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

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)

<table>
<thead>
<tr>
<th>C/C++</th>
<th>Fortran</th>
<th>Chapel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>Java</td>
<td>R</td>
</tr>
<tr>
<td>Perl</td>
<td>Haskell, Scala, ...</td>
<td>Other</td>
</tr>
</tbody>
</table>
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.

Conceptual global address space

- 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](http://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](http://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](http://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

- Specifically a 2D heat diffusion problem
  - 2D diffusion equation is above. Mathematical details: [wikipedia.org/wiki/Heat_equation](https://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

\[
\begin{align*}
    u_{i,j}^{n+1} &= u_{i,j}^n + \frac{\nu \Delta t}{\Delta x^2} (u_{i+1,j}^n - 2u_{i,j}^n + u_{i-1,j}^n) \\
    &+ \frac{\nu \Delta t}{\Delta y^2} (u_{i,j+1}^n - 2u_{i,j}^n + u_{i,j-1}^n)
\end{align*}
\]

Simplified form

Assume \( \Delta x = \Delta y \), and let \( \alpha = \frac{\nu \Delta t}{\Delta x^2} \)

\[
    u_{i,j}^{n+1} = u_{i,j}^n + \alpha \left( u_{i+1,j}^n + u_{i-1,j}^n - 4u_{i,j}^n + u_{i,j+1}^n + u_{i,j-1}^n \right)
\]
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
What do you want to learn about Chapel, UPC++, or Coarray Fortran today?
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
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 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
Laurendeau, Bourgault-Côté, Parenteau, Plante, et al.
École Polytechnique Montréal

ChplUltra: Simulating Ultralight Dark Matter
Nikhil Padmanabhan, J. Luna Zagorac, et al.
Yale University et al.

Arkouda: Interactive Data Science at Massive Scale
Mike Merrill, Bill Reus, et al.
U.S. DoD

ChOp: Chapel-based Optimization
INRIA, IMEC, et al.

Chapel-based Hydrological Model Calibration
Marjan Asgari et al.
University of Guelph

Chapel-based Hydrological Model Calibration
Nelson Luis Dias
The Federal University of Paraná, Brazil

Lattice-Symmetries: a Quantum Many-Body Toolbox
Tom Westerhout
Radboud University

CrayAI HyperParameter Optimization (HPO)
Ben Albrecht et al.
Cray Inc. / HPE

ChapQG: Layered Quasigeostrophic CFD
Ian Grooms and Scott Bachman
University of Colorado, Boulder et al.

RapidQ: Mapping Coral Biodiversity
Rebecca Green, Helen Fox, Scott Bachman, et al.
The Coral Reef Alliance

CHIUW 2021
CHIUW 2022
CHIUW 2020
CHIUW 2023

(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](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’

Locales array:

- locale 0
- locale 1
- locale 2
- locale 3

User’s code starts running as a single task on locale 0
const numTasks = here.numPUs();

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

coforall tid in 1..numTasks do
  printf("Hello from task %n of %n on %s\n", tid, numTasks, here.name);
TASK-PARALLEL “HELLO WORLD”

```chpl
const numTasks = here.numPUs();
csofarall tid in 1..numTasks do
    printf("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”

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

So far, this is a shared-memory program

Nothing refers to remote locales, explicitly or implicitly
```chapel
coforall loc in Locales {
  on loc {
    const numTasks = here.maxTaskPar;
    coforall tid in 1..numTasks do
      printf("Hello from task %n of %n on %s\n", tid, numTasks, here.name);
  }
}
```

**Locales array:**

-Locale 0
-Locale 1
-Locale 2
-Locale 3
**TASK-PARALLEL “HELLO WORLD” (DISTRIBUTED VERSION)**

```chpl
hello-dist-node-names.chpl

coforall loc in Locales {
  on loc {
    const numTasks = here.numPUs();
    coforall tid in 1..numTasks do
      printf("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 n1034
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 '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

See [https://go.lbl.gov/cuf23-repo](https://go.lbl.gov/cuf23-repo) for more info and for example code.
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 \( \Delta x = \Delta y \), 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+1, j] + 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

