



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



**Hewlett Packard
Enterprise**



SOURCERY INSTITUTE



LEADERSHIP
COMPUTING
FACILITY



National Energy Research
Scientific Computing Center



EXASCALE COMPUTING PROJECT

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.

When poll is active, respond at pollev.com/michellestrout402

Text **MICHELLESTROUT402** to **22333** once to join

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

Conceptual global address space

Process
w/virtual
address
space

Process
w/virtual
address
space

Process
w/virtual
address
space

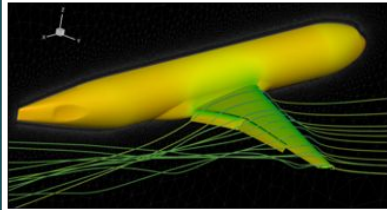
Process
w/virtual
address
space

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
 - 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



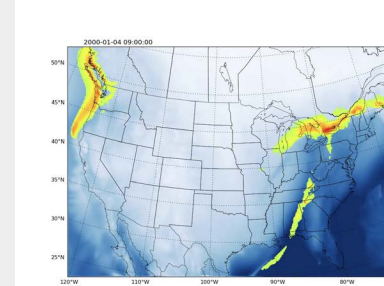
Production Applications using these Programming Models



CHAMPS: 3D Unstructured CFD

(~100K lines of Chapel)

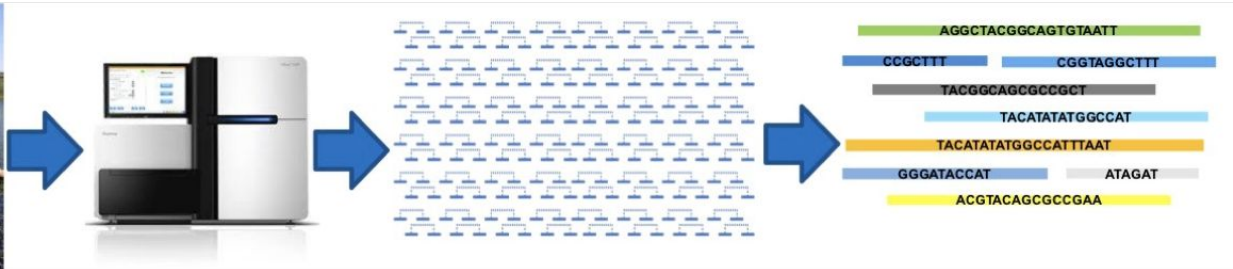
Éric Laurendeau, Simon Bourgault-Côté,
Matthieu Parenteau, et al.
École Polytechnique Montréal



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](https://en.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 $\Delta \mathbf{x} = \Delta \mathbf{y}$, and let $\alpha = \nu \Delta t / \Delta \mathbf{x}^2$

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

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.



⚠ When survey is active, respond at pollev.com/michellestrout402

Three questions about how you program

0 done

↻ **0 underway**

Start the presentation to see live content. For screen share software, share the entire screen. Get help at pollev.com/app



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

go.lbl.gov/cuf23

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





**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,

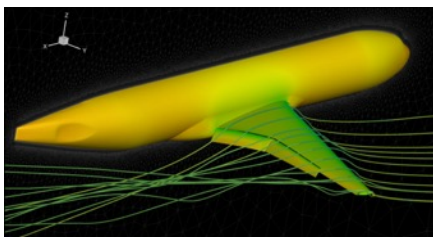
developing new codes,

serving high performance to Python codes **(Chapel server with Python client)**, and

providing distributed and shared memory parallelism for existing codes.

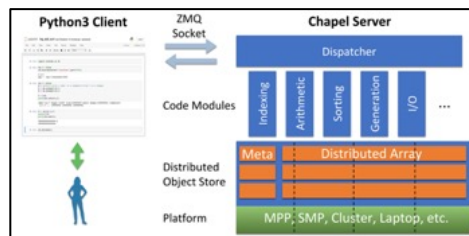


APPLICATIONS OF CHAPEL: LINKS TO USERS' TALKS (SLIDES + VIDEO)



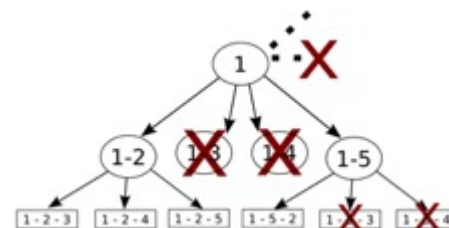
CHAMPS: 3D Unstructured CFD

[CHIOW 2021](#) [CHIOW 2022](#)



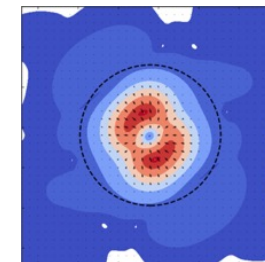
Arkouda: Interactive Data Science at Massive Scale

[CHIOW 2020](#) [CHIOW 2023](#)



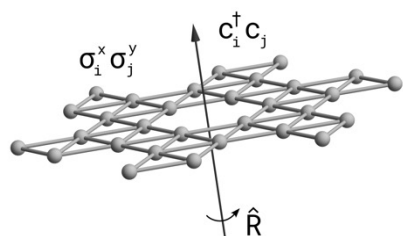
ChOp: Chapel-based Optimization

[CHIOW 2021](#) [CHIOW 2023](#)



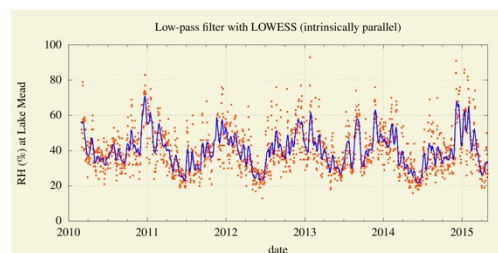
ChpUltra: Simulating Ultralight Dark Matter

[CHIOW 2020](#) [CHIOW 2022](#)



Lattice-Symmetries: a Quantum Many-Body Toolbox

[CHIOW 2022](#)



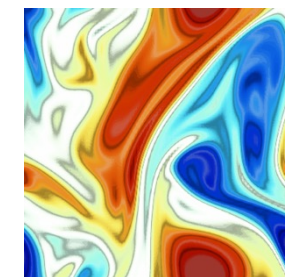
Desk dot chpl: Utilities for Environmental Eng.

[CHIOW 2022](#)

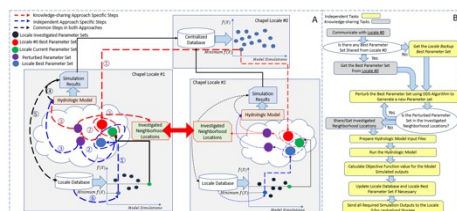


RapidQ: Mapping Coral Biodiversity

[CHIOW 2023](#)

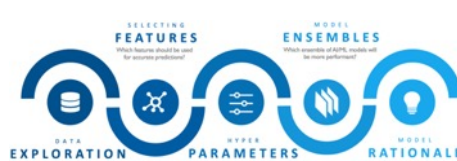


ChapQG: Layered Quasigeostrophic CFD



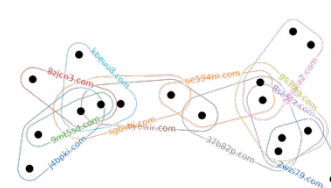
Chapel-based Hydrological Model Calibration

[CHIOW 2023](#)



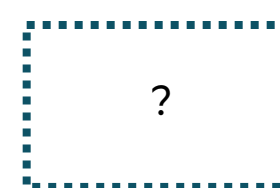
CrayAI HyperParameter Optimization (HPO)

[CHIOW 2021](#)



CHGL: Chapel Hypergraph Library

[CHIOW 2020](#)

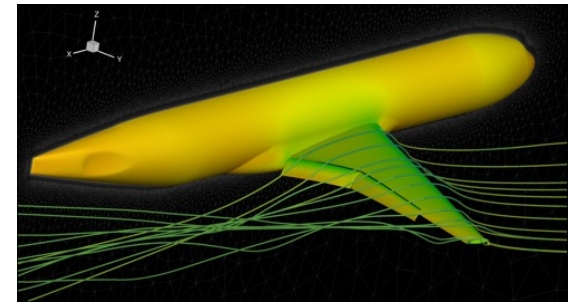


Your Application Here?

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)



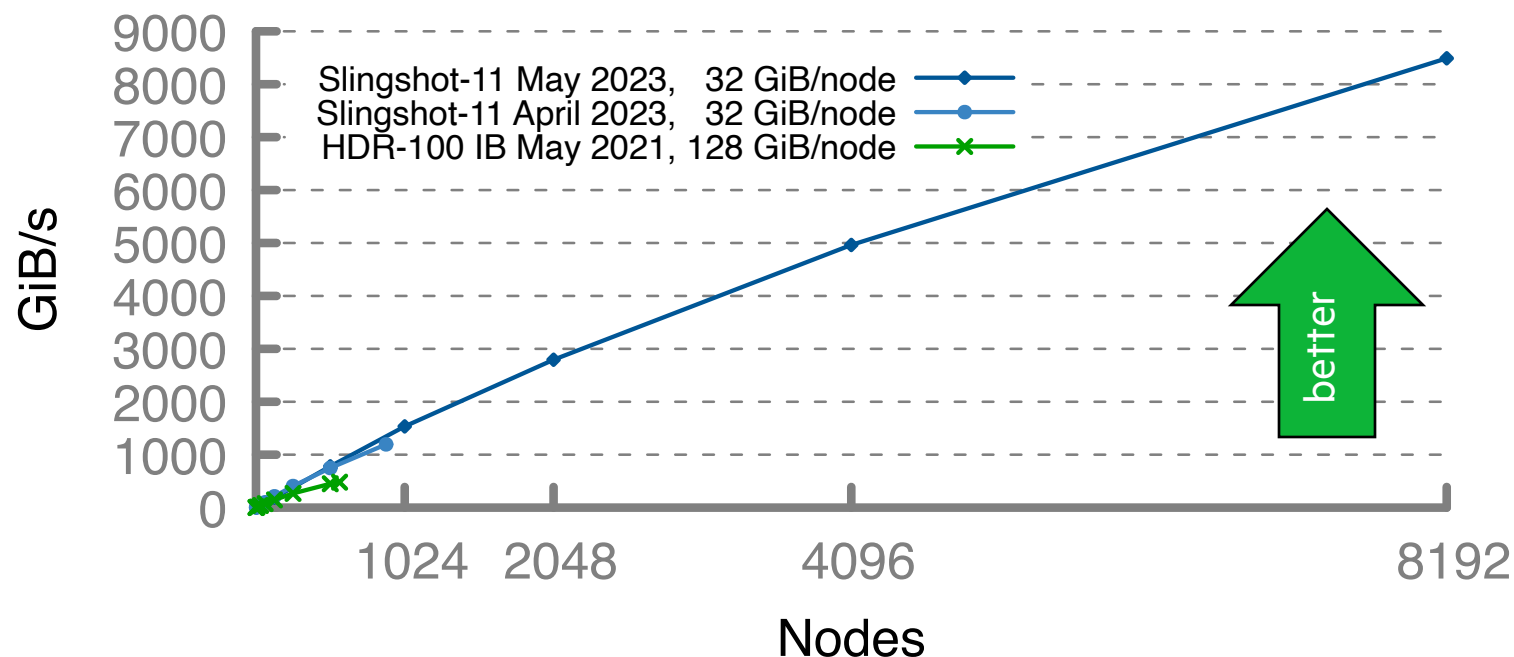
- Slingshot-11 network (200 Gb/s)
- 896 compute nodes
- 28 TiB of 8-byte values
- ~1200 GiB/s (~24 seconds)

