P0361R1

Invoking Algorithms Asynchronously

Hartmut Kaiser, Thomas Heller, Bryce Adelstein Lelbach,
John Biddiscombe, Michael Wong

Motivation

- Parallel algorithms in C++17 are synchronous
  - Fork/join, imposes implicit barrier
  - Introduces implicit barrier impeding parallel efficiency
- No means of controlling when and how this barrier is imposed
- This paper introduces asynchronous algorithms
  - Algorithms return a future
  - Does not remove barrier, but allows to overlap the 'tapering off' with other work

New Execution Policies

- 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)`
  - `parallel_unsequenced_task_execution_policy` (asynchronous version of `parallel_unsequenced_execution_policy`), generated with `par_unseq(task)`
  - In all cases the formerly synchronous functions return a future<> representing the overall result
  - Instruct the parallel construct to be executed asynchronously
  - Allows integration with asynchronous control flow

STE||AR GROUP

Example

```
using namespace std::experimental::parallel::v1;
std::vector<int> data = { ... };

// NEW: asynchronous, sequential execution
std::future<void> f1 = sort(seq(task), data.begin(), data.end());
// ... perform other work
f1.get(); // synchronize with the asynchronous sequential sort()

// NEW: asynchronous execution, allow for parallelization of the algorithm
std::future<void> f2 = sort(par(task), data.begin(), data.end());
// ... perform other work
f2.get(); // synchronize with the asynchronous parallel sort()

// NEW: asynchronous execution, allow for parallelization and vectorization
// of the algorithm
std::future<void> f3 = sort(par_unseq(task), data.begin(), data.end());
// ... perform other work
f3.get(); // synchronize with the asynchronous parallel vectorized sort()
```

Implementation

- Implementation experience in HPX
- Use experience for 2 years in large scientific applications
- Has shown to help improving parallel efficiency of applications by factor of 2
  - Figure shows the utilization of cores in a time-step based stencil code, comparing synchronous and asynchronous operation

STE||AR GROUP

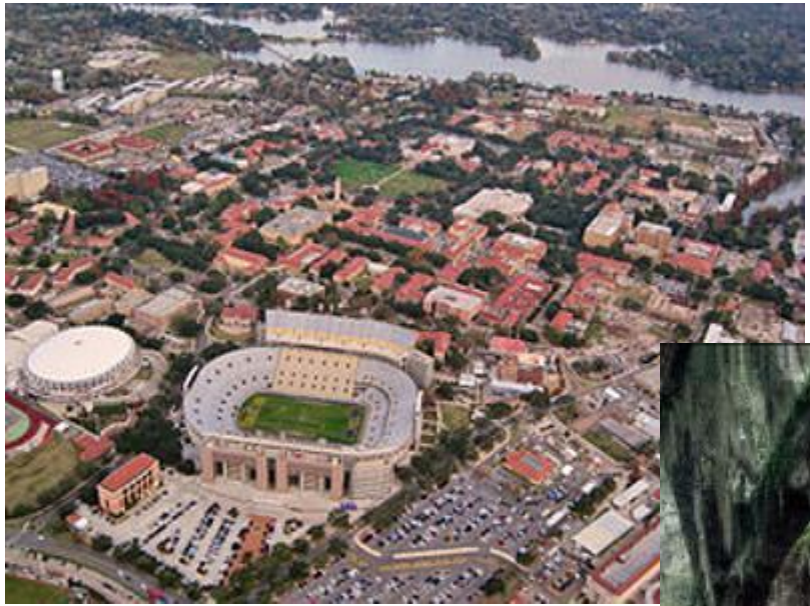
Discussion

- Use separate overloads for asynchronous algorithms instead of new execution policies
- Rationale:
  - Asynchronous algorithms may need different set of algorithms
- This is certainly a possible solution
  - Prevents generic programming
  - In our implementation we have seen no need for different set of arguments

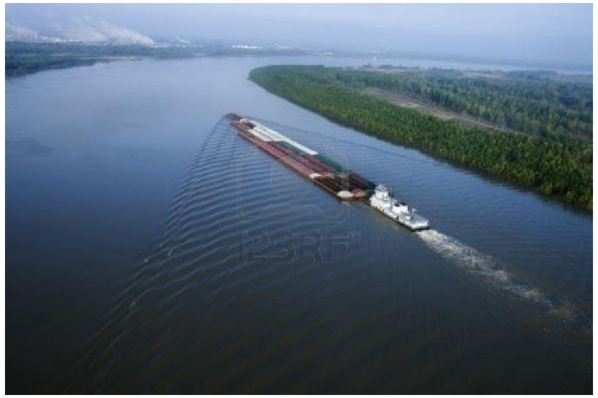
Discussion

- Do not introduce asynchronous algorithms now as those may be subsumed by core language functionalities (such as suspendable functions, see [P0071R2]) which are currently proposed and under discussion.
  - Asynchronous algorithms are a feature requested by several people in SG1 and SG14.
  - Solid implementation and usage experience
- The suspendable functions proposal without any doubt has merit and may partially or fully subsume the features proposed
  - Requires compiler support
  - Unclear if and when available
  - No implementation experience
- We would rather move forward with an existing and proven solution now to give users more experience with possible implementations

STE||AR GROUP



LSU CENTER FOR COMPUTATION & TECHNOLOGY



STE||AR GROUP