Parallelizing the Standard Algorithms Library

Jared Hoberock
NVIDIA
Programming Systems and Applications Research Group

CppCon
September, 2014

Bringing Parallelism to C++

Technical Specification for Parallel Algorithms

Multi-vendor collaboration

Based on proven technologies

- Thrust (NVIDIA)
- PPL (Microsoft)
- TBB (Intel)

Multiple implementations in progress

Targeting C++17

Roadmap

Parallelism?

Motivating example

What’s included in the box

The details

Future work

What do I mean by parallelism?

That’s like threads, right?

When I say “parallel”, think “independent”

- Concurrency is an optimization
- Concurrency can be hard
- locking, exclusion, communication, shared state, data races...

What do I mean by parallelism?

Parallel programming => identifying tasks which may be performed independently

How to communicate this information to the system?

It’s easy!

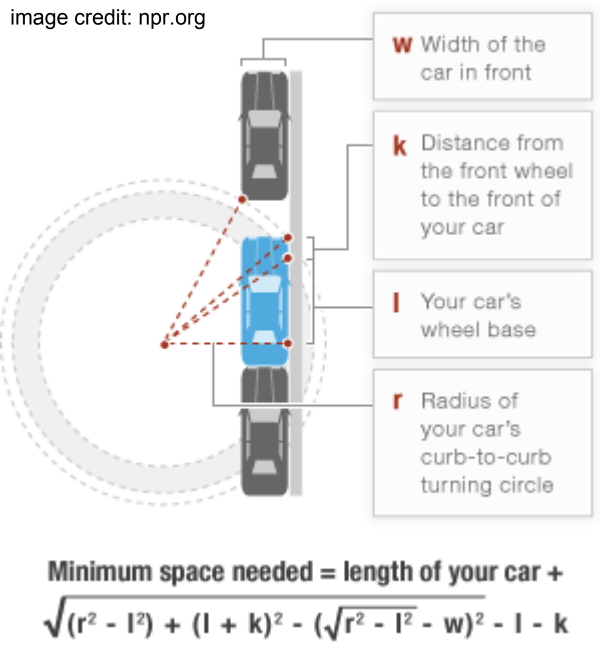


image credit: npr.org

w Width of the car in front
k Distance from the front wheel to the front of your car
l Your car's wheel base
r Radius of your car's curb-to-curb turning circle
Minimum space needed = length of your car + $\sqrt{(r^2 - l^2) + (l + k)^2 - (\sqrt{r^2 - l^2} - w)^2} - l - k$

Simple parallelism for everyone

Easy to access

Interoperability with existing codes

Supported as broadly as possible

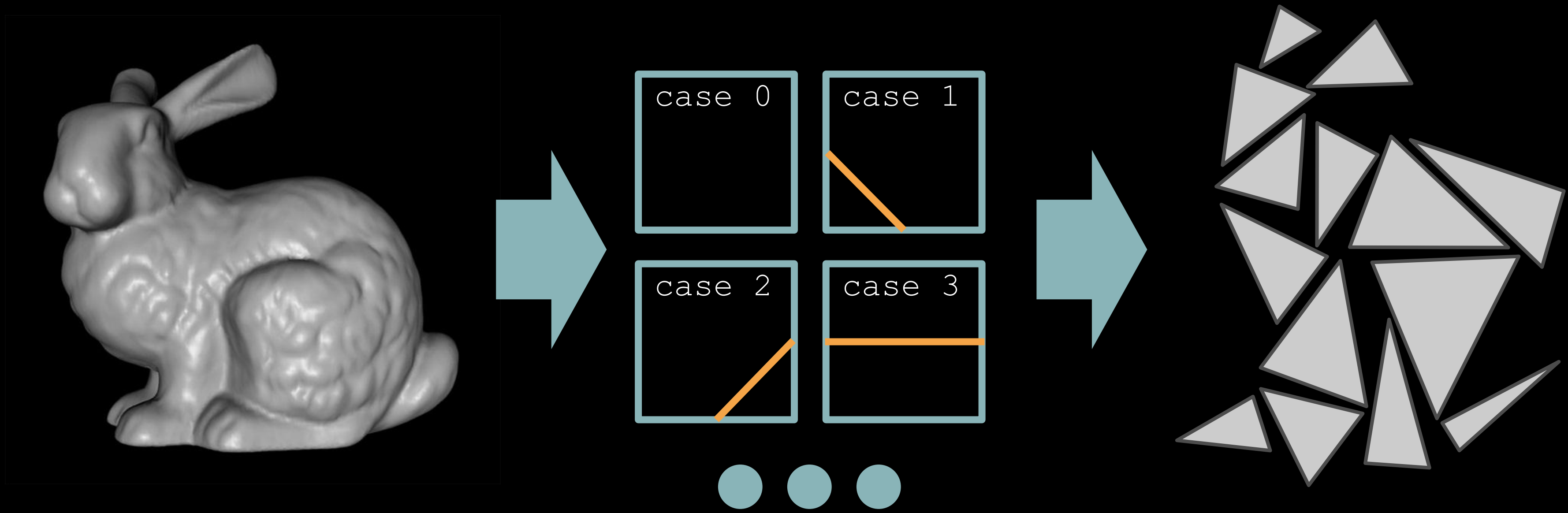
Concurrency is an invisible optimization

