



Chapel: Productive Parallel Programming at Scale

Elliot Ronaghan, Chapel Team, Cray Inc.

EAGE 2017

June 16th, 2017



COMPUTE

| STORE

| ANALYZE

Safe Harbor Statement

This presentation may contain forward-looking statements that are based on our current expectations. Forward looking statements may include statements about our financial guidance and expected operating results, our opportunities and future potential, our product development and new product introduction plans, our ability to expand and penetrate our addressable markets and other statements that are not historical facts. These statements are only predictions and actual results may materially vary from those projected. Please refer to Cray's documents filed with the SEC from time to time concerning factors that could affect the Company and these forward-looking statements.



Motivation for Chapel

Q: Why doesn't HPC programming have an equivalent to Python / Matlab / Scala / Swift / (your favorite programming language here) ?

- one that makes it easy to get programs up and running quickly
- one that is portable across system architectures and scales
- one that bridges the HPC, data analysis, and mainstream communities

A: We believe this is due not to any particular technical challenge, but rather a lack of sufficient...

- ...long-term efforts
- ...resources
- ...community will
- ...patience

Chapel is our attempt to reverse this trend!



What is Chapel?

Chapel: A productive parallel programming language

- portable
- open-source
- a collaborative effort

Goals:

- Support general parallel programming
 - “any parallel algorithm on any parallel hardware”
- Make parallel programming far more productive

What does “Productivity” mean to you?

Recent Graduates:

“something similar to what I used in school: Python, Matlab, Swift, ...”

Seasoned HPC Programmers:

“that sugary stuff that I don’t need because I ~~was born to suffer~~
want full control
to ensure performance”

Computational Scientists:

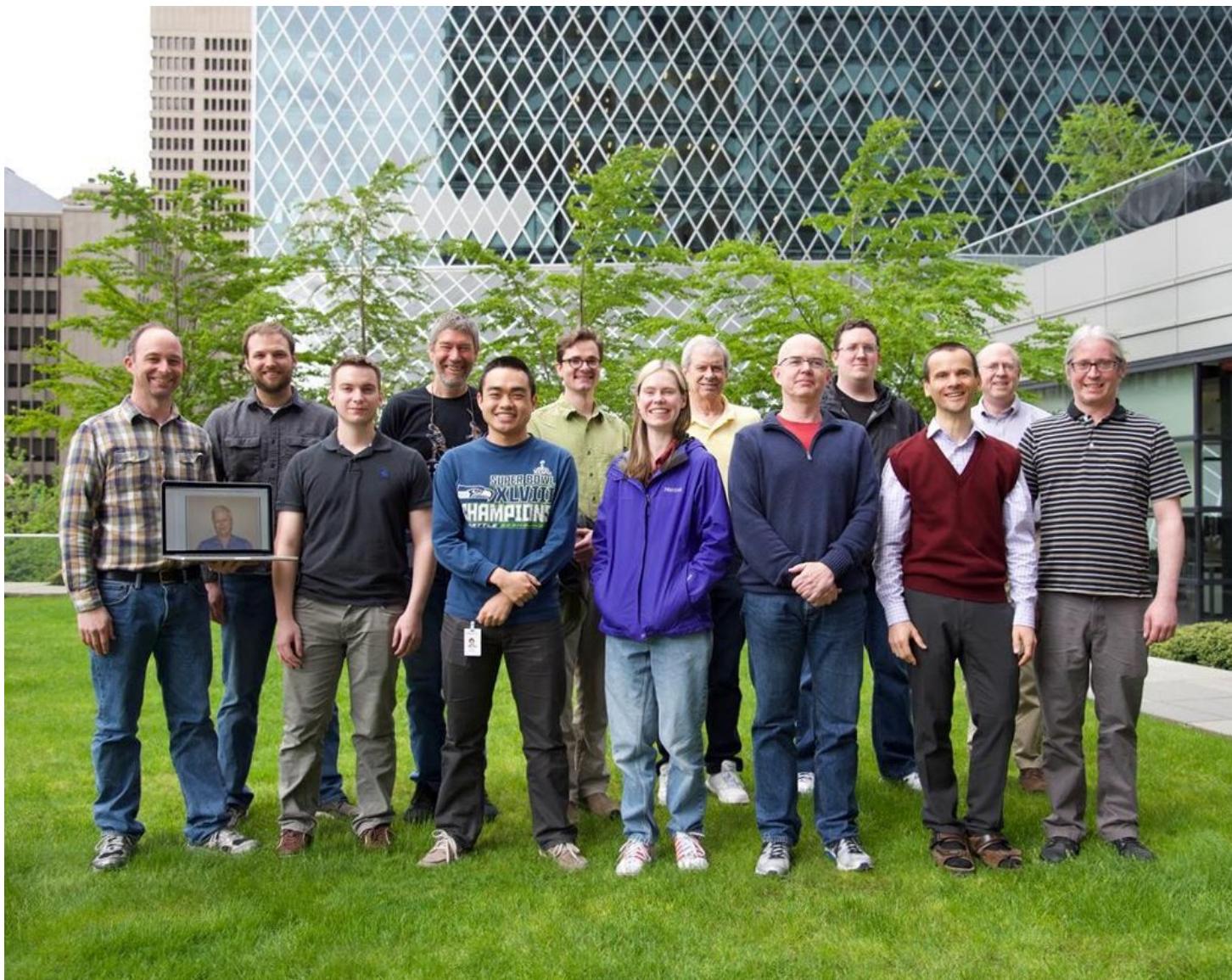
“something that lets me express my parallel computations
without having to wrestle with architecture-specific details”

Chapel Team:

“something that lets computational scientists express what they want,
without taking away the control that HPC programmers want,
implemented in a language as attractive as recent graduates want.”

