Standard Parallel Algorithm Tasks in Taskflow

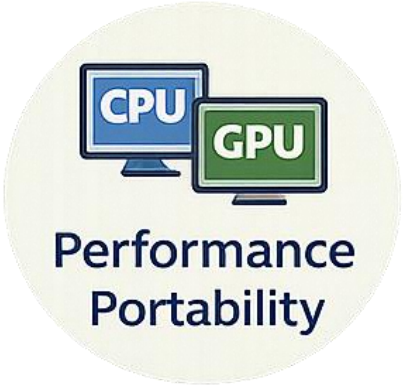
Dr. Tsung-Wei (TW) Huang
Department of Electrical and Computer Engineering
University of Wisconsin at Madison, Madison, WI
https://taskflow.github.io/

Takeaways

- **Understand the need of standard parallel algorithm tasks**
- **Learn how to create standard algorithm tasks in Taskflow**
  - Parallel for
  - Parallel reduction
  - Parallel sort
- **Launch algorithm tasks with asynchronous tasking**
- **Conclude**

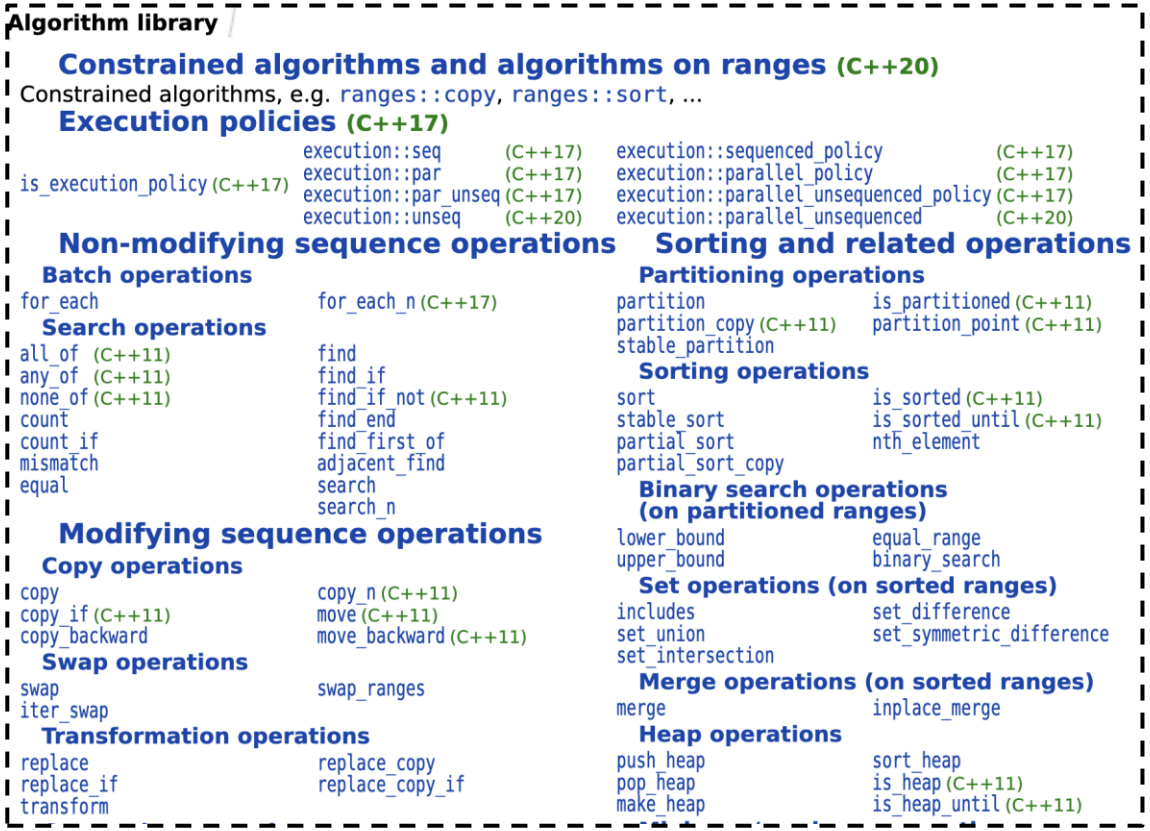
Standard Algorithms: What and Why?

