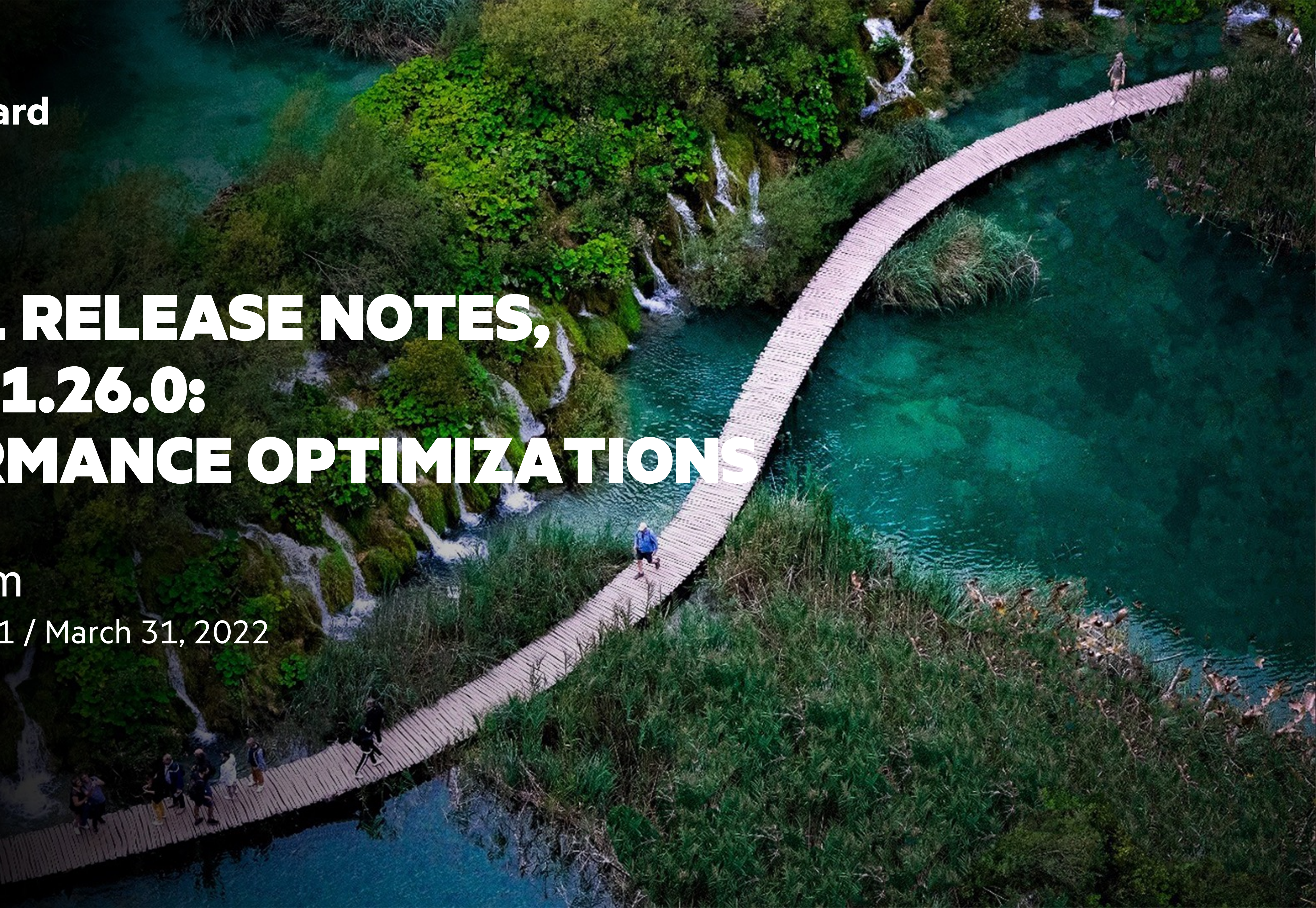
Hewlett Packard
Enterprise



CHAPEL RELEASE NOTES, 1.25.1 / 1.26.0: PERFORMANCE OPTIMIZATIONS

Chapel Team

December 9, 2021 / March 31, 2022

OUTLINE

- Slice Serialization
- Memory Tracking Opt.
- Regex Optimizations

SLICE SERIALIZATION IMPROVEMENTS

SLICE SERIALIZATION IMPROVEMENTS

Background and This Effort

Background

- Array slices have been expensive to create due to privatization costs
- A non-user-facing '-schpl_serializeSlices=true' flag has been developed to reduce the cost
  - Disables privatization for array slices
  - Uses serialization and remote value forwarding instead
- Want to enable by default to benefit codes like ChplUltra, which opts into the flag
  - However, this flag increases communication in some cases, so currently off-by-default

