Intel Thread Building Blocks, Part III

SPD course 2013-14

Massimo Coppola

16/05/2014

reduce

- Reduce has also two forms
  - "*Functional*" from, nice with lambda function definitions
  - "*Imperative*" form, minimizes data copying
  - *Please remember this is just TBB terminology*

```
template<typename Range, typename Value, typename
        Func, typename Reduction>

Value parallel_reduce( const Range& range,
        const Value& identity, const Func& func,
        const Reduction& reduction,
        [, partitioner[, task_group_context& group]] );

template<typename Range, typename Body>

void parallel_reduce( const Range& range,
        const Body& body
        [, partitioner[, task_group_context& group]] );
```

"Functional" form

- Beside the function, several other objects have to be passed to the reduce

- Value Identity
  - left identity for the operator
- Value Func::operator()(const Range& range, const Value& x)
  - must accumulate a whole subrange of values starting from x ("sequential reduction")
- Value Reduction::operator()(const Value& x, const Value& y);
  - Combines two values ("parallel" reduction)

Object-oriented form

- Computes the reduction on its Body object together with the associated Range
  - Data (reference) is held within the Body
  - The reduce can split() the body parameter, and will split() the range accordingly
  - Can also split only the range, and compute over a range that is smaller than the Body's data
    - This may allow saving some data copy operation when we exploit parallel slackness together with affinity
  - Results form each side will the be combined
- Body object's state contains the reduced value
  - Final result is accumulated in initial Body object

Reduce

- Both the function-based form and the OO one can specify a custom partitioner
- Both forms can specify a task group that will be used for the execution

Reduce – deterministic variant

- parallel_deterministic_reduce
- Performs a deterministically chosen sets of splits, joins and computations
- Exploits the simple_partitioner → no partitioner argument allowed
- Computes the same regardless of the number of threads in execution
  - no adaptive work assignment is ever performed
  - grain size must be carefully chosen in order to achieve ideal parallelism
- Has both the functional form and the OO one

pipeline

- Pipeline pattern
  - pipeline class not strongly typed
  - parallel_pipeline strongly typed interface
- Implements the pipeline pattern
  - A series of filter applied to a stream
    - You need to subclass the abstract filter class
  - Each filter can work in one of three modes
    - Parallel
    - Serial in order
    - Serial out of order

Pipeline class

- Pipeline is dynamically constructed
  - pipeline() create an empty pipeline
  - ~pipeline() destructor
  - void add_filter(filter& f) add a filter
  - clear() remove all filters
  - void run(size_t max_number_of_live_tokens
    [, task_group_context& group])
- Run until the first filter returns NULL
- Actual parallelism depends on pipeline structure, and on parameter
  - max_number_of_live_tokens
- Pipelines can be reused, but NOT concurrently
- Stages can be added in between runs
- Can have all tasks belong in a specified optional group, by default a new group is created

filter

- Abstract class implementing filters for pipelines
- Three modes, specified in the constructor
  - Parallel can process/produce any number of item in any order (e.g. nested parallelism)
  - Serial out of order filter processes items one at a time, and in no particular order
  - Serial in order filter processes items one at a time, in the received order
- Computation is specified by overriding the operator ()
  - virtual void* operator()(void * item)
  - Process one item and return result, via pointers
  - First stage signals with NULL the end of the stream
  - Result of last stage is ignored

Parallel_pipeline

- void parallel_pipeline(
  size_t max_number_of_live_tokens,
  const filter_t<void,void>& filter_chain
  [, task_group_context& group]);
- Strongly typed, can use lambdas
  - parallel_pipeline(max_number_of_live_tokens,
    make_filter<void,I1>(mode0,g0) &
    make_filter<I1,I2>(mode1,g1) & ...
    make_filter<In,void>(moden,gn));
- Employ the make_filter template to build filters on the spot from their operator() function
- Types are checked at compilation time
  - First stage must invoke fc.stop() and return a dummy value to terminate the stream

Parallel_do

```
template<typename InputIterator,
                 typename Body>
void parallel_do( InputIterator first,
                  InputIterator last, Body body
                  [, task_group_context& group] );
```

- Only has the object oriented syntax
- Applies a function object body to a specified interval
  - The body can add additional tasks dynamically
  - Replaces completely the deprecated parallel_while
  - Iterator is a standard C++ one
  - A purely serial input iterator is a bottleneck: use iterators over random-access data structures

Adding items in a do

B::operator()(T& item,
parallel_do_feeder<T>& feeder) const

B::operator()(T& item) const

- The body class need to operate on the template T type
- It needs a copy constructor and a destroyer
- Two possible signatures for Body operator()
  - You can't define both!
  - First signature, with extra parameter, allows each item to add more items dinamically in the do → e.g. dynamically bound parallel do, divide & conquer
  - Second signature means the do task set is static

Containers

- container_range
  - extends the range to use a container class
- maps and sets:
  - concurrent_unordered_map
  - concurrent_unordered_set
  - concurrent_hash_map
- Queues:
  - concurrent_queue
  - concurrent_bounded_queue
  - concurrent_priority_queue
- concurrent_vector