- **Reusable, well-defined algorithmic building blocks**
  - Encapsulate common parallel patterns with clean, composable interfaces so we don't need to rewrite them over and over
    - Ex: Parallel for-loop, reduction, transform, scan, sort, etc.
- **Advantages of standard algorithms:**
  - Higher productivity: write less code for commonly used parallel patterns
  - Better readability: algorithms are easier to understand
  - Fewer bugs: reuse well-tested, well-defined patterns



Example: Standard Algorithms in C++

- **A collection of generic, reusable algorithms provided by C++ STL[1]**
  - Mainly in `<algorithm>`, `<numeric>`, and `<execution>`
- **Common algorithm patterns:**
  - Searching and querying
  - Modification
  - Sorting and ordering
  - Set and map operations
  - Numeric algorithms
  - …
- **Parallel algorithms since C++17**
  - Execution policy
    - `std::execution::seq`,
    - `std::execution::par`,
    - `std::execution::par_unseq`



[1]: C++ standard algorithms library: https://en.cppreference.com/w/cpp/algorithm.html

Example: Standard Algorithms in C++ (cont'd)

```
#include <algorithm>
int main() {
  std::vector<int> a(1000000), b(1000000);
  std::transform(
    a.begin(), a.end(), b.begin(),
    [](int x) { return x * x; }
  );
}
```

Use `std::transform` to transform vector a into vector b by squaring each element.

We parallelize this transform operation by passing a parallel execution policy as the first argument.

```
#include <algorithm>
#include <execution>
int main() {
  std::vector<int> a(1000000), b(1000000);
  std::transform(
    std::execution::par, a.begin(), a.end(), b.begin(),
    [](int x) { return x * x; }
  );
}
```

[1]: C++ transform algorithm: https://en.cppreference.com/w/cpp/algorithm/transform.html

Example: Standard Algorithms in Intel TBB

- **TBB is a widely used C++ library for parallel programming**
  - Support many standard parallel algorithms (e.g., parallel-for, scan, reduction, etc.)

```
for(size_t i = 0; i < a.size(); ++i) {
  b[i] = a[i] * a[i];
}
```

A simple for-loop that squares each element of array `a` and stores the result in array `b`.

```
#include <tbb/parallel_for.h>
#include <tbb/blocked_range.h>
tbb::parallel_for(
  tbb::blocked_range<size_t>(0, a.size()),
  [&](const tbb::blocked_range<size_t>& r) {
    for(size_t i = r.begin(); i < r.end(); ++i) {
      b[i] = a[i] * a[i];
    }
  }
);
```

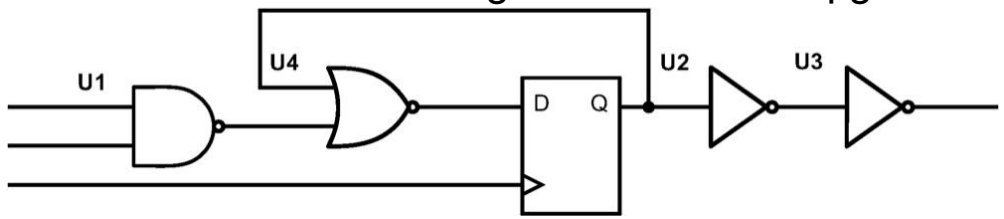
You can parallelize this for-loop using TBB's `parallel_for`. Instead of writing a loop from zero to `N`, you describe the iteration range using a `blocked_range`. TBB automatically splits this range into smaller chunks and distributes them across worker threads.

[1]: oneTBB parallel_for library: https://uxlfoundation.github.io/oneTBB/main/tbb_userguide/parallel_for_os.html

Limitations of Existing Parallel Algorithms Libraries

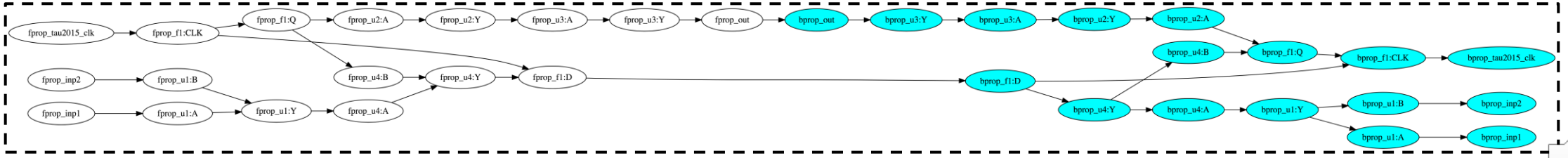
- **Expose algorithms as procedure calls rather than composable tasks**
  - Algorithms are invoked as procedures (e.g., parallel-for, reduction, sort)
  - Executions are triggered immediately at the call site
- **Composition relies on control flow and ordering, not explicit dependencies**
  - Hard to express complex workloads (e.g., task-parallel VLSI timing analysis[1])

A circuit of four comb gates and one seq gate



Derive a timing propagation task graph

Each task may run a parallel algorithm, such as parallel-for for multi-corner analysis and parallel-reduction for top-k critical values finding, but ordering these tasks along with the circuit dependencies, which can be millions of them, without explicit dependencies management is very difficult...



