Distributed Ranges:
A Model for Building Distributed Data Structures, Algorithms, and Views

BENJAMIN BROCK

Cppcon
The C++ Conference
2023
October 01 - 06

Notices and Disclaimers

For notices, disclaimers, and details about performance claims, visit www.intel.com/PerformanceIndex or scan the QR code:

Human Readable **Disclaimer**

- The **views** in this talk are **mine**, not necessarily those of my employer.

- This is a **speculative**, **academic-style talk**. I’m not describing anything about future Intel products.

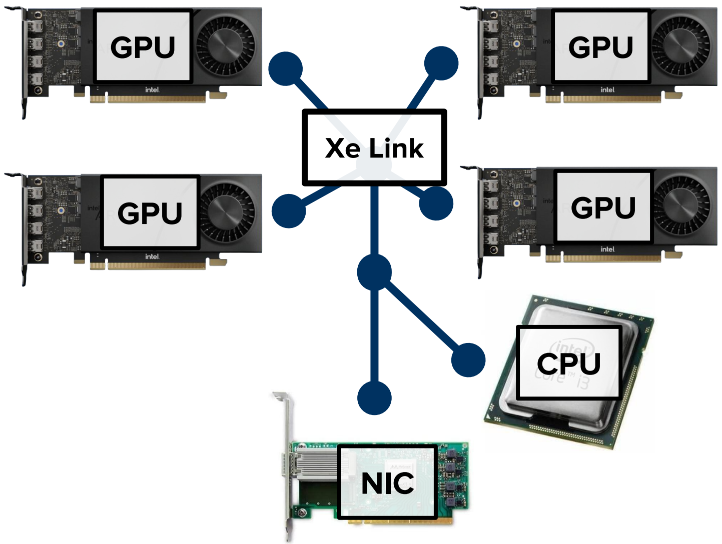
- I work in Intel’s **research labs**. Work described here will involve **experimental prototypes** and early research.

Problem: writing **parallel programs** is hard

- **Multi-GPU**, **multi-CPU** systems require **partitioning** data

- Users must **manually split up data** amongst GPUs / nodes

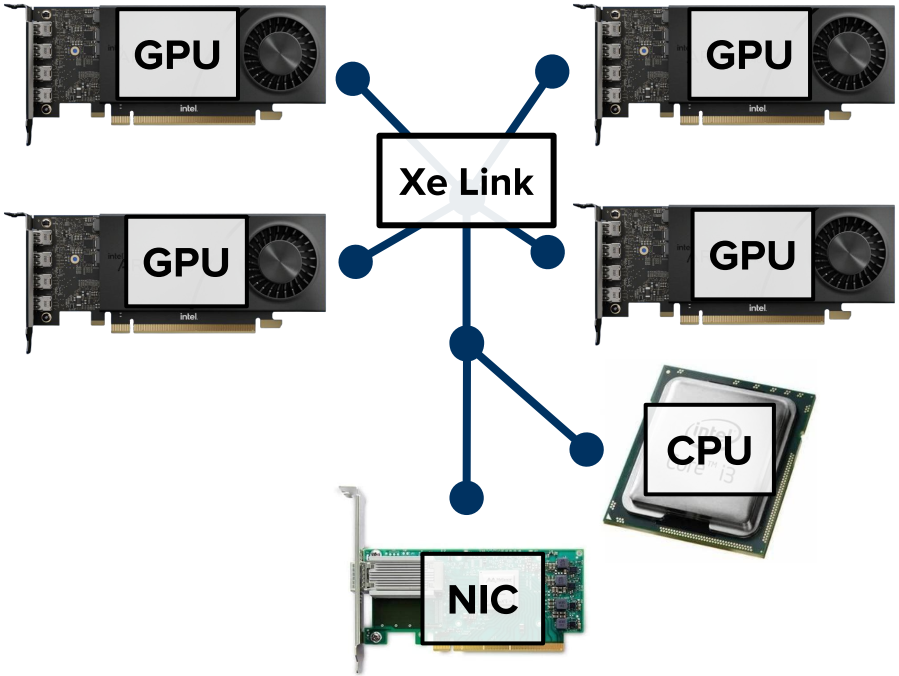
- High-level mechanisms for **data distribution** / **execution** necessary.



intel labs

Multi-GPU Systems

- NUMA regions:
  - **4+ GPUs**
  - **2+ CPUs**



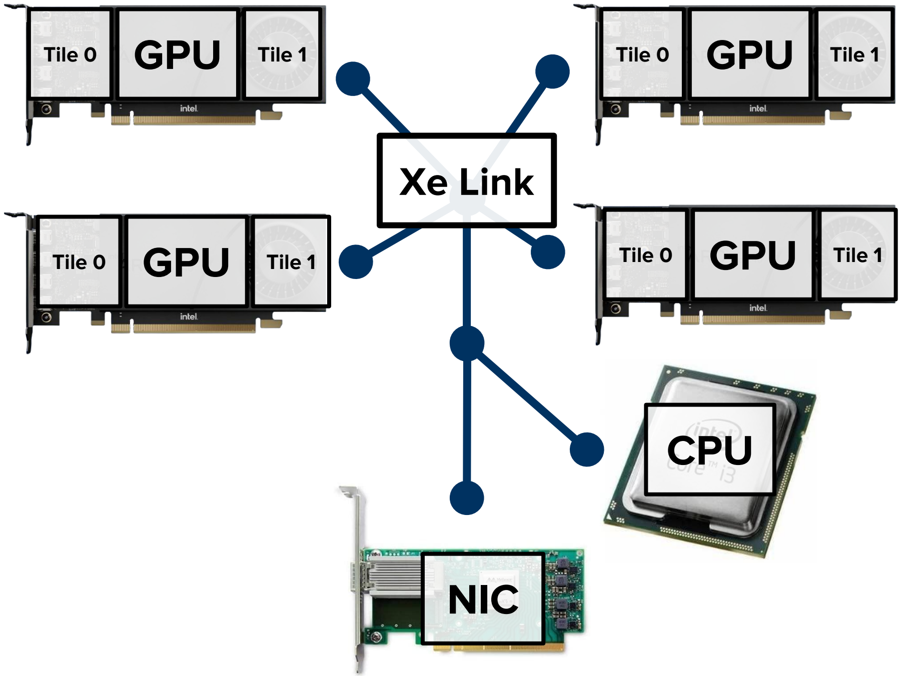
intel labs

Multi-GPU Systems

- NUMA regions:
  - **4+ GPUs**
  - **2+ CPUs**

- Systems becoming more **hierarchical**: even more **memory domains**

- Software needed to **reduce complexity**



intel labs

Project Goals

- Offer high-level, standard C++ **distributed data structures**
- Support **distributed algorithms**
- Achieve **high performance** for both **multi-GPU, NUMA,** and **multi-node** execution

```
float dot_product(vector<float>& x,
                  vector<float>& y) {

  auto z =  views::zip(x, y)
          | views::transform([](auto element) {
             auto [a, b] = element;
             return a * b;
            });

  return reduce(par_unseq, z, 0, std::plus());
}
```

Outline

- Background (Ranges, Parallelism, Distributed Data Structures)
- Distributed Ranges (Concepts)
- Implementation (Algorithms and views)
- Complex Data Structures (Dense and sparse matrices)
- Lessons learned

Outline

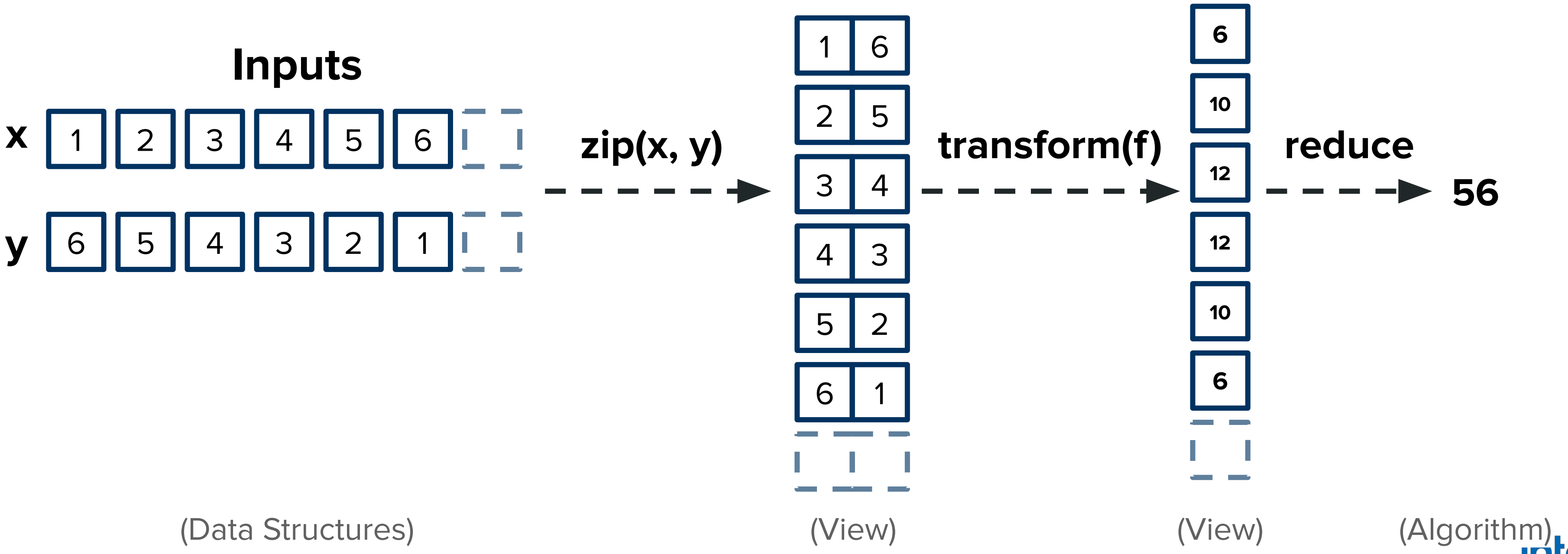
- **Background** (Ranges, Parallelism, Distributed Data Structures)
- Distributed Ranges (Concepts)
- Implementation (Algorithms and views)
- Complex Data Structures (Dense and sparse matrices)
- Lessons learned

Standard C++ Parallelism

- **Data structures**
  - Hold and organize data
- **Views**
  - Lightweight objects, views of data
- **Algorithms**
  - Operate on and modify data

```cpp
using namespace std;
using namespace std::ranges;
using namespace std::execution;

template <range R>
auto dot_product(R&& x, R&& y) {
  using T = range_value_t<R>;

  auto z =  views::zip(x, y)
          | views::transform([](auto element) {
             auto [a, b] = element;
             return a * b;
            });

  return reduce(par_unseq, z.begin(), z.end(),
                T(0), std::plus());
}
```