Vendor extensible

Motivating Example

Motivating Example: Weld Vertices

Marching Cubes Algorithm



case 0

case 1

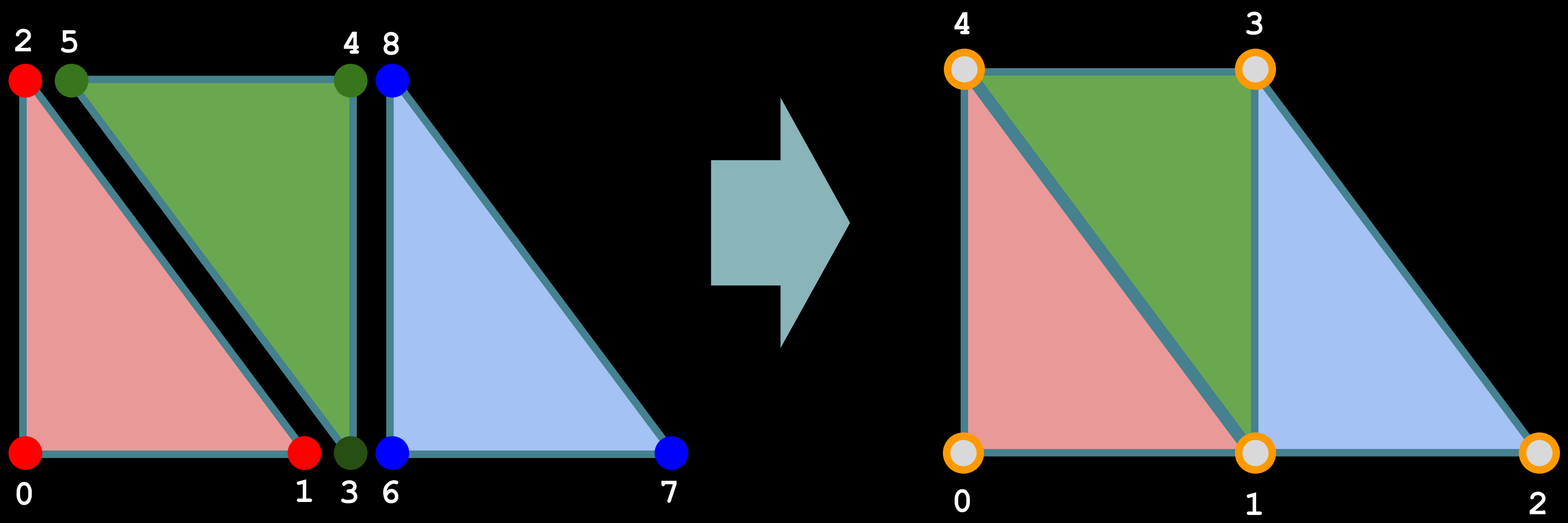
case 2

case 3

Motivating Example: Weld Vertices

Problem: Marching Cubes produces “triangle soup”

