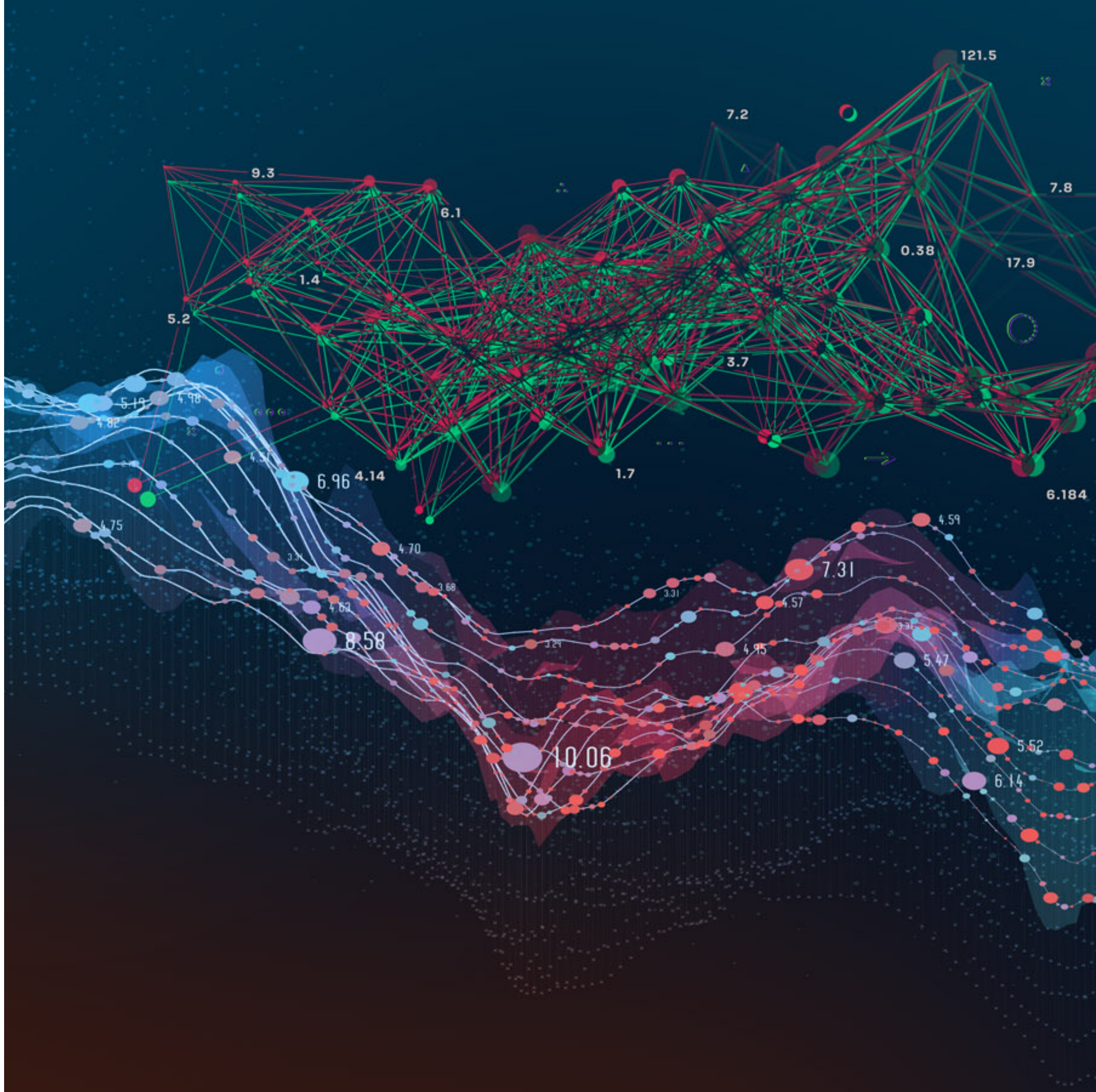
MCL + Rust

PPoPP '22

Ryan Friese, Roberto Gioiosa, Alok Kamatar



Pacific Northwest NATIONAL LABORATORY

RUST High Level Overview

Why Rust?

Rust is a modern systems programing language that is an alternative to C and C++.
It prioritizes reliability, performance, and productivity.

Reliability

- Memory safety
- Thread safety
- Strong compile-time guarantees

Productivity

- **Cargo** build tool/package manager
- Verbose and helpful compiler errors
- rustfmt

"Fearless Concurrency!"

Performance

- No garbage collector
- Zero-cost abstractions
- No runtime required

Rust Memory Model

Rust's memory model is inspired by **Cyclone**:

> **Cyclone: a Type-safe Dialect of C**. Dan Grossman, Michael Hicks, Trevor Jim, and Greg Morrisett. C/C++ Users Journal, 23(1), January 2005.
>
> *http://www.cs.umd.edu/~mwh/papers/cyclone-cuj.pdf*

Cyclone introduced the idea of a region-based type system, which in Rust is expressed through what is known as a **borrow-checker**.

Variables are **immutable by default**, and have lifetimes that are scoped by memory regions (typically, blocks of code such as functions).

We need to **borrow ownership** of a variable if we are to access it outside of its defining scope and explicitly declare values as mutable if we wish to change them.

Rust RAII

Rust enforces **RAII** (Resource Acquisition Is Initialization).

Whenever an object goes out of scope, its destructor is called, and its owned resources are freed.

Allows us to automate aspects of memory management at compile-time, without explicit calls to free() or using a Garbage Collector (GC).