Dot Product Algorithm



Standard C++ Parallelism

- **Data structures**
  - Hold and organize data
- **Views**
  - Lightweight objects, views of data
- **Algorithms**
  - Operate on and modify data

```cpp
using namespace std;
using namespace std::ranges;
using namespace std::execution;

template <range R>
auto dot_product(R&& x, R&& y) {
  using T = range_value_t<R>;

  auto z =  views::zip(x, y)
          | views::transform([](auto element) {
             auto [a, b] = element;
             return a * b;
           });

  return reduce(par_unseq, z.begin(), z.end(),
                T(0), std::plus());
}
```

intel labs

Standard C++ Parallelism

- All depends on **ranges library**, iteration concepts

- **Extensible:** execution policies allow **parallel execution**

```
using namespace std;
using namespace std::ranges;
using namespace std::execution;

template <range R>
auto dot_product(R&& x, R&& y) {
  using T = range_value_t<R>;

  auto z =  views::zip(x, y)
          | views::transform([](auto element) {
             auto [a, b] = element;
             return a * b;
            });

  return reduce(par_unseq, z.begin(), z.end(),
                T(0), std::plus());
}
```

Standard C++ Parallelism

- All depends on **ranges library**, iteration concepts
- **Extensible:** execution policies allow **parallel execution**
- Standard allows **implementation-defined** execution policies

```
using namespace std;
using namespace std::ranges;
using namespace std::execution;
using namespace oneapi;

template <range R>
float dot_product(R&& x, R&& y) {
  using T = range_value_t<R>;

  auto z =  views::zip(x, y)
          | views::transform([](auto element) {
             auto [a, b] = element;
             return a * b;
           });

  auto policy = device_policy(/*...*/);

  return reduce(policy, z.begin(), z.end(),
                T(0), std::plus());
}
```

intel labs

Standard C++ Parallelism

- All depends on **ranges library**, iteration concepts

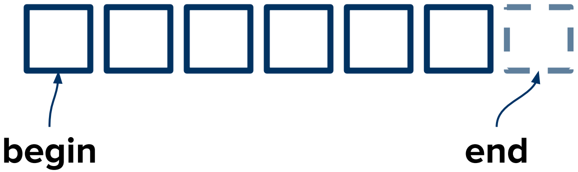
- **Extensible:** execution policies allow **parallel execution**

- Standard allows **implementation-defined** execution policies

```
using namespace std;
using namespace std::r
using namespace std::e
using namespace oneapi;

template <range R>
float dot_product(R&& x, R&& y) {
  using T = range_value_t<R>;

  auto z =  views::zip(x, y)
          | views::transform([](auto element) {
             auto [a, b] = element;
             return a * b;
           });

  auto policy = device_policy(/*...*/);

  return reduce(policy, z.begin(), z.end(),
                T(0), std::plus());
}
```

Question: where's the data?

Ranges Library



C++ 20 added the **ranges library**

A **range** is a **collection of values**

Range concepts provide a **standard way** to iterate over values

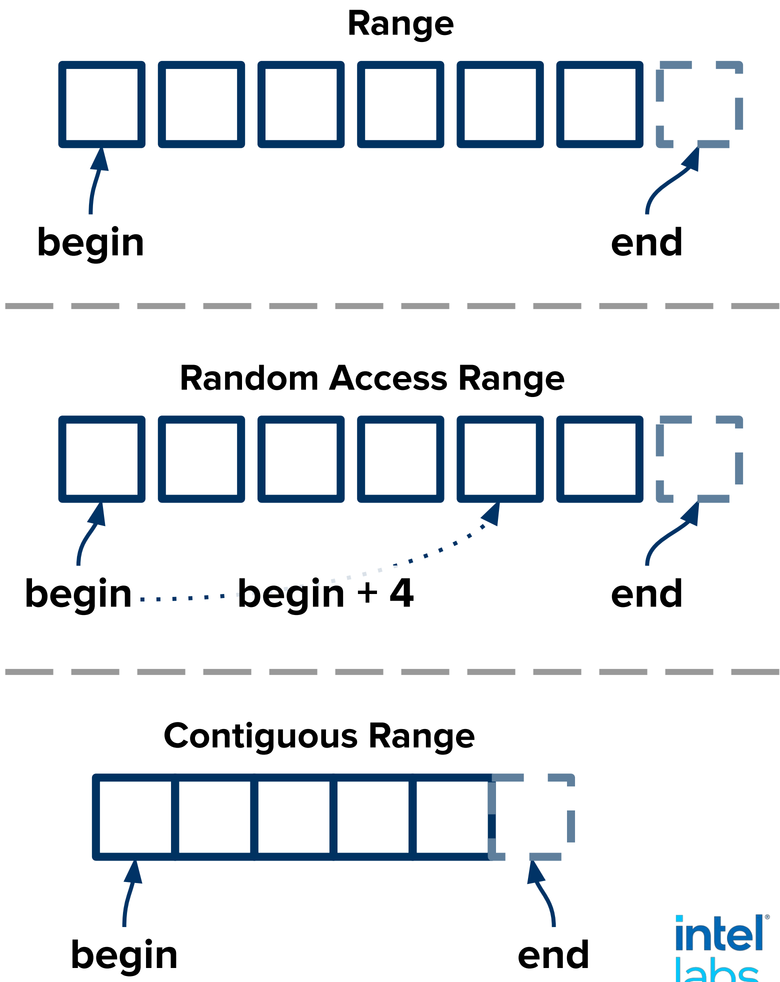
```
// Iteration
for (auto&& value : range) {
  printf("%d\n", value);
}
```

```
// Algorithms
auto r = std::ranges::reduce(range);
auto r = std::ranges::partial_sum(range);
```

```
// Views
auto add_two = [](auto v) { return v + 2; };
auto view =
     std::ranges::transform_view(range, add_two);
```

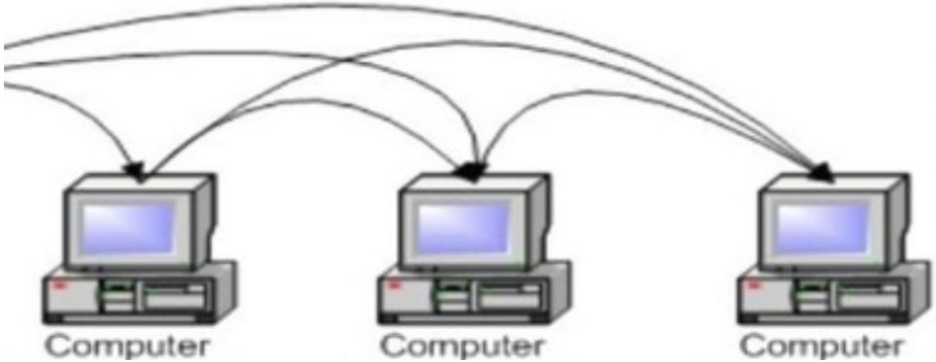
Ranges Library

- Have **`begin()`** and **`end()`**
- Often have **`size()`**
- **Random access**: access any element *at random* in **constant time**
- **Contiguous**: a **contiguous block of memory**



Distributed (Data Structures)

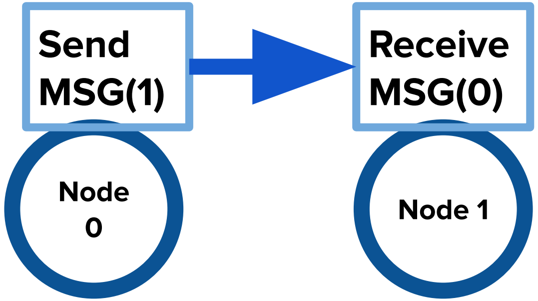
- A collection of **nodes**, connected by a **network**.



“PGAS in C++” CppCon’21

Communication Mechanisms

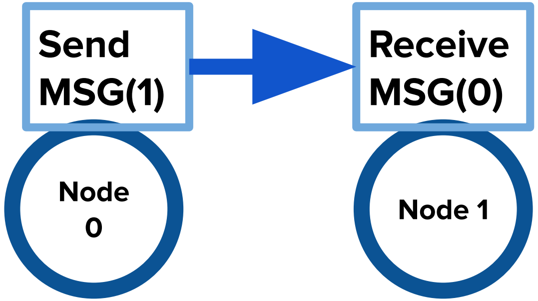
- **Message Passing** - processes issue matching **send** and **receive** calls



intel labs

Communication Mechanisms

- **Message Passing** - processes issue matching **send** and **r** … **s**



Process 0

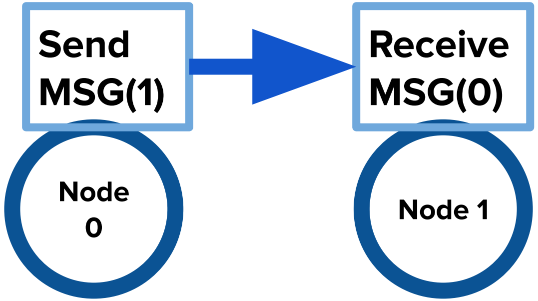
```
// Calculate data
auto values =
   algorithm(1.0f, 3,
             data);

// Send data to proc. 1
MPI_Send(values.data(),
         values.size(),
         MPI_FLOAT, 1,
         0, MPI_COMM_WORLD);

// Data is now sent.
```

Process 1

```
// Allocate space for data
std::vector<float>
recv_values(num_values);

// Receive data from proc. 0
MPI_Recv(recv_values.data(),
         num_values,
         MPI_FLOAT, 0,
         0, MPI_COMM_WORLD);

// Data is now in
// `recv_values`
```

Communication Mechanisms

- **Message Passing** - processes issue matching **send** and **r** s



Process 0

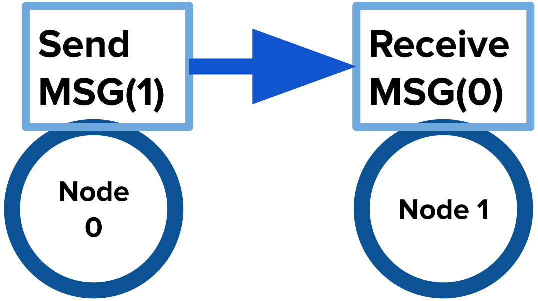
```
// Calculate data
auto values =
   algorithm(1.0f, 3,
             data);

// Send data to proc. 1
MPI_Send(values.data(),
         values.size(),
         MPI_FLOAT, 1,
         0, MPI_COMM_WORLD);

// Data is now sent.
```

Process 1

```
// Allocate space for data
std::vector<float>
recv_values(num_values);

// Receive data from proc. 0
MPI_Recv(recv_values.data(),
         num_values,
         MPI_FLOAT, 0,
         0, MPI_COMM_WORLD);

// Data is now in
// `recv_values`
```

Communication Mechanisms

- **Message Passing** - processes issue matching **send** and **r** s



Process 0

```
// Calculate data
auto values =
    algorithm(1.0f, 3.

// Send dat
MPI_Send(va
         values.size(),
         MPI_FLOAT, 1,
         0, MPI_COMM_WORLD);

// Data is now sent.
```

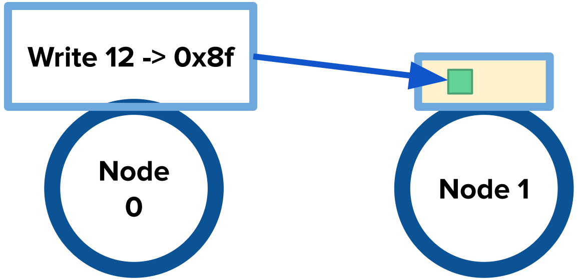
Process 1

```
// Allocate space for data
std::vector<float>
            s(num_values);

             data from proc. 0
            cv_values.data(),
            num_values,
            MPI_FLOAT, 0,
            0, MPI_COMM_WORLD);

// Data is now in
// `recv_values`
```

P0 and P1 must both participate in message.