[1]: Tsung-Wei Huang, et al, “OpenTimer v2: A New Parallel Incremental Timing Analysis Engine,” *IEEE TCAD*, 2022

Takeaways

- **Understand the need of standard parallel algorithm tasks**
- **Learn how to create standard algorithm tasks in Taskflow**
  - Parallel for
  - Parallel reduction
  - Parallel sort
- **Launch algorithm tasks with asynchronous tasking**
- **Conclude**

Parallel-for Algorithm Task in Taskflow[1]

- **Represents a parallel execution of the following range of two versions:**
  - Index-based range: [`first`, `last`) with a positive step size or a negative step size
  - Iterator-based range: [`first`, `last`) with a step size of one

```
// index-based loop
// positive step size
for(auto i=first; i<last; i+=step) {
  callable(i);
}
// negative step size
for(auto i=first; i>last; i+=step) {
  callable(i);
}
```

```
// iterator-based loop
for(auto i=first; i<last; ++i) {
  callable(*i);
}
```

- **Applies the given callable to each index or dereferenced item**
  - No dependencies exist between iterations

[1]: Parallel-for algorithm task in Taskflow: https://taskflow.github.io/taskflow/ParallelIterations.html

Index-based Parallel-for Algorithm Task

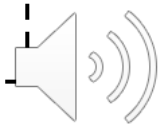
- **tf::Taskflow::for_each_index(first, last, step_size, callable)**

```
// 50 parallel iterations at 0, 2, 4, 6, 8, ..., 46, 48
tf::Task t1 = taskflow.for_each_index(0, 100,  2, [](int i) {});

// 50 parallel iterations at 100, 98, 96, 94, ..., 4, 2
tf::Task t2 = taskflow.for_each_index(100, 0, -2, [](int i) {});
```

- **tf::Taskflow::for_each_by_index(index_range, callable)**

```
// initialize each element in data to 10 using tf::IndexRange
std::vector<int> data;
tf::IndexRange<int> range(0, 100, 1);  // an index range [0, 100) of step size one
tf::Task t = taskflow.for_each_by_index(range, [&](tf::IndexRange<int> subrange){
  for(int i=subrange.begin(); i<subrange.end(); i+=subrange.step_size()) {
    data[i] = 10;
  }
});
```



Iterator-based Parallel-for Algorithm Task

- **tf::Taskflow::for_each(first, last, callable)**

```
// iterates every integer in a vector simultaneously
std::vector<int> vec = {1, 2, 3, 4, 5};
auto t1 = taskflow.for_each(vec.begin(), vec.end(), [](int item){
  printf("parallel for on item %d", item);
});
// iterates every string in a list simultaneously
std::list<std::string> list = {"hi", "from", "t", "a", "s", "k", "f", "low"};
auto t2 = taskflow.for_each(list.begin(), list.end(), [](const std::string& str){
  printf("parallel for on item %s", str.c_str());
});
// asks the first for-each task to run before the second for-each task
t1.precede(t2);
```

This is the key difference between Taskflow's algorithm tasks and existing libraries, where Taskflow allows algorithm tasks to be composed with other tasks in the task graph.

Capture Range by Reference using `std::ref`

- **Useful when the range is unknown at the time of creating a parallel-for task**
  - Ex: Input data needs to be loaded from a file whose range is only available at runtime

```
std::vector<int> vec;
std::vector<int>::iterator first, last;

tf::Task init = taskflow.emplace([&](){
  vec.resize(1000);
  first = vec.begin();
  last  = vec.end();
});

tf::Task task = taskflow.for_each(std::ref(first), std::ref(last), [&](int i) {
  printf("parallel iteration on item %d\n", item);
});

init.precede(task);
```

