Classical CPU tools training at CERN

# Performance Libraries
# Intel® oneAPI Threading Building Blocks
# Changes from previous versions

Aleksei Fedotov, Software Development Engineer at DSE PRE, Intel

# Agenda

- Revamp of the Intel® Threading Building Blocks (TBB)
- Motivation of the Revamp
- What is Intel® oneAPI Threading Building Blocks (oneTBB)?
- How it Differs from Intel® Threading Building Blocks (TBB)
- Migration from TBB to oneTBB
- Conclusion

# Revamp of Intel® Threading Building Blocks (TBB)

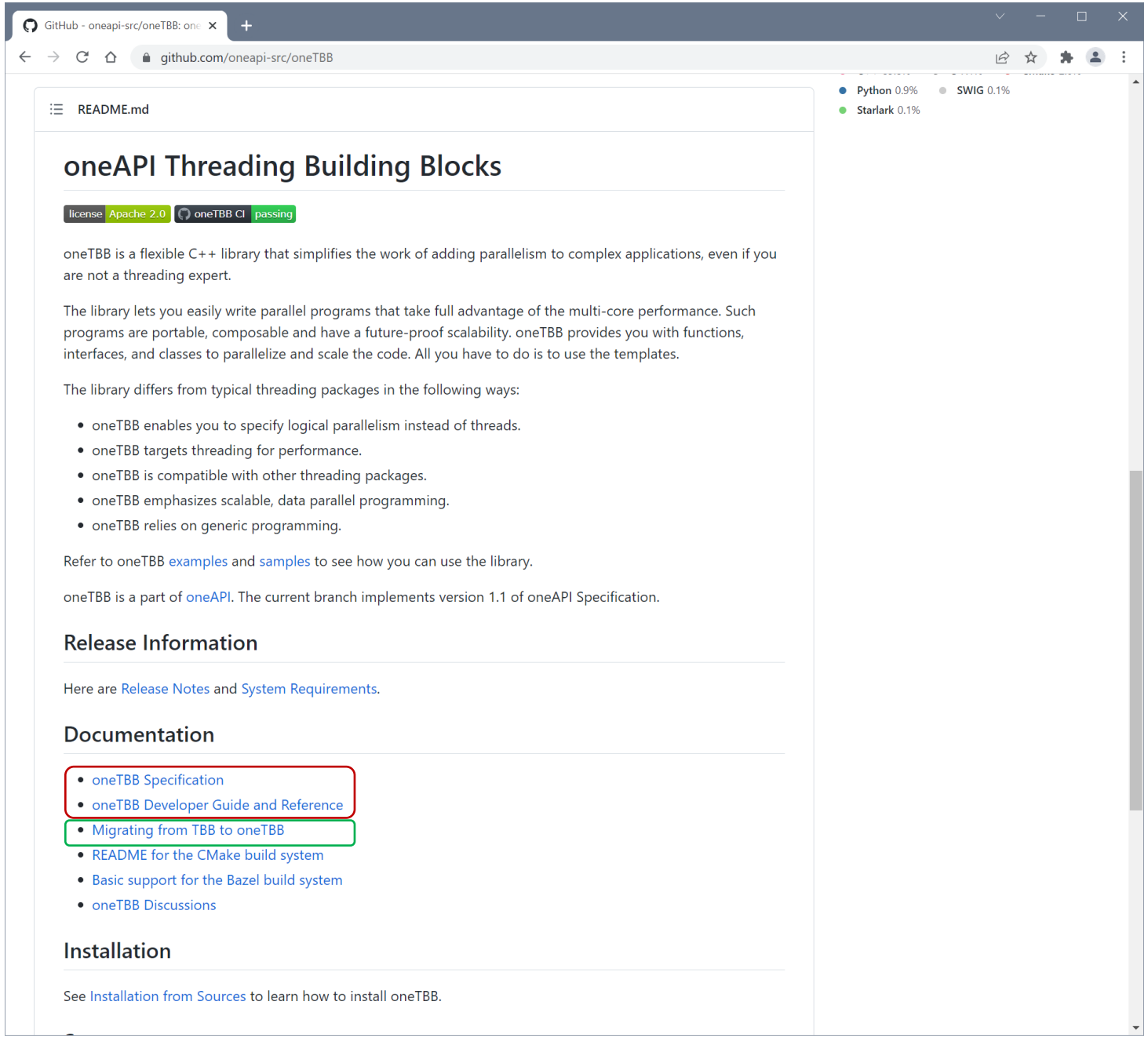
- The classic Intel® Threading Building Blocks (TBB) library had been revamped into Intel® oneAPI Threading Building Blocks (oneTBB)
- The latest version of the TBB library before the revamp is TBB 2020.3
- oneTBB is not binary compatible with TBB
- Among other things, name of the library had been changed to reflect its inclusion into Intel® oneAPI Toolkits
  - Its official name now is Intel® oneTBB Threading Building Blocks (oneTBB)
  - It can also be downloaded standalone