This Effort

- Implemented transitive serialization for reducing communication in one of the cases

SLICE SERIALIZATION IMPROVEMENTS

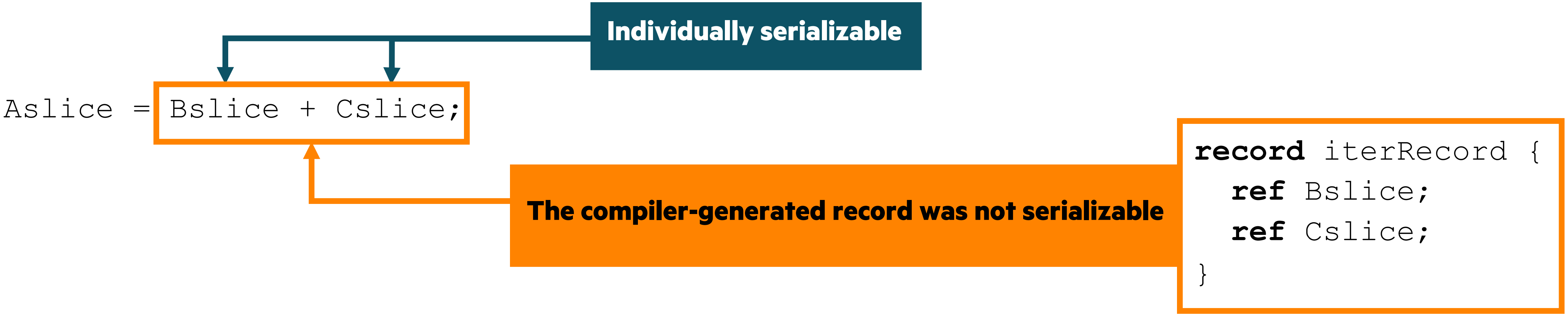
Impact

- In Chapel 1.25.x, basic slice operations were cheap with '-schpl_serializeSlices=true'

```
Aslice = Bslice;
```

// both slices can be serialized and forwarded in 'on' statements

- However, complicated promotions with slices involve a compiler-generated "iterator record"
  - This record could not be serialized and forwarded, causing extra communication



Individually serializable

```
Aslice = Bslice + Cslice;
```

The compiler-generated record was not serializable

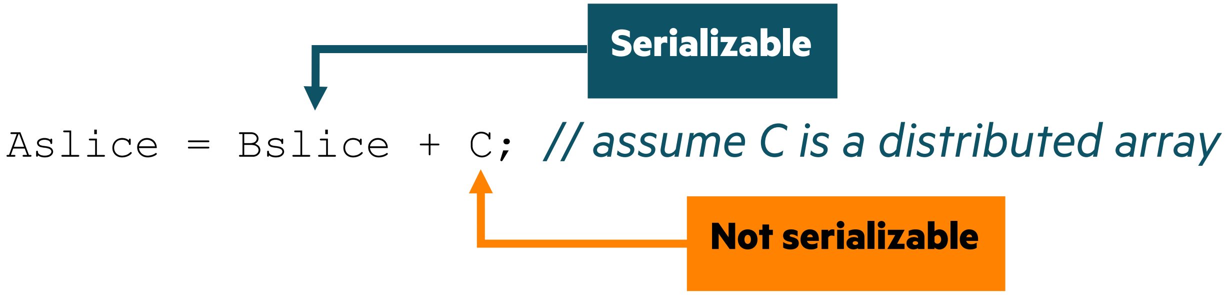
```
record iterRecord {
  ref Bslice;
  ref Cslice;
}
```

