Parallelism in C++
Lecture 1
hkaiser@cct.lsu.edu
Amdahl’s Law (Strong Scaling)

\[ S = \frac{1}{(1 - P) + \frac{P}{N}} \]

- S: Speedup
- P: Proportion of parallel code
- N: Number of processors

Figure courtesy of Wikipedia (http://en.wikipedia.org/wiki/Amdahl%27s_law)
Rule 1

Parallelize Applications as Much as Humanly Possible
The 4 Horsemen of the Apocalypse
The 4 Horsemen of the Apocalypse

- **Starvation**
  - Insufficient concurrent work to maintain high utilization of resources

- **Latencies**
  - Time-distance delay of remote resource access and services

- **Overheads**
  - Work for management of parallel actions and resources on critical path which are not necessary in sequential variant

- **Waiting for Contention resolution**
  - Delays due to lack of availability of oversubscribed shared resources

**Impose upper bound on both, weak and strong scaling**
Universal Scalability Law

\[ X(N) = \frac{\lambda N}{1 + \sigma (N - 1) + \kappa N (N - 1)} \]

- \( \lambda \): Scaling efficiency
- \( \delta \): Contention
- \( \kappa \): Latencies (‘Crosstalk’)
- \( N \): Number of processors
Real-world Problems

• Insufficient parallelism imposed by the programming model
  • OpenMP: enforced barrier at end of parallel loop
  • MPI: global (communication) barrier after each time step

• Over-synchronization of more things than required by algorithm
  • MPI: Lock-step between nodes (ranks)

• Insufficient coordination between on-node and off-node parallelism
  • MPI+X: insufficient co-design of tools for off-node, on-node, and accelerators

• Distinct programming models for different types of parallelism
  • Off-node: MPI, On-node: OpenMP, Accelerators: CUDA, etc.
Real-world Problems

- Even standard algorithms added to C++17 enforce fork-join semantics
Fork/Join Parallelism

[Diagram showing fork/join parallelism with barrier and reduction stages, active and idle threads]

Hartmut Kaiser
Rule 2

Use a Programming Environment that Embraces SLOW
Overheads: Thought-Experiment

Execution Time over Grain Size
(for different amounts of overheads per thread, 16 Cores)

Execution time (relative to sequential time)

Grain Size (amount of work per thread)
Overheads: The Worst of All?

• Even relatively small amounts of work can benefit from being split into smaller tasks
  • Possibly huge amount of ‘threads’
    • In the previous thought-experiment we ended up considering up to 10 million threads
    • Best possible scaling is predicted to be reached when using 10000 threads (for 1 second worth of work)

• Several problems
  • Impossible to work with that many kernel threads (p-threads)
  • Impossible to reason about this amount of tasks
  • Requires abstraction mechanism
Rule 3

Allow for your Grainsize to be Variable
Overheads: The Worst of All?

Execution Time over Grainsize
(1D Stencil, Solving Heat Diffusion)

Execution time (relative to sequential time)

Grainsize (amount of work per thread)

Sequential Time
Time (16 Cores)
Rule 4

Oversubscribe and Balance Adaptively
The Challenges

• We need to find a usable way to fully parallelize our applications

• Goals are:
  • Expose asynchrony to the programmer without exposing additional concurrency
  • Make data dependencies explicit, hide notion of ‘thread’ and ‘communication’
  • Provide manageable paradigms for handling parallelism
The Future of Computation
What is a (the) Future?

• Many ways to get hold of a (the) 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
}
```
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
- Represents a data-dependency
- Makes asynchrony manageable
- Allows for composition of several asynchronous operations
- (Turns concurrency into parallelism)
Ways to Create a future

• Standard defines 3 possible ways to create a future,
  • 3 different ‘asynchronous providers’
    • std::async
    • std::packaged_task
    • std::promise
Packaging a Future

- std::packaged_task is a function object
  - It gives away a future representing the result of its invocation

- Can be used as a synchronization primitive
  - Pass to std::thread

- Converting a callback into a future
  - Observer pattern, allows to wait for a callback to happen
template<typename F, typename ...Arg>
std::future<typename std::result_of<F(Arg...)>::type>
simple_async(F func, Arg&&... arg)
{
    std::packaged_task<F> pt(func);
    auto f = pt.get_future();

    std::thread t(std::move(pt), std::forward<Arg>(arg)...);
    t.detach(); // detach the thread from this object

    return f;
}
Promising a Future

- `std::promise` is also an asynchronous provider ("an object that provides a result to a shared state")
  - The promise is the thing that you set a result on, so that you can get it from the associated future.
  - The promise initially creates the shared state
  - The future created by the promise shares the state with it
  - The shared state stores the value
Promising a future

```cpp
template <typename F> class simple_packaged_task;

template <typename R, typename... Args>
class simple_packaged_task<R(Args...)> // must be move-only
{
    std::function<R(Args...)> fn;
    std::promise<R> p; // the promise for the result
    // ...

public:
    template <typename F>
    explicit simple_packaged_task(F && f) : fn(std::forward<F>(f)) {} 

    template <typename ...T>
    void operator()(T&&... t) { p.set_value(fn(std::forward<T>(t)...)); } 

    std::future<R> get_future() { return p.get_future(); } 
};
```
Extending std::future
Extending std::future

• Several proposals for next C++ Standard, also HPX
  • Extension for std::future
    • Compositional facilities
      • Parallel composition
      • Sequential composition
  • Parallel Algorithms
  • Parallel Task Regions
  • Extended async semantics: dataflow
Make a ready Future

• Create a future which is ready at construction (N3857)

```cpp
future<int> compute(int x)
{
    if (x < 0) return make_ready_future(-1);
    if (x == 0) return make_ready_future(0);

    return async(
        [](int param) { return do_work(param); },
        x);
}
```
Compositional facilities

- Sequential composition of futures

```cpp
string make_string()
{
    future<int> f1 = async([]() { return 123; });
    future<string> f2 = f1.then(
        [] (future<int> f) { return to_string(f.get()); } // here .get() won’t block
    );
}
```
Compositional facilities

- Parallel composition of futures

```c++
void test_when_all()
{
    shared_future<int> shared_future1 = async([]() -> int { return 125; });
    future<string> future2 = async([]() -> string { return string("hi"); });

    future<tuple<shared_future<int>, future<string>>> all_f =
        when_all(shared_future1, future2); // also: when_any, when_some, etc.

    future<int> result = all_f.then(
        [](auto f) -> int {
            return do_work(f.get());
        });
}
```
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>
Parallel Algorithms (C++17)

- Add Execution Policy as first argument

- Execution policies have associated default executor and default executor parameters
  - `execution::parallel_policy`, generated with `par`
    - parallel executor, static chunk size
  - `execution::sequenced_policy`, generated with `seq`
    - sequential executor, no chunking

```cpp
// add execution policy
std::fill(
  std::execution::par,
  begin(d), end(d), 0.0);
```
// uses default executor: par
std::vector<double> d = { ... };
fill(execution::par, begin(d), end(d), 0.0);

// rebind par to user-defined executor (where and how to execute)
my_executor my_exec = ...;
fill(execution::par.on(my_exec), begin(d), end(d), 0.0);

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

- Extensions: asynchronous execution policies

  - `parallel_task_execution_policy` (asynchronous version of `parallel_execution_policy`), generated with `par(task)`
  - `sequenced_task_execution_policy` (asynchronous version of `sequenced_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