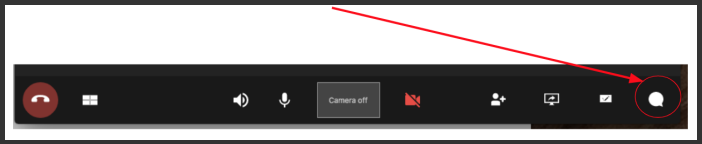
Working with external C libraries in Chapel

ALEX RAZOUMOV
alex.razoumov@westgrid.ca

To ask questions

- Websteam: email **info@westgrid.ca**
- Vidyo: use the GROUP CHAT to ask questions



- Please mute your microphone unless you have a question
- Feel free to ask questions via audio at any time

Why another language? http://chapel.cray.com

- High-level parallel programming language
  - "Python for parallel programming"
  - much easier to use and learn than MPI; few lines of Chapel typically replace tens of lines of MPI code
  - abstractions for data distribution/parallelism, task parallelism
  - optimization for data-driven placement of subcomputations
  - granular ("multi-resolution") design: can bring closer to machine level if needed
  - everything you can do in MPI (and OpenMP!), you should be able to do in Chapel
- Focus on performance
  - compiled language; simple Chapel codes perform as well as optimized C/C++/Fortran code
  - very complex Chapel codes run at ∼70% performance of a well-tuned MPI code (some room for improvement)
- Perfect language for learning parallel programming for beginners
- Open-source: can compile on any Unix-like platform
  - precompiled for MacOS (single-locale via Homebrew)
  - Docker image `http://dockr.ly/2vJbi06` (simulates a multi-locale environment)

Task- vs. data-parallel

	single locale shared memory parallelism	multiple locales distributed memory parallelism + likely shared memory parallelism

task parallel	`config var numtasks = 2;`
	`coforall taskid in 1..numtasks do`
	`  writeln("this is task ", taskid);`
		`forall loc in Locales do`
		`  on loc do`
		`  writeln("this locale is named ", here.name);`

data parallel	`var A, B, C: [1..1000] real;`
	`forall (a,b,c) in zip(A,B,C) do`
	`  c = a + b;`
		`use BlockDist;`
		`const mesh = {1..100,1..100} dmapped`
		`  Block(boundingBox={1..100,1..100});`
		`var T: [mesh] real;`
		`forall (i,j) in T.domain do`
		`  T[i,j] = i + j;`

- Watch our introductory Chapel lecture series
  `https://westgrid.github.io/trainingMaterials/programming`

- Cedar (OmniPath) / Graham (InfiniBand) /Béluga (InfiniBand)
  - `https://docs.computecanada.ca/wiki/Chapel`

```
$ source /home/razoumov/startSingleLocale.sh
$ chpl --version

$ source /home/razoumov/startMultiLocale.sh
$ chpl --version
```

- Fairly small community at the moment:
  too few people know/use Chapel $\Longleftrightarrow$ relatively few libraries

- You can use functions/libraries written in other languages, e.g. in C
  1. direct calls will always be serial
  2. high-level Chapel parallel libraries can use C/F90/etc libraries underneath

External C functions in C

dependencies.c

```c
#include <stdio.h>
void printSquared(int x) {
  printf("x = %d\n", x*x);
}
```

driver.c

```c
void printSquared(int); // function prototype to declare printSquared(),
                        // `extern` keyword implicitly assumed by the compiler
int main(void) {
  printSquared(5);
}
```

```
$ gcc -O2 driver.c dependencies.c -o driver
$ ./driver
x = 25
```

External C functions in Chapel

The Chapel compiler will:

1. Convert Chapel code to C
   - all C functions must be prototyped in Chapel with an `extern` keyword
   - note that these declarations will not be translated to last slide's `extern` C function prototypes!
2. Translate all `require ``*.h''` statements in Chapel to `#include ``*.h''` *at the start of* the just-created C code
   - this would include any C function definitions in C header files ⇒ no need for a separate C function declaration
   - alternatively, you can remove `require` and supply the header file in the command line:
     `chpl --fast test.chpl dependencies.h -o test`
3. Compile the resulting C code with included headers

test.chpl

```
require "dependencies.h";
extern proc printSquared(x: c_int): void;
printSquared(5);
```

dependencies.h

```
void printSquared(int x) {
  printf("x = %d\n", x*x);
}
```

```
$ chpl --fast test.chpl -o test
$ ./test
x = 25
```

External C functions in Chapel (cont.)