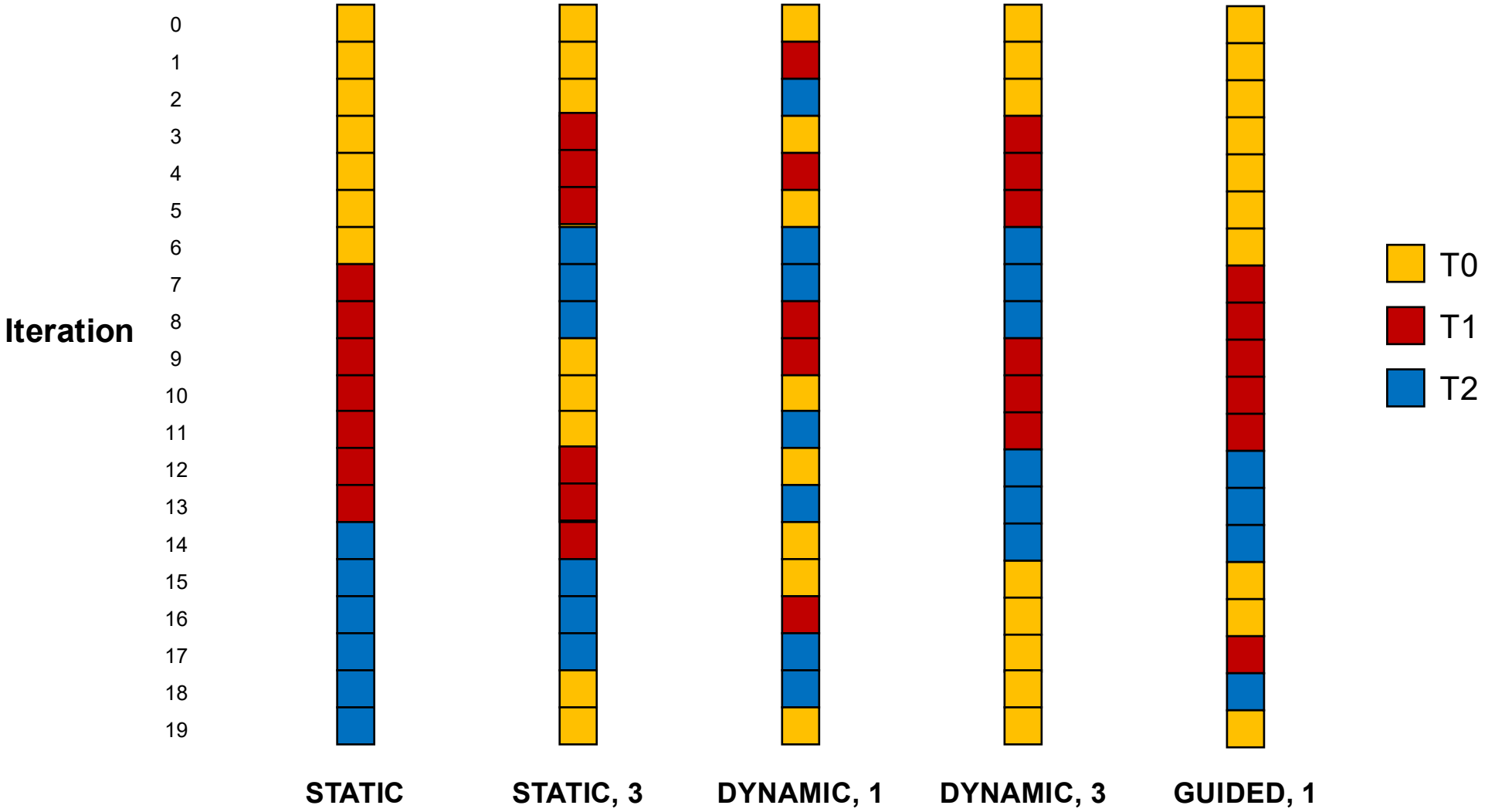
The initialization task sets up the range, after which another task performs the parallel-for operation.

The parallel-for task takes the range by reference using `std::ref`, reflecting the changes made by the initialization task.

Handle the Load-balancing Problem

- **When per-iteration work varies a lot, we have the load-balancing problem**
  - A common problem in parallelizing irregular applications (e.g., sparse algebra, rendering)
- **Taskflow partitions iterations to *chunks* and assign them to worker threads**
  - **Static partitioning** (`tf::StaticPartitioner`)
    - Each thread grabs a chunk of iterations in a round-robin fashion, where each round has a fixed number of iterations equal to `num_workers*chunk_size`
    - Low scheduling and synchronization overhead, but may cause load imbalance
  - **Dynamic partitioning** (`tf::DynamicPartitioner`)
    - Each thread grabs a chunk of iterations and request the next chunk of iterations in no particular order when finished
    - Higher scheduling overhead (e.g., `chunk_size = 1`), but can reduce load imbalance
  - **Guided partitioning** (`tf::GuidedPartitioner`)
    - Threads are initially assigned large chunks of iterations to reduce scheduling overhead; as threads complete their chunks, subsequent chunks gradually become smaller to prevent load imbalance
    - Decent scheduling overhead, and can balance the load in most applications

Visualization of Different Partitioners



Configure a Partitioning Method

- **Pass a partitioner as the last argument to a parallel-for task**
  - Static, dynamic, and guided partitioners

```
std::vector<int> data = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

// creates different partitioners
// 1. Guided partitioner with a chunk size of 1
// 2. Static partitioner without any chunk size specified (evenly distributed)
// 3. Dynamic partitioner with a chunk size of 5
tf::GuidedPartitioner guided_partitioner(1);
tf::StaticPartitioner static_partitioner;
tf::DynamicPartitioner dynamic_partitioner(5);

// creates four parallel-iteration tasks from the four execution policies
taskflow.for_each(data.begin(), data.end(), [](int i){}, guided_partitioner);
taskflow.for_each(data.begin(), data.end(), [](int i){}, static_partitioner);
taskflow.for_each(data.begin(), data.end(), [](int i){}, dynamic_partitioner);
```

