



Intel

Threading Building Blocks

Alexandre David

1.2.05

adavid@cs.aau.dk





What is TBB?

- C++ library for multi-threading.
 - Internally uses pthreads (Linux).
 - Abstracts from threading details.
 - Based on tasks.
 - Offers concurrent data-structures.
 - C++
 - Dual licensed GPL/commercial.



Benefits

- Specify tasks instead of thread.
 - Thread programming: map work to threads, do the load balancing etc...
 - Task programming lets the library schedule threads for you.
 - Abstraction on raw threads, more portable.
- Threading for performance.
 - Higher level simple solutions for computationally intensive work.
- Compatible with other threading packages.
 - Mix with OpenMP or pthreads.



Benefits

- TBB emphasizes scalable data-parallel programming.
 - Data-parallel programming scales well with large problems – partition data set.
 - Special constructs to do the partitioning.
- Generic programming.
 - Write best possible algorithms with as few constraints as possible.



Important Concepts

- Recursive splitting.
 - Break problems recursively down to some minimal size.
 - Works better than static division, works well with task stealing.
- Task stealing.
 - A way to manage load balancing.
- Generic algorithms
 - algorithm templates.



Overview

- Algorithms
 - parallel_for
 - parallel_reduce
 - parallel_scan
 - parallel_while
 - pipeline
 - parallel_sort
- Concurrent containers
 - concurrent_queue
 - concurrent_vector
 - concurrent_hash_map



Basic Algorithms

- Loop parallelization
 - `parallel_for`
 - `parallel_reduce`
 - `parallel_scan`
 - → building blocks.



Start & End

- Need to start task scheduler.
- Declaring: `task_scheduler_init` init;
in main does the job.
- Can be tweaked but the default is usually good enough.
 - Number of threads automatic.



parallel_for

Original code:

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



parallel_for

Algorithm class:

```
#include "tbb/blocked_range.h"
class ApplyFoo
{
    float *const my_a;
public:
    void operator()(const blocked_range<size_t>& r) const
    {
        float *a = my_a;
        for(size_t i = r.begin(); i != r.end(); ++i) Foo(a[i]);
    }
    ApplyFoo(float a[]) : my_a(a) {}
};
```



parallel_for

Algorithm call:

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

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



Recursive Splitting

- General form of the constructor:
`blocked_range<T>(begin,end,grainsize)`
 - [Setting the grain to 10000 is a good rule of thumb. The grain should take 10000-100000 instructions at least.]
- This range is used to do recursive splitting automatically.
 - If *currentSize* > grainsize then split.
 - It's not the minimal size of the data-sets.
 - Minimum threshold for parallelization.
 - Concept → minimum block size.



Automatic Grain Size

- New version of TBB support automatic grain sizes.
 - The algorithms (`parallel_for...`) need a partitioner.
 - There's a default `auto_partitioner()`.
 - It's using heuristics.



Aha - Recursive Algorithms

- How to implement recursive algorithms using `parallel_for`?
 - Define your own range splitting class.
 - Call `parallel_for`.
 - TBB will split recursively as needed.



parallel_reduce

Original code:

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



parallel_reduce

Algorithm class:

```
class SumFoo
{
    float* my_a;
public:
    float sum;
    void operator()(const blocked_range<size_t>& r)
    {
        float *a = my_a;
        for(size_t i = r.begin(); i != r.end(); ++i) sum += Foo(a[i]);
    }
    SumFoo(SumFoo& x, split) : my_a(x.my_a), sum(0) {}
    void join(const SumFoo& y) { sum += y.sum; }
    SumFoo(float a[]) : my_a(a), sum(0) {}
};
```




Reduce

- Associative operator.
- Recursive algorithm to compute it.
 - Schwartz' algorithm.
- TBB:
 - splitting constructor
 - non-const method to compute on blocks
 - join to combine results



parallel_reduce

Call:

```
float ParallelSumFoo(const float a[], size_t n)
{
    SumFoo sf(a);
    parallel_reduce(blocked_range<size_t>(0,n,GrainSize),
                   sf);
    return sf.sum;
}
```



parallel_scan

Methods needed:

```
class Body {
    T reduced_result; ... x & y data
public:
    Body(x & y)...
    T get_reduced_result() const { return reduced_result; }
    void operator()(range, tag) {
        T temp = reduced_result;
        for(i : range) {
            temp <op>= x[i];
            if (tag::is_final_scan()) y[i] = temp;
        }
        reduced_result = temp;
    }
    Body(Body&b, split) - split constructor
    void reverse_join(Body& a) {
        reduced_result = a.reduced_result <op> reduced_result;
    }
    void assign(Body& b) { reduced_result = b.reduced_result; } };
```



parallel_scan

- One class to define the operations for both passes of the algorithm (recall 2 passes).
 - Differentiation with `is_final_scan()`.
 - `prescan` computes the reduction, doesn't touch `y`.
 - `final scan` updates `y`.
 - `reverse_join`: *this* is the right argument.



