Hewlett Packard
Enterprise

CHAPEL 1.27.0/1.28.0 RELEASE NOTES: COMPILER, PERFORMANCE, AND TOOL IMPROVEMENTS

Chapel Team

June 30, 2022 / September 15, 2022

OUTLINE

LLVM-14 SUPPORT

LLVM-14

Background, This Effort and Status

Background:

- LLVM is the default back-end for Chapel
- The LLVM project releases new major versions about twice per year

This Effort:

- Updated Chapel to use LLVM-14, the latest major version
  - Updated the version in the third-party directory
  - Updated the Chapel compiler to address API differences
- Maintained compatibility with older versions as well

Status:

- Started using LLVM-14 for most test configurations
  - Continued testing versions 11–13 for a subset of test configurations

Next Steps: Continue tracking new releases of LLVM

USING LLVM TYPES TO ACCELERATE COMPILATION

LLVM ADTS

Background and This Effort

Background:

- The LLVM project includes some data structures designed for use in compilers
  - These Abstract Data Types (ADTs) are alternatives to standard C++ data structures
  - Some examples:

```
SmallVector    // alternative to std::vector optimized for short vectors
SmallPtrSet    // alternative to std::set optimized for small sets
DenseSet       // alternative to std::set
DenseMap       // alternative to std::map
```

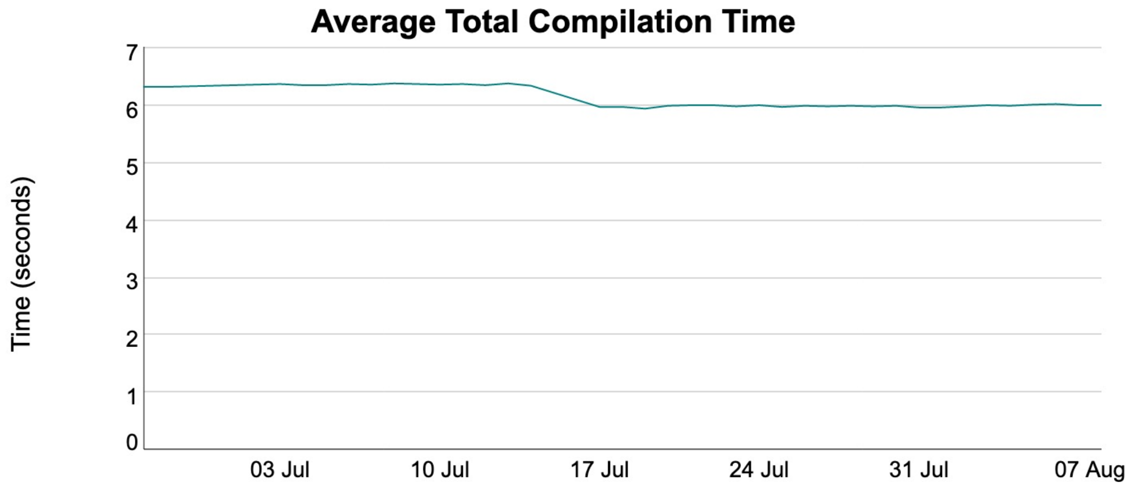
This Effort:

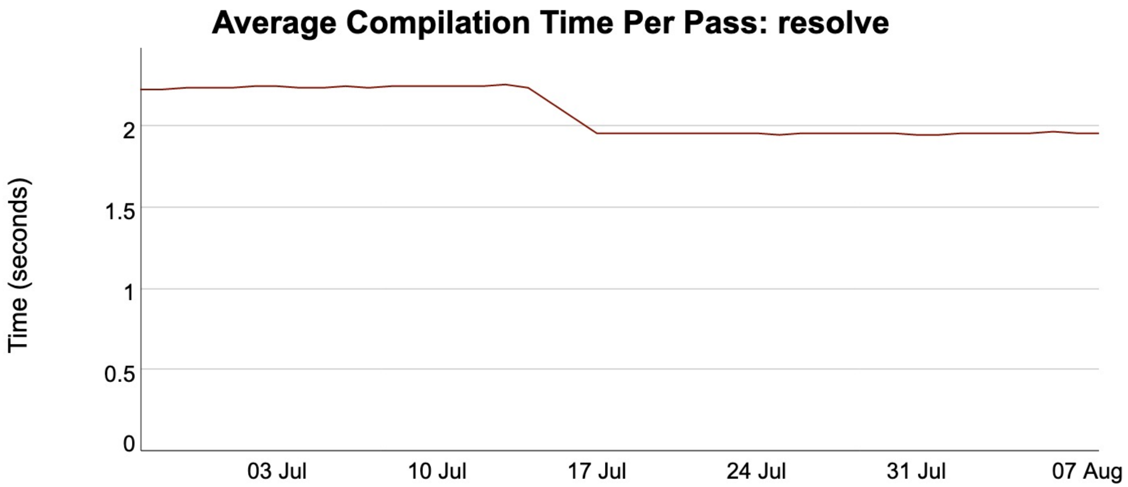
- Added some uses of these LLVM ADTs to the production compiler, which improved performance
  - As a result, the LLVM Support Library is now required to build the compiler
  - If no system install of LLVM is found, the LLVM Support Library will be built from the bundled LLVM
- Also began making use of these ADTs in new ‘dyno’ compiler code

LLVM ADTS
Impact and Next Steps

Impact: Modest improvement in average total compilation time (6%)

- 33% improvement in average scope resolve time
- 15% improvement in average resolution time
- No significant improvement for larger applications



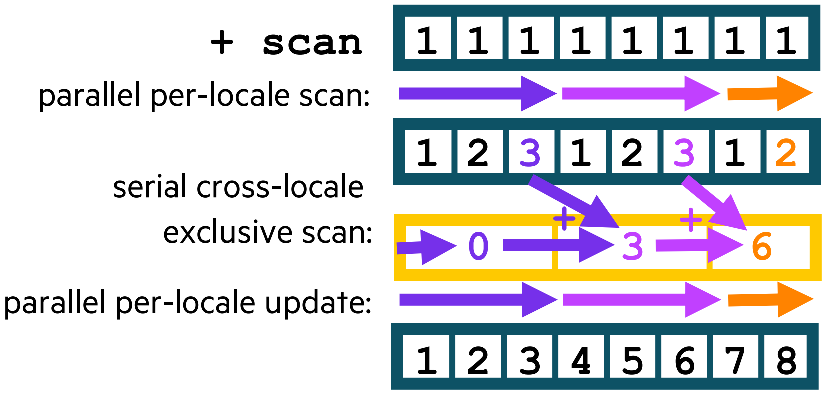


Next Steps: Look for additional use-case opportunities in both ‘dyno’ and the production compiler

SCAN OPTIMIZATIONS

SCAN OPTIMIZATIONS
Background

- Scans on block-distributed arrays were parallelized in Chapel 1.20
  - Uses a multi-pass implementation
    - Each locale does a parallel scan on its region of the array, stores per-locale state into replicated array
    - Initial locale gathers per-locale state, does a serial cross-locale scan, stores results into a replicated array
    - Each locale updates its region of the array with the cross-locale results



SCAN OPTIMIZATIONS
This Effort

- Identified that replicated arrays have high creation cost due to large amount of communication

- Updated block-distributed array scan implementation to avoid using replicated arrays
  - Use local array on initial locale to store first-pass results
    - Allows remote locale to store results in parallel, speeding up serial cross-locale scan
  - Use custom replicated-like data structure to store cross-locale scan results
    - Scan algorithm permits creating per-locale storage during first-pass, avoiding separate comm to create distributed array

