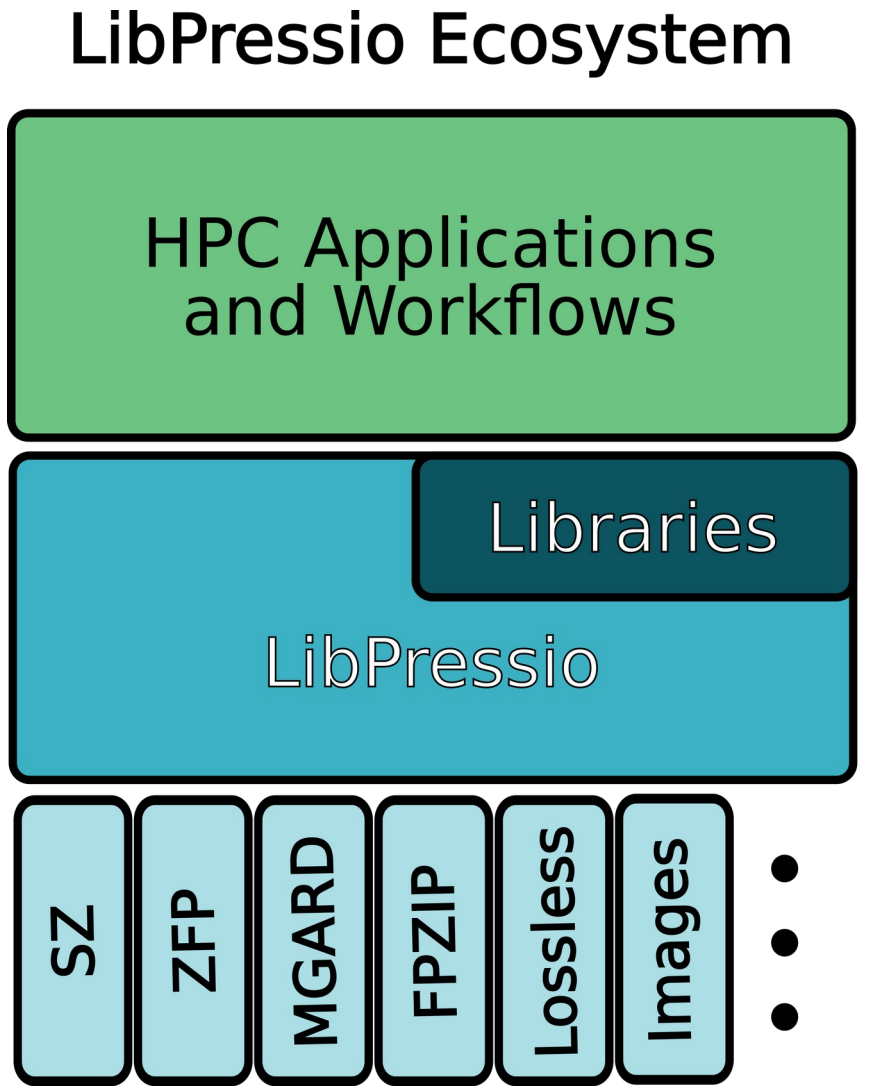
LibPressio

Robert Underwood – Clemson University

LibPressio – Generic Abstraction for Compression

- Problem: Every library has its own API
  - Harder to learn and use
  - Requires rewrite to use new compressors
  - Limited collaboration and comparison
- Solution: One C/C++ API for all compressors
- Who: Application Users and Compression Developers
- Features:
  - Supports many compressors
    - SZ, ZFP, MGARD, FPZIP
    - Also: Images (i.e. JPEG, WEBP), Lossless (BLOSC)
  - Safe, Consistent, Simple, Introspectable, Fully Documented
  - Tooling Interface for Analysis



