Threading Building Block (TBB)

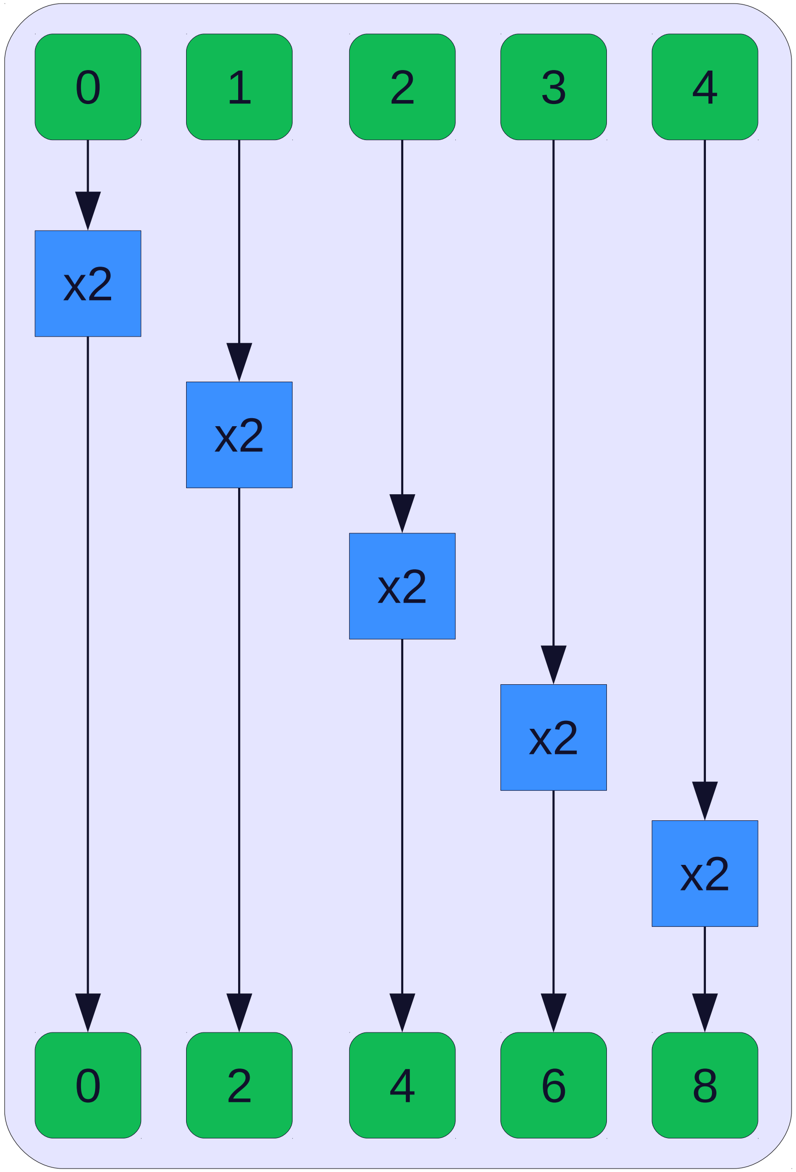
- Open source C++ library for parallel algorithms
- Review C++ lambdas: super handy with TBB!
- apt-get install libtbb-dev
- #include “tbb/tbb.h”
- LIBS += -ltbb
- CONFIG += c++14 (c++11 required for lambdas)

C++ lambda

- Syntax: [capture](parameters) -> return_type { body }
- std::function<return_type (parameters)> lambda