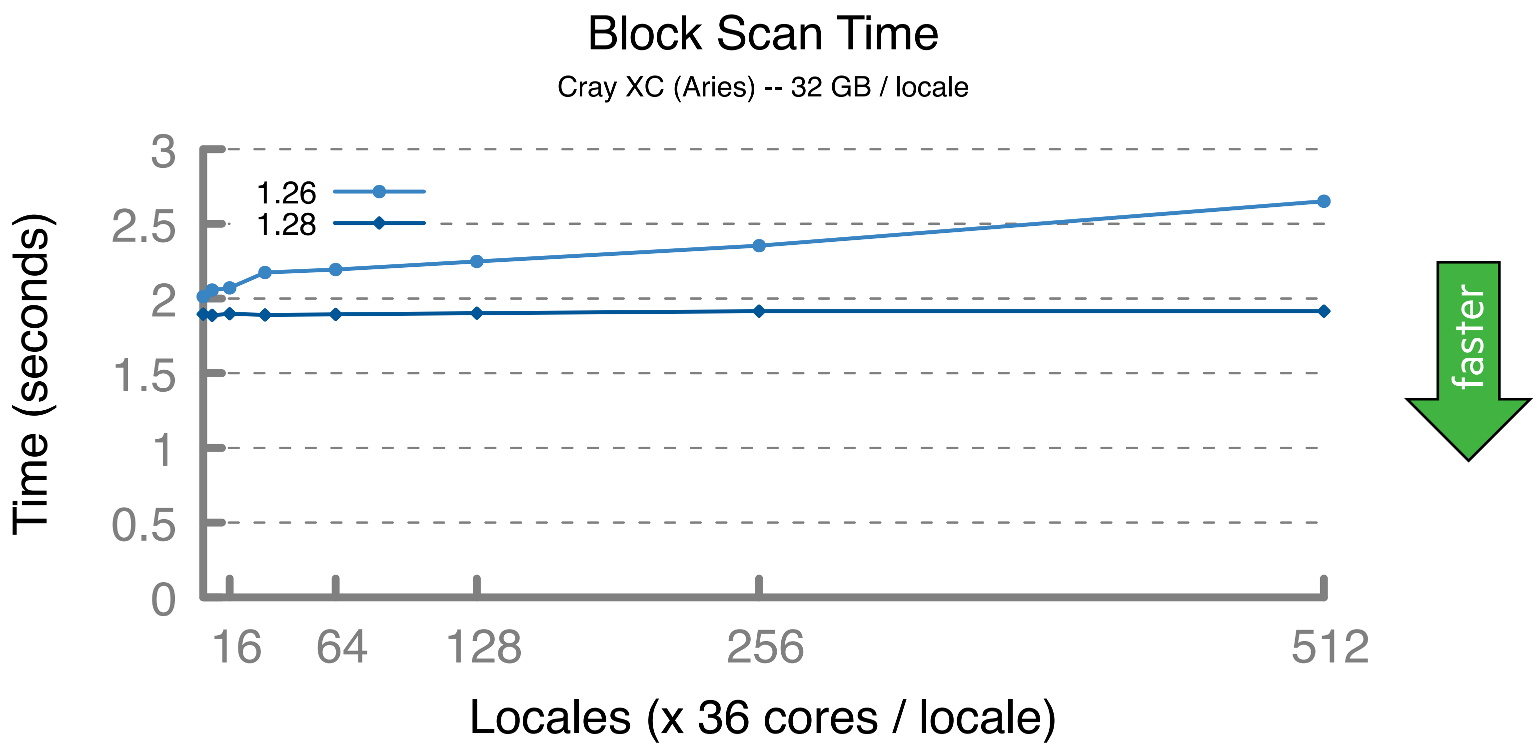
- Made micro-optimizations to further reduce scan communication

- Updated per-locale portion of scan to operate on local views when input and output distributions match
  - Reduces overhead for indexing into arrays

```
var A = newBlockArr(1..n, int);
var B = + scan A;             // A and B have same distribution, can operate on local views of A
var C = + scan A[{1..10}]; // A and C have different distribution, must operate on global view of A
```