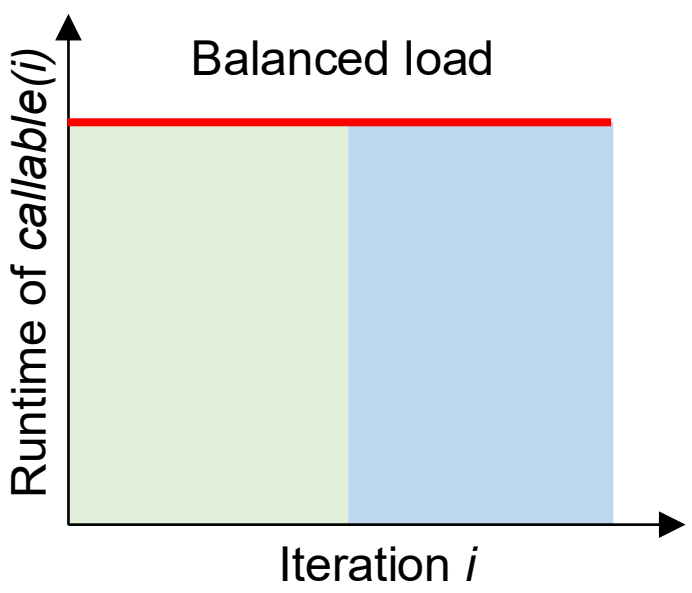
When constructing a partitioner, you can specify a chunk size.

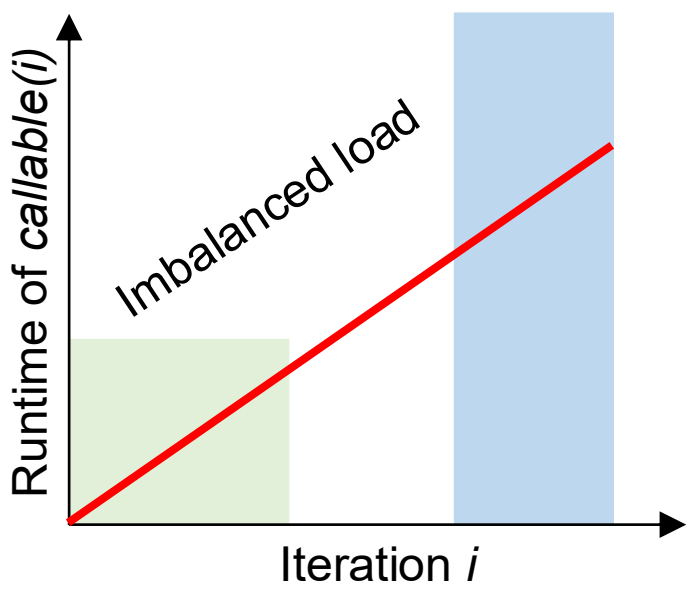
Which Partitioning Method to Use?

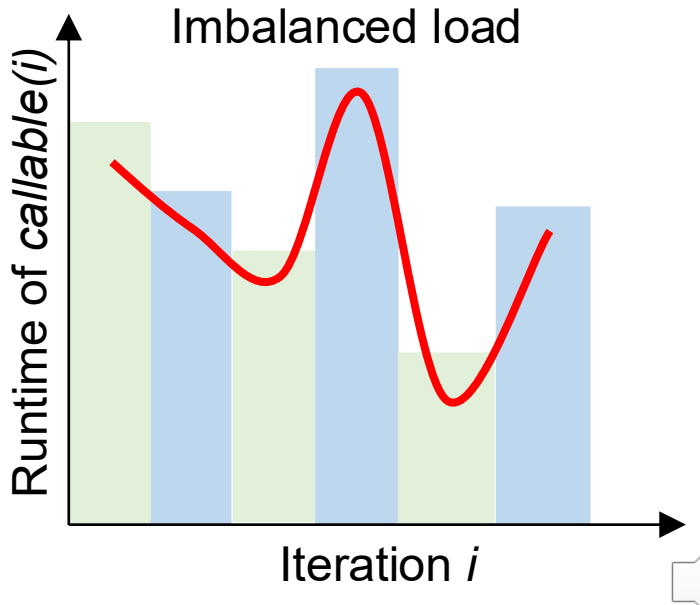
- **The right partitioner really depends on the workload characteristics**
  - Uniform work → static partitioning (scheduling overhead is the least)
  - Irregular work → dynamic or guided partitioning (work is redistributed at runtime)

Static partitioner is the best for balanced load.

Dynamic partitioner or guided partitioner (Taskflow's default) are often better for imbalanced load than a static partitioner.







