ASYNCHRONOUS COMPUTING IN C++

Hartmut Kaiser  (Hartmut.Kaiser@gmail.com)

CppCon 2014
WHAT IS ASYNCHRONOUS COMPUTING?

- Spawning off some work without immediately waiting for the work to finish
  - Asynchronous work
  - May produce result (some value) or not (just trigger)
- Either: wait for the asynchronous work at some later point
- Or: attach a continuation which is automatically run once the work is done
- While this sounds like parallelism, it is not directly related, however
  - May be used to (auto-) parallelize code (this talk will show an example)
  - Runs just as well in single threaded environments
  - Runs just as well in environments with an arbitrary number (millions) of threads
WHAT IS ASYNCHRONOUS COMPUTING?

- Also called ‘reactive computing’, ‘actor computing’, or ‘observer pattern’
  - Propagation of change using the concepts of (static and dynamic) dataflow
- There are many existing asynchronous environments
  - JavaScript, C#, widely adopted in functional languages
  - In this talk, the term ‘asynchronous computing’ is used
    - Presented concepts are not ‘strictly’ reactive
    - Attempt to integrate dataflow with ‘normal’ imperative C++
- All content of this talk is based on using such an environment: HPX
  - HPX is a general purpose parallel runtime system for applications of any scale
WHY ASYNCHRONOUS COMPUTING?

Tianhe-2's projected theoretical peak performance: 54.9 PetaFLOPs
16,000 nodes, ~3,200,000 computing cores (32,000 Intel Ivy Bridge Xeons, 48,000 Xeon Phi Accelerators)
ASYNCHRONOUS ENVIRONMENTS

• Asynchronous computing requires an appropriate runtime system which supports scheduling of work
  • All existing asynchronous environments have such a runtime system

• C++ has the standard library
  • Surprisingly the existing concepts are suitable for this (with some extensions)
  • Main facility is the type ‘future<T>’

• Default implementations of ‘future<T>’ are based on kernel threads
  • Too coarse grain, too much overhead
WHY IS STD::THREAD TOO SLOW?

Execution Time over Grain Size
(for different amounts of overheads per thread)

- Sequential Time
- 1µs Overhead
- 100µs Overhead
- 10ms Overhead

Execution time (relative to sequential time)
Grain Size (amount of work per thread)
ASYNCHRONOUS ENVIRONMENTS

• Even relatively small amounts of work can benefit from being split into smaller tasks
  • Possibly huge amount of ‘threads’
    • In the previous gedankenexperiment we ended up considering up to 10 million threads
    • Best possible scaling is predicted to be reached when using 10000 threads (for 10s worth of work)

• Several problems
  • Impossible to work with that many kernel threads (p-threads)
  • Impossible to reason about this amount of tasks
  • Requires abstraction mechanism
CURRENT STD::FUTURE
WHAT IS A (THE) FUTURE

- A future is an object representing a result which has not been calculated yet
  - Enables transparent synchronization with producer
  - Hides notion of dealing with threads
  - Makes asynchrony manageable
  - Allows for composition of several asynchronous operations
  - (Turns concurrency into parallelism)

Hartmut Kaiser: Asynchronous Computing in C++
Many ways to get hold of a future, simplest way is to use (std) async:

```c++
int universal_answer() { return 42; }

void deep_thought()
{
    future<int> promised_answer = async(&universal_answer);

    // do other things for 7.5 million years

    cout << promised_answer.get() << endl;  // prints 42, eventually
}
```
WAYS TO CREATE A FUTURE

• Standard defines 3 possible ways to create a future,
  • 3 different ‘asynchronous providers’
    • std::async
      • See previous example, std::async has caveats
    • std::packaged_task
    • std::promise
PACKAGING A FUTURE

• std::packaged_task is a function object
  • It gives away a future representing the result of its invocation

