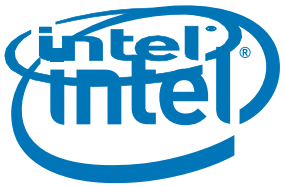


Threading for Performance with Intel® Threading Building Blocks

Lab Document

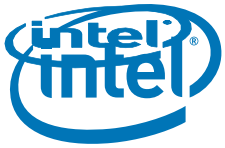
April 2008

Revision 1.0

Intel® Academic Community

Time Required	Seventy-five minutes
Objectives	In this lab, you will practice writing threaded code using Intel® Threading Building Blocks. At the successful completion of these lab activities, you will be able to: • Apply the TBB parallel_for and parallel_reduce generic algorithms for loop parallel computations • Generate a recursive execution tree of tasks that are scheduled by the TBB task scheduler • Use Intel Threading Building blocks concurrent containers and scalable memory allocation

Module Number: MC325_1_0



INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, or life sustaining applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

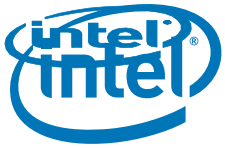
Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

*Other names and brands may be claimed as the property of others.

Copyright © 2008, Intel Corporation. All rights reserved.

Contents

© April 2008 Intel® Corporation



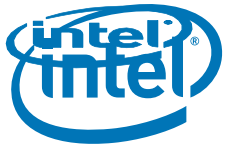
Revision History

Document Number	Revision Number	Description	Revision Date
MC325_1_0	1.0	Initial release.	April 2008

Note: On Windows platforms, if you experience link errors, try switching the configuration between "win32" and "x64" and relinking. Depending on the library available on your system, there may be compatibility issues that could be cleared up by matching the library version to the compiled code

§

© April 2008 Intel® Corporation



Activity 1: Using `parallel_for`

Time Required	Fifteen minutes
Objective	Modify a serial matrix multiplication code to do computations in parallel through the Intel TBB `parallel_for` generic algorithm.

The application generates two NxN matrices and then does a matrix multiplication on these two matrices. The code contains two separate matrix multiplication calls: one in serial and one in parallel (though the parallel version initially calls the serial function). The two calls are timed in order to see if the parallel version (when run on a multi-core processor) will run in less time than the serial version.

Build and Run Serial Program

1. Locate and change to the **01 Matrix Multiply** directory. You should find Visual Studio Solution and project files and a source file, mxm_serial.cpp, in this directory.
2. Double-click on the solution icon and examine the source file.
3. Be sure the Release configuration is selected and build the executable binary.
4. After successfully compiling and linking, run the executable.

   This can be done from the Visual Studio IDE by selecting the "Start without Debugging" command (CTRL+F5). The output reports a serial and a parallel time, that should be close to each other, and a computed speed up of the parallel over the serial time.

Modify Serial Program to use Intel TBB `parallel_for`

1. Modify the ParallelMxM function to perform the matrix multiplication in parallel using Intel TBB

   a. Create a Body class and define the operator() to perform multiplications across a range of index values. At what loop level should the parallelism be implemented?

   b. Replace original body of the ParallelMxM function with an execution(s) of `parallel_for` using your defined Body class and the TBB defined **blocked_range**.

2. Compile and debug the application. Once you have a clean compilation, run the parallel executable.

   What was the speedup of the parallel version? _____

© April 2008 Intel® Corporation

Activity 2: Using `parallel_reduce`

Time Required	Twenty minutes
Objective	Modify a numerical integration code to do computations in parallel and collect a computed reduction through the Intel TBB `parallel_reduce` generic algorithm.

The application computes an approximation of pi (3.1415926…) through numerical integration using the midpoint rectangle rule. Thus, for a given number of steps between 0.0 and 1.0, the function f(x) = 4.0 / (1 + x*x) is evaluated, which corresponds to the height of the rectangles. The sum of all the rectangle heights is computed and this is multiplied by the inverse of the number of steps (width of rectangles) to compute pi.

The computation is timed in order to see if the parallel version (when run on a multi-core processor) will run in less time than the serial version.

Build and Run Serial Program

1. Locate and change to the **02 Numerical Integration** directory. You should find Visual Studio Solution and project files and a source file, pi.cpp, in this directory.
2. Double-click on the solution icon and examine the source file.
3. Be sure the Release configuration is selected and build the executable binary.
4. After successfully compiling and linking, run the executable. This can be done from the Visual Studio IDE by selecting the "Start without Debugging" command (CTRL+F5).

