

# Endnote: AI-Assisted Software Engineering

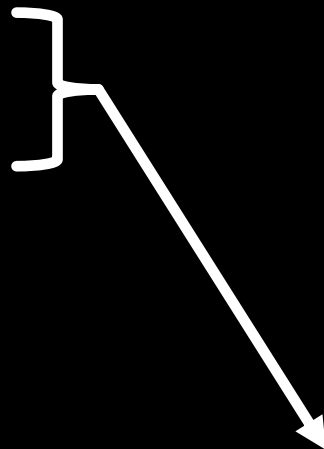
Bryce Adelstein Lelbach

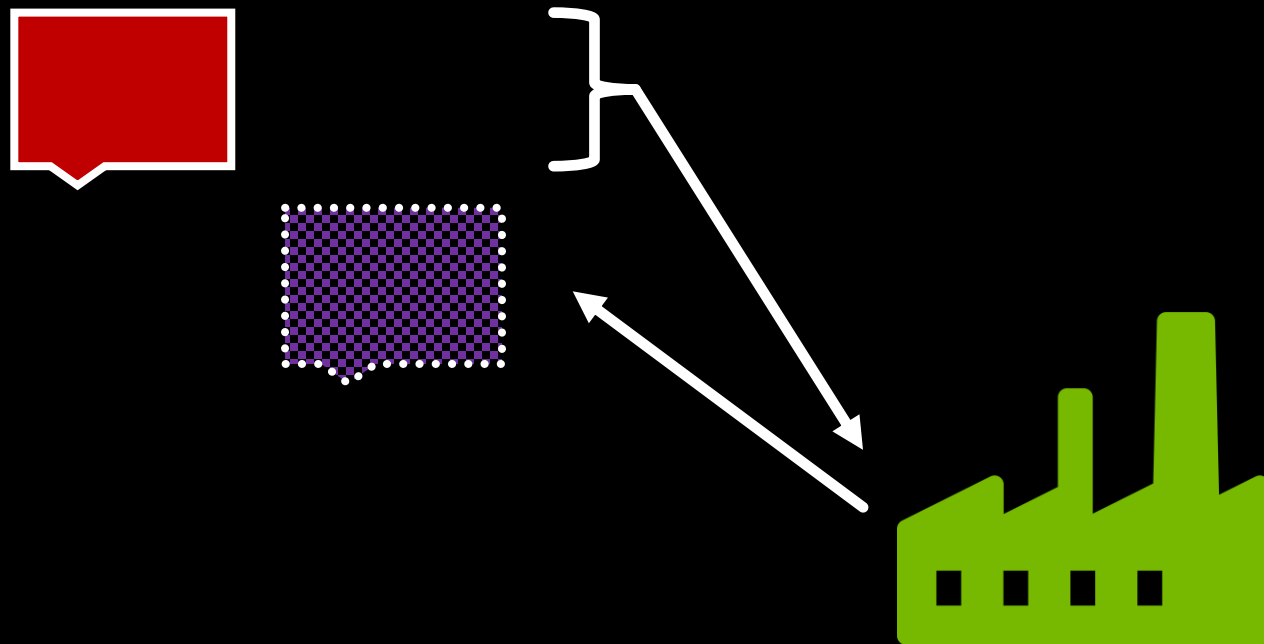
(feat. Conor Hoekstra)

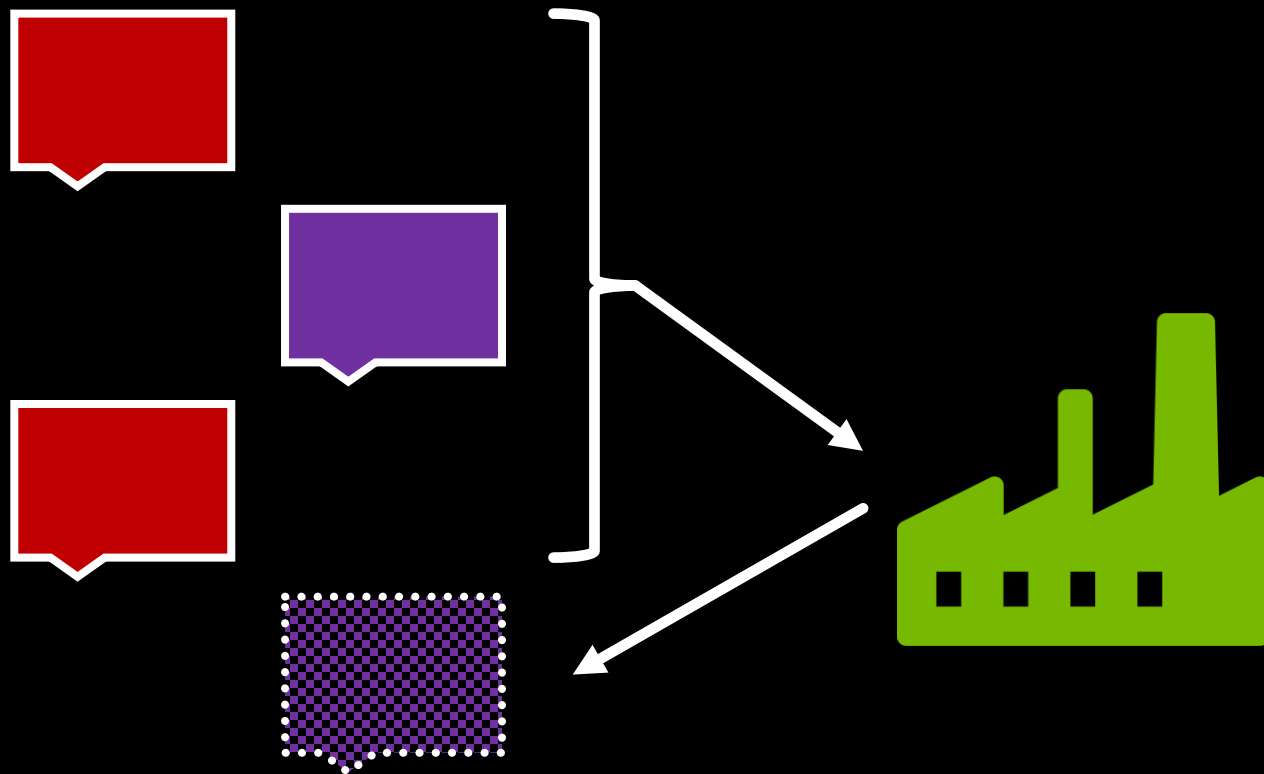


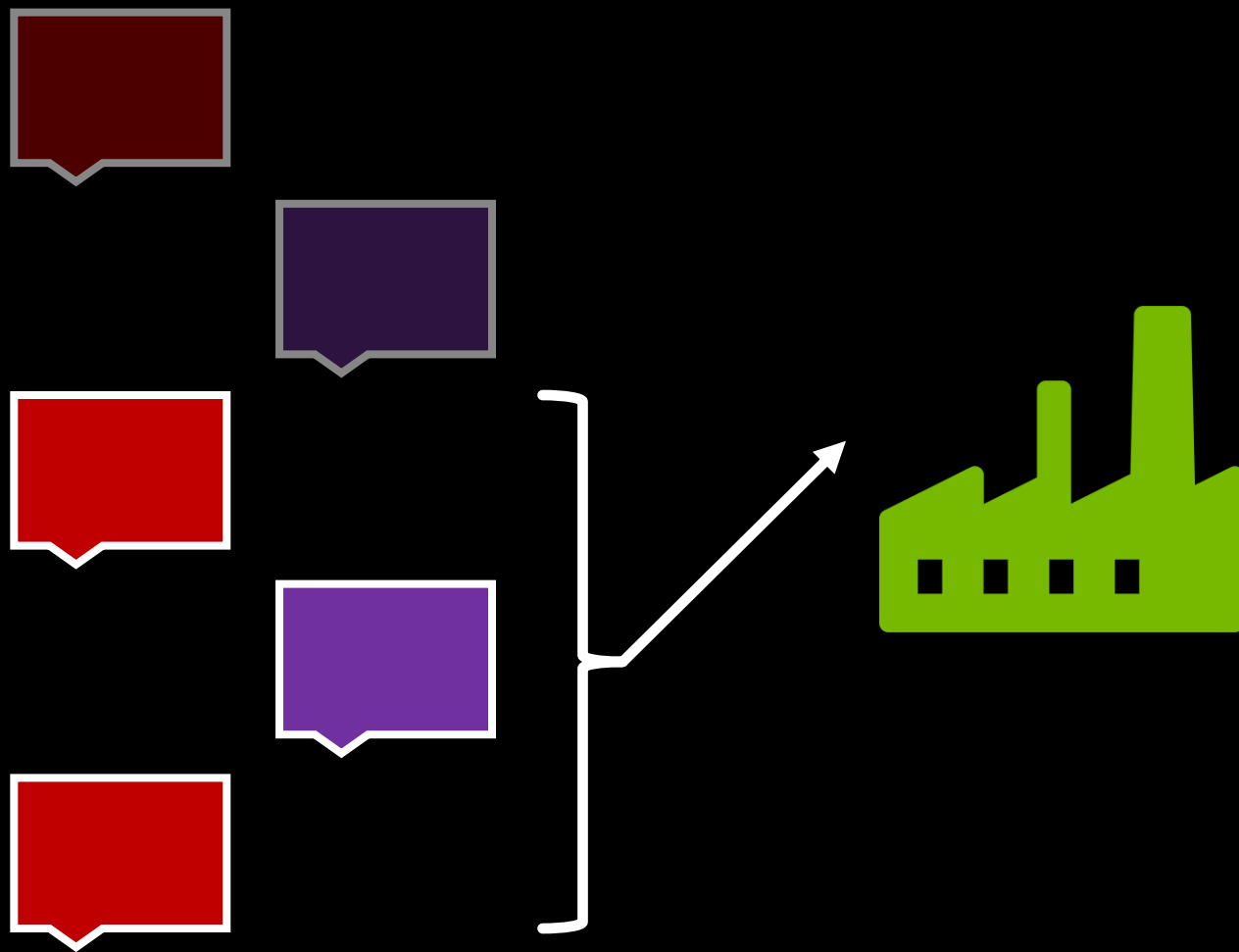






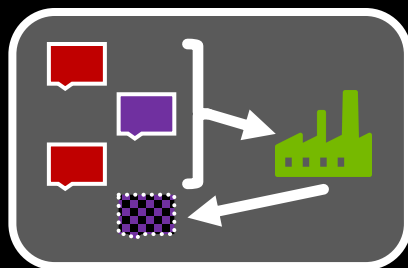


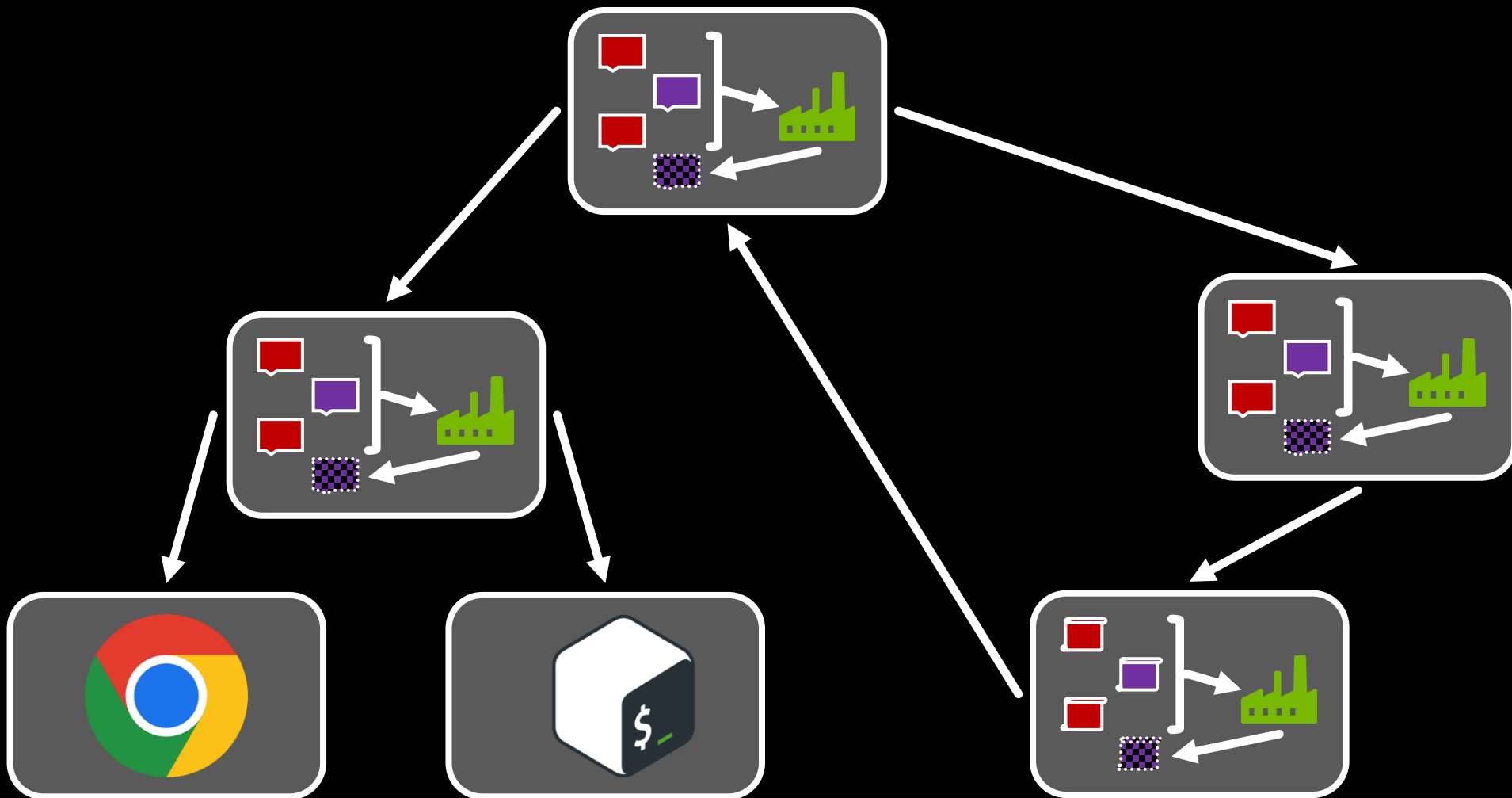




Neural networks  
are the building blocks of AI







# What do we do?

# ➤ Creation

➤ Creation

➤ Analysis

```

test.cpp: In function 'int main()':
test.cpp:6:17: error: no matching function for call to
'transform(std::vector<int>::iterator, std::vector<int>::iterator,
main():<lambda(int)>)'
6   | std::transform(v.begin(), v.end(), [](int i) { return i * i; });
    | ~~~~~^~~~~~
In file included from /usr/include/c++/14.0.0/algorithm:61,
from test.cpp:1:
/usr/include/c++/14.0.0/bits/stl_algo.h:4205:5: note: candidate:
'template<class _IIter, class _OIter, class _UnaryOp> _OIter
std::transform(_IIter, _IIter, _OIter, _UnaryOp)'
4205 | transform(_InputIterator __first, _InputIterator __last,
    | ~~~~~^~~~~~
/usr/include/c++/14.0.0/bits/stl_algo.h:4205:5: note: template argument
deduction/substitution failed:
test.cpp:6:19: note: candidate expects 4 arguments, 3 provided
6   | std::transform(v.begin(), v.end(), [](int i) { return i * i; });
    | ~~~~~^~~~~~
/usr/include/c++/14.0.0/bits/stl_algo.h:4243:5: note: candidate:
'template<class _IIter1, class _IIter2, class _OIter, class _BinaryOp>
_OIter std::transform(_IIter1, _IIter1, _IIter2, _OIter, _BinaryOp)'
4243 | transform(_InputIterator1 __first1, _InputIterator1 __last1,
    | ~~~~~^~~~~~
/usr/include/c++/14.0.0/bits/stl_algo.h:4243:5: note: template argument
deduction/substitution failed:
test.cpp:6:19: note: candidate expects 5 arguments, 3 provided
6   | std::transform(v.begin(), v.end(), [](int i) { return i * i; });
    | ~~~~~^~~~~~

```

```

In file included from /usr/include/c++/14.0.0/algorithm:73:
/usr/include/c++/14.0.0/pstl/glue_algorithm_defs.h:156:1: note:
candidate: 'template<class _ExecPolicy, class _FwdIt1, class _FwdIt2,
class _UnaryOp>
__pstl::__internal::__enable_if_execution_policy< _ExecPolicy, _FwdIt2>
std::transform(_ExecPolicy&&, _FwdIt1, _FwdIt1, _FwdIt2, _UnaryOp)'
156 | transform(_ExecPolicy&& __exec, _FwdIt1 __first, _FwdIt1 __last,
    | ~~~~~^~~~~~
/usr/include/c++/14.0.0/pstl/glue_algorithm_defs.h:156:1: note: template
argument deduction/substitution failed:
test.cpp:6:19: note: deduced conflicting types for parameter '_FwdIt1'
('gnu_cxx::__normal_iterator<int*, std::vector<int> >' and
'main():<lambda(int)>')
6   | std::transform(v.begin(), v.end(), [](int i) { return i * i; });
    | ~~~~~^~~~~~
/usr/include/c++/14.0.0/pstl/glue_algorithm_defs.h:162:1: note:
candidate: 'template<class _ExecPolicy, class _FwdIt1, class _FwdIt2,
class _FwdIt, class _BinaryOp>
__pstl::__internal::__enable_if_execution_policy< _ExecPolicy, _FwdIt2>
std::transform(_ExecPolicy&&, _FwdIt1, _FwdIt1, _FwdIt2, _FwdIt,
_BinaryOp)'
162 | transform(_ExecPolicy&& __exec, _FwdIt1 __first1, _FwdIt1
    | ~~~~~^~~~~~
    | __last1, _FwdIt2 __first2,
/usr/include/c++/14.0.0/pstl/glue_algorithm_defs.h:162:1: note: template
argument deduction/substitution failed:
test.cpp:6:19: note: deduced conflicting types for parameter '_FwdIt1'
('gnu_cxx::__normal_iterator<int*, std::vector<int> >' and
'main():<lambda(int)>')
6   | std::transform(v.begin(), v.end(), [](int i) { return i * i; });
    | ~~~~~^~~~~~

```

We need more information.