- All `require ``*.c''` statements in Chapel will not be `#include`'d into the C code, but will simply add those C files to compilation
  - ⇒ you will need to include a separate C function declaration
  - alternatively, you can remove `require` and compile with:
    `chpl --fast test.chpl dependencies.c dependencies.h -o test`

test.chpl

```
require "dependencies.c", "dependencies.h";
extern proc printSquared(x: c_int): void;
printSquared(5);
```

dependencies.c

```
#include <stdio.h>
void printSquared(int x) {
  printf("x = %d\n", x*x);
}
```

dependencies.h

```
void printSquared(int);
```

```
$ chpl --fast test.chpl -o test
$ ./test
x = 25
```

- You can also put C code into `extern{...}` blocks in Chapel, as described in `http://bit.ly/39FvOT8`, but that requires a special Chapel build

C types

- Within `extern` C procedures in Chapel, you need to describe their types and the types of their variables
- These must exactly match the corresponding C types (see next example):
  - how many bits
  - order of bits/bytes
  - other conventions
- So, you must use one of these types to pass functions and variables to C:

c_int	c_uint	c_long	c_ulong	c_longlong
c_ulonglong	c_char	c_schar	c_uchar	c_short
c_ushort	ssize_t	size_t	c_float	c_double

  and pointer types:

c_void_ptr	c_ptr(T)	c_string	c_fn_ptr	c_array(T,n)

- Chapel variables passed to C functions as arguments will need to match the declared C types (can also be converted on the fly)

Parallel safety

- `extern` C code does not have any support for multiple locales and is in general single-threaded
- When calling C code from parallel Chapel, be aware of the context
  - your C code will run locally in the current thread
  - local (this thread, this node) copies of global variables will have their own values
  - C pointers only work in the context of shared memory $\Rightarrow$ do not pass them across nodes

Let's pass an actual Chapel variable (call by value):

test.chpl

```
require "dependencies.h";
extern proc printSquared(x: c_int): void;
var a: c_int = 5;
printSquared(a);
```

dependencies.h

```
void printSquared(int x) {
  printf("x = %d\n", x*x);
}
```

```
$ chpl --fast test.chpl -o test
$ ./test
x = 25
```

Instead of a pointer argument to a C function, you can pass the variable itself prototyped with `ref` intent (call by reference):

test.chpl

```
require "dependencies.h";
extern proc increment(ref x: c_int): void;
var a: c_int = 5;
increment(a);
writeln("a = ", a);
```

dependencies.h

```
void increment(int* x) {
  *x += 1;
}
```

```
$ chpl --fast test.chpl -o test
$ ./test
a = 6
```

We can pass a C fixed-size array via its name (in this case a pointer to its first element; also call by reference):

test.chpl

```
require "dependencies.h";
extern proc reverse(x: c_ptr(c_float),
                    len: size_t): void;
var A: c_array(c_float,10); // C fixed-size array,
                            // indices start from 0,
                            // can't iterate over values
for j in 0..9 do
  A[j] = j: c_float;
writeln(A);
reverse(A, 10);
writeln(A);   // now in reverse
```

dependencies.h

```
void reverse(float x[], size_t len) {
  for (int i = 0; i < len/2; ++i) {
    float tmp = x[i];
    x[i] = x[len-1-i];
    x[len-1-i] = tmp;
  }
}
```

```
$ chpl --fast test.chpl -o test
$ ./test
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
[9.0, 8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0, 0.0]
```

Now pass a Chapel array over a range as a reference argument to the same C function:

test.chpl

```
require "dependencies.h";

// same external function as before
// but declared differently
extern proc reverse(ref x: c_float, len: size_t): void;

// Chapel fixed-size array of C types over a range,
// arbitrary starting index, can iterate over values
var B: [1..10] c_float;
var count = 1;
for b in B do {
  b = count: c_float;
  count += 1;
 }
writeln(B);
reverse(B[1], 10);
writeln(B);   // now in reverse
```

dependencies.h

```
void reverse(float x[], size_t len) {
  for (int i = 0; i < len/2; ++i) {
    float tmp = x[i];
    x[i] = x[len-1-i];
    x[len-1-i] = tmp;
  }
}
```

```
$ chpl --fast test.chpl -o test
$ ./test
1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0
10.0 9.0 8.0 7.0 6.0 5.0 4.0 3.0 2.0 1.0
```

