HPX

The C++ Standards Library for Concurrency and Parallelism

Hartmut Kaiser (hkaiser@cct.lsu.edu)

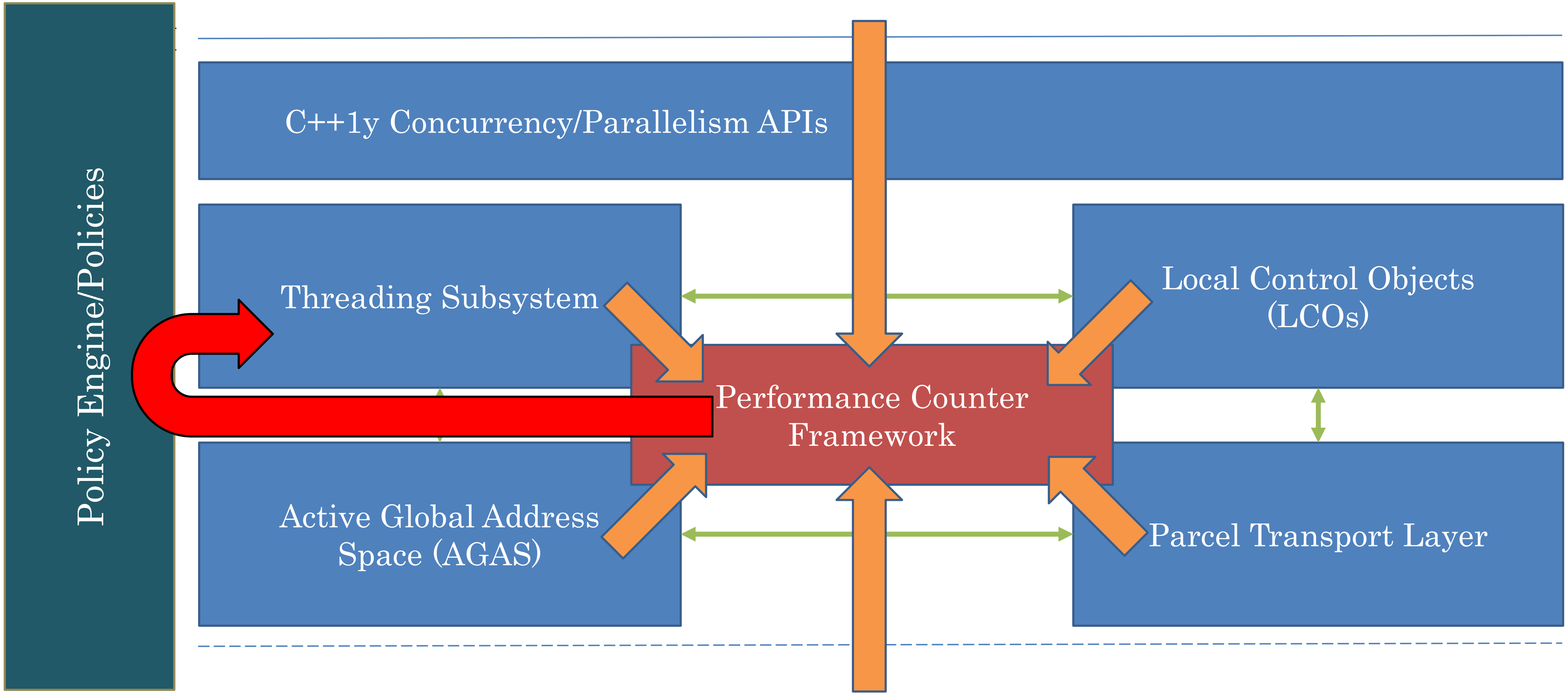
HPX – A General Purpose Runtime System

- 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
  - 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



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	hpx::thread
• std::mutex	hpx::mutex
• std::future	hpx::future (including N4538, ‘Concurrency TS’)
• 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::for_each(par, …), etc.	hpx::parallel::for_each (N4507, ‘Parallelism TS’, C++17)
• std::experimental::task_block	hpx::parallel::task_block (N4411)

STE||AR GROUP

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)

adjacent_difference	adjacent_find	all_of	any_of
copy	copy_if	copy_n	count
count_if	equal	exclusive_scan	fill
fill_n	find	find_end	find_first_of
find_if	find_if_not	for_each	for_each_n
generate	generate_n	includes	inclusive_scan
inner_product	inplace_merge	is_heap	is_heap_until
is_partitioned	is_sorted	is_sorted_until	lexicographical_compare
max_element	merge	min_element	minmax_element
mismatch	move	none_of	nth_element
partial_sort	partial_sort_copy	partition	partition_copy
reduce	remove	remove_copy	remove_copy_if
remove_if	replace	replace_copy	replace_copy_if
replace_if	reverse	reverse_copy	rotate
rotate_copy	search	search_n	set_difference
set_intersection	set_symmetric_difference	set_union	sort
stable_partition	stable_sort	swap_ranges	transform
uninitialized_copy	uninitialized_copy_n	uninitialized_fill	uninitialized_fill_n
unique	unique_copy		

