Parallelism in Modern C++

Task-based parallelism as the basis for all higher-level APIs

Hartmut Kaiser (hkaiser@cct.lsu.edu)
HPX

A General Purpose Parallel Runtime System for Applications of any Scale
HPX – A General Purpose Runtime System

- General purpose parallel runtime system for applications of any scale
- 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.
- 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
  - Full semantic equivalence of local and remote execution, and
  - Explicit support for hardware accelerators and vectorization
HPX – A General Purpose Runtime System

• 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 General Purpose Runtime System

- C++1y Parallelism APIs
- Threading Subsystem
- Active Global Address Space (AGAS)
- Local Control Objects (LCOs, Synchronization)
- Parcel Transport Layer (Network)
- Performance Counter Framework

Policy Engine/Policies
HPX – A General Purpose Runtime System

C++1y Parallelism APIs

Threading Subsystem

Active Global Address Space (AGAS)

Local Control Objects (LCOs, Synchronization)

Parcel Transport Layer (Network)

Policy Engine/Policies

Performance Counter Framework
HPX — The API

• As close as possible to C++1y standard library, where appropriate, for instance
  • `std::thread` to `hpx::thread`
  • `std::mutex` to `hpx::mutex`
  • `std::future` to `hpx::future (including N4107, ‘Concurrency TS’)`
  • `std::async` to `hpx::async (including N3632)`
  • `std::bind` to `hpx::bind`
  • `std::function` to `hpx::function`
  • `std::tuple` to `hpx::tuple`
  • `std::any` to `hpx::any (P0220, ‘Library Fundamentals TS’)`
  • `std::cout` to `hpx::cout`
  • `std::parallel::for_each, etc.` to `hpx::parallel::for_each (N4105, ‘Parallelism TS’)`
  • `std::parallel::task_block` to `hpx::parallel::task_block (N4411)`
  • `std::vector` to `hpx::vector, hpx::partitioned_vector`

• Extensions to the standard APIs, where necessary
  • While maintaining full compatibility
Parallelism in C++

A Vision for Coherent Higher-level APIs without the need for OpenMP, OpenAcc, or CUDA, etc.
Concepts and Types of Parallelism
Types of Parallelism

• Current state of standard C++:
  • Parallelism TS: iterative parallelism (moved to be included into C++17)
  • Concurrency TS: task-based, asynchronous, and continuation style parallelism
  • N4411: task blocks for fork-join parallelism of heterogeneous tasks
  • N4406, PR0008R0: executors
  • PR0057R0: resumable functions (co_await, etc.)

• Missing:
  • Integration of the above
  • Parallel ranges
  • Vectorization is being discussed
  • Extensions for GPUs, many-core, distributed, and high-performance computing

• The goal has to be to make parallelism in C++ independent of any external solutions such as OpenMP, OpenACC, etc.
  • HPX makes C++ independent of MPI as well
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)
What is a (the) Future?

- Many ways to get hold of a 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
}
```
Parallel Algorithms
**Parallel Algorithms**

<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>
Parallel Algorithms

- Similar to standard library facilities known for years
  - Add execution policy as first argument

- Execution policies have associated default executor and default executor parameters
  - `par` → parallel executor, static chunk size
  - `seq` → sequential executor, no chunking

- Rebind executor and executor parameters:

```cpp
// Simplest case: parallel execution policy
std::vector<double> d(1000);
parallel::fill(par,
    begin(d), end(d), 0.0);
```
Parallel Algorithms

• Similar to standard library facilities known for years
  • Add execution policy as first argument

• Execution policies have associated default executor and default executor parameters
  • `par` → parallel executor, static chunk size
  • `seq` → sequential executor, no chunking

• Rebind executor and executor parameters:

```cpp
// rebind execution policy
//     .on(): executor object, ‘where and when’
//     .with(): parameter object(s), possibly executor specific parameters

