Introduction to High-Performance Parallel Distributed Computing using Chapel, UPC++, and Coarray Fortran

ECP/NERSC/OLCF 2023 Tutorial
30-minute Intro session
go.lbl.gov/cuf23

Introduction to High-Performance Parallel Distributed Computing using Chapel, UPC++ and Coarray Fortran

Dr. Michelle Mills Strout

Dr. Damian Rouson

Dr. Amir Kamil

Other Contributors:

Dan Bonachea, Jeremiah Corrado, Paul H. Hargrove, Katherine Rasmussen, Sameer Shende, Daniel Waters

Acknowledgements

This work was supported in part by the **Exascale Computing Project** (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, in support of the nation's exascale computing imperative.

This work used resources of the **National Energy Research Scientific Computing Center (NERSC)**, a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231, as well as This research used resources of the **Oak Ridge Leadership Computing Facility** at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

Schedule for Chapel, UPC++ and Fortran Tutorial

Wed July 26, noon - 3:15pm (all times US Eastern)

- noon - 1:30: Tutorial Overview
  - including a 20-minute intro to each programming model
- *1:30 - 1:45: Coffee Break*
- 1:45 - 3:15: Parallel programming in Chapel

Thu July 27, noon - 3:15pm

- noon - 1:30: Parallel programming with UPC++
- *1:30 - 1:45: Coffee Break*
- 1:45 - 3:15: Parallel programming with Fortran Coarrays

Audience questions

Slack is preferred:
go.lbl.gov/cuf23-slack

alternatively use Zoom chat

Motivation

- You have …
  - A lot of data to process and analyze
  - A big simulation to run
  - Or both of the above
- Resources are available
  - Your laptop has multiple cores that can process in parallel
  - Your lab/institution has a cluster
  - Or your lab/institution has a supercomputer
- Writing a parallel program enables you to analyze data and/or perform simulations significantly faster.

Which programming language(s) do you use the most? (you can respond to this question 3 times)

C/C++

Fortran

Chapel

Python

Java

R

Perl

Haskell, Scala, ...

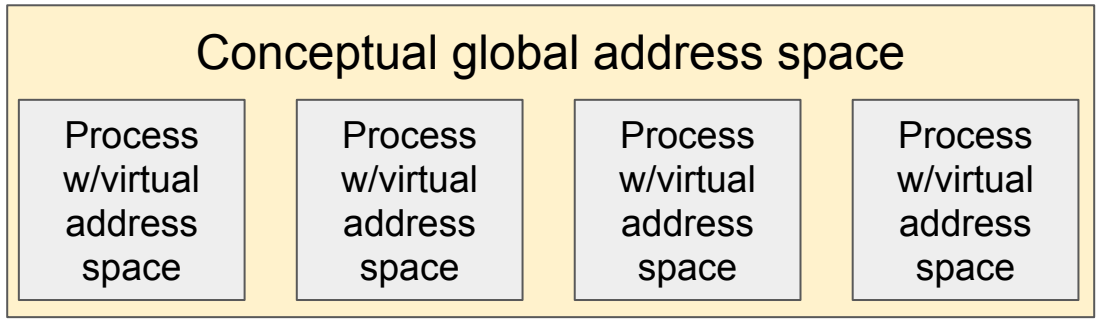
Other

PGAS Programming Models

- PGAS: Partitioned Global Address space
- **Chapel**, **UPC++**, and **Fortran with coarrays** are PGAS programming models
- A programming model provides an interface and code patterns to a programmer along with a concept of how code will execute at runtime.

PGAS Programming Models

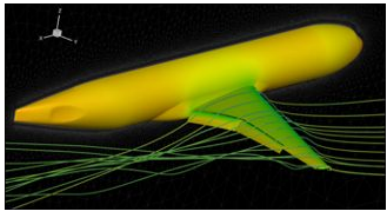
- Can access variables in global address space from each node
- Implemented with puts and gets (RMA: remote memory access)
- Can partition/organize data and computation to reduce RMA



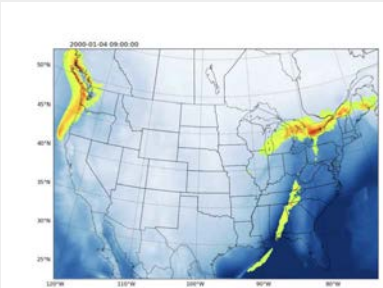
This tutorial: Chapel, UPC++, Fortran with coarrays

- Shared example shown in all three: **2D heat diffusion**
- Then other examples per programming model
  - Chapel: k-mer counting, image analysis, processing files in parallel
  - UPC++: 1-d Jacobi solver, distributed hash table
  - Fortran: 2-d heat equation, hello world variants
- Hands On
  - Providing a cloud instance, Perlmutter, and Frontier instructions for obtaining a tarball containing all example programs: [go.lbl.gov/cuf23](go.lbl.gov/cuf23)
  - You are encouraged to compile, run, and experiment with the examples throughout
- Q&A Protocol
  - Model experts are available to answer questions in Slack: go.lbl.gov/cuf23-slack
    - You should have received an email invite, or can follow the link above

Production Applications using these Programming Models



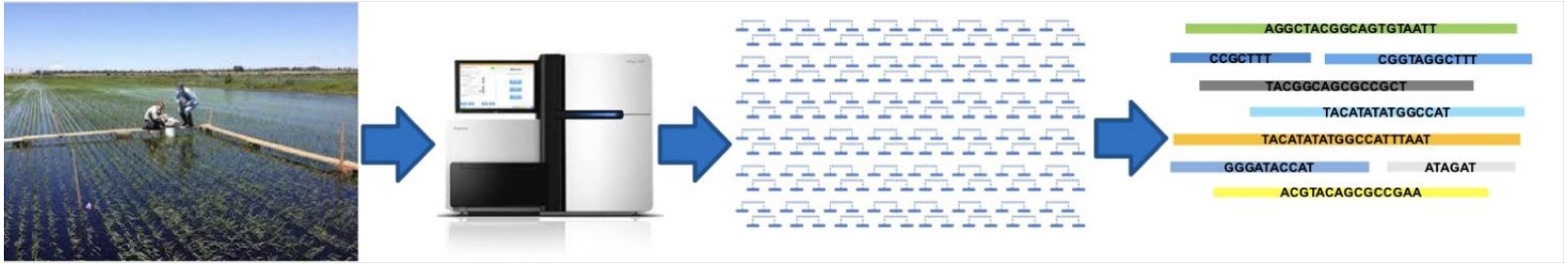
CHAMPS: 3D Unstructured CFD
(~100K lines of Chapel)
Éric Laurendeau, Simon Bourgault-Côté,
Matthieu Parenteau, et al.
École Polytechnique Montréal



ICAR: Intermediate Complexity Atmospheric Research model written in Coarray Fortran

https://github.com/NCAR/icar

MetaHipMer, a genome assembler written in UPC++



Hands On: Compiling and Running **Hello Worlds**

- Instructions on how to compile and run a **hello world** for all three programming models.
- Hands-on examples and instructions: go.lbl.gov/cuf23
  - Options include:
    - NERSC Perlmutter, OLCF Frontier, AWS Cloud, Docker, …
  - Pause here for attendees to setup their programming environment

Do you have any parallel programming experience? If so, what tools have you used?

Shared Problem: 2D Heat Diffusion

$$\frac{\partial u}{\partial t} = \nu \frac{\partial^2 u}{\partial x^2} + \nu \frac{\partial^2 u}{\partial y^2}$$

- Specifically a 2D heat diffusion problem
  - 2D diffusion equation is above. Mathematical details: wikipedia.org/wiki/Heat_equation
  - Discretization solving for the unknown at time step n+1 and spatial coordinate i,j
- Steps in sample codes
  - Set some initial conditions for u^0
  - Estimate u over time and space as shown below
  - Show how to parallelize these computations

$$u_{i,j}^{n+1} = u_{i,j}^n + \frac{\nu \Delta t}{\Delta x^2}(u_{i+1,j}^n - 2u_{i,j}^n + u_{i-1,j}^n) + \frac{\nu \Delta t}{\Delta y^2}(u_{i,j+1}^n - 2u_{i,j}^n + u_{i,j-1}^n)$$

Simplified form

assume $\Delta x = \Delta y$, and let $\alpha = \nu \Delta t / \Delta x^2$

$$u_{i,j}^{n+1} = u_{i,j}^n + \alpha \begin{pmatrix} u_{i+1,j}^n + u_{i-1,j}^n \\ -4u_{i,j}^n + u_{i,j+1}^n + u_{i,j-1}^n \end{pmatrix}$$

Three questions about how you program

- Have you used a cluster or supercomputer before? If so, what were their characteristics (number of nodes, threads per node, etc)?
- Where do you go when you have programming questions? A colleague, stack overflow, google search, documentation, …
- For your code, what computations/libraries are most important for your work?

NOTE: The pollEV survey starts on the next slide, but it won't show the above questions. This slide is to show you what those questions will be.

Three questions about how you program

0 done

0 underway

What do you want to learn about Chapel, UPC++, or Coarray Fortran today?

Top

Schedule for Chapel, UPC++ and Fortran Tutorial

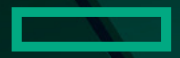
Wed July 26, noon - 3:15pm (all times US Eastern)

- noon - 1:30: Tutorial Overview, 20-minute intro to each programming model
  - **Chapel Intro**
  - **Fortran with co-arrays Intro**
  - **UPC++ Intro**
- *1:30 - 1:45: Coffee Break*
- 1:45 - 3:15: Parallel programming in Chapel

Thu July 27, noon - 3:15pm

- noon - 1:30: Parallel programming with UPC++
- *1:30 - 1:45: Coffee Break*
- 1:45 - 3:15: Parallel programming with Fortran Coarrays

go.lbl.gov/cuf23



Hewlett Packard
Enterprise

INTRODUCTION TO CHAPEL PARALLEL PROGRAMMING LANGUAGE

Michelle Strout and Jeremiah Corrado

CUF23: Sponsored by OLCF, NERSC, and ECP
July 26-27, 2023

INTRODUCTION TO CHAPEL

- What Chapel is and how programmers are using Chapel in their applications
- Chapel execution model with a parallel and distributed "Hello World"
- 2D Heat Diffusion example: variants and how to compile and run them
- Learning objectives for today's 90-minute Chapel tutorial

CHAPEL PROGRAMMING LANGUAGE

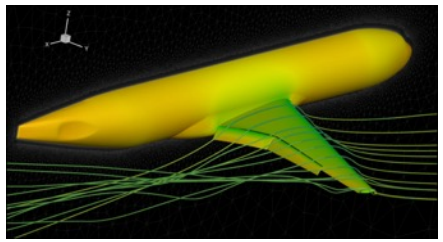
Chapel is a general-purpose programming language that provides

ease of parallel programming,
high performance, and
portability.

And is being used in applications in various ways:

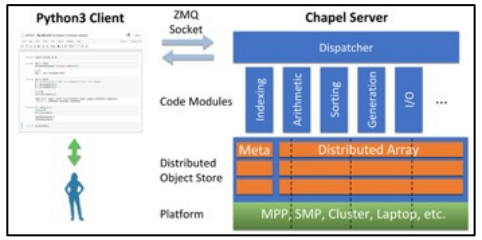
refactoring existing codes,
developing new codes,
serving high performance to Python codes **(Chapel server with Python client),** and
providing distributed and shared memory parallelism for existing codes.

APPLICATIONS OF CHAPEL: LINKS TO USERS' TALKS (SLIDES + VIDEO)



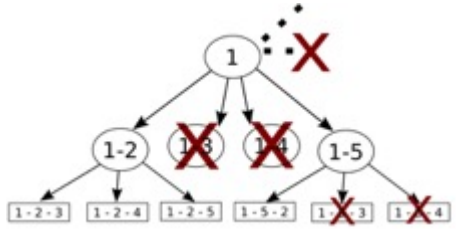
CHAMPS: 3D Unstructured CFD

CHIUW 2021 CHIUW 2022



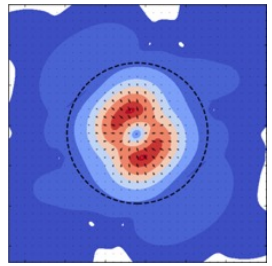
Arkouda: Interactive Data Science at Massive Scale

CHIUW 2020 CHIUW 2023



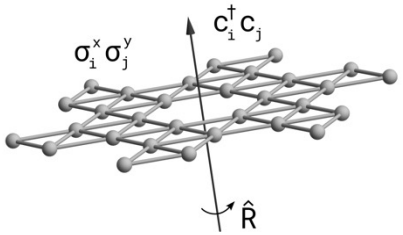
ChOp: Chapel-based Optimization

CHIUW 2021 CHIUW 2023



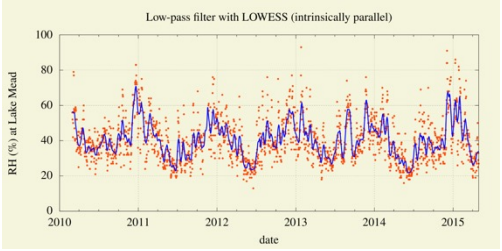
ChplUltra: Simulating Ultralight Dark Matter

CHIUW 2020 CHIUW 2022



Lattice-Symmetries: a Quantum Many-Body Toolbox

CHIUW 2022

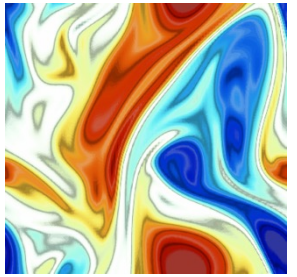


Desk dot chpl: Utilities for Environmental Eng.

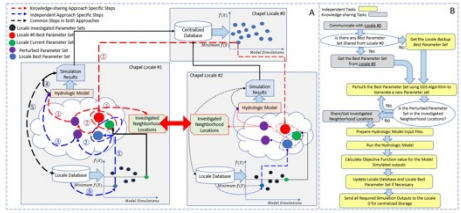
CHIUW 2022

RapidQ: Mapping Coral Biodiversity

CHIUW 2023

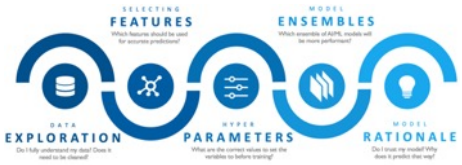


ChapQG: Layered Quasigeostrophic CFD



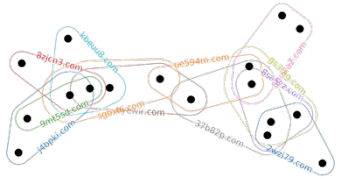
Chapel-based Hydrological Model Calibration

CHIUW 2023



CrayAI HyperParameter Optimization (HPO)

CHIUW 2021



CHGL: Chapel Hypergraph Library

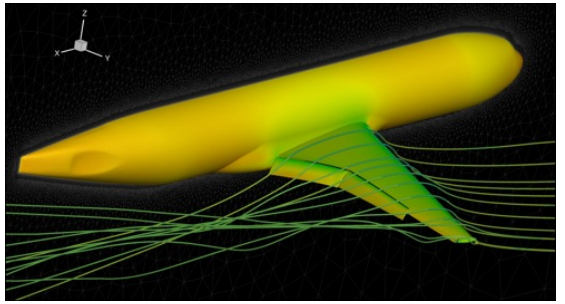
CHIUW 2020

?

Your Application Here?

(images provided by their respective teams and used with permission)

HIGHLIGHTS OF CHAPEL USAGE



CHAMPS: Computational Fluid Dynamics framework for airplane simulation

- Professor Eric Laurendeau's team at Polytechnique Montreal
- Performance: achieves competitive results w.r.t. established, world-class frameworks from Stanford, MIT, etc.
- Programmability: "*We ask students at the master's degree to do stuff that would take 2 years and they do it in 3 months.*"

Arkouda: data analytics framework (https://github.com/Bears-R-Us/arkouda)

- Mike Merrill, Bill Reus, et al., US DOD
- Python front end client, Chapel server that processes dozens of terabytes in seconds
- April 2023: 1200 GiB/s for argsort on an HPE EX system

Recent Journal Paper on using Chapel for calibrating hydrologic models

- Marjan Asgari et al, "Development of a knowledge-sharing parallel computing approach for calibrating distributed watershed hydrologic models", Environmental Modeling and Software.
- They report super-linear speedup

ARKOUDA ARGSORT PERFORMANCE

HPE Apollo (May 2021)

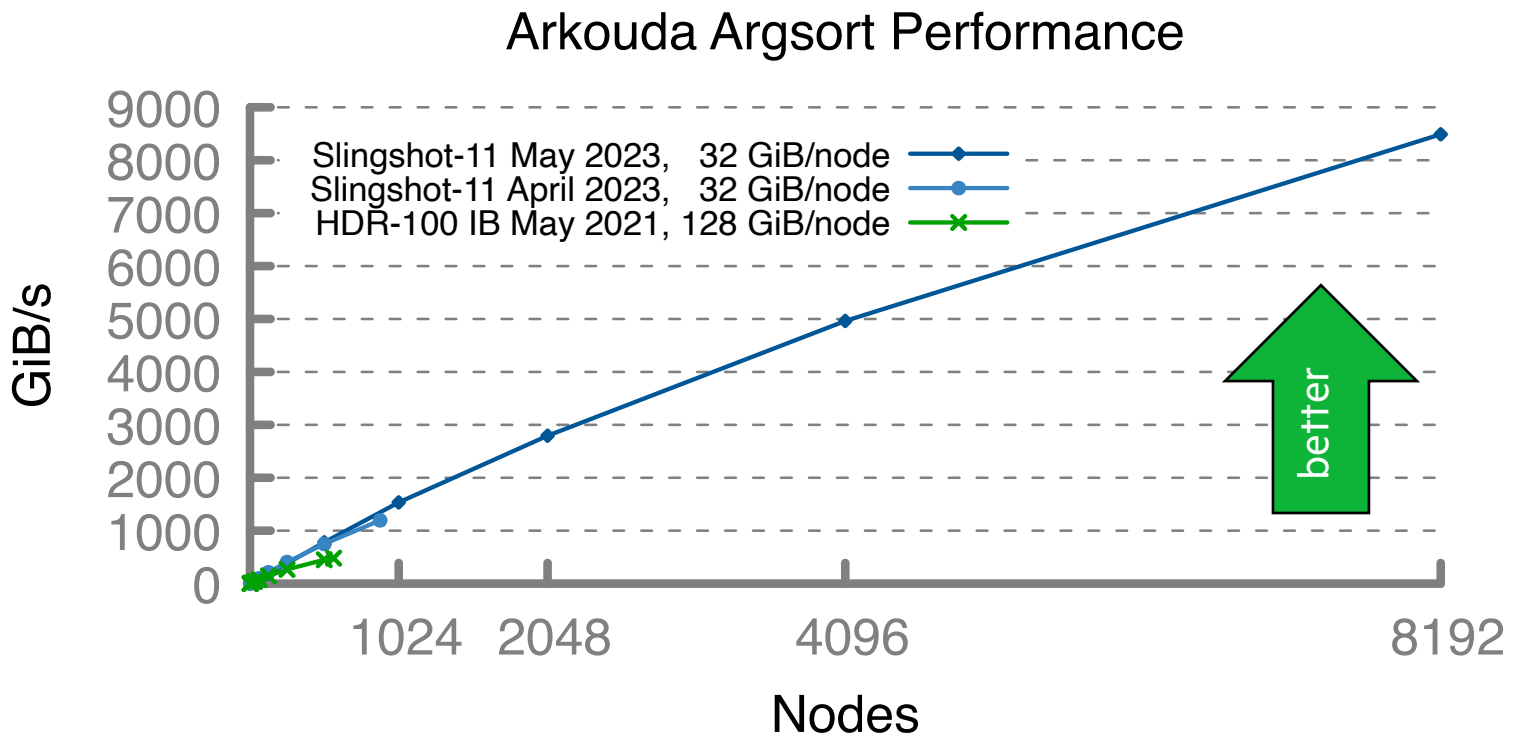
- HDR-100 Infiniband network (100 Gb/s)
- 576 compute nodes
- 72 TiB of 8-byte values
- ~480 GiB/s (~150 seconds)

HPE Cray EX (April 2023)

- Slingshot-11 network (200 Gb/s)
- 896 compute nodes
- 28 TiB of 8-byte values
- ~1200 GiB/s (~24 seconds)

HPE Cray EX (May 2023)

- Slingshot-11 network (200 Gb/s)
- 8192 compute nodes
- 256 TiB of 8-byte values
- ~8500 GiB/s (~31 seconds)



A notable performance achievement in ~100 lines of Chapel

INTRODUCTION TO CHAPEL

- What Chapel is and how programmers are using Chapel in their applications

- Chapel execution model with a parallel and distributed "Hello World"

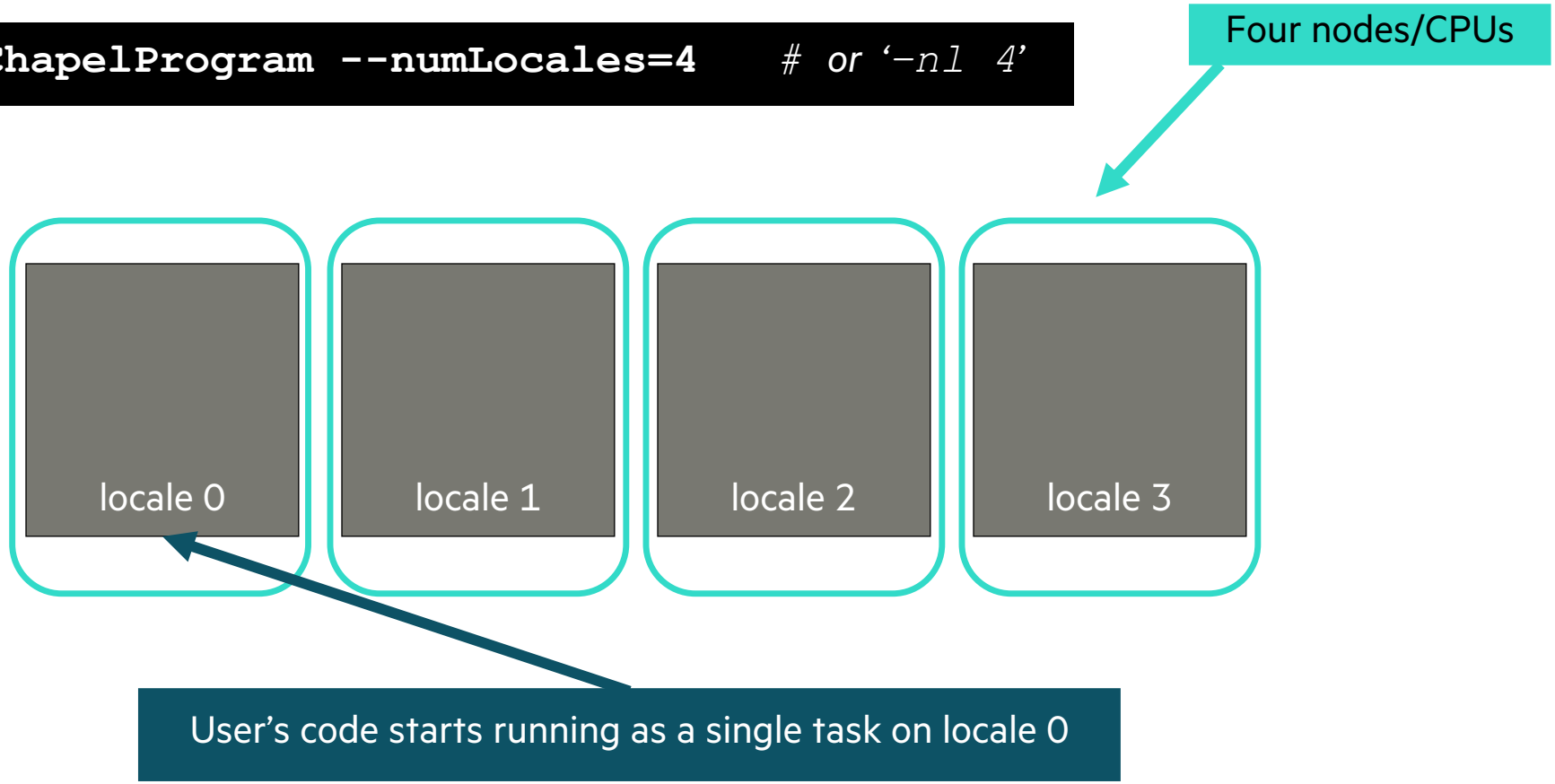
- 2D Heat Diffusion example: variants and how to compile and run them

- Learning objectives for today's 90-minute Chapel tutorial

CHAPEL EXECUTION MODEL AND TERMINOLOGY: LOCALES

- Locales can run tasks and store variables
  - Each locale executes on a “compute node” on a parallel system
  - User specifies number of locales on executable’s command-line

```
prompt> ./myChapelProgram --numLocales=4    # or ‘-nl 4’
```



TASK-PARALLEL “HELLO WORLD”

hello-dist-node-names.chpl

```
const numTasks = here.numPUs();
coforall tid in 1..numTasks do
  writef("Hello from task %n of %n on %s\n",
         tid, numTasks, here.name);
```

TASK-PARALLEL "HELLO WORLD"

hello-dist-node-names.chpl

```
const numTasks = here.numPUs();
coforall tid in 1..numTasks do
  writef("Hello from task %n of %n on %s\n",
         tid, numTasks, here.name);
```

'here' refers to the locale on which we're currently running

how many processing units (think "cores") does my locale have?

what's my locale's name?

TASK-PARALLEL “HELLO WORLD”

hello-dist-node-names.chpl

```
const numTasks = here.numPUs();
coforall tid in 1..numTasks do
  writef("Hello from task %n of %n on %s\n",
         tid, numTasks, here.name);
```

a 'coforall' loop executes each iteration as an independent task

```
> chpl hello-dist-node-names.chpl
> ./hello-dist-node-names
Hello from task 1 of 4 on n1032
Hello from task 4 of 4 on n1032
Hello from task 3 of 4 on n1032
Hello from task 2 of 4 on n1032
```

TASK-PARALLEL “HELLO WORLD”

hello-dist-node-names.chpl

```
const numTasks = here.numPUs();
coforall tid in 1..numTasks do
  writef("Hello from task %n of %n on %s\n",
         tid, numTasks, here.name);
```

```
> chpl hello-dist-node-names.chpl
> ./hello-dist-node-names
Hello from task 1 of 4 on n1032
Hello from task 4 of 4 on n1032
Hello from task 3 of 4 on n1032
Hello from task 2 of 4 on n1032
```

So far, this is a shared-memory program

Nothing refers to remote locales,
explicitly or implicitly

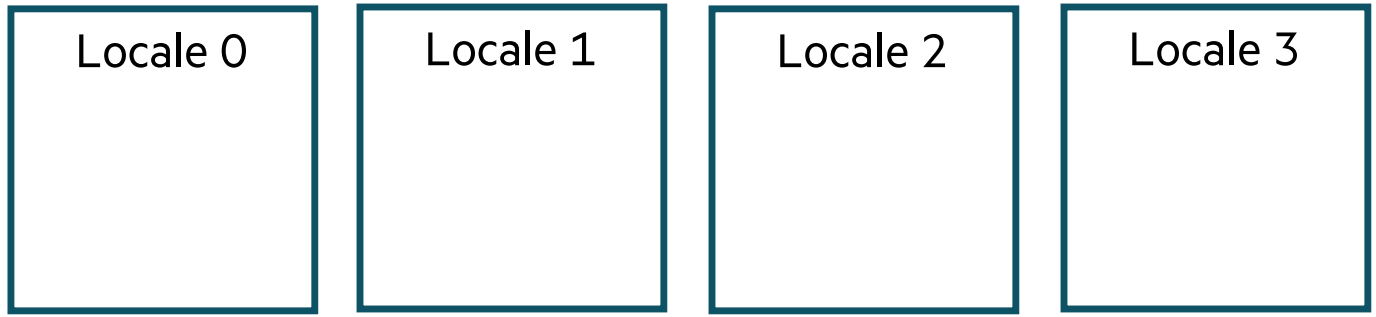
TASK-PARALLEL "HELLO WORLD" (DISTRIBUTED VERSION)

hello-dist-node-names.chpl

```
coforall loc in Locales {
  on loc {
    const numTasks = here.maxTaskPar;
    coforall tid in 1..numTasks do
      writef("Hello from task %n of %n on %s\n",
             tid, numTasks, here.name);
  }
}
```

the array of locales we're running on (introduced a few slides back)

Locales array:



TASK-PARALLEL "HELLO WORLD" (DISTRIBUTED VERSION)

hello-dist-node-names.chpl

```
coforall loc in Locales {
  on loc {
    const numTasks = here.numPUs();
    coforall tid in 1..numTasks do
      writef("Hello from task %n of %n on %s\n",
             tid, numTasks, here.name);
  }
}
```

create a task per locale on which the program is running

have each task run 'on' its locale

then print a message per core, as before

```
> chpl hello-dist-node-names.chpl
> ./hello-dist-node-names -nl=4
Hello from task 1 of 4 on n1032
Hello from task 4 of 4 on n1032
Hello from task 1 of 4 on n1034
Hello from task 2 of 4 on n1032
Hello from task 1 of 4 on n1033
Hello from task 3 of 4 on n1034
Hello from task 1 of 4 on n1035
…
```

INTRODUCTION TO CHAPEL

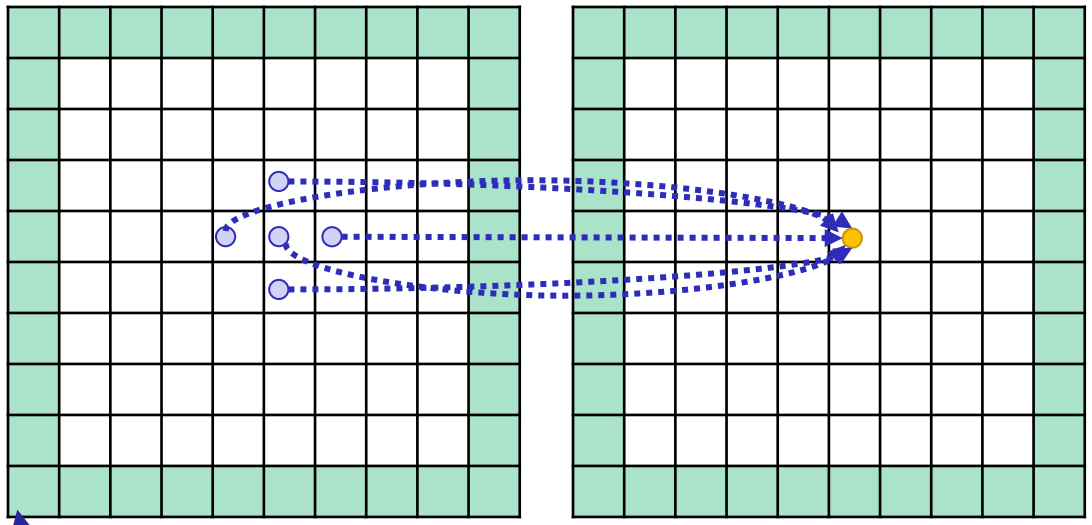
- What Chapel is and how programmers are using Chapel in their applications
- Chapel execution model with a parallel and distributed "Hello World"
- 2D Heat Diffusion example: variants and how to compile and run them
- Learning objectives for today's 90-minute Chapel tutorial

2D HEAT DIFFUSION EXAMPLE

See https://go.lbl.gov/cuf23-repo for more info and for example code.

- **See 'heat_2D.*.chpl' in the Chapel examples**
  - 'heat_2D.chpl' - shared memory parallel version that runs in locale 0
  - 'heat_2D_dist.chpl' - parallel and distributed version that is the same as 'heat_2D.chpl' but with distributed arrays
  - 'heat_2D_dist_buffers.chpl' - parallel and distributed version that copies to neighbors landing pad and then into local halos

PARALLEL HEAT DIFFUSION IN HEAT_2D.CHPL



- 2D heat diffusion PDE

$$\frac{\partial u}{\partial t} = \nu\frac{\partial^2 u}{\partial x^2} + \nu\frac{\partial^2 u}{\partial y^2}$$

Simplified form for below assume **Δx=Δy**, and let **α=νΔt/Δx²**

- Solving for next temperatures at each time step using finite difference method

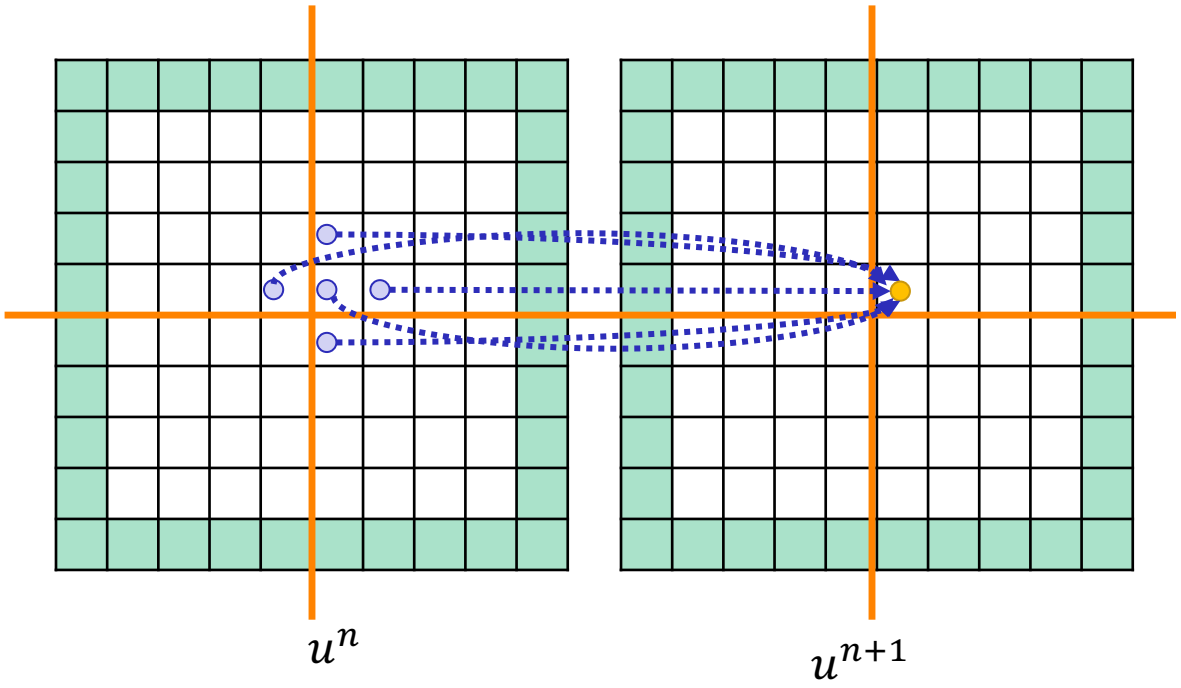
$$u_{i,j}^{n+1} = u_{i,j}^{n} + \alpha\left(u_{i+1,j}^{n} + u_{i-1,j}^{n} - 4u_{i,j}^{n} + u_{i,j+1}^{n} + u_{i,j-1}^{n}\right)$$

- All updates in a timestep can be done in parallel

```
forall (i, j) in indicesInner do
    u[i, j] = un[i, j] + alpha *
      (un[i, j-1] + un[i-1, j] + un[i+1, j] +
       un[i, j+1] - 4 * un[i, j]);
```

- Output is the mean and standard deviation of all the values and time to solution

DISTRIBUTED AND PARALLEL HEAT DIFFUSION IN HEAT_2D_DIST.CHPL



- Declaring 'u' and 'un' arrays

```
const indices = {0..<nx, 0..<ny}
var u: [indices] real;
```

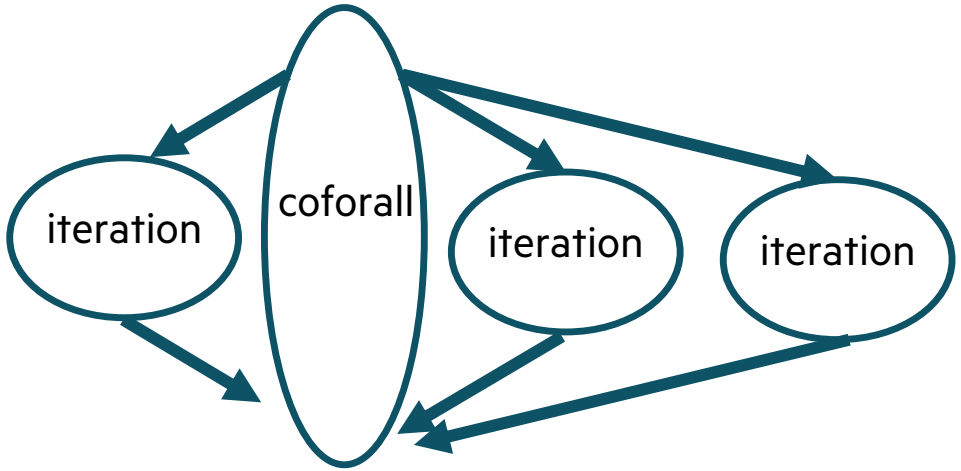
- Declaring 'u' and 'un' arrays as distributed (e.g., 2x2 distribution is shown)

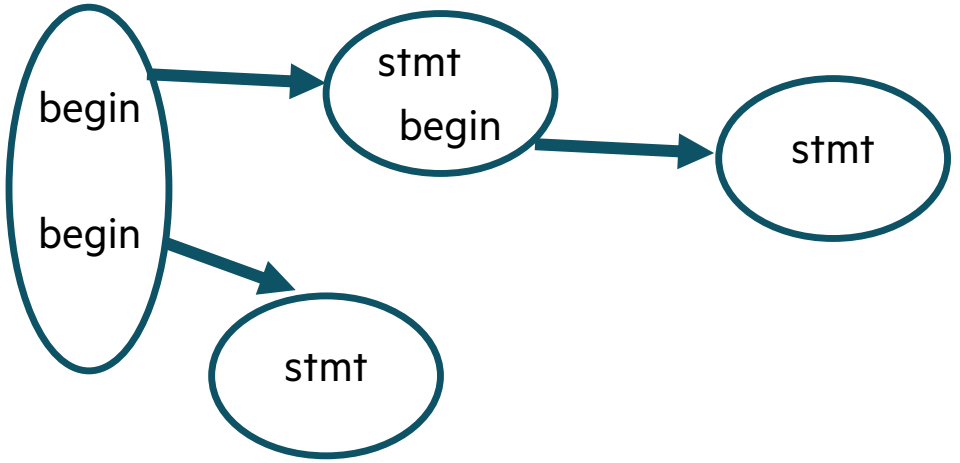
```
const indices = {0..<nx, 0..<ny},
      INDICES = Block.createDomain(indices);
var u: [INDICES] real;
```

- Reads that cross the distribution boundary will result in a remote get

PARALLELISM SUPPORTED BY CHAPEL

- **Synchronous parallellism**
  - 'coforall', distributed memory parallelism across processes/locales with 'on' syntax
  - 'coforall', shared-memory parallelism over threads
  - 'cobegin', executes all statements in block in parallel
- **Asynchronous parallelism**
  - 'begin', creates an asynchronous task
  - 'sync' and 'atomic' vars for task coordination
  - spawning subprocesses
- **Higher-level parallelism abstractions**
  - 'forall', data parallelism and iterator abstraction
  - 'foreach', SIMD parallelism
  - 'scan', operations such as cumulative sums
  - 'reduce', operations such as summation





LEARNING OBJECTIVES FOR TODAY'S CHAPEL TUTORIAL

- Compile and run Chapel programs
- Familiarity with the Chapel execution model including how to run codes in parallel on a single node, across nodes, and both

- Learn Chapel concepts by compiling and running provided code examples
  - Serial code using map/dictionary, (k-mer counting from bioinformatics)
  - Parallelism and locality in Chapel
  - Distributed parallelism and 1D arrays, (processing files in parallel)
  - Distributed parallelism and 2D arrays, (heat diffusion problem will see in UPC++ and CAF)
  - Distributed parallel image processing, (coral reef diversity example)
  - GPU parallelism (stream example)

- Where to get help and how you can participate in the Chapel community

BERKELEY LAB
Bringing Science Solutions to the World

U.S. DEPARTMENT OF ENERGY
Office of Science

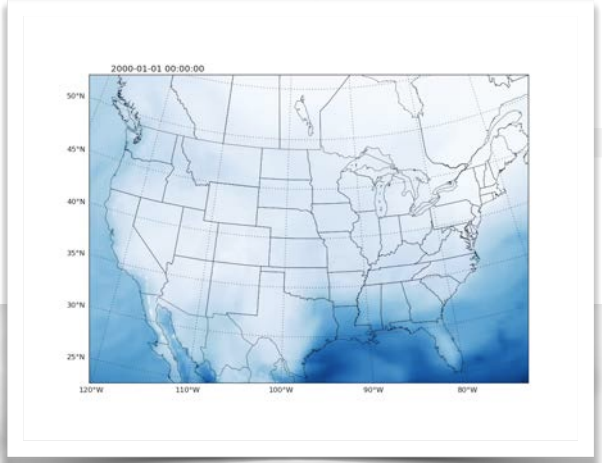
Coarray Fortran Tutorial

Damian Rouson
Computer Languages & System Software

Hosted by ECP, NERSC, and OLCF, 26-27 July 2023

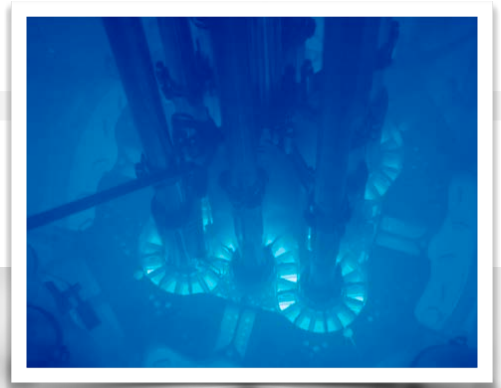
- Introduction to Coarray Fortran (“CAF”)
  - Why Fortran Matters
  - SPMD parallel execution
  - PGAS data structures & RMA
- Heat Conduction Solver
  - Compiling and running it
  - Understanding it

Why Fortran Matters



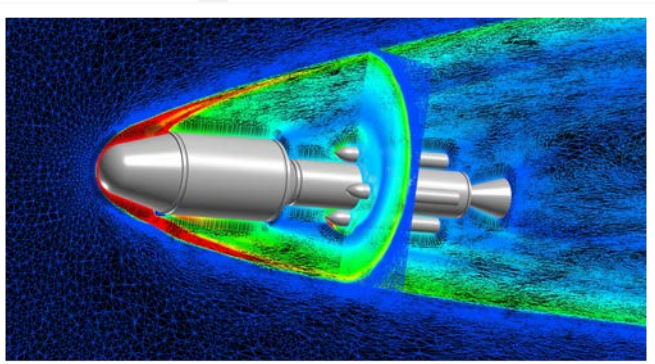
Intermediate Complexity Atmospheric Research (ICAR) Model
Courtesy of Ethan Gutmann, NCAR

Weather & Climate



U.S. Nuclear Regulatory Commission File Photo

Nuclear Energy



FUN3D Mesh Adaptation for Mars Ascent Vehicle, Courtesy of Eric Nielsen & Ashley Korzun, NASA Langley

Aerospace

CAF Philosophy

"The underlying philosophy of our design is to make the smallest number of changes to the language required to obtain a robust and efficient parallel language without requiring the programmer to learn very many new rules."

Reid, J., & Numrich, R. W. (2007). Co-arrays in the next Fortran standard. *Scientific Programming*, *15*(1), 9-26.

Seminal paper:

Numrich, R. W., & Reid, J. (1998, August). Co-Array Fortran for parallel programming. In *ACM SIGPLAN Fortran Forum* (Vol. 17, No. 2, pp. 1-31). New York, NY, USA: ACM.

