



# Recursive Task Parallelism 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

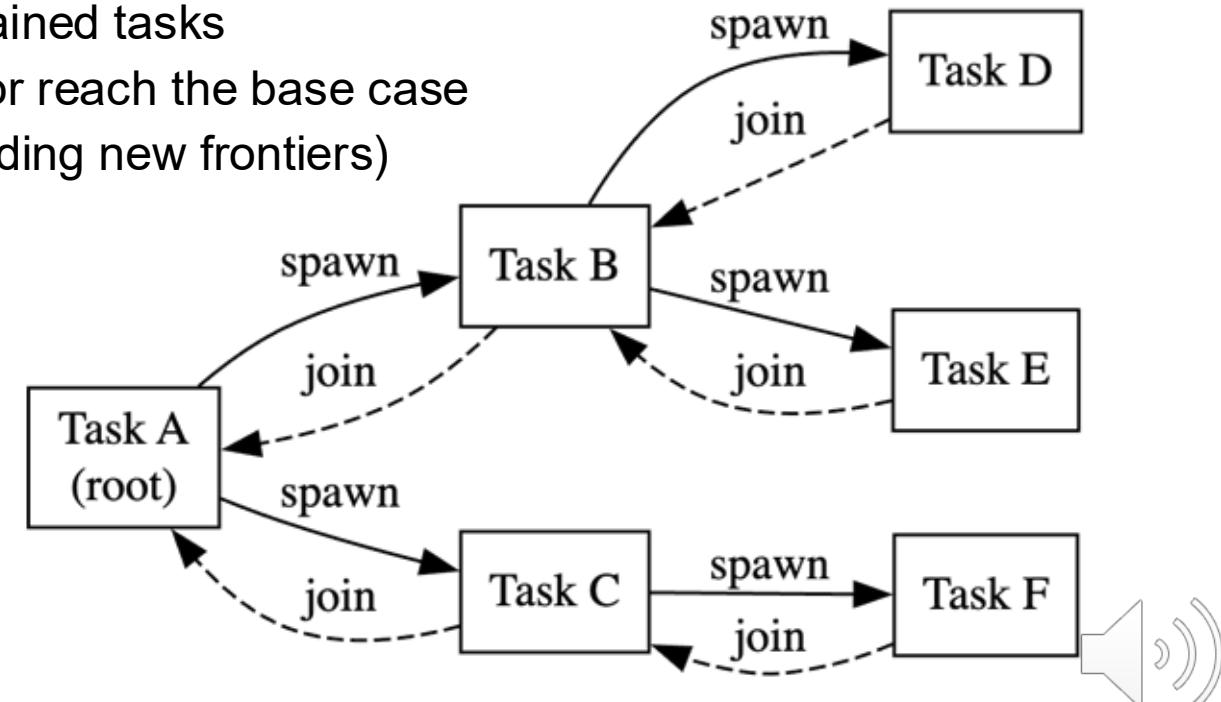
---

- Learn how to program recursive task parallelism in Taskflow
- Understand the role of cooperative execution in recursive task parallelism
- Showcase two real-world applications of recursive task parallelism
- Conclude the talk



# Recursive Task Parallelism (RTP): What and Why?

- RTP expresses parallelism through recursive task decomposition
  - A parent task recursively spawns multiple child tasks and joins their executions
- RTP is a powerful programming model for parallelizing irregular workloads
  - Algorithms that leverage **divide-and-conquer** to solve problems with the following patterns:
    1. Recursively partitions a problem into smaller blocks of interest
    2. Each partitioned block spawns finer-grained tasks
    3. Join when interested regions stabilize or reach the base case
  - Graph traversals (e.g., spawn tasks when finding new frontiers)
  - Recursive search (e.g., n-queens)
  - Adaptive mesh (e.g., PDE solver)
  - High-performance matrix multiplication
  - Dynamic programming
- Many libraries support RTP
  - Taskflow, OpenCilk, TBB, OpenMP, etc.

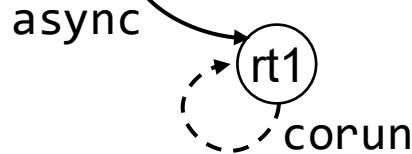




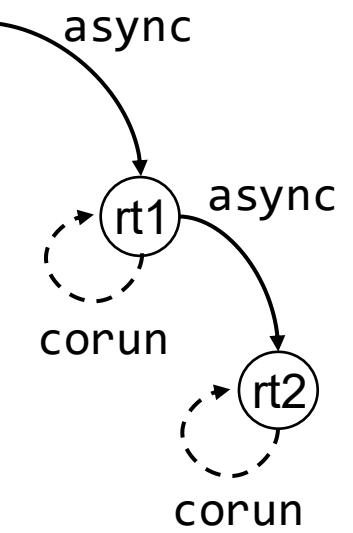
# RTP in Taskflow: Runtime Task

- A task taking a reference to a `tf::Runtime` object created by its executor
- A runtime task allows you to interact with the scheduling runtime
  - Performs async tasking via `tf::Runtime::async` / `tf::Runtime::dependent_async`
  - Performs cooperative execution via `tf::Runtime::corun`
- All tasks spawned from a runtime task are parented to that runtime task
  - Allows you to explicitly join child tasks at any parent or ancestor in the call tree

```
auto task = taskflow.emplace( [] (tf::Runtime& rt) {  
    rt.async([](){});  
    rt.corun();  
}  
);
```



```
auto task = taskflow.emplace( [] (tf::Runtime& rt1) {  
    rt1.async([](tf::Runtime& rt2){  
        rt2.async([](){});  
        rt2.corun();  
    });  
    rt1.corun();  
});
```





# Implicit Synchronization of a Runtime Task

- All tasks spawned by a runtime task are implicitly joined at scope exit

```
tf::Executor executor(num_threads);
tf::Taskflow taskflow;
std::atomic<size_t> counter(0);

tf::Task A = taskflow.emplace([&](tf::Runtime& rt){
    // spawn 1000 asynchronous tasks from this runtime task
    for(size_t i=0; i<1000; i++) {
        rt.silent_async([&](){ counter.fetch_add(1, std::memory_order_relaxed); });
    }
    // all tasks spawned from rt will join at the end of its scope
});
tf::Task B = taskflow.emplace([&](){ assert(counter == 1000); });

A.precede(B);
executor.run(taskflow).wait();
```

Those 1000 asynchronous tasks are guaranteed to finish after leaving the scope of runtime task A.





# Create a Dynamic Task Graph within a Runtime Task

```
tf::Task X = taskflow.emplace([&](tf::Runtime& rt){
    // create a dynamic task graph of four tasks, A, B, C, and D1
    // A runs before B and C, and D runs after B and C
    tf::AsyncTask A = rt.silent_dependent_async([&](){ std::cout << "A\n"; });
    tf::AsyncTask B = rt.silent_dependent_async([&](){ std::cout << "B\n"; }, A);
    tf::AsyncTask C = rt.silent_dependent_async([&](){ std::cout << "C\n"; }, A);
    tf::AsyncTask D = rt.silent_dependent_async([&](){ std::cout << "D\n"; }, B, C);
    // all tasks spawned from rt will join at the end of its scope
});
tf::Task Y = taskflow.emplace([&](){
    std::cout << "A, B, C, and D must have already finished\n";
});
X.precede(Y);
executor.run(taskflow).wait();
```

The dynamic task graph created within X is guaranteed to finish after leaving the scope of runtime task X.

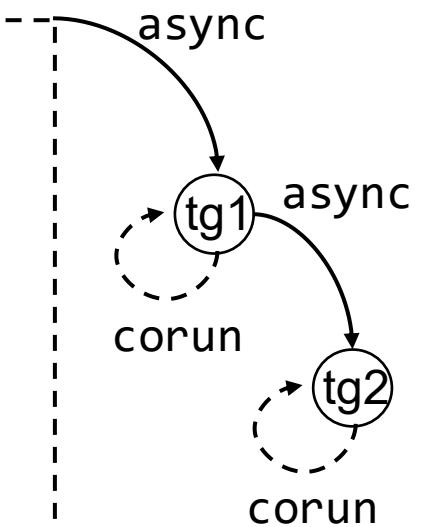


# RTP in Taskflow: Task Group

- A more **self-managed mechanism than `tf::Runtime` to perform RTP**
  - Creates a `tf::TaskGroup` explicitly rather than receiving it as an input argument
  - Performs async tasking via `tf::TaskGroup::async` / `tf::TaskGroup::dependent_async`
  - Performs cooperative execution via `tf::TaskGroup::corun`
- **All tasks spawned from a task group are parented to that task group**
  - Allows you to explicitly join child tasks at any parent or ancestor in the call tree

```
auto task = taskflow.emplace([&executor]() {
    tf::TaskGroup task_group_level1 = executor.task_group();
    task_group.silent_async([&](){
        tf::TaskGroup task_group_level2 = executor.task_group();
        task_group_level2.silent_async([](){});
        task_group_level2.corun();
    });
    task_group_level1.corun();
});
```

Usage of `tf::TaskGroup` is similar to `tf::Runtime`, except that it can be explicitly created and managed





# Important Semantics of the Task Group

- A `tf::TaskGroup` can only be created inside a task context

- Otherwise, a non-worker thread may mistakenly participate in the work-stealing loop

```
executor.silent_async([&](){  
    auto tg = executor.task_group();  
    ...  
});
```

vs

```
auto tg = executor.task_group();  
...  
); // will throw, since tg is not  
// created inside a task context
```

- While created by an executor, the executor does not own any task group

```
auto task = taskflow.emplace([&]() {  
    std::atomic<size_t> n(0);  
    auto tg = executor.task_group();  
    tg.silent_async([&](){ n++; });  
    tg.silent_async([&](){ n++; });  
    tg.corun();  
    assert(n == 2); // pass  
}); // tg is safely destroyed
```

vs

```
auto task = taskflow.emplace([&]() {  
    std::atomic<size_t> n(0);  
    auto tg = executor.task_group();  
    tg.silent_async([&](){ n++; });  
    tg.silent_async([&](){ n++; });  
    assert(n==2); // undefined behavior  
}); // tg is destroyed while its tasks  
// may still be running
```





# Takeaways

---

- Learn how to program recursive task parallelism in Taskflow
- Understand the role of cooperative execution in recursive task parallelism
- Showcase two real-world applications of recursive task parallelism
- Conclude the talk





# Cooperative Execution within a Runtime Task

- Runtime task offers a corun method to perform cooperative execution

- Same as tf::Executor::corun, the calling worker of the runtime cooperatively executes other available tasks while waiting for all tasks spawned from that runtime to complete
- The caller worker *cooperatively* participates in the execution by helping other workers in the same executor to make overall progress

```
tf::Executor executor(2);
tf::Taskflow taskflow;
std::array<tf::Taskflow, 1000> others;

// non-cooperative execution: deadlock
for(size_t n=0; n<1000; n++) {
    taskflow.emplace([&, &tf=others[n]]{
        executor.run(tf).wait();
    });
} May incur deadlock w/o cooperative execution!
executor.run(taskflow).wait();
```

```
tf::Executor executor(2);
tf::Taskflow taskflow;
std::array<tf::Taskflow, 1000> others;

// cooperative execution: no deadlock
for(size_t n=0; n<1000; n++) {
    taskflow.emplace([&, &tf=others[n]]{
        (tf::Runtime& rt){
            rt.corun(tf);
        });
} No deadlock with cooperative execution!
executor.run(taskflow).wait();
```





# Corun Asynchronous Tasks from a Runtime Task

- While preserving the caller's execution context and stack state

```
size_t fib(size_t N, tf::Runtime& rt) {  
    if(N < 2) return N;  
    size_t res1, res2;  
    rt.silent_async([N, &res1](tf::Runtime& rt1){ res1 = fib(N-1, rt1); });  
    rt.silent_async([N, &res2](tf::Runtime& rt2){ res2 = fib(N-2, rt2); });  
    rt.corun();  
    return res1 + res2;  
}  
  
int main() {  
    tf::Executor executor;      Each asynchronous task can itself be a runtime task.  
    size_t N = 4, res;  
    executor.silent_async([N, &res](tf::Runtime& rt){ res = fib(N, rt); });  
    executor.wait_for_all();  
    std::cout << N << "-th Fibonacci number is " << res << '\n';  
    return 0;  
}
```

The runtime task spawns two asynchronous tasks to compute  $\text{fib}(N-1)$  and  $\text{fib}(N-2)$ .