LibPressio – Getting Started

- Easy to Install/Use (Spack, Container, Cmake)
- Extensible with:
  - Compressors – Writing a Compressor Plugin
  - Tooling Modules – Writing a Metrics Plugin
- A simple, consistent workflow
  - Get a reference to a compressor
  - Configure and assign options
  - Describe input and output buffers
  - Compress
  - Decompress
  - Release Resources
- Resources
  - Watch the "LibPressio Tutorial" on YouTube
  - Reference the extensive developer documentation

```c
//get the compressor
struct pressio* library = pressio_instance();
struct pressio_compressor* sz = pressio_get_compressor(library,
↪  "sz");

//configure, validate, and assign the options
struct pressio_options* config =
↪  pressio_compressor_get_options(sz);
pressio_options_set_integer(config, "sz:error_bound_mode",
↪  REL);
pressio_options_set_double(config, "sz:rel_err_bound", 0.01);
pressio_compressor_set_options(sz, config);

//read in an input buffer
size_t dims[] = {500,500,100};
struct pressio_data* description =
↪  pressio_data_new_empty(pressio_float_dtype, 3, dims);
struct pressio_data* input_data =
↪  pressio_io_data_path_read(description, "CLOUDf48.bin.f32");

//create output buffers
struct pressio_data* compressed_data =
↪  pressio_data_new_empty(pressio_byte_dtype, 0, NULL);
struct pressio_data* decompressed_data =
↪  pressio_data_new_owning(pressio_float_dtype, 3, dims);

//compress and decompress the data
pressio_compressor_compress(sz, input_data, compressed_data);
pressio_compressor_decompress(sz, compressed_data,
↪  decompressed_data);
```

LibPressio – Progress and Direction

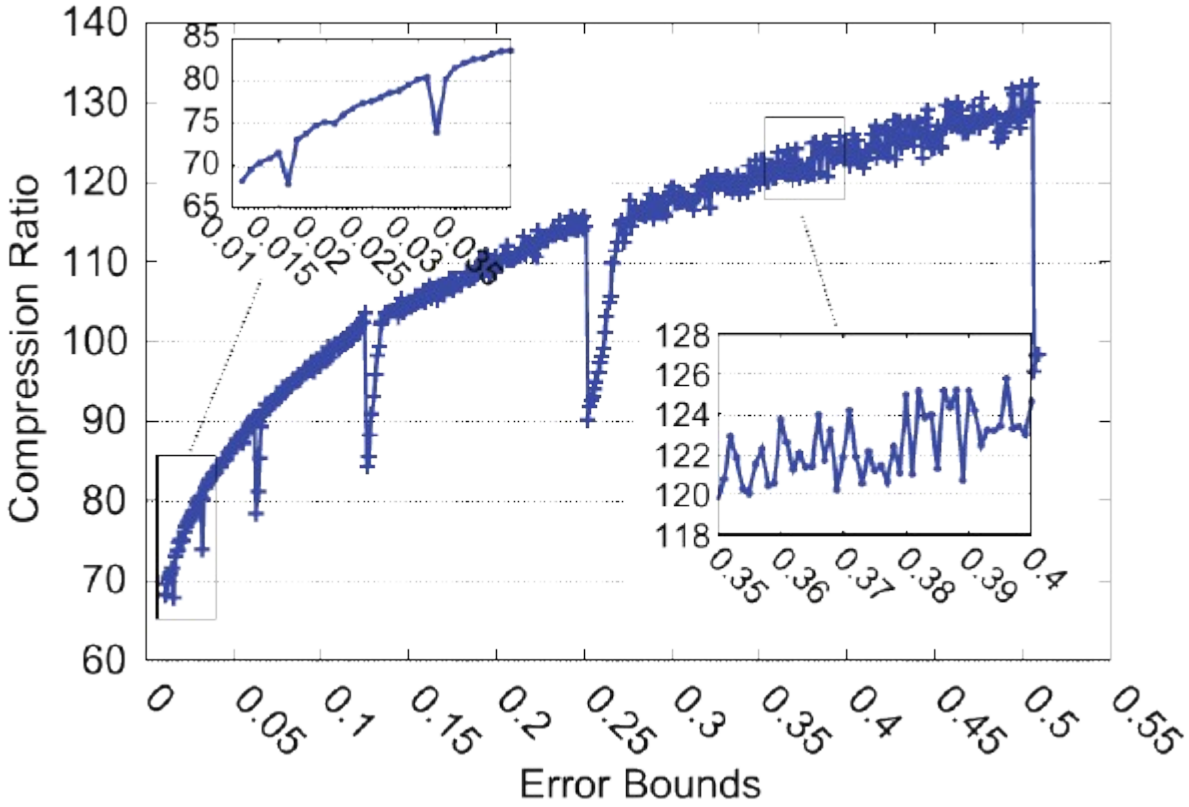
- Since last year:

- Several new and improved compressor plugins
- Better integration with IO libraries
- Improved metrics and metrics execution plugins
- Significant improvements to language bindings

- Current Uses:
  - Language Bindings (Bash, Python, Julia, R, Rust)
  - FRaZ/Opt – Autotuning Frameworks for EBLC
  - Z-Checker – Error Analysis Framework
  - Fault Injection Workflow
  - Distributed Compression Benchmarking
  - And many other research workflows!

LibPressioOpt – Why is this difficult?

- What: Tune compressors using user's metrics
- Why is this needed:
  - Users need lossy compression:
    - to reduce storage footprint
    - to achieve "best fit" compression
    - to manage streaming volume
  - Users care how their analysis are affected
  - Many user metrics are hard to bound analytically
  - Sometimes we can improve over analytical methods
- Why is this hard?
  - The relationships between bounds and metrics are complex



Formulating the Optimization Problem

- **Given:**

  Original Dataset $D_{f,t}$

  Decompressed Dataset $D'_{f,t}$

  Fixed Compression Parameters θ

  Acceptable Compressor Error Bound U

  Real compression ratio $\rho_r(D_{f,t}, e, \theta)$

  Target compression ratio $\rho_t(D_{f,t})$

  Target compression ratio relative tolerance ϵ

Let: Compressor Error Bound e

- **Minimize over e:**

  $(\rho_r(D_{f,t}, e, \theta) - \rho_t(D_{f,t}))^2$ s.t. $0 \leq e \leq U$

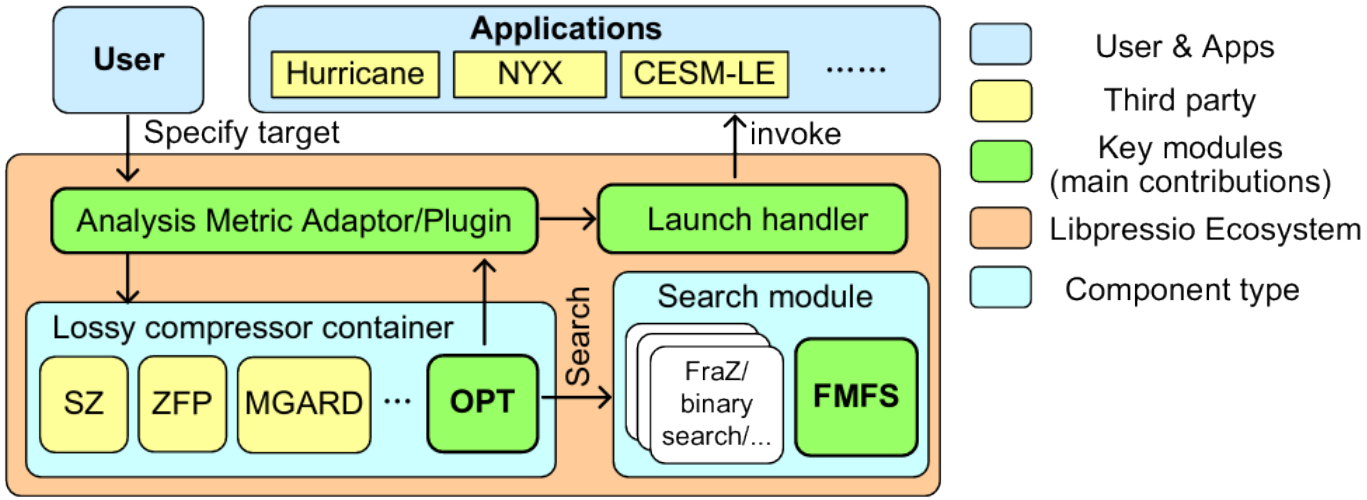
  if $(\rho_r(D_{f,t}, e, \theta) - \rho_t(D_{f,t}))^2 \leq \epsilon^2 \rho_t(D_{f,t})$, terminate