Communication Mechanisms

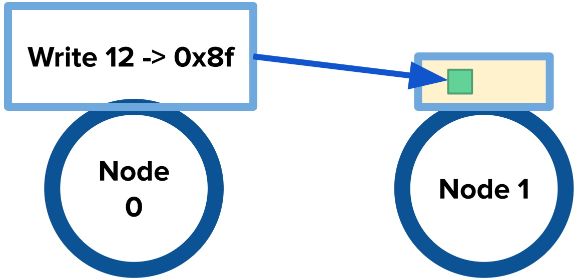
- **Message Passing** - processes issue matching **send** and **receive** calls
- **RDMA** - directly read/write to **remote memory**



Communication Mechanisms

- **Message Passing** - processes issue matching **send** and **receive** calls

- **RDMA** - directly read/write to



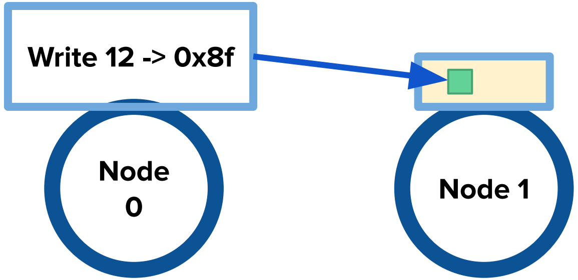
Process 0

```
auto remote_ptr = ...;
// Calculate data
auto values = algorithm(1.0f, 3, data);

// Send data to proc. 1
memcpy(remote_ptr, values.data(),
       values.size()*sizeof(float));

flush();

// Data is copied.
```

Communication Mechanisms

- **Message Passing** - processes issue matching **send** and **receive** calls

- **RDMA** - directly read/write to



Process 0

```
auto remote_ptr = ...;
// Calculate da
                                      , data);

                           .data(),
           values.size()*sizeof(float));

flush();

// Data is copied.
```

P1 does not participate in remote write.

intel labs

Remote Pointers

```cpp
template <typename T>
struct remote_ptr {

  ...

private:
  size_t rank_;
  size_t offset_;
};
```

Remote Pointers

```
template <typename T>
struct remote_ptr {

  ...

private:
  size_t rank_;
  size_t offset_;
};
```

```
void memcpy(void* dest, remote_ptr<void> src,
            size_t count) {
  // Issue remote get operation to
  // copy `count` bytes from `src` to `dest`
  MPI_Get(dest, count, MPI_BYTE,
          src.rank(), src.offset(), count,
          MPI_BYTE, __global::mpi_window_);
}
```

Remote Pointers

```cpp
template <typename T>
struct remote_ptr {

  remote_ref<T> operator*() const {
    return remote_ref(*this);
  }

private:
  size_t rank_;
  size_t offset_;
};
```

Remote Pointers

```
template <typename T>
struct remote_ptr {

  remote_ref<T> operator*() {
    return remote_ref(*this);
  }

private:
  size_t rank_;
  size_t offset_;
};
```

Cannot return T&, since memory may be remote.

Remote Pointers

```
template <typename T>
struct remote_ptr {

  remote_ref<T> operator
    return remote_ref(*t
  }

private:
  size_t rank_;
  size_t offset_;
};
```

```
template <typename T>
struct remote_ref {
  T& operator=(const T& value) {
    memcpy(ptr_, &value, sizeof(T));
    return value;
  }

  operator T() {
    T value;
    memcpy(&value, ptr_, sizeof(T));
    return value;
  }

private:
  remote_ptr<T> ptr_;
};
```

Remote Pointers

```
template <typename T>
struct remote_ptr {

  remote_ref<T> operator
    return remote_ref(*t
  }

private:
  size_t rank_;
  size_t offset_;
};
```

```
template <typename T>
struct remote_ref {
  T& operator=(const T& value) {
    memcpy(ptr_, &value, sizeof(T));
    return value;
  }

  operator T() {
    T value;
    memcpy(&value, ptr_, sizeof(T));
    return value;
  }

private:
  remote_ptr<T> ptr_;
};
```

```
remote_ptr<int> p = ...;
*p = 12;
```

Remote Pointers

```
template <typename T>
struct remote_ptr {

  remote_ref<T> operator
    return remote_ref(*t
  }

private:
  size_t rank_;
  size_t offset_;
};
```

```
template <typename T>
struct remote_ref {
  T& operator=(const T& value) {
    memcpy(ptr_, &value, sizeof(T));
    return value;
  }

  operator T() {
    T value;
    memcpy(&value, ptr_, sizeof(T));
    return value;
  }

private:
  remote_ptr<T> ptr_;
};
```

```
remote_ptr<int> p = ...;
*p = 12;
```

```
remote_ptr<int> p = ...;
int x = *p;
```

Remote Pointers

- Reference **memory on another process**
- Can support **memcpy**, **copy**, **atomics**, etc.
- Support **dereferencing**, with **proxy reference** type (no T&)
- Limited to **trivially copyable types**

Remote Pointers

- **What exactly** are these things?

- Remote pointers **do not fulfill *LegacyRandomAccessIterator*** 😔

Remote Pointers

- **What exactly** are these things?

- Remote pointers **do not fulfill *LegacyRandomAccessIterator*** 😔

[forward.iterators]

25.3.5.5 Forward iterators

1 A class or pointer type `X` meets the requirements of a forward iterator if

(1.1) — `X` meets the *Cpp17InputIterator* requirements ([input.iterators]),

(1.2) — `X` meets the *Cpp17DefaultConstructible* requirements ([utility.arg.requirements]),

(1.3) — if `X` is a mutable iterator, `reference` is a reference to `T`; if `X` is a constant iterator, `reference` is a reference to `const T`,

(1.4) — the expressions in Table 89 are valid and have the indicated semantics, and

(1.5) — objects of type `X` offer the multi-pass guarantee, described below.

Remote Pointers

- **What exactly** are these things?
- Remote pointers **do not fulfill *LegacyRandomAccessIterator*** 😔
- The **ranges library**, however is designed differently! 🥳

Indirectly Readable

- **Views** don’t necessarily have underlying data in memory
- e.g. **transform**, **iota**, **zip**
- View **reference types** are not necessarily references

```
using namespace std;

vector<int> x = {1, 2, 3, 4};
vector<int> y = {4, 3, 2, 1};

auto view = views::zip(x, y);

// [(1, 4), (2, 3), (3, 2), (4, 1)]
print("{}\n", view);
```

Indirectly Readable

- **Views** don’t necessarily have underlying data in memory
- e.g. **transform**, **iota**, **zip**
- View **reference types** are not necessarily references

```
using namespace std;

vector<int> x = {1, 2, 3, 4};
vector<int> y = {4, 3, 2, 1};

auto view = views::zip(x, y);

// [(1, 4), (2, 3), (3, 2), (4, 1)]
print("{}\n", view);

// What is decltype(v)?
auto&& v = view[1];
```

Indirectly Readable

- **Views** don't necessarily have underlying data in memory
- e.g. **transform**, **iota**, **zip**
- View **reference types** are not necessarily references

```
using namespace std;

vector<int> x = {1, 2, 3, 4};
vector<int> y = {4, 3, 2, 1};

auto view = views::zip(x, y);

// [(1, 4), (2, 3), (3, 2), (4, 1)]
print("{}\n", view);

// What is decltype(v)?
auto&& v = view[1];
```

`std::pair<int&, int&>`, ***not*** `std::pair<int, int>&`

intel labs

Indirectly Readable

- The **iterator concepts** that ranges depend on are **less strict** than the named requirements

25.3.4.11 Concept `forward_iterator` [iterator.concept.forward]

1 The `forward_iterator` concept adds copyability, equality comparison, and the multi-pass guarantee, specified below.

```
template<class I>
  concept forward_iterator =
    input_iterator<I> &&
    derived_from<ITER_CONCEPT(I), forward_iterator_tag> &&
    incrementable<I> &&
    sentinel_for<I, I>;
```

Indirectly Readable

- The **iterator concepts** that ranges depend on are **less strict** than the named requirements

25.3.4.11 Concept `forward_iterator` [iterator.concept.forward]

1 The `forward_iterator` concept adds copyability, equality comparison, and the multi-pass guarantee, specified below.

```
template<class I>
  concept forward_iterator =
    input_iterator<I> &&
    derived_from<ITER_CONCEPT(I), forward_iterator_tag> &&
    incrementable<I> &&
    sentinel_for<I, I>;
```

Indirectly Readable

- The **iterator concepts** that ranges depend on are **less strict** than the named requirements

- `*std::declval<Iter&>()` needs to be **valid** and **non-void**

25.3.4.11 Concept `forward_iterator` [iterator.concept.forward]

1 The `forward_iterator` concept adds copyability, equality comparison, and the multi-pass guarantee, specified below.

```
template<class I>
  concept forward_iterator =
    input_iterator<I> &&
    derived_from<ITER_CONCEPT(I), forward_iterator_tag> &&
    incrementable<I> &&
    sentinel_for<I, I>;
```

Indirectly Readable

- The **iterator concepts** that ranges depend on are **less strict** than the named requirements

- `*std::declval<Iter&>()` needs to be **valid** and **non-void**

Proxy references are fine!

25.3.4.11 Concept `forward_iterator` [iterator.concept.forward]

1 The `forward_iterator` concept adds copyability, equality comparison, and the multi-pass guarantee, specified below.

```
template<class I>
  concept forward_iterator =
    input_iterator<I> &&
    derived_from<ITER_CONCEPT(I), forward_iterator_tag> &&
    incrementable<I> &&
    sentinel_for<I, I>;
```

intel labs

Remote Pointers

- Remote pointers can **reference memory** that lives on **another node**
- They still fulfill the concept **`random_access_iterator`**
- We can use them to build a **`random_access_range`** containers/views

```
using namespace std::ranges;

remote_ptr<int> p = ...;

// v is a range!
subrange v(p, p + 100);
```

intel labs

Remote Pointers

- Remote pointers can **reference memory** that lives on **another node**
- They still fulfill the concept **`random_access_iterator`**
- We can use them to build a **`random_access_range`** containers/views

```
using namespace std::ranges;

remote_ptr<int> p = …;

// v is a range!
subrange v(p, p + 100);

reduce(v.begin(), v.end());
```

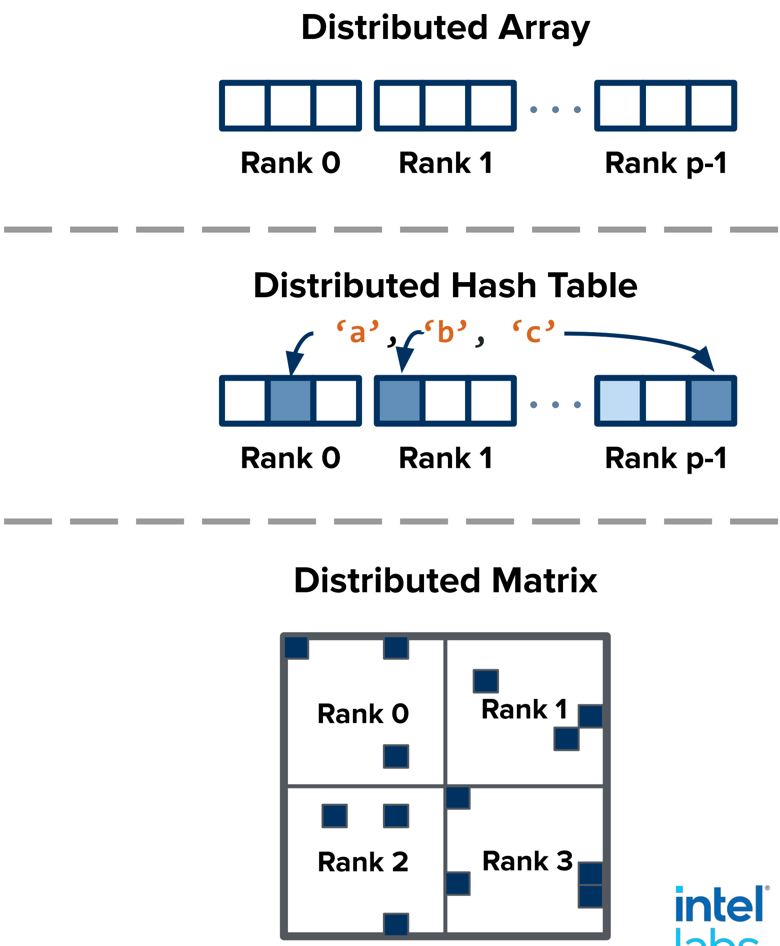
(Naively using them as such might not be smart.)