Takeaways

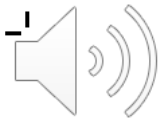
- **Understand the need of standard parallel algorithm tasks**
- **Learn how to create standard algorithm tasks in Taskflow**
  - Parallel for
  - Parallel reduction
  - Parallel sort
- **Launch algorithm tasks with asynchronous tasking**
- **Conclude**

Parallel-reduction Algorithm Task in Taskflow[1]

- **Represents a parallel execution of the following reduction loop:**

```
// reduces a range of items [first, last) to a result using a binary operator bop
for(auto i=first; i<last; ++i) {
  result = bop(result, *i);
}
```

```
// creates a parallel-reduction task that accumulates all vector elements into
// the initial value sum
int sum = 100;
std::vector<int> vec = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
tf::Task task = taskflow.reduce(vec.begin(), vec.end(), sum,
  [] (int l, int r) { return l + r; }
);
executor.run(taskflow).wait();
assert(sum == 100 + 55);
```



[1]: Parallel-reduction algorithm task in Taskflow: https://taskflow.github.io/taskflow/ParallelReduction.html

Capture Range by Reference using `std::ref`

- **Useful when the range is unknown at the time of creating a reduction task**
  - Ex: Input data needs to be loaded from a file whose range is only available at runtime

```
int sum = 100;
std::vector<int> vec;
std::vector<int>::iterator first, last;
tf::Task init = taskflow.emplace([&](){
  vec   = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  first = vec.begin();
  last  = vec.end();
});
tf::Task task = taskflow.reduce(std::ref(first), std::ref(last), sum,
  [] (int l, int r) { return l + r; }
);
init.precede(task);
executor.run(taskflow).wait();
assert(sum == 100 + 55);
```

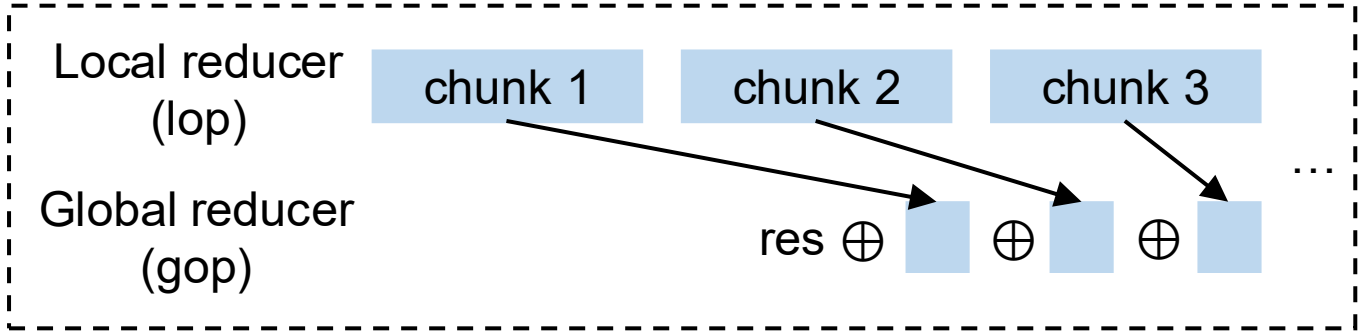
The initialization task sets up the range, after which another task performs the parallel-reduction task.

The parallel-reduction task takes the range by reference, reflecting the changes made by the initialization task.

Index-based Parallel-reduction Algorithm Task

- **tf::Taskflow::reduce_by_index(range, res, lop, gop)**

```
std::vector<double> data(1000000, 1.0);
double res = 1.0;
taskflow.reduce_by_index(tf::IndexRange<size_t>(0, 1000000, 1), res,
  // local reducer (lop)
  [&](tf::IndexRange<size_t> subrange, std::optional<double> running_total) {
    double residual = running_total ? *running_total : 0.0;
    for(size_t i=subrange.begin(); i<subrange.end(); i+=subrange.step_size()) {
      residual += data[i];
    }
    return residual;
  },
  // global reducer (gop)
  std::plus<double>()
);
executor.run(taskflow).wait();
assert(res == 1000001);
```

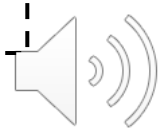


Configure a Partitioning Method

- **Pass a partitioner as the last argument to the parallel-reduction task**
  - Static, dynamic, and guided partitioners

```
tf::StaticPartitioner static_partitioner(10);
tf::GuidedPartitioner guided_partitioner;
int sum1 = 100, sum2 = 100;
std::vector<int> vec = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
// create a parallel-reduction task with a static partitioner of a chunk size of 10
auto t1 = taskflow.reduce(vec.begin(), vec.end(), sum1,
  [] (int l, int r) { return l + r; },
  static_partitioner
);
// create a parallel-reduction task with guided partitioner
auto t2 = taskflow.reduce(vec.begin(), vec.end(), sum2,
  [] (int l, int r) { return l + r; },
  guided_partitioner
);
t1.precede(t2);
```

When constructing a partitioner, you can specify a chunk size.