- **Many Algorithms preform poorly:**

  We don't have a analytic forms for ρ_r, $\rho_r\prime$, or $\rho_r\prime\prime$

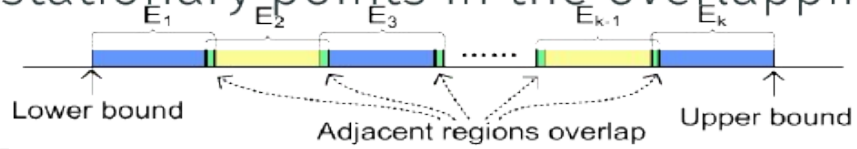
  Numerical derivatives are costly, $O(sec) - O(min)$

  Empirically, ρ_r often is non-convex many local optima



Parallelizing the Algorithm

1. By Field – embarrassingly parallel
2. By Timestep
   - Do first timestep as normal
   - Guess next solution is same as last
   - If wrong, do full tuning again
3. By Error Bound Range
   - Split search range $[0, U]$ into n similarly sized subranges run an independent search on each as hardware allows
   - a slight overlap (i.e. 10%) improves performance allowing for sufficient stationary points in the overlapping region



Algorithm 2 TRAINING

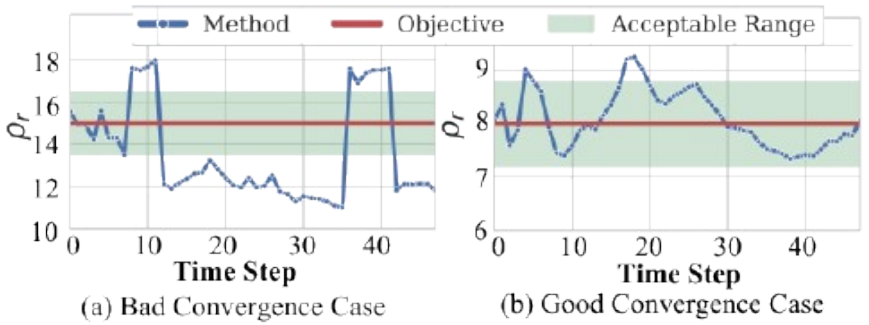
Input: target compression ratio $\rho_t(D_{f,t})$, acceptable error ϵ, dataset D_t, max allowed compression error U
Output: real compression ratio $\rho_r(D_{f,t}, e)$, recommended error bound setting e

```
tasks[N]
done ← false
for (i, (l, u)) ∈ make_error_bounds(U) do
    tasks[i] ← launch_task(D_t, l, u, ρ_t(D_{f,t}), ϵ, h)
end for
while notdone do
    last_task ← next_completed(tasks)
    candiate ← compression_ratio(last_task)
    if ρ_t(D_{f,t})(1 − ϵ) ≤ candidate ≤ ρ_t(D_{f,t})(1 + ϵ) then
        done ← true
        for task ∈ tasks do
            cancel_if_not_finished(task)
        end for
    end if
    done ← has_next(completed)
end while
ρ_r(D_{f,t}, e) = ∞
for task ∈ tasks do
    if finished(task) then
        ρ ← compression_ratio(task)
        if (ρ_r − ρ)^2 < (ρ_t − ρ)^2 then
            ρ_r = ρ
        end if
    end if
end for
return ρ_r(D_{f,t}, e), error_bound(task)
```

