Tutorial Handout and Lab Instructions

Intel® Threading Building Blocks (Intel® TBB)

This tutorial aims to familiarize you with Intel Threading Building Blocks. Intel TBB is a versatile and rich library that provides task-parallelism with the spirit of C++.

Introduction

This tutorial aims to familiarize you with Intel Threading Building Blocks (TBB). Intel TBB is a versatile and rich library that provides task-parallelism with the spirit of C++.

Source code files are numbered according to the order of activities i.e., "`01`" (parallel for), "`02`", and so on. A file containing the suggested solution contains "`_solution`" in its name. Note, solutions may be distributed later during the tutorial, or they are given by the next exercise. Other useful resources are:
https://software.intel.com/en-us/intel-tbb
https://www.threadingbuildingblocks.org/
http://software.intel.com/en-us/intel-software-technical-documentation/
http://software.intel.com/en-us/forums/intel-threading-building-blocks/

The build system is based on GNU Make. To build (or run) an example type, `make <N>` where N is lab number. Also, you can use "`build.sh`" script that sets Intel TBB environment automatically.

Note: answers to questions are not included into given solutions. Take your own notes, and keep your own solutions as a reference!

Lab 1: Parallel For

Introduction

This activity performs general matrix-vector multiplication ("gemm" with α = 1, and β = 0).

Learn about loop parallelization and how λ-expressions (C++ 11) help to make the code more readable. Have a look at `01_parallel_for.cpp`!

Activity

1. Have a look at `01_parallel_for.cpp`. Ask questions to warm-up with C++ and the given code in general. Pick a candidate loop to parallelize the `gemm` function!
2. Make the Intel TBB environment available in your shell environment. For example, type:
   ```
   source /opt/intel/compilers_and_libraries/linux/tbb/bin/tbbvars.sh
   intel64
   ```
3. Learn the supplied `Makefile`. Type "`make`" without parameters to learn possible target values.
4. Build and run the sample:
   ```
   make 01
   ./01_parallel_for
   ```
5. Apply `parallel_for` (by using `blocked_range`) to the outer loop of `gemm`, and implement the loop body by using a lambda-expression.
6. Notice `tbb:: tick_count` calls used for time measurement.
7. Build and run the solution:
   ```
   make 01s
   ./01_parallel_for_solution
   ```

Note, there is an overloaded `parallel_for` with a signature that takes (*begin index*, *end index*, *body*) instead of taking a `blocked_range` object.

Bonus

I. Choose the lower scheduling overhead: (a) a single parallel loop that uses `blocked_range2d`, or (b) two nested one-dimensional `parallel_for` loops.
II. Introduce a preprocessor symbol `USE_GEMM_USE_PARALLEL_FOR`. In case this symbol is defined, use `parallel_for`, otherwise reuse your functor to execute in a serial fashion!
III. What is the grain size? Is "grain size" an argument of `parallel_for`?
IV. Read about the STL allocator (or about Intel TBB allocators), and use the `cache_aligned_allocator` for all buffers based on `std::vector`.
V. Introduce a `task_scheduler_init` object to ask for a specific number of threads. How to use all cores similar to the implicit/default initialization?

Lab 2: Reduction Operations

Introduction

Reductions are a class of collective operations that redistribute work during a chain of fork-join phases. Data locality might be still exploited by employing a scheme that does local work on a per-block basis. For reduction operations with low computational intensity, the whole process is often bound by the memory bandwidth.

In this activity, a linear buffer (1d) is reduced to a single value (0d). The parallel reduce function will be turned into a deterministic reduction that is able to show run-to-run reproducibility of the final result. What is the cost of determinism in terms of performance?

Activity

1. Have a look at implementation of `sum_reduce` (`02_reduction.cpp`). Adjust the functor of the reduction to accept a value in order to be more explicit about the initial state!
2. Build and run the sample:
   ```
   make 02
   ./02_reduction
   ```
3. Keep notes of the performance of the non-deterministic reduction for different problem sizes, and then employ TBB's deterministic reduction algorithm.
4. Rerun the previously recorded problem sizes, and compare the performance numbers. What is a “bathtub curve”? Fix the performance, and compare again!
5. Build and run the solution:
   ```
   make 02s
   ./02_reduction_solution
   ```

Bonus

I. Choose what likely gives higher performance: (a) `parallel_for` that uses a synchronization primitive inside the loop body, or (b) a `parallel_reduce`.