Takeaways

- **Understand the need of standard parallel algorithm tasks**
- **Learn how to create standard algorithm tasks in Taskflow**
  - Parallel for
  - Parallel reduction
  - Parallel sort
- **Launch algorithm tasks with asynchronous tasking**
- **Conclude**

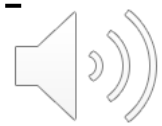
Parallel-sort Algorithm Task in Taskflow

- **Performs parallel sort to rank a range of elements in increasing order**
  - Basically, a parallel version of `std::sort`

```
tf::Taskflow taskflow;
tf::Executor executor;
std::vector<int> data = {1, 4, 9, 2, 3, 11, -8};
// creates a parallel-sort task to sort the data vector in increasing order
tf::Task sort = taskflow.sort(data.begin(), data.end());
executor.run(taskflow).wait();
```

- **A general-purpose soring algorithm that accepts a custom comparator**

```
tf::Task sort = taskflow.sort(data.begin(), data.end(),
  [](int a, int b) { return a > b; }  // sort the data vector in decreasing order
);
executor.run(taskflow).wait();
```



[1]: Parallel-sort algorithm task in Taskflow: https://taskflow.github.io/taskflow/ParallelSort.html

Capture Range by Reference using `std::ref`

- **Useful when the range is unknown at the time of creating a sort task**
  - Ex: Input data needs to be loaded from a file whose range is only available at runtime

```
tf::Taskflow taskflow;
tf::Executor executor;
std::vector<int> data;
std::vector<int>::iterator first, last;
tf::Task init = taskflow.emplace([&](){
  data  = {1, 4, 9, 2, 3, 11, -8};
  first = data.begin();
  last  = data.end();
});
tf::Task sort = taskflow.sort(
  std::ref(first), std::ref(last), [] (int l, int r) { return l < r; }
);
init.precede(sort);
executor.run(taskflow).wait();
assert(std::is_sorted(data.begin(), data.end()));
```

The initialization task sets up the range, after which another task performs the parallel-sort task.

The parallel-sort task takes the range by reference, reflecting the changes made by the initialization task.

Other Algorithm Tasks in Taskflow

- **Parallel-transform algorithm task**
  - Performs parallel transforms over a range of items
- **Parallel-find algorithm task**
  - Performs parallel search for an item in a range
- **Parallel-scan algorithm task**
  - Performs parallel prefix sum over a range of items
- **Parallel-pipeline algorithm task**
  - Performs pipeline parallelism over a series of data tokens
- **Module algorithm task**
  - Encapsulates a task graph
- **... more[1]**

[1]: Taskflow Algorithms: https://taskflow.github.io/taskflow/Algorithms.html

Takeaways

- **Understand the need of standard parallel algorithm tasks**
- **Learn how to create standard algorithm tasks in Taskflow**
  - Parallel for
  - Parallel reduction
  - Parallel sort
- **Launch algorithm tasks with asynchronous tasking**
- **Conclude**

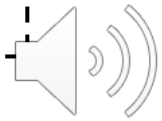
All Algorithm Tasks can Work with Async Tasks

- **Very useful when you need to launch an algorithm task on the fly**
  - `algorithm` (e.g., `for_each`) → `make_algorithm_task` (e.g., `make_for_each_task`)

```
tf::Executor executor;
std::vector<int> vec(1000000);

// launch a parallel-for task asynchronously to initialize a vector randomly
auto fu1 = executor.async(tf::make_for_each_task(
  vec.begin(), vec.end(), [](int& item){ item = rand(); }
));
f1.wait();

// launch a parallel-sort task asynchronously to sort the vecctor
auto fu2 = executor.async(
  tf::make_sort_task(vec.begin(), vec.end(), std::less<int>{})
);
fu2.wait();
```