# Motivation of the Library Revamp

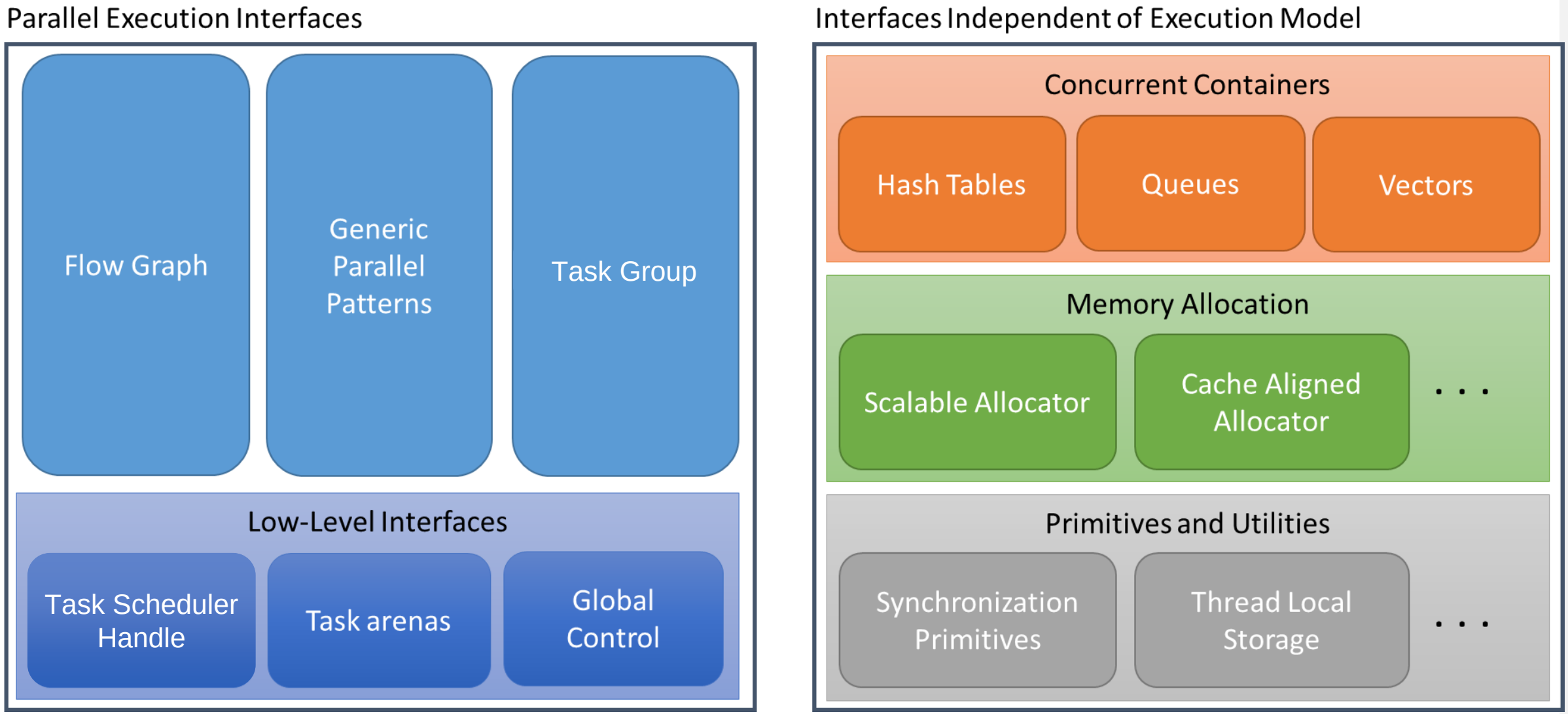
- Compliance with the latest C++ standards
- Improve the usability and simplicity of the library
- Deprecation and eventual removal of legacy TBB features