std::vector<double> d(1000);
parallel::fill(par.on(exec).with(par1, par2, ...), begin(d), end(d), 0.0);
```
Rebind Execution Policies

numa_executor exec;
auto policy1 = par.on(exec); // rebind only executor

static_chunk_size param;
auto policy2 = par.with(param); // rebind only executor parameter

auto policy3 = par.on(exec).with(param); // rebind both
// uses default execution policy: par
std::vector<double> d = { ... };
parallel::fill(par, begin(d), end(d), 0.0);

// rebind par to user-defined executor
my_executor my_exec = ...;
parallel::fill(par.on(my_exec), begin(d), end(d), 0.0);

// rebind par to user-defined executor and user defined executor parameters
my_params my_par = ...;
parallel::fill(par.on(my_exec).with(my_par), begin(d), end(d), 0.0);
Execution Policies (HPX Extensions)

- Extensions: asynchronous execution policies
  
  - `parallel_task_execution_policy` (asynchronous version of `parallel_execution_policy`), generated with `par(task)`
  - `sequential_task_execution_policy` (asynchronous version of `sequential_execution_policy`), generated with `seq(task)`

- In all cases the formerly synchronous functions return a future<>
- Instruct the parallel construct to be executed asynchronously
- Allows integration with asynchronous control flow
Execution Policies (HPX Extensions)

• Extensions: vectorization execution policies

  • `datapar_task_execution_policy` (asynchronous version of `datapar_execution_policy`), generated with `datapar`, `datapar(task)`
  • `dataseq_task_execution_policy` (asynchronous version of `dataseq_execution_policy`), generated with `dataseq`, `dataseq(task)`

• Instruct the algorithm to apply certain transformations to used data types allowing for vectorization of code
  • Requires external library: currently Vc ([https://github.com/VcDevel/Vc](https://github.com/VcDevel/Vc)), possibly Boost.SIMD
  • Requires use of generic lambdas (C++14) or polymorphic function objects
Executors

• Executors must implement one function: `async_execute(F && f)`

• Invocation of executors happens through `executor_traits` which exposes (emulates) additional functionality:

  ```cpp
  executor_traits<my_executor_type>::execute(
      my_executor,
      [](...){ // perform task },
      ...
  );
  ```

• Four modes of invocation: single async, single sync, bulk async and bulk sync
  • The async calls return a future
Executor Examples

- sequential_executor, parallel_executor:
  - Default executors corresponding to par, seq

- this_thread_executor

- distribution_policy_executor
  - Use one of HPX’s (distributed) distribution policies, specify node(s) to run on

- host::parallel_executor
  - Specify core(s) to run on (NUMA aware)

- cuda::default_executor
  - Use for running things on GPU

- Etc.
Executor Parameters (HPX Extension)

• Same scheme as for executor/executor_traits:
  • parameter/executor_parameter_traits

• Various execution parameters, possibly executor specific

• For instance:
  • Allow to control the grain size of work
    • i.e. amount of iterations of a parallel for_each run on the same thread
    • Similar to OpenMP scheduling policies: static, guided, dynamic
      • auto_chunk_size, static_chunk_size, dynamic_chunk_size
    • Much more fine control
    • Used by parallel algorithms to adjust chunk size
  • Specify GPU-kernel name for certain platforms
    • gpu_kernel<foobar>
  • Specify which other arrays to prefetch
Data placement
Data Placement

• Different strategies for different platforms
  • Need interface to control explicit placement of data
    • NUMA architectures
    • GPUs
    • Distributed systems
  • Use `std::allocator<T>` interfaces
    • NUMA architectures: first touch
    • Slightly extended: bulk-operations for allocation, construction, destruction, and deallocation
Data Placement

- **HPX:**
  - `hpx::vector<T, Alloc>`
    - Same interface as `std::vector<T>`
    - Manages data locality through allocator
    - Uses execution target objects for data placement
    - Allows for direct manipulation of data on NUMA domains, GPUs, remote nodes, etc.

- `hpx::partitioned_vector<T>`
  - Same interface as `std::vector<T>`
  - Segmented data store
    - Segments can be `hpx::vector<T, Alloc>`
  - Uses `distribution_policy` for data placement
  - Allows for manipulation of data on several targets
Data Placement

- Extending std::allocator_traits
  - Adding functionality to copy data
    - CPU: trivial
    - GPU: platform specific data transfer, hooked into parallel::copy
    - Distributed: maps onto network, possibly RDMA (put/get)
  - Adding functionality to access single elements
    - CPU: trivial
    - GPU: slow, but possible
    - Distributed: maps onto network
Execution Targets
One Ring to Rule them All
Execution Targets

- Opaque types which represent a place in the system
  - Used to identify data placement
  - Used to specify execution site close to data

- Targets encapsulate architecture specifics
  - E.g. `cuda::target`, `host::target`

- Allocators to be initialized from targets
  - Customization of data placement
    - NUMA domain: `host::block_allocator`
    - (possibly remote) GPU device: `cuda::allocator`
    - Locality, i.e. (possibly remote) node

- Executors to be initialized from targets as well
  - Make sure code is executed close to placed data
Examples and Results
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);
    return make_pair(co_await f1, co_await f2);
}
```
STREAM Benchmark