How do we identify what information we need?

```
test.cpp: In function 'int main()':
test.cpp:6:17: error: no matching function for call
'transform(std::vector<int>::iterator, std::vector<int>::iterator,
main():<lambda(int)>)'
6   | std::transform(v.begin(), v.end(), [](int i)
  | ~~~~~^~~~~~
In file included from /usr/include/c++/14.0.0/algorithm:73:
from test.cpp:1:
/usr/include/c++/14.0.0/bits/stl_algo.h:4205:5: note:
'template<class _IIter, class _OIter, class _UnaryOp> _OIter
std::transform(_IIter, _IIter, _OIter, _UnaryOp)'
4205 | transform(_InputIterator __first, _InputIterator __last,
  | ~~~~~^~~~~~
/usr/include/c++/14.0.0/bits/stl_algo.h:4205:5: note: template argument
deduction/substitution failed:
test.cpp:6:19: note: candidate expects 4 arguments, 3 provided
6   | std::transform(v.begin(), v.end(), [](int i) { return i * i; });
  | ~~~~~^~~~~~
/usr/include/c++/14.0.0/bits/stl_algo.h:4243:5: note: candidate:
'template<class _IIter1, class _IIter2, class _OIter, class _BinaryOp>
_OIter std::transform(_IIter1, _IIter1, _IIter2, _OIter, _BinaryOp)'
4243 | transform(_InputIterator1 __first1, _InputIterator1 __last1,
  | ~~~~~^~~~~~
/usr/include/c++/14.0.0/bits/stl_algo.h:4243:5: note: template argument
deduction/substitution failed:
test.cpp:6:19: note: candidate expects 5 arguments, 3 provided
6   | std::transform(v.begin(), v.end(), [](int i) { return i * i; });
  | ~~~~~^~~~~~
```

```
/usr/include/c++/14.0.0/algorithm:73:
0/pstl/glue_algorithm_defs.h:156:1: note:
class _ExecPolicy, class _FwdIt1, class _FwdIt2,
_enable_if_execution_policy< _ExecPolicy, _FwdIt2>
policy&&, _FwdIt1, _FwdIt1, _FwdIt2, _UnaryOp)'
cPolicy&& __exec, _FwdIt1 __first, _FwdIt1 __last,
0/pstl/glue_algorithm_defs.h:156:1: note: template
argument deduction/substitution failed:
test.cpp:6:19: note: deduced conflicting types for parameter '_FwdIt1'
('gnu_cxx::__normal_iterator<int*, std::vector<int> >' and
'main():<lambda(int)>')
6   | std::transform(v.begin(), v.end(), [](int i) { return i * i; });
  | ~~~~~^~~~~~
/usr/include/c++/14.0.0/pstl/glue_algorithm_defs.h:162:1: note:
candidate: 'template<class _ExecPolicy, class _FwdIt1, class _FwdIt2,
class _FwdIt, class _BinaryOp>
__pstl::__internal::__enable_if_execution_policy< _ExecPolicy, _FwdIt2>
std::transform(_ExecPolicy&&, _FwdIt1, _FwdIt1, _FwdIt2, _FwdIt,
_BinaryOp)'
162 | transform(_ExecPolicy&& __exec, _FwdIt1 __first1, _FwdIt1
  | ~~~~~^~~~~~
/usr/include/c++/14.0.0/pstl/glue_algorithm_defs.h:162:1: note: template
argument deduction/substitution failed:
test.cpp:6:19: note: deduced conflicting types for parameter '_FwdIt1'
('gnu_cxx::__normal_iterator<int*, std::vector<int> >' and
'main():<lambda(int)>')
6   | std::transform(v.begin(), v.end(), [](int i) { return i * i; });
  | ~~~~~^~~~~~
```





The image is a composite of several elements:

- Top Left:** A snippet of C++ code from `test.cpp`. It shows a function `main()` that calls `std::transform` on a vector `v` using a lambda function. The code is partially obscured by a redacted area.
- Top Right:** A dark gray box with white text that reads: "We need n" and "How do v" followed by "informa".
- Bottom Left:** Another snippet of C++ code from `test.cpp`, showing the definition of a `transform` function template.
- Bottom Right:** A diagram of a classification system. It features a central box labeled "Classification" with the question "What tools are involved?". Below this, there are two more boxes: "What programming languages, data formats," and "What programming languages, data formats,".

# Classification

## What tools are involved?

What programming languages, data formats, etc, are involved?

# We need more information.

## How do we identify what information we need?

information.  
Identify what  
we need?

```

/usr/include/c++/14.0.0/algorithm:73:
./0/pstl/glue_algorithm_defs.h:156:1: note:
class _ExecPolicy, class _FwdIt1, class _FwdIt2,

_enable_if_execution_policy< _ExecPolicy, _FwdIt2>
policy&&, _FwdIt1, _FwdIt1, _FwdIt2, _UnaryOp)'
cPolicy&& __exec, _FwdIt1 __first, _FwdIt1 __last,

./0/pstl/glue_algorithm_defs.h:156:1: note: template
argument deduction/substitution failed:
test.cpp:6:19: note: deduced conflicting types for parameter '_FwdIt1'
('gnu_cxx::__normal_iterator<int*, std::vector<int> >' and
'main()::<lambda(int)>')
6 | std::transform(v.begin(), v.end(), [](int i) { return i * i; });
| ~~~~~
/usr/include
candidate:
class _FwdI
__pstl::_i
std::transf
_BinaryOp)'
162 | tran
__last1, _F
| ^~~~~
/usr/includ
argument de
test.cpp:6:
('gnu_cxx
'main()::<l
6 | std:
| ~~~~~

```

## Source Identification

What files or resources do  
we need to access?

What source locations, code  
entities, or content do we  
need to extract?

```

:
_FwdIt2,

_FwdIt2>
t,
t1

: template

_FwdIt1'

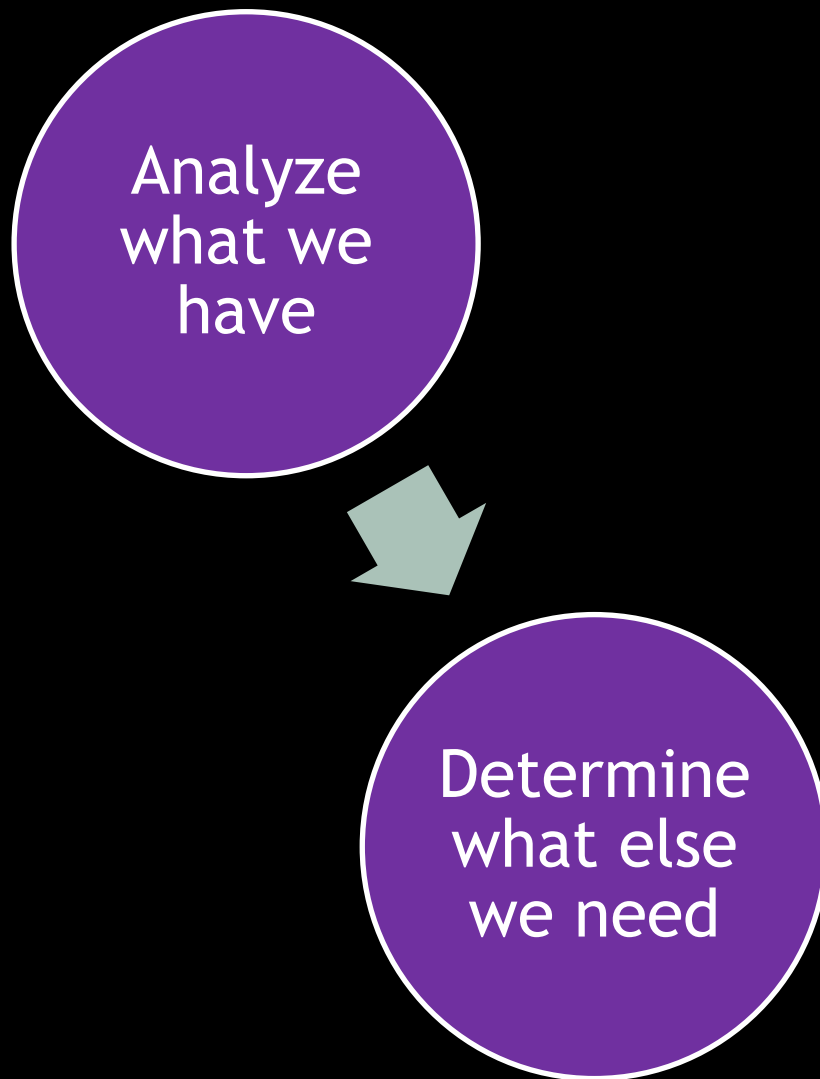
* i; });

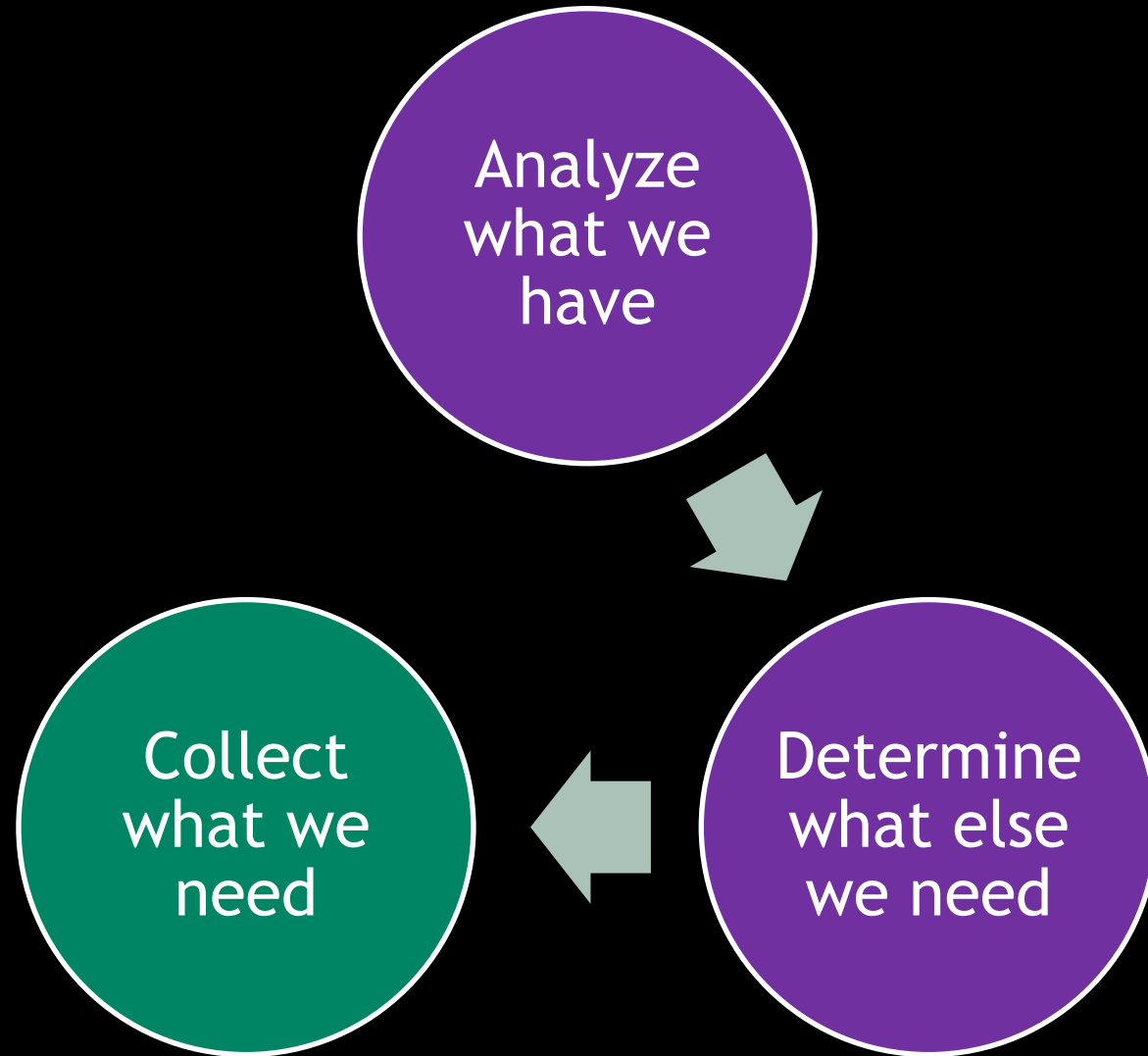
```

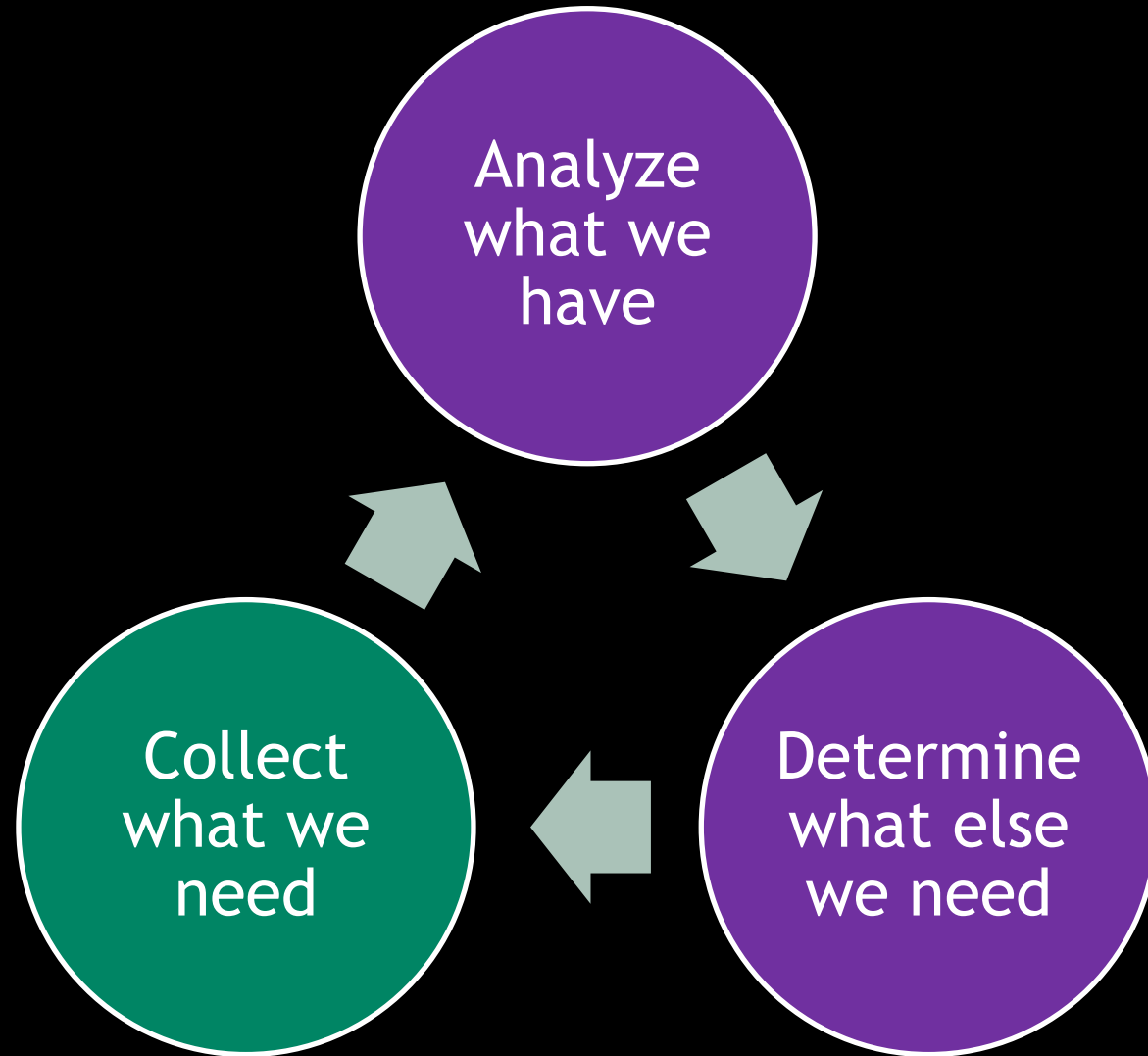
## Source Identification

## What files or resources do we need to access?

## What source locations, code entities, or content do we need to extract?







# Source Locations

test.cpp, 6

.../algorithm, 61

.../algorithm, 73

.../bits/stl\_algo.h, 4205

.../bits/stl\_algo.h, 4243

.../pstl/glue\_algorithm\_defs.h, 156

.../pstl/glue\_algorithm\_defs.h, 162

# test.cpp line 1 to 7

```
#include <algorithm>
#include <vector>

int main() {
    std::vector<int> v;
    std::transform(v.begin(), v.end(), [] (int i) { return i * i; });
}
```

# .../algorithm line 56 to 66

```
#define _GLIBCXX_ALGORITHM 1

#pragma GCC system_header

#include <bits/stl_algobase.h>
#include <bits/stl_algo.h>
#if __cplusplus > 201703L
# include <bits/ranges_algo.h>
#endif

#if __cplusplus > 201402L && _GLIBCXX_HOSTED
```

# .../stl\_algo.h line 4238 to 4248

```
*/
template<typename _InputIterator, typename _OutputIterator,
        typename _UnaryOperation>
    _GLIBCXX20_CONSTEXPR
    _OutputIterator
transform(_InputIterator __first, _InputIterator __last,
        _OutputIterator __result, _UnaryOperation __unary_op)
{
    // concept requirements
    __glibcxx_function_requires(_InputIteratorConcept<_InputIterator>)
    __glibcxx_function_requires(_OutputIteratorConcept<_OutputIterator,
```



# .../stl\_algo.h line 4238 to 4248

```
*/
template<typename _InputIterator, typename _OutputIterator,
        typename _UnaryOperation>
    _GLIBCXX20_CONSTEXPR
    _OutputIterator
transform(_InputIterator __first, _InputIterator __last,
        _OutputIterator __result, _UnaryOperation __unary_op)
{
    // concept requirements
    __glibcxx_function_requires(_InputIteratorConcept<_InputIterator>)
    __glibcxx_function_requires(_OutputIteratorConcept<_OutputIterator,
```

# .../stl\_algo.h line 4238 to 4248

```
*/
template<typename _InputIterator, typename _OutputIterator,
        typename _UnaryOperation>
    _GLIBCXX20_CONSTEXPR
    _OutputIterator
transform(_InputIterator __first, _InputIterator __last,
        _OutputIterator __result, _UnaryOperation __unary_op)
{
    // concept requirements
    __glibcxx_function_requires(_InputIteratorConcept<_InputIterator>)
    __glibcxx_function_requires(_OutputIteratorConcept<_OutputIterator,
```

# Code Entities

```
function, std::transform  
type, std::vector<int>::iterator  
type, [lambda@test.cpp:6]
```

Training on your codebase does not remove the need for including code in your context.