SCAN OPTIMIZATIONS
Impact

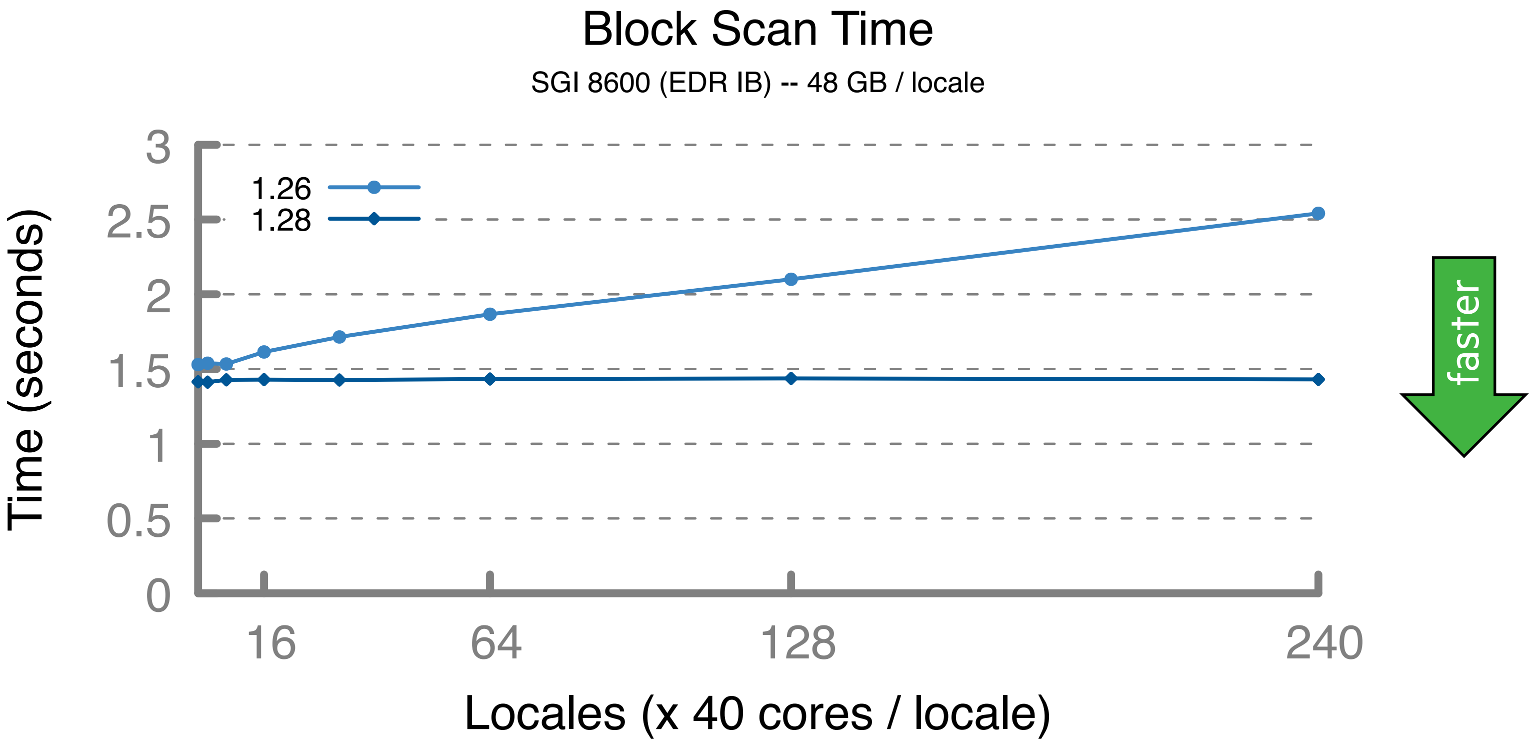
- Improved performance and scalability of scans on block-distributed arrays



SCAN OPTIMIZATIONS

Impact

- Improved performance and scalability of scans on block-distributed arrays
  - Particularly for configurations with less optimized fine-grained communication