• Can be used as a synchronization primitive
  • Pass to std::thread

• Converting a callback into a future
  • Observer pattern, allows to wait for a callback to happen
template <typename F, typename ...Arg>
std::future<typename std::result_of<F(Arg...)>::type>
simple_async(F func, Arg&& arg...)
{
    std::packaged_task<F> pt(func);
    auto f = pt.get_future();

    std::thread t(std::move(pt), std::forward<Arg>(arg)...);
    t.detach();

    return std::move(f);
}
PROMISING A FUTURE

• std::promise is also an *asynchronous provider* ("an object that provides a result to a shared state")
  • The promise is the thing that you *set* a result on, so that you can *get* it from the associated future.
  • The promise initially creates the shared state
  • The future created by the promise shares the state with it
  • The shared state stores the value
template <typename F> class simple_packaged_task;

template <typename R, typename ...Args>
class simple_packaged_task<R(Args...)>
    // must be move-only
{
    std::function<R(Args...)>
        fn;
    std::promise<R>
        p;                    // the promise for the result
    // ...
public:
    template <typename F> explicit simple_packaged_task(F && f)
        : fn(std::forward<F>(f)) {}

    template <typename ...T>
    void operator()(T &&... t) { p.set_value(fn(std::forward<T>(t)...)); }

    std::future<R> get_future() { return p.get_future(); }

};
EXTENDING STD::FUTURE
EXTENDING STD::FUTURE

• Several proposals (draft technical specifications) for next C++ Standard
  • Extension for future<>  
    • Compositional facilities  
      • Parallel composition  
      • Sequential composition  
  • Parallel Algorithms  
  • Parallel Task Regions  
• Extended async semantics: dataflow
MAKE A READY FUTURE

• Create a future which is ready at construction (N3857)

```cpp
future<int> compute(int x)
{
    if (x < 0) return make_ready_future<int>(-1);
    if (x == 0) return make_ready_future<int>(0);

    return async([](int par) { return do_work(par); }, x);
}
```
COMPOSITIONAL FACILITIES

• Sequential composition of futures (see N3857)

```cpp
string make_string()
{
    future<int> f1 = async([]() -> int { return 123; });
    future<string> f2 = f1.then(
        [](future<int> f) -> string {
            return to_string(f.get()); // here .get() won't block
        });
}
```
COMPOSITIONAL FACILITIES

- Parallel composition of futures (see N3857)

```c++
void test_when_all() {
    shared_future<int> shared_future1 = async([]() { return 125; });
    future<string> future2 = async([]() { return string("hi"); });

    future<
        tuple<
            shared_future<int>,
            future<string>
        >
    > all_f =
        when_all(shared_future1, future2);  // also: when_any, when_some, etc.

    future<int> result = all_f.then(
        [](future<
            tuple<
                shared_future<int>,
                future<string>
            > f) { return do_work(f.get()); }
    );
}
```
PARALLEL ALGORITHMS

- Parallel algorithms (N4071)
  - Mostly, same semantics as sequential algorithms
  - Additional, first argument: execution_policy (seq, par, etc.)
- Extension
  - task_execution_policy
  - Algorithm returns future<>