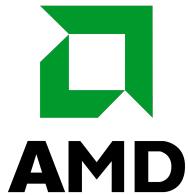


The Chapel Team at Cray (May 2017)





Chapel Community R&D Efforts



THE GEORGE
WASHINGTON
UNIVERSITY
WASHINGTON, DC



Lawrence Berkeley
National Laboratory



Yale

<http://chapel.cray.com/collaborations.html>



COMPUTE

STORE

ANALYZE

High Performance Computing (HPC) Programming Models by Example



COMPUTE

| STORE

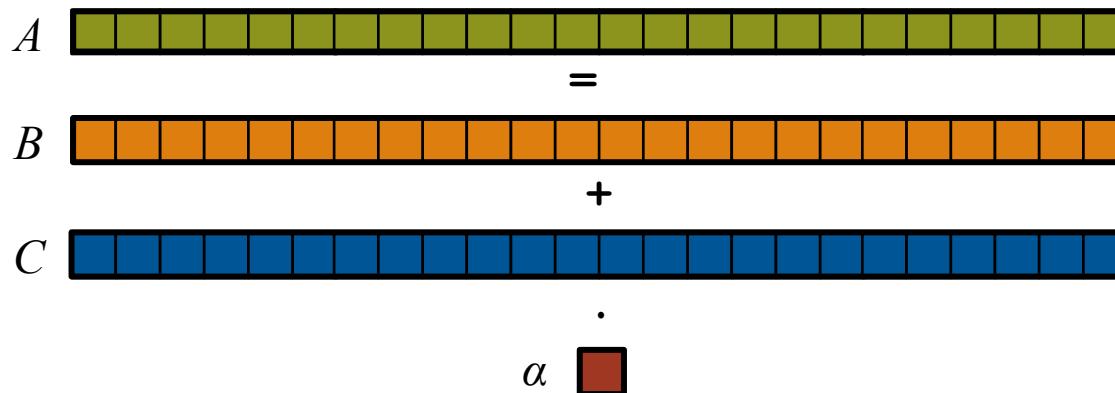
| ANALYZE

STREAM Triad: a trivial parallel computation

Given: m -element vectors A, B, C

Compute: $\forall i \in 1..m, A_i = B_i + \alpha \cdot C_i$

In pictures:

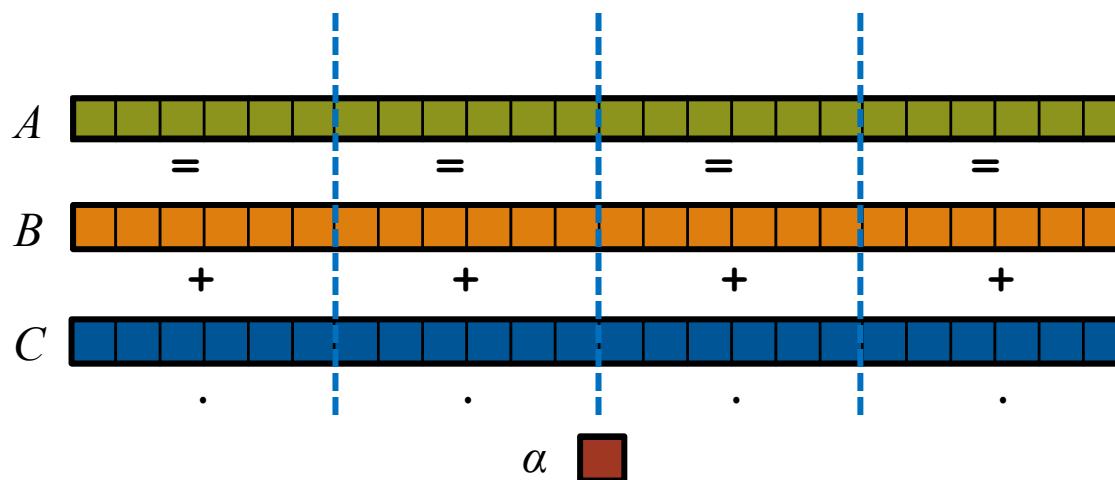


STREAM Triad: a trivial parallel computation

Given: m -element vectors A, B, C

Compute: $\forall i \in 1..m, A_i = B_i + \alpha \cdot C_i$

In pictures, in parallel:

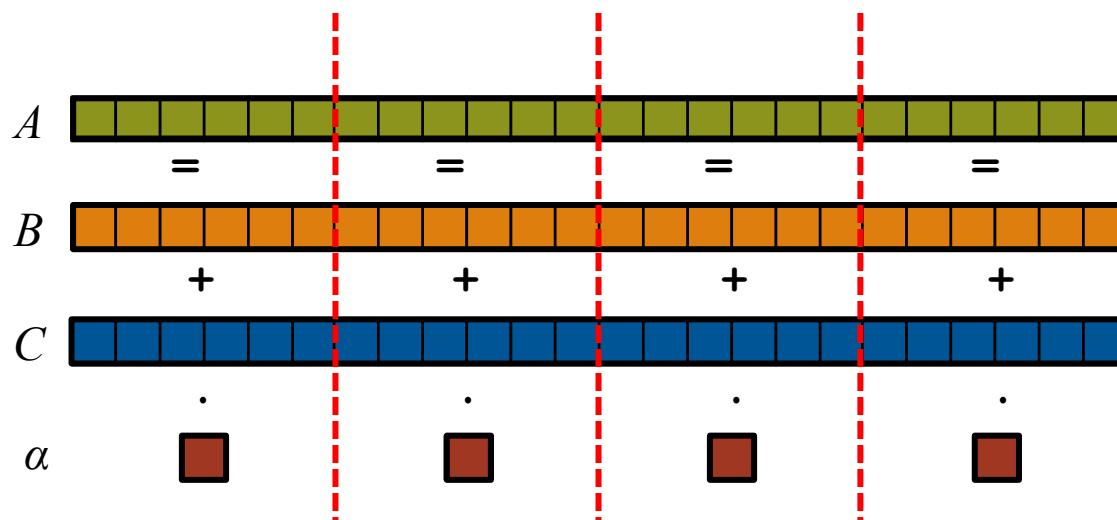


STREAM Triad: a trivial parallel computation

Given: m -element vectors A, B, C

Compute: $\forall i \in 1..m, A_i = B_i + \alpha \cdot C_i$

In pictures, in parallel (distributed memory):