Fixed boundary values

\[ u^n \] Stored in un

\[ u^{n+1} \] Stored in u
• Declaring 'u' and 'un' arrays

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

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

```chapel
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
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

Weather & Climate
- Intermediate Complexity Atmospheric Research (ICAR) Model
  Courtesy of Ethan Gutmann, NCAR

Nuclear Energy
- U.S. Nuclear Regulatory Commission File Photo

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


Seminal paper:

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

```fortran
1  program main
2    implicit none
3    print *, "Hello from image ", this_image(), "of", num_images()
4  end program
```
Compiling and Running `hi.f90`

cuf23-tutorial:
SPMD Execution Sequence

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)

Coarrays:

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

```fortran
program main
  !! One-sided communication of distributed greetings
  implicit none
  integer, parameter :: max_greeting_length=64, writer = 1
  integer image
  character(len=max_greeting_length) :: greeting[*,*] ! scalar coarray
  associate(me => this_image(), ni=>num_images())
    write(greeting,*) "Hello from image",me,"of",ni ! local (no "[]")
    sync all ! image control
    if (me == writer) then
      do image = 1, ni
        print *, greeting[image] ! one-sided communication: "get"
      end do
    end if
  end associate
end program
```
Compiling & Running `hello.f90`

cuf23-tutorial:  
Compiling and Running the Heat Equation Solver
Heat Equation

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

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

\[ T = T + dt \star \text{alpha} \star \text{.laplacian.} \ T \]
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

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

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

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
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.
Acknowledgements

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


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

Irregular and unpredictable data movement:

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

Summit: IBM Power9, dual-rail EDR InfiniBand, IBM Spectrum MPI

Perlmutter Phase-I: AMD Milan, Slingshot-10, HPE Cray MPICH

Cori Phase-I: Intel Haswell, Cray Aries, Cray MPICH

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 Global Address Space

- 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.

```cpp
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

```cpp
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 \textit{target} process
2. Target process executes $\text{fn}(\text{arg1, arg2})$ 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

```
upcxx::rpc(target, fn, arg1, arg2)
```

Execute $\text{fn}(\text{arg1, arg2})$ on process target

Result available via a future

Process (initiator)

Process (target)
Hands-on: 2D heat diffusion

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:

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

Command to run in the terminal

```
$ make run-heat2d
upcxx heat2d.cpp -Wall -o heat2d
upcxx-run -N 1 -n 4 ./heat2d
```

My Neighbors: (1, 3)     My Domain: (2048,3072)
My Neighbors: (2, -1)     My Domain: (3072,4096)
My Neighbors: (-1, 1)     My Domain: (0,1024)
My Neighbors: (0, 2)     My Domain: (1024,2048)
[0] mean temperature=1.06256 | Solve time: 0.734826 seconds

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

Check out the chapel-quickReference.pdf in the cuf23/chapel/ subdirectory
SERIAL CODE USING MAP/DICTIONARY: K-MER COUNTING

```
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

• 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 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

```chapel
cforall loc in Locales do on loc { /* ... */ }
cforall 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
Laurendeau, Bourgault-Côté, Parenteau, Plante, et al.
École Polytechnique Montréal

CHplUltra: Simulating Ultralight Dark Matter
Nikhil Padmanabhan, J. Luna Zagorac, et al.
Yale University et al.

Arkouda: Interactive Data Science at Massive Scale
Mike Merrill, Bill Reus, et al.
U.S. DoD

ChAp: Chapel-based Optimization
INRIA, IMEC, et al.

Chapel-based Hydrological Model Calibration
Marjan Asgari et al.
University of Guelph

ChapQG: Layered Quasigeostrophic CFD
Ian Grooms and Scott Bachman
University of Colorado, Boulder et al.

Nelson Luis Dias
The Federal University of Paraná, Brazil

RapidQ: Mapping Coral Biodiversity
Rebecca Green, Helen Fox, Scott Bachman, et al.
The Coral Reef Alliance

CHIUW 2021
CHIUW 2022
CHIUW 2020
CHIUW 2023

(images provided by their respective teams and used with permission)
## USE OF PARALLELISM IN SOME APPLICATIONS AND BENCHMARKS

<table>
<thead>
<tr>
<th>Application</th>
<th>Distributed 'coforall'</th>
<th>Threaded 'coforall'</th>
<th>Asynchronous 'begin'</th>
<th>'cobegin'</th>
<th>sync or atomic vars</th>
<th>subprocesses</th>
<th>forall</th>
<th>scan</th>
</tr>
</thead>
<tbody>
<tr>
<td>HPO</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arkouda</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHAMPS</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ChOp</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ParFlow</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coral Reef</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task Graph</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In this tutorial will be working with examples of parallelism from the yellow highlighted columns.
PARALLELISM ACROSS LOCALES AND WITHIN LOCALES

- **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 '//'