Distributed Data Structures

Distributed data structures **split up** data across multiple **segments**

Segments may be stored in **different memory regions**

We need a unified concept for accessing these distributed data structures!



Distributed Data Structures

Data is typically **partitioned** amongst processors into **segments**

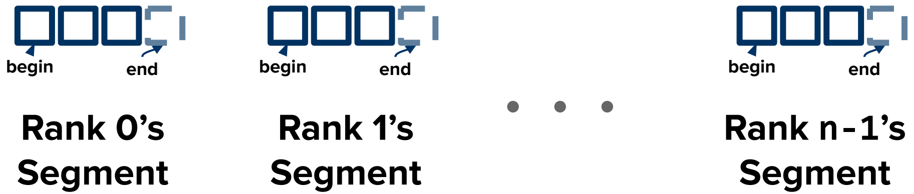
Segments are **remotely accessible**, and are located on a **single rank**

Distributed Data Structures

Data is typically **partitioned** amongst processors into **segments**

Segments are **remotely accessible**, and are located on a **single rank**

We can **access** the desired data using **remote pointers**



Distributed Data Structures

- Data structure stores a number of **segments**
- **Access data** through corresponding segment
- We can build **iteration** on top of this (although it may be slow)

```
template <typename T>
class distributed_vector {
public:

  remote_ref<T> operator[](size_t pos) {
    auto&& seg = segments_[pos / segment_size_];
    return seg[pos % segment_size_];
  }

  ...

private:
  std::size_t segment_size_;
  vector<remote_vector<T>> segments_;
};
```

intel labs

Distributed Data Structures

- Data structure stores a number of **segments**
- **Access data** through corresponding segment
- We can build **iteration** on top of this (although it may be slow)

```
template <typename T>
class distributed_vector {
public:

  remote_ref<T> operator[](size_t pos) {
    auto&& seg = segments_[pos / segment_size_];
    return seg[pos % segment_size_];
  }

  ...

private:
  std::size_t segment_size_;
  vector<remote_vector<T>> segments_;
};
```

intel labs

Distributed Data Structures

- Data structure stores a number of **segments**
- **Access data** through corresponding segment
- We can build **iteration** on top of this (although it may be slow)

```
template <typename T>
class distributed_vector {
public:

  remote_ref<T> operator[](size_t pos) {
    auto&& seg = segments_[pos / segment_size_];
    return seg[pos % segment_size_];
  }

  ...

private:
  std::size_t segment_size_;
  vector<remote_vector<T>> segments_;
};
```

intel labs

Distributed Data Structures

- Data structure stores a number of **segments**

- **Access data** through corresponding segment

- We can build **iteration** on top of this (although it may be slow)

```
template <typename T>
class distributed_vector {
public:

  remote_ref<T> operator[](size_t pos) {
    auto&& seg = segments_[pos / segment_size_];
    return seg[pos % segment_size_];
  }

  ...

private:
  std::size_t segment_size_;
  vector<remote_vector<T>> segments_;
};
```

Distributed Data Structures

- With iteration, our distributed data structures are **ranges**
- Global iteration is **useful** (e.g. printing), but **slow**
- We need a generic **range concept** for **distributed data structures**

```
void add_two(distributed_vector<int>& v) {
  for (auto iter = v.begin(); iter != v.end(); ++iter) {
    *iter += 2;
  }
}

...

distributed_vector<int>& v(10, 0);

if (rank() == 0) {
  // [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
  print("{}\n", v);

  add_two(v);

  // [2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
  print("{}\n", v);
}
```

intel labs

Outline

- Background (Ranges and Standard Parallelism)
- **Distributed Ranges** (Concepts)
- Implementation (Algorithms and views)
- Complex Data Structures (Dense and sparse matrices)
- Lessons learned

Distributed Range Concepts

Remote Range

A remote range:

1) Is a **range** (satisfies `forward_range`)
2) Has a **rank** (implements `rank` CPO)

```
template <typename R>
concept remote_range = forward_range<R>
                       && requires(R &r) {
                         rank(r);
                       };
```

Remote Range

A remote range:

1) Is a **range** (satisfies `forward_range`)
2) Has a **rank** (implements `rank` CPO)

```
template <typename R>
concept remote_range = forward_range<R>
                       && requires(R &r) {
                         rank(r);
                       };
```

Can operate on this as a **normal range**.

intel labs

Remote Range

A remote range:

1) Is a **range** (satisfies `forward_range`)
2) Has a **rank** (implements `rank` CPO)

Has a concept of ***locality***.

```
template <typename R>
concept remote_range = forward_range<R>
                       && requires(R &r) {
                         rank(r);
                       };
```

Remote Range

A remote range:

1) Is a **range** (satisfies `forward_range`)
2) Has a **rank** (implements `rank` CPO)

```
template <typename R>
concept remote_range = forward_range<R>
                       && requires(R &r) {
                            rank(r);
                          };
```

Given a **remote range**, we can ask:

Which memory space is this data located in? (`rank(r)`)

The data lives there, so it will be fastest to **execute in that memory space**.

Distributed Range

A distributed range:

1) Is a **range** (satisfies `forward_range`)
2) Has **segments** (implements `segments` CPO)

```
template <typename R>
concept distributed_range = forward_range<R>
                            && requires(R &r) {
                                 segments(r);
                               };
```

intel labs

Distributed Range

A distributed range:

1) Is a **range** (satisfies `forward_range`)
2) Has **segments** (implements `segments` CPO)

```
template <typename R>
concept distributed_range = forward_range<R>
                            && requires(R &r) {
                                 segments(r);
                               };
```

Can operate on this as a **normal range**.

intel labs

Distributed Range

A distributed range:

1) Is a **range** (satisfies `forward_range`)
2) Has **segments** (implements `segments` CPO)

```
template <typename R>
concept distributed_range = forward_range<R>
                            && requires(R &r) {
                              segments(r);
                            };
```

Segments returns a **range** of **remote ranges**.

This exposes **distribution** and **locality** of the distributed range.

Distributed Range

1) Global view

2) Segmented view

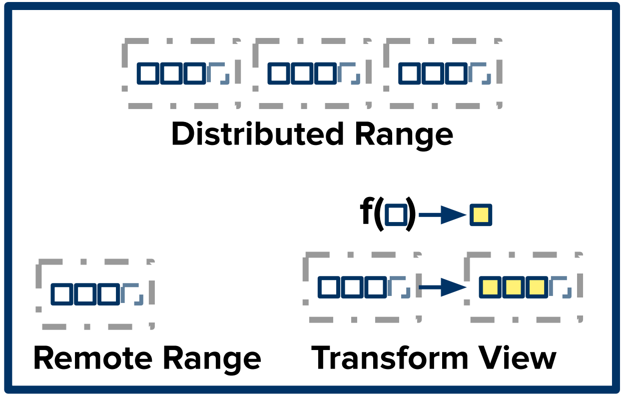
Segmented view exposes distribution, allows **hierarchical implementation** of **algorithms** and **views**.

Outline

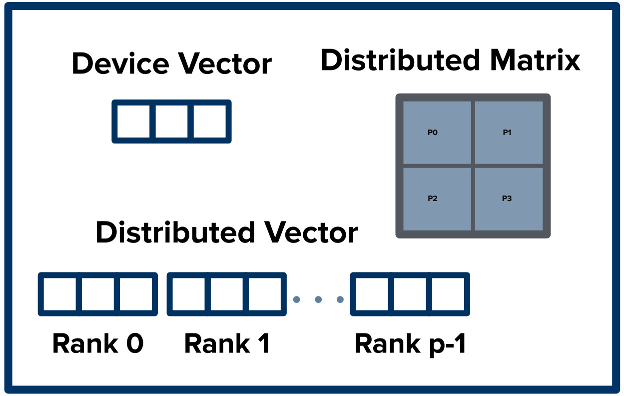
- Background (Ranges and Standard Parallelism)
- Distributed Ranges (Concepts)
- **Implementation** (Algorithms and views)
- Complex Data Structures (Dense and sparse matrices)
- Lessons learned

Distributed Ranges Project

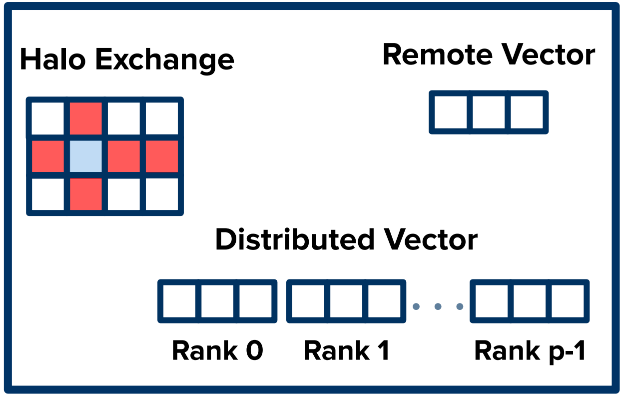
Shared Concepts and Views



GPU Data Structures and Algorithms (“shp”)



MPI Data Structures and Algorithms (“mhp”)



Building a Distributed Data Structure

- Data structure contains a bunch of segments (**remote ranges**)

```
template <typename T>
class distributed_vector {
public:

  ...

private:
  std::size_t segment_size_;
  vector<remote_vector<T>> segments_;
};
```

intel labs

Building a Distributed Data Structure

- Data structure contains a bunch of segments (**remote ranges**)

```
template <typename T>
class distributed_vector {
public:

   ...

private:
  std::size_t segment_size_;
  vector<remote_vector<T>> segments_;
};
```