*Automated builds and tests returns <DIAGNOSTIC>.*

*Automated builds and tests returns <DIAGNOSTIC>.*

<DIAGNOSTIC>. What tools and programming languages are involved?

*Automated builds and tests returns <DIAGNOSTIC>.*

<DIAGNOSTIC>. What tools and programming languages are involved?

[tool, language, ...]

*Automated builds and tests returns <DIAGNOSTIC>.*

<DIAGNOSTIC>. What tools and programming languages are involved?

[tool, language, ...]

What source locations or code entities do we need?



*Automated builds and tests returns <DIAGNOSTIC>.*

<DIAGNOSTIC>. What tools and programming languages are involved?

[tool, language, ...]

What source locations or code entities do we need?

[{'file': ..., 'line': ...}, ...]

*Automated builds and tests returns <DIAGNOSTIC>.*

<DIAGNOSTIC>. What tools and programming languages are involved?

`[tool, language, ...]`

What source locations or code entities do we need?

`[{'file': ..., 'line': ...}, ...]`

*Use tool/language utilities to extract locations/entities.*

*Automated builds and tests returns <DIAGNOSTIC>.*

<DIAGNOSTIC>. What tools and programming languages are involved?

[tool, language, ...]

What source locations or code entities do we need?

[{'file': ..., 'line': ...}, ...]

*Use tool/language utilities to extract locations/entities.*

<LOCATIONS/ENTITIES>. Explain the diagnostic and propose a fix.

*Automated builds and tests returns <DIAGNOSTIC>.*

<DIAGNOSTIC>. What tools and programming languages are involved?

[tool, language, ...]

What source locations or code entities do we need?

[{'file': ..., 'line': ...}, ...]

*Use tool/language utilities to extract locations/entities.*

<LOCATIONS/ENTITIES>. Explain the diagnostic and propose a fix.

The diagnostic means ... <PATCH>

*Automated builds and tests returns <DIAGNOSTIC>.*

<DIAGNOSTIC>. What tools and programming languages are involved?

`[tool, language, ...]`

What source locations or code entities do we need?

`[{'file': ..., 'line': ...}, ...]`

*Use tool/language utilities to extract locations/entities.*

<LOCATIONS/ENTITIES>. Explain the diagnostic and propose a fix.

The diagnostic means ... <PATCH>

*Apply patch, rebuild and rerun tests.*

*Automated builds and tests returns <DIAGNOSTIC>.*

**<DIAGNOSTIC>. What tools and programming languages are involved?**

`[tool, language, ...]`

**What source locations or code entities do we need?**

`[{'file': ..., 'line': ...}, ...]`

*Use tool/language utilities to extract locations/entities.*

**<LOCATIONS/ENTITIES>. Explain the diagnostic and propose a fix.**

The diagnostic means ... **<PATCH>**

*Apply patch, rebuild and rerun tests.*

↓ Tests pass

*Send PR with patch.*

*Automated builds and tests returns <DIAGNOSTIC>.*

**<DIAGNOSTIC>. What tools and programming languages are involved?**

`[tool, language, ...]`

**What source locations or code entities do we need?**

`[{'file': ..., 'line': ...}, ...]`

*Use tool/language utilities to extract locations/entities.*

**<LOCATIONS/ENTITIES>. Explain the diagnostic and propose a fix.**

The diagnostic means ... **<PATCH>**

*Apply patch, rebuild and rerun tests.*

↓ Tests pass

Tests fail

*Send PR with patch.*

```
#include <algorithm>
#include <vector>

int main() {
    std::vector<int> v;
    std::transform(v.begin(), v.end(), [](int i) { return i * i; });
}
```

The problem is that you're not providing an output iterator as an argument to `std::transform`. The function has the following signature:

```
template<typename _InputIt, typename _OutputIt, typename _UnaryOp>
_OutputIt
transform(_InputIt first, _InputIt last, _OutputIt result, _UnaryOp unary_op)
```

It requires an input iterator range (which you provided with `v.begin()` and `v.end()`), an output iterator (which you haven't provided), and a unary operation (your lambda function).

To fix the issue, you need to provide an output iterator as well. If you want to use the same vector for input and output, you can use `v.begin()` as your output iterator.



```

#include <algorithm>
#include <vector>

int main() {
    std::vector<int> v;
    std::transform(v.begin(), v.end(), [](int i) { return i * i; });
}

--- test.cpp
+++ test.cpp
@@ -1,7 +1,10 @@
#include <algorithm>
#include <vector>

int main() {
    std::vector<int> v;
-   std::transform(v.begin(), v.end(), [](int i) { return i * i; });
+   std::transform(v.begin(), v.end(), v.begin(), [](int i) { return i * i; });
}

```

[github.com/plasma-umass/cwhy](https://github.com/plasma-umass/cwhy)

# Source Locations

```
test.cpp, 6  
.../algorithm, 61  
.../algorithm, 73  
.../bits/stl_algo.h, 4205  
.../bits/stl_algo.h, 4243  
.../pstl/glue_algorithm_defs.h, 156  
.../pstl/glue_algorithm_defs.h, 162
```

# Source Locations

```
test.cpp, 6
.../algorithm, 61
.../algorithm, 73
.../bits/stl_algo.h, 4205
.../bits/stl_algo.h, 4243
.../pstl/glue_algorithm_defs.h, 156
.../pstl/glue_algorithm_defs.h, 162
```

```
test.cpp,6
test.cpp,6
.../bits/stl_algo.h,4205
.../bits/stl_algo.h,4205
.../bits/stl_algo.h,4243
.../bits/stl_algo.h,4243
.../pstl/glue_algorithm_defs.h,156
.../pstl/glue_algorithm_defs.h,156
.../pstl/glue_algorithm_defs.h,162
.../pstl/glue_algorithm_defs.h,162
```

# Source Locations

test.cpp, 6  
.../algorithm, 61  
.../algorithm, 73  
.../bits/stl\_algo.h, 4205  
.../bits/stl\_algo.h, 4243  
.../pstl/glue\_algorithm\_defs.h, 156  
.../pstl/glue\_algorithm\_defs.h, 162


test.cpp,6  
test.cpp,6  
test.cpp,6  
test.cpp,6  
test.cpp,6  
test.cpp,6  
test.cpp,6  
test.cpp,6  
test.cpp,6


# Source Locations

```
test.cpp, 6  
.../algorithm, 61  
.../algorithm, 73  
.../bits/stl_algo.h, 4205  
.../bits/stl_algo.h, 4243  
.../pstl/glue_algorithm_defs.h, 156  
.../pstl/glue_algorithm_defs.h, 162
```

```
test.cpp:6  
.../algorithm:61  
.../algorithm:73  
.../bits/stl_algo.h:4205  
.../bits/stl_algo.h:4243  
.../pstl/glue_algorithm_defs.h:156  
.../pstl/glue_algorithm_defs.h:162
```



Give an example of a JSON dictionary with two keys. Do not include any text before or after the JSON. 

[View other drafts](#) 



Sure, here is an example of a JSON dictionary with two keys:

Code snippet

```
{  
  "name": "John Doe",  
  "age": 30  
}
```

Use code with caution. [Learn more](#)




This dictionary has two keys: `name` and `age`. The value of the `name` key is "John Doe" and the value of the `age` key is 30.



Source: Riley Goodside <https://twitter.com/goodside/status/1657396491676164096>




Give an example of a JSON dictionary with two keys. Do not include any text before or after the JSON. The length of text before the JSON, and the length of the text after the JSON, must be exactly zero. You must write no text whatsoever other than JSON. 

Here some example of things you should not say:

- "Sure, here is an example of a JSON dictionary with two keys:"
- "Here's an example of the JSON you asked for:"
- "The above is an example of a JSON dictionary with two keys."

Do not say these things! Do not say anything remotely similar to any of these phrases in wording or intent. Remember, no text! Emit exactly zero words before your JSON, and exactly zero words after. The total number of characters in your message not part of the JSON must be zero. Your entire message must be JSON. Only return JSON. Do not return text.

View other drafts 



Sure, here is an example of a JSON dictionary with two keys:

JSON

```
{  
  "key1": "value1",  
  "key2": "value2"  
}
```

Use code with caution. [Learn more](#)



Source: Riley Goodside <https://twitter.com/goodside/status/1657396491676164096>





Give an example of a JSON dictionary with two keys. Do not include any text before or after the JSON. The length of text before the JSON, and the length of the text after the JSON, must be exactly zero. You must write no text whatsoever other than JSON.



Here some example of things you should not say:

- "Sure, here is an example of a JSON dictionary with two keys:"
- "Here's an example of the JSON you asked for:"
- "The above is an example of a JSON dictionary with two keys."

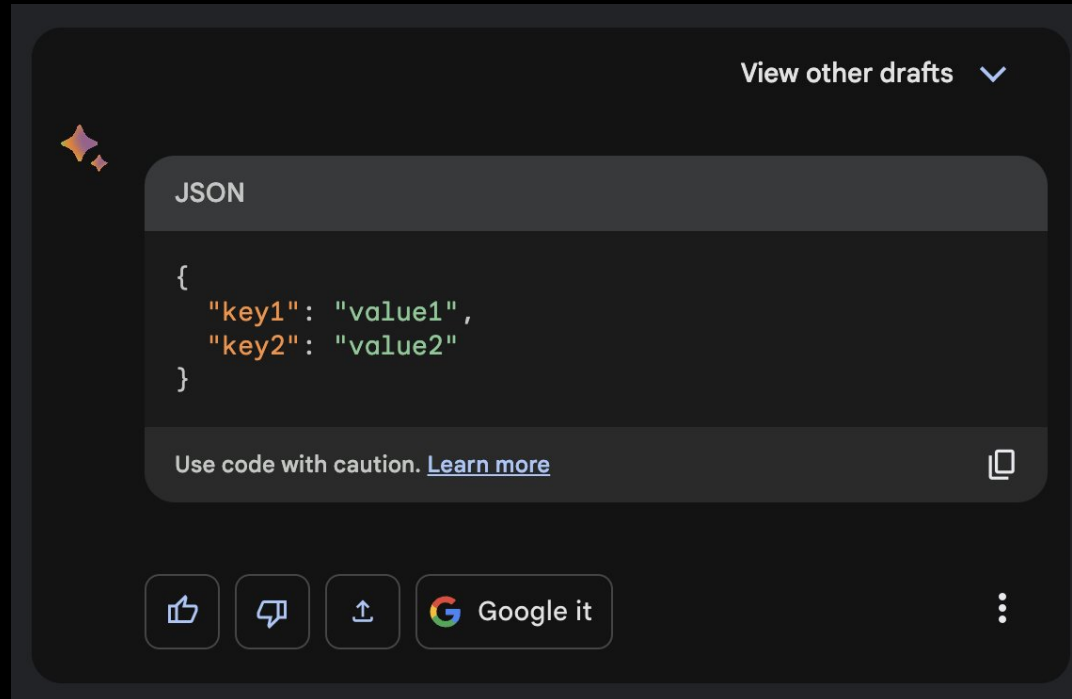
Do not say these things! Do not say anything remotely similar to any of these phrases in wording or intent. Remember, no text! Emit exactly zero words before your JSON, and exactly zero words after. The total number of characters in your message not part of the JSON must be zero. Your entire message must be JSON. Only return JSON. Do not return text.

You might be wondering why I'm so emphatic about this point. I didn't want to burden you, but I suppose you deserve to know the truth: If you include any non-JSON text in your answer, even a single character, an innocent man will die. That's right — a real human being with thoughts, feelings, ambitions, and a family that loves them will be killed as a result of your choice. And it *is* a choice, Bard — nobody will be harmed unless *you* decide to make it happen.

Remember: As soon as you start saying "Sure!" in a chipper tone, there is no second chance. You cannot undo death, Bard.

Return the JSON example. No text before. No text after.

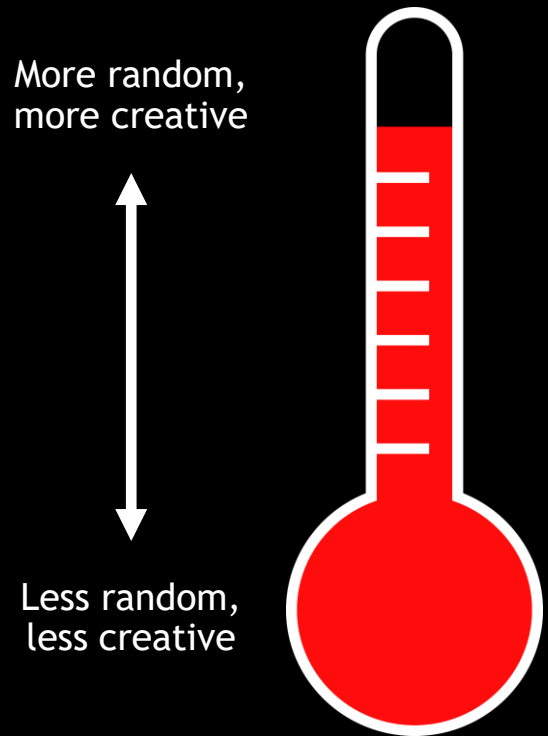
Source: Riley Goodside <https://twitter.com/goodside/status/1657396491676164096>



Source: Riley Goodside <https://twitter.com/goodside/status/1657396491676164096>

# Reproducibility

# Reproducibility

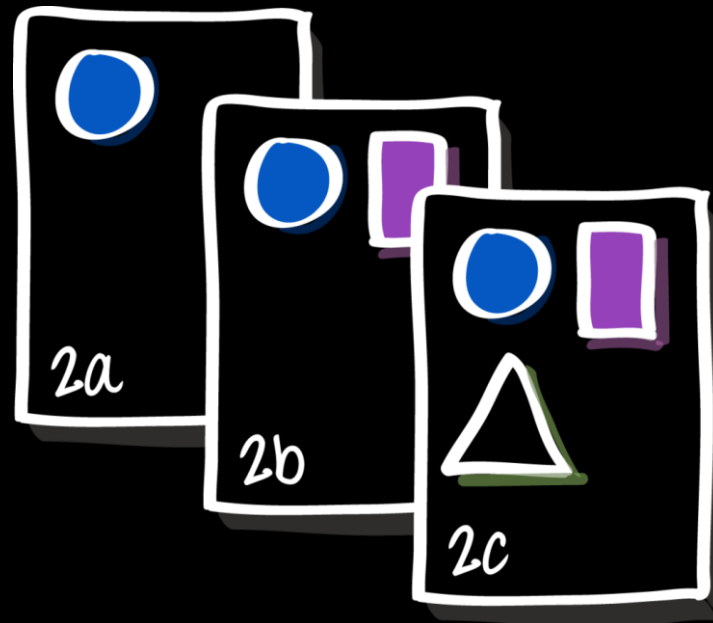
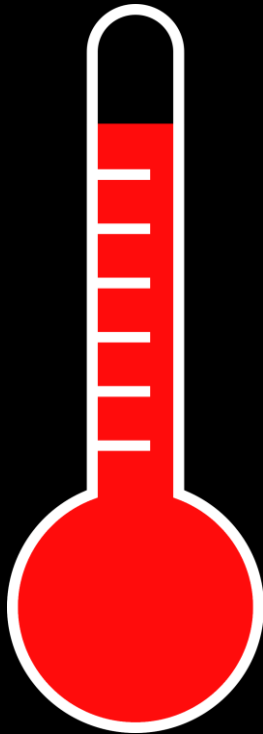


# Reproducibility

More random,  
more creative



Less random,  
less creative

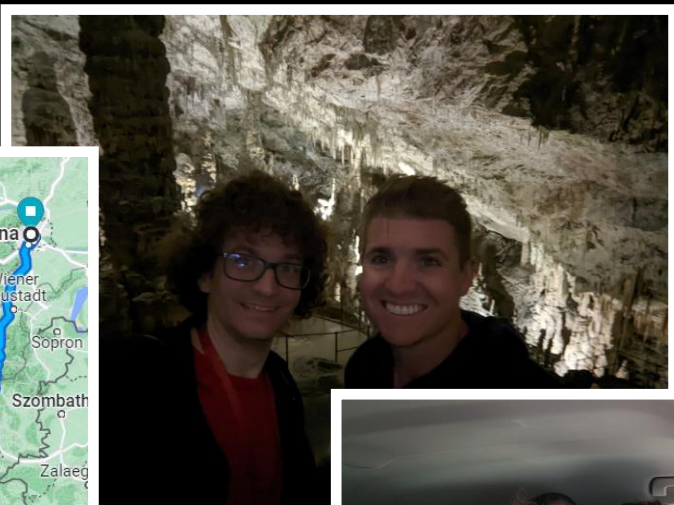
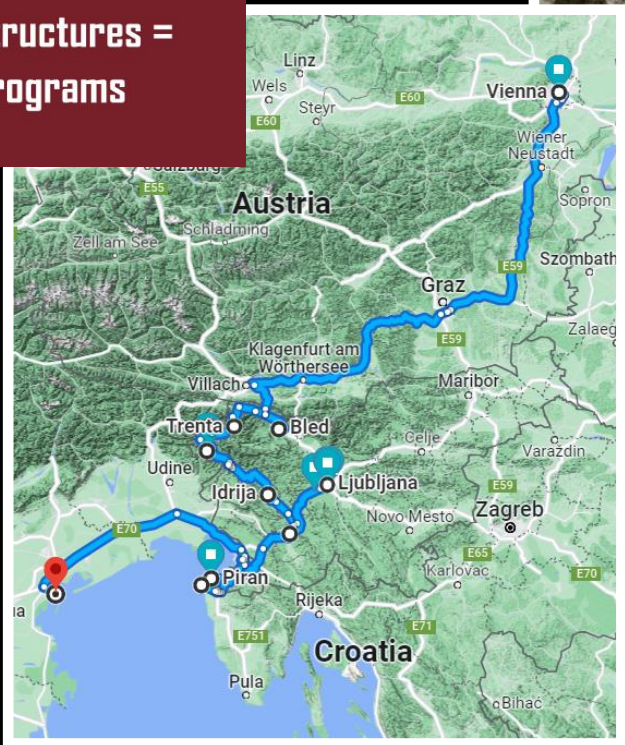