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)

Arkouda ArgSORT Performance



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



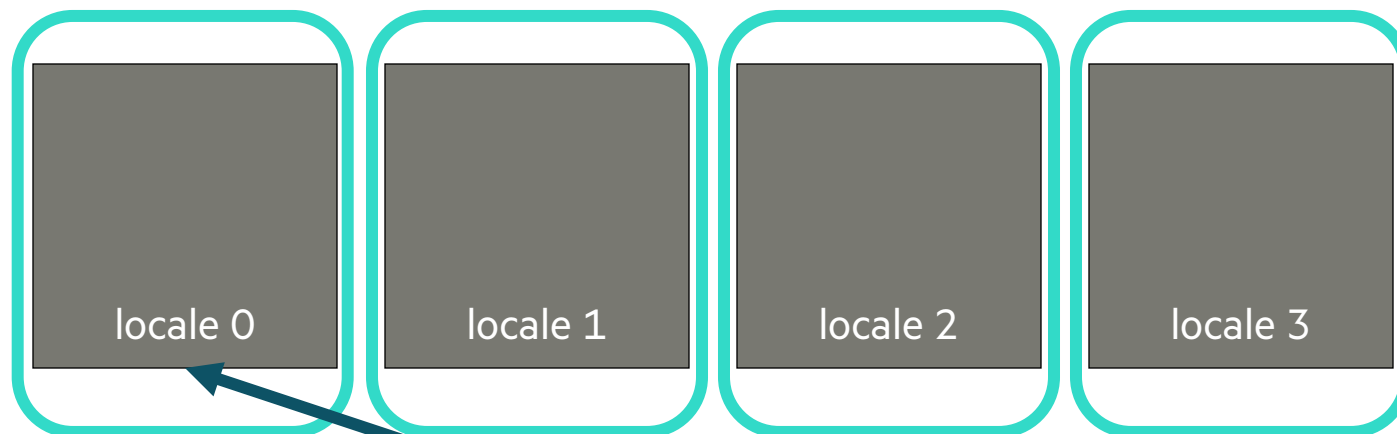
CHAPEL EXECUTION MODEL AND TERMINOLOGY: LOCALES