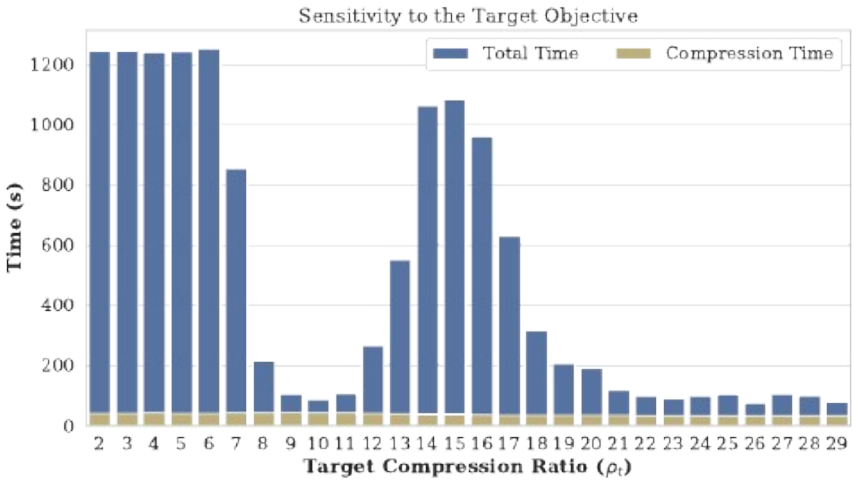
Worker Algorithm

Results: Runtime

- Runtime depends substantially if the requested target is feasible:
  - Good (feasible) Case: We terminate early most of the time
  - Bad (infeasible) Case: We alternate between a compression ratio which is too small or too large
- Very small compression ratios are often infeasible because there is a minimum compressed size
- There are also gaps between feasible and infeasible. For this figure $\rho_t(D_{f,t}) \in [14, 16]$ are infeasible for the specified ϵ
- In the feasible case, overhead is often $\approx$ 2x just compressing with the correct error bound.



