Accelerating SuperDARN File I/O with Rust

REMINGTON ROHEL

SNAKES ON A SPACESHIP: CEDAR-GEM WORKSHOP

JUNE 23-27, 2025 DES MOINES IA, USA

Outline

- Motivation
- Writing Rust code for Python
- Testing
- Benchmarking

Motivation

- **pyDARNio** – Python package for reading/writing SuperDARN files
  - Used by pyDARN which plots SuperDARN data
  - Very slow (>10s to read 1 day of data, i.e. 100-600 MB of data)
  - Scientists often want years of SuperDARN data
- **Goal**: Speed up pyDARNio

pyDARNio

https://pydarnio.readthedocs.io/en/latest/ and https://github.com/SuperDARN/pyDARNio

Basic usage:

```
import pydarnio
file = "path/to/file"

# read from file
reader = pydarnio.SDarnRead(file)
fitacf_data = reader.read_fitacf()

# write to file
writer = pydarnio.SDarnWrite("/path/to/file")
writer.write_fitacf(fitacf_data)
```

DMAP Format

- Bespoke binary data format
- 3 data structures:

1. Record

code int32	**size** int32	**# of scalars** int32	**# of vectors** int32	**scalar fields**	**vector fields**

2. Scalar

name null-terminated string	**type** int8	**data** determined by "type"

3. Vector

name null-terminated string	**type** int8	**# of dimensions** int32	**dimensions** [int32]	**data** N-dimensional array of "type"

https://radar-software-toolkit-rst.readthedocs.io/en/latest/references/general/dmap_data/

Accelerating with Rust

- compiled programming language
- C-like speed but more guardrails, like enforced memory safety
- 4 main types:
  - Primitives, e.g. int8, float32, str, bool
  - Structs (can have associated methods)
  - Enums (can have associated methods)
  - Traits, which Structs or Enums can inherit
- Easy to create Python bindings for Rust packages. Users can clone your project and compile locally, or the developer generates wheel files and uploads to PyPI for many OS's and CPUs (using *pyO3* and *maturin*)
- Python API exposed, but installs Rust code behind-the-scenes in wheel file

Inside the Rust codebase

https://github.com/SuperDARNCanada/dmap

```rust
use crate::error::DmapError;
use crate::formats::dmap::{GenericRecord, Record};
use pyo3::prelude::*;

/// Functions for SuperDARN DMAP file format I/O.
#[pymodule]
fn dmap(m: &Bound<'_, PyModule>) -> PyResult<()> {
    m.add_function(wrap_pyfunction!(read_dmap_py, m)?)?;
    Ok(())
}
```

```
|
| Imports
| << pyo3 is the crate for converting to Python

<< Docstring for the library
<< Decorator that denotes this as a module for use
in Python
| Special function which specifies name + members
|    of Python module
|  Members (functions or classes) are added here
```

Inside the Rust codebase

```rust
/// Reads a generic DMAP file, returning a list of dictionaries containing the fields.
#[pyfunction]
#[pyo3(name = "read_dmap")]
#[pyo3(text_signature = "(infile: str, /)")]
fn read_dmap_py(infile: PathBuf) -> PyResult<Vec<IndexMap<String, DmapField>>>
{
    read_generic::<GenericRecord>(infile).map_err(PyErr::from)
}

/// Reads the data from infile into a collection of `IndexMap`s
fn read_generic<T: for<'a> Record<'a> + Send>(
    infile: PathBuf,
) -> Result<Vec<IndexMap<String, DmapField>>, DmapError> {
    match T::read_file(&infile) {
        Ok(recs) => {
            let new_recs = recs.into_iter().map(|rec| rec.inner()).collect();
            Ok(new_recs)
        }
        Err(e) => Err(e),
    }
}
```

```
<< Docstring for the Python function

<< Decorator that denotes this as a function usable in Python
<< Specifies the name the Python function will have
<< Specifies the parameter hints for the function
| The Rust function, takes some input and returns special Py types
|
|   Calls your Rust code to do something
|


|
| Rust function which is called above, has type bounds "T"
|
|    Returns special "Result" type, either Ok or Err
|    Calls method of type "T" (method specified in "Record" trait)
|
|    Matches return type of T::read_file(), either Ok or Err
|      and does some computation on the results
|
|
|    Implicitly returns output of match statement
|
```