Solution: “Weld” redundant vertices together



Motivating Example: Weld Vertices

Easy with the right high-level algorithms

Procedure:

1. Sort triangle vertices
2. Collapse spans of like vertices
3. Search for each vertex’s unique index

Motivating Example: Weld Vertices

```
using namespace std;

using vertex = tuple<float,float>;
vector<vertex> vertices = input;
vector<size_t> indices(input.size());

// sort vertices to bring duplicates together
sort(vertices.begin(), vertices.end());

// find unique vertices and erase redundancies
auto redundant_begin = unique(vertices.begin(), vertices.end());
vertices.erase(redundant_begin, vertices.end());

// find index of each vertex in the list of unique vertices
my_find_all_lower_bounds(vertices.begin(), vertices.end(),
                         input.begin(), input.end(),
                         indices.begin());
```

Now do it in parallel?

Easy!

```
using namespace std;
using namespace std::experimental::parallel;

using vertex = tuple<float,float>;
vector<vertex> vertices = input;
vector<size_t> indices(input.size());

// sort vertices to bring duplicates together
sort(par, vertices.begin(), vertices.end());

// find unique vertices and erase redundancies
auto redundant_begin = unique(par, vertices.begin(), vertices.end());
vertices.erase(redundant_begin, vertices.end());

// find index of each vertex in the list of unique vertices
my_find_all_lower_bounds(par, vertices.begin(), vertices.end(),
                         input.begin(), input.end(),
                         indices.begin());
```

Wait, I changed my mind...

```
using namespace std;
using namespace std::experimental::parallel;

using vertex = tuple<float,float>;
vector<vertex> vertices = input;
vector<size_t> indices(input.size());

// sort vertices to bring duplicates together
sort(seq, vertices.begin(), vertices.end());

// find unique vertices and erase redundancies
auto redundant_begin = unique(seq, vertices.begin(), vertices.end());
vertices.erase(redundant_begin, vertices.end());

// find index of each vertex in the list of unique vertices
my_find_all_lower_bounds(seq, vertices.begin(), vertices.end(),
                         input.begin(), input.end(),
                         indices.begin());
```

Don’t make me choose!

```
using namespace std;
using namespace std::experimental::parallel;

...

execution_policy exec = seq;
if(input.size() > some_threshold) exec = par;

// sort vertices to bring duplicates together
sort(exec, vertices.begin(), vertices.end());

// find unique vertices and erase redundancies
auto redundant_begin = unique(exec, vertices.begin(), vertices.end());
vertices.erase(redundant_begin, vertices.end());

// find index of each vertex in the list of unique vertices
my_find_all_lower_bounds(exec, vertices.begin(), vertices.end(),
                         input.begin(), input.end(),
                         indices.begin());
```

my_find_all_lower_bounds

```
template<class ExecutionPolicy,
         class ForwardIterator,
         class InputIterator,
         class OutputIterator>
OutputIterator my_find_all_lower_bounds(ExecutionPolicy& exec,
                                        ForwardIterator haystack_begin,
                                        ForwardIterator haystack_end,
                                        InputIterator needles_begin,
                                        InputIterator needles_end,
                                        OutputIterator result)
{
  return transform(exec, needles_begin, needles_end, result,
    [=](auto& needle)
  {
    auto iter = std::lower_bound(haystack_begin, haystack_end, needle);
    return std::distance(haystack_begin, iter);
  });
}
```

`my_find_all_lower_bounds`

Truly general

- generic in data types (via iterators)
- generic in execution (via execution policy)

Composing our higher-level algorithms from lower-level primitives gives us parallelism for free!