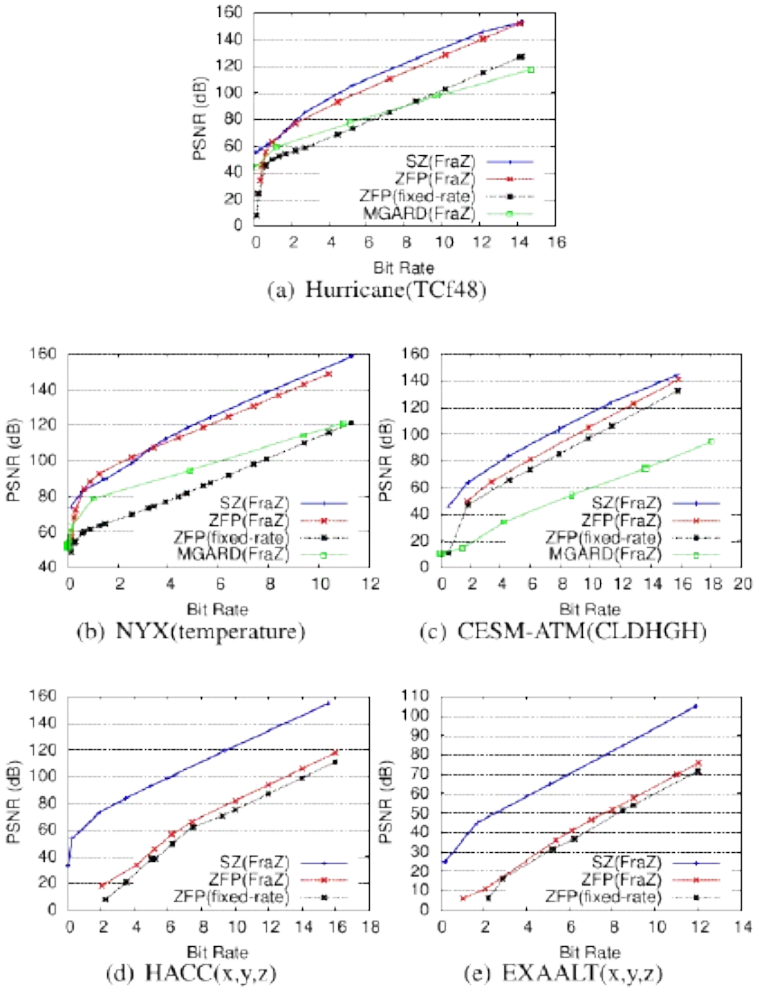
Solutions in good/bad case



Time to solution for many targets

Results: Quality of Solution

- Fixed Ratio SZ/ZFP is generally better than ZFP Fixed Rate at each compression ratio:
  - Better Rate Distortion (higher PSNR per bit rate)
  - Higher SSIM
  - Higher PSNR
  - Better visual quality
- Figure 1: Rate Distortion for Several Datasets
- Figure 2: Visual Quality for Several Compressors



(a) Hurricane(TCf48)

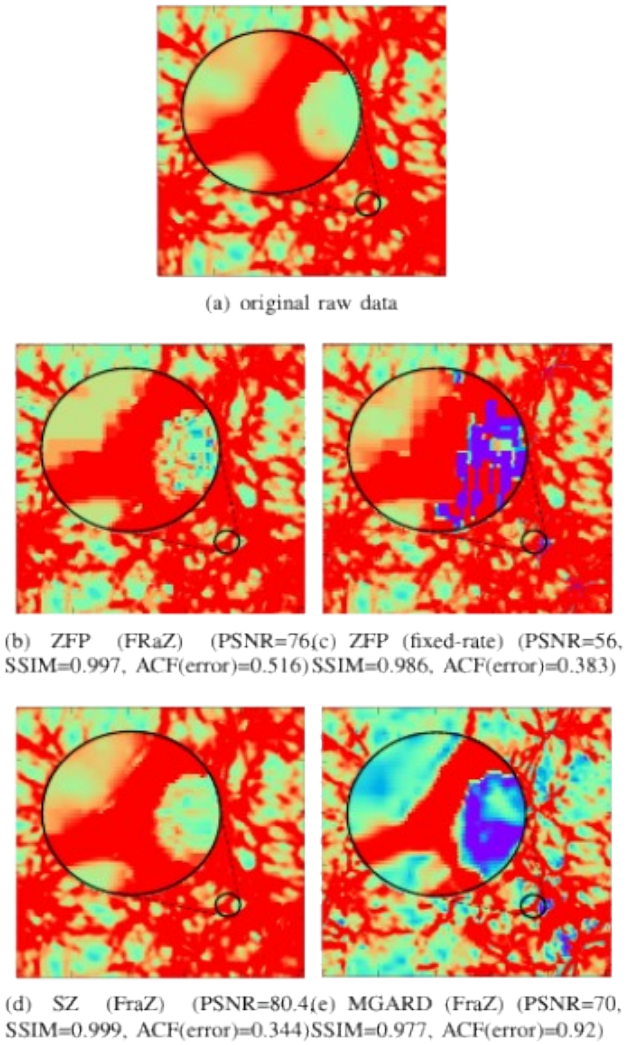
(b) NYX(temperature)

(c) CESM-ATM(CLDHGH)

(d) HACC(x,y,z)

(e) EXAALT(x,y,z)

1



(a) original raw data

(b) ZFP (FRaZ) (PSNR=76, SSIM=0.997, ACF(error)=0.516)

(c) ZFP (fixed-rate) (PSNR=56, SSIM=0.986, ACF(error)=0.383)

(d) SZ (FraZ) (PSNR=80.4, SSIM=0.999, ACF(error)=0.344)

(e) MGARD (FraZ) (PSNR=70, SSIM=0.977, ACF(error)=0.92)

2

Thanks

This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing

Argonne NATIONAL LABORATORY

Lawrence Livermore National Laboratory

EXASCALE COMPUTING PROJECT