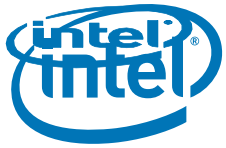
   What is the serial time of the application? _____

Modify Serial Program to use Intel TBB `parallel_reduce`

1. Modify the application to perform the repeated computations of the function being integrated in parallel using Intel TBB. Each task created will compute a separate, local copy of the sum of rectangle heights that need to be gathered and summed (reduced) in to the final global sum.

   a. Create a Body class and define the operator() to perform function computations across a range of index values. You will also need to define a **split** (to initialize the local sum) and a **join** method (to combine two local sum values) in order to use the `parallel_reduce`.

© April 2008 Intel® Corporation



b. Replace original body of the for-loop computations with a call to `parallel_reduce` using your defined Body class and the TBB defined **blocked_range**.

2. Compile and debug the application. Once you have a clean compilation, run the parallel executable.

   What is the execution time of the parallel version? _____

   What was the speedup of the parallel version? _____

© April 2008 Intel® Corporation

Activity 3: Generating Recursive Tasks

Time Required	Fifteen minutes
Objective	Modify a partially parallelized binary tree traversal application. The traversal is done in parallel by creating recursive of TBB tasks for each branch of the tree.

Modify, Build, and Run Threaded Program

1. Locate and change to the **03 recursive_tasks** directory.

    a. The `original.h` file contains the SerialTreeTraversal function, which implements serial tree traversal. The nodes of the tree can be processed independently. The task scheduler interface that Intel(R) Threading Buidling Blocks (Intel(R) TBB) provides naturally supports recursive parallelism.

    b. The `TODO.h` contains an incomplete an implementation of the `MyRecursiveTask::execute` method. The goal of this exercise is to learn the basic elements of the task scheduler interface and complete the implementation of `MyRecursiveTask::execute` method.

2. Review the serial implementation of the recursive tree traversal algorithm (`original.h`).
3. Open `TODO.h` and find the implementation of `MyRecursiveTask::execute` method - this is a body of TBB task.
4. Complete the code that processes the "right" tree.

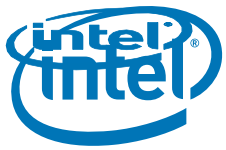
    There are a number of changes that need to be applied to the serial implementation to make it work with TBB, but they still look similar:

    a. The recursion will not change; it stops when the tree is empty.

    b. Because the task spawns new children, there is a variable "`count`" that contains the number of new children tasks.

    c. The new children tasks process "left" and "right" sub-trees. (This section is very similar to the section in the serial version.) There is a complete example how a new task is created to process the "left" sub-tree: `tbb::task::allocate_child` method is used to allocate the memory for the child task, added to the list of tasks, and the task counter is incremented.

© April 2008 Intel® Corporation



d. When all tasks are created and the task counter is equal to the number of children + 1, the method `tbb::task::spawn_and_wait_for_all` is called to spawn the tasks from the list.

5. Open main.cpp and find the function "`void improved()`". Its implementation demonstrates how a parallel tree traversal should be initialized:

   - It first creates root_task calling tbb::task::allocate_root method.
   - Then, root_task is spawned. This function returns when all children tasks of root_task are finished.

6. Build and run the application.

   Does the version using the TBB task scheduler interface perform better?

   _____

Note: The file "`solution.h`" contains the complete implementation for this exercise. If you would like to test it, just uncomment the line '`#include "solution.h"`' in main.cpp.

© April 2008 Intel® Corporation

Activity 4: Using the `concurrent_hash_map` Container

Time Required	Fifteen minutes
Objective	Modify a partially parallelized string counting application. The counts for strings are kept in the associative container mapping strings to the number of occurrences seen.

Modify, Build, and Run Threaded Application

1. Locate and change to the **04 concurrent_hash_map** directory.

   a. The `original.h` file contains an implementation of a function object for counting the occurrences of strings - class `CountStringsLocked`. Strings are stored in the array. The associative STL container map is used to map strings to integer counters: each individual parallel task executing `CountStringsLocked::operator()` will search for the string in the map and increment its counter.

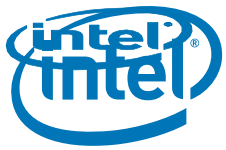
   Although this class uses the STL map method in a manner that permits many tasks to run in parallel, it is not thread-safe. A global lock (e.g. critical section) must be used to protect the map from concurrent access and modifications. Only one task can access the table at a time, and creates a performance bottleneck.