- 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'
```

Four nodes/CPU's

Locales array :



User's code starts running as a single task on locale 0

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:

Locale 0

Locale 1

Locale 2

Locale 3

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

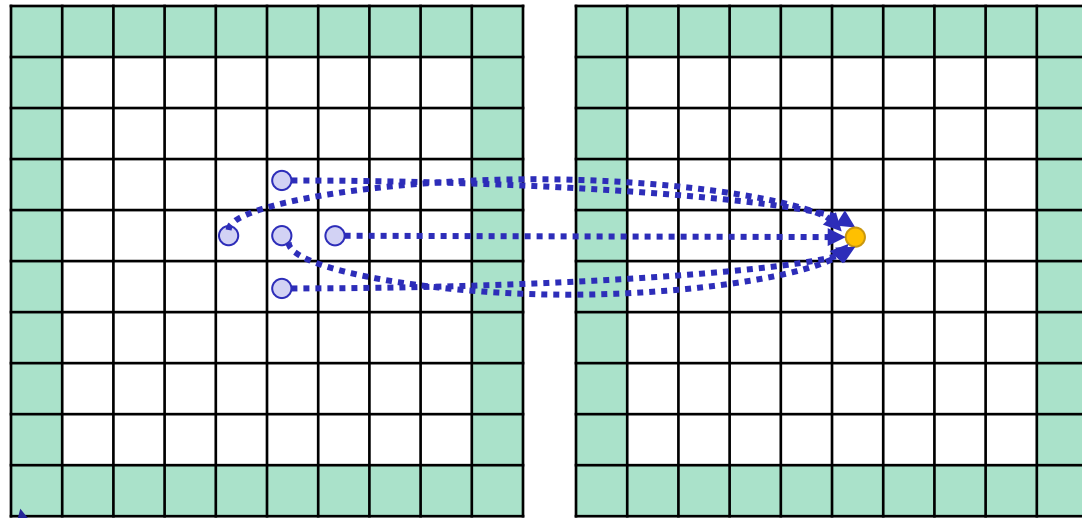
See <https://go.lbl.gov/cuf23-repo> for more info and for example code.

- **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



u^n

Stored in un

u^{n+1}

Stored in u

Fixed
boundary
values

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

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

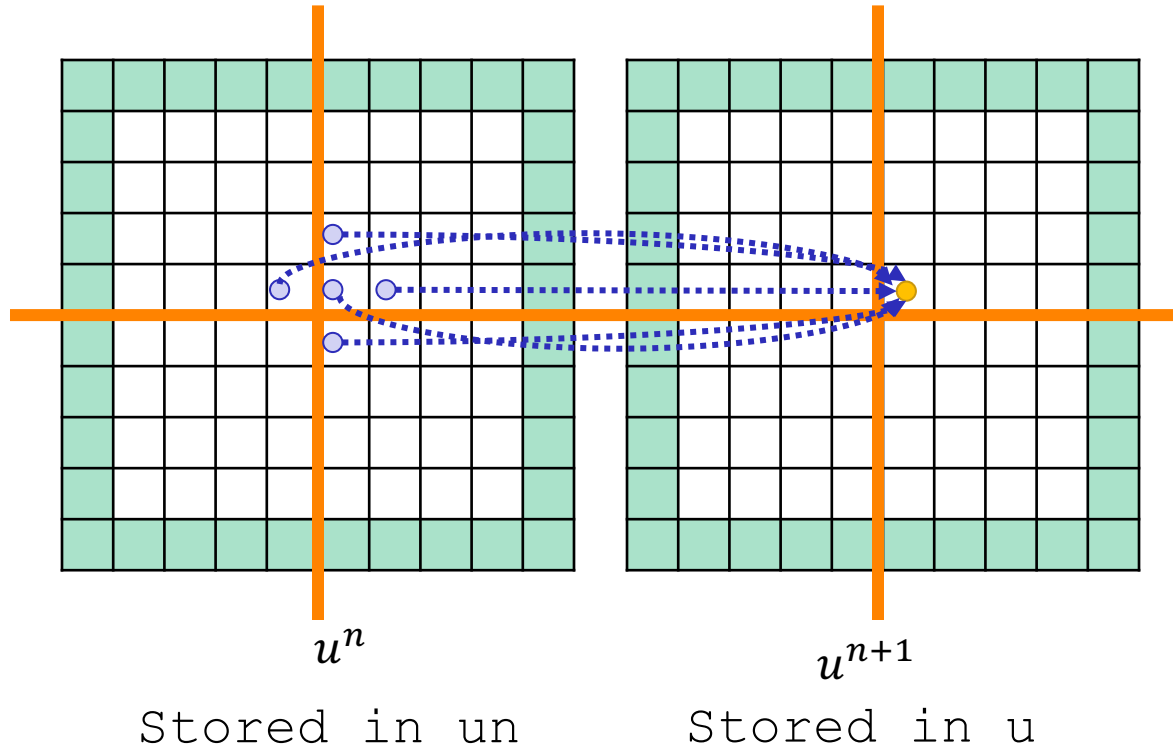
$$u_{i,j}^{n+1} = u_{i,j}^n + \alpha (u_{i+1,j}^n + u_{i-1,j}^n - 4u_{i,j}^n + u_{i,j+1}^n + u_{i,j-1}^n)$$

- 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 parallelism

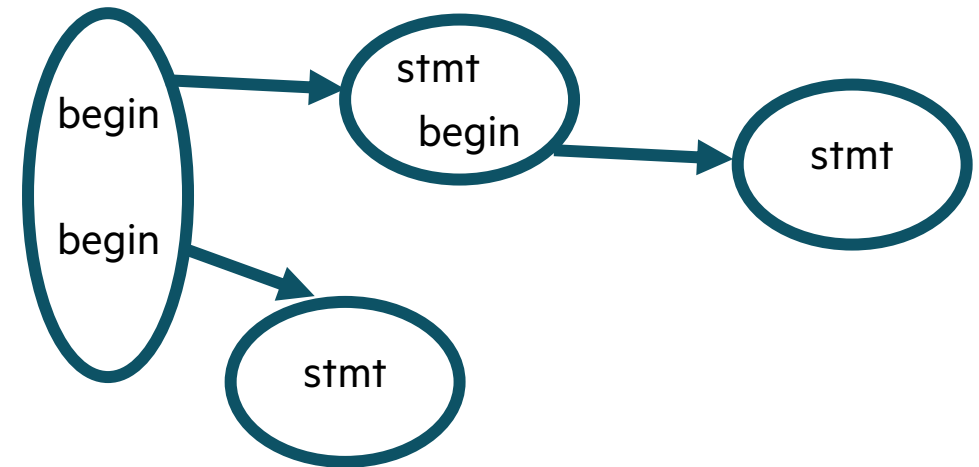
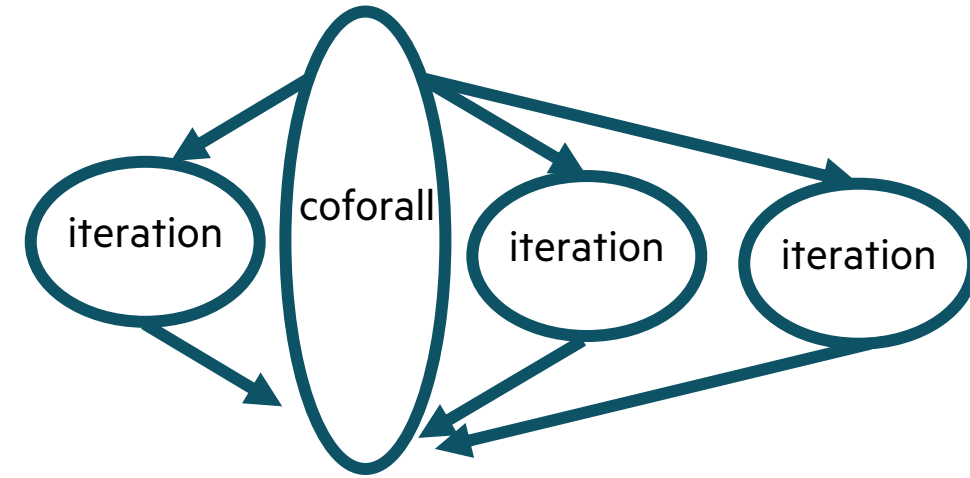
- 'coforall', distributed memory parallelism across processes/locales with 'on' syntax
- 'coforall', shared-memory parallelism over threads
- 'cobegin', executes all statements in block in parallel

• 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”)

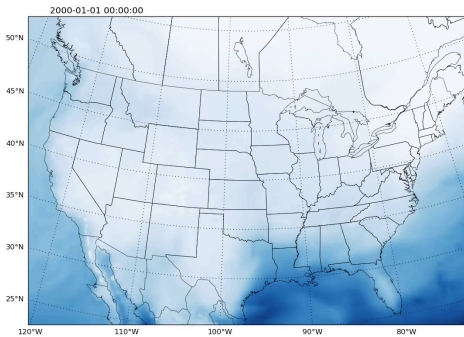
- Why Fortran Matters
- SPMD parallel execution
- PGAS data structures & RMA



Heat Conduction Solver

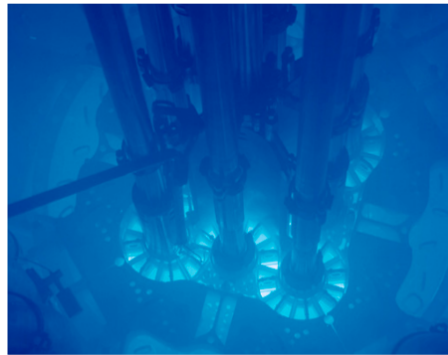
- Compiling and running it
- Understanding it

Why Fortran Matters



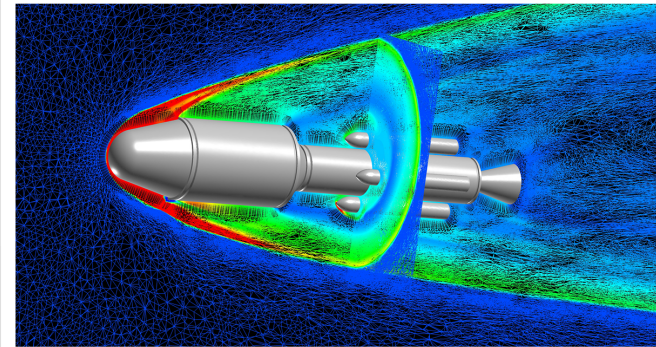
Intermediate Complexity Atmospheric
Research (ICAR) Model
Courtesy of Ethan Gutmann, NCAR

**Weather &
Climate**



U.S. Nuclear Regulatory Commission
File Photo

Nuclear Energy

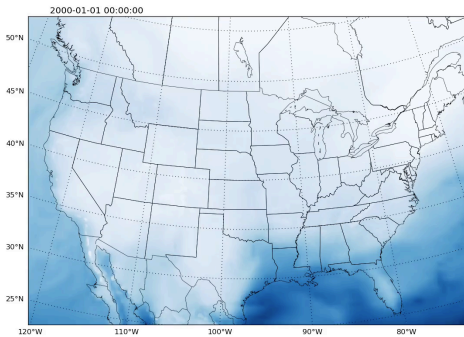


FUN3D Mesh Adaptation for Mars Ascent
Vehicle, Courtesy of Eric Nielsen & Ashley
Korzun, NASA Langley

3

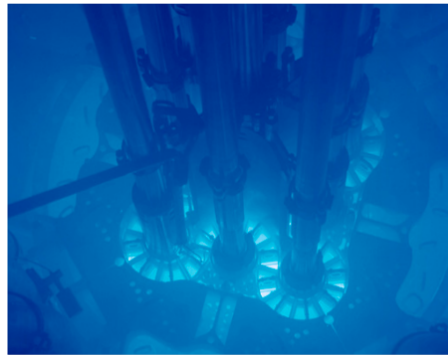
Aerospace

Why Fortran Matters



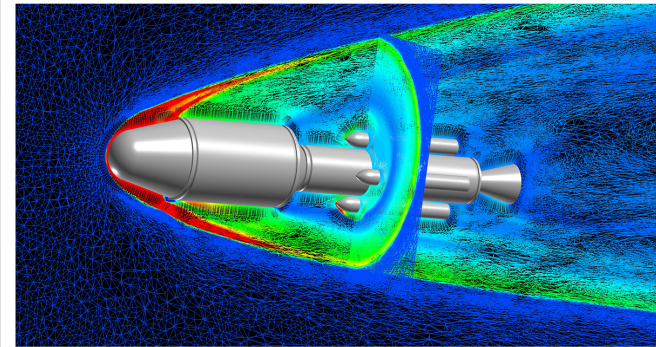
Intermediate Complexity Atmospheric
Research (ICAR) Model
Courtesy of Ethan Gutmann, NCAR

**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

3

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

A screenshot of a vim editor window. The title bar shows "rouson — vim hi.f90 — 67x5". The code is as follows:

```
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

A terminal window with a title bar containing three colored circles (red, yellow, green) and the text "rouson — zsh — 64x19". The terminal content shows the prompt "cuf23-tutorial:" followed by a cursor.

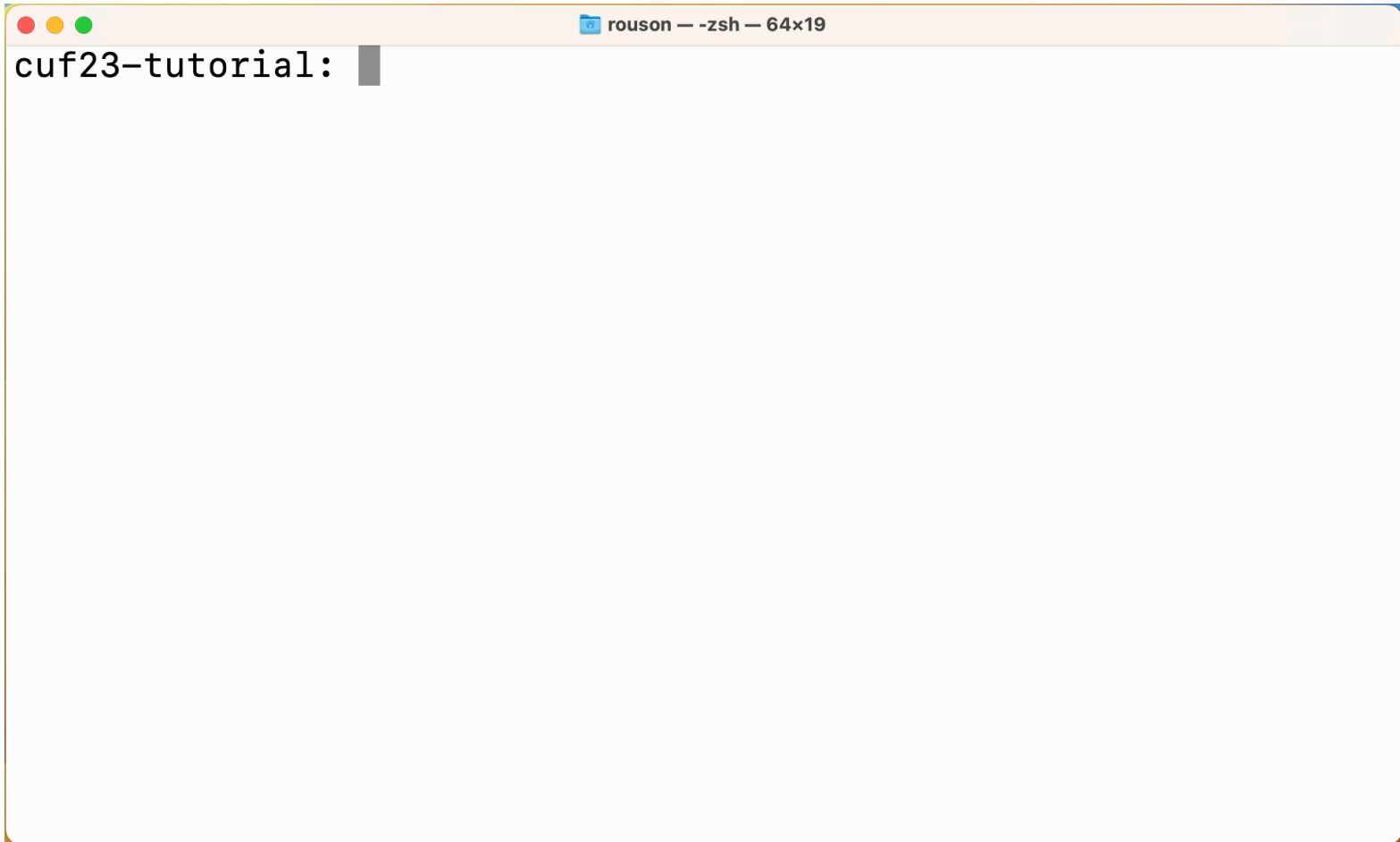
```
cuf23-tutorial: 
```

Compiling and Running hi.f90



BERKELEY LAB

Bringing Science Solutions to the World

A terminal window with a title bar containing three colored buttons (red, yellow, green) and a text label "rouson — zsh — 64x19". The terminal content shows the prompt "cuf23-tutorial:" followed by a cursor.

```
rouson — zsh — 64x19
cuf23-tutorial: █
```


SPMD Execution Sequence



BERKELEY LAB

Bringing Science Solutions to the World

Image 1

A screenshot of a code editor window titled "rouson - vim hi.f90 - 67x5". It contains the following Fortran code:

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

Time

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.

