THINK PARALLEL: SCANS

Bryce Adelstein Lelbach

Principal Architect

✉ brycelelbach@gmail.com 🐦 @blelbach

THINK PARALLEL: SCANS

Bryce Adelstein Lelbach

Principal Architect

✉ brycelelbach@gmail.com @blelbach

THINK PARALLEL: SCANS

Bryce Adelstein Lelbach

Principal Architect

✉ brycelelbach@gmail.com 🐦 @blelbach

$$y_0 = x_0$$

$$y_1 = x_0 + x_1$$

$$y_2 = x_0 + x_1 + x_2$$

…

$$y_i = y_{i-1} + x_i$$

Commutativity

```
x+y == y+x
```

Not Required

```
4+2 == 2+4
```

```
"a"+"b" != "b"+"a"
```

Commutativity

```
x+y == y+x
```

Not Required

```
4+2 == 2+4
"a"+"b" != "b"+"a"
```

Associativity

```
(x+y)+z == x+(y+z)
```

Required

```
("a"+"b")+"c" == "a"+("b"+"c")
(4+2)+1 == 4+(2+1)
```

Left Identity

```
Φ+x == x
```

Required

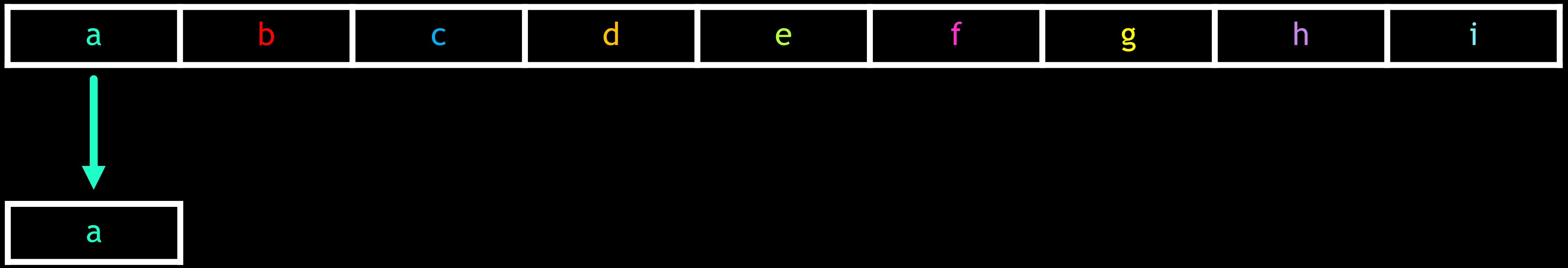
```
""+"a" == "a"
0+4 == 4
```

```
void inclusive_scan(auto&& in) {
  for (auto i : stdv::iota(1, size(in)))
    in[i] = in[i - 1] + in[i];
}
```

```
void inclusive_scan(auto&& in) {
  for (auto i : stdv::iota(1, size(in)))
    in[i] = in[i - 1] + in[i];
}
```

a	b	c	d	e	f	g	h	i

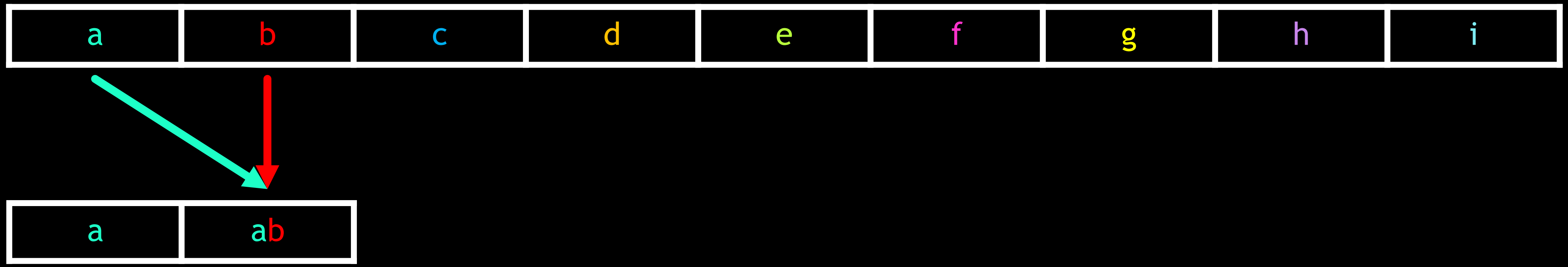
```
void inclusive_scan(auto&& in) {
  for (auto i : stdv::iota(1, size(in)))
    in[i] = in[i - 1] + in[i];
}
```



a	b	c	d	e	f	g	h	i

a

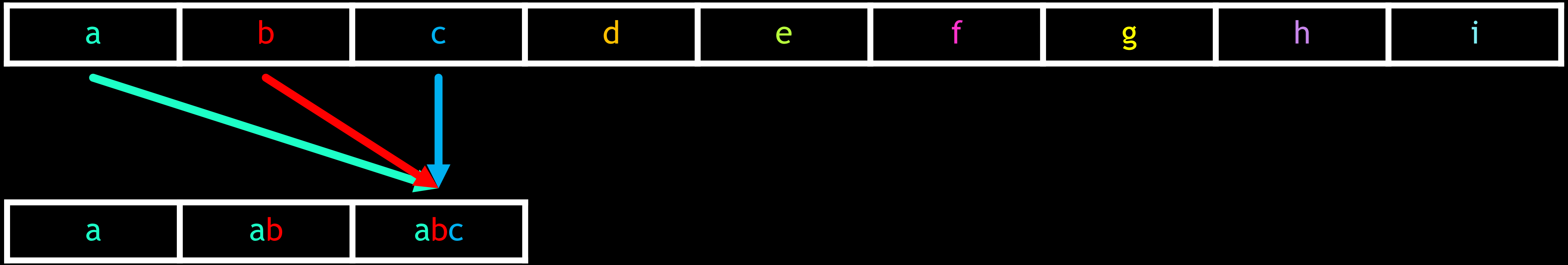
```
void inclusive_scan(auto&& in) {
  for (auto i : stdv::iota(1, size(in)))
    in[i] = in[i - 1] + in[i];
}
```



a	b	c	d	e	f	g	h	i

a	ab

```
void inclusive_scan(auto&& in) {
  for (auto i : stdv::iota(1, size(in)))
    in[i] = in[i - 1] + in[i];
}
```



a	b	c	d	e	f	g	h	i

a	ab	abc

```
void inclusive_scan(auto&& in) {
  for (auto i : stdv::iota(1, size(in)))
    in[i] = in[i - 1] + in[i];
}
```

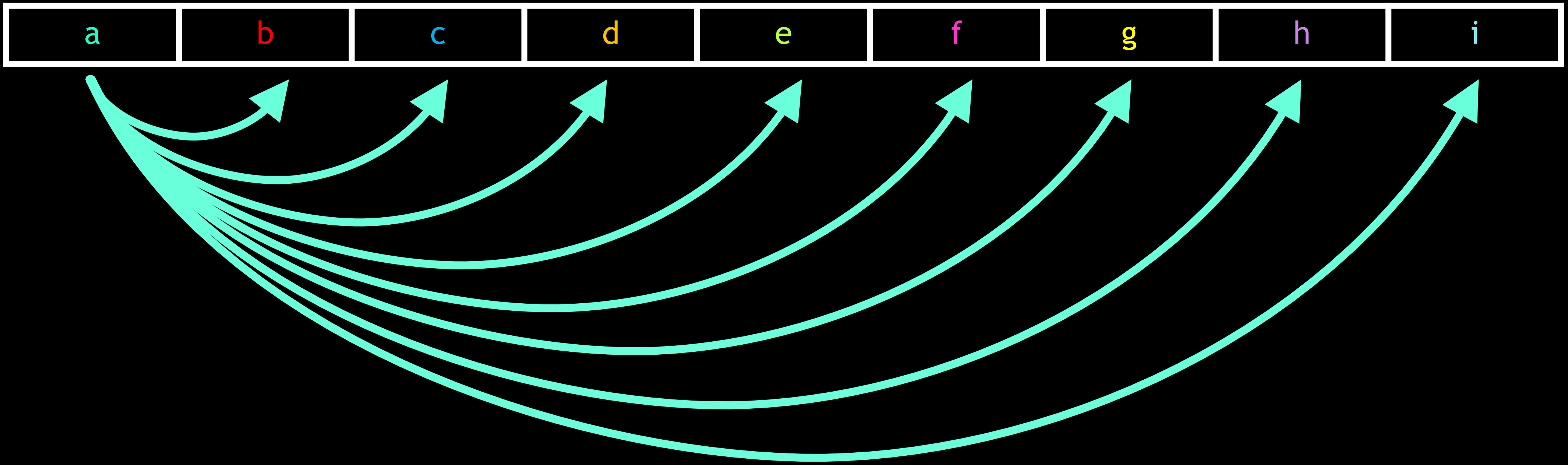
a	b	c	d	e	f	g	h	i

a	ab	abc	abcd	abcde	abcdef	abcdefg	abcdefgh	abcdefghi

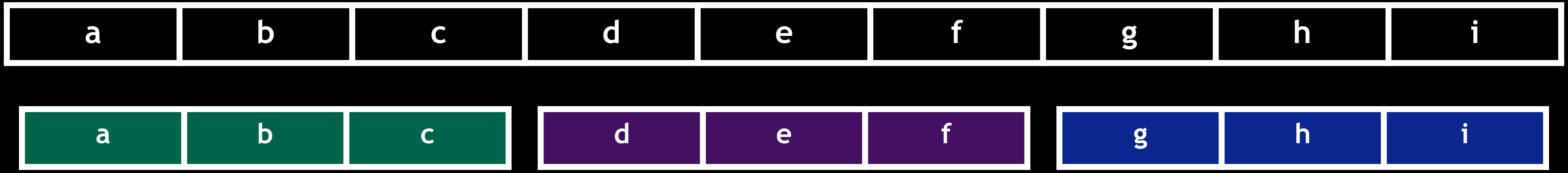
- Distribute
- Calculate
- Communicate

Communication is everything

Everything is communication



a	b	c	d	e	f	g	h	i



a b c d e f g h i

a b c d e f g h i

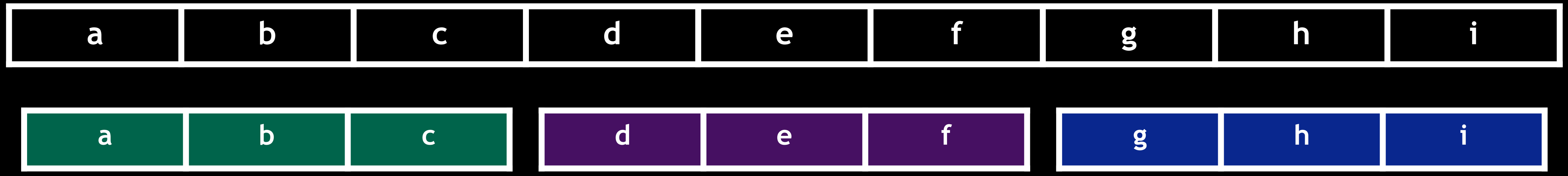
```
(a + b + c)    +    (d + e + f)    +    (g + h + i)
```

```
void inclusive_scan(stdr::range auto&& in, std::size_t num_tiles) {
  …
}
```

```
void inclusive_scan(stdr::range auto&& in, std::size_t num_tiles) {
  …

  stdr::for_each(stde::par, stdv::iota(0, num_tiles),
    …);

  …
}
```

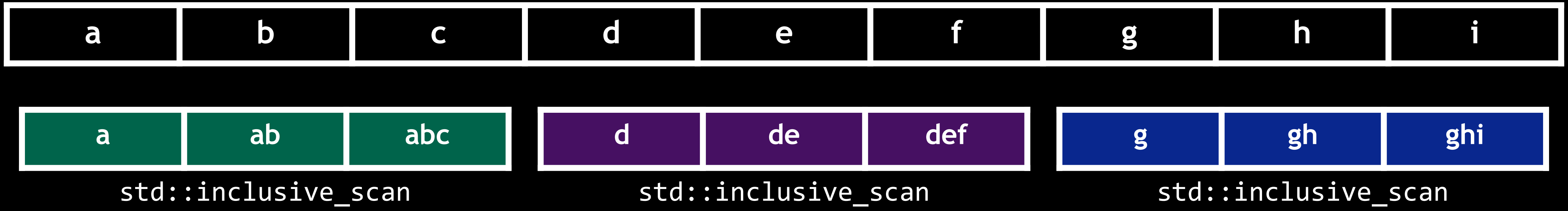
```
void inclusive_scan(stdr::range auto&& in, std::size_t num_tiles) {
  …

  stdr::for_each(stde::par, stdv::iota(0, num_tiles),
    [&] (std::size_t tile) {
      auto sub_in = range_for_tile(in, tile, num_tiles);
      …
    });

  …
}
```

```
auto range_for_tile(stdr::range auto&& in,
                    std::size_t tile,
                    std::size_t num_tiles)
{
  auto tile_size = (size(in) + num_tiles - 1) / num_tiles;
  auto start     = std::min(tile * tile_size, size(in));
  auto end       = std::min((tile + 1) * tile_size, size(in));
  return stdr::subrange(next(begin(in), start), next(begin(in), end));
}
```



a b c d e f g h i

a b c | d e f | g h i

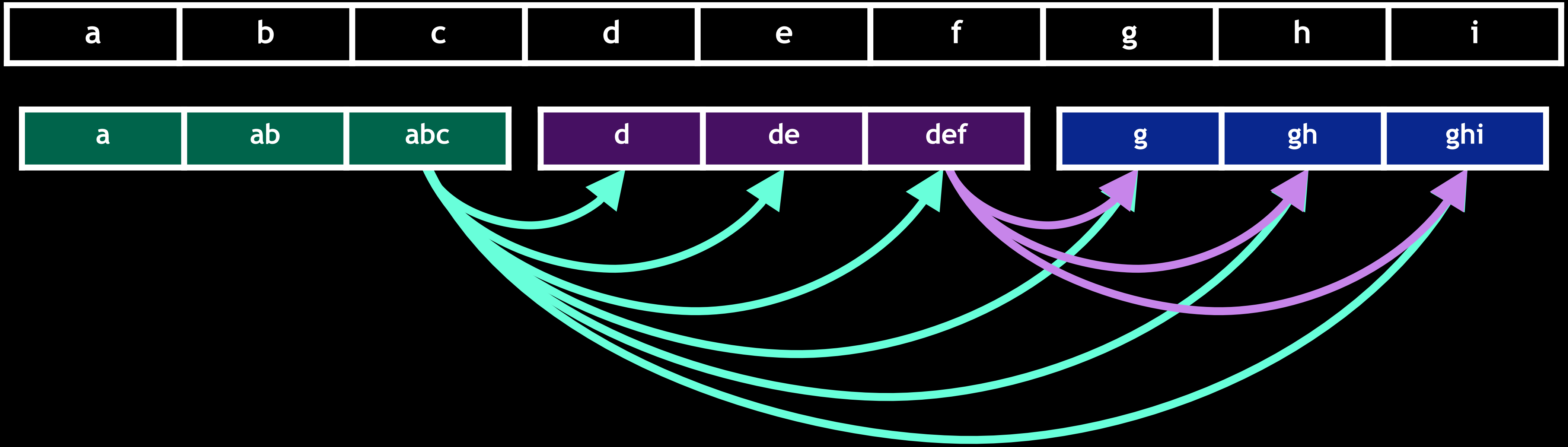


a	b	c	d	e	f	g	h	i

a	ab	abc	d	de	def	g	gh	ghi

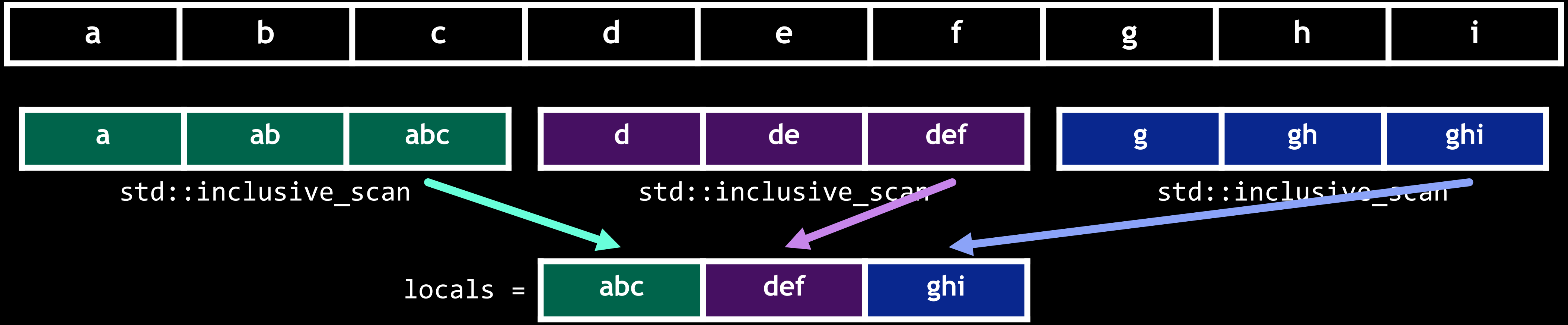
`std::inclusive_scan` `std::inclusive_scan` `std::inclusive_scan`

```
void inclusive_scan(stdr::range auto&& in, std::size_t num_tiles) {
  …

  stdr::for_each(stde::par, stdv::iota(0, num_tiles),
    [&] (std::size_t tile) {
      auto sub_in = range_for_tile(in, tile, num_tiles);
                        stdr::inclusive_scan(sub_in, begin(sub_in));
    });

  …
}
```

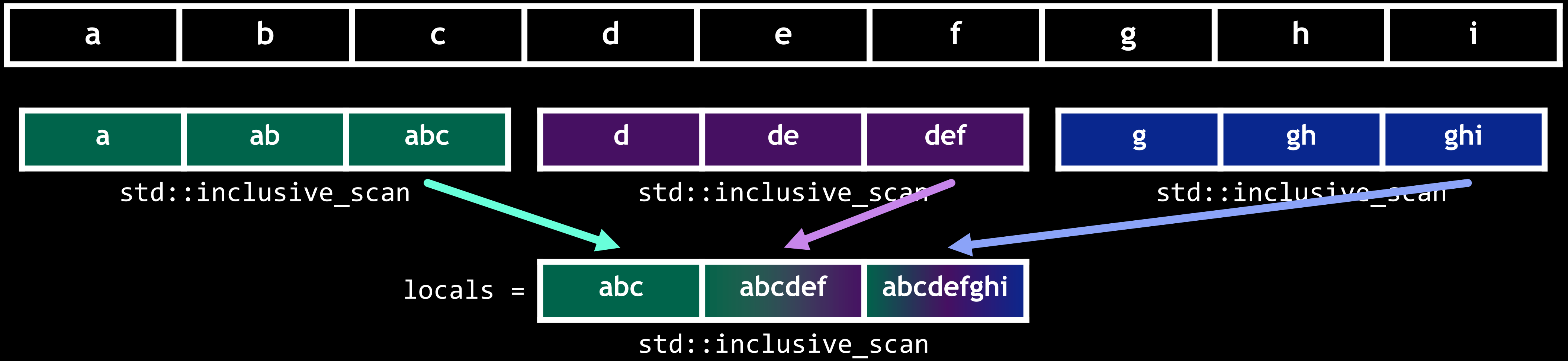

a
b
c
d
e
f
g
h
i
a
ab
abc
d
de
def
g
gh
ghi

```
void inclusive_scan(stdr::range auto&& in, std::size_t num_tiles) {
  std::vector<stdr::range_value_t<decltype(in)>> locals(num_tiles);

  stdr::for_each(stde::par, stdv::iota(0, num_tiles),
    [&] (std::size_t tile) {
      auto sub_in = range_for_tile(in, tile, num_tiles);
                        stdr::inclusive_scan(sub_in, begin(sub_in));
    });

  …
}
```

```
void inclusive_scan(stdr::range auto&& in, std::size_t num_tiles) {
  std::vector<stdr::range_value_t<decltype(in)>> locals(num_tiles);

  stdr::for_each(stde::par, stdv::iota(0, num_tiles),
    [&] (std::size_t tile) {
      auto sub_in = range_for_tile(in, tile, num_tiles);
      locals[tile] = *--stdr::inclusive_scan(sub_in, begin(sub_in));
    });

  …
}
```


a
b
c
d
e
f
g
h
i
a
ab
abc
d
de
def
g
gh
ghi
std::inclusive_scan
std::inclusive_scan
std::inclusive_scan
locals =
abc
def
ghi

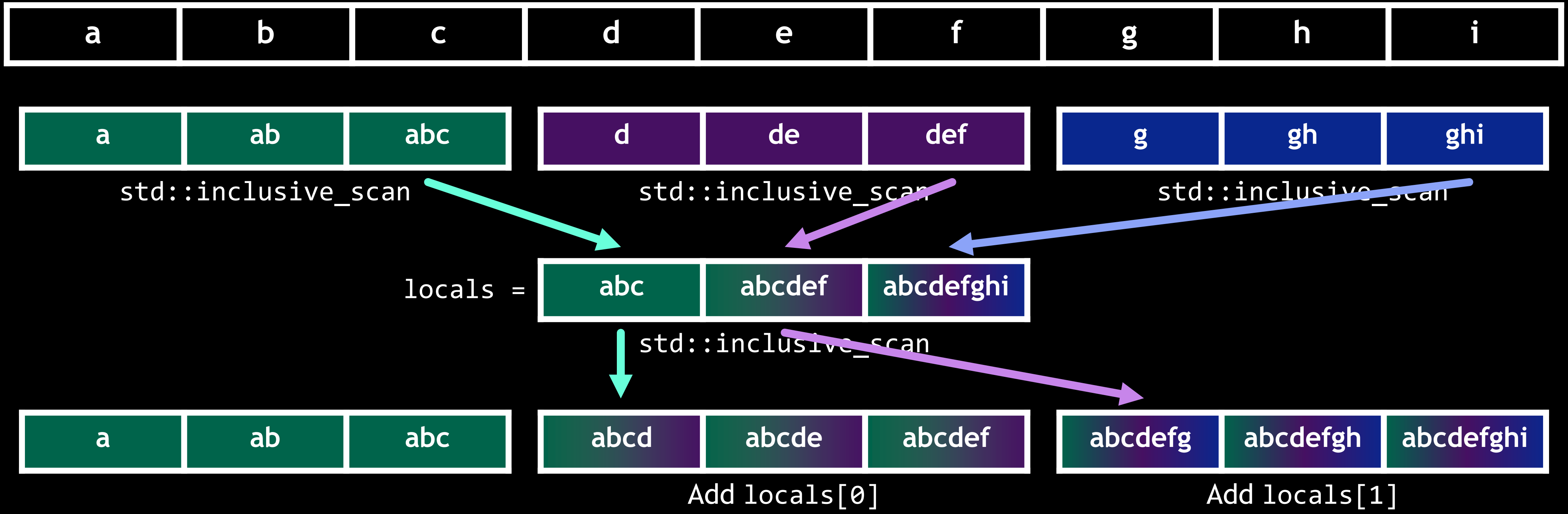
```
void inclusive_scan(stdr::range auto&& in, std::size_t num_tiles) {
  std::vector<stdr::range_value_t<decltype(in)>> locals(num_tiles);

  stdr::for_each(stde::par, stdv::iota(0, num_tiles),
    [&] (std::size_t tile) {
      auto sub_in = range_for_tile(in, tile, num_tiles);
      locals[tile] = *--stdr::inclusive_scan(sub_in, begin(sub_in));
    });

  stdr::inclusive_scan(locals, begin(locals));

  …
}
```


a
b
c
d
e
f
g
h
i
a
ab
abc
d
de
def
g
gh
ghi
std::inclusive_scan
std::inclusive_scan
std::inclusive_scan
locals =
abc
abcdef
abcdefghi
std::inclusive_scan

```
void inclusive_scan(stdr::range auto&& in, std::size_t num_tiles) {
  std::vector<stdr::range_value_t<decltype(in)>> locals(num_tiles);

  stdr::for_each(stde::par, stdv::iota(0, num_tiles),
    [&] (std::size_t tile) {
      auto sub_in = range_for_tile(in, tile, num_tiles);
      locals[tile] = *--stdr::inclusive_scan(sub_in, begin(sub_in));
    });

  stdr::inclusive_scan(locals, begin(locals));

  stdr::for_each(stde::par, stdv::iota(1, num_tiles),
    [&] (std::size_t tile) {
      auto sub_in = range_for_tile(in, tile, num_tiles);
      …
    });
}
```

```
void inclusive_scan(stdr::range auto&& in, std::size_t num_tiles) {
  std::vector<stdr::range_value_t<decltype(in)>> locals(num_tiles);

  stdr::for_each(stde::par, stdv::iota(0, num_tiles),
    [&] (std::size_t tile) {
      auto sub_in = range_for_tile(in, tile, num_tiles);
      locals[tile] = *--stdr::inclusive_scan(sub_in, begin(sub_in));
    });

  stdr::inclusive_scan(locals, begin(locals));

  stdr::for_each(stde::par, stdv::iota(1, num_tiles),
    [&] (std::size_t tile) {
      auto sub_in = range_for_tile(in, tile, num_tiles);
      …
    });
}
```

```
void inclusive_scan(stdr::range auto&& in, std::size_t num_tiles) {
  std::vector<stdr::range_value_t<decltype(in)>> locals(num_tiles);

  stdr::for_each(stde::par, stdv::iota(0, num_tiles),
    [&] (std::size_t tile) {
      auto sub_in = range_for_tile(in, tile, num_tiles);
      locals[tile] = *--stdr::inclusive_scan(sub_in, begin(sub_in));
    });

  stdr::inclusive_scan(locals, begin(locals));

  stdr::for_each(stde::par, stdv::iota(1, num_tiles),
    [&] (std::size_t tile) {
      auto sub_in = range_for_tile(in, tile, num_tiles);
      stdr::for_each(sub_in, [&] (auto& e) { e = locals[tile - 1] + e; });
    });
}
```



a	b	c	d	e	f	g	h	i

a	ab	abc

std::inclusive_scan

d	de	def

std::inclusive_scan

g	gh	ghi

std::inclusive_scan

`locals =`

abc	abcdef	abcdefghi

std::inclusive_scan

a	ab	abc

abcd	abcde	abcdef

Add locals[0]

abcdefg	abcdefgh	abcdefghi

Add locals[1]

```
void inclusive_scan(stdr::range auto&& in, std::size_t num_tiles) {
  std::vector<stdr::range_value_t<decltype(in)>> locals(num_tiles);

  stdr::for_each(stde::par, stdv::iota(0, num_tiles),
    [&] (std::size_t tile) {
      auto sub_in = range_for_tile(in, tile, num_tiles);
      locals[tile] = *--stdr::inclusive_scan(sub_in, begin(sub_in));
    });

  stdr::inclusive_scan(locals, begin(locals));

  stdr::for_each(stde::par, stdv::iota(1, num_tiles)
    [&] (std::size_t tile) {
      auto sub_in = range_for_tile(in, tile, num_tiles);
      stdr::for_each(sub_in, [&] (auto& e) { e = locals[tile - 1] + e; });
    });
}
```

