HPX
The C++ Standards Library for Concurrency and Parallelism

Hartmut Kaiser (hkaiser@cct.lsu.edu)
HPX – A General Purpose Runtime System

- The C++ Standards Library for Concurrency and Parallelism

- Exposes a coherent and uniform, C++ standards-conforming 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 seamless data parallelism orthogonally to task-based parallelism

- HPX represents an innovative mixture of
  - A global system-wide address space (AGAS - Active Global Address Space)
  - Fine grain parallelism and lightweight synchronization
  - Combined with implicit, work queue based, message driven computation
  - Support for hardware accelerators
HPX – A C++ Standard Library

- Widely portable
  - Platforms: x86/64, Xeon/Phi, ARM 32/64, Power, BlueGene/Q
  - Operating systems: Linux, Windows, Android, OS/X

- Well integrated with compiler’s C++ Standard libraries

- Enables writing applications which out-perform and out-scale existing applications based on OpenMP/MPI
  - [http://stellar-group.org/libraries/hpx](http://stellar-group.org/libraries/hpx)
  - [https://github.com/STEllAR-GROUP/hpx/](https://github.com/STEllAR-GROUP/hpx/)

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

- Can be used as a platform for research and experimentation
HPX – A C++ Standard Library

- C++1y Concurrency/Parallelism APIs
- Threading Subsystem
- Active Global Address Space (AGAS)
- Performance Counter Framework
- Local Control Objects (LCOs)
- Parcel Transport Layer
Programming Model

- Focus on the logical composition of data processing, rather than the physical orchestration of parallel computation
- Provide useful abstractions that shield programmer from low-level details of parallel and distributed processing
- Centered around data dependencies not communication patterns
- Make data dependencies explicit to system thus allows for auto-magic parallelization
- Basis for various types of higher level parallelism, such as iterative, fork-join, continuation-style, asynchronous, data-parallelism
- Enable runtime-based adaptivity while applying application-defined policies
Programming Model

• The consequent application of the Concept of Futures
  • Make data dependencies explicit and visible to the runtime

• Implicit and explicit asynchrony
  • Transparently hide communication and other latencies
  • Makes over-subscription manageable
  • Uniform API for local and remote operation
    • Local operation: create new thread
    • Remote operation: send parcel (active message), create thread on behalf of sender

• Work-stealing scheduler
  • Inherently multi-threaded environment
  • Supports millions of concurrently active threads, minimal thread overhead
  • Enables transparent load balancing of work across all execution resources inside a locality

• API is fully conforming with C++11/C++14 and ongoing standardization efforts
HPX – The API

- As close as possible to C++11/14/17 standard library, where appropriate, for instance
  - std::thread
  - std::mutex
  - std::future
  - std::async
  - std::bind
  - std::function
  - std::tuple
  - std::any
  - std::cout
  - std::for_each(par, ...), etc.
  - std::experimental::task_block

- std::thread
- std::mutex
- std::future (including N4538, ‘Concurrency TS’)
- std::async (including N3632)
- std::bind
- std::function
- std::tuple
- std::any (N3508)
- std::cout
- std::parallel::for_each (N4507, ‘Parallelism TS’, C++17)
- std::parallel::task_block (N4411)
Control Model

- How is parallelism achieved?
  - Explicit parallelism:
    - Low-level: thread
    - Middle-level: async(), dataflow(), future::then()
  - Higher-level constructs
    - Parallel algorithms (parallel::for_each and friends, fork-join parallelism for homogeneous tasks)
      - Asynchronous algorithms (alleviates bad effect of fork/join)
    - Task-block (fork-join parallelism of heterogeneous tasks)
      - Asynchronous task-blocks
    - Continuation-style parallelism based on composing futures (task-based parallelism)
    - Data-parallelism on accelerator architectures (vector-ops, GPUs)
      - Same code used for CPU and accelerators
## 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>
STREAM Benchmark

```cpp
std::vector<double> a, b, c;    // data

// ... init data

auto a_begin = a.begin(), a_end = a.end(), b_begin = b.begin() ...;

// STREAM benchmark
parallel::copy(par, a_begin, a_end, c_begin);                       // copy step: c = a
parallel::transform(par, c_begin, c_end, b_begin,                    // scale step: b = k * c
    [](double val) { return 3.0 * val; });
parallel::transform(par, a_begin, a_end, b_begin, b_end, c_begin,     // add two arrays: c = a + b
    [](double val1, double val2) { return val1 + val2; });
parallel::transform(par, b_begin, b_end, c_begin, c_end, a_begin,     // triad step: a = b + k * c
    [](double val1, double val2) { return val1 + 3.0 * val2; });
```
Dot-product: Vectorization

```cpp
std::vector<float> data1 = {...};
std::vector<float> data2 = {...};

double p = parallel::inner_product(
    datapar, // parallel and vectorized execution
    std::begin(data1), std::end(data1),
    std::begin(data2),
    0.0f,
    [](auto t1, auto t2) { return t1 + t2; }, // std::plus<>()
    [](auto t1, auto t2) { return t1 * t2; } // std::multiplies<>()
);
```
Control Model

• How is synchronization expressed?
  • Low-level (thread-level) synchronization: mutex, condition_variable, etc.
  • Replace (global) barriers with finer-grain synchronization (synchronize of a ‘as-need-basis’)
    • Wait only for immediately necessary dependencies, forward progress as much as possible
  • Many APIs hand out a future representing the result
    • Parallel and sequential composition of futures (future::then0, when_all0, etc.)
    • Orchestration of parallelism through launching and synchronizing with asynchronous tasks
  • Synchronization primitives: barrier, latch, semaphores, channel, etc.
    • Synchronize using futures
Synchronization with Futures

- 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)
Data Model

• AGAS essential underpinning for all data management
  • Foundation for syntactic semantic equivalence of local and remote operations

• Full spectrum of C++ data structures are available
  • Either as distributed data structures or for SPMD style computation

• Explicit data partitioning, manually orchestrated boundary exchange
  • Using existing synchronization primitives (for instance channels)

• Use of distributed data structures, like partitioned_vector
  • Use of parallel algorithms
  • Use of co-array like layer (FORTRAN users like that)

• Load balancing: migration
  • Move objects around in between nodes without stopping the application
Small Example
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 = parallel::stable_partition(par(task), f, p, not1(pred));
    future<BiIter> f2 = parallel::stable_partition(par(task), p, l, pred);
    return dataflow(
        unwrapped([](BiIter r1, BiIter r2) { return make_pair(r1, r2); }),
        f1, f2);
}
```
Extending Parallel Algorithms (await)

- 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 = parallel::stable_partition(par(task), f, p, not1(pred));
  future<BiIter> f2 = parallel::stable_partition(par(task), p, l, pred);
  co_return make_pair(co_await f1, co_await f2);
}
```