<table>
<thead>
<tr>
<th>adjacent_difference</th>
<th>adjacent_find</th>
<th>all_of</th>
<th>any_of</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy</td>
<td>copy_if</td>
<td>copy_n</td>
<td>count</td>
</tr>
<tr>
<td>count_if</td>
<td>equal</td>
<td>exclusive_scan</td>
<td>fill</td>
</tr>
<tr>
<td>fill_n</td>
<td>find</td>
<td>find_end</td>
<td>find_first_of</td>
</tr>
<tr>
<td>find_if</td>
<td>find_if_not</td>
<td>for_each</td>
<td>for_each_n</td>
</tr>
<tr>
<td>generate</td>
<td>generate_n</td>
<td>includes</td>
<td>inclusive_scan</td>
</tr>
<tr>
<td>in-place_merge</td>
<td>in-place_merge</td>
<td>is_heap</td>
<td>is_heap_until</td>
</tr>
<tr>
<td>is_parted</td>
<td>is_sorted</td>
<td>is_sorted_until</td>
<td>lexicographical_compare</td>
</tr>
<tr>
<td>max_element</td>
<td>merge</td>
<td>min_element</td>
<td>minmax_element</td>
</tr>
<tr>
<td>mismatch</td>
<td>move</td>
<td>none_of</td>
<td>nth_element</td>
</tr>
<tr>
<td>partial_sort</td>
<td>partial_sort_copy</td>
<td>partition</td>
<td>partition_copy</td>
</tr>
<tr>
<td>reduce</td>
<td>remove</td>
<td>remove_copy</td>
<td>remove_copy_if</td>
</tr>
<tr>
<td>remove_if</td>
<td>replace</td>
<td>replace_copy</td>
<td>replace_copy_if</td>
</tr>
<tr>
<td>replace_if</td>
<td>reverse</td>
<td>reverse_copy</td>
<td>rotate</td>
</tr>
<tr>
<td>rotate_copy</td>
<td>search</td>
<td>search_n</td>
<td>set_difference</td>
</tr>
<tr>
<td>set_intersection</td>
<td>set_symmetric_difference</td>
<td>set_union</td>
<td>sort</td>
</tr>
<tr>
<td>stable_partition</td>
<td>stable_sort</td>
<td>swap_ranges</td>
<td>transform</td>
</tr>
<tr>
<td>uninitialized_copy</td>
<td>uninitialized_copy_n</td>
<td>uninitialized_fill</td>
<td>uninitialized_fill_n</td>
</tr>
</tbody>
</table>

Hartmut Kaiser: Asynchronous Computing in C++  September 9th, 2014 22
HPX – A GENERAL PURPOSE RUNTIME SYSTEM

• Solidly based on a theoretical foundation - ParalleX
  • A general purpose parallel runtime system for applications of any scale
    • http://stellar-group.org/libraries/hpx
    • https://github.com/STEllAR-GROUP/hpx/

• Exposes an uniform, standards-oriented API for ease of programming parallel and distributed applications.
  • Enables to write fully asynchronous code using hundreds of millions of threads.
  • Provides unified syntax and semantics for local and remote operations.

• Enables writing applications which out-perform and out-scale existing ones

• Is published under Boost license and has an open, active, and thriving developer community.

• Can be used as a platform for research and experimentation
THREAD OVERHEADS

Threads (Tasks) Executed per Second

- HPX
- Qthreads
- TBB

Number of Cores

Threads (Tasks) Executed per Second

Millions

0 5 10 15 20 25 30

0 5 10 15 20 25

Hartmut Kaiser: Asynchronous Computing in C++
HPX – THE API

- As close as possible to C++11/14 standard library, where appropriate, for instance
  - `std::thread` `hpx::thread`
  - `std::mutex` `hpx::mutex`
  - `std::future` `hpx::future` (including N3857)
  - `std::async` `hpx::async` (including N3632)
  - `std::bind` `hpx::bind`
  - `std::function` `hpx::function`
  - `std::tuple` `hpx::tuple`
  - `std::any` `hpx::any` (N3508)
  - `std::cout` `hpx::cout`
  - `std::parallel::for_each`, etc. `hpx::parallel::for_each` (N4071)
  - `std::parallel::task_region` `hpx::parallel::task_region` (N4088)
What if one or more arguments to ‘async’ are futures themselves?

