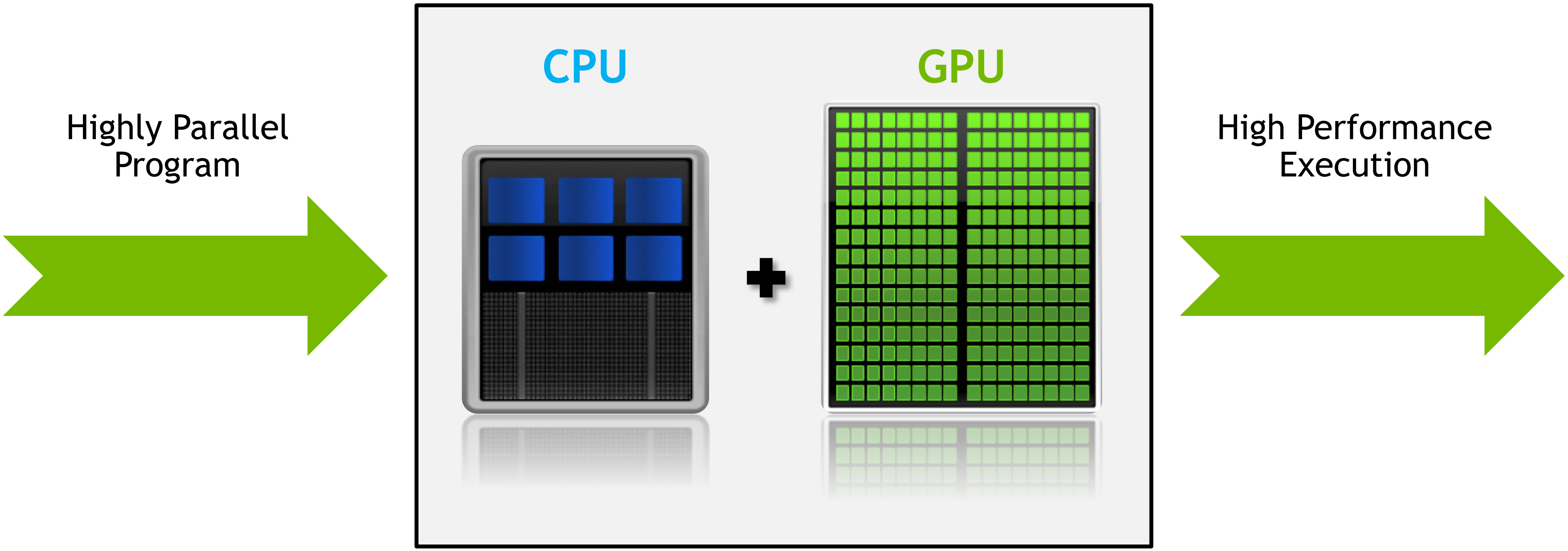
PARALLELISM BEYOND LOOPS

Michael Garland, March 2016
Programming Systems & Applications Research

HIGH-PERFORMANCE COMPUTING



RESEARCH QUESTION

How might programmers write highly parallel programs in a mainstream language like C++?

BEGIN AT THE BEGINNING

Independent loop iterations represent latent parallelism

```
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

```
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

OpenMP	`#pragma omp parallel for` `for(int i=a; i<b; ++i) { ... }`
OpenACC	`#pragma acc loop` `for(int i=a; i<b; ++i) { ... }`
Fortran 2008	`DO CONCURRENT (I=1:N) ... END DO`
C++17	`std::for_each(std::par, begin, end, [](int i) { ... });`

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);
```

↑

Execution Policy

Specify *how* operation may execute.

C++ Extensions for Parallelism

Draft technical specification: http://wg21.link/n4507

EXECUTION POLICIES

Specify how algorithms may execute

POLICY NAME	MEANING
`seq`	Sequential execution alone is permitted.
`par`	Parallel execution is permitted.
`par_vec`	Vectorized parallel execution is permitted.

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	New additions for parallelism
`for_each`	`reduce`
`transform`	`exclusive_scan`
`copy_if`	`inclusive_scan`
`sort`	`transform_reduce`
`set_intersection`	`transform_inclusive_scan`
etc.	`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);
    });
----------------------------------------------------------------------- wait

    // reduce the partial sums
    return reduce(policy.inner(), partial_sums, init, op);
----------------------------------------------------------------------- wait
}
```

ADDING ASYNCHRONY

Future-based machinery

```
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?

