

Intel Thread Building Blocks

SPD course 2012-13

Massimo Coppola

9/11/2012

Thread Building Blocks : History

- A library to simplify writing thread-parallel programs and debugging them
- Originated circa 2006 as a commercial product
 - First version was still very low-level
 - Little more than a debugging tool
 - Strong emphasis was put on how to performance debug thread-parallel programs
- Several releases improved the abstraction level
 - Nowadays a programming model

Thread Building Blocks Today

- Version 4.0 released Sept. 2011
 - V4.1 released Sept 2012
- A C++ based pattern language for threads
 - Supports generic programming
 - Supports nested parallelism
- Nowadays double licensed
 - Commercial version for industrial users
 - Open source version under GPLv2
 - Stable versions aligned with commercial ones
 - Developer, source-only versions
 - Multi-OS
 - Windows*, Linux, OS X direct support
 - Other o.s. support in the open source

What is TBB today

- A runtime + template library for C++
- Eases writing thread programs by raising the abstraction level
 - Tasks production and processing instead of threads
- Templates and classes are defined for
 - Common parallelism forms
 - Data structures to pass to these parallel “skeletons”
 - Heavy use of generics for expressiveness
 - Data structures to control parallelism
 - e.g. **range** to define the set of values of a parameter
 - Operators to specify each skeleton semantics
 - A form of encapsulation of sequential behaviour

TBB Runtime support

- Two support libraries
- Provide operating system-independent basic primitives
- One library for
 - Task generation
 - Parallel patterns
 - Task scheduling to threads,
- A specific library for scalable memory allocation
- The two can also be used independently

Some supported abstractions

More patterns added with each version

- **parallel_for**
- **lambda expressions**
- **parallel_reduce**
- parallel_do
- pipeline
 - Extended to dags as supersets of pipelines
- **concurrency-safe containers**
- **mutex helper objects**
- atomic<t> template (atomic operations)

- Express independent task computations
 - `parallel_for` (iteration space , function)
- Exploit a `blocked_range` template to express iteration space
 - Ranges can be recursively split by the library
 - 1D, 2D, 3D blocked ranges as of TBB 4.0
- Automatic dispatch to independent threads
 - Heuristics within the library
 - Can be customized
 - Specify optional *partitioner* function to the `parallel_for`
 - Specify *grainsize* parameter in the range

Example

```
#include "tbb/tbb.h"
using namespace tbb;
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)
    {}
};

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


Scheduling tasks to threads

- Task scheduler
 - Automatically created by the library
 - Customizable by program to suit user needs
 - Define scheduler creation/destruction time
 - Number of created threads
 - Stack size for threads
 - Customizable per construct
 - via construct parameters
- Much more in the docs about the scheduler
 - Task scheduler deals with pipelines and workflows

Choosing grain size

- As always, small grain size → high overhead
 - Intel suggests 100.000 clock cycles as grain size
 - Also suggests experimental procedure to set
 - You are expected to already know the issues, and take into account the number of cores and load balancing details in your algorithm
- Cache affinity can impact performance
 - *affinity partitioner* tries to exploit it when scheduling tasks to threads

Type	Use	Conditions
simple	Chunks given by grain size (Default until TBB 2.2)	$g/2 < \text{chunk size} < g$
auto	Automatic size (heuristics, default nowadays)	$g/2 < \text{chunks size}$
affinity	Automatic size (heuristics to exploit affinity)	$g/2 < \text{chunks size}$

Lambda expression

- Unnamed functions defined by the latest C++ 0x standard (ISO/IEC 14882:2011)
 - Released September 2011
- Use a stereotype for in-place defining an unnamed free function (some support for storing the def)
[variable_scope] type_def function_def;
- Capture all variable references which are used inside, but defined outside the function
- Variable scope spec can dictate capturing by reference, by value, or disallow use
 - In general, e.g. [] disallow [=] by value [&] ref.
 - For specific variable(s)
[=,&z] all by value, with only z by reference

Parallel reduce

- Expresses the parallel reduction pattern
 - Basic form is analogous to the parallel for `parallel_reduce (iteration_space, function)`
 - Iteration space defined as `blocked_range`
 - The function to apply has different C++ type template w.r.t to parallel loop
 - Reduce operator does not have the same const-requirements as the one used in a for
 - Also accepts an optional partitioner

- Data structures
 - which are very often used in programs,
 - whose thread-safe implementation is not trivial
 - or it does not match standard semantics
- Special care taken to avoid decreasing program performance
- `concurrent_hash_map`
 - Constant or update access to elements
 - Access to elements can block other threads

- `concurrent_vector`
 - Can grow as normal vector
 - Does not move its elements in memory
 - Destroying elements is not thread safe
- `concurrent_queue`
 - Simultaneous push/pop from concurrent threads
 - Ensure serialization and preserve object order
 - Bottleneck if improperly used
 - `pop` / `push` / `try_push` / `size`

- Classes to build *lock objects*
- The new lock object will generally
 - Wait according to specific semantics for locking
 - Lock the object
 - Release lock when destroyed
- Several characteristics of mutexes
 - Scalable
 - Fair
 - Recursive
 - Yield / Block
- Check implementations in the docs:
 - mutex, recursive_mutex, spin_mutex, queueing_mutex, spin_rw_mutex, queueing_rw_mutex, null_mutex, null_rw_mutex
 - Specific reader/writer locks
 - Upgrade/downgrade operation to change r/w role

- Download docs and code from <http://threadingbuildingblocks.org/>
- Check the accompanying docs
 - Getting started – install and first compilation example **← TRY IT**
 - Tutorial – tour of main functionalities with examples
 - Reference
- Quick summary to lambda expressions in C++
 - http://www.nacad.ufrj.br/online/intel/Documentation/en_US/compiler_c/main_cls/cref_cls/common/cppref_lambda_desc.htm