Normal behavior: pass futures through to function

Extended behavior: wait for futures to become ready before invoking the function:

```
template <typename F, typename... Arg>
future<typename result_of<F(Args...)>::type> dataflow(F&& f, Arg&&... arg);
```

- If ArgN is a future, then the invocation of F will be delayed
- Non-future arguments are passed through
TWO EXAMPLES
EXTENDING PARALLEL ALGORITHMS
EXTENDING PARALLEL ALGORITHMS

• New algorithm: gather

```cpp
template <typename BiIter, typename Pred>
pair<BiIter, BiIter> gather(BiIter f, BiIter l, BiIter p, Pred pred)
{
    return make_pair(stable_partition(f, p, not1(pred)), stable_partition(p, l, pred));
}
```
New algorithm: `gather_async`

```cpp
template <typename BiIter, typename Pred>
future<pair<BiIter, BiIter>> gather_async(BiIter f, BiIter l, BiIter p, Pred pred)
{
    return dataflow(
        unwrapped([](BiIter r1, BiIter r2) { return make_pair(r1, r2); }),
        parallel::stable_partition(task, f, p, not1(pred)),
        parallel::stable_partition(task, p, l, pred));
}
```
• Iteratively simulating 1D heat diffusion
1D HEAT EQUATION

- Kernel: simple iterative heat diffusion solver, 3 point stencil

```cpp
double heat(double left, double middle, double right)
{
    return middle + (k*dt/dx*dx) * (left - 2*middle + right);
}
```
1D HEAT EQUATION

• One time step, periodic boundary conditions:

```cpp
void heat_timestep(std::vector<double>& next, std::vector<double> const& curr)
{
    #pragma omp parallel for
    for (std::size_t i = 0; i != nx; ++i)
        next[i] = heat(current[idx(i-1, nx)], current[i], current[idx(i+1, nx)]);
}
```
1D HEAT EQUATION

• Time step iteration:

```cpp
std::array<std::vector<double>, 2> U = { std::vector<double>(nx), std::vector<double>(nx) };
for (std::size_t t = 0; t != nt; ++t)
{
    std::vector<double> const & current = U[t % 2];
    std::vector<double> & next = U[(t + 1) % 2];

    heat_timestep(next, curr);
}
```
One time step, periodic boundary conditions:

```cpp
void heat_timestep(
    std::vector<shared_future<double>>& next,
    std::vector<shared_future<double>> const& curr)
{
    for (std::size_t i = 1; i != nx-1; ++i) {
        next[i] = dataflow(unwrapped(heat),
            current[idx(i-1, nx)], current[i], current[idx(i+1, nx)]);
    }
}
```
1D HEAT EQUATION, PARTITIONED

- Partitioning data into parts to control grain size of work
1D HEAT EQUATION, FUTURIZED

• Time step iteration:

```cpp
std::array<std::vector<std::vector<std::shared_future<std::vector<double>>>>, 2> U { ... };
for (std::size_t t = 0; t != nt; ++t)
{
    std::vector<std::shared_future<std::vector<double>>> const& current = U[t % 2];
    std::vector<std::shared_future<std::vector<double>>>& next = U[(t + 1) % 2];

    heat_timestep(next, curr);
}
```
• One time step, periodic boundary conditions:

```cpp
void heat_timestep(
    std::vector<shared_future<std::vector<double>>> & next,
    std::vector<shared_future<std::vector<double>>> const & curr)
{
    for (std::size_t i = 0; i != np; ++i) {
        next[i] = dataflow(unwrapped(heat_partition),
            current[idx(i-1, np)], current[i], current[idx(i+1, np)]);
    }
}
```
1D HEAT EQUATION, RESULTS

Execution Time over Grainsize

- - - - Sequential Time
--- Time (16 Cores)

1D Heat Equation, Strong Scaling SMP
(100,000,000 Elements, Stampede)

- - - - Peak
- - - OpenMP
--- HPX (part)
• Asynchronous computing is fun
  • And a possible approach to solve massive parallelization problems
• C++11/14 (and proposals) cover large amount of necessary interfaces
  • However more fine grain parallelism necessary to take full advantage
• One possible option would be to use HPX as a runtime environment
  • HPX also implements a couple of extensions which have proven to be beneficial