SPMD Execution Sequence



BERKELEY LAB

Bringing Science Solutions to the World

Image 1

```
rouson — vim hi.f90 — 67x5
1 program main
2   implicit none
3   print *, "Hello from image ", this_image(), " of ", num_images()
4 end program
```

Image 2

```
rouson — vim hi.f90 — 67x5
1 program main
2   implicit none
3   print *, "Hello from image ", this_image(), " of ", num_images()
4 end program
```

Time

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.

SPMD Execution Sequence



BERKELEY LAB

Bringing Science Solutions to the World

Image 1

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

Image 2

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

Time

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.

SPMD Execution Sequence



BERKELEY LAB

Bringing Science Solutions to the World

Image 1

```
rouson — vim hi.f90 — 67x5
1 program main
2   implicit none
3   print *, "Hello from image ", this_image(), " of ", num_images()
4 end program
```

Image 2

```
rouson — vim hi.f90 — 67x5
1 program main
2   implicit none
3   print *, "Hello from image ", this_image(), " of ", num_images()
4 end program
```

Time

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.

SPMD Execution Sequence



BERKELEY LAB

Bringing Science Solutions to the World

Image 1

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

Image 2

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



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.

SPMD Execution Sequence



BERKELEY LAB

Bringing Science Solutions to the World

Image 1

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

Image 2

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



```
print *, "Hello from image ", this_image(), "of", num_images()
```

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.

SPMD Execution Sequence



BERKELEY LAB

Bringing Science Solutions to the World

Image 1

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

Image 2

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



```
print *, "Hello from image ", this_image(), "of", num_images()
```

```
print *, "Hello from image ", this_image(), "of", num_images()
```

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.

SPMD Execution Sequence



BERKELEY LAB

Bringing Science Solutions to the World

Image 1

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

Image 2

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



```
print *, "Hello from image ", this_image(), "of", num_images()
```

```
print *, "Hello from image ", this_image(), "of", num_images()
```

end program

end program



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.

SPMD Execution Sequence



BERKELEY LAB

Bringing Science Solutions to the World

Image 1

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

Image 2

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



```
print *, "Hello from image ", this_image(), "of", num_images()
```

```
print *, "Hello from image ", this_image(), "of", num_images()
```

end program

end program

} Image control statement

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

Coarrays:

- Distributed data structures — greeting
- Facilitate Remote Memory Access (RMA) — line 15

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

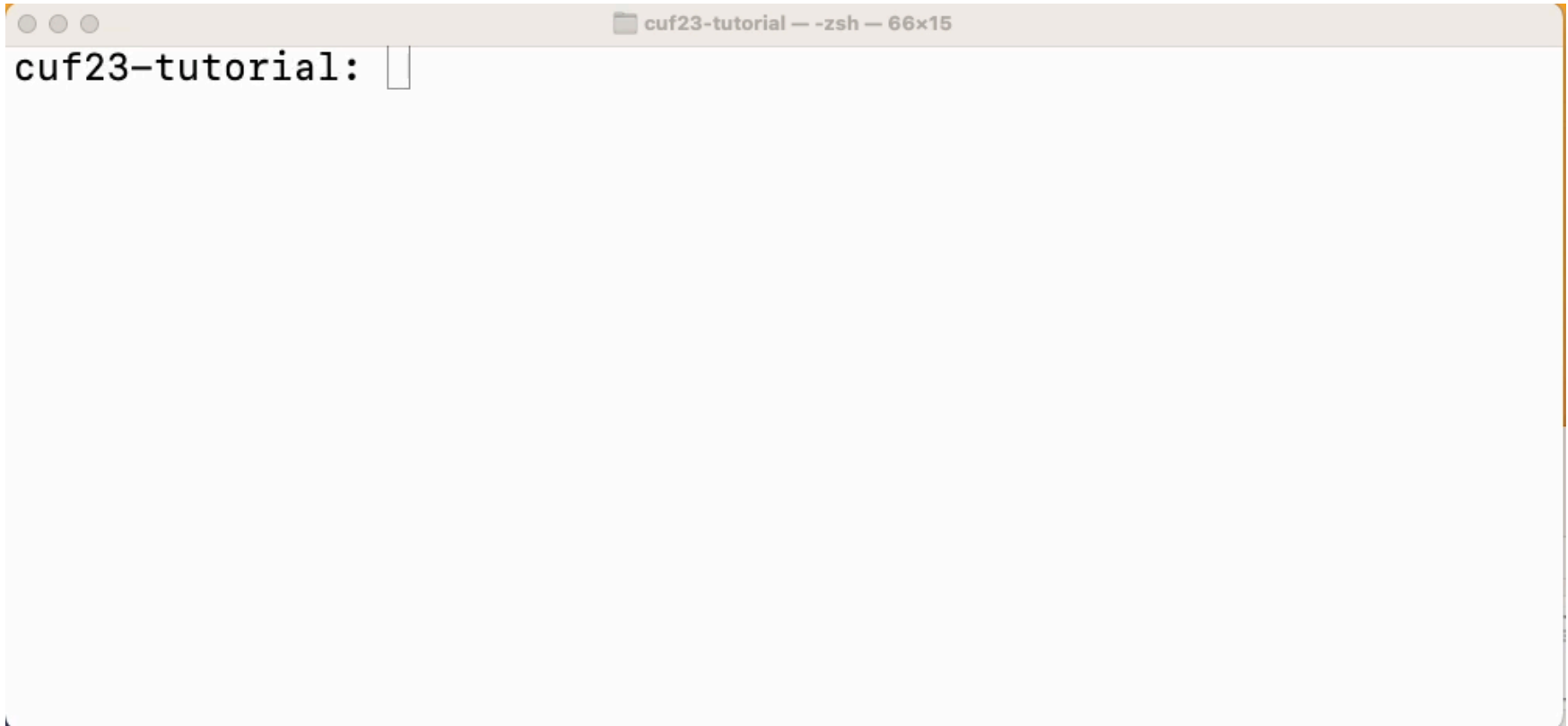
```
cuf23-tutorial — vim hello.f90 — 74x21
1 program main
2  !! One-sided communication of distributed greetings
3  implicit none
4  integer, parameter :: max_greeting_length=64, writer = 1
5  integer image
6  character(len=max_greeting_length) :: greeting[*] ! scalar coarray
7
8  associate(me => this_image(), 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

A terminal window with a title bar that reads "cuf23-tutorial — zsh — 66x15". The terminal content shows the prompt "cuf23-tutorial:" followed by a cursor. The terminal has a light gray background and a dark gray title bar.

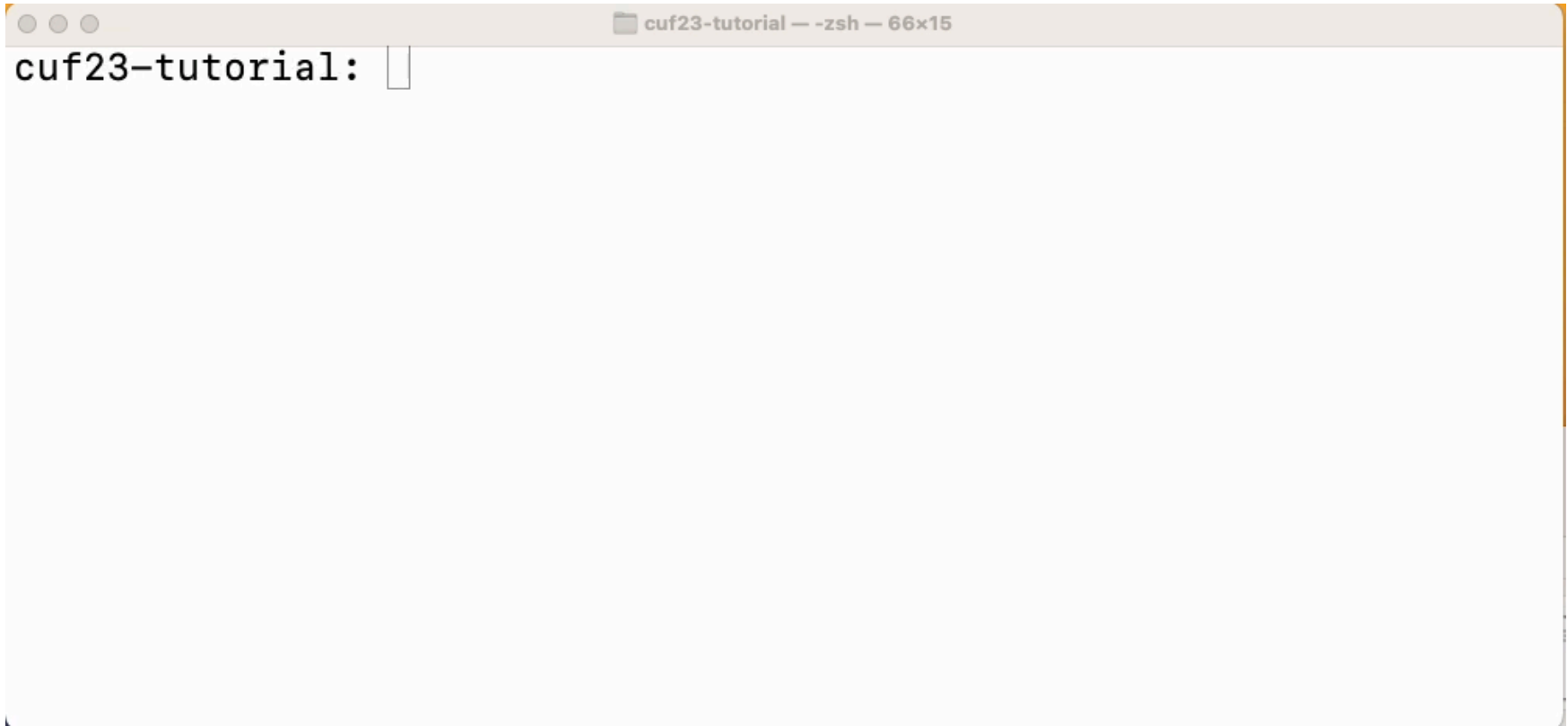
```
cuf23-tutorial: 
```

Compiling & Running hello.f90



BERKELEY LAB

Bringing Science Solutions to the World

A terminal window with a title bar that reads "cuf23-tutorial — zsh — 66x15". The terminal content shows the prompt "cuf23-tutorial:" followed by a cursor. The terminal has a light gray background and a dark gray title bar.

