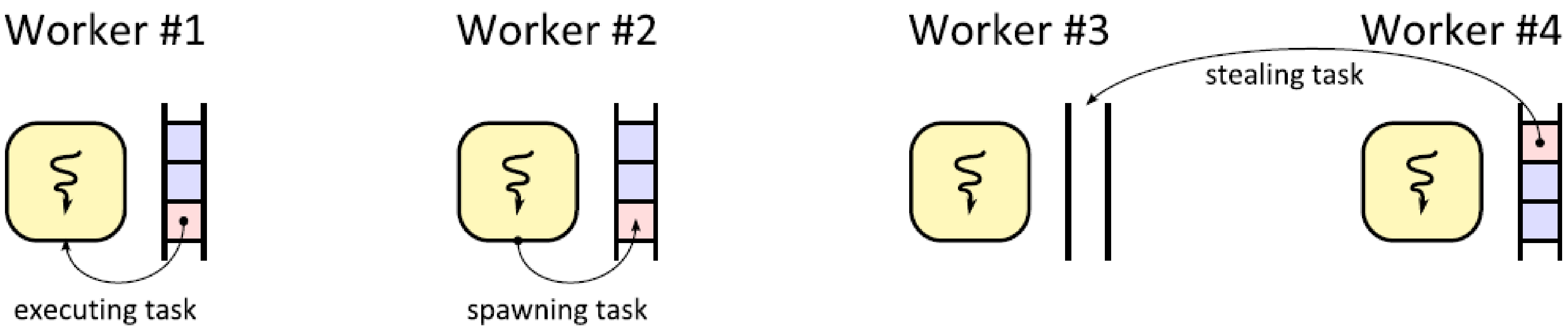


Task Scheduling

- TBB offer interface to directly work with tasks
- Used to explicitly create tasks and their relations
- Declarative
  - provides data to the task scheduler
  - does not control the scheduler itself

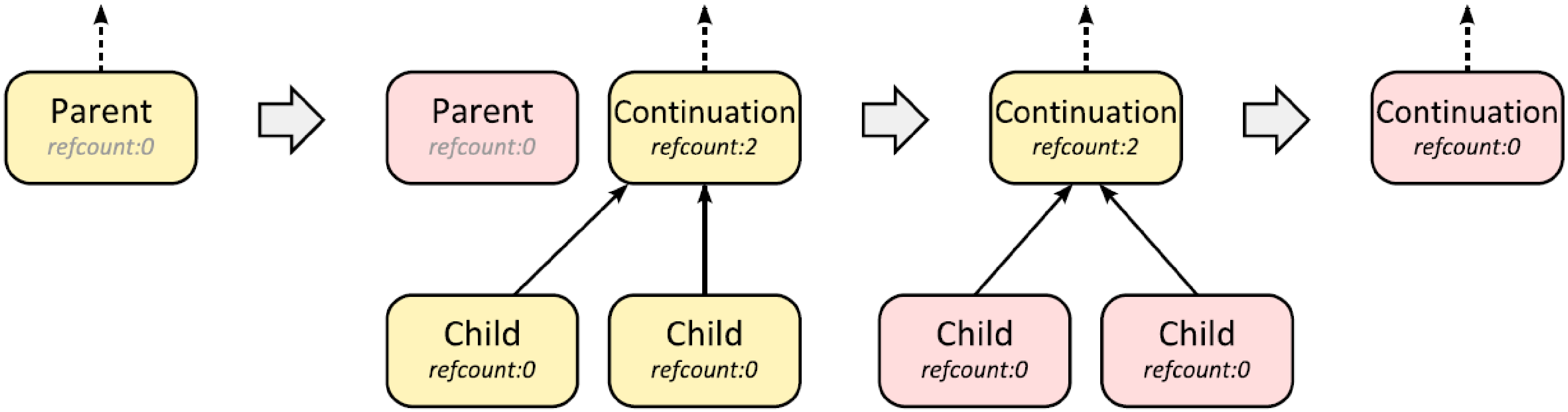
Task Scheduling

- Each thread has its own pool of ready tasks
  - task is ready when it has no children
  - new tasks are spawned to the front of the stack
  - tasks are stolen from the end of the stack



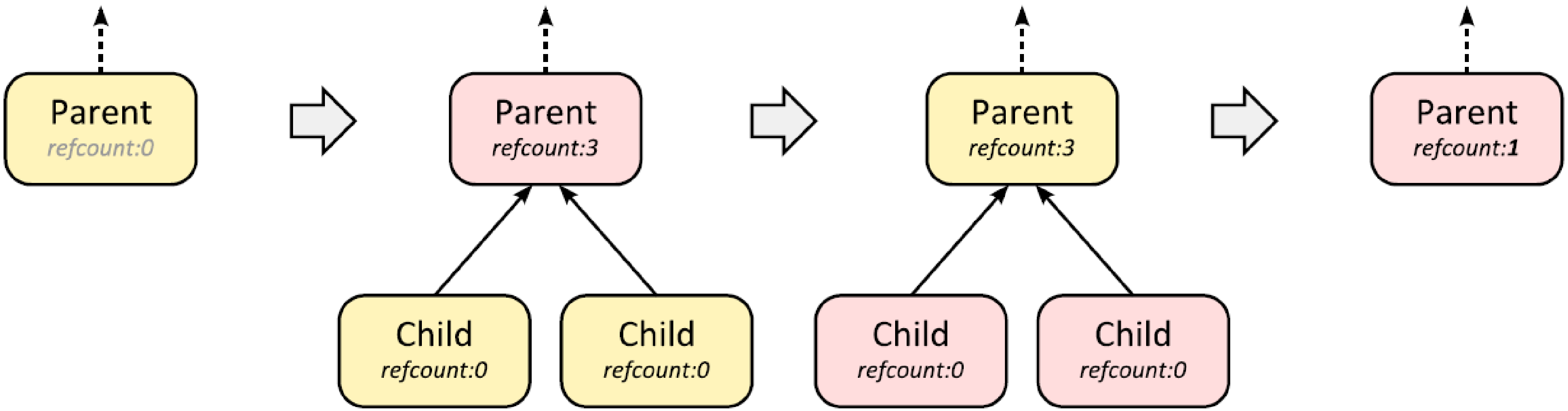
Spawning Child Tasks

- Spawning children with continuation
  - Parent spawns children and continuation task
  - Continuation is executed automatically when children terminate



Spawning Child Tasks

- Blocking task spawning
  - Parent spawns the children
  - Blocks on wait operation until the children terminate



Task Scheduling

- Summary
  - other algorithms (parallel_for, …) are translated into tasks
    - therefore, the algorithms can be easily nested
    - use tasks explicitly only if you cannot fit any of the parallel algorithms provided
  - task as scheduled non-preemptively
    - should not block