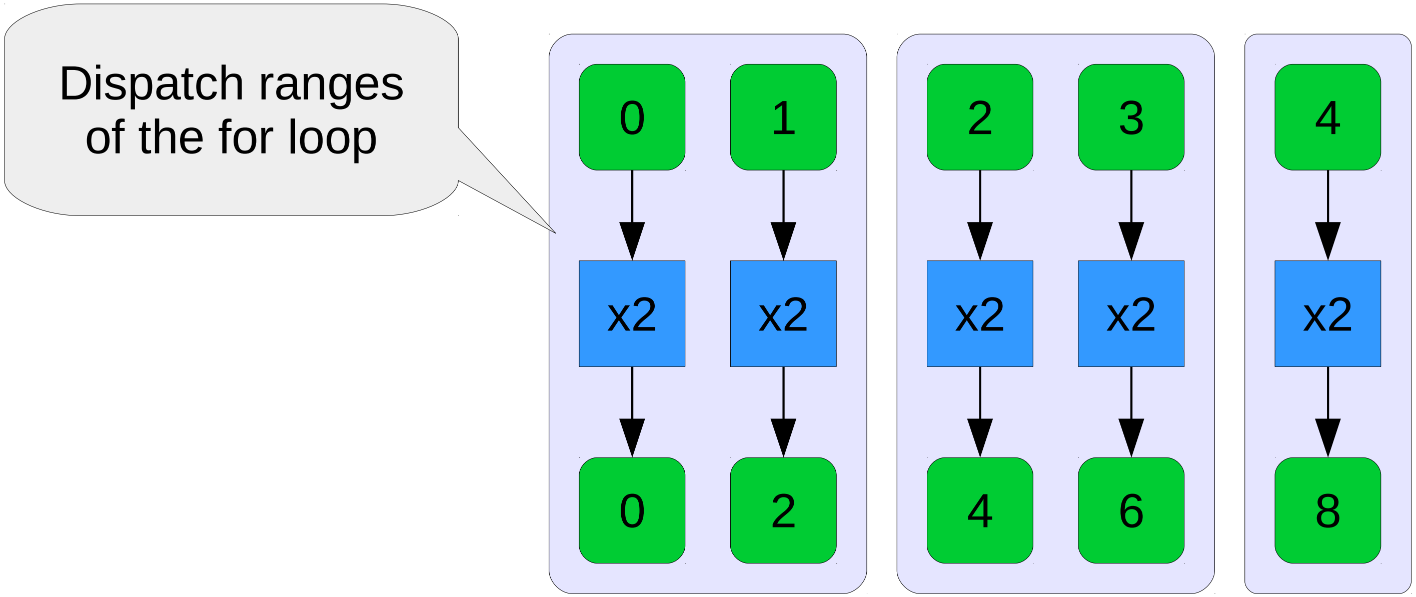
```
// basic example: no capture, no arguments, no return value
std::function<void ()> f1 = [] () { qDebug() << "lambda"; };
f1();               // explicit call
execute(f1);        // pass as parameter

// let the compiler figure the type
auto f2 = [] () { qDebug() << "lambda"; };
f2();

// you can access variables from the parent scope...
QString msg("variable in parent scope by reference");
auto f3 = [&] () { qDebug() << "lambda" << msg; };
f3();
```

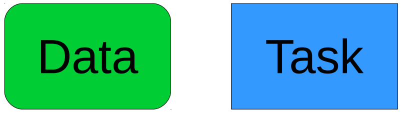
Serial loop



Map

- Dispatch iterations to threads
- Loop iterations must be independent





tbb::parallel_for

```
int n = 100;
QVector<int> array(n);

// serial
for (int i = 0; i < n; i++) {
    array[i] = i;
}

// lambda
tbb::parallel_for(tbb::blocked_range<int>(0, n),
    [&] (tbb::blocked_range<int>& range) {
        for (int i = range.begin(); i < range.end(); i++) {
            array[i] = i;
        }
});

// lambda without inner loop
tbb::parallel_for(0, n, 1, [&] (int& i) { array[i] = i; } );
```

Exercice

- 16-tbb-image: Implements the image processing with TBB
- Verify the resulting image
- What is the speedup you obtain?
- Compare with 01-pthread-img

Reduction

- Outputs a summary if a collection
- Example: sum of a large set

```
int n = 100;
QVector<int> v(n);

// initialization
for (int i = 0; i < n; i++) {
    v[i] = i;
}

// serial
int exp = 0;
for (int i = 0; i < n; i++) {
    exp += v[i];
}
```

Reduction (cont)