```
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, ")");
    }
}
```
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.
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

```chapel
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

Locale 0

Locale 1

Locale 2

Locale 3
BASIC FEATURES FOR LOCALITY

```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**

```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**

- **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
  ```

```bash
// 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:

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

• This is a distributed, but serial program:

```chapel
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:

```chapel
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

• 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

```chapl
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
    ...
}
```

- 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

```
prompt> chpl --fast parfilekmer.chpl
prompt> ./parfilekmer -nl 1
prompt> ./parfilekmer -nl 4
```
**BLOCK DISTRIBUTION OF ARRAY OF STRINGS**

<table>
<thead>
<tr>
<th>Locale 0</th>
<th>Locale 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;filename1&quot;</td>
<td>&quot;filename2&quot;</td>
</tr>
</tbody>
</table>

- 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

```
prompt> chpl --fast parfilekmer.chpl
prompt> ./parfilekmer -nl 2
```
PROCESSING FILES IN PARALLEL

• 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 GLOBALNAMESPACE WITH PUTS AND GETS

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

```chapel
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

- 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
  ```

make run-heat_2D
make run-heat_2D_dist
make run-heat_2D_buffers
ARRAY-BASED PARALLELISM AND LOCALITY

Chapel also supports distributed domains (index sets) and arrays.

They also result in parallel distributed computation.

basics-distarr.chpl

```chapel
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;
```
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 \( \Delta x = \Delta y \), 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
• Declaring 'u' array

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

• Declaring 'u' array as distributed

```chpl
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

\[ u^n \text{ Stored in } u \]
\[ u^{n+1} \text{ Stored in } u \]
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

(1) write to neighbor halo
(2) copy into local halo
(3) compute next u in parallel locally
**HALO BUFFER OPTIMIZATION CODE**

```plaintext
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

• 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_distBuffers.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_distBuffers.chpl' shows an optimization that explicitly copies subarrays into buffers

make run-heat_2D
make run-heat_2D_dist
make run-heat_2D_dist_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
• **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
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**

- **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](https://chapel-lang.org/docs/technotes/gpu.html)

```chapel
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());
```
This program uses all CPUs and GPUs across all of your compute nodes.

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

One task runs our multi-GPU triad.

The other runs the multi-CPU triad.
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-locals* in Chapel
STREAM TRIAD: DISTRIBUTED MEMORY, CPUS ONLY

These programs are both CPU-only

Nothing refers to GPUs, explicitly or implicitly
STREAM TRIAD: DISTRIBUTED MEMORY, GPUS ONLY

```chapel
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.
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
• 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](https://chapel-lang.org)
- (points to all other resources)

**Social Media:**
- Twitter: [@ChapelLanguage](https://twitter.com/ChapelLanguage)
- Facebook: [@ChapelLanguage](https://facebook.com/ChapelLanguage)
- YouTube: [http://www.youtube.com/c/ChapelParallelProgrammingLanguage](http://www.youtube.com/c/ChapelParallelProgrammingLanguage)

**Community Discussion / Support:**
- Discourse: [https://chapel.discourse.group/](https://chapel.discourse.group/)
- Gitter: [https://gitter.im/chapel-lang/chapel](https://gitter.im/chapel-lang/chapel)
- Stack Overflow: [https://stackoverflow.com/questions/tagged/chapel](https://stackoverflow.com/questions/tagged/chapel)
- GitHub Issues: [https://github.com/chapel-lang/chapel/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/](https://chapel-lang.org/docs/)
  - The primers can be particularly valuable for learning a concept: [https://chapel-lang.org/docs/primers/index.html](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