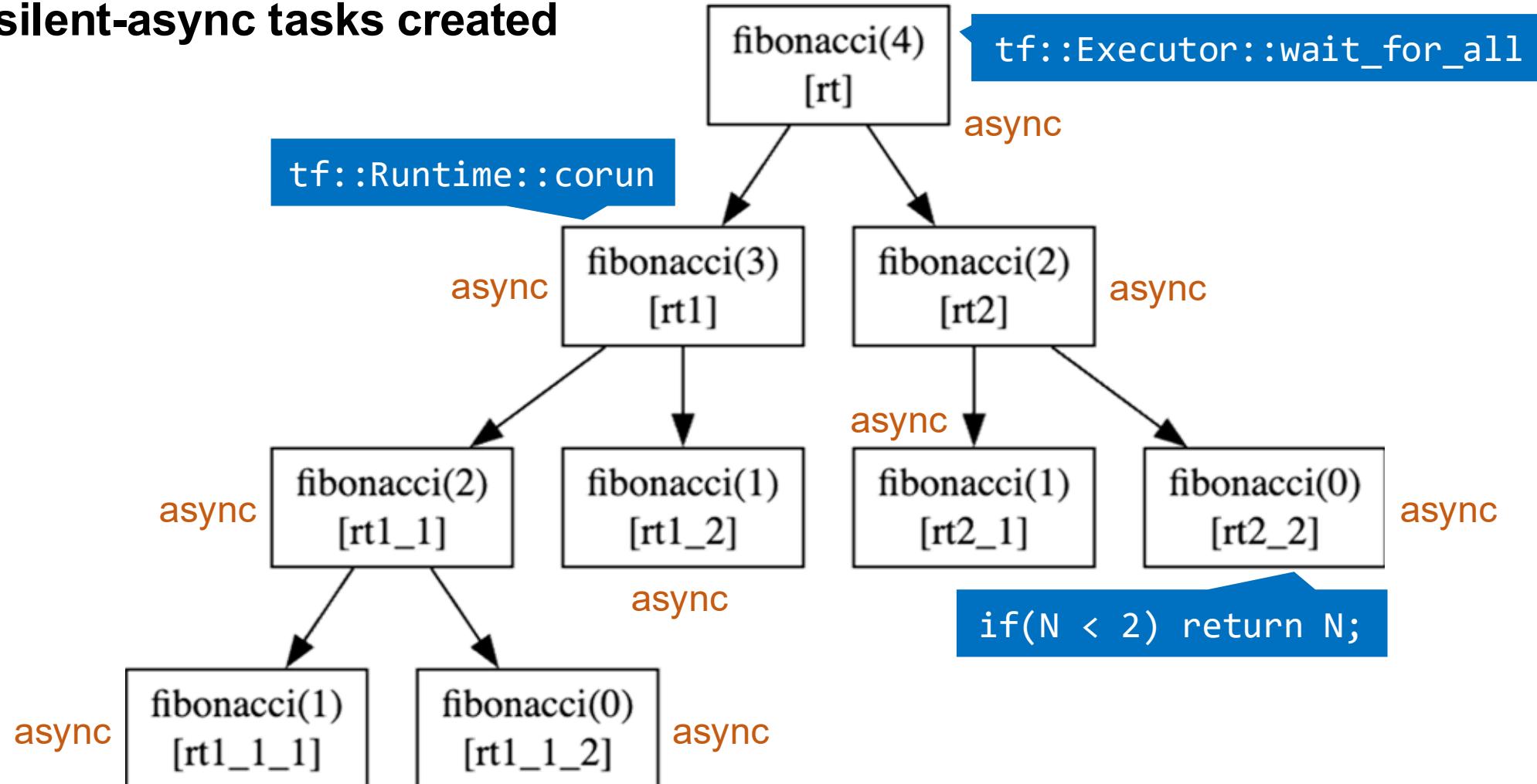
Each asynchronous task can itself be a runtime task.

Each asynchronous task can itself be a runtime task.



# Execution Diagram

- 9 silent-async tasks created





# Reduce Tasking Overhead through Tail Optimization

```
size_t fib(size_t N, tf::Runtime& rt) {
    if(N < 2) return N;
    size_t res1, res2;
    rt.silent_async([N, &res1](tf::Runtime& rt1){ res1 = fib(N-1, rt1); });
    // tail optimization for the right child
    res2 = fib(N-2, rt);
    rt.corun();
    return res1 + res2;
}
int main() {
    tf::Executor executor;
    size_t N = 30, res;
    executor.silent_async([N, &res](tf::Runtime& rt){ res = fib(N, rt); });
    executor.wait_for_all();
    std::cout << N << "-th Fibonacci number is " << res << '\n';
    return 0;
}
```

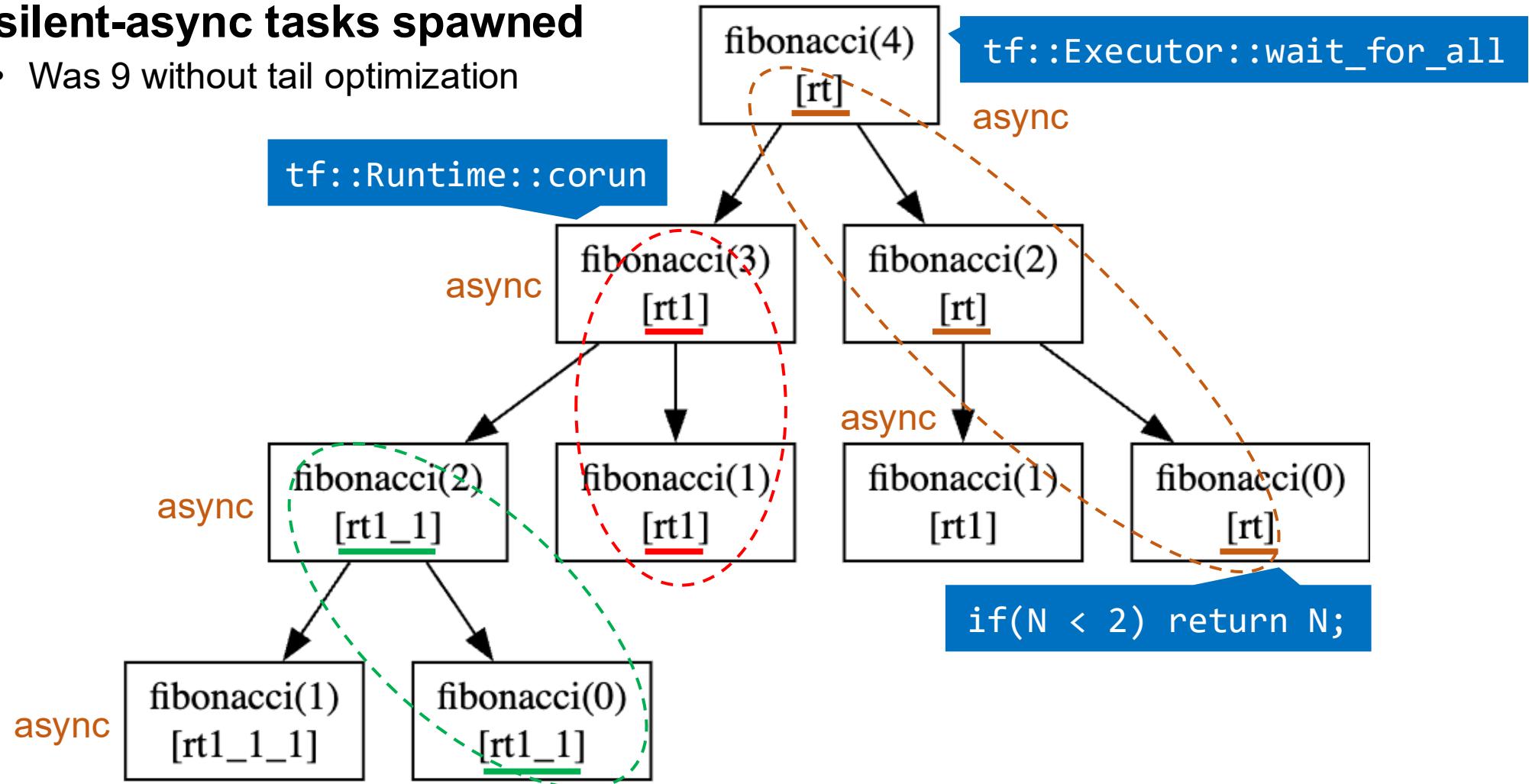
At the same time, the left recursive call still runs asynchronously, and we maintain the same parallelism.

Instead of spawning a new asynchronous task for `fib(N-2)`, we compute it directly in the current runtime task. This reduces the number of created asynchronous tasks, which improves the performance dramatically.



# Execution Diagram w/ Tail Optimization

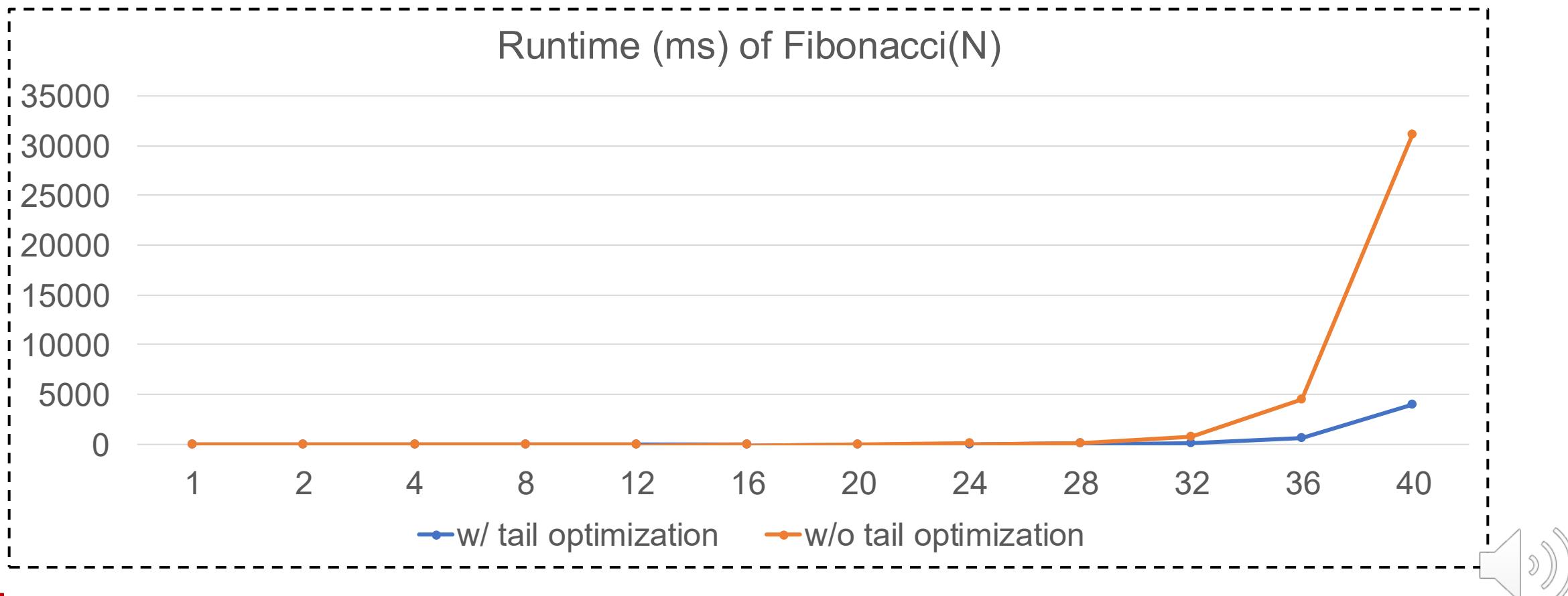
- **5 silent-async tasks spawned**
    - Was 9 without tail optimization