Advanced Algorithms

- Different kinds of parallelizations:
 - parallel_while
 - suitable for streams of data
 - pipeline
 - parallel_sort



parallel_while

Original code:

```
void SerialApplyFooToList(Item *root)
{
    for(Item* ptr = root; ptr != NULL; ptr = ptr->next)
        Foo(ptr->data);
}
```



parallel_while

```
class ItemStream
{
    Item *my_ptr;
public:
    bool pop_if_present(Item* & item) {
        if (my_ptr) {
            item = my_ptr;
            my_ptr = my_ptr->next;
            return true;
        } else {
            return false;
        }
    }
    ItemStream(Item* root) : my_ptr(root) {}
};
```



parallel_while

- The class acts as an item generator and writes items where specified.
- The `pop_if_present` does not need to be thread safe because it is never called concurrently.
 - This makes it non-scalable – could be a bottleneck.
 - It makes more sense when `parallel_while` can acquire more work: call to `parallel_while::add(item)`.



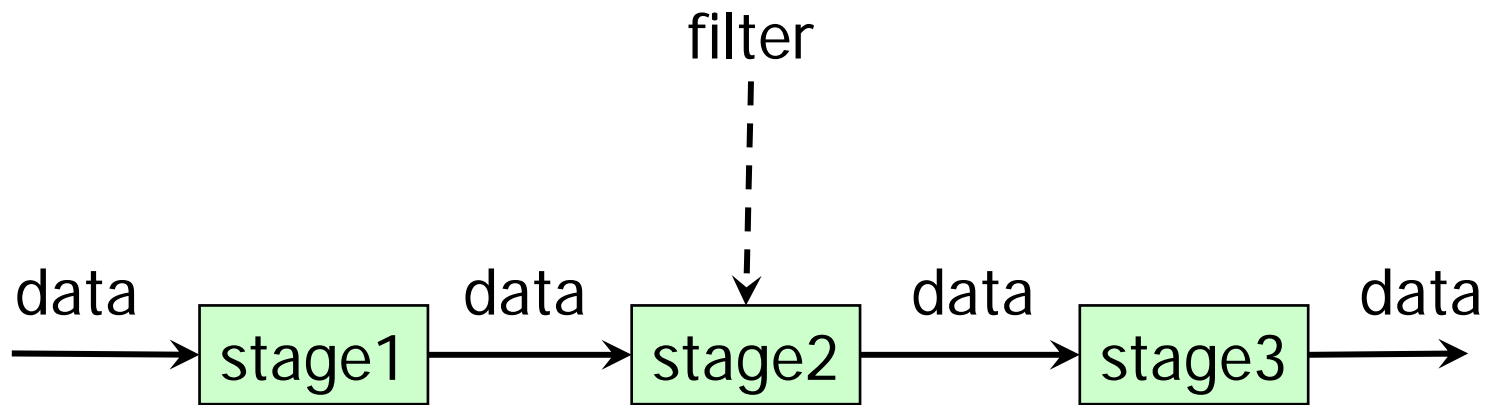
parallel_while

(functor)

```
class ApplyFoo {
public:
    void operator()(Item* item) const {
        Foo(item->data);
    }
    typedef Item* argument_type;
};

void ParallelApplyFooToList(Item* root) {
    parallel_while<ApplyFoo> w;
    ItemStream stream;
    ApplyFoo body;
    w.run(stream,body);
}
```

Pipelining



TBB: One stream of data – linear pipeline.