```
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(...)`

Multiplicative Explosion

✕

Diverse Execution Resources

Operating System Threads

Thread pool schedulers

OpenMP runtime

SIMD vector units

GPU runtime

Fibers

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

```
class sample_executor
{
public:
  using execution_category = ...;

  using shape_type = tuple<size_t,size_t>;

  template<class T>
  using future = ...;

  template<class T>
  future<T> make_ready_future(T&& value);

  template<class Container, class Function, class Future>
  future<Container> then_execute(Function f,
                                 shape_type shape,
                                 Future& dependency);
  ...
};
```

See http://wg21.link/p0058r1 for details.

EXECUTORS

Provide control over where/how execution happens

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

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

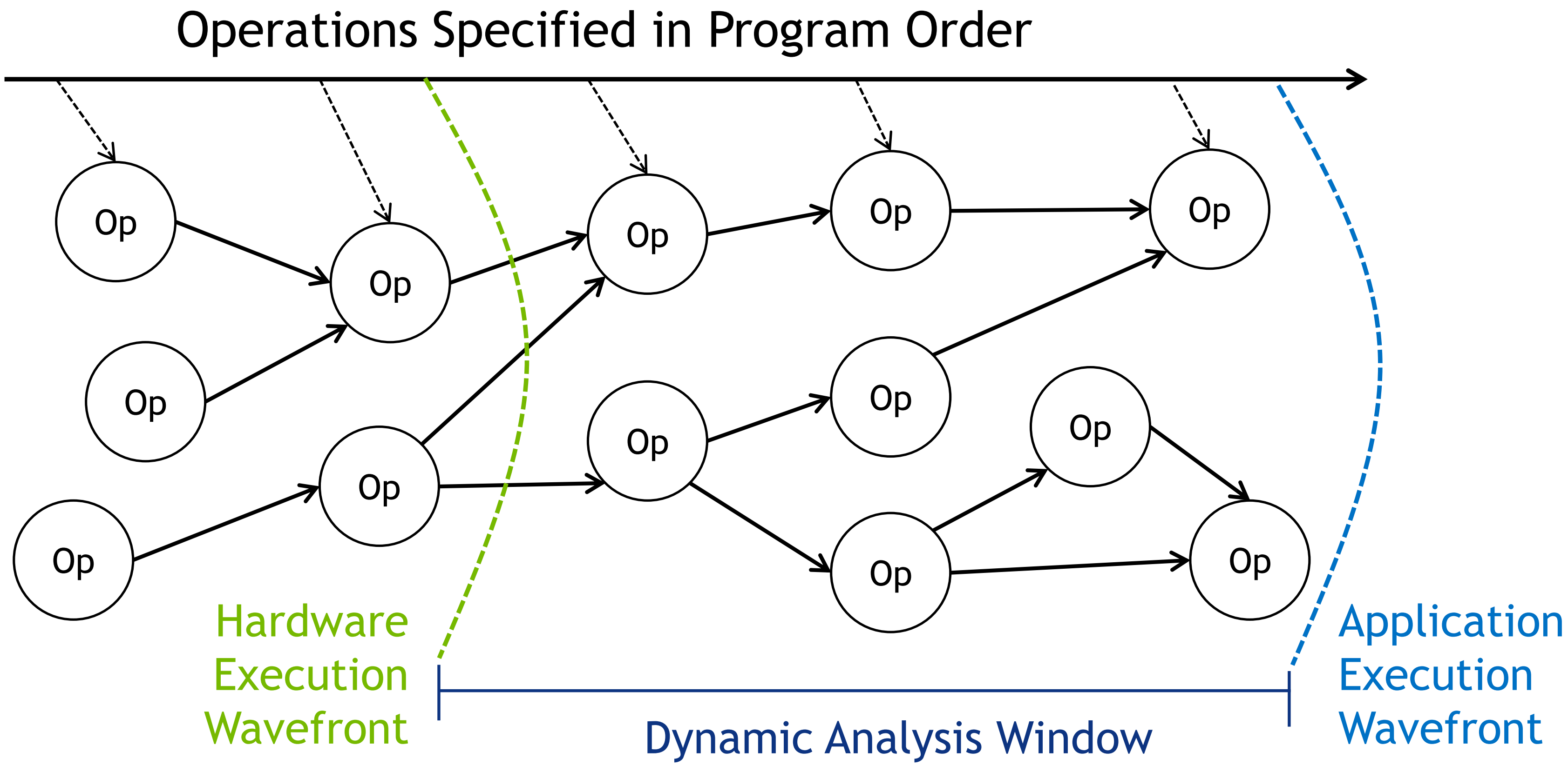
In some cases, the programmer might want to control placement.

```
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



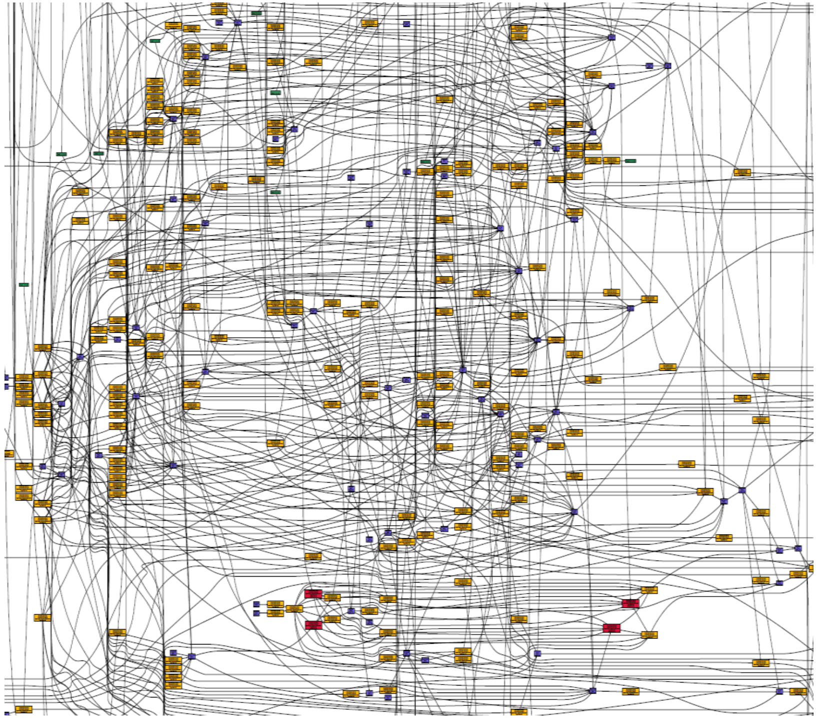
BENEFITS OF DEFERRAL

Programming system can do more for the programmer

Optimize mapping of work and data

Optimization of activities (e.g., fusion)

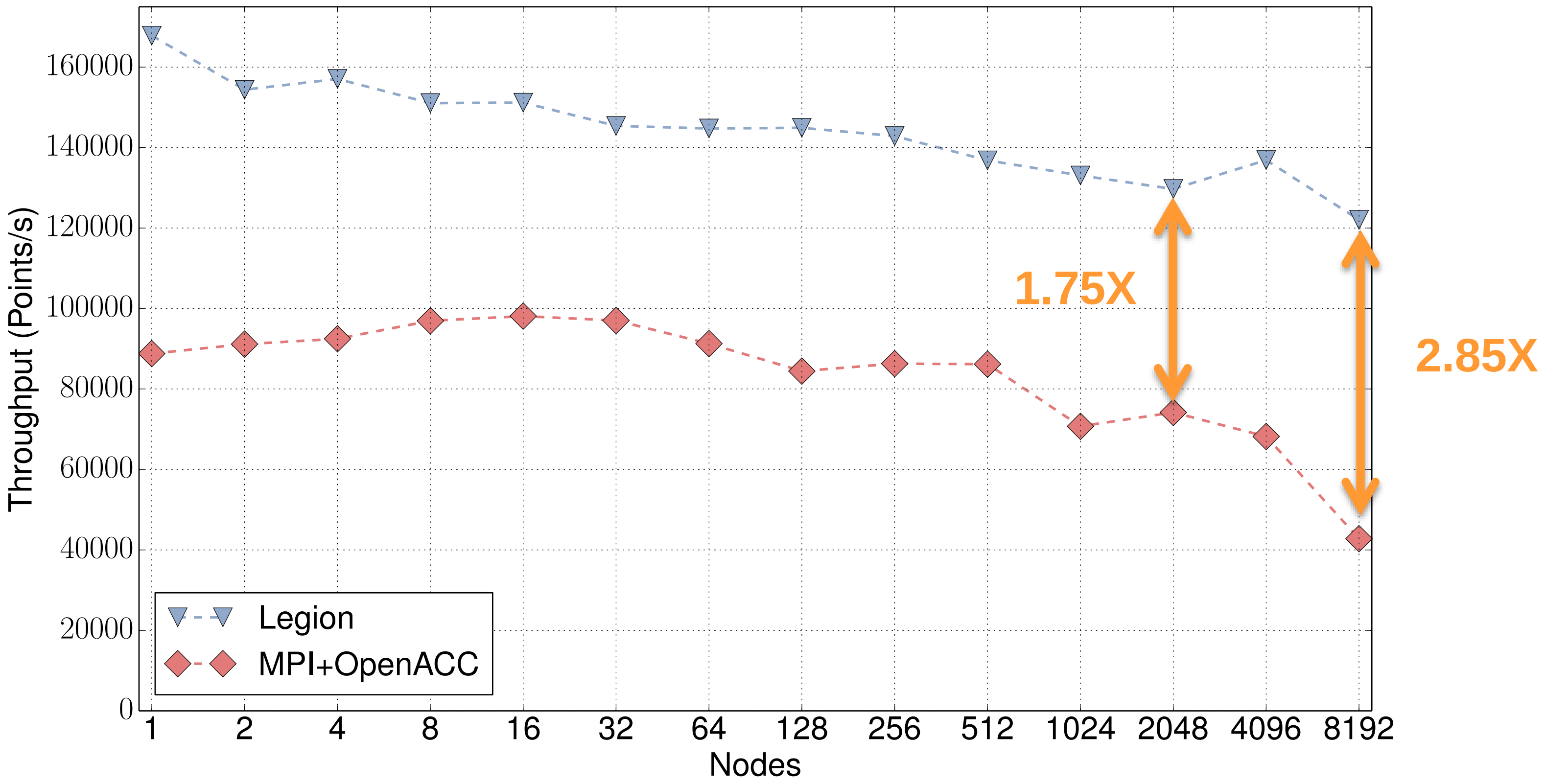
Resilience using data provenance (cf. Spark RDDs)

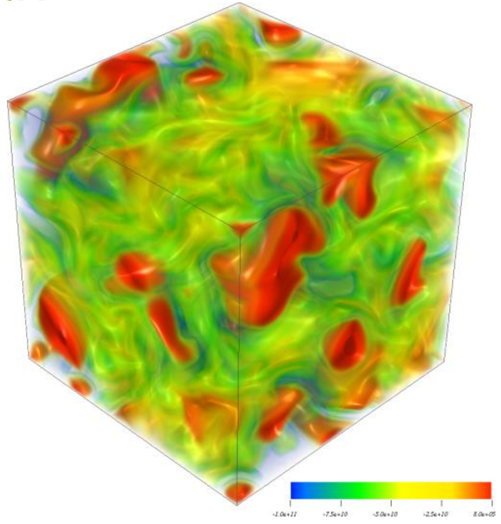


PENNANT mini-app task graph
one time step on one node

DEFERRED EXECUTION: LEGION

Weak scaling results for S3D on ORNL Titan





Collaboration amongst:

Stanford, NVIDIA, Los Alamos National Laboratory, Sandia National Laboratories

DEFERRED EXECUTION: APACHE SPARK

Resilient Distributed Datasets (RDDs)