How to write parallel programs

High-level algorithms

Control sequential/parallel execution with policies

Communicate dependencies

What’s included in the box

Execution Policies

Parallel Algorithms

Parallel Exceptions

Execution Policies

```
using namespace std::experimental::parallelism;
std::vector<int> data = ...

// vanilla sort
sort(data.begin(), data.end());

// explicitly sequential sort
sort(seq, data.begin(), data.end());

// permitting parallel execution
sort(par, data.begin(), data.end());

// permitting vectorization as well
sort(par_vec, data.begin(), data.end());
```

Execution Policies

```
// sort with dynamically-selected execution
size_t threshold = ...
execution_policy = seq;
if(vec.size() > threshold)
{
  exec = par;
}

sort(exec, vec.begin(), vec.end());
```

Execution Policies

```
sort(vectorize_in_this_thread, vec.begin(), vec.end());

sort(submit_to_my_thread_pool, vec.begin(), vec.end());

sort(execute_on_that_gpu, vec.begin(), vec.end());

sort(offload_to_my_fpga, vec.begin(), vec.end());

sort(send_to_the_cloud, vec.begin(), vec.end());

sort(launder_through_botnet, vec.begin(), vec.end());
```

Implementation-Defined
(Non-Standard)

What is an execution policy?

Promise that a particular kind of reordering will preserve the meaning of a program

Request that the implementation use some sort of execution strategy

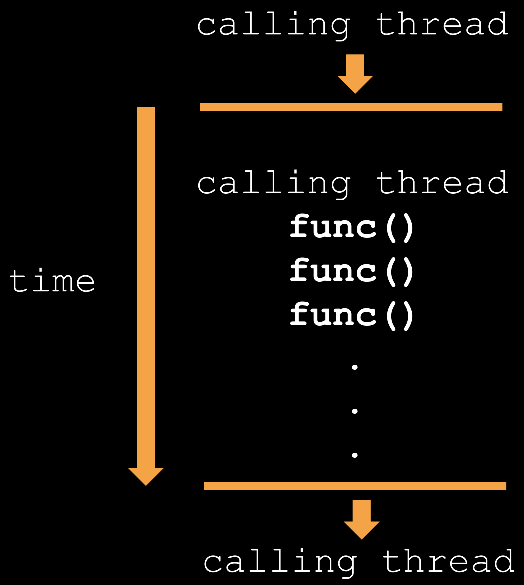
What sorts of reordering are allowable?

The Details

Sequential Execution

```
algo(seq, begin, end, func);
```

`algo` invokes function and iterator operations in sequential order in the calling thread



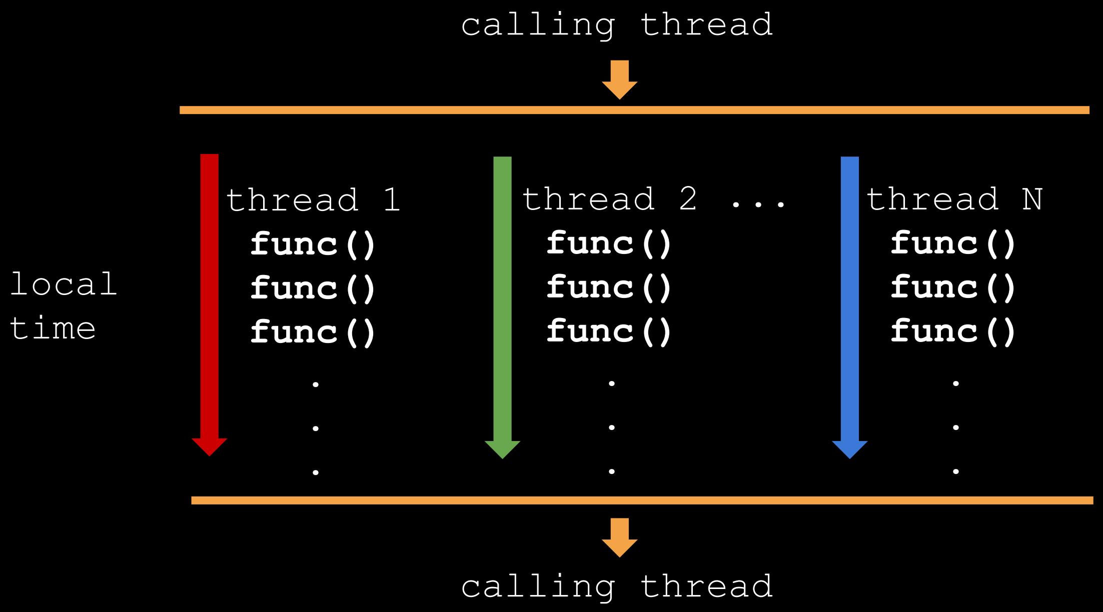
Parallelizable Execution

```
algo(par, begin, end, func);
```