```plaintext
begin stmt; // fire off an asynchronous task to run ‘stmt’
```

```plaintext
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

```plaintext
var sum: atomic int; // supports various atomic methods like .add(), .compareExchangeO, …
var cursor: sync int; // stores a full/empty bit governing reads/writes, supporting .readEFO, .writeEFO
```

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

```plaintext
coforall i in 1..nites 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

```chapel
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

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

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

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

is equivalent to:

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

as is:

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

• So, in the Jacobi computation,

```java
abs(A[D] - Temp[D]); == forall (a,t) in zip(A[D], Temp[D]) do abs(a - t);
```
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 \( \text{fn}(\text{arg1}, \text{arg2}) \) 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

\[
\text{upcxx::rpc} \left( \text{target, fn, arg1, arg2} \right)
\]

Execute \( \text{fn}(\text{arg1}, \text{arg2}) \) on process target

Result available via a future
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](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-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

Kamil / UPC++ / CUF23 Tutorial / upcxx.lbl.gov
Example: 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();
}

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)

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

```c++
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.

```cpp
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

```cpp
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

```cpp
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
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.

```cpp
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):

- **Process 0**: 12 1 2 3 4 5
- **Process 1**: 4 5 6 7 8 9
- **Process 2**: 8 9 10 11 12 1

**Local grid size**

**Ghost cells**

**Periodic boundary**
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.

Ghost cells

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 1 2 3 4 5</td>
<td>4 5 6 7 8 9</td>
<td>8 9 10 11 12 1</td>
</tr>
</tbody>
</table>
Jacobi computation (version 1)

We can use RPC to communicate boundary cells

```cpp
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[i] = 0.25 *
        (old_grid[i-1] + 2*old_grid[i] + old_grid[i+1]);

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

new_grid[N-2] = 0.25 *

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

<table>
<thead>
<tr>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 5 6 7 8 9</td>
</tr>
</tbody>
</table>

Initiate communication

Do interior computation

Wait for communication to complete and do boundary computation
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

```cpp
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)
    /* ... */;


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:

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

• Vector put and get:

```cpp
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:

```c
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

```cpp
std::vector<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()`

```cpp
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 \texttt{rget()}

\begin{verbatim}
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)
    /* ... */;
left_get.wait();
right_get.wait();
\end{verbatim}
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

```cpp
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()`

```cpp
    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:

```plaintext
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();
```

Kamil / UPC++ / CUF23 Tutorial / upcxx.lbl.gov
Jacobi RMA with puts and conjoining

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

\[
\text{future<> puts = when\_all(}
\begin{align*}
& \text{rput(old\_grid[1], left\_old\_grid + N - 1),} \\
& \text{rput(old\_grid[N-2], right\_old\_grid));}
\end{align*}
\]

\[
\text{for (long i = 2; i < N - 2; ++i)}
\]
/* ... */

\[
\text{puts.wait();}
\]
\[
\text{barrier();}
\]

Ensure outgoing puts have completed
Ensure incoming puts have completed

\[
\]
\[
\]
2D heat diffusion data layout

Fixed boundary values

Global (Abstract) View

Local (Concrete) View

Process 2

Process 1

Process 0

Global (Abstract) View

Local (Concrete) View

"Landing zone" for receiving data from downward neighbor

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

make run-heat2d
2D heat diffusion computation

Computation loop:

```
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();
}
```

\[
u_{i,j}^{n+1} = u_{i,j}^n + \alpha(u_{i+1,j}^n + u_{i-1,j}^n - 4u_{i,j}^n + u_{i,j+1}^n + u_{i,j-1}^n)
\]
Distributed objects

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

\[ \text{dist_object}<T>(T \ value, \ \text{team} \ &\text{team} = \text{world}()); \]

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
Distributed objects in 2D heat diffusion

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