Single Program Multiple Data

BERKELEY LAB
Bringing Science Solutions to the World

```
cd fortran
make run-hi
```

Single Program Multiple Data (SPMD) parallel execution

— Synchronized launch of multiple “images” (process/threads/ranks)

— Asynchronous execution except where program explicitly synchronizes

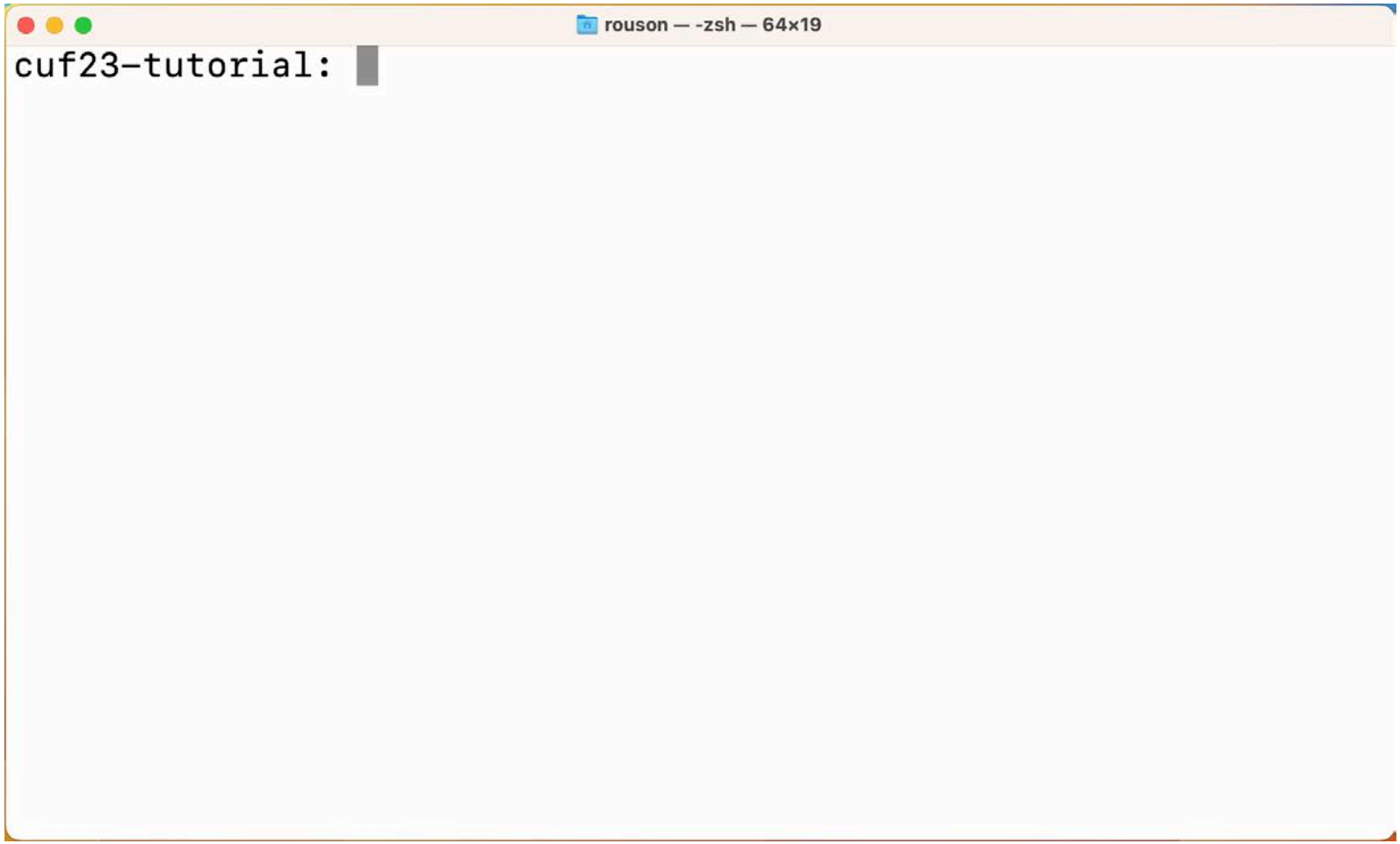
— Error termination or synchronized normal termination

rouson — vim hi.f90 — 67×5

```
1 program main
2   implicit none
3   print *,"Hello from image ", this_image(), "of", num_images()
4 end program
```

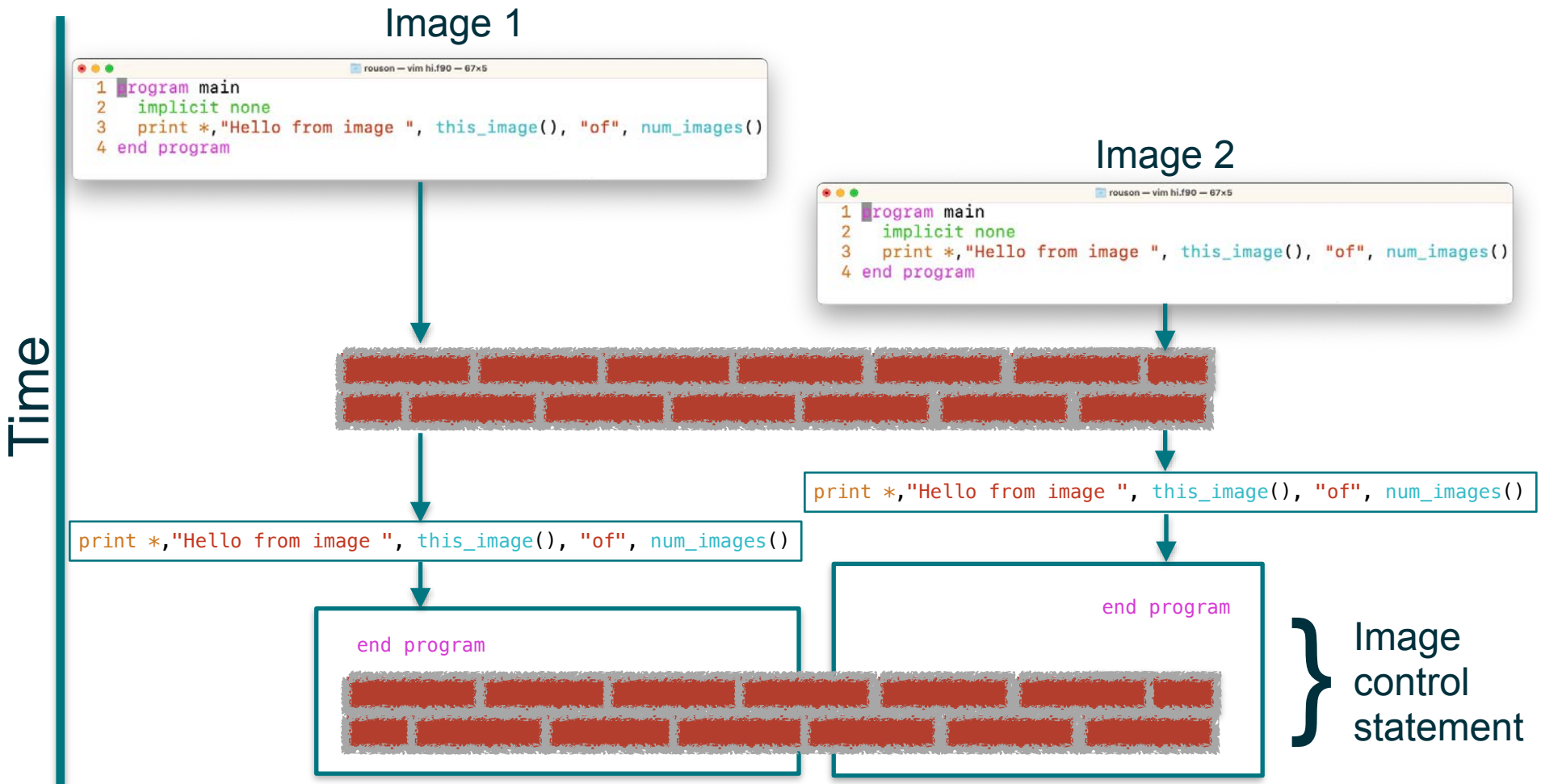
Compiling and Running `hi.f90`

BERKELEY LAB
Bringing Science Solutions to the World



SPMD Execution Sequence

BERKELEY LAB
Bringing Science Solutions to the World



1. After the creation of a fixed number of images, each image's first "segment" (sequence of statements) executes.
2. Image control statements totally order segments executed by a single image and partially order segments executed by separate images.

Partitioned Global Address Space (PGAS)

BERKELEY LAB
Bringing Science Solutions to the World

```
cd fortran
make run-hello
```

Coarrays:

— Distributed data structures — `greeting`

— Facilitate Remote Memory Access (RMA) — line 15

cuf23-tutorial — vim hello.f90 — 74×21

```
 1 program main
 2   !! One-sided communication of distributed greetings
 3   implicit none
 4   integer, parameter :: max_greeting_length=64, writer = 1
 5   integer image
 6   character(len=max_greeting_length) :: greeting[*] ! scalar coarray
 7
 8   associate(me => this_image(), ni=>num_images())
 9
10     write(greeting,*) "Hello from image",me,"of",ni ! local (no "[]")
11     sync all ! image control
12
13     if (me == writer) then
14       do image = 1, ni
15         print *,greeting[image] ! one-sided communication: "get"
16       end do
17     end if
18
19   end associate
20 end program
```

Compiling & Running `hello.f90`

BERKELEY LAB
Bringing Science Solutions to the World

Compiling and Running the Heat Equation Solver

BERKELEY LAB
Bringing Science Solutions to the World

Heat Equation

BERKELEY LAB
Bringing Science Solutions to the World

```
cd fortran
make run-heat-equation
```

$$\frac{\partial T}{\partial t} = \alpha \nabla^2 T$$

$$\{T\}^{n+1} = \{T\}^n + \Delta t \cdot \alpha \cdot \nabla^2 \{T\}^n$$

```
T =  T + dt * alpha * .laplacian. T
```

Heat Equation

BERKELEY LAB
Bringing Science Solutions to the World

```
cd fortran
make run-heat-equation
```

$$\frac{\partial T}{\partial t} = \alpha \nabla^2 T$$

$$\{T\}^{n+1} = \{T\}^n + \Delta t \cdot \alpha \cdot \nabla^2 \{T\}^n$$

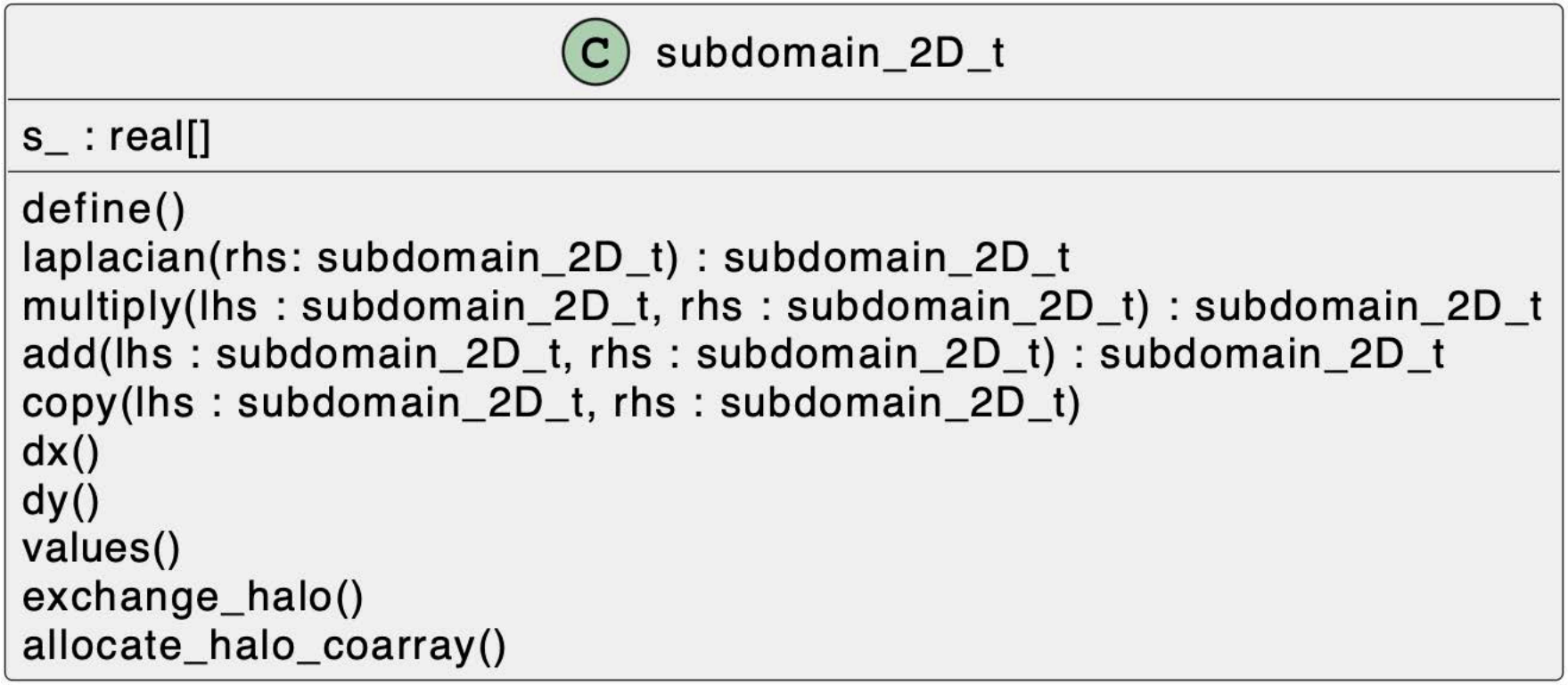
```
T = T + dt * alpha * .laplacian. T
```

local objects

`pure` user-defined operators

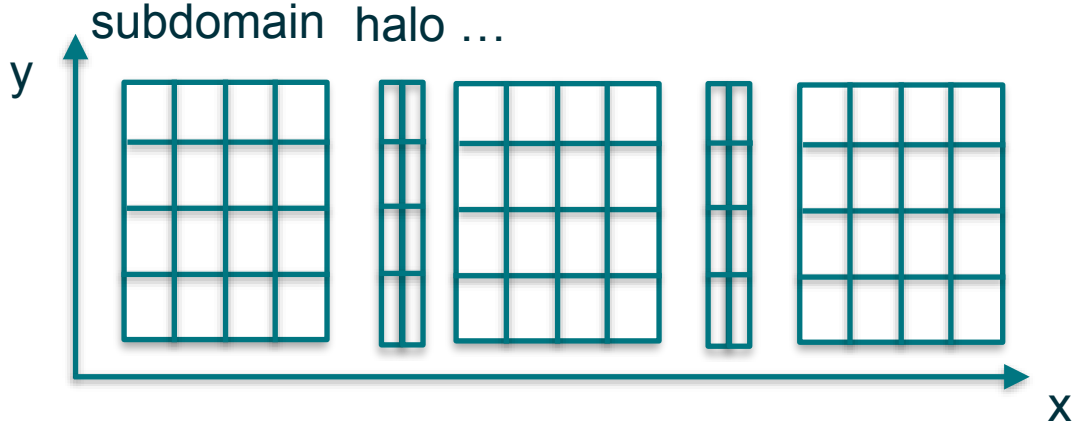
Class Diagram

BERKELEY LAB
Bringing Science Solutions to the World



Halo Exchange

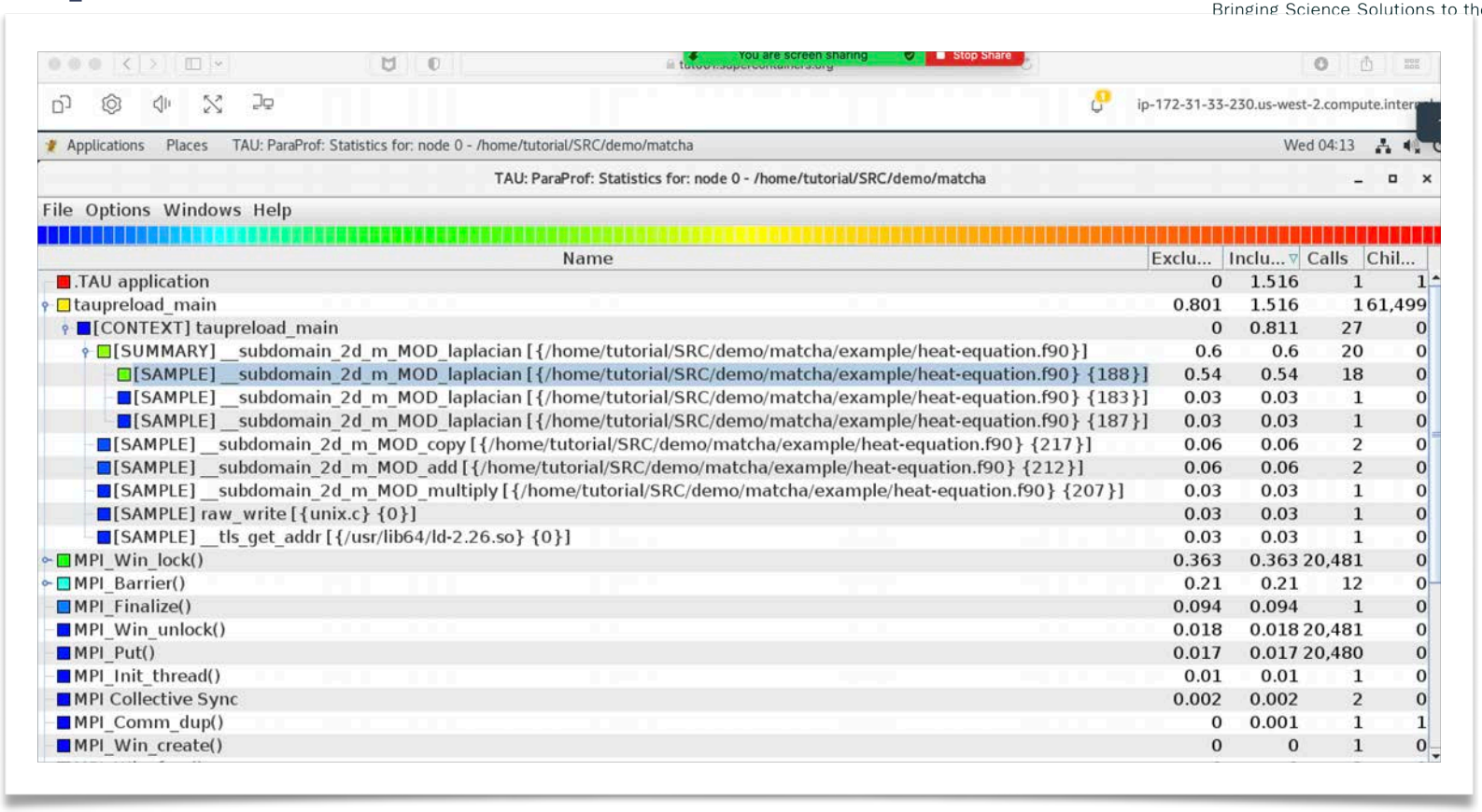
BERKELEY LAB
Bringing Science Solutions to the World




```
116 real(rkind), allocatable :: halo_x(:,:)[:]
117 integer, parameter :: west=1, east=2

134 me = this_image()
135 num_subdomains = num_images()
137 my_nx = nx/num_subdomains + merge(1, 0, me <= mod(nx, num_subdomains))

232 subroutine exchange_halo(self)
233   class(subdomain_2D_t), intent(in) :: self
234   if (me>1) halo_x(east,:)[me-1] = self%s_(1,:)
235   if (me<num_subdomains) halo_x(west,:)[me+1] = self%s_(my_nx,:)
236 end subroutine
```

Loop-Level Parallelism

BERKELEY LAB
Bringing Science Solutions to the World



Name	Exclu...	Inclu...	Calls	Chil...
.TAU application	0	1.516	1	1
taupreload_main	0.801	1.516	1	61,499
[CONTEXT] taupreload_main	0	0.811	27	0
[SUMMARY] __subdomain_2d_m_MOD_laplacian [{/home/tutorial/SRC/demo/matcha/example/heat-equation.f90}]	0.6	0.6	20	0
[SAMPLE] __subdomain_2d_m_MOD_laplacian [{/home/tutorial/SRC/demo/matcha/example/heat-equation.f90} {188}]	0.54	0.54	18	0
[SAMPLE] __subdomain_2d_m_MOD_laplacian [{/home/tutorial/SRC/demo/matcha/example/heat-equation.f90} {183}]	0.03	0.03	1	0
[SAMPLE] __subdomain_2d_m_MOD_laplacian [{/home/tutorial/SRC/demo/matcha/example/heat-equation.f90} {187}]	0.03	0.03	1	0
[SAMPLE] __subdomain_2d_m_MOD_copy [{/home/tutorial/SRC/demo/matcha/example/heat-equation.f90} {217}]	0.06	0.06	2	0
[SAMPLE] __subdomain_2d_m_MOD_add [{/home/tutorial/SRC/demo/matcha/example/heat-equation.f90} {212}]	0.06	0.06	2	0
[SAMPLE] __subdomain_2d_m_MOD_multiply [{/home/tutorial/SRC/demo/matcha/example/heat-equation.f90} {207}]	0.03	0.03	1	0
[SAMPLE] raw_write [{unix.c} {0}]	0.03	0.03	1	0
[SAMPLE] __tls_get_addr [{/usr/lib64/ld-2.26.so} {0}]	0.03	0.03	1	0
MPI_Win_lock()	0.363	0.363	20,481	0
MPI_Barrier()	0.21	0.21	12	0
MPI_Finalize()	0.094	0.094	1	0
MPI_Win_unlock()	0.018	0.018	20,481	0
MPI_Put()	0.017	0.017	20,480	0
MPI_Init_thread()	0.01	0.01	1	0
MPI Collective Sync	0.002	0.002	2	0
MPI_Comm_dup()	0	0.001	1	1
MPI_Win_create()	0	0	1	0

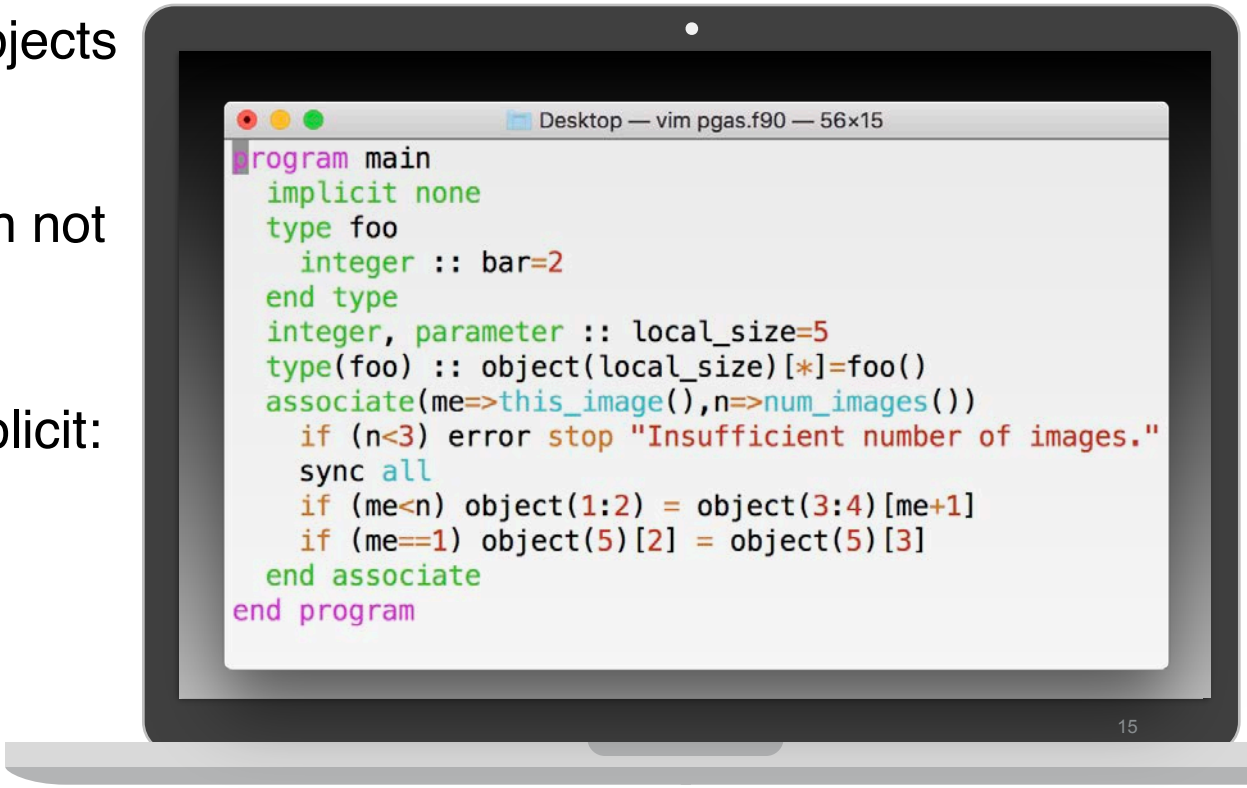
```
188 do concurrent(j=2:ny-1)
189   laplacian_rhs%s_(i, j) = &
      (halo_left(j)   - 2*rhs%s_(i, j) + rhs%s_(i+1,j  ))/dx_**2 + &
190   (rhs%s_(i, j-1) - 2*rhs%s_(i, j) + rhs%s_(i  ,j+1))/dy_**2
191 end do
```

line continuation

Comments

BERKELEY LAB
Bringing Science Solutions to the World

- Coarray Fortran began as a syntactically small extension to Fortran 95:
  - Square-bracketed “cosubscripts” distribute & communicate data
- Integration with other features:
  - Array programming: colon subscripts
  - OOP: distributed objects
- Minimally invasive:
  - Drop brackets when not communicating
- Communication is explicit:
  - Use brackets when communicating



BERKELEY LAB
Bringing Science Solutions to the World

U.S. DEPARTMENT OF ENERGY
Office of Science

Acknowledgements

This presentation includes efforts on the part of contributors to the Caffeine, GASNet-EX. Inference-Engine, Matcha, Nexport, and OpenCoarrays software libraries and members of the Computer Languages and Systems Software (CLaSS) Group and our collaborators:

Dan Bonachea, Jeremiah Bailey, Tobias Burnus, Alessandro Fanfarillo, Daniel Ceils Garza, Ethan Gutmann, Jeff Hammond, Peter Hill, Paul Hargrove, Dominick Martinez, Tan Nguyen, Katherine Rasmussen, Soren Rasmussen, Brad Richardson, Sameer Shende, David Torres, Andre Vehreschild, Jordan Welsman, Nathan Weeks, Yunhao Zhang

This research was supported in part by the **Exascale Computing Project** (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, in support of the nation's exascale computing imperative.

This research used resources of the **National Energy Research Scientific Computing Center (NERSC)**, a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231, as well as This research used resources of the **Oak Ridge Leadership Computing Facility** at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

BERKELEY LAB
LAWRENCE BERKELEY NATIONAL LABORATORY

U.S. DEPARTMENT OF ENERGY

UPC++: An Asynchronous RMA/RPC Library for Distributed C++ Applications

Amir Kamil

https://go.lbl.gov/CUF23
pagoda@lbl.gov

Applied Mathematics and Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, California, USA

Acknowledgements

This presentation includes the efforts of the following past and present members of the Pagoda group and collaborators:

Hadia Ahmed, John Bachan, Scott B. Baden, Dan Bonachea, Johnny Corbino, Rob Egan, Max Grossman, Paul H. Hargrove, Steven Hofmeyr, Mathias Jacquelin, Amir Kamil, Colin MacLean, Damian Rouson, Erich Strohmaier, Daniel Waters, Katherine Yelick

This research was supported in part by the **Exascale Computing Project** (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, in support of the nation's exascale computing imperative.

This research used resources of the **National Energy Research Scientific Computing Center (NERSC)**, a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231, as well as This research used resources of the **Oak Ridge Leadership Computing Facility** at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

What does UPC++ offer?

Asynchronous behavior

- **RMA**:
  - Get/put to a remote location in another address space
  - Low overhead, zero-copy, one-sided communication.
- **RPC: Remote Procedure Call**:
  - Moves computation to the data

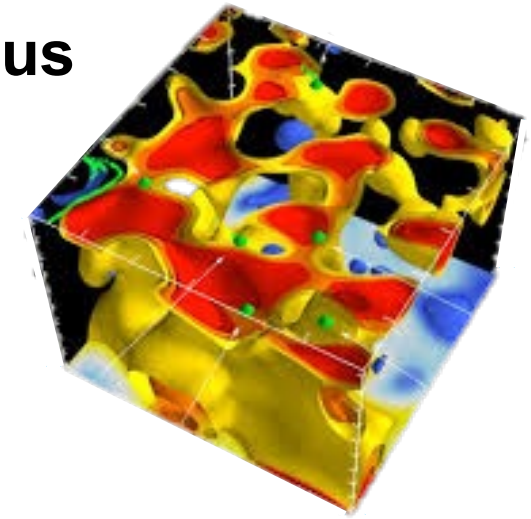
Design principles for performance

- All communication is syntactically explicit
- All communication is asynchronous: futures and promises
- Scalable data structures that avoid unnecessary replication

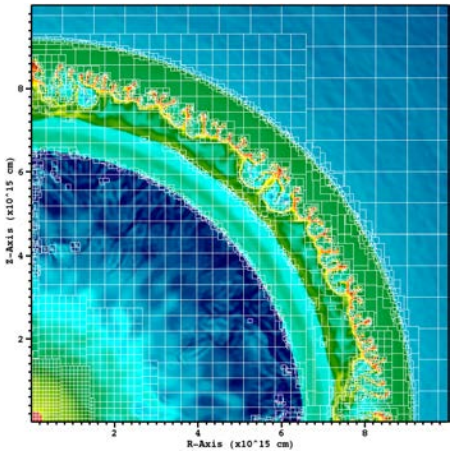
Some motivating applications

Many applications involve asynchronous updates to irregular data structures

- Adaptive meshes
- Sparse matrices
- Hash tables and histograms
- Graph analytics
- Dynamic work queues



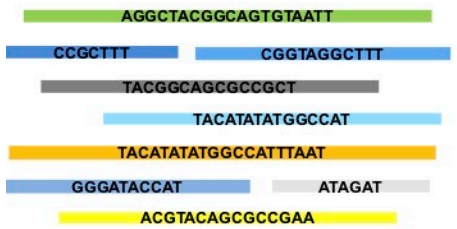
Seismo,Berkeley



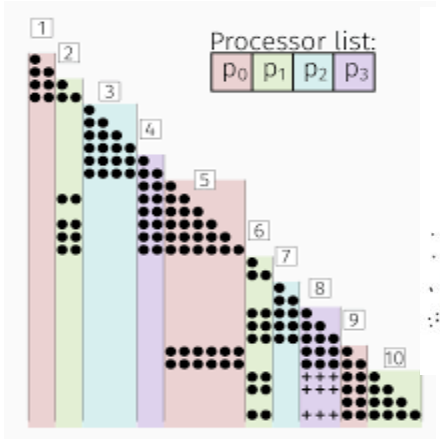
AMReX

Irregular and unpredictable data movement:

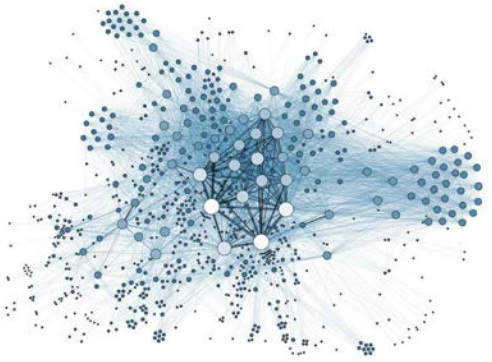
- *Space:* Pattern across processors
- *Time:* When data moves
- *Volume:* Size of data



ExaBiome



SymPACK



Graph analytics

Some motivating system trends

The first exascale systems appeared in 2022

- Cores per node is growing
- Accelerators (e.g. GPUs) are becoming more important
- Latency is not improving

Need to reduce communication costs in software

- Overlap communication to hide latency
- Reduce memory using smaller, more frequent messages
- Minimize software overhead
- Use simple messaging protocols (RDMA)

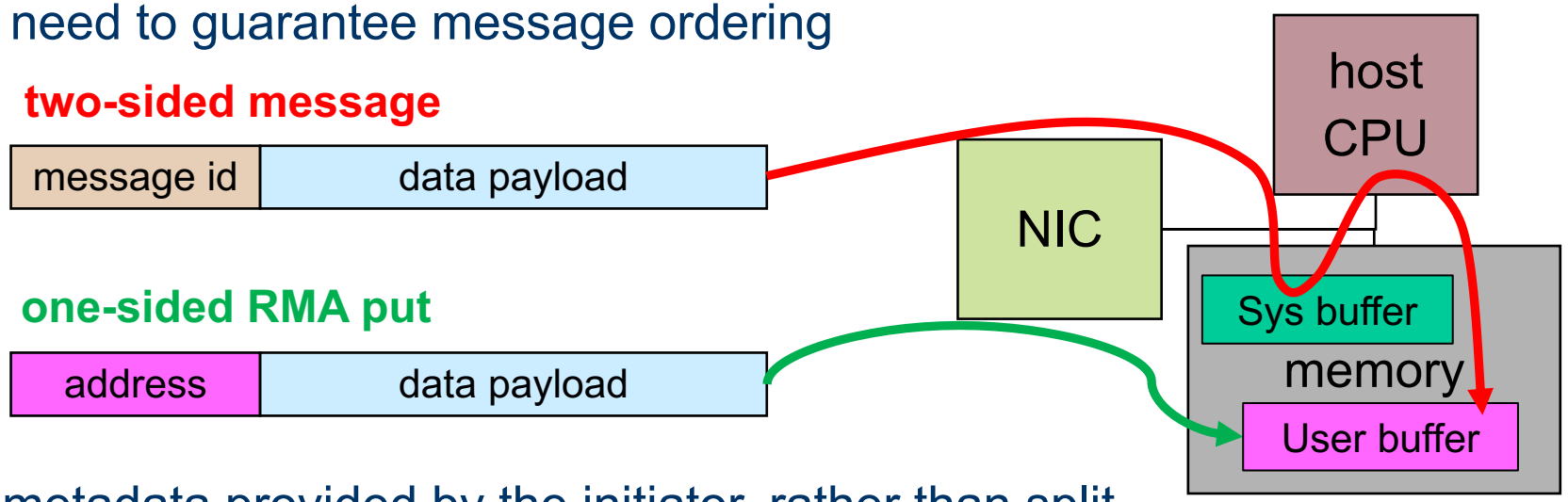
upc++ BERKELEY LAB

Reducing communication overhead

Let each process directly access another's memory via a global pointer

Communication is **one-sided** – there is no "receive" operation

- No need to match sends to receives
- No unexpected messages
- No need to guarantee message ordering



- All metadata provided by the initiator, rather than split between sender and receiver
- Supported in hardware through RDMA (Remote Direct Memory Access)

Looks like shared memory: shared data structures with asynchronous access

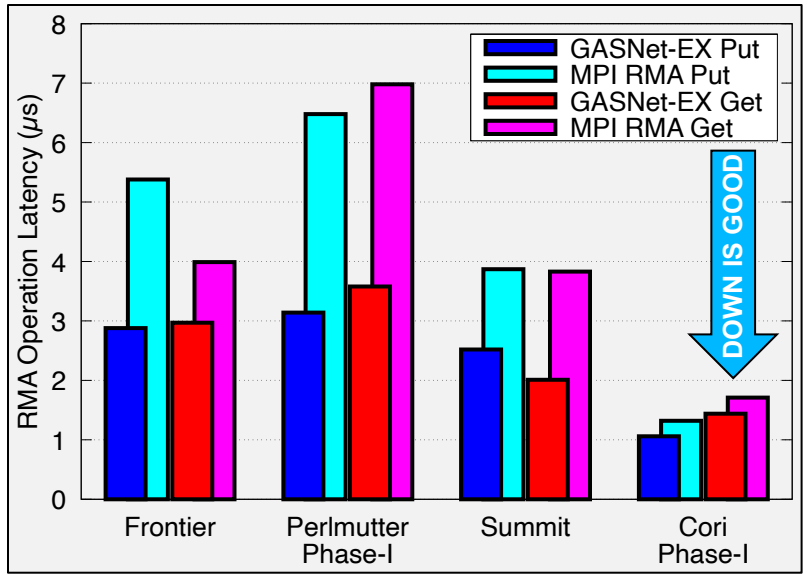
One-sided GASNet-EX vs one- and two-sided MPI

Four distinct network hardware types

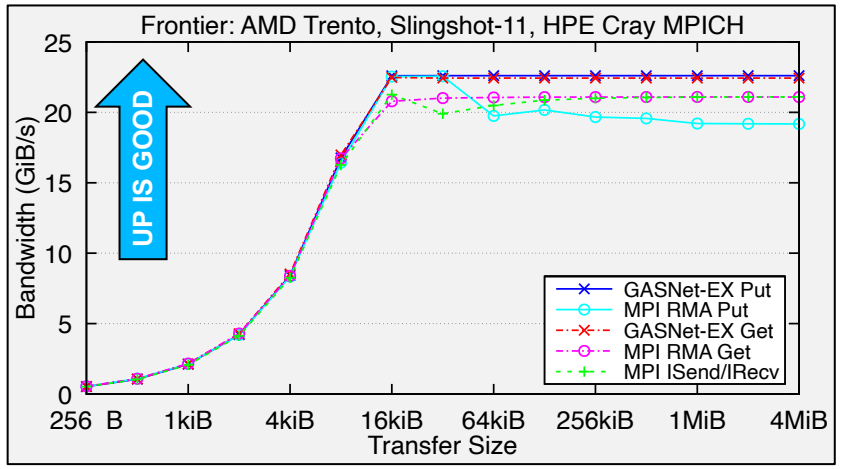
The performance of one-sided GASNet-EX matches or exceeds that of MPI RMA and message-passing:

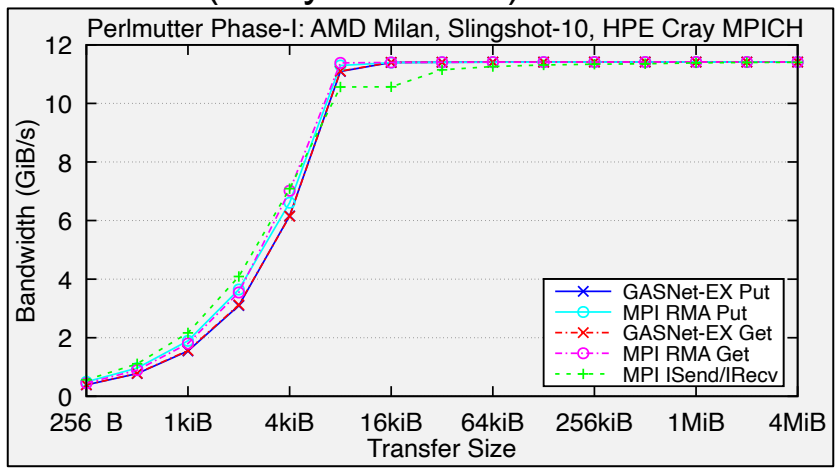
- 8-byte Put latency 19 - 52% better
- 8-byte Get latency 16 - 49% better
- Better flood bandwidth efficiency: often reaching same or better peak at ½ or ¼ the transfer size

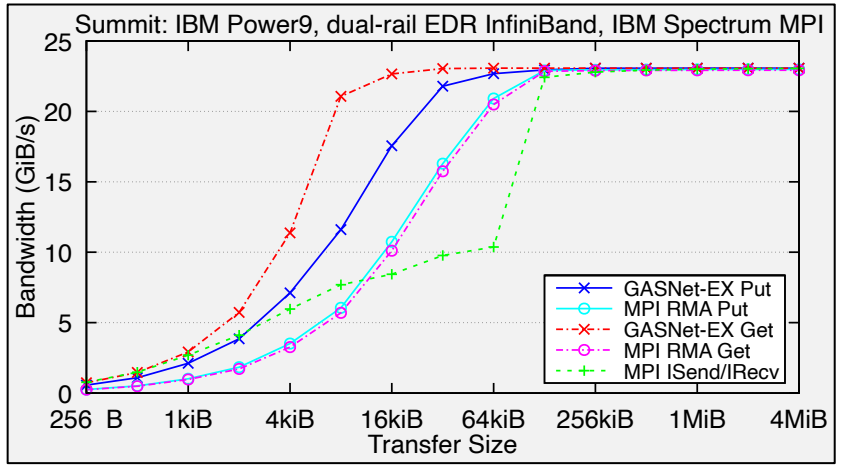
8-Byte RMA Operation Latency (one-at-a-time)

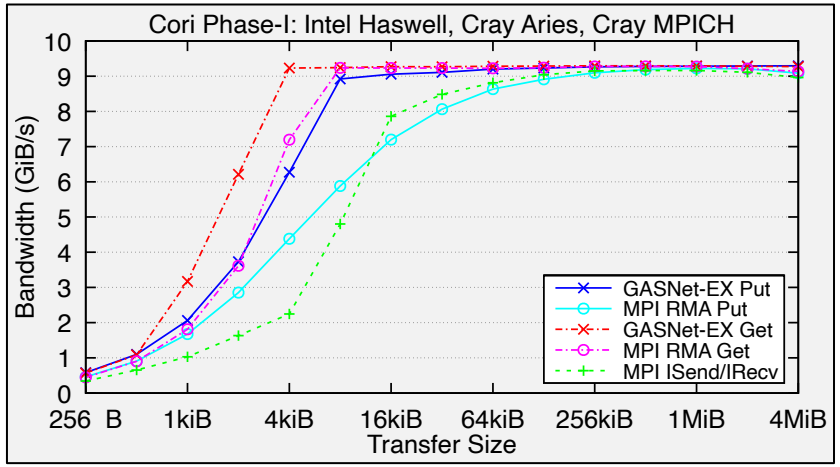


Uni-directional Flood Bandwidth (many-at-a-time)









Perlmutter Phase-I results collected July 2022, all others collected April 2023.
GASNet-EX tests were run using then-current GASNet library and its tests.
MPI tests were run using then-current center default MPI version and Intel MPI Benchmarks.
All tests use two nodes and one process per node.
For details see LCPC'18 doi.org/10.25344/S4QP4W and PAW-ATM'22 doi.org/10.25344/S40C7D
See also: gasnet.lbl.gov/performance

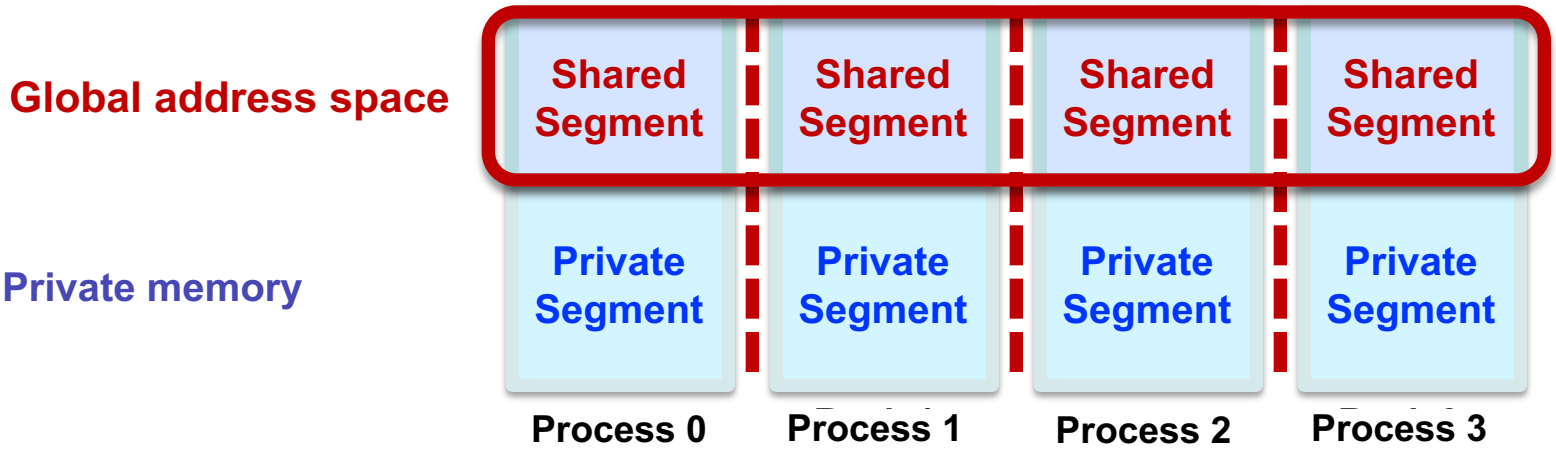
A Partitioned Global Address Space programming model

Global Address Space

- Processes may read and write *shared segments* of memory
- Global address space = union of all the shared segments

Partitioned

- *Global pointers* to objects in shared memory have an affinity to a particular process
- Explicitly managed by the programmer to optimize for locality
- In conventional shared memory, pointers do not encode affinity



The PGAS model

Partitioned **G**lobal **A**ddress **S**pace

- Support global memory, leveraging the network's RDMA capability
- Distinguish private and shared memory
- Separate synchronization from data movement

Languages that provide PGAS: **Chapel**, **Co-Array Fortran (Fortran 2008)**, UPC, Titanium, X10

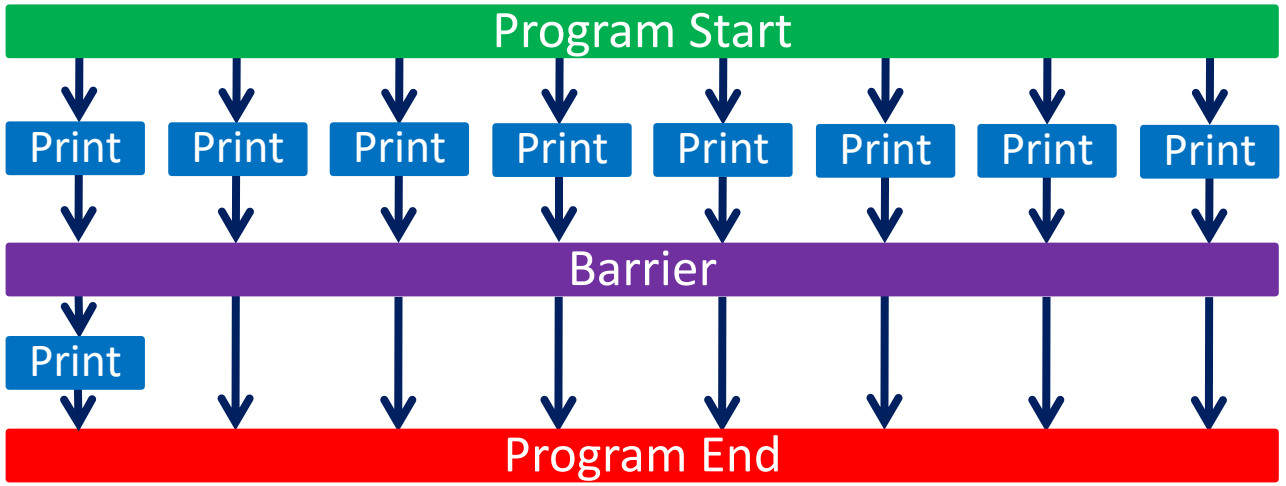
Libraries that provide PGAS: OpenSHMEM, Co-Array C++, Global Arrays, DASH, MPI-RMA

This presentation is about UPC++, a C++ library developed at Lawrence Berkeley National Laboratory

Execution model: SPMD

Like MPI and Coarray Fortran, UPC++ uses a SPMD model of execution, where a fixed number of processes run the same program

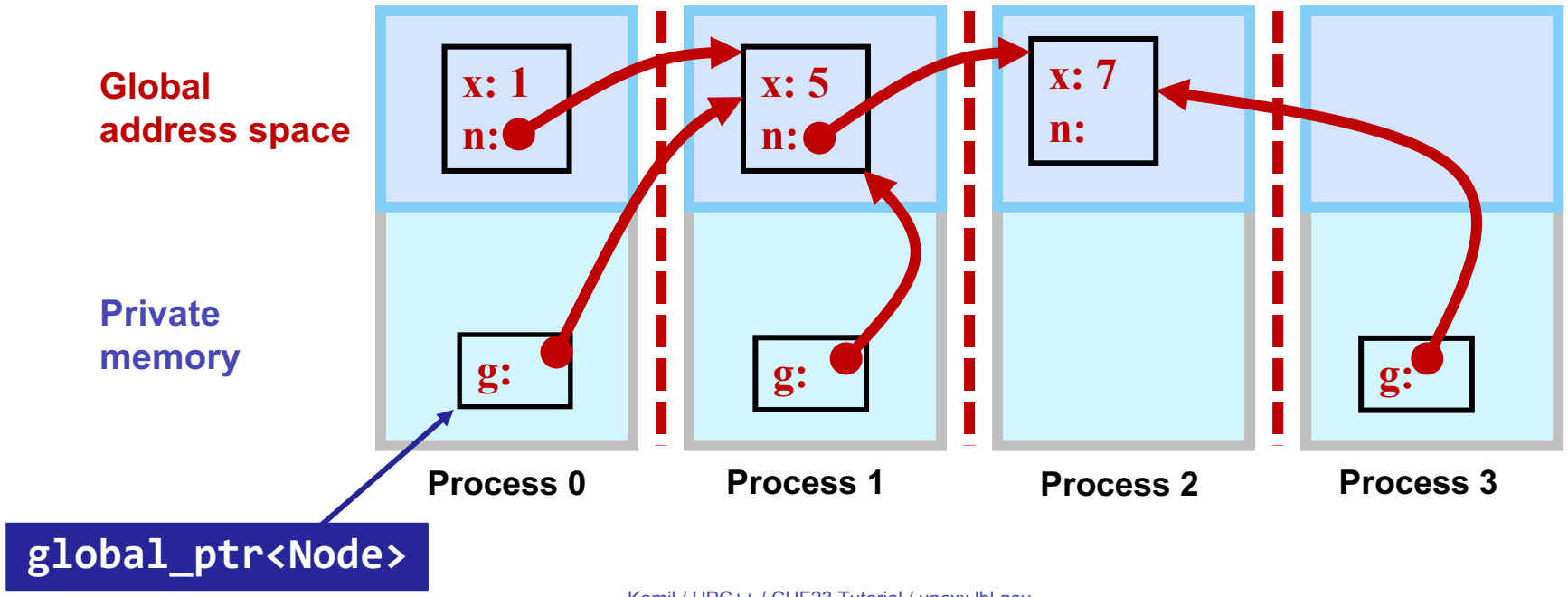
```
int main() {
  upcxx::init();
  cout << "Hello from " << upcxx::rank_me() << endl;
  upcxx::barrier();
  if (upcxx::rank_me() == 0) cout << "Done." << endl;
  upcxx::finalize();
}
```



Global pointers

Global pointers are used to create logically shared but physically distributed data structures

Parameterized by the type of object it points to, as with a C++ (raw) pointer: e.g. `global_ptr<double>`, `global_ptr<Node>`

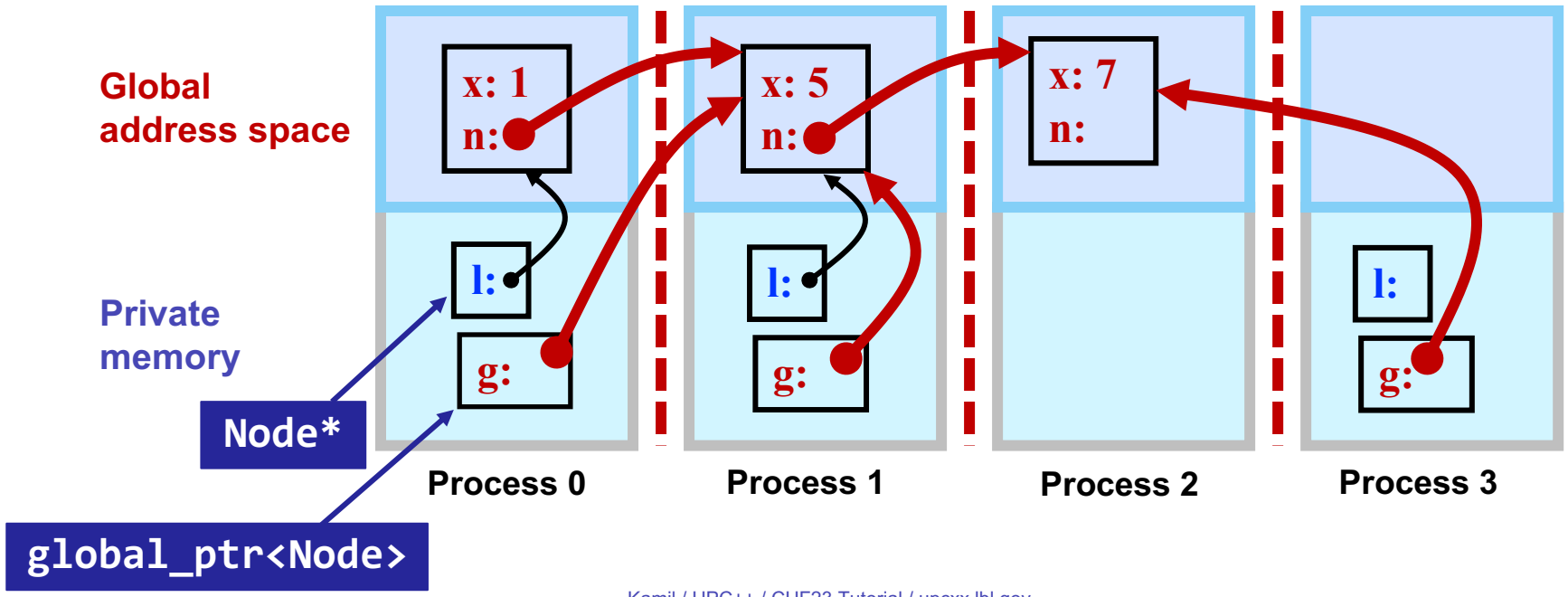


Global vs raw pointers and affinity

The affinity identifies the process that created the object

Global pointer carries both an address and the affinity for the data

Raw C++ pointers (e.g. `Node*`) can be used on a process to refer to objects in the global address space that have affinity to that process



How does UPC++ deliver the PGAS model?