STE||AR GROUP

STREAM Benchmark

```
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; });
```

STE||AR GROUP

Dot-product: Vectorization

```
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<>()
);
```

STE||AR GROUP

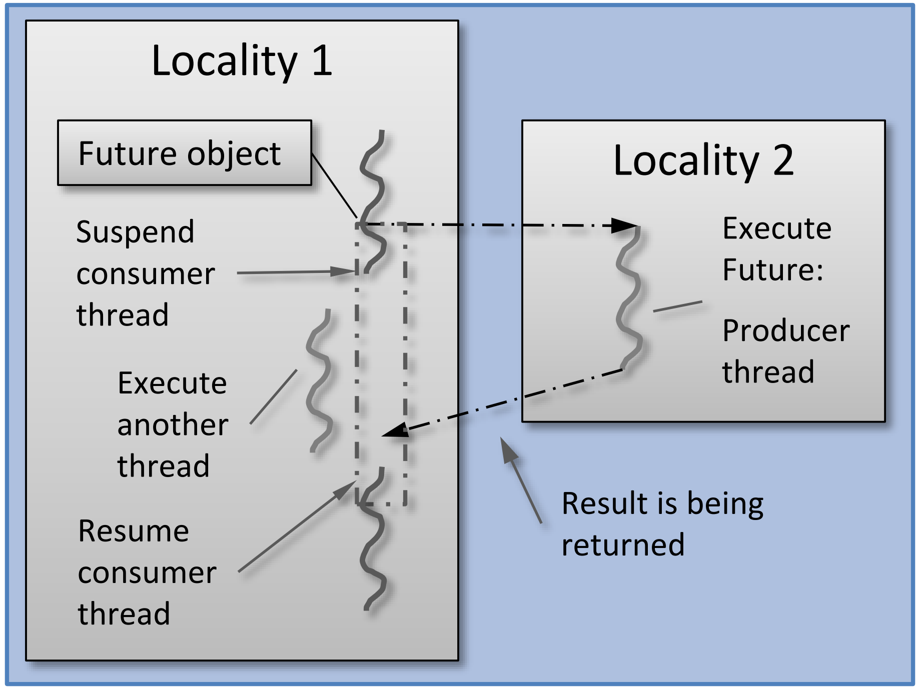
Control Model

- 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::then(), when_all(), etc.)
    - Orchestration of parallelism through launching and synchronizing with asynchronous tasks
  - Synchronization primitives: barrier, latch, semaphores, channel, etc.
    - Synchronize using futures

STE||AR GROUP

Synchonization 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)