This reduces wide classes of memory faults that are caught at compile time (double free, use after free, resource leaks) while preserving performance (no stop the world GC).

Variable assignment has **move semantics**, unless primitive values have known size (e.g., fixed # of words) and defaults to stack allocation.

Ownership Rules

- Each value in Rust has a variable that’s called its *owner*.
- There can only be one owner at a time.
- When the *owner* goes out of scope, the value will be dropped (destructor is called, owned resources freed).

```rust
let s1 = String::from("hello");
let s2 = s1;
println!("{}, world!", s1);
```

```
error[E0382]: use of moved value: `s1`
 --> src/main.rs:5:28
  |
3 |     let s2 = s1;
  |         -- value moved here
4 |
5 |     println!("{}, world!", s1);
  |                            ^^ value used
here after move
  |
  = note: move occurs because `s1` has type
`std::string::String`, which does
  not implement the `Copy` trait
```

Rust Type System

Rust is:

- Strongly typed.
- Statically typed.
- With type inference.

Some types available include:

- Scalar types (e.g., integer, floating point, Boolean)
- Compound types (e.g., tuples, arrays)
- Enums (similar to algebraic data types (ADTs) from Ocaml, Haskell)
- Standard Collections (Vector, HashMap, etc.).
- Smart Pointers

References and Borrowing

Default parameter use in function **call by value**:

- A function F takes ownership of values placed in its call stack:
  - F will **copy** the value if its size is statically allocated.
  - F will **move** the value if it is size is dynamically allocated.

- If moved, the value will:
  - Go out of scope at the end of the called function.
  - Be automatically deallocated.

References and Borrowing

This is memory safe, but not always convenient.

- We can refer to a value `val` by using the syntax `&val` without taking ownership of it.
- This allows the function argument value to **stay in scope** after the function returns.
- References are *immutable* by default.
- Explicit *mutable* references (`&mut val`) are allowed if we want to alter the value in the new scope.
  - Compiler ensures there is only one mutable reference to a variable at any given time
  - Unless explicitly marked as a "sync" data structure (e.g. Atomic Integers)

Rust Structs

Rust structs are named tuples:

```
struct ActiveMessage {

      source_locale: u64,
      dest_locale: u64,
      data: Vec<u64>,
      one_sided : bool,
}
```

Rust Enums

Similar to algebraic data types (ADTS) from Ocaml or Haskell:

```
•         enum ConduitKind {
                  IBV,
                  OFI,
                  UGNI,
          }

          fn send(conduit_type: ConduitKind, data: [u64]) {

                  match type {
                    ConduitKind::IBV  =>  send_ibv(data),
                    ConduitKind::OFI  =>  send_ofi(data),
                    ConduitKind::UGNI =>  send_ugni(data),
                    _ => (),
                  }
          }
```

- Matches are exhaustive, so all cases of the defining **enum** need to be covered.
- The _ operator is a catch-all.

Pacific Northwest NATIONAL LABORATORY

Rust Traits

- **Traits** are (if you squint) similar to type classes in Haskell or Interfaces in Java and Go
- Allows for default method implementations.
  - Attached to the trait itself.
- Allows for generic programming.
  - Implementations may specialize behavior
- By default, Rust monomorphizes called instances of generic functions at compile time for static dispatch.
  - Can result in larger binaries
  - Dynamic dispatch is available but impacts performance (i.e., vtable)
- Rust trait objects are similar to C++ classes, but:
  - Not possible to derive functionality through inheritance, only through implementation
- **Composition** Over **Inheritance** OOP design principle.
  - No multiple inheritance. . . But a data object can *impl* many traits
  - Rust structs can not inherit other structs

Rust Traits

```
enum Option<T> {
    Some(T),
    None,
}

pub trait Ordered {
   fn greater(&self, &other) -> u64;
}

fn largest<T: Ordered>( list: &[T] ) -> T {
    let mut largest = list[0];
    for &item in list.iter() {
        if item.greater(&largest) {
            largest = item;
        }
    }
    largest
}
```

<T> is a placeholder for a generic trait.

<T: Ordered> means the generic must implement **Ordered**. This is called a trait bound

Rust Traits – Deriving functionality

- There exist a subset of Traits that are so common, the language provides a mechanism to automatically *derive* default implementations at compile time
- This is accomplished by preceding your data structure declaration with the #[derive(…)] macro
- Derivable traits:
  - Debug – debug formatting for printing
  - PartialEq, Eq – equality comparisons
  - ParialOrd, Ord – ordering comparisons
  - Clone,Copy – duplicating data
  - Hash – mapping to value of fixed size
  - Default – default values
- 3rd party crates also introduce their own derivable traits
  - e.g. #[derive(serde::serialize)]

```rust
#[derive(Clone,Debug,Hash)]
enum Cmd {
    Send,
    Recv,
}

#[derive(Clone,Debug)]
struct Packet<T> {
    cmd: Cmd
    msg: &str
    payload: T
}
```

<T> in this case must also *impl* Clone and Debug, otherwise we will get a compiler error

- It is always possible to manually derive any of these traits based on your use case

Rust Macros

- Macros are a significant part of the Rust language
  - We just saw a very useful Macro on the previous slide #[derive]
  - Even the simplest "Hello, World" application uses a macro
- Macros at their heart are a way of writing code that writes other code
- Macros vs Functions (high level)
  - Function signatures must declare the number and type of parameters
  - Macros can take a variable number of parameters
    - ✓ Macros are evaluated (expanded) before compiler begins
- Two types of Macros
  - Declarative macros (macro_rules!) – essentially perform pattern matching and replacement
  - Procedural macros – more function like: accepts code, dynamically operates on/transforms code (you have access to AST), produce new code
- Macro implementation can be difficult to develop, read, and maintain