UPC++ uses a "compiler-free," library approach

- UPC++ leverages C++ standards, needs only a standard C++ compiler

Relies on GASNet-EX for low-overhead communication

- Efficiently utilizes network hardware, including RDMA
- Provides Active Messages on which UPC++ RPCs are built
- Enables portability (laptops to supercomputers)

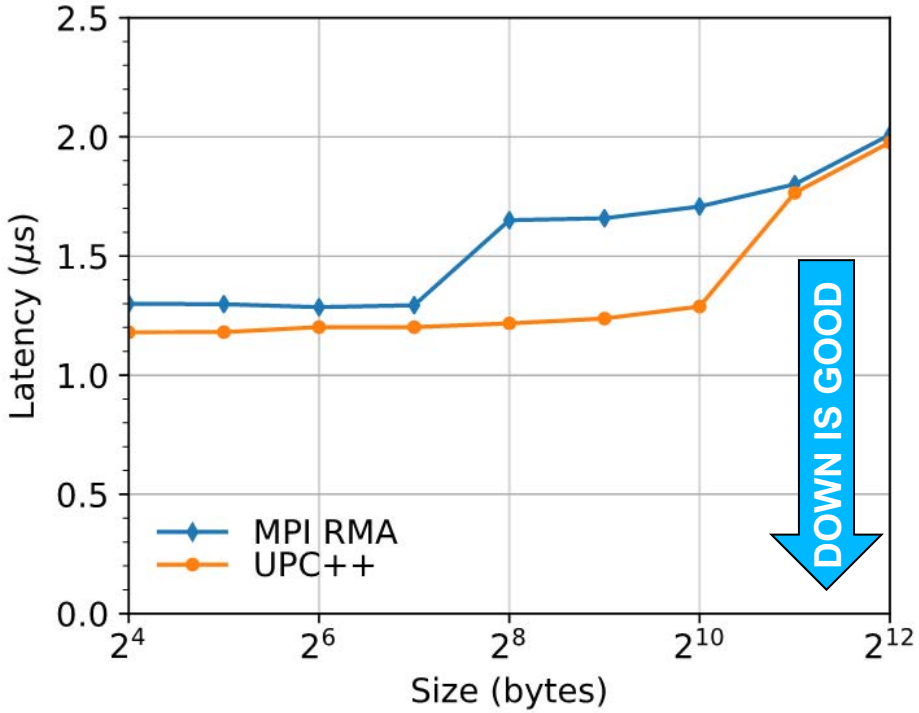
Designed for interoperability

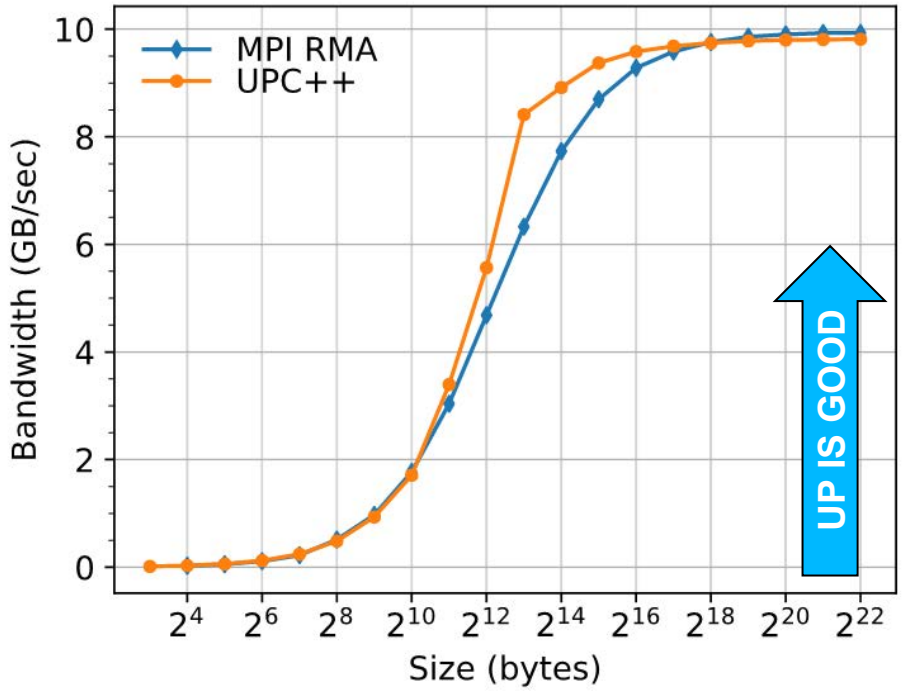
- Same process model as MPI, enabling hybrid applications
- On-node compute models (e.g. OpenMP, CUDA, HIP, Kokkos) can be mixed with UPC++ as in MPI+X

UPC++ on top of GASNet

Experiments on NERSC Cori:
- Cray XC40 system

Two processor partitions:
- Intel Haswell (2 x 16 cores per node)
- Intel KNL (1 x 68 cores per node)





Round-trip Put Latency (lower is better)

Flood Put Bandwidth (higher is better)

Data collected on Cori Haswell (https://doi.org/10.25344/S4V88H)

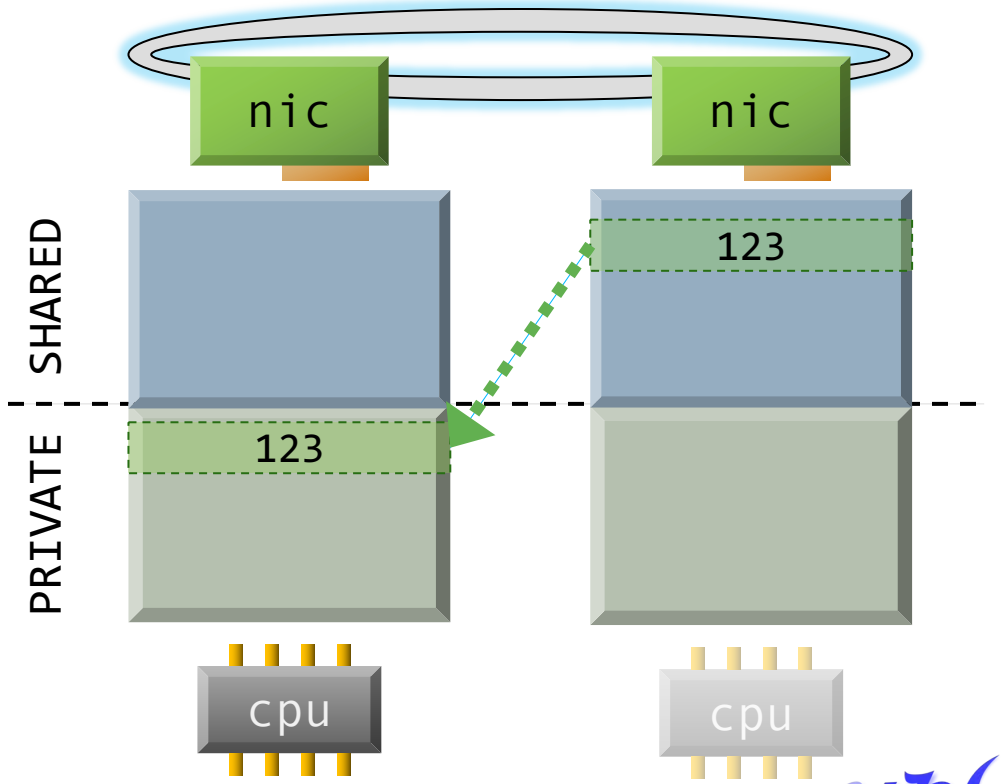
Asynchronous communication (RMA)

By default, all communication operations are split-phased

- **Initiate** operation
- **Wait** for completion
  A future holds a value and a state: ready/not-ready

```
global_ptr<int> gptr1 = ...;
future<int> f1 = rget(gptr1);
// unrelated work...
int t1 = f1.wait();
```

Wait returns the result when the rget completes

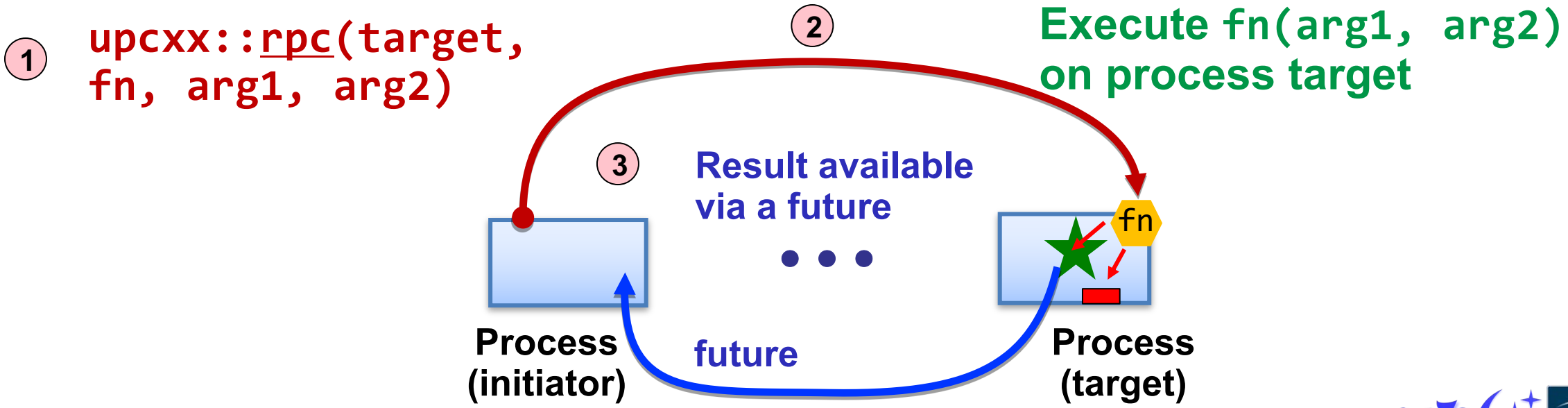


Remote procedure call (RPC)

Execute a function on another process, sending arguments and returning an optional result

1. Initiator injects the RPC to the *target* process
2. Target process executes `fn(arg1, arg2)` at some later time determined at the target
3. Result becomes available to the initiator via the future

Many RPCs can be active simultaneously, hiding latency



Hands-on: 2D heat diffusion

$$u_{i,j}^{n+1} = u_{i,j}^{n} + \alpha\left(u_{i+1,j}^{n} + u_{i-1,j}^{n} - 4u_{i,j}^{n} + u_{i,j+1}^{n} + u_{i,j-1}^{n}\right)$$

Everything needed for the hands-on activities is at:

https://go.lbl.gov/CUF23

Online materials include:

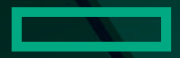
- Module info for NERSC Perlmutter, OLCF Frontier, and other machines
- Download links to install UPC++

Once you have set up your environment, copied the tutorial materials, and changed to the `cuf23/upcxx` directory:

Command to run in the terminal

Copy this and add arguments to change the problem size, e.g.:
`upcxx-run -N 1 -n 4 ./heat2d 8192 8192`

```
$ make run-heat2d
upcxx heat2d.cpp -Wall  -o heat2d
upcxx-run -N 1 -n 4 ./heat2d
[2]   My Neighbors: (1, 3)       My Domain: (2048,3072)
[3]   My Neighbors: (2, -1)      My Domain: (3072,4096)
[0]   My Neighbors: (-1, 1)      My Domain: (0,1024)
[1]   My Neighbors: (0, 2)       My Domain: (1024,2048)
[0] mean temperature=1.06256 | Solve time: 0.734826 seconds
```



Hewlett Packard
Enterprise

PROGRAMING IN CHAPEL

Michelle Strout and Jeremiah Corrado

CUF23: Sponsored by OLCF, NERSC, and ECP
July 26-27, 2023

LEARNING OBJECTIVES FOR TODAY'S CHAPEL TUTORIAL

- Compile and run Chapel programs
- Familiarity with the Chapel execution model including how to run codes in parallel on a single node, across nodes, and both

- Learn Chapel concepts by compiling and running provided code examples
  - Serial code using map/dictionary, (k-mer counting from bioinformatics)
  - Parallelism and locality in Chapel
  - Distributed parallelism and 1D arrays, (processing files in parallel)
  - Distributed parallelism and 2D arrays, (heat diffusion problem will see in UPC++ and CAF)
  - Distributed parallel image processing, (coral reef diversity example)
  - GPU parallelism (stream example)

- Where to get help and how you can participate in the Chapel community

HOW TO PARTICIPATE IN THIS TUTORIAL AND AFTERWARDS

- **During the tutorial today and tomorrow (July 26-27, 2023)**
  - Download the tarball of examples and follow the instructions in the README

```
curl -LO https://go.lbl.gov/cuf23.tar.gz
tar xzf cuf23.tar.gz
cd cuf23/
```

Check out the chapel-quickReference.pdf in the cuf23/chapel/ subdirectory

- **After the tutorial**
  - **The cuf23 tarball will still be available or clone from** *https://go.lbl.gov/cuf23-repo* *for Chapel code*
  - **Attempt this Online website for running Chapel code**
    - Go to main Chapel webpage at https://chapel-lang.org/ and click on the ATO icon on the lower left
  - **Using a container on your laptop**
    - First, install docker for your machine and then start it up
    - Then, the below commands work with docker


```
docker pull docker.io/chapel/chapel-gasnet    # takes about 5 minutes
docker run --rm -v "$PWD":/myapp -w /myapp chapel/chapel-gasnet chpl hello.chpl
docker run --rm -v "$PWD":/myapp -w /myapp chapel/chapel-gasnet ./hello -nl 1
```

SERIAL CODE USING MAP/DICTIONARY: K-MER COUNTING

kmer.chpl

```
use Map, IO;

config const infilename = "kmer_large_input.txt";
config const k = 4;

var sequence, line : string;
var f = open(infilename, ioMode.r);
var infile = f.reader();
while infile.readLine(line) {
 sequence += line.strip();
}

var nkmerCounts : map(string, int);

for ind in 0..<(sequence.size-k) {
 nkmerCounts[sequence[ind..#k]] += 1;
}
```

'Map' and 'IO' are two of the standard libraries provided in Chapel. A 'map' is like a dictionary in python.

'config const' indicates a configuration constant, which result in built-in command-line parsing

Reading all of the lines from the input file into the string 'sequence'.

The variable 'nkmerCounts' is being declared as a dictionary mapping strings to ints

Counting up each kmer in the sequence

EXPERIMENTING WITH THE K-MER EXAMPLE

```
make run-kmer
```

- **Some things to try out with 'kmer.chpl'**

```
chpl kmer.chpl
./kmer -nl 1

./kmer -nl 1 --k=10                                  # can change k
./kmer -nl 1 --infilename="kmer.chpl"                # changing infilename
./kmer -nl 1 --k=10 --infilename="kmer.chpl" # can change both
```

- **Key concepts**
  - 'use' command for including modules
  - configuration constants, 'config const'
  - reading from a file
  - 'map' data structure

LEARNING OBJECTIVES FOR TODAY'S CHAPEL TUTORIAL

- Compile and run Chapel programs
- Familiarity with the Chapel execution model including how to run codes in parallel on a single node, across nodes, and both

- Learn Chapel concepts by compiling and running provided code examples
  - ✓Serial code using map/dictionary, (k-mer counting from bioinformatics)
  - Parallelism and locality in Chapel
  - Distributed parallelism and 1D arrays, (processing files in parallel)
  - Distributed parallelism and 2D arrays, (heat diffusion problem will see in UPC++ and CAF)
  - Distributed parallel image processing, (coral reef diversity example)
  - GPU parallelism (stream example)

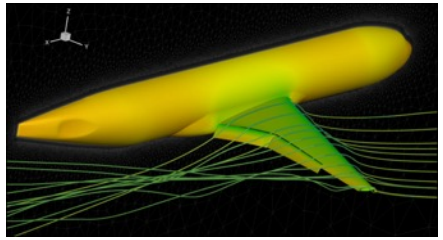
- Where to get help and how you can participate in the Chapel community

PARALLELISM SUPPORTED BY CHAPEL

- **Synchronous parallellism**
  - 'coforall', distributed memory parallelism across processes/locales with 'on' syntax
  - 'coforall', shared-memory parallelism over threads
  - 'cobegin', executes all statements in block in parallel
- **Asynchronous parallelism**
  - 'begin', creates an asynchronous task
  - 'sync' and 'atomic' vars for task coordination
  - spawning subprocesses
- **Higher-level parallelism abstractions**
  - 'forall', data parallelism and iterator abstraction
  - 'foreach', SIMD parallelism
  - 'scan', operations such as cumulative sums
  - 'reduce', operations such as summation

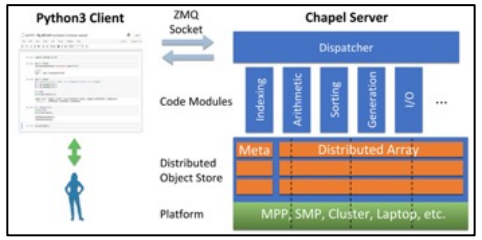
```
coforall loc in Locales do on loc { /* ... */ }
coforall tid in 0..<numTasks { /* ... */ }

cobegin { doTask0(); doTask1(); ... doTaskN(); }

var x : atomic int = 0, y : sync int = 0;
sync {
  begin x.add(1);
  begin y.writeEF(1);
  begin x.sub(1);
  begin y.writeFF(0);
}
assert(x.read() == 0);
assert(y.readFE() == 0);

var n = [i in 1..10] i*i;
forall x in n do x += 1;

var nPartialSums = + scan n;
var nSum = + reduce n;
```

APPLICATIONS OF CHAPEL: LINKS TO USERS' TALKS (SLIDES + VIDEO)



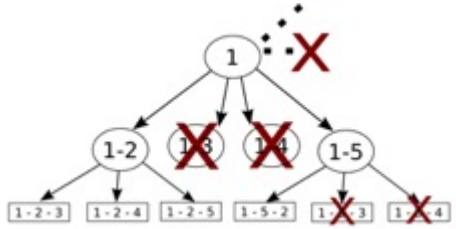
CHAMPS: 3D Unstructured CFD

CHIUW 2021 CHIUW 2022



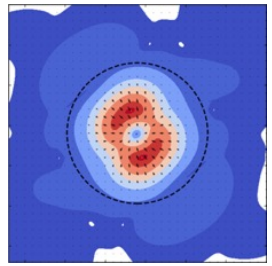
Arkouda: Interactive Data Science at Massive Scale

CHIUW 2020 CHIUW 2023



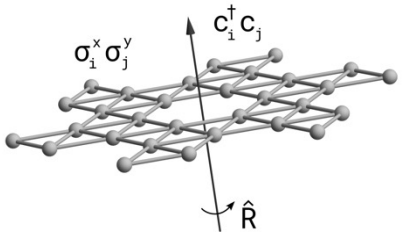
ChOp: Chapel-based Optimization

CHIUW 2021 CHIUW 2023



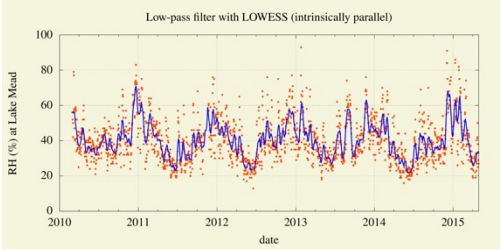
ChplUltra: Simulating Ultralight Dark Matter

CHIUW 2020 CHIUW 2022



Lattice-Symmetries: a Quantum Many-Body Toolbox

CHIUW 2022

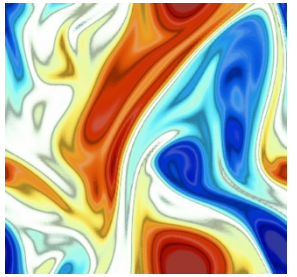


Desk dot chpl: Utilities for Environmental Eng.

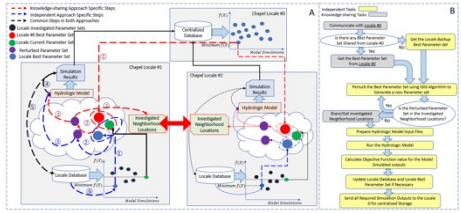
CHIUW 2022

RapidQ: Mapping Coral Biodiversity

CHIUW 2023

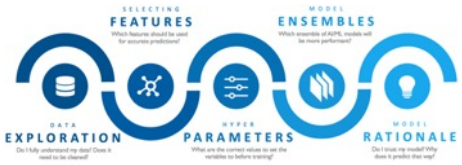


ChapQG: Layered Quasigeostrophic CFD



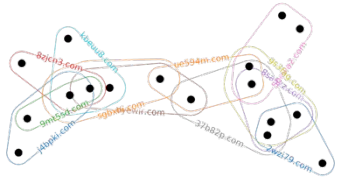
Chapel-based Hydrological Model Calibration

CHIUW 2023



CrayAI HyperParameter Optimization (HPO)

CHIUW 2021



CHGL: Chapel Hypergraph Library

CHIUW 2020

?

Your Application Here?

(images provided by their respective teams and used with permission)

USE OF PARALLELISM IN SOME APPLICATIONS AND BENCHMARKS

Application	Distributed 'coforall'	Threaded 'coforall'	Asynchronous 'begin'	'cobegin'	sync or atomic vars	subprocesses	forall	scan
HPO	✓	✓				✓		
Arkouda	✓	✓					✓	✓
CHAMPS	✓	✓						
ChOp	✓		✓		✓		✓	
ParFlow							✓	
Coral Reef	✓	✓		✓			✓	
Task Graph			✓		✓			

In this tutorial will be working with examples of parallelism from the yellow highlighted columns.

PARALLELISM ACROSS LOCALES AND WITHIN LOCALES

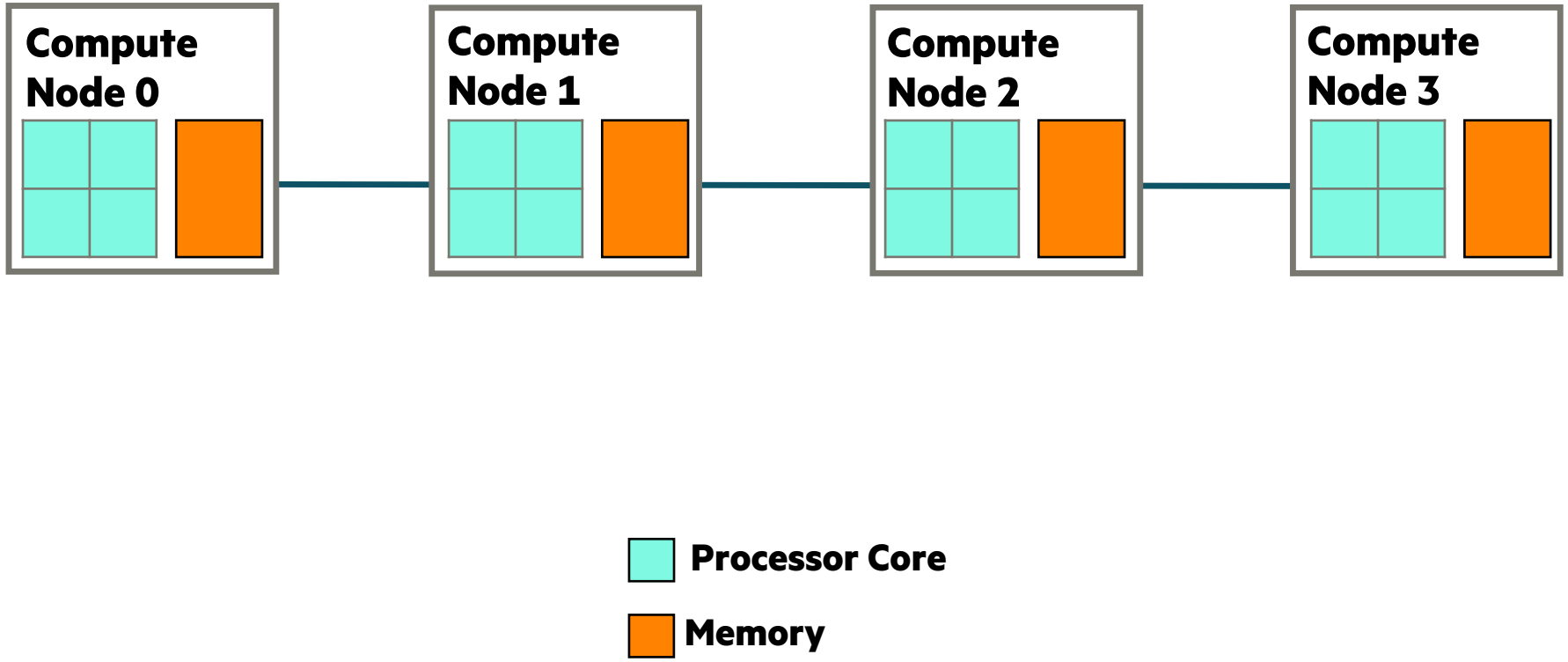
```
make run-hellopar
```

- **Parallel hello world**
  - hellopar.chpl
- **Key concepts**
  - 'coforall' over the `Locales` array with an `on` statement
  - 'coforall' creating some number of tasks per locale
  - configuration constants, 'config const'
  - range expression, '0..<tasksPerLocale'
  - 'writeln'
  - inline comments start with '//'

```
// can be set on the command line with --tasksPerLocale=2
config const tasksPerLocale = 1;

// parallel loops over nodes and then over threads
coforall loc in Locales do on loc {
  coforall tid in 0..<tasksPerLocale {

    writeln("Hello world! ",
            "(from task ", tid,
            " of ", tasksPerLocale,
            " on locale ", here.id,
            " of ", numLocales, ")" );
  }
}
```

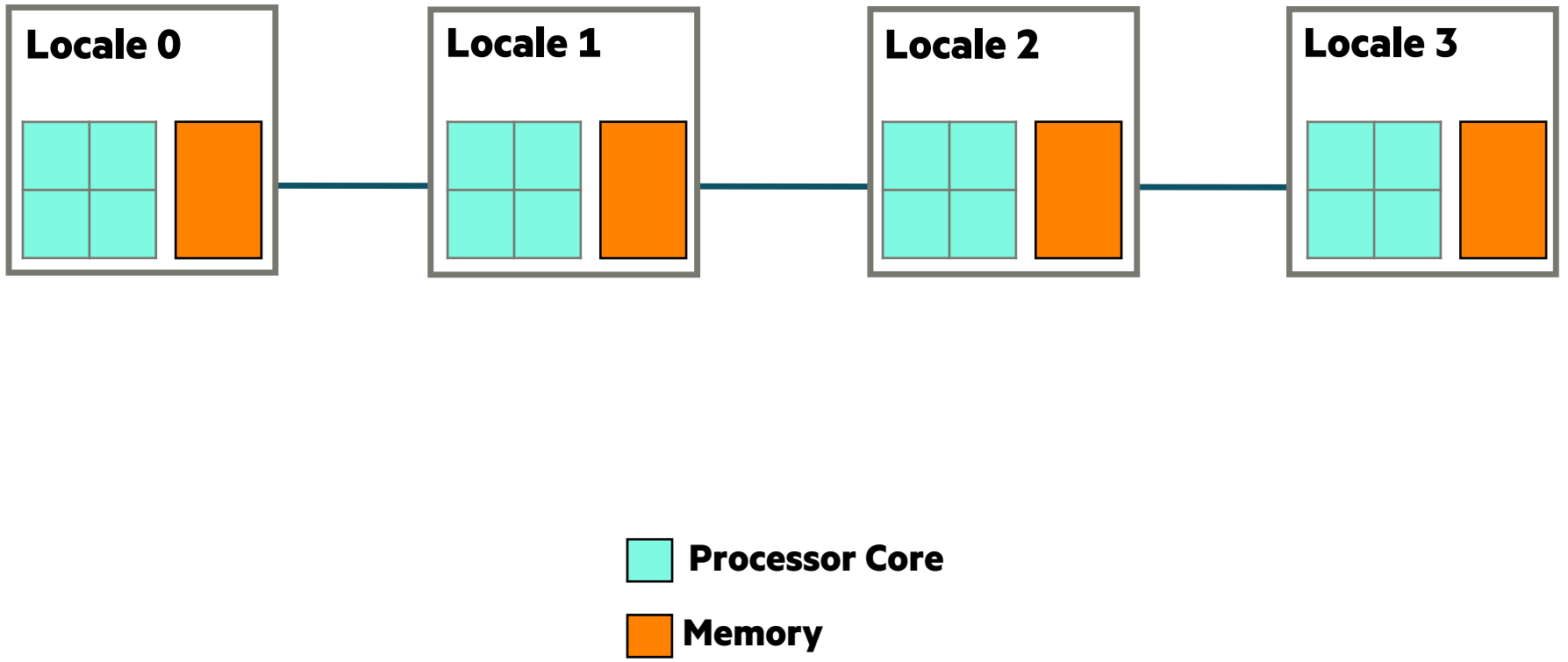
LOCALES AND EXECUTION MODEL IN CHAPEL

- In Chapel, a *locale* refers to a compute resource with…
  - processors, so it can run tasks
  - memory, so it can store variables
- For now, think of each compute node as having one locale run on it



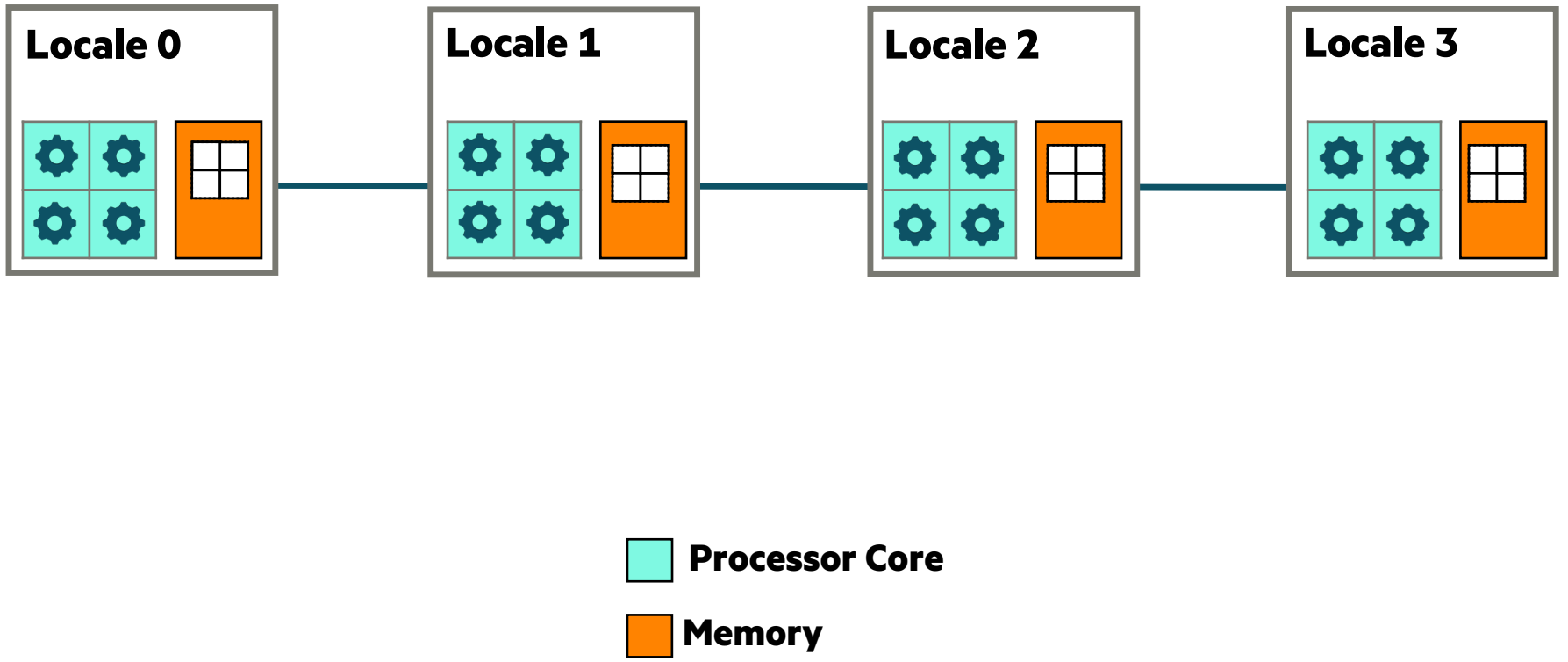
LOCALES AND EXECUTION MODEL IN CHAPEL

- Two key built-in variables for referring to locales in Chapel programs:
  - **`Locales`**: An array of locale values representing the system resources on which the program is running
  - **`here`**: The locale on which the current task is executing



KEY CONCERNS FOR SCALABLE PARALLEL COMPUTING

1. **parallelism:** Which tasks should run simultaneously?
2. **locality:** Where should tasks run? Where should data be allocated?



BASIC FEATURES FOR LOCALITY

basics-on.chpl

```
writeln("Hello from locale ", here.id);

var A: [1..2, 1..2] real;

on Locales[1] {
  var B: [1..2, 1..2] real;

  B = 2 * A;
}
```

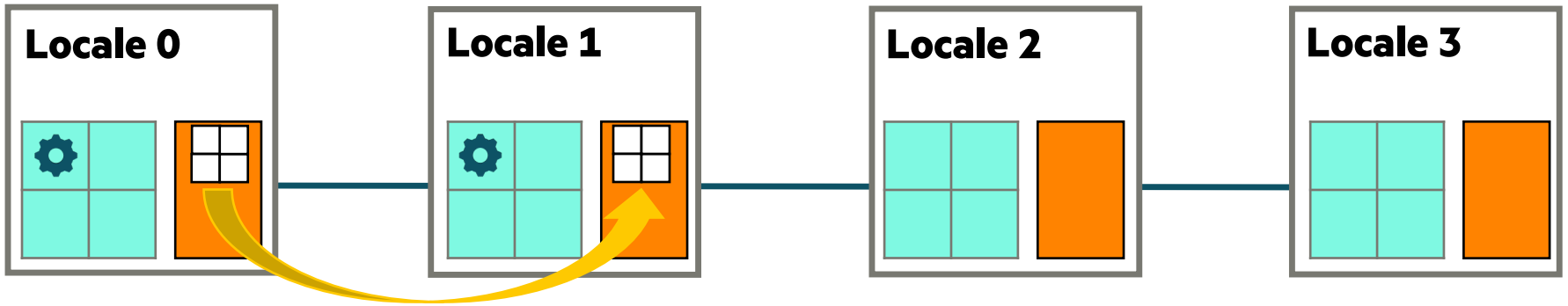
All Chapel programs begin running as a single task on locale 0

Variables are stored using the memory local to the current task

on-clauses move tasks to other locales

remote variables can be accessed directly

This is a serial, but distributed computation



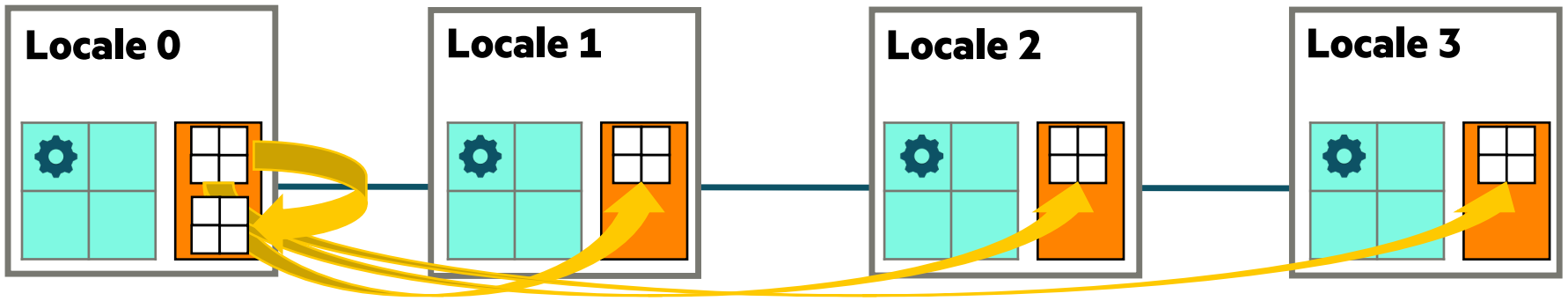
BASIC FEATURES FOR LOCALITY

basics-for.chpl

```
writeln("Hello from locale ", here.id);

var A: [1..2, 1..2] real;

for loc in Locales {
  on loc {
    var B = A;
  }
}
```

This loop will serially iterate over the program's locales

This is also a serial, but distributed computation



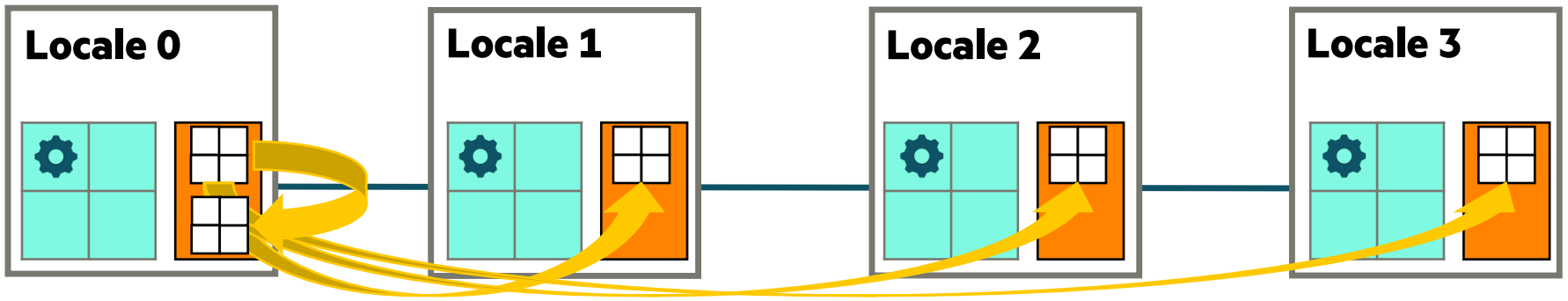
MIXING LOCALITY WITH TASK PARALLELISM

basics-coforall.chpl

```
writeln("Hello from locale ", here.id);

var A: [1..2, 1..2] real;

coforall loc in Locales {
  on loc {
    var B = A;
  }
}
```

The coforall loop creates a parallel task per iteration

This results in a parallel distributed computation



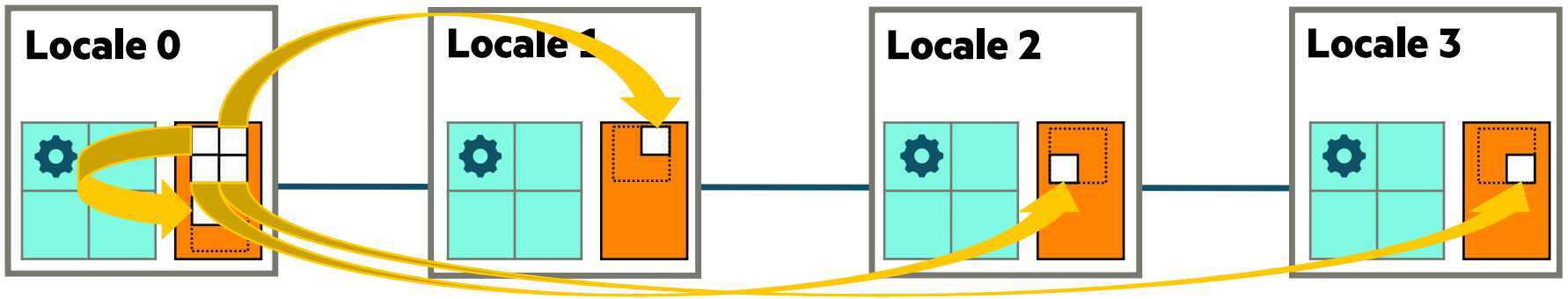
ARRAY-BASED PARALLELISM AND LOCALITY

basics-distarr.chpl

```
writeln("Hello from locale ", here.id);

var A: [1..2, 1..2] real;

use BlockDist;

var D = Block.createDomain({1..2, 1..2});
var B: [D] real;
B = A;
```

Chapel also supports distributed domains (index sets) and arrays

They also result in parallel distributed computation



PARALLELISM ACROSS LOCALES AND WITHIN LOCALES

```
make run-hellopar
```

- **Parallel hello world**
  - hellopar.chpl
- **Key concepts**
  - 'coforall' over the `Locales` array with an `on` statement
  - 'coforall' creating some number of tasks per locale
  - configuration constants, 'config const'
  - range expression, '0..<tasksPerLocale'
  - 'writeln'
  - inline comments start with '//'
- **Things to try**

```
./run-hellopar -nl 1 --tasksPerLocale=3
./run-hellopar -nl 2 --tasksPerLocale=3
```

```
// can be set on the command line with --tasksPerLocale=2
config const tasksPerLocale = 1;

// parallel loops over nodes and then over threads
coforall loc in Locales do on loc {
  coforall tid in 0..<tasksPerLocale {

    writeln("Hello world! ",
            "(from task ", tid,
            " of ", tasksPerLocale,
            " on locale ", here.id,
            " of ", numLocales, ")" );
  }
}
```

PARALLELISM AND LOCALITY ARE ORTHOGONAL IN CHAPEL

- This is a parallel, but local program:

```
coforall i in 1..msgs do
  writeln("Hello from task ", i);
```

- This is a distributed, but serial program:

```
writeln("Hello from locale 0!");
on Locales[1] do writeln("Hello from locale 1!");
on Locales[2] {
  writeln("Hello from locale 2!");
  on Locales[0] do writeln("Hello from locale 0!");
}
writeln("Back on locale 0");
```

- This is a distributed parallel program:

```
coforall i in 1..msgs do
  on Locales[i%numLocales] do
    writeln("Hello from task ", i, " running on locale ", here.id);
```

LEARNING OBJECTIVES FOR TODAY'S CHAPEL TUTORIAL

- Compile and run Chapel programs
- Familiarity with the Chapel execution model including how to run codes in parallel on a single node, across nodes, and both

- Learn Chapel concepts by compiling and running provided code examples
  - ✓Serial code using map/dictionary, (k-mer counting from bioinformatics)
  - ✓Parallelism and locality in Chapel
  - Distributed parallelism and 1D arrays, (processing files in parallel)
  - Distributed parallelism and 2D arrays, (heat diffusion problem will see in UPC++ and CAF)
  - Distributed parallel image processing, (coral reef diversity example)
  - GPU parallelism (stream example)

- Where to get help and how you can participate in the Chapel community

PROCESSING FILES IN PARALLEL

```
make run-parfilekmer
```

- **See 'parfilekmer.chpl' in the repository**

- **Some things to try out with 'parfilekmer.chpl'**

```
chpl parfilekmer.chpl --fast
./parfilekmer -nl 2 --dir="SomethingElse/"     # change dir with inputs files

./parfilekmer -nl 2 --k=10                     # can also change k
```

ANALYZING MULTIPLE FILES USING PARALLELISM

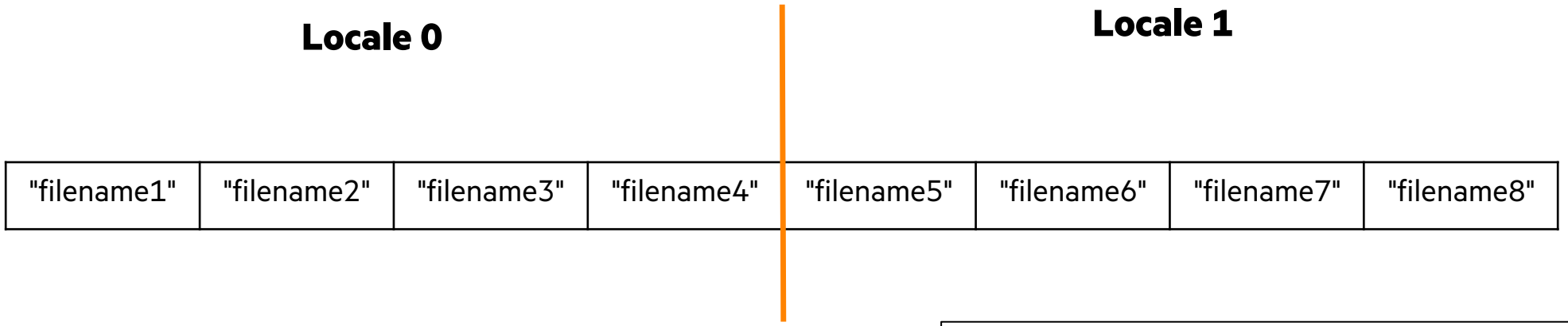
parfilekmer.chpl

```
use FileSystem;
config const dir = "DataDir";
var fList = findFiles(dir);
var filenames =
  Block.createArray(0..<fList.size,string);
filenames = fList;

// per file word count
forall f in filenames {
  ...
  // code from kmer.chpl
  ...
}
```

```
prompt> chpl --fast parfilekmer.chpl
prompt> ./parfilekmer -nl 1
prompt> ./parfilekmer -nl 4
```