```
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

DEFERRED EXECUTION: TENSORFLOW

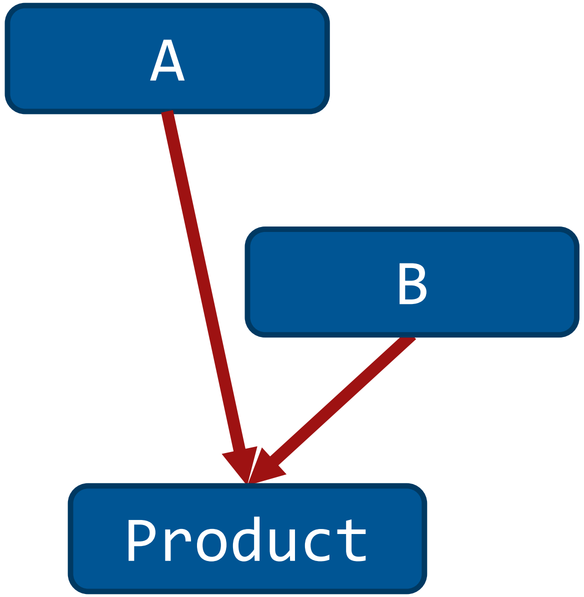
Constructing computation graphs for machine learning

```
# 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()
```