SCAN OPTIMIZATIONS

Status and Next Steps

Status:

- Scans on block-distributed arrays are well-tuned with minimal communication
  - No known remaining optimization opportunities remain

Next Steps:

- Parallel scan improvements:
  - ensure scans of 1D array-like expressions are parallelized

    ```
    B = + scan (A: int);
    ```

  - parallelize scans of multidimensional arrays
  - consider extending parallelism to challenging/less mature distributions (e.g., Cyclic, Block-Cyclic)
  - generalize implementation to support cases where the ‘result’ and ‘state’ types don’t match
- Add support for partial scans, exclusive scans, directional scans
- Finalize and document the user-defined reduction/scan interface
- Reduce the overheads associated with creating replicated / distributed / privatized arrays

DYNO-CHPLDOC

DYNO-CHPLDOC
Background

- The ‘chpldoc’ tool generates ‘.rst’/‘.html’ documentation files by parsing commented ‘.chpl’ source files
  - ‘sphinx’ is leveraged under the hood to generate ‘.html’ files from ‘.rst’

- Historically, ‘chpldoc’ was implemented as an optional pass within the ‘chpl’ compiler
  - This approach resulted in several display issues with ‘chpldoc’ output that had never been addressed

- Since the compiler front-end is being rewritten for ‘dyno’, ‘chpldoc’ needed to be revisited as well
  - Since ‘dyno’ adds a new compiler library interface, a standalone ‘chpldoc’ tool is an ideal test case for it
    - Demonstrates how linters or code formatting tools could be similarly based on the ‘dyno’ compiler library

- As of 1.26, had a rough prototype of this new ‘dyno’-based ‘chpldoc’
  - Only 15/150 tests of ‘chpldoc’ passed using it at that time

DYNO-CHPLDOC

This Effort

- In 1.28, we have replaced ‘chpldoc’ with this ‘dyno’-based version of ‘chpldoc’
  - Serves as a drop-in replacement for ‘chpldoc’
  - Improves several cases that were not handled well with the previous ‘chpldoc’

- Increased number of documentation tests by ~10%

- Updated ‘sphinx’ Domain for the Chapel language, ‘sphinxcontrib-chapeldomain’, to v0.0.23
  - Now handles ‘operator’ keyword

DYNO-CHPLDOC
Impact

- Improved ability to control '.rst' output
  - The 'dyno' parser maintains a more accurate representation of the original Chapel source code

- Operators are now labeled with 'operator' keyword rather than 'proc':
  - was:
    ```
    proc *(s: bytes, n: integral): bytes
    ```
  - now:
    ```
    operator *(s: bytes, n: integral): bytes
    ```

- Internal rewrites of language features are no longer revealed:
  - was:
    ```
    const myLocaleSpace = 0..chpl__nudgeHighBound(numLocales)
    ```
  - now:
    ```
    const myLocaleSpace = 0..<numLocales
    ```
  - was:
    ```
    var infoLevels = new set(LogLevel, chpl__buildArrayExpr(LogLevel.INFO, LogLevel.DEBUG))
    ```
  - now:
    ```
    var infoLevels = new set(LogLevel, [LogLevel.INFO, LogLevel.DEBUG])
    ```

DYNO-CHPLDOC

Impact (continued)

- Literals are now displayed as they appear in source code
  - ‘string’ values are quoted:
    - was:
      ```
      param defaultBuffSize = if CHPL_COMM == ugni then 4096 else 8192
      ```
    - now:
      ```
      param defaultBuffSize = if CHPL_COMM == "ugni" then 4096 else 8192
      ```
  - ‘real’ values display all significant decimal places:
    - was:
      ```
      param pi = 3.14159
      ```
    - now:
      ```
      param pi = 3.14159265358979323846
      ```
  - Hex and octal values display in proper format:
    - was:
      ```
      param H5F_ACC_DEFAULT = 65535: c_uint
      ```
      ```
      proc mkdir(name: string, mode: int = 511, parents: bool = false) throws
      ```
    - now:
      ```
      param H5F_ACC_DEFAULT = 0xffff: c_uint
      ```
      ```
      proc mkdir(name: string, mode: int = 0o777, parents: bool = false) throws
      ```