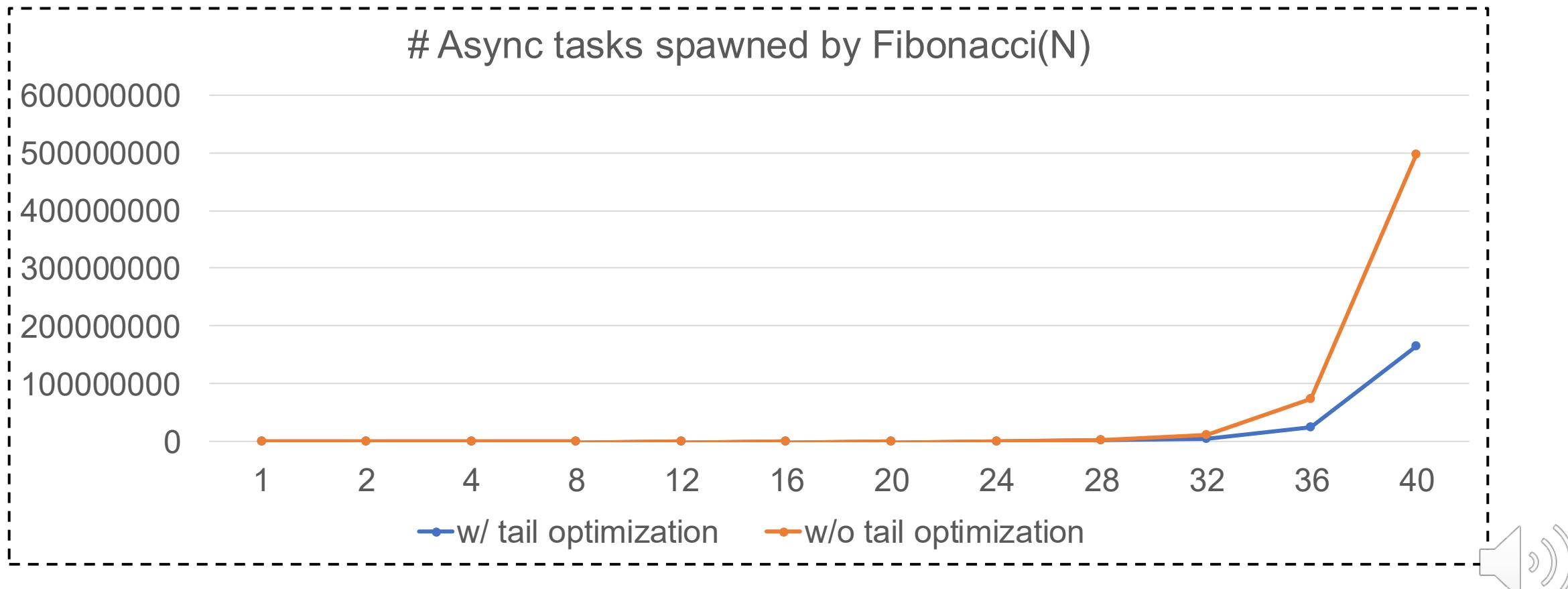
# How Good is Tail Optimization?

- Measured at different Fibonacci numbers up to 40 under 16 threads
  - 34%–105% performance improvement over the implementation w/o tail optimization



# How Good is Tail Optimization? (cont'd)

- **Largely improved performance due to largely reduced # of async tasks**
  - Ex: 2x fewer tasks at the 40<sup>th</sup> Fibonacci number (165,580,141 vs 331,160,281)





# Implementation using Task Group

```
size_t fib(size_t N, tf::Executor& executor) {
    if(N < 2) return N;
    size_t res1, res2;
    tf::TaskGroup tg = executor.task_group();
    tg.silent_async([N, &res1](){ res1 = fib(N-1, executor); });
    res2 = fib(N-2, executor);
    tg.corun();
    return res1 + res2;
}

int main() {
    tf::Executor executor;
    size_t N = 30, res;
    res = executor.async([N, &executor](){ return fib(N, executor); }).get();
    std::cout << N << "-th Fibonacci number is " << res << '\n';
    return 0;
}
```

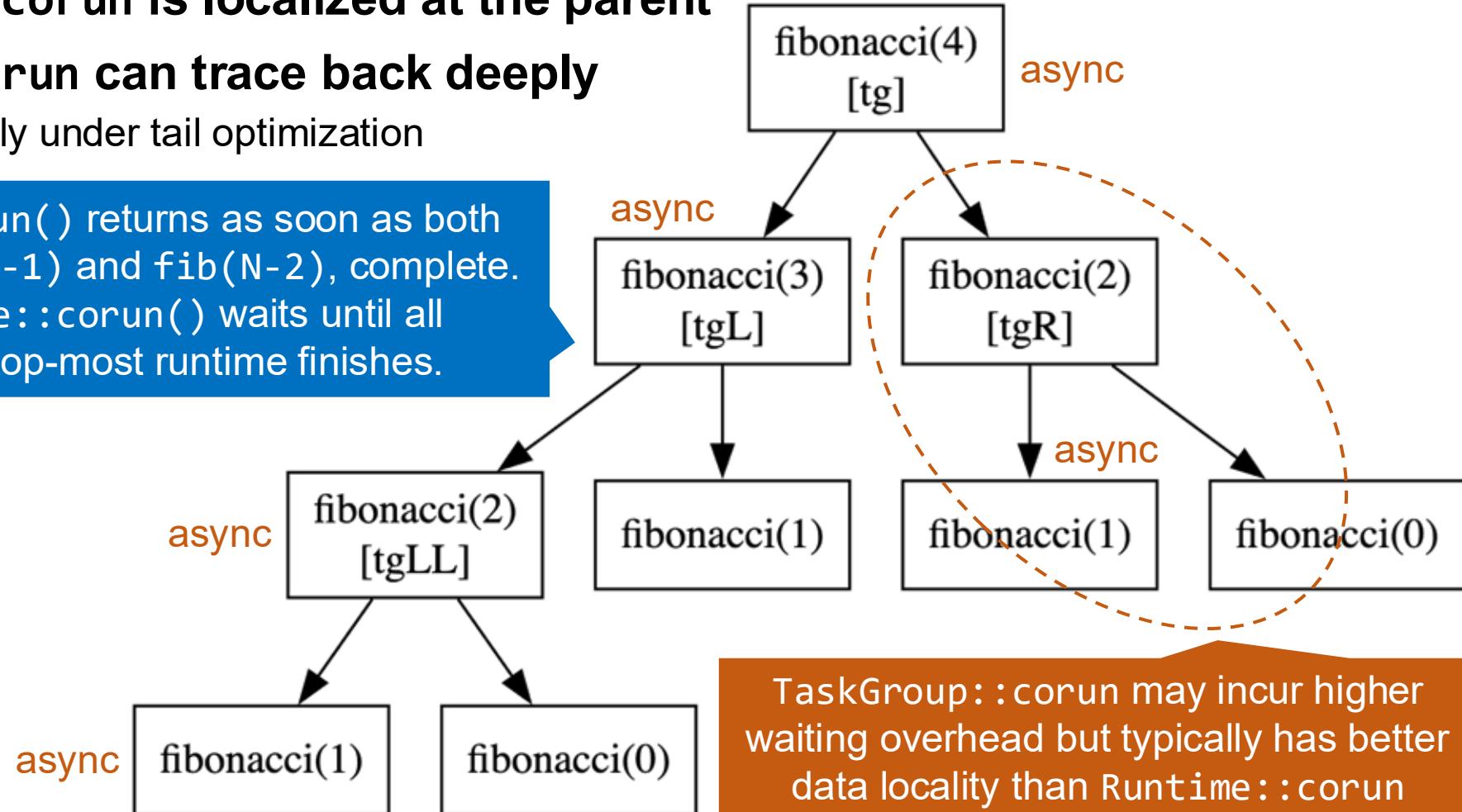
Similar to the one implemented using runtime tasking, except that we need to pass the executor as an input argument so we can create a task group inside the Fibonacci function.



# Difference between Task Group and Runtime

- **TaskGroup::corun is localized at the parent**
- **Runtime::corun can trace back deeply**
  - Happens only under tail optimization

TaskGroup::corun() returns as soon as both child tasks, fib(N-1) and fib(N-2), complete. However, Runtime::corun() waits until all child tasks of the top-most runtime finishes.





# Takeaways

---

- Learn how to program recursive task parallelism in Taskflow
- Understand the role of cooperative execution in recursive task parallelism
- Showcase two real-world applications of recursive task parallelism
- Conclude the talk



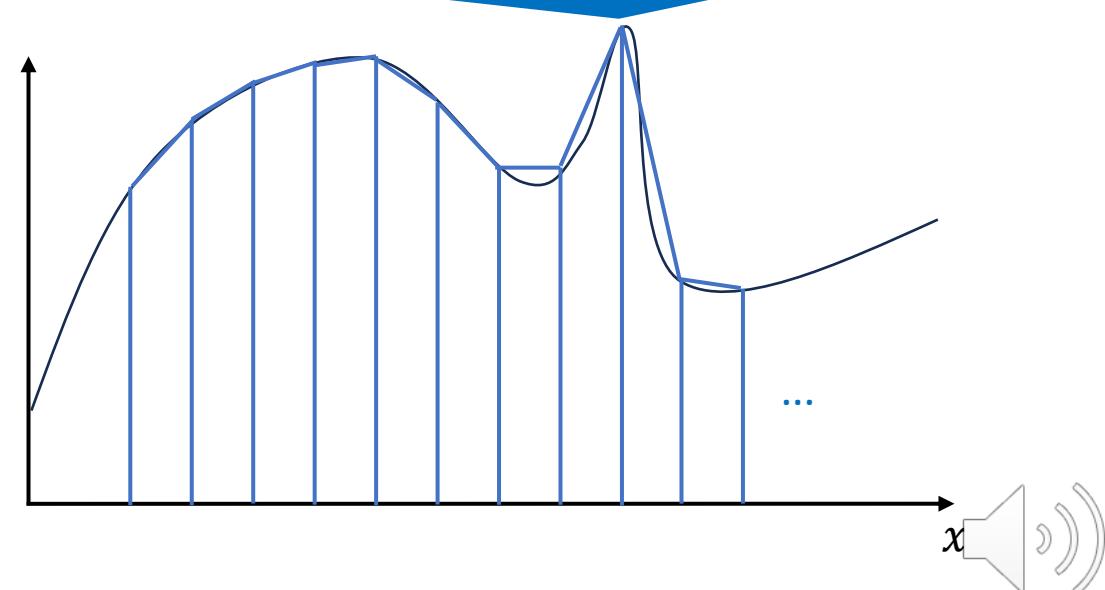
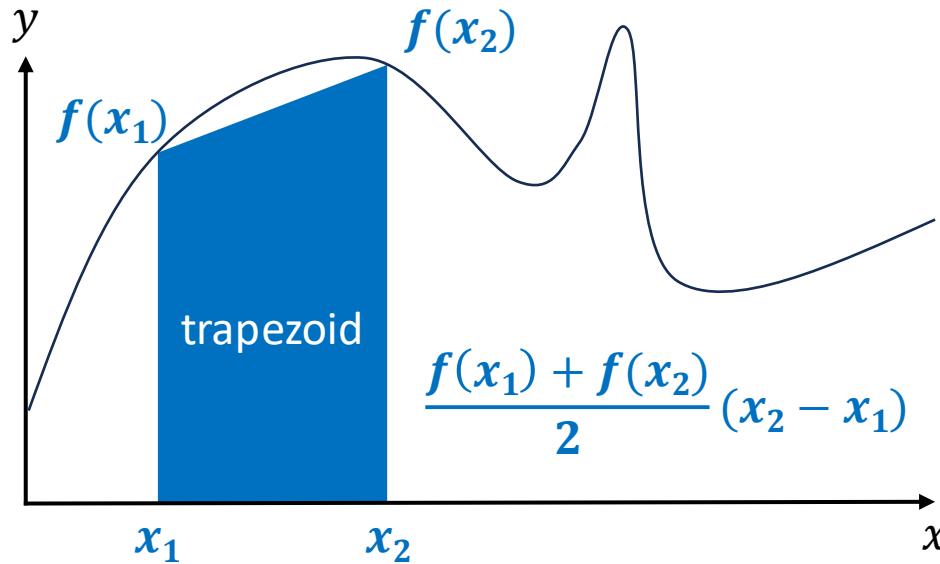
# Case Study #1: Parallel Trapezoidal Integration

- A numerical method to approximate a curve area using trapezoids

$$\int_{x_1}^{x_2} f(x)dx \approx \sum \text{trapezoids}$$

- Example:**

When the number of splits approaches infinity, the sum of all trapezoid areas will coverage to the exact area under the curve.