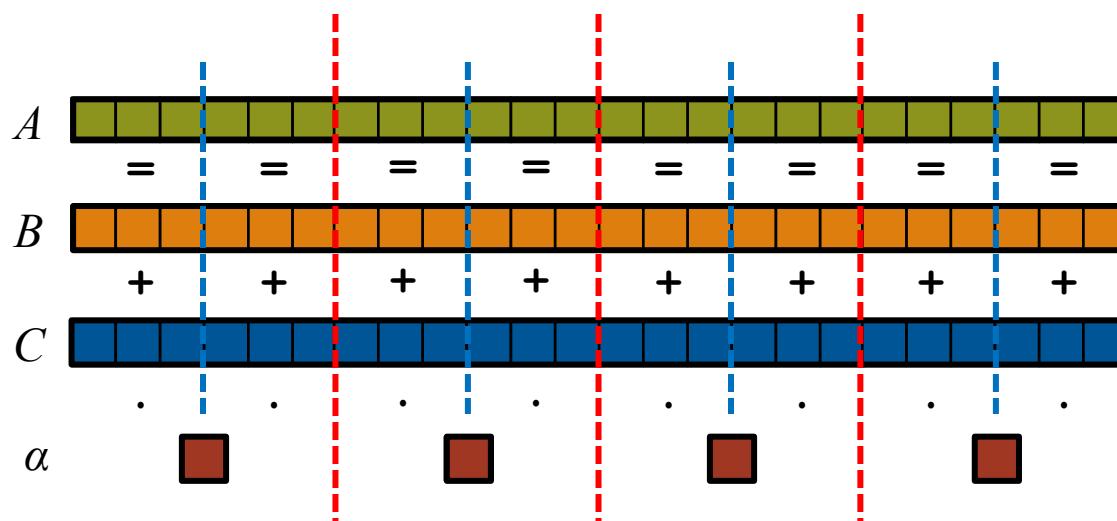


STREAM Triad: a trivial parallel computation

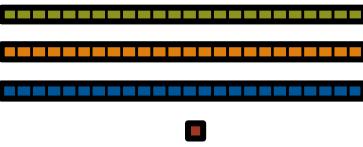
Given: m -element vectors A, B, C

Compute: $\forall i \in 1..m, A_i = B_i + \alpha \cdot C_i$

In pictures, in parallel (distributed memory multicore):



STREAM Triad: Python



Python

```
m = 1000  
alpha = 3.0
```

```
A = [0.0] * m  
B = [2.0] * m  
C = [1.0] * m
```

```
for j in range(m):  
    A[j] = B[j] + alpha * C[j];
```



COMPUTE

STORE

ANALYZE

STREAM Triad: MPI



MPI

```
#include <hpcc.h>

static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Parms *params) {
    int myRank, commSize;
    int rv, errCount;
    MPI_Comm comm = MPI_COMM_WORLD;

    MPI_Comm_size( comm, &commSize );
    MPI_Comm_rank( comm, &myRank );

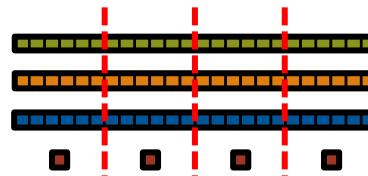
    rv = HPCC_Stream( params, 0 == myRank );
    MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM,
                0, comm );

    return errCount;
}

int HPCC_Stream(HPCC_Parms *params, int doIO) {
    register int j;
    double scalar;

    VectorSize = HPCC_LocalVectorSize( params, 3,
                                       sizeof(double), 0 );

    a = HPCC_XMALLOC( double, VectorSize );
    b = HPCC_XMALLOC( double, VectorSize );
    c = HPCC_XMALLOC( double, VectorSize );
```



```
if (!a || !b || !c) {
    if (c) HPCC_free(c);
    if (b) HPCC_free(b);
    if (a) HPCC_free(a);
    if (doIO) {
        fprintf( outFile, "Failed to allocate memory
(%d).\n", VectorSize );
        fclose( outFile );
    }
    return 1;
}

for (j=0; j<VectorSize; j++) {
    b[j] = 2.0;
    c[j] = 0.0;
}

scalar = 3.0;

for (j=0; j<VectorSize; j++)
    a[j] = b[j]+scalar*c[j];

HPCC_free(c);
HPCC_free(b);
HPCC_free(a);
```