`algo` is permitted to invoke function unsequenced if invoked in different threads, and indeterminate order if invoked in the same thread

Parallelizable Execution

```
algo(par, begin, end, func);
```



calling thread

local time

thread 1

```
func()
func()
func()
.
.
.
```

thread 2 ...

```
func()
func()
func()
.
.
.
```

thread N

```
func()
func()
func()
.
.
.
```

calling thread

Parallelizable Execution

It is the caller’s responsibility to ensure correctness, for example that the invocation does not introduce data races or deadlocks.

Parallelizable Execution

```
// data race
int a[] = {0,1};
std::vector<int> v;
for_each(par, a, a + 2, [&](int& i)
{
  v.push_back(i);
});
```

Parallelizable Execution

```
// OK (don't do this):
int a[] = {0,1};
std::vector<int> v;

std::mutex mut;
for_each(par, a, a + 2, [&](int& i)
{
  mut.lock();
  v.push_back(i);
  mut.unlock();
});
```

Parallelizable Execution

```
// OK (do this):
int a[] = {0,1};
std::vector<int> v(2);

for_each(par, a, a + 2, [&](int& i)
{
  v[i] = i;
});
```

Parallelizable Execution

```
// may deadlock (don't do this):
std::atomic<int> counter = 0;
int a[] = {0,1};

for_each(par, a, a + 2, [&](int& i)
{
  counter++;

  // spin wait for both lambdas to arrive
  while(counter.load() != 2)
  {
    ;
  }

  // try to do something crazy
  ...
});
```

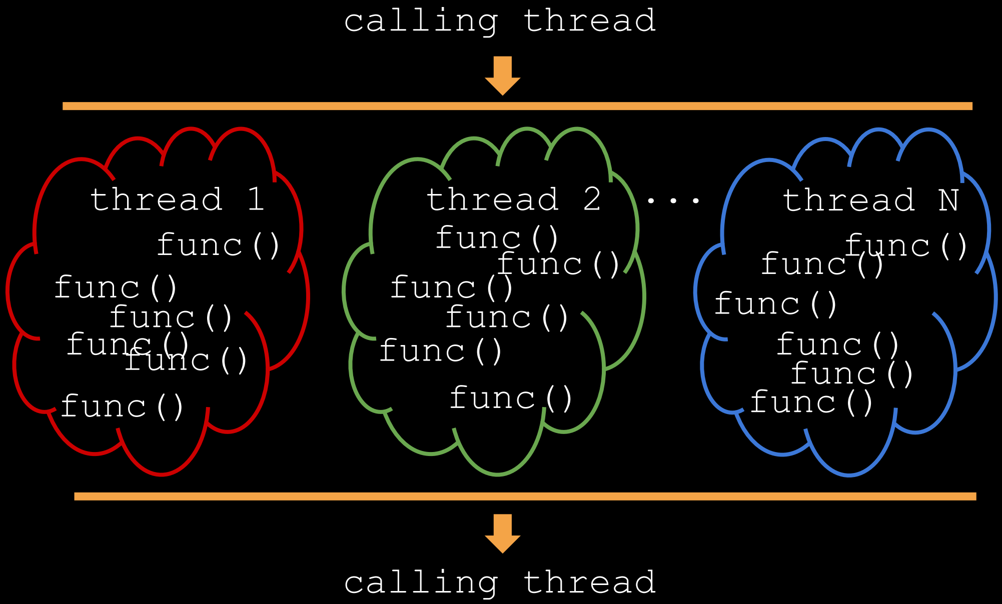
Parallelizable + Vectorizable Execution

```
algo(par_vec, begin, end, func);
```

`algo` is permitted to invoke function unsequenced if invoked in different threads, and unsequenced if invoked in the same thread

Parallelizable + Vectorizable Execution

```
algo(par_vec, begin, end, func);
```



Difference between `par` & `par_vec`

Function invocation is unsequenced when in different threads

When executed in the same thread

- **`par`**: unspecified, sequenced invocations
- **`par_vec`**: no sequence exists at all

Parallelizable + Vectorizable Execution

It is the caller’s responsibility to ensure correctness, for example that the invocation of functions do not attempt to synchronize with each other.