Let's pass a multi-dimensional array as a reference argument:

test.chpl

```
require "dependencies.h";

extern proc matrix(nx: size_t, ny: size_t,
                   ref x: c_float): void;
var C: [1..10, 1..10] c_float;
matrix(10, 10, C[1,1]);
writeln(C);
```

dependencies.h

```
void matrix(size_t nx, size_t ny,
            float *x) {
  for (int i = 0; i < nx; ++i) {
    for (int j = 0; j < ny; ++j) {
      *((x+i*ny) + j) = abs(i-j);
    }
  }
}
```

```
$ chpl --fast test.chpl -o test
$ ./test
0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0
1.0 0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0
2.0 1.0 0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0
3.0 2.0 1.0 0.0 1.0 2.0 3.0 4.0 5.0 6.0
4.0 3.0 2.0 1.0 0.0 1.0 2.0 3.0 4.0 5.0
5.0 4.0 3.0 2.0 1.0 0.0 1.0 2.0 3.0 4.0
6.0 5.0 4.0 3.0 2.0 1.0 0.0 1.0 2.0 3.0
7.0 6.0 5.0 4.0 3.0 2.0 1.0 0.0 1.0 2.0
8.0 7.0 6.0 5.0 4.0 3.0 2.0 1.0 0.0 1.0
9.0 8.0 7.0 6.0 5.0 4.0 3.0 2.0 1.0 0.0
```

ASCII $\rightarrow$ binary $\rightarrow$ scientific data format (NetCDF, HDF5, VTK)

1. portable binary encoding (little vs. big endian byte order)
2. compression
3. random access
4. parallel I/O (in development, partially available)

- In Chapel `NetCDF`, `HDF5` are provided as package modules (libraries outside of the Standard Library)

NetCDF

- Chapel's `NetCDF.C_NetCDF` module contains NetCDF `extern` C function/type/constant prototypes for Chapel
  - NetCDF C library must be installed on the system
  - currently serial NetCDF only
  - when decribing data types to NetCDF-4 functions, use their C atomic data types:

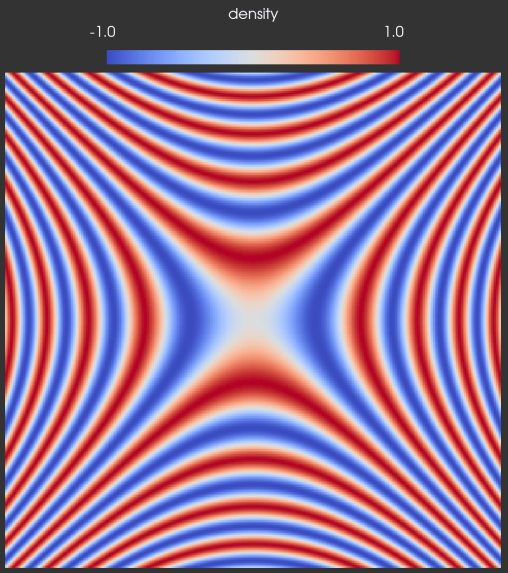
type	C definition	bits
int	NC_INT	32
byte	NC_BYTE	8
char	NC_CHAR	8
short	NC_SHORT	16
float	NC_FLOAT	32
double	NC_DOUBLE	64
...	...	...