Parallelization

```
use rayon::prelude::*;

fn read_records(mut dmap_data: impl Read) -> Result<Vec<Self>, DmapError> {

    // some setup here

    let mut dmap_results: Vec<Result<Self, DmapError>> = vec![];

    // single-threaded
    dmap_results.extend(
        slices
            .iter_mut()
            .map(|cursor| Self::parse_record(cursor)),
    );

    // parallelized
    dmap_results.par_extend(
        slices
            .par_iter_mut()
            .map(|cursor| Self::parse_record(cursor)),
    );
}
```

Very similar code, only have to change a few lines to get full CPU utilization!

How to get it to Python?

Cargo.toml:

```toml
[lib]
name = "dmap"
# "cdylib" is necessary to produce a shared library for Python to import from.
crate-type = ["cdylib", "rlib"]

[dependencies]
pyo3 = { version = "0.22.5", features = ["extension-module", "indexmap", "abi3-py38"] }
```

pyproject.toml:

```toml
[build-system]
requires = ["maturin>=1,<2", "numpy<3"]
build-backend = "maturin"

[tool.maturin]
bindings = "pyo3"
profile = "release"
compatibility = "manylinux2014"
auditwheel = "repair"
strip = true
```

- Use *maturin,* run `maturin develop` to build and install in your virtual environment (https://www.maturin.rs/)
- *maturin-action* builds GitHub action workflows for automating builds + shipping to PyPI for a range of OS's and CPUs (https://github.com/PyO3/maturin-action)
- This work in *dmap* project: https://github.com/SuperDARNCanada/dmap

Python side

```
>>> import dmap
>>> dmap.__doc__
'Functions for SuperDARN DMAP file format I/O.'

>>> dmap.read_dmap.__doc__
'Reads a generic DMAP file, returning a list of dictionaries containing
the fields.'

>>> dmap.read_dmap.__text_signature__
'(infile: str, /)'
```

Functions are re-exported by pyDARNio for seamless integration with SuperDARN software

Testing

```
#[test]
fn read_write_generic() {
    // [testing code here]
}
```

- Functions decorated with the `#[test]` macro are auto-detected when `cargo test` is invoked.
- Tests can be put in the same file next to where a function is defined.
- Tests can be embedded in docstrings
- Only functions exposed via Python API can be tested in Python – everything else must be tested in Rust

Benchmarking (criterion.rs)