# Trapezoidal Integration w/ Recursive Task Parallelism

- A numerical method to approximate a curve area using trapezoid sum

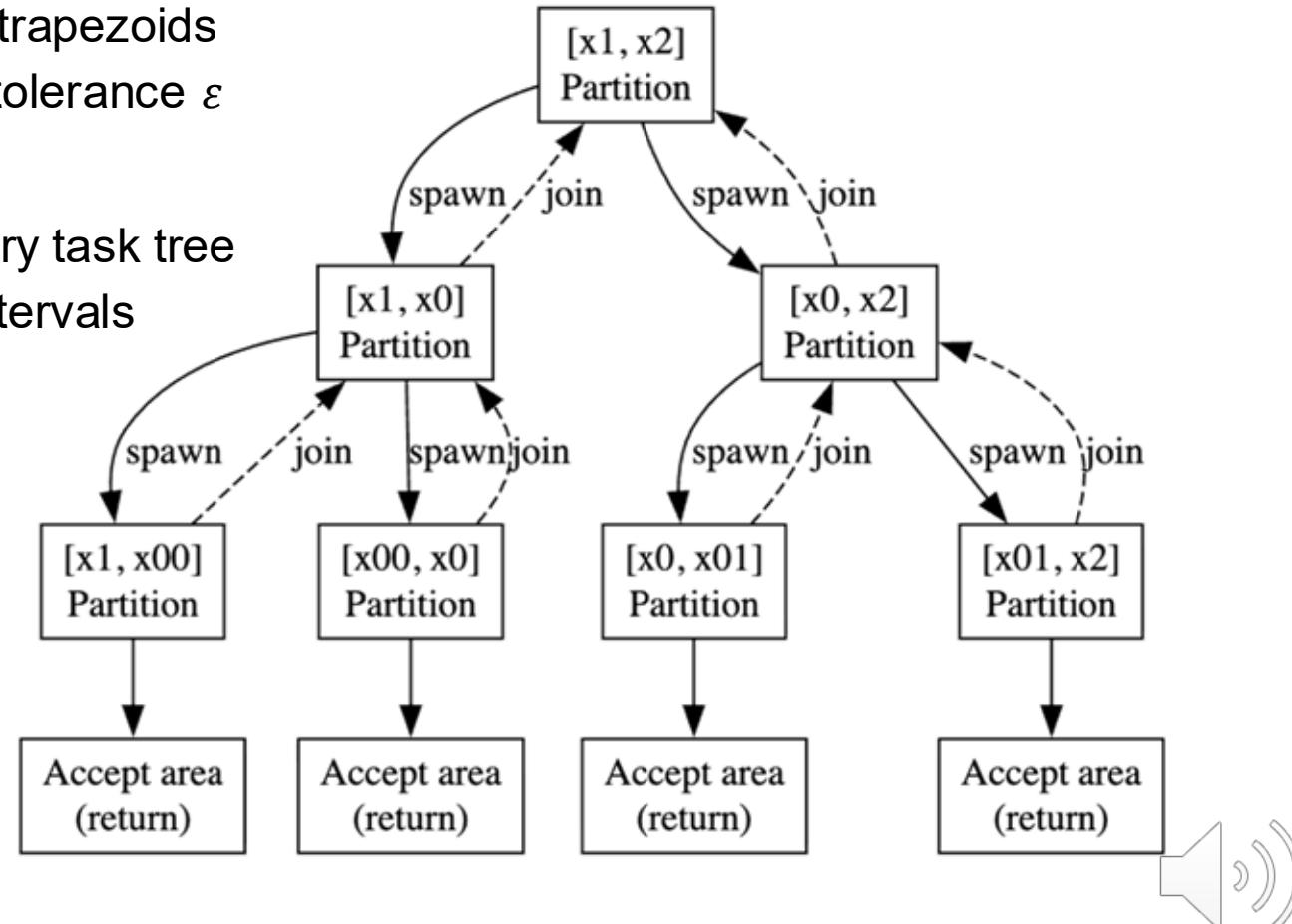
- Start with a coarse trapezoid over the given range  $[x_1, x_2]$
- Partition the interval recursively to refine trapezoids
- Stop when the approximation is within a tolerance  $\varepsilon$

- Observation

- Recursive partition naturally forms a binary task tree
- No shared state among partitioned subintervals
- Each subinterval is independent

- A natural fit for RTP!

- Recursive divide-and-conquer structure
  - Algorithm structure = task structure
- RTP can express this structure directly
  - No flattening to iterative tasks

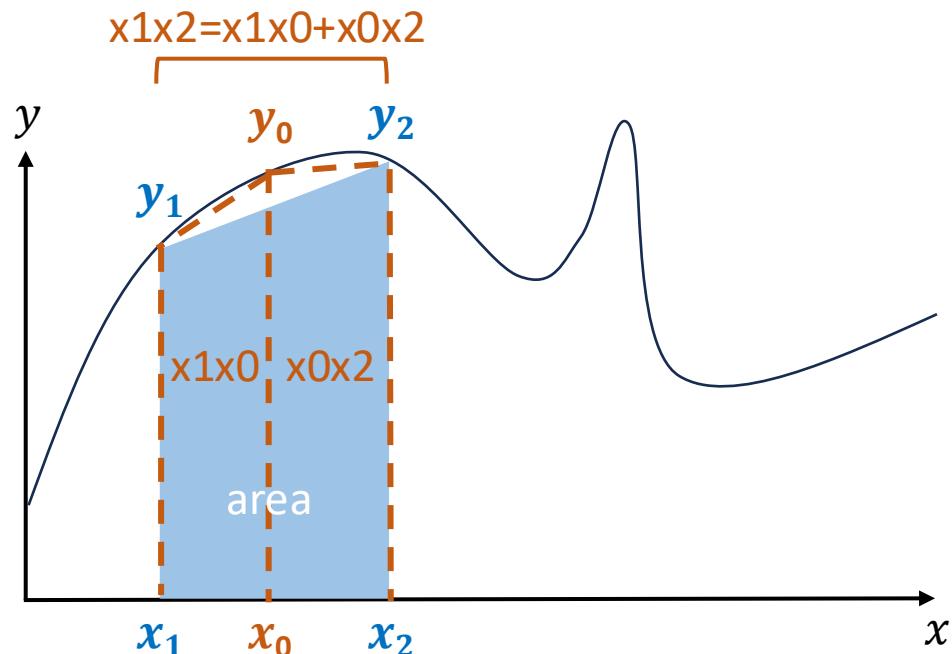


# Sequential Implementation: $f(x) = x^3 + x$ over $[x_1, x_2]$

```
float fn(float x) { return (x * x + 1.0) * x; } //  $f(x) = x^3 + x$ 
```

```
float integrate(float x1, float y1, float x2, float y2, float area) {
    float half = (x2 - x1) / 2;
    float x0 = x1 + half;
    float y0 = fn(x0);
    float x1x0 = (y1 + y0) / 2 * half;
    float x0x2 = (y0 + y2) / 2 * half;
    float x1x2 = x1x0 + x0x2;
    if (std::fabs(x1x2-area) < 1e-9) {
        return x1x2;
    }
    x1x0 = integrate(x1, y1, x0, y0, x1x0);
    x0x2 = integrate(x0, y0, x2, y2, x0x2);
    return x1x0 + x0x2;
}
```

```
float area = integrate(x1, y1, x2, y2, A);
```



// A is  $(y_1+y_2)*(x_2-x_1)/2$



# Parallel Implementation using Runtime Tasking

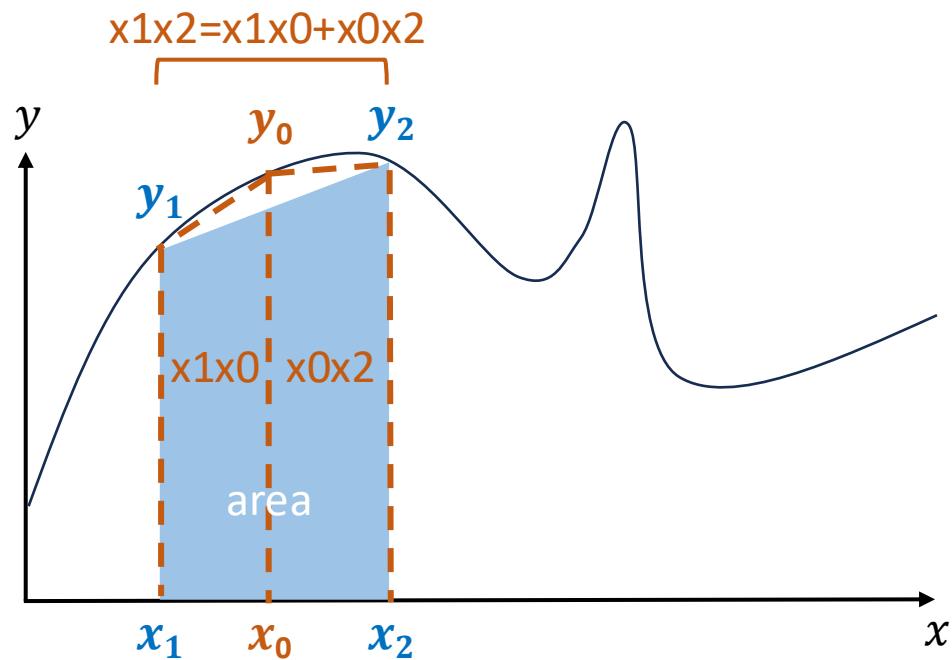
```

float integrate(float x1, float y1, float x2, float y2, float area, tf::Runtime& rt) {
    float half = (x2 - x1) / 2.0;
    float x0 = x1 + half;
    float y0 = fn(x0);
    float x1x0 = (y1 + y0) / 2 * half;
    float x0x2 = (y0 + y2) / 2 * half;
    float x1x2 = x1x0 + x0x2;
    if (std::fabs(x1x2-area) < 1e-9) {
        return x1x2;
    }
    rt.silent_async([=, &x1x0](tf::Runtime& rt1){
        x1x0 = integrate(x1, y1, x0, y0, x1x0, rt1);
    });
    x0x2 = integrate(x0, y0, x2, y2, x0x2, rt);
    rt.corun();
    return x1x0 + x0x2;
}

float area;
executor.async([=, &area](tf::Runtime& rt){ area=integrate(x1, y1, x2, y2, A, rt); }).get();