netcdfWrite.chpl

```
use NetCDF.C_NetCDF;
proc cdfError(e) {
  if e != NC_NOERR {
    writeln("Error: ", nc_strerror(e): string);
    exit(2);
  }
}
config const nx = 300, ny = 300, h = 5.0;
var T: [1..nx, 1..ny] c_float, x, y: real;
var ncid, xDimID, yDimID, varID: c_int;
var dimIDs: [0..1] c_int;   // two elements
for i in 1..nx do {
  x = (i-0.5)/nx*2*h - h;   // square -h to +h on each side
  for j in 1..ny do {
    y = (j-0.5)/ny*2*h - h;
    T[i,j] = (sin(x*x-y*y)): c_float;  }}

cdfError(nc_create("300x300.nc", NC_NETCDF4, ncid));   // const NC_NETCDF4 => file in netCDF-4 standard
cdfError(nc_def_dim(ncid, "x", nx: size_t, xDimID));   // define the dimensions
cdfError(nc_def_dim(ncid, "y", ny: size_t, yDimID));
dimIDs = [xDimID, yDimID];                             // set up dimension IDs array
cdfError(nc_def_var(ncid, "density", NC_FLOAT, 2, dimIDs[0], varID));   // define the 2D data variable
cdfError(nc_def_var_deflate(ncid, varID, NC_SHUFFLE, deflate=1, deflate_level=9)); // compress 0=no 9=max
cdfError(nc_enddef(ncid));                             // done defining metadata
cdfError(nc_put_var_float(ncid, varID, T[1,1]));       // write data to file
cdfError(nc_close(ncid));
```

```
$ chpl --fast netcdfWrite.chpl -o netcdfWrite \
    -I/usr/local/include \
    -L/usr/local/lib -lnetcdf
$ ./netcdfWrite
```

- $300^2 \times 4$ bytes = 352kB
- NetCDF without compression 360kB
- NetCDF with l=9 compression 224kB



HDF5

- Chapel's `HDF5.C_HDF5` module contains HDF5 `extern` C function/type/constant prototypes for Chapel
  - HDF5 C library must be installed on the system
  - when decribing data types to NetCDF-4 functions, use their C atomic data types
- In the upcoming Chapel 1.21 `HDF5.IOusingMPI` module contains several higher-level functions for parallel HDF5 reads/writes using multiple Chapel locales
  - requires parallel HDF5 library, which in turn requires MPI library
  - this signals future capabilities of parallel libraries in Chapel

Parallel HDF5 write example with Chapel 1.21

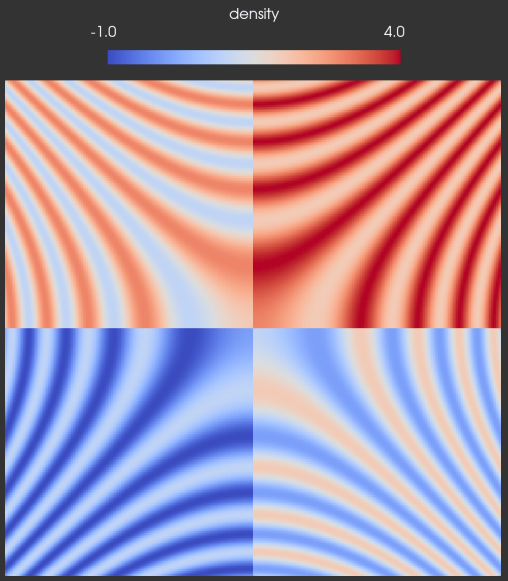
hdf5Write.chpl

```
use BlockDist, HDF5.IOusingMPI;

config const nx = 300, ny = 300, fileName = "300x300.h5";
var h = 5.0;
var T = newBlockArr({1..nx, 1..ny}, c_float);
var x: [1..nx] real, y: [1..ny] real;

forall loc in Locales do
  on loc do
    writeln("node", loc.id, " >> ", T.localSubdomain());

for i in 1..nx do   // square -h to +h on each side
  x[i] = (i-0.5)/nx*2*h - h;
for j in 1..ny do
  y[j] = (j-0.5)/ny*2*h - h;
forall (i,j) in T.domain do
  T[i,j] = (sin(x[i]**2 - y[j]**2) + T[i,j].locale.id): c_float;

hdf5WriteDistributedArray(T, fileName, "density");
```

Parallel HDF5 write example (cont.)

```
$ source startDevMultiLocale.sh
$ salloc --fast --time=0-00:05 --ntasks=4 \
         --nodes=4 --mem-per-cpu=1000
$ chpl hdf5Write.chpl -o hdf5Write
$ ./hdf5Write -nl 4

node0 >> {1..150, 1..150}
node2 >> {151..300, 1..150}
node1 >> {1..150, 151..300}
node3 >> {151..300, 151..300}
```