II. What level of determinism is covered by `parallel_deterministic_reduce`? Make your choice: (1) run to run reproducible result on the same computer with a non-varying number of threads, (2) reproducible results even for a varying number of threads, or (3) reproducible results regardless of the number of threads, and regardless of the particular processor type i.e., the particular SIMD instruction set.

III. Use the functional form of `parallel_deterministic_reduce` along with a λ-expression of the operation in order to perform the reduction without the need for a separate functor.

IV. Try the affinity partitioner, and check whether it improves the performance of the non-deterministic reduction. Would an improvement be visible for the first run of `sum_reduce`?

Lab 3: Concurrent Container

Introduction

Intel TBB includes several STL-alike containers that permit multiple threads to simultaneously invoke certain methods of the same container. The term "thread-safe" is meant to not only cover concurrent reads, but to also allow concurrent mutual operations. The motivation behind the concurrent containers is an additional piece of performance compared to cases where any mutual exclusive synchronization permits "concurrent" modification of the corresponding STL-container.

Familiarize with an example application which benefits from a concurrent container. First, make use of a mutual exclusive lock, and finally optimize the lock contention to achieve a higher scalability by just using an Intel TBB concurrent container. Note, that C++ 11 features have been used for this example application (λ-expressions and `std::unordered_map`).

Activity

1. Study the main program in `03_container.cpp`.
2. Build and run the sample. Why does the validation step eventually fails?
   ```
   make 03
   ./03_container
   ```
3. Lock the access to the container inside the body of the parallel loop. Use `scoped_lock` (a type that is nested into `spin_mutex`) to acquire and release the `lock` object.
4. Discover both types of `map_type`, and measure the time for at least 10M entries!
5. Introduce `concurrent_unordered_map` (`std::unordered_map` uses the same type arguments), and check that no lock is required in order to get correct results.
6. Build and run the solution:
   ```
   make 03s
   ./03_container_solution
   ```

Bonus

I. Use Intel® Inspector XE to detect the race condition in (a) `03_container.cpp` with no synchronization object, or in (b) `03_container_solution.cpp` with no atomic type.

II. Look into the Intel TBB reference manual, and find the table that compares the synchronization primitives. Is there any obviously better candidate than `spin_mutex`? Experiment, and measure the time!

Lab 4: Flow Graph

Introduction

The flow graph pattern can be used to model dependencies between tasks. Intel TBB automatically extracts the parallelism in presence of these dependencies. For example, this can be used to express concurrency over any kind of non-threaded functionality that needs to be executed according to a scheme. In contrast to other parallel programming models, information flow and dependencies are explicit as well as runtime-dynamic rather than implicitly controlled by program logic.

This activity aims to build up a more complex expression that requires multiple evaluations of `gemv` and `dot` functions.

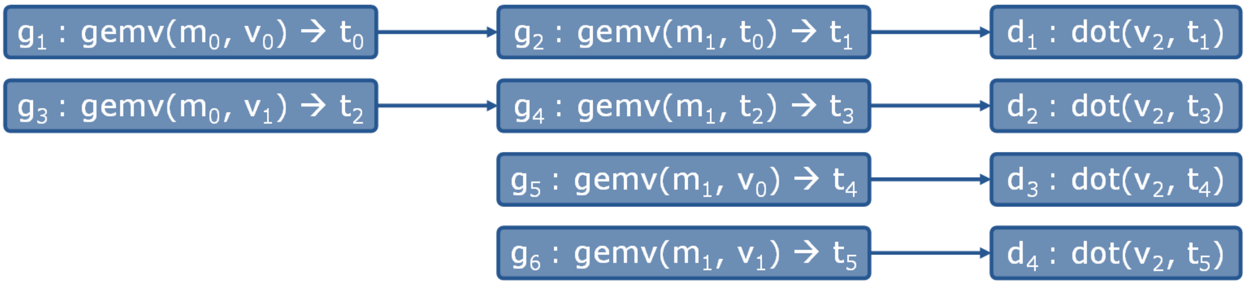

Figure 1: An expression is built of `gemv` and `dot` functions, and represented by `flow::graph`. Each node of the graph is given by `flow::function_node` objects (an action node i.e., `gemv` or `dot` are executed).

