HIGH-PERFORMANCE COMPUTING

Highly Parallel Program

CPU

GPU

High Performance Execution
How might programmers write highly parallel programs in a mainstream language like C++?
BEGIN AT THE BEGINNING
Independent loop iterations represent latent parallelism

```c
void saxpy(int n, float a, float *x, float *y)
{
    // Sequential code with latent parallelism
    for(int i=0; i<n; ++i)
    {
        y[i] = a*x[i] + y[i];
    }
}
```
PARALLEL LOOPS IN C++17
Library implementation of parallel constructs

```cpp
void saxpy(int n, float a, float *x, float *y)
{
    auto I = interval(0, n);

    std::for_each(std::par, I.begin(), I.end(), [=](int i)
    {
        y[i] = a*x[i] + y[i];
    });
}
```
## PARALLEL LOOPS
Increasingly common in standard languages

<table>
<thead>
<tr>
<th>Language</th>
<th>Code Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMP</td>
<td><code>#pragma omp parallel for</code> for(int i=a; i&lt;b; ++i) { ... }</td>
</tr>
<tr>
<td>OpenACC</td>
<td><code>#pragma acc loop</code> for(int i=a; i&lt;b; ++i) { ... }</td>
</tr>
<tr>
<td>Fortran 2008</td>
<td>DO CONCURRENT (I=1:N) ... END DO</td>
</tr>
<tr>
<td>C++17</td>
<td><code>std::for_each(std::par, begin, end, [](int i) { ... });</code></td>
</tr>
</tbody>
</table>
STANDARD TEMPLATE LIBRARY
Higher-level library built around algorithms

for_each(begin, end, function);

Operator
A named pattern of computation and communication.

Data
One or more collections to operate on.

Function
Caller-provided function object injected in pattern.
C++17 PARALLEL STL
Algorithms + Execution Policies

`for_each(par, begin, end, function);`

Specify *how* operation may execute.
**EXECUTION POLICIES**
Specify how algorithms may execute

<table>
<thead>
<tr>
<th>POLICY NAME</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq</td>
<td>Sequential execution alone is permitted.</td>
</tr>
<tr>
<td>par</td>
<td>Parallel execution is permitted.</td>
</tr>
<tr>
<td>par_vec</td>
<td>Vectorized parallel execution is permitted.</td>
</tr>
</tbody>
</table>

*parallel*: provided function objects can be executed in any order on one or more threads.

*vectorized*: provided function objects can be also be interleaved when on one thread.
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
IMPLEMENTING ALGORITHMS
Requires suitable lower level constructs

template<class Policy, class Range, class T, class Op>
T reduce(Policy&, Range&& data, T init, Op op)
{
    // organize data into partitions
    auto partitions = make_partitioned_view(data, policy.executor().shape());

    // reduce each partition
    auto partial_sums = bulk_invoke(policy, [=](auto& self)
    {
        return reduce(par.on(self.inner()), partitions[self.index()], op);
    });

    // reduce the partial sums
    return reduce(policy.inner(), partial_sums, init, op);
}
IMPLEMENTING ALGORITHMS

But waiting can be harmful

template<class Policy, class Range, class T, class Op>
T reduce(Policy&, Range&& data, T init, Op op)
{
    // organize data into partitions
    auto partitions = make_partitioned_view(data, policy.executor().shape);

    // reduce each partition
    auto partial_sums = bulk_invoke(policy, [=](auto& self)
    {
        return reduce(par.on(self.inner()), partitions[self.index()], op);
    });

    // reduce the partial sums
    return reduce(policy.inner(), partial_sums, init, op);
}
template<class Policy, class Range, class T, class Op>
future<T> async_reduce(Policy&, Range&& data, T init, Op op)
{
    // organize data into partitions
    auto partitions = make_partitioned_view(data, policy.executor().shape());

    // reduce each partition
    auto partial_sums = bulk_async(policy, [=](auto& self) {
        return reduce(par.on(self.inner()), partitions[self.index()], op);
    });

    // reduce the partial sums
    return partial_sums.then([=](auto& partial_sums) {
        reduce(policy.inner(), partial_sums, init, op);
    });
}
A CAUTIONARY TALE
How much actual parallelism does this loop generate?

```c
void saxpy(int n, float a, float *x, float *y)
{
    for(int i=0; i<n; ++i)
    {
        async([=]{ y[i] = a*x[i] + y[i]; });
    }
}
```
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(...)  

Uniform Abstraction

Executors

Diverse Execution Resources

Operating System Threads  Thread pool schedulers  OpenMP runtime
SIMD vector units  GPU runtime  Fibers
EXECUTOR FRAMEWORK
Abstract platform details of execution

Create execution agents

Manage data they share

Advertise semantics

Mediate dependencies

See [http://wg21.link/p0058r1](http://wg21.link/p0058r1) for details.
EXECUTORS
Provide control over where/how execution happens

Placement is, by default, at discretion of the system.

```cpp
for_each(par, I.begin(), I.end(), [](int i) { y[i] += a*x[i]; });
```

In some cases, the programmer might want to control placement.

```cpp
auto place1 = choose_some_place();
auto place2 = choose_another_place();

for_each(par.on(place1), I.begin(), I.end(), ...);
for_each(par.on(place2), I.begin(), I.end(), ...);
```
DEFERRED EXECUTION
Providing opportunities for analysis and optimization

Operations Specified in Program Order

[Diagram with nodes labeled Op, Hardware Execution Wavefront, Application Execution Wavefront, and Dynamic Analysis Window]
BENEFITS OF DEFERRAL

Programming system can do more for the programmer

Optimize mapping of work and data

Optimization of activities (e.g., fusion)

Resilience using data provenance (cf. Spark RDDs)

PENNANT mini-app task graph
one time step on one node
DEFERRED EXECUTION: LEGION

Weak scaling results for S3D on ORNL Titan

Collaboration amongst:

- Legion
- MPI + OpenACC
DEFERRED EXECUTION: APACHE SPARK
Resilient Distributed Datasets (RDDs)

```python
sc = SparkContext(...)  

def sample(p):
    x, y = random(), random()
    return 1 if x*x + y*y < 1 else 0

count = sc.parallelize(xrange(0, NUM_SAMPLES)) \  
    .map(sample) \  
    .reduce(lambda a, b: a + b)

print "Pi is roughly \%f\% (4.0 * count / NUM_SAMPLES)"
```

Example from http://spark.apache.org/examples.html
# Create a Constant op that produces a 1x2 matrix. The op is added as a node to the default graph.
# The value returned by the constructor represents the output of the Constant op.
matrix1 = tf.constant([[3., 3.]], name="A")

# Create another Constant that produces a 2x1 matrix.
matrix2 = tf.constant([[2.], [2.]], name="B")

# Create a Matmul op that takes 'matrix1' and 'matrix2' as inputs. The returned value, 'product', represents the result of the matrix multiplication.
product = tf.matmul(matrix1, matrix2, name="Product")

# Execute the computation and print the result with tf.Session():
print product.eval()
Convolution Neural Network Graph

Abadi et al., 2015.

TensorFlow: Large-scale machine learning on heterogeneous distributed systems.
See tensorflow.org.
Parallel Control Structures
Capture common patterns of parallelism and asynchrony.

Executor Framework
Common low-level abstraction to support parallel control structures.

Deferred Execution
Enable runtime analysis and optimization of execution.