Building a Distributed Data Structure

- Data structure contains a bunch of segments (**remote ranges**)

```
template <typename T>
class distributed_vector {
public:


  ...



private:
  std::size_t segment_size_;
  vector<remote_vector<T>> segments_;
};
```

Building a Distributed Data Structure

- Data structure contains a bunch of segments (**remote ranges**)
- To fulfill distributed range, implement **`segments`** CPO
- (Return a range of remote ranges)

```
template <typename T>
class distributed_vector {
public:


    ...


private:
  std::size_t segment_size_;
  vector<remote_vector<T>> segments_;
};
```

Building a Distributed Data Structure

- Data structure contains a bunch of segments (**remote ranges**)
- To fulfill distributed range, implement **`segments`** CPO
- (Return a range of remote ranges)

```
template <typename T>
class distributed_vector {
public:

  // Return a view of the remote ranges
  auto segments() {
    return views::all(segments_);
  }

  …

private:
  std::size_t segment_size_;
  vector<remote_vector<T>> segments_;
};
```

Building a Distributed Data Structure

- Data structure contains a bunch of segments (**remote ranges**)
- To fulfill distributed range, implement **`segments`** CPO
- (Return a range of remote ranges)

```
template <typename T>
class distributed_vector {
public:

  // Return a view of the remote ranges
  auto segments() {
    return views::all(segments_);
  }

  ...

private:
  std::size_t segment_size_;
  vector<remote_vector<T>> segments_;
};
```

Building a Distributed Data Structure

- Data structure contains a bunch of segments (**remote ranges**)
- To fulfill distributed range, implement **`segments`** CPO
- (Return a range of remote ranges)

```
template <typename T>
class distributed_vector {
public:

  // Return a view of the remote ranges
  auto segments() {
    return views::all(segments_);
  }

  auto begin() { return global_view_.begin(); }

  ...

private:
  std::size_t segment_size_;
  vector<remote_vector<T>> segments_;
  join_view<...> global_view_;
};
```

Building a Distributed Data Structure

- Data structure contains a bunch of segments (**remote ranges**)
- To fulfill distributed range, implement **`segments`** CPO
- (Return a range of remote ranges)

```
template <typename T>
class distributed_vector {
public:

  // Return a view of the remote ranges
  auto segments() {
    return views::all(segments_);
  }

  auto begin() { return global_view_.begin(); }

  ...

private:
  std::size_t segment_size_;
  vector<remote_vector<T>> segments_;
  join_view<...> global_view_;
};
```

intel labs

Distributed Algorithms

- Algorithms accept **distributed range** parameters

- Access data using **`segments`** CPO

- Launch data on execution agent corresponding to **memory space**

```
using namespace std::ranges;
using namespace dr::shp;
using namespace oneapi;

template <distributed_range R, typename T>
T reduce(R&& r, T init) {
  /*...*/
}
```

intel labs

Distributed Algorithms

- Algorithms accept **distributed range** parameters
- Access data using **segments** CPO
- Launch data on execution agent corresponding to **memory space**

```
using namespace std::ranges;
using namespace dr::shp;
using namespace oneapi;

template <distributed_range R, typename T>
T reduce(R&& r, T init) {
  std::vector</* future type */> futures;

  for (auto&& segment : segments(r)) {
    if (size(segment) > 0) {
      auto rank = rank(segment);

      auto&& policy = get_device_policy(rank);
      auto f = dpl::reduce_async(policy, segment);
      futures.push_back(f);
    }
  }

  for (auto&& f : futures) {
    init += f.get();
  }

  return init;
}
```

intel labs

Distributed Algorithms

- Algorithms accept **distributed range** parameters
- Access data using **segments** CPO
- Launch data on execution agent corresponding to **memory space**

```
using namespace std::ranges;
using namespace dr::shp;
using namespace oneapi;

template <distributed_range R, typename T>
T reduce(R&& r, T init) {
  std::vector</* future type */> futures;

  for (auto&& segment : segments(r)) {
    if (size(segment) > 0) {
      auto rank = rank(segment);

      auto&& policy = get_device_policy(rank);
      auto f = dpl::reduce_async(policy, segment);
      futures.push_back(f);
    }
  }

  for (auto&& f : futures) {
    init += f.get();
  }

  return init;
}
```

Distributed Algorithms

- Algorithms accept **distributed range** parameters
- Access data using **segments** CPO
- Launch data on execution agent corresponding to **memory space**

```cpp
using namespace std::ranges;
using namespace dr::shp;
using namespace oneapi;

template <distributed_range R, typename T>
T reduce(R&& r, T init) {
  std::vector</* future type */> futures;

  for (auto&& segment : segments(r)) {
    if (size(segment) > 0) {
      auto rank = rank(segment);

      auto&& policy = get_device_policy(rank);
      auto f = dpl::reduce_async(policy, segment);
      futures.push_back(f);
    }
  }

  for (auto&& f : futures) {
    init += f.get();
  }

  return init;
}
```

intel labs

Distributed Algorithms

- Algorithms accept **distributed range** parameters

- Access data using **segments** CPO

- Launch data on execution agent corresponding to **memory space**

```
using namespace std::ranges;
using namespace dr::shp;
using namespace oneapi;

template <distributed_range R, typename T>
T reduce(R&& r, T init) {
  std::vector</* future type */> futures;

  for (auto&& segment : segments(r)) {
    if (size(segment) > 0) {
      auto rank = rank(segment);

      auto&& policy = get_device_policy(rank);
      auto f = dpl::reduce_async(policy, segment);
      futures.push_back(f);
    }
  }

  for (auto&& f : futures) {
    init += f.get();
  }

  return init;
}
```

Distributed Algorithms

- Algorithms accept **distributed range** parameters
- Access data using **segments** CPO
- Launch data on execution agent corresponding to **memory space**

```
using namespace std::ranges;
using namespace dr::shp;
using namespace oneapi;

template <distributed_range R, typename T>
T reduce(R&& r, T init) {
  std::vector</* future type */> futures;

  for (auto&& segment : segments(r)) {
    if (size(segment) > 0) {
      auto rank = rank(segment);

      auto&& policy = get_device_policy(rank);
      auto f = dpl::reduce_async(policy, segment);
      futures.push_back(f);
    }
  }

  for (auto&& f : futures) {
    init += f.get();
  }

  return init;
}
```

Views

- Views: take in a **base view V**
- Fulfill range concepts

```
std::vector<int> x = {1, 2, 3, 4};

auto v = x
       | views::transform([](auto&& x) {
                            return x * 2;
                          });

// v is now a random_access_range
int y = v[2];

// 6
print("{}\n", y);
```

Views

- Views: take in a **base view V**
- Fulfill range concepts

- Many (but not all) views have a way to get at the **base view**

```
std::vector<int> x = {1, 2, 3, 4};

auto v = x
       | views::transform([](auto&& x) {
                             return x * 2;
                           });

// v is now a random_access_range
int y = v[2];

// 6
print("{}\n", y);

// base is a ref_view<vector<int>>
auto base = v.base();

// o is a std::vector<int>&
auto&& o = base.base();
```

intel labs

Remote Views

- To be a **remote range**, we must implement the **rank** CPO

```
remote_vector<int> v(/*...*/);

auto view = v
          | views::transform([](auto x) {
                                   return x*2;
                                 });

// We'd like this to work
auto rank = rank(view);
```

Remote Views

- To be a **remote range**, we must implement the **rank** CPO

- We have two options:

1) Implement **views from scratch** 😔

```
remote_vector<int> v(/*...*/);

auto view = v
          | views::transform([](auto x) {
                                return x*2;
                             });

// We'd like this to work
auto rank = rank(view);
```

intel labs

Remote Views

- To be a **remote range**, we must implement the **rank** CPO

- We have two options:

1) Implement **views from scratch** 😔
2) Implement **rank** CPO for **library implementation** 🤔

```
remote_vector<int> v(/*...*/);

auto view = v
          | views::transform([](auto x) {
                                 return x*2;
                             });

// We'd like this to work
auto rank = rank(view);
```

intel labs

Remote Views

- If a view's base is a remote range:

1) The **derived view** must be in the same **memory space**

```
template <remote_range V>
auto rank(const ref_view<V>& view) {
  auto& base = view.base();
  return rank(base);
}
```

2) We can obtain the rank by calling **rank** on its **base**

```
template <remote_range V>
auto rank(const transform_view<V>& view) {
  auto&& base = view.base();
  return rank(base);
}
```

intel labs

Distributed Views

To be a distributed range, we must implement the **`segments`** CPO

```
distributed_vector<int> v(/*...*/);

auto view = v
          | views::transform([](auto x) {
                                return x*2;
                              });

// We'd like this to work
auto&& segments = segments(view);
```

Can we use **the same trick**? (**Kind of.**)

Distributed Views

To be a distributed range, we must implement the **`segments`** CPO

```
template <distributed_range R>
auto segments(transform_view<R>& v) {
  auto&& base = v.base();

  return segments(base)
       | transform(
           [=](auto&& seg) {
             // Doh! No way to access fun.
             return seg | transform(fun);
           });
}
```

Distributed Views

To be a distributed range, we must implement the **`segments`** CPO

```
template <distributed_range R>
auto segments(transform_view<R>& v) {
  auto&& base = v.base();

  return segments(base)
       | transform(
           [=](auto&& seg) {
             // Doh! No way to access fun.
             return seg | transform(fun);
           });
}
```

Distributed Views

To be a distributed range, we must implement the **`segments`** CPO

```
template <distributed_range R>
auto segments(transform_view<R>& v) {
  auto&& base = v.base();

  return segments(base)
       | transform(
           [=](auto&& seg) {
             // Doh! No way to access fun.
             return seg | transform(fun);
           });
}
```

intel labs

Distributed Views

To be a distributed range, we must implement the **`segments`** CPO

However, once we create **`transform_view`**, no way to access **fun**.
(exposition-only private member)

```
template <distributed_range R>
auto segments(transform_view<R>& v) {
  auto&& base = v.base();

  return segments(base)
       | transform(
           [=](auto&& seg) {
             // Doh! No way to access fun.
             return seg | transform(fun);
           });
}
```

intel labs

Distributed Views

To be a distributed range, we must implement the **`segments`** CPO

However, once we create **`transform_view`**, no way to access **fun**.
(exposition-only private member)

```
template <distributed_range R>
auto segments(transform_view<R>& v) {
  auto&& base = v.base();

  return segments(base)
       | transform(
           [=](auto&& seg) {
             // Doh! No way to access fun.
             return seg | transform(fun);
           });
}
```

Distributed Views

Workaround: implement our own **`transform_view`**, implement segments as a method

(Some views such as **`take`**, **`drop`**, and **`subrange`**, can be still be implemented **just using CPO**.)