```cpp
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)

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

```
make run-dmap-insert-test
```
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
Comes owner for the given key
DHT insertion

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

```cpp
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);
}
```

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

Send RPC to the process determined by key hash

Key and value passed as arguments to the remote function
DHT find

Find also uses RPC and returns a future

```cpp
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);
}
```

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

Key passed as argument to the remote function

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

Key passed as argument to the remote function

Process 0

Process p
Additional DHT operations

// 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);
}
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
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)
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 at the seasonal fluctuations in a wetland mangrove? (1.6 TB)

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

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

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
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
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
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

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

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

Stephan 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

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.”

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

Application:
- European Centre for Medium Range Weather Forecasts (ECMWF) operational weather forecast model

Programming models studied:
- CAF or PGAS
- Two-sided MPI

Highlights:
- Simulations on > 60K cores
- Performance improvement from switching to CAF peaks at 21% around 40K cores

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

Extended Abstract

Soren Rasmussen, Ethan D Gutmann, Brian Friesen, Damian Rouson, Salvatore Filippone, Irene Moulitsas

1Cranfield University, UK
2National Center for Atmospheric Research, USA
3Lawrence Berkeley National Laboratory, USA
4Sourcery Institute, USA

ABSTRACT

As an application of the Intermediate Complexity Research (ICAR) Model, this paper focuses on the comparison of the cost and performance of the Message Passing Interface (MPI) versus coarrays. Two methods of communication across processes: the application-specific message-passing communication of OpenSHMEM, which is performed with either MPI or OpenSHMEM, 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 studies 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 v1 was released in 1994 and research and development has continued across academia and industry. A method of interfacing Fortran, MPI introduces its concepts to Fortran, which is an enhancement to Fortran that was introduced by Robert W. Pannekoek 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, while MPI introduces a layer of abstraction between the Fortran

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

New Frontiers: T-Cell Motility

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

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

https://go.lbl.gov/inference-engine
Implicitly Parallel Training

iterate_across_batches: &
  do iter = 1, size(mini_batches)
  cost = 0.; dcw = 0.; dcdb = 0.
  associate(input_output_pairs => mini_batches(iter)%input_output_pairs())
  inputs = input_output_pairs%inputs()
  expected_outputs = input_output_pairs%expected_outputs()
  mini_batch_size = size(input_output_pairs)
  end associate

iterate_through_batch: &
  do pair = 1, mini_batch_size
    a(1:input_length,0) = inputs(pair)%values()
    y = expected_outputs(pair)%outputs()
    feed_forward: &
      do l = 1, output_layer
        z(1:n(l),1) = matmul(w(1:n(l),1:n(l-1),1), a(1:n(l-1),1-1)) + b(1:n(l),1)
        a(1:n(l),1) = self%differentiable_activation_strategy%Activation(z(1:n(l),1))
      end do feed_forward
    cost = cost + sum((y(1:n(output_layer)) - a(1:n(output_layer),output_layer)) ** 2)/(2.0e0*mini_batch_size)
    delta(1:n(output_layer),output_layer) &=
      (a(1:n(output_layer),output_layer) - y(1:n(output_layer))) &
      * self%differentiable_activation_strategy%Activation_derivative(z(1:n(output_layer),output_layer))
    back_propagate_error: &
      do l = n_hidden, 1,-1
        delta(1:n(l),1) = matmul(transpose(w(1:n(l-1),1:n(l),1-1)), delta(1:n(l+1),1-1))
        delta(1:n(l),1) = delta(1:n(l),1) * self%differentiable_activation_strategy%Activation_derivative(z(1:n(l),1))
      end do back_propagate_error
    sum_gradients: &
      do l = 1, output_layer
        dcdb(1:n(l),1) = dcdb(1:n(l),1) + delta(1:n(l),1)
      end do concurrent(j = 1:n(l))
      dcwb(j,1:n(l-1),1) = dcwb(j,1:n(l-1),1) + a(1:n(l-1),1-1)*delta(j,1)
    end do
    end do sum_gradients
  end do iterate_through_batch