- shared and distributed-memory parallelism using 'forall'
  - in other words, parallelism within the locale/node and across locales/nodes
- a distributed array
- command line options to indicate number of locales

BLOCK DISTRIBUTION OF ARRAY OF STRINGS




```
prompt> chpl --fast parfilekmer.chpl
prompt> ./parfilekmer -nl 2
```

- Array of strings for filenames is distributed across locales
- 'forall' will do parallelism across locales and then within each locale to take advantage of multicore

PROCESSING FILES IN PARALLEL

```
make run-parfilekmer
```

- **See 'parfilekmer.chpl' in the repository**

- **Some things to try out with 'parfilekmer.chpl'**

```
chpl parfilekmer.chpl --fast
./parfilekmer -nl 2 --dir="SomethingElse/"     # change dir with inputs files

./parfilekmer -nl 2 --k=10                     # can also change k
```

- **Concepts illustrated**
  - 'forall' provides distributed and shared memory parallelism when do a 'forall' over the Block distributed array
  - No puts and gets happening yet

LEARNING OBJECTIVES FOR TODAY'S CHAPEL TUTORIAL

- Compile and run Chapel programs
- Familiarity with the Chapel execution model including how to run codes in parallel on a single node, across nodes, and both

- Learn Chapel concepts by compiling and running provided code examples
  - ✓Serial code using map/dictionary, (k-mer counting from bioinformatics)
  - ✓Parallelism and locality in Chapel
  - ✓Distributed parallelism and 1D arrays, (processing files in parallel)
  - Distributed parallelism and 2D arrays, (heat diffusion problem will see in UPC++ and CAF)
  - Distributed parallel image processing, (coral reef diversity example)
  - GPU parallelism (stream example)

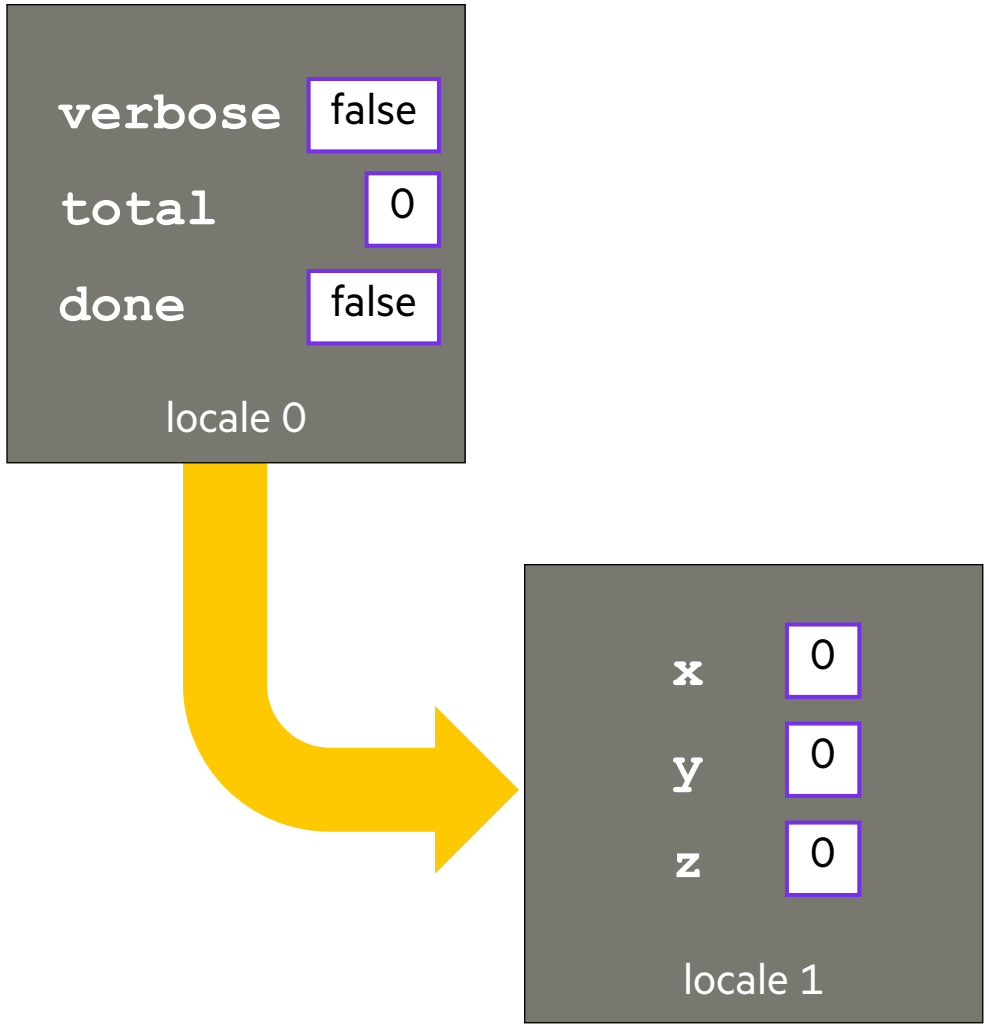
- Where to get help and how you can participate in the Chapel community

CHAPEL SUPPORTS A GLOBAL NAMESPACE WITH PUTS AND GETS

Note 1: Variables are allocated on the locale where the task is running

onClause.chpl

```
config const verbose = false;
var total = 0,
    done = false;

…

on Locales[1] {
  var x, y, z: int;
  …



}
```

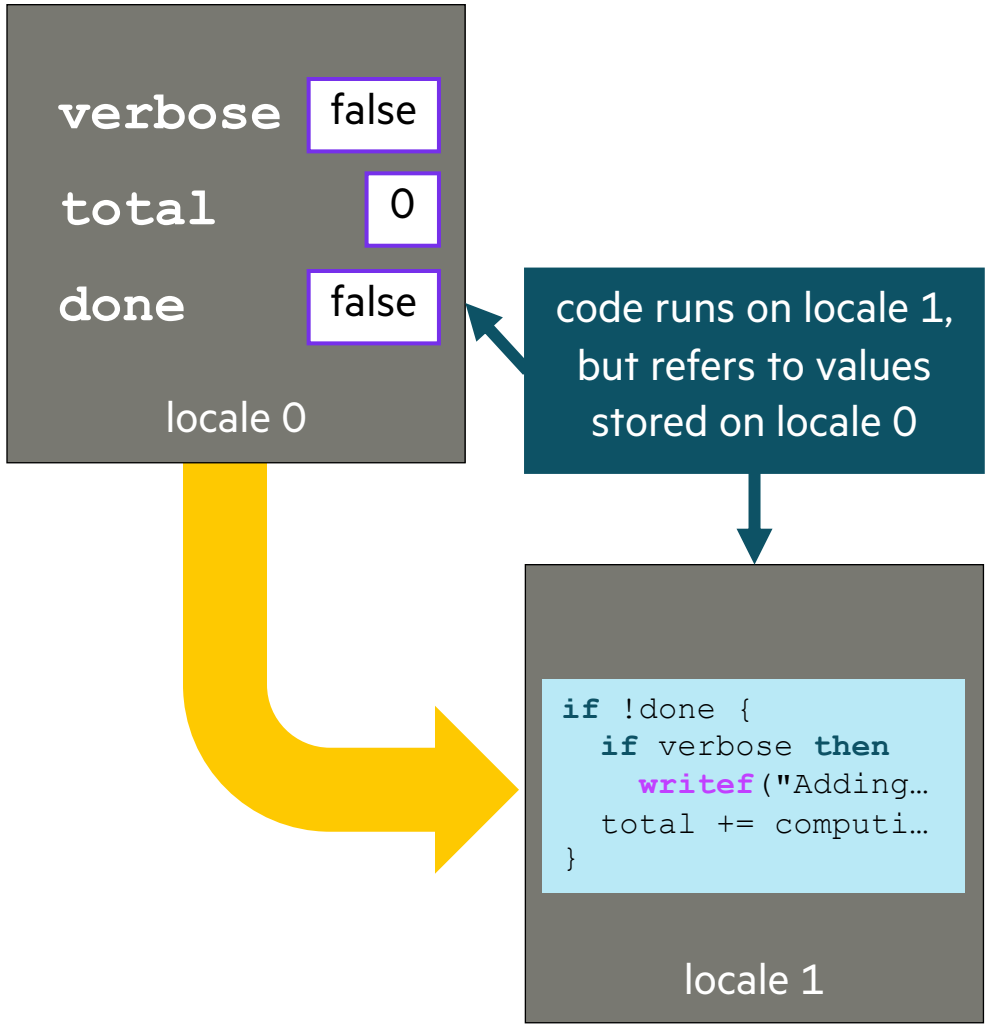


CHAPEL SUPPORTS A GLOBAL NAMESPACE

Note 2: Tasks can refer to lexically visible variables, whether local or remote

onClause.chpl

```
config const verbose = false;
var total = 0,
    done = false;

…

on Locales[1] {
  if !done {
    if verbose then
      writef("Adding locale 1's contribution");
    total += computeMyContribution();
  }
}
```



2D HEAT DIFFUSION EXAMPLE

```
make run-heat_2D
make run-heat_2D_dist
make run-heat_2D_buffers
```

- **See 'heat_2D.*.chpl' in the Chapel examples**
  - 'heat_2D.chpl' - shared memory parallel version that runs in locale 0
  - 'heat_2D_dist.chpl' - parallel and distributed version that is the same as 'heat_2D.chpl' but with distributed arrays
  - 'heat_2D_dist_buffers.chpl' - parallel and distributed version that copies to neighbors landing pad and then into local halos

- **Some things to try out with these variants**

```
chpl heat_2D.chpl
./heat_2D -nl 1

--nt 10 --nx=2048 --ny=2048    # decreases the number of time steps
                               # and reduces the size of the domain
                               # along each dimension from default 4096
```

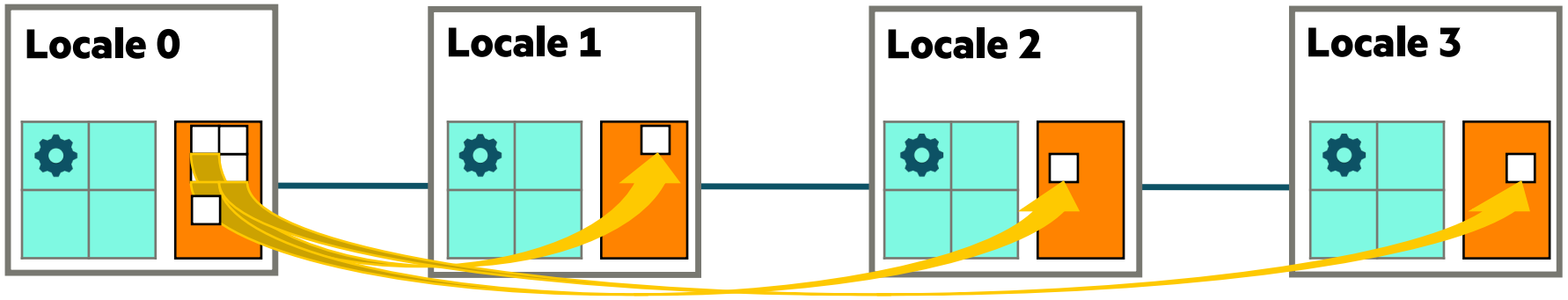
ARRAY-BASED PARALLELISM AND LOCALITY

basics-distarr.chpl

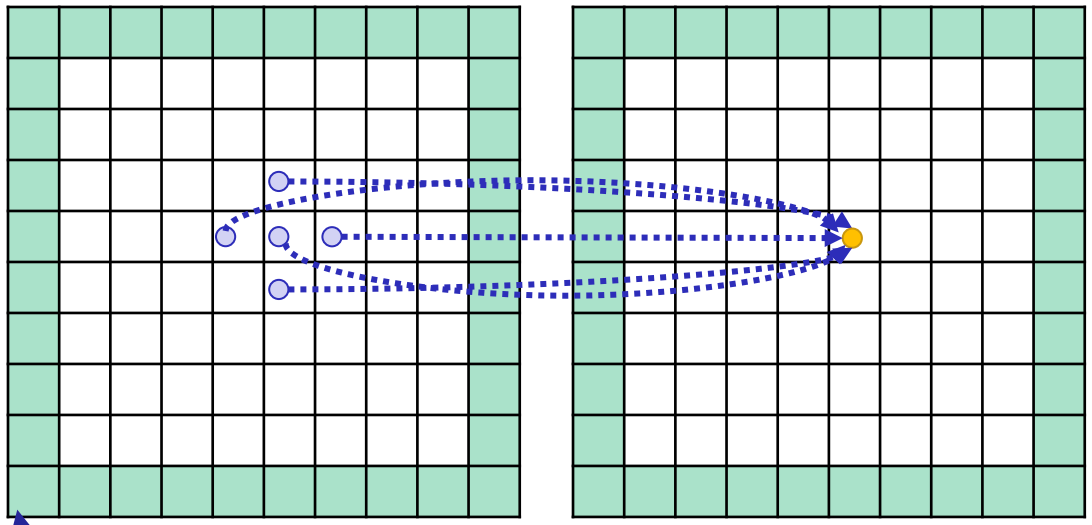
```
writeln("Hello from locale ", here.id);

var A: [1..2, 1..2] real;

use BlockDist;

var D = Block.createDomain({1..2, 1..2});
var B: [D] real;
B = A;
```

Chapel also supports distributed domains (index sets) and arrays

They also result in parallel distributed computation



PARALLEL HEAT DIFFUSION IN HEAT_2D.CHPL



- 2D heat diffusion PDE

$$\frac{\partial u}{\partial t} = \nu \frac{\partial^2 u}{\partial x^2} + \nu \frac{\partial^2 u}{\partial y^2}$$

Simplified form for below assume **Δx=Δy**, and let **α=νΔt/Δx²**

- Solving for next temperatures at each time step using finite difference method

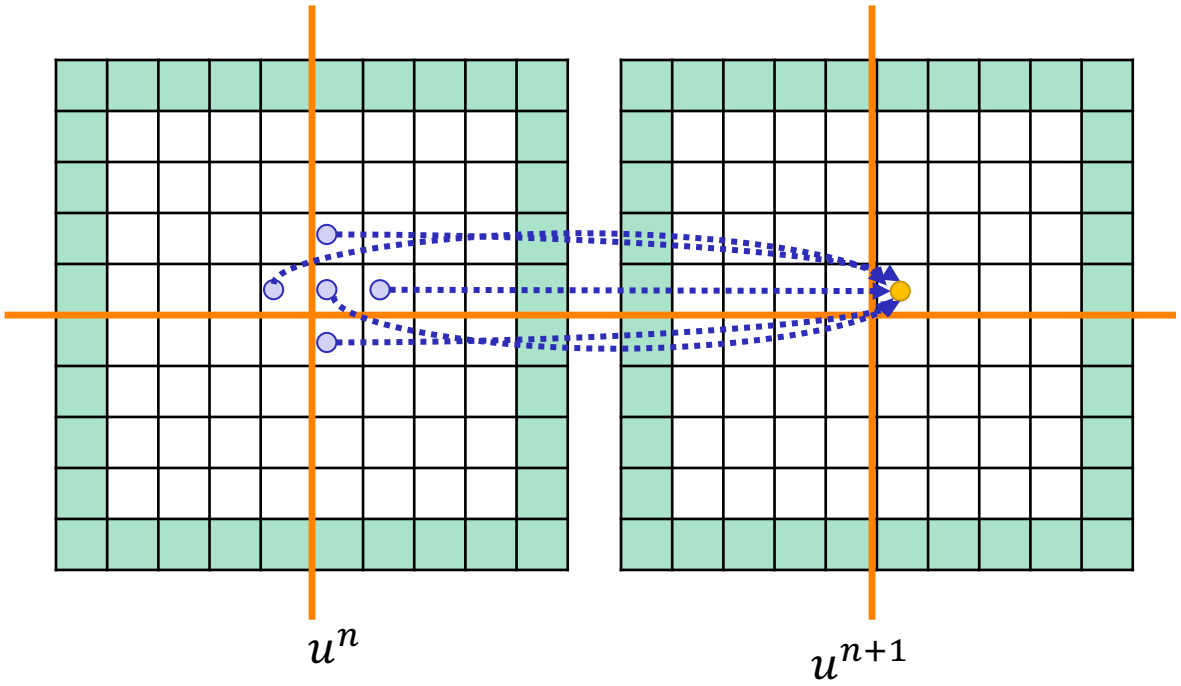
$$u_{i,j}^{n+1} = u_{i,j}^n + \alpha\left(u_{i+1,j}^n + u_{i-1,j}^n - 4u_{i,j}^n + u_{i,j+1}^n + u_{i,j-1}^n\right)$$

- All updates in a timestep can be done in parallel

```
forall (i, j) in indicesInner do
    u[i, j] = un[i, j] + alpha *
      (un[i, j-1] + un[i-1, j] + un[i+1, j] +
       un[i, j+1] - 4 * un[i, j]);
```

- Output is the mean and standard deviation of all the values and time to solution

DISTRIBUTED AND PARALLEL HEAT DIFFUSION IN HEAT_2D_DIST.CHPL



- Declaring 'u' array

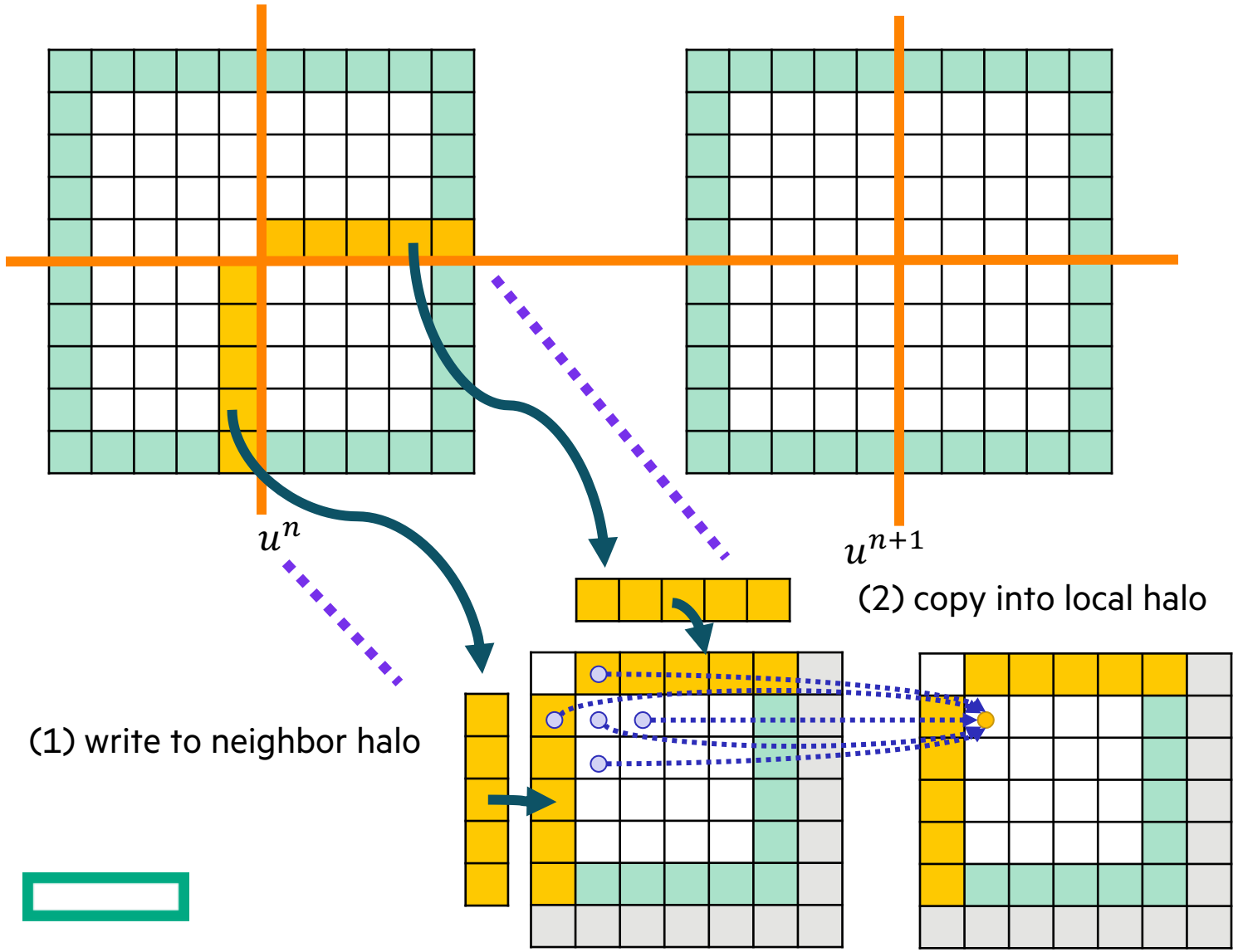
```
const indices = {0..<nx, 0..<ny}
var u: [indices] real;
```

- Declaring 'u' array as distributed

```
const indices = {0..<nx, 0..<ny},
      INDICES = Block.createDomain(indices);
var u: [INDICES] real;
```

- Reads that cross the distribution boundary will result in a remote get

HALO BUFFER OPTIMIZATION IN HEAT_2D_DIST_BUFFERS.CHPL



- Each locale has own copies of 'u' and 'un' subdomains with a one-cell halo
- (1) Array assignment writes edge values into neighbors' halo landing pads
- (2) copy into local halo
- (3) compute next u in parallel locally

(3) compute next u in parallel locally

HALO BUFFER OPTIMIZATION CODE

```
const indices = {0..<nx, 0..<ny},
      indicesInner = indices.expand(-1),
      INDICES = Block.createDomain(indices);
const u: [INDICES] real;
...
var   LOCALE_DOM = Block.createDomain(u.targetLocales().domain);
var   haloArrays: [LOCALE_DOM][0..<4] haloArray;
param N = 0, S = 1, E = 2, W = 3;
...
for 1..nt {
  haloArrays[tidX, tidY-1][E].v = uLocal2[.., WW+1];
  ...
  b.barrier();
  uLocal1 <=> uLocal2;

  uLocal1[.., WW] = haloArrays[tidX, tidY][W].v;
  ...
  forall (i,j) in localIndicesInner do
    uLocal2[i,j] = uLocal1[i,j] + alpha*(uLocal1[i-1,j] + uLocal1[i+1,j]
      + uLocal1[i,j-1] + uLocal1[i,j+1] - 4*uLocal1[i,j]);
  b.barrier();
}
```

Declare and distribute 'u' array.

Declare North, South, East, and West halo arrays per locale

Copy local edge results into neighbor's halo array. 'tidX' and 'tidY' are the locale's task id X and Y coordinates. Using array slicing in 'uLocal2[..,WW+1]'.

Copy halo array into local halo.

Compute u[I,j] in local subdomain.

Barrier over all locales

2D HEAT DIFFUSION EXAMPLE

```
make run-heat_2D
make run-heat_2D_dist
make run-heat_2D_dist_buffers
```

- **See 'diffusion/heat_2D.*.chpl' in the Chapel examples**
  - 'heat_2D.chpl' - shared memory parallel version that runs in locale 0
  - 'heat_2D_dist.chpl' - parallel and distributed version that is the same as 'heat_2D.chpl' but with distributed arrays
  - 'heat_2D_dist_buffers.chpl' - parallel and distributed version that copies to neighbors landing pad and then into local halos

- **Concepts illustrated**
  - 'forall' provides distributed and shared memory parallelism when do a 'forall' over the 2D Block distributed array
  - 'heat_2D_dist.chpl' version doesn't do any special handling of the halo exchange
  - 'heat_2D_dist_buffers.chpl' shows an optimization that explicitly copies subarrays into buffers

IMAGE PROCESSING EXAMPLE

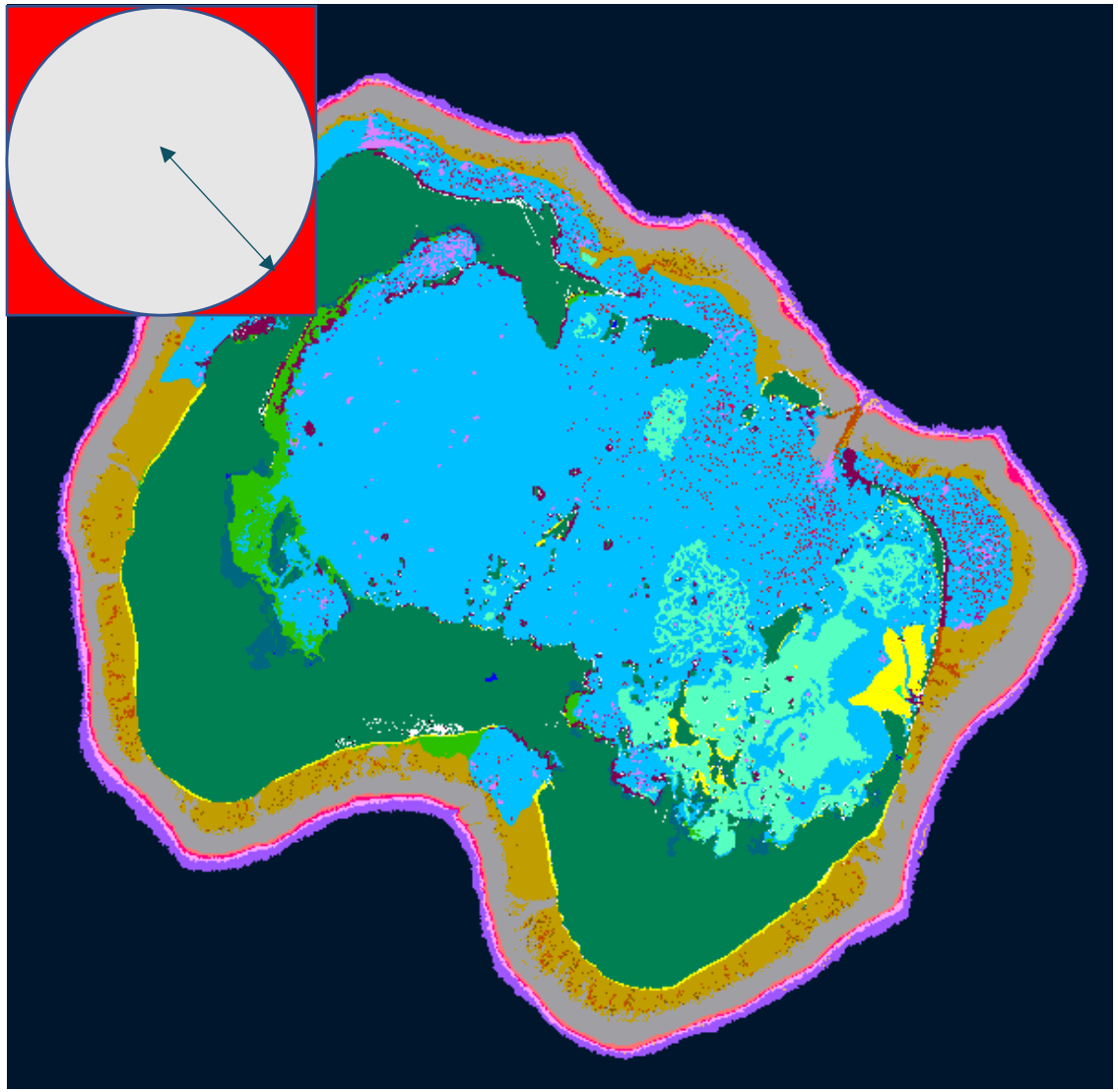
- **See 'image_analysis/' subdirectory in the Chapel examples**
  - Coral reef diversity analysis written by Scott Bachman
  - Reads a single file in parallel
  - Uses distributed and shared memory parallelism
  - Is being used and modified by Scott and collaborators for climate research

- **'image_analysis/README' explains how to compile and run it**

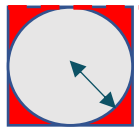
```
cd image_analysis
chpl main.chpl --fast
./main -nl 2 --in_name=banda_ai --map_type=benthic --window_size=100000
```

IMAGE PROCESSING FOR CORAL REEF DISSIMILARITY

- **Analyzing images for coral reef diversity**
  - Important for prioritizing interventions
- **Algorithm implemented productively**
  - Add up weighted values of all points in a neighborhood, i.e., convolution over image
  - Developed by Scott Bachman, NCAR scientist who is a visiting scholar on the Chapel team
  - Scott started learning Chapel in Sept 2022, started Coral Reef app in Dec 2022, already had collaborators presenting results in Feb 2023
  - Last week with ~5 lines changed, ran on a GPU
- **Performance**
  - Less than 300 lines of Chapel code scales out to 100s of processors on Cheyenne (NCAR)
  - Full maps calculated in *seconds*, rather than days



Distributed Parallelism: Divide the domain into “strips” and allocate a task per strip



Task 1

Task 2

…

Task (n-1)

Task n

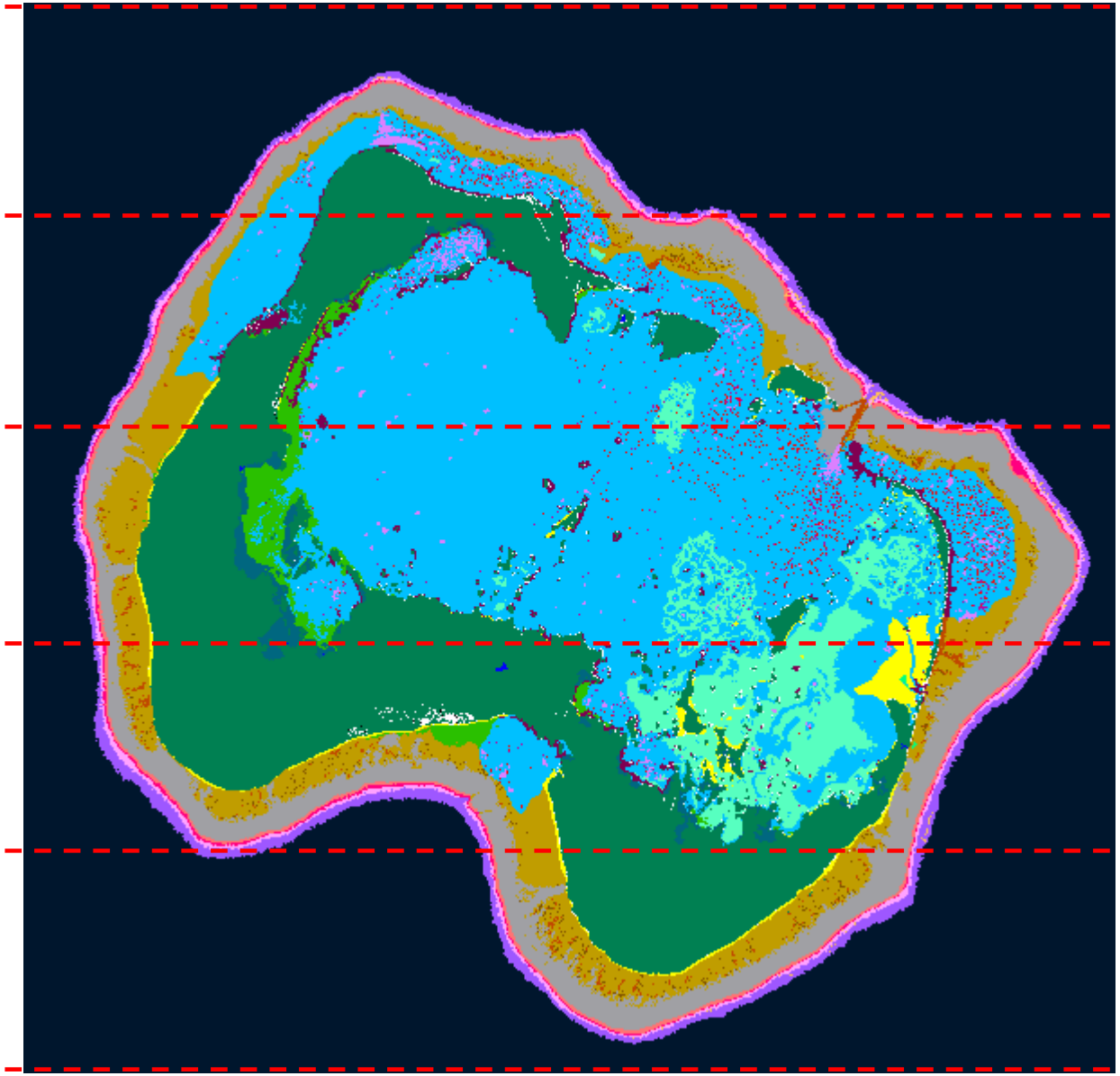


IMAGE PROCESSING EXAMPLE

- **See 'image_analysis/' subdirectory in the Chapel examples**
  - Coral reef diversity analysis written by Scott Bachman
  - Reads a single file in parallel
  - Uses distributed and shared memory parallelism
  - Is being used and modified by Scott and collaborators for climate research

- **'image_analysis/README' explains how to compile and run it**

- **Concepts illustrated**
  - User-defined modules
  - Reading a single file in parallel
  - Sparse domains used to create masks in 'distance_mask.chpl'
  - Creating a 1D block distribution by reshaping the 'Locales' array
  - Gets to locale 0 will occur for some smaller arrays that live on locale 0

GPU SUPPORT IN CHAPEL

- **Generate code for GPUs**
  - Support for NVIDIA and AMD GPUs
  - Exploring Intel support
- **Chapel code calling CUDA examples**
  - https://github.com/chapel-lang/chapel/blob/main/test/gpu/interop/stream/streamChpl.chpl
  - https://github.com/chapel-lang/chapel/blob/main/test/gpu/interop/cuBLAS/cuBLAS.chpl
- **Key concepts**
  - Using the 'locale' concept to indicate execution and data allocation on GPUs
  - 'forall' and 'foreach' loops are converted to kernels
  - Arrays declared within GPU sublocale code blocks are allocated on the GPU
- **For more info…**
  - https://chapel-lang.org/docs/technotes/gpu.html

gpuExample.chpl

```
use GpuDiagnostics;
startGpuDiagnostics();

var operateOn =
if here.gpus.size>0 then here.gpus
                    else [here,];

// Same code can run on GPU or CPU
coforall loc in operateOn do on loc {
 var A : [1..10] int;
 foreach a in A do a+=1;
  writeln(A);
}

stopGpuDiagnostics();
writeln(getGpuDiagnostics());
```

STREAM TRIAD: DISTRIBUTED MEMORY, GPUS AND CPUS

stream-ep.chpl

```
config var n = 1_000_000,
           alpha = 0.01;

coforall loc in Locales do on loc {
  cobegin {
      coforall gpu in here.gpus do on gpu {
        var A, B, C: [1..n] real;
        A = B + alpha * C;
      }
      {
        var A, B, C: [1..n] real;
        A = B + alpha * C;
      }
    }
  }
}
```

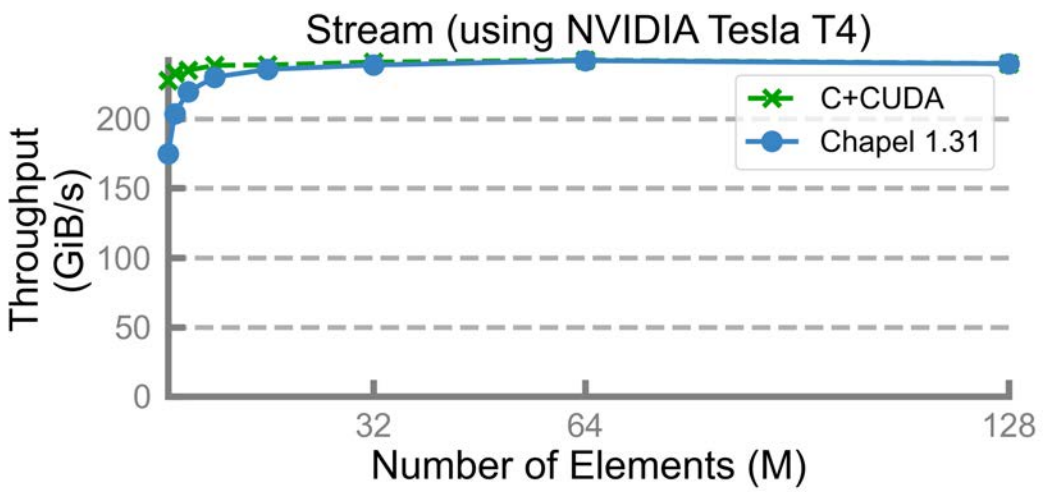
'cobegin { ... }' creates a task per child statement

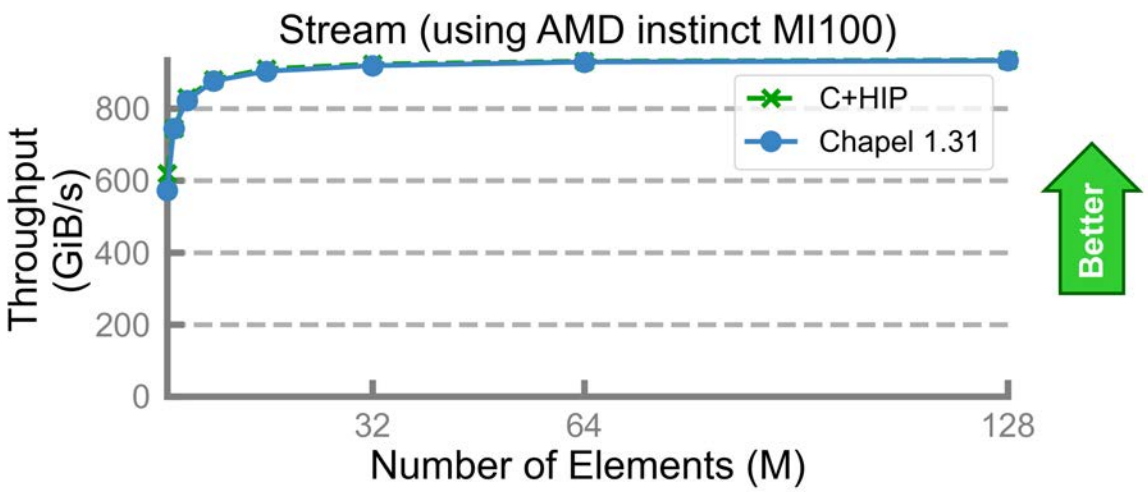
one task runs our multi-GPU triad

the other runs the multi-CPU triad

This program uses all CPUs and GPUs across all of your compute nodes

STREAM TRIAD: PERFORMANCE VS. REFERENCE VERSIONS

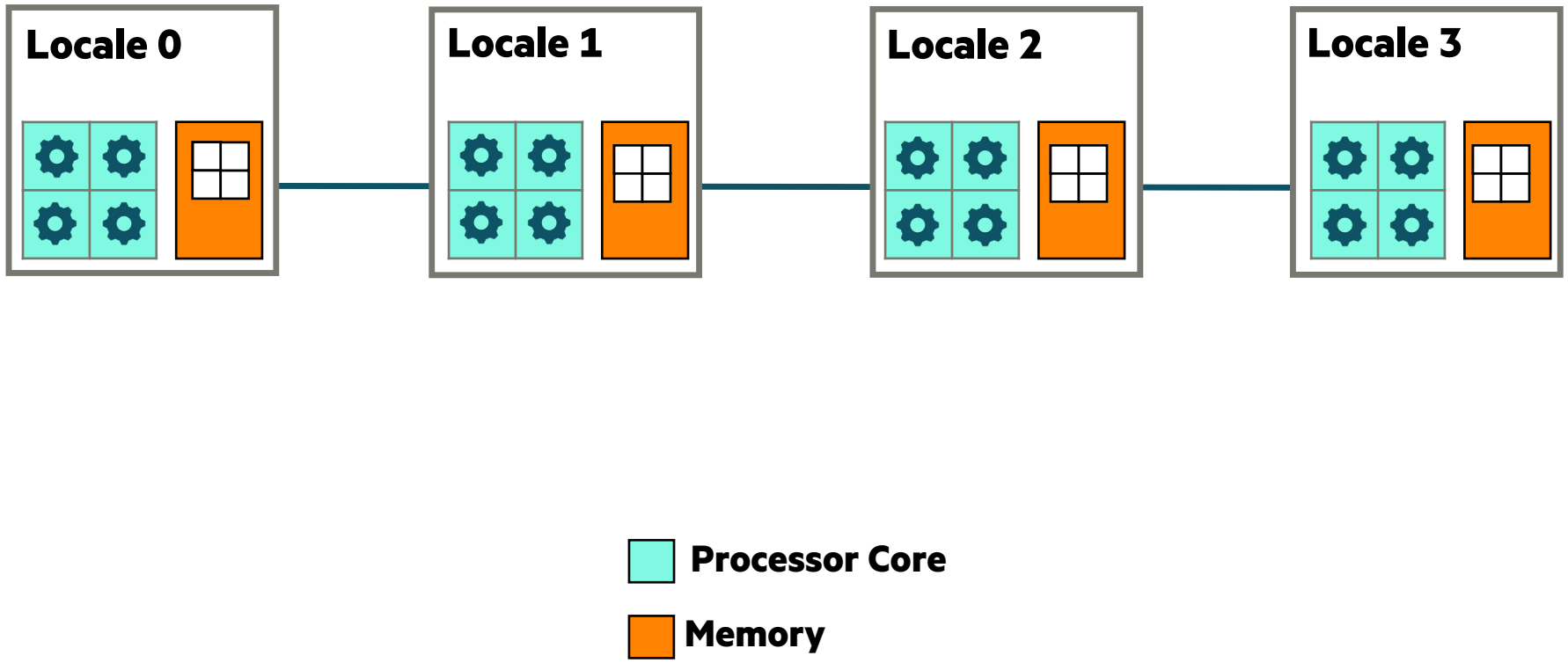




Performance vs. reference versions has become competitive as of the last release

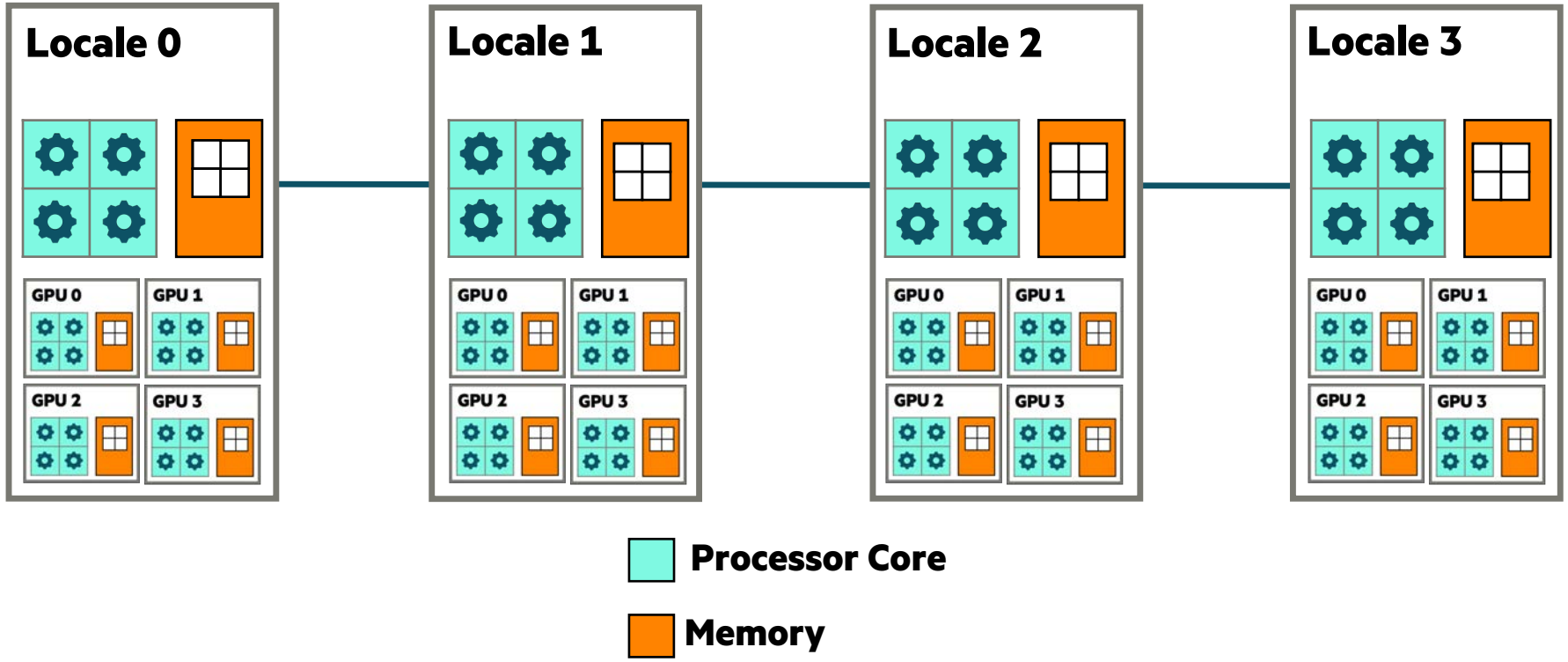
KEY CONCERNS FOR SCALABLE PARALLEL COMPUTING

1. **parallelism:** What tasks should run simultaneously?
2. **locality:** Where should tasks run? Where should data be allocated?
   - complicating matters, compute nodes now often have GPUs with their own processors and memory



KEY CONCERNS FOR SCALABLE PARALLEL COMPUTING

1. **parallelism:** What tasks should run simultaneously?
2. **locality:** Where should tasks run? Where should data be allocated?
   - complicating matters, compute nodes now often have GPUs with their own processors and memory
   - we represent these as *sub-locales* in Chapel



STREAM TRIAD: DISTRIBUTED MEMORY, CPUS ONLY

stream-glbl.chpl

```
config const n = 1_000_000,
             alpha = 0.01;

use BlockDist;

const Dom = Block.createDomain({1..n});
var A, B, C: [Dom] real;

A = B + alpha * C;
```

These programs are both CPU-only

Nothing refers to GPUs,
explicitly or implicitly

stream-ep.chpl

```
config const n = 1_000_000,
             alpha = 0.01;

coforall loc in Locales {
  on loc {
    var A, B, C: [1..n] real;
    A = B + alpha * C;
  }
}
```

STREAM TRIAD: DISTRIBUTED MEMORY, GPUS ONLY

stream-ep.chpl

```
config const n = 1_000_000,
             alpha = 0.01;

coforall loc in Locales do on loc {

    coforall gpu in here.gpus do on gpu {
      var A, B, C: [1..n] real;
      A = B + alpha * C;
    }



}
```

Use a similar ‘coforall’ + ‘on’ idiom to run a Triad concurrently on each of this locale’s GPUs

This is a GPU-only program

Nothing other than coordination code runs on the CPUs

STREAM TRIAD: DISTRIBUTED MEMORY, GPUS AND CPUS

stream-ep.chpl

```
config const n = 1_000_000,
             alpha = 0.01;

coforall loc in Locales do on loc {
  cobegin {
    coforall gpu in here.gpus do on gpu {
      var A, B, C: [1..n] real;
      A = B + alpha * C;
    }
    {
      var A, B, C: [1..n] real;
      A = B + alpha * C;
    }
  }
}
```