COMPUTE

STORE

ANALYZE

STREAM Triad: MPI+OpenMP

MPI + OpenMP

```
#include <hpcc.h>
#ifndef _OPENMP
#include <omp.h>
#endif

static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Parms *params) {
    int myRank, commSize;
    int rv, errCount;
    MPI_Comm comm = MPI_COMM_WORLD;

    MPI_Comm_size( comm, &commSize );
    MPI_Comm_rank( comm, &myRank );

    rv = HPCC_Stream( params, 0 == myRank );
    MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM,
                0, comm );

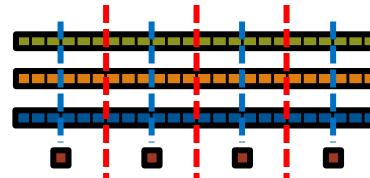
    return errCount;
}

int HPCC_Stream(HPCC_Parms *params, int doIO) {
    register int j;
    double scalar;

    VectorSize = HPCC_LocalVectorSize( params, 3,
                                       sizeof(double), 0 );

    a = HPCC_XMALLOC( double, VectorSize );
    b = HPCC_XMALLOC( double, VectorSize );
    c = HPCC_XMALLOC( double, VectorSize );
}

```



```
if (!a || !b || !c) {
    if (c) HPCC_free(c);
    if (b) HPCC_free(b);
    if (a) HPCC_free(a);
    if (doIO) {
        fprintf( outFile, "Failed to allocate memory
(%d).\n", VectorSize );
        fclose( outFile );
    }
    return 1;
}

#ifndef _OPENMP
#pragma omp parallel for
#endif
for (j=0; j<VectorSize; j++) {
    b[j] = 2.0;
    c[j] = 0.0;
}

scalar = 3.0;

#ifndef _OPENMP
#pragma omp parallel for
#endif
for (j=0; j<VectorSize; j++)
    a[j] = b[j]+scalar*c[j];

HPCC_free(c);
HPCC_free(b);
HPCC_free(a);
}

```

STREAM Triad: MPI+OpenMP vs. CUDA

MPI + OpenMP

```
#ifdef _OPENMP
#include <omp.h>
#endif

static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *params) {
    int myRank, commSize;
    int rv, errCount;
    MPI_Comm comm = MPI_COMM_WORLD;
    MPI_Comm_size( comm, &commSize );
    MPI_Comm_rank( comm, &myRank );
    rv = HPCC_Stream( params, 0 == myRank );
    MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM, 0, comm );
    return errCount;
}

int HPCC_Stream(HPCC_Params *params, int doIO) {
    register int j;
    double scalar;
    VectorSize = HPCC_LocalVectorSize( params, 3, sizeof(double), 0 );
    a = HPCC_XMALLOC( double, VectorSize );
    b = HPCC_XMALLOC( double, VectorSize );
    c = HPCC_XMALLOC( double, VectorSize );
    if (!a || !b || !c) {
        if (c) HPCC_free(c);
        if (b) HPCC_free(b);
        if (a) HPCC_free(a);
        if (doIO) {
            fprintf( outFile, "Failed to allocate memory (%d).\n", VectorSize );
            fclose( outFile );
        }
        return 1;
    }
    return 0;
}

#endif _OPENMP
#pragma omp parallel for
for (j=0; j<VectorSize; j++) {
    b[j] = 2.0;
    c[j] = 0.0;
}
scalar = 3.0;

#endif _OPENMP
#pragma omp parallel for
for (j=0; j<VectorSize; j++)
    a[j] = b[j]+scalar*c[j];

HPCC_free(c);
HPCC_free(b);
HPCC_free(a);
return 0;
}
```

CUDA

```
#define N 2000000

int main() {
    float *d_a, *d_b, *d_c;
    float scalar;

    cudaMalloc((void**)&d_a, sizeof(float)*N);
    cudaMalloc((void**)&d_b, sizeof(float)*N);
    cudaMalloc((void**)&d_c, sizeof(float)*N);

    dim3 dimBlock(128);
    dim3 dimGrid(N/dimBlock.x );
    if( N % dimBlock.x != 0 ) dimGrid

    set_array<<<dimGrid,dimBlock>>>(d_b, .5f, N);
    set_array<<<dimGrid,dimBlock>>>(d_c, .5f, N);

    scalar=3.0f;
    STREAM_Triad<<<dimGrid,dimBlock>>>(d_b, d_c, d_a, scalar, N);
    cudaThreadSynchronize();

    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    __global__ void set_array(float *a, float value, int len) {
        int idx = threadIdx.x + blockIdx.x * blockDim.x;
        if (idx < len) a[idx] = value;
    }

    __global__ void STREAM_Triad( float *a, float *b, float *c,
                                float scalar, int len) {
        int idx = threadIdx.x + blockIdx.x * blockDim.x;
        if (idx < len) c[idx] = a[idx]+scalar*b[idx];
    }
}
```

COMPUTE

STORE

ANALYZE

STREAM Triad: MPI+OpenMP vs. CUDA

MPI + OpenMP

```
#ifdef __OPENMP
#include <omp.h>
#endif

static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *params) {
    int myRank, commSize;
    int rv, errCount;
    MPI_Comm comm = MPI_COMM_WORLD;

    MPI_Comm_size( comm, &commSize );
    MPI_Comm_rank( comm, &myRank );

    rv = HPCC_Stream( params, 0 == myRank );
    MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM, 0, comm );

    return errCount;
}

int HPCC_Stream(HPCC_Params *params, int doIO) {
    register int j;
    double scalar;
```

CUDA

```
#define N 2000000

int main() {
    float *d_a, *d_b, *d_c;
    float scalar;

    cudaMalloc((void**)&d_a, sizeof(float)*N);
    cudaMalloc((void**)&d_b, sizeof(float)*N);
    cudaMalloc((void**)&d_c, sizeof(float)*N);

    dim3 dimBlock(128);
    dim3 dimGrid(N/dimBlock.x);
    if( N % dimBlock.x != 0 ) dimGrid
```