Still a fairly **modular implementation**.

```
template <forward_range V,
          copy_constructible F>
class transform_view {
public:

  /*...*/

  auto segments()
  requires(distributed_range<V>)
  {
    return segments(base_)
        | transform(
            [=](auto&& seg) {
              return seg | transform(fun_);
            });
  }

private:
  V base_;
  F fun_;
};
```

SYCL Codebase (shp)

- Data **automatically distributed** amongst **multiple GPUs**

- **Distributed algorithms**: each GPU calls into **oneDPL algorithms**

```
using namespace dr::shp;


template <distributed_range R>
auto dot_product(R& x, R& y) {
  using T = range_value_t<R>;

  auto z =  views::zip(x, y)
          | views::transform([](auto element) {
             auto [a, b] = element;
             return a * b;
            });

  return reduce(par_unseq, z.begin(), z.end(),
                T(0), std::plus());
}
```

intel labs

SYCL Codebase (shp)

- Data **automatically distributed** amongst **multiple GPUs**

- **Distributed algorithms**: each GPU calls into **oneDPL algorithms**

```
using namespace dr::shp;


template <distributed_range R>
auto dot_product(R& x, R& y) {
  using T = range_value_t<R>;

  auto z =  views::zip(x, y)
          | views::transform([](auto element) {
             auto [a, b] = element;
             return a * b;
            });

  return reduce(par_unseq, z.begin(), z.end(),
                T(0), std::plus());
}
```

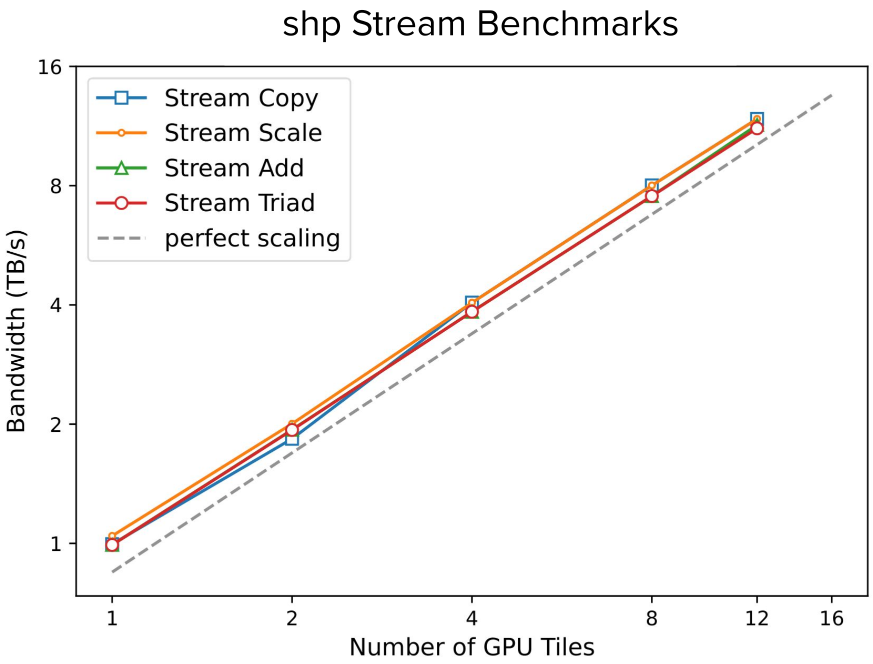
Multi-Node Codebase (mhp)

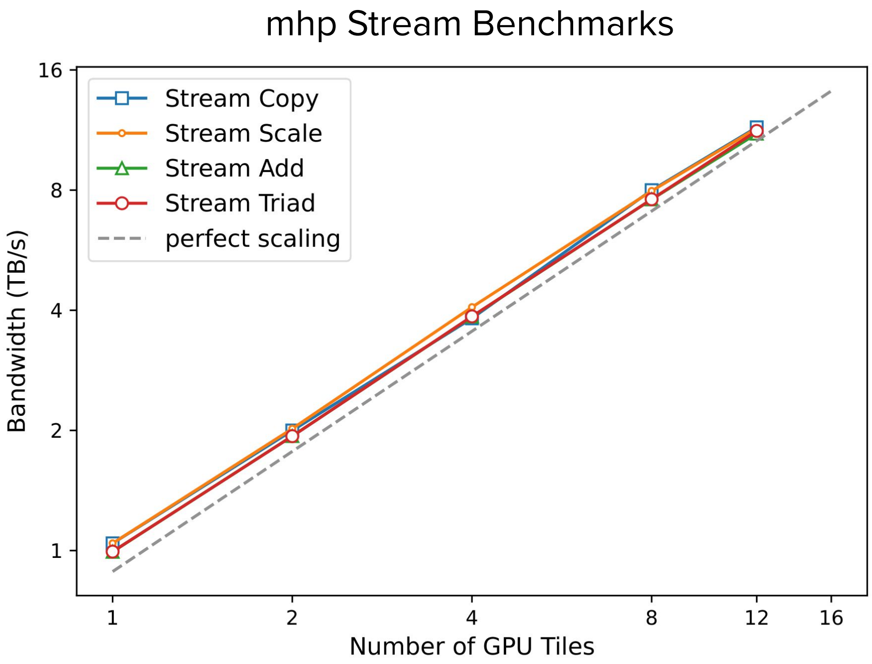
- **Multi-process, SPMD** program
- Data structures **automatically distributed** on **multiple nodes** using MPI
- Data structure **constructors** and **algorithms** are collective

```
using namespace dr::mhp;
```

```
template <distributed_range R>
auto dot_product(R& x, R& y) {
  using T = range_value_t<R>;

  auto z =  views::zip(x, y)
          | views::transform([](auto element) {
             auto [a, b] = element;
             return a * b;
            });

  return reduce(par_unseq, z.begin(), z.end(),
                T(0), std::plus());
}
```

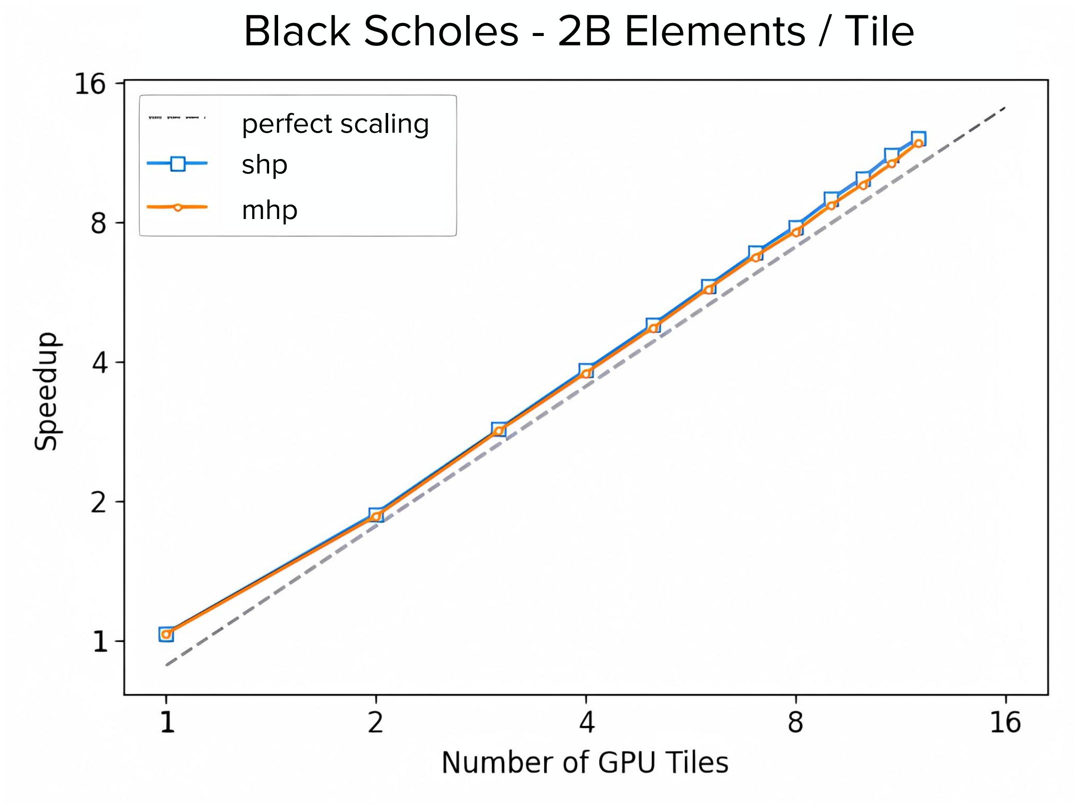
Stream Benchmarks





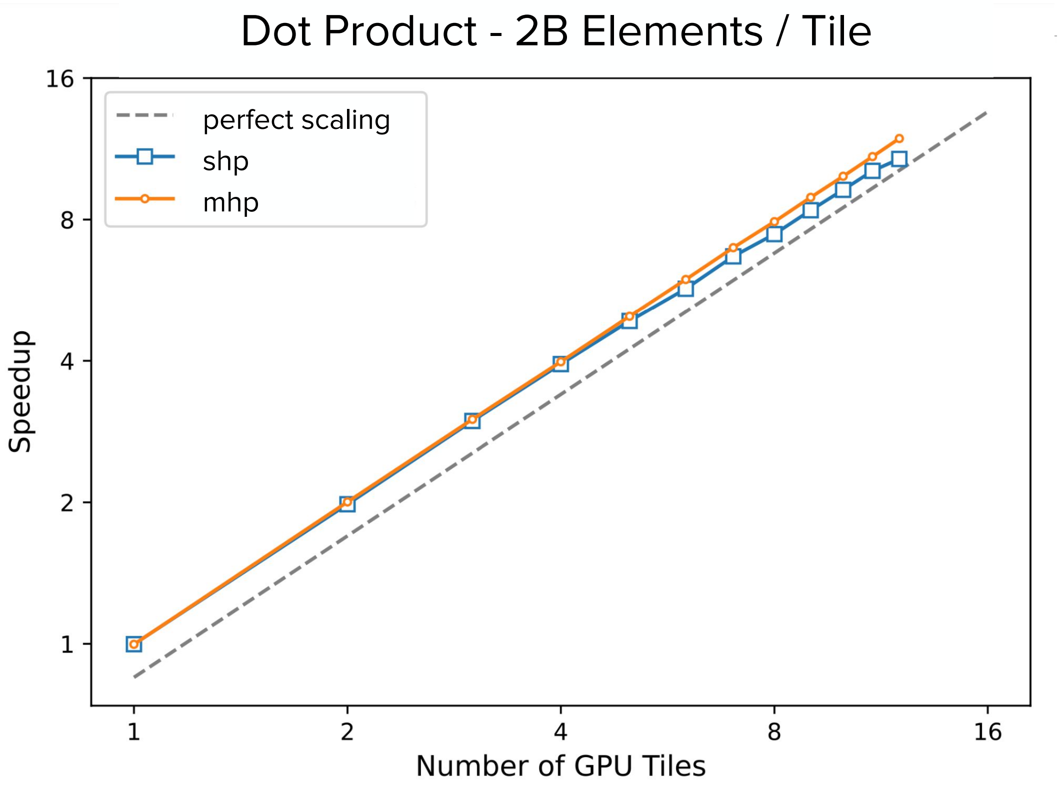
intel labs

Black Scholes




```
auto black_scholes_kernel = [=](auto &&e) {
  auto &&[s0, x, t, vcall, vput] = e;
  T d1 = (std::log(s0 / x) + (r + T(0.5) * sig * sig) * t) /
         (sig * std::sqrt(t));
  T d2 = (std::log(s0 / x) + (r - T(0.5) * sig * sig) * t) /
         (sig * std::sqrt(t));
  vcall = s0 * normalCDF(d1) - std::exp(-r * t) * x *
          normalCDF(d2);
  vput = std::exp(-r * t) * x * normalCDF(-d2) - s0 *
         normalCDF(-d1);
};


void black_scholes(auto&& s0, auto&& x, auto&& t,
                   auto&& vcall, auto&& vput) {
  for_each(zip(s0, x, t, vcall, vput),
           black_scholes_kernel);
}
```