```
cuf23-tutorial: 
```

Compiling and Running the Heat Equation Solver



BERKELEY LAB

Bringing Science Solutions to the World

A terminal window titled "cuf23-tutorial — -zsh — 78x23". The prompt is "cuf23-tutorial:" followed by a cursor. A cursor is also visible in the center of the terminal area.

```
cuf23-tutorial: 
```

Compiling and Running the Heat Equation Solver



BERKELEY LAB

Bringing Science Solutions to the World

A terminal window titled "cuf23-tutorial — -zsh — 78x23". The prompt is "cuf23-tutorial:" followed by a cursor. A cursor is also visible in the center of the terminal area.

```
cuf23-tutorial: 
```

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

Heat Equation



BERKELEY LAB

Bringing Science Solutions to the World

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

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

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

$$T = T + dt * \alpha * \text{.laplacian.} T$$

local objects

pure user-defined operators

Class Diagram



BERKELEY LAB

Bringing Science Solutions to the World

C subdomain_2D_t

s_ : real[]

define()

laplacian(rhs: subdomain_2D_t) : subdomain_2D_t

multiply(lhs : subdomain_2D_t, rhs : subdomain_2D_t) : subdomain_2D_t

add(lhs : subdomain_2D_t, rhs : subdomain_2D_t) : subdomain_2D_t

copy(lhs : subdomain_2D_t, rhs : subdomain_2D_t)

dx()

dy()

values()

exchange_halo()

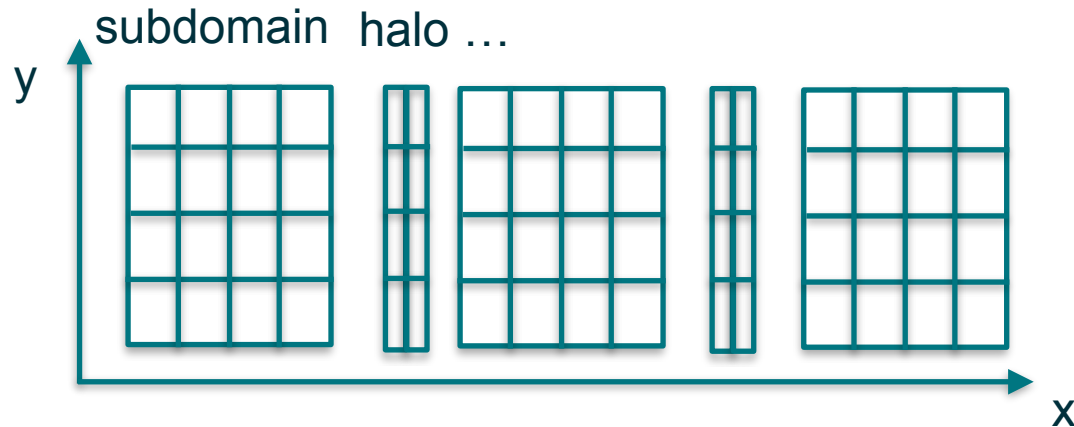
allocate_halo_coarray()

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