- In Chapel 1.26, the compiler can transitively create serialization/deserialization for these records
  - They can be forwarded with '-schpl_serializeSlices=true', avoiding extra communication

SLICE SERIALIZATION IMPROVEMENTS

Next Steps

- Avoid communication in the last known cases
  - When a distributed array's slice is used alongside a distributed array in a promoted expression



Serializable

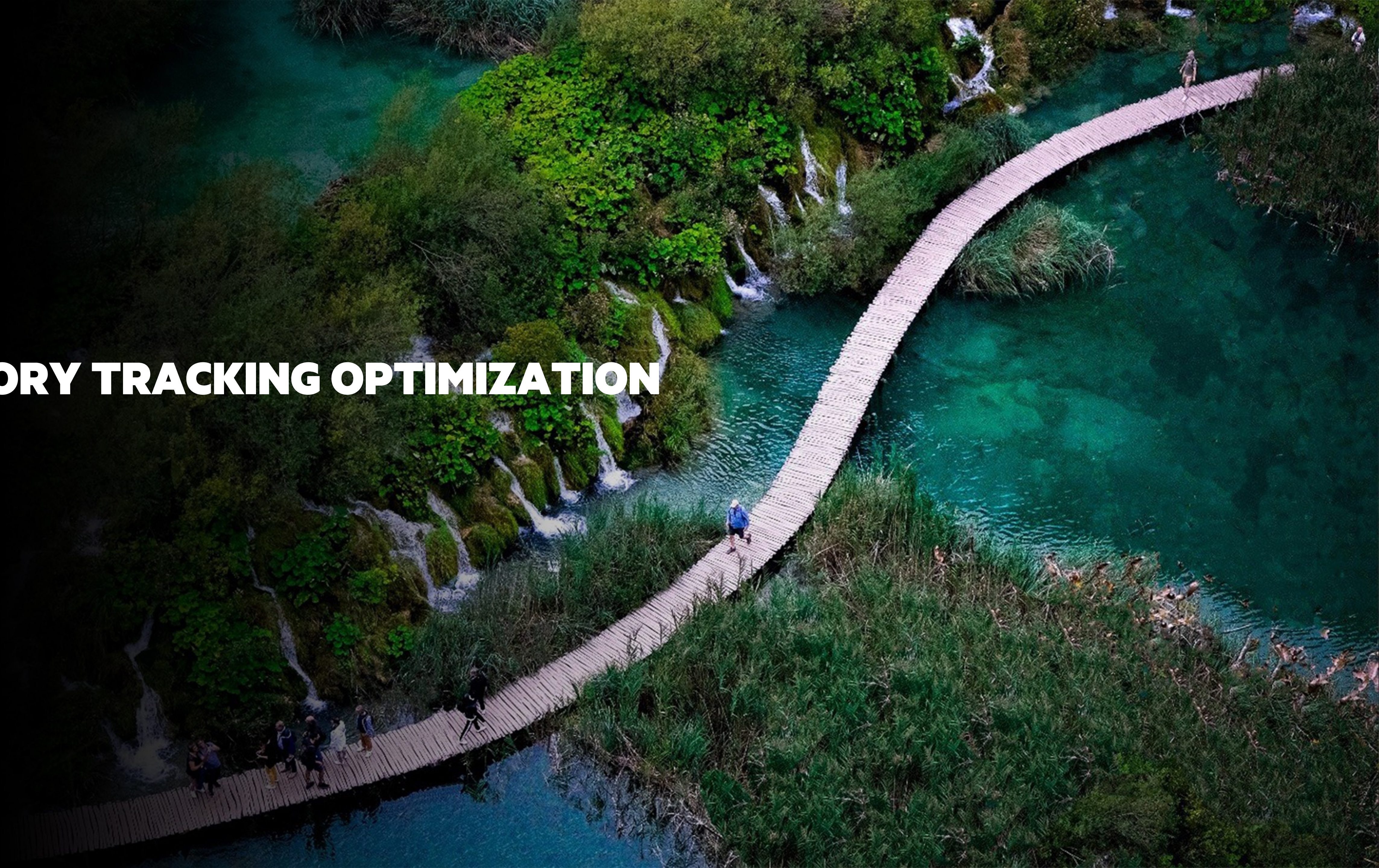
```
Aslice = Bslice + C; // assume C is a distributed array
```