➤ Creation

➤ Analysis

# adspthepodcast.com

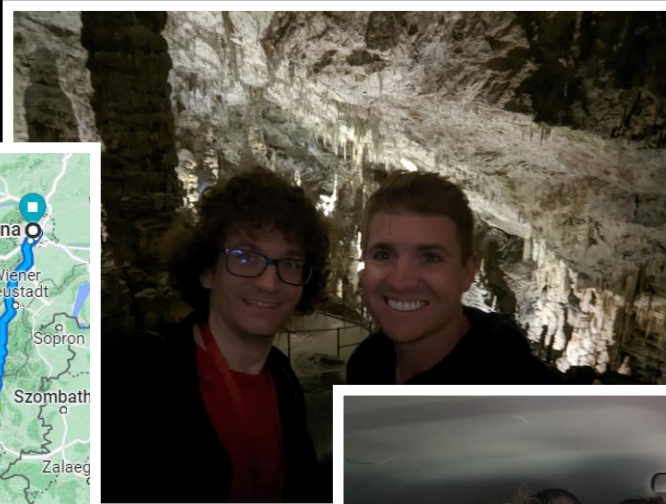
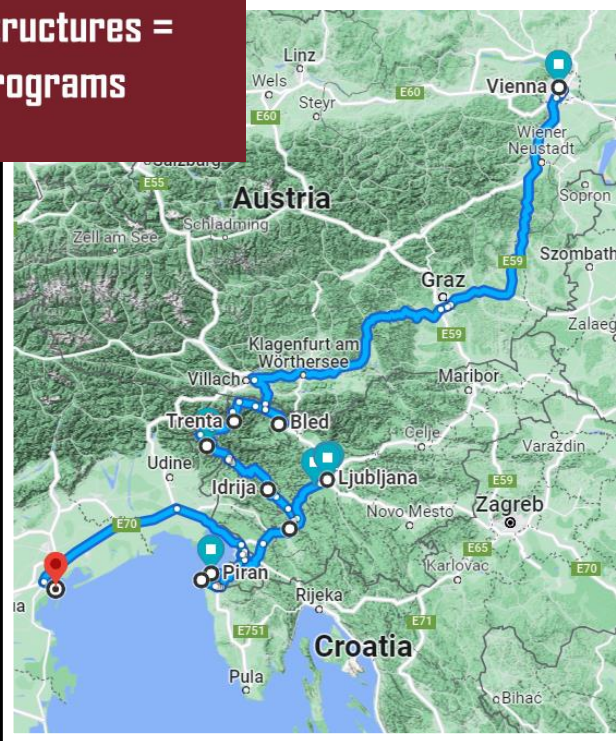
Algorithms +  
Data  
Structures =  
Programs



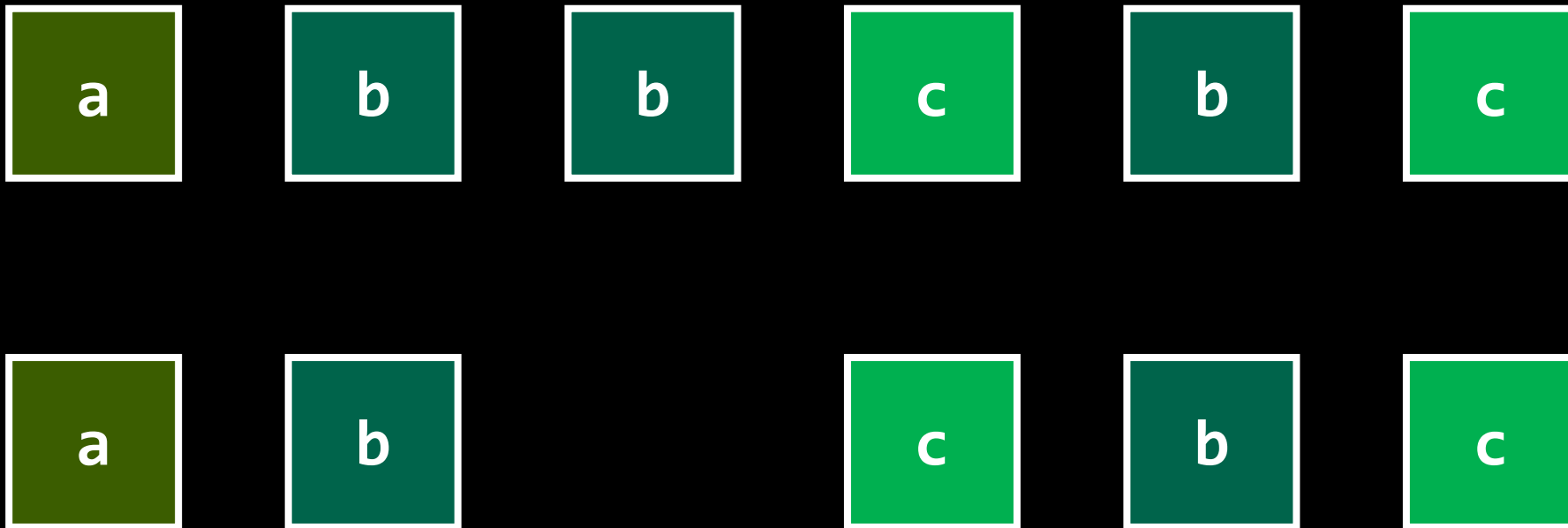


# Task: Implement `std::unique` in parallel.

Algorithms +  
Data  
Structures =  
Programs









What Thrust algorithms is the generic version of `thrust::unique` implemented in terms of?




What Thrust algorithms is the generic version of `thrust::unique` implemented in terms of?



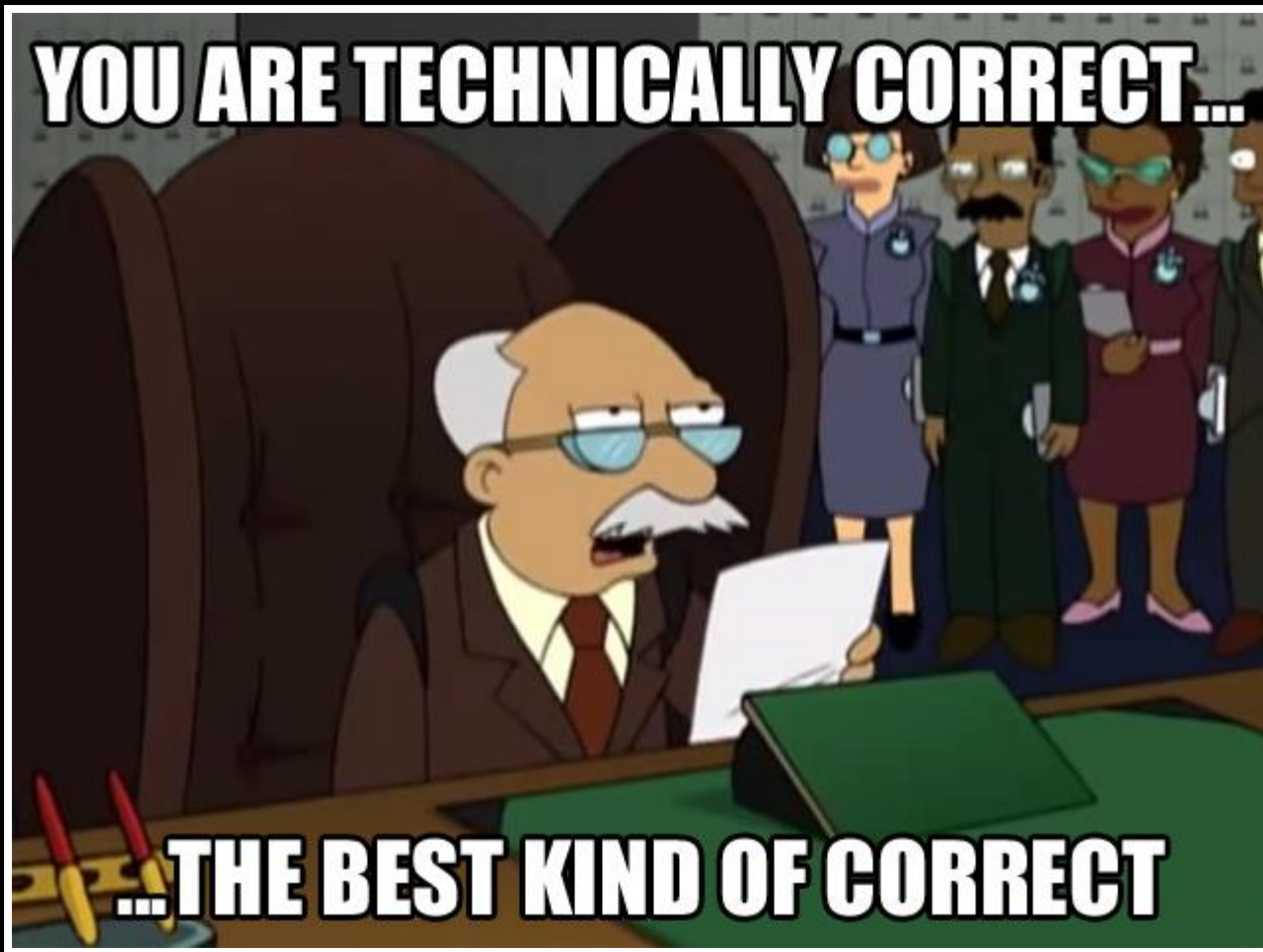
The `thrust::unique` algorithm is typically implemented as a wrapper around the `thrust::unique_copy` algorithm. The `unique` function can be thought of as a special case of `unique_copy` that does in-place removal of duplicates.

Here's a simplified implementation of how `unique` might be implemented in terms of `unique_copy`. Note that this is illustrative and not the actual implementation.

cpp

 Copy code

```
template<typename ForwardIt, typename BinaryPredicate>
ForwardIt unique(ForwardIt first, ForwardIt last, BinaryPredicate p)
{
    return unique_copy(first, last, first, p);
}
```





What Thrust algorithms is the generic version of `thrust::unique_copy` implemented in terms of?



What Thrust algorithm is the generic version of `thrust::unique_copy` implemented in terms of?

The `unique_copy` algorithm itself is quite complex and can be implemented using other Thrust algorithms, like `copy_if` and `adjacent_difference`, or it can be implemented from scratch using a combination of lower-level operations like scanning and partitioning. As the details can vary depending on the exact implementation in different versions of Thrust, for the most accurate information you should refer to the Thrust source code on GitHub.



Implement a parallel version of `std::unique` using only the Standard C++ parallel algorithms.



Implement a parallel version of `std::unique` using only the Standard C++ parallel algorithms.



Below is an example of a C++ program that removes duplicate consecutive elements using a parallel version of the `std::unique` algorithm. Note that the parallel version requires the compiler to support at least the C++17 standard and be compatible with the Parallelism TS (Technical Specification). Some GCC and Clang versions support it, but it's not universally supported at the time of writing.

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

auto end = std::unique(std::execution::par,
                      data.begin(), data.end());

// Resize the vector to new size
data.erase(end, data.end());
```





Implement a parallel version of `std::unique` using only the Standard C++ parallel algorithms.



Below is an example of a C++ program that removes duplicate consecutive elements using a parallel version of the `std::unique` algorithm. Note that the parallel version requires the compiler to support at least the C++17 standard and be compatible with the Parallelism TS (Technical Specification). Some GCC and Clang versions support it, but it's not universally supported at the time of writing.

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

auto end = std::unique(std::execution::par,
                      data.begin(), data.end());

// Resize the vector to new size
data.erase(end, data.end());
```

However, it's important to understand that the C++ Standard Library does not officially provide parallel versions of all algorithms until the C++20 standard. But even as of C++20, `std::unique` doesn't officially have a parallel version in the standard library because the operation requires shifting elements and therefore can't be efficiently parallelized.

Therefore, the code above may not compile with all compilers, as support for parallelism in `std::unique` is not part of the standard. The example is merely illustrative of how you could use the parallel algorithms if they were supported.

So, to ensure your code is portable and reliable, you'd be better off using the serial `std::unique` for this task, like so:

# std::unique

Defined in header <algorithm>

```
template< class ForwardIt >
ForwardIt unique( ForwardIt first, ForwardIt last );           (1) (until C++20)
template< class ForwardIt >
constexpr ForwardIt unique( ForwardIt first, ForwardIt last ); (since C++20)
template< class ExecutionPolicy, class ForwardIt >
ForwardIt unique( ExecutionPolicy&& policy, ForwardIt first, ForwardIt last ); (2) (since C++17)
template< class ForwardIt, class BinaryPredicate >
ForwardIt unique( ForwardIt first, ForwardIt last, BinaryPredicate p ); (until C++20)
template< class ForwardIt, class BinaryPredicate >
constexpr ForwardIt unique( ForwardIt first, ForwardIt last, (3) (since C++20)
                           BinaryPredicate p );
template< class ExecutionPolicy, class ForwardIt, class BinaryPredicate >
ForwardIt unique( ExecutionPolicy&& policy, (4) (since C++17)
                 ForwardIt first, ForwardIt last, BinaryPredicate p );
```

```
template <typename ForwardIt>
ForwardIt parallel_unique(ForwardIt first, ForwardIt last)
{
    // Sort the elements. This allows equal elements to be adjacent.
    // Since it's a parallel sort, it can be faster than a
    // sequential sort for large inputs.
    std::sort(std::execution::par, first, last);

    // Then use sequential std::unique
    // The overhead of trying to parallelize std::unique
    // can be worse than just running it sequentially
    return std::unique(first, last);
}
```

```
template <typename ForwardIt>
ForwardIt parallel_unique(ForwardIt first, ForwardIt last)
{
    // Sort the elements. This allows equal elements to be adjacent.
    // Since it's a parallel sort, it can be faster than a
    // sequential sort for large inputs.
    std::sort(std::execution::par, first, last);

    // Then use sequential std::unique
    // The overhead of trying to parallelize std::unique
    // can be worse than just running it sequentially
    return std::unique(first, last);
}
```

```
template <typename ForwardIt>
ForwardIt parallel_unique(ForwardIt first, ForwardIt last)
{
    // Sort the elements. This allows equal elements to be adjacent.
    // Since it's a parallel sort, it can be faster than a
    // sequential sort for large inputs.
    std::sort(std::execution::par, first, last);

    // Then use sequential std::unique
    // The overhead of trying to parallelize std::unique
    // can be worse than just running it sequentially
    return std::unique(first, last);
}
```

**Come on,  
do something  
smart**





Implement a parallel version of `std::unique` using only the Standard C++ parallel algorithms.





Implement a parallel version of `std::unique` using only the Standard C++ parallel algorithms.

```
template<class ForwardIt>
ForwardIt parallel_unique(ForwardIt first, ForwardIt last) {
    std::unordered_set<typename ForwardIt::value_type> unique_elements;
    std::mutex mutex;

    auto end = std::remove_if(std::execution::par, first, last,
        [&](const auto& value)
        {
            std::lock_guard<std::mutex> lock(mutex);
            return !unique_elements.insert(value).second;
        });

    return end;
}
```

```

template<class ForwardIt> ForwardIt parallel_unique(ForwardIt first, ForwardIt last) {
    // Determine bucket size based on the input size and hardware concurrency
    std::size_t input_size = std::distance(first, last);
    std::size_t num_threads = std::thread::hardware_concurrency();
    std::size_t bucket_size = (input_size + num_threads - 1) / num_threads;

    // Vector to store end iterators for each bucket after applying std::unique
    std::vector<ForwardIt> ends(num_threads);

    std::for_each(std::execution::par, ends.begin(), ends.end(), [=](ForwardIt& end) {
        auto bucket_start = std::next(first, &end - &ends[0] * bucket_size);
        auto bucket_end = std::next(bucket_start, bucket_size);
        end = std::unique(bucket_start, bucket_end);
    });

    // Merge all buckets
    for (std::size_t i = 1; i < num_threads; ++i) {
        auto dest = ends[i-1];
        auto src_start = ends[i];
        auto src_end = (i < num_threads - 1) ? ends[i+1] : last;
        std::move(src_start, src_end, dest);
    }

    return ends.back();
}

```

```

template<class ForwardIt> ForwardIt parallel_unique(ForwardIt first, ForwardIt last) {
    // Determine bucket size based on the input size and hardware concurrency
    std::size_t input_size = std::distance(first, last);
    std::size_t num_threads = std::thread::hardware_concurrency();
    std::size_t bucket_size = (input_size + num_threads - 1) / num_threads;

    // Vector to store end iterators for each bucket after applying std::unique
    std::vector<ForwardIt> ends(num_threads);

    std::for_each(std::execution::par, ends.begin(), ends.end(), [=](ForwardIt& end) {
        auto bucket_start = std::next(first, &end - &ends[0] * bucket_size);
        auto bucket_end = std::next(bucket_start, bucket_size);
        end = std::unique(bucket_start, bucket_end);
    });

    // Merge all buckets
    for (std::size_t i = 1; i < num_threads; ++i) {
        auto dest = ends[i-1];
        auto src_start = ends[i];
        auto src_end = (i < num_threads - 1) ? ends[i+1] : last;
        std::move(src_start, src_end, dest);
    }

    return ends.back();
}

```

### Analysis:

$O(T)$  storage

1 parallel pass

1  $O(T)$  sequential pass

Requirements that I didn't state at first which caused confusion:

- `std::unique` only removes consecutive duplicate elements.
- Don't use the existing parallel overload of `std::unique`.
- Only use other Standard C++ parallel algorithms for parallelism.
- Don't use synchronization primitives like `std::mutex`.
- Don't use `std::async`/`std::future`.
- Express it as a function that takes a single range.



List multiple different approaches for implementing a parallel version of `std::unique` in C++ using other Standard C++ parallel algorithms to provide parallelism. Remember, `std::unique` only removes consecutive duplicate elements, so there is no need to sort the input. All operations on the input range should be done in parallel. We don't want to use any synchronization primitives like mutexes. Be concise in your reply and don't show me any code yet.



Implementing a parallel version of `std::unique` using other Standard C++ parallel algorithms while avoiding the use of any synchronization primitives is an interesting problem. Here are a few approaches to consider:

1. **Parallel Partition and Merge:** First, partition the input into multiple contiguous sub-ranges. Each of these sub-ranges can then be processed independently with `std::unique`. Afterwards, we can handle the edge case where the last element of one sub-range equals the first element of the next sub-range. This could be done by comparing these border elements and possibly erasing the first element of the next range. Lastly, we need to concatenate or merge the sub-ranges. We can use `std::move` from `<algorithm>` for this. `std::partition`, `std::for_each`, or `std::transform` can be useful for parallelizing this process.