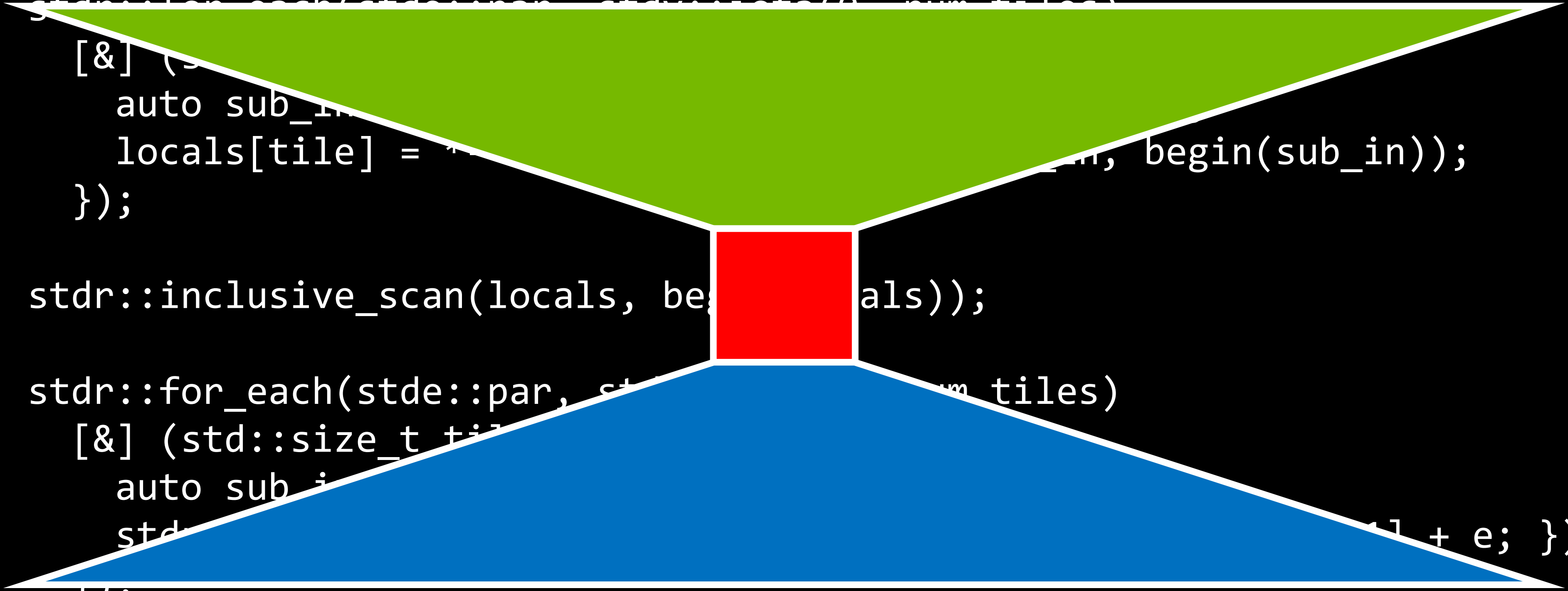
```
void inclusive_scan(stdr::range auto&& in, std::size_t num_tiles) {
  std::vector<stdr::range_value_t<decltype(in)>> locals(num_tiles);

  stdr::for_each(stde::par, stdv::iota(0, num_tiles),
    [&] (std::size_t tile) {
      auto sub_in = range_for_tile(in, tile, num_tiles);
      locals[tile] = *--                            egin(sub_in));
    });

  stdr::inclusive_scan(l

  stdr::for_each(stde::par, stdv::iota(1, num_tiles),
    [&] (std::size_t tile) {
      auto sub_in = range_for_tile(in, tile, num_tiles);
      stdr::for_each(sub_in, [&] (auto& e) { e = locals[tile - 1] + e; });
    });
}
```

Analysis

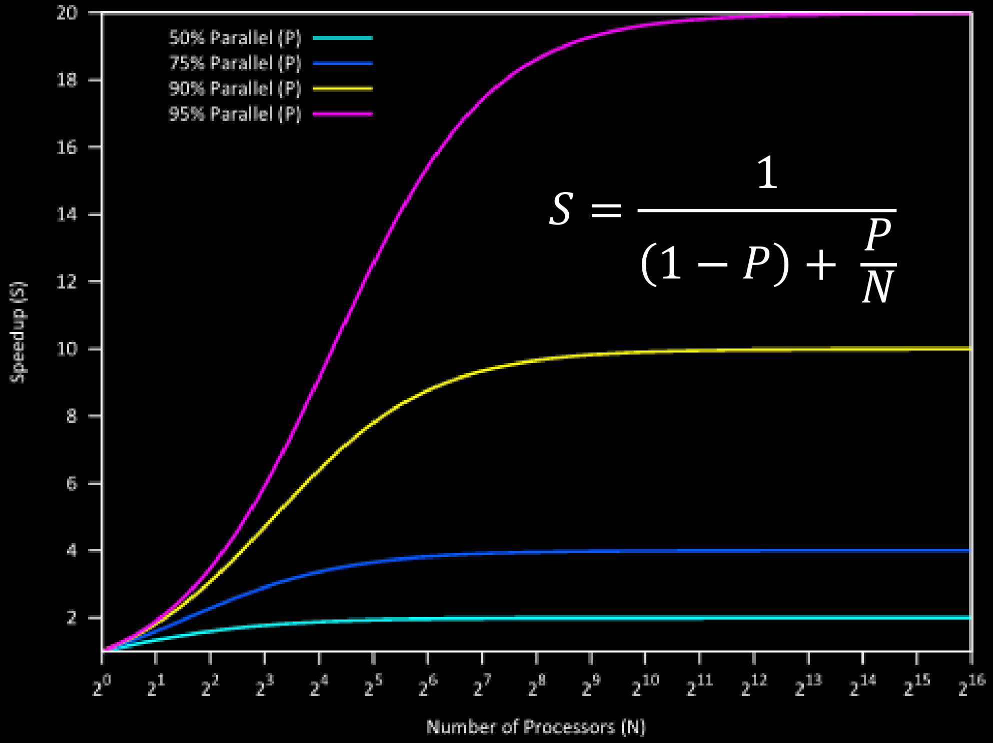
O(input) storage

2 global synchronizations

```
void inclusive_scan(stdr::range auto&& in, std::size_t num_tiles) {
  std::vector<stdr::range_value_t<decltype(in)>> locals(num_tiles);

  stdr::for_each(stde::par, stdv::iota(0, num_tiles),
    [&] (s
      auto sub_in
      locals[tile] = *                              n, begin(sub_in));
    });

  stdr::inclusive_scan(locals, be          als));

  stdr::for_each(stde::par, st               m_tiles)
    [&] (std::size_t til
      auto sub_i
      std                                                 1] + e; });
    });
}
```

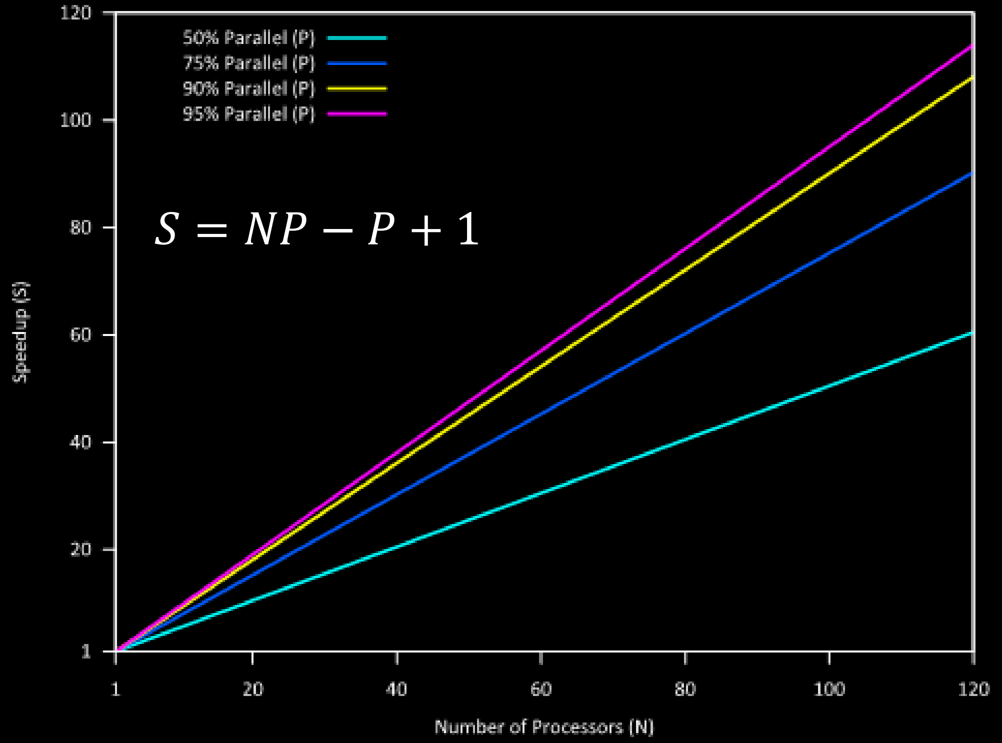


Amdahl's Law (Strong Scaling)



$$S = \frac{1}{(1 - P) + \frac{P}{N}}$$

Gustafson’s Law (Weak Scaling)

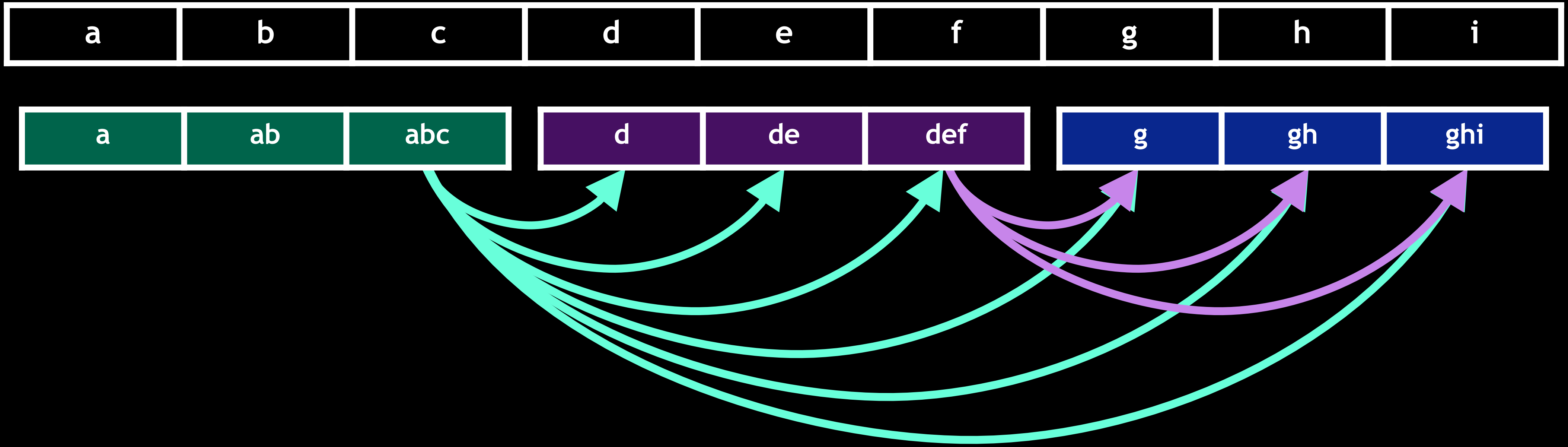


$$S = NP - P + 1$$

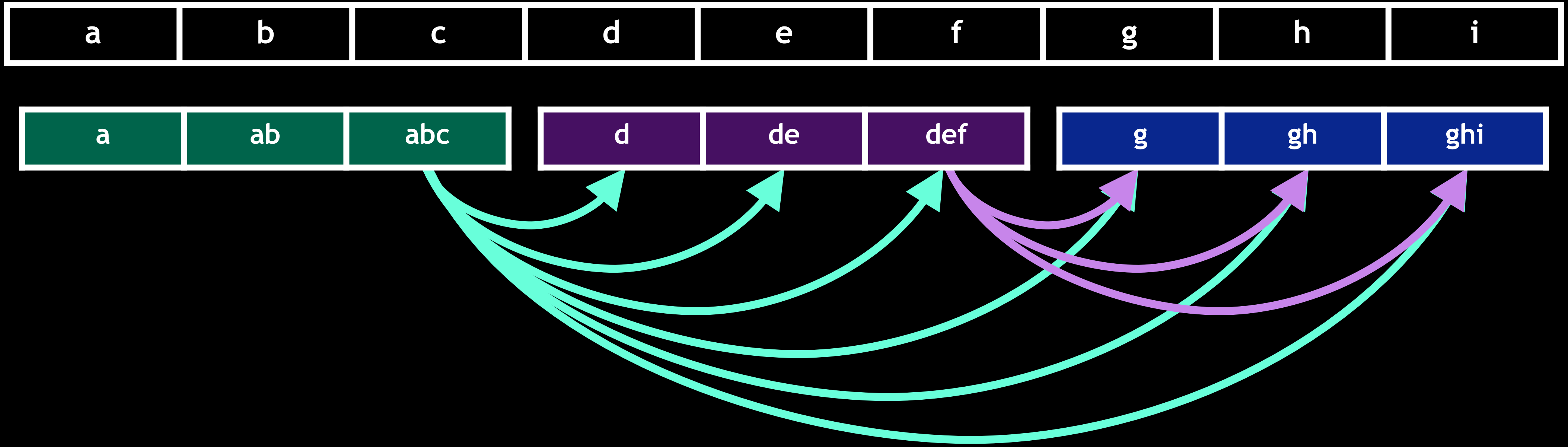
S: Speedup
P: Proportion of parallel code
N: Number of processors

- **Localize synchronization**
- **Hide latency**

What does each tile depend on?



What does each tile depend on?



Only the tiles preceding it!

```
void inclusive_scan(stdr::range auto&& in, std::size_t num_tiles) {
  scan_tile_state<stdr::range_value_t<decltype(in)>> sts(num_tiles);


  …
}
```

```
void inclusive_scan(stdr::range auto&& in, std::size_t num_tiles) {
  scan_tile_state<stdr::range_value_t<decltype(in)>> sts(num_tiles);


  stdr::for_each(stde::par, stdv::iota(0, num_tiles),
    [&] (std::size_t tile) {
      …
    });
}
```

```
void inclusive_scan(stdr::range auto&& in, std::size_t num_tiles) {
  scan_tile_state<stdr::range_value_t<decltype(in)>> sts(num_tiles);


  stdr::for_each(stde::par, stdv::iota(0, num_tiles),
    [&] (std::size_t tile) {

      auto sub_in = range_for_tile(in, tile, num_tiles);

      …
    });
}
```

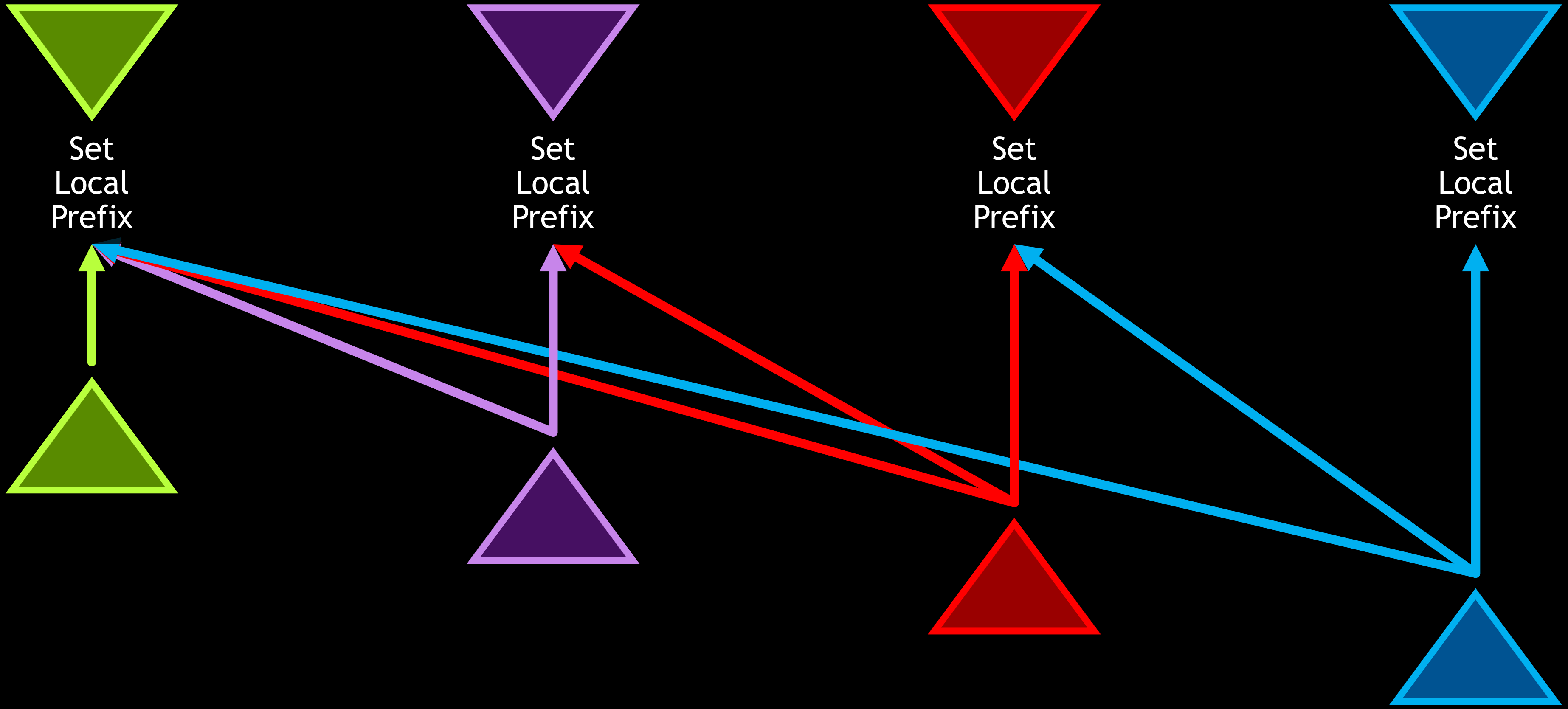
```
void inclusive_scan(stdr::range auto&& in, std::size_t num_tiles) {
  scan_tile_state<stdr::range_value_t<decltype(in)>> sts(num_tiles);


  stdr::for_each(stde::par, stdv::iota(0, num_tiles),
    [&] (std::size_t tile) {

      auto sub_in = range_for_tile(in, tile, num_tiles);


           stdr::inclusive_scan(sub_in, begin(sub_in))

      …
    });
}
```

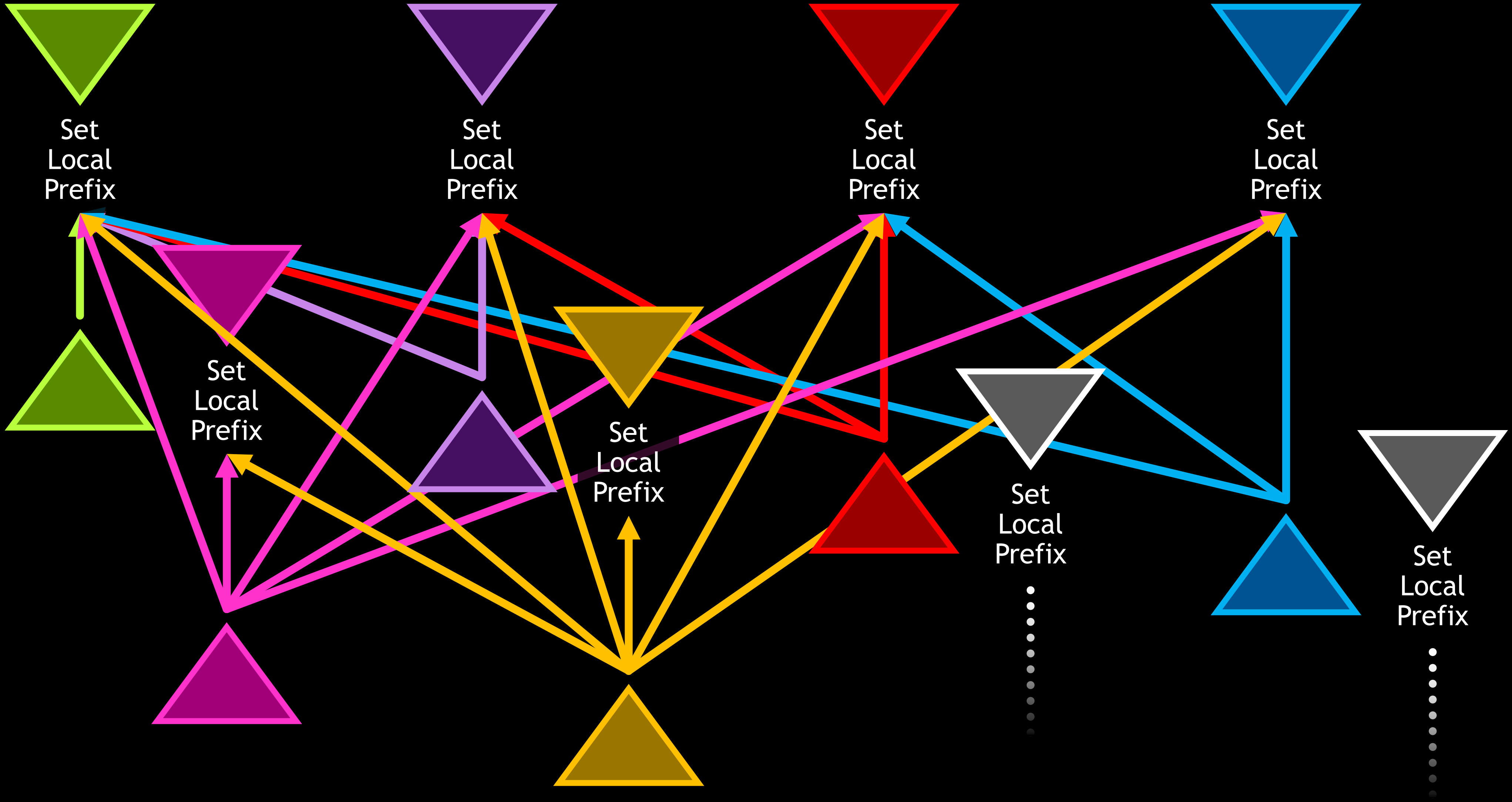
```
void inclusive_scan(stdr::range auto&& in, std::size_t num_tiles) {
  scan_tile_state<stdr::range_value_t<decltype(in)>> sts(num_tiles);


  stdr::for_each(stde::par, stdv::iota(0, num_tiles),
    [&] (std::size_t tile) {

      auto sub_in = range_for_tile(in, tile, num_tiles);

      sts.set_local_prefix(tile,
        *--stdr::inclusive_scan(sub_in, begin(sub_in)));

      …
    });
}
```