# Intel® oneAPI Threading Building Blocks (oneTBB)

- A portable C++ library for parallel programming
- Supports multiple platforms (some of them by community), e.g., x86, ARM, MIPS and others
- Distributed:
  - via open-source project at GitHub (https://github.com/oneapi-src/oneTBB)
  - as part of Intel oneAPI Base Toolkit
  - via package managers like apt and yum
  - via native channels like nuget.org and anaconda.org
- Qualified in accordance with ISO 26262 for Functional Safety and distributed to Mobileye


GitHub - oneapi-src/oneTBB: one
github.com/oneapi-src/oneTBB
README.md
Python 0.9%
SWIG 0.1%
Starlark 0.1%
oneAPI Threading Building Blocks
license Apache 2.0
oneTBB CI passing
oneTBB is a flexible C++ library that simplifies the work of adding parallelism to complex applications, even if you are not a threading expert.
The library lets you easily write parallel programs that take full advantage of the multi-core performance. Such programs are portable, composable and have a future-proof scalability. oneTBB provides you with functions, interfaces, and classes to parallelize and scale the code. All you have to do is to use the templates.
The library differs from typical threading packages in the following ways:
oneTBB enables you to specify logical parallelism instead of threads.
oneTBB targets threading for performance.
oneTBB is compatible with other threading packages.
oneTBB emphasizes scalable, data parallel programming.
oneTBB relies on generic programming.
Refer to oneTBB examples and samples to see how you can use the library.
oneTBB is a part of oneAPI. The current branch implements version 1.1 of oneAPI Specification.
Release Information
Here are Release Notes and System Requirements.
Documentation
oneTBB Specification
oneTBB Developer Guide and Reference
Migrating from TBB to oneTBB
README for the CMake build system
Basic support for the Bazel build system
oneTBB Discussions
Installation
See Installation from Sources to learn how to install oneTBB.

# Intel® oneAPI Threading Building Blocks (oneTBB)



# Functionality Removed in oneTBB (Pre C++11 Compatibility API)

| Deprecated/removed TBB functionality | Replacement |
|---|---|
| tbb::atomic | std::atomic |
| tbb::flow::tuple (incl. helper classes) | std::tuple |
| tbb::mutex, tbb::critical_section (incl. tbb::improper_lock) | std::mutex |
| tbb::recursive mutex | std::recursive mutex |
| tbb::hash (incl. tbb::hasher) | std::hash |
| tbb::tbb_thread / std::thread / std::this_thread | std::thread / std::this_thread with possible minimal changes related to std::chrono |
| std::lock_guard / std::unique_lock (incl. helper classes) | Minimal changes related to std::chrono might be required |
| std::condition_variable (incl. std::cv_status, std::timeout, std::no_timeout) | Minimal changes related to std::chrono might be required |
| tbb::aligned_space | std::aligned_storage |
| tbb::tbb_exception / tbb::captured_exception / tbb::movable_exception | No more needed due to TBB exact exception propagation |

# Compatibility with Microsoft* Parallel Patterns Library (PPL)

| Deprecated/removed TBB functionality | Replacement |
|---|---|
| Concurrency::critical section | std::mutex |
| Concurrency::reader_writer_lock (incl. Concurrency::improper_lock) | std::shared_mutex |
| Concurrency::parallel_invoke | tbb::parallel_invoke |
| Concurrency::parallel_for (first, last, f) | tbb::parallel_for (first, last, f) |
| Concurrency::parallel_for_each | tbb::parallel_for_each |
| Concurrency::task_group (incl. helper classes) | tbb::task_group |
| Concurrency::structured_task_group (incl. helper classes) | tbb::task_group |

# Other Functionality

| Deprecated/removed TBB functionality | Replacement |
|---|---|
| Task API (tbb::task, tbb::empty_task, tbb::task_list and related functions) | No direct replacement, the majority use cases can be covered with tbb::task_group, tbb::flow::graph. Task priorities can be covered with Flow graph node priorities and static arena-level priorities. |
| tbb::task_scheduler_init | tbb::task_arena, tbb::global_control, task_scheduler_handle |
| tbb::pipeline (incl. tbb::filter, tbb::thread_bound_filter) | tbb::parallel_pipeline, tbb::flow_async_node, resumable tasks |
| tbb::flow::sender/receiver/continue_receiver | Remain as unspecified base types for flow graph classes |
| Allocator template parameter for the flow graph nodes | No replacement is planned |
| tbb::flow::async_msg, tbb::flow::streaming_node, tbb::flow::opencl_node | No replacement is planned. To interact with asynchronous/heterogeneous activity use tbb::flow::async_node or resumable tasks |
| (preview) tbb::serial::parallel_for | Limit the number of threads to 1 with task_arena or global_control |
| (preview) runtime_loader (aka tbbproxy library) | No replacement is planned |
| tbb::structured_task_group (incl. helper classes) | tbb: :task_group |
| tbb::parallel_do | tbb::parallel_for_each |
| tbb::flow::source node | tbb::flow::input_node |
| tbb::reader_writer_lock | std::shared_mutex |