Parallelizable + Vectorizable Execution

```
// may deadlock (don't do this):
int counter = 0;
int a[] = {0,1};

std::mutex m;
for_each(par_vec, a, a + 2, [&](int)
{
  mut.lock();
  ++counter;
  mut.unlock();
});
```

Parallelizable + Vectorizable Execution

```
// OK:
std::atomic<int> counter = 0;
int a[] = {0,1};
for_each(par_vec, a, a + 2, [&](int)
{
  ++counter;
});
```

Parallelizable + Vectorizable Execution

```
// Best:
int count = count_if(par_vec, ...);
```

Use the highest-level algorithm that does the job!

Parallel Algorithms

Provide overloads of standard algorithm which receive execution policy as parameter

To the degree possible, parallelize everything

New Algorithms

`reduce` : reduction over a collection

```
result = init + a[0] + a[1] + … + a[N-1]
```

`exclusive_scan` : exclusive prefix sum

```
result[i] = init + a[0] + a[1] + … + a[i-1]
```

`inclusive_scan` : inclusive prefix sum

```
result[i] = init + a[0] + a[1] + … + a[i]
```

No Parallelism

Binary search algorithms

Heap algorithms

Permutation algorithms

Shuffling algorithms

Sequential numeric algorithms

Parallel Exceptions

Parallel algorithms throw one of two:

- If no temporary storage is available, `bad_alloc`

- `exception_list` of exceptions thrown by user-provided code

Exceptions Example

```
struct superstition_error { const char* what() { return "eek"; } };

std::vector<int> data = ...

try
{
  for_each(data.begin(), data.end(), [](auto x)
  {
    if(x == 13)
    {
      throw superstition_error();
    }
  });
}
catch(superstition_error& error)
{
  std::cerr << error.what() << std::endl;
}
```

Exceptions Example

```
struct superstition_error { const char* what() { return "eek"; } };

std::vector<int> data = ...

using namespace std::experimental::parallelism;

try
{
  for_each(par, data.begin(), data.end(), [](auto x)
  {
    if(x == 13)
    {
      throw superstition_error();
    }
  });
}
catch(exception_list& error)
{
  std::cerr << “Encountered “ << errors.size() << “ unlucky numbers” << std::endl;

  reduce(par, errors.begin(), errors.end(), my_handler());
}
```

Exceptions Example

```
struct superstition_error { const char* what() { return "eek"; } };

std::vector<int> data = ...

using namespace std::experimental::parallelism;

try
{
  for_each(seq, data.begin(), data.end(), [](auto x)
  {
    if(x == 13)
    {
      throw superstition_error();
    }
  });
}
catch(exception_list& error)
{
  std::cerr << “Encountered “ << errors.size() << “ unlucky numbers” << std::endl;

  reduce(seq, errors.begin(), errors.end(), my_handler());
}
```