TAU: ParaProf: Statistics for: node 0 - /home/tutorial/SRC/demo/matcha

TAU: ParaProf: Statistics for: node 0 - /home/tutorial/SRC/demo/matcha

Name	Exclu...	Inclu...	Calls	Chil...
TAU application	0	1.516	1	1
taupreload_main	0.801	1.516	161,499	
[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



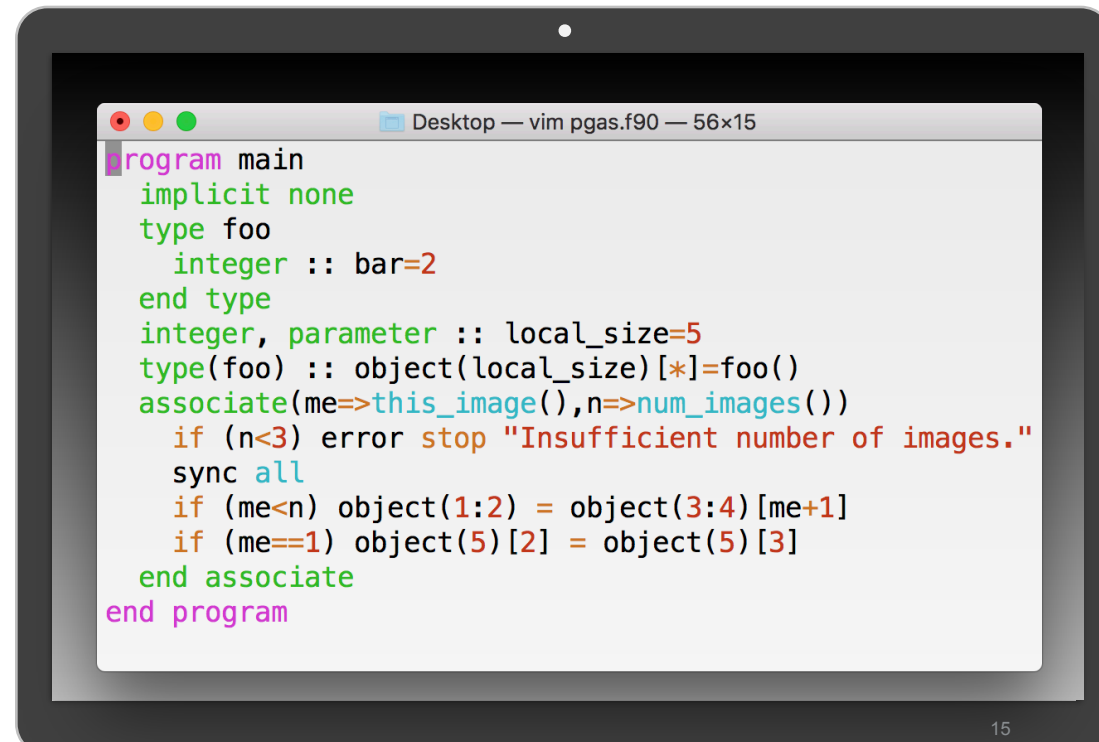
Minimally invasive:

- Drop brackets when not communicating



Communication is explicit:

- Use brackets when communicating





BERKELEY LAB

Bringing Science Solutions to the World



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.

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

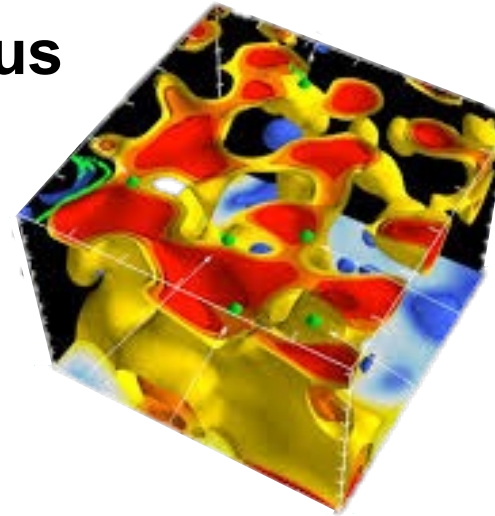
Design principles for performance

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

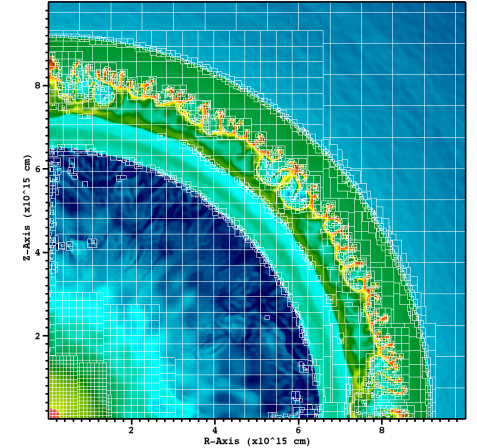
Some motivating applications

Many applications involve asynchronous updates to irregular data structures

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



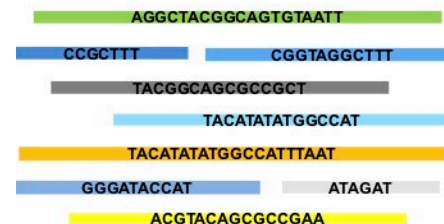
Seismo, Berkeley



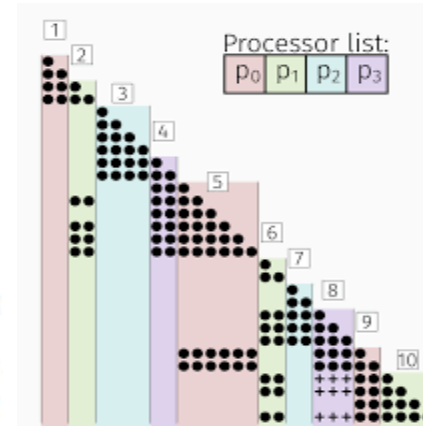
AMReX

Irregular and unpredictable data movement:

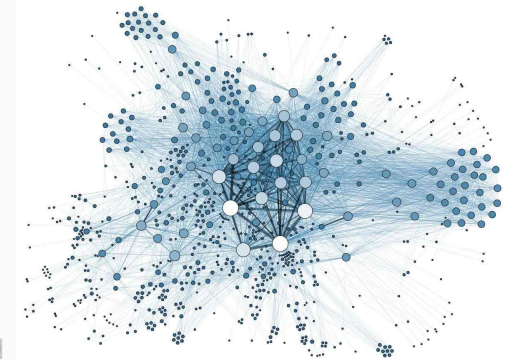
- *Space*: Pattern across processors
- *Time*: When data moves
- *Volume*: Size of data



ExaBiome



SymPACK



Graph analytics

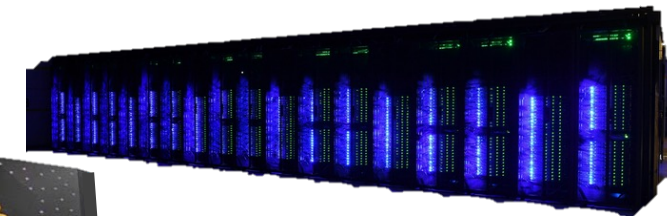
Some motivating system trends

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)



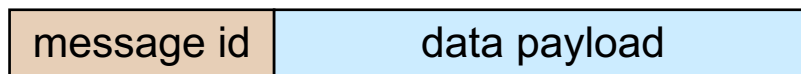
Reducing communication overhead

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

