HIGH-PERFORMANCE EXECUTION
Across potentially many nodes

Parallelism + Asynchrony + Data coordination

×1000
BEYOND PARALLELISM
Provide the things needed for scalable execution

Continue to evolve

Parallelism

Asynchrony

Data coordination

Common mechanisms are needed
C++ STANDARDIZATION
What is most important for the Standard Library?

Common concepts which serve to organize & compose software modules

Iterators are a prime example
KEY INGREDIENTS
Concepts needed for structuring libraries & applications

Identify things: pointers, iterators, ranges

Identify place to allocate storage: allocators

Identify where/how to execute: execution policies, executors

Identify dependencies: futures

Identify affinity of workers and data: conforming index spaces
for_each(par, begin, end, function);

Execution Policy

Specify *how* operation may execute.
C++17 PARALLEL ALGORITHMS
Many useful patterns beyond loops

Parallelizable algorithms in STL

for_each
transform
copy_if
sort
set_intersection
etc.

New additions for parallelism
reduce
exclusive_scan
inclusive_scan
transform_reduce
transform_inclusive_scan
transform_exclusive_scan
NVIDIA’S THRUST LIBRARY
CUDA C++ Parallel Algorithms

for_each(par, begin, end, function);

Device-accessible Data

Device Function

e.g., via unified memory

for GPU execution.
Diverse Execution Resources

Diverse Control Structures

async(...) for_each(...)
define_task_block(...) bulk_invoke(...)
your_favorite_control_structure(...)
Diverse Control Structures

- async(...)
- for_each(...)
- define_task_block(...)
- bulk_invoke(...)
- your_favorite_control_structure(...)
EXECUTORS

Compositional, uniform control of where/how execution occurs

```cpp
auto ex = my_thread_pool.executor(...);
```

Executor

Used to *submit work* for some operation.

Serves to identify *where* work will be run.
EXECUTORS
Compositional, uniform control of where/how execution occurs

```cpp
auto ex = obtain_an_executor(...);
async(ex, ...);
for_each(par.on(ex), ...);
parallel_linear_solver(ex, ...);
```
C++17 PARALLEL ALGORITHMS

Specified to be synchronous

```cpp
// Input parameters must be available immediately
std::transform(par, x.begin(), x.end(), y.begin(), F);

// ... synchronize with transform() and resume this thread ...
float sum = std::reduce(par, x.begin(), x.end());

return sum;
```
ASYNCHRONOUS ALGORITHMS

Need suitable ranges and futures

// Inputs x & y are future<Rangetype> objects
auto z = async_transform(par, x, y, F);

// ... z is also a future<Rangetype>; no synchronization here
auto sum = async_reduce(par, z);

return sum.get();
CHAINED ASYNCHRONY
Mediated in C++ by futures and executors

```cpp
auto future = launch(A)
when A completes: launch(B, future.get())
```

Execution timeline:

- A Signal
- Evaluate
- Schedule
- Launch
- B

Maximize flexibility, composability, and performance here
CHAINED ASYNCHRONY

Don’t hide dependencies

auto future = launch(A)

Running code to decide what to do next:

future.then([](...) { ex.execute(B); });

A Signal Evaluate Schedule Launch B

Forces at least some of these stages to run on a thread somewhere
CHAINED ASYNCHRONY

Expose dependencies

auto future = launch(A)

Telling the platform what to do next:

ex.then_execute(future, B);

Permits more implementations, including hardware mechanisms & construction of task graphs
PAST
Singleton actions for immediate submission

- Task run by collection of threads
- Memory copy
- Memory allocation
- ...

NVIDIA Platform

Immediately submitted to specified GPU(s)
PRESENT

Actions with predecessors which can be deferred

NVIDIA Platform

Action

Predecessor actions

Immediately submitted to specified GPU(s)

– or –

Deferred into a graph
CUDA GRAPHS

Mechanism for communicating these dependencies to CUDA

NVIDIA Platform

Immediately submitted to specified GPU(s)

Action

Predecessor actions

Deferred into a graph

A → B → C → D → E → Y → X → End