   However, multiple threads can safely search or modify the table concurrently if they access different parts of the data container. Intel® Threading Building Blocks (Intel® TBB) provides a container that is concurrency friendly -- `tbb::concurrent_hash_map` which uses local locks. If threads modify different parts of the container, they don't block competing tasks for the lock.

   b. The `TODO.h` file contains an implementation of `CountStringsNoLocks` class which counts strings occurrences using `tbb::concurrent_hash_map`. The goal of the exercise is to modify the body of `CountStringsNoLocks::operator()` with `tbb::concurrent_hash_map`, thus avoiding use of a global lock.

   c. Within the main.cpp source file, tbb::parallel_for is used to count the occurences of strings as parallelized task. Each parallel task (body of CountStrings*::operator()) is assigned an independent sub-range of the array Data. You can play with the problem size by changing the number of the strings in array. The Intel(R) TBB version of the algorithm doesn't use a global lock; all of the locks are local. It is expected to perform better than the version that uses global lock to protect concurrent modifications of STL map: all modifications of STL map are serial, while many modifications of tbb::oncurrent_hash_map are parallel.

© April 2008 Intel® Corporation



2. Open `TODO.h` and search for `MyHashCompare` structure. This structure is a required parameter to `tbb::concurrent_hash_map` template class. It defines hashing and comparison operations for user's type. The method "`equal`" will return true if the 2 keys are equal, and the method "`hash`" will generate the corresponding value for the key.
3. Modify the attributes of the CountStringsNoLocks class - `CRITICAL_SECTION` is not needed because `tbb::concurrent_hash_map` doesn't require global synchronization.
4. Modify the body of `CountStringsNoLocks::operator()`:
   - Remove calls to critical section API
   - Create the accessor object: "ConcurrentStringTable::accessor a;". The constructor of this object will acquire a local lock, and destructor will release this lock at the end of the code block
   - Use the method tbb::concurrent_hash_map::insert to access the table element key that is equal to the string from the array (*p)
   - Now "a" points to the table element of interest. Each table element is `std::pair` where `a->first` is a key and `a->second` is the corresponding value. Increment `a->second` to count this occurrence of the string *p.
5. Build and run the application.

   Does the version using the TBB container perform better? _____

Note: The file "`solution.h`" contains the complete implementation for this exercise. If you would like to test it, just uncomment the line '`#include "solution.h`"' in main.cpp.

© April 2008 Intel® Corporation

Activity 5: Using Scalable Memory Allocators

Time Required	Ten minutes
Objective	Modify a parallelized complete binary tree construction and traversal application to use scalable memory allocator within Intel TBB.

The application constructs a complete binary tree of given depth and traverses the tree. During the construction phase, half of the tree is done serially and the other half is done in parallel. After the full tree is constructed, a serial and a parallel traversal of the tree (summing up the values stored at each node) are run, one after the other. All four of these phases (serial construction, parallel construction, serial traversal, parallel traversal) are timed and the times are reported upon completion. The initial code uses the default "new" memory allocation method for each node in the construction phases.

Build and Run Serial Program

1. Locate and change to the **05 Scalable Allocator** directory. You should find Visual Studio Solution and project files and several source files in this directory.
2. Double-click on the solution icon and examine the main.cpp source file.
3. Be sure the Release configuration is selected and build the executable binary.
4. After successfully compiling and linking, run the executable. This can be done from the Visual Studio IDE by selecting the "Start without Debugging" command (CTRL+F5). The output reports a serial and a parallel construction time and a serial and a parallel traversal time.

   What is the serial construction time? _____

   What is the parallel construction time? _____

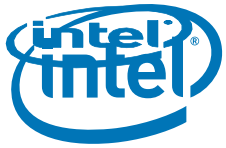
   What is the serial traversal time? _____

   What is the parallel traversal time? _____

Modify Program to use Intel TBB `scalable_allocator`

1. Modify the allocate_node method to allocate a new TreeNode object by using the scallable_allocator allocate() method.
2. Compile and debug the application. Once you have a clean compilation, run the parallel executable.

© April 2008 Intel® Corporation



What is the serial construction time? _____

What is the parallel construction time? _____

What is the serial traversal time? _____

What is the parallel traversal time? _____

What was the speedup of the new version? _____

© April 2008 Intel® Corporation