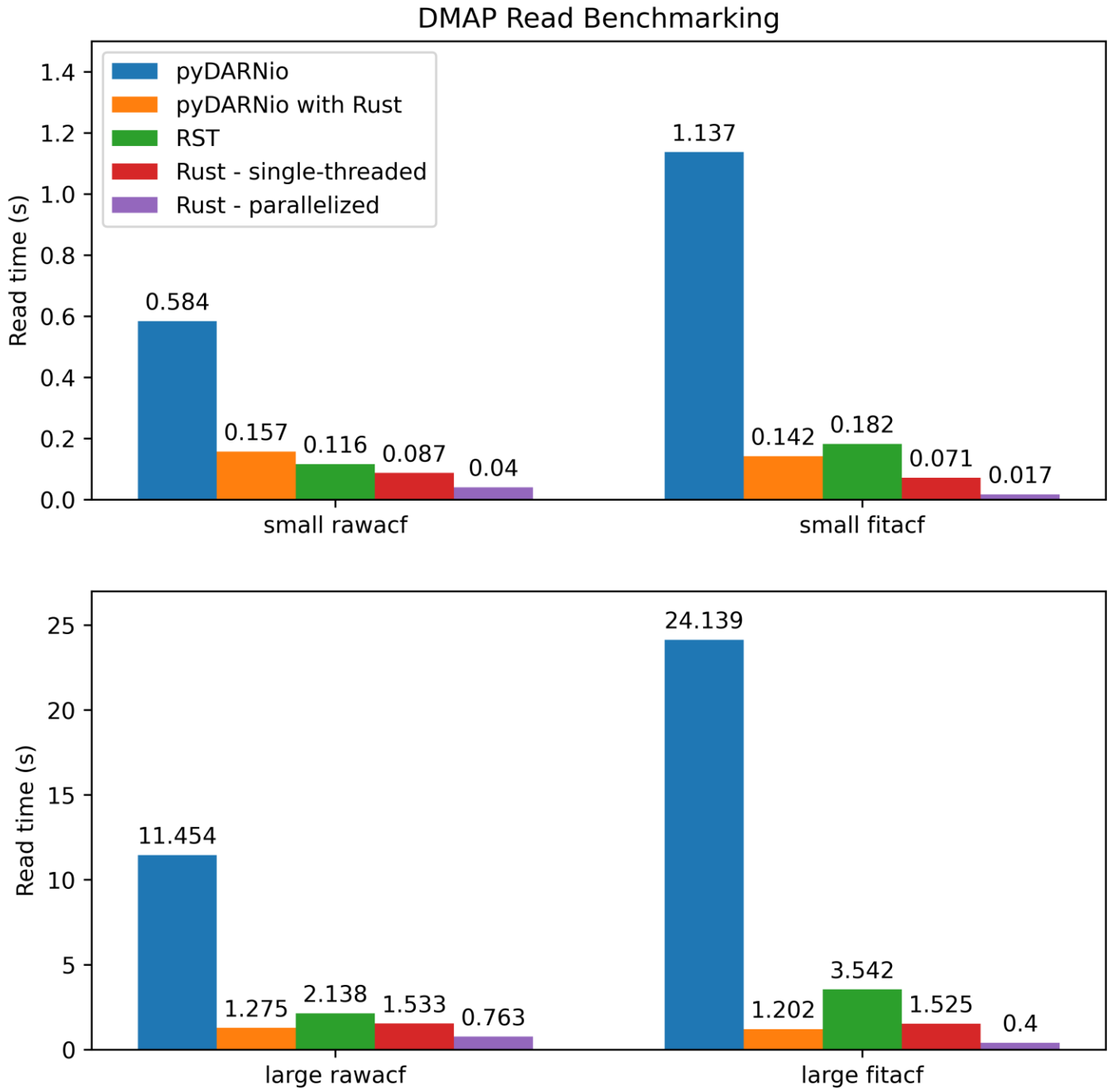
```rust
use criterion::{criterion_group, criterion_main, Criterion};

fn criterion_benchmark(c: &mut Criterion) {
    c.bench_function("Read RAWACF", |b| b.iter(|| read_rawacf()));
}

fn read_rawacf() -> Vec<RawacfRecord> {
    let file = File::open("tests/test_files/test.rawacf").expect("Test file not found");
    RawacfRecord::read_records(file).unwrap()
}

criterion_group!(benches, criterion_benchmark);
criterion_main!(benches);
```

- Run with `cargo bench`
- Generates HTML report with statistics and plots, compares performance to previous benchmarks
- Could also benchmark using the Python API, e.g. using *hyperfine*

Benchmarking

(using *hyperfine* https://github.com/sharkdp/hyperfine)

File type	# of records	Size (MB)
Small rawacf	1423	50
Small fitacf	1423	7.1
Large rawacf	30592	814
Large fitacf	30592	141



Notes on benchmarking

The Rust code is faster than C equivalent, even single-threaded

- slower when passing the data from Rust to Python

All tests conducted on:

- OS: openSUSE Leap 15.4
- CPU: Intel(R) Core(TM) i7-8700K @ 3.70 GHz, 12 core
- Python version: 3.8
- Files on SSD with SATA connection

Summary

- Rust can be a great tool for accelerating Python packages
- Testing is simple, parallelization is simple, benchmarking is simple
- For pyDARNio, saw up to 20x speedup with Rust
- Tools exist for automating builds and publishing to PyPI, making installation easy for everyone

Thanks to Funding Agencies

Links

pyDARNio documentation:
https://pydarnio.readthedocs.io/en/latest/

Rust code:
https://github.com/SuperDARNCanada/dmap