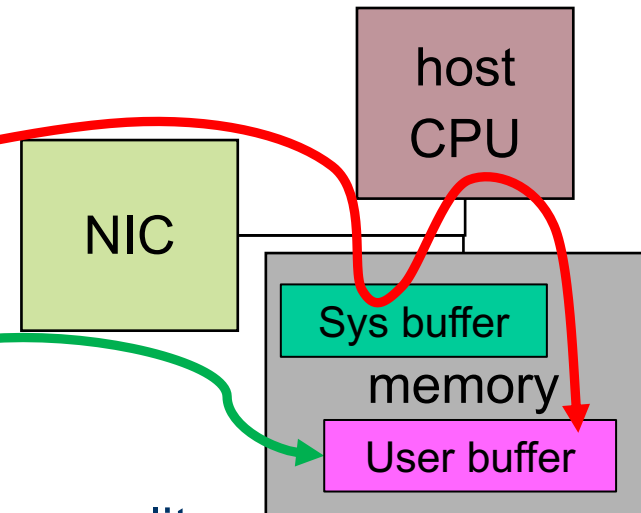
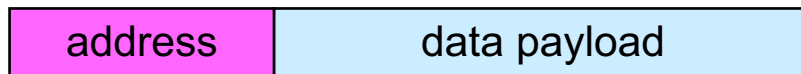
Communication is **one-sided** – there is no “receive” operation

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

two-sided message



one-sided RMA put



- 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

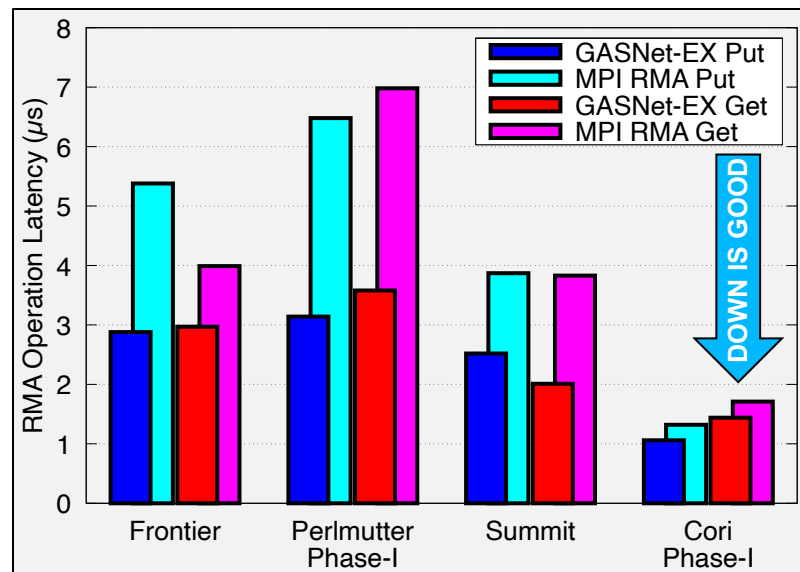
One-sided GASNet-EX vs one- and two-sided MPI

Four distinct network hardware types

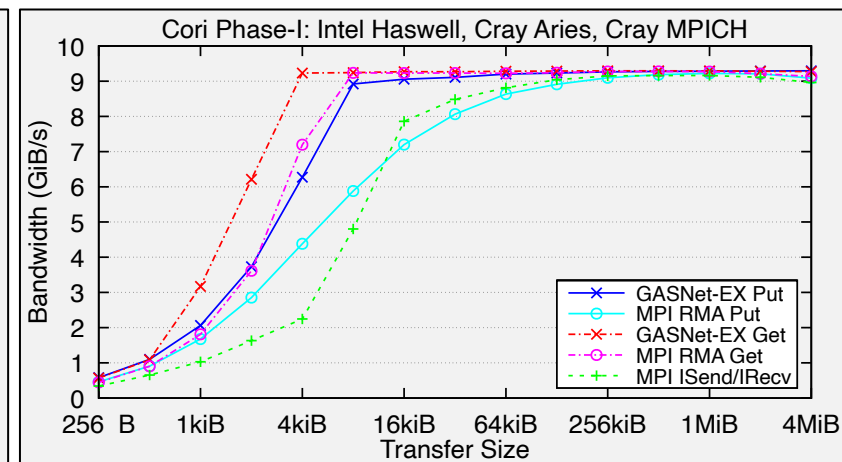
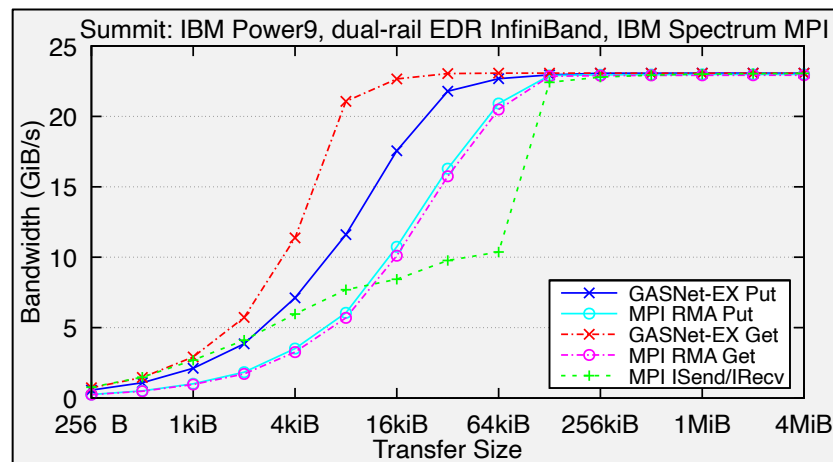
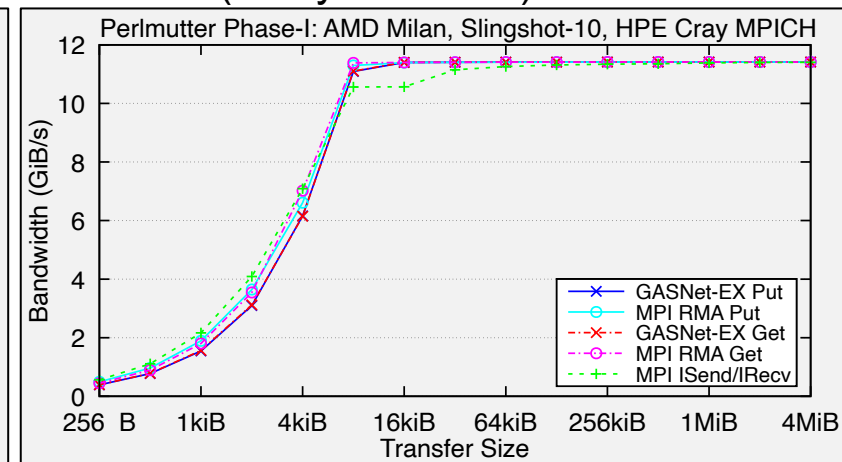
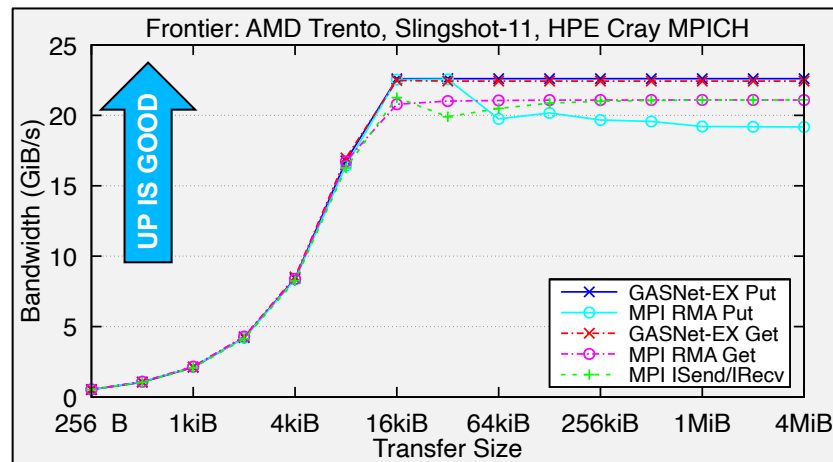
The performance of one-sided GASNet-EX matches or exceeds that of MPI RMA and message-passing:

- 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 $\frac{1}{2}$ or $\frac{1}{4}$ 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

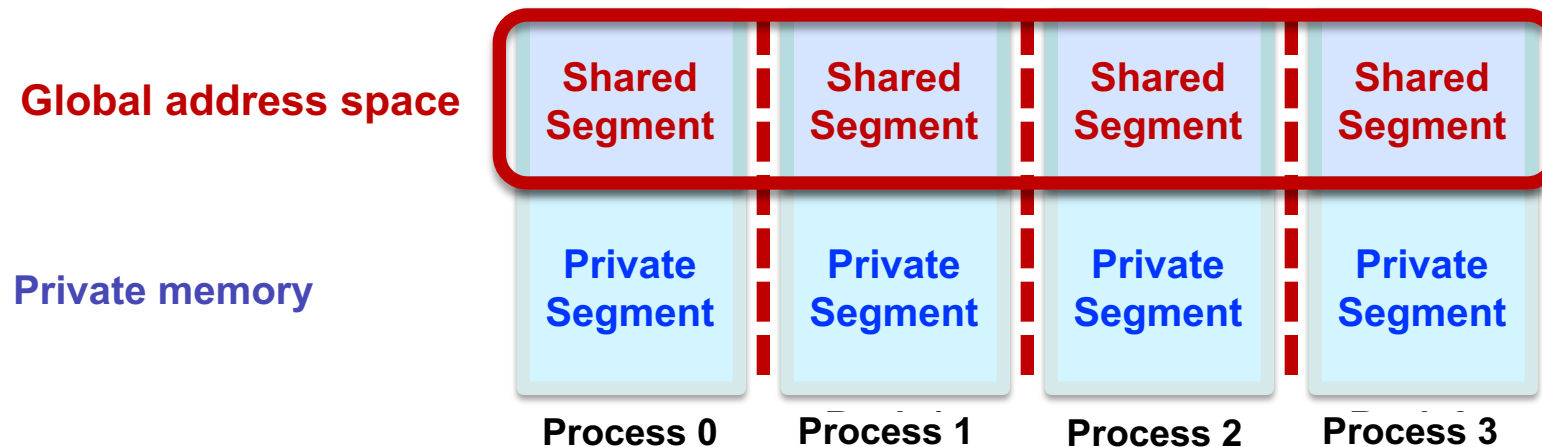
A Partitioned Global Address Space programming model

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

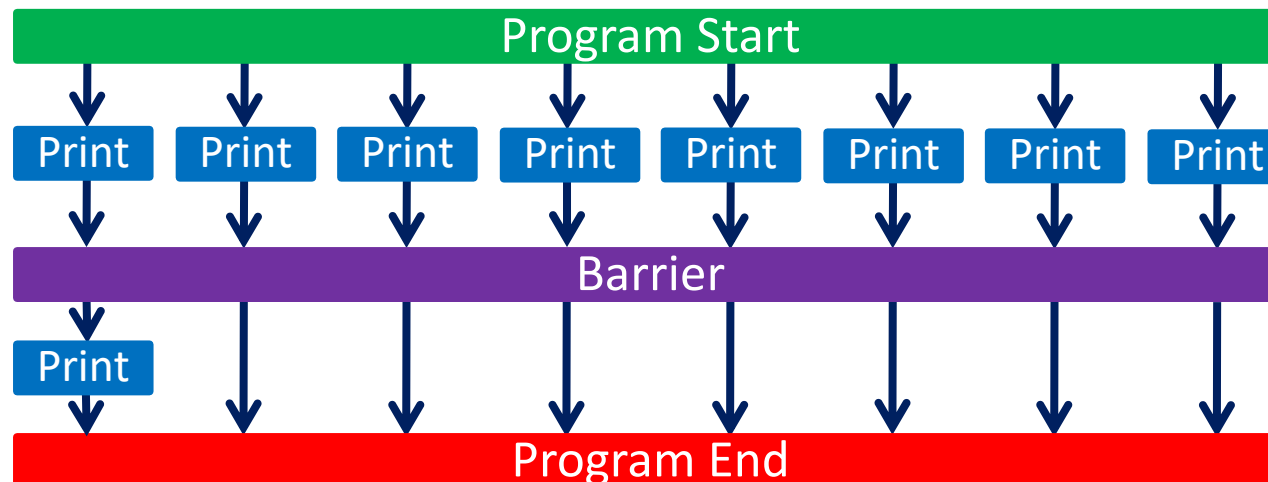
Libraries that provide PGAS: OpenSHMEM, Co-Array C++, Global Arrays, DASH, MPI-RMA

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

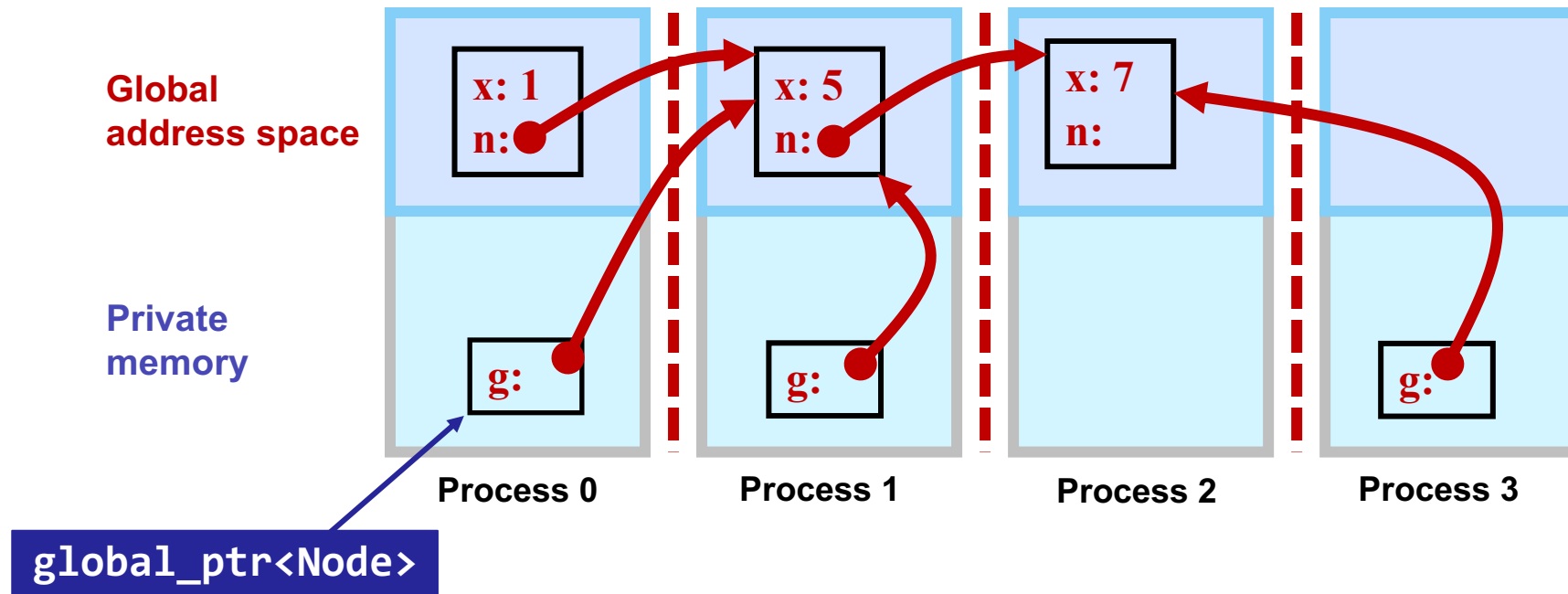
```
int main() {  
    upcxx::init();  
    cout << "Hello from " << upcxx::rank_me() << endl;  
    upcxx::barrier();  
    if (upcxx::rank_me() == 0) cout << "Done." << endl;  
    upcxx::finalize();  
}
```



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>`

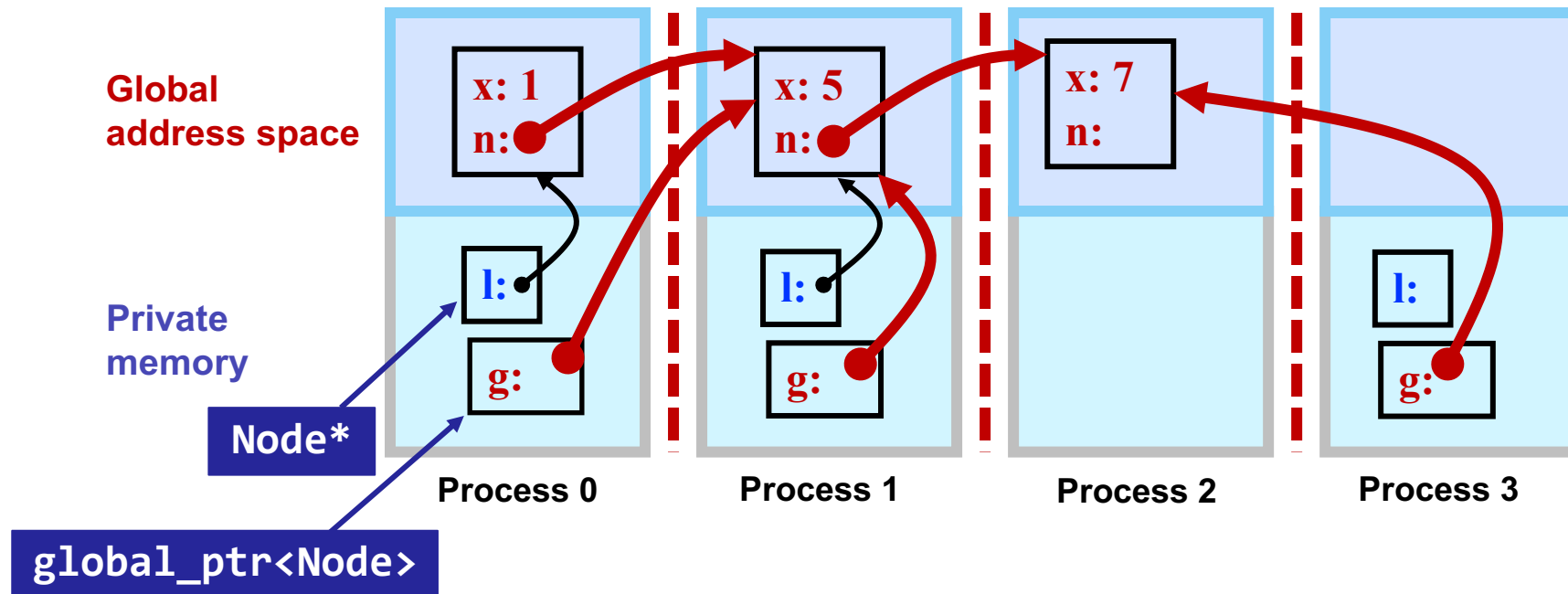


Global vs raw pointers and affinity