This is a macro!

```
fn main{
    println!( "Hello, world" );
}
```

1 argument

2 arguments

```
fn main{
    println!( "Hello {}", "world" );
}
```

Additional Features

- **Associated Types** are type placeholders within a trait definition.
  - Allow the implementation to specify the concretized type value.
  - For example:
    - `.next()` method definition in an **Iterator** trait.
    - Operator Overloading (+,-, …)
- **Closures** are lambda expressions (anonymous functions) that capture the enclosing environment.
  - Closures can be saved in a variable
  - Closures can be passed as arguments to other functions.
- **Smart Pointers** to force:
  - Heap Allocation
  - Explicit reference counting (and thread safe atomic versions)
  - Runtime borrow checking
  - Explicit deferencing (destructor-like behavior).

Closures and Iterators

Closures are lambda expressions allow us to define anonymous functions that capture the enclosing environment. We can then save the closure in a variable or pass it as arguments to other functions:

```
let square_x = |x| { x * x }

let accum_x = | x , a | { x + a }
```

Higher Order Functions (HOFs)

HOFs take one or more functions as arguments. Combining with lazy iterators gives a somewhat functional flavor:

```
fn is_odd(n: u32) -> bool {
    n % 2 == 1
}

let sum_of_squared_odd_numbers: u32 =
        (0..).map(|n| n * n)                    // All natural numbers squared
             .take_while(|&n_squared| n_squared < upper) // Below upper limit
             .filter(|&n_squared| is_odd(n_squared))     // That are odd
             .sum(); // Sum them
```

Rust Error Handling

Rust enforces error handle (or at least explicit acknowledgement you are ignoring an error)

```
enum Result<T, E> { Ok(T), Err(E), }

use std::fs::File;

fn main() {
  let f = File::open("hello.txt");
  let f = f.unwrap(); //MINIMUM – ignore an error may happen, panics with generic message
  let f = f.expect(“error better opening file"); //BETTER – panics with custom error message
  let f = match f {
    Ok(file)   => file,
    Err(error) => {                            // BEST – explicitly handle/recover
      panic!("There was a problem opening     // from an error or at lease provide
              the file: {:?}", error)        // more detailed error message on panic
    },
  };
}
```

Rust Concurrency

Rust has support for message passing inspired by **golang** via a multiple producer single consumer (mpsc) model standard library feature.

```
use std::thread; //explicit OS level thread (e.g. pthreads)
use std::sync::mpsc;

fn main() {
  let (tx, rx) = mpsc::channel();
  thread::spawn( move || {
    let val = String::from("hi");
    tx.send(val).unwrap();
   }
  );
}
```

The `move` keyword moves tx into the spawned thread.

Async/await design pattern for futures is also available in the standard library

Somewhat similar to green threads or user level threads

Rust Build Environment: Cargo

- Cargo is the build tool for Rust.
- Provides a canonical package layout and manifest.
- Includes robust dependency management
  - Download/resolution of libraries
  - Compilation of dependencies
  - Environment support (debug/release, benchmarks, integration tests)
- Integration of foreign code with a Rust project requires leveraging and working within the conventions of the Cargo package manager
  - There is built-in support to do this. . .
  - And there are convenience crates (libraries) such as CC

Cargo Structure

- Cargo.toml and Cargo.lock are stored in the root of your package (*package root*).
- Source code goes in the src directory.
  - The default library file is src/lib.rs.
  - The default executable file is src/main.rs.
- Integration tests go in the tests directory
  - *Unit tests go in each file they're testing*.
- Benchmarks go in the benches directory.

```
.
├── Cargo.lock
├── Cargo.toml
├── benches
│   └── large-input.rs
├── examples
│   └── simple.rs
├── src
│   ├── bin
│   │   └── another_executable.rs
│   ├── lib.rs
│   └── main.rs
└── tests
    └── some-integration-tests.rs
```

Cargo Basics

- To start a new project:

```
cargo  new [--library]  my_project
```

- The **Cargo.toml** file lists the top-level dependencies of an existing project.
  - Allows for external libraries, version pinning, etc.
- The Cargo.lock file captures the exact environment used for a build

- To build a project:

```
cargo build [--release]
```

- Dependencies will be downloaded, built, linked, etc. . .

MCL + Rust

C – Rust FFI

- What is an FFI?
  - Foreign function interface – the mechanism in which a program in one language can call routines and service written in another
- But… Rust is a “safe” language… C… isn’t…
  - Unfortunately, this is true, once a rust program calls into a C library all safety guarantees are off
- So why have a Rust interface?
  - We can limit the “unsafe” code to only the C library itself
    - ✓ In an ideal world the library is completely bug free…
  - User level code benefits from all the features of Rust
- Great.. But can’t you just port the C library to Rust?
  - Yup! And some libraries have certainly done that (time + money…)
  - But at some level you will eventually need to call in to a C library (e.g. libc)
    - ✓ Unless you use a Rust OS (these are very young)

Rust “sys” Crates

- The idiomatic way to implement a C library FFI is via a “sys” crate
  - For MCL we have implemented “libmcl-sys”
- At a high level this is simply a mapping of the C types and the Rust types of the public interfaces in the library
  - E.g