# Migration to oneTBB

- Dedicated page that helps in migrating
  - https://oneapi-src.github.io/oneTBB/main/tbb_userguide/Migration_Guide.html
  - Regularly populated with new and not yet covered use cases
  - Now includes explanations on how to:
    - Migrate from tbb::task_scheduler_init
    - Migrate from low-level tasking API
    - Mix to runtimes
- PDF document describing classic interfaces to replace with (https://www.intel.com/content/www/us/en/developer/articles/technical/tbb-revamp.html)

# Example 1: Spawning of Individual Tasks Using classic TBB Task API

```
#include <tbb/task.h>
int main() {
    // Assuming RootTask, ChildTask1, and ChildTask2 are defined.
    RootTask& root = *new(tbb::task::allocate_root()) RootTask{};
    ChildTask1& child1 = *new(root.allocate_child()) ChildTask1{/*params*/};
    ChildTask2& child2 = *new(root.allocate_child()) ChildTask2{/*params*/};
    root.set_ref_count(3);
    tbb::task::spawn(child1);
    tbb::task::spawn(child2);
    root.wait_for_all();
}
```

# Example 1: Spawning of Individual Tasks Using classic TBB Task API

```
#include <tbb/task.h>
int main() {
    // Assuming RootTask, ChildTask1, and ChildTask2 are defined.
    RootTask& root = *new(tbb::task::allocate_root()) RootTask{};
    ChildTask1& child1 = *new(root.allocate_child()) ChildTask1{/*params*/};
    ChildTask2& child2 = *new(root.allocate_child()) ChildTask2{/*params*/};
    root.set_ref_count(3);
    tbb::task::spawn(child1);
    tbb::task::spawn(child2);
    root.wait_for_all();
}
```

# Example 1: Spawning of Individual Tasks Using oneapi::tbb::task_group

```
#include <oneapi/tbb/task_group.h>


int main() {
    // Assuming ChildTask1, and ChildTask2 are defined.
    oneapi::tbb::task_group tg;
    tg.run(ChildTask1{/*params*/});
    tg.run(ChildTask2{/*params*/});
    tg.wait();
}
```

# Example 1: Spawning of Individual Tasks Using oneapi::tbb::parallel_invoke

```
#include <oneapi/tbb/parallel_invoke.h>


int main() {
    // Assuming ChildTask1, and ChildTask2 are defined.
    oneapi::tbb::parallel_invoke(
        ChildTask1{/*params*/},
        ChildTask2{/*params*/}
    );
}
```

# Example 2: Adding More Work During Task Execution in Classic TBB

```
#include <tbb/task.h>
// Assuming necessary entities are defined and implement required interfaces
struct Task : public tbb::task {
    Task(tbb::task& root, int i) : m_root(root), m_i(i) {}
    tbb::task* execute() override {
        // ... do some work for item m_i ...
        if (add_more_parallel_work) {
            tbb::task& child = *new(m_root.allocate_child()) OtherWork;
            tbb::task::spawn(child);
        }
        return nullptr;
    }
    // …
};
```

# Example 2: Adding More Work During Task Execution in Classic TBB

```
#include <tbb/task.h>
// Assuming necessary entities are defined and implement required interfaces
struct Task : public tbb::task {
    Task(tbb::task& root, int i) : m_root(root), m_i(i) {}
    tbb::task* execute() override {
        // ... do some work for item m_i ...
        if (add_more_parallel_work) {
            tbb::task& child = *new(m_root.allocate_child()) OtherWork;
            tbb::task::spawn(child);
        }
        return nullptr;
    }
    // …
};
```

# Example 2: Adding More Work During Task Execution Using oneapi::tbb::parallel_for_each

```
#include <vector>
#include <oneapi/tbb/parallel_for_each.h>

int main() {
    std::vector<int> items = { 0, 1, 2, 3, 4, 5, 6, 7 };
    oneapi::tbb::parallel_for_each(
        items.begin(), items.end(),
        [](int& i, tbb::feeder<int>& feeder) {
            // ... do some work for item i ...
            if (add_more_parallel_work)
                feeder.add(i);
        }
    );
}
```