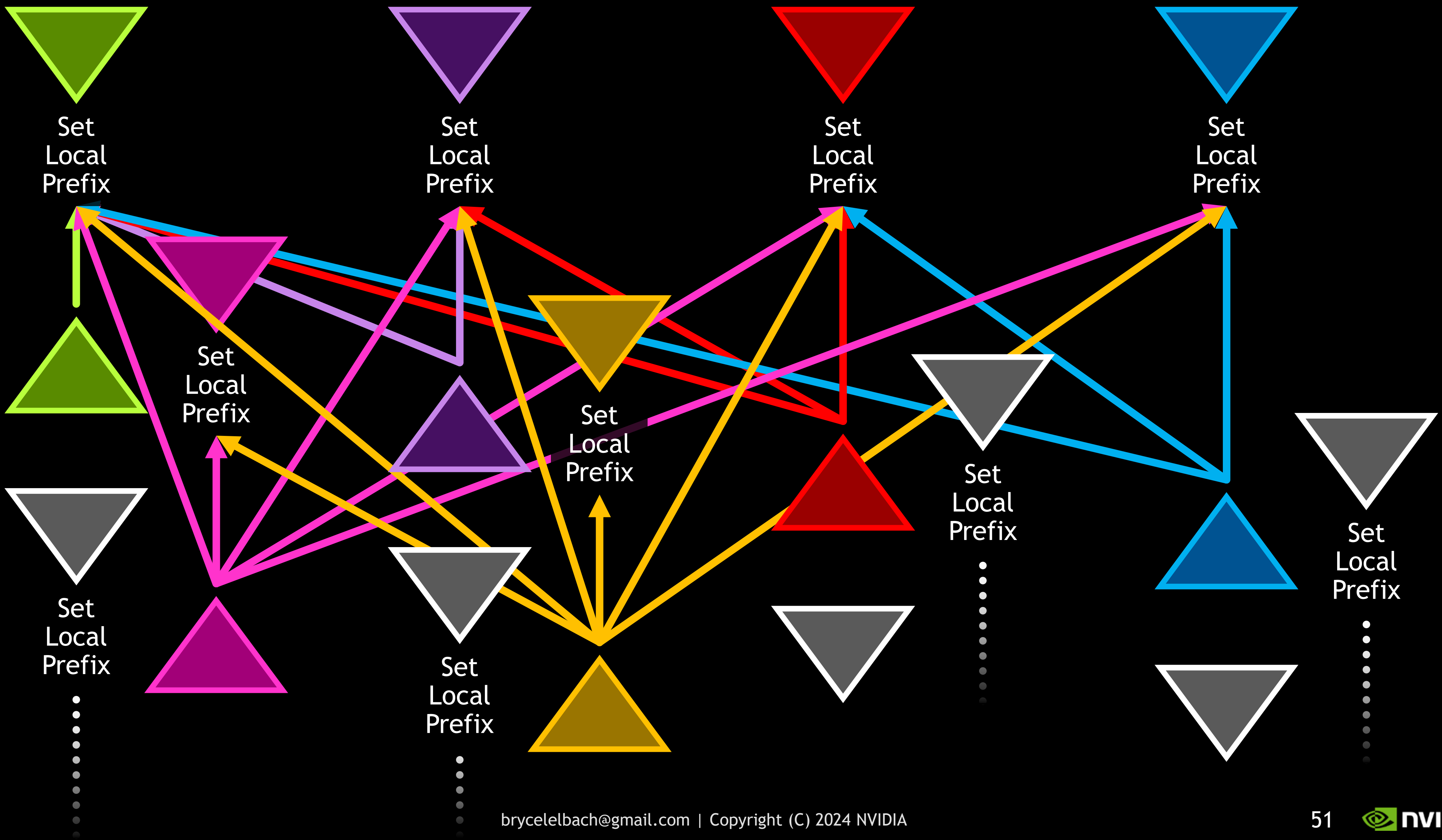
```
void inclusive_scan(stdr::range auto&& in, std::size_t num_tiles) {
  scan_tile_state<stdr::range_value_t<decltype(in)>> sts(num_tiles);


  stdr::for_each(stde::par, stdv::iota(0, num_tiles),
    [&] (std::size_t tile) {

      auto sub_in = range_for_tile(in, tile, num_tiles);

      sts.set_local_prefix(tile,
        *--stdr::inclusive_scan(sub_in, begin(sub_in)));

      if (tile != 0) {
        …
      }
    });
}
```

```
void inclusive_scan(stdr::range auto&& in, std::size_t num_tiles) {
  scan_tile_state<stdr::range_value_t<decltype(in)>> sts(num_tiles);


  stdr::for_each(stde::par, stdv::iota(0, num_tiles),
    [&] (std::size_t tile) {

      auto sub_in = range_for_tile(in, tile, num_tiles);

      sts.set_local_prefix(tile,
        *--stdr::inclusive_scan(sub_in, begin(sub_in)));

      if (tile != 0) {
        auto pred = sts.wait_for_predecessor_prefix(tile);
        …
      }
    });
}
```

```
void inclusive_scan(stdr::range auto&& in, std::size_t num_tiles) {
  scan_tile_state<stdr::range_value_t<decltype(in)>> sts(num_tiles);


  stdr::for_each(stde::par, stdv::iota(0, num_tiles),
    [&] (std::size_t tile) {

      auto sub_in = range_for_tile(in, tile, num_tiles);

      sts.set_local_prefix(tile,
        *--stdr::inclusive_scan(sub_in, begin(sub_in)));

      if (tile != 0) {
        auto pred = sts.wait_for_predecessor_prefix(tile);
        stdr::for_each(sub_in, [&] (auto& e) { e = pred + e; });
      }
    });
}
```

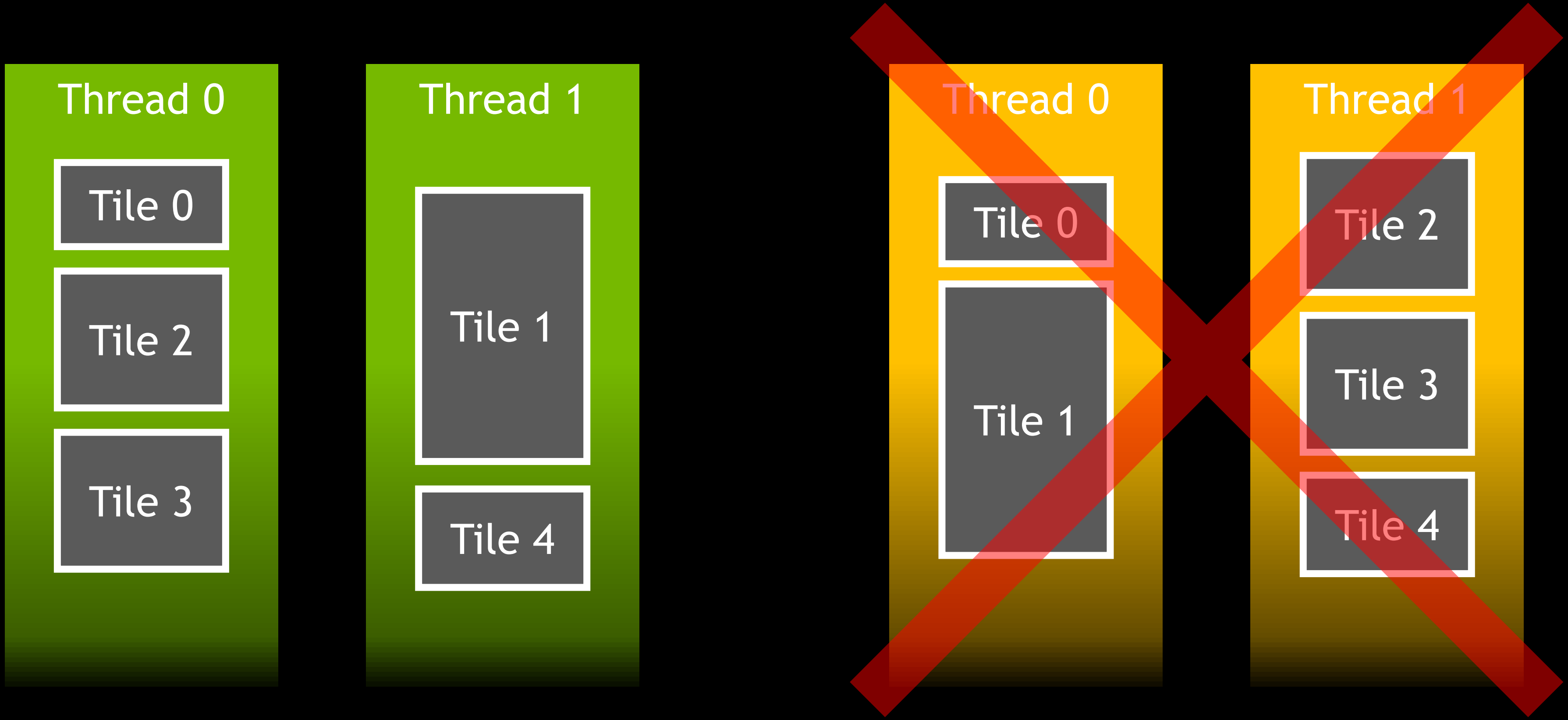

Set Local Prefix
Set Local Prefix
Set Local Prefix
Set Local Prefix


Set Local Prefix
Set Local Prefix
Set Local Prefix
Set Local Prefix
Set Local Prefix
Set Local Prefix
Set Local Prefix
Set Local Prefix


Set Local Prefix
Set Local Prefix
Set Local Prefix
Set Local Prefix
Set Local Prefix
Set Local Prefix
Set Local Prefix
Set Local Prefix
Set Local Prefix
Set Local Prefix
Set Local Prefix

Monotonic Progress

If tile X is executing, all tiles < X must be executing or completed.



```
void inclusive_scan(stdr::range auto&& in, std::size_t num_tiles) {
  scan_tile_state<stdr::range_value_t<decltype(in)>> sts(num_tiles);


  stdr::for_each(stde::par, stdv::iota(0, num_tiles),
    [&] (std::size_t tile) {

      auto sub_in = range_for_tile(in, tile, num_tiles);

      sts.set_local_prefix(tile,
        *--stdr::inclusive_scan(sub_in, begin(sub_in)));

      if (tile != 0) {
        auto pred = sts.wait_for_predecessor_prefix(tile);
        stdr::for_each(sub_in, [&] (auto& e) { e = pred + e; });
      }
    });
}
```

```
void inclusive_scan(stdr::range auto&& in, std::size_t num_tiles) {
  scan_tile_state<stdr::range_value_t<decltype(in)>> sts(num_tiles);
  std::atomic<std::size_t> tile_counter(0);

  stdr::for_each(stde::par, stdv::iota(0, num_tiles),
    [&] (std::size_t) {
      auto tile = tile_counter.fetch_add(1, std::memory_order_relaxed);
      auto sub_in = range_for_tile(in, tile, num_tiles);

      sts.set_local_prefix(tile,
        *--stdr::inclusive_scan(sub_in, begin(sub_in)));

      if (tile != 0) {
        auto pred = sts.wait_for_predecessor_prefix(tile);
        stdr::for_each(sub_in, [&] (auto& e) { e = pred + e; });
      }
    });
}
```

```
void inclusive_scan(stdr::range auto&& in, std::size_t num_tiles) {
  scan_tile_state<stdr::range_value_t<decltype(in)>> sts(num_tiles);
  std::atomic<std::size_t> tile_counter(0);

  stdr::for_each(stde::par, stdv::iota(0, num_tiles),
    [&] (std::size_t) {
      auto tile = tile_counter.fetch_add(1, std::memory_order_relaxed);
      auto sub_in = range_for_tile(in, tile, num_tiles);

      sts.set_local_prefix(tile,
        *--stdr::inclusive_scan(sub_in, begin(sub_in)));

      if (tile != 0) {
        auto pred = sts.wait_for_predecessor_prefix(tile);
        stdr::for_each(sub_in, [&] (auto& e) { e = pred + e; });
      }
    });
}
```

```
template <typename T>
struct scan_tile_state {
  …

  struct descriptor {
    …
  };

  std::vector<descriptor> prefixes;

  …
};
```

```
template <typename T>
struct scan_tile_state {
  …

  struct descriptor {
    T local = {};
    T complete = {};
    …
  };

  std::vector<descriptor> prefixes;

  …
};
```

```
template <typename T>
struct scan_tile_state {
  enum status { status_unavailable, status_local, status_complete };

  struct descriptor {
    T local = {};
    T complete = {};
    std::atomic<status> state = status_unavailable;
  };

  std::vector<descriptor> prefixes;

  …
};
```

```
template <typename T>
struct scan_tile_state {
  enum status { status_unavailable, status_local, status_complete };

  struct descriptor {
    T local = {};
    T complete = {};
    std::atomic<status> state = status_unavailable;
  };

  std::vector<descriptor> prefixes;

  scan_tile_state(std::size_t num_tiles) : prefixes(num_tiles) {}

  …
};
```

```
template <typename T>
struct scan_tile_state {
  enum status { status_unavailable, status_local, status_complete };

  struct descriptor {
    T local = {};
    T complete = {};
    std::atomic<status> state = status_unavailable;
  };

  std::vector<descriptor> prefixes;

  scan_tile_state(std::size_t num_tiles) : prefixes(num_tiles) {}

  void set_local_prefix(std::size_t i, T local);

  …
};
```

```
void scan_tile_state<T>::set_local_prefix(std::size_t i, T local) {
  if (i == 0) {
    …
  }
  …
}
```

```
void scan_tile_state<T>::set_local_prefix(std::size_t i, T local) {
  if (i == 0) {
    prefixes[i].local = local;
    prefixes[i].complete = local;
    …
  } else {
    …
  }
  …
}
```

```
void scan_tile_state<T>::set_local_prefix(std::size_t i, T local) {
  if (i == 0) {
    prefixes[i].local = local;
    prefixes[i].complete = local;
    prefixes[i].state.store(status_complete, std::memory_order_release);
  } else {
    …
  }
  …
}
```

```
void scan_tile_state<T>::set_local_prefix(std::size_t i, T local) {
  if (i == 0) {
    prefixes[i].local = local;
    prefixes[i].complete = local;
    prefixes[i].state.store(status_complete, std::memory_order_release);
  } else {
    prefixes[i].local = local;
    …
  }
  …
}
```

```
void scan_tile_state<T>::set_local_prefix(std::size_t i, T local) {
  if (i == 0) {
    prefixes[i].local = local;
    prefixes[i].complete = local;
    prefixes[i].state.store(status_complete, std::memory_order_release);
  } else {
    prefixes[i].local = local;
    prefixes[i].state.store(status_local, std::memory_order_release);
  }
  …
}
```

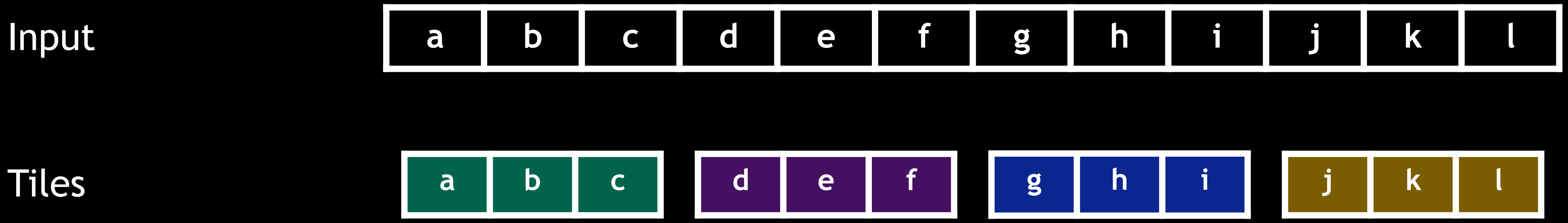
```
void scan_tile_state<T>::set_local_prefix(std::size_t i, T local) {
  if (i == 0) {
    prefixes[i].local = local;
    prefixes[i].complete = local;
    prefixes[i].state.store(status_complete, std::memory_order_release);
  } else {
    prefixes[i].local = local;
    prefixes[i].state.store(status_local, std::memory_order_release);
  }
  prefixes[i].state.notify_all();
}
```

```
template <typename T>
struct scan_tile_state {
  enum status { status_unavailable, status_local, status_complete };

  struct descriptor {
    T local = {};
    T complete = {};
    std::atomic<status> state = status_unavailable;
  };

  std::vector<descriptor> prefixes;

  scan_tile_state(std::size_t num_tiles) : prefixes(num_tiles) {}

  void set_local_prefix(std::size_t i, T local);

  T wait_for_predecessor_prefix(std::size_t i);
};
```

Prefix	Sum of…
Local	Elements in tile X

Prefix	Sum of…
Local	Elements in tile X
Predecessor (not stored)	Elements before tile X AKA Local prefixes from tiles < X

Prefix	Sum of…
Local	Elements in tile X
Predecessor (not stored)	Elements before tile X AKA Local prefixes from tiles < X
Complete	Elements up to the end of tile X AKA Local prefixes from tiles <= X AKA Predecessor prefix + local prefix

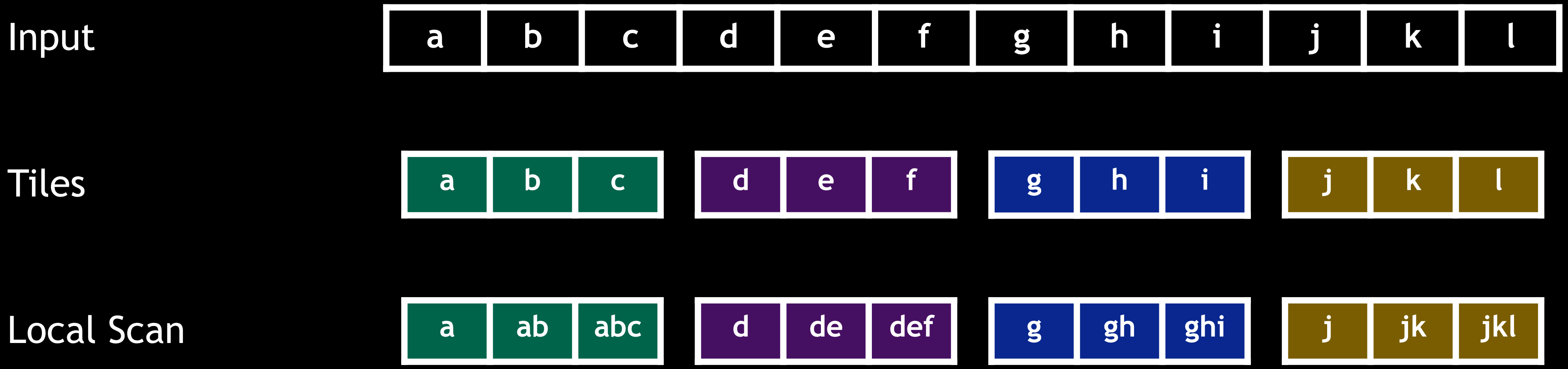


Input

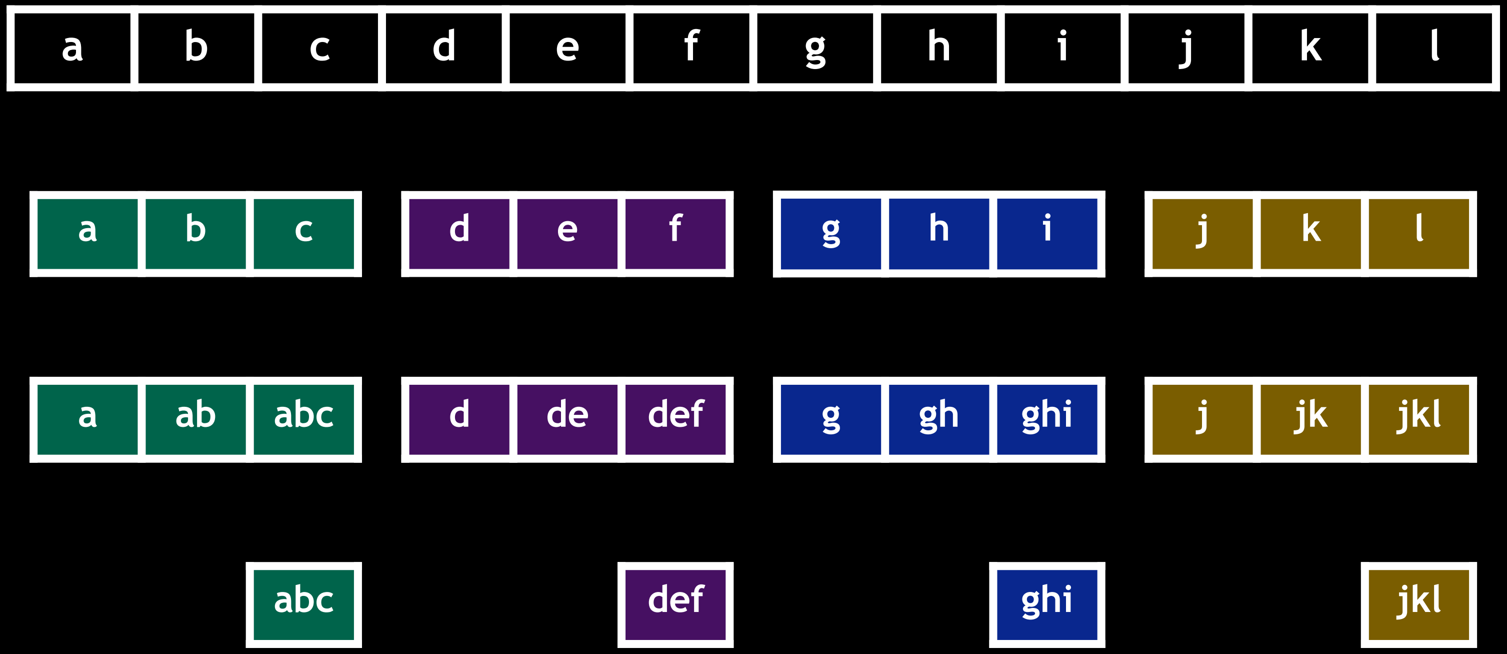
a b c d e f g h i j k l

Tiles

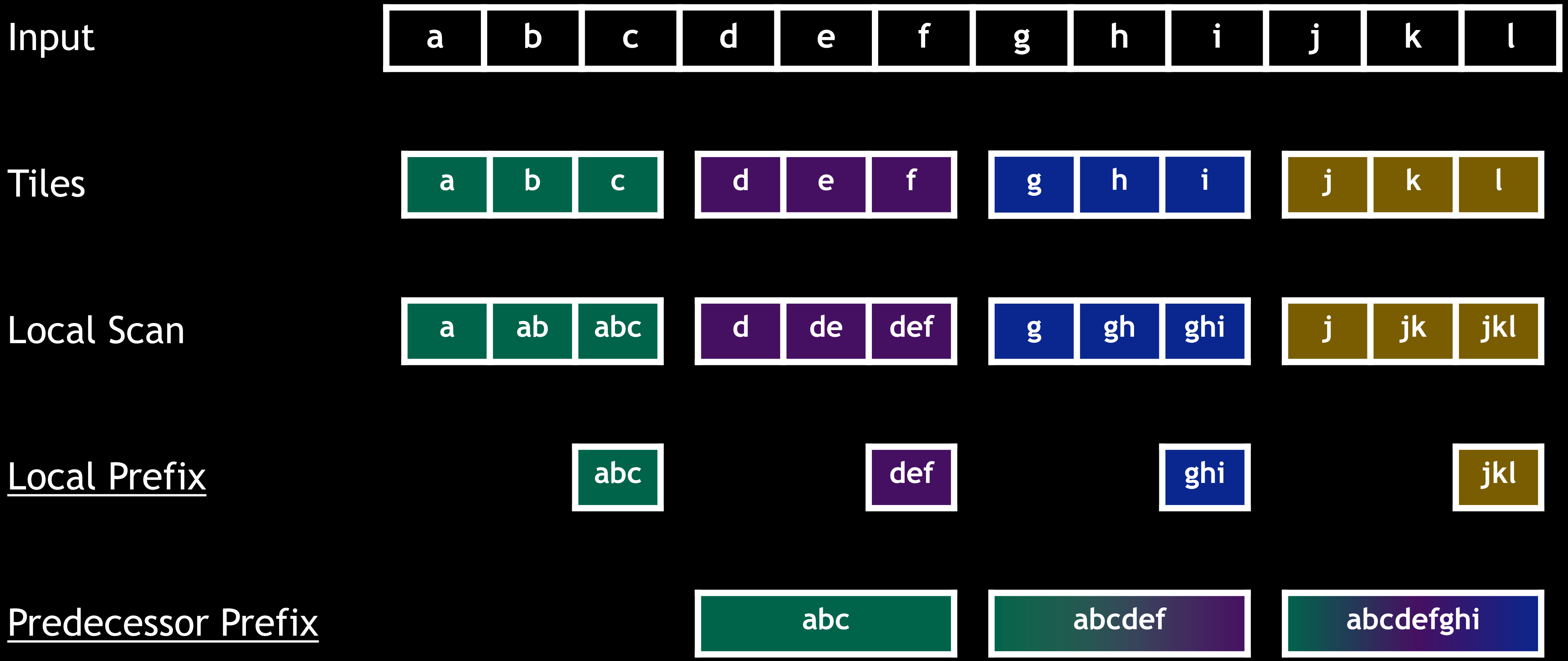
a b c | d e f | g h i | j k l



Input	a	b	c	d	e	f	g	h	i	j	k	l
Tiles	a	b	c	d	e	f	g	h	i	j	k	l
Local Scan	a	ab	abc	d	de	def	g	gh	ghi	j	jk	jkl