'cobegin { ... }' creates a task per child statement

one task runs our multi-GPU triad

the other runs the multi-CPU triad

This program uses all CPUs and GPUs across all of our compute nodes

OTHER CHAPEL EXAMPLES & PRESENTATIONS

- **Primers**
  - https://chapel-lang.org/docs/primers/index.html
- **Blog posts for Advent of Code**
  - https://chapel-lang.org/blog/index.html
- **Test directory in main repository**
  - https://github.com/chapel-lang/chapel/tree/main/test
- **Presentations**
  - https://chapel-lang.org/presentations.html

TUTORIAL SUMMARY

- **Takeaways**
  - Chapel is a PGAS programming language designed to leverage parallelism
  - It is being used in some large production codes
  - Our team is responsive to user questions and would enjoy having you participate in our community
- **How to get more help**
  - Ask the Chapel team and users questions on discourse, gitter, or stack overflow
  - Also feel free to email me at michelle.strout@hpe.com
- **Engaging with the community**
  - Share your sample codes with us and your research community!
  - Join us at our free, virtual workshop in June, https://chapel-lang.org/CHIUW.html

CHAPEL RESOURCES

Chapel homepage: https://chapel-lang.org

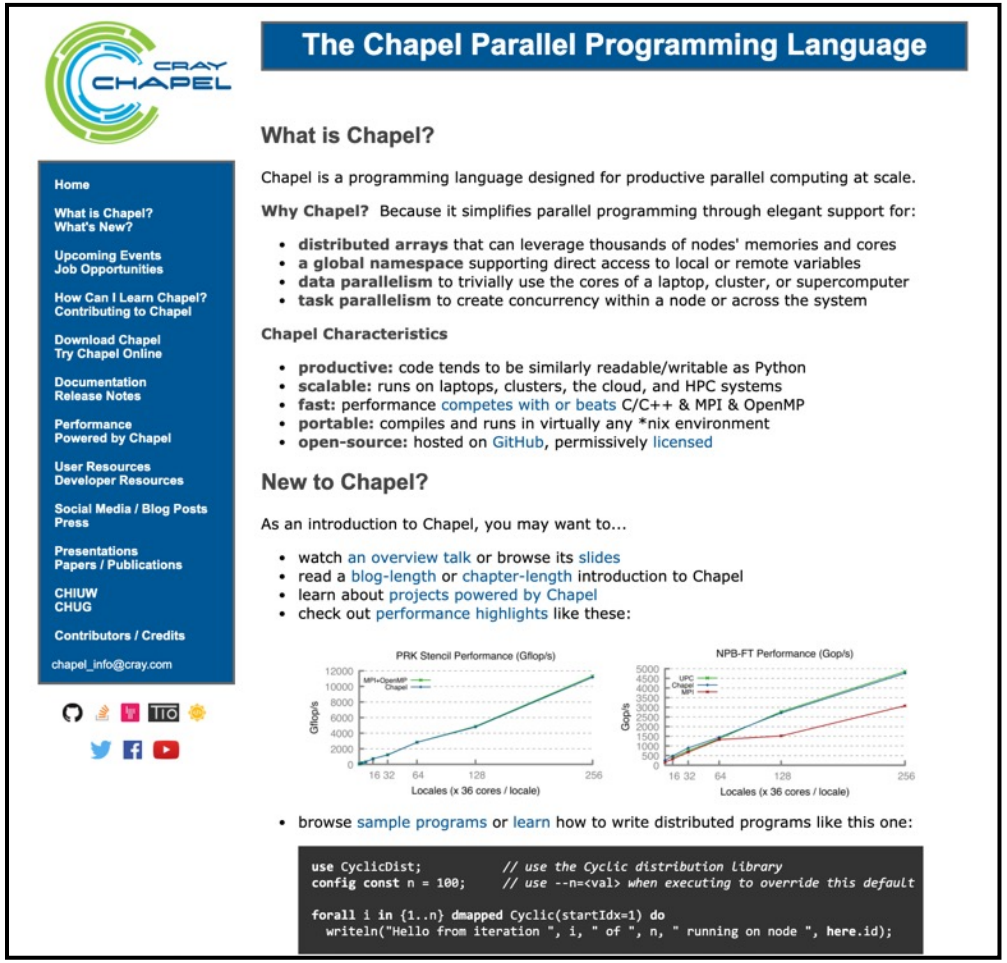
- (points to all other resources)

Social Media:

- Twitter: @ChapelLanguage
- Facebook: @ChapelLanguage
- YouTube: http://www.youtube.com/c/ChapelParallelProgrammingLanguage

Community Discussion / Support:

- Discourse: https://chapel.discourse.group/
- Gitter: https://gitter.im/chapel-lang/chapel
- Stack Overflow: https://stackoverflow.com/questions/tagged/chapel
- GitHub Issues: https://github.com/chapel-lang/chapel/issues



CURRENT CHAPEL TEAM AT HPE

BACKUP SLIDES AND ADDITIONAL CONTENT

GENERAL TIPS WHEN GETTING STARTED WITH CHAPEL (ALSO IN README)

- Online **documentation** is here: https://chapel-lang.org/docs/
  - The primers can be particularly valuable for learning a concept: https://chapel-lang.org/docs/primers/index.html
    - These are also available from a Chapel release in '$CHPL_HOME/examples/primers/'
      or '$CHPL_HOME/test/release/examples/primers/' if you clone from GitHub

- When debugging, **almost anything in Chapel can be printed out** with 'writeln(expr1, expr2, expr3);'
  - Types can be printed after being cast to strings, e.g. 'writeln("Type of ", expr, " is ", expr.type:string);'
  - A quick way to print a bunch of values out clearly is to print a tuple made up of them 'writeln((x, y, z));'

- Once your code is correct, before doing any performance timings, be sure to re-compile with **'--fast'**
  - Turns on optimizations, turns off safety checks, slows down compilation, speeds up execution significantly
  - Then, when you go back to making modifications, be sure to stop using `--fast` in order to turn checks back on

- For vim / emacs users, **syntax highlighters** are in $CHPL_HOME/highlight
  - Imperfect, but typically better than nothing
  - Emacs MELPA users may want to use the chapel-mode available there (better in many ways, weird in others)

OTHER TASK PARALLEL FEATURES

- **begin / cobegin statements:** the two other ways of creating tasks

```
begin stmt;      // fire off an asynchronous task to run 'stmt'
```

```
cobegin {        // fire off a task for each of 'stmt1', 'stmt2', ...
  stmt1;
  stmt2;
  stmt3;
  ...
}                // wait here for these tasks to complete before proceeding
```

- **atomic / synchronized variables:** types for safe data sharing & coordination between tasks

```
var sum: atomic int;     // supports various atomic methods like .add(), .compareExchange(), ...
var cursor: sync int;    // stores a full/empty bit governing reads/writes, supporting .readEF(), .writeEF()
```

- **task intents / task-private variables:** control how variables and tasks relate

```
coforall i in 1..niters with (ref x, + reduce y, var z: int) { … }
```

SPECTRUM OF CHAPEL FOR-LOOP STYLES

for loop: each iteration is executed serially by the current task

- predictable execution order, similar to conventional languages

foreach loop: all iterations executed by the current task, but in no specific order

- a candidate for vectorization, SIMD execution on GPUs

forall loop: all iterations are executed by one or more tasks in no specific order

- implemented using one or more tasks, locally or distributed, as determined by the iterand expression

```
forall i in 1..n do …                    // forall loops over ranges use local tasks only
forall (i,j) in {1..n, 1..n} do …        // ditto for local domains…
forall elem in myLocArr do …             // …and local arrays
forall elem in myDistArr do …            // distributed arrays use tasks on each locale owning part of the array
forall i in myParIter(…) do …            // you can also write your own iterators that use the policy you want
```

coforall loop: each iteration is executed concurrently by a distinct task

- explicit parallelism; supports synchronization between iterations (tasks)

SIDEBAR: PROMOTION OF SCALAR SUBROUTINES

- Any function or operator that takes scalar arguments can be called with array expressions instead

```
proc foo(x: real, y: real, z: real) {
  return x**y + 10*z;
}
```

- Interpretation is similar to that of a zippered forall loop, thus:

```
C = foo(A, 2, B);
```

is equivalent to:

```
forall (c, a, b) in zip(C, A, B) do
  c = foo(a, 2, b);
```

as is:

```
C = A**2 + 10*B;
```

- So, in the Jacobi computation,

```
abs(A[D] - Temp[D]);
```

==

```
forall (a,t) in zip(A[D], Temp[D]) do abs(a - t);
```

BERKELEY LAB
LAWRENCE BERKELEY NATIONAL LABORATORY

U.S. DEPARTMENT OF ENERGY

UPC++: An Asynchronous RMA/RPC Library for Distributed C++ Applications

Amir Kamil

https://go.lbl.gov/CUF23
pagoda@lbl.gov

Applied Mathematics and Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, California, USA

What does UPC++ offer?

Asynchronous behavior

- **RMA**:
  - Get/put to a remote location in another address space
  - Low overhead, zero-copy, one-sided communication.
- **RPC: Remote Procedure Call**:
  - Moves computation to the data

Design principles for performance

- All communication is syntactically explicit
- All communication is asynchronous: futures and promises
- Scalable data structures that avoid unnecessary replication

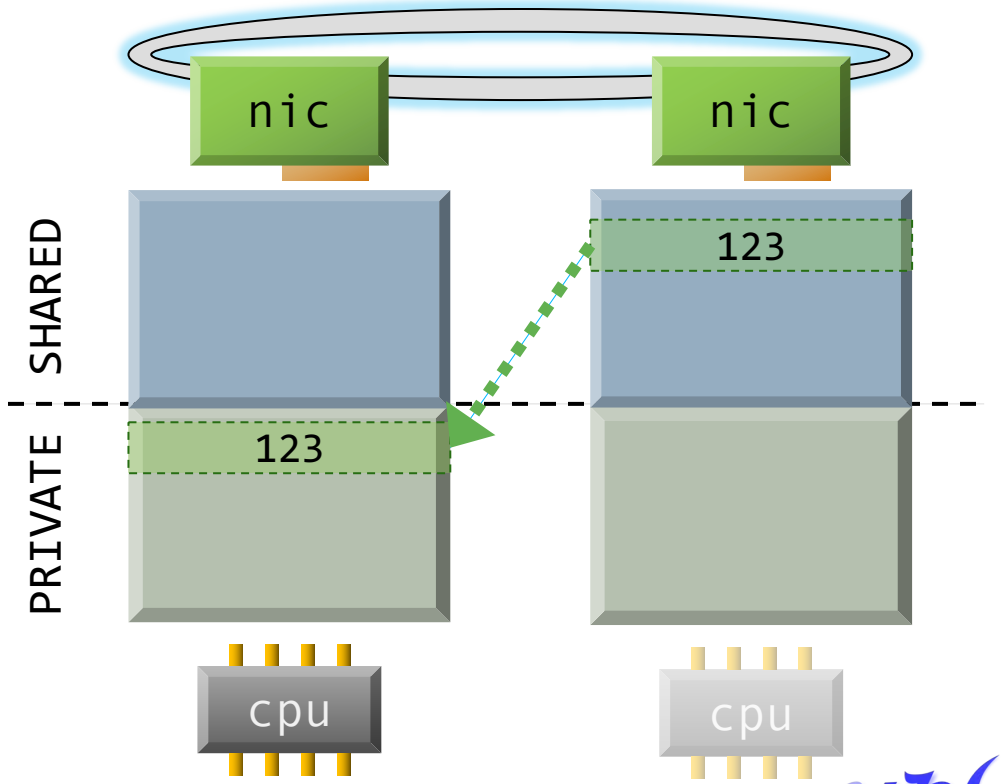
Review: Asynchronous communication (RMA)

By default, all communication operations are split-phased

- **Initiate** operation
- **Wait** for completion
  A future holds a value and a state: ready/not-ready

```
global_ptr<int> gptr1 = ...;
future<int> f1 = rget(gptr1);
// unrelated work...
int t1 = f1.wait();
```

Wait returns the result when the rget completes

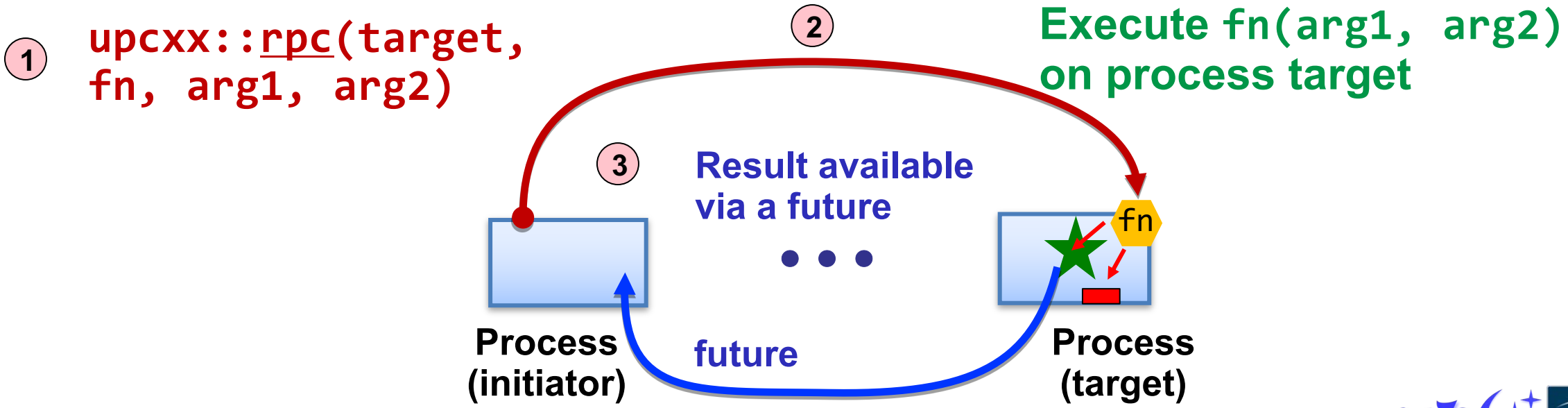


Review: Remote procedure call (RPC)

Execute a function on another process, sending arguments and returning an optional result

1. Initiator injects the RPC to the *target* process
2. Target process executes `fn(arg1, arg2)` at some later time determined at the target
3. Result becomes available to the initiator via the future

Many RPCs can be active simultaneously, hiding latency



Compiling and running a UPC++ program

UPC++ provides tools for ease-of-use

Compiler wrapper:

```
$ upcxx -g hello-world.cpp -o hello-world.exe
```

- Invokes a normal backend C++ compiler with the appropriate arguments (`-I`/`-L` etc).
- We also provide other mechanisms for compiling
  - `upcxx-meta`
  - CMake package

Launch wrapper:

```
$ upcxx-run -N 1 -n 4 ./hello-world.exe
```

- Arguments similar to other familiar tools
- Also support launch using platform-specific tools, such as **`srun`**, **`jsrun`** and **`aprun`**.

Using UPC++ at US DOE Office of Science Centers

UPC++ installations available at ALCF (Polaris, Theta, Sunspot), NERSC (Perlmutter), and OLCF (Summit, Frontier, Crusher)

Info and examples for all three centers are available from
https://upcxx.lbl.gov/site

Also contains links to UPC++ source and build instructions

UPC++ works on laptops, workstations, and clusters too

Instructions for the hands-on activities in this tutorial:
https://go.lbl.gov/CUF23

Hands-on: Hello world compile and run

Everything needed for the hands-on activities is at:

https://go.lbl.gov/CUF23

Online materials include:

- Module info for NERSC Perlmutter, OLCF Frontier, and other machines
- Download links to install UPC++

Once you have set up your environment, copied the tutorial materials, and changed to the `cuf23/upcxx` directory:

Command to run in the terminal

Copy this and change the number after `-n` to use a different number of processes, e.g.:
`upcxx-run -N 1 -n 8 ./hello-world`

```
$ make run-hello-world
upcxx hello-world.cpp -Wall  -o hello-world
upcxx-run -N 1 -n 4 ./hello-world
Hello world from process 2 out of 4 processes
Hello world from process 0 out of 4 processes
Hello world from process 3 out of 4 processes
Hello world from process 1 out of 4 processes
```

Example: Hello world

```
make run-hello-world
```

```
#include <iostream>
#include <upcxx/upcxx.hpp>
using namespace std;

int main() {
  upcxx::init();
  cout << "Hello world from process "
       << upcxx::rank_me()
       << " out of " << upcxx::rank_n()
       << " processes" << endl;
  upcxx::finalize();
}
```

Set up UPC++ runtime

Close down UPC++ runtime

```
Hello world from process 0 out of 4 processes
Hello world from process 2 out of 4 processes
Hello world from process 3 out of 4 processes
Hello world from process 1 out of 4 processes
```

Hello world with RPC (synchronous)

```
make run-hello-world-rpc-to-0
```

We can rewrite hello world by having each process launch an RPC to process 0

```
int main() {
  upcxx::init();
  for (int i = 0; i < upcxx::rank_n(); ++i) {
    if (upcxx::rank_me() == i) {

      upcxx::rpc(0, [](int rank) {
        cout << "Hello from process " << rank << endl;
      }, upcxx::rank_me()).wait();
    }

    upcxx::barrier();
  }
  upcxx::finalize();
}
```

C++ lambda function

Wait for RPC to complete before continuing

Rank number is the argument to the lambda

Barrier prevents any process from proceeding until all have reached it

Futures

RPC returns a *future* object, which represents a computation that may or may not be complete

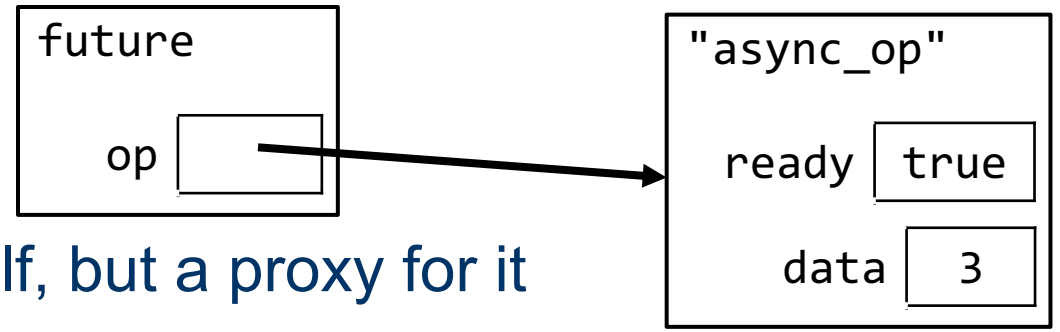
Calling `wait()` on a future causes the current process to wait until the future is ready


```
upcxx::future<> fut =
  upcxx::rpc(0, [](int rank) {
    cout << "Hello from process " << rank << endl;
  }, upcxx::rank_me());

fut.wait();
```

What is a future?

A future is a handle to an asynchronous operation, which holds:

- The status/readiness of the operation
- The results (zero or more values) of the completed operation



The future is not the result itself, but a proxy for it

The `wait()` method blocks until a future is ready and returns the result

```
upcxx::future<int> fut = /* ... */;
int result = fut.wait();
```

The `then()` method can be used instead to attach a callback to the future

Overlapping communication

Rather than waiting on each RPC to complete, we can launch every RPC and then wait for each to complete

```
vector<upcxx::future<int>> results;
for (int i = 0; i < upcxx::rank_n(); ++i) {
  upcxx::future<int> fut = upcxx::rpc(i, []() {
    return upcxx::rank_me();
  }));
  results.push_back(fut);
}

for (auto fut : results) {
  cout << fut.wait() << endl;
}
```

We'll see better ways to wait on groups of asynchronous operations later

`make run-jac1d`

1D 3-point Jacobi in UPC++

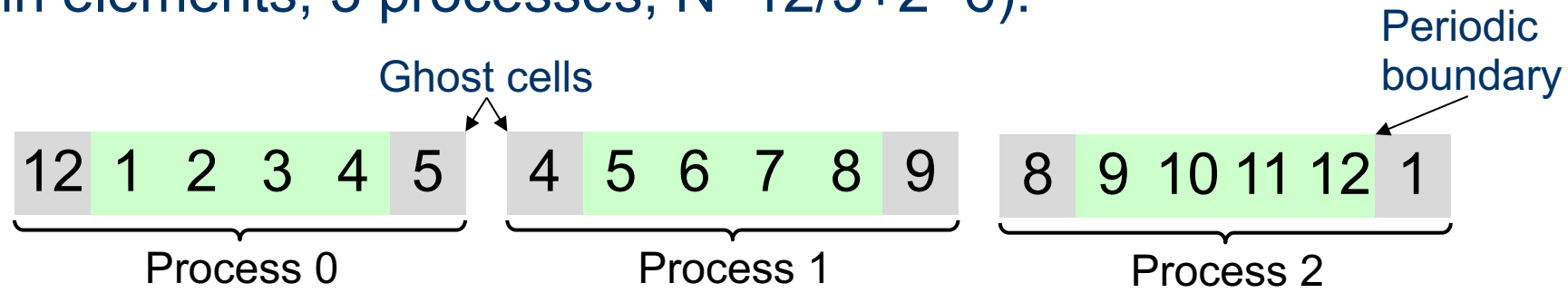
Iterative algorithm that updates each grid cell as a function of its old value and those of its immediate neighbors

Out-of-place computation requires two grids

Local grid size

```
for (long i = 1; i < N - 1; ++i)
  new_grid[i] = 0.25 *
      (old_grid[i - 1] + 2*old_grid[i] + old_grid[i + 1]);
```

Sample data distribution of each grid
(12 domain elements, 3 processes, N=12/3+2=6):

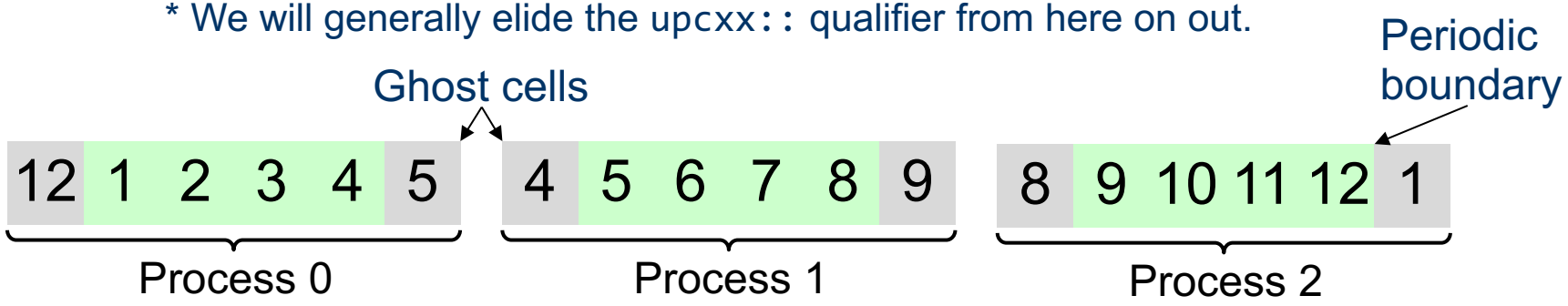


Jacobi boundary exchange (version 1)

RPCs can refer to static variables, so we use them to keep track of the grids

```
double *old_grid, *new_grid;

double get_cell(long i) {
  return old_grid[i];
}

...

double val = rpc(right, get_cell, 1).wait();
```

* We will generally elide the `upcxx::` qualifier from here on out.



Jacobi computation (version 1)

We can use RPC to communicate boundary cells

```
future<double> left_ghost = rpc(left, get_cell, N-2);
future<double> right_ghost = rpc(right, get_cell, 1);
```

Initiate communication

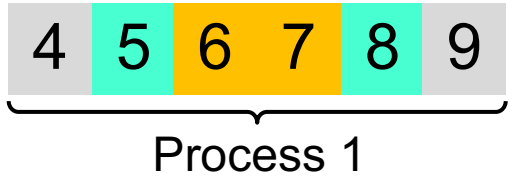
```
for (long i = 2; i < N - 2; ++i)
  new_grid[i] = 0.25 *
    (old_grid[i-1] + 2*old_grid[i] + old_grid[i+1]);
```

Do interior computation

```
new_grid[1] = 0.25 *
  (left_ghost.wait() + 2*old_grid[1] + old_grid[2]);

new_grid[N-2] = 0.25 *
  (old_grid[N-3] + 2*old_grid[N-2] + right_ghost.wait());
```

Wait for communication to complete and do boundary computation

```
std::swap(old_grid, new_grid);
```

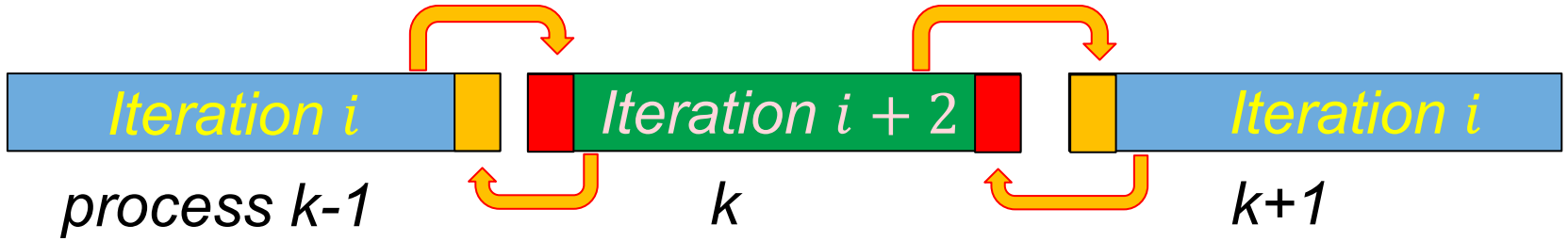


Race conditions

Since processes are unsynchronized, it is possible that a process can move on to later iterations while its neighbors are still on previous ones

- One-sided communication decouples data movement from synchronization for better performance

A *straggler* in iteration i could obtain data from a neighbor that is computing iteration $i + 2$, resulting in incorrect values



This behavior is unpredictable and may not be observed in testing

Naïve solution: barriers

Barriers at the end of each iteration provide sufficient synchronization

```
future<double> left_ghost = rpc(left, get_cell, N-2);
future<double> right_ghost = rpc(right, get_cell, 1);

for (long i = 2; i < N - 2; ++i)
  /* ... */;

new_grid[1] = 0.25 *
  (left_ghost.wait() + 2*old_grid[1] + old_grid[2]);
new_grid[N-2] = 0.25 *
  (old_grid[N-3] + 2*old_grid[N-2] + right_ghost.wait());

barrier();
std::swap(old_grid, new_grid);
barrier();
```

Barriers around the swap ensure that incoming RPCs in both this iteration and the next one use the correct grids

One-sided put and get (RMA)

UPC++ provides APIs for one-sided puts and gets

Implemented using network RDMA if available – most efficient way to move large payloads

- Scalar put and get:

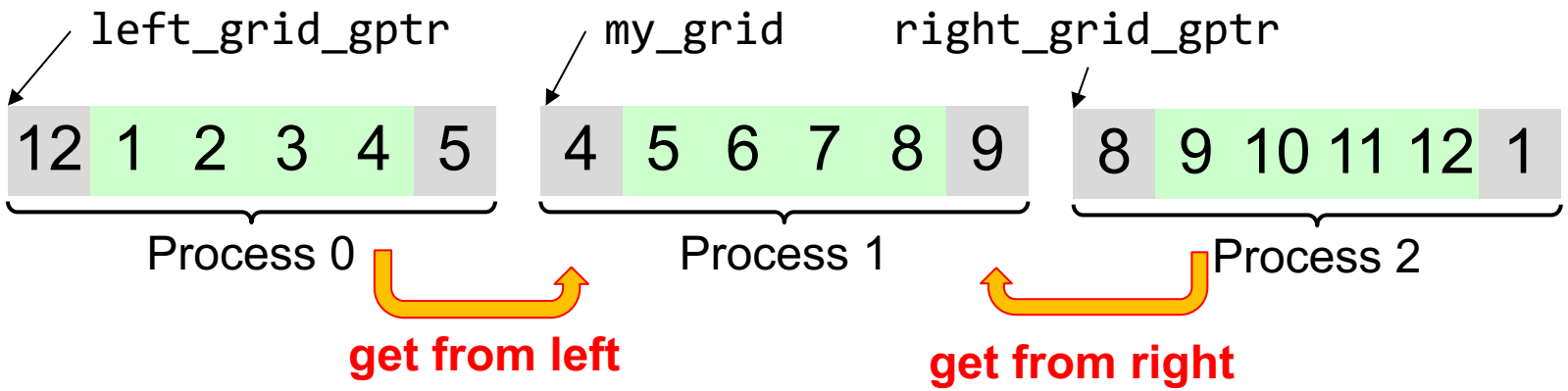
```
global_ptr<int> remote = /* ... */;
future<int> fut1 = rget(remote);
int result = fut1.wait();
future<> fut2 = rput(42, remote);
fut2.wait();
```

- Vector put and get:

```
int *local = /* ... */;
future<> fut3 = rget(remote, local, count);
fut3.wait();
future<> fut4 = rput(local, remote, count);
fut4.wait();
```

Jacobi with ghost cells

Each process maintains *ghost cells* for data from neighboring processes



Assuming we have *global pointers* to our neighbor grids, we can do a one-sided put or get to communicate the ghost data:

```
double *my_grid;
global_ptr<double> left_grid_gptr, right_grid_gptr;
my_grid[0] = rget(left_grid_gptr + N - 2).wait();
my_grid[N-1] = rget(right_grid_gptr + 1).wait();
```

Storage management

Memory must be allocated in the shared segment in order to be accessible through RMA

```
global_ptr<double> old_grid_gptr, new_grid_gptr;
...
old_grid_gptr = new_array<double>(N);
new_grid_gptr = new_array<double>(N);
```

These are not collective calls – each process allocates its own memory, and there is no synchronization

- Explicit synchronization may be required before retrieving another process's pointers with an RPC
- The pointers must be communicated to other processes before they can access the data

Downcasting global pointers

If a process has direct load/store access to the memory referenced by a global pointer, it can *downcast* the global pointer into a raw pointer with `local()`

```
global_ptr<double> old_grid_gptr, new_grid_gptr;
double *old_grid, *new_grid;

void make_grids(size_t N) {
  old_grid_gptr = new_array<double>(N);
  new_grid_gptr = new_array<double>(N);
  old_grid = old_grid_gptr.local();
  new_grid = new_grid_gptr.local();
}
```

Downcasting can also be used to optimize for co-located processes that share physical memory

Jacobi RMA with gets

Each process obtains boundary data from its neighbors with `rget()`

Remote source (global_ptr) Local dest ptr

```
future<> left_get = rget(left_old_grid + N - 2, old_grid, 1);
future<> right_get = rget(right_old_grid + 1, old_grid + N - 1, 1);

for (long i = 2; i < N - 2; ++i)
  /* ... */;
```

Begin asynchronous RMA gets

Overlapped computation on interior cells

Wait for communication, then consume values

```
left_get.wait();
new_grid[1] = 0.25*(old_grid[0] + 2*old_grid[1] + old_grid[2]);

right_get.wait();
new_grid[N-2] = 0.25*(old_grid[N-3] + 2*old_grid[N-2] + old_grid[N-1]);
```

Callbacks

The `then()` method attaches a callback to a future

- The callback will be invoked after the future is ready, with the future's values as its arguments

```
future<> left_update =
  rget(left_old_grid + N - 2, old_grid, 1)
  .then([]() {
    new_grid[1] = 0.25 *
      (old_grid[0] + 2*old_grid[1] + old_grid[2]);
  });

future<> right_update =
  rget(right_old_grid + N - 2)
  .then([](double value) {
    new_grid[N-2] = 0.25 *
      (old_grid[N-3] + 2*old_grid[N-2] + value);
  });
```

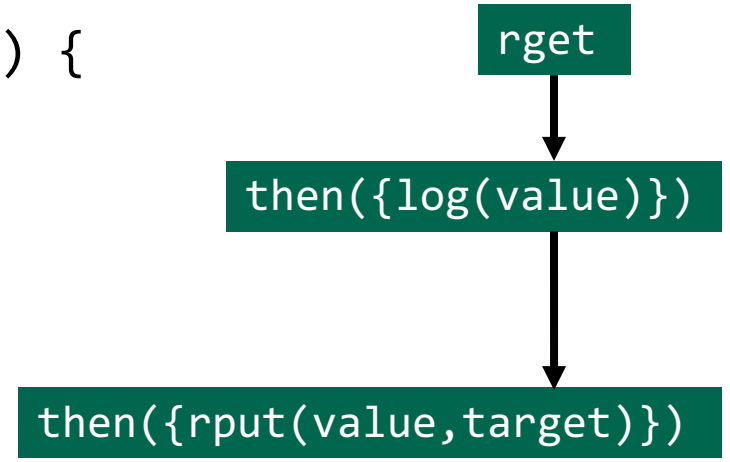
Vector get does not produce a value

Scalar get produces a value

Chaining callbacks

Callbacks can be chained through calls to `then()`

```
global_ptr<int> source = /* ... */;
global_ptr<double> target = /* ... */;
future<int> fut1 = rget(source);
future<double> fut2 = fut1.then([](int value) {
  return std::log(value);
});
future<> fut3 =
  fut2.then([target](double value) {
    return rput(value, target);
  });
fut3.wait();
```



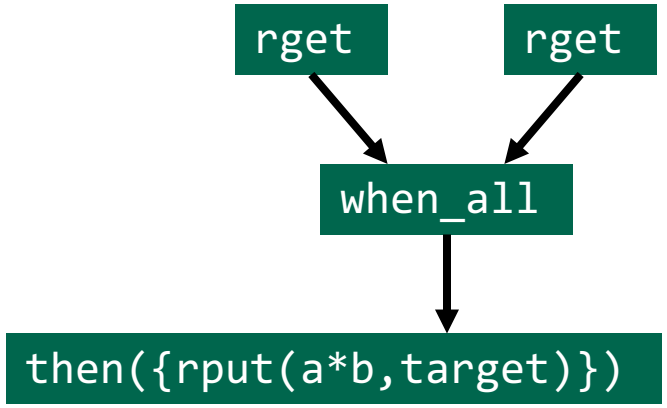
This code retrieves an integer from a remote location, computes its log, and then sends it to a different remote location

Conjoining futures

Multiple futures can be *conjoined* with when_all() into a single future that encompasses all their results

Can be used to specify multiple dependencies for a callback

```
global_ptr<int>    source1 = /* ... */;
global_ptr<double> source2 = /* ... */;
global_ptr<double> target = /* ... */;
future<int>    fut1 = rget(source1);
future<double> fut2 = rget(source2);
future<int, double> both =
    when_all(fut1, fut2);
future<> fut3 =
    both.then([target](int a, double b) {
        return rput(a * b, target);
    });
fut3.wait();
```



Jacobi RMA with puts and conjoining

Each process sends boundary data to its neighbors with rput(), and the resulting futures are conjoined

```
future<> puts = when_all(
    rput(old_grid[1], left_old_grid + N - 1),
    rput(old_grid[N-2], right_old_grid));

for (long i = 2; i < N - 2; ++i)
  /* ... */;

puts.wait();
barrier();

new_grid[1] = 0.25 * (old_grid[0] + 2*old_grid[1] + old_grid[2]);
new_grid[N-2] = 0.25 * (old_grid[N-3] + 2*old_grid[N-2] + old_grid[N-1]);
```

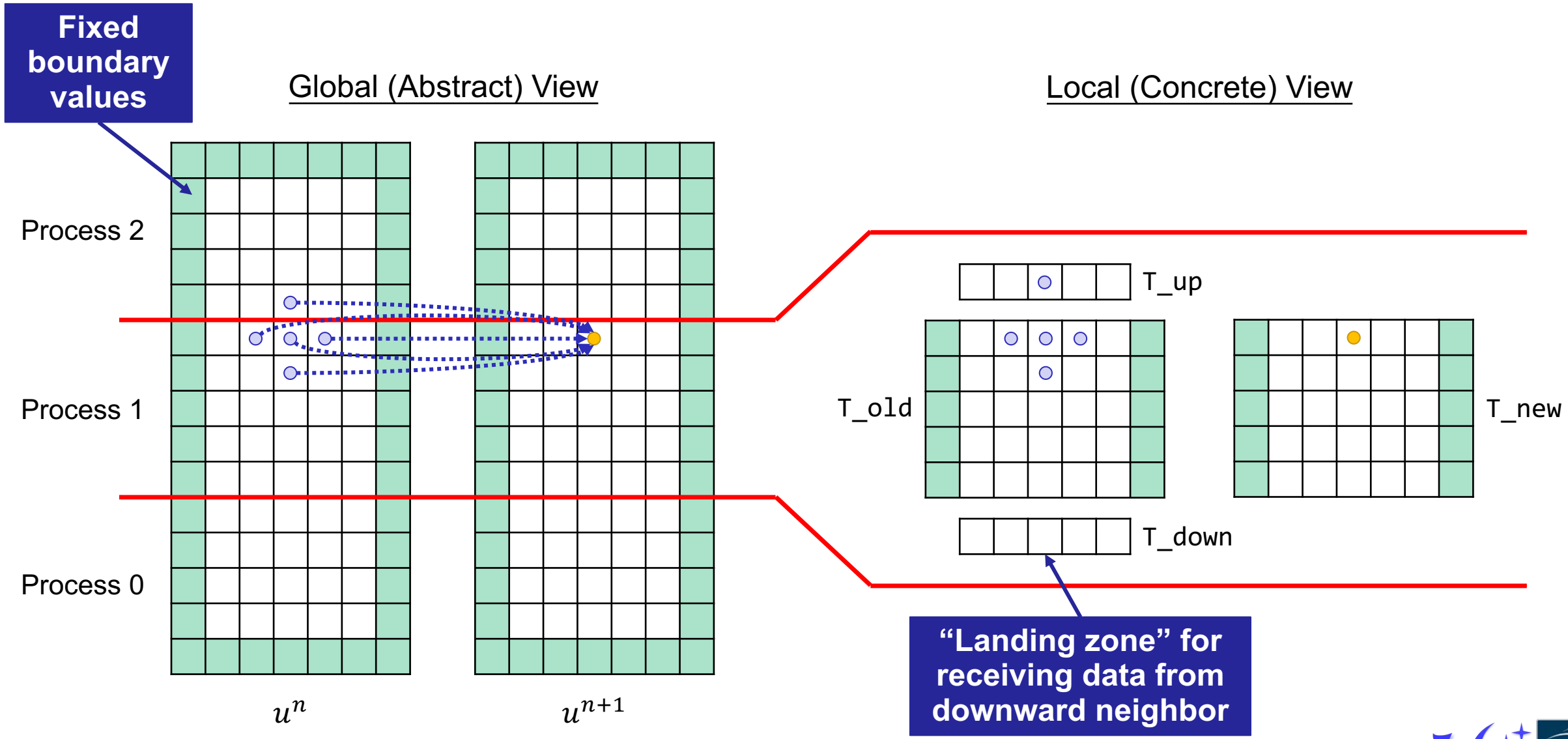
Ensure outgoing puts have completed

Ensure incoming puts have completed

2D heat diffusion data layout

$$u_{i,j}^{n+1} = u_{i,j}^{n} + \alpha\left(u_{i+1,j}^{n} + u_{i-1,j}^{n} - 4u_{i,j}^{n} + u_{i,j+1}^{n} + u_{i,j-1}^{n}\right)$$

```
make run-heat2d
```



2D heat diffusion computation

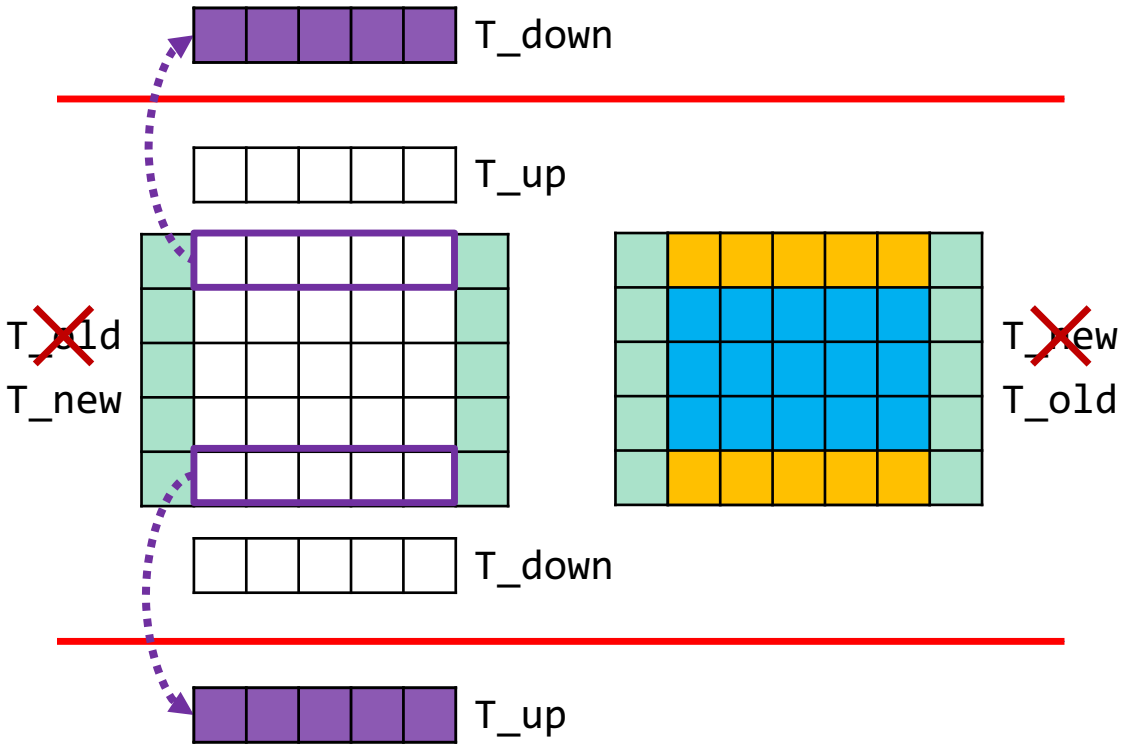
$$u_{i,j}^{n+1} = u_{i,j}^{n} + \alpha\left(u_{i+1,j}^{n} + u_{i-1,j}^{n} - 4u_{i,j}^{n} + u_{i,j+1}^{n} + u_{i,j-1}^{n}\right)$$

```
make run-heat2d
```

Computation loop:

Global pointer to neighbor's landing zone

```
for (int t = 0; t < num_timesteps; t++) {
  // initiate asynchronous puts to neighbors
  future<> fut =
    when_all(rput(T_old, gptr_down, X),
             rput(T_old+offset, gptr_up, X));
  // overlapped computation of interior
  compute_inner_T_new();
  // wait for my puts to complete
  fut.wait();
  // ensure everyone's puts have completed
  barrier();
  // compute boundaries using data received from neighbors
  compute_surface_T_new();
  // set up next timestep
  std::swap(T_new, T_old);
  barrier();
}
```



Distributed objects

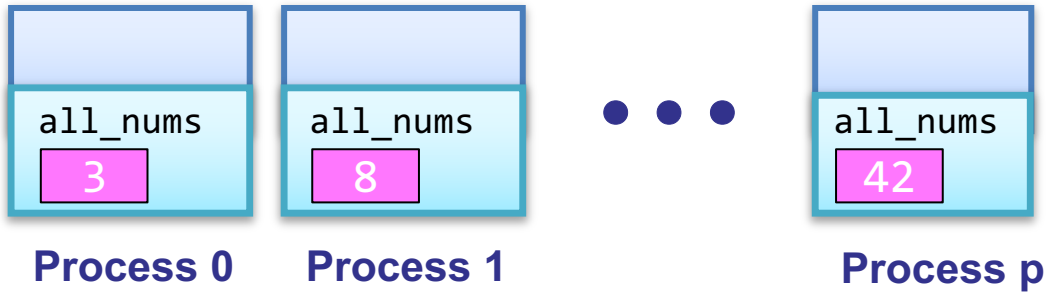
A *distributed object* is an object that is partitioned over a set of processes

```
dist_object<T>(T value, team &team = world());
```

The processes share a universal name for the object, but each has its own local value

Similar in concept to a co-array, but with advantages

- Scalable metadata representation
- Does not require a symmetric heap
- No communication to set up or tear down




```
dist_object<int>
  all_nums(rand());
```

Distributed objects in 2D heat diffusion

Distributed objects can be used to obtain global pointers to other processes' landing zones

```
global_ptr<double> down_in, up_in;
if (lo != 0) {
  down_in = new_array<double>(X);
  T_down = down_in.local();
}
if (hi != Y) {
  up_in = new_array<double>(X);
  T_up = up_in.local();
}
dist_object<global_ptr<double>> dist_up{down_in};
dist_object<global_ptr<double>> dist_down{up_in};
if (lo != 0) gptr_down = dist_down.fetch(down).wait();
if (hi != Y) gptr_up = dist_up.fetch(up).wait();
barrier();
```

Construct landing zones for each neighbor (if necessary)

Construct distributed objects containing pointers to each process's landing zones

Fetch landing-zone pointer from the neighbor below

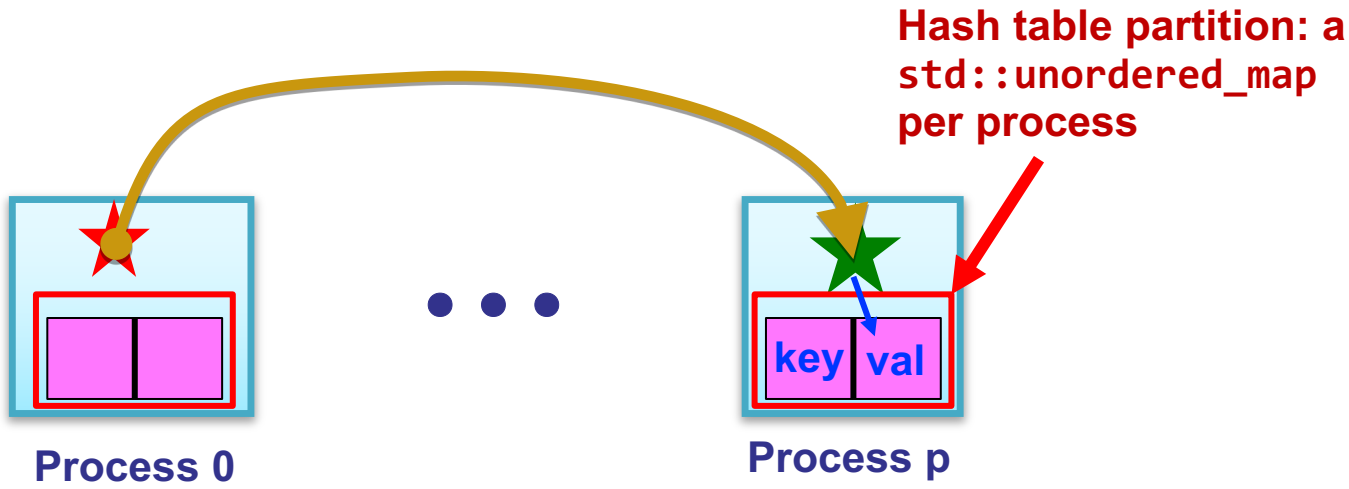
Ensure that all fetches have completed before the distributed objects are destroyed

Hands-on: Distributed hash table (DHT)

```
make run-dmap-insert-test
```

Distributed analog of `std::unordered_map` (similar to Python `dict`, Java `HashMap`)