# Example 2: Adding More Work During Task Execution Using oneapi::tbb::task_group

```
#include <vector>
#include <oneapi/tbb/task_group.h>
int main() {
    std::vector<int> items = { 0, 1, 2, 3, 4, 5, 6, 7 };
    oneapi::tbb::task_group tg;
    for (std::size_t i = 0; i < items.size(); ++i) {
        tg.run([&i = items[i], &tg] {
            // ... do some work for item i ...
            if (add_more_parallel_work)
                tg.run(OtherWork{});
        });
    }
    tg.wait();
}
```

# Example 3: Deferred Task Creation Using Classic TBB

```
#include <tbb/task.h>
int main() {
    // Assuming RootTask, ChildTask, and CallBackTask are defined.
    RootTask& root = *new(tbb::task::allocate_root()) RootTask{};
    ChildTask&    child    = *new(root.allocate_child()) ChildTask{/*params*/};
    CallBackTask& cb_task  = *new(root.allocate_child()) CallBackTask{/*params*/};
    root.set_ref_count(3);
    tbb::task::spawn(child);
    register_callback( [cb_task&]() { tbb::task::enqueue(cb_task); } );
    root.wait_for_all();
    // Control flow will reach here only after both ChildTask and CallBackTask are
    // executed, i.e. after the callback is called
}
```

# Example 3: Deferred Task Creation Using Classic TBB

```
#include <tbb/task.h>
int main() {
    // Assuming RootTask, ChildTask, and CallBackTask are defined.
    RootTask& root = *new(tbb::task::allocate_root()) RootTask{};
    ChildTask&    child    = *new(root.allocate_child()) ChildTask{/*params*/};
    CallBackTask& cb_task  = *new(root.allocate_child()) CallBackTask{/*params*/};
    root.set_ref_count(3);
    tbb::task::spawn(child);
    register_callback( [cb_task&]() { tbb::task::enqueue(cb_task); } );
    root.wait_for_all();
    // Control flow will reach here only after both ChildTask and CallBackTask are
    // executed, i.e. after the callback is called
}
```

# Example 3: Deferred Task Creation Using oneapi::tbb::task_group

```
#include <oneapi/tbb/task_group.h>
int main(){
    oneapi::tbb::task_group tg;
    oneapi::tbb::task_arena arena;
    // Assuming ChildTask and CallBackTask are defined.

    auto cb = tg.defer(CallBackTask{/*params*/});
    register_callback( [&tg, c = std::move(cb), &arena]{ arena.enqueue(c); } );

    tg.run(ChildTask{/*params*/});
    tg.wait();
    // Control flow gets here once both ChildTask and CallBackTask are executed
    // i.e. after the callback is called
}
```

# Example 3: Deferred Task Creation Using oneapi::tbb::task_group (alternative)

```
#include <oneapi/tbb/task_group.h>
int main(){
    oneapi::tbb::task_group tg;
    // Assuming ChildTask and CallBackTask are defined.

    auto cb = tg.defer(CallBackTask{/*params*/});
    register_callback( [&tg, c = std::move(cb)]{ tbb::this_task_arena::enqueue(c); } );

    tg.run(ChildTask{/*params*/});
    tg.wait();
    // Control flow gets here once both ChildTask and CallBackTask are executed
    // i.e. after the callback is called
}
```

# Other Changes: New Community Preview Features

- Task arena interface extension to support Hybrid CPUs
- Collaborative call once
- Extended the high-level task API to simplify migration from TBB to oneTBB
- Task scheduler handle to support waiting of worker thread termination
- Added heterogeneous lookup, erase, insert in concurrent hash map
- etc.

# Other Changes: New Features

- Defer execution of a task in the task group
- Specify arbitrary task_group_context for the task_group
- Task arena interface extension to specify priority of the arena
- Support of latest C++ standards and compilers
- Support of Address Sanitizer and Thread Sanitizer
- Several community preview features are now officially supported
  - Concurrent ordered containers
  - Task arena interface extension for NUMA
  - Flow Graph API to support relative priorities for functional nodes
  - Resumable tasks

# Conclusion

- oneTBB is a revamped version of TBB
  - Included into Intel® oneAPI Toolkits
  - Migration Guide (https://oneapi-src.github.io/oneTBB/main/tbb_userguide/Migration_Guide.html)
  - oneTBB team can help in migration!
- Available on GitHub (https://github.com/oneapi-src/oneTBB)
  - Anyone can open an Issue, suggest a Pull Request
- Includes CMake support and is included into popular package managers
- Backward compatible with previous releases of oneTBB


intel