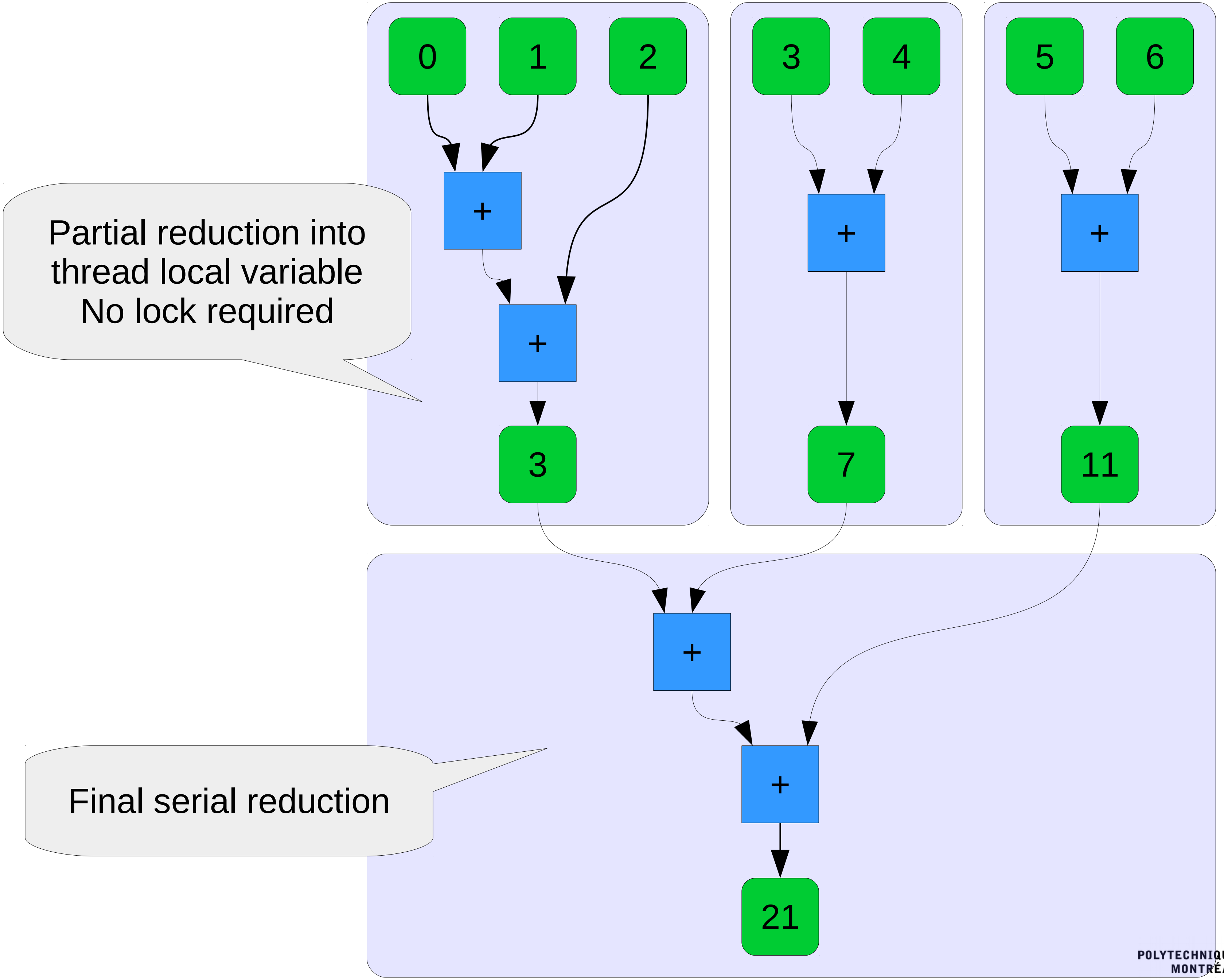
```
// reduction with parallel_for
int sum0 = 0;
tbb::parallel_for(0, v.size(), [&](int &i) {
    sum0 += v[i];
});
```

Concurrent access to a shared variable!

```
// reduction with parallel_for
sum0 = 0;
QMutex mutex;
tbb::parallel_for(0, v.size(), [&](int &i) {
    mutex.lock();
    sum0 += v[i];
    mutex.unlock();
});
```

Access is serialized, defeats the purpose!

Two stages reduction



```
void do_sum(work *arg)
{
    int begin = arg->data->size() * arg->n / arg->rank;
    int end = arg->data->size() * (arg->n + 1) / arg->rank;
    int *v = arg->data->data();
    int sum = 0;
    for (int i = begin; i < end; i++) {
        sum += v[i];
    }
    arg->local_sum = sum;
}
```