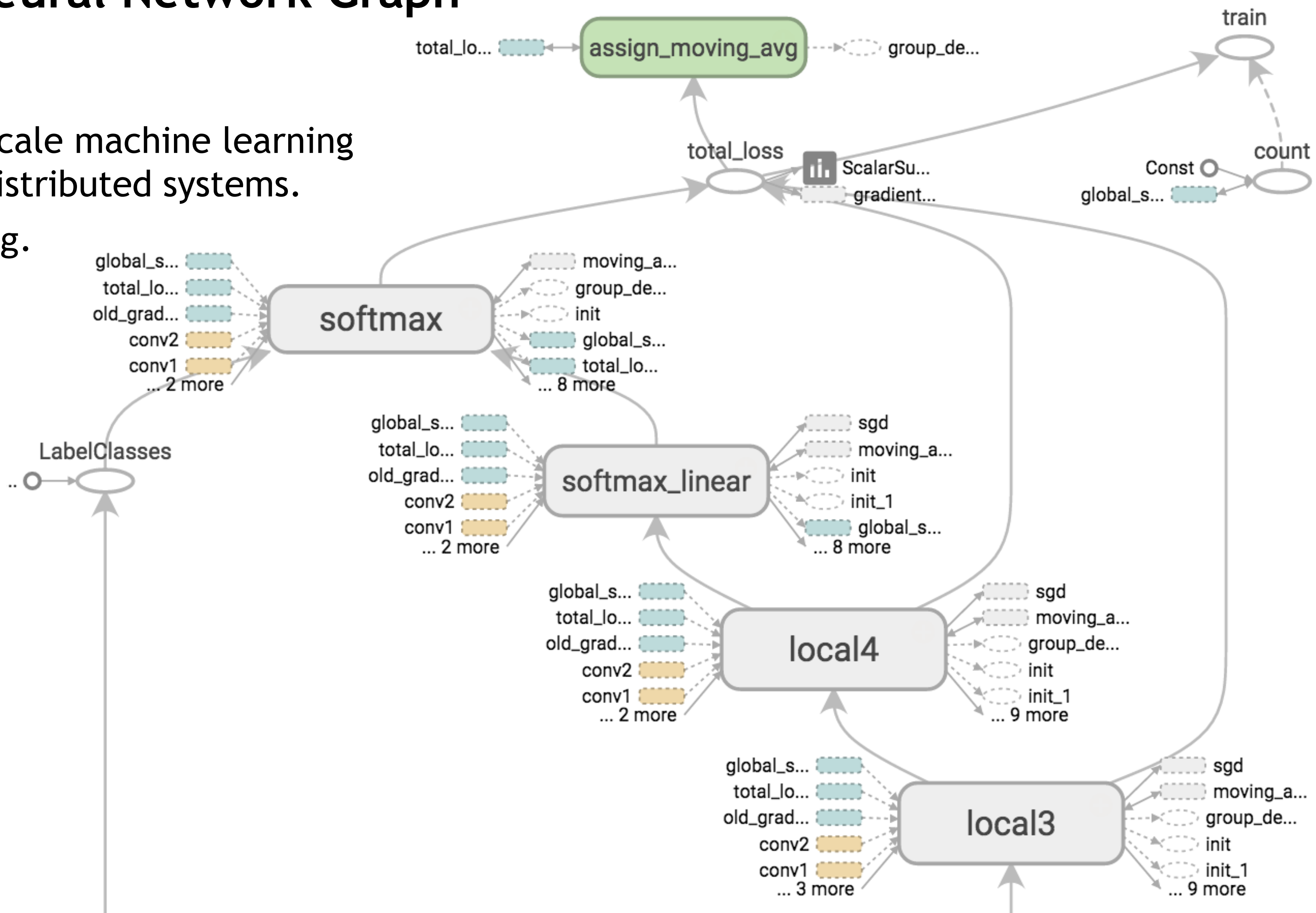
Original example: https://www.tensorflow.org/versions/r0.7/get_started/basic_usage.html

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.

NVIDIA