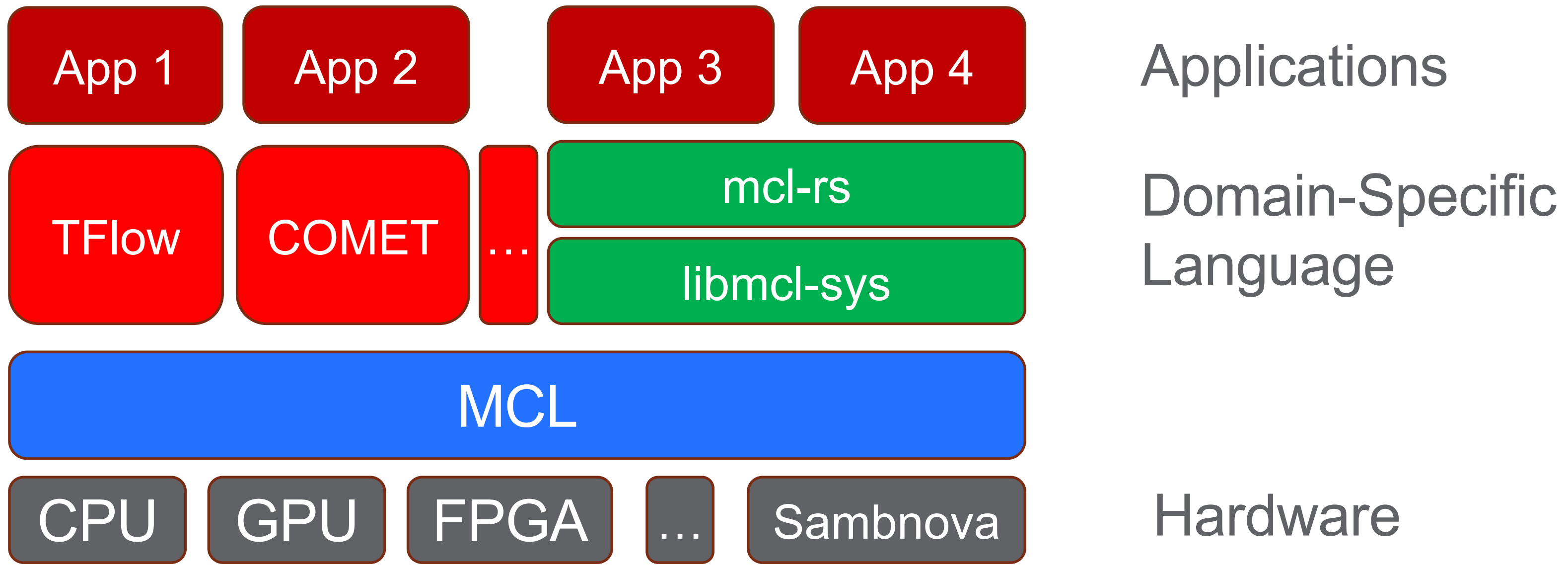
```c
/**
 * @brief Initialize MCL
 *
 * @param num_workers Number of …
 * @param flags Unimplemented
 * @return int 0 on success, non-zero otherwise
 */
int        mcl_init(uint64_t num_workers, uint64_t flags);
```

```rust
extern "C" {
    #[doc = " @brief Initialize MCL"]
    #[doc = ""]
    #[doc = " @param num_workers Number of …"]
    #[doc = " @param flags Unimplemented"]
    #[doc = " @return int 0 on success, non-zero otherwise"]
    pub fn mcl_init(num_workers: u64, flags: u64) -> ::std::os::raw::c_int;
}
```

- The Rust Bindgen assists in automatically generating most of these bindings
- The resultant “sys” crate provides unsafe Rust compatible interfaces
- It also appropriately links the library so it can be used in any other Rust app
- It is common for developers to also a safe interface for the library using the “sys” crate as a dependency

MCL + Rust

- We have implemented two crates to enable MCL programming in Rust
  - libmcl-sys – the low-level unsafe Rust-C bindings
    - ✓ Not really intended for direct user interaction
  - mcl-rs – high level safe abstractions
    - ✓ Fits into the Domain-Specific language layer in the MCL stack
    - ✓ We want to program at this layer!



Enough talk… lets see some code!

- Lets start with a simple mcl-rs "vector_add" – our version of "hello world"
- Step 0 – install Rust –

```
$ curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

- Step 1 – create a new Rust project

```
$ cargo new vector_add --bin
```

  - Step 1.5 lets see what happened
    - ✓ Lets peek in main.rs
    - ✓ Everything we need to build and execute a Rust app!

```
$ cd vector_add
vector_add$ tree .
.
├── Cargo.toml
└── src
    └── main.rs

1 directory, 2 files
```

```
fn main() {
    println!("Hello, world!");
}
```

- Step 2 – lets build it!

```
vector_add$ cargo build
Compiling vector_add v0.1.0 (/…/vector_add)
Finished dev [unoptimized + debuginfo] target(s) in 1.19s
```

- Step 3 – lets run it!

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.03s
    Running `target/debug/vector_add`
Hello, world!
```

What about MCL?