DYNO-CHPLDOC

Impact (continued)

- Postfix '?' operator is now displayed to indicate a nilable class type
  - was: `proc this(tbl: string) ref: shared nilable Toml throws`
  - now: `proc this(tbl: string) ref: shared Toml? throws`
- Multi-declarations declared outside of records and classes are now handled

```
module M {
  var x, y, z: int; // previously would not print any of these
}
```

- 'use'/'import' hints for submodules now include their parent module's name
  - was: `use Diagnostics;`
  - now: `use Memory.Diagnostics;`

DYNO-CHPLDOC

Status

- Default ‘chpldoc’ tool is now ‘dyno-chpldoc’
  - Both ‘chpldoc’ and ‘chpldoc-legacy’ are built with the ‘make chpldoc’ command

- Previous version of ‘chpldoc’ can still be accessed if desired
  - Use ‘chpldoc --legacy’ or ‘chpldoc-legacy’ to invoke previous version

- Any documentation differences for Chapel modules are improvements or innocuous [#20558]
  - Also verified Arkouda-generated documentation

- Performance is roughly equivalent to the previous version of ‘chpldoc’
  - e.g., timed results from running full documentation test suite
    - 1m14s ‘chpldoc’
    - 1m14s ‘chpldoc --legacy’

DYNO-CHPLDOC

Next Steps

- Tune performance
  - Opportunities exist for improvements to execution time, and possibly to memory overhead

- Add support for automated testing of code examples within chpldoc comments

- Get feedback from users

- Remove support for ‘chpdoc-legacy’ and simplify compiler code that was supporting it

MASON IMPROVEMENTS

MASON IMPROVEMENTS
Background

- Mason is Chapel's package manager
  - Design inspired by Rust's Cargo
- Mason aims to standardize and simplify the build process for Chapel programs
  - Compiling a Chapel program can get complicated with flags, etc., so Mason aims to handle builds for users
  - If all Chapel users used Mason, there would be a common feel to building and running all projects
    - i.e., just run 'mason build' and the project compiles as expected
- Mason aims to create a community around Chapel package development
  - Registry hosted online to store packages, but does not have many packages today
- Mason aims to handle dependency management, creating reproducible builds
  - Keeping version dependencies straight can be tedious when done by hand

MASON IMPROVEMENTS
This Effort: Mason Package Types

- Mason previously assumed that all packages were going to be libraries
  - Library packages do not run as standalone projects and are only expected to be ‘use’d by other projects
  - Made Mason unusable for applications and small projects
- Implemented a “Library”, “Application”, “Lightweight” distinction
  - Library: use Mason to create and publish a library to the Mason registry
    - Not intended to be run as a standalone application
  - Application: use Mason as a build tool and dependency manager
    - Designed to assist in the development of standalone applications, benchmarks, etc.
  - Lightweight: use Mason only as a dependency manager
    - Useful for projects like Arkouda that already have a build process, but would like to use Mason packages
- These changes were inspired by Rust’s Cargo

MASON IMPROVEMENTS

This Effort: Initialization

- Simplified mason package initialization to only create essential files (matches Cargo initialization)

Result of ‘mason new’ in 1.26

```
MyPackage/
│
├── Mason.toml
├── example/
├── src/
│   └── myPackage.chpl
└── test/
```

Result of ‘mason new’ in 1.28

```
MyPackage/
│
├── Mason.toml
└── src/
    └── myPackage.chpl
```

- Removed confusing interactive initialization
  - Would prompt users to input information about licensing and Chapel versioning — unnecessary in most cases
    - Fields are populated with defaults and can be modified as needed by users
- Aligned behavior of ‘mason new’ and ‘mason init’
  - ‘mason new’ creates a mason package given a location, ’mason init’ creates a mason package in current directory