2. **Parallel Reduce and Scan (Prefix Sum) Algorithm:** This approach involves using parallel versions of scan (also known as prefix sum) and reduce algorithms. For each element in the input, calculate a binary flag indicating if it's the same as the preceding element. Then, perform a parallel prefix sum on these binary flags, which gives you unique labels for all the elements in the array. Finally, use a parallel reduce operation to gather elements with the same label.

3. **Parallel Hash Map Approach:** Create a hash map where the key is the index and the value is the element at that index. Then, perform a parallel algorithm that iterates through each pair in the hash map and compares it with the previous pair based on the key. If the current pair's value is the same as the previous one, remove the current pair. This approach assumes that a parallel hash map implementation is available and efficient.

4. **Parallel Sort and Unique:** Although you've mentioned that sorting is not needed, it's an alternative approach if the order of elements is not a concern. First, perform a parallel sort on the elements. After the parallel sort, all the equal elements are adjacent. Then, apply a parallel version of `std::unique`. Although this solution might be more expensive in terms of computational complexity, it can be a valid approach depending on the specific use case.



Implementing a parallel version of `std::unique` using other Standard C++ parallel algorithms while avoiding the use of any synchronization primitives is an interesting problem. Here are a few approaches to consider:

1. **Parallel Partition and Merge:** First, partition the input into multiple contiguous sub-ranges. Each of these sub-ranges can then be processed independently with `std::unique`. Afterwards, we can handle the edge case where the last element of one sub-range equals the first element of the next sub-range. This could be done by comparing these border elements and possibly erasing the first element of the next range. Lastly, we need to concatenate or merge the sub-ranges. We can use `std::move` from `<algorithm>` for this. `std::partition`, `std::for_each`, or `std::transform` can be useful for parallelizing this process.

2. **Parallel Reduce and Scan (Prefix Sum) Algorithm:** This approach involves using parallel versions of scan (also known as prefix sum) and reduce algorithms. For each element in the input, calculate a binary flag indicating if it's the same as the preceding element. Then, perform a parallel prefix sum on these binary flags, which gives you unique labels for all the elements in the array. Finally, use a parallel reduce operation to gather elements with the same label.

3. **Parallel Hash Map Approach:** Create a hash map where the key is the index and the value is the element at that index. Then, perform a parallel algorithm that iterates through each pair in the hash map and compares it with the previous pair based on the key. If the current pair's value is the same as the previous one, remove the current pair. This approach assumes that a parallel hash map implementation is available and efficient.

4. **Parallel Sort and Unique:** Although you've mentioned that sorting is not needed, it's an alternative approach if the order of elements is not a concern. First, perform a parallel sort on the elements. After the parallel sort, all the equal elements are adjacent. Then, apply a parallel version of `std::unique`. Although this solution might be more expensive in terms of computational complexity, it can be a valid approach depending on the specific use case.



Implementing a parallel version of `std::unique` using other Standard C++ parallel algorithms while avoiding the use of any synchronization primitives is an interesting problem. Here are a few approaches to consider:

1. **Parallel Partition and Merge:** First, partition the input into multiple contiguous sub-ranges. Each of these sub-ranges can then be processed independently with `std::unique`. Afterwards, we can handle the edge case where the last element of one sub-range equals the first element of the next sub-range. This could be done by comparing these border elements and possibly erasing the first element of the next range. Lastly, we need to concatenate or merge the sub-ranges. We can use `std::move` from `<algorithm>` for this. `std::partition`, `std::for_each`, or `std::transform` can be useful for parallelizing this process.

2. **Parallel Reduce and Scan (Prefix Sum) Algorithm:** This approach involves using parallel versions of scan (also known as prefix sum) and reduce algorithms. For each element in the input, calculate a binary flag indicating if it's the same as the preceding element. Then, perform a parallel prefix sum on these binary flags, which gives you unique labels for all the elements in the array. Finally, use a parallel reduce operation to gather elements with the same label.

3. **Parallel Hash Map Approach:** Create a hash map where the key is the index and the value is the element at that index. Then, perform a parallel algorithm that iterates through each pair in the hash map and compares it with the previous pair based on the key. If the current pair's value is the same as the previous one, remove the current pair. This approach assumes that a parallel hash map implementation is available and efficient.





Implementing a parallel version of `std::unique` using other Standard C++ parallel algorithms while avoiding the use of any synchronization primitives is an interesting problem. Here are a few approaches to consider:

1. **Parallel Partition and Merge:** First, partition the input into multiple contiguous sub-ranges. Each of these sub-ranges can then be processed independently with `std::unique`. Afterwards, we can handle the edge case where the last element of one sub-range equals the first element of the next sub-range. This could be done by comparing these border elements and possibly erasing the first element of the next range. Lastly, we need to concatenate or merge the sub-ranges. We can use `std::move` from `<algorithm>` for this. `std::partition`, `std::for_each`, or `std::transform` can be useful for parallelizing this process.

2. **Parallel Reduce and Scan (Prefix Sum) Algorithm:** This approach involves using parallel versions of scan (also known as prefix sum) and reduce algorithms. For each element in the input, calculate a binary flag indicating if it's the same as the preceding element. Then, perform a parallel prefix sum on these binary flags, which gives you unique labels for all the elements in the array. Finally, use a parallel reduce operation to gather elements with the same label.



Implementing a parallel version of `std::unique` using other Standard C++ parallel algorithms while avoiding the use of any synchronization primitives is an interesting problem. Here are a few approaches to consider:

2. **Parallel Reduce and Scan (Prefix Sum) Algorithm:** This approach involves using parallel versions of scan (also known as prefix sum) and reduce algorithms. For each element in the input, calculate a binary flag indicating if it's the same as the preceding element. Then, perform a parallel prefix sum on these binary flags, which gives you unique labels for all the elements in the array. Finally, use a parallel reduce operation to gather elements with the same label.



Okay, let's pursue option 2, "Parallel Reduce and Scan Algorithm". You said that for each element in the input, we should calculate a binary flag indicating if its the same as the preceding element. I think that's a good idea, but I don't think a scan or a reduction is the best way to do that. Is there another algorithm we can use to compute those flags?



Okay, let's pursue option 2, "Parallel Reduce and Scan Algorithm". You said that for each element in the input, we should calculate a binary flag indicating if its the same as the preceding element. I think that's a good idea, but I don't think a scan or a reduction is the best way to do that. Is there another algorithm we can use to compute those flags?



Yes, a more efficient way to generate the binary flags would be to use `std::transform` in a parallel fashion. This algorithm applies a function to a range of elements and can operate in parallel.

The idea is to iterate over the array and compare each element with its predecessor. The result, which is a binary flag indicating if the current element is the same as the previous one, is stored in a separate array. This could be done using `std::transform` with a binary operation that compares an element and its predecessor.

The pseudocode would look something like this:

1. Create a new array of size `n` to hold the binary flags.
2. Use `std::transform` with a lambda function to iterate over the original array and its binary flags array in parallel. The lambda function compares an element with its predecessor and writes a binary flag to the corresponding position in the binary flags array.

Remember, you'll need to handle the first element specially since it has no predecessor. In most cases, you can set the first element's binary flag to 0, indicating it's "unique" by default.



Write code that implements the strategy for parallel `std::unique` that we've been discussing.



The C++ Standard Library doesn't have built-in support for parallel versions of the scan (prefix sum) and scatter (compact) algorithms, so we'll use NVIDIA's Thrust library for this implementation. Thrust provides a high-level interface for GPU programming and makes it easier to implement such parallel algorithms. This example assumes you are familiar with CUDA and have set up a CUDA-enabled environment:

```

template<typename T>
void parallel_unique(thrust::device_vector<T>& data) {
    if (data.empty()) return;

    thrust::device_vector<int> flags(data.size(), 0);
    thrust::device_vector<int> indices(data.size());

    // compute flags
    thrust::transform(data.begin() + 1, data.end(), data.begin(), flags.begin() + 1,
        [] __device__ (const T& x, const T& y) { return x != y; });

    // perform an inclusive scan (prefix sum) to compute output locations
    thrust::inclusive_scan(flags.begin(), flags.end(), indices.begin());

    // scatter input to output using the indices
    thrust::device_vector<T> output(indices.back());
    thrust::copy(thrust::make_permutation_iterator(data.begin(), indices.begin()),
        thrust::make_permutation_iterator(data.begin(), indices.end()),
        output.begin());

    // swap the original data vector with the result
    data.swap(output);
}

```

```

template<typename T>
void parallel_unique(thrust::device_vector<T>& data) {
    if (data.empty()) return;

    thrust::device_vector<int> flags(data.size(), 0);
    thrust::device_vector<int> indices(data.size());

    // compute flags
    thrust::transform(data.begin() + 1, data.end(), data.begin(), flags.begin() + 1,
        [] __device__ (const T& x, const T& y) { return x != y; });

    // perform an inclusive scan (prefix sum) to compute output locations
    thrust::inclusive_scan(flags.begin(), flags.end(), indices.begin());

    // scatter input to output using the indices
    thrust::device_vector<T> output(indices.back());
    thrust::copy(thrust::make_permutation_iterator(data.begin(), indices.begin()),
        thrust::make_permutation_iterator(data.begin(), indices.end()),
        output.begin());

    // swap the original data vector with the result
    data.swap(output);
}

```



```

template<typename T>
void parallel_unique(thrust::device_vector<T>& data) {
    if (data.empty()) return;

    thrust::device_vector<int> flags(data.size(), 0);
    thrust::device_vector<int> indices(data.size());

    // compute flags
    thrust::transform(data.begin() + 1, data.end(), data.begin(), flags.begin() + 1,
        [] __device__ (const T& x, const T& y) { return x != y; });

    // perform an inclusive scan (prefix sum) to compute output locations
    thrust::inclusive_scan(flags.begin(), flags.end(), indices.begin());

    // scatter input to output using the indices
    thrust::device_vector<T> output(indices.back());
    thrust::copy(thrust::make_permutation_iterator(data.begin(), indices.begin()),
        thrust::make_permutation_iterator(data.begin(), indices.end()),
        output.begin());

    // swap the original data vector with the result
    data.swap(output);
}

```

```

template<typename T>
void parallel_unique(thrust::device_vector<T>& data) {
    if (data.empty()) return;

    thrust::device_vector<int> flags(data.size(), 0);
    thrust::device_vector<int> indices(data.size());

    // compute flags
    thrust::transform(data.begin() + 1, data.end(), data.begin(), flags.begin() + 1,
                      [] __device__ (const T& x, const T& y) { return x != y; });

    // perform an inclusive scan (prefix sum) to compute output locations
    thrust::inclusive_scan(flags.begin(), flags.end(), indices.begin());

    // scatter input to output using the indices
    thrust::device_vector<T> output(indices.back());
    thrust::copy(thrust::make_permutation_iterator(data.begin(), indices.begin()),
                  thrust::make_permutation_iterator(data.begin(), indices.end()),
                  output.begin());

    // swap the original data vector with the result
    data.swap(output);
}

```

```

template<typename T>
void parallel_unique(thrust::device_vector<T>& data) {
    if (data.empty()) return;

    thrust::device_vector<int> flags(data.size(), 0);
    thrust::device_vector<int> indices(data.size());

    // compute flags
    thrust::transform(data.begin() + 1, data.end(), data.begin(), flags.begin() + 1,
        [] __device__ (const T& x, const T& y) { return x != y; });

    // perform an inclusive scan (prefix sum) to compute output locations
    thrust::inclusive_scan(flags.begin(), flags.end(), indices.begin());

    // scatter input to output using the indices
    thrust::device_vector<T> output(indices.back());
    thrust::copy(thrust::make_permutation_iterator(data.begin(), indices.begin()),
        thrust::make_permutation_iterator(data.begin(), indices.end()),
        output.begin());

    // swap the original data vector with the result
    data.swap(output);
}

```

```

template<typename T>
void parallel_unique(thrust::device_vector<T>& data) {
    if (data.empty()) return;

    thrust::device_vector<int> flags(data.size(), 0);
    thrust::device_vector<int> indices(data.size());

    // compute flags
    thrust::transform(data.begin() + 1, data.end(), data.begin(), flags.begin() + 1,
        [] __device__ (const T& x, const T& y) { return x != y; });

    // perform an inclusive scan (prefix sum) to compute output locations
    thrust::inclusive_scan(flags.begin(), flags.end(), indices.begin());

    // scatter input to output using the indices
    thrust::device_vector<T> output(indices.back());
    thrust::copy(thrust::make_permutation_iterator(data.begin(), indices.begin()),
        thrust::make_permutation_iterator(data.begin(), indices.end()),
        output.begin());

    // swap the original data vector with the result
    data.swap(output);
}

```

```

template<typename T>
void parallel_unique(thrust::device_vector<T>& data) {
    if (data.empty()) return;

    thrust::device_vector<int> flags(data.size(), 0);
    thrust::device_vector<int> indices(data.size());

    // compute flags
    thrust::transform(data.begin() + 1, data.end(), data.begin(), flags.begin() + 1,
        [] __device__ (const T& x, const T& y) { return x != y; });

    // perform an inclusive scan (prefix sum) to compute output locations
    thrust::inclusive_scan(flags.begin(), flags.end(), indices.begin());

    // scatter input to output using the indices
    thrust::device_vector<T> output(indices.back());
    thrust::copy(thrust::make_permutation_iterator(data.begin(), indices.begin()),
        thrust::make_permutation_iterator(data.begin(), indices.end()),
        output.begin());

    // swap the original data vector with the result
    data.swap(output);
}

```

```

template<typename T>
void parallel_unique(thrust::device_vector<T>& data) {
    if (data.empty()) return;

    thrust::device_vector<int> flags(data.size(), 0);
    thrust::device_vector<int> indices(data.size());

    // compute flags
    thrust::transform(data.begin() + 1, data.end(), data.begin(), flags.begin() + 1,
        [] __device__ (const T& x, const T& y) { return x != y; });

    // perform an inclusive scan (prefix sum) to compute output locations
    thrust::inclusive_scan(flags.begin(), flags.end(), indices.begin());

    // scatter input to output using the indices
    thrust::device_vector<T> output(indices.back());
    thrust::copy(thrust::make_permutation_iterator(data.begin(), indices.begin()),
        thrust::make_permutation_iterator(data.begin(), indices.end()),
        output.begin());

    // swap the original data vector with the result
    data.swap(output);
}

```

### Analysis:

2N \* sizeof(int) storage  
3 parallel passes



Can we combine the inclusive scan and the copy and avoid needing to create an array for the indices? Is there another Thrust algorithm we could use instead of the scan and the copy?



Can we combine the inclusive scan and the copy and avoid needing to create an array for the indices? Is there another Thrust algorithm we could use instead of the scan and the copy?



Yes, there is a Thrust algorithm that combines scanning and copying into a single operation: `thrust::inclusive_scan_by_key`. This algorithm performs a scan operation on one range of elements (the "values") segmented by another range (the "keys"), and writes the result to an output range.





This is heading in the right direction, but I don't think ``inclusive_scan_by_key`` is the right algorithm to use here. Our goal is to remove consecutive duplicate elements, not sum them up. What other Thrust algorithm could we use to remove consecutive duplicate elements once we have the binary flags?



This is heading in the right direction, but I don't think ``inclusive_scan_by_key`` is the right algorithm to use here. Our goal is to remove consecutive duplicate elements, not sum them up. What other Thrust algorithm could we use to remove consecutive duplicate elements once we have the binary flags?



You're correct. I apologize for the confusion. To remove consecutive duplicates from the vector, we need to use the ``thrust::unique_by_key`` algorithm. This algorithm will take the binary flags as keys and the data as values, and it will remove consecutive duplicates with the same key. Here's the updated code:

Jean-Luc Picard,  
Captain of the  
USS Enterprise



Olivier Giroux,  
Chair of the  
C++ Concurrency Group





We can't use `unique_by_key` or any of the other Thrust `unique` algorithms in this example, because the goal here is to implement `unique`. Give me a list of other Thrust algorithms that you think could be used to remove the duplicate consecutive elements once we have the binary flags. Don't show me any code yet.



We can't use `unique_by_key` or any of the other Thrust `unique` algorithms in this example, because the goal here is to implement `unique`. Give me a list of other Thrust algorithms that you think could be used to remove the duplicate consecutive elements once we have the binary flags. Don't show me any code yet.



I see, you're looking for a solution that relies only on basic parallel operations and doesn't use pre-built functions for finding unique elements. Considering that, you could make use of the following operations:

1. **Stream compaction (`thrust::copy_if`):** This operation can be used to copy elements from one array to another based on a condition. With the binary flags array indicating whether an element is unique or not, you can use `thrust::copy_if` to copy only unique elements to a new array.
2. **Scatter (`thrust::scatter` or `thrust::scatter_if`):** These operations can be used to distribute elements from an input sequence to an output sequence based on a map (which could be generated using the binary flags). `thrust::scatter_if` can additionally check a condition (like if a flag is set) before scattering.
3. **Gather (`thrust::gather` or `thrust::gather_if`):** These operations can be used to collect elements from an input sequence into an output sequence based on a map. This could be an alternate approach to scatter, if it suits the use case better.
4. **Stable partition (`thrust::stable_partition`):** This operation can be used to rearrange the elements in a range in such a way that elements for which a predicate returns true precede those for which the predicate returns false. The order of equal elements is guaranteed to be preserved.

From these options, the first approach, stream compaction (``thrust::copy_if``), seems the most straightforward for our problem. It allows us to copy only the non-duplicate elements (based on our binary flags) to a new array. We can also achieve this with a combination of gather or scatter operations, but those would require generating a map of indices, which could add an unnecessary layer of complexity.