adjust_weights_and_biases: &
  do l = 1, output_layer
    dcwb(1:n(l),1) = dcwb(1:n(l),1)/mini_batch_size
    b(l) = b(l) - eta*dcdb(1:n(l),1)
  end do
  dcwb(1:n(l),1) = dcwb(1:n(l),1)/mini_batch_size
  w(1:n(l),1:n(l-1),1) = w(1:n(l),1:n(l-1),1) - eta*dcwb(1:n(l),1:n(l-1),1)
end do adjust_weights_and_biases
end do iterate_across_batches
Iterating sequentially across and within mini-batches of input/output pairs facilitates \textit{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.
The only other sequential logic is the (mostly) necessary stepping through layers:

All other logic is implicitly parallel array statements or do concurrent blocks:
FASTGPT: FASTER THAN PYTORCH IN 300 LINES OF FORTRAN

March 14, 2023
Authors: Ondřej Čertík, Brian Beckman

In this blog post I am announcing fastGPT, fast GPT-2 inference written in Fortran. In it, I show

1. Fortran has speed at least as good as default PyTorch on Apple M1 Max.

2. Fortran code has statically typed arrays, making maintenance of the code easier than with Python.

3. It seems that the bottleneck algorithm in GPT-2 inference is matrix-matrix multiplication. For physicists like us, matrix-matrix multiplication is very familiar, unlike other aspects of AI and ML. Finding this familiar ground inspired us to approach GPT-2 like any other numerical computing problem.

4. Fixed an unintentional single-to-double conversion that slowed down the original Python.

5. I am asking others to take over and parallelize fastGPT on CPU and offload to GPU and see how fast you can make it.

About one month ago, I read the blogpost GPT in 60 Lines of NumPy, and it piqued my curiosity. I looked at the corresponding code (picoGPT) and was absolutely amazed, for two reasons. First, I hadn't known it could be so simple to implement the GPT-2 inference. Second, this looks just like a typical computational physics code, similar to many that I have developed and maintained throughout my career.

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

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

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

Legend

- Default communication mechanisms
- Optional communication mechanism
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   associate(my_team=>mod(this_image(),2) + 1)
13
14   form team(my_team,home) ! Map even|odd images->teams 1|2
15   change team(home)
16     call assert(team_number() == my_team)
17   end team
18
19   call assert(team_number() == standard_initial_value)
20   end associate
21
22   sync all
23
24   if (this_image() == 1) print *, "Test passed."
25
26 end program
Image Enumeration

Obtaining an image index:

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

Obtaining an image count:

- `num_images()`
- `num_images(team)`
- `num_images(team_number)`
Image Enumeration

<table>
<thead>
<tr>
<th>b(0)</th>
<th>b(1)</th>
<th>b(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[-1,0]</td>
<td>[0,0]</td>
<td>[1,0]</td>
</tr>
<tr>
<td>[-1,-1]</td>
<td>[0,-1]</td>
<td>[1,-1]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>[1]</td>
<td>[2]</td>
<td>[3]</td>
<td>[4]</td>
<td></td>
</tr>
</tbody>
</table>

```bash
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 *, lcbound(a), ucobound(a)
8 end if
```

cuf23-tutorial: cafrun -n 5 ./image-notation
3
5
5
-1
-1
cuf23-tutorial: ^5^1
cafrun -n 1 ./image-notation
-1
0
0
-1
-1
cuf23-tutorial: 
```
**Synchronization**

Coffee Image barriers (“meet-ups”):

```
sync all(stat, errmsg)
sync images(image-set, stat, errmsg)
allocate()   } for coarrays only, including implicit
deallocate() (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.