- Supports insertion and lookup
- We will assume the key and value types are `std::string`
- Represented as a collection of individual unordered maps across processes
- We use RPC to move hash-table operations to the owner



DHT data representation

A distributed object represents the directory of unordered maps

```
class DistrMap {
  using dobj_map_t =
    dist_object<std::unordered_map<std::string, std::string>>;

  // Construct empty map
  dobj_map_t local_map{{}};


  int get_target_rank(const std::string &key) {
    return std::hash<string>{}(key) % rank_n();
  }
};
```

Define an abbreviation for a helper type

Computes owner for the given key

DHT insertion

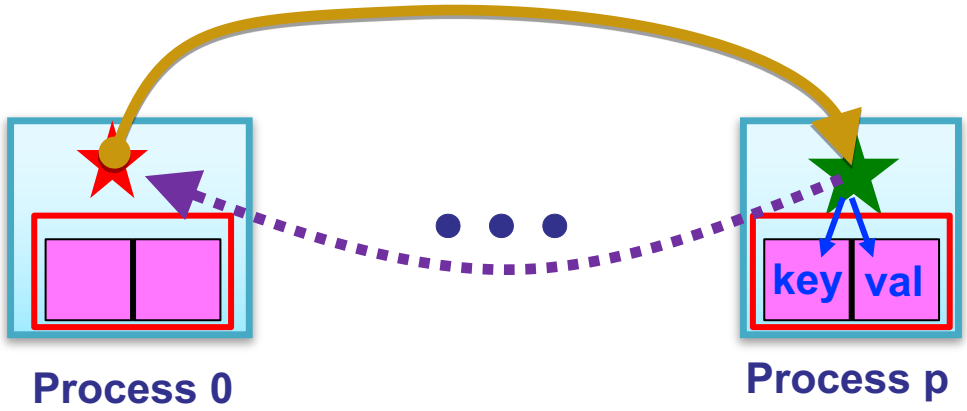
Insertion initiates an RPC to the owner and returns a future that represents completion of the insert

```
future<> insert(const string &key,
                const string &val) {
  return rpc(get_target_rank(key),
    [](dobj_map_t &lmap, const string &key, const string &val) {
      (*lmap)[key] = val;
    }, local_map, key, val);
}
```

Send RPC to the process determined by key hash

Key and value passed as arguments to the remote function

UPC++ uses the distributed object's universal name to look it up on the remote process



DHT find

Find also uses RPC and returns a future

```
future<string> find(const string &key) {
  return rpc(get_target_rank(key),
    [](dobj_map_t &lmap, const string &key) {
      if (lmap->count(key) == 0)
        return string("NOT FOUND");
      else
        return (*lmap)[key];
    }, local_map, key);
}
```

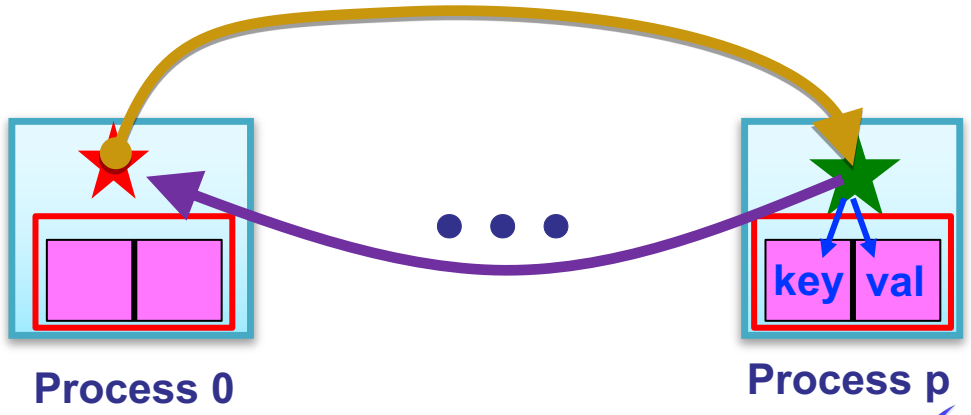
Send RPC to the process determined by key hash

Check whether key exists in local map

Retrieve corresponding value from the local map and return it

UPC++ uses the distributed object's universal name to look it up on the remote process

Key passed as argument to the remote function



Additional DHT operations

```
make run-dmap-erase-update-test
```

```
// Erases the given key from the DHT.
future<> erase(const string &key) {
  return rpc(get_target_rank(key),
             [](dobj_map_t &lmap, const string &key) {
                 lmap->erase(key);
             }, local_map, key);
}

// Replaces the value associated with the given key and returns the old
// value with which it was previously associated.
future<string> update(const string &key,
                      const string &value) {
  return rpc(get_target_rank(key),
             [](dobj_map_t &lmap, const string &key,
                const string &value) {
                  return local_update(*lmap, key, value);
             }, local_map, key, value);
}
```

Lambda to remove the key from the local map at the target

Lambda to update the key in the local map at the target

Helper function to update local map

```
make run-drmap-insert-test
```

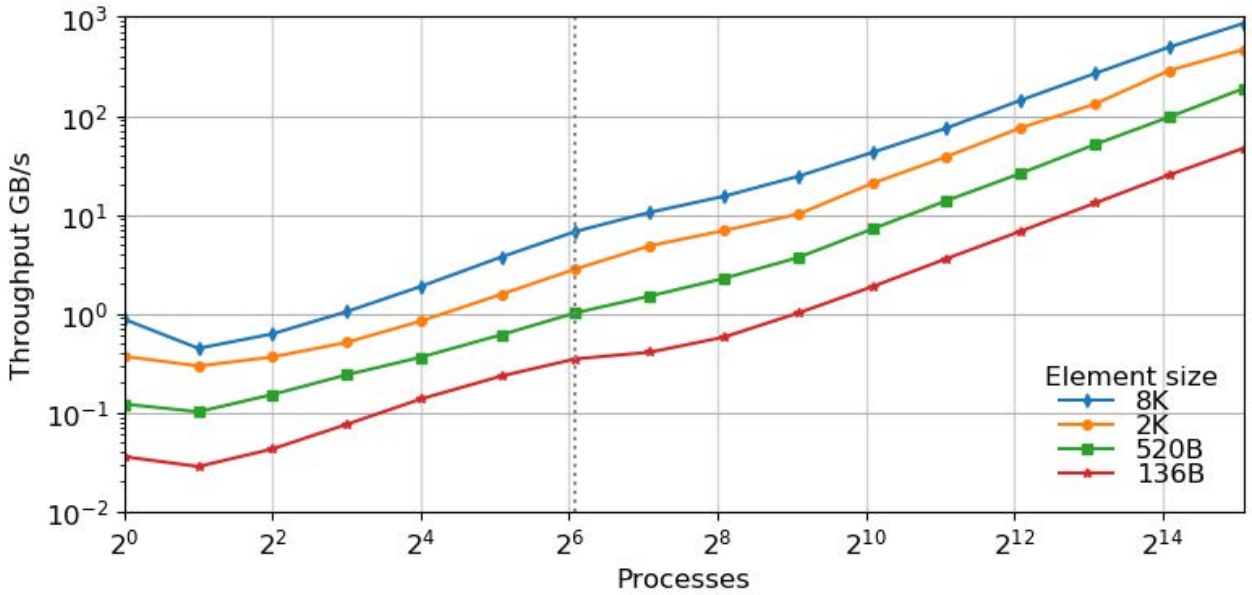
Optimized DHT scales well

Excellent weak scaling up to 32K cores [IPDPS19]

- Randomly distributed keys

RPC and RMA lead to simplified and more efficient design

- Key insertion and storage allocation handled at target
- Without RPC, complex updates would require explicit synchronization and two-sided coordination



Cori @ NERSC (KNL)

Cray XC40

UPC++ advanced features

UPC++ has many advanced features that enable further optimizations

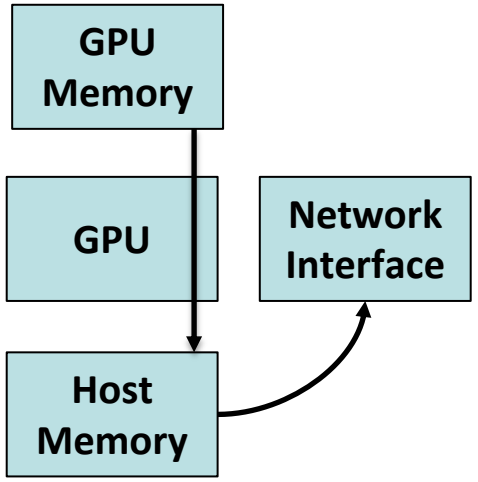
- Team-based barrier, reduction, and broadcast collectives
- Remote atomic operations that utilize hardware offload capabilities of modern networks
- Serialization of complex standard-library and user types in RPC's
- Shared-memory bypass for co-located processes on many-core nodes
- Additional forms of communication completion notification such as promises and "signaling put"
- Non-contiguous RMA with automated packing and aggregation of strided or sparse data
- Memory kinds for data transfer between remote or local host (CPU) and device (e.g. GPU) memory
- …

Memory kinds: Accelerated RMA to/from GPU memory

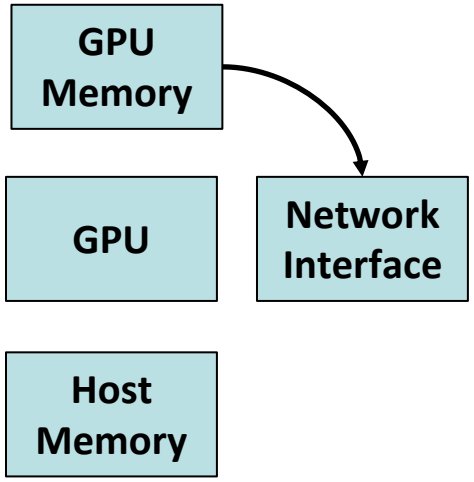
Modern GPUs and NICs can support peer-to-peer data transfers

Example: Put with source on GPU

- In the absence of necessary hardware and OS support:
  1. Data must be copied from GPU memory to host memory
  2. RDMA from host memory's copy
- With support:
  1. RDMA directly from GPU memory (no copies)



Data movement without acceleration



Data movement with acceleration

Memory kinds: Accelerated RMA to/from GPU memory

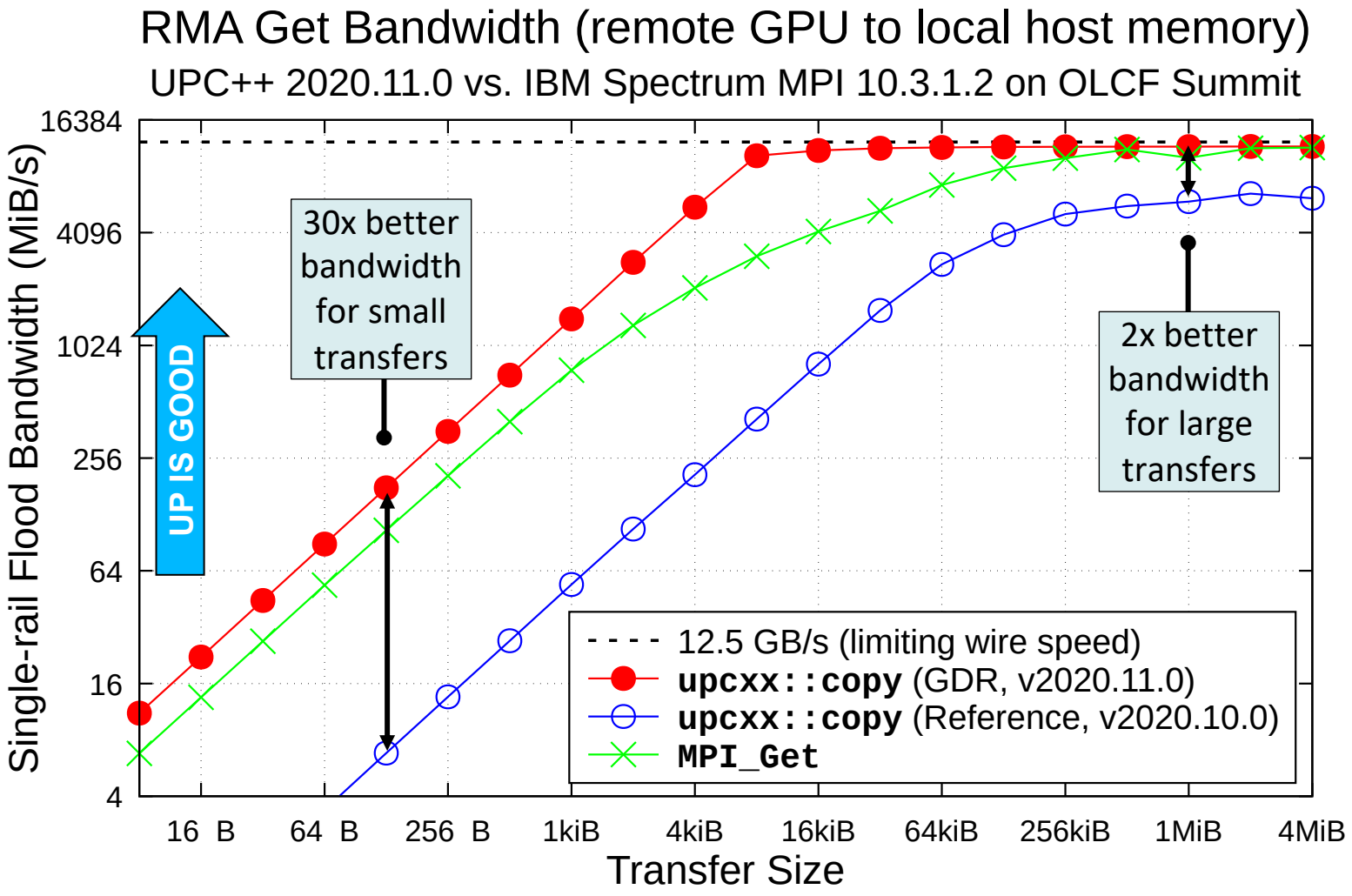
Measurements of flood bandwidth of `upcxx::copy()` on OLCF's Summit

Difference between two consecutive releases shows benefit of GASNet-EX's support for accelerated transfers via Nvidia's "GDR".

- No longer staging through host memory
- Large xfers: 2x better bandwidth
- Small xfers: up to 30x better bandwidth

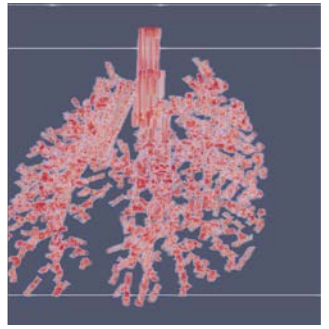
Get operations to/from GPU memory now perform comparably to host memory

Comparisons to MPI RMA in GDR-enabled IBM MPI show UPC++ saturating more quickly to the peak



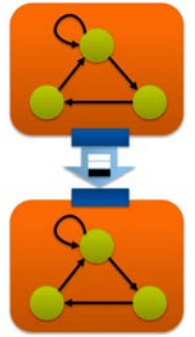
UPC++ results were collecting using the version of the **`cuda_benchmark`** test that appears in the 2020.11.0 release.
MPI results are from **`osu_get_bw`** test in a CUDA-enabled build of OSU Micro-Benchmarks 5.6.3.
All tests were run on OLCF Summit, between two nodes with one process per node, over its EDR InfiniBand network.

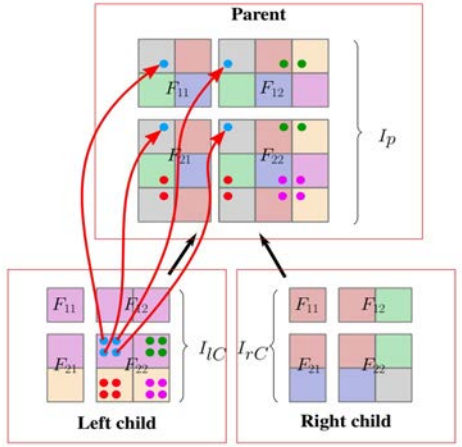
UPC++ applications

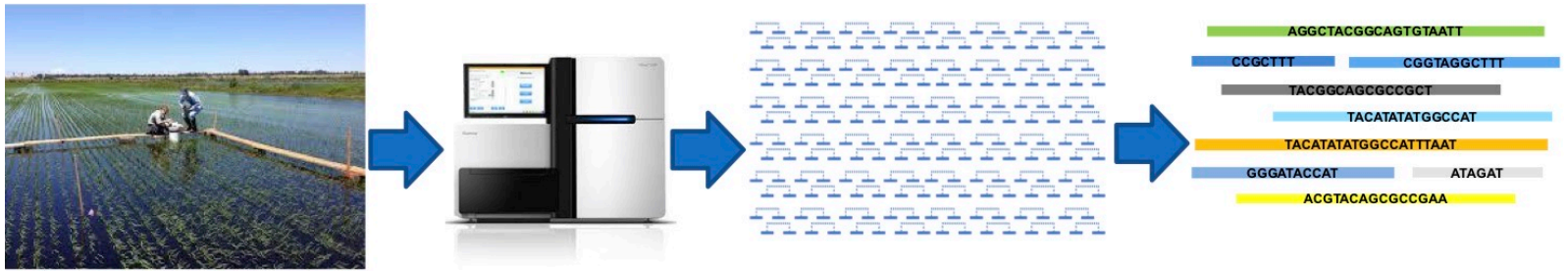


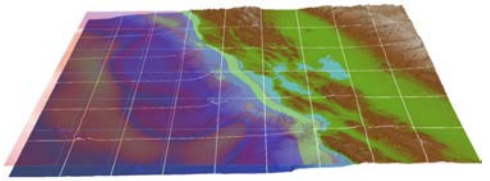
UPC++ has been used successfully in several applications to improve programmer productivity and runtime performance, including:

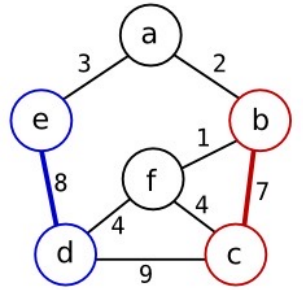
- symPack, a sparse symmetric matrix solver
- SIMCoV, agent-based simulation of lungs with COVID
- MetaHipMer, a genome assembler
- Actor-UPCXX, used in the Pond tsunami simulator
- A UPC++ backend for NWChemEx/TAMM
- UPC++ DepSpawn, a library for data-flow computing
- Mel-UPX, half-approximate graph matching solver











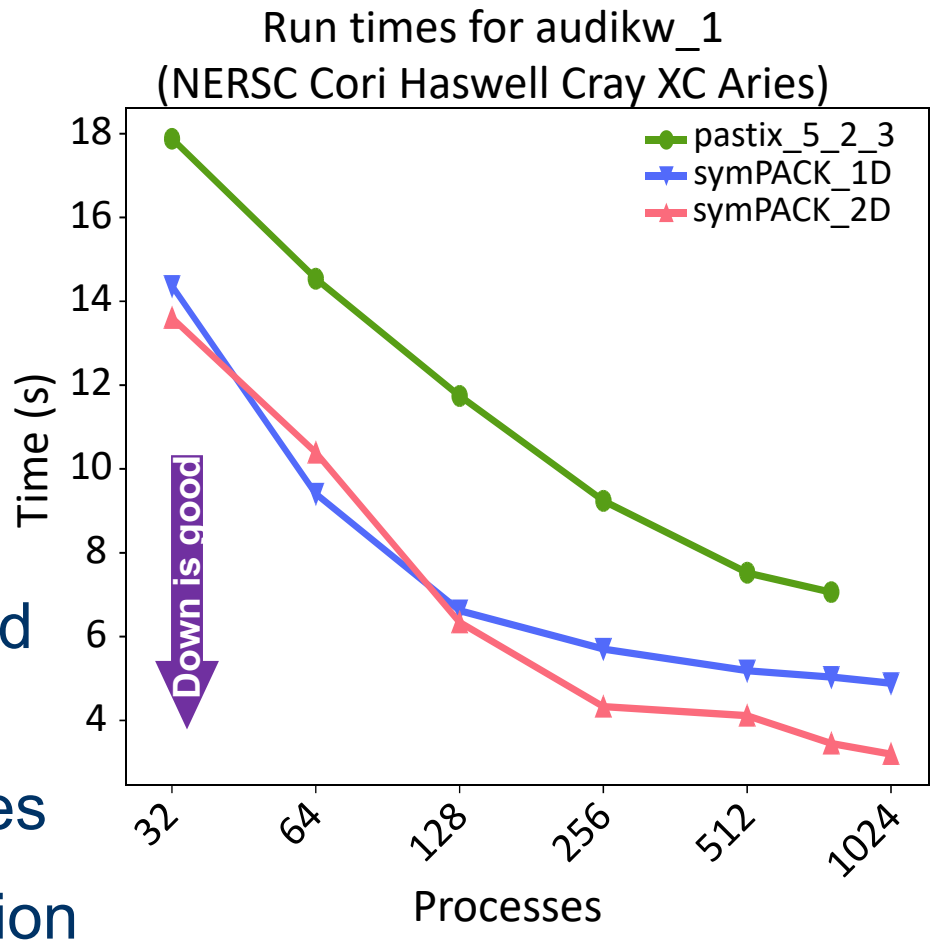
symPACK: UPC++ provides productivity + performance

Productivity

- RPC allowed very simple notify-get system
- Interoperates with MPI
- Non-blocking API

Reduced communication costs

- Low overhead reduces the cost of fine-grained communication
- Overlap communication via asynchrony/futures
- Increased efficiency in the extend-add operation
- Outperform state-of-the-art sparse symmetric solvers



https://upcxx.lbl.gov/sympack

SIMCoV: Spatial Model of Immune Response to Viral Lung Infection

Model the entire lung at the cellular level:

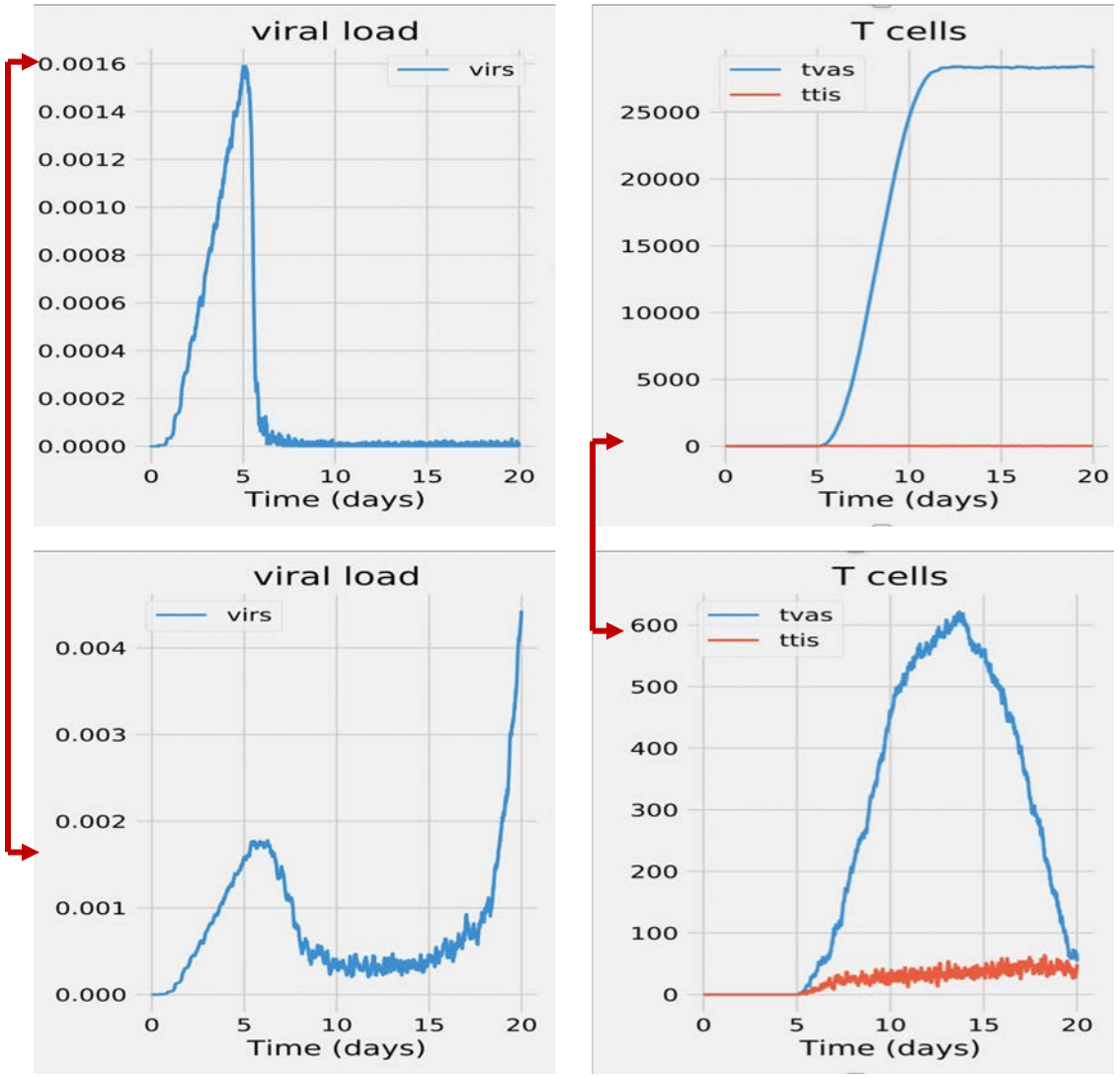
- 100 billion epithelial cells
- 100s of millions of T cells
- Complex branching fractal structure
- Time resolution in seconds for 20 to 30 days

SIMCoV in UPC++

- Distributed 3D spatial grid
- Particles move over time, but computation is localized
- Load balancing is tricky: active near infections

UPC++ benefits:

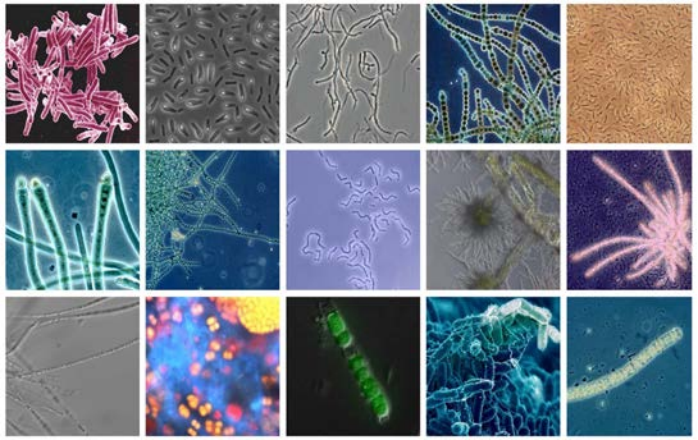
- Heavily uses RPCs
- Easy to develop first prototype
- Good distributed performance and avoids explicit locking
- Extensive support for asynchrony improves computation/communication overlap



https://github.com/AdaptiveComputationLab/simcov

ExaBiome: Exascale Solutions for Microbiome Analysis

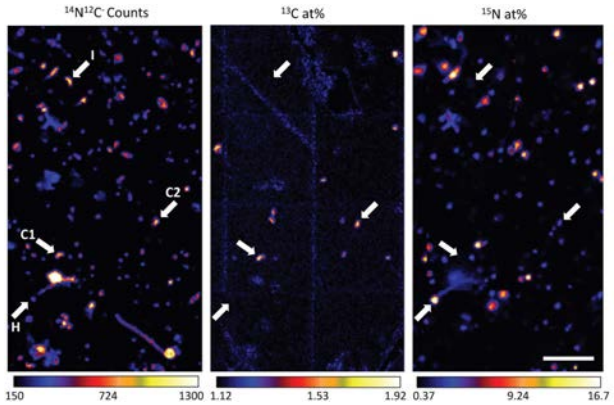
What happens to microbes after a wildfire? (1.5TB)



What are the microbial dynamics of soil carbon cycling? (3.3 TB)

How do microbes affect disease and growth of switchgrass for biofuels (4TB)

What at the seasonal fluctuations in a wetland mangrove? (1.6 TB)



Combine genomics with isotope tracing methods for improved functional understanding (8TB)

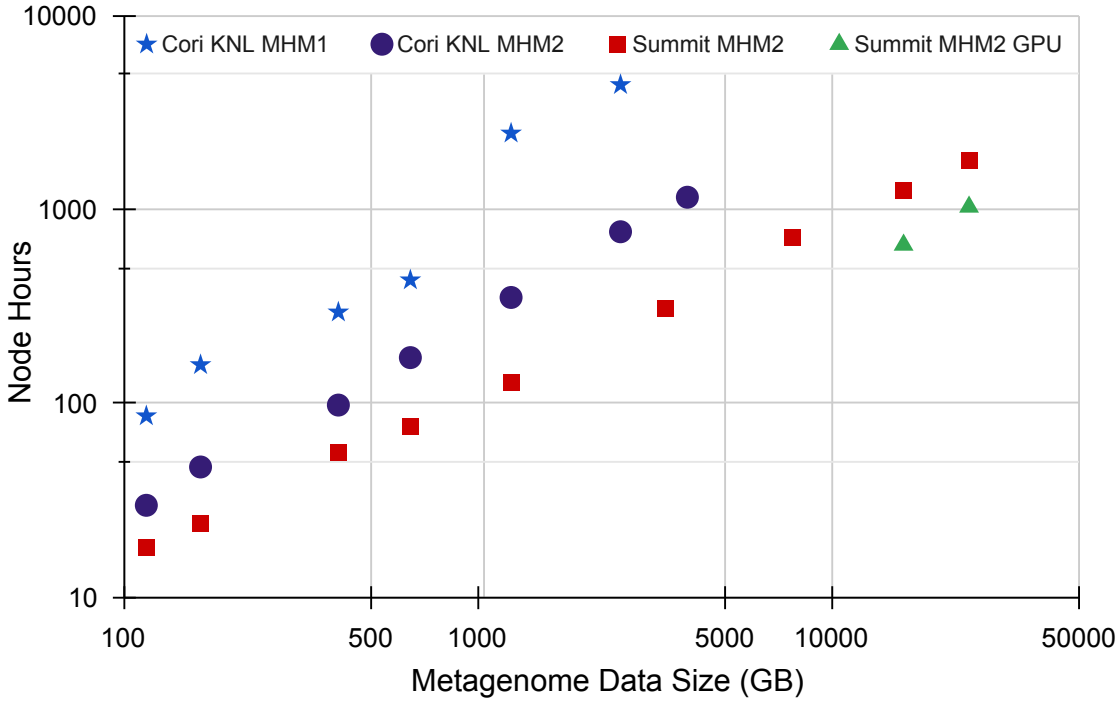
Co-Assembly improves quality and is an HPC problem

Full wetlands data: 2.6 TB of data in 21 lanes (samples)

- Time-series samples from multiple sites of Twitchell Wetlands in the San Francisco Bay-Delta
- Previously assembled 1 lane at a time (multiassembly)
- MetaHipMer coassembled together – higher quality assembly, in **3.5 hours on 16K cores**

Multiassembly
1 lane at a time

Coassembly all assembled together – more new genomes at higher completeness



This was the largest, high-quality de novo metagenome assembly completed at the time
More recently: new record 30TB metagenome assembly on 1500 nodes (63K cores and 9K GPUs) of OLCF Summit in 2022

Evangelos Georganas, Rob Egan, Steven Hofmeyr, Eugene Goltsman, Bill Arndt, Andrew Tritt, Aydın Buluc, Leonid Oliker, Katherine Yelick, **SC18 best paper finalist**

MetaHipMer utilized UPC++ features

C++ templates – efficient code reuse

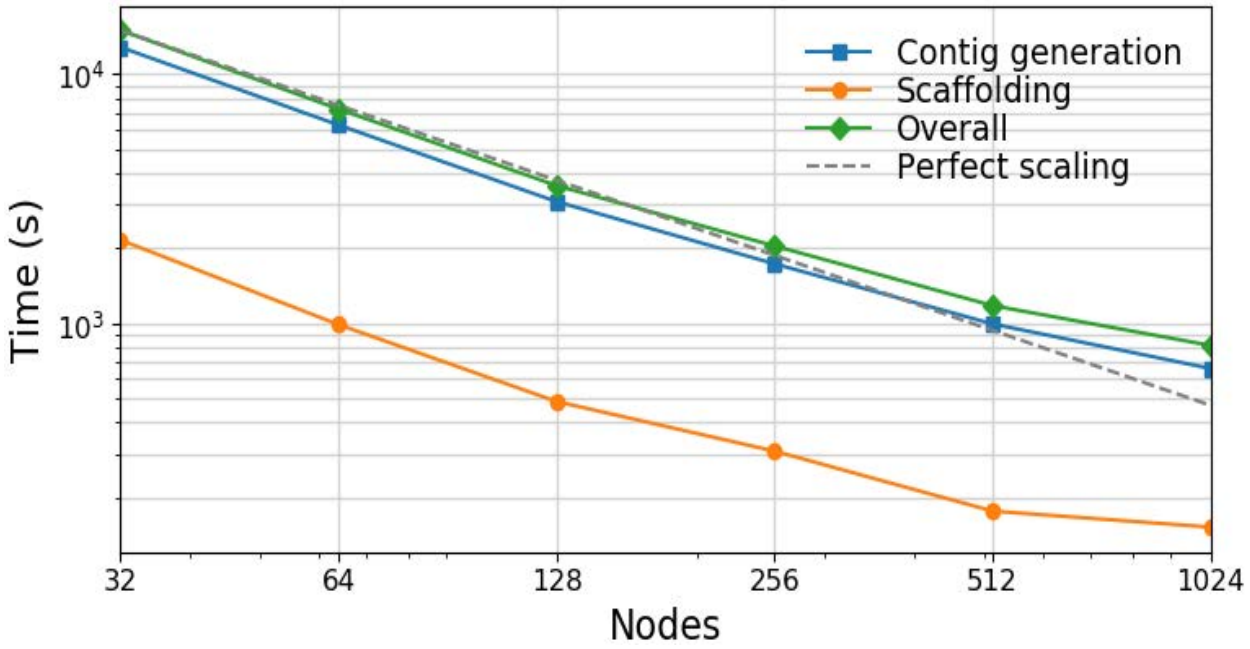
`dist_object` – as a templated functor & data store

Asynchronous all-to-all exchange – not batch synchronous

- 5x improvement at scale relative to previous MPI implementation

Future-chained workflow

- Multi-level RPC messages
- Send by node, then by process



Promise & fulfill (advanced UPC++ feature) – for a fixed-size memory footprint

- Issue promise when full, fulfill when available

Work and results by Rob Egan, funded by ECP ExaBiome Group

https://sites.google.com/lbl.gov/exabiome/downloads

UPC++ additional resources

Website: **upcxx.lbl.gov** includes the following content:

- Open-source/free library implementation
  - Portable from laptops to supercomputers
- Tutorial resources at **upcxx.lbl.gov/training**
  - UPC++ Programmer's Guide
  - Videos and exercises from past tutorials
- Formal UPC++ specification
  - All the semantic details about all the features
- Links to various UPC++ publications
- Links to optional extensions and partner projects
- Contact information and support forum

"We found UPC++ to be a very powerful and flexible tool for the development of parallel applications in distributed memory environments that enabled us to reach the high level of performance required by our DepSpawn project, so that we could outperform the state-of-the-art approaches. It is also particularly important in our opinion that, while supporting a really wide range of mechanisms, it is very well documented and supported."
-- Basilio Bernardo Fraguela Rodríguez, Universidade da Coruña, Spain

"If your code is already written in a one-sided fashion, moving from MPI RMA or SHMEM to UPC++ RMA is quite straightforward and intuitive; it took me about 30 minutes to convert MPI RMA functions in my application to UPC++ RMA, and I am getting similar performance to MPI RMA at scale."
-- Sayan Ghosh, PNNL

BERKELEY LAB
Bringing Science Solutions to the World

U.S. DEPARTMENT OF ENERGY
Office of Science

Coarray Fortran Tutorial

Damian Rouson
Computer Languages & System Software

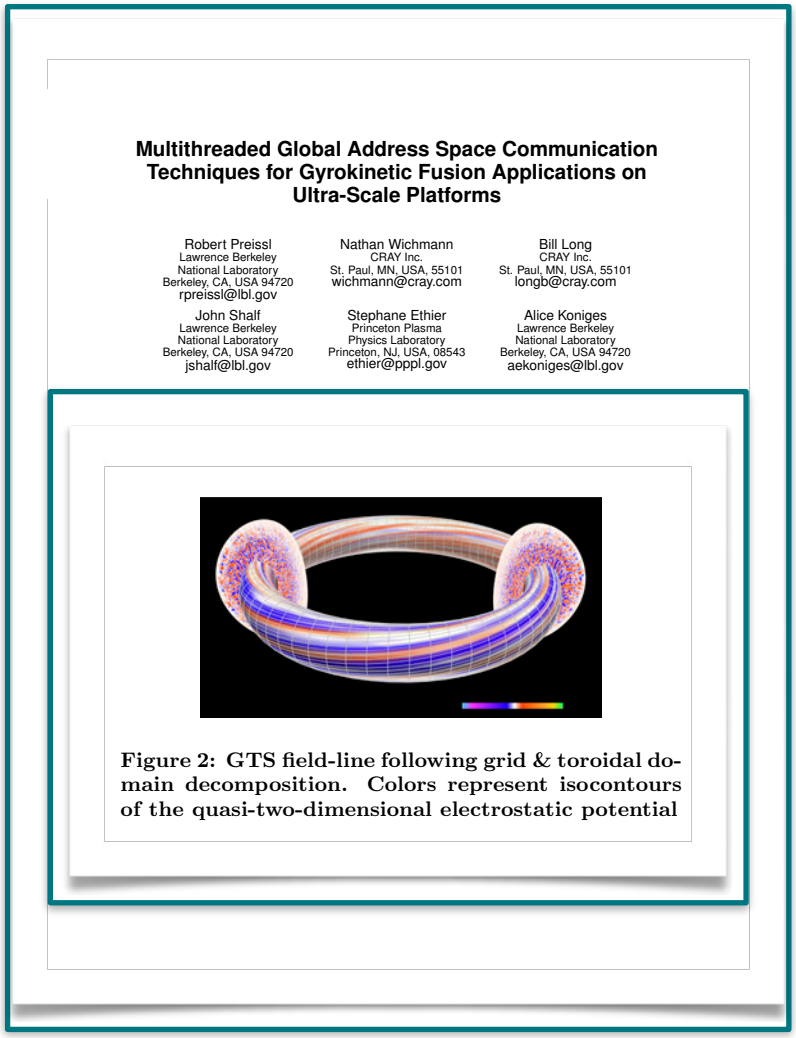
Hosted by ECP, NERSC, and OLCF, 26-27 July 2023

Day 2

- CAF at Scale
- Teams
- Image enumeration
- Synchronization
- Collective Subroutines
- Coarrays
- Events

CAF at Scale: Magnetic Fusion

BERKELEY LAB
Bringing Science Solutions to the World



Multithreaded Global Address Space Communication Techniques for Gyrokinetic Fusion Applications on Ultra-Scale Platforms

Robert Preissl
Lawrence Berkeley National Laboratory
Berkeley, CA, USA 94720
rpreissl@lbl.gov

Nathan Wichmann
CRAY Inc.
St. Paul, MN, USA, 55101
wichmann@cray.com

Bill Long
CRAY Inc.
St. Paul, MN, USA, 55101
longb@cray.com

John Shalf
Lawrence Berkeley National Laboratory
Berkeley, CA, USA 94720
jshalf@lbl.gov

Stephane Ethier
Princeton Plasma Physics Laboratory
Princeton, NJ, USA, 08543
ethier@pppl.gov

Alice Koniges
Lawrence Berkeley National Laboratory
Berkeley, CA, USA 94720
aekoniges@lbl.gov

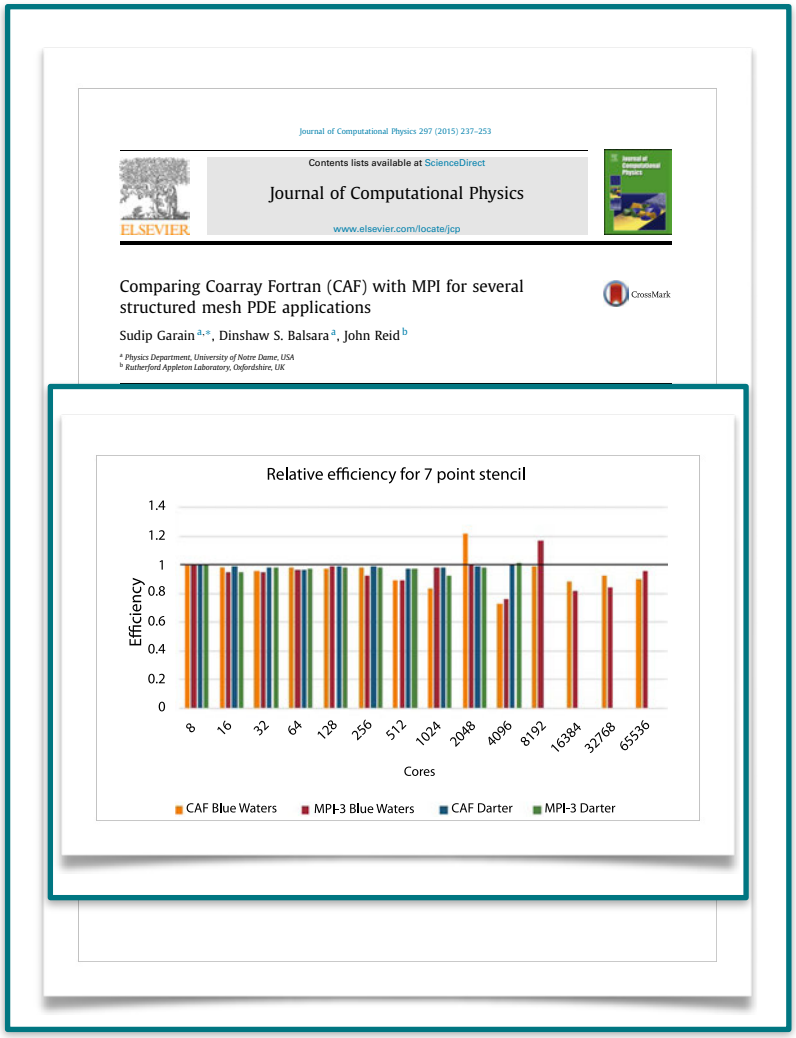
Figure 2: GTS field-line following grid & toroidal domain decomposition. Colors represent isocontours of the quasi-two-dimensional electrostatic potential

- Application focus:
  - The shift phase of charged particles in a tokamak simulation code
- Programming models studied:
  - CAF + OpenMP or
  - Two-sided MPI + OpenMP
- Highlights:
  - Experiments on up to 130,560 processors
  - 58% speed-up of the CAF implementation over the best multithreaded MPI shifter algorithm on largest scale
  - “the complexity required to implement … MPI-2 one-sided, in addition to several other semantic limitations, is prohibitive.”

Preissl, R., Wichmann, N., Long, B., Shalf, J., Ethier, S., & Koniges, A. (2011, November). Multithreaded global address space communication techniques for gyrokinetic fusion applications on ultra-scale platforms. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (pp. 1-11).

CAF at Scale: CFD, FFTs, Multigrid

BERKELEY LAB
Bringing Science Solutions to the World



Garain, S., Balsara, D. S., & Reid, J. (2015). Comparing Coarray Fortran (CAF) with MPI for several structured mesh PDE applications. *Journal of Computational Physics*, *297*, 237-253.

Applications studied:

- Magnetohydrodynamics (MHD)
- 3D Fast Fourier Transforms (FFTs) used in infinite-order accurate spectral methods
- Multigrid methods with point-wise smoothers requiring fine-grained messaging

Programming models studied:

- CAF or
- One-sided MPI-3

Highlights:

- Simulations on up to 65,536 cores
- "… CAF either draws level with MPI-3 or shows a slight advantage over MPI-3."
- "CAF and MPI-3 are shown to provide substantial advantages over MPI-2.
- "CAF code is of course much easier to write and maintain…"

CAF at Scale: Weather

BERKELEY LAB
Bringing Science Solutions to the World

Article

International Journal of HIGH PERFORMANCE COMPUTING APPLICATIONS

A Partitioned Global Address Space implementation of the European Centre for Medium Range Weather Forecasts Integrated Forecasting System

George Mozdzynski, Mats Hamrud and Nils Wedi

The International Journal of High Performance Computing Applications
2015, Vol. 29(3) 261–273
© The Author(s) 2015
Reprints and permissions:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/1094342015576773
hpc.sagepub.com
SAGE

Figure 7. EQ_REGIONS partitioning of grid-point space, showing a partition at the poles and then an increasing number of partitions as we approach the equator.

- Application:
  - — European Centre for Medium Range Weather Forecasts (ECMWF) operational weather forecast model
- Programming models studied:
  - — CAF or
  - — Two-sided MPI
- Highlights:
  - — Simulations on > 60K cores
  - — performance improvement from switching to CAF peaks at 21% around 40K cores

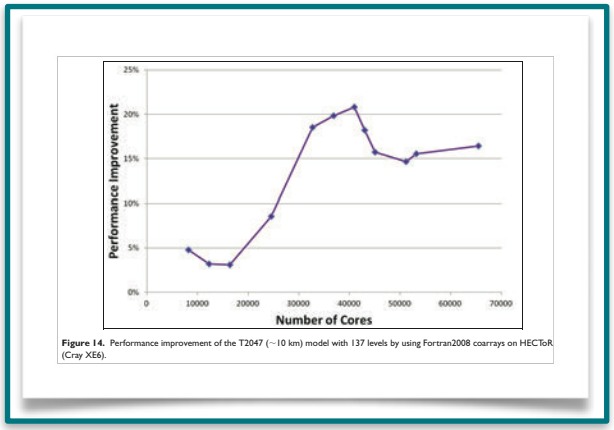


Figure 14. Performance improvement of the T2047 (~10 km) model with 137 levels by using Fortran2008 coarrays on HECToR (Cray XE6).

Mozdzynski, G., Hamrud, M., & Wedi, N. (2015). A partitioned global address space implementation of the European centre for medium range weather forecasts integrated forecasting system. *The International Journal of High Performance Computing Applications*, *29*(3), 261-273.

CAF at Scale: Climate

BERKELEY LAB
Bringing Science Solutions to the World

Development and performance comparison of MPI and Fortran Coarrays within an atmospheric research model

Extended Abstract

Soren Rasmussen[1], Ethan D Gutmann[2], Brian Friesen[3], Damian Rouson[4], Salvatore Filippone [1], Irene Moulitsas [1]
[1]Cranfield University, UK
[2]National Center for Atmospheric Research, USA
[3]Lawrence Berkeley National Laboratory, USA
[4]Sourcery Institute, USA

ABSTRACT

A mini-application of The Intermediate Complexity Research (ICAR) Model offers an opportunity to compare the costs and performance of the Message Passing Interface (MPI) versus coarray Fortran, two methods of communication across processes. The application requires repeated communication of halo regions, which is performed with either MPI or coarrays. The MPI communication is done using non-blocking two-sided communication, while the coarray library is implemented using a one-sided MPI or OpenSHMEM communication backend. We examine the development cost in addition to strong and weak scalability analysis to understand the performance costs.