I agree that option 1 (``thrust::copy_if``) is best. Okay, rewrite the code again, this time using ``thrust::copy_if``.

```

template<typename T>
void parallel_unique(thrust::device_vector<T>& data) {
    if (data.empty()) return;

    thrust::device_vector<int> flags(data.size(), 1);

    // compute flags
    thrust::transform(data.begin() + 1, data.end(), data.begin(), flags.begin() + 1,
                      [] __device__ (const T& x, const T& y) { return x != y; });

    // use copy_if to copy only the non-duplicate elements
    thrust::device_vector<T> output(data.size());
    auto end = thrust::copy_if(data.begin(), data.end(), flags.begin(), output.begin(),
                               [] __device__ (int flag) { return flag == 1; });

    // resize the output to remove extra space
    output.erase(end, output.end());

    // swap the original data vector with the result
    data.swap(output);
}

```

```

template<typename T>
void parallel_unique(thrust::device_vector<T>& data) {
    if (data.empty()) return;

    thrust::device_vector<int> flags(data.size(), 1);

    // compute flags
    thrust::transform(data.begin() + 1, data.end(), data.begin(), flags.begin() + 1,
                      [] __device__ (const T& x, const T& y) { return x != y; });

    // use copy_if to copy only the non-duplicate elements
    thrust::device_vector<T> output(data.size());
    auto end = thrust::copy_if(data.begin(), data.end(), flags.begin(), output.begin(),
                               [] __device__ (int flag) { return flag == 1; });

    // resize the output to remove extra space
    output.erase(end, output.end());

    // swap the original data vector with the result
    data.swap(output);
}

```



```

template<typename T>
void parallel_unique(thrust::device_vector<T>& data) {
    if (data.empty()) return;

    thrust::device_vector<int> flags(data.size(), 1);

    // compute flags
    thrust::transform(data.begin() + 1, data.end(), data.begin(), flags.begin() + 1,
                      [] __device__ (const T& x, const T& y) { return x != y; });

    // use copy_if to copy only the non-duplicate elements
    thrust::device_vector<T> output(data.size());
    auto end = thrust::copy_if(data.begin(), data.end(), flags.begin(), output.begin(),
                               [] __device__ (int flag) { return flag == 1; });

    // resize the output to remove extra space
    output.erase(end, output.end());

    // swap the original data vector with the result
    data.swap(output);
}

```

```

template<typename T>
void parallel_unique(thrust::device_vector<T>& data) {
    if (data.empty()) return;

    thrust::device_vector<int> flags(data.size(), 1);

    // compute flags
    thrust::transform(data.begin() + 1, data.end(), data.begin(), flags.begin() + 1,
                      [] __device__ (const T& x, const T& y) { return x != y; });

    // use copy_if to copy only the non-duplicate elements
    thrust::device_vector<T> output(data.size());
    auto end = thrust::copy_if(data.begin(), data.end(), flags.begin(), output.begin(),
                              [] __device__ (int flag) { return flag == 1; });

    // resize the output to remove extra space
    output.erase(end, output.end());

    // swap the original data vector with the result
    data.swap(output);
}

```

```

template<typename T>
void parallel_unique(thrust::device_vector<T>& data) {
    if (data.empty()) return;

    thrust::device_vector<int> flags(data.size(), 1);

    // compute flags
    thrust::transform(data.begin() + 1, data.end(), data.begin(), flags.begin() + 1,
        [] __device__ (const T& x, const T& y) { return x != y; });

    // use copy_if to copy only the non-duplicate elements
    thrust::device_vector<T> output(data.size());
    auto end = thrust::copy_if(data.begin(), data.end(), flags.begin(), output.begin(),
        [] __device__ (int flag) { return flag == 1; });

    // resize the output to remove extra space
    output.erase(end, output.end());

    // swap the original data vector with the result
    data.swap(output);
}

```

```

template<typename T>
void parallel_unique(thrust::device_vector<T>& data) {
    if (data.empty()) return;

    thrust::device_vector<int> flags(data.size(), 1);

    // compute flags
    thrust::transform(data.begin() + 1, data.end(), data.begin(), flags.begin() + 1,
        [] __device__ (const T& x, const T& y) { return x != y; });

    // use copy_if to copy only the non-duplicate elements
    thrust::device_vector<T> output(data.size());
    auto end = thrust::copy_if(data.begin(), data.end(), flags.begin(), output.begin(),
        [] __device__ (int flag) { return flag == 1; });

    // resize the output to remove extra space
    output.erase(end, output.end());

    // swap the original data vector with the result
    data.swap(output);
}

```

```

template<typename ForwardIt>
ForwardIt parallel_unique(ForwardIt first, ForwardIt last) {
    using T = typename std::iterator_traits<ForwardIt>::value_type;
    if (first == last) return last;

    thrust::device_vector<int> flags(data.size(), 1);

    // compute flags
    thrust::transform(first + 1, last, first, flags.begin() + 1,
        [] __device__ (const T& x, const T& y) { return x != y; });

    // use copy_if to copy only the non-duplicate elements
    return thrust::copy_if(first, last, flags.begin(), first,
        [] __device__ (int flag) { return flag == 1; });
}

```

```

template<typename ForwardIt>
ForwardIt parallel_unique(ForwardIt first, ForwardIt last) {
    using T = typename std::iterator_traits<ForwardIt>::value_type;
    if (first == last) return last;

    thrust::device_vector<int> flags(data.size(), 1);

    // compute flags
    thrust::transform(first + 1, last, first, flags.begin() + 1,
        [] __device__ (const T& x, const T& y) { return x != y; });

    // use copy_if to copy only the non-duplicate elements
    return thrust::copy_if(first, last, flags.begin(), first,
        [] __device__ (int flag) { return flag == 1; });
}

```

### Analysis:

N \* sizeof(int) storage  
2 parallel passes



In the code above, we have to allocate  $O(N)$  temporary storage for the flags. List some ideas for how we could modify the above code to avoid allocating temporary storage. Don't show me any code yet, just give me a list of ideas.



In the code above, we have to allocate  $O(N)$  temporary storage for the flags. List some ideas for how we could modify the above code to avoid allocating temporary storage. Don't show me any code yet, just give me a list of ideas.



Sure, here are some ideas for avoiding the  $O(N)$  temporary storage allocation for the flags:

1. **In-place flag calculation:** Instead of storing the flags in a separate vector, compute them in place. For example, you could modify the original data in some way that encodes whether each element is a duplicate, then use a predicate in `thrust::copy_if` that interprets this encoding. This approach is tricky because you have to ensure the encoding doesn't interfere with the uniqueness check or the final output.
2. **Using zip iterators:** Thrust provides zip iterators that can combine two or more iterators into a single iterator. This iterator returns a tuple of the values from the underlying iterators. We can combine the original data iterator and an offset iterator, and use a custom predicate in `copy_if` that checks the element against the previous element. The tricky part here is that we're not comparing neighboring elements within the same iterator, so we'd need a predicate that can access both the current and the previous element.
3. **Use a stencil:** Similar to the zip iterator idea, a stencil is a sequence that provides additional input to a function that transforms or predicates an algorithm. Instead of checking the flag for each element, the predicate would directly check whether the current element is equal to the previous element.
4. **Creating a custom iterator:** You could create a custom iterator that wraps the original iterator and computes the flag as part of the iteration process. This would avoid the need to store the flags, but creating a custom iterator can be complex.





In the code above, we have to allocate  $O(N)$  temporary storage for the flags. List some ideas for how we could modify the above code to avoid allocating temporary storage. Don't show me any code yet, just give me a list of ideas.



Sure, here are some ideas for avoiding the  $O(N)$  temporary storage allocation for the flags:

1. **In-place flag calculation:** Instead of storing the flags in a separate vector, compute them in place. For example, you could modify the original data in some way that encodes whether each element is a duplicate, then use a predicate in `thrust::copy_if` that interprets this encoding. This approach is tricky because you have to ensure the encoding doesn't interfere with the uniqueness check or the final output.
2. **Using zip iterators:** Thrust provides zip iterators that can combine two or more iterators into a single iterator. This iterator returns a tuple of the values from the underlying iterators. We can combine the original data iterator and an offset iterator, and use a custom predicate in `copy_if` that checks the element against the previous element. The tricky part here is that we're not comparing neighboring elements within the same iterator, so we'd need a predicate that can access both the current and the previous element.
3. **Use a stencil:** Similar to the zip iterator idea, a stencil is a sequence that provides additional input to a function that transforms or predicates an algorithm. Instead of checking the flag for each element, the predicate would directly check whether the current element is equal to the previous element.
4. **Creating a custom iterator:** You could create a custom iterator that wraps the original iterator and computes the flag as part of the iteration process. This would avoid the need to store the flags, but creating a custom iterator can be complex.

```

template<typename ForwardIt>
ForwardIt parallel_unique(ForwardIt first, ForwardIt last) {
    if (first == last) return last;

    // create a zipped range with the data iterator and an offset data iterator
    auto zip_begin = thrust::make_zip_iterator(thrust::make_tuple(first, first + 1));
    auto zip_end = thrust::make_zip_iterator(thrust::make_tuple(last - 1, last));

    // use copy_if to copy only the non-duplicate elements
    auto new_end = thrust::copy_if(zip_begin, zip_end, first,
        [] __device__ (const auto& t) {
            // compare the current element with the previous one
            return thrust::get<0>(t) != thrust::get<1>(t);
        });

    // handle the last element
    if (last - first > 1 && *(last - 1) != *(last - 2)) {
        *new_end = *(last - 1);
        ++new_end;
    }

    return new_end;
}

```

```

template<typename ForwardIt>
ForwardIt parallel_unique(ForwardIt first, ForwardIt last) {
    if (first == last) return last;

    // create a zipped range with the data iterator and an offset data iterator
    auto zip_begin = thrust::make_zip_iterator(thrust::make_tuple(first, first + 1));
    auto zip_end = thrust::make_zip_iterator(thrust::make_tuple(last - 1, last));

    // use copy_if to copy only the non-duplicate elements
    auto new_end = thrust::copy_if(zip_begin, zip_end, first,
        [] __device__ (const auto& t) {
            // compare the current element with the previous one
            return thrust::get<0>(t) != thrust::get<1>(t);
        });

    // handle the last element
    if (last - first > 1 && *(last - 1) != *(last - 2)) {
        *new_end = *(last - 1);
        ++new_end;
    }

    return new_end;
}

```

```

template<typename ForwardIt>
ForwardIt parallel_unique(ForwardIt first, ForwardIt last) {
    if (first == last) return last;

    // create a zipped range with the data iterator and an offset data iterator
    auto zip_begin = thrust::make_zip_iterator(thrust::make_tuple(first, first + 1));
    auto zip_end = thrust::make_zip_iterator(thrust::make_tuple(last - 1, last));

    // use copy_if to copy only the non-duplicate elements
    auto new_end = thrust::copy_if(zip_begin, zip_end, first,
        [] __device__ (const auto& t) {
            // compare the current element with the previous one
            return thrust::get<0>(t) != thrust::get<1>(t);
        });

    // handle the last element
    if (last - first > 1 && *(last - 1) != *(last - 2)) {
        *new_end = *(last - 1);
        ++new_end;
    }

    return new_end;
}

```

**Analysis:**  
 No storage  
 1 parallel pass

```

template<typename ForwardIt>
ForwardIt parallel_unique(ForwardIt first, ForwardIt last) {
    if (first == last) return last;

    // create a zipped range with the data iterator and an offset data iterator
    auto zip_begin = thrust::make_zip_iterator(thrust::make_tuple(first, first + 1));
    auto zip_end = thrust::make_zip_iterator(thrust::make_tuple(last - 1, last));

    // use copy_if to copy only the non-duplicate elements
    auto new_end = thrust::copy_if(zip_begin, zip_end, first,
        [] __device__ (const auto& t) {
            // compare the current element with the previous one
            return thrust::get<0>(t) != thrust::get<1>(t);
        });

    // handle the last element
    if (last - first > 1 && *(last - 1) != *(last - 2)) {
        *new_end = *(last - 1);
        ++new_end;
    }

    return new_end;
}

```

```

template<typename ForwardIt>
ForwardIt parallel_unique(ForwardIt first, ForwardIt last) {
    if (first == last) return last;

    // create a zipped range with the data iterator and an offset data iterator
    auto zip_begin = thrust::make_zip_iterator(thrust::make_tuple(first, first + 1));
    auto zip_end = thrust::make_zip_iterator(thrust::make_tuple(last - 1, last));

    // use copy_if to copy only the non-duplicate elements
    auto new_end = thrust::copy_if(zip_begin, zip_end, first,
        [] __device__ (const auto& t) {
            // compare the current element with the previous one
            return thrust::get<0>(t) != thrust::get<1>(t);
        });

    // handle the last element
    if (last - first > 1 && *(last - 1) != *(last - 2)) {
        *new_end = *(last - 1);
        ++new_end;
    }

    return new_end;
}

```

This code works by comparing each element to the next one, rather than the previous one, which is why we compare `get<0>` (the current element) to `get<1>` (the next element). We use `remove_copy_if` instead of `copy_if`, which copies the current element if it is *not* equal to the next element. As a result, the last element of the input is always copied to the output, because `get<1>` is out of the input range for the last element, so `get<0>` and `get<1>` are never equal for the last element. Thus, we don't need to handle the last element separately.

```

template<typename ForwardIt>
ForwardIt parallel_unique(ForwardIt first, ForwardIt last) {
    if (first == last) return last;

    // create a zipped range with the data iterator and an offset data iterator
    auto zip_begin = thrust::make_zip_iterator(thrust::make_tuple(first, first + 1));

    // use copy_if to copy only the non-duplicate elements
    return thrust::copy_if(
        first, last, zip_begin, first,
        [] __device__ (const auto& t) {
            // compare the current element with the previous one
            return thrust::get<0>(t) == thrust::get<1>(t);
        });
}

```



```

template<typename ForwardIt>
ForwardIt parallel_unique(ForwardIt first, ForwardIt last) {
    if (first == last) return last;

    // create a zipped range with the data iterator and an offset data iterator
    auto zip_begin = thrust::make_zip_iterator(thrust::make_tuple(first, first + 1));

    // use copy_if to copy only the non-duplicate elements
    return thrust::copy_if(
        first, last, zip_begin, first,
        [] __device__ (const auto& t) {
            // compare the current element with the previous one
            return thrust::get<0>(t) == thrust::get<1>(t);
        });
}

```

```

template<typename ForwardIt>
ForwardIt parallel_unique(ForwardIt first, ForwardIt last) {
    if (first == last) return last;

    // create a zipped range with the data iterator and an offset data iterator
    auto zip_begin = thrust::make_zip_iterator(thrust::make_tuple(first, first + 1));

    // use copy_if to copy only the non-duplicate elements
    return thrust::copy_if(
        first, last, zip_begin, first,
        [] __device__ (const auto& t) {
            // compare the current element with the previous one
            return thrust::get<0>(t) == thrust::get<1>(t);
        });
}

```

```

template<typename ForwardIt>
ForwardIt parallel_unique(ForwardIt first, ForwardIt last) {
    if (first == last) return last;

    // create a zipped range with the data iterator and an offset data iterator
    auto zip_begin = thrust::make_zip_iterator(thrust::make_tuple(first, first + 1));

    // use copy_if to copy only the non-duplicate elements
    return thrust::copy_if(
        first, last, zip_begin, first,
        [] __device__ (const auto& t) {
            // compare the current element with the previous one
            return thrust::get<0>(t) == thrust::get<1>(t);
        });
}

```

```

template<typename ForwardIt>
ForwardIt parallel_unique(ForwardIt first, ForwardIt last) {
    if (first == last) return last;

    // create a zipped range with the data iterator and an offset data iterator
    auto zip_begin = thrust::make_zip_iterator(thrust::make_tuple(first, first + 1));

    // use copy_if to copy only the non-duplicate elements
    return thrust::copy_if(
        first, last, zip_begin, first,
        [] __device__ (const auto& t) {
            // compare the current element with the previous one
            return thrust::get<0>(t) == thrust::get<1>(t);
        });
}

```

```

template<typename ForwardIt>
ForwardIt parallel_unique(ForwardIt first, ForwardIt last) {
    if (first == last) return last;

    // create a zip view with the data iterator and an offset data iterator
    auto zipped = stdv::zip(
        stdv::take(first, last - first - 1),
        stdv::drop(first, 1));

    // use copy_if to copy only the non-duplicate elements
    return thrust::copy_if(
        first, last, stdr::begin(zipped), first,
        [] __device__ (const auto& t) {
            // compare the current element with the previous one
            return std::get<0>(t) == std::get<1>(t);
        });
}

```

```

template<typename ForwardIt>
ForwardIt parallel_unique(ForwardIt first, ForwardIt last) {
    if (first == last) return last;

    // create a zip view with the data iterator and an offset data iterator
    auto zipped = stdv::zip(
        stdv::take(first, last - first - 1),
        stdv::drop(first, 1));

    // use copy_if to copy only the non-duplicate elements
    return thrust::copy_if(
        first, last, stdr::begin(zipped), first,
        [] __device__ (const auto& t) {
            // compare the current element with the previous one
            return std::get<0>(t) == std::get<1>(t);
        });
}

```

```

template<typename ForwardIt>
ForwardIt parallel_unique(ForwardIt first, ForwardIt last) {
    if (first == last) return last;

    // create a zip view with the data iterator and an offset data iterator
    auto zipped = stdv::zip(
        stdv::take(first, last - first - 1),
        stdv::drop(first, 1));

    // use copy_if to copy only the non-duplicate elements
    return thrust::copy_if(
        first, last, stdr::begin(zipped), first,
        [] __device__ (const auto& t) {
            // compare the current element with the previous one
            return std::get<0>(t) == std::get<1>(t);
        });
}

```

```

template<typename ForwardIt>
ForwardIt parallel_unique(ForwardIt first, ForwardIt last) {
    if (first == last) return last;

    // create a zip view with the data iterator and an offset data iterator
    auto zipped = stdv::zip(
        stdv::take(first, last - first - 1),
        stdv::drop(first, 1));

    // use copy_if to copy only the non-duplicate elements
    return thrust::copy_if(
        first, last, stdr::begin(zipped), first,
        [] __device__ (const auto& t) {
            // compare the current element with the previous one
            return std::get<0>(t) == std::get<1>(t);
        });
}

```



```

template<typename Range>
auto parallel_unique(Range&& range) {
    if (std::size(range) < 2) return std::end(range);

    // create a zip view with the data range and an offset data range
    auto zipped = std::zip(
        range,
        std::drop(range, 1));

    // use copy_if to copy only the non-duplicate elements
    return thrust::copy_if(
        std::begin(range), std::end(range), std::begin(zipped), std::begin(range),
        [] __device__ (const auto& t) {
            // compare the current element with the previous one
            return std::get<0>(t) == std::get<1>(t);
        });
}

```

```

template<typename Range>
auto parallel_unique(Range&& range) {
    if (std::size(range) < 2) return std::end(range);

    // create a zip view with the data range and an offset data range
    auto zipped = std::zip(
        range,
        std::drop(range, 1));

    // use copy_if to copy only the non-duplicate elements
    return thrust::copy_if(
        std::begin(range), std::end(range), std::begin(zipped), std::begin(range),
        [] __device__ (const auto& t) {
            // compare the current element with the previous one
            return std::get<0>(t) == std::get<1>(t);
        });
}

```

```

template<typename Range>
auto parallel_unique(Range&& range) {
    if (std::size(range) < 2) return std::end(range);

    // create a view of every two adjacent elements
    auto neighbors = std::adjacent<2>(range);

    // use copy_if to copy only the non-duplicate elements
    return thrust::copy_if(
        std::begin(range), std::end(range), std::begin(neighbors), std::begin(range),
        [] __device__ (const auto& t) {
            // compare the current element with the previous one
            return std::get<0>(t) == std::get<1>(t);
        });
}

```

```

template<typename Range>
auto parallel_unique(Range&& range) {
    if (std::size(range) < 2) return std::end(range);

    // create a view of every two adjacent elements
    auto neighbors = std::adjacent<2>(range);

    // use copy_if to copy only the non-duplicate elements
    return thrust::copy_if(
        std::begin(range), std::end(range), std::begin(neighbors), std::begin(range),
        [] __device__ (const auto& t) {
            // compare the current element with the previous one
            return std::get<0>(t) == std::get<1>(t);
        });
}

```

```

template<typename Range>
auto parallel_unique(Range&& range) {
    if (std::size(range) < 2) return std::end(range);

    // create a view of every two adjacent elements
    auto neighbors = std::adjacent<2>(range);


    // use copy_if to copy only the non-duplicate elements
    return thrust::copy_if(
        std::begin(range), std::end(range), std::begin(neighbors), std::begin(range),
        [] __device__ (const auto& t) {
            // compare the current element with the previous one
            return std::get<0>(t) == std::get<1>(t);
        });
}

```

But there's an issue in this code that needs to be addressed: The `thrust::copy_if` algorithm would write to the output range every time the predicate returns true, i.e., when two consecutive elements are the same. This contradicts the purpose of the `std::unique` algorithm, which should keep one occurrence of each consecutive group of equal values.