Example: Estimating Pi

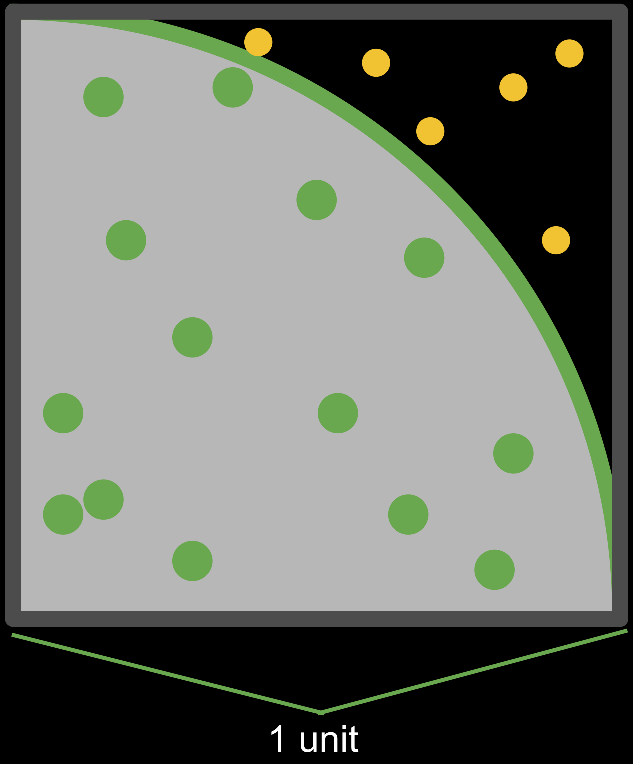
Example: Estimating Pi

Area of circle = pi * radius^2

Throw darts at unit square

pi ~ 4 * green / total

Algorithm: Count the number of darts which fall within quarter circle



Example: Estimating Pi

```
// generate a random point and test whether it
// lies in the quarter circle
bool test_quarter_circle(int seed)
{
  // seed an RNG
  std::default_random_engine rng(my_hash(seed));

  // generate numbers uniformly
  // in the unit interval
  std::uniform_real_distribution<float> u01(0,1);

  // generate a point within the unit square
  float x = u01(rng);
  float y = u01(rng);

  // measure distance from the origin
  float dist2 = std::sqrt(x*x + y*y);

  return dist2 <= 1.0f;
}
```

```
// burtleburtle.net/bob/hash/integer.html
int my_hash(int a)
{
  a = (a+0x7ed55d16) + (a<<12);
  a = (a^0xc761c23c) ^ (a>>19);
  a = (a+0x165667b1) + (a<<5);
  a = (a+0xd3a2646c) ^ (a<<9);
  a = (a+0xfd7046c5) + (a<<3);
  a = (a^0xb55a4f09) ^ (a>>16);
  return a;
}
```

Example: Estimating Pi

```
// throw 300M darts
auto n = 300 << 20;

// create the integers [0..n)
auto iter = boost::make_counting_iterator(0);

// count the number of points in the quarter circle
using namespace std::experimental::parallel;

auto num_within_quarter_circle =
  count_if(par, iter, iter + n, test_quarter_circle);

double pi_estimate = (4.0 * num_within_quarter_circle) / n;
```

Performance Portability

```
single CPU thread
Intel i7 860

time ./pi_seq
pi is approximately
3.14

real 0m5.097s
user 0m5.098s
sys  0m0.004s
```

```
8 CPU threads
OpenMP, Intel i7 860

time ./pi_par
pi is approximately
3.14

real 0m0.971s
user 0m7.565s
sys  0m0.015s
```

```
many GPU threads
CUDA, NVIDIA Tesla K20

time ./pi_gpu
pi is approximately
3.14

real 0m0.260s
user 0m0.047s
sys  0m0.188s
```

1x

5.25x

19.6x

Future Work

Scheduling?

```
algo(exec, begin, end, func);
```

`algo` needs to compose with scheduling decisions in the surrounding application

exec specifies how `algo` is allowed to execute

- specifies **what** work an implementation is allowed to create
- does not specify **where** the work should be executed

Placement is **orthogonal**

Scheduling?

```
algo(exec(sched), begin, end,
func);
```

We anticipate extending our execution policies to accept scheduling requirements as parameters

`sched` specifies **where** the work should be executed

Scheduling?

`sched` could be:

- hard requirement to execute vectorizable work in the current thread
- number of threads to use
- a thread pool to use
- an executor to use
- which GPU(s) to use

Implementations in progress

github.com/n3554

- based on Thrust

parallelstl.codeplex.com

- based on PPL?

github.com/t-lutz/ParallelSTL

- based on **`std::thread`**

Summary

High-level algorithms make parallelism easy

- Portable & Composable
- Concurrency is invisible

Standardization

- On track for C++17
- Experimental Tech Spec in the meantime
- github.com/cplusplus/parallelism-ts

Questions?

jhoberock@nvidia.com
github.com/jaredhoberock