1 INTRODUCTION

1.1 Motivation and Background

In high performance computing MPI has been the de facto method for memory communication across a system's nodes for many years. MPI 1.0 was released in 1994 and research and development has continued across academia and industry. A method in Fortran 2008, known as coarray Fortran, was introduced to express the communication within the language [5]. This work was based on an extension to Fortran that was introduced by Robert W. Numrich and John Reid in 1998 [7]. Coarray Fortran, like MPI, is a single-program, multiple-data (SPMD) programming technique. Coarray Fortran's single program is replicated across multiple processes, which are called "images". Unlike MPI, it is based on the Partitioned

(c) 400 points per process (d) Cray weak scaling

Figure 3: (a-c) Weak scaling results for 25, 100, and 400 points per process (d) weak scaling for Cray.

Application:

- Intermediate Complexity Atmospheric Research (ICAR) model
- Regional impacts of global climate change

Programming models studied:

- CAF over one-sided MPI
- CAF over OpenSHMEM
- Two-sided MPI
- Cray CAF

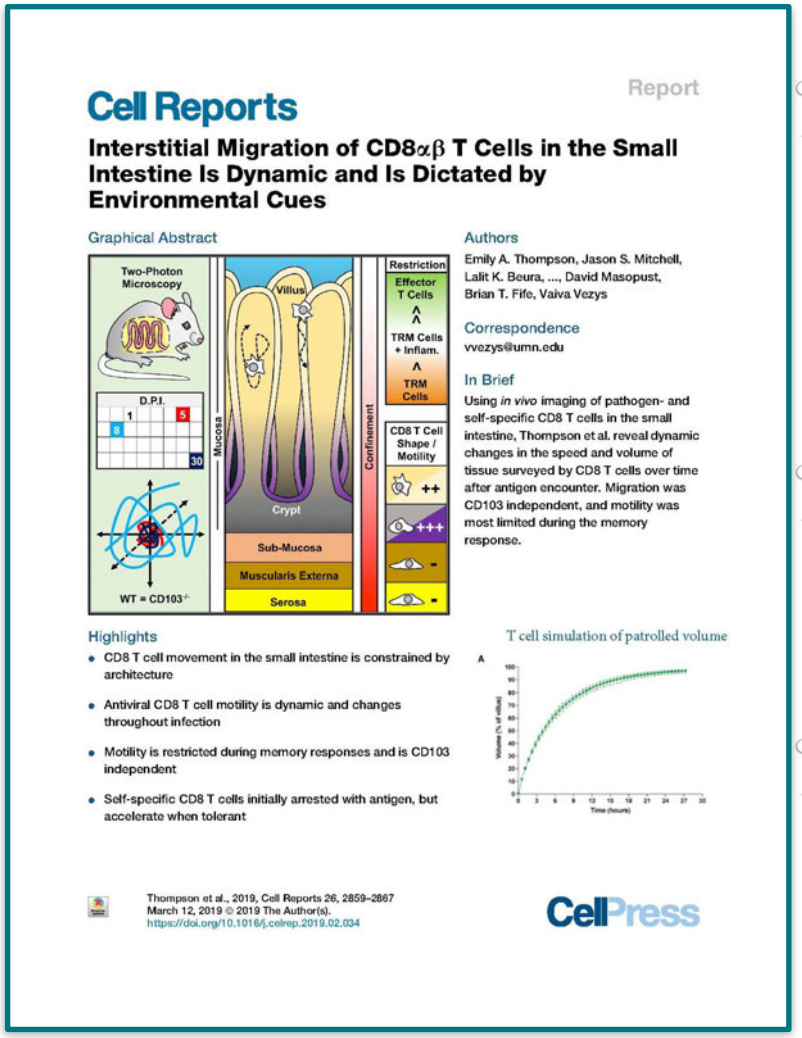
Highlights:

- "… we used up to 25,600 processes and found that at every data point OpenSHMEM was outperforming MPI."
- "The coarray Fortran with MPI backend stopped being usable as we went over 2,000 processes… the initialization time started to increase exponentially."

Rasmussen, S., Gutmann, E. D., Friesen, B., Rouson, D., Filippone, S., & Moulitsas, I. (2018). Development and performance comparison of MPI and Fortran Coarrays within an atmospheric research model. *Parallel Applications Workshop - Alternatives to MPI+x (PAW-ATM)*, Dallas, Texas, USA.

New Frontiers: T-Cell Motility

BERKELEY LAB
Bringing Science Solutions to the World



Report

Cell Reports

Interstitial Migration of CD8αβ T Cells in the Small Intestine Is Dynamic and Is Dictated by Environmental Cues

Graphical Abstract

Authors

Emily A. Thompson, Jason S. Mitchell, Lalit K. Beura, ..., David Masopust, Brian T. Fife, Vaiva Vezys

Correspondence

vvezys@umn.edu

In Brief

Using *in vivo* imaging of pathogen- and self-specific CD8 T cells in the small intestine, Thompson et al. reveal dynamic changes in the speed and volume of tissue surveyed by CD8 T cells over time after antigen encounter. Migration was CD103 independent, and motility was most limited during the memory response.

Highlights

- CD8 T cell movement in the small intestine is constrained by architecture
- Antiviral CD8 T cell motility is dynamic and changes throughout infection
- Motility is restricted during memory responses and is CD103 independent
- Self-specific CD8 T cells initially arrested with antigen, but accelerate when tolerant

Thompson et al., 2019, Cell Reports 26, 2859–2867
March 12, 2019 © 2019 The Author(s).
https://doi.org/10.1016/j.celrep.2019.02.034

CellPress

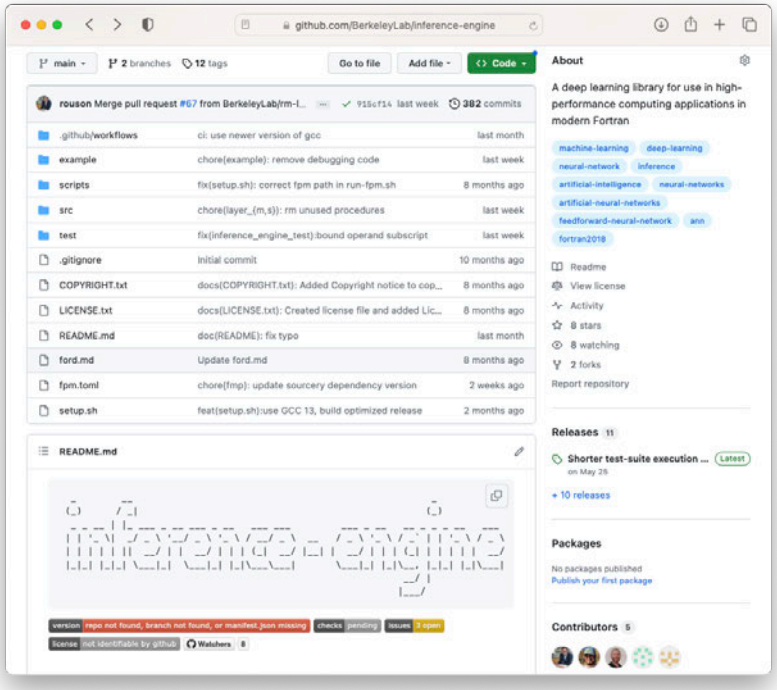
Thompson, E. A., Mitchell, J. S., Beura, L. K., Torres, D. J., Mrass, P., Pierson, M. J., ... & Vezys, V. (2019). Interstitial migration of CD8αβ T cells in the small intestine is dynamic and is dictated by environmental cues. *Cell reports*, *26*(11), 2859-2867.

- Application:
  - — Matcha: Motility Analysis of T Cells in Activation
  - — Matching the speed & turning angle distributions to observed T cells, simulations can explore large spatial volumes and parameter spaces.
- Programming models:
  - — Coarray halo exchanges in a 3D diffusion PDE solver.
  - — `Do concurrent` for automatic GPU offloading
- Highlights:
  - — This tutorial's 2D heat equation solver was the prototype for the 3D diffusion solver.

https://go.lbl.gov/matcha

New Frontiers: Deep Learning

BERKELEY LAB
Bringing Science Solutions to the World



https://go.lbl.gov/inference-engine

- Application:
  - Inference-Engine
  - *In situ* neural network training and large-batch inference for HPC applications
- Language-based parallel & GPU programming:
  - Extensive use of array statements, `elemental` procedures, `do concurrent`
  - Functional programming pattern:

    Every procedure is `pure` except those that create and consume JSON file objects.
  - *Coming soon:*

    Parallel mini-batch training via `co_sum`

Implicitly Parallel Training

inference-engine — vim src/inference_engine/trainable_engine_s.f90 — 132×55

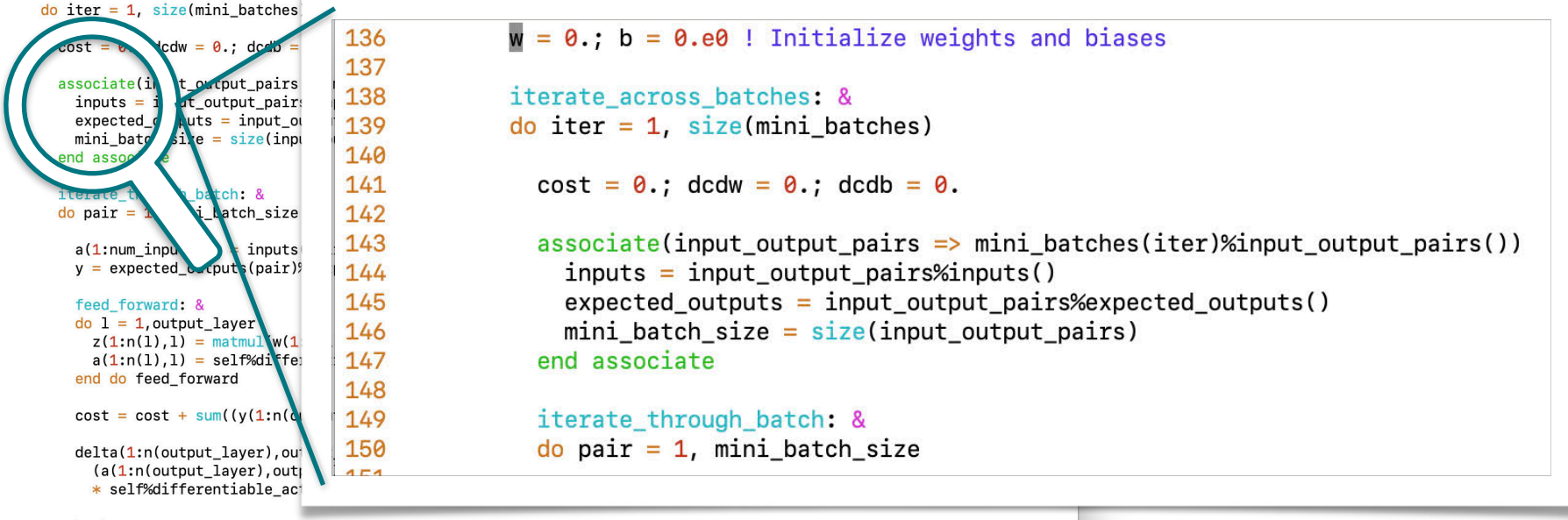
```
136       w = 0.; b = 0.e0 ! Initialize weights and biases
137
138       iterate_across_batches: &
139       do iter = 1, size(mini_batches)
140
141         cost = 0.; dcdw = 0.; dcdb = 0.
142
143         associate(input_output_pairs => mini_batches(iter)%input_output_pairs())
144           inputs = input_output_pairs%inputs()
145           expected_outputs = input_output_pairs%expected_outputs()
146           mini_batch_size = size(input_output_pairs)
147         end associate
148
149         iterate_through_batch: &
150         do pair = 1, mini_batch_size
151
152           a(1:num_inputs,0) = inputs(pair)%values()
153           y = expected_outputs(pair)%outputs()
154
155           feed_forward: &
156           do l = 1,output_layer
157             z(1:n(l),l) = matmul(w(1:n(l),1:n(l-1),l), a(1:n(l-1),l-1)) + b(1:n(l),l)
158             a(1:n(l),l) = self%differentiable_activation_strategy_%activation(z(1:n(l),l))
159           end do feed_forward
160
161           cost = cost + sum((y(1:n(output_layer))-a(1:n(output_layer),output_layer))**2)/(2.e0*mini_batch_size)
162
163           delta(1:n(output_layer),output_layer) = &
164             (a(1:n(output_layer),output_layer) - y(1:n(output_layer))) &
165             * self%differentiable_activation_strategy_%activation_derivative(z(1:n(output_layer),output_layer))
166
167           back_propagate_error: &
168           do l = n_hidden,1,-1
169             delta(1:n(l),l) = matmul(transpose(w(1:n(l+1),1:n(l),l+1)), delta(1:n(l+1),l+1))
170             delta(1:n(l),l) = delta(1:n(l),l) * self%differentiable_activation_strategy_%activation_derivative(z(1:n(l),l))
171           end do back_propagate_error
172
173           sum_gradients: &
174           do l = 1,output_layer
175             dcdb(1:n(l),l) = dcdb(1:n(l),l) + delta(1:n(l),l)
176             do concurrent(j = 1:n(l))
177               dcdw(j,1:n(l-1),l) = dcdw(j,1:n(l-1),l) + a(1:n(l-1),l-1)*delta(j,l)
178             end do
179           end do sum_gradients
180         end do iterate_through_batch
181
182         adjust_weights_and_biases: &
183         do l = 1,output_layer
184           dcdb(1:n(l),l) = dcdb(1:n(l),l)/mini_batch_size
185           b(1:n(l),l) = b(1:n(l),l) - eta*dcdb(1:n(l),l) ! Adjust biases
186           dcdw(1:n(l),1:n(l-1),l) = dcdw(1:n(l),1:n(l-1),l)/mini_batch_size
187           w(1:n(l),1:n(l-1),l) = w(1:n(l),1:n(l-1),l) - eta*dcdw(1:n(l),1:n(l-1),l) ! Adjust weights
188         end do adjust_weights_and_biases
189       end do iterate_across_batches
                                                                    136,9         72%
```

“Loop” Structure

BERKELEY LAB
Bringing Science Solutions to the World

inference-engine — vim src/inference_engine/trainable_engine_s.f90 — 132×55

```
136       w = 0.; b = 0.e0 ! Initialize weights and biases
137
138       iterate_across_batches: &
139       do iter = 1, size(mini_batches
140
141         cost = 0.; dcdw = 0.; dcdb =
142
143         associate(input_output_pairs
144           inputs = input_output_pairs
145           expected_outputs = input_ou
146           mini_batch_size = size(inpu
147         end associate
148
149         iterate_through_batch: &
150         do pair = 1, mini_batch_size
151
152           a(1:num_inputs) = inputs
153           y = expected_outputs(pair)
154
155           feed_forward: &
156           do l = 1,output_layer
157             z(1:n(l),l) = matmul(w(1
158             a(1:n(l),l) = self%diffe
159           end do feed_forward
160
161           cost = cost + sum((y(1:n(
162
163           delta(1:n(output_layer),ou
164             (a(1:n(output_layer),outp
165             * self%differentiable_ac
166
167           back_propagate_error: &
168           do l = n_hidden,1,-1
169             delta(1:n(l),l) = matmul(transpose(
170             delta(1:n(l),l) = delta(1:n(l),l) *
171           end do back_propagate_error
172
173           sum_gradients: &
174           do l = 1,output_layer
175             dcdb(1:n(l),l) = dcdb(1:n(l),l) + d
176             do concurrent(j = 1:n(l))
177               dcdw(j,1:n(l-1),l) = dcdw(j,1:n(l
178             end do
179           end do sum_gradients
180         end do iterate_through_batch
181
182         adjust_weights_and_biases: &
183         do l = 1,output_layer
184           dcdb(1:n(l),l) = dcdb(1:n(l),l)/mini_batch_size
185           b(1:n(l),l) = b(1:n(l),l) - eta*dcdb(1:n(l),l) ! Adjust biases
186           dcdw(1:n(l),1:n(l-1),l) = dcdw(1:n(l),1:n(l-1),l)/mini_batch_size
187           w(1:n(l),1:n(l-1),l) = w(1:n(l),1:n(l-1),l) - eta*dcdw(1:n(l),1:n(l-1),l) ! Adjust weights
188         end do adjust_weights_and_biases
189       end do iterate_across_batches
                                                                    136,9         72%
```



Iterating sequentially across and within mini-batches of input/output pairs facilitates *in situ* training at application runtime, potentially eliminating the export of large training data sets or at least making it so that the resulting network can be trained off-line in fewer iterations.

“Loop” Structure

BERKELEY LAB
Bringing Science Solutions to the World

inference-engine — vim src/inference_engine/trainable_engine_s.f90 — 132×55

```
136       w = 0.; b = 0.e0 ! Initialize weights and biases
137
138       iterate_across_batches: &
139       do iter = 1, size(mini_batches)
140
141         cost = 0.; dcdw = 0.; dcdb = 0.
142
143         associate(input_output_pairs => mini_batches(iter)%input_output_pairs())
144           inputs = input_output_pairs%inputs()
145           expected_outputs = input_output_pairs%expected_outputs()
146           mini_batch_size = size(input_output_pairs)
147         end associate
148
149         iterate_through_batch: &
150         do pair = 1, mini_batch_size
151
152           a(1:num_inputs,0) = inputs(pair)%values()
153           y = expected_outputs(pair)%outputs()
154
155           feed_forward: &
156           do l = 1,output_layer
157             z(1:n(l),l) = matmul(w(1:n(l),1:n(l-1),l), a(1:n(l-1),l-1)) + b(1:n(l),l)
158             a(1:n(l),l) = self%differentiable_activation_strategy_%activation(z(1:n(l),l))
159
160
161                                                                    **2)/(2.e0*m
162
163
164
165                                                                    utput_layer)
166
167
168
169           delta(1:n(l),l) = matmul(
170           delta(1:n(l),l) = delta(
171
172
173           sum_grad      ts: &
174           do l = 1,     put_layer
175             dcdb(1:n    ),l) = dcdb(1:
176             do concu      ent(j = 1:n(l)
177               dcdw(     :n(l-1),l) = c
178             end do
179           end do      gradients
180
181
182         adjust_weigh      d_biases: &
183         do l = 1,outpu      er
184           dcdb(1:n(l),1     dcdb(1:n(
185           b(1:n(l),l) = b(1:n(l),l)
186           dcdw(1:n(l),1:n(l-1),l) =
187           w(1:n(l),1:n(l-1),l) = w(1:n(l),1:n(l-1),l) - eta*dcdw(1:n(l),1:n(l-1),l) ! Adjust weights
188         end do adjust_weights_and_biases
189       end do iterate_across_batches
                                                                    136,9         72%
```

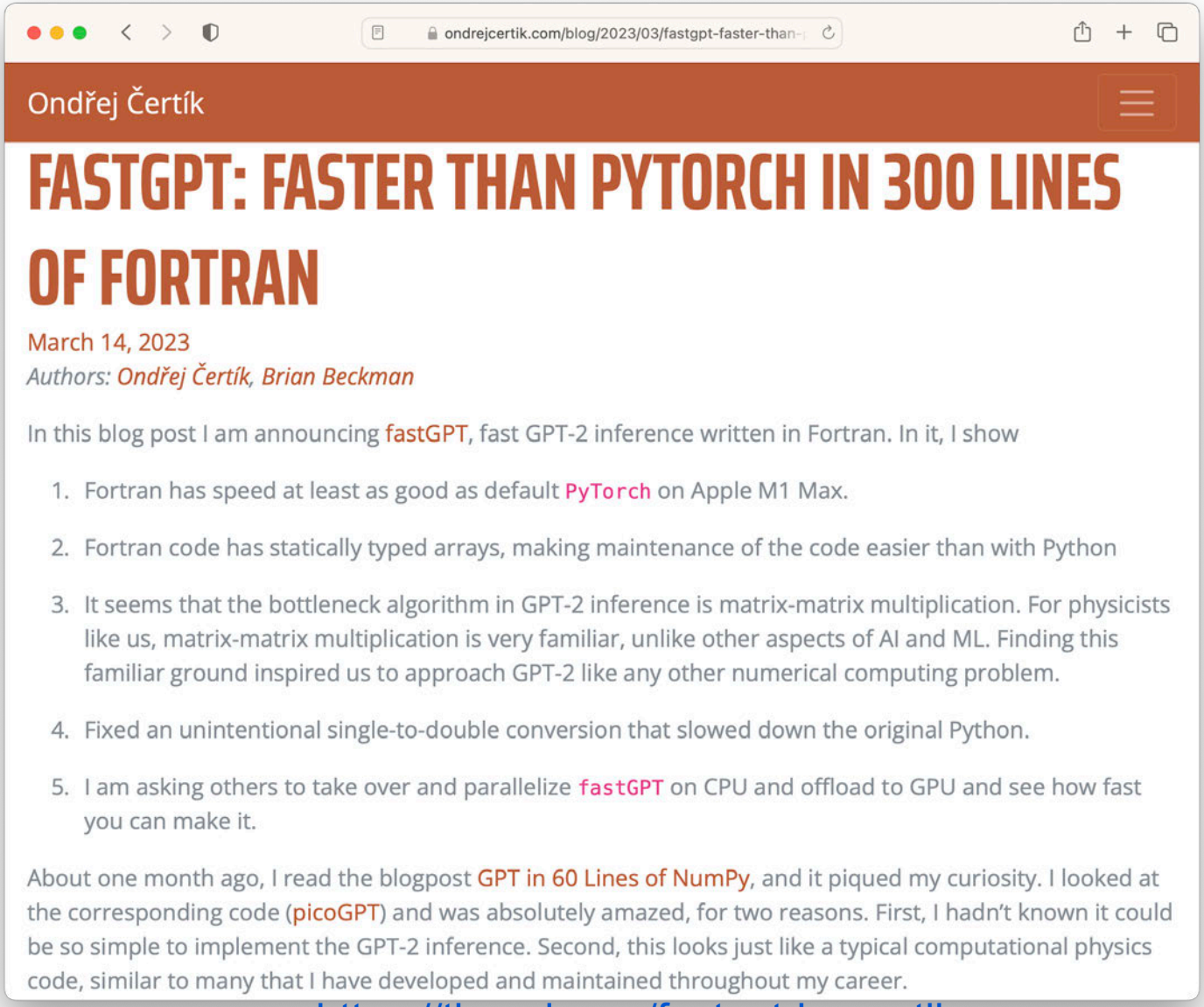
The only other sequential logic is the (mostly) necessary stepping through layers:

All other logic is implicitly parallel array statements or `do concurrent` blocks:

```
173           sum_gradients: &
174           do l = 1,output_layer
175             dcdb(1:n(l),l) = dcdb(1:n(l),l) + delta(1:n(l),l)
176             do concurrent(j = 1:n(l))
177               dcdw(j,1:n(l-1),l) = dcdw(j,1:n(l-1),l) + a(1:n(l-1),l-1)*delta(j,l)
178             end do
179           end do sum_gradients
180         end do iterate_through_batch
```

Fast-GPT

BERKELEY LAB
Bringing Science Solutions to the World



https://tinyurl.com/fastgpt-by-certik

Teams

BERKELEY LAB
Bringing Science Solutions to the World

An ordered set of images created by execution of a **`form team`** statement, or the initial ordered set of all images.

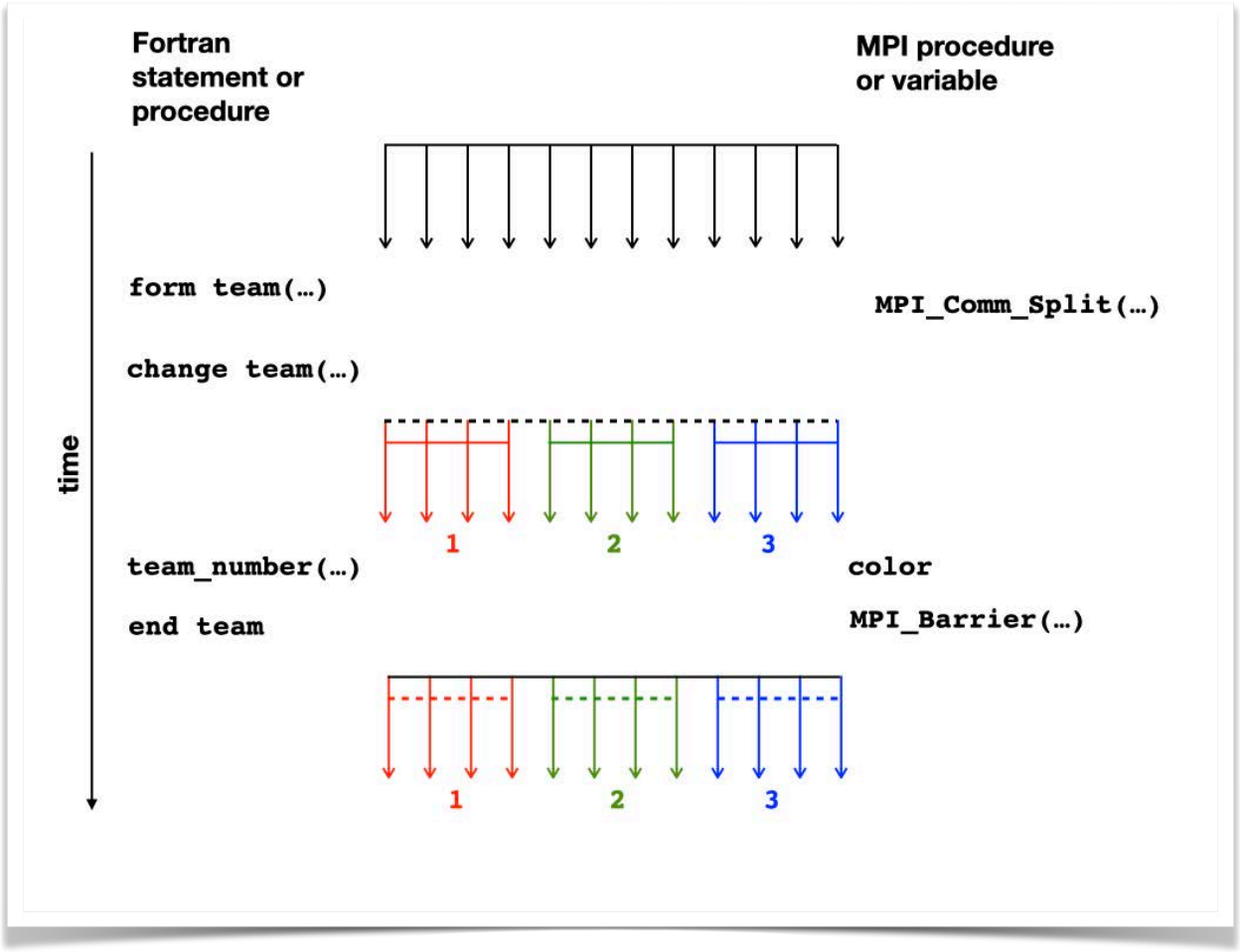
Teams facilitate the execution of an image sets independently from other image sets, e.g., a **`sync all`** statement synchronizes the current team only.

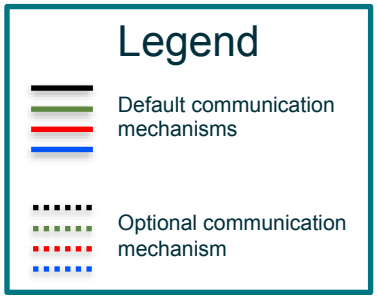
An extensible derived type `team_type` with private components describes a team after the successful execution of a **`form team`** statement.

CAF/MPI Rosetta Stone

BERKELEY LAB
Bringing Science Solutions to the World

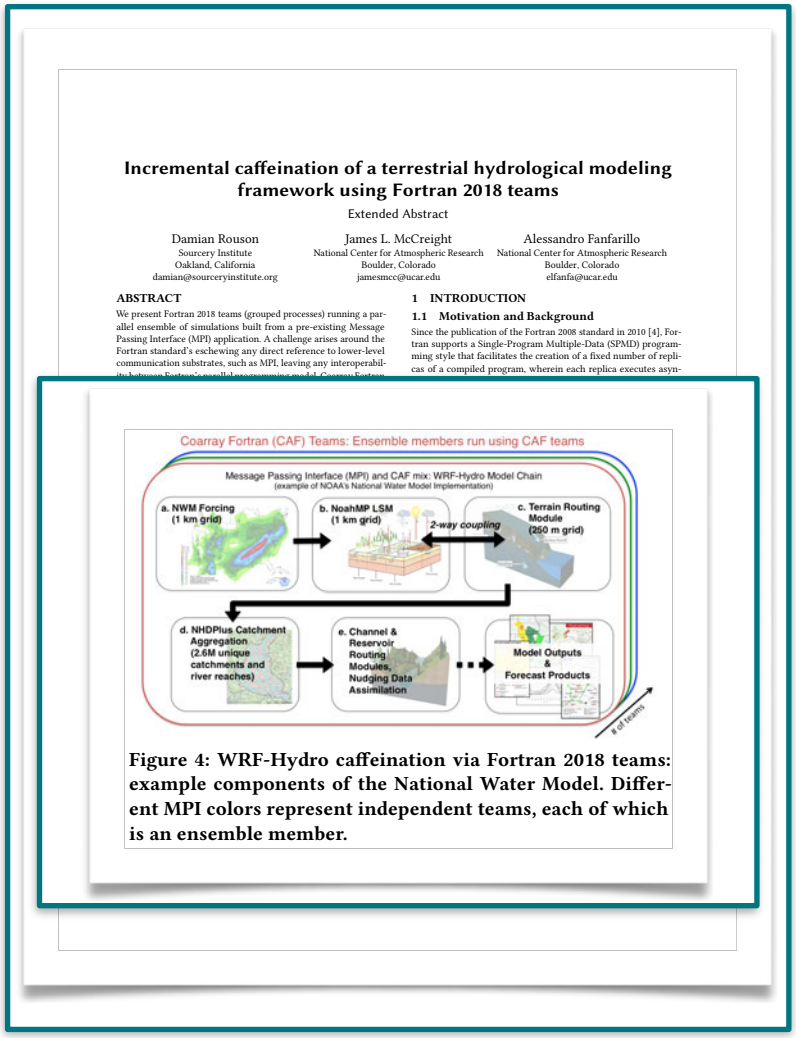
Program execution sequence over time (left axis) in 12 images (top) initially globally and then within subgroups.





Teams Test Code

BERKELEY LAB
Bringing Science Solutions to the World



Incremental caffeination of a terrestrial hydrological modeling framework using Fortran 2018 teams

Extended Abstract

Damian Rouson
Sourcery Institute
Oakland, California
damian@sourceryinstitute.org

James L. McCreight
National Center for Atmospheric Research
Boulder, Colorado
jamesmcc@ucar.edu

Alessandro Fanfarillo
National Center for Atmospheric Research
Boulder, Colorado
elfanfa@ucar.edu

ABSTRACT

We present Fortran 2018 teams (grouped processes) running a parallel ensemble of simulations built from a pre-existing Message Passing Interface (MPI) application. A challenge arises around the Fortran standard's eschewing any direct reference to lower-level communication substrates, such as MPI, leaving any interoperability between Fortran's parallel programming model, Coarray Fortran

1 INTRODUCTION

1.1 Motivation and Background

Since the publication of the Fortran 2008 standard in 2010 [4], Fortran supports a Single-Program Multiple-Data (SPMD) programming style that facilitates the creation of a fixed number of replicas of a compiled program, wherein each replica executes asyn-

Figure 4: WRF-Hydro caffeination via Fortran 2018 teams: example components of the National Water Model. Different MPI colors represent independent teams, each of which is an ensemble member.

```
 1 program main
 2   !! Test team_number intrinsic function
 3   use iso_fortran_env, only : team_type
 4   use assertions_module , only : assertions
 5
 6   implicit none
 7
 8   integer , parameter :: standard_initial_value = -1
 9   type(team_type), target :: home
10
11   call assert(team_number() == standard_initial_value)
12
13   associate(my_team=>mod(this_image(),2) + 1)
14
15     form team(my_team,home) ! Map even|odd images->teams 1|2
16     change team(home)
17       call assert(team_number() == my_team)
18     end team
19
20     call assert(team_number() == standard_initial_value)
21
22   end associate
23
24   sync all
25
26   if (this_image() == 1) print *, "Test passed."
27
28 end program
```

Rouson, D., McCreight, J. L., & Fanfarillo, A. (2017, November). Incremental caffeination of a terrestrial hydrological modeling framework using Fortran 2018 teams. In *Proceedings of the Second Annual PGAS Applications Workshop* (pp. 1-4).

Image Enumeration

BERKELEY LAB
Bringing Science Solutions to the World

- Obtaining an image index:

```
this_image([team])
this_image(coarray [,team])
this_image(coarray, dim [,team])
```

```
image_index(coarray, sub, team_number)
image_index(coarray, sub, team)
image_index(coarray, sub)
```

- Obtaining an image count:

```
num_images()
num_images(team)
num_images(team_number)
```

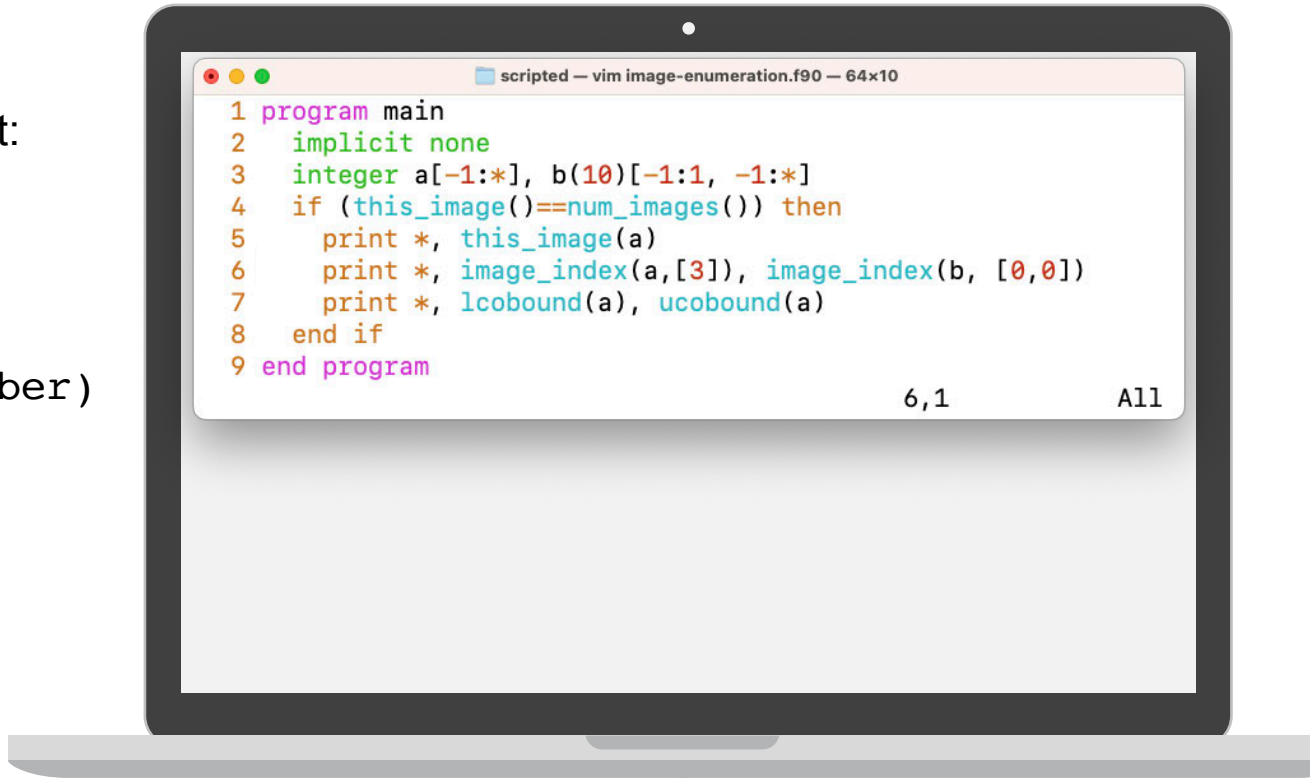


Image Enumeration

BERKELEY LAB
Bringing Science Solutions to the World

b(:)

[-1,1]	[0,1]	[1,1]
[-1,0]	[0,0]	[1,0]
[-1,-1]	[0,-1]	[1,-1]

a

[0]	[1]	[2]	[3]	[4]

scripted — vim image-enumeration.f90 — 64×10

```
1 program main
2   implicit none
3   integer a[-1:*], b(10)[-1:1, -1:*]
4   if (this_image()==num_images()) then
5     print *, this_image(a)
6     print *, image_index(a,[3]), image_index(b, [0,0])
7     print *, lcobound(a), ucobound(a)
8   end if
```

All

scripted — -zsh — 64×10

```
[cuf23-tutorial: cafrun -n 5 ./image-enumeration
           3
           5           5
          -1           3
[cuf23-tutorial: ^5^1
cafrun -n 1 ./image-enumeration
          -1
           0           0
          -1          -1
cuf23-tutorial:
```

Synchronization

BERKELEY LAB
Bringing Science Solutions to the World

- Image barriers ("meet-ups"):

```
sync all(stat, errmsg)
sync images(image-set, stat, errmsg)
allocate()
deallocate()
```

} for coarrays only, including implicit (de)allocation at end of a block or procedure

```
stop stop_code
```
(integer or character codes allowed)

```
end program
```

```
call move_alloc(from,to)
```
with coarray arguments.

Any statement causing an implicit coarray deallocation by completing a block or procedure.

- Deprecated by Metcalf, Reid & Cohen (2018):

```
sync memory(stat, errmsg)
```

Other Image Control Statements

BERKELEY LAB
Bringing Science Solutions to the World

- Locks:

```
lock(lock-variable, errmsg)
unlock(lock-variable, stat, errmsg)
```

A lock variable is a coarray object of the extensible intrinsic type `lock_type` with private components.

- Critical blocks:

```
critical(stat, errmsg)
end critical
```

- Teams

```
form team(team_number, team_variable)
change team(team_value, …)
end team
```

- Events

```
event post(event-variable, stat, errmsg)
event wait(event-variable, stat, errmsg)
```

An event variable is a coarray object of the extensible intrinsic type `event_type` with private components.

Collective Subroutines

BERKELEY LAB
Bringing Science Solutions to the World

- Behavior:
  - Successful execution of a collective subroutine performs a calculation on all the images of the current team and assigns a computed value on one or all of them.
  - If it is invoked by one image, it shall be invoked by the same statement on all active images of its current team in segments that are not ordered with respect to each other
  - Corresponding references participate in the same collective computation.
- Complete list:
  - co_sum(a, result_image, stat, errmsg)
  - co_max(a, result_image, stat, errmsg)
  - co_min(a, result_image, stat, errmsg)
  - co_broadcast(a, source_image, stat, errmsg)
  - co_reduce(a, operation, result_image, stat, errmsg)

co_sum

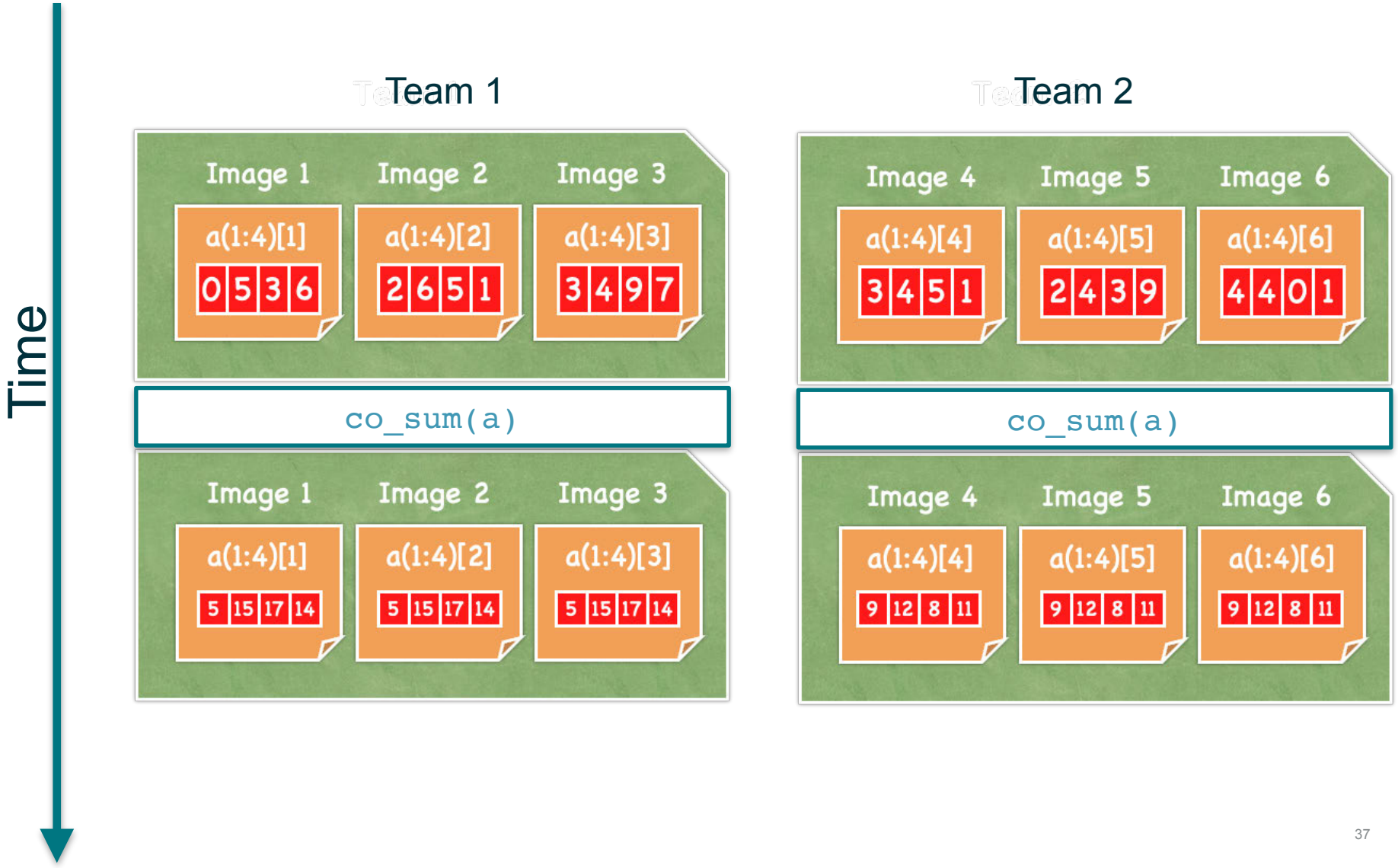
BERKELEY LAB
Bringing Science Solutions to the World

```
co_sum(a, result_image, stat, errmsg)
```

- Argument `a`
  - — shall be of numeric type,
  - — shall have the same shape, type, & type parameter values, in corresponding references.
  - — shall not be a coindexed object
  - — is an `intent(inout)` argument
- Argument `result_image` (optional)
  - — shall be of scalar type `integer`
  - — is an `intent(in)` argument
  - — If present, it shall be present on all images of the current team, have the same value on all images of the current team, and shall be an image index of the current team

co_sum

BERKELEY LAB
Bringing Science Solutions to the World



co_max

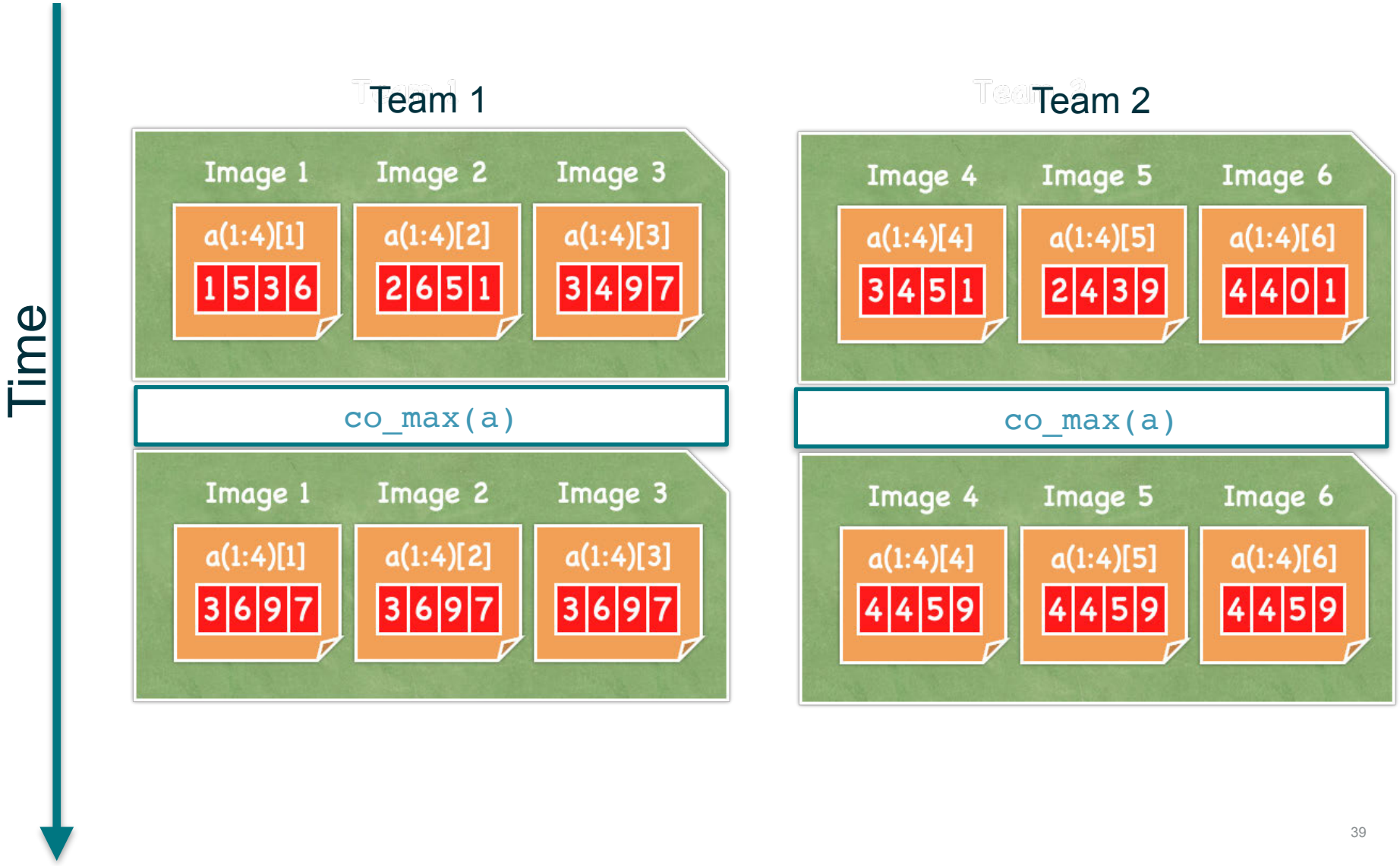
BERKELEY LAB
Bringing Science Solutions to the World

```
co_max(a, result_image, stat, errmsg)
```