```

Tail optimization to reduce corun overhead



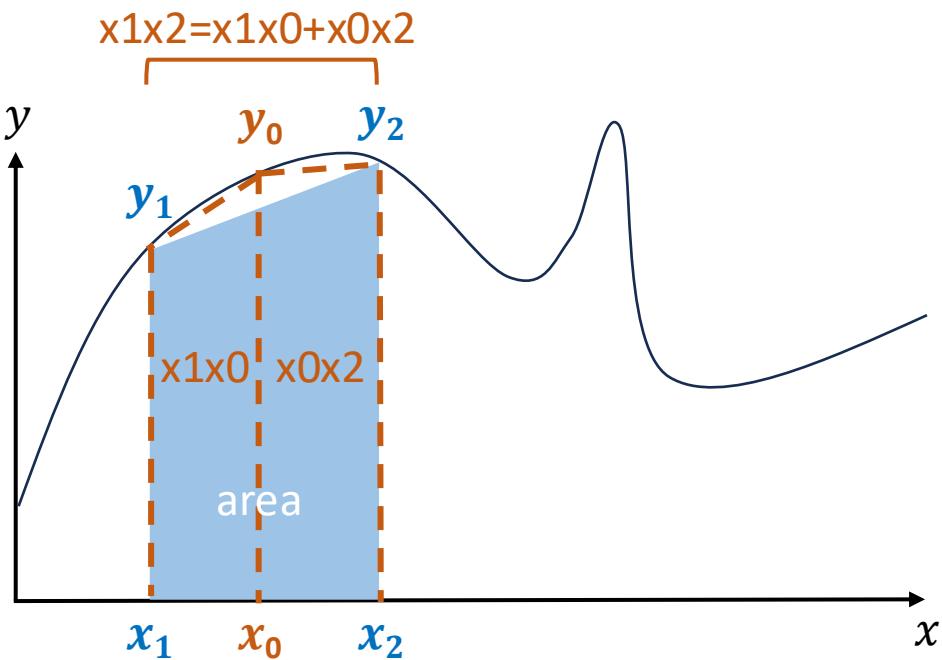
# Parallel Implementation using Task Group

```

float integrate(float x1, float y1, float x2, float y2, float area, tf::Executor& exe) {
    float half = (x2 - x1) / 2.0;
    float x0 = x1 + half;
    float y0 = fn(x0);
    float x1x0 = (y1 + y0) / 2 * half;
    float x0x2 = (y0 + y2) / 2 * half;
    float x1x2 = x1x0 + x0x2;
    if (std::fabs(x1x2-area) < 1e-9) {
        return x1x2;
    }
    tf::TaskGroup tg = exe.task_group();
    tg.silent_async([=, &x1x0, &exe](){
        x1x0 = integrate(x1, y1, x0, y0, x1x0, exe);
    });
    x0x2 = integrate(x0, y0, x2, y2, x0x2, exe);
    tg.corun();
    return x1x0 + x0x2;
}

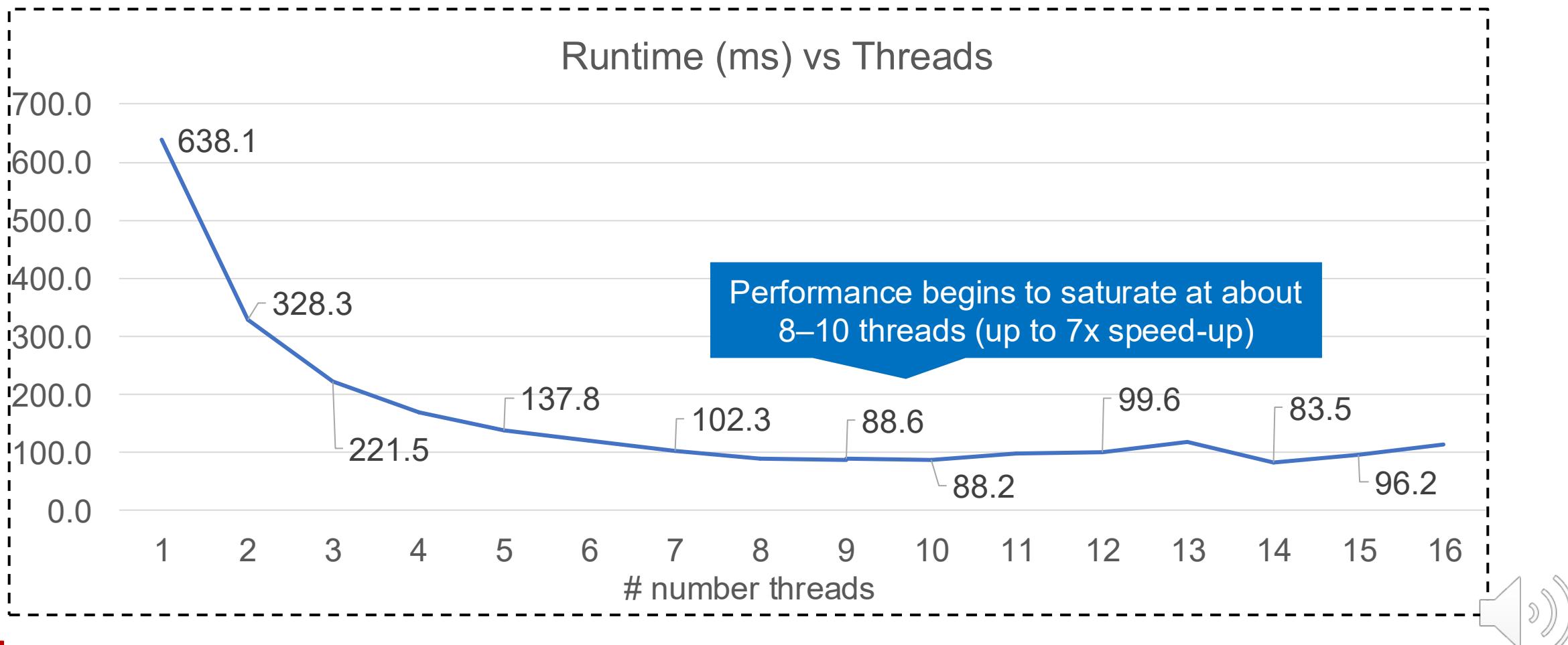
float area = executor.async([=, &exe](){ return integrate(x1, y1, x2, y2, A, exe); }).get();

```



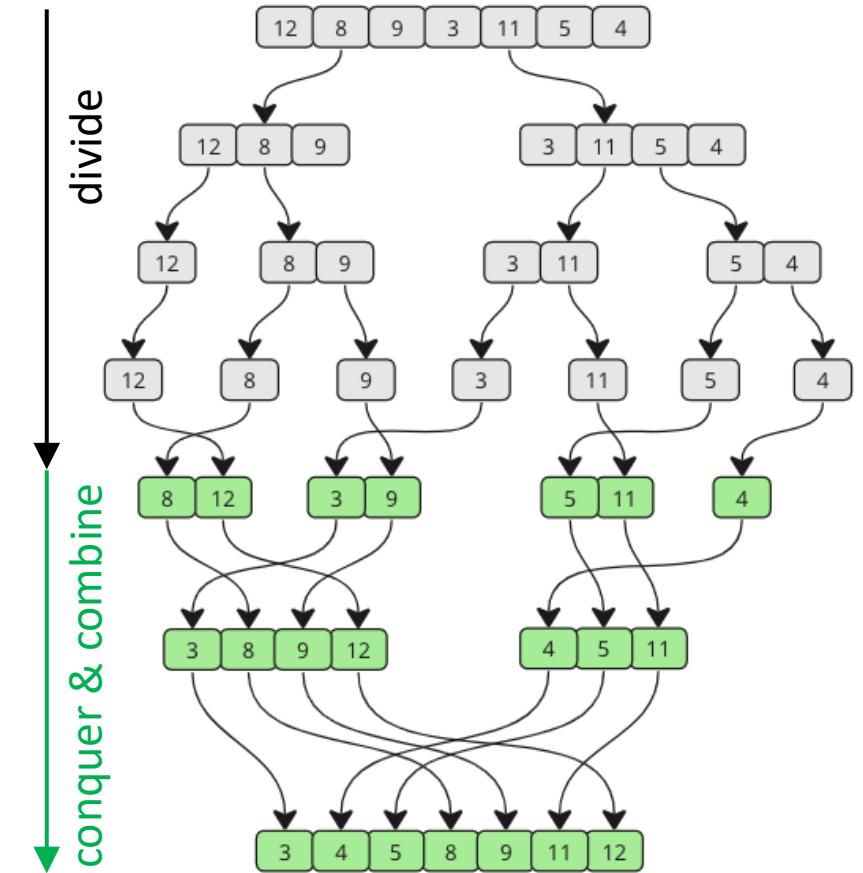
# Performance of Task-parallel Trapezoidal Integration

- Measured with integration over  $[0, 1000]$  on a 16-core Linux machine



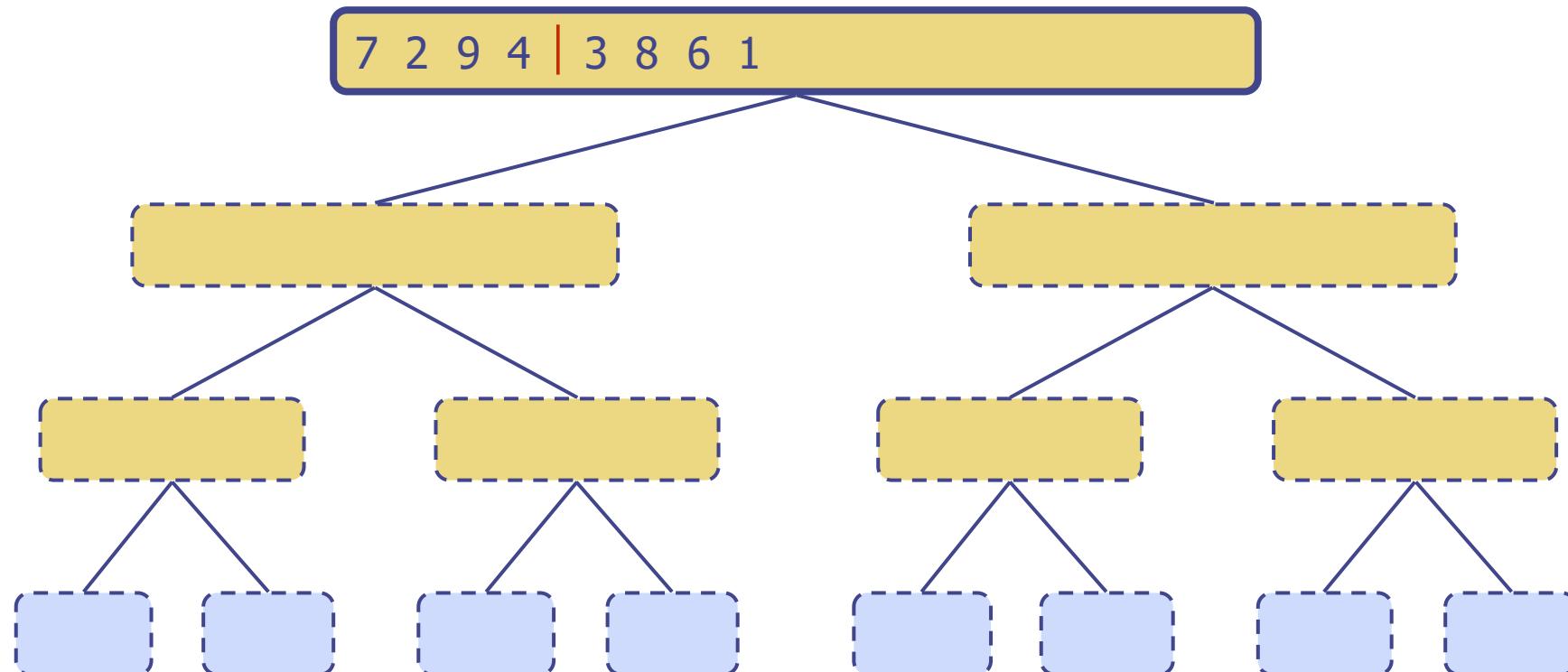
# Case Study #2: Parallel Merge Sort

- A classic divide-and-conquer algorithm that sorts an array by
  - Divide: recursively partitioning the array o two halves
  - Conquer: sorting each half independently
  - Combine: merging two sorted halves into a sorted one
- Observation
  - Merge sort has a naturally recursive structure
  - No shared state among partitioned arrays
  - Partitioned arrays can run concurrently at every level
- A natural fit for RTP!
  - Recursive divide-and-conquer structure
    - Algorithm structure = task structure
  - RTP can express this structure directly
    - No flattening to iterative tasks



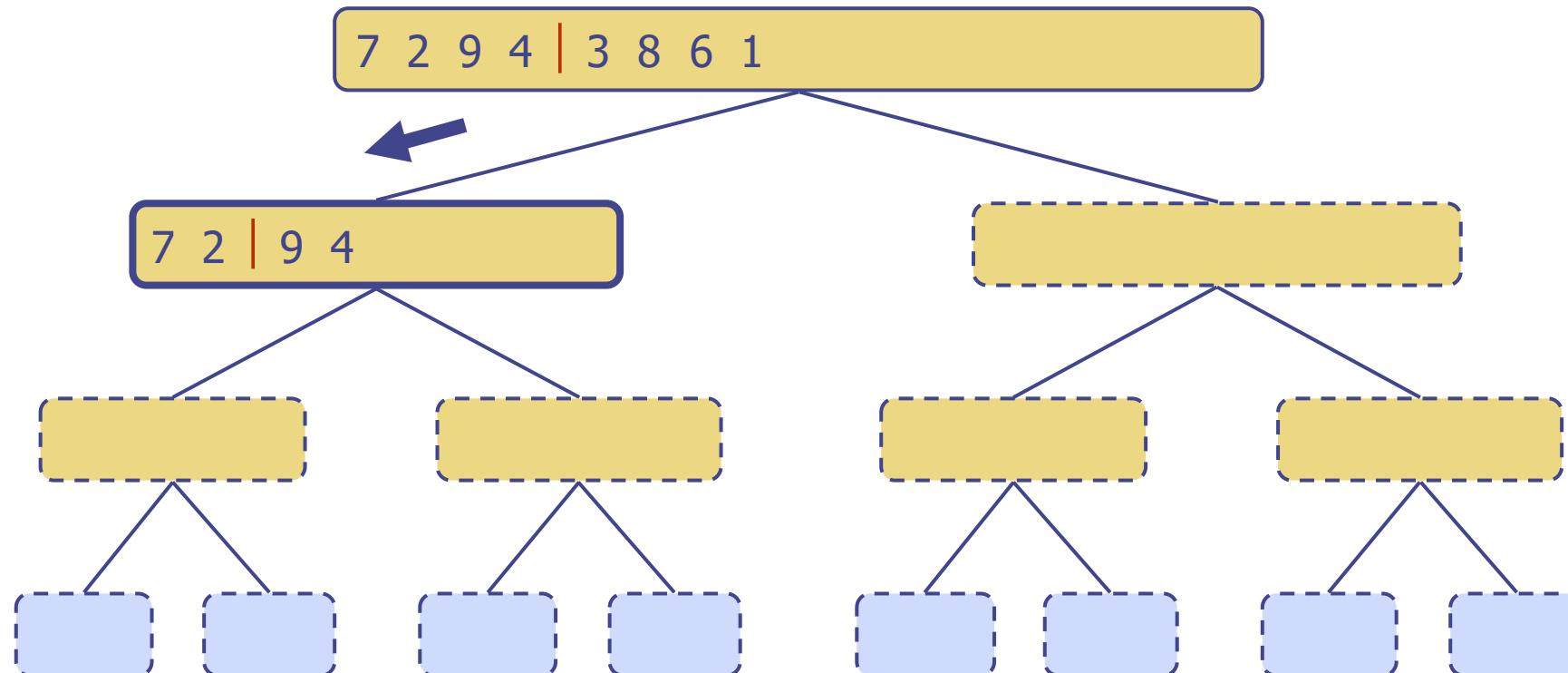
# Merge Sort Example – 1/10

- Sort an input array of eight numbers  $\{7, 2, 9, 4, 3, 8, 6, 1\}$ 
  - At the top-most level, we have eight numbers, not reaching the base case yet
  - Partition the array into two halves, subarray  $\{7, 2, 9, 4\}$  and subarray  $\{3, 8, 6, 1\}$