Partial reduction

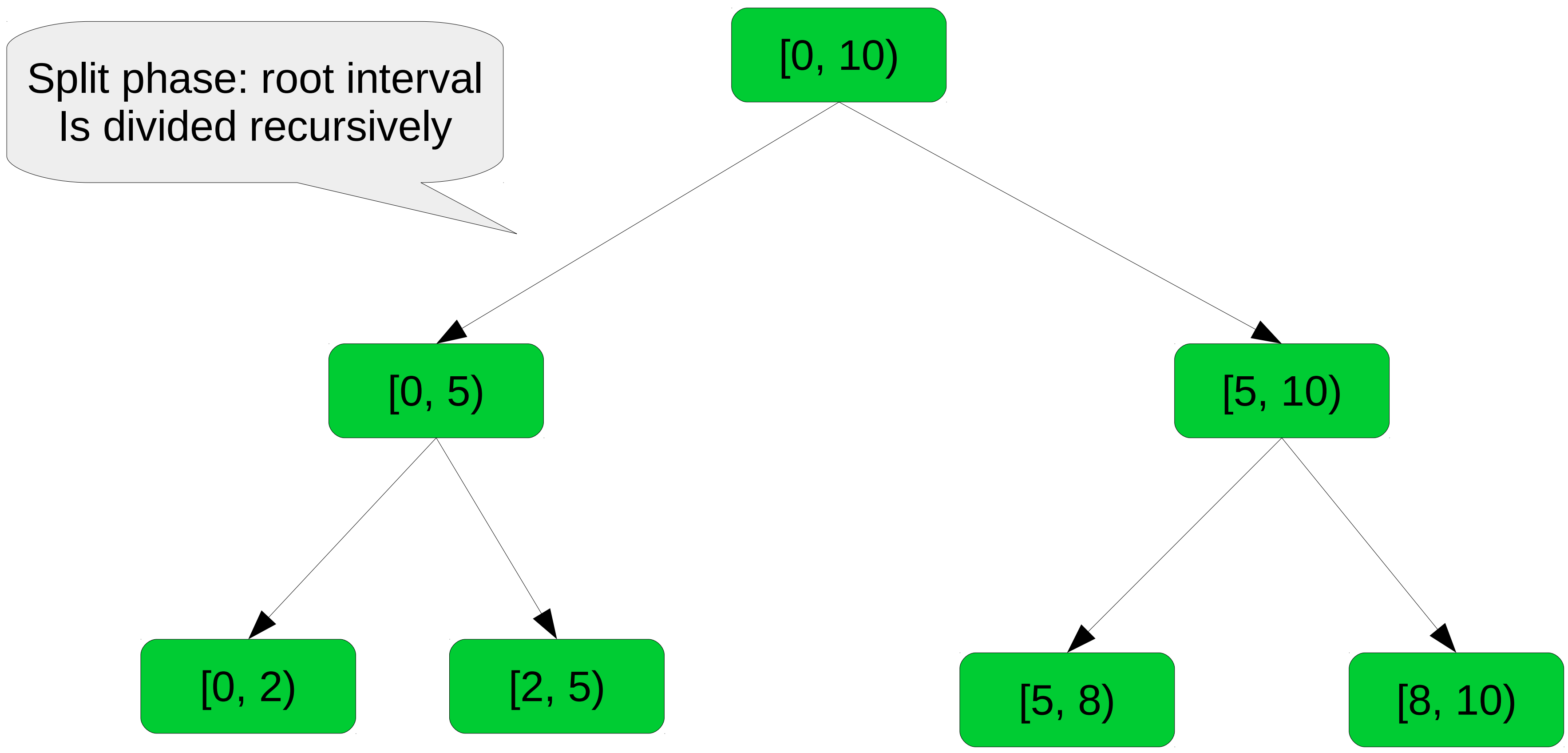
```
int reduce_pthread(QVector<int> &v, int n)
{
    thread threads[n];
    work items[n];
    int final_sum = 0;

    for (int i = 0; i < n; i++) {
        items[i] = work{n, i, &v, 0};
        threads[i] = thread(do_sum, &items[i]);
    }

    for (int i = 0; i < n; i++) {
        threads[i].join();
        final_sum += items[i].local_sum;
    }
    return final_sum;
}
```

Final serial reduction

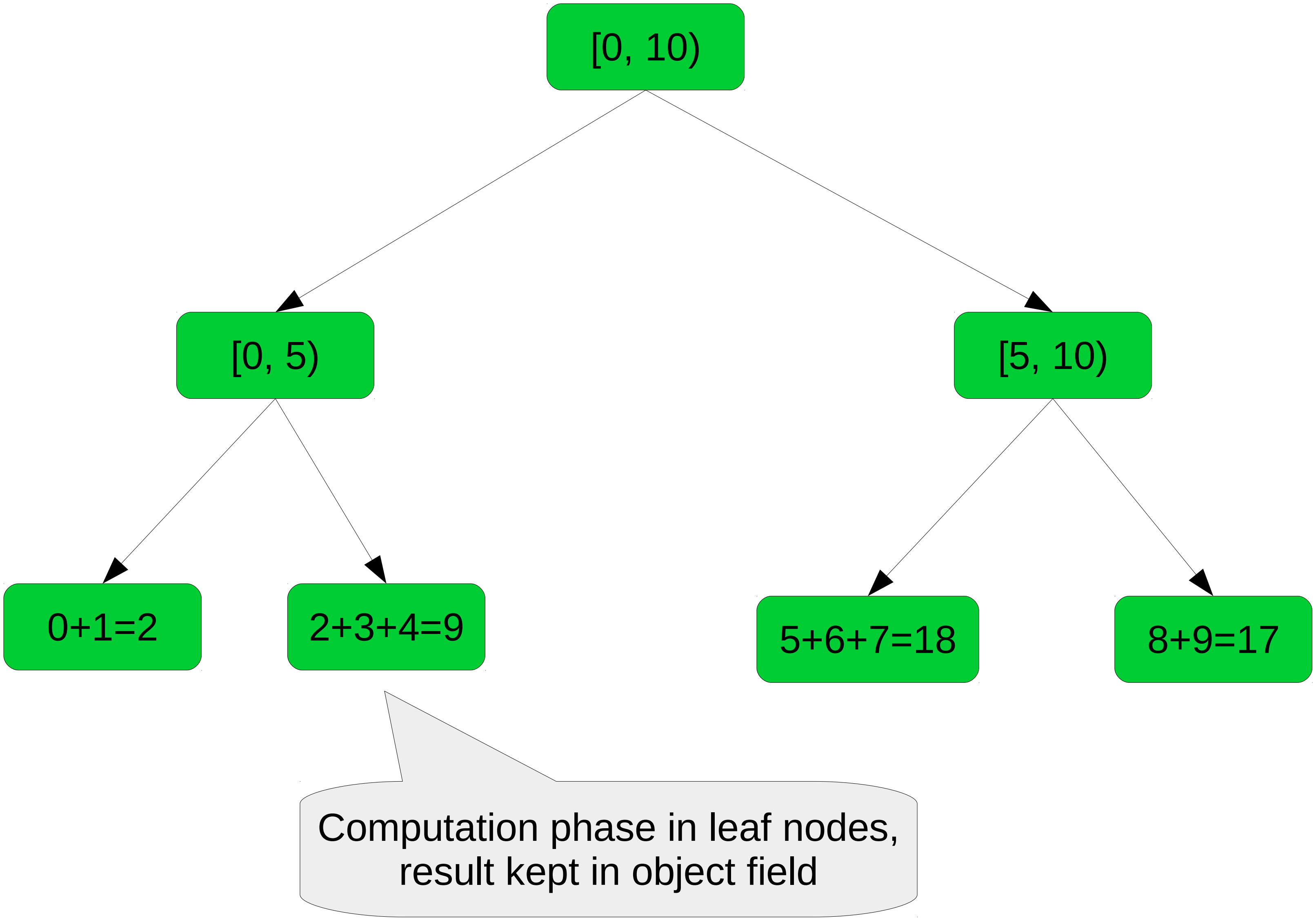
tbb::parallel_reduce

```
// lambda
int sum2 = tbb::parallel_reduce(
    tbb::blocked_range<int>(0, n),  // global range to process
    0,                              // initial value

    // first lambda: compute the reduce for a subrange
    [&] (const tbb::blocked_range<int>& range, int sum) -> int {
        for (int i = range.begin(); i < range.end(); i++) {
            sum += v[i];
        }
        return sum;
    },

    // second lambda: merge range results
    [] (int x, int y) -> int {
        return x + y;
    }
);
```