Input	a	b	c	d	e	f	g	h	i	j	k	l
Tiles	a	b	c	d	e	f	g	h	i	j	k	l
Local Scan	a	ab	abc	d	de	def	g	gh	ghi	j	jk	jkl
<u>Local Prefix</u>			abc			def			ghi			jkl



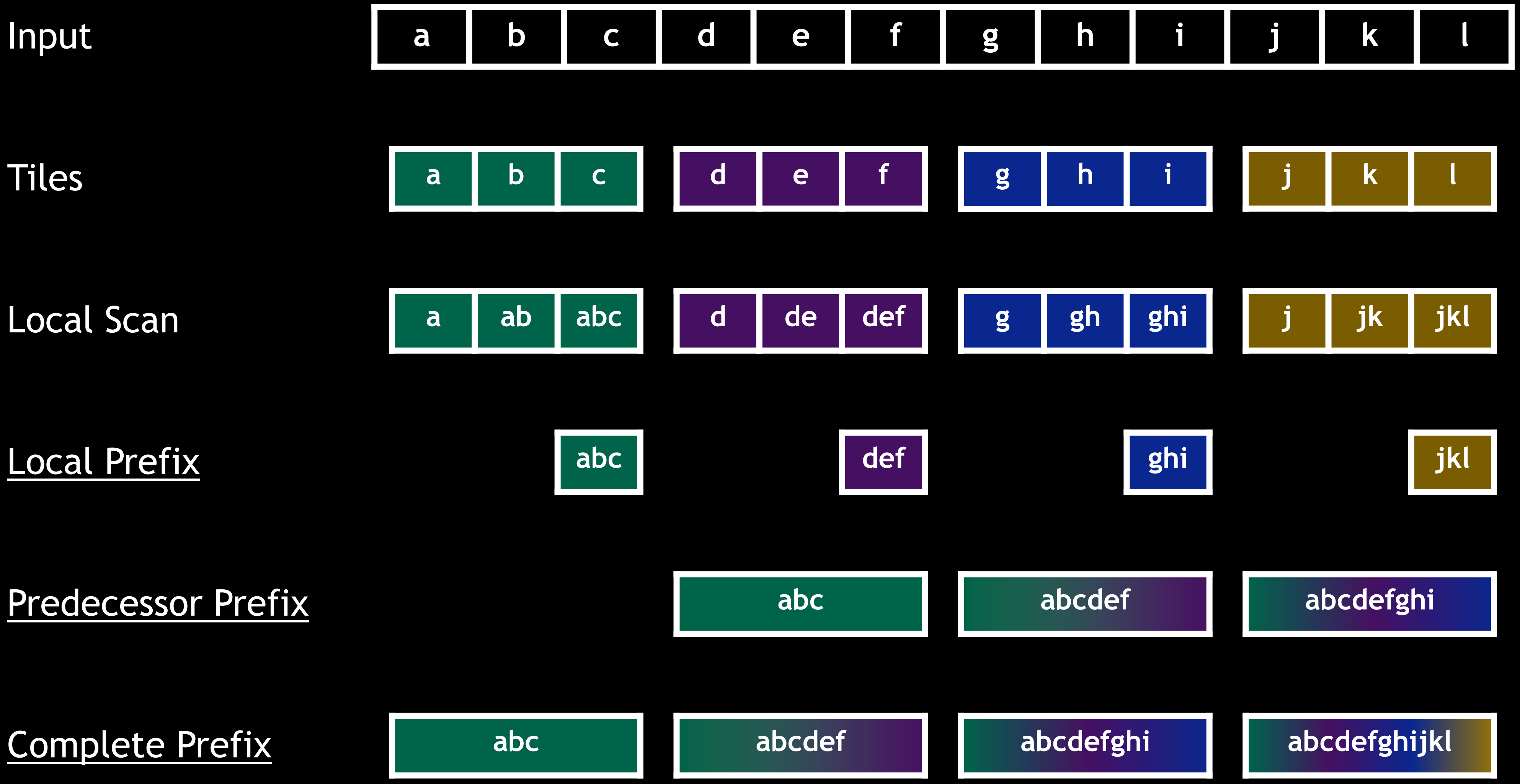
Input: a | b | c | d | e | f | g | h | i | j | k | l

Tiles: a b c | d e f | g h i | j k l

Local Scan: a ab abc | d de def | g gh ghi | j jk jkl

Local Prefix: abc | def | ghi | jkl

Predecessor Prefix: abc | abcdef | abcdefghi



Input	a	b	c	d	e	f	g	h	i	j	k	l
Tiles	a	b	c	d	e	f	g	h	i	j	k	l
Local Scan	a	ab	abc	d	de	def	g	gh	ghi	j	jk	jkl
Local Prefix			abc			def			ghi			jkl
Predecessor Prefix				abc			abcdef			abcdefghi		
Complete Prefix	abc			abcdef			abcdefghi			abcdefghijkl		

Tile X queries Tile X-1, X-2, …, 0:

- If it is unavailable, wait.
- If it is local, add it to the predecessor prefix & continue.
- If it is complete, add it to the predecessor prefix & terminate.

Tile X queries Tile X-1, X-2, …, 0:

- If it is unavailable, wait.

Tile X queries Tile X-1, X-2, …, 0:

- If it is unavailable, wait.
- If it is local, add it to the predecessor prefix & continue.

Tile X queries Tile X-1, X-2, …, 0:

- If it is unavailable, wait.
- If it is local, add it to the predecessor prefix & continue.
- If it is complete, add it to the predecessor prefix & terminate.

Decoupled: Each tile searches independently.

Lookback: Each tile searches backwards from its position.

```
T scan_tile_state<T>::wait_for_predecessor_prefix(std::size_t i) {
  T predecessor_prefix = {};
  …
}
```

```
T scan_tile_state<T>::wait_for_predecessor_prefix(std::size_t i) {
 T predecessor_prefix = {};
 for (std::intptr_t p = i - 1; p >= 0; --p) {
  …
 }

 …
}
```

```
T scan_tile_state<T>::wait_for_predecessor_prefix(std::size_t i) {
  T predecessor_prefix = {};
  for (std::intptr_t p = i - 1; p >= 0; --p) {
    prefixes[p].state.wait(status_unavailable, std::memory_order_acquire);
    …
  }

  …
}
```

```
T scan_tile_state<T>::wait_for_predecessor_prefix(std::size_t i) {
  T predecessor_prefix = {};
  for (std::intptr_t p = i - 1; p >= 0; --p) {
    prefixes[p].state.wait(status_unavailable, std::memory_order_acquire);
    state = prefixes[p].state.load(std::memory_order_acquire);
    …
  }

  …
}
```

```
T scan_tile_state<T>::wait_for_predecessor_prefix(std::size_t i) {
  T predecessor_prefix = {};
  for (std::intptr_t p = i - 1; p >= 0; --p) {
    prefixes[p].state.wait(status_unavailable, std::memory_order_acquire);
    state = prefixes[p].state.load(std::memory_order_acquire);
    if (state == status_local) {
      …
    }
    …
  }

  …
}
```

```
T scan_tile_state<T>::wait_for_predecessor_prefix(std::size_t i) {
 T predecessor_prefix = {};
 for (std::intptr_t p = i - 1; p >= 0; --p) {
  prefixes[p].state.wait(status_unavailable, std::memory_order_acquire);
  state = prefixes[p].state.load(std::memory_order_acquire);
  if (state == status_local) {
   predecessor_prefix = prefixes[p].local + predecessor_prefix;
  }
  …
 }

 …
}
```

```
T scan_tile_state<T>::wait_for_predecessor_prefix(std::size_t i) {
  T predecessor_prefix = {};
  for (std::intptr_t p = i - 1; p >= 0; --p) {
    prefixes[p].state.wait(status_unavailable, std::memory_order_acquire);
    state = prefixes[p].state.load(std::memory_order_acquire);
    if (state == status_local) {
      predecessor_prefix = prefixes[p].local + predecessor_prefix;
    } else if (state == status_complete) {
      …
    }
  }

  …
}
```

```
T scan_tile_state<T>::wait_for_predecessor_prefix(std::size_t i) {
  T predecessor_prefix = {};
  for (std::intptr_t p = i - 1; p >= 0; --p) {
    prefixes[p].state.wait(status_unavailable, std::memory_order_acquire);
    state = prefixes[p].state.load(std::memory_order_acquire);
    if (state == status_local) {
      predecessor_prefix = prefixes[p].local + predecessor_prefix;
    } else if (state == status_complete) {
      predecessor_prefix = prefixes[p].complete + predecessor_prefix;
      …
    }
  }

  …
}
```

```
T scan_tile_state<T>::wait_for_predecessor_prefix(std::size_t i) {
  T predecessor_prefix = {};
  for (std::intptr_t p = i - 1; p >= 0; --p) {
    prefixes[p].state.wait(status_unavailable, std::memory_order_acquire);
    state = prefixes[p].state.load(std::memory_order_acquire);
    if (state == status_local) {
      predecessor_prefix = prefixes[p].local + predecessor_prefix;
    } else if (state == status_complete) {
      predecessor_prefix = prefixes[p].complete + predecessor_prefix;
      break;
    }
  }

  …
}
```

```
T scan_tile_state<T>::wait_for_predecessor_prefix(std::size_t i) {
  T predecessor_prefix = {};
  for (std::intptr_t p = i - 1; p >= 0; --p) {
    prefixes[p].state.wait(status_unavailable, std::memory_order_acquire);
    state = prefixes[p].state.load(std::memory_order_acquire);
    if (state == status_local) {
      predecessor_prefix = prefixes[p].local + predecessor_prefix;
    } else if (state == status_complete) {
      predecessor_prefix = prefixes[p].complete + predecessor_prefix;
      break;
    }
  }

  prefixes[i].complete = predecessor_prefix + prefixes[i].local;
  …
}
```

```
T scan_tile_state<T>::wait_for_predecessor_prefix(std::size_t i) {
  T predecessor_prefix = {};
  for (std::intptr_t p = i - 1; p >= 0; --p) {
    prefixes[p].state.wait(status_unavailable, std::memory_order_acquire);
    state = prefixes[p].state.load(std::memory_order_acquire);
    if (state == status_local) {
      predecessor_prefix = prefixes[p].local + predecessor_prefix;
    } else if (state == status_complete) {
      predecessor_prefix = prefixes[p].complete + predecessor_prefix;
      break;
    }
  }

  prefixes[i].complete = predecessor_prefix + prefixes[i].local;
  prefixes[i].state.store(status_complete, std::memory_order_release);
  prefixes[i].state.notify_all();

  …
}
```

```
T scan_tile_state<T>::wait_for_predecessor_prefix(std::size_t i) {
  T predecessor_prefix = {};
  for (std::intptr_t p = i - 1; p >= 0; --p) {
    prefixes[p].state.wait(status_unavailable, std::memory_order_acquire);
    state = prefixes[p].state.load(std::memory_order_acquire);
    if (state == status_local) {
      predecessor_prefix = prefixes[p].local + predecessor_prefix;
    } else if (state == status_complete) {
      predecessor_prefix = prefixes[p].complete + predecessor_prefix;
      break;
    }
  }

  prefixes[i].complete = predecessor_prefix + prefixes[i].local;
  prefixes[i].state.store(status_complete, std::memory_order_release);
  prefixes[i].state.notify_all();

  return predecessor_prefix;
}
```

Tile	Status	Local	Complete
0	Complete	a	a
1	Local	b	ab
2	Local	c	
3	Local	d	
4	Local	e	
5	Local	f	
6	Unavailable		
7	Local	h	

Tile	Status	Local	Complete
0	Complete	a	a
1	Local	b	ab
2	Local	c	
3	Local	d	
4	Local	e	
5	Local	f	
6	Unavailable		
7	Local	h	

Tile 3

Tile	Status	Local	Complete
0	Complete	a	a
1	Local	b	ab
2	*Local*	*c*	
3	Local	d	
4	Local	e	
5	Local	f	
6	Unavailable		
7	Local	h	

Tile 3

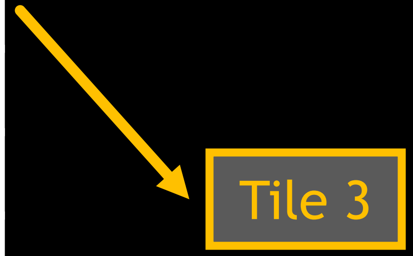
```
pred = c
```

Tile	Status	Local	Complete
0	Complete	a	a
1	*Local*	*b*	*ab*
2	Local	c	
3	Local	d	
4	Local	e	
5	Local	f	
6	Unavailable		
7	Local	h	

Tile 3

```
pred = bc
```

Tile	Status	Local	Complete
0	*Complete*	*a*	*a*
1	Local	b	ab
2	Local	c	
3	Local	d	
4	Local	e	
5	Local	f	
6	Unavailable		
7	Local	h	



Tile 3

```
pred = abc
```

Tile	Status	Local	Complete
0	Complete	a	a
1	Local	b	ab
2	Local	c	
3	Complete	d	abcd
4	Local	e	
5	Local	f	
6	Unavailable		
7	Local	h	

Tile 3

```
pred = abc
```

Tile	Status	Local	Complete
0	Complete	a	a
1	Local	b	ab
2	Local	c	
3	Complete	d	abcd
4	Local	e	
5	Local	f	
6	Unavailable		
7	Local	h	

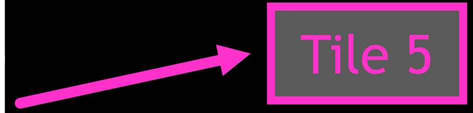
Tile 5

Tile	Status	Local	Complete
0	Complete	a	a
1	Local	b	ab
2	Local	c	
3	Complete	d	abcd
4	*Local*	*e*	
5	Local	f	
6	Unavailable		
7	Local	h	

Tile 5

`pred = e`

Tile	Status	Local	Complete
0	Complete	a	a
1	Local	b	ab
2	Local	c	
3	*Complete*	*d*	*abcd*
4	Local	e	
5	Local	f	
6	Unavailable		
7	Local	h	



`pred = abcde`

Tile	Status	Local	Complete
0	Complete	a	a
1	Local	b	ab
2	Local	c	
3	Complete	d	abcd
4	Local	e	
5	Complete	f	abcdef
6	Unavailable		
7	Local	h	

Tile 5

`pred = abcde`

```
void inclusive_scan(stdr::range auto&& in, std::size_t num_tiles) {
  scan_tile_state<stdr::range_value_t<decltype(in)>> sts(num_tiles);
  std::atomic<std::size_t> tile_counter(0);

  stdr::for_each(stde::par, stdv::iota(0, num_tiles),
    [&] (std::size_t) {
      auto tile = tile_counter.fetch_add(1, std::memory_order_relaxed);
      auto sub_in = range_for_tile(in, tile, num_tiles);

      sts.set_local_prefix(tile,
        *--stdr::inclusive_scan(sub_in, begin(sub_in)));

      if (tile != 0) {
        auto pred = sts.wait_for_predecessor_prefix(tile);
        stdr::for_each(sub_in, [&] (auto& e) { e = pred + e; });
      }
    });
};
```

```
void inclusive_scan(stdr::range auto&& in, std::size_t num_tiles) {
  scan_tile_state<stdr::range_value_t<decltype(in)>> sts(num_tiles);
  std::atomic<std::size_t

  stdr::for_each(stde::pa
    [&] (std::size_t) {
      auto tile = tile_co                            ry_order_relaxed);
      auto sub_in = range                            s);

      sts.set_local_prefix(tile,
        *--stdr::inclusive_scan(sub_in, begin(sub_in)));

      if (tile != 0) {
        auto pred = sts.wait_for_predecessor_prefix(tile);
        stdr::for_each(sub_in, [&] (auto& e) { e = pred + e; });
      }
    });
};
```

Analysis

O(tiles) storage

1 global synchronization

```
void inclusive_scan(stdr::range auto&& in, std::size_t num_tiles) {
  scan_tile_state<stdr::range_value_t<decltype(in)>> sts(num_tiles);
  std::atomic<std::size_t

  stdr::for_each(stde::pa
    [&] (std::size_t) {
      auto tile = tile_co                                ry_order_relaxed);
      auto sub_in = range                                s);

      sts.set_local_prefix(tile,
        *--stdr::inclusive_scan(sub_in, begin(sub_in)));

      if (til
        auto
        stdr:
      }
    });
};
```

Analysis

O(tiles) storage

1 global synchronization

Performance

3x faster than two pass implementation

NVC++ 24.3, 2x 32 core EPYC 7513, 4GB 32 bit int input, 1024 tiles

Scan is a building block

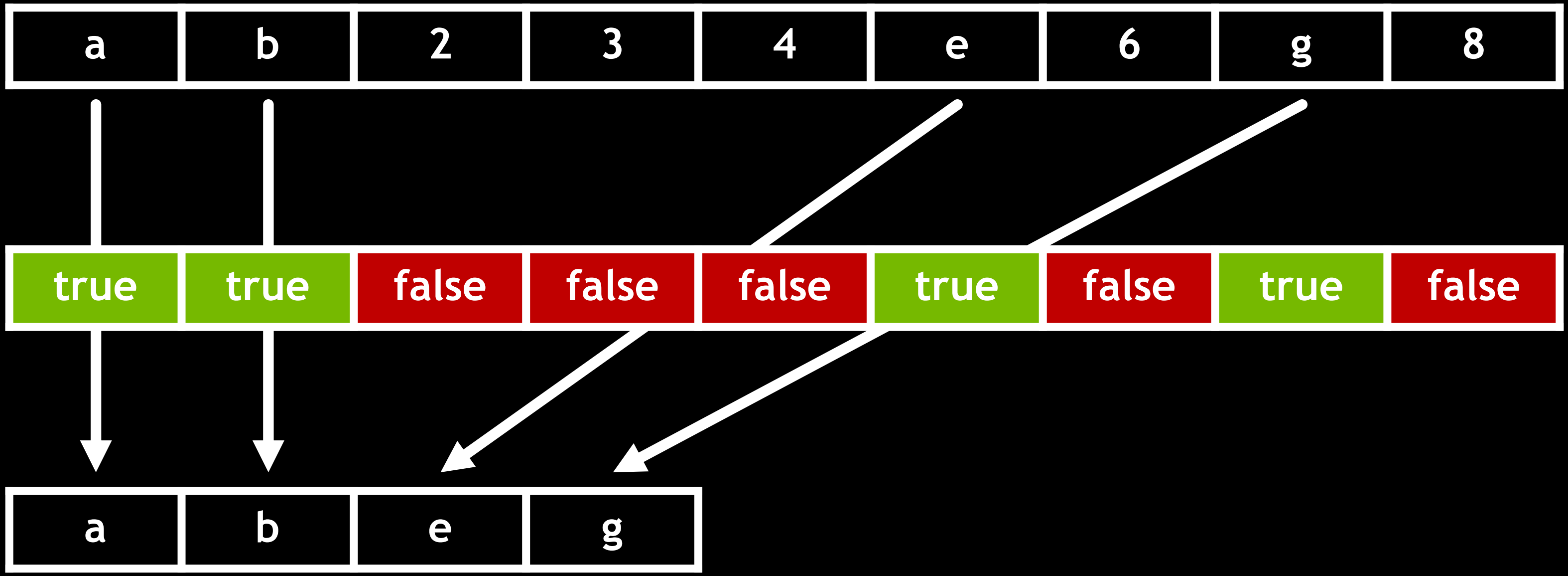
a	b	2	3	4	e	6	g	8

`is_letter`

a	b	2	3	4	e	6	g	8

`is_letter`

true	true	false	false	false	true	false	true	false



a	b	2	3	4	e	6	g	8
true	true	false	false	false	true	false	true	false
a	b	e	g					

```
auto copy_if(stdr::range auto&& in, auto out, auto op) {
  …
}
```

```
auto copy_if(stdr::range auto&& in, auto out, auto op) {
  std::vector<std::uint8_t> flags(size(in));

  …
}
```

```
auto copy_if(stdr::range auto&& in, auto out, auto op) {
  std::vector<std::uint8_t> flags(size(in));

  stdr::transform(stde::par, in, begin(flags), op);

  …
}
```

```
auto copy_if(stdr::range auto&& in, auto out, auto op) {
  std::vector<std::uint8_t> flags(size(in));

  stdr::transform(stde::par, in, begin(flags), op);

  std::vector<std::size_t> indices(size(in) + 1);

  …
}
```

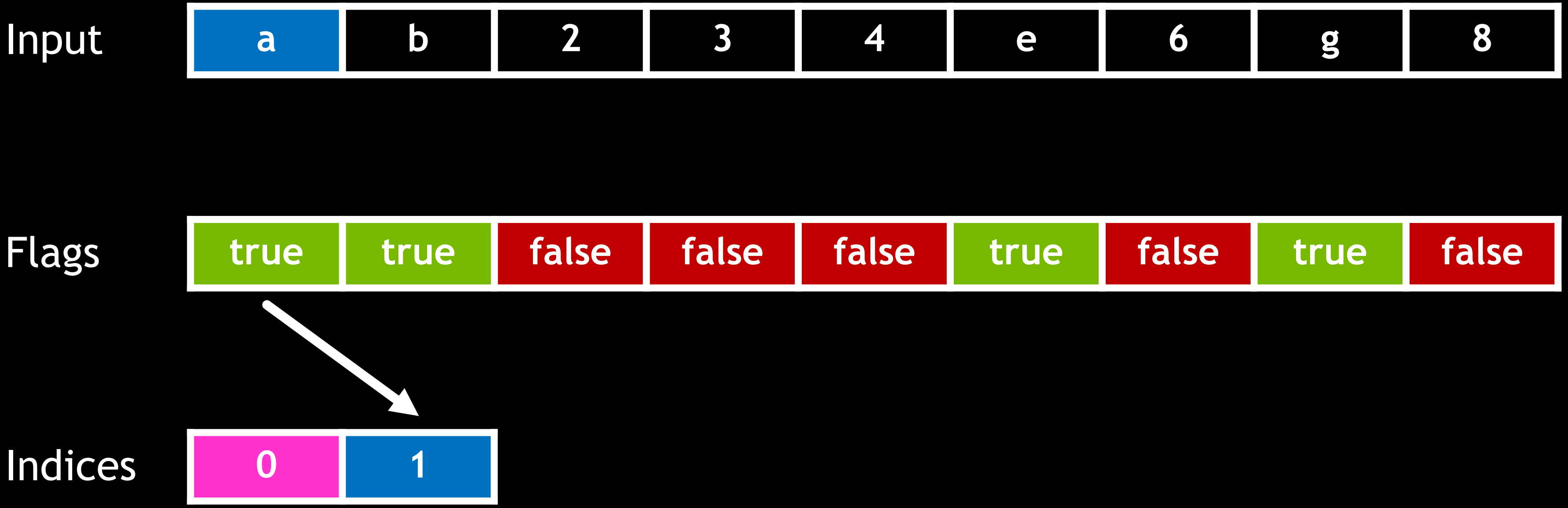
```
auto copy_if(stdr::range auto&& in, auto out, auto op) {
  std::vector<std::uint8_t> flags(size(in));

  stdr::transform(stde::par, in, begin(flags), op);

  std::vector<std::size_t> indices(size(in) + 1);

  stdr::inclusive_scan(stde::par, flags, begin(indices) + 1);

  …
}
```