- Argument `a`
  - — shall be of numeric type,
  - — shall have the same shape, type, & type parameter values, in corresponding references.
  - — shall not be a coindexed object
  - — is an `intent(inout)` argument
- Argument `result_image` (optional)
  - — shall be of scalar type `integer`
  - — is an `intent(in)` argument
  - — If present, it shall be present on all images of the current team, have the same value on all images of the current team, and shall be an image index of the current team

co_max

BERKELEY LAB
Bringing Science Solutions to the World



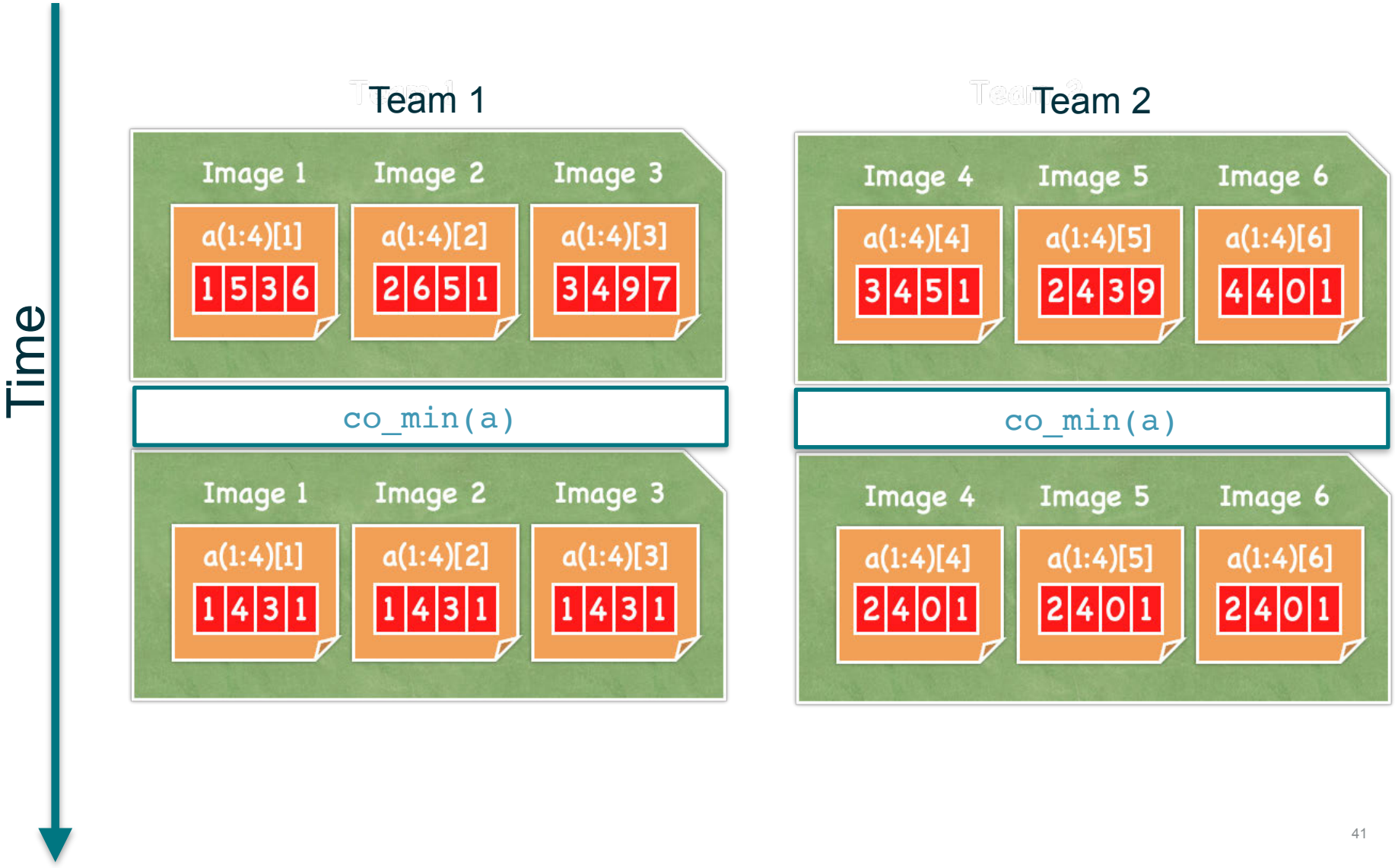
co_min

BERKELEY LAB
Bringing Science Solutions to the World

```
co_min(a, result_image, stat, errmsg)
```

- Argument `a`
  - — shall be of numeric type,
  - — shall have the same shape, type, & type parameter values, in corresponding references.
  - — shall not be a coindexed object
  - — is an `intent(inout)` argument
- Argument `result_image` (optional)
  - — shall be of scalar type `integer`
  - — is an `intent(in)` argument
  - — If present, it shall be present on all images of the current team, have the same value on all images of the current team, and shall be an image index of the current team

co_min


Time
Team 1
Team 2
Image 1
Image 2
Image 3
a(1:4)[1]
1536
a(1:4)[2]
2651
a(1:4)[3]
3497
Image 4
Image 5
Image 6
a(1:4)[4]
3451
a(1:4)[5]
2439
a(1:4)[6]
4401
co_min(a)
Image 1
Image 2
Image 3
a(1:4)[1]
1431
a(1:4)[2]
1431
a(1:4)[3]
1431
Image 4
Image 5
Image 6
a(1:4)[4]
2401
a(1:4)[5]
2401
a(1:4)[6]
2401

co_broadcast

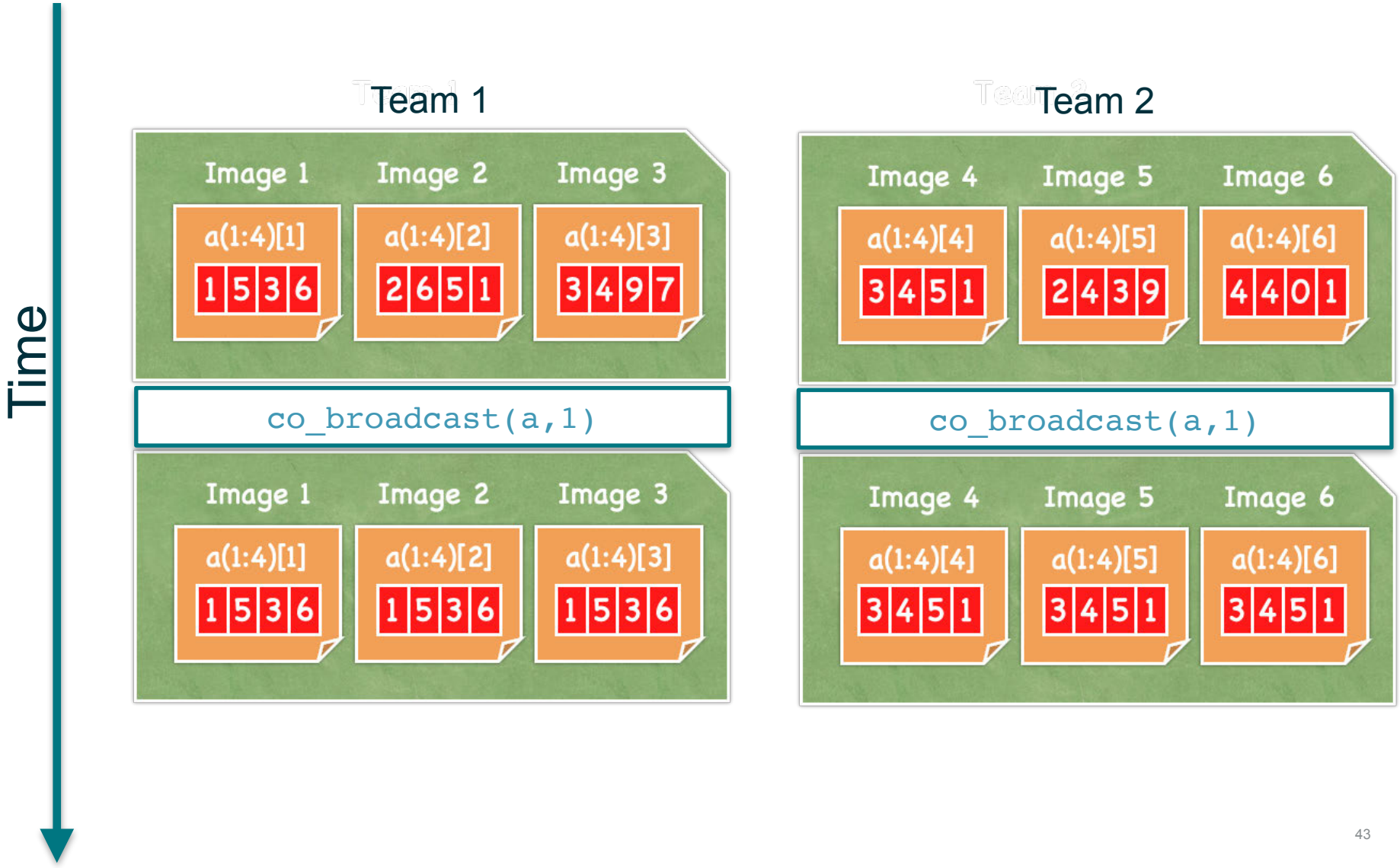
BERKELEY LAB
Bringing Science Solutions to the World

```
co_broadcast(a, source_image, stat, errmsg)
```

- Argument `a`
  - — shall have the same shape, dynamic type, & type parameter values, in corresponding references.
  - — shall not be a coindexed object
  - — is an `intent(inout)` argument
  - — successful execution causes `a` to become defined as if by intrinsic assignment on all images in the current team with the value of `a` on the source_image
- Argument `source_image`
  - — shall be of scalar type `integer`
  - — is an `intent(in)` argument
  - — If present, it shall be present on all images of the current team, have the same value on all images of the current team, and shall be an image index of the current team

co_broadcast

BERKELEY LAB
Bringing Science Solutions to the World



co_reduce

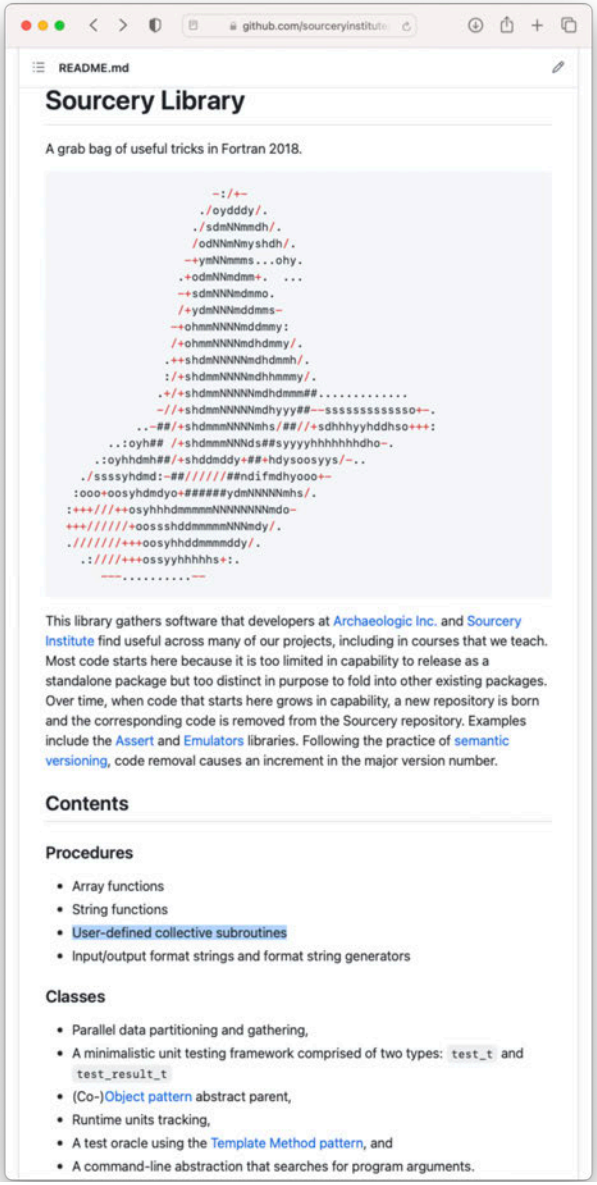
BERKELEY LAB
Bringing Science Solutions to the World

```
co_reduce(a, operation, result_image, stat, errmsg)
```

- Argument `a`
  - shall be `intent(inout)`, non-polymorphic and not coindexed
  - shall have the same shape, dynamic type, & type parameter values, in corresponding references.
  - becomes the result of applying the reduction operation to values of a in the corresponding references, and likewise on an element-wise basis if a is an array
- Argument operation
  - shall implement an associative operation via a `pure` function with two arguments
- Argument `result_image`
  - shall be of scalar `integer, intent(in) argument`
  - if present, it shall have the same value on all images of the current team and shall be an image index of the current team

Hands-on co_reduce

BERKELEY LAB
Bringing Science Solutions to the World



README.md

Sourcery Library

A grab bag of useful tricks in Fortran 2018.

This library gathers software that developers at Archaeologic Inc. and Sourcery Institute find useful across many of our projects, including in courses that we teach. Most code starts here because it is too limited in capability to release as a standalone package but too distinct in purpose to fold into other existing packages. Over time, when code that starts here grows in capability, a new repository is born and the corresponding code is removed from the Sourcery repository. Examples include the Assert and Emulators libraries. Following the practice of semantic versioning, code removal causes an increment in the major version number.

Contents

Procedures

- Array functions
- String functions
- User-defined collective subroutines
- Input/output format strings and format string generators

Classes

- Parallel data partitioning and gathering,
- A minimalistic unit testing framework comprised of two types: `test_t` and `test_result_t`
- (Co-)Object pattern abstract parent,
- Runtime units tracking,
- A test oracle using the Template Method pattern, and
- A command-line abstraction that searches for program arguments.

https://github.com/sourceryinstitute/sourcery

```
 1 module co_all_m
 2   implicit none
 3
 4   interface
 5     module subroutine co_all(a)
 6       implicit none
 7       logical, intent(inout) :: a
 8     end subroutine
 9   end interface
10
11 end module
12
13 submodule(co_all_m) co_all_s
14   implicit none
15 contains
16   module procedure co_all
17      call co_reduce(a, and)
18   contains
19      pure function and(lhs, rhs) result(lhs_and_rhs)
20        logical, intent(in) :: lhs, rhs
21        logical lhs_and_rhs
22        lhs_and_rhs = lhs .and. rhs
23      end function
24   end procedure
25 end submodule
26
27 program main
28   use co_all_m, only : co_all
29   implicit none
30   logical :: operand = .true.
31
32   associate(me=>this_image())
33     call co_all(operand)
34     if (me==1) print *, operand
35     if (me==num_images()) operand = .false.
36     call co_all(operand)
37     if (me==1) print *, operand
38   end associate
39 end program
```

Heat Equation Solver

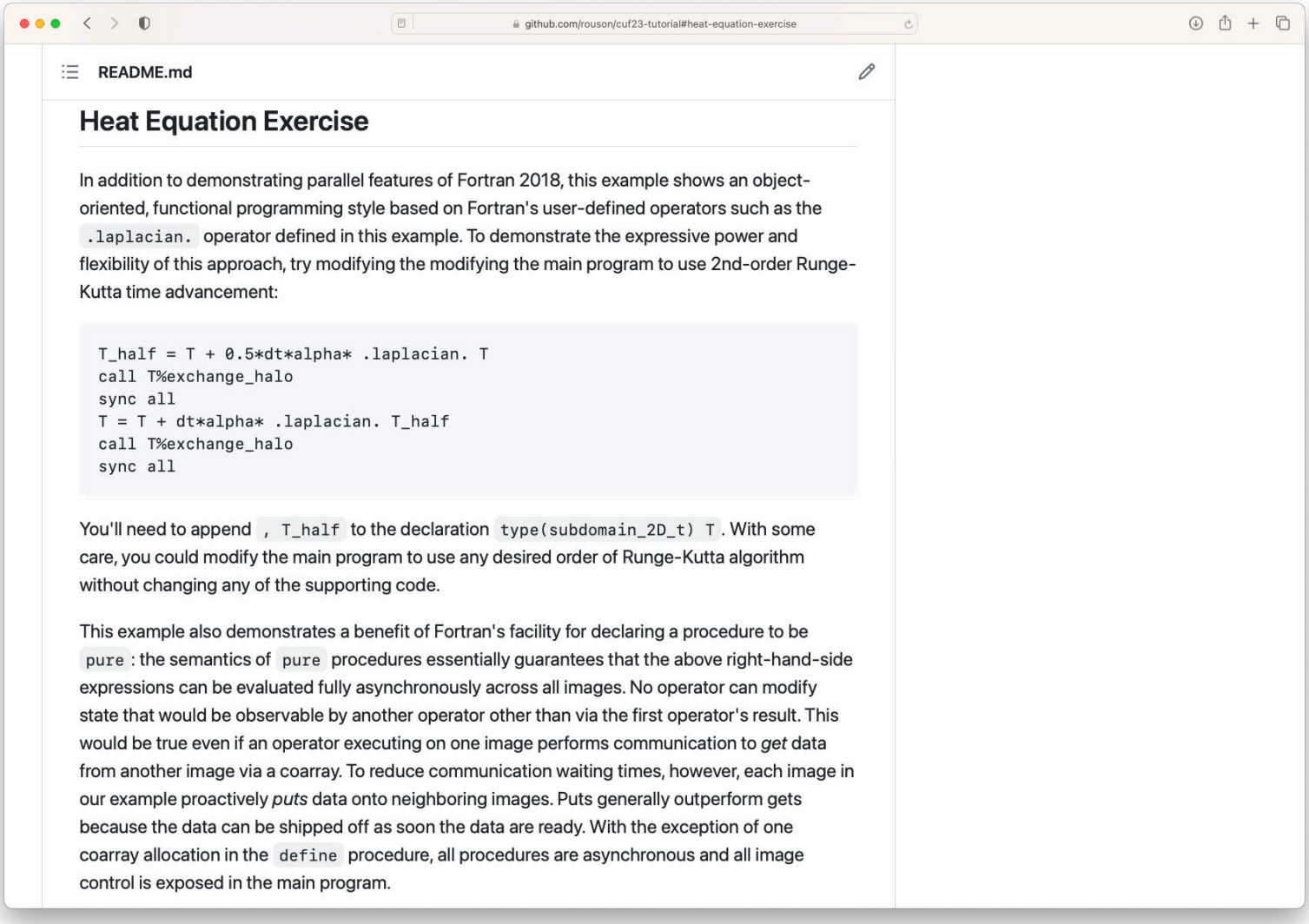
BERKELEY LAB
Bringing Science Solutions to the World

cuf23-tutorial — vim heat-equation.f90 — 110×39

```
240 program heat_equation
241   !! Parallel finite difference solver for the 2D, unsteady heat conduction partial differential equation
242   use subdomain_2D_m, only : subdomain_2D_t
243   use iso_fortran_env, only : int64
244   use kind_parameters_m, only : rkind
245   implicit none
246   type(subdomain_2D_t) T
247   integer, parameter :: nx = 4096, ny = nx, steps = 50
248   real(rkind), parameter :: alpha = 1._rkind
249   real(rkind) T_sum
250   integer(int64) t_start, t_finish, clock_rate
251   integer step
252
253   call T%define(side=1._rkind, boundary_val=1._rkind, internal_val=2._rkind, n=nx)! Initial/boundary cond.
254   call T%allocate_halo_coarray ! implicit synchronization
255
256   associate(dt => T%dx()*T%dy()/(4*alpha)) ! set time step
257
258     call system_clock(t_start)
259
260     do step = 1, steps
261       call T%exchange_halo ! put subdomain boundary values on neighboring images
262       sync all
263       T =  T + dt * alpha * .laplacian. T ! asynchronous parallel user-defined operators
264       sync all
265     end do
266
267   end associate
268
269   T_sum = sum(T%values()) ! local sum
270   call co_sum(T_sum, result_image=1) ! distributed collective sum
271
272   call system_clock(t_finish, clock_rate)
273   if (this_image()==1) then
274     print *, "walltime: ", real(t_finish - t_start, rkind) / real(clock_rate, rkind)
275     print *,"T_avg = ", T_sum/(nx*ny)
276   end if
277 end program
```

Hands-On Heat Equation

BERKELEY LAB
Bringing Science Solutions to the World



Coarrays

BERKELEY LAB
Bringing Science Solutions to the World

Non-allocatable (static):

```
character(len=max_greeting_length) :: greeting[*]
```

Dynamically allocatable:

```
real(rkind), allocatable :: halo_x(:,:)[:]
```

Derived type components:

```
type global_field_t
  real, allocatable :: values_(:)[:]
end type
```

Local coarrays:

```
subroutine gather_image_numbers
  integer, allocatable :: images(:)[:]
  allocate(images(num_images())[*])
end subroutine
```

Derived type coarrays:

```
type payload_list_t
  type(payload_t), allocatable :: payloads(:)
end type

type(payload_list_t), allocatable :: mailbox[:]
```

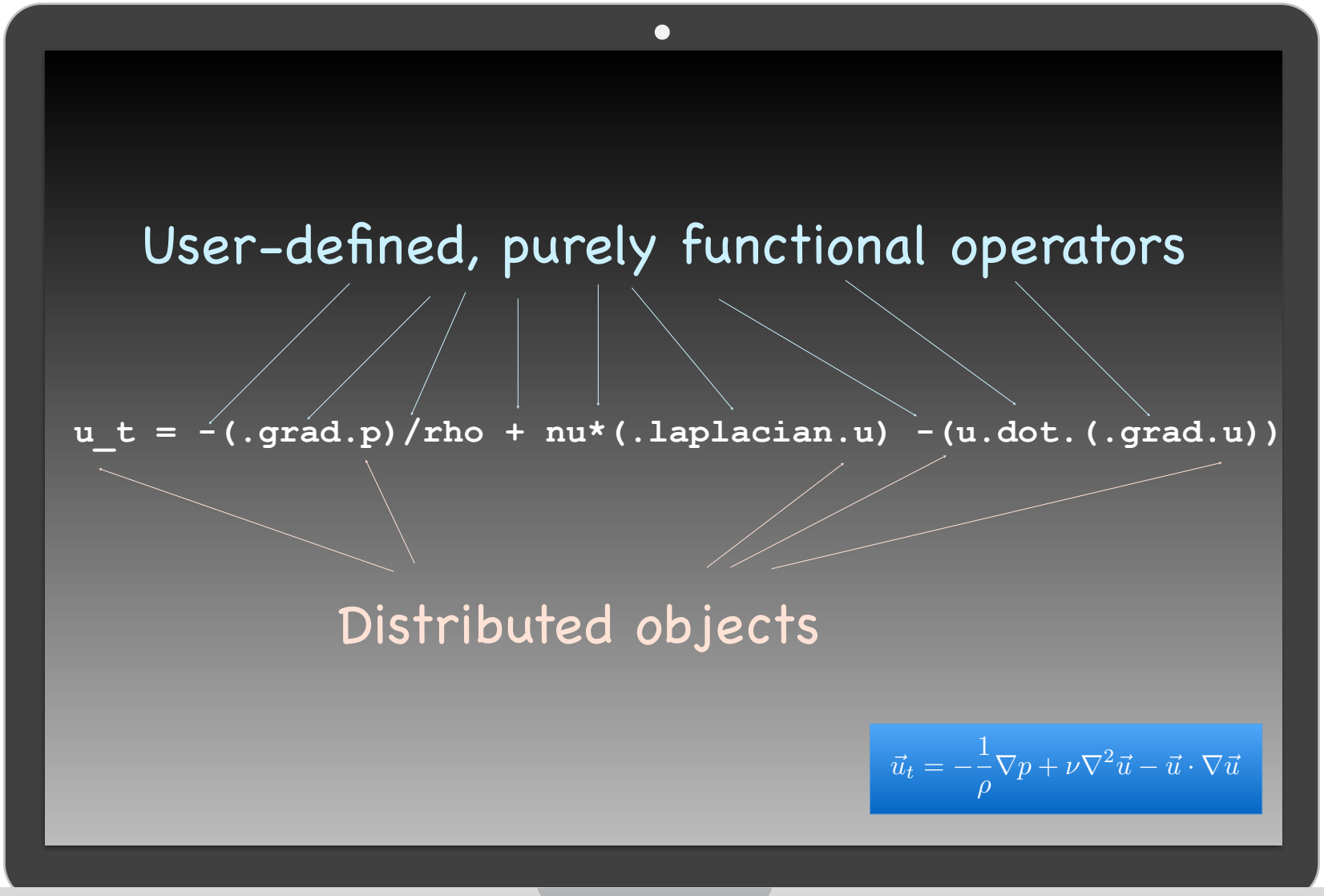
A coarray is a data entity that has nonzero corank; it can be directly referenced or defined by other images. It may be a scalar or an array.

For each coarray on an image, there is a corresponding coarray with the same type, type parameters, and bounds on every other image of a team in which it is established

=> Symmetric memory
if intrinsic-type coarray

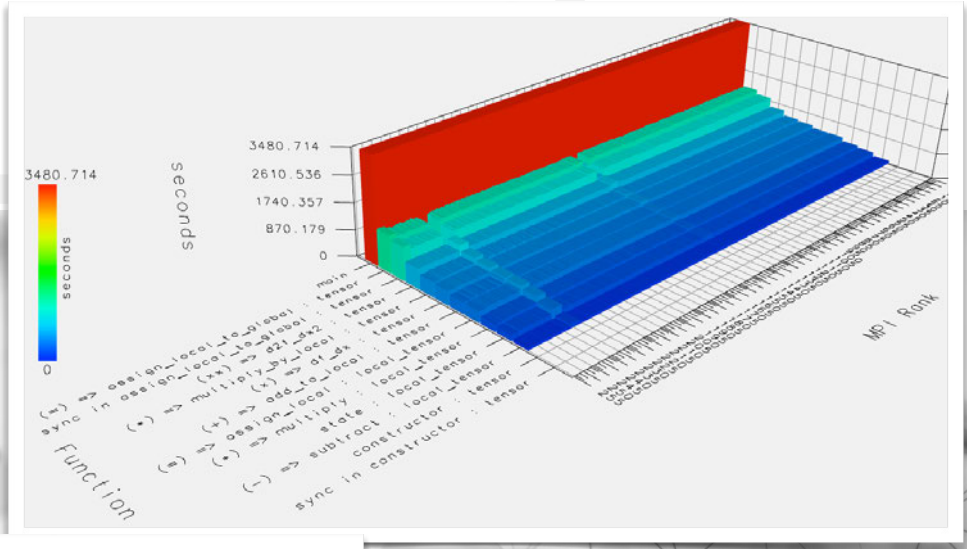
} Allow for asymmetric memory

Abstract Calculus Pattern



Burgers Eq. Solver

$$u_t = \nu u_{xx} - \left(\frac{u^2}{2}\right)_x$$



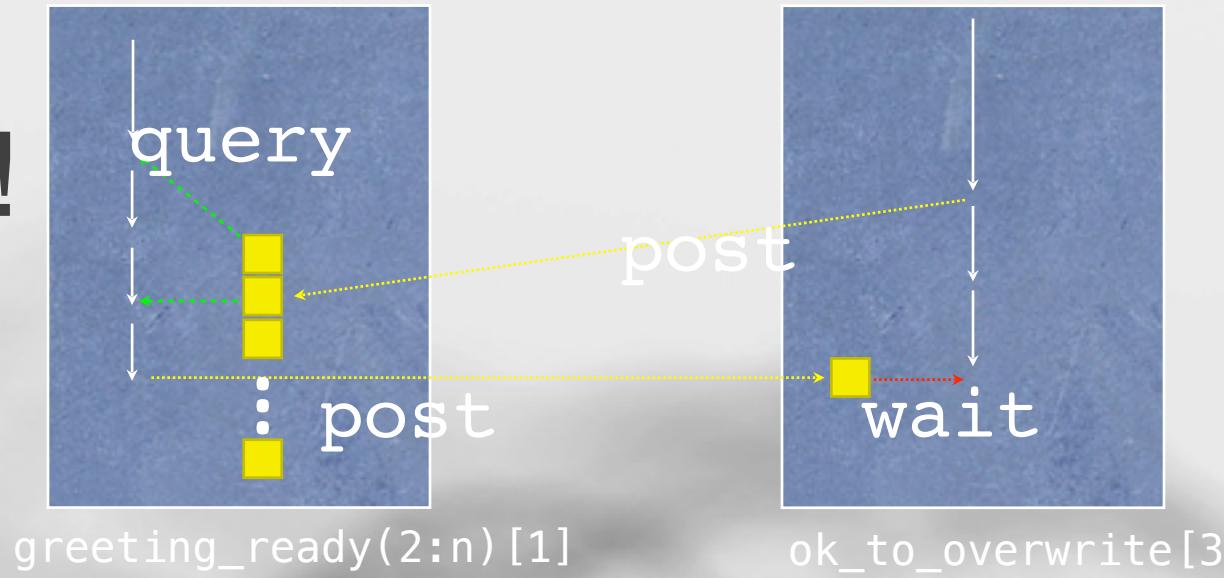
Burgers, J. M. (1948). A mathematical model illustrating the theory of turbulence. *Adv. Appl. Mech.* (1), 25-27.

Platform: Cray XE6 (Hopper at NERSC)

Rouson, Xia, & Xu (2011). *Scientific Software Design: The Object-Oriented Way.* Cambridge University Press.

Events

Hello, world!



Performance-oriented constraints:
- Query and wait must be local.
- Post and wait are disallowed in do concurrent constructs.

Pro tips:
- Overlap communication and computation.
- Wherever safety permits, query without waiting.

Segment Ordering: Events

An intrinsic module provides the derived type `event_type`, which encapsulates an `atomic_int_kind` integer component default-initialized to zero.

- An image increments the event count on a remote image by executing `event post`.
- The remote image obtains the post count by executing `event_query`.

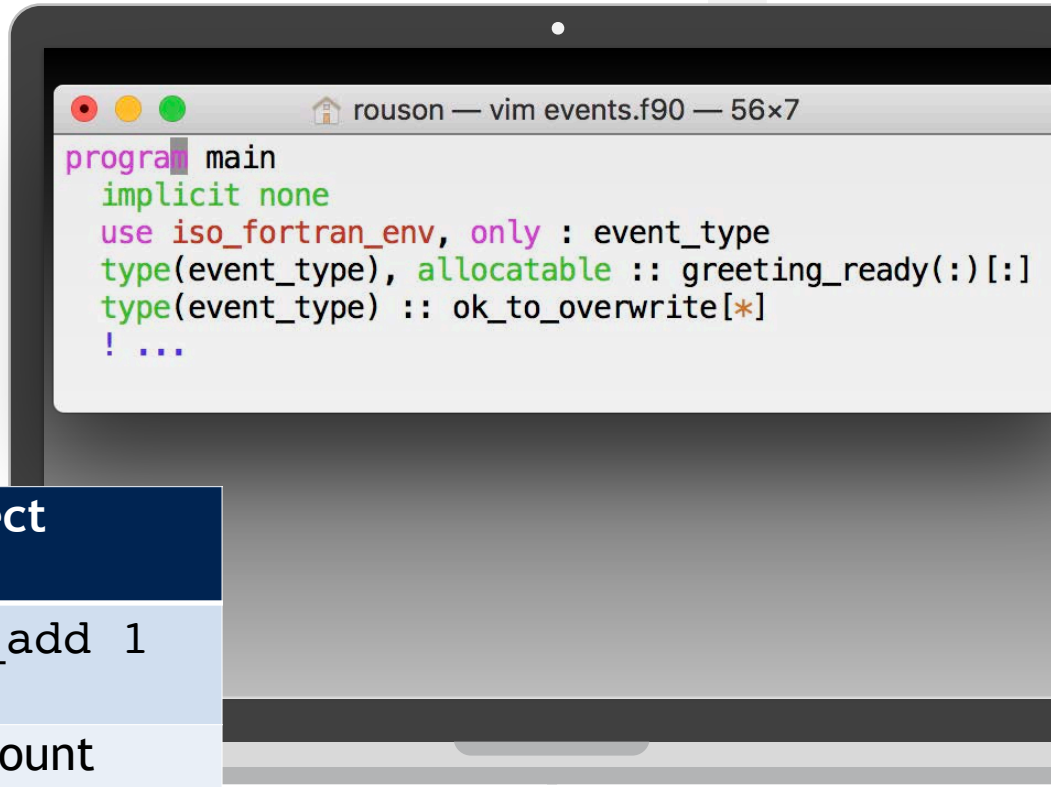
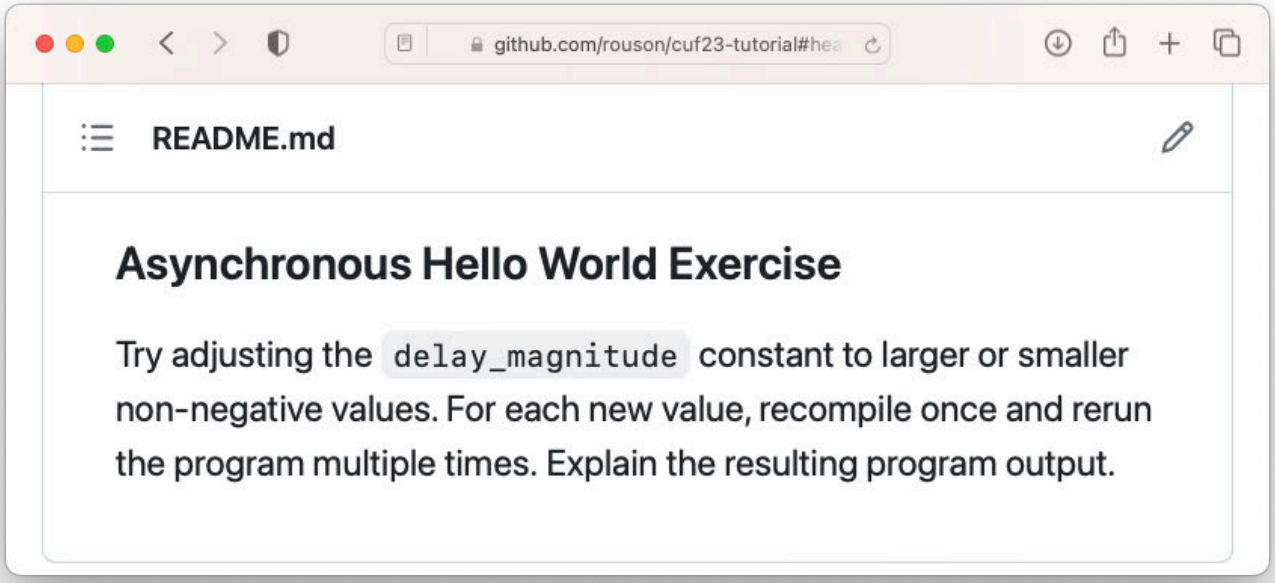


	Image Control	Side Effect
event post	☒	atomic_add 1
event_query		defines count
event wait	☒	atomic_add -1

Hands-On Asynchronous “Hello, World!”

BERKELEY LAB
Bringing Science Solutions to the World



FEATS:

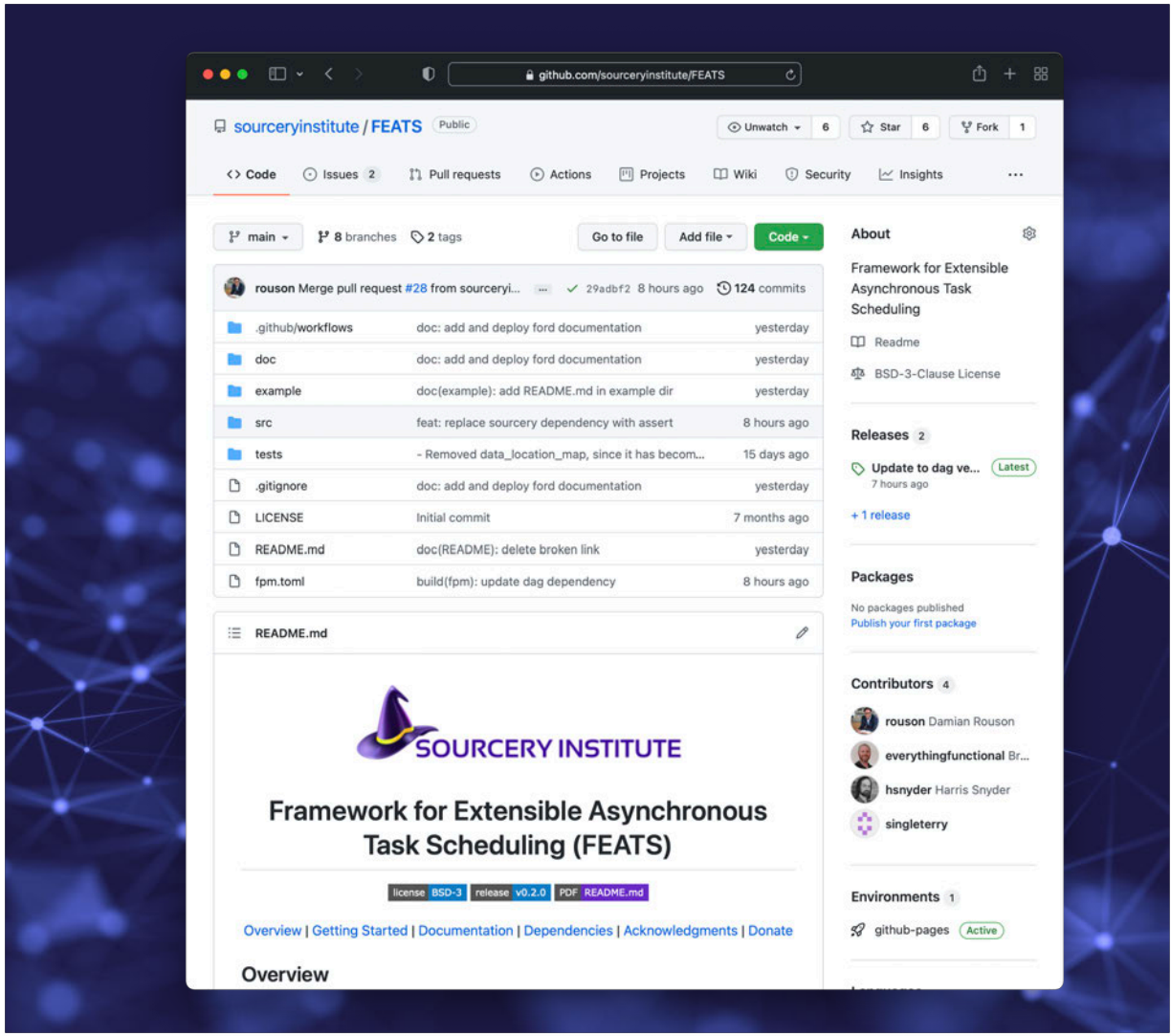
Framework for Extensible Asynchronous Task Scheduling

Execution:

- In each team, establish one scheduler image and one or more compute images.
- Schedulers post task_assigned events to compute images in an order that respects dependencies in a directed acyclic graph (DAG).
- Compute images post ready_for_next_task events to scheduler.
- A task_payload_map_t abstraction maps task task identifiers to locations in a payload_t mailbox coarray.

Initial target applications:

- NASA's Online Tool for the Assessment of Radiation in Space (OLTARIS)
- NCAR's Intermediate Complexity Atmospheric Research (ICAR) model: work-sharing/work-stealing.
- Fortran Package Manager: parallel builds.



FEATS:

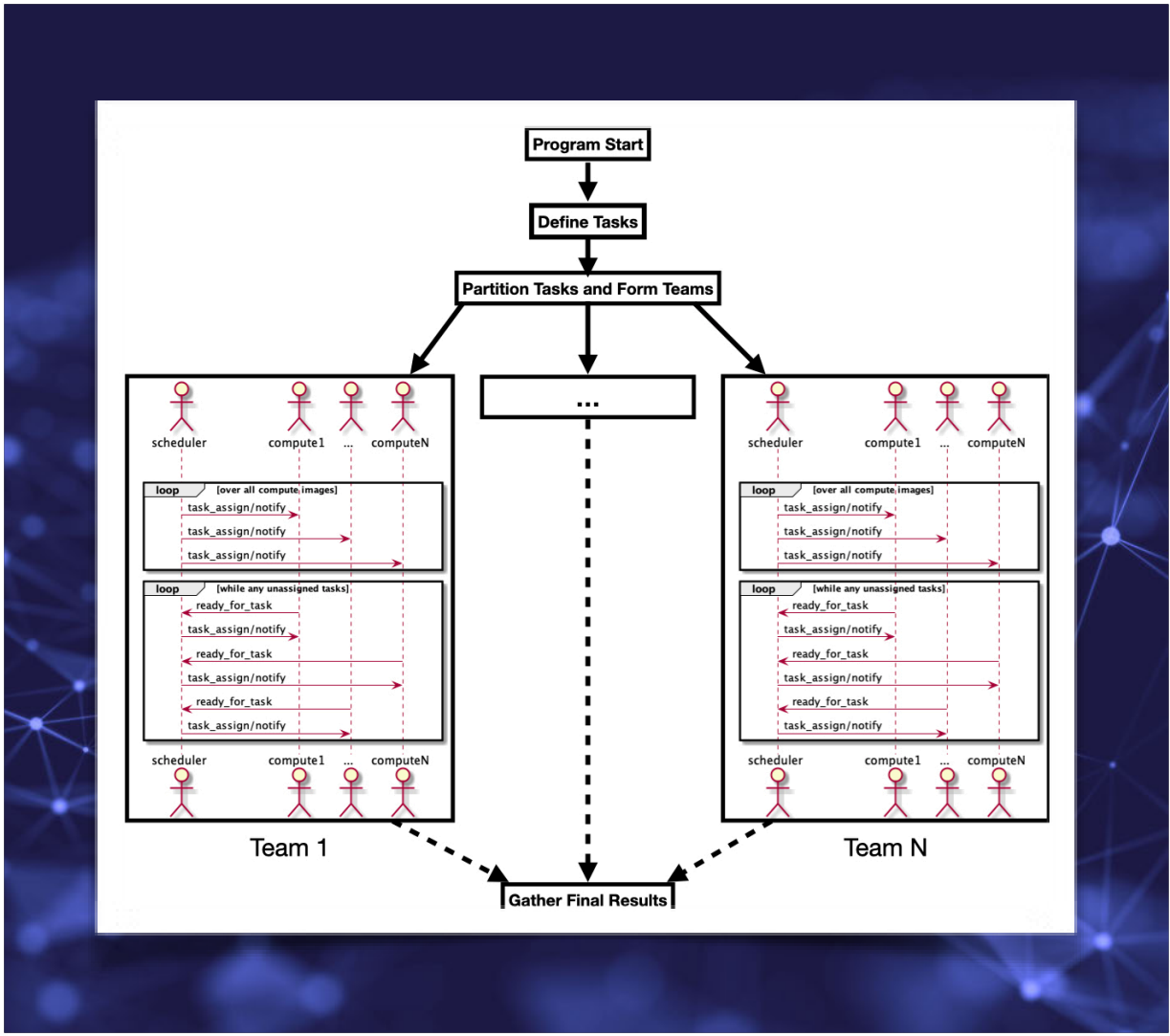
Framework for Extensible Asynchronous Task Scheduling

Execution:

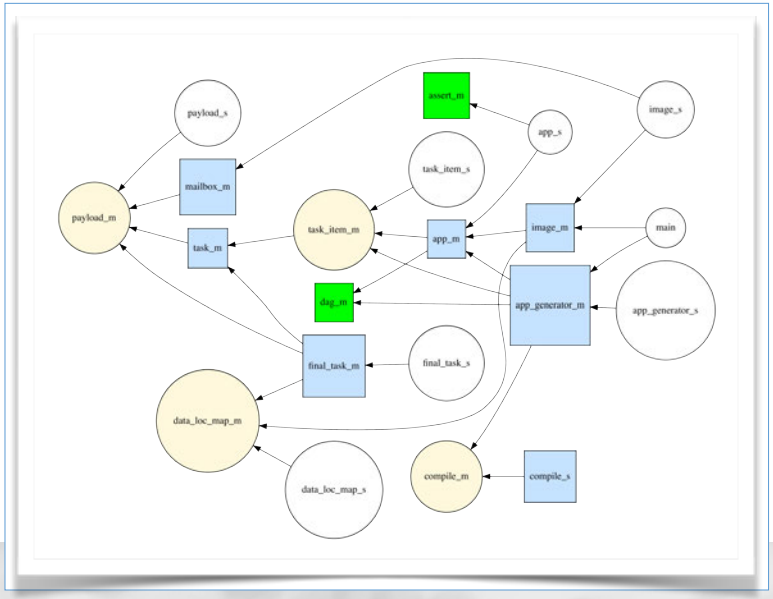
- In each team, establish one scheduler image and one or more compute images.
- Schedulers post task_assigned events to compute images in an order that respects dependencies in a directed acyclic graph (DAG).
- Compute images post ready_for_next_task events to scheduler.
- A task_payload_map_t abstraction maps task task identifiers to locations in a payload_t mailbox coarray.

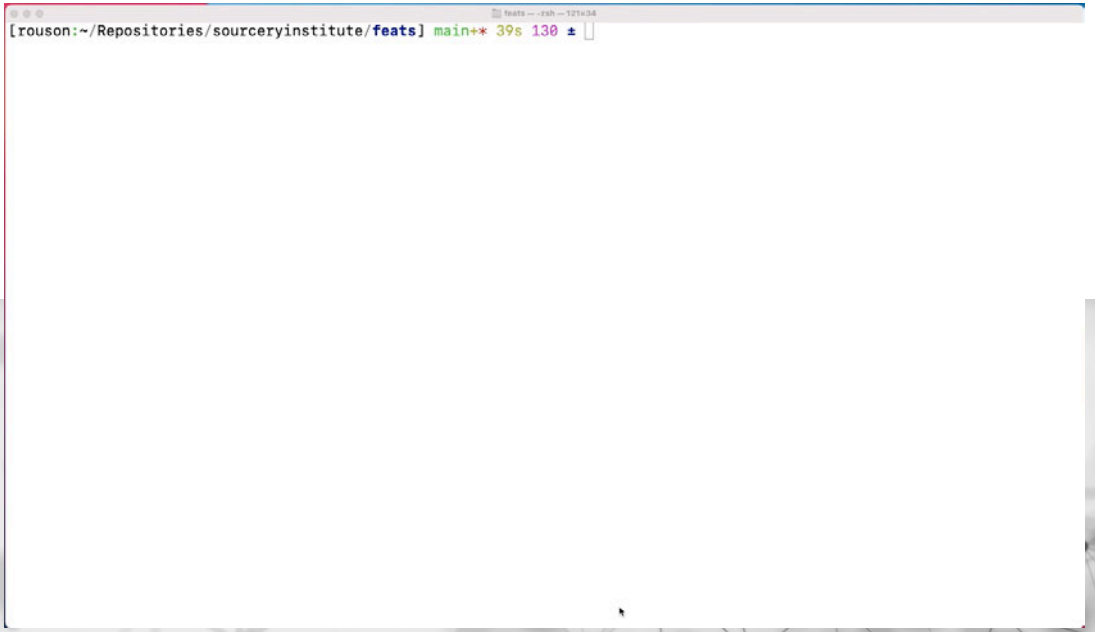
Initial target applications:

- NASA's Online Tool for the Assessment of Radiation in Space (OLTARIS)
- NCAR's Intermediate Complexity Atmospheric Research (ICAR) model: work-sharing/work-stealing.
- Fortran Package Manager: parallel builds.



Demo


payload_s
assert_m
image_s
app_s
task_item_s
mailbox_m
payload_m
task_item_m
app_m
image_m
main
task_m
dag_m
app_generator_m
app_generator_s
final_task_m
final_task_s
data_loc_map_m
data_loc_map_s
compile_m
compile_s


[rouson:~/Repositories/sourceryinstitute/feats] main+* 39s 130 ±

Coming Soon to a Computer Screen Near You

- Fortran 2023
  - Reductions in `do concurrent`
  - Notified access for remote coarray data

- Fortran 202Y (Y ~ 8)
  - Type-safe generic programming
  - Task-based parallel programming