- Step 4 – include “mcl-rs” as a dependency for our App
  - Open “Cargo.toml”
  - Add mcl-rs=0.1 to dependency section

```
//Cargo.toml
[package]
name = “vector_add"
version = "0.1.0"
edition = "2021"
[dependencies]
mcl-rs = "0.1.2”
```

- Step 5 – make sure we can build with mcl-rs
  - Cargo automatically downloads our declared dependencies and all “inherited dependencies
  - Uhh ohh!!!
  - Depending on your system this error may not have happened (OCL was found)
  - Let try again with env vars set to proper locations
  - Dang it!
  - One last time setting the MCL_PATH

```
vector_add$ cargo build
   Updating crates.io index
   Compiling libc v0.2.121
   Compiling autocfg v1.1.0
   …
```

```
error: failed to run custom build command for `libmcl-sys v0.1.0`
Please set the paths to OpenCL: env variables OCL_PATH_INC, OCL_PATH_LIB
```

```
Vector_add$ export OCL_PATH_INC=${..} && export OCL_PATH_LIB={…}
vector_add$ cargo build
   error: failed to run custom build command for `libmcl-sys v0.1.0`
MCL_PATH environmental variable is not set     …
```

```
vector_add$ MCL_PATH=${mcl_install_path} cargo build
    Compiling libmcl-sys v0.1.0
    Compiling mcl-rs v0.1.0
    Compiling vector_add v0.1.0
   Finished dev [unoptimized + debuginfo] target(s) in 4.88s
```

SUCCESS!

Now we can actually do MCL code

- Step 6 – add vadd.cl to the src folder

```
__kernel void VADD (__global int* out, __global int* x, __global int* y)
{
    const int i = get_global_id(0);
    out[i] = x[i] + y[i];
}
```

- Step 7 – lets update main.rs
  - 7.1 – include the mcl-rs module
  - 7.2 – initialize mcl
    - ✓ Builders are a common Rust design pattern
    - ✓ No need for explicit "finit" mcl-rs handles automatically

```
use mcl_rs; //7.1 imports the module

fn main() {
 let mcl = rust_rs::MclEnvBuilder::new()
    .num_workers(workers) //number of threads
    .initialize();
} //mcl is "dropped" here
```

  - 7.3 – lets test everything is working
    - ✓ Cargo does a lot… but it struggles embedding paths to dynamic libraries

```
vector_add$ cargo run
error: while loading shared libraries: libmcl.so.0: cannot open shared object file
```

    - ✓ Ensure LD_LIBRARY_PATH contain your MCL lib path (and OpenCL)
    - ✓ And again…
      - So close, but we forgot to start "mcl_sched"

```
vector_add$ cargo run
[MCL ERR][…/src/lib/core.c 554] Error opening shared memory object mcl_shm.
shm_open: No such file or directory
[MCL ERR][…/src/lib/api.c 322] Error setting up MCL library. Aborting.
thread 'main' panicked at 'Error -1. Could not initialize MCL', …/mcl-rs/src/lib.rs:1836:17
```

```
vector_add$ path/to/mcl_sched &
vector_add$ cargo run
 Finished dev [unoptimized + debuginfo] target(s) in 0.19s
    Running `target/debug/vector_add`
```

No errors!

Implementing Vector Add

- First let’s implement a sequential version on the CPU
- Build and run to check

```
vector_add$ vector_add$ cargo run
 Finished dev [unoptimized + debuginfo] target(s) in 1.12s
     Running `target/debug/vector_add`
z= [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
```

```rust
use mcl_rs; //imports the module

fn add_seq(x: &[i32], y: &[i32], z: &mut [i32]) {// z must be declared as
    for i in 0..z.len(){                         // mutable so we can update
        z[i] = x[i] + y[i];
    }
}

fn main() {
    let _mcl = mcl_rs::MclEnvBuilder::new()
    .num_workers(1)
    .initialize();

    let vec_size = 10;

    let x: Vec::<i32> = (0..vec_size).map(|_| 1).collect(); //vector of ones
    let y: Vec::<i32> = (0..vec_size).map(|_| 2).collect(); //vector of twos

    let mut z = vec![0; vec_size]; //the compiler will infer the correct type

    add_seq(&x, &y, &mut z); // a ref of a Vec<T> coerces into a slice &[T]
    println!("z= {:?}",z);
}
```

Implementing MCL Vector Add

- Now for the good part!
- MCL + Rust!
- We use our “mcl” object to create and execute a task
- This should look somewhat familiar
- Lets run it!

```
__kernel void VADD (__global int* out, __global int* x, __global int* y)
{
    const int i = get_global_id(0);
    out[i] = x[i] + y[i];
}
```

```
use mcl_rs::{TaskArg,DevType}; //specific structs
we will use


fn main() {
    …
     //all the code from previous slide
    let vec_size = 10;
    let mut z_mcl = vec![0; vec_size];

    let global_work_dims= [vec_size as u64,1,1];

    //create and execute an MCL task
    mcl.task("src/vadd.cl", "VADD", 3)
    .arg(TaskArg::output_slice(&mut z_mcl))
    .arg(TaskArg::input_slice(&x))
    .arg(TaskArg::input_slice(&y))
    .dev(DevType::ANY)
    .exec(global_work_dims)
    .wait();
    println!("z_mcl= {:?}",z_mcl);
    assert_eq!(z, z_mcl);
}
```

