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

Dr. Michelle Mills Strout

Dr. Damian Rouson

Dr. Amir Kamil
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](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>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++</td>
</tr>
<tr>
<td>Fortran</td>
</tr>
<tr>
<td>Chapel</td>
</tr>
<tr>
<td>Python</td>
</tr>
<tr>
<td>Java</td>
</tr>
<tr>
<td>R</td>
</tr>
<tr>
<td>Perl</td>
</tr>
<tr>
<td>Haskell, Scala, ...</td>
</tr>
<tr>
<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](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](go.lbl.gov/cuf23-slack)
    - You should have received an email invite, or can follow the link above
Production Applications using these Programming Models

MetaHipMer, a genome assembler written in UPC++

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
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
  - 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

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

Simplified form

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

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

Top
Schedule for Chapel, UPC++ and Fortran Tutorial

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

● noon - 1:30: Tutorial Overview, 20-minute intro to each programming model
  o Chapel Intro
  o Fortran with co-arrays Intro
  o 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)

CHAMS: 3D Unstructured CFD

Arkouda: Interactive Data Science at Massive Scale

ChOp: Chapel-based Optimization

ChpiUltra: Simulating Ultralight Dark Matter

Lattice-Symmetries: a Quantum Many-Body Toolbox


RapidQ: Mapping Coral Biodiversity

ChapQG: Layered Quasigeostrophic CFD

Chapel-based Hydrological Model Calibration

CrayAI HyperParameter Optimization (HPO)

CHGL: Chapel Hypergraph Library

Your Application Here?

(images provided by their respective teams and used with permission)
HIGHLIGHTS OF CHAPEL USAGE

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

Arkouda: data analytics framework (https://github.com/Bears-R-Us/arkouda)
- Mike Merrill, Bill Reus, et al., US DOD
- Python front end client, Chapel server that processes dozens of terabytes in seconds
- April 2023: 1200 GiB/s for argsort on an HPE EX system

Recent Journal Paper on using Chapel for calibrating hydrologic models
- Marjan Asgari et al, "Development of a knowledge-sharing parallel computing approach for calibrating distributed watershed hydrologic models", Environmental Modeling and Software.
- They report super-linear speedup
ARKOUDA ARGSORT PERFORMANCE

HPE Apollo (May 2021)
- HDR-100 Infiniband network (100 Gb/s)
- 576 compute nodes
- 72 TiB of 8-byte values
- ~480 GiB/s (~150 seconds)

HPE Cray EX (April 2023)
- Slingshot-11 network (200 Gb/s)
- 896 compute nodes
- 28 TiB of 8-byte values
- ~1200 GiB/s (~24 seconds)

HPE Cray EX (May 2023)
- Slingshot-11 network (200 Gb/s)
- 8192 compute nodes
- 256 TiB of 8-byte values
- ~8500 GiB/s (~31 seconds)

A notable performance achievement in ~100 lines of Chapel
INTRODUCTION TO CHAPEL

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

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

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

• Learning objectives for today's 90-minute Chapel tutorial
CHAPEL EXECUTION MODEL AND TERMINOLOGY: LOCALES

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

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

Locales array:

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

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

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

> chpl hello-dist-node-names.chpl 
> ./hello-dist-node-names 
Hello from task 1 of 4 on n1032 
Hello from task 4 of 4 on n1032 
Hello from task 3 of 4 on n1032 
Hello from task 2 of 4 on n1032
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
Task-Parallel “Hello World” (Distributed Version)

```
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
", tid, numTasks, here.name);
    }
}
```

Locales array:

- Locale 0
- Locale 1
- Locale 2
- Locale 3

The array of locales we're running on (introduced a few slides back).
**TASK-PARALLEL “HELLO WORLD” (DISTRIBUTED VERSION)**

```chapl
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**

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

- All updates in a timestep can be done in parallel

```chpl
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
• Declaring 'u' and 'un' arrays

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

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

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

• Reads that cross the distribution boundary will result in a remote get
PARALLELISM SUPPORTED BY CHAPEL

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

- **Asynchronous parallelism**
  - 'begin', creates an asynchronous task
  - 'sync' and 'atomic' vars for task coordination
  - spawning subprocesses

- **Higher-level parallelism abstractions**
  - 'forall', data parallelism and iterator abstraction
  - 'foreach', SIMD parallelism
  - 'scan', operations such as cumulative sums
  - 'reduce', operations such as summation
LEARNING OBJECTIVES FOR TODAY’S CHAPEL TUTORIAL