• Assess memory bandwidth

• Series of parallel for loops, 3 arrays (a, b, c)
  • copy step: \( c = a \)
  • scale step: \( b = k \times c \)
  • add two arrays: \( c = a + b \)
  • triad step: \( a = b + k \times c \)

• Best possible performance possible only if data is placed properly
  • Data has to be located in memory of NUMA-domain where thread runs

• OpenMP: implicitly by using ‘first touch’, i.e. run initialization and actual benchmark using same thread
  • #pragma omp parallel for schedule(static)
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; });
```

9/19/2016

Parallelism in Modern C++ (Talk at CppCon 2016),
Hartmut Kaiser
STREAM Benchmark (CPU)

```cpp
class Benchmark {
public:
    Benchmark() {
        // Initialize data
        vector<double, allocator> a(allocator), b(allocator), c(allocator);
    }

    // Define execution site
    executor exec(tgt);

    // Define data placement
    allocator alloc(tgt, ...);

    // Bound execution policy
    auto policy = par.on(exec).with(static_chunk_size());

    // STREAM benchmark
    parallel::copy(policy, a_begin, a_end, c_begin);
};
```

// where and when, here CPU, NUMA domain 0
STREAM Benchmark: HPX vs. OpenMP

TRIAD STREAM Results
(50 million data points)

- HPX (1 NUMA Domain)
- OpenMP (1 NUMA Domain)
- HPX (2 NUMA Domains)
- OpenMP (2 NUMA Domains)
Extending to GPUs
STREAM Benchmark (GPU)

cuda::target target("Tesla C2050");  // where and when, here NVidia GPU (CUDA)

using executor = cuda::default_executor;
using allocator = cuda::allocator<double>;

executor exec(tgt);                   // define execution site
allocator alloc(tgt);                 // define data placement

std::vector<double> data = { ... };  // init data on host
hpdx::vector<double, allocator> a(alloc), b(alloc), c(alloc);  // data on device

parallel::copy(par, data.begin(), data.end(), a_begin);  // copy data to device

// STREAM benchmark
// ...
STREAM Benchmark: HPX vs. OpenCL
Vectorization
Dot-product: Parallel Execution

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

inner_product(
    par,
    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<>()
);
```
Dot-product: Vectorization

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

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<>()
);
```
Dot-Product: Results

Dot-product of 100,000 Points

Dot-product of 1,000,000 Points

Dot-product of 10,000,000 Points
Partitioned Vector
Finding Min/Max on Host

\begin{verbatim}
std::vector<targets> targets = host::get_numa_targets();
partitioned_vector<int> v(size, 
    host::target_distribution_policy(targets));

host::numa_executor exec(targets);
generate(par.on(exec), v.begin(), v.end(), rand);

auto iters = minmax_element(par.on(exec), v.begin(), v.end());

std::cout << "Minimal element: " << *(iter.first);
std::cout << "Maximal element: " << *(iter.second);
\end{verbatim}
Finding Min/Max on GPU

```cpp
std::vector<targets> targets = cuda::get_device_targets();
partitioned_vector<int> v(size,
    host::target_distribution_policy(targets));

cuda::default_executor exec(targets);
generate(par.on(exec), v.begin(), v.end(), rand);

auto iters = minmax_element(par.on(exec), v.begin(), v.end());

std::cout << "Minimal element: " << *(iter.first);
std::cout << "Maximal element: " << *(iter.second);
```
Loop Prefetching
Automatic Loop Prefetching

```cpp
std::vector<double> a, b, c;
parallel::for_loop(par, 0, a.size(),
    [&](int i) { a[i] = b[i] + 3.0 * c[i]; });

// add automatic prefetching for b and c
std::vector<double> a, b, c;
parallel::for_loop(par.with(prefetch(b, c)), 0, a.size(),
    [&](int i) { a[i] = b[i] + 3.0 * c[i]; });
```
Automatic Loop Prefetching (Results)

TRIAD STREAM Results (1 NUMA Domain)

- HPX(With Prefetching) - 100,000 data
- HPX(Without Prefetching) - 100,000 data
- HPX(With Prefetching) - 1,000 data
- HPX(Without Prefetching) - 1,000 data

Bandwidth [MB/s] vs. Number of Threads