Input	a	b	2	3	4	e	6	g	8

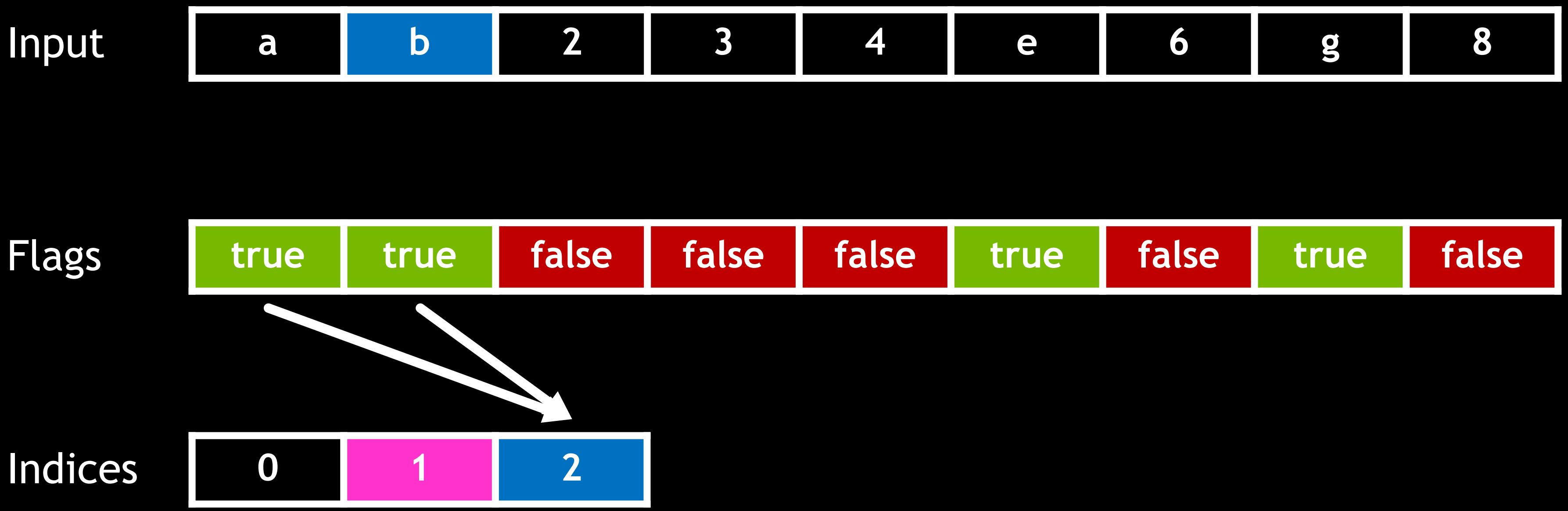
`is_letter`

Flags	true	true	false	false	false	true	false	true	false

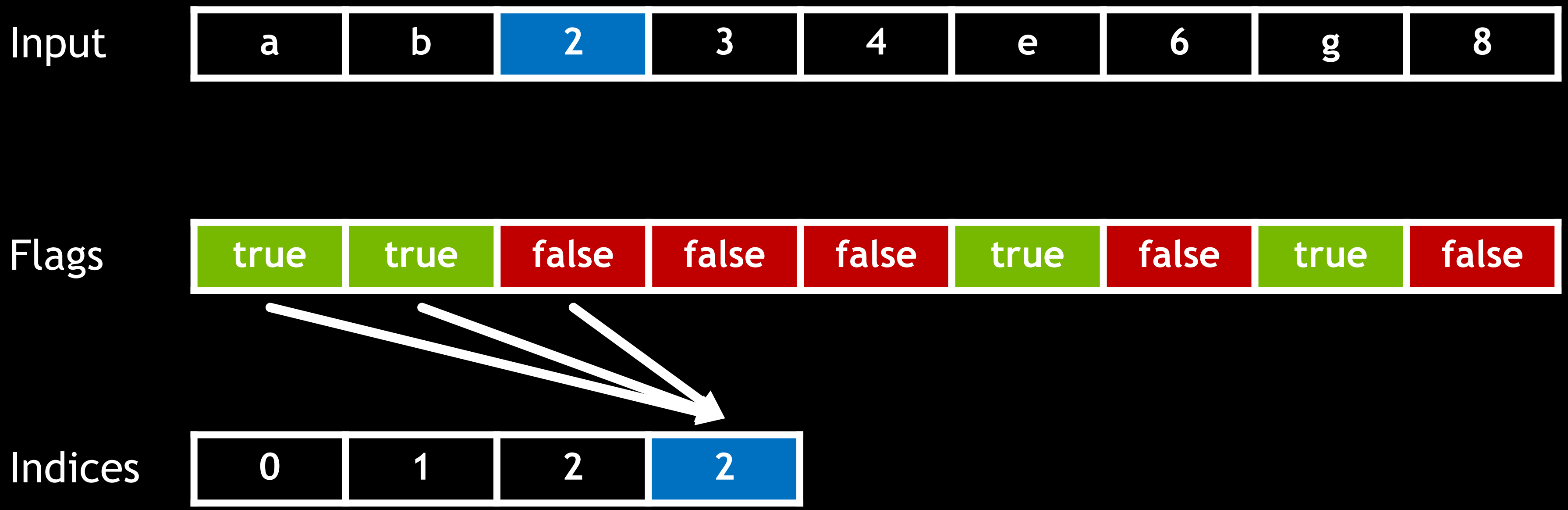
Input	a	b	2	3	4	e	6	g	8
Flags	true	true	false	false	false	true	false	true	false
Indices	0								



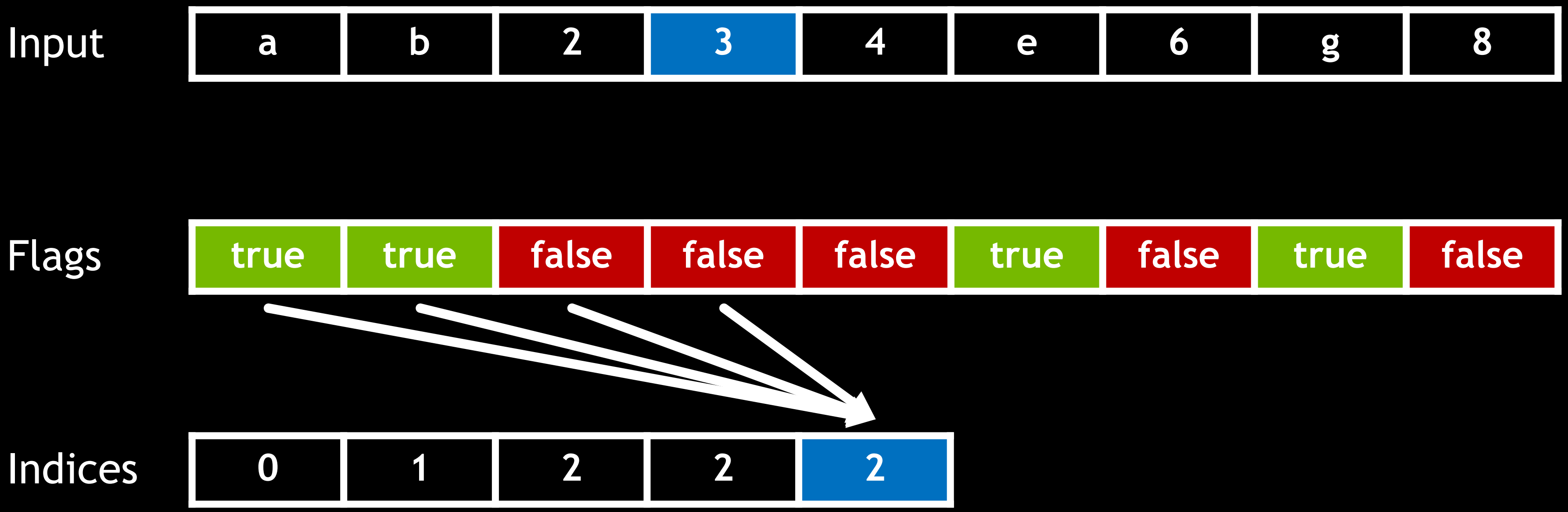
Input	a	b	2	3	4	e	6	g	8
Flags	true	true	false	false	false	true	false	true	false
Indices	0	1							



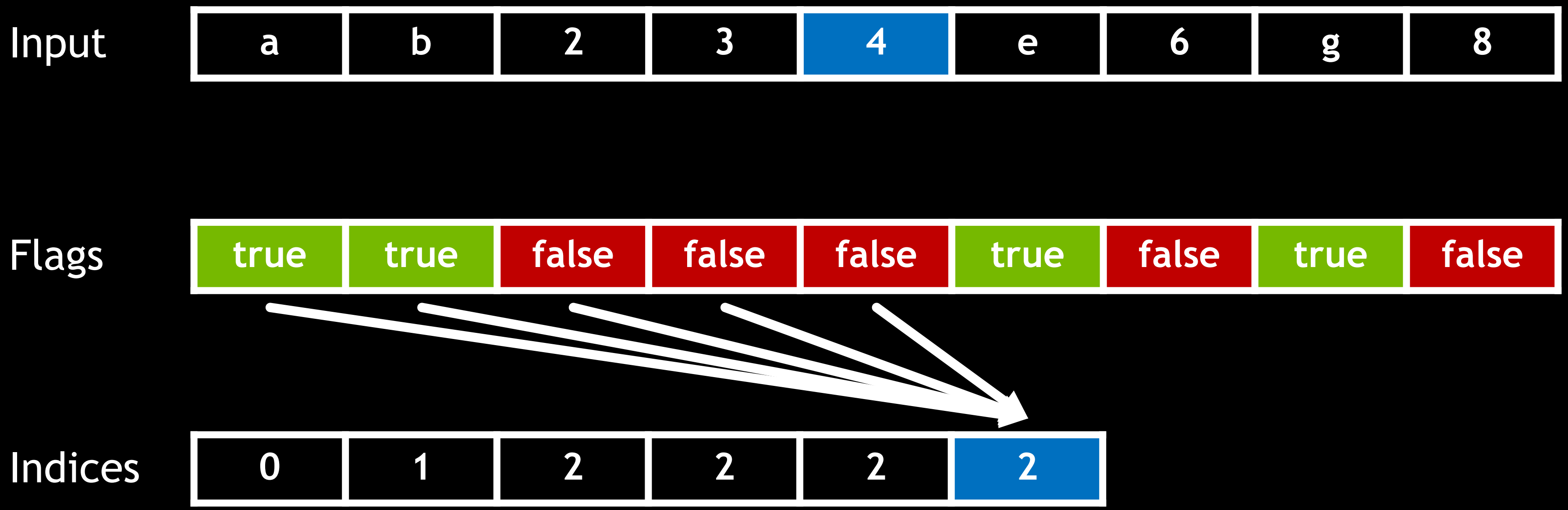
Input	a	b	2	3	4	e	6	g	8
Flags	true	true	false	false	false	true	false	true	false
Indices	0	1	2						



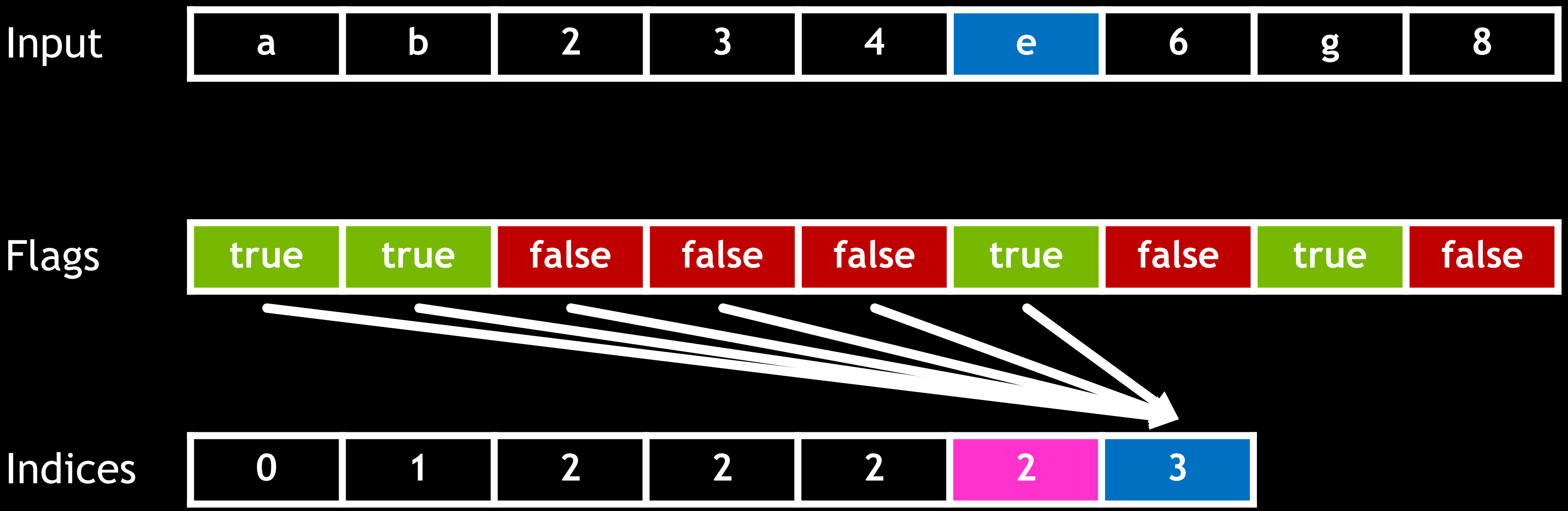
Input	a	b	2	3	4	e	6	g	8
Flags	true	true	false	false	false	true	false	true	false
Indices	0	1	2	2					



Input	a	b	2	3	4	e	6	g	8
Flags	true	true	false	false	false	true	false	true	false
Indices	0	1	2	2	2				



Input	a	b	2	3	4	e	6	g	8
Flags	true	true	false	false	false	true	false	true	false
Indices	0	1	2	2	2	2			

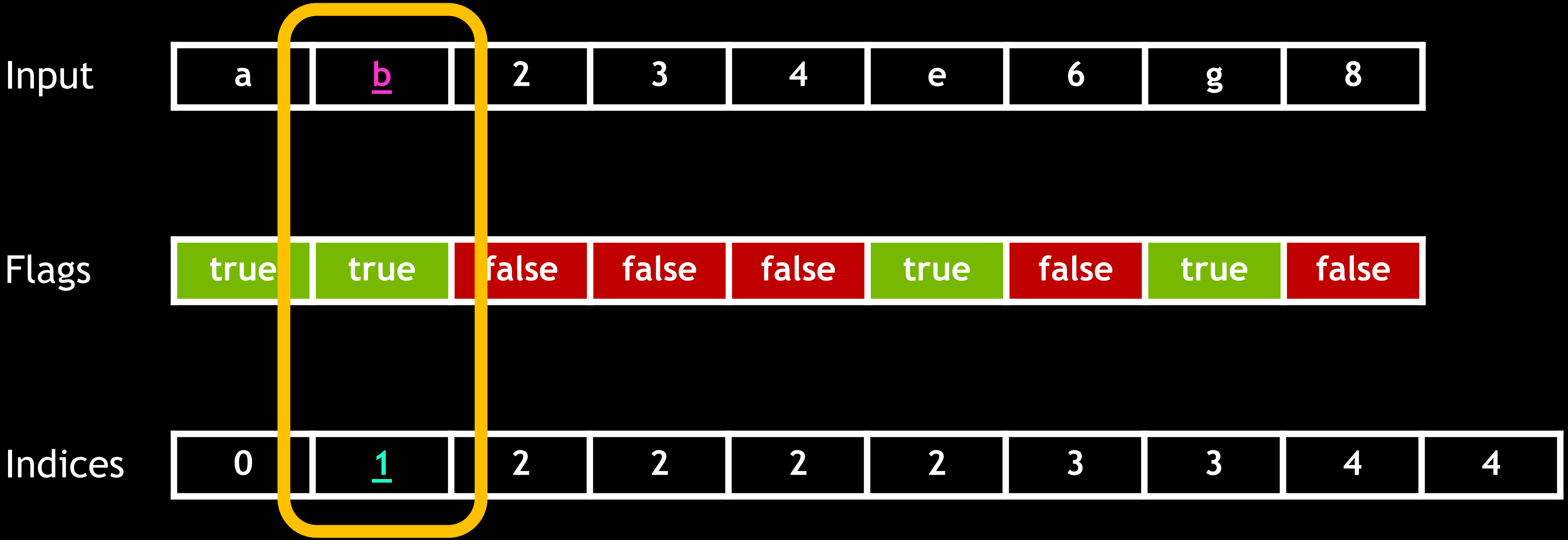


Input	a	b	2	3	4	e	6	g	8
Flags	true	true	false	false	false	true	false	true	false
Indices	0	1	2	2	2	2	3		

Input	a	b	2	3	4	e	6	g	8	
Flags	true	true	false	false	false	true	false	true	false	
Indices	0	1	2	2	2	2	3	3	4	4

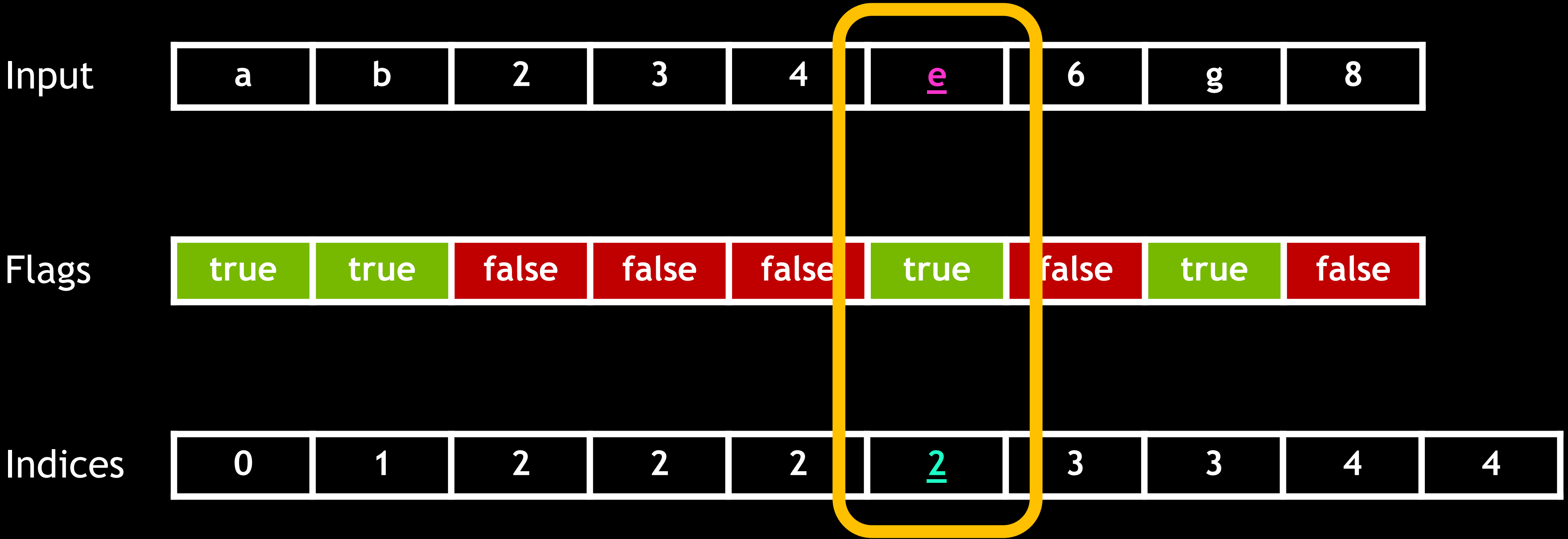
Input	a	b	2	3	4	e	6	g	8	
Flags	true	true	false	false	false	true	false	true	false	
Indices	0	1	2	2	2	2	3	3	4	4

```
out[0] = a
```



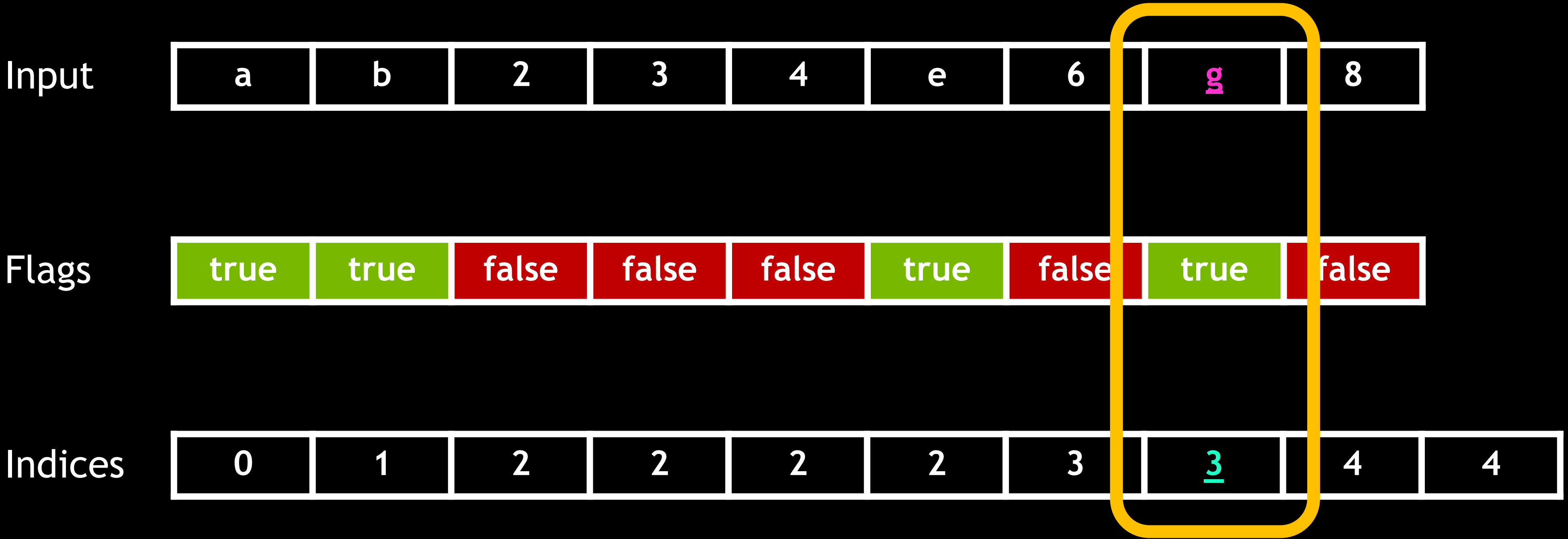
Input	a	b	2	3	4	e	6	g	8	
Flags	true	true	false	false	false	true	false	true	false	
Indices	0	1	2	2	2	2	3	3	4	4

```
out[1] = b
```



Input	a	b	2	3	4	e	6	g	8	
Flags	true	true	false	false	false	true	false	true	false	
Indices	0	1	2	2	2	2	3	3	4	4

```
out[2] = e
```



Input	a	b	2	3	4	e	6	g	8	
Flags	true	true	false	false	false	true	false	true	false	
Indices	0	1	2	2	2	2	3	3	4	4

`out[3] = g`

Input	a	b	2	3	4	e	6	g	8	
Flags	true	true	false	false	false	true	false	true	false	
Indices	0	1	2	2	2	2	3	3	4	4

```
auto copy_if(stdr::range auto&& in, auto out, auto op) {
  std::vector<std::uint8_t> flags(size(in));

  stdr::transform(stde::par, in, begin(flags), op);

  std::vector<std::size_t> indices(size(in) + 1);

  stdr::inclusive_scan(stde::par, flags, begin(indices) + 1);

  stdr::for_each(stde::par, stdv::zip(in, flags, indices),
    …);

  …
}
```

```
auto copy_if(stdr::range auto&& in, auto out, auto op) {
  std::vector<std::uint8_t> flags(size(in));

  stdr::transform(stde::par, in, begin(flags), op);

  std::vector<std::size_t> indices(size(in) + 1);

  stdr::inclusive_scan(stde::par, flags, begin(indices) + 1);

  stdr::for_each(stde::par, stdv::zip(in, flags, indices),
    [&] (auto z) { auto [e, flag, index] = z;
      if (flag) out[index] = e;
    });

  …
}
```

```
auto copy_if(stdr::range auto&& in, auto out, auto op) {
  std::vector<std::uint8_t> flags(size(in));

  stdr::transform(stde::par, in, begin(flags), op);

  std::vector<std::size_t> indices(size(in) + 1);

  stdr::inclusive_scan(stde::par, flags, begin(indices) + 1);

  stdr::for_each(stde::par, stdv::zip(in, flags, indices),
    [&] (auto z) { auto [e, flag, index] = z;
      if (flag) out[index] = e;
    });

  return stdr::subrange(out, next(out, indices.back()));
}
```

```
auto copy_if(stdr::range auto&& in, auto out, auto op) {
  std::vector<std::uint8_t> flags(size(in));

  stdr::transform(stde::par, in, begin(flags), op);

  std::vector<std::size_t> indices(size(in) + 1);

  stdr::inclusive_scan(stde::par, flags, begin(indices) + 1);

  stdr::for_each(stde::par, stdv::zip(in, flags, indices),
    [&] (auto z) { auto [e, flag, index] = z;
      if (flag) out[index] = e;
    });

  return stdr::subrange(out, next(out, indices.back()));
}
```

```
auto copy_if(stdr::range auto&& in, auto out, auto op) {
  std::vector<std::uint8_t> flags(size(in));

  stdr::transform(stde::par, in, begin(flags), op);

  std::vector<std                             1);

  stdr::inclusive                             (indices) + 1);

  stdr::for_each(                             gs, indices),
    [&] (auto z)
      if (flag) out[index] = e;
    });

  return stdr::subrange(out, next(out, indices.back()));
}
```

Analysis

O(input) storage (2 * input)