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

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

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

Nuclear Energy

Aerospace
Why Fortran Matters

- Weather & Climate
- Nuclear Energy
- Aerospace
“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
```
Compilation and Running `hi.f90`

```
cuf23-tutorial:  
```
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.

Image 1
```
1  program main
2    implicit none
3    print *, 'Hello from image ', this_image(), ' of ', num_images()
4  end program
```
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.
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.
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.
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.
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.
1. After the creation of a fixed number of images, each image’s first “segment” (sequence of statements) executes.
2. Image control statements totally order segments executed by a single image and partially order segments executed by separate images.
SPMD Execution Sequence

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

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
1 program main
2   !! One-sided communication of distributed greetings
3   implicit none
4   integer, parameter :: max_greeting_length=64, writer = 1
5   integer image
6   character(len=max_greeting_length) :: greeting[*] ! scalar coarray
7
8   associate(me => this_image(), ni=>num_images())
9
10  write(greeting,*), "Hello from image", me, "of", ni ! local (no "["])"  
11  sync all ! image control
12
13  if (me == writer) then
14    do image = 1, ni
15      print *, greeting[image] ! one-sided communication: "get"
16    end do
17  end if
18
19  end associate
20 end program
```
Compiling & Running `hello.f90`

cuf23-tutorial:
Compiling & Running `hello.f90`

cuf23-tutorial:
Compiling and Running the Heat Equation Solver

cuf23-tutorial:
Compiling and Running the Heat Equation Solver

cuf23-tutorial:
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

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

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

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

local objects

cd fortran
make run-heat-equation
Heat Equation

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

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

local objects

pure user-defined operators

cd fortran
make run-heat-equation

T = T + dt * alpha * .laplacian. T
### Class Diagram

<table>
<thead>
<tr>
<th>C</th>
<th>subdomain_2D_t</th>
</tr>
</thead>
<tbody>
<tr>
<td>s_</td>
<td>real[]</td>
</tr>
<tr>
<td>define()</td>
<td></td>
</tr>
<tr>
<td>laplacian(rhs: subdomain_2D_t) : subdomain_2D_t</td>
<td></td>
</tr>
<tr>
<td>multiply(lhs : subdomain_2D_t, rhs : subdomain_2D_t) : subdomain_2D_t</td>
<td></td>
</tr>
<tr>
<td>add(lhs : subdomain_2D_t, rhs : subdomain_2D_t) : subdomain_2D_t</td>
<td></td>
</tr>
<tr>
<td>copy(lhs : subdomain_2D_t, rhs : subdomain_2D_t)</td>
<td></td>
</tr>
<tr>
<td>dx()</td>
<td></td>
</tr>
<tr>
<td>dy()</td>
<td></td>
</tr>
<tr>
<td>values()</td>
<td></td>
</tr>
<tr>
<td>exchange_halo()</td>
<td></td>
</tr>
<tr>
<td>allocate_halo_coarray()</td>
<td></td>
</tr>
</tbody>
</table>
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
Loop-Level Parallelism

188 do concurrent(j=2:ny-1)
189    laplacian_rhs%s_(i, j) = 
      (halo_left(j) - 2*rhs%s_(i, j) + rhs%s_(i+1,j))/dx_**2 + 
      (rhs%s_(i, j-1) - 2*rhs%s_(i, j) + rhs%s_(i ,j+1))/dy_**2
191 end do
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.
UPC++: An Asynchronous RMA/RPC Library for Distributed C++ Applications

Amir Kamil

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

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

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


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

Seismo, Berkeley

ExaBiome

AMReX

SymPACK

Graph analytics
Some motivating system trends

The first exascale systems appeared in 2022

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

Need to reduce communication costs in software

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

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

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

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

**two-sided message**

- message id
- data payload

**one-sided RMA put**

- address
- data payload

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

Looks like shared memory: shared data structures with asynchronous access
One-sided GASNet-EX vs one- and two-sided MPI

Four distinct network hardware types

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

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

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

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> gp1 = ...;
future<int> f1 = rget(gp1);
// unrelated work...
int t1 = f1.wait();
```

Wait returns the result when the rget completes
Remote procedure call (RPC)

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

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

Many RPCs can be active simultaneously, hiding latency

```
// Emit an RPC
upcxx::rpc(target, fn, arg1, arg2)
```

Execute fn(arg1, arg2) on process target

Result available via a future

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

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:

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

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