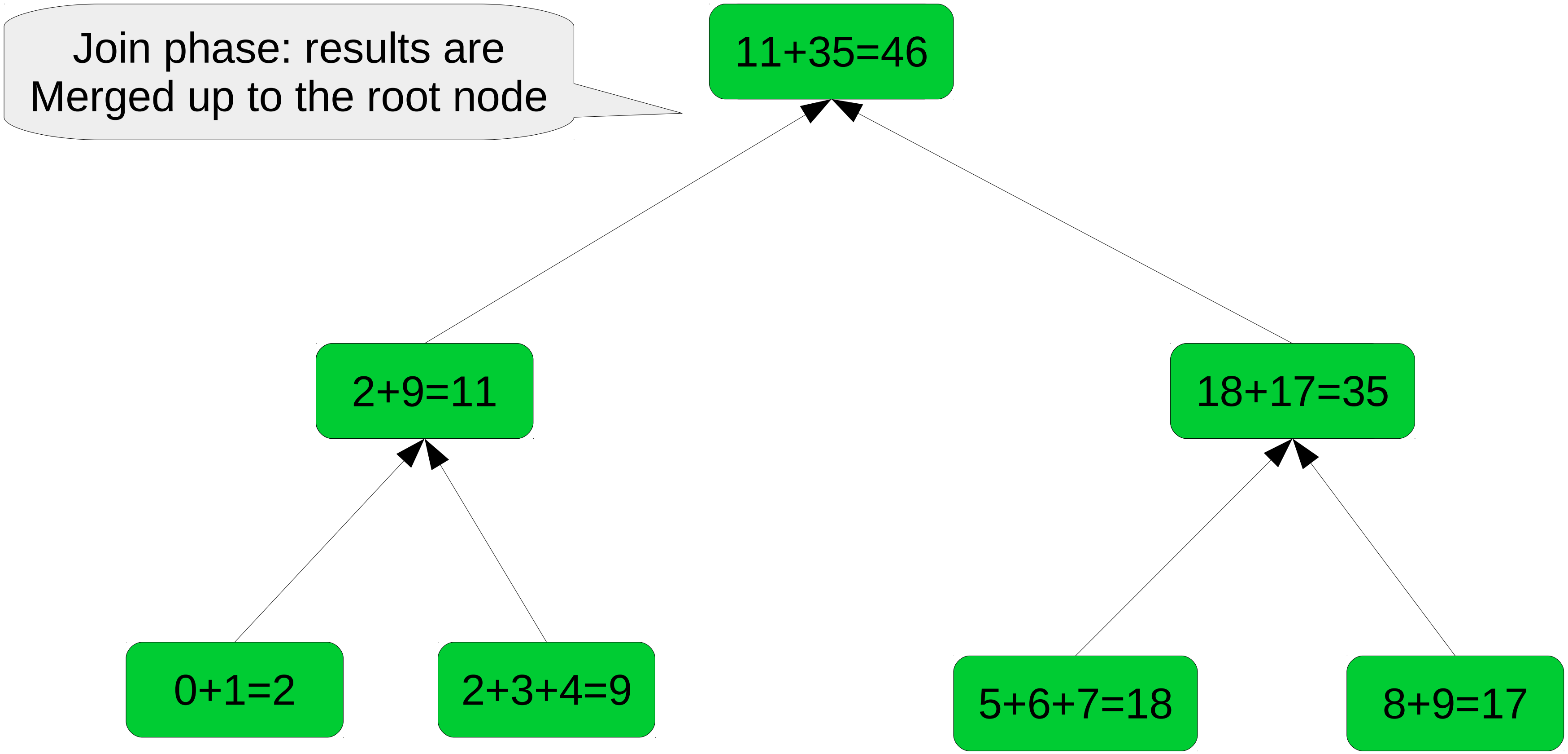
tbb::parallel_reduce



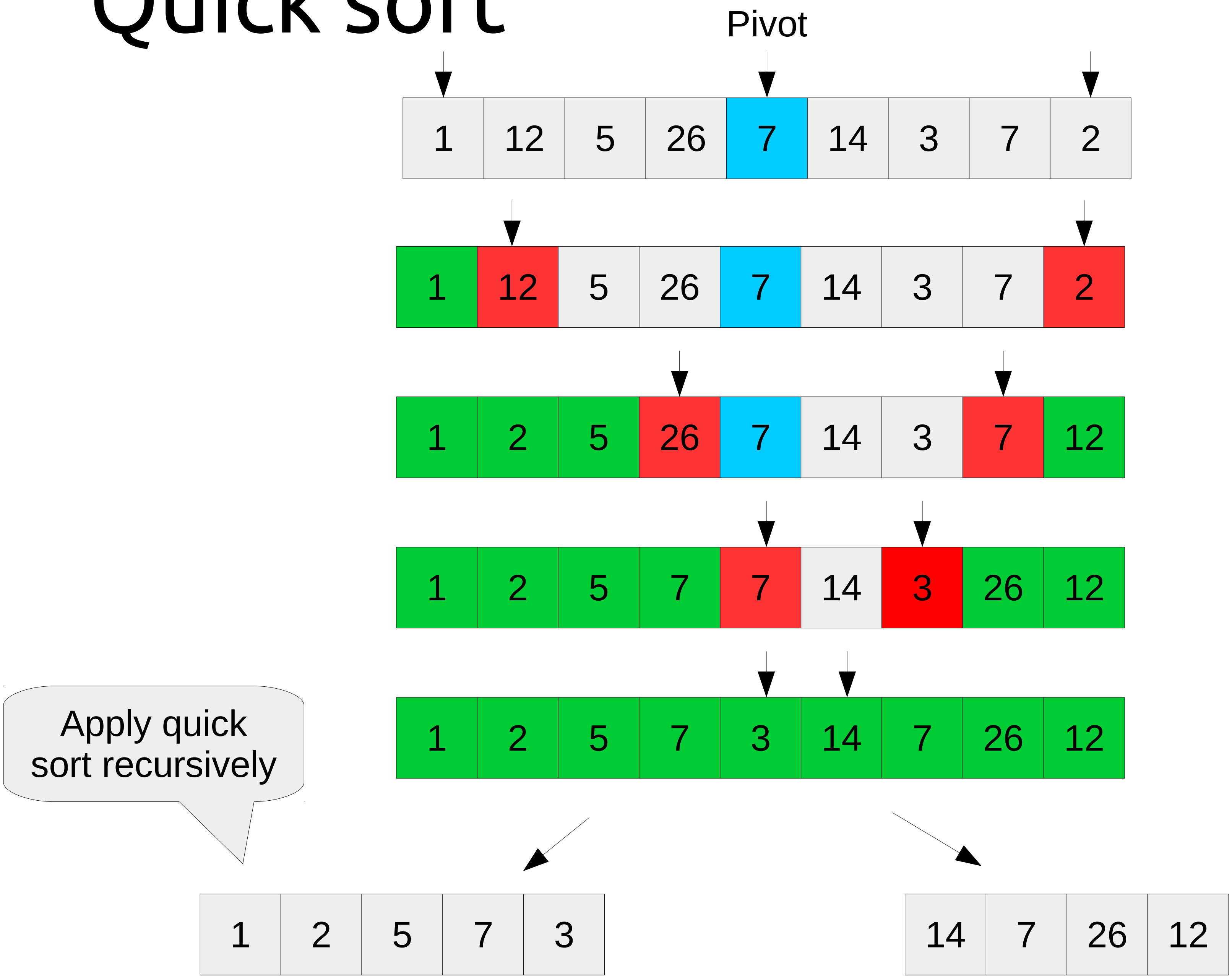
tbb::parallel_reduce



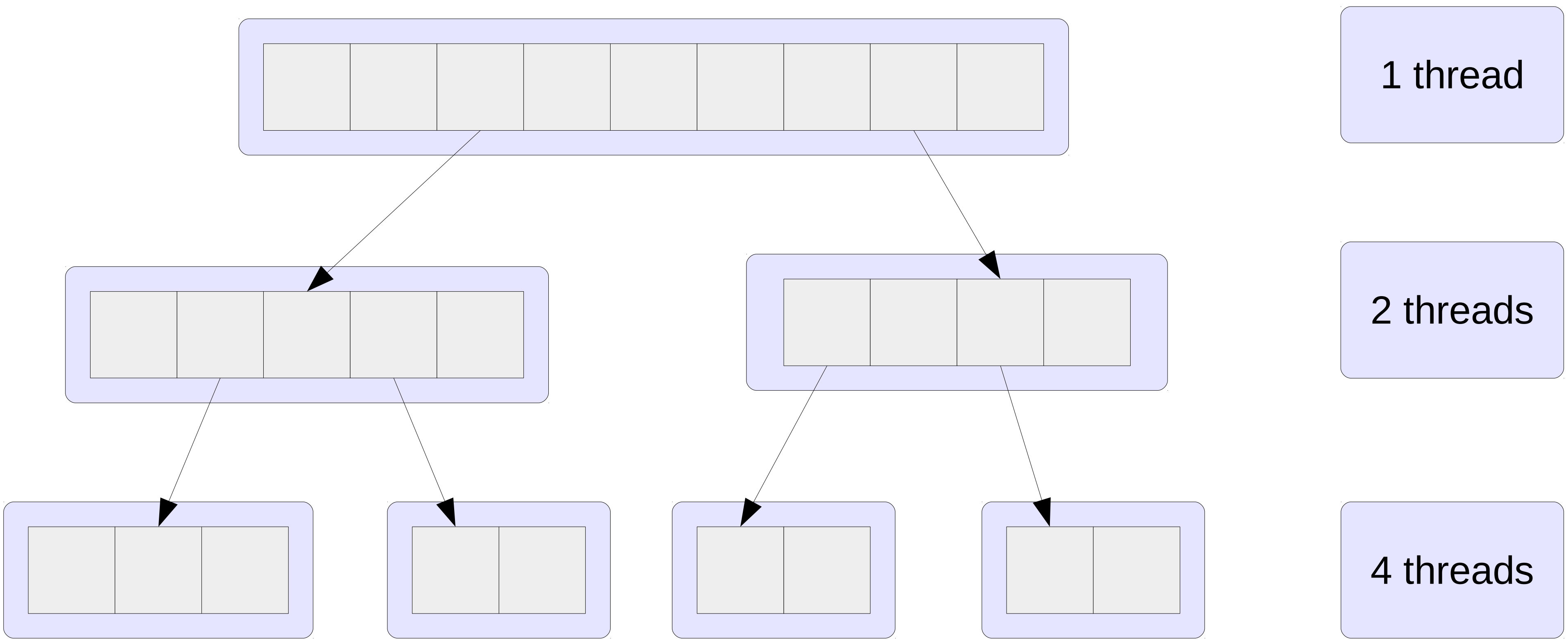
tbb::parallel_reduce



Quick sort


Pivot
1 12 5 26 7 14 3 7 2
1 12 5 26 7 14 3 7 2
1 2 5 26 7 14 3 7 12
1 2 5 7 7 14 3 26 12
1 2 5 7 3 14 7 26 12
Apply quick sort recursively
1 2 5 7 3