Not serializable

- **Option 1:** Enable distributed array serialization in these scenarios only
  - Easier implementation, smaller impact

- **Option 2:** Disable privatization, always use serialization for distributed arrays instead
  - Potentially more challenging implementation
  - Yet, removing privatization can help other areas (e.g., GPU support)

MEMORY TRACKING OPTIMIZATION



MEMORY TRACKING OPTIMIZATION

Background and This Effort

Background: Chapel supports tracking memory allocation/deallocation for debugging purposes

- Not implemented efficiently, allocations tracked in a hash table protected by a global lock
- Primarily used by core team for '--memLeaks' support
- Arkouda uses '--memTrack' in combination with 'memoryUsed()' to estimate memory usage
  - Used to report operations that will likely result in out-of-memory conditions instead of crashing the server
  - Initially thought performance impact would be minimal since Arkouda mostly allocates massive arrays
  - Discovered large slowdowns for regular expression operations that had many small concurrent allocations

This Effort: Optimize and use existing '--memThreshold' option

- '--memThreshold' avoids tracking allocations smaller than the specified threshold
  - Previously, deallocating still required table lookup, as the size was unknown at deallocation time
  - Optimized now to query memory layer for actual size and skip lookup when size is below threshold

MEMORY TRACKING OPTIMIZATION

Impact

- Significantly faster tracking for concurrent allocations below ‘--memThreshold’

```
coforall 1..here.maxTaskPar do
  for i in 1..1_000_000 do
    var s = i:string;
```

Configuration (128 core CPU)	Time
w/o --memTrack	0.19s
w/ --memTrack	144.50s
w/ --memThreshold before	33.06s
w/ --memThreshold now	0.22s