*HPC suffers from too many distinct notations for expressing parallelism and locality.
This tends to be a result of bottom-up language design.*

```
    if (a) HPCC_free(a);
    if (a) HPCC_free(a);
    if (doIO) {
        fprintf( outFile, "Failed to allocate memory (%d).\n", VectorSize );
        fclose( outFile );
    }
    return 1;
}

#ifndef __OPENMP
#pragma omp parallel for
#endif
for (j=0; j<VectorSize; j++) {
    b[j] = 2.0;
    c[j] = 0.0;
}
scalar = 3.0;

#ifndef __OPENMP
#pragma omp parallel for
#endif
for (j=0; j<VectorSize; j++)
    a[j] = b[j]+scalar*c[j];

HPCC_free(c);
HPCC_free(b);
HPCC_free(a);
return 0;
}
```

```
STREAM_Triad<<<dimGrid, dimBlock>>>(d_b, d_c, d_a, scalar, N);
cudaThreadSynchronize();

cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);

__global__ void set_array(float *a, float value, int len) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < len) a[idx] = value;
}

__global__ void STREAM_Triad( float *a, float *b, float *c,
                             float scalar, int len) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < len) c[idx] = a[idx]+scalar*b[idx];
}
```

COMPUTE

STORE

ANALYZE

STREAM Triad: Chapel

MPI + OpenMP

```
#include <hpcc.h>
#ifndef _OPENMP
#include <omp.h>
#endif

static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *params,
int myRank, commSize;
int rv, errCount;
MPI_Comm comm = MPI_COMM_WORLD;

MPI_Comm_size( comm, &commSize );
MPI_Comm_rank( comm, &myRank );

rv = HPCC_Stream( params, 0 == myRank );
MPI_Reduce( &rv, &errCount, 1, MPI_
return errCount;

int HPCC_Stream(HPCC_Params *params,
register int j;
double scalar;
VectorSize = HPCC_LocalVectorSize();
a = HPCC_XMALLOC( double, VectorSize );
b = HPCC_XMALLOC( double, VectorSize );
c = HPCC_XMALLOC( double, VectorSize );

if (!a || !b || !c) {
    if (c) HPCC_free(c);
    if (b) HPCC_free(b);
    if (a) HPCC_free(a);
    if (doIO) {
        fprintf( outFile, "Failed to allocate memory (%d).\n" VectorSize );
        cudaThreadSynchronize();
        fclose( outFile );
    }
}
```

Chapel

```
config const m = 1000,
alpha = 3.0;

const ProblemSpace = {1..m} dmapped ...;

var A, B, C: [ProblemSpace] real;

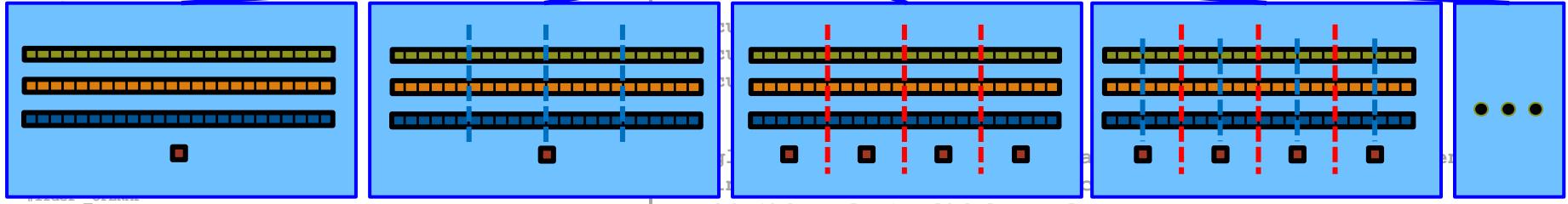
B = 2.0;
C = 1.0;

A = B + alpha * C;
```

The special sauce:
How should this index set—and any arrays and computations over it—be mapped to the system?

```
N);
N);

_c, d_a, scalar, N);
```



Philosophy: Good, *top-down* language design can tease system-specific implementation details away from an algorithm, permitting the compiler, runtime, applied scientist, and HPC expert to each focus on their strengths.

Outline

- ✓ Chapel Motivation and Background
- Chapel in a Nutshell
 - Chapel Project: Status
 - Chapel Resources