# Merge Sort Example – 2/10

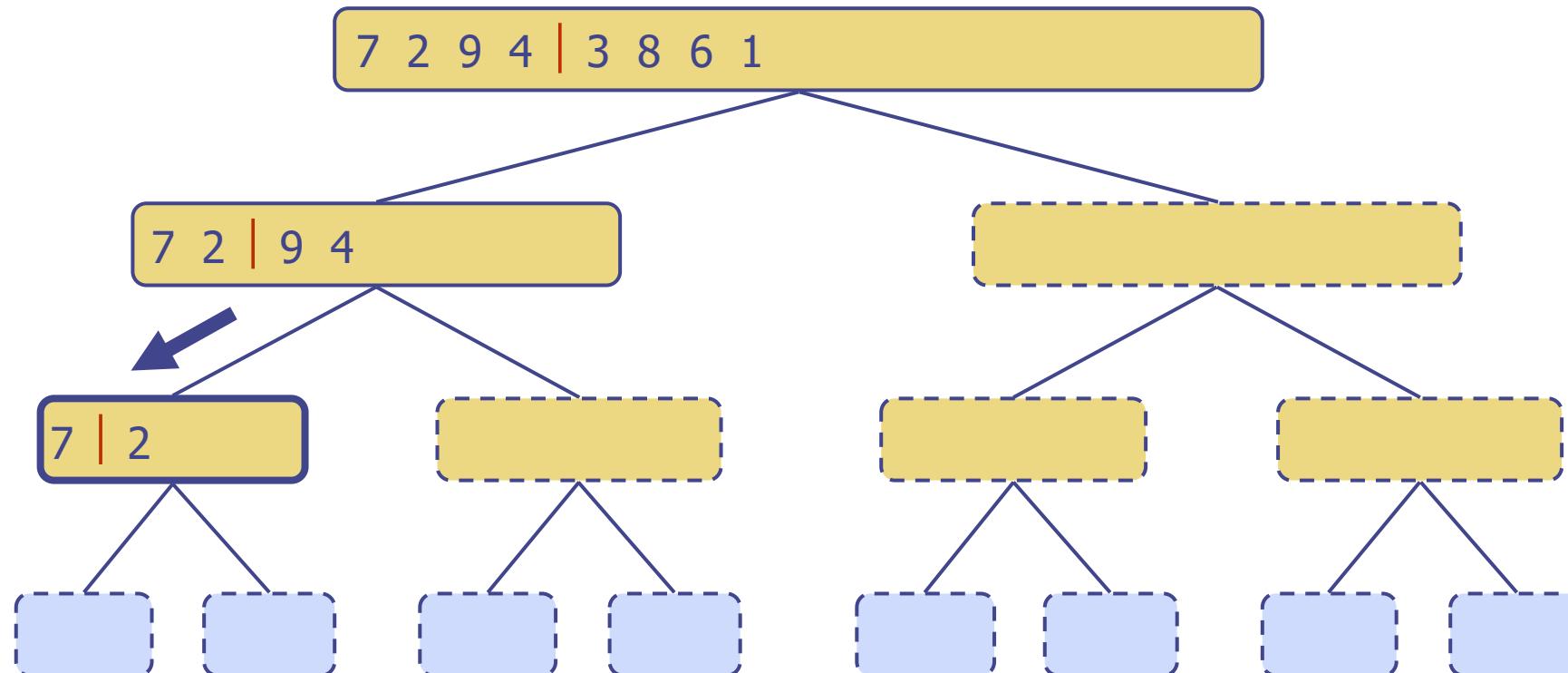
- **Process the left half {7, 2, 9, 4}**
  - The array has four numbers, not reaching the base case yet
  - Partition the subarray {7, 2, 9, 4} to two halves, subarray {7, 2} and subarray {9, 4}



# Merge Sort Example – 3/10

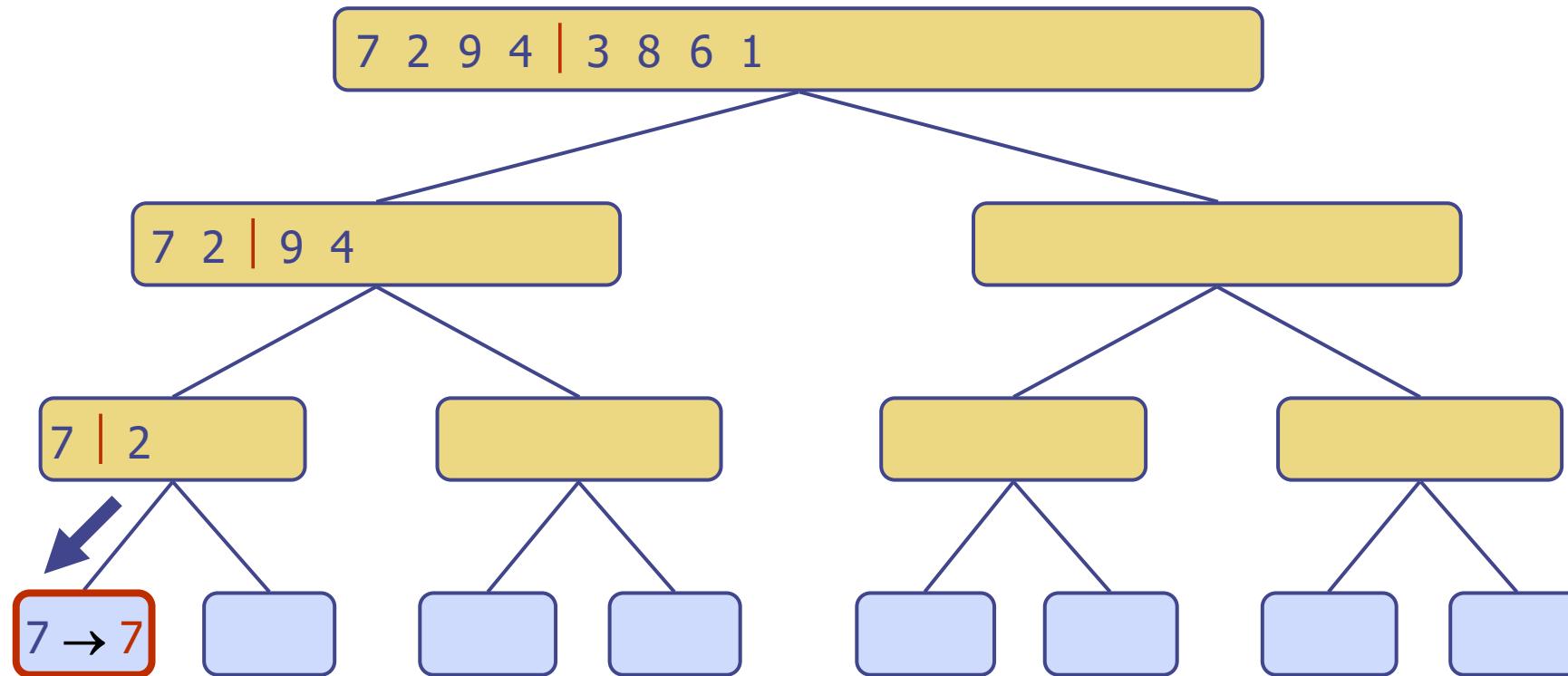
- **Process the left half {7, 2}**

- The array has two numbers, not reaching the base case yet
- Partition the subarray {7, 2} to two halves, subarray {7} and subarray {2}



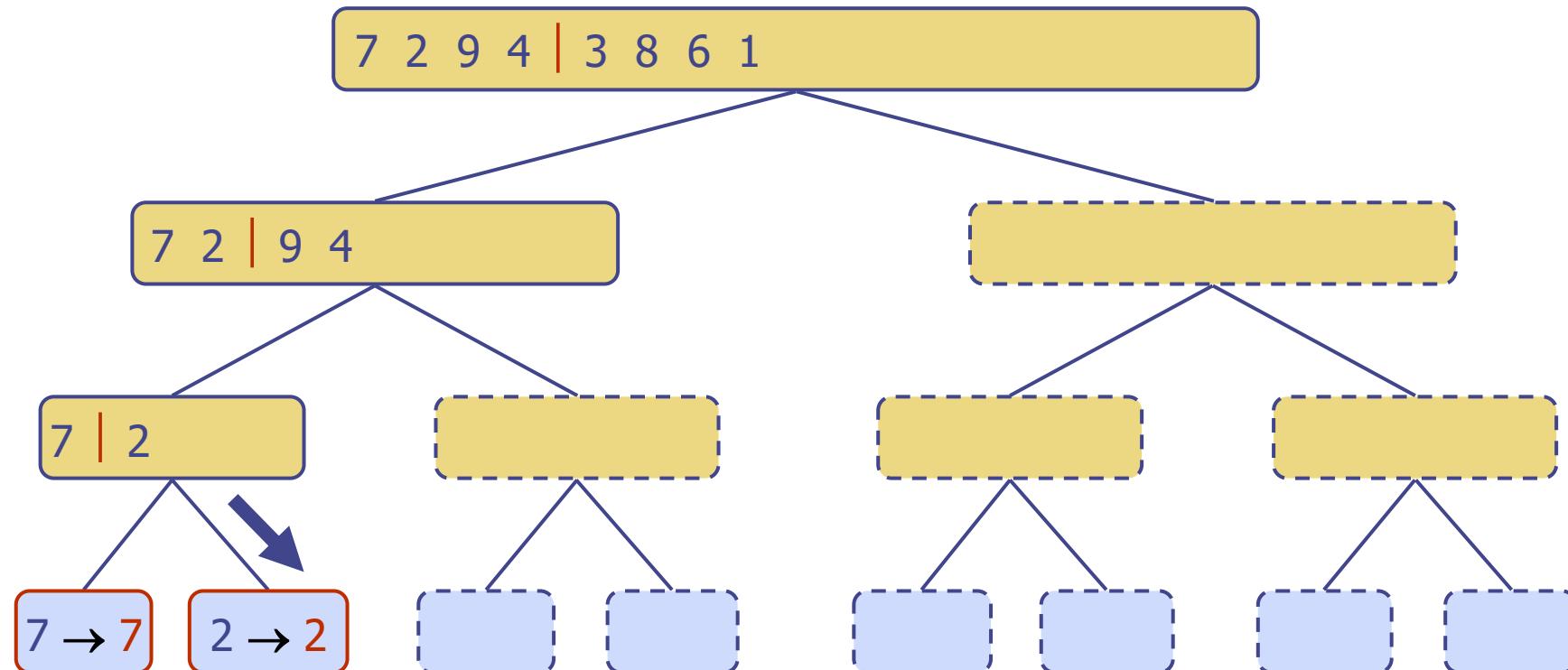
# Merge Sort Example – 4/10

- The left-most partition has only one number {7}
  - Obviously, a subarray of a single number is sorted, reaching the base case, so we return