- The startup script loads `gcc/7.3.0`, `openmpi/3.1.2`, `hdf5-mpi/1.10.3` underneath ⇒ Chapel is able to link to parallel HDF5 without the explicit flags
- Note the random printout order
- To open `300x300.h5` in ParaView, create `300x300.xdmf` XML wrapper



LinearAlgebra package module

Details at `https://chapel-lang.org/docs/modules/packages/LinearAlgebra.html`

```
use LinearAlgebra;
```

- Many LinearAlgebra functions require BLAS or LAPACK $\Rightarrow$ in these cases you should link to external BLAS or LAPACK

```
[cluster]$ module load openblas/0.3.4
[cluster]$ chpl --fast test.chpl -o test -lopenblas
```

```
[macos]$ brew install lapack openblas
[macos]$ chpl test.chpl -o test -I/usr/local/opt/lapack/include \
           -L/usr/local/opt/lapack/lib -llapacke -llapack
           -I/usr/local/opt/openblas/include -L/usr/local/opt/openblas/lib
           -lblas -lopenblas
```

- LinearAlgebra is under active development!!!
  - both dense and sparse arrays
  - dozens of linear algebra operations implemented: inverse, linear solve, eigen{values,vectors}, ...
  - my understanding is that LinearAlgebra functions should work on distributed arrays out-of-the-box, in many cases calling serial BLAS and LAPACK functions underneath – in my tests, they mostly work ...

Serial inverse of a square matrix

inverse.chpl

```
use LinearAlgebra;

config const n = 10;

proc sinMatrix(n) {
  var A = Matrix(n);
  const fac0 = 1.0/(n+1.0);
  const fac1 = sqrt(2.0*fac0);
  for (i,j) in {1..n,1..n} do
    A[i,j] = fac1*sin(i*j*pi*fac0) + 0.1; // without 0.1 A=inv(A)
  return A;
}

var A = sinMatrix(n);
writeln("A = ", A);

var B = inv(A);
writeln("inverse = ", B);

writeln("their product = ", dot(A,B));
```

Serial linear solve

solve.chpl

```
use LinearAlgebra;

var A: [{1..5,1..5}] real;
A[1,..] = [1.0, 0.0,-5.0,-1.0, 2.0];
A[2,..] = [2.0, 6.0,-2.0, 3.0, 0.0];
A[3,..] = [2.0, 5.0, 2.0, 1.0, 1.0];
A[4,..] = [-2.0,1.0, 2.5, 3.0, 1.0];
A[5,..] = [-1.0,1.0, 0.0, 3.5, 0.5];
var b: [{1..5}] real = [1,2,3,4,5];
var c: [{1..5}] real = b;

var x = solve(A, b);   // b gets modified too
writeln("solution = ", x);
writeln("check = ", c - dot(A,x));
```

Inner product of a distributed vector

dotMulti.chpl

```
use LinearAlgebra, BlockDist;

config const n = 1e6: int;
const space = {1..n};   // 1D domain
const distributedSpace = space dmapped Block(boundingBox=space);
var A : [distributedSpace] real = [i in distributedSpace] (i:real / n:real);

write("A is distributed as ");
for loc in Locales do
  on loc do
    write(A.localSubdomain()," ");
writeln();

writeln("LinearAlgebra product = ", dot(A, A));

var p2 = + reduce (A * A);
writeln("reduction product = ", p2);
```

Inner product of a distributed vector (cont.)

```
$ source startDevMultiLocale
$ chpl dotMulti.chpl -o dotMulti       # no BLAS or LAPACK dependency

$ ./dotMulti -nl 1
A is distributed as {1..1000000}
LinearAlgebra product = 3.33334e+05
reduction product = 3.33334e+05

$ salloc --time=0-00:05 --ntasks=4 --nodes=4 --mem-per-cpu=1000
$ ./dotMulti -nl 4
A is distributed as {1..250000} {250001..500000} {500001..750000} {750001..1000000}
LinearAlgebra product = 3.33334e+05
reduction product = 3.33334e+05
```

