Asynchronous Programming in Modern C++

Hartmut Kaiser (hkaiser@cct.lsu.edu)
Today’s Parallel Applications
Real-world Problems

- Insufficient parallelism imposed by the programming model
  - OpenMP: enforced barrier at end of parallel loop
  - MPI: global (communication) barrier after each time step

- Over-synchronization of more things than required by algorithm
  - MPI: Lock-step between nodes (ranks)

- Insufficient coordination between on-node and off-node parallelism
  - MPI+X: insufficient co-design of tools for off-node, on-node, and accelerators

- Distinct programming models for different types of parallelism
  - Off-node: MPI, On-node: OpenMP, Accelerators: CUDA, etc.
The Challenges

• We need to find a usable way to fully parallelize our applications

• Goals are:
  • Expose asynchrony to the programmer without exposing additional concurrency
  • Make data dependencies explicit, hide notion of ‘thread’ and ‘communication’
  • Provide manageable APIs and paradigms for uniformly handling parallelism
HPX
The C++ Standards Library for Concurrency and Parallelism

https://github.com/STEllAR-GROUP/hpx
HPX – The C++ Standards Library for Concurrency and Parallelism

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

• Enables using the Asynchronous C++ Standard Programming Model
  • Emergent auto-parallelization, intrinsic hiding of latencies,
HPX – The API

- As close as possible to C++ standard library, where appropriate, for instance
  - `std::thread, std::jthread`
  - `std::mutex`
  - `std::future`
  - `std::async`
  - `std::for_each(par, ...), etc.`
  - `std::latch, std::barrier`
  - `std::experimental::task_block`
  - `std::experimental::for_loop`
  - `std::bind`
  - `std::function`
  - `std::any`
  - `std::cout`
  - `hpx::thread (C++11), hpx::jthread (C++20)`
  - `hpx::mutex`
  - `hpx::future (including N4538, ‘Concurrency TS’)`
  - `hpx::async (including N3632)`
  - `hpx::for_each (C++17)`
  - `hpx::latch, hpx::barrier`
  - `hpx::experimental::task_block (TS V2)`
  - `hpx::experimental::for_loop (TS V2)`
  - `hpx::bind`
  - `hpx::function`
  - `hpx::any (C++20)`
  - `hpx::cout`
# Parallel Algorithms (C++17)

<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>inner_product</td>
<td>inplace_merge</td>
<td>is_heap</td>
<td>is_heap_until</td>
</tr>
<tr>
<td>is_partitioned</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>
<tr>
<td>unique</td>
<td>unique_copy</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Future of Computation
What is a (the) Future?

• Many ways to get hold of a (the) future, simplest way is to use (std) async:

```cpp
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
}
```
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
- Represents a data-dependency
- Makes asynchrony manageable
- Allows for composition of several asynchronous operations
- (Turns concurrency into parallelism)
Recursive Parallelism
Parallel Quicksort

template<typename RandomIter>
void quick_sort(RandomIter first, RandomIter last)
{
  ptrdiff_t size = last - first;
  if (size > 1) {
    RandomIter pivot = partition(first, last,
                                [p = first[size / 2]](auto v) { return v < p; });

    quick_sort(first, pivot);
    quick_sort(pivot, last);
  }
}
Parallel Quicksort: Parallel

template <typename RandomIter>
void quick_sort(RandomIter first, RandomIter last)
{
    ptrdiff_t size = last - first;
    if (size > threshold) {
        RandomIter pivot = partition(par, first, last,
            [p = first[size / 2]="auto v`: return v < p; `]);

        quick_sort(first, pivot);
        quick_sort(pivot, last);
    }
    else if (size > 1) {
        sort(seq, first, last);
    }
}
Parallel Quicksort: Futurized

template <typename RandomIter>
future<void> quick_sort(RandomIter first, RandomIter last)
{
    ptrdiff_t size = last - first;
    if (size > threshold) {
        future<RandomIter> pivot = partition(par(task), first, last,
            [p = first[size / 2]](auto v) { return v < p; });

        return pivot.then([=](auto pf) {
            auto pivot = pf.get();
            return when_all(quick_sort(first, pivot), quick_sort(pivot, last));
        });
    } else if (size > 1) {
        sort(seq, first, last);
    }
    return make_ready_future();
}
Parallel Quicksort: co_await (C++20)

template<typename RandomIter>
future<void> quick_sort(RandomIter first, RandomIter last)
{
    ptrdiff_t size = last - first;
    if (size > threshold) {
        RandomIter pivot = co_await partition(par(task), first, last,
            [p = first[size / 2]](auto v) { return v < p; });

        co_await when_all(
            quick_sort(first, pivot), quick_sort(pivot, last));
    }
    else if (size > 1) {
        sort(seq, first, last);
    }
}
Iterative Parallelism
Extending Parallel Algorithms

Sean Parent: C++ Seasoning, Going Native 2013
Extending Parallel Algorithms

- New algorithm: gather

```cpp
template <typename BiIter, typename Pred>
pair<BiIter, BiIter> gather(BiIter f, BiIter l, BiIter p, Pred pred)
{
    BiIter it1 = stable_partition(f, p, not1(pred));
    BiIter it2 = stable_partition(p, l, pred);
    return make_pair(it1, it2);
}
```
Extending Parallel Algorithms

• New algorithm: gather_async

```cpp
template <typename BiIter, typename Pred>
future<pair<BiIter, BiIter>> gather_async(BiIter f, BiIter l, BiIter p, Pred pred)
{
    future<BiIter> f1 = stable_partition(par(task), f, p, not1(pred));
    future<BiIter> f2 = stable_partition(par(task), p, l, pred);
    co_return make_pair(co_await f1, co_await f2);
}
```
Futurization

- Technique allowing to automatically transform code
  - Delay direct execution in order to reduce synchronization
  - Turns ‘straight’ code into ‘futurized’ code
  - Code no longer calculates results, but generates an execution tree representing the original algorithm
  - If the tree is executed it produces the same result as the original code
  - The execution of the tree is performed with maximum speed, depending only on the data dependencies of the original code
- Execution exposes the emergent property of being auto-parallelized
Recent Results
Merging White Dwarfs

Orbits: 4.13005

Primary Star Density

Secondary Star Density

3e+3 Max
1e-3 Refine
1e-5

1e-3 Max
1e-3 Refine
1e-5
1e-6
1e-7
Adaptive Mesh Refinement
Adaptive Mesh Refinement

Strong-scaling efficiency: 68.1%

Weak-scaling efficiency: 78.4%
The Solution to the Application Problem
The Solution to the Application Problems