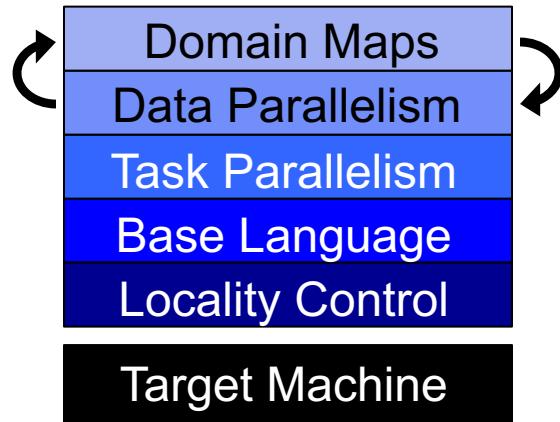


Chapel's Multiresolution Philosophy

Multiresolution Design: Support multiple tiers of features

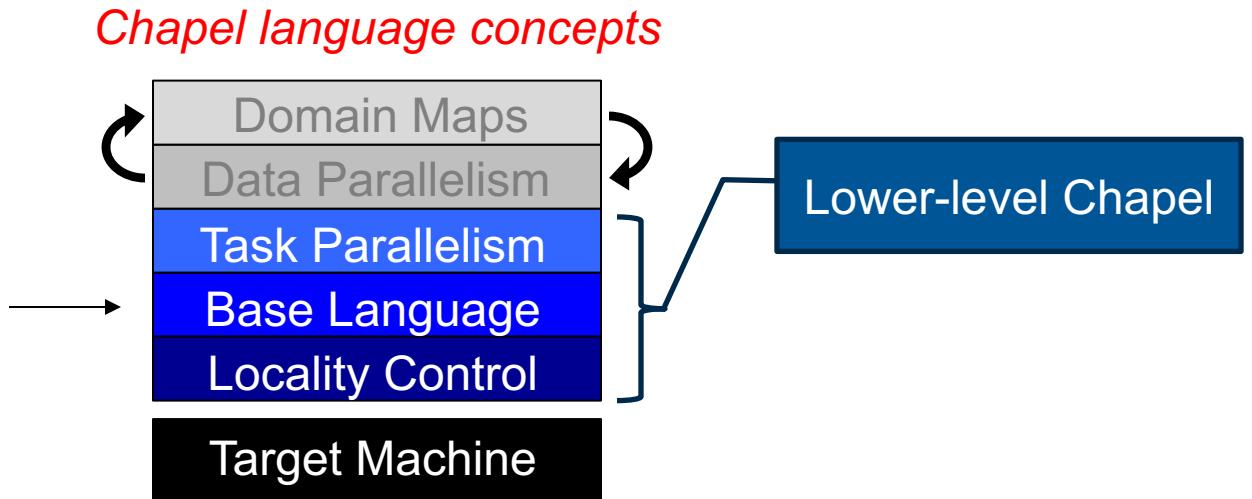
- higher levels for programmability, productivity
- lower levels for greater degrees of control

Chapel language concepts



- build the higher-level concepts in terms of the lower
- permit the user to intermix layers arbitrarily

Lower-Level Features



Base Language Features: Fibonacci Example

```
iter fib(n) {
    var current = 0,
        next = 1;

    for i in 1..n {
        yield current;
        current += next;
        current <=> next;
    }
}
```

```
for (i,f) in zip(0..#n, fib(n)) do
    writeln("fib #", i, " is ", f);
```

```
fib #0 is 0
fib #1 is 1
fib #2 is 1
fib #3 is 2
fib #4 is 3
fib #5 is 5
fib #6 is 8
...
```

Base Language Features: Fibonacci Example

iterators

```
iter fib(n) {
    var current = 0,
        next = 1;

    for i in 1..n {
        yield current;
        current += next;
        current <=> next;
    }
}
```

```
for (i,f) in zip(0..#n, fib(n)) do
    writeln("fib #", i, " is ", f);
```

```
fib #0 is 0
fib #1 is 1
fib #2 is 1
fib #3 is 2
fib #4 is 3
fib #5 is 5
fib #6 is 8
...
```

Base Language Features: Fibonacci Example

```
iter fib(n) {
    var current = 0
        next = 1

    for i in 1..n {
        yield current;
        current += next;
        current <=> next;
    }
}
```

built-in range types
and operators

```
for (i,f) in zip(0..#n, fib(n)) do
    writeln("fib #", i, " is ", f);
```

```
fib #0 is 0
fib #1 is 1
fib #2 is 1
fib #3 is 2
fib #4 is 3
fib #5 is 5
fib #6 is 8
...
```

Base Language Features: Fibonacci Example

```
iter fib(n) {
    var current = 0,
        next = 1;

    for i in 1..n {
        yield current;
        current += next;
        current <=> next;
    }
}
```

```
for (i,f) in zip(0..#n, fib(n)) do
    writeln("fib #", i, " is ", f);
```

fib #0 is 0
fib #1 is 1
fib #2 is 1
fib #3 is 2
fib #4 is 3
fib #5 is 5
fib #6 is 8
...

zippered iteration

Base Language Features: Fibonacci Example

```
iter fib(n) {
    var current = 0,
        next = 1;

    for i in 1..n {
        yield current;
        current += next;
        current <=> next;
    }
}
```

tuples

```
for (i,f) in zip(0..#n, fib(n)) do
    writeln("fib #", i, " is ", f);
```

```
fib #0 is 0
fib #1 is 1
fib #2 is 1
fib #3 is 2
fib #4 is 3
fib #5 is 5
fib #6 is 8
...
```

Base Language Features: Fibonacci Example

Static Type Inference for:

- arguments
- return types
- variables

```
iter fib(n) {
    var current = 0,
        next = 1;

    for i in 1..n {
        yield current;
        current += next;
        current <=> next;
    }
}
```

```
for (i,f) in zip(0..#n, fib(n)) do
    writeln("fib #", i, " is ", f);
```

```

fib #0 is 0
fib #1 is 1
fib #2 is 1
fib #3 is 2
fib #4 is 3
fib #5 is 5
fib #6 is 8
...