3 global synchronizations

```
auto copy_if(stdr::range auto&& in, auto out, auto op, std::size_t num_tiles) {
  …
}
```

```
auto copy_if(stdr::range auto&& in, auto out, auto op, std::size_t num_tiles) {
  scan_tile_state<std::size_t> sts(num_tiles);
  std::atomic<std::size_t> tile_counter(0);

  …
}
```

```
auto copy_if(stdr::range auto&& in, auto out, auto op, std::size_t num_tiles) {
  scan_tile_state<std::size_t> sts(num_tiles);
  std::atomic<std::size_t> tile_counter(0);

  stdr::for_each(stde::par, stdv::iota(0, num_tiles), [&] (std::size_t) {
      …
    });

  …
}
```

```
auto copy_if(stdr::range auto&& in, auto out, auto op, std::size_t num_tiles) {
 scan_tile_state<std::size_t> sts(num_tiles);
 std::atomic<std::size_t> tile_counter(0);

 stdr::for_each(stde::par, stdv::iota(0, num_tiles), [&] (std::size_t) {
   auto tile = tile_counter.fetch_add(1, std::memory_order_relaxed);
   auto sub_in = range_for_tile(in, tile, num_tiles);

   …
  });

 …
}
```

```
auto copy_if(stdr::range auto&& in, auto out, auto op, std::size_t num_tiles) {
 scan_tile_state<std::size_t> sts(num_tiles);
 std::atomic<std::size_t> tile_counter(0);

 stdr::for_each(stde::par, stdv::iota(0, num_tiles), [&] (std::size_t) {
   auto tile = tile_counter.fetch_add(1, std::memory_order_relaxed);
   auto sub_in = range_for_tile(in, tile, num_tiles);

   std::vector<std::uint8_t> flags(size(sub_in));
   stdr::transform(sub_in, begin(flags), op);

   …
  });

 …
}
```

```
auto copy_if(stdr::range auto&& in, auto out, auto op, std::size_t num_tiles) {
  scan_tile_state<std::size_t> sts(num_tiles);
  std::atomic<std::size_t> tile_counter(0);

  stdr::for_each(stde::par, stdv::iota(0, num_tiles), [&] (std::size_t) {
      auto tile = tile_counter.fetch_add(1, std::memory_order_relaxed);
      auto sub_in = range_for_tile(in, tile, num_tiles);

      std::vector<std::uint8_t> flags(size(sub_in));
      stdr::transform(sub_in, begin(flags), op);

      std::vector<std::size_t> indices(size(sub_in) + 1);

      …
    });

  …
}
```

```
auto copy_if(stdr::range auto&& in, auto out, auto op, std::size_t num_tiles) {
  scan_tile_state<std::size_t> sts(num_tiles);
  std::atomic<std::size_t> tile_counter(0);

  stdr::for_each(stde::par, stdv::iota(0, num_tiles), [&] (std::size_t) {
      auto tile = tile_counter.fetch_add(1, std::memory_order_relaxed);
      auto sub_in = range_for_tile(in, tile, num_tiles);

      std::vector<std::uint8_t> flags(size(sub_in));
      stdr::transform(sub_in, begin(flags), op);

      std::vector<std::size_t> indices(size(sub_in) + 1);

      sts.set_local_prefix(tile, *--stdr::inclusive_scan(flags, begin(indices) + 1));
      if (tile != 0) {
        auto pred = sts.wait_for_predecessor_prefix(tile);
        stdr::for_each(indices, [&] (auto& e) { e = pred + e; });
      }

      …
    });

  …
}
```

```
auto copy_if(stdr::range auto&& in, auto out, auto op, std::size_t num_tiles) {
 scan_tile_state<std::size_t> sts(num_tiles);
 std::atomic<std::size_t> tile_counter(0);

 stdr::for_each(stde::par, stdv::iota(0, num_tiles), [&] (std::size_t) {
   auto tile = tile_counter.fetch_add(1, std::memory_order_relaxed);
   auto sub_in = range_for_tile(in, tile, num_tiles);

   std::vector<std::uint8_t> flags(size(sub_in));
   stdr::transform(sub_in, begin(flags), op);

   std::vector<std::size_t> indices(size(sub_in) + 1);

   sts.set_local_prefix(tile, *--stdr::inclusive_scan(flags, begin(indices) + 1));
   if (tile != 0) {
    auto pred = sts.wait_for_predecessor_prefix(tile);
    stdr::for_each(indices, [&] (auto& e) { e = pred + e; });
   }

   stdr::for_each(stdv::zip(sub_in, flags, indices),
    [&] (auto z) { auto [e, flag, index] = z; if (flag) out[index] = e; });
  });

 …
}
```

```
auto copy_if(stdr::range auto&& in, auto out, auto op, std::size_t num_tiles) {
 scan_tile_state<std::size_t> sts(num_tiles);
 std::atomic<std::size_t> tile_counter(0);

 stdr::for_each(stde::par, stdv::iota(0, num_tiles), [&] (std::size_t) {
   auto tile = tile_counter.fetch_add(1, std::memory_order_relaxed);
   auto sub_in = range_for_tile(in, tile, num_tiles);

   std::vector<std::uint8_t> flags(size(sub_in));
   stdr::transform(sub_in, begin(flags), op);

   std::vector<std::size_t> indices(size(sub_in) + 1);

   sts.set_local_prefix(tile, *--stdr::inclusive_scan(flags, begin(indices) + 1));
   if (tile != 0) {
    auto pred = sts.wait_for_predecessor_prefix(tile);
    stdr::for_each(indices, [&] (auto& e) { e = pred + e; });
   }

   stdr::for_each(stdv::zip(sub_in, flags, indices),
    [&] (auto z) { auto [e, flag, index] = z; if (flag) out[index] = e; });
  });

 return stdr::subrange(out, next(out, sts.prefixes.back().complete));
}
```

```cpp
auto copy_if(stdr::range auto&& in, auto out, auto op, std::size_t num_tiles) {
  scan_tile_state<std::size_t> sts(num_tiles);
  std::atomic<std::size_t> tile_counter(0);

  stdr::for_each(stde::par, stdv::iota(0, num_tiles), [&] (std::size_t) {
      auto tile = tile_counter.fetch_add(1, std::memory_order_relaxed);
      auto sub_in = range_for_tile(in, tile, num_tiles);

      std::vector<std::uint8_t> flags(size(sub_in));
      stdr::transform(sub_in, begin(flags), op);

      std::vector<std::size_t> indices(size(sub_in) + 1);

      sts.set_local_prefix(tile, *--stdr::inclusive_scan(flags, begin(indices) + 1));
      if (tile != 0) {
        auto pred = sts.wait_for_predecessor_prefix(tile);
        stdr::for_each(indices, [&] (auto& e) { e = pred + e; });
      }

      stdr::for_each(stdv::zip(sub_in, flags, indices),
        [&] (auto z) { auto [e, flag, index] = z; if (flag) out[index] = e; });
    });

  return stdr::subrange(out, next(out, sts.prefixes.back().complete));
}
```

```
auto copy_if(stdr::range auto&& in, auto out, auto op, std::size_t num_tiles) {
  scan_tile_state<std::size_t> sts(num_tiles);
  std::atomic<std::size_t> tile_counter(0);

  stdr::for_each(stde::p                                      d::size_t) {
      auto tile = tile_c                                      relaxed);
      auto sub_in = rang

      std::vector<std::u
      stdr::transform(su

      std::vector<std::size_t> indices(size(sub_in) + 1);

      sts.set_local_prefix(tile, *--stdr::inclusive_scan(flags, begin(indices) + 1));
      if (tile != 0) {
        auto pred = sts.wait_for_predecessor_prefix(tile);
        stdr::for_each(indices, [&] (auto& e) { e = pred + e; });
      }

      stdr::for_each(stdv::zip(sub_in, flags, indices),
        [&] (auto z) { auto [e, flag, index] = z; if (flag) out[index] = e; });
    });

  return stdr::subrange(out, next(out, sts.prefixes.back().complete));
}
```

Analysis

O(tiles) storage

1 global synchronization

```
auto copy_if(stdr::range auto&& in, auto out, auto op, std::size_t num_tiles) {
  scan_tile_state<std::size_t> sts(num_tiles);
  std::atomic<std::size_t> tile_counter(0);

  stdr::for_each(stde::p                                   d::size_t) {
      auto tile = tile_c                                   _relaxed);
      auto sub_in = rang

      std::vector<std::u
      stdr::transform(su

      std::vector<std::size_t> indices(size(sub_in) + 1);

      sts.set_local_prefix(tile, *--stdr::inclusive_scan(flags, begin(indices) + 1));
      if (tile != 0) {
        a
        s
      }

      std
        [                                                                  });
    });

  return stdr::subrange(out, next(out, sts.prefixes.back().complete));
}
```

Analysis

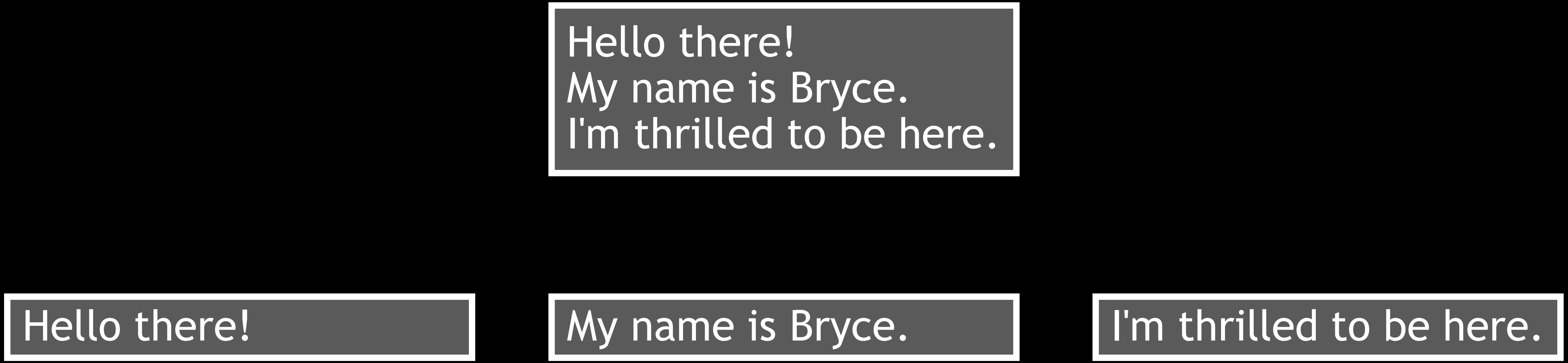
O(tiles) storage

1 global synchronization

Performance

11x faster than three pass implementation

NVC++ 24.3, 2x 32 core EPYC 7513, 4GB 32 bit int input, 1024 tiles



Hello there!
My name is Bryce.
I'm thrilled to be here.

Hello there!

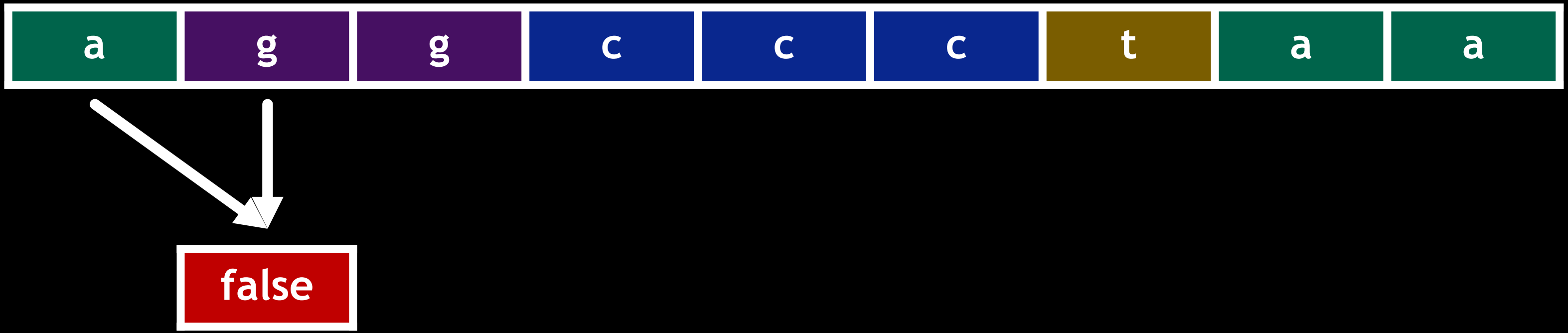
My name is Bryce.

I'm thrilled to be here.

```
[] (auto l, auto r) { return !(l == '\n'); };
```

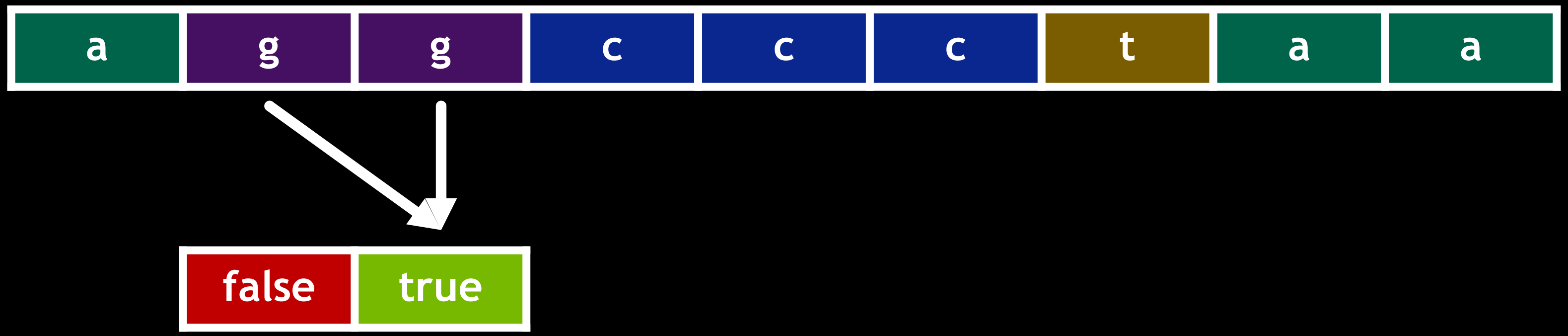
a	g	g	c	c	c	t	a	a

`equal_to`



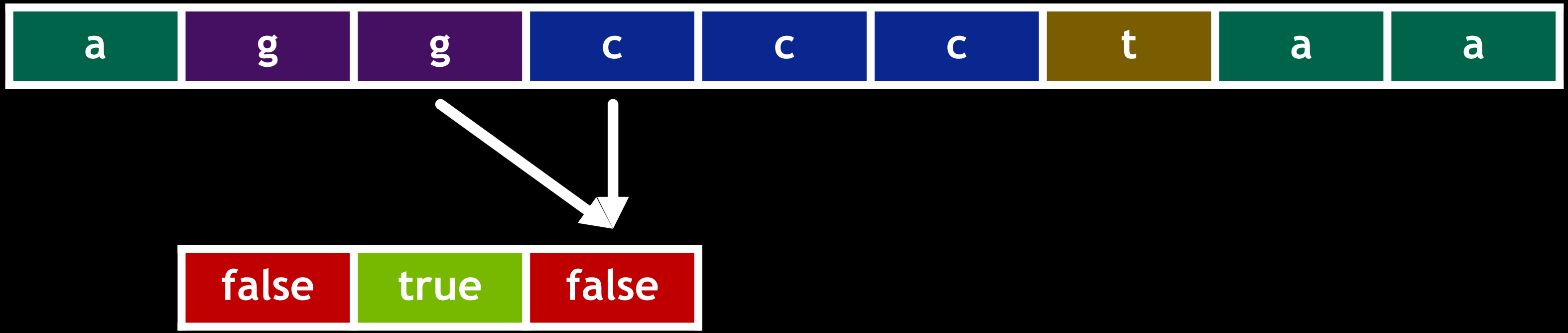
a	g	g	c	c	c	t	a	a

false



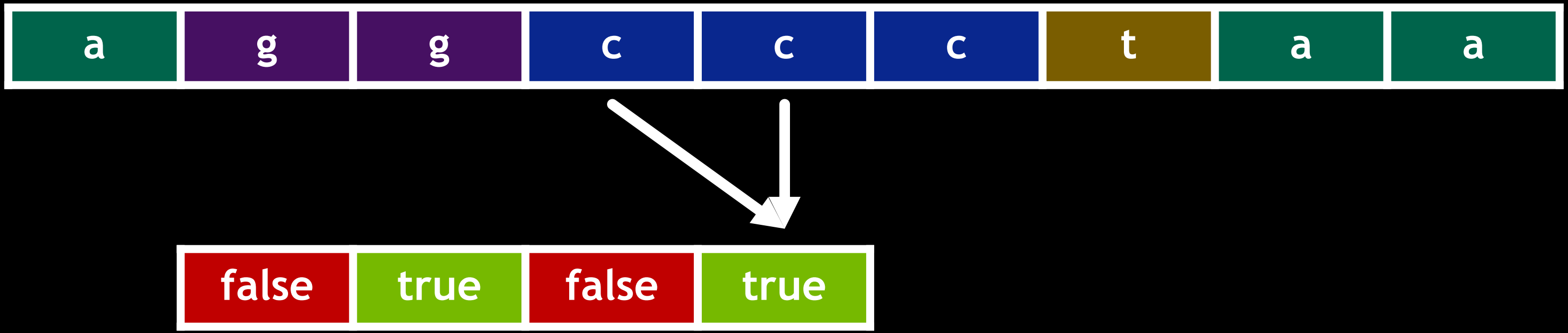
a	g	g	c	c	c	t	a	a

false	true



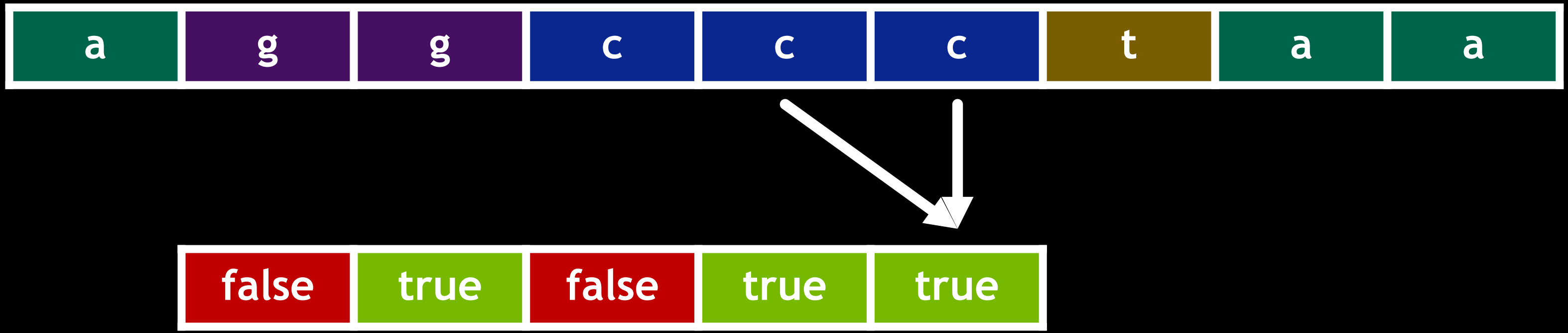
a	g	g	c	c	c	t	a	a

false	true	false



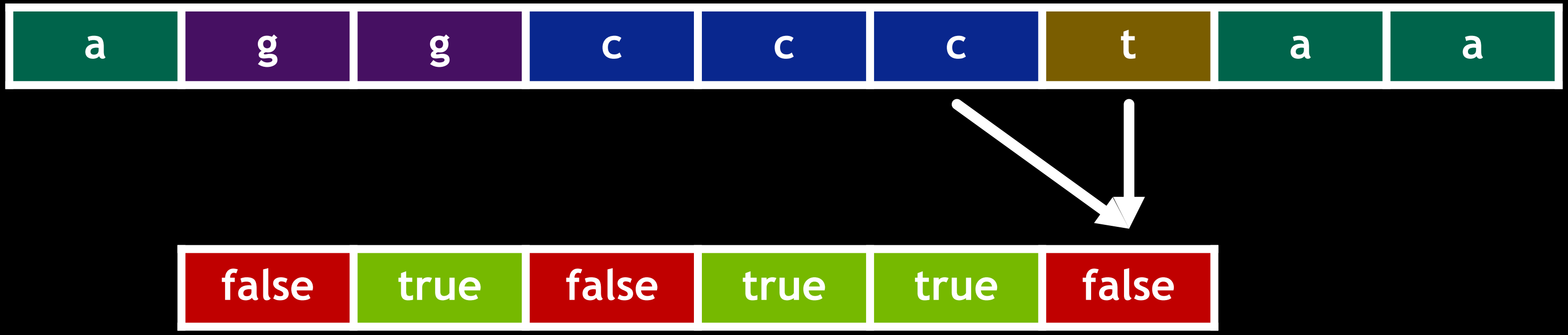
a	g	g	c	c	c	t	a	a

false	true	false	true



a	g	g	c	c	c	t	a	a

false	true	false	true	true



a	g	g	c	c	c	t	a	a

false	true	false	true	true	false

a	g	g	c	c	c	t	a	a
	false	true	false	true	true	false	false	true

- What index is this chunk?

- What index is this chunk?
- Where does this chunk start?

- What index is this chunk?
- Where does this chunk start?
- Where does this chunk end?

```
struct interval {
  …
};
```

```
struct interval {
  bool flag = true;
  …
};
```

```
struct interval {
  bool flag = true;
  std::size_t index = 0; // Plus scan of inverse flags.
  …
};
```

```
struct interval {
  bool flag = true;
  std::size_t index = 0;  // Plus scan of inverse flags.
  std::size_t count = 0;  // Counts elements with this index.
  …
};
```

```
struct interval {
  bool flag = true;
  std::size_t index = 0; // Plus scan of inverse flags.
  std::size_t count = 0; // Counts elements with this index.
  std::size_t end = 0;   // Counts all elements.
};
```

```
struct interval {
  bool flag = true;
  std::size_t index = 0; // Plus scan of inverse flags.
  std::size_t count = 0; // Counts elements with this index.
  std::size_t end = 0;   // Counts all elements.
};

interval operator+(interval l, interval r) {
  return {…};
}
```

```
struct interval {
  bool flag = true;
  std::size_t index = 0; // Plus scan of inverse flags.
  std::size_t count = 0; // Counts elements with this index.
  std::size_t end = 0;   // Counts all elements.
};

interval operator+(interval l, interval r) {
  return {r.flag,
          …};
}
```

```
struct interval {
  bool flag = true;
  std::size_t index = 0; // Plus scan of inverse flags.
  std::size_t count = 0; // Counts elements with this index.
  std::size_t end = 0;   // Counts all elements.
};

interval operator+(interval l, interval r) {
  return {r.flag,
          l.index + r.index,
          …};
}
```

```
struct interval {
  bool flag = true;
  std::size_t index = 0; // Plus scan of inverse flags.
  std::size_t count = 0; // Counts elements with this index.
  std::size_t end = 0;   // Counts all elements.
};

interval operator+(interval l, interval r) {
  return {r.flag,
          l.index + r.index,
          r.index ? r.count : l.count + r.count,
          …};
}
```

```
struct interval {
  bool flag = true;
  std::size_t index = 0; // Plus scan of inverse flags.
  std::size_t count = 0; // Counts elements with this index.
  std::size_t end = 0;   // Counts all elements.
};

interval operator+(interval l, interval r) {
  return {r.flag,
          l.index + r.index,
          r.index ? r.count : l.count + r.count,
          l.end + r.end};
}
```

```
struct interval {
  bool flag = true;
  std::size_t index = 0;  // Plus scan of inverse flags.
  std::size_t count = 0;  // Counts elements with this index.
  std::size_t end = 0;    // Counts all elements.
};

interval operator+(interval l, interval r) {
  return {r.flag,
          l.index + r.index,
          r.index ? r.count : l.count + r.count,
          l.end + r.end};
}
```

```
auto chunk_by(stdr::range auto&& in, stdr::range auto&& out, auto op) {
  …
}
```

```
auto chunk_by(stdr::range auto&& in, stdr::range auto&& out, auto op) {
  std::vector<interval> intervals(size(in) + 1);

  …
}
```

```
auto chunk_by(stdr::range auto&& in, stdr::range auto&& out, auto op) {
  std::vector<interval> intervals(size(in) + 1);


  stdr::transform(stde::par, in | stdv::adjacent<2>, begin(intervals) + 1,

    …

     );


  …
}
```

a	g	g	c	c	c	t	a	a

a	g	g	c	c	c	t	a
g	g	c	c	c	t	a	a

a	g	g	c	c	c	t	a	a

a	g	g	c	c	c	t	a
g	g	c	c	c	t	a	a

Φ^0	a	g	g	c	c	c	t	a	a
a	g	g	c	c	c	t	a	a	Φ^1

```
auto chunk_by(stdr::range auto&& in, stdr::range auto&& out, auto op) {
  std::vector<interval> intervals(size(in) + 1);

  intervals[0] = …;
  stdr::transform(stde::par, in | stdv::adjacent<2>, begin(intervals) + 1,
    [&] (auto lr) { auto [l, r] = lr;
      bool b = op(l, r);
      return interval{b, !b, 1, 1};
    });
  intervals.back() = …;

  …
}
```

```
auto chunk_by(stdr::range auto&& in, stdr::range auto&& out, auto op) {
  std::vector<interval> intervals(size(in) + 1);

  intervals[0] = …;
  stdr::transform(stde::par, in | stdv::adjacent<2>, begin(intervals) + 1,
    [&] (auto lr) { auto [l, r] = lr;
      bool b = op(l, r);
      return interval{b, !b, 1, 1};
    });
  intervals.back() = …;

  …
}
```

a	g	g	c	c	c	t	a	a

Φ^0	a	g	g	c	c	c	t	a	a
a	g	g	c	c	c	t	a	a	Φ^1

Intervals Before Scan

Flags	false	true	false	true	true	false	false	true
Index	1	0	1	0	0	1	1	0
Count	1	1	1	1	1	1	1	1
End	1	1	1	1	1	1	1	1

```
auto chunk_by(stdr::range auto&& in, stdr::range auto&& out, auto op) {
  std::vector<interval> intervals(size(in) + 1);

  intervals[0] = interval{true, 0, 1, 1};
  stdr::transform(stde::par, in | stdv::adjacent<2>, begin(intervals) + 1,
    [&] (auto lr) { auto [l, r] = lr;
      bool b = op(l, r);
      return interval{b, !b, 1, 1};
    });
  intervals.back() = …;

  …
}
```

```
auto chunk_by(stdr::range auto&& in, stdr::range auto&& out, auto op) {
  std::vector<interval> intervals(size(in) + 1);

  intervals[0] = interval{true, 0, 1, 1};
  stdr::transform(stde::par, in | stdv::adjacent<2>, begin(intervals) + 1,
    [&] (auto lr) { auto [l, r] = lr;
      bool b = op(l, r);
      return interval{b, !b, 1, 1};
    });
  intervals.back() = interval{false, 1, 1, 1};

  …
}
```

a	g	g	c	c	c	t	a	a

Φ^0	a	g	g	c	c	c	t	a	a
a	g	g	c	c	c	t	a	a	Φ^1

Intervals Before Scan

Flags	true	false	true	false	true	true	false	false	true	false
Index	0	1	0	1	0	0	1	1	0	1
Count	1	1	1	1	1	1	1	1	1	1
End	1	1	1	1	1	1	1	1	1	1

```
auto chunk_by(stdr::range auto&& in, stdr::range auto&& out, auto op) {
  std::vector<interval> intervals(size(in) + 1);

  intervals[0] = interval{true, 0, 1, 1};
  stdr::transform(stde::par, in | stdv::adjacent<2>, begin(intervals) + 1,
    [&] (auto lr) { auto [l, r] = lr;
      bool b = op(l, r);
      return interval{b, !b, 1, 1};
    });
  intervals.back() = interval{false, 1, 1, 1};

  stdr::inclusive_scan(stde::par, intervals, begin(intervals));

  …
};
```

a	g	g	c	c	c	t	a	a

Before Scan

Flags	true	false	true	false	true	true	false	false	true	false
Index	0	1	0	1	0	0	1	1	0	1