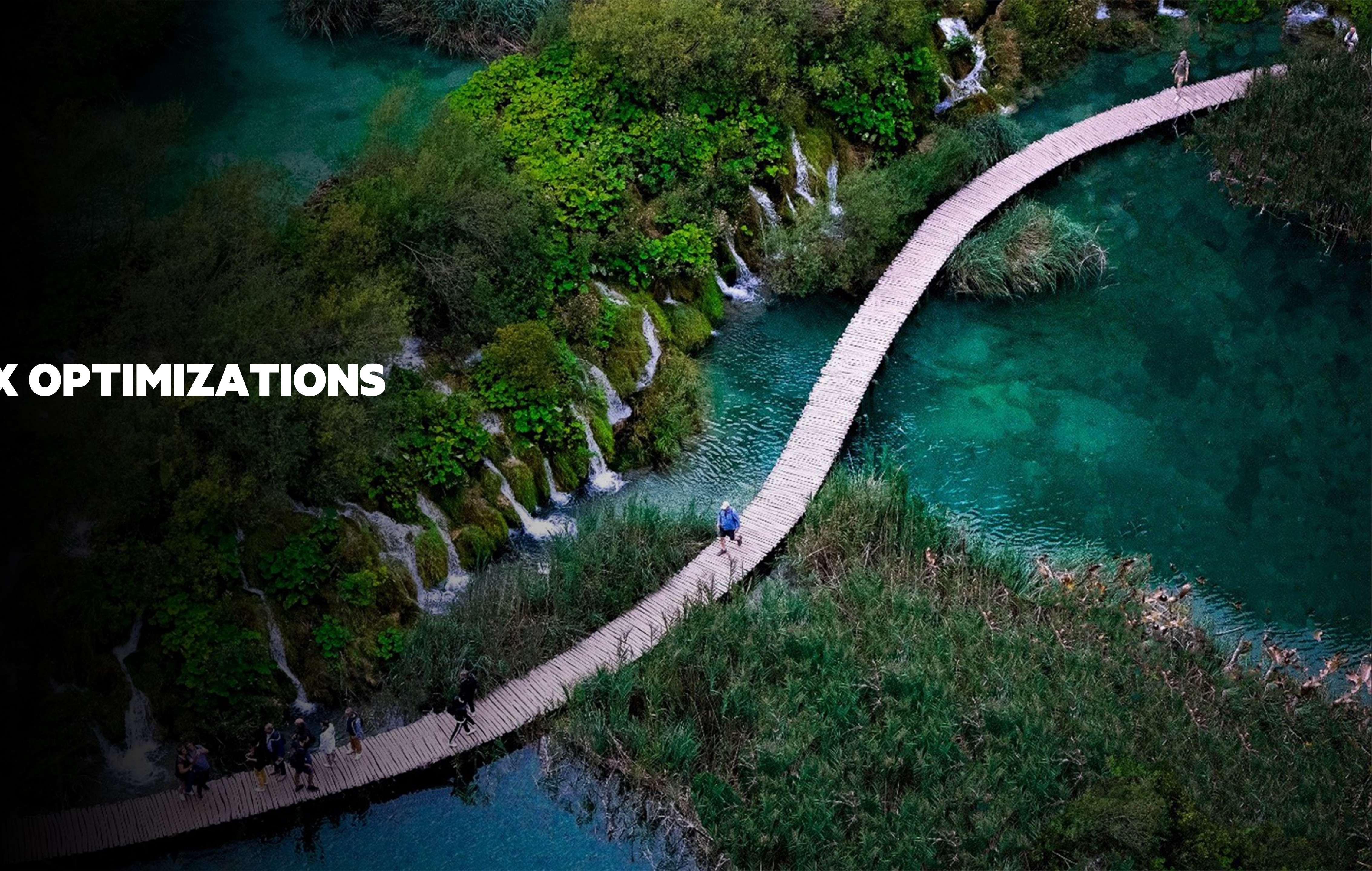
MEMORY TRACKING OPTIMIZATION

Next Steps

- Optimize memory tracking
  - For just ‘memoryUsed()’, an atomic counter could be used
    - Faster than hashtable w/ lock, but contended atomics are still somewhat slow
  - May be able to provide less precise tracking (e.g., track jemalloc chunks instead of each allocation)

- Provide a more principled mechanism for reporting out-of-memory conditions
  - Current Arkouda approach requires hardcoding memory estimates for key operations
    - Error-prone and invasive, makes it difficult to separate core routines out into mason packages

REGEX OPTIMIZATIONS



REGEX OPTIMIZATIONS

Background:

- The ‘Regex’ module provides regular expression support through C interop and the RE2 library
- Compiled regular expressions are stored in the ‘regex’ record which needs special care to contain a C pointer:
  - Save the home locale where the C pointer is valid, use ’on this.home { … }’ when searching, matching, etc.

This Effort:

- Eagerly localize ‘regex’ values on assignment by recompiling on the current locale
  - the recompilation process amounts to creating a new local RE2 object
- Implement serialization for ‘regex’ to reduce communication
  - If the pattern is a small string, it will be sent inside the arg bundle for a remote ‘on’ execution

Impact:

- No remote operations for common metods: ‘search’, ‘match’, ‘split’, ‘matches’
- Removing ‘on this.home’ enabled turning heap allocations into stack allocations
- Enables creating task-private regular expressions

```
forall s in strings with (var r = compile(pattern)) { … }
```

OTHER PERFORMANCE IMPROVEMENTS

OTHER PERFORMANCE IMPROVEMENTS

For a more complete list of performance changes and improvements in the 1.25.1 and 1.26.0 releases, refer to the following sections in the CHANGES.md file:

- ‘Performance Optimizations/Improvements’
- ‘Compilation Time / Generated Code Improvements’
- ‘Memory Improvements’

THANK YOU

https://chapel-lang.org
@ChapelLanguage