```

Base Language Features: Fibonacci Example

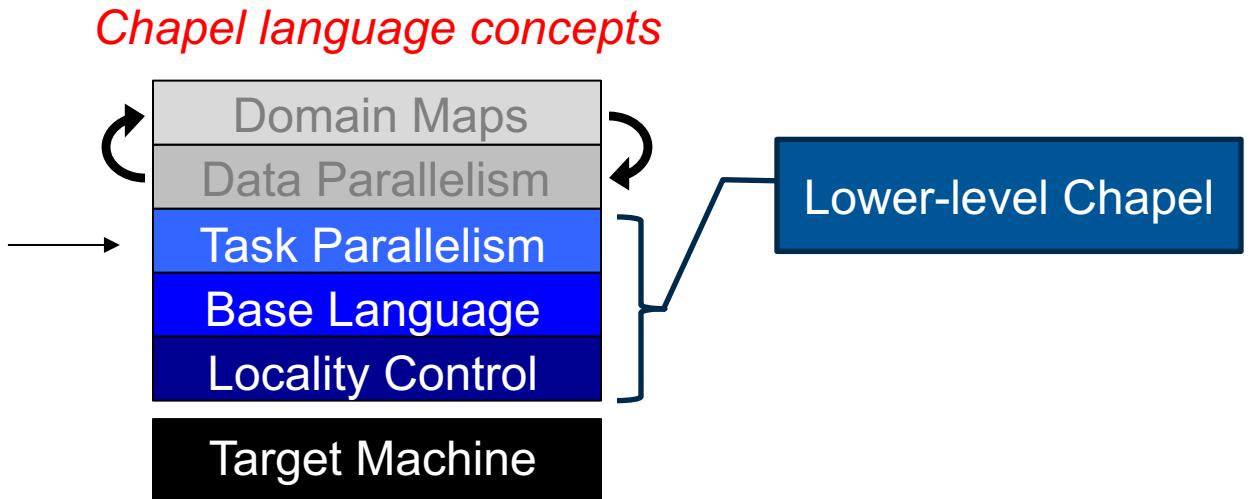
```
iter fib(n) {
    var current = 0,
        next = 1;

    for i in 1..n {
        yield current;
        current += next;
        current <=> next;
    }
}
```

```
for (i,f) in zip(0..#n, fib(n)) do
    writeln("fib #", i, " is ", f);
```

```
fib #0 is 0
fib #1 is 1
fib #2 is 1
fib #3 is 2
fib #4 is 3
fib #5 is 5
fib #6 is 8
...
```

Lower-Level Features



Task Parallelism

beginTask.chpl

```
begin writeln("Hello!");
writeln("Goodbye...");
```

```
prompt> chpl beginTask.chpl -o beginTask
prompt> ./beginTask
Hello!
Goodbye...
prompt> ./beginTask
Goodbye...
Hello!
```

Task Parallelism

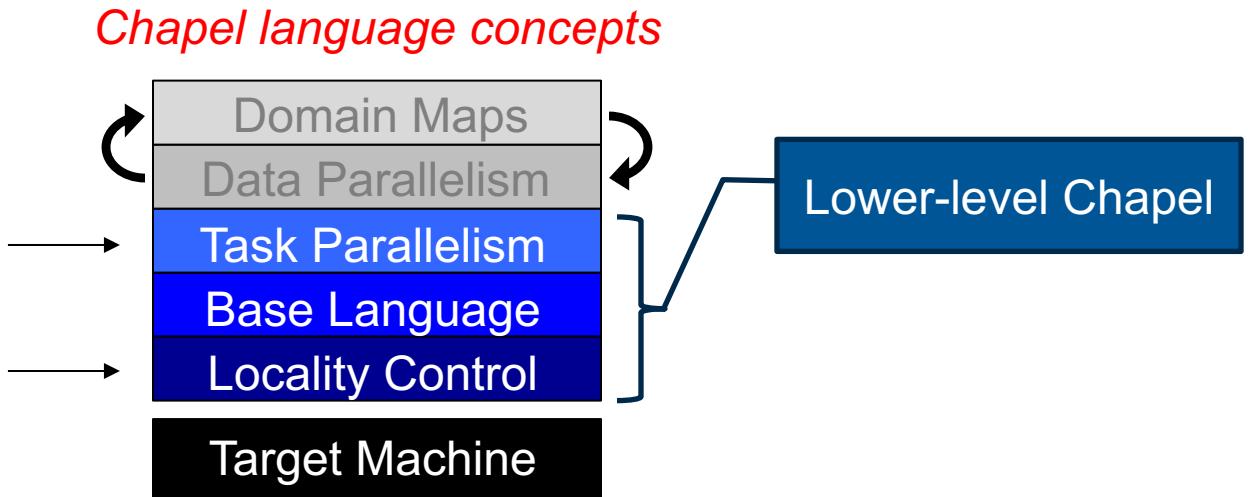
Creates a new task

beginTask.chpl

```
begin writeln("Hello!");  
writeln("Goodbye...");
```

```
prompt> chpl beginTask.chpl -o beginTask  
prompt> ./beginTask  
Hello!  
Goodbye...  
prompt> ./beginTask  
Goodbye...  
Hello!
```

Lower-Level Features



Task Parallelism & Locality Control

taskParallel.chpl

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

```
prompt> chpl taskParallel.chpl -o taskParallel
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032
```



Task Parallelism & Locality Control

High-Level
Task Parallelism

taskParallel.chpl

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

```
prompt> chpl taskParallel.chpl -o taskParallel
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032
```



COMPUTE

STORE

ANALYZE

Task Parallelism & Locality Control

Abstraction of System Resources

taskParallel.chpl

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

```
prompt> chpl taskParallel.chpl -o taskParallel
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032
```



Task Parallelism & Locality Control

Control of Locality/Affinity

taskParallel.chpl

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

```
prompt> chpl taskParallel.chpl -o taskParallel
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032
```



COMPUTE

| STORE

| ANALYZE

Task Parallelism & Locality Control

Abstraction of System Resources

taskParallel.chpl

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

```
prompt> chpl taskParallel.chpl -o taskParallel
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032
```

Task Parallelism & Locality Control

High-Level
Task Parallelism

taskParallel.chpl

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

```
prompt> chpl taskParallel.chpl -o taskParallel
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032
```



Parallelism and Locality: Orthogonal in Chapel

- This is a **parallel**, but local program:

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

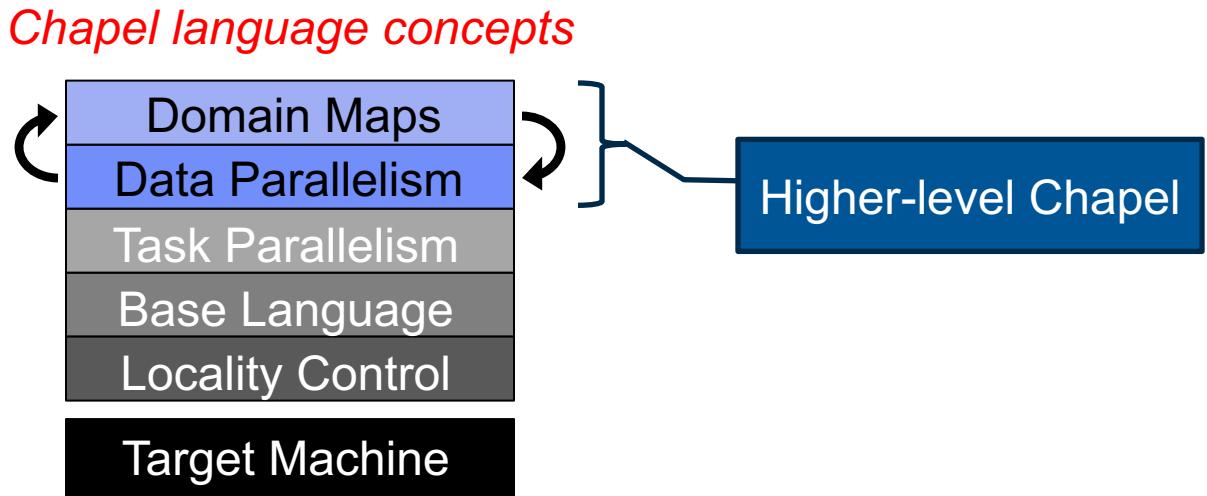
- This is a **distributed**, but serial program:

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

- This is a **distributed parallel** program:

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

Higher-Level Features



Data Parallelism

dataParallel.chpl

```
config const n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i,j) in D do
    A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl -o dataParallel
prompt> ./dataParallel --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```



COMPUTE | STORE | ANALYZE

Data Parallelism

Domains (Index Sets)

dataParallel.chpl

```
config const n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i,j) in D do
    A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl -o dataParallel
prompt> ./dataParallel --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```



COMPUTE | STORE | ANALYZE

Data Parallelism

Arrays

dataParallel.chpl

```
config const n = 1000;  
  
var D = {1..n, 1..n};  
  
var A: [D] real;  
  
forall (i,j) in D do  
    A[i,j] = i + (j - 0.5)/n;  
  
writeln(A);
```

```
prompt> chpl dataParallel.chpl -o dataParallel  
prompt> ./dataParallel --n=5  
1.1 1.3 1.5 1.7 1.9  
2.1 2.3 2.5 2.7 2.9  
3.1 3.3 3.5 3.7 3.9  
4.1 4.3 4.5 4.7 4.9  
5.1 5.3 5.5 5.7 5.9
```



Data Parallelism

Data-Parallel Forall Loops

dataParallel.chpl

```
config const n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i,j) in D do
    A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl -o dataParallel
prompt> ./dataParallel --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```



Distributed Data Parallelism

Domain Maps
(Map Data Parallelism to the System)

dataParallel.chpl

```
use CyclicDist;
config const n = 1000;
var D = {1..n, 1..n}
        dmapped Cyclic(startIdx = (1,1));
var A: [D] real;
forall (i,j) in D do
    A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl -o dataParallel
prompt> ./dataParallel --n=5 --numLocales=4
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```



COMPUTE

STORE

ANALYZE

Distributed Data Parallelism

Distributions

- BlockCycDist
- BlockDist
- CyclicDist
- DimensionalDist2D
- PrivateDist
- ReplicatedDist
- SparseBlockDist
- StencilDist

Layouts

- CSR

dataParallel.chpl

```
use CyclicDist;
config const n = 1000;
var D = {1..n, 1..n}
      dmapped Cyclic(startIdx = (1,1));
var A: [D] real;
forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl -o dataParallel
prompt> ./dataParallel --n=5 --numLocales=4
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

Intel Xeon Phi (“KNL”) locale model

- Chapel can target KNL’s MCDRAM via on-clauses

```
on here.highBandwidthMemory() {  
    x = new myClass();           // placed in MCDRAM  
    ...  
    on here.defaultMemory() {  
        y = new myClass();     // placed in DDR  
        ...  
    }  
}  
  
on y.locale.highBandwidthMemory() {  
    z = new myClass();           // same locale as y, but using MCDRAM  
    ...  
}
```

Intel Xeon Phi (“KNL”) locale model

- Working towards distributions that can target MCDRAM

on here
x = r
...
on he
config const m = 1000,
alpha = 3.0;

Y =
const ProblemSpace = {1..m}
 dmapped KNLDist(...);

}
}
var A, B, C: [ProblemSpace] **real**;

B = 2.0;
C = 1.0;

on y.lo
z = r
A = B + alpha * C;
...
}

Chapel

Outline

- ✓ Chapel Motivation and Background
- ✓ Chapel in a Nutshell
- Chapel Project: Status
- Chapel Resources



Chapel is a Work-in-Progress

- **Currently being picked up by early adopters**
 - Users who try it like what they see
- **Most features are functional and working well**
 - some areas are under active development, particularly:
 - Initializers
 - Error handling
- **Performance is improving**
 - shared memory performance is competitive with C+OpenMP
 - some distributed memory applications now competitive with C+MPI
 - many cases continue to need more work



Single-Locale Performance



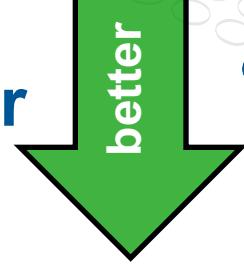
COMPUTE

| STORE