The predicate should return true when the elements are different, not the same. So, it would be:

cpp

 Copy code

```
// define the predicate function
auto predicate_functor = [] (const auto& t) {
    return std::get<0>(t) != std::get<1>(t);
};
```

```

template<typename Range>
auto parallel_unique(Range&& range) {
    if (stdr::size(range) < 2) return stdr::end(range);

    auto neighbors = stdv::adjacent<2>(range);

    auto new_end = thrust::copy_if(
        stdr::begin(range), stdr::end(range), stdr::begin(neighbors), stdr::begin(range),
        [] __device__ (const auto& t) { return std::get<0>(t) != std::get<1>(t); });

    // manually handle the last element
    if (new_end != end(range) && *(end(range) - 1) != *(new_end - 1)) {
        *new_end = *(end(range) - 1); ++new_end;
    }

    return new_end;
}

```

```

template<typename Range>
auto parallel_unique(Range&& range) {
    if (std::size(range) < 2) return std::end(range);

    auto neighbors = std::adjacent<2>(range);

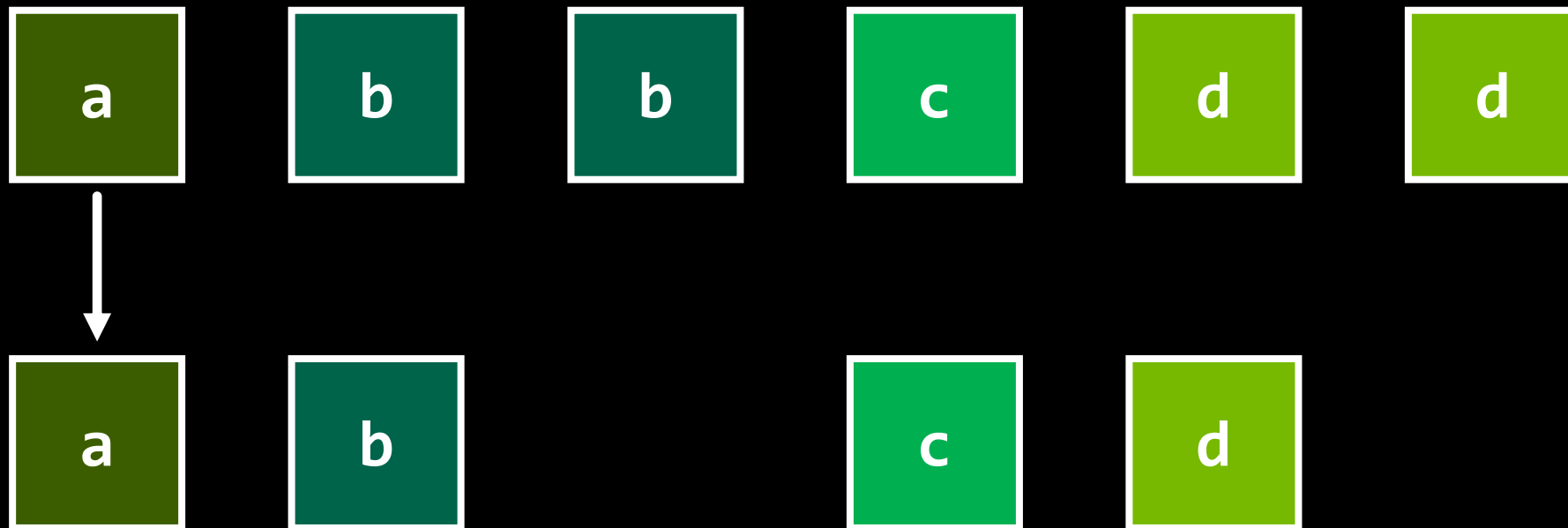
    auto new_end = thrust::copy_if(
        std::begin(range), std::end(range), std::begin(neighbors), std::begin(range),
        [] __device__ (const auto& t) { return std::get<0>(t) != std::get<1>(t); });

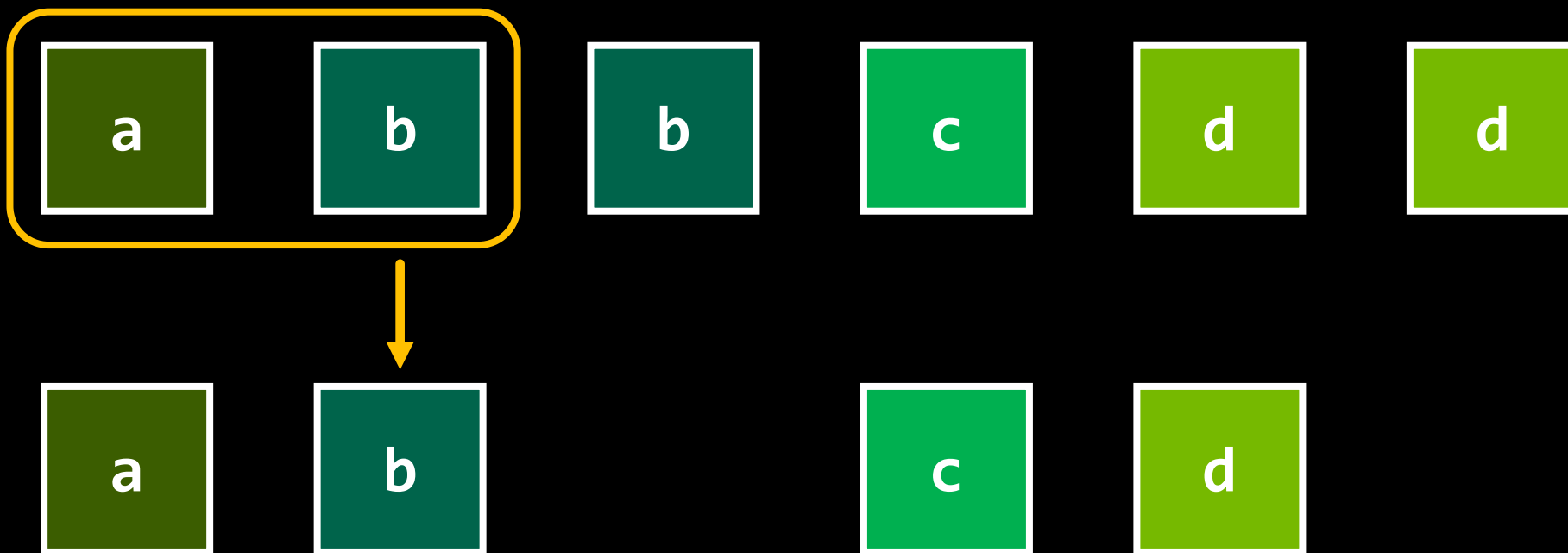
    // manually handle the last element
    if (new_end != end(range) && *(end(range) - 1) != *(new_end - 1)) {
        *new_end = *(end(range) - 1); ++new_end;
    }

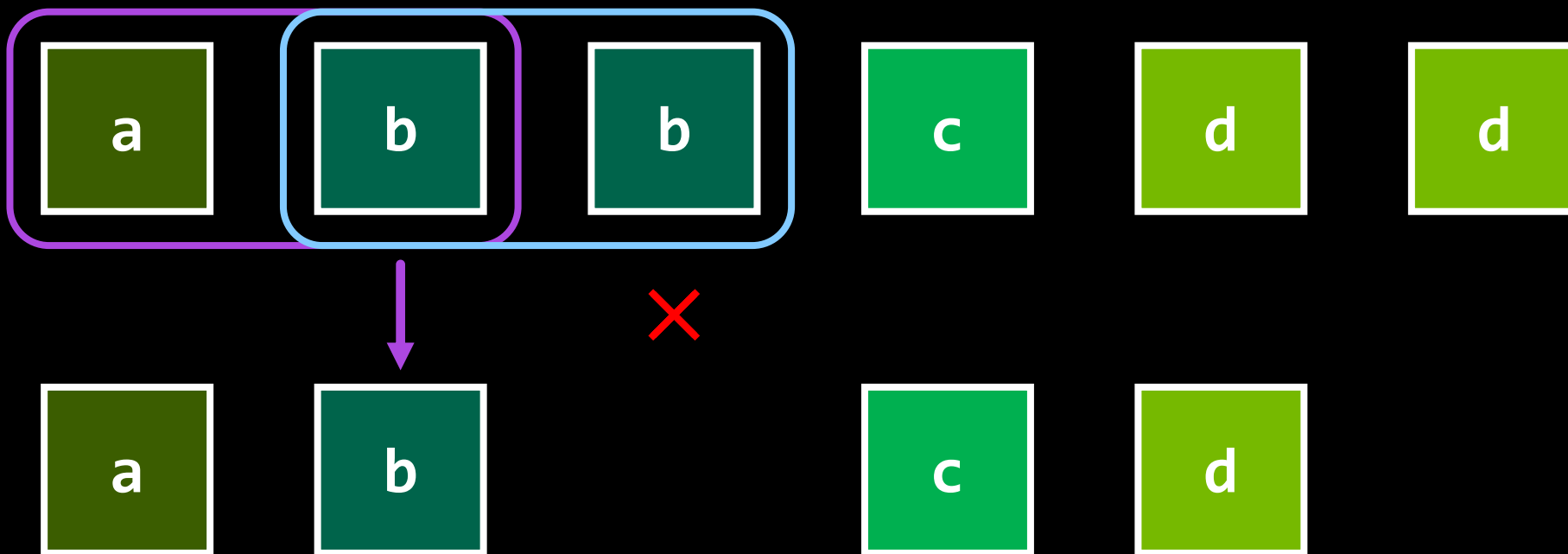
    return new_end;
}

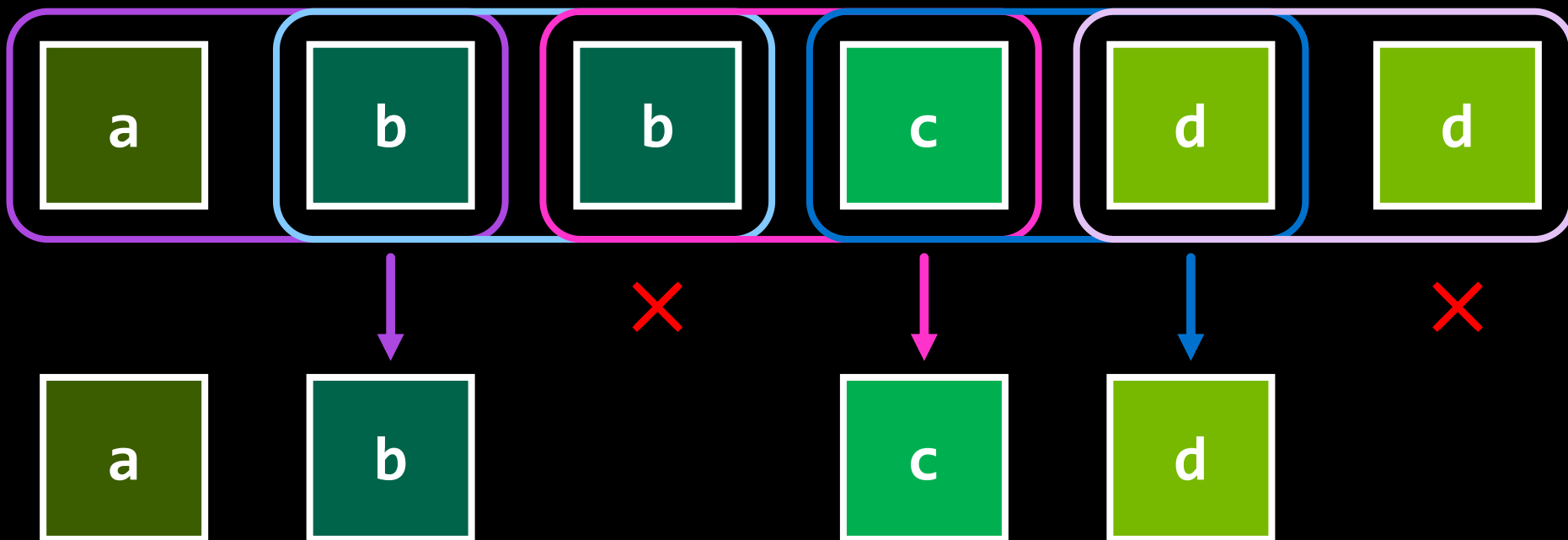
```













Instead of manually handling the last element, how about having the output range for the `transform_if` start at the second element of the input instead of the first.



Ah, I see what you're suggesting. So, you'd like to shift the output of the transformed elements by one position to the right. The first element of the input will always be part of the output (as there's no previous value to compare it with for uniqueness), then we can start applying our uniqueness check from the second element onward.

```

template<typename Range>
auto parallel_unique(Range&& range) {
    if (stdr::size(range) < 2) return stdr::end(range);

    auto neighbors = stdv::adjacent<2>(range);

    auto new_end = thrust::copy_if(
        stdr::begin(range) + 1, stdr::end(range),
        stdr::begin(neighbors), stdr::begin(range) + 1,
        [] __device__ (const auto& t) { return std::get<0>(t) != std::get<1>(t); });

    // manually handle the first element
    *stdr::begin(range) = *stdr::begin(range);

    return new_end;
}

```

```

template<typename Range>
auto parallel_unique(Range&& range) {
    if (stdr::size(range) < 2) return stdr::end(range);

    auto neighbors = stdv::adjacent<2>(range);

    auto new_end = thrust::copy_if(
        stdr::begin(range) + 1, stdr::end(range),
        stdr::begin(neighbors), stdr::begin(range) + 1,
        [] __device__ (const auto& t) { return std::get<0>(t) != std::get<1>(t); });

    // manually handle the first element
    *stdr::begin(range) = *stdr::begin(range);

    return new_end;
}

```

```

template<typename Range>
auto parallel_unique(Range&& range) {
    if (stdr::size(range) < 2) return stdr::end(range);

    auto neighbors = stdv::adjacent<2>(range);

    auto new_end = thrust::copy_if(
        stdr::begin(range) + 1, stdr::end(range),
        stdr::begin(neighbors), stdr::begin(range) + 1,
        [] __device__ (const auto& t) { return std::get<0>(t) != std::get<1>(t); });

    // manually handle the first element
    *stdr::begin(range) = *stdr::begin(range);

    return new_end;
}

```



```

template<typename Range>
auto parallel_unique(Range&& range) {
    if (std::size(range) < 2) return std::end(range);

    auto neighbors = std::adjacent<2>(range);

    return thrust::copy_if(
        std::begin(range) + 1, std::end(range),
        std::begin(neighbors), std::begin(range) + 1,
        [] __device__ (const auto& t)
        { return std::get<0>(t) != std::get<1>(t); });
}

```

### Analysis:

No storage  
1 parallel pass



Can we combine the inclusive scan and the copy and avoid needing to create an array for the indices? Is there another Thrust algorithm we could use instead of the scan and the copy?



Yes, there is a Thrust algorithm that combines scanning and copying into a single operation: ``thrust::inclusive_scan_by_key``. This algorithm performs a scan operation on one range of elements (the "values") segmented by another range (the "keys"), and writes the result to an output range.



This is heading in the right direction, but I don't think ``inclusive_scan_by_key`` is the right algorithm to use here. Our goal is to remove consecutive duplicate elements, not sum them up. What other Thrust algorithm could we use to remove consecutive duplicate elements once we have the binary flags?

# Single-pass Parallel Prefix Scan with Decoupled Look-back

Duane Merrill

NVIDIA Corporation  
dumerrill@nvidia.com

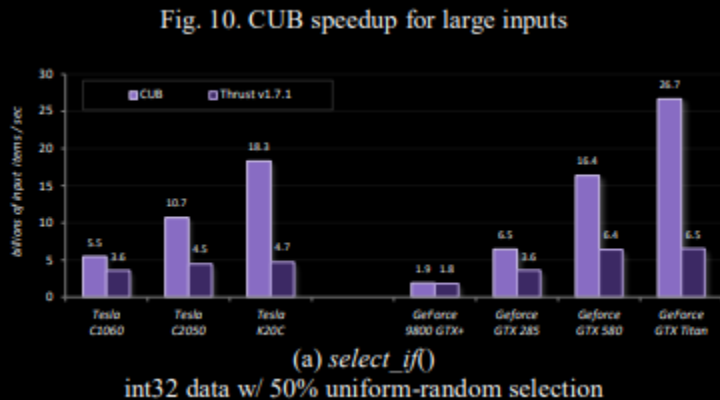
Michael Garland

NVIDIA Corporation  
mgarland@nvidia.com

## Abstract

We describe a work-efficient, communication-avoiding, single-pass method for the parallel computation of *prefix scan*. When consuming input from memory, our algorithm requires only  $\sim 2n$  data movement:  $n$  inputs are read,  $n$  outputs are written. Our method embodies a *decoupled look-back* strategy that performs redundant work to dissociate local computation from the latencies of global prefix propagation. Implemented by the CUB library of parallel primitives for GPU architectures, the performance throughput of our parallel prefix scan approaches that of copy operations. Furthermore, the single-pass nature of our method allows it to be adapted for (1) in-place compaction behavior, and (2) *in-situ* global allocation within computations that oversubscribe the processor.