MASON IMPROVEMENTS
This Effort: Other Improvements

- Added user-requested ability to include git repositories as Mason dependencies
  - Package does not need to be in Mason registry
  - Package does not need to conform to Mason “release” requirements
  - Can use a specific revision or branch of the package

```
HelloWorld = { git = "https://github.com/myrepo/HelloWorld",
               branch = "test-branch" }
```

- Added a ‘mason modules’ command that generates command-line flags
  - Enables usage of Mason packages outside of the Mason package directory structure
  - Result is the absolute path to Mason packages in TOML (e.g., /path/to/mason-home/MyPackage.chpl)
- Reworked Mason documentation, splitting into multiple sections instead of one monolithic page
  - Added tutorials on using Mason from a package-user perspective
    - Previous documentation was written assuming library-developer perspective

MASON IMPROVEMENTS

This Effort: Experimenting with Arkouda

- Have been exploring usage of Mason in Arkouda to see where it could add value to existing projects
  - Given Mason's goal of being a build tool, wanted to see what it would take to replace the Arkouda 'Makefile'
  - Converting Arkouda modules to Mason packages could provide values to other users (e.g., argsort)
- 'mason modules' command was motivated by Arkouda, enabling integration into existing build process
  - Provides compiler flags to use Mason packages without changing the Arkouda directory structure
  - An experimental Arkouda branch using this approach helped identify areas in need of improvement
- Using Mason in offline environments is an ongoing effort
- Additional Mason development will be needed in order to provide value to Arkouda

MASON IMPROVEMENTS

Next Steps

- Improve Mason build support
  - Allow users to programmatically specify build flags, override commands, etc.
- Work towards providing a greater set of Mason packages
  - Port some existing Chapel package modules to Mason packages
- Enable Mason to be built in all configurations for portability
  - Today, can only be built with a ‘CHPL_COMM=none’ runtime
- Further improve Mason documentation
  - Add central document where supported commands are outlined and explained from user perspective
- Further improve Mason flexibility and usability
  - Allow different module names for Mason projects, improve testing infrastructure, etc.

PORTABILITY IMPROVEMENTS

PORTABILITY IMPROVEMENTS

Background: New requirements to build Chapel can introduce portability challenges

- cmake 3.13.4 or newer is now required
- the Chapel compiler now requires the LLVM Support library, but the bundled version is built if it is not found

This Effort: Continued to improve portability and packaging

- Addressed build problems in several configurations:
  - GCC 12, Alpine Linux, or Amazon Linux 2022
- Improved several aspects of Chapel configuration and build:
  - stopped saving the path to the linker in case it changes after 'chpl' is built
  - the quickstart environment now uses a system LLVM package when available
  - 'CHPL_LLVM=none' can use a system LLVM support library when available
  - now linking dynamically with the system LLVM on Mac OS X
- Chapel 1.28 was tested with 47 different OS configurations and prerequisite install commands were generated
- A community member has created a Chapel AUR package for Arch Linux!

Impact: Users are less likely to run into build issues in the field

OTHER COMPILER, PERFORMANCE, AND TOOL IMPROVEMENTS

OTHER COMPILER, PERFORMANCE, AND TOOL IMPROVEMENTS

For a more complete list of compiler, performance, and tool changes and improvements in the 1.27.0 and 1.28.0 releases, refer to the following sections in the CHANGES.md file:

- ‘Compiler Improvements’
- ‘Compilation-Time / Generated Code Improvements’
- ‘Error Messages / Semantic Checks’
- ‘[Platform-Specific] Performance Optimizations / Improvements’
- ‘Tool Improvements `
- ‘Packaging / Configuration Changes’
- ‘Build System Improvements’
- ‘Portability / Platform-specific Improvements’
- ‘Bug Fixes [for Build Issues | for Tools]’ / ‘Platform-specific Bug Fixes’
- ‘Launchers’
- ‘Third-Party Software Changes’

THANK YOU

https://chapel-lang.org
@ChapelLanguage