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

```
sync memory(stat, errmsg)
```
Other Image Control Statements

**Locks:**

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

**Critical blocks:**

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

**Teams**

```plaintext
form team(team_number, team_variable)
change team(team_value, ...)
end team
```

**Events**

```plaintext
event post(event-variable, stat, errmsg)
event wait(event-variable, stat, errmsg)
```
Collective Subroutines

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)`
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
For Team 1:

- Image 1: \(a(1:4)[1]\), values: 0536
- Image 2: \(a(1:4)[2]\), values: 2651
- Image 3: \(a(1:4)[3]\), values: 3497

\[\text{co_sum}(a)\]

For Team 2:

- Image 4: \(a(1:4)[4]\), values: 3451
- Image 5: \(a(1:4)[5]\), values: 2439
- Image 6: \(a(1:4)[6]\), values: 4401

\[\text{co_sum}(a)\]
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(a)

Team 1

Image 1: a(1:4)[1] = 1536
Image 2: a(1:4)[2] = 2651
Image 3: a(1:4)[3] = 3497

co_max(a)

Image 1: a(1:4)[1] = 3697
Image 2: a(1:4)[2] = 3697
Image 3: a(1:4)[3] = 3697

Team 2

Image 4: a(1:4)[4] = 3451
Image 5: a(1:4)[5] = 2439
Image 6: a(1:4)[6] = 4401

co_max(a)

Image 4: a(1:4)[4] = 4459
Image 5: a(1:4)[5] = 4459
Image 6: a(1:4)[6] = 4459
**co_min**

```
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
<table>
<thead>
<tr>
<th>Team 1</th>
<th>Team 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image 1</td>
<td>Image 4</td>
</tr>
<tr>
<td>$a(1:4)[1]$</td>
<td>$a(1:4)[4]$</td>
</tr>
<tr>
<td>1536</td>
<td>3451</td>
</tr>
<tr>
<td>Image 2</td>
<td>Image 5</td>
</tr>
<tr>
<td>$a(1:4)[2]$</td>
<td>$a(1:4)[5]$</td>
</tr>
<tr>
<td>2651</td>
<td>2439</td>
</tr>
<tr>
<td>Image 3</td>
<td>Image 6</td>
</tr>
<tr>
<td>$a(1:4)[3]$</td>
<td>$a(1:4)[6]$</td>
</tr>
<tr>
<td>3497</td>
<td>4401</td>
</tr>
</tbody>
</table>

$\text{co\_min}(a)$
**co_broadcast**

```
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

Team 1

Image 1: \(a(1:4)[1]\) 
\[1 5 3 6\]
Image 2: \(a(1:4)[2]\) 
\[2 6 5 1\]
Image 3: \(a(1:4)[3]\) 
\[3 4 9 7\]

co_broadcast\((a, 1)\)

Team 2

Image 4: \(a(1:4)[4]\) 
\[3 4 5 1\]
Image 5: \(a(1:4)[5]\) 
\[2 4 3 9\]
Image 6: \(a(1:4)[6]\) 
\[4 4 0 1\]

co_broadcast\((a, 1)\)
co_reduce

\[
\text{co\_reduce}(a, \text{operation}, \text{result\_image}, \text{stat}, \text{errmsg})
\]

**Argument** \( a \)

— shall be \text{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 \text{operation} to values of \( a \) in the corresponding references, and likewise on an element-wise basis if \( a \) is an array

**Argument** \( \text{operation} \)

— shall implement an associative operation via a \text{pure} function with two arguments

**Argument** \( \text{result\_image} \)

— shall be of scalar \text{integer}, \text{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

module co_all_m
  implicit none
  interface
    module subroutine co_all(a)
      implicit none
      logical, intent(inout) :: a
    end subroutine
  end interface
end module

submodule(co_all_m) co_all_s
  implicit none
  contains
    module procedure co_all
      call co_reduce(a, and)
    end procedure
  contains
    pure function and(lhs, rhs) result(lhs_and_rhs)
      logical, intent(in) :: lhs, rhs
      logical lhs_and_rhs
      lhs_and_rhs = lhs .and. rhs
    end function
end submodule

program main
  use co_all_m, only : co_all
  implicit none
  logical :: operand = .true.
  associate(me=>this_image())
    call co_all(operand)
    if (me==1) print *, operand
    if (me==num_images()) operand = .false.
    call co_all(operand)
    if (me==1) print *, operand
  end associate
end program