Dot Product




```
float dot_product(vector<float>& x,
                  vector<float>& y) {

  auto z =  views::zip(x, y)
          | views::transform([](auto element) {
             auto [a, b] = element;
             return a * b;
           });

  return reduce(par_unseq, z, 0, std::plus());
}
```

Data Structure/Algorithms Demo

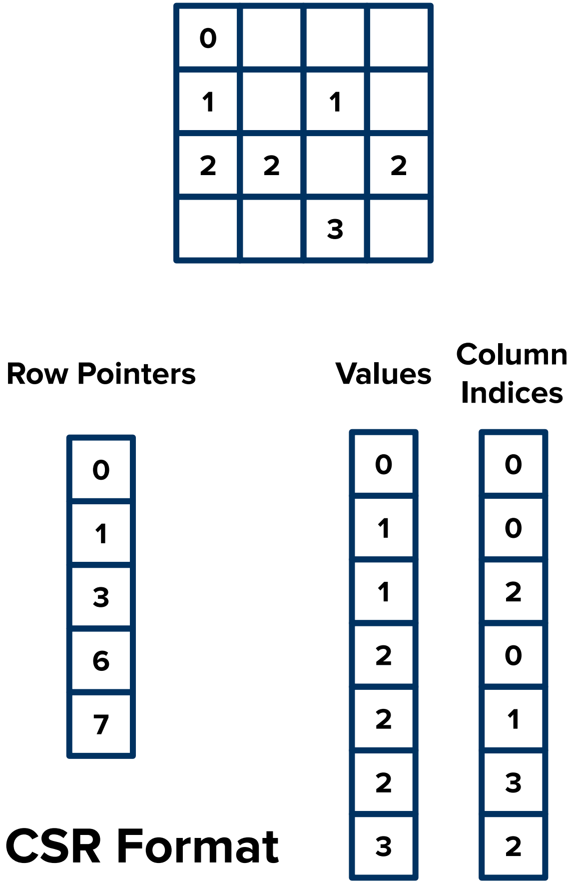
Outline

- Background (Ranges and Standard Parallelism)
- Distributed Ranges (Concepts)
- Implementation (Algorithms and views)
- **Complex Data Structures** (Dense and sparse matrices)
- Lessons learned

Dense and Sparse Matrices

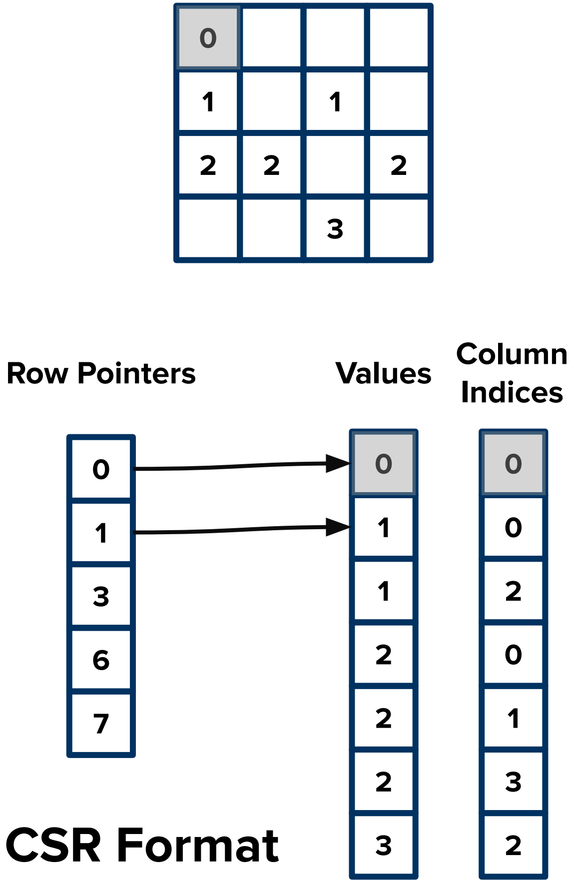
- **Sparse matrices** can have many different formats

- Each format may support different **iteration orders**



Dense and Sparse Matrices

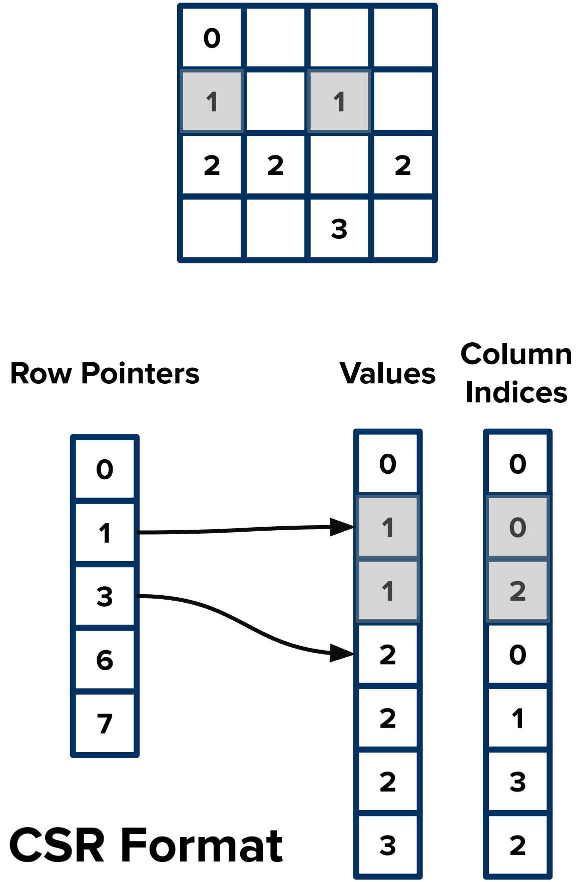
- **Sparse matrices** can have many different formats
- Each format may support different **iteration orders**



Dense and Sparse Matrices

- **Sparse matrices** can have many different formats

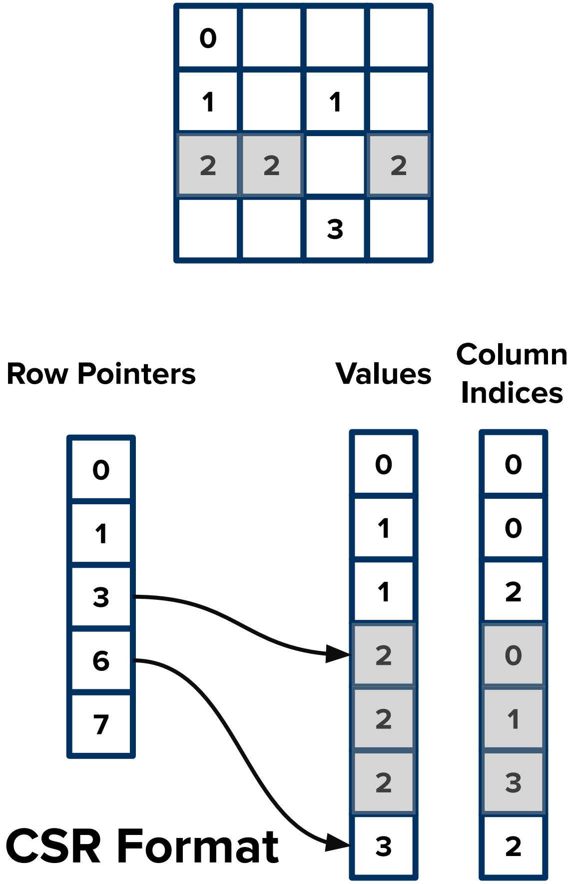
- Each format may support different **iteration orders**



Dense and Sparse Matrices

- **Sparse matrices** can have many different formats

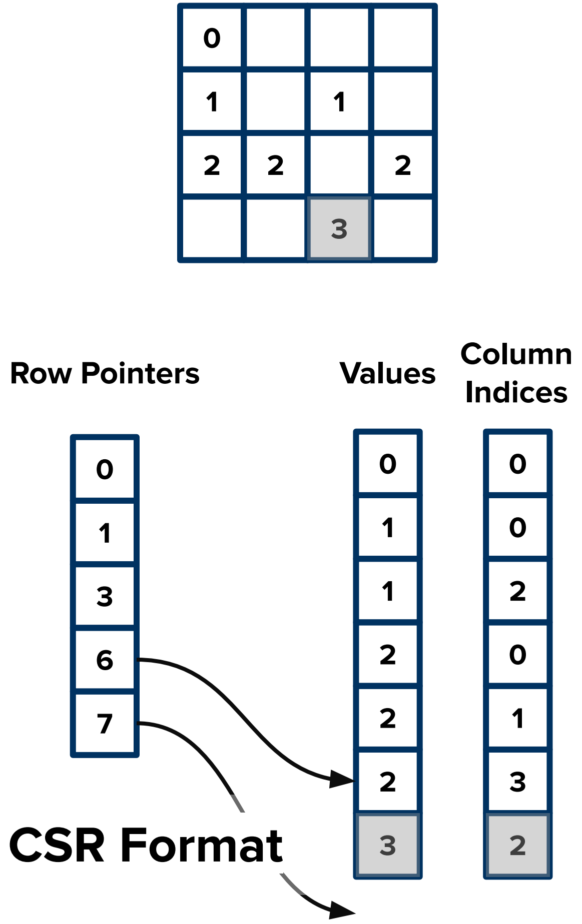
- Each format may support different **iteration orders**



Dense and Sparse Matrices

- **Sparse matrices** can have many different formats

- Each format may support different **iteration orders**



Dense and Sparse Matrices

- **Sparse matrices** can have many different formats
- Each format may support different **iteration orders**