After Scan

Flags	true	false	true	false	true	true	false	false	true	false

a	g	g	c	c	c	t	a	a

Before Scan

Flags	true	false	true	false	true	true	false	false	true	false
Index	0	1	0	1	0	0	1	1	0	1

After Scan

Flags	true	false	true	false	true	true	false	false	true	false
Index	0	1	1	2	2	2	3	4	4	5

a	g	g	c	c	c	t	a	a

Before Scan

Flags	true	false	true	false	true	true	false	false	true	false
Index	0	1	0	1	0	0	1	1	0	1
Count	1	1	1	1	1	1	1	1	1	1

After Scan

Flags	true	false	true	false	true	true	false	false	true	false
Index	0	1	1	2	2	2	3	4	4	5

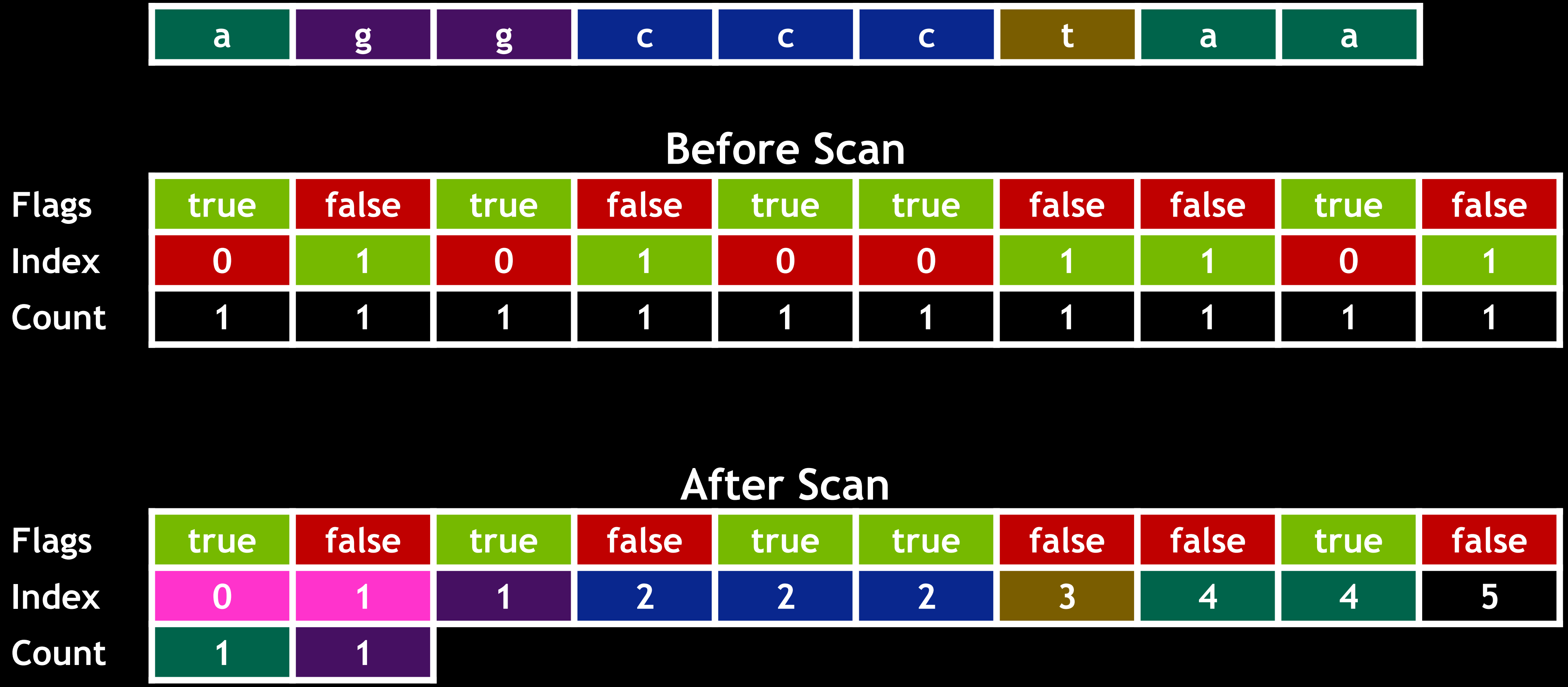
a	g	g	c	c	c	t	a	a

Before Scan

Flags	true	false	true	false	true	true	false	false	true	false
Index	0	1	0	1	0	0	1	1	0	1
Count	1	1	1	1	1	1	1	1	1	1

After Scan

Flags	true	false	true	false	true	true	false	false	true	false
Index	0	1	1	2	2	2	3	4	4	5
Count	1									



a	g	g	c	c	c	t	a	a

Before Scan

Flags	true	false	true	false	true	true	false	false	true	false
Index	0	1	0	1	0	0	1	1	0	1
Count	1	1	1	1	1	1	1	1	1	1

After Scan

Flags	true	false	true	false	true	true	false	false	true	false
Index	0	1	1	2	2	2	3	4	4	5
Count	1	1								

a	g	g	c	c	c	t	a	a

Before Scan

Flags	true	false	true	false	true	true	false	false	true	false
Index	0	1	0	1	0	0	1	1	0	1
Count	1	1	1	1	1	1	1	1	1	1

After Scan

Flags	true	false	true	false	true	true	false	false	true	false
Index	0	1	1	2	2	2	3	4	4	5
Count	1	1	2							

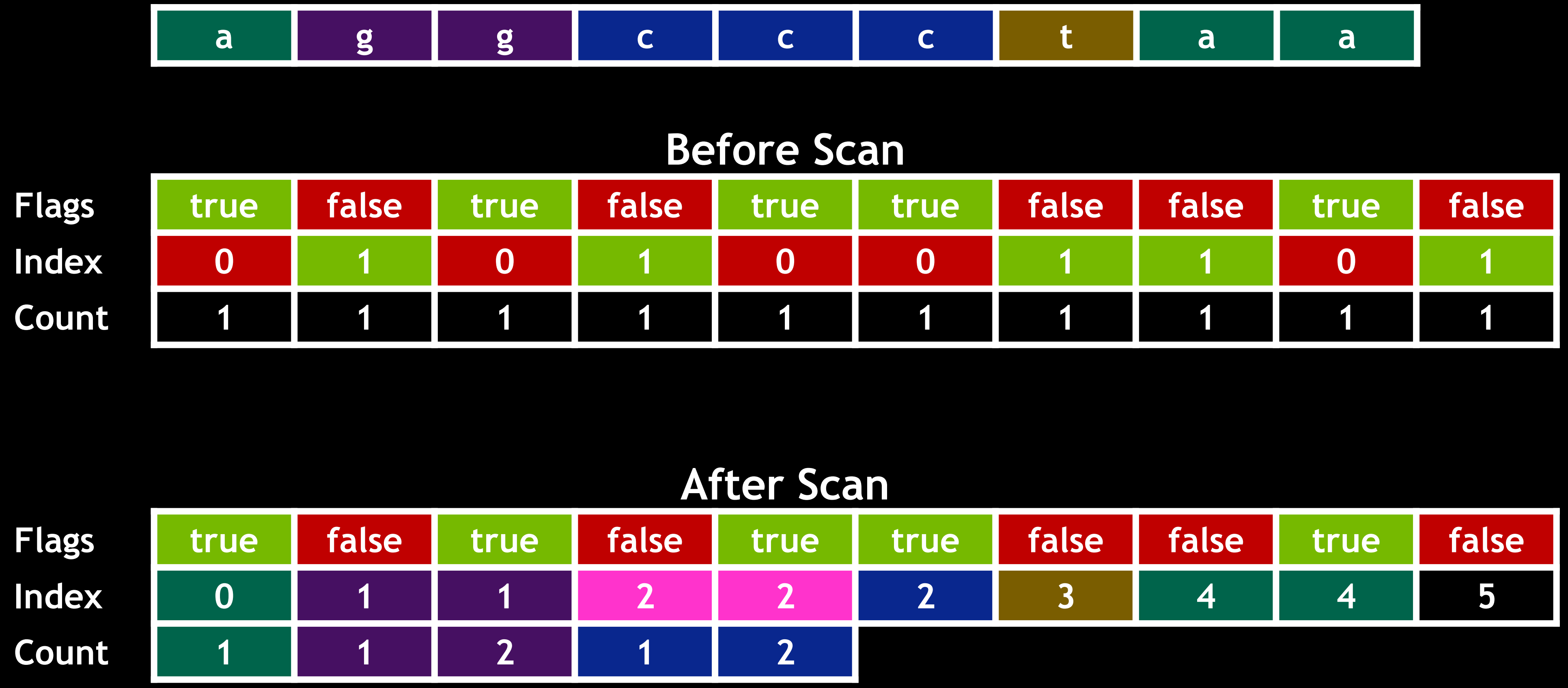
a	g	g	c	c	c	t	a	a

Before Scan

Flags	true	false	true	false	true	true	false	false	true	false
Index	0	1	0	1	0	0	1	1	0	1
Count	1	1	1	1	1	1	1	1	1	1

After Scan

Flags	true	false	true	false	true	true	false	false	true	false
Index	0	1	1	2	2	2	3	4	4	5
Count	1	1	2	1						



a	g	g	c	c	c	t	a	a

Before Scan

Flags	true	false	true	false	true	true	false	false	true	false
Index	0	1	0	1	0	0	1	1	0	1
Count	1	1	1	1	1	1	1	1	1	1

After Scan

Flags	true	false	true	false	true	true	false	false	true	false
Index	0	1	1	2	2	2	3	4	4	5
Count	1	1	2	1	2					

a	g	g	c	c	c	t	a	a

Before Scan

Flags	true	false	true	false	true	true	false	false	true	false
Index	0	1	0	1	0	0	1	1	0	1
Count	1	1	1	1	1	1	1	1	1	1

After Scan

Flags	true	false	true	false	true	true	false	false	true	false
Index	0	1	1	2	2	2	3	4	4	5
Count	1	1	2	1	2	3				

a	g	g	c	c	c	t	a	a

Before Scan

Flags	true	false	true	false	true	true	false	false	true	false
Index	0	1	0	1	0	0	1	1	0	1
Count	1	1	1	1	1	1	1	1	1	1

After Scan

Flags	true	false	true	false	true	true	false	false	true	false
Index	0	1	1	2	2	2	3	4	4	5
Count	1	1	2	1	2	3	1	1	2	1

a	g	g	c	c	c	t	a	a

Before Scan

Flags	true	false	true	false	true	true	false	false	true	false
Index	0	1	0	1	0	0	1	1	0	1
Count	1	1	1	1	1	1	1	1	1	1
End	1	1	1	1	1	1	1	1	1	1

After Scan

Flags	true	false	true	false	true	true	false	false	true	false
Index	0	1	1	2	2	2	3	4	4	5
Count	1	1	2	1	2	3	1	1	2	1

a	g	g	c	c	c	t	a	a

Before Scan

Flags	true	false	true	false	true	true	false	false	true	false
Index	0	1	0	1	0	0	1	1	0	1
Count	1	1	1	1	1	1	1	1	1	1
End	1	1	1	1	1	1	1	1	1	1

After Scan

Flags	true	false	true	false	true	true	false	false	true	false
Index	0	1	1	2	2	2	3	4	4	5
Count	1	1	2	1	2	3	1	1	2	1
End	1	2	3	4	5	6	7	8	9	10

```
auto chunk_by(stdr::range auto&& in, stdr::range auto&& out, auto op) {
  std::vector<interval> intervals(size(in) + 1);

  intervals[0] = interval{true, 0, 1, 1};
  stdr::transform(stde::par, in | stdv::adjacent<2>, begin(intervals) + 1,
    [&] (auto lr) { auto [l, r] = lr;
      bool b = op(l, r);
      return interval{b, !b, 1, 1};
    });
  intervals.back() = interval{false, 1, 1, 1};

  stdr::inclusive_scan(stde::par, intervals, begin(intervals));

  stdr::for_each(stde::par, intervals | stdv::adjacent<2>,
    [&] (auto lr) { auto [l, r] = lr;
      …
    });

  …
}
```

```
auto chunk_by(stdr::range auto&& in, stdr::range auto&& out, auto op) {
  std::vector<interval> intervals(size(in) + 1);

  intervals[0] = interval{true, 0, 1, 1};
  stdr::transform(stde::par, in | stdv::adjacent<2>, begin(intervals) + 1,
    [&] (auto lr) { auto [l, r] = lr;
      bool b = op(l, r);
      return interval{b, !b, 1, 1};
    });
  intervals.back() = interval{false, 1, 1, 1};

  stdr::inclusive_scan(stde::par, intervals, begin(intervals));

  stdr::for_each(stde::par, intervals | stdv::adjacent<2>,
    [&] (auto lr) { auto [l, r] = lr;
      if (!r.flag)
        …
    });

  …
}
```

a	g	g	c	c	c	t	a	a

After Scan

Flags	true	false	true	false	true	true	false	false	true	false
Index	0	1	1	2	2	2	3	4	4	5
Count	1	1	2	1	2	3	1	1	2	1
End	1	2	3	4	5	6	7	8	9	10

Write Pass

Flags		false		false			false	false		false
Index	0		1			2	3		4	

```
auto chunk_by(stdr::range auto&& in, stdr::range auto&& out, auto op) {
  std::vector<interval> intervals(size(in) + 1);

  intervals[0] = interval{true, 0, 1, 1};
  stdr::transform(stde::par, in | stdv::adjacent<2>, begin(intervals) + 1,
    [&] (auto lr) { auto [l, r] = lr;
      bool b = op(l, r);
      return interval{b, !b, 1, 1};
    });
  intervals.back() = interval{false, 1, 1, 1};

  stdr::inclusive_scan(stde::par, intervals, begin(intervals));

  stdr::for_each(stde::par, intervals | stdv::adjacent<2>,
    [&] (auto lr) { auto [l, r] = lr;
      if (!r.flag)
        out[l.index] = …;
    });

  …
}
```

```
auto chunk_by(stdr::range auto&& in, stdr::range auto&& out, auto op) {
  std::vector<interval> intervals(size(in) + 1);

  intervals[0] = interval{true, 0, 1, 1};
  stdr::transform(stde::par, in | stdv::adjacent<2>, begin(intervals) + 1,
    [&] (auto lr) { auto [l, r] = lr;
      bool b = op(l, r);
      return interval{b, !b, 1, 1};
    });
  intervals.back() = interval{false, 1, 1, 1};

  stdr::inclusive_scan(stde::par, intervals, begin(intervals));

  stdr::for_each(stde::par, intervals | stdv::adjacent<2>,
    [&] (auto lr) { auto [l, r] = lr;
      if (!r.flag)
        out[l.index] = stdr::subrange(next(begin(in), …),
                                      next(begin(in), …));
    });

  …
}
```

a	g	g	c	c	c	t	a	a

After Scan

Flags	true	false	true	false	true	true	false	false	true	false
Index	0	1	1	2	2	2	3	4	4	5
Count	1	1	2	1	2	3	1	1	2	1
End	1	2	3	4	5	6	7	8	9	10

Write Pass

Flags		false		false			false	false		false
Index	0		1			2	3		4	
Count	1		2			3	1		2	
End	1		3			6	7		9	

```
auto chunk_by(stdr::range auto&& in, stdr::range auto&& out, auto op) {
  std::vector<interval> intervals(size(in) + 1);

  intervals[0] = interval{true, 0, 1, 1};
  stdr::transform(stde::par, in | stdv::adjacent<2>, begin(intervals) + 1,
    [&] (auto lr) { auto [l, r] = lr;
      bool b = op(l, r);
      return interval{b, !b, 1, 1};
    });
  intervals.back() = interval{false, 1, 1, 1};

  stdr::inclusive_scan(stde::par, intervals, begin(intervals));

  stdr::for_each(stde::par, intervals | stdv::adjacent<2>,
    [&] (auto lr) { auto [l, r] = lr;
      if (!r.flag)
        out[l.index] = stdr::subrange(next(begin(in), …),
                                      next(begin(in), l.end));
    });

  …
}
```

```
auto chunk_by(stdr::range auto&& in, stdr::range auto&& out, auto op) {
  std::vector<interval> intervals(size(in) + 1);

  intervals[0] = interval{true, 0, 1, 1};
  stdr::transform(stde::par, in | stdv::adjacent<2>, begin(intervals) + 1,
    [&] (auto lr) { auto [l, r] = lr;
      bool b = op(l, r);
      return interval{b, !b, 1, 1};
    });
  intervals.back() = interval{false, 1, 1, 1};

  stdr::inclusive_scan(stde::par, intervals, begin(intervals));

  stdr::for_each(stde::par, intervals | stdv::adjacent<2>,
    [&] (auto lr) { auto [l, r] = lr;
      if (!r.flag)
        out[l.index] = stdr::subrange(next(begin(in), l.end - l.count),
                                      next(begin(in), l.end));
    });

  …
}
```

a	g	g	c	c	c	t	a	a

After Scan

Flags	true	false	true	false	true	true	false	false	true	false
Index	0	1	1	2	2	2	3	4	4	5
Count	1	1	2	1	2	3	1	1	2	1
End	1	2	3	4	5	6	7	8	9	10

Write Pass

Flags		false		false			false	false		false
Index	0		1			2	3		4	
Count	1		2			3	1		2	
End	1		3			6	7		9	
Range	[0, 1)		[1, 3)			[3, 6)	[6, 7)		[7, 9)	

a	g	g	c	c	c	t	a	a

After Scan

Flags	true	false	true	false	true	true	false	false	true	false
Index	0	1	1	2	2	2	3	4	4	5
Count	1	1	2	1	2	3	1	1	2	1
End	1	2	3	4	5	6	7	8	9	10

Write Pass

Flags		false		false			false	false		false
Index	0		1			2	3		4	
Count	1		2			3	1		2	
End	1		3			6	7		9	
Range	[0, 1)		1, 3)			[3, 6)	[6, 7)		[7, 9)	

a	g	g	c	c	c	t	a	a

After Scan

Flags	true	false	true	false	true	true	false	false	true	false
Index	0	1	1	2	2	2	3	4	4	5
Count	1	1	2	1	2	3	1	1	2	1
End	1	2	3	4	5	6	7	8	9	10

Write Pass

Flags		false		false			false	false		false
Index	0		1			2	3		4	
Count	1		2			3	1		2	
End	1		3			6	7		9	
Range	[0, 1		[1, 3)			[3, 6)	[6, 7)		[7, 9)	

a	g	g	c	c	c	t	a	a

After Scan

Flags	true	false	true	false	true	true	false	false	true	false
Index	0	1	1	2	2	2	3	4	4	5
Count	1	1	2	1	2	3	1	1	2	1
End	1	2	3	4	5	6	7	8	9	10

Write Pass

Flags		false		false			false	false		false
Index	0		1			2	3		4	
Count	1		2			3	1		2	
End	1		3			6	7		9	
Range	[0, 1)		[1, 3)			[3, 6)	[6, 7)		[7, 9)	

a	g	g	c	c	c	t	a	a

After Scan

Flags	true	false	true	false	true	true	false	false	true	false
Index	0	1	1	2	2	2	3	4	4	5
Count	1	1	2	1	2	3	1	1	2	1
End	1	2	3	4	5	6	7	8	9	10

Write Pass

Flags		false		false			false	false		false
Index	0		1			2	3		4	
Count	1		2			3	1		2	
End	1		3			6	7		9	
Range	[0, 1)		[1, 3)			[3, 6	[6, 7)		7, 9)	

a	g	g	c	c	c	t	a	a

After Scan

Flags	true	false	true	false	true	true	false	false	true	false
Index	0	1	1	2	2	2	3	4	4	5
Count	1	1	2	1	2	3	1	1	2	1
End	1	2	3	4	5	6	7	8	9	10

Write Pass

Flags		false		false			false	false		false
Index	0		1			2	3		4	
Count	1		2			3	1		2	
End	1		3			6	7		9	
Range	[0, 1)		[1, 3)			[3, 6)	[6, 7)		[7, 9)	

```
auto chunk_by(stdr::range auto&& in, stdr::range auto&& out, auto op) {
  std::vector<interval> intervals(size(in) + 1);

  intervals[0] = interval{true, 0, 1, 1};
  stdr::transform(stde::par, in | stdv::adjacent<2>, begin(intervals) + 1,
    [&] (auto lr) { auto [l, r] = lr;
      bool b = op(l, r);
      return interval{b, !b, 1, 1};
    });
  intervals.back() = interval{false, 1, 1, 1};

  stdr::inclusive_scan(stde::par, intervals, begin(intervals));

  stdr::for_each(stde::par, intervals | stdv::adjacent<2>,
    [&] (auto lr) { auto [l, r] = lr;
      if (!r.flag)
        out[l.index] = stdr::subrange(next(begin(in), l.end - l.count),
                                      next(begin(in), l.end));
    });

  return stdr::subrange(begin(out), next(begin(out), intervals.back().index));
}
```

a	g	g	c	c	c	t	a	a

After Scan

Flags	true	false	true	false	true	true	false	false	true	false
Index	0	1	1	2	2	2	3	4	4	5
Count	1	1	2	1	2	3	1	1	2	1
End	1	2	3	4	5	6	7	8	9	10

Write Pass

Flags		false		false			false	false		false
Index	0		1			2	3		4	
Count	1		2			3	1		2	
End	1		3			6	7		9	
Range	[0, 1)		[1, 3)			[3, 6)	[6, 7)		[7, 9)	

```
auto chunk_by(stdr::range auto&& in, stdr::range auto&& out, auto op) {
  std::vector<interval> intervals(size(in) + 1);

  intervals[0] = interval{true, 0, 1, 1};
  stdr::transform(stde::par, in | stdv::adjacent<2>, begin(intervals) + 1,
    [&] (auto lr) { auto [l, r] = lr;
      bool b = op(l, r);
      return interval{b, !b, 1, 1};
    });
  intervals.back() = interval{false, 1, 1, 1};

  stdr::inclusive_scan(stde::par, intervals, begin(intervals));

  stdr::for_each(stde::par, intervals | stdv::adjacent<2>,
    [&] (auto lr) { auto [l, r] = lr;
      if (!r.flag)
        out[l.index] = stdr::subrange(next(begin(in), l.end - l.count),
                                      next(begin(in), l.end));
    });

  return stdr::subrange(begin(out), next(begin(out), intervals.back().index));
}
```

```
auto chunk_by(stdr::range auto&& in, stdr::range auto&& out, auto op) {
  std::vector<interval> intervals(size(in) + 1);

  intervals[0] = interval{true, 0, 1, 1};
  stdr::transform(stde::par, in | stdv::adjacent<2>, begin(intervals) + 1,
    [&] (auto lr) { auto [l, r] = lr;
      bool b = op(l, r);
      return interval{b, !b, 1, 1};
    });
  intervals.back() = i

  stdr::inclusive_scan                                 vals));

  stdr::for_each(stde:                                 >,
    [&] (auto lr) { auto [l, r] = lr;
      if (!r.flag)
        out[l.index] = stdr::subrange(next(begin(in), l.end - l.count),
                                      next(begin(in), l.end));
    });

  return stdr::subrange(begin(out), next(begin(out), intervals.back().index));
}
```