https://github.com/sourceryinstitute/sourcery
program heat_equation

!! Parallel finite difference solver for the 2D, unsteady heat conduction partial differential equation
use subdomain_2D_m, only : subdomain_2D_t
use iso_fortran_env, only : int64
use kind_parameters_m, only : rkind
implicit none

type(subdomain_2D_t) T
integer, parameter :: nx = 4096, ny = nx, steps = 50
real(rkind), parameter :: alpha = 1._rkind
real(rkind) T_sum
integer(int64) t_start, t_finish, clock_rate
integer step

call T%define(side=1._rkind, boundary_val=1._rkind, internal_val=2._rkind, n=nx)! Initial/boundary cond.
call T%allocate_halo_coarray ! Implicit synchronization

associate(dt => T%dx() * T%dy() / (4 * alpha)) ! set time step

call system_clock(t_start)

do step = 1, steps
   call T%exchange_halo ! put subdomain boundary values on neighboring images
   sync all
   T = T + dt * alpha * .laplacian. T ! asynchronous parallel user-defined operators
   sync all
end do

end associate

T_sum = sum(T%values()) ! local sum
call co_sum(T_sum, result_image=1) ! distributed collective sum

call system_clock(t_finish, clock_rate)

if (this_image() == 1) then
   print *, "walltime: ", real(t_finish - t_start, rkind) / real(clock_rate, rkind)
   print *, "T_avg = ", T_sum / (nx*ny)
end if

end program
Heat Equation Exercise

In addition to demonstrating parallel features of Fortran 2018, this example shows an object-oriented, functional programming style based on Fortran's user-defined operators such as the .laplacian. operator defined in this example. To demonstrate the expressive power and flexibility of this approach, try modifying the main program to use 2nd-order Runge-Kutta time advancement:

```fortran
T_half = T + 0.5*dt*alpha* .laplacian. T
call T%exchange_halo
sync all
T = T + dt*alpha* .laplacian. T_half
call T%exchange_halo
sync all
```

You'll need to append , T_half to the declaration type(subdomain_2D_t) T. With some care, you could modify the main program to use any desired order of Runge-Kutta algorithm without changing any of the supporting code.

This example also demonstrates a benefit of Fortran's facility for declaring a procedure to be pure: the semantics of pure procedures essentially guarantees that the above right-hand-side expressions can be evaluated fully asynchronously across all images. No operator can modify state that would be observable by another operator other than via the first operator's result. This would be true even if an operator executing on one image performs communication to get data from another image via a coarray. To reduce communication waiting times, however, each image in our example proactively puts data onto neighboring images. Puts generally outperform gets because the data can be shipped off as soon the data are ready. With the exception of one coarray allocation in the define procedure, all procedures are asynchronous and all image control is exposed in the main program.
Coarrays

Non-allocatable (static):

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

Dynamically allocatable:

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

Derived type components:

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

Local coarrays:

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

Derived type coarrays:

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

```fortran
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

User-defined, purely functional operators

\[ u_t = -\frac{\text{grad}.p}{\rho} + \nu \text{laplacian}.u - (u \cdot \text{grad}.u) \]

Distributed objects

\[ \bar{u}_t = -\frac{1}{\rho} \nabla p + \nu \nabla^2 \bar{u} - \bar{u} \cdot \nabla \bar{u} \]


Platform: Cray XE6 (Hopper at NERSC)
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.
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`.

<table>
<thead>
<tr>
<th>Image Control</th>
<th>Side Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>event_post</code></td>
<td><code>atomic_add 1</code></td>
</tr>
<tr>
<td><code>event_query</code></td>
<td>defines count</td>
</tr>
<tr>
<td><code>event_wait</code></td>
<td><code>atomic_add -1</code></td>
</tr>
</tbody>
</table>
Asynchronous Hello World Exercise

Try adjusting the `delay_magnitude` constant to larger or smaller non-negative values. For each new value, recompile once and rerun the program multiple times. Explain the resulting program output.
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 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 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
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