0			
1		1	
2	2		2
		3	

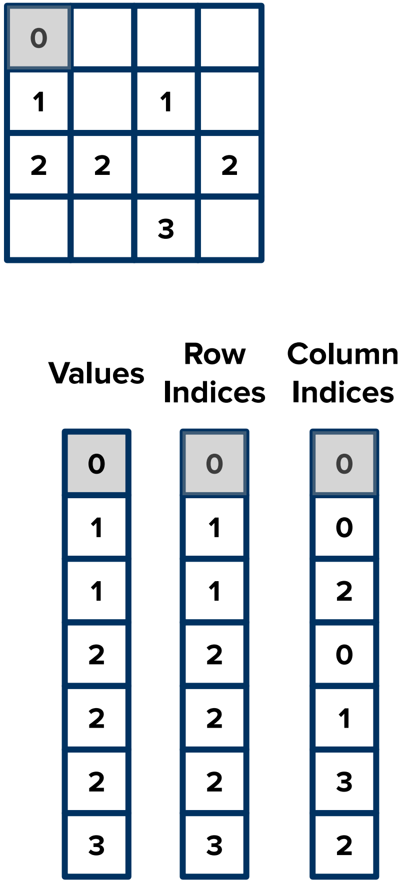
COO Format

Values	Row Indices	Column Indices
0	0	0
1	1	0
1	1	2
2	2	0
2	2	1
2	2	3
3	3	2

Dense and Sparse Matrices

- **Sparse matrices** can have many different formats

- Each format may support different **iteration orders**

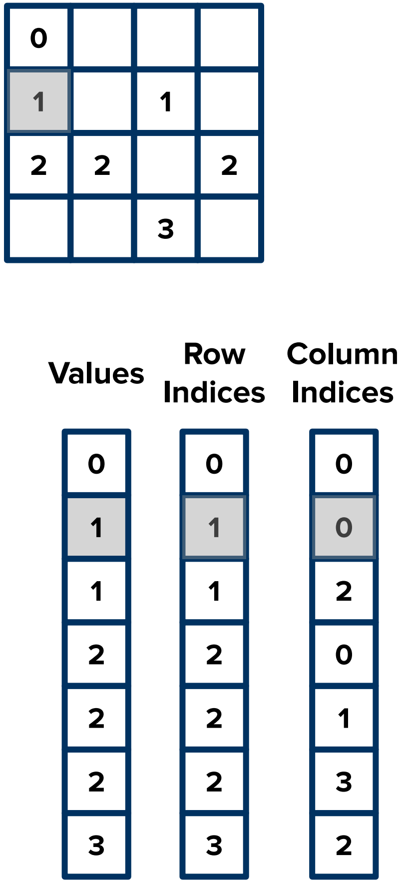


COO Format

Dense and Sparse Matrices

- **Sparse matrices** can have many different formats

- Each format may support different **iteration orders**



COO Format

Dense and Sparse Matrices

- **Sparse matrices** can have many different formats

- Each format may support different **iteration orders**

0			
1		1	
2	2		2
		3	

Values	Row Indices	Column Indices
0	0	0
1	1	0
1	1	2
2	2	0
2	2	1
2	2	3
3	3	2

COO Format

Dense and Sparse Matrices

- **Sparse matrices** can have many different formats

- Each format may support different **iteration orders**

0			
1		1	
2	2		2
		3	

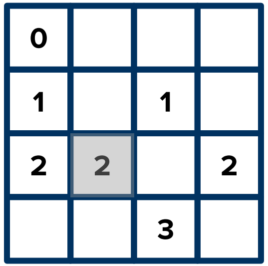
COO Format

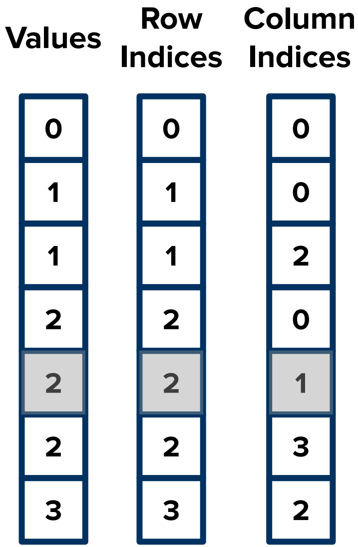
Values	Row Indices	Column Indices
0	0	0
1	1	0
1	1	2
2	2	0
2	2	1
2	2	3
3	3	2

Dense and Sparse Matrices

- **Sparse matrices** can have many different formats

- Each format may support different **iteration orders**





COO Format

Dense and Sparse Matrices

- **Sparse matrices** can have many different formats

- Each format may support different **iteration orders**

0			
1		1	
2	2		2
		3	

Values	Row Indices	Column Indices
0	0	0
1	1	0
1	1	2
2	2	0
2	2	1
2	2	3
3	3	2

COO Format

Dense and Sparse Matrices

- **Sparse matrices** can have many different formats

- Each format may support different **iteration orders**

0			
1		1	
2	2		2
		3	

Values	Row Indices	Column Indices
0	0	0
1	1	0
1	1	2
2	2	0
2	2	1
2	2	3
3	3	2

COO Format

Dense and Sparse Matrices

- GraphBLAS **sparse matrix** concept: **unordered iteration** through matrix tuples

```
sparse_matrix<float> x = read_matrix("a.mtx");

for (auto&& [idx, v] : x) {
  auto&& [i, j] = idx;

  print("Tuple at idx {}, {} with value {}\n",
        i, j, v);
}
```

Dense and Sparse Matrices

- GraphBLAS **sparse matrix** concept: **unordered iteration** through matrix tuples

- (Possibly some additional CPOs for ordered multi-dimensional iteration)

```
sparse_matrix<float> x = read_matrix("a.mtx");

for (auto&& [idx, v] : x) {
  auto&& [i, j] = idx;

  print("Tuple at idx {}, {} with value {}\n",
        i, j, v);
}

/* *If* `x` supports row iteration. */
for (auto&& [i, row] : rows(x)) {
  /* Do something with row */
}

/* *If* `x` supports column iteration. */
for (auto&& [j, column] : columns(x)) {
  /* Do something with column */
}
```

Dense and Sparse Matrices

- GraphBLAS **sparse matrix** concept: **unordered iteration** through matrix tuples

- (Possibly some additional CPOs for ordered multi-dimensional iteration)

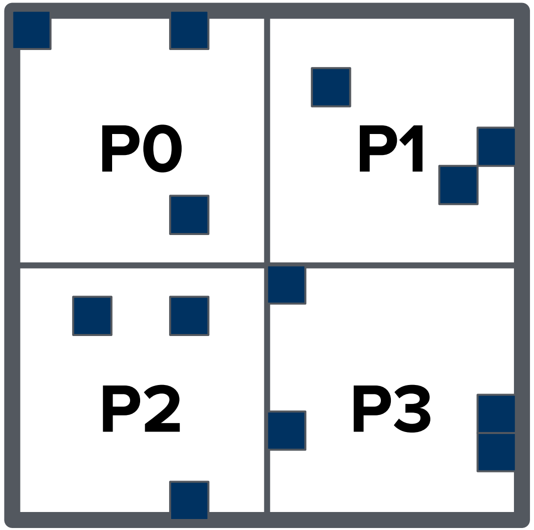
Shout out to Graph Library Proposal P1709

```
sparse_matrix<float> x = read_matrix("a.mtx");

for (auto&& [idx, v] : x) {
  auto&& [i, j] = idx;

  print("Tuple at idx {}, {} with value {}\n",
        i, j, v);
}

/* *If* `x` supports row iteration. */
for (auto&& [i, row] : rows(x)) {
  /* Do something with row */
}

/* *If* `x` supports column iteration. */
for (auto&& [j, column] : columns(x)) {
  /* Do something with column */
}
```

intel labs

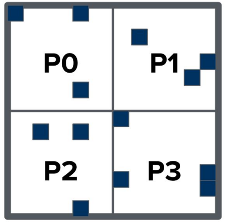
Distributed Dense and Sparse Matrices

- Distributed matrix data structures split up matrix into **tiles**
- Each tile stored in **different memory space**
- Tiles can be **sparse** or **dense**



intel labs

Distributed Dense and Sparse Matrices

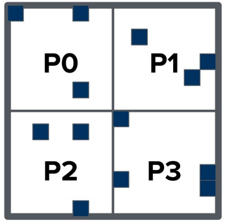


- Each tile represents a **submatrix**
- Tiles satisfy **`remote_range`**

```
distributed_sparse_matrix<float> x = ...;

// tile is a remote_range
auto tile = x.tile({0, 0});


// [{{0, 0}, 12.f}, {{5, 0}, 3.f}, {{5, 5}, 1.f}]
print("{}\n", tile);
```

intel labs

Distributed Dense and Sparse Matrices



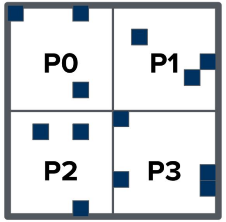
- Each tile represents a **submatrix**
- Tiles satisfy **`remote_range`**
- Also have **`get_tile()`** to retrieve copy of tile

(And `get_tile_async()` for asynchronous copying.)

```
distributed_sparse_matrix<float> x = ...;

// tile is a remote_range
auto tile = x.tile({0, 0});


// [{{0, 0}, 12.f}, {{5, 0}, 3.f}, {{5, 5}, 1.f}]
print("{}\n", tile);


// tile2 is a local copy of tile
auto tile2 = x.get_tile({0, 0});

// *printed faster*
// [{{0, 0}, 12.f}, {{5, 0}, 3.f}, {{5, 5}, 1.f}]
print("{}\n", tile2);
```

intel labs

Distributed Dense and Sparse Matrices



- By creating a range that returns these tiles, we can implement **segments** for our matrix

- This allows us to use **normal range algorithms** on matrices.

(A local → global index transformation is required, although not discussed here.)

```
distributed_sparse_matrix<float> x = ...;

// Return view of values in matrix
auto v = x | transform([](auto&& e) {
                       auto&& [idx, v] = e;
                       return v;
                     });

// Compute sum of values in matrix
auto sum = reduce(x);
```

Outline

- Background (Ranges and Standard Parallelism)
- Distributed Ranges (Concepts)
- Implementation (Algorithms and views)
- Complex Data Structures (Dense and sparse matrices)
- **Lessons learned**

Standard C++ Parallelism

- **Data structures**
  - Hold and organize data
- **Views**
  - Lightweight objects, views of data
- **Algorithms**
  - Operate on and modify data

```
using namespace std;
using namespace std::ranges;
using namespace std::execution;

template <range R>
auto dot_product(R&& x, R&& y) {
  using T = range_value_t<R>;

  auto z =  views::zip(x, y)
          | views::transform([](auto element) {
             auto [a, b] = element;
             return a * b;
            });

  return reduce(par_unseq, z.begin(), z.end(),
                T(0), std::plus());
}
```

Standard C++ Parallelism

- **Data structures**
  - Hold and organize data
- **Views**
  - Light
    views
- **Algorithms**
  - Operate on and modify data

```
using namespace std;
using namespace std::ranges;
using namespace std::execution;

template <range R
                                    R&& y) {
                     _value_t<R>;

                     ws::zip(x, y)
                   | views::transform([](auto element) {
                       auto [a, b] = element;
                       return a * b;
                     });

  return reduce(par_unseq, z.begin(), z.end(),
                T(0), std::plus());
}
```

Ranges are the **glue** that hold this all together!

Lessons Learned - Building on top of Ranges

- The **ranges library** provides many useful tools to build on top of
- We can **refine** ranges concepts to add functionality (**distributed/remote** ranges)
- Ranges has ***almost*** everything we'd need to build on top of...

Wish List for Ranges

- We ended up building some views **from scratch**
- Would be nice if **every view** exposed its **base(s)** (including zip!)
- Would be nice if **`transform_view`** exposed fun

Limitations

Many things are still a **work in progress**:

- “**Distributed iterator**” concepts
  (Would be nice if **view iterators** exposed **base iterators**)

- Ranges of **ranges**
  (Can we have **nested distributed ranges**?)

- Don’t have full coverage

- **rank** should be formalized

Wrap Up

- We can extend **ranges** to support **distributed data structures**

- Unites **data structures**, **algorithms**, and **views** with **one abstraction**

- Utilize **vendor-supplied** local algorithms for **competitive performance**

- Our work is **open source**: https://github.com/oneapi-src/distributed-ranges

intel labs