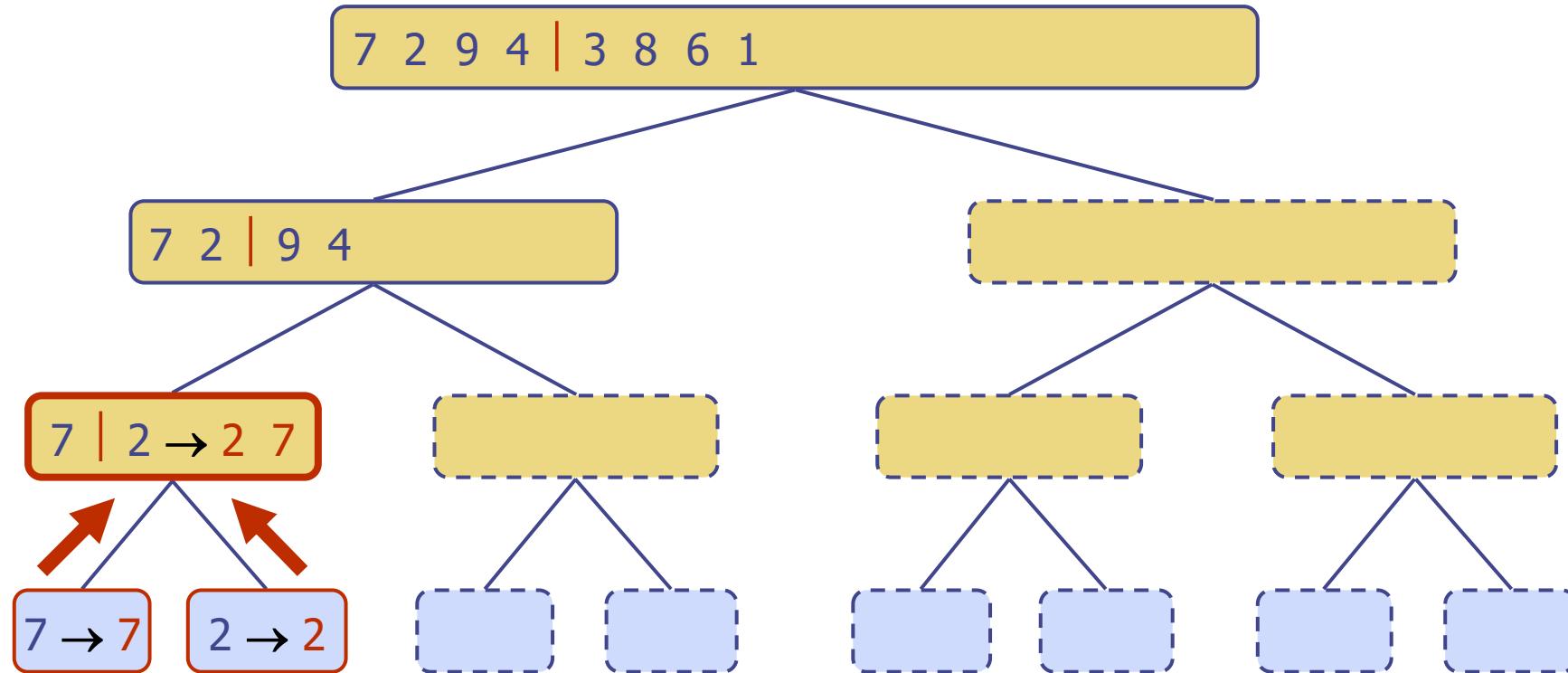
# Merge Sort Example – 5/10

- The right partition has only one number {2}
- Obviously, a subarray of a single number is sorted, reaching the base case, so we return



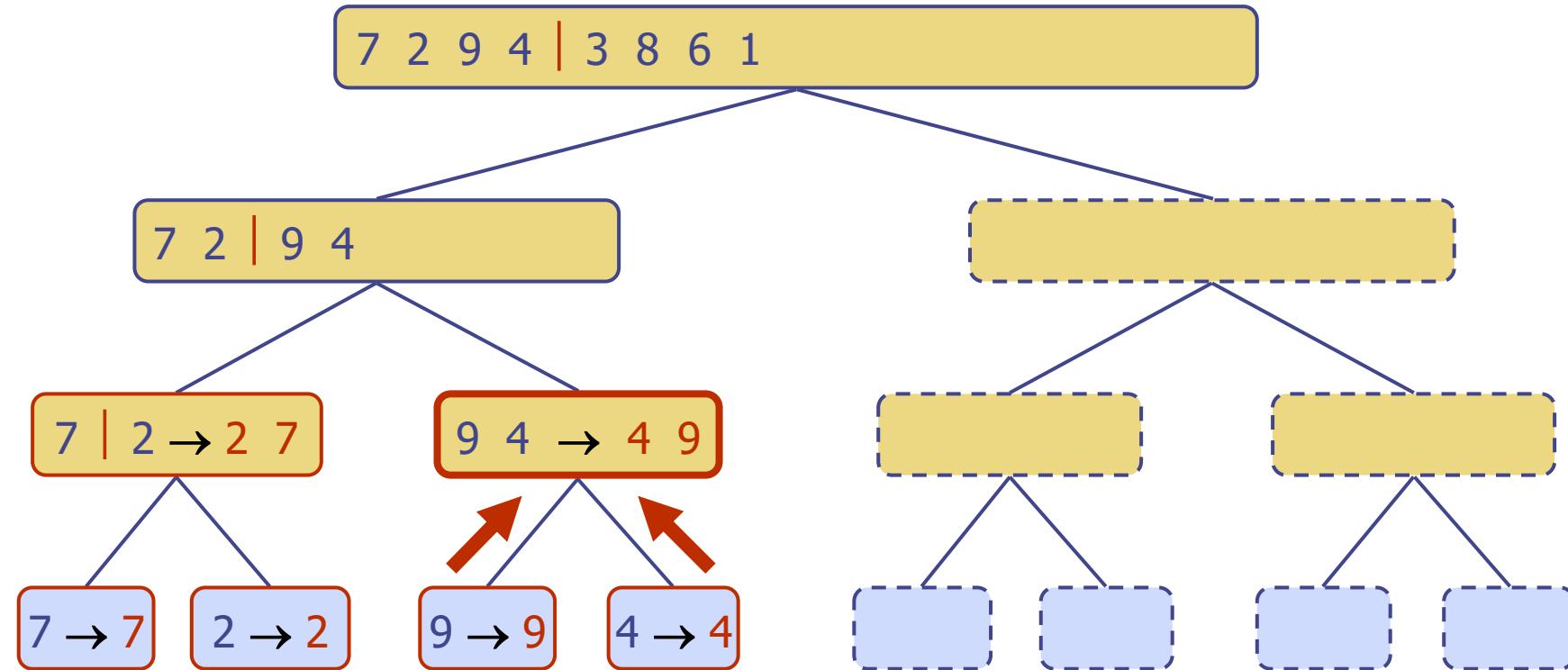
# Merge Sort Example – 6/10

- **Merge two sorted subarrays {7} and {2} into a sorted subarray {2, 7}**
  - This is the conquer stage, combining partial results into a larger sorted subarray
  - In C++, this operation can be implemented using `std::merge`<sup>1</sup> with linear time complexity



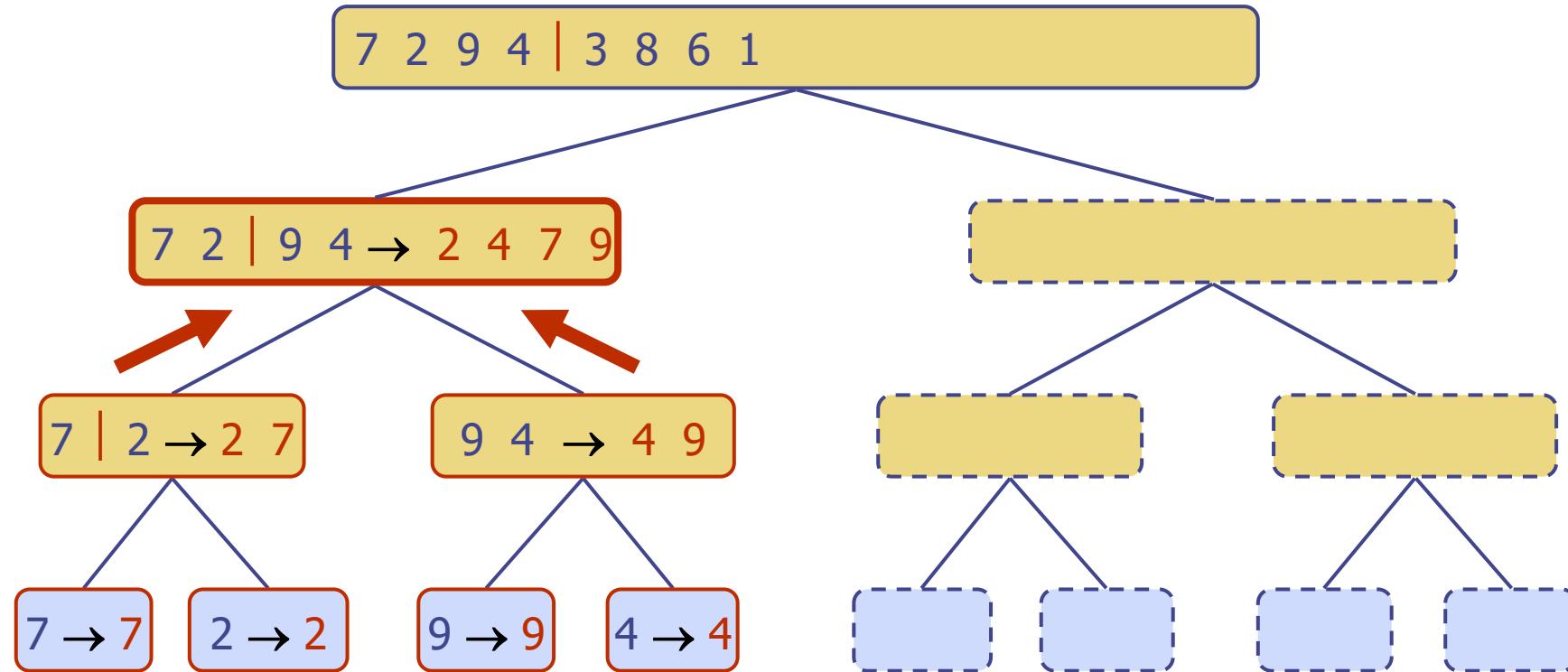
# Merge Sort Example – 7/10

- Similarly, merge two sorted subarrays {9} and {4} to a sorted subarray {4, 9}
  - This is the conquer stage, combining partial results into a larger sorted subarray
  - In C++, this operation can be implemented using `std::merge`<sup>1</sup> with linear time complexity