Filter Interface

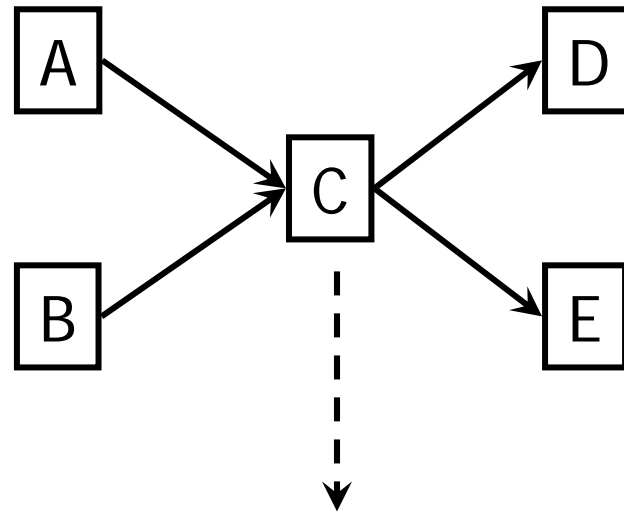
```
namespace tbb {  
    class filter {  
    protected:  
        filter(bool is_serial);  
    public:  
        bool is_serial() const;  
        virtual void* operator()(void* item) = 0;  
        virtual ~filter();  
    };  
}
```



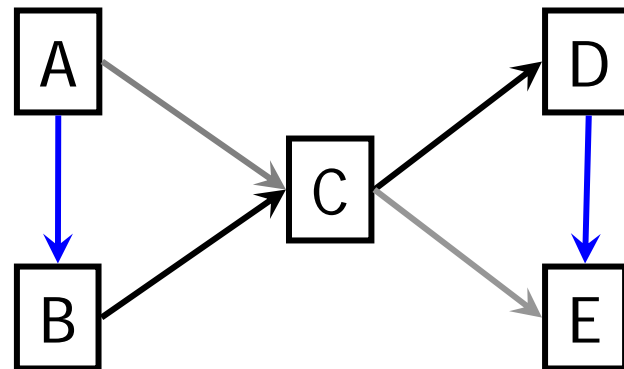
Building Pipelines

```
tbb::pipeline pipeline;  
  
MyInputFilter input(args);  
pipeline.add_filter(input);  
  
MyTransformFilter transform(args);  
pipeline.add_filter(transform);  
  
MyOutputFilter output(args);  
pipeline.add_filter(output);  
  
pipeline.run(buffer_args);  
  
pipeline.clear();
```

Non-Linear Pipelines



Topologically sorted pipeline





parallel_sort

- `parallel_sort(i,j,comp)`.
- Types `i` and `j` are compared using `comp` (functor).
- Types `i` and `j` must be accessible randomly (are `std::RandomAccessIterator`).
- Uses quicksort internally, average time $O(n \log n)$.



Concurrent Queue

- `concurrent_queue<T>`
 - no allocator argument, uses scalable allocators.
 - `pop_if_present`, `pop` (blocks).
 - `size()` (signed) = `#push - #started pop`
if `<0` then there are pending pops.
 - `empty()`
 - no `front()` or `back()` – could be unsafe.
- Inherently bottlenecks, threading explicit, passive structure.



Concurrent Vector

- `concurrent_vector<T>`
 - similar to `stl`
- Iterators supported.



Concurrent Hash Table

- `concurrent_hash_map<Key, T, HashCompare>`
- `HashCompare` is a trait.
 - `static size_t hash(const Key& x)`
`static bool equal(const Key& x, const Key& y)`
- Read/write access by accessor classes
 - `const_accessor`
`accessor`
 - ~ smart pointers.
 - Accessors lock elements.



concurrent_hash_map

- Interesting methods:
 - `bool insert(const accessor& result, const Key& key);`
 - `bool erase(const Key& key);`
 - `bool find(const accessor& result, const Key& key) const;`
- Iterators supported too.



Memory Allocation

- You know of false sharing.
- Scalable allocator allocates in multiple of cache line sizes and pads memory.



Locks

- Support for locks.
 - `scoped_lock` object, keeps exception safety.
 - Can use constructor argument to avoid lock-unlock, like `synchronized` in Java.

```
typedef spin_mutex MyMutex;  
MyMutex myMutex;  
  
...  
{  
    MyMutex::scoped_lock mylock(myMutex);  
    ...  
}  
or  
MyMutex::scoped_lock lock;  
lock.acquire(myMutex);  
  
...  
lock.release();
```

Different types of locks available, good to use a typedef to change if needed.

`mutex`, `spin_mutex`, `queuing_mutex`...



Atomic Operations

- `atomic<T>`
 - some simple scalar atomic operations supported,
 - compare and swap