The affinity identifies the process that created the object

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

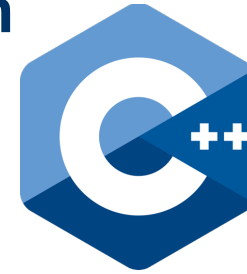
Raw C++ pointers (e.g. `Node*`) can be used on a process to refer to objects in the global address space that have affinity to that process



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)

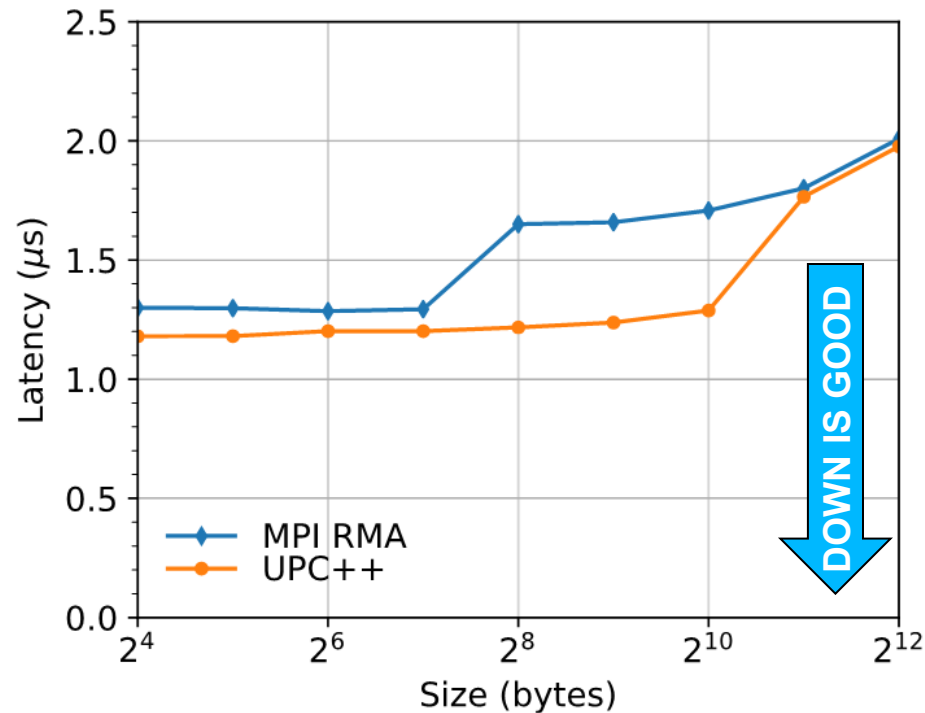
Designed for interoperability

- 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:

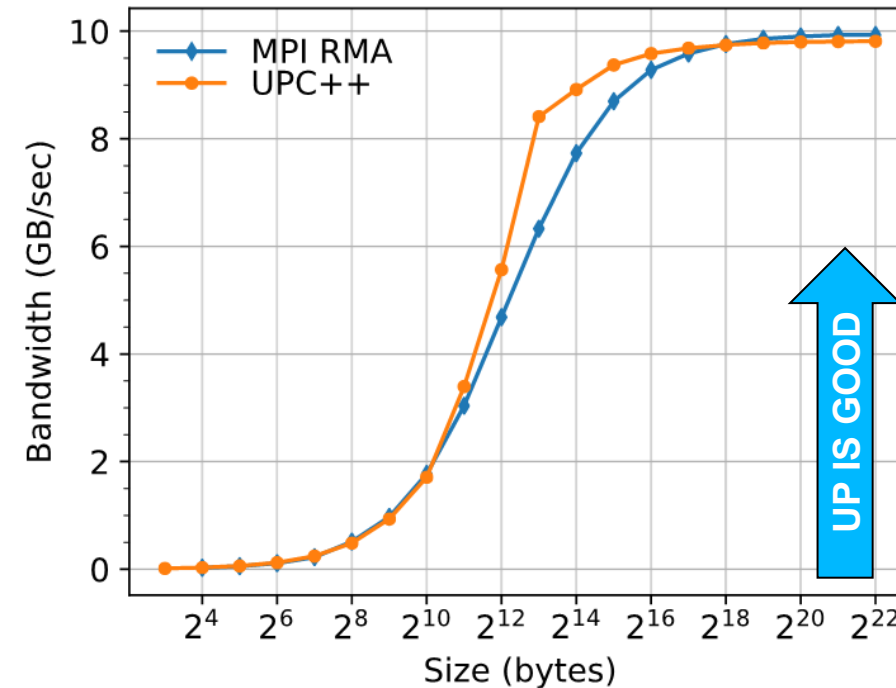
- Cray XC40 system



Round-trip Put Latency (lower is better)

Two processor partitions:

- Intel Haswell (2 x 16 cores per node)
- Intel KNL (1 x 68 cores per node)



Flood Put Bandwidth (higher is better)

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

Asynchronous communication (RMA)

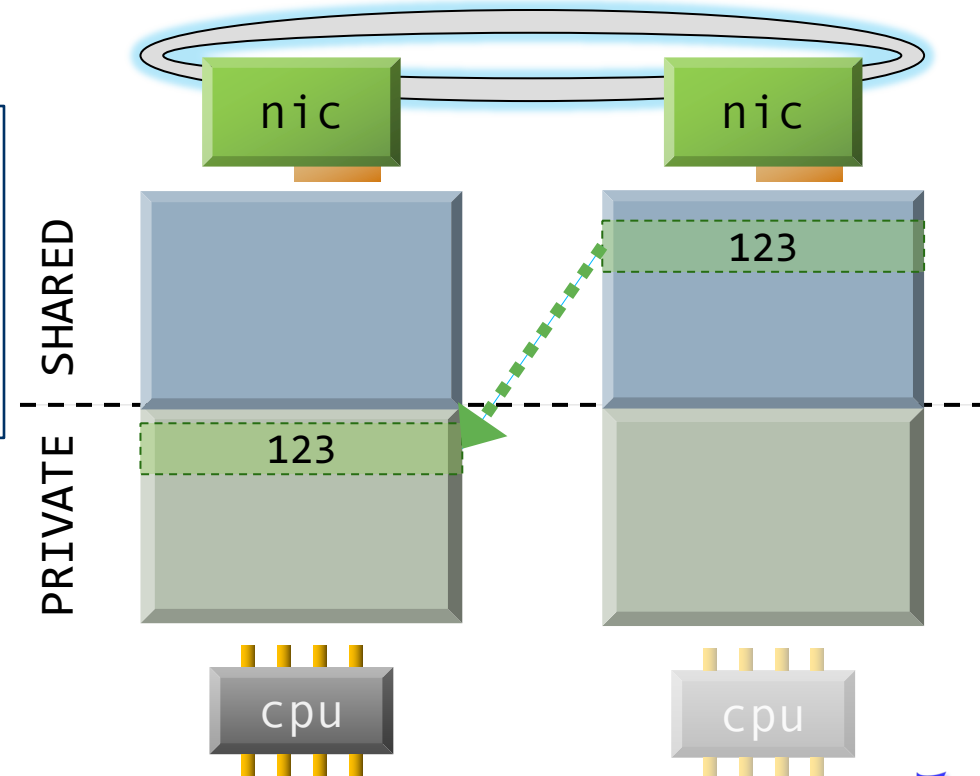
By default, all communication operations are split-phased

- **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

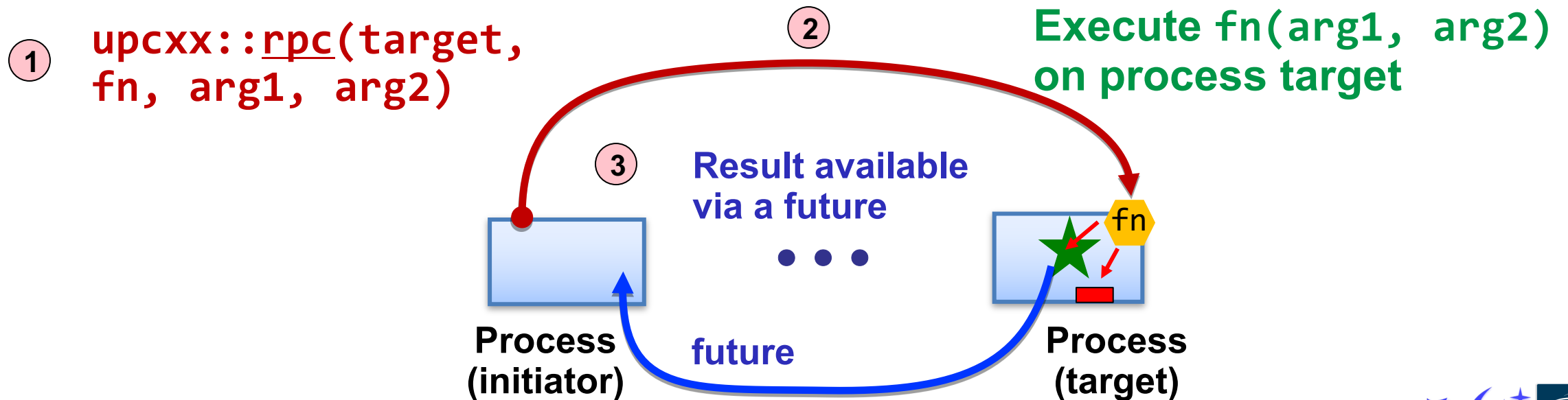


Remote procedure call (RPC)

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

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

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:

```
$ make run-heat2d
```

Command to run
in the terminal

```
upcxx heat2d.cpp -Wall -o heat2d
```

```
upcxx-run -N 1 -n 4 ./heat2d
```

Copy this and add arguments to change the
problem size, e.g.:

```
upcxx-run -N 1 -n 4 ./heat2d 8192 8192
```

```
[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
```