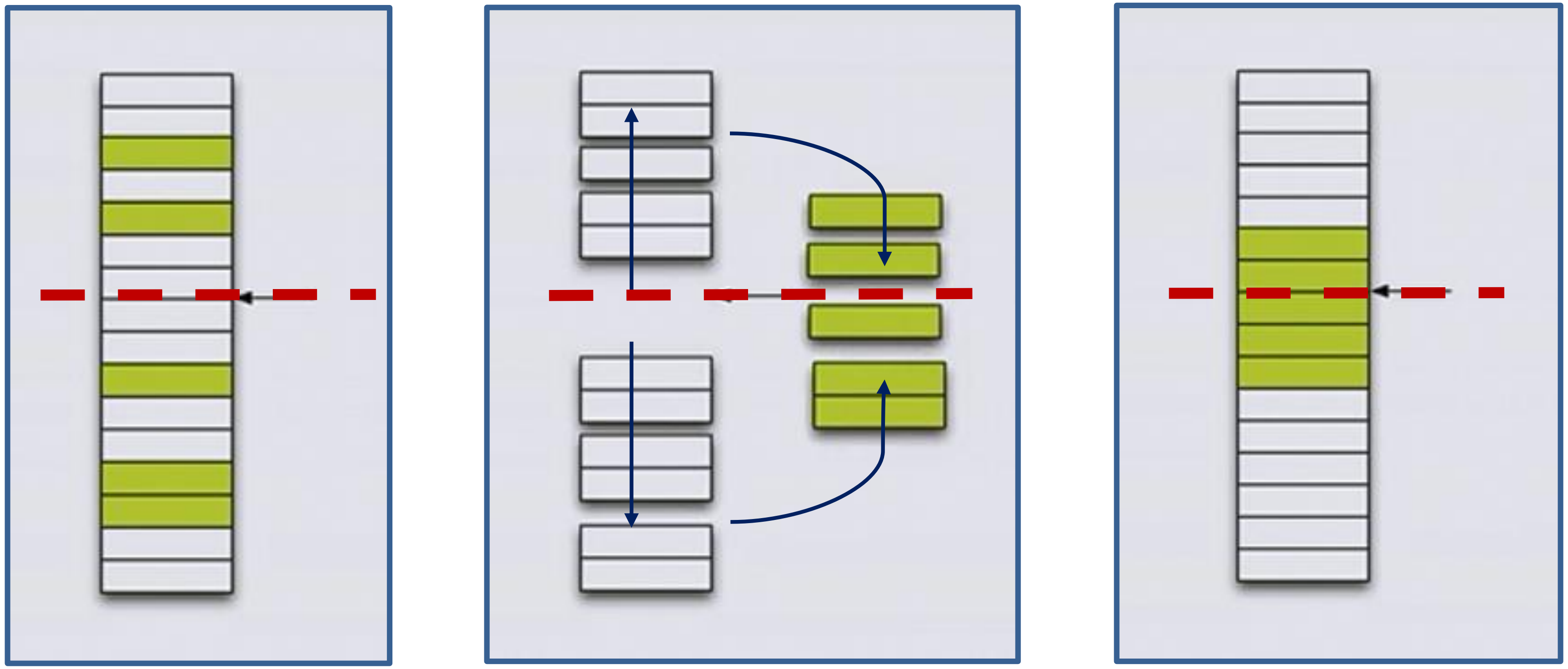
STE||AR GROUP

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

```
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);
}
```

Sean Parent: C++ Seasoning, Going Native 2013

Extending Parallel Algorithms

- New algorithm: gather_async

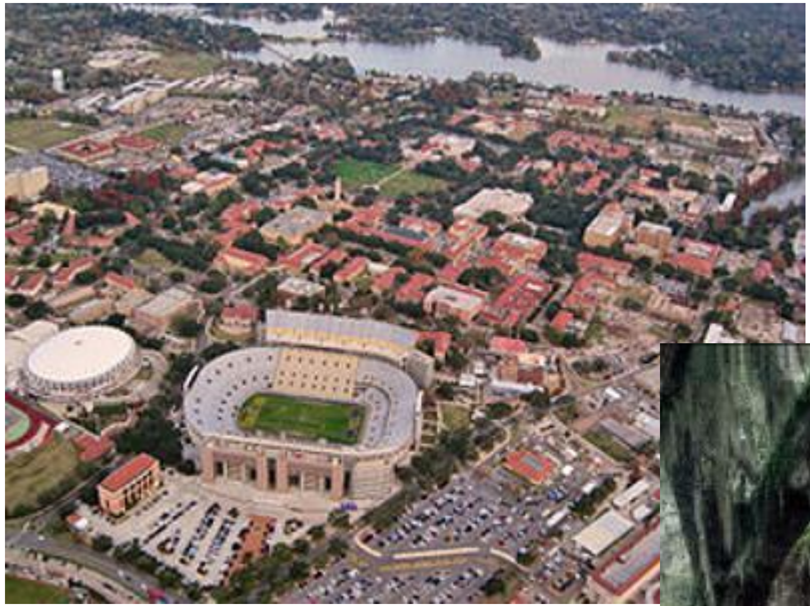
```
template <typename BiIter, typename Pred>
future<pair<BiIter, BiIter>> gather_async(BiIter f, BiIter l, BiIter p, Pred pred)
{
    future<BiIter> f1 = parallel::stable_partition(par(task), f, p, not1(pred));
    future<BiIter> f2 = parallel::stable_partition(par(task), p, l, pred);
    return dataflow(
        unwrapped([](BiIter r1, BiIter r2) { return make_pair(r1, r2); }),
        f1, f2);
}
```

STE||AR GROUP

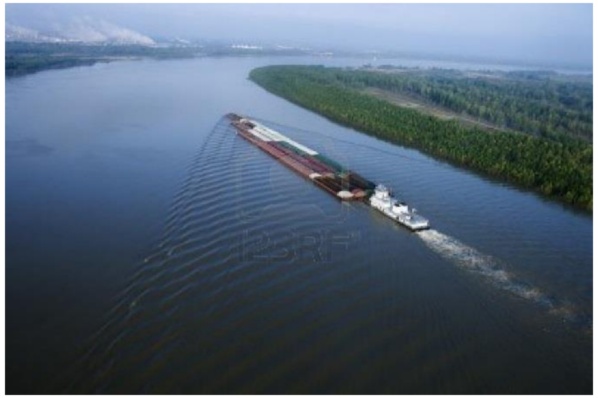
Extending Parallel Algorithms (await)

- New algorithm: gather_async

```
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);
}
```



LSU CENTER FOR COMPUTATION & TECHNOLOGY



STE||AR GROUP