Path to *.cl file, Kernel Name, Num Arguments

Equivalent to calling mcl_task_set_arg with MCL_ARG_OUTPUT|MCL_ARG_BUFFER flags

Equivalent to calling mcl_task_set_arg with MCL_ARG_INPUT|MCL_ARG_BUFFER flags

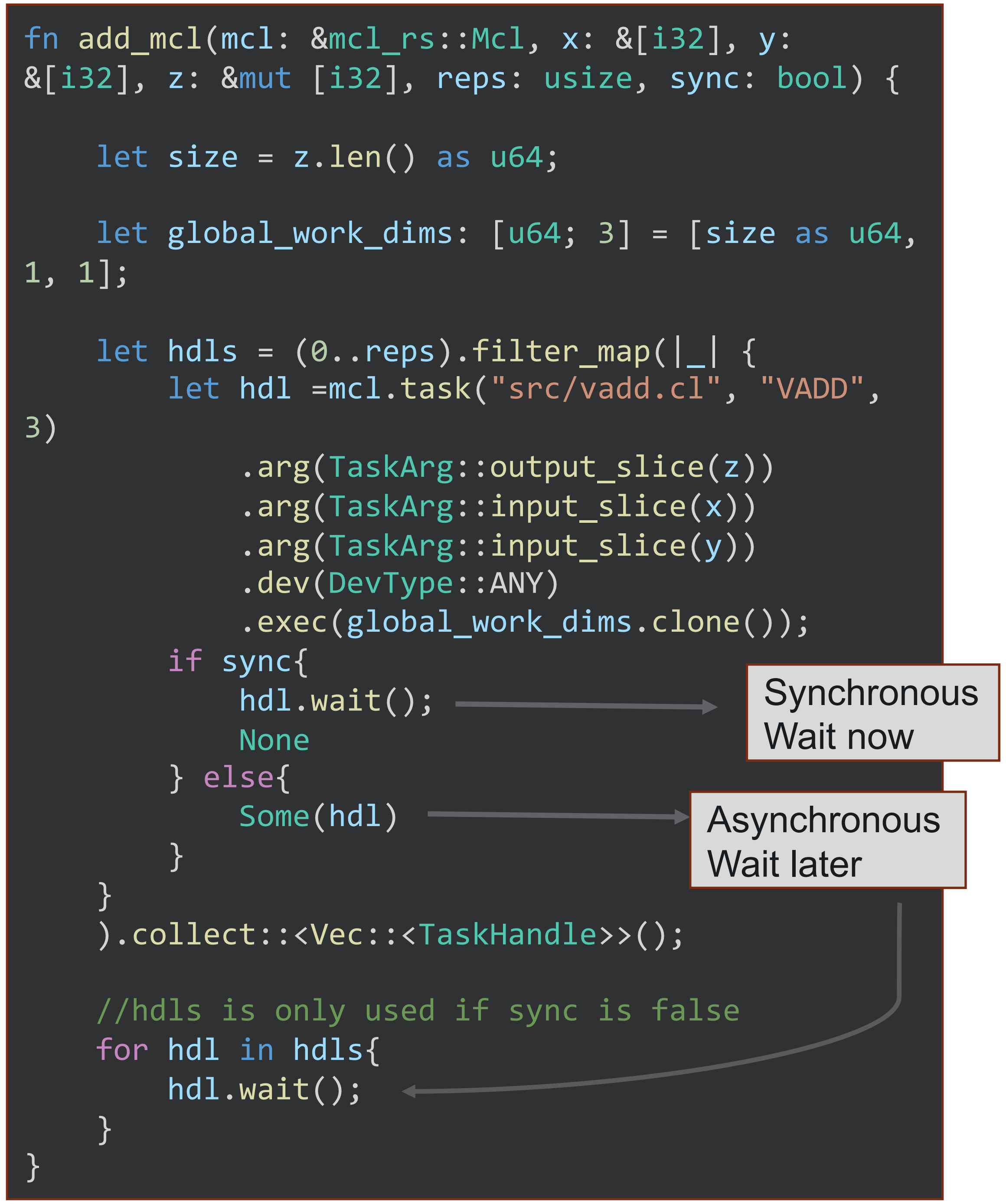
Equivalent to the MCL_TASK_ANY flag

Synchronous execution

```
vector_add$ vector_add$ cargo run
 Finished dev [unoptimized + debuginfo] target(s) in 1.12s
    Running `target/debug/vector_add`
z= [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
z=mcl= [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
```

How about async?

- Really no different, we just don't immediately wait on the handle
- I hear Rust has really nice async/await functionality… do you support it?
  - Rust does have nice async/await!
  - Not yet... but we will in a *future* (no pun intended) release



```
fn add_mcl(mcl: &mcl_rs::Mcl, x: &[i32], y:
&[i32], z: &mut [i32], reps: usize, sync: bool) {

    let size = z.len() as u64;

    let global_work_dims: [u64; 3] = [size as u64,
1, 1];

    let hdls = (0..reps).filter_map(|_| {
        let hdl =mcl.task("src/vadd.cl", "VADD",
3)
            .arg(TaskArg::output_slice(z))
            .arg(TaskArg::input_slice(x))
            .arg(TaskArg::input_slice(y))
            .dev(DevType::ANY)
            .exec(global_work_dims.clone());
        if sync{
            hdl.wait();
            None
        } else{
            Some(hdl)
        }
    }
    ).collect::<Vec::<TaskHandle>>();

    //hdls is only used if sync is false
    for hdl in hdls{
        hdl.wait();
    }
}
```

Testing the final version!