14 7 26 12

Parallel quick sort

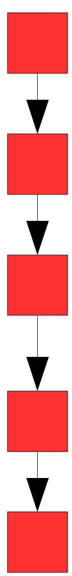


Log(9, 2) ~= 3
Serial = 9 * 3 = 27
Parallel = 9 + 9 / 2 + 9 / 4 = 15.75
Speedup = 27 / 15.75 ~= 1.7x

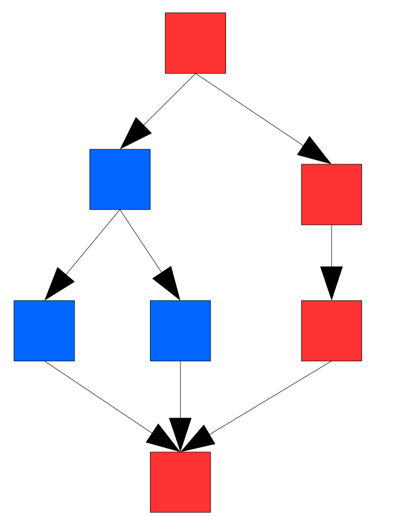
Log(1E6, 2) ~= 20
Serial = 1E6 * 20 = 2E7
Parallel = 1E6 + 1E6 / 2 + 1E6 / 4 + (17 * 1E6 / 8) = 3.9E6
Speedup = 2E7 / 3.9E6 ~= 5.1x

