Denis Blank - GSoC 2017

STEllAR-GROUP/hpx

Project Proposal

Please provide a description of your proposed work.

Proposal for re-implementing `hpx::util::unwrapped` and unifying the API of `hpx::wait_(all|any|each|some)` and `hpx::when_(all|any|each|some)`

Overview My plans for a potential GSoC stipend embrace the plans on improving the following functions:

- `hpx::util::unwrapped`
- `hpx::wait_(all|any|each|some)`
- `hpx::when_(all|any|each|some)`
- `hpx::dataflow`

in order to fully support the requirements for accepting argument types described in #2456, #1404, #1400 and #1126. When the proposal is finished, all the functions listed above should accept the same set of arguments (unification).

The features added as part of a potential GSoC project are described below and will include the corresponding implementation, unit-tests as well as Doxygen documentation comments.

(1) Non future types For easier usage, especially when doing template meta-programming, where we don't want to handle special cases, the functions listed above should treat non future types as ready futures so we can elide an explicit conversion into a `hpx::future` (for instance `f(0)` instead of `f(hpx::make_ready_future(0))`). This behavior is requested through issue #1400.

(2) Accepting arbitrary homogeneous container Currently `std::vector` is used as a fixed type when passing a single *homogeneous* container to `wait_*` or `when_*`. I propose to improve the API of these functions in order to take arbitrary objects satisfying the container (`begin()` and `end()`) requirements instead of a `std::vector`.

This could also remove additional overloads for special types such as `std::array` which supports `begin()` and `end()` by design.

```
std::list<hpx::future<int>> my_list;
auto all = hpx::when_all(my_list);
```

(3) Accepting arbitrary heterogeneous container In addition of supporting *homogeneous* container, support for *heterogeneous* container types such as `std::tuple` or `std::pair` is planned, when passed as a single argument to `wait_*` or `when_*`. A *heterogeneous* container must be unpack able through a call to `hpx::get<I>`.

```
std::tuple<hpx::future<int>, hpx::future<float>> my_tuple;
hpx::wait_all(my_tuple);
```

This becomes important when covering the improvement of composed sequences in proposal *(4)*.

(4) Nested `hpx::future<...>` types and composed sequences Issue #1126 describes that the functions listed above don't work with sequences and nested futures inside containers. This point is by far the most time-consuming one.

As solution I would propose to add exactly 2 overloads to each `wait_*` and `when_*` function:

```
// [1]
template<typename... Args>
auto wait_all(Args&&... args) { /* ... */ }

// [2]
template<typename Begin, typename End, typename std::enable_if<
    // Test whether `begin` and `end` are future iterators
>::type* = nullptr>
auto wait_all(Begin begin, End end) { /* ... */ }

// Note that we have to convert the `auto` into `decltype(FUNCTION_BODY)` for C++11.
```

Function overload *[2]* just treats 2 iterators as arguments special, this behavior isn't different from the current code base.

Function *[1]* however, will work completely different because we can treat every single argument in the pack as one of a:

- Non future type: `int`
- Single future object: `hpx::future<int>`
- Homogeneous container holding futures: `std::deque<hpx::future<int>>`
- Heterogeneous container holding futures: `std::tuple<hpx::future<int>>`

This change doesn't deprecate the current API but additionally allows us to pass various arguments to the functions like requested in issue #1126 and shown below:

```
int my_value = 0;
hpx::future<int> my_future;
std::list<hpx::future<int>> my_list;
std::tuple<hpx::future<int>, hpx::future<float>> my_tuple;

hpx::wait_all(my_value, my_future, my_list, my_tuple);
```

This should make it possible to pass an `hpx::future<...>` which is nested with 1 level depth to the API. A deeper level of nestings such as `std::vector<std::vector<hpx::future<int>>>` is feasible, but we need to evaluate first, whether we want to support unlimited nesting depths or limit the nesting we can access to a certain depth.

(5) Unwrapping facilities for nested and composed future sequences As of now, the 8 headers implementing various `wait_*` and `when_*` functions contain a lot of duplicated code for the wait functionality.

I believe that the duplication can be significantly removed through implementing two internal facilities for:

- Accessing nested `hpx::futures` inside parameter packs for:
  - unwrapping (usable for `hpx::util::unwrapped`)
  - access (for registering wait handler)
- Waiting on a fixed amount of futures (proposed by issue #1132).

By design `wait_*` functions return `void` whereas `when_*` functions mostly return a persistent copy of its arguments. Theoretically, we could use the same continuation handlers across both function types, the only difference is that `wait_*` waits on the current thread while `when_*` makes the future, which is returned, ready.

The facility for unwrapping a composed sequence of potential nested `hpx::futures` could also be used to re-implement `hpx::util::unwrapped`.
After implementing the proposed changes `hpx::util::unwrapped` should be additionally able to unwrap nested types described in **(2)**, **(3)** and **(4)**:

- Futures inside arbitrary homogeneous containers:
  `std::list<hpx::future<int>>` -> `std::list<int>`:

```
hpx::util::unwrapped([](std::list<int> my_list) {
  // ...
})(std::list<hpx::future<int>>{});
```

- Futures inside arbitrary heterogeneous containers:
  `std::pair<hpx::future<int>, hpx::future<float>>` -> `int, float`.

```
hpx::util::unwrapped([](int, float) {
  // ...
})(std::pair<hpx::future<int>, hpx::future<float>>{});
```

The purpose of this changes is that we should be able to unwrap all `hpx::future` types we received from `hpx::when_*` functions, which is currently not the case as indicated by issue #1126.

(6) Move-only types Issue #1404 describes that `hpx::util::unwrapped` doesn't work with move-only types. As part of the internal API improvement, I will resolve this Issue.

(7) Improving the documentation Additionally, I plan to improve the documentation of the functions listed above. The `hpx::util::unwrapped` and `hpx::dataflow` function isn't documented currently and thus leads to a lot of confusion on how to use it.

Benefits of this proposal

In my opinion the proposed changes are highly beneficial for the HPX project, because it could increase the expressiveness of the `hpx::wait_(all|any|each|some)` and `hpx::when_(all|any|each|some)` API, while decreasing the workarounds needed, to express a certain data flow or dependency.

When looking at pure numbers, the proposal would resolve **five** out of the current 112 present issues in the HPX issue tracker on GitHub, which seems like a great profitable outcome for a GSoC stipend.

Improving the `hpx::wait_(all|any|each|some)` and `hpx::when_(all|any|each|some)` functions, which belong to the most important API of HPX, will increase the advantages of using HPX over alternative libraries such as the cpprestsdk or the C++17 standard library, that will support similar capabilities in later standards and thus will become a notable competitor when targeting CPU parallelism in the future.

May 4, 2017