Analysis

O(input) storage

3 global synchronizations

```
auto chunk_by(stdr::range auto&& in, stdr::range auto&& out, auto op, std::uint32_t num_tiles) {
  …
}
```

```
auto chunk_by(stdr::range auto&& in, stdr::range auto&& out, auto op, std::uint32_t num_tiles) {
  scan_tile_state<interval> sts(num_tiles);
  std::atomic<std::uint32_t> tile_counter(0);

  …
}
```

```
auto chunk_by(stdr::range auto&& in, stdr::range auto&& out, auto op, std::uint32_t num_tiles) {
  scan_tile_state<interval> sts(num_tiles);
  std::atomic<std::uint32_t> tile_counter(0);

  stdr::for_each(stde::par, stdv::iota(0, num_tiles), [&] (std::size_t) {
      …
    });
  …
}
```

```
auto chunk_by(stdr::range auto&& in, stdr::range auto&& out, auto op, std::uint32_t num_tiles) {
  scan_tile_state<interval> sts(num_tiles);
  std::atomic<std::uint32_t> tile_counter(0);

  stdr::for_each(stde::par, stdv::iota(0, num_tiles), [&] (std::size_t) {
      auto tile = tile_counter.fetch_add(1, std::memory_order_relaxed);
      …
    });
  …
}
```

```
auto chunk_by(stdr::range auto&& in, stdr::range auto&& out, auto op, std::uint32_t num_tiles) {
  scan_tile_state<interval> sts(num_tiles);
  std::atomic<std::uint32_t> tile_counter(0);

  stdr::for_each(stde::par, stdv::iota(0, num_tiles), [&] (std::size_t) {
      auto tile = tile_counter.fetch_add(1, std::memory_order_relaxed);
      bool is_first_tile    = tile == 0;
      bool is_last_tile     = tile == num_tiles - 1;
      bool is_interior_tile = tile > 0 && tile < num_tiles - 1;

      …
    });
  …
}
```

```
auto chunk_by(stdr::range auto&& in, stdr::range auto&& out, auto op, std::uint32_t num_tiles) {
  scan_tile_state<interval> sts(num_tiles);
  std::atomic<std::uint32_t> tile_counter(0);

  stdr::for_each(stde::par, stdv::iota(0, num_tiles), [&] (std::size_t) {
      auto tile = tile_counter.fetch_add(1, std::memory_order_relaxed);
      bool is_first_tile    = tile == 0;
      bool is_last_tile     = tile == num_tiles - 1;
      bool is_interior_tile = tile > 0 && tile < num_tiles - 1;

      auto sub_in = range_for_tile(in, tile, num_tiles);
      if (!is_first_tile) sub_in = stdr::subrange(--begin(sub_in), end(sub_in));

      …
    });
  …
}
```

```
auto chunk_by(stdr::range auto&& in, stdr::range auto&& out, auto op, std::uint32_t num_tiles) {
  scan_tile_state<interval> sts(num_tiles);
  std::atomic<std::uint32_t> tile_counter(0);

  stdr::for_each(stde::par, stdv::iota(0, num_tiles), [&] (std::size_t) {
      auto tile = tile_counter.fetch_add(1, std::memory_order_relaxed);
      bool is_first_tile    = tile == 0;
      bool is_last_tile     = tile == num_tiles - 1;
      bool is_interior_tile = tile > 0 && tile < num_tiles - 1;

      auto sub_in = range_for_tile(in, tile, num_tiles);
      if (!is_first_tile) sub_in = stdr::subrange(--begin(sub_in), end(sub_in));

      std::vector<interval> intervals(…);

      stdr::transform(sub_in | stdv::adjacent<2>, …,
        …);


      …
    });
  …
}
```

```
auto chunk_by(stdr::range auto&& in, stdr::range auto&& out, auto op, std::uint32_t num_tiles) {
  scan_tile_state<interval> sts(num_tiles);
  std::atomic<std::uint32_t> tile_counter(0);

  stdr::for_each(stde::par, stdv::iota(0, num_tiles), [&] (std::size_t) {
      auto tile = tile_counter.fetch_add(1, std::memory_order_relaxed);
      bool is_first_tile    = tile == 0;
      bool is_last_tile     = tile == num_tiles - 1;
      bool is_interior_tile = tile > 0 && tile < num_tiles - 1;

      auto sub_in = range_for_tile(in, tile, num_tiles);
      if (!is_first_tile) sub_in = stdr::subrange(--begin(sub_in), end(sub_in));

      std::vector<interval> intervals(size(sub_in) - is_interior_tile);

      stdr::transform(sub_in | stdv::adjacent<2>, …,
        …);


      …
    });
  …
}
```

```
auto chunk_by(stdr::range auto&& in, stdr::range auto&& out, auto op, std::uint32_t num_tiles) {
 scan_tile_state<interval> sts(num_tiles);
 std::atomic<std::uint32_t> tile_counter(0);

 stdr::for_each(stde::par, stdv::iota(0, num_tiles), [&] (std::size_t) {
   auto tile = tile_counter.fetch_add(1, std::memory_order_relaxed);
   bool is_first_tile    = tile == 0;
   bool is_last_tile     = tile == num_tiles - 1;
   bool is_interior_tile = tile > 0 && tile < num_tiles - 1;

   auto sub_in = range_for_tile(in, tile, num_tiles);
   if (!is_first_tile) sub_in = stdr::subrange(--begin(sub_in), end(sub_in));

   std::vector<interval> intervals(size(sub_in) - is_interior_tile);
   if (is_first_tile) intervals[0] = interval{true, 0, 1, 1};
   stdr::transform(sub_in | stdv::adjacent<2>, …,
    …);
   if (is_last_tile) intervals.back() = interval{false, 1, 1, 1};

   …
  });
 …
}
```

```
auto chunk_by(stdr::range auto&& in, stdr::range auto&& out, auto op, std::uint32_t num_tiles) {
  scan_tile_state<interval> sts(num_tiles);
  std::atomic<std::uint32_t> tile_counter(0);

  stdr::for_each(stde::par, stdv::iota(0, num_tiles), [&] (std::size_t) {
      auto tile = tile_counter.fetch_add(1, std::memory_order_relaxed);
      bool is_first_tile    = tile == 0;
      bool is_last_tile     = tile == num_tiles - 1;
      bool is_interior_tile = tile > 0 && tile < num_tiles - 1;

      auto sub_in = range_for_tile(in, tile, num_tiles);
      if (!is_first_tile) sub_in = stdr::subrange(--begin(sub_in), end(sub_in));

      std::vector<interval> intervals(size(sub_in) - is_interior_tile);
      if (is_first_tile) intervals[0] = interval{true, 0, 1, 1};
      stdr::transform(sub_in | stdv::adjacent<2>, begin(intervals) + is_first_tile,
        …);
      if (is_last_tile) intervals.back() = interval{false, 1, 1, 1};

      …
    });
  …
}
```

```
auto chunk_by(stdr::range auto&& in, stdr::range auto&& out, auto op, std::uint32_t num_tiles) {
  scan_tile_state<interval> sts(num_tiles);
  std::atomic<std::uint32_t> tile_counter(0);

  stdr::for_each(stde::par, stdv::iota(0, num_tiles), [&] (std::size_t) {
      auto tile = tile_counter.fetch_add(1, std::memory_order_relaxed);
      bool is_first_tile    = tile == 0;
      bool is_last_tile     = tile == num_tiles - 1;
      bool is_interior_tile = tile > 0 && tile < num_tiles - 1;

      auto sub_in = range_for_tile(in, tile, num_tiles);
      if (!is_first_tile) sub_in = stdr::subrange(--begin(sub_in), end(sub_in));

      std::vector<interval> intervals(size(sub_in) - is_interior_tile);
      if (is_first_tile) intervals[0] = interval{true, 0, 1, 1};
      stdr::transform(sub_in | stdv::adjacent<2>, begin(intervals) + is_first_tile,
        [&] (auto lr) { auto [l, r] = lr; bool b = op(l, r); return interval{b, !b, 1, 1}; });
      if (is_last_tile) intervals.back() = interval{false, 1, 1, 1};

      …
    });
  …
}
```

```
auto chunk_by(stdr::range auto&& in, stdr::range auto&& out, auto op, std::uint32_t num_tiles) {
  scan_tile_state<interval> sts(num_tiles);
  std::atomic<std::uint32_t> tile_counter(0);

  stdr::for_each(stde::par, stdv::iota(0, num_tiles), [&] (std::size_t) {
      auto tile = tile_counter.fetch_add(1, std::memory_order_relaxed);
      bool is_first_tile    = tile == 0;
      bool is_last_tile     = tile == num_tiles - 1;
      bool is_interior_tile = tile > 0 && tile < num_tiles - 1;

      auto sub_in = range_for_tile(in, tile, num_tiles);
      if (!is_first_tile) sub_in = stdr::subrange(--begin(sub_in), end(sub_in));

      std::vector<interval> intervals(size(sub_in) - is_interior_tile);
      if (is_first_tile) intervals[0] = interval{true, 0, 1, 1};
      stdr::transform(sub_in | stdv::adjacent<2>, begin(intervals) + is_first_tile,
        [&] (auto lr) { auto [l, r] = lr; bool b = op(l, r); return interval{b, !b, 1, 1}; });
      if (is_last_tile) intervals.back() = interval{false, 1, 1, 1};

      sts.set_local_prefix(tile, *--stdr::inclusive_scan(stde::par, intervals, begin(intervals)));
      if (!is_first_tile) {
        auto pred = sts.wait_for_predecessor_prefix(tile);
        stdr::for_each(intervals, [&] (auto& e) { e = pred + e; });
      }

      …
    });
  …
}
```

```
auto chunk_by(stdr::range auto&& in, stdr::range auto&& out, auto op, std::uint32_t num_tiles) {
  scan_tile_state<interval> sts(num_tiles);
  std::atomic<std::uint32_t> tile_counter(0);

  stdr::for_each(stde::par, stdv::iota(0, num_tiles), [&] (std::size_t) {
      auto tile = tile_counter.fetch_add(1, std::memory_order_relaxed);
      bool is_first_tile    = tile == 0;
      bool is_last_tile     = tile == num_tiles - 1;
      bool is_interior_tile = tile > 0 && tile < num_tiles - 1;

      auto sub_in = range_for_tile(in, tile, num_tiles);
      if (!is_first_tile) sub_in = stdr::subrange(--begin(sub_in), end(sub_in));

      std::vector<interval> intervals(size(sub_in) - is_interior_tile);
      if (is_first_tile) intervals[0] = interval{true, 0, 1, 1};
      stdr::transform(sub_in | stdv::adjacent<2>, begin(intervals) + is_first_tile,
        [&] (auto lr) { auto [l, r] = lr; bool b = op(l, r); return interval{b, !b, 1, 1}; });
      if (is_last_tile) intervals.back() = interval{false, 1, 1, 1};

      sts.set_local_prefix(tile, *--stdr::inclusive_scan(stde::par, intervals, begin(intervals)));
      if (!is_first_tile) {
        auto pred = sts.wait_for_predecessor_prefix(tile);
        stdr::for_each(intervals, [&] (auto& e) { e = pred + e; });
      }

      stdr::for_each(intervals | stdv::adjacent<2>,
        …);
    });
  …
}
```

```
auto chunk_by(stdr::range auto&& in, stdr::range auto&& out, auto op, std::uint32_t num_tiles) {
  scan_tile_state<interval> sts(num_tiles);
  std::atomic<std::uint32_t> tile_counter(0);

  stdr::for_each(stde::par, stdv::iota(0, num_tiles), [&] (std::size_t) {
      auto tile = tile_counter.fetch_add(1, std::memory_order_relaxed);
      bool is_first_tile    = tile == 0;
      bool is_last_tile     = tile == num_tiles - 1;
      bool is_interior_tile = tile > 0 && tile < num_tiles - 1;

      auto sub_in = range_for_tile(in, tile, num_tiles);
      if (!is_first_tile) sub_in = stdr::subrange(--begin(sub_in), end(sub_in));

      std::vector<interval> intervals(size(sub_in) - is_interior_tile);
      if (is_first_tile) intervals[0] = interval{true, 0, 1, 1};
      stdr::transform(sub_in | stdv::adjacent<2>, begin(intervals) + is_first_tile,
        [&] (auto lr) { auto [l, r] = lr; bool b = op(l, r); return interval{b, !b, 1, 1}; });
      if (is_last_tile) intervals.back() = interval{false, 1, 1, 1};

      sts.set_local_prefix(tile, *--stdr::inclusive_scan(stde::par, intervals, begin(intervals)));
      if (!is_first_tile) {
        auto pred = sts.wait_for_predecessor_prefix(tile);
        stdr::for_each(intervals, [&] (auto& e) { e = pred + e; });
      }

      stdr::for_each(intervals | stdv::adjacent<2>,
        [&] (auto lr) { auto [l, r] = lr;
          if (!r.flag) out[l.index] = stdr::subrange(next(begin(in), l.end - l.count), next(begin(in), l.end));
        });
    });
  …
}
```

```
auto chunk_by(stdr::range auto&& in, stdr::range auto&& out, auto op, std::uint32_t num_tiles) {
  scan_tile_state<interval> sts(num_tiles);
  std::atomic<std::uint32_t> tile_counter(0);

  stdr::for_each(stde::par, stdv::iota(0, num_tiles), [&] (std::size_t) {
      auto tile = tile_counter.fetch_add(1, std::memory_order_relaxed);
      bool is_first_tile    = tile == 0;
      bool is_last_tile     = tile == num_tiles - 1;
      bool is_interior_tile = tile > 0 && tile < num_tiles - 1;

      auto sub_in = range_for_tile(in, tile, num_tiles);
      if (!is_first_tile) sub_in = stdr::subrange(--begin(sub_in), end(sub_in));

      std::vector<interval> intervals(size(sub_in) - is_interior_tile);
      if (is_first_tile) intervals[0] = interval{true, 0, 1, 1};
      stdr::transform(sub_in | stdv::adjacent<2>, begin(intervals) + is_first_tile,
        [&] (auto lr) { auto [l, r] = lr; bool b = op(l, r); return interval{b, !b, 1, 1}; });
      if (is_last_tile) intervals.back() = interval{false, 1, 1, 1};

      sts.set_local_prefix(tile, *--stdr::inclusive_scan(stde::par, intervals, begin(intervals)));
      if (!is_first_tile) {
        auto pred = sts.wait_for_predecessor_prefix(tile);
        stdr::for_each(intervals, [&] (auto& e) { e = pred + e; });
      }

      stdr::for_each(intervals | stdv::adjacent<2>,
        [&] (auto lr) { auto [l, r] = lr;
          if (!r.flag) out[l.index] = stdr::subrange(next(begin(in), l.end - l.count), next(begin(in), l.end));
        });
    });
  return stdr::subrange(begin(out), next(begin(out), sts.prefixes.back().complete.index));
}
```

```
auto chunk_by(stdr::range auto&& in, stdr::range auto&& out, auto op, std::uint32_t num_tiles) {
  scan_tile_state<interval> sts(num_tiles);
  std::atomic<std::uint32_t> tile_counter(0);

  stdr::for_each(stde::par, stdv::iota(0, num_tiles), [&] (std::size_t) {
      auto tile = tile_counter.fetch_add(1, std::memory_order_relaxed);
      bool is_first_tile    = tile == 0;
      bool is_last_tile     = tile == num_tiles - 1;
      bool is_interior_tile = tile > 0 && tile < num_tiles - 1;

      auto sub_in = range_for_tile(in, tile, num_tiles);
      if (!is_first_tile) sub_in = stdr::subrange(--begin(sub_in), end(sub_in));

      std::vector<interval> intervals(size(sub_in) - is_interior_tile);
      if (is_first_tile) intervals[0] = interval{true, 0, 1, 1};
      stdr::transform(sub_in | stdv::adjacent<2>, begin(intervals) + is_first_tile,
        [&] (auto lr) { auto [l, r] = lr; bool b = op(l, r); return interval{b, !b, 1, 1}; });
      if (is_last_tile) intervals.back() = interval{false, 1, 1, 1};

      sts.set_local_prefix(tile, *--stdr::inclusive_scan(stde::par, intervals, begin(intervals)));
      if (!is_first_tile) {
        auto pred = sts.wait_for_predecessor_prefix(tile);
        stdr::for_each(intervals, [&] (auto& e) { e = pred + e; });
      }

      stdr::for_each(intervals | stdv::adjacent<2>,
        [&] (auto lr) { auto [l, r] = lr;
          if (!r.flag) out[l.index] = stdr::subrange(next(begin(in), l.end - l.count), next(begin(in), l.end));
        });
    });
  return stdr::subrange(begin(out), next(begin(out), sts.prefixes.back().complete.index));
}
```

```
auto chunk_by(stdr::range auto&& in, stdr::range auto&& out, auto op, std::uint32_t num_tiles) {
  scan_tile_state<interval> sts(num_tiles);
  std::atomic<std::uint32_t> tile_counter(0);

  stdr::for_each(stde::par, stdv::iota(0, num_tiles), [&] (std::size_t) {
      auto tile = tile_counter.fe
      bool is_first_tile    = til
      bool is_last_tile     = til
      bool is_interior_tile = til

      auto sub_in = range_for_til
      if (!is_first_tile) sub_in                                            );

      std::vector<interval> inter
      if (is_first_tile) intervals[0] = interval{true, 0, 1, 1};
      stdr::transform(sub_in | stdv::adjacent<2>, begin(intervals) + is_first_tile,
        [&] (auto lr) { auto [l, r] = lr; bool b = op(l, r); return interval{b, !b, 1, 1}; });
      if (is_last_tile) intervals.back() = interval{false, 1, 1, 1};

      sts.set_local_prefix(tile, *--stdr::inclusive_scan(stde::par, intervals, begin(intervals)));
      if (!is_first_tile) {
        auto pred = sts.wait_for_predecessor_prefix(tile);
        stdr::for_each(intervals, [&] (auto& e) { e = pred + e; });
      }

      stdr::for_each(intervals | stdv::adjacent<2>,
        [&] (auto lr) { auto [l, r] = lr;
          if (!r.flag) out[l.index] = stdr::subrange(next(begin(in), l.end - l.count), next(begin(in), l.end));
        });
    });
  return stdr::subrange(begin(out), next(begin(out), sts.prefixes[num_tiles - 1].complete.index));
}
```

Analysis

O(tiles) storage

1 global synchronization

```
auto chunk_by(stdr::range auto&& in, stdr::range auto&& out, auto op, std::uint32_t num_tiles) {
  scan_tile_state<interval> sts(num_tiles);
  std::atomic<std::uint32_t> tile_counter(0);

  stdr::for_each(stde::par, stdv::iota(0, num_tiles), [&] (std::size_t) {
      auto tile = tile_counter.fe
      bool is_first_tile    = til
      bool is_last_tile     = til
      bool is_interior_tile = til

      auto sub_in = range_for_til
      if (!is_first_tile) sub_in                                                  );

      std::vector<interval> inter
      if (is_first_tile) intervals[0] = interval{true, 0, 1, 1};
      stdr::transform(sub_in | stdv::adjacent<2>, begin(intervals) + is_first_tile,
        [&] (auto lr) { auto [l, r] = lr; bool b = op(l, r); return interval{b, !b, 1, 1}; });
      if (is_last_tile) intervals.back() = interval{false, 1, 1, 1};

      sts.set_loc                                                                  ));
      if (!is_fir
        auto pred
        stdr::for
      }

      stdr::for_e
        [&] (auto
          if (!r.flag) out[l.index] = stdr::subrange(next(begin(in), l.end - l.count), next(begin(in), l.end));
        });
    });
  return stdr::subrange(begin(out), next(begin(out), sts.prefixes[num_tiles - 1].complete.index));
}
```

Analysis

O(tiles) storage

1 global synchronization

Performance

9x faster than three pass implementation

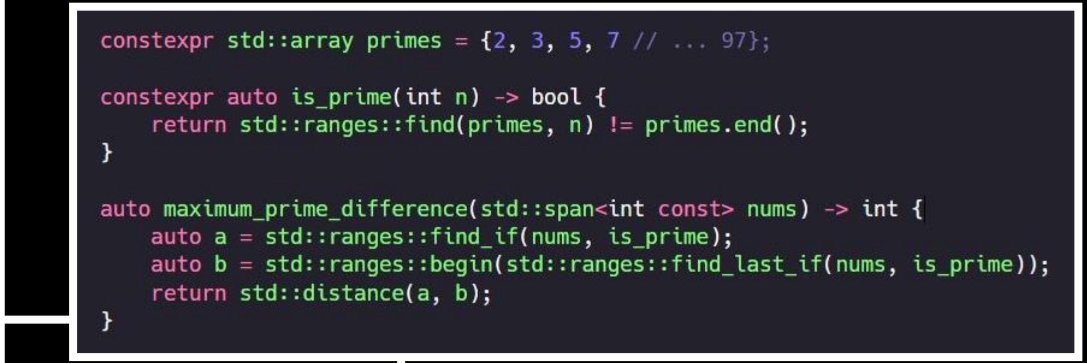
NVC++ 24.3, 2x 32 core EPYC 7513, 1GB 8 bit char input, 1024 tiles

Scan is a building block

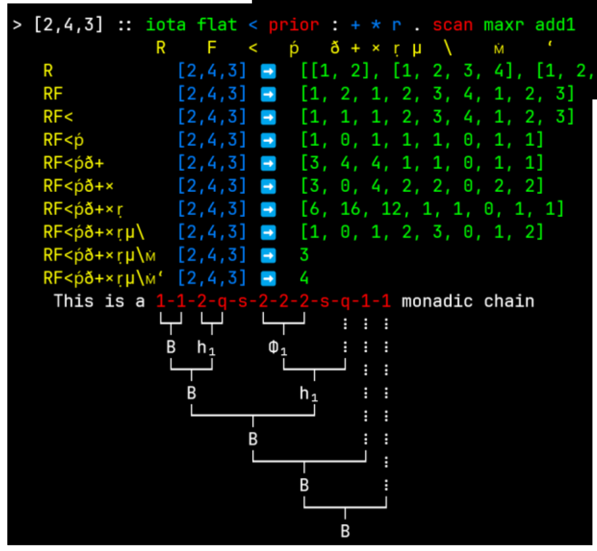
- **Focus on communication**
- **Localize synchronization**
- **Hide latency**

adspthepodcast.com

Algorithms + Data Structures = Programs



```
constexpr std::array primes = {2, 3, 5, 7 // ... 97};

constexpr auto is_prime(int n) -> bool {
    return std::ranges::find(primes, n) != primes.end();
}

auto maximum_prime_difference(std::span<int const> nums) -> int {
    auto a = std::ranges::find_if(nums, is_prime);
    auto b = std::ranges::begin(std::ranges::find_last_if(nums, is_prime));
    return std::distance(a, b);
}
```



```
> [2,4,3] :: iota flat < prior : + * r . scan maxr add1
                 R    F   <   ṕ   ð + × ṛ µ  \    ṁ    ʻ
   R              [2,4,3] ➡  [[1, 2], [1, 2, 3, 4], [1, 2, 3]]
   RF             [2,4,3] ➡  [1, 2, 1, 2, 3, 4, 1, 2, 3]
   RF<            [2,4,3] ➡  [1, 1, 1, 2, 3, 4, 1, 2, 3]
   RF<ṕ           [2,4,3] ➡  [1, 0, 1, 1, 1, 0, 1, 1]
   RF<ṕð+         [2,4,3] ➡  [3, 4, 4, 1, 1, 0, 1, 1]
   RF<ṕð+×        [2,4,3] ➡  [3, 0, 4, 2, 2, 0, 2, 2]
   RF<ṕð+×ṛ       [2,4,3] ➡  [6, 16, 12, 1, 1, 0, 1, 1]
   RF<ṕð+×ṛµ\     [2,4,3] ➡  [1, 0, 1, 2, 3, 0, 1, 2]
   RF<ṕð+×ṛµ\ṁ    [2,4,3] ➡  3
   RF<ṕð+×ṛµ\ṁʻ   [2,4,3] ➡  4
    This is a 1-1-2-q-s-2-2-2-s-q-1-1 monadic chain
              └─┬┘ └─┬┘  └──┬──┘    ⁞ ⁞ ⁞
                B   h₁      Φ₁      ⁞ ⁞ ⁞
                └─┬──┘      └──┬────┘ ⁞ ⁞
                  B            h₁     ⁞ ⁞
                  └──────┬─────┘      ⁞ ⁞
                         B            ⁞ ⁞
                         └─────┬──────┘ ⁞
                               B        ⁞
                               └───┬────┘
                                   B
```

THINK PARALLEL: SCANS

Bryce Adelstein Lelbach

Principal Architect

✉ `brycelelbach@gmail.com` 🐦 `@blelbach`