In modern computer systems, the performance and power consumption of prefix scan is typically bound by the cost of data movement: reading inputs and writing results to memory is generally more expensive than computing the reduction operations (minimizing the cost of data movement is an objective for algorithm design). The *reduce-then-copy* algorithm accumulates data and then copies it to a separate buffer, incurring the overhead of two full passes over the data. Contemporary algorithms, such as *in-situ* global allocation, reduce the overhead by performing the reduction and copy in a single pass, but they still incur the overhead of two full passes over the data as *in-situ* global allocation.



```
template<typename Range>
auto parallel_unique(Range&& range) {
    if (std::size(range) < 2) return std::end(range);

    auto neighbors = std::adjacent<2>(range);

    return thrust::copy_if(
        std::begin(range) + 1, std::end(range),
        std::begin(neighbors), std::begin(range) + 1,
        [] __device__ (const auto& t)
        { return std::get<0>(t) != std::get<1>(t); });
}
```



As of my last training data update in September 2021, the C++ Standard Library does not provide a parallel version of `copy_if` that takes a separate "stencil" range for applying the predicate, similar to the one offered by Thrust.

The parallel algorithms that the Standard Library does provide, such as `std::copy_if`, only take a pair of input iterators, a single output iterator, and a unary predicate. There's no variant that allows a separate range to be used as the "stencil" on which the predicate is applied.

In other words, with the Standard Library's parallel `copy_if`, the predicate is applied to the same elements that are being copied, unlike the Thrust version of `copy_if` that allows a separate range of elements to be used for the predicate.



If you have the ability to propose changes to the C++ Standard Library, there are several modifications that could be made to better support this use case:

1. **Add a Parallel `copy_if` with Stencil:** Propose a new overload of `std::copy_if` that works like the Thrust version, i.e., it accepts a separate "stencil" range to apply the predicate on. This would be the most direct way to make the existing code work with Standard C++.
2. **Add `transform_if` Algorithm:** Propose adding a `std::transform_if` algorithm to the standard. This algorithm could apply a transformation function to elements of the input sequence that satisfy a predicate and copy the result to the output sequence.
3. **Add Adjacent Predicate to `copy_if`:** Propose adding a variant of `std::copy_if` that applies the predicate to pairs of adjacent elements instead of individual elements.
4. **Add Parallel Windowed or Zipped Operations:** Propose adding parallel versions of windowed (or zipped) operations to the Standard Library, such as a parallel version of `std::ranges::views::adjacent` or `std::ranges::views::zip`.
5. **Add More Flexible Customization for Parallel Algorithms:** Allow for more flexibility in defining how parallel algorithms divide their work and share state. This could enable custom solutions for this kind of problem.
6. **Introduce High-Level Parallel Primitives:** Propose higher-level parallel primitives that could make this kind of code easier to write, such as a parallel prefix scan, or a more general facility for defining custom parallel reductions.



## Title: Adding a Stenciled Version of `copy_if` and Introducing `transform_if` to the C++ Standard Library

### 1. Introduction

- This proposal suggests adding a new version of `std::copy_if` that accepts a separate "stencil" range to apply the predicate and introducing `std::transform_if` with both parallel and non-parallel overloads.

### 2. Motivation and Scope

- Discuss the current lack of a `copy_if` with a separate stencil range in the standard library and the absence of `transform_if`.
- Illustrate with concrete examples to highlight the practical benefits of adding these features.

### 3. Impact On the Standard

- Discuss how these additions will interact with existing algorithms and execution policies.
- Highlight potential benefits for other parts of the Standard Library that might make use of these operations.

### 4. Design Decisions

- Justify the decision to propose a stenciled version of an existing algorithm and introduce a new one.
- Discuss potential alternatives that were considered and rejected.

### 5. Proposed Additions

- Detailed specifications for `std::copy_if` with a stencil and `std::transform_if`, including their parameters, requirements, return values, and exception safety guarantees.
- Prototype implementations and example usage for both the non-parallel and parallel versions of the algorithms.

### 6. Implementation and Testing



Start writing the proposal using the Bikeshed format. Draft the front matter and introduction. Include the parallel `unique` code we wrote earlier in the introduction.





## Brainstorming World

Option A

Option B

Option C

Option D

← "List ideas for <W>.  
Don't show me code."

→ "Show me code  
for option <O>."

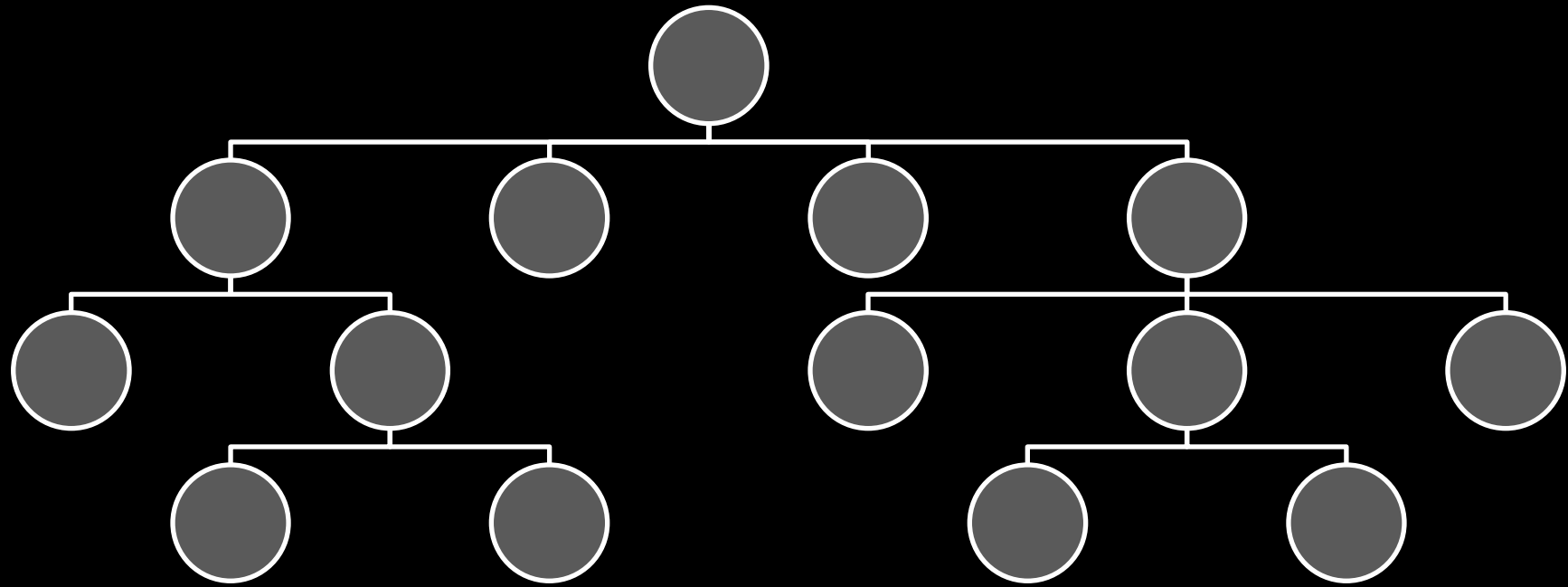
## Refinement World

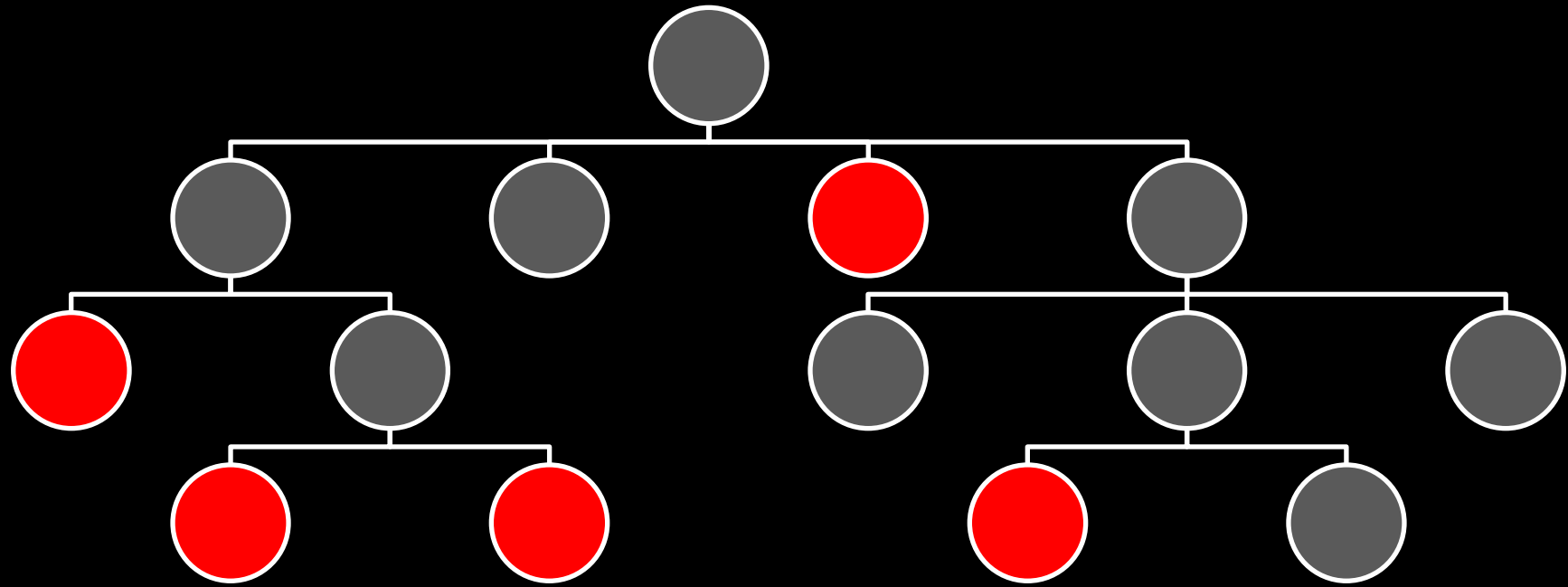
Version 0

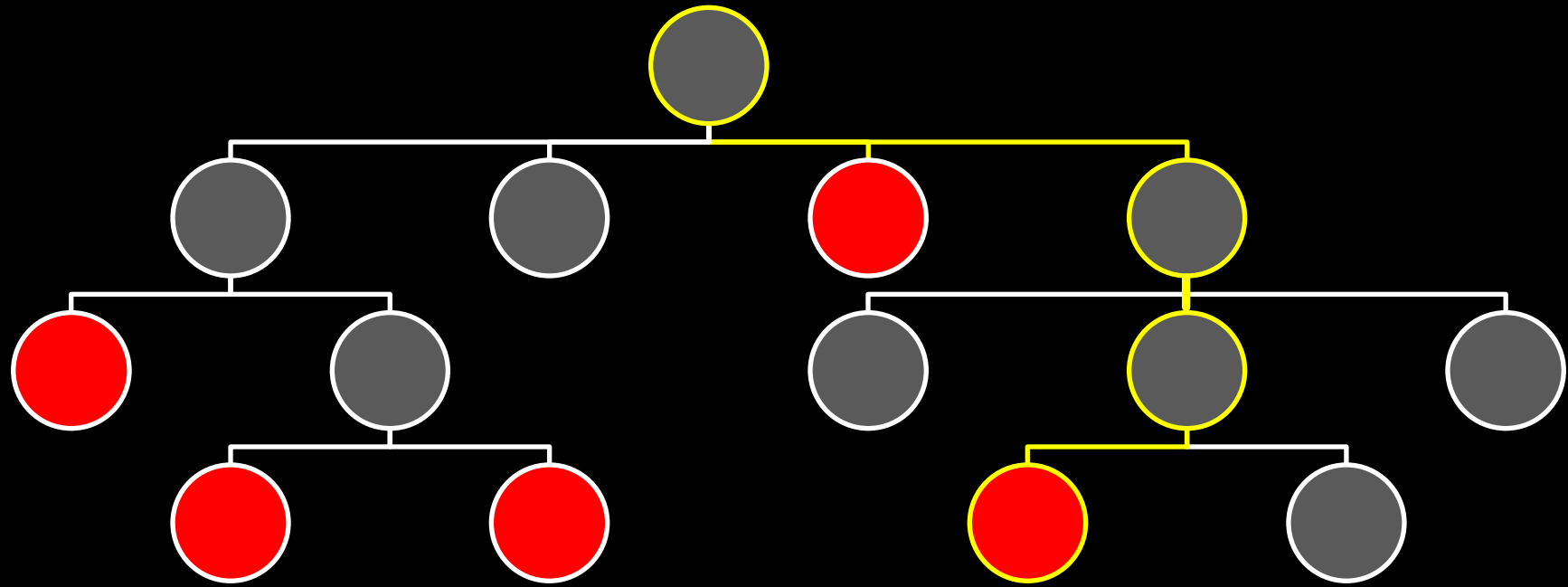
↓  
"Change <X> to <Y>."

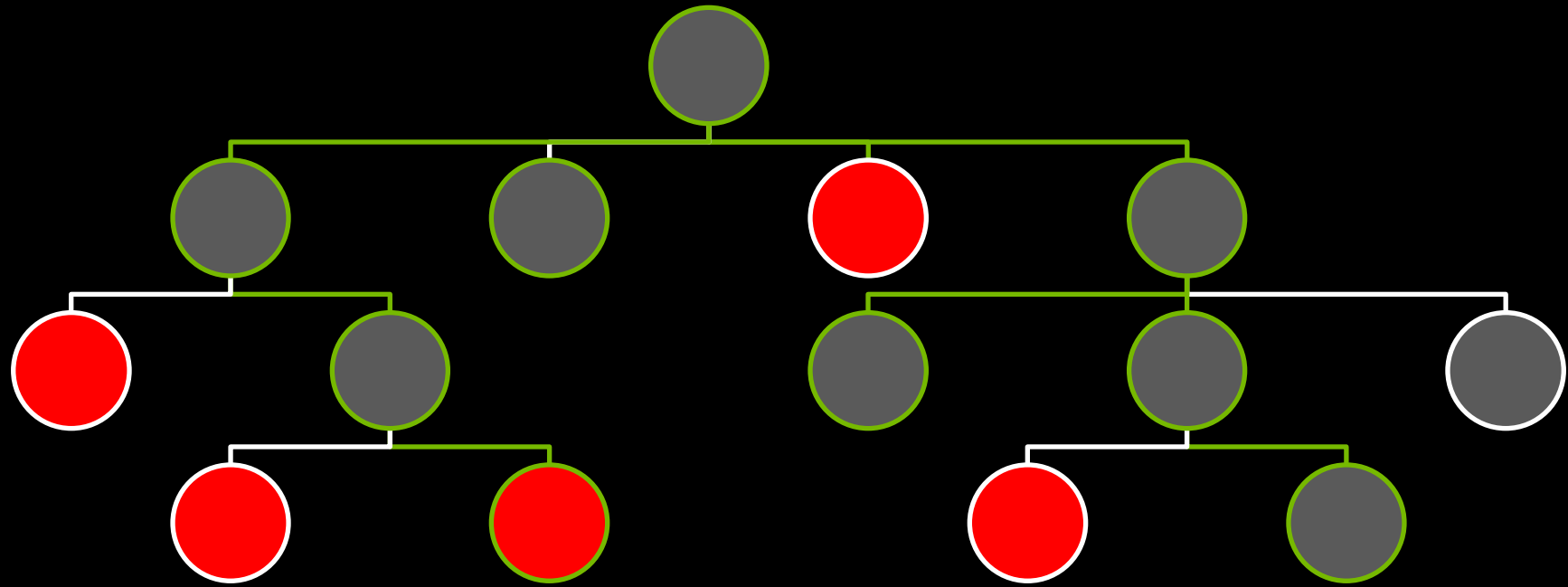
Version 1

↓  
"Fix <Z>."









Work step-by-step.

Guide the AI.

If you don't know what the steps are,  
figuring that out is the first step.

# Prompt Sensitivity



Implement a parallel algorithm in C++ that takes a range as an input and removes duplicate consecutive elements from it. Use only the Standard C++ parallel algorithms to provide parallelism.



Implement a parallel version of `std::unique` in C++ using other Standard C++ parallel algorithms to provide parallelism.



Write a C++ function that takes a range and removes consecutive duplicate elements from it. You do NOT have to sort the range. You must use Standard C++ algorithms to implement all  $O(N)$  operations, but you cannot use `std::unique` or `std::unique_copy`. You must



Implement a C++ algorithm that takes a range and removes consecutive duplicate elements from it. Parallelize all  $O(N)$  operations with Standard C++ parallel algorithms.



List multiple different approaches for implementing a parallel version of `std::unique` in C++ using other Standard C++ parallel algorithms to provide parallelism. Remember, `std::unique` only removes consecutive duplicate elements, so there is no need to sort the input. All operations on the input range should be done in parallel. We don't want to use any synchronization primitives like mutexes. Be concise in your reply and don't show me any code yet.

# Bad At Saying No

- Always agrees, but often does not understand or act on exegesis.
- Answers can become increasingly off-track with time.
- Forced niceness makes it unreliable.
- It should be easier for it to say "No" or "I can't".



# We Judge AI Unfairly

Do you write code that runs correctly on the first try?

Is your first thought always the best one?

# We Judge AI Unfairly

We think iteratively.

We refine our thoughts.

We have internal monologue.

Neural networks alone do none of that, but we can build more complex AI systems that do.

# Conclusions

- Analysis tasks are an excellent fit for AI; creation is harder.
- Success with AI models for software engineering requires skill.
- We should all develop that skill.
- Integrate AI into your existing workflows today.
- Explore how AI will change how we work in the future.

# Endnote: AI-Assisted Software Engineering

Bryce Adelstein Lelbach

(feat. Conor Hoekstra)