- Performing some simple timing of our different approaches

```
vector_add$ vector_add$ cargo run
Compiling vector_add v0.1.0
Finished dev [unoptimized + debuginfo] target(s) in 1.22s
Running `target/debug/vector_add`
seq time: 28.968750326 z[0..10] = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
mcl sync time: 8.169609996 z[0..10] = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
mcl async time: 5.779657659 z[0..10] = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
```

- Both MCL are faster!
  - Running on a GPU
- Async faster than Sync
  - As expected!

```rust
fn main() {
    let mcl = mcl_rs::MclEnvBuilder::new().num_workers(10).initialize();

    let vec_size = 1000000;

    let x: Vec<i32> = (0..vec_size).map(|_| 1).collect(); //vector of ones
    let y: Vec<i32> = (0..vec_size).map(|_| 2).collect(); //vector of twos
    let mut z = vec![0; vec_size]; //the compiler will infer the correct type
    let mut z_mcl_sync = vec![0; vec_size]; //the compiler will infer the correct type
    let mut z_mcl_async = vec![0; vec_size]; //the compiler will infer the correct type

    let reps = 1000;

    let mut timer = Instant::now();
    for _i in 0..reps {
        add_seq(&x, &y, &mut z);
    }
    println!("seq time: {} z[0..10] = {:?} ",timer.elapsed().as_secs_f64(),&z[0..10]);

    timer = Instant::now();
    add_mcl(&mcl, &x, &y, &mut z_mcl_sync,reps, true);
    println!("mcl sync time: {} z[0..10] = {:?} ",timer.elapsed().as_secs_f64(),&z[0..10]);

    timer = Instant::now();
    add_mcl(&mcl, &x, &y, &mut z_mcl_async,reps, false);
    println!("mcl async time: {} z[0..10] = {:?} ",timer.elapsed().as_secs_f64(),&z[0..10]);
}
```

But wait…

- I keep seeing “unoptimized” and “target/debug” everytime I compile and run…
- By default cargo builds in debug mode with limited optimizations

```
vector_add$ vector_add$ cargo run
Compiling vector_add v0.1.0
Finished dev [unoptimized + debuginfo] target(s) in 1.22s
Running `target/debug/vector_add`
seq time: 28.968750326 z[0..10] = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
mcl sync time: 8.169609996 z[0..10] = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
mcl async time: 5.779657659 z[0..10] = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
```

- We can build in “release” mode to enable optimizations (at the expense of compile time)
  - In some cases this can be and order of magnitude longer…

```
vector_add$ vector_add$ cargo run --release
Compiling vector_add v0.1.0
Finished release [optimized] target(s) in 4.04s
Running `target/release/vector_add`
seq time: 0.814003001 z[0..10] = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
mcl sync time: 7.938727178 z[0..10] = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
mcl async time: 5.619023865 z[0..10] = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
```

Utilizing CPU SIMD instructions + no data movement

TL;DR Recap MCL + Rust

- Add “mcl-rs” to dependency section of Cargo.toml file
- Set env vars
  - OCL_INC_PATH: OpenCL include directory
  - OCL_LIB_PATH: OpenCL lib directory
  - MCL_PATH: Mcl install directory (contains include & lib)
  - Ensure $OCL_LIB_PATH and $MCL_PATH/lib are in the LD_LIBRARY_PATH
- Launch mcl_schec: $MCL_PATH/bin/mcl_sched
- Build “cargo build [--release]” or
- Run “cargo run [--release]”

Thank you!

Our Rust code is hosted in our main MCL GitHub repository

https://github.com/pnnl/mcl

libmcl-sys and mcl-rs are available on crates.io

https://crates.io/crates/libmcl-sys

https://crates.io/crates/mcl-rs

Documentation is available at:

https://docs.rs/libmcl-sys

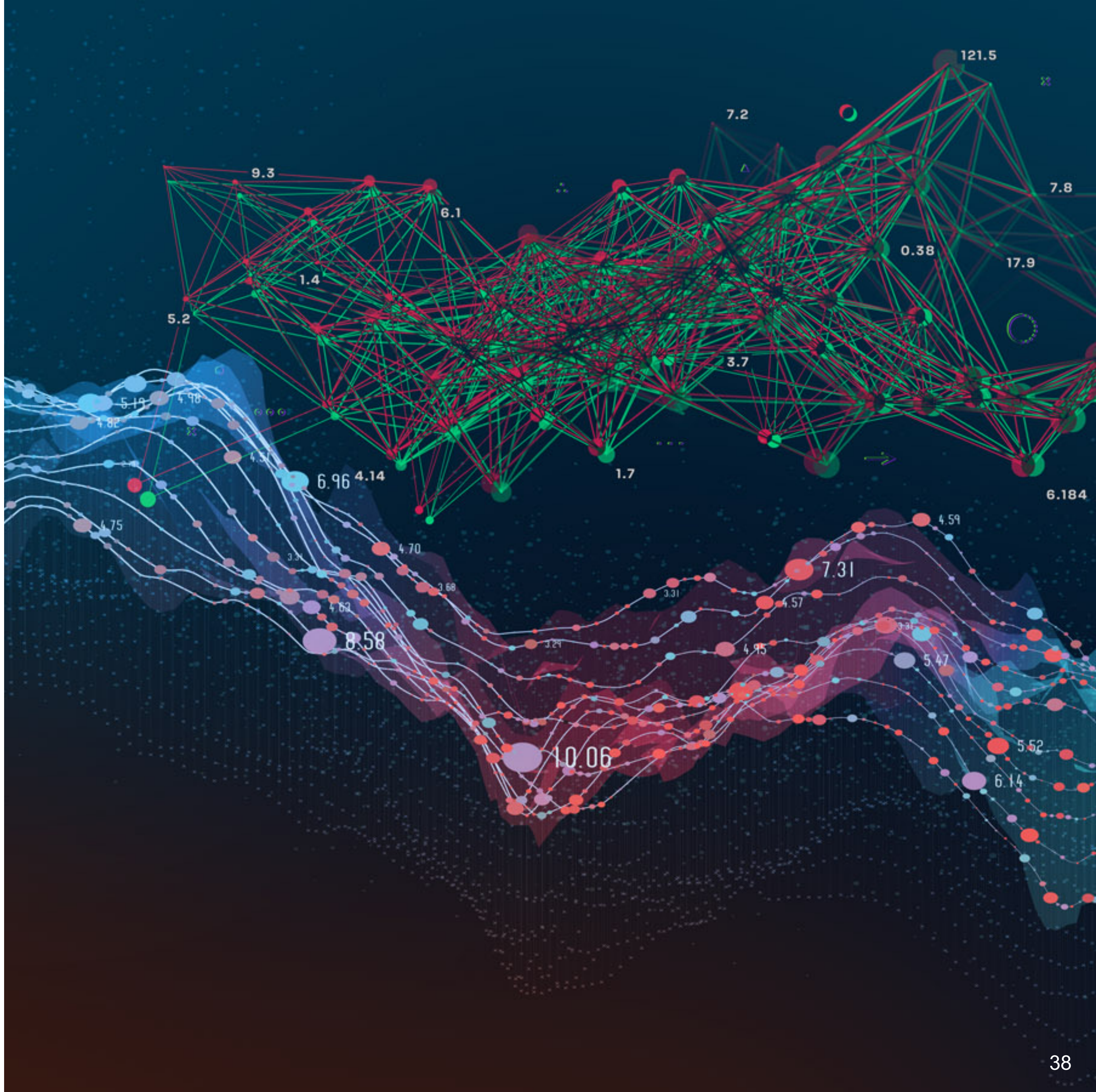
https://docs.rs/mcl-rs

Please feel free to reach out if you are interested in MCL/Rust

ryan.friese@pnnl.gov

Pacific Northwest NATIONAL LABORATORY

Thank you



BACKUP MATERIAL

Reference Lifetimes

Each reference has a lifetime, which *usually* can be elided.

Lifetimes can be made explicit with **lifetime traits**: <‘a>

- Can be subtyped to specify relationship between lifetimes: *one lifetime can explicitly be made to outlive another*.
- Can be given bounds*: help verify references in generic types will not outlive the data they refer to*.
- Can be inferred with traits.

This is useful when fighting the borrow checker…

```
struct Ref<'a, T>(&'a T);
struct Ref<'a, T: 'a>(&'a T);
struct StaticRef<T: 'static>
    (&'static T);
```

T must live as long as `'a`

T and its referents must live for the entire program.

Associated Types

A type placeholder can be *associated* with a trait, whose **concrete type** will be specified in an implementation:

```
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}

impl Iterator for Counter {
    type Item = u32;
    fn next(&mut self) -> Option<Self::Item> {
    . . .
    }
}
```

Operator Overloading works this way…

```
trait Add<RHS=Self> {
    type Output;
    fn add(self, rhs: RHS) -> Self::Output;
}
```

Smart Pointers

- Heap allocators: Box<T> Allows for recursive data types.
- Reference counting pointers: Rc<T>
- Runtime borrow checking: Ref<T>, RefMut<T>
- Dereference operator * and custom **Deref** trait impls.

```
enum List {
    Cons(i32,
    Box<List>),
    Nil,
}
use crate::List::{Cons,
Nil};
```

```
use std::ops::Deref;

impl<T> Deref for FooBox<T>
{
    type Target = T;

    fn deref(&self) -> &T {
        &self.0
    }
}
```

Cargo Basics

```
$ cargo help
Rust's package manager


USAGE:
    cargo [OPTIONS] [SUBCOMMAND]


OPTIONS:
    -V, --version           Print version info and exit
        --list              List installed commands
        --explain <CODE>    Run `rustc --explain CODE`
    -v, --verbose           Use verbose output (-vv very verbose/build.rs output)
    -q, --quiet             No output printed to stdout
        --color <WHEN>      Coloring: auto, always, never
        --frozen            Require Cargo.lock and cache are up to date
        --locked            Require Cargo.lock is up to date
    -Z <FLAG>...            Unstable (nightly-only) flags to Cargo, see 'cargo -Z help'
for details
    -h, --help              Prints help information
```

Cargo Basics

```
Some common cargo commands are (see all commands with --list):
    build       Compile the current package
    check       Analyze the current package and report errors, but don't build object
files
    clean       Remove the target directory
    doc         Build this package's and its dependencies' documentation
    new         Create a new cargo package
    init        Create a new cargo package in an existing directory
    run         Run a binary or example of the local package
    test        Run the tests
    bench       Run the benchmarks
    update      Update dependencies listed in Cargo.lock
    search      Search registry for crates
    publish     Package and upload this package to the registry
    install     Install a Rust binary. Default location is $HOME/.cargo/bin
    uninstall   Uninstall a Rust binary


See 'cargo help <command>' for more information on a specific command.
```

Rust Error Handling

This code pattern winds up being boilerplate often enough that an **unwrap** and **expect** convenience methods are provided in the Result type:

```
use std::fs::File;

fn main() {
  let f = File::open("hello.txt").unwrap();
}

fn main() {
  let f = File::open("hello.txt").expect(
    ("Failed to open hello.txt" );
}
```

Rust Error Handling

Errors can be propagated explicitly or via the **?** operator:

```rust
use std::io;
use std::io::Read;

use std::fs::File;

fn read_username_from_file() -> Result<String, io::Error> {
  let mut s = String::new();
  File::open("hello.txt")?.read_to_string(&mut s)?;
  Ok(s)
}
```

This function will short-circuit return if error condition occurs on lines terminated with ? and will return Ok(s) otherwise.