Intel® TBB Lab

If you have any problems, don't hesitate to ask the lecturers.

Getting Started

Introduction

Source code files are numbered according to the order of activities i.e., "`00`" (Getting Started), "`01`", and so on. A file containing the solution is suggested by appending "`_solution`" to the base name. Note, solutions may be distributed later during the tutorial, or they are given by the next exercise.

Other useful resources are:
https://www.threadingbuildingblocks.org/
http://software.intel.com/en-us/node/467860

http://software.intel.com/en-us/intel-software-technical-documentation/
http://software.intel.com/en-us/forums/intel-threading-building-blocks/
http://software.intel.com/en-us/search/site/

The build system is based on GNU Make (via MinGW/msys on Windows). To build (or run) an example e.g., type:

```
make TGT=00_getting_started
make TGT=00_getting_started ARGS="1 5 1981" run
```

To run the program e.g., type:

```
bin/intel64/00_getting_started 1975
```

Please refer to the file `BUILD.txt` to learn more about the `Makefile`-based build system.

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

Activity

Learn how a main program that employs Intel TBB may look like. In the bonus part, learn about utilities such as a function to measure the execution time.

1. Have a look at `00_getting_started.cpp` and bcome familiar with it. Try to identify which might be a candidate loop within the gemm function to parallelize.
2. Compile the code as is by using the supplied `Makefile` . For example, type `make`.

Bonus

I. 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?

II. Use `tick_count::now()` to measure the duration of the calculation

Parallel For

Introduction

Although the library is not specifically tailored towards loop-nested HPC workloads, Intel TBB scales into data-parallel and compute-intensive domains. In fact, the task scheduler is unfair and is intended to schedule actual work rather than managing situations that are highly lock-contented.

Activity

This activity continues with the introduced general matrix-vector multiplication ("gemm" with $\alpha = 1$, and $\beta = 0$). Learn about loop parallelization and how λ-expressions (C++ 11) frequently help to make the code more readable. Have a look at `01_parallel_for.cpp`!

1. Apply `parallel_for` (by using `blocked_range`) to the outer loop of `gemm`, and implement the loop body by using a functor! Have a look at the functor below, and note its state in contrast to a function that is conceptually stateless.

```
struct functor {
  int i;
  explicit functor(int i_): i(i_) {}
  void operator()(int j) const {
    std::cout << "i + j = " << (i + j) << '\n'
  }
};
functor f(2507);
f(1975);
```

2. Use an `affinity_partitioner` as argument of your `parallel_for`. Why is this partitioner supplied in a non-const manner compared to the default `auto_partitioner`?

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. Choose what likely gives higher performance: (a) `parallel_for` that uses a synchronization primitive inside the loop body, or (b) a `parallel_reduce`.
III. 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!
IV. What is the grain size? Is "grain size" an argument of `parallel_for`?
V. Read about the STL allocator (or about Intel TBB allocators), and use the `cache_aligned_allocator` for all buffers based on `std::vector`.
VI. Introduce the preprocessor symbol `GEMM_USE_LAMBDA`. Write a λ-function that becomes the "body" of `parallel_for`.

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.
Note: prior to Intel TBB 4.1, the preprocessor symbol `TBB_PREVIEW_DETERMINISTIC_REDUCE` must have a non-zero value in order to make `parallel_deterministic_reduce` available.

Activity

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?

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

Bonus

I. 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.

II. 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`?

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 limits "concurrent" modification of the corresponding STL-container.

Activity

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`).

1. Study the main program in `03_container.cpp`, build, and run the application! Why does the validation step eventually fails?
2. Lock the access to the container inside the body of the parallel loop. Have a look at the *resource-allocation-is-initialization* idiom below, and use `scoped_lock` (a type that is nested into `spin_mutex`) to acquire and release the `lock` object.

```
{ // construct RAII
  raii_type raii(/*args*/);
  // ...
} // destruct RAII
```

3. Discover both types of `map_type`, and measure the time for at least 10M entries!
4. 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.

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.

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!

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. The flow graph feature got introduced by Intel TBB 4.0. Note: when looking for references, the flow graph is distinct from the "task graphs" as known before Intel TBB 4.0!

Activity

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

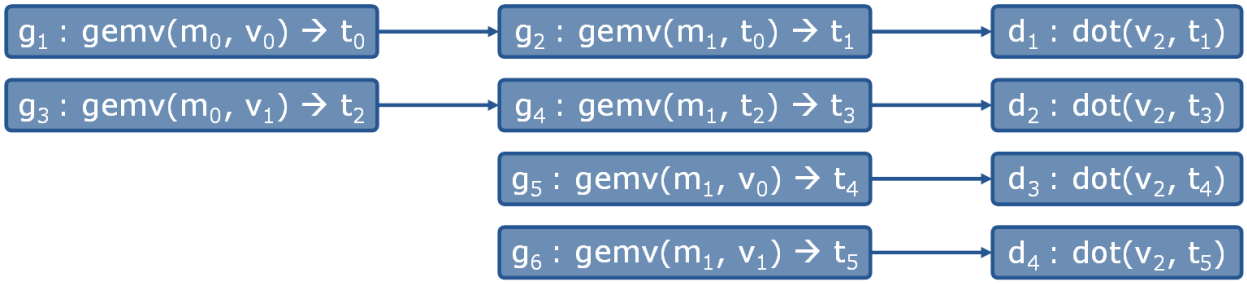

Figure 1: An expression is built of `gemv` and `dot` functions, and represented by `flow::graph`. Each node of the graph is given by `flow::function_node` objects (an action node i.e., `gemv` or `dot` are executed).

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.

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. 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. Why is it

possible that the parallel implementation executes reasonably faster than the shortest path of the serial implementation?

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

Bonus

I. Create two `flow::broadcast_node` objects, and exploit that two function nodes are always fed by the same input (see `try_put`).

Remember the background about stateless vs. stateful functors (cf.

II. 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`!

III. Find the note in the TBB reference documentation about how the body of a node is passed along. Is this done by reference/pointer, or by-value? (Hint: Body Objects)

Tasks

Introduction

Intel TBB is a versatile library for parallel programming with e.g., parallel patterns, generic algorithms, tasks, and threads where each benefits from a common lower level. It is important to realize that "lower level" is not necessarily equivalent with "higher performance". It always depends on the context where it used and the configuration.

Activity

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.

1. Have a look at `parallel_quicksort` (`05_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. Sketch a parallel Quicksort based on `tbb::task`, or have a look into `04_task_solution.cpp` and think about why `wait_for_all` is called during the time the tree of tasks is built up.
3. What happens when the root task (`qsort`) is not created by every repetition that determines the execution time of the Quicksort algorithm?

Bonus

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