Can also Work with Dependent-async Tasks

- **Useful for asynchronously launching algorithm tasks with dependencies**

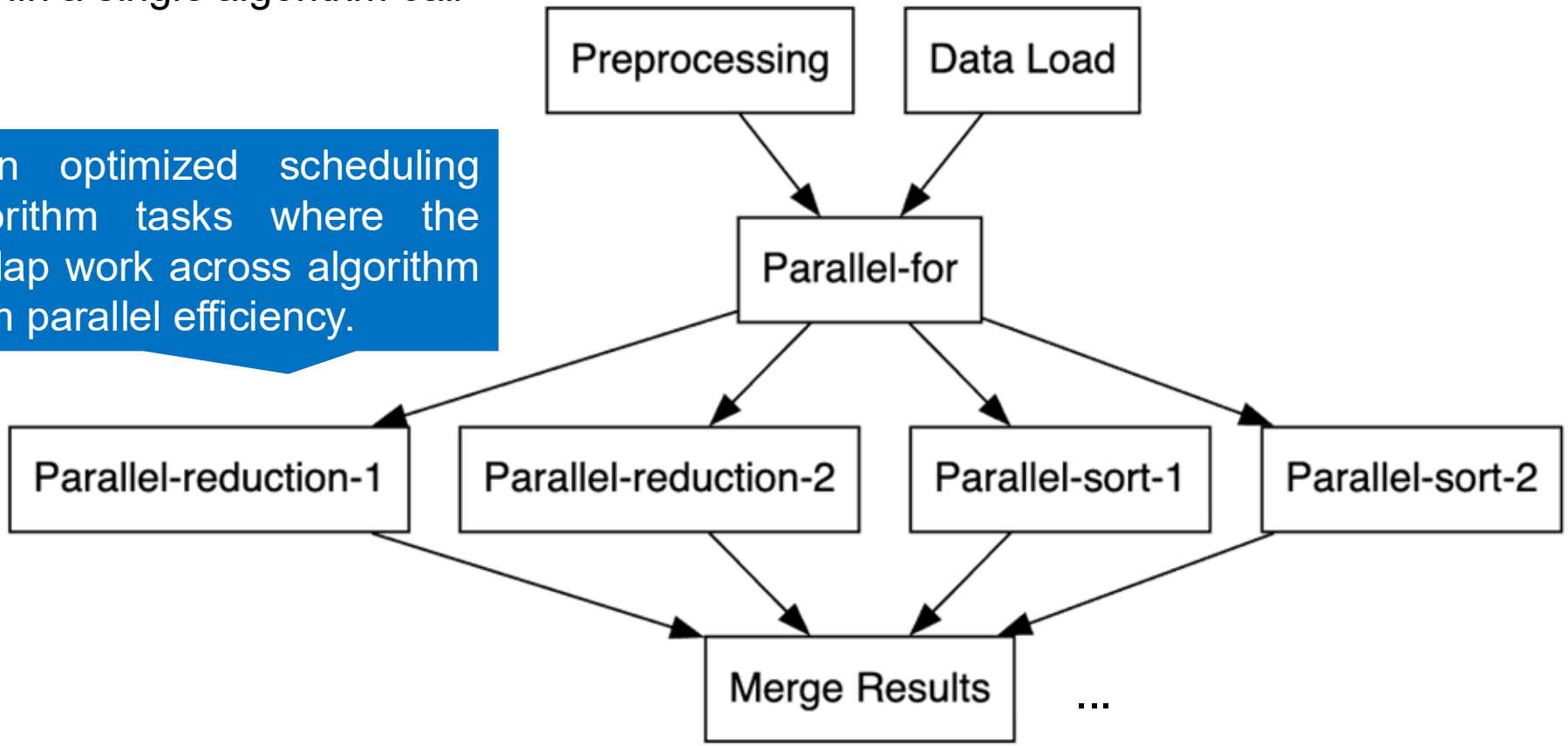
```
tf::Executor executor;
std::vector<int> vec(2000000);

// launches two parallel-for tasks asynchronously to initialize a vector randomly
auto init1 = executor.silent_dependent_async(tf::make_for_each_task(
  vec.begin(), vec.begin() + 1000000, [](int& item){ item = rand(); }
));
auto init2 = executor.silent_dependent_async(tf::make_for_each_task(
  vec.begin() + 1000000, vec.end(), [](int& item){ item = rand(); }
));
// launches a parallel-sort task asynchronously after initialization is done
auto [task, future] = executor.dependent_async(tf::make_sort_task(
  vec.begin(), vec.end(), std::less<int>{}
), init1, init2);
// waits for the parallel-sort task finishes
future.wait();
```

Compose Parallel Algorithms as a Dynamic Task Graph

- **Expose task-level parallelism across different algorithm tasks**
  - Not just within a single algorithm call

Taskflow has an optimized scheduling method for algorithm tasks where the executor will overlap work across algorithm tasks for maximum parallel efficiency.



Takeaways

- **Understand the need of standard parallel algorithm tasks**
- **Learn how to create standard algorithm tasks in Taskflow**
  - Parallel for
  - Parallel reduction
  - Parallel sort
- **Launch algorithm tasks with asynchronous tasking**
- **Conclude**

Question?

Static Task Graph Programming (STGP)

```
// Live: https://godbolt.org/z/j8hx3xnnx

tf::Taskflow taskflow;
tf::Executor executor;
auto [A, B, C, D] = taskflow.emplace(
  [](){ std::cout << "TaskA\n"; }
  [](){ std::cout << "TaskB\n"; },
  [](){ std::cout << "TaskC\n"; },
  [](){ std::cout << "TaskD\n"; }
);

A.precede(B, C);
D.succeed(B, C);
executor.run(taskflow).wait();
```

Taskflow: https://taskflow.github.io

Dynamic Task Graph Programming (DTGP)

```
// Live: https://godbolt.org/z/T87PrTarx

tf::Executor executor;
auto A = executor.silent_dependent_async([]{
  std::cout << "TaskA\n";
});
auto B = executor.silent_dependent_async([]{
  std::cout << "TaskB\n";
}, A);
auto C = executor.silent_dependent_async([]{
  std::cout << "TaskC\n";
}, A);
auto D = executor.silent_dependent_async([]{
  std::cout << "TaskD\n";
}, B, C);
executor.wait_for_all();
```