Here, `gemv` and `dot` do not employ multiple threads by themself e.g., no `parallel_for` is used. To start this activity, have a look at the driver program (`04_flow_graph.cpp`) where the flow graph description needs to be completed. This application will then show how much parallelism has been automatically extracted by Intel TBB.

Activity

1. Study the main program in `04_flow_graph.cpp`.
2. Build and run the sample:
   ```
   make 04
   ./04_flow_graph
   ```
1. Setup the `flow::function_node` objects which are missed from the expression as shown in Figure 1. The code can be placed right behind the first component of the graph made from the variables `g1`, `g2`, and `d1` (which represents the first line in Figure 1). When finished, seven node objects have been described in addition to the given three nodes.
2. Connect the graph nodes using `flow::make_edge`. When finished, six edges have been created according to the six arrows shown in Figure 1.
3. Build and run the solution:
   ```
   make 04s
   ./04_flow_graph_solution
   ```
4. Experiment with your solution (or `04_flow_graph_solution.cpp`), and try varying problem sizes (command line argument e.g., 512, 1024, 2048, 4096, and 8192). Take notes about the amount of parallelism (percentage) that has been exploited.

Note, that function nodes require single-argument actions, therefore `gemv` and `dot` are adapted by `gemv_body` and `dot_body`.

Bonus

I. Why is it possible that the parallel implementation executes reasonably faster than the shortest path of the serial implementation?
II. Create two `flow::broadcast_node` objects, and exploit that two function nodes are always fed by the same input (see `try_put`).
III. Remember the background about stateless vs. stateful functors (cf. Lab 1: Parallel For exercise), and have a look at the given code (graph actions) where data is updated but stored outside of the graph (referenced by pointers). Modify `gemv_body` to store the results of `gemv`!
IV. Find the note in the TBB reference documentation about how the body of an executable node is taken and stored. Is this done by reference/pointer, or by-value?

Lab 5: Tasks

Introduction

Intel TBB is a versatile library for parallel programming with e.g., parallel patterns, generic algorithms, tasks, and threads where each leverages its lower level. It is important to realize that "lower level" is not necessarily equivalent with "higher performance" e.g., task-based programming allows for unfair and lightweight scheduling of work.

In this activity, two ways are shown to organize work that is not data-parallel but intended to exploit compute resources as opposed to permanently run in the background, or as opposed to require fair time slices.

Activity

1. Have a look at `parallel_quicksort` (`04_task.cpp`) and how it invokes this algorithm for the left partition as well as the right partition. Is the scalability of `parallel_invoke` limited by just launching two functors?
2. Build and run the sample:
   ```
   make 05
   ./05_task
   ```
3. Sketch a parallel Quicksort based on `tbb::task`, or have a look into `05_task_solution.cpp`.
4. Build and run the solution:
   ```
   make 05s
   ./05_task_solution
   ```

Bonus

I. Think about why `wait_for_all` is called during the time the tree of tasks is built up.
II. What happens when the root task (`qsort`) is not created by every repetition that determines the execution time of the Quicksort algorithm?

III. Implement the Quicksort algorithm by using `tbb::task_group`.

Legal Information

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel.

Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Trademark Information

BlueMoon, BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino Inside, Cilk, Core Inside, E-GOLD, Flexpipe, i960, Intel, the Intel logo, Intel AppUp, Intel Atom, Intel Atom Inside, Intel CoFluent, Intel Core, Intel Inside, Intel Insider, the Intel Inside logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel Sponsors of Tomorrow., the Intel Sponsors of Tomorrow. logo, Intel StrataFlash, Intel vPro, Intel Xeon Phi, Intel XScale, InTru, the InTru logo, the InTru Inside logo, InTru soundmark, Itanium, Itanium Inside, MCS, MMX, Pentium, Pentium Inside, Puma, skoool, the skoool logo, SMARTi, Sound Mark, Stay With It, The Creators Project, The Journey Inside, Thunderbolt, Ultrabook, vPro Inside, VTune, Xeon, Xeon Inside, X-GOLD, XMM, X-PMU and XPOSYS are trademarks of Intel Corporation in the U.S. and/or other countries.

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

Technical Collateral Disclaimer

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.

A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

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. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm

Software Source Code Disclaimer

Any software source code reprinted in this document is furnished under a software license and may only be used or copied in accordance with the terms of that license.