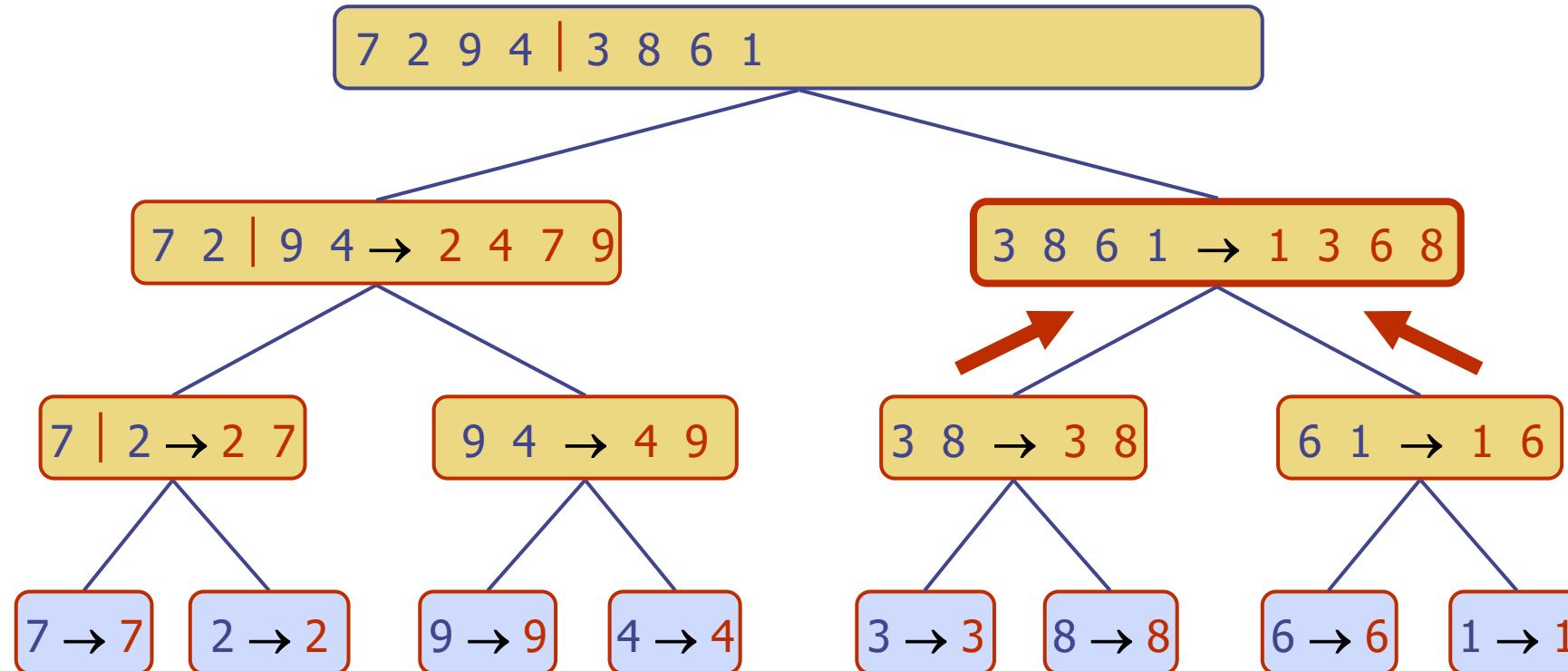
# Merge Sort Example – 8/10

- Merge two sorted subarrays  $\{2, 7\}$  and  $\{4, 9\}$  to a sorted subarray  $\{2, 4, 7, 9\}$ 
  - This is the conquer stage, combining partial results into a larger sorted subarray
  - In C++, this operation can be implemented using `std::merge`<sup>1</sup> with linear time complexity



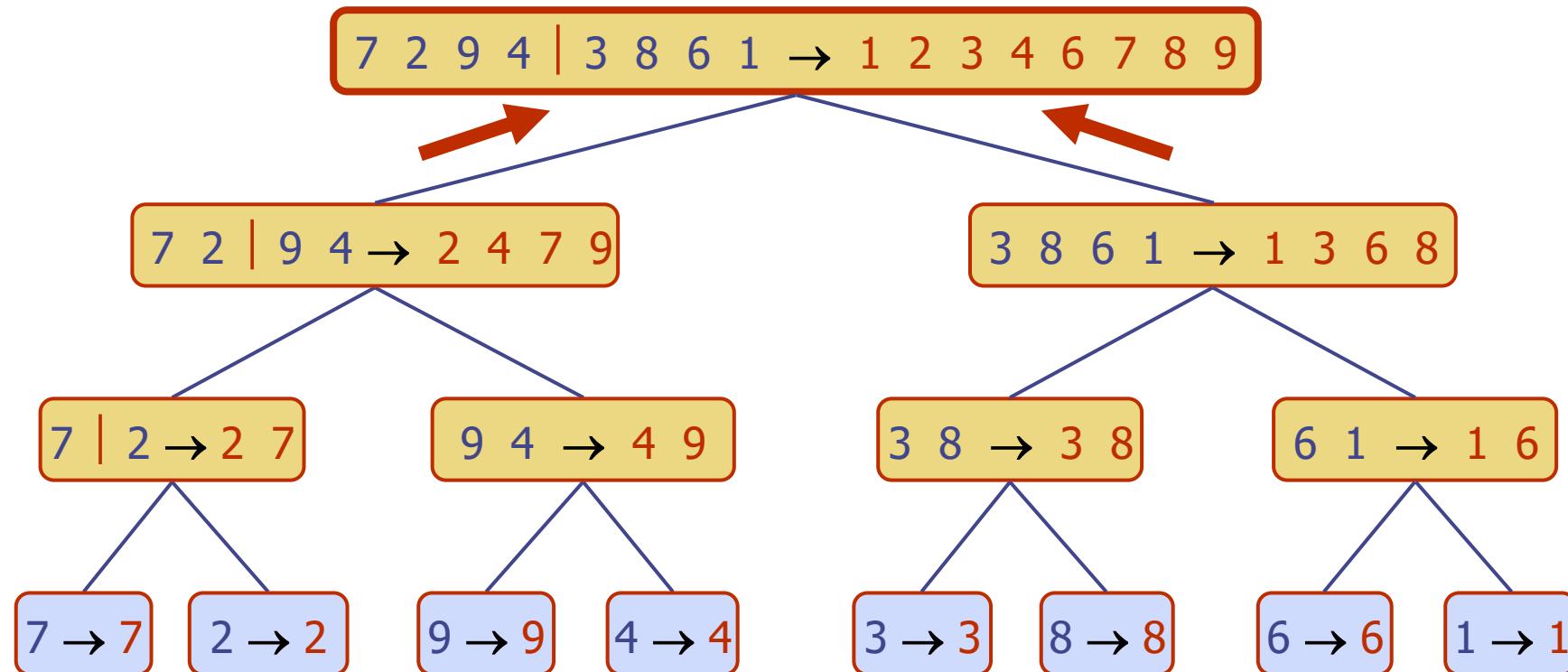
# Merge Sort Example – 9/10

- Apply the same recursive procedure to the right partitions ...
  - Merge two sorted subarrays  $\{3, 8\}$  and  $\{1, 6\}$  to a sorted subarray  $\{1, 3, 6, 8\}$
  - In C++, this operation can be implemented using `std::merge`<sup>1</sup> with linear time complexity



# Merge Sort Example – 10/10

- Merge  $\{2, 4, 7, 9\}$  and  $\{1, 3, 6, 8\}$  to a fully sorted array  $\{1, 2, 3, 4, 6, 7, 8, 9\}$ 
  - This is the conquer stage, combining partial results into the final solution (all numbers are sorted)
  - In C++, this operation can be implemented using `std::merge`<sup>1</sup> with linear time complexity





# Sequential Implementation using C++ Algorithm

```
template <typename T>
void msort(std::vector<T>& data, std::vector<T>& buffer, size_t begin, size_t end) {
    size_t n = end - begin;
    if (n <= 1024) {
        std::sort(data.begin() + begin, data.begin() + end); // Sequential cutoff on 1024 elements
        return;
    }
    size_t mid = begin + n / 2;
    msort(data, buffer, mid, end);
    msort(data, buffer, begin, mid);
    std::merge(data.begin() + begin, data.begin() + mid, data.begin() + mid, data.begin() + end,
               buffer.begin() + begin); // Merging the two sorted ranges, [begin, mid) and [mid, end), into a sorted one using std::merge
    std::copy(buffer.begin() + begin, buffer.begin() + end, data.begin() + begin);
}

template <typename T>
void msort(std::vector<T>& data) {
    std::vector<T> buffer(data.size()); // temporary storage for merging two subarray
    msort(data, buffer, 0, data.size());
}
```

Merging the two sorted ranges, [begin, mid) and [mid, end), into a sorted one using std::merge

Merging two sorted subarrays efficiently usually needs a temporary buffer to store intermediate results.





# Parallel Implementation using Task Group

```
template <typename T>
void msort(std::vector<T>& data, std::vector<T>& buffer, size_t begin, size_t end) {
    size_t n = end - begin;
    if (n <= 1024) { std::sort(data.begin() + begin, data.begin() + end); return; }
    size_t mid = begin + n / 2;
    tf::TaskGroup tg = executor.task_group();
    tg.silent_async([=, &data, &buffer](){ msort(data, buffer, mid, end); });
    msort(data, buffer, begin, mid);
    tg.corun();
    std::merge(data.begin() + begin, data.begin() + mid, data.begin() + mid, data.begin() + end,
               buffer.begin() + begin);
    std::copy(buffer.begin() + begin, buffer.begin() + end, data.begin() + begin);
}
template <typename T>
void msort(std::vector<T>& data) {
    std::vector<T> buffer(data.size());
    executor.async([&]() { msort(data, buffer, 0, data.size()); }).wait();
}
```





# Parallel Implementation using OpenMP

```
template <typename T>
void msort(std::vector<T>& data, std::vector<T>& buffer, size_t begin, size_t end) {
    size_t n = end - begin;
    if (n <= 1024) { std::sort(data.begin() + begin, data.begin() + end); return; }
    size_t mid = begin + n / 2;
    #pragma omp task
    {
        msort(data, buffer, mid, end);
    }
    msort(data, buffer, begin, mid);

    #pragma omp taskwait Synchronization

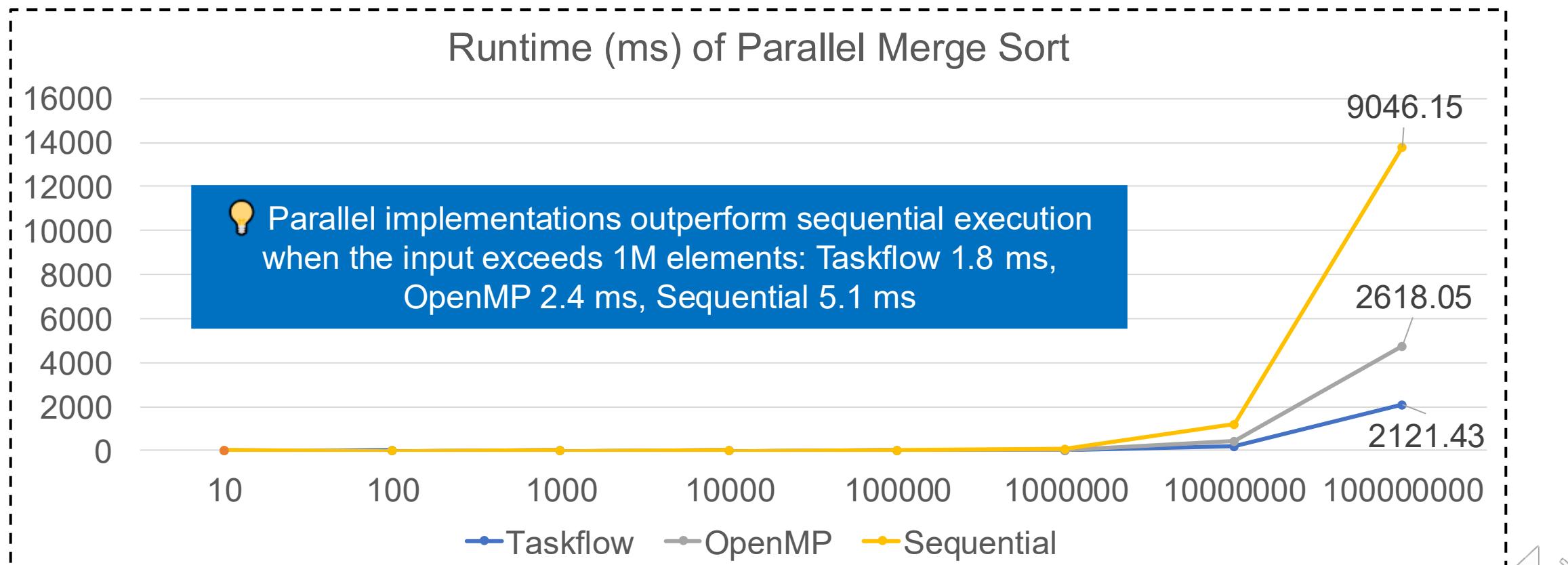
    std::merge(
        data.begin() + begin, data.begin() + mid,
        data.begin() + mid, data.begin() + end,
        buffer.begin() + begin
    );
    std::copy(buffer.begin() + begin, buffer.begin() + end, data.begin() + begin);
}
```

```
template <typename T>
void msort(std::vector<T>& data) {
    std::vector<T> buffer(data.size());
    #pragma omp parallel
    {
        #pragma omp single
        { msort(data, buffer, 0, data.size()); }
    }
}
```



# Performance of Parallel Merge Sort

- Measured on sorting a `std::vector<double>` of up to 100 million elements
  - Taskflow (8 threads) vs OpenMP (8 threads) vs Sequential (1 thread)





# Takeaways

---

- Learn how to program recursive task parallelism in Taskflow
- Understand the role of cooperative execution in recursive task parallelism
- Showcase two real-world applications of recursive task parallelism
- **Conclude the talk**





# 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();
```