Distributed linear solve

solveMulti.chpl

```
use LinearAlgebra, BlockDist, Random;
config const n = 20;
proc distribution(object) {
  for loc in Locales do
    on loc do
      write(object.localSubdomain()," ");
  writeln();
}

var A = newBlockArr({1..n, 1..n}, real);
fillRandom(A, seed=0);   // pseudo-random
write("A is distributed as ");
distribution(A);

var b = newBlockArr({1..n}, real);
[i in b.domain] b[i] = i:real;
write("b is distributed as ");
distribution(b);

var x = solve(A, b);   // b gets modified too
writeln("solution = ", x);
write("x is distributed as ");
distribution(x);
```

Distributed linear solve (cont.)

```
$ source startDevMultiLocale.sh
$ module load openblas/0.3.4
$ chpl solveMulti.chpl -o solveMulti -lopenblas   # needs BLAS

$ ./solveMulti -nl 1
A is distributed as {1..20, 1..20}
b is distributed as {1..20}
solution = 31.1783 73.0722 -51.5904 -48.2999 ... 143.333
x is distributed as {1..20}

$ salloc --time=0-00:05 --ntasks=4 --nodes=4 --mem-per-cpu=1000
$ ./solveMulti -nl 4
A is distributed as {1..10, 1..10} {1..10, 11..20} {11..20, 1..10} {11..20, 11..20}
b is distributed as {1..5} {6..10} {11..15} {16..20}
solution = 31.1783 73.0722 -51.5904 -48.2999 ... 143.333
x is distributed as {1..5} {6..10} {11..15} {16..20}
```

BLAS and LAPACK modules

Details at `https://chapel-lang.org/docs/modules/packages/BLAS.html` and `https://chapel-lang.org/docs/modules/packages/LAPACK.html`

```
use BLAS;
```

- BLAS provides standard building blocks for performing basic vector and matrix operations
  - commonly used in other linear algebra packages, e.g. LAPACK and ScaLAPACK
- Chapel's `BLAS.C_BLAS` module contains C function/type/constant prototypes for Chapel
  - built and tested with Netlib's C_BLAS
  - compatible with many other implementations (OpenBLAS, MKL, ATLAS, ...)

```
use LAPACK;
```

- LAPACK is a standard linear algebra package, written in Fortran 90, built on top of BLAS
- Chapel's `LAPACK` module contains function/type/constant prototypes for Chapel

BLAS and LAPACK modules (cont.)

- Many examples in Chapel's source code (details in the last slide)
- To compile your Chapel code, you will need LAPACK and libgfortran on your system
- General recommendation is to use LinearAlgebra module:
  - higher-level interface than the original BLAS and LAPACK
  - will use BLAS and/or LAPACK underneath
  - bonus parallelism

Summary

Questions?

- Official latest (1.20) documentation
  `https://chapel-lang.org/docs/index.html`
- Pre-release (1.21) documentation
  `https://chapel-lang.org/docs/master/index.html`
- C interoperability
  `https://chapel-lang.org/docs/master/language/spec/interoperability.html`
- Lots of great examples inside Chapel's source code (a few adapted for this webinar):

```
$ git clone https://github.com/chapel-lang/chapel.git
$ ls chapel/test/library/packages
./                  Curl/               HDF5.skipif*        LockFreeQueue/        Search/             UnitTest.skipif@
../                 Curl.skipif         HDFS/               LockFreeQueue.skipif@ SharedObject/       ZMQ/
BLAS/               DistributedIters/   HDFS.skipif         LockFreeStack/        Sort/               ZMQ.skipif*
BLAS.skipif@        EpochManager/       Itertools/          LockFreeStack.skipif@ TOML/               canCompileNoLink/
Buffers/            EpochManager.skipif LAPACK/             MPI/                  TOML.skipif         csv/
Collection/         FFTW/               LAPACK.skipif*      MatrixMarket/         TensorFlow/
Crypto/             Futures/            LinearAlgebra/      NetCDF/               TensorFlow.skipif*
Crypto.skipif       HDF5/               LinearAlgebraJama/  NetCDF.skipif*        UnitTest/
```

- Email me "alex.razoumov@westgrid.ca" if you need help