Work-span model

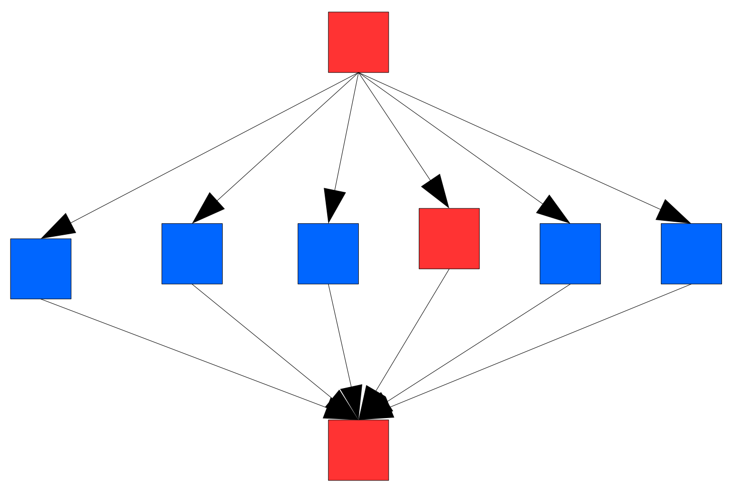
- Amdhal’s law
- Model the processing as directed acyclic graph
- Each node is a work item
- Longest path is the span (depth)



Work = 5, span = 5



Work = 6, span = 4

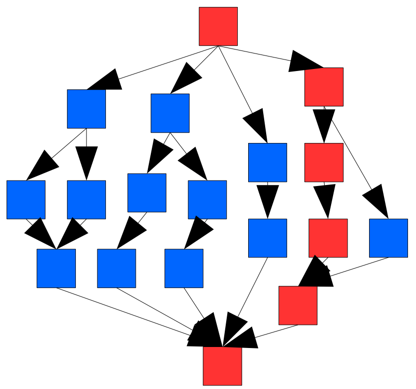


Work = 8, span = 3

Work-span speedup

- Fair lower and upper bound for speedup
- Brent's lemma (using uniform work items)
- Example:
  - 4 cores computer

$$\frac{processors \cdot work}{(processors \cdot span) - span + work} \leqslant S \leqslant \frac{work}{span}$$



Work = 18, span = 6

$$\frac{8 \cdot 18}{(8 \cdot 6) - 6 + 18} \leqslant S \leqslant \frac{18}{6}$$

$$2.4 \leqslant S \leqslant 3$$

$$slack = \frac{work}{processors \cdot span} = \frac{18}{4 \cdot 6} = 0.75$$

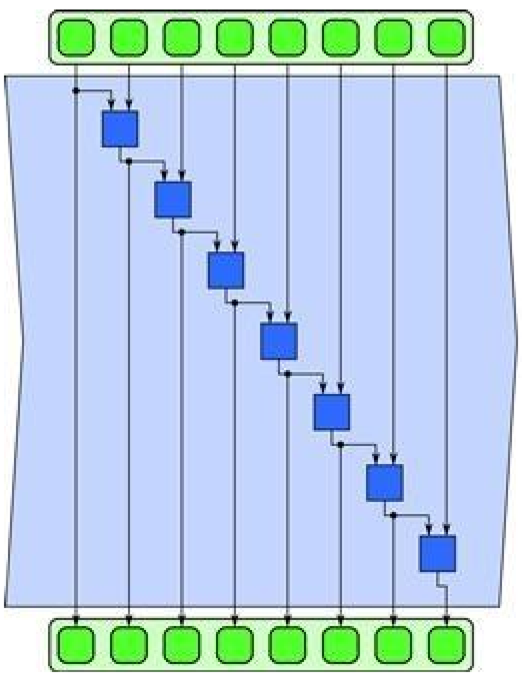
In practice, a slack of 8 is good

Data dependecies

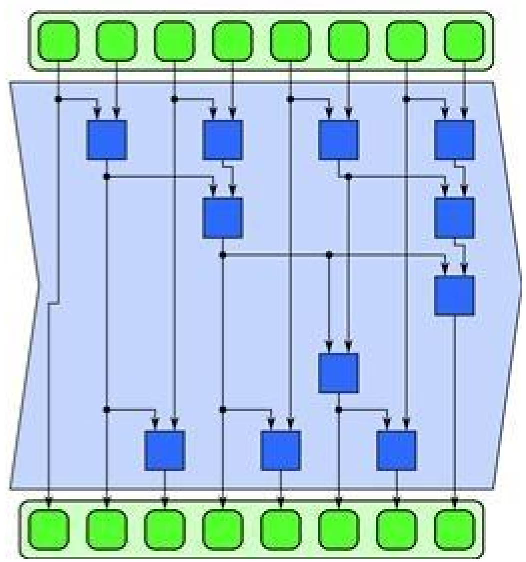
```
b[0] = a[0];
for (int i = 1; i < len; i++) {
    b[i] = b[i - 1] + a[i];
}
```

The current iteration depends on the result of the previous one.

work =7, span = 7



work =11, span = 5



Reduction tree: duplicated work, but breaks dependencies
Work: O(n) -> O(2n)
Span: O(n) -> O(log(n))

See 14-tbb-parallel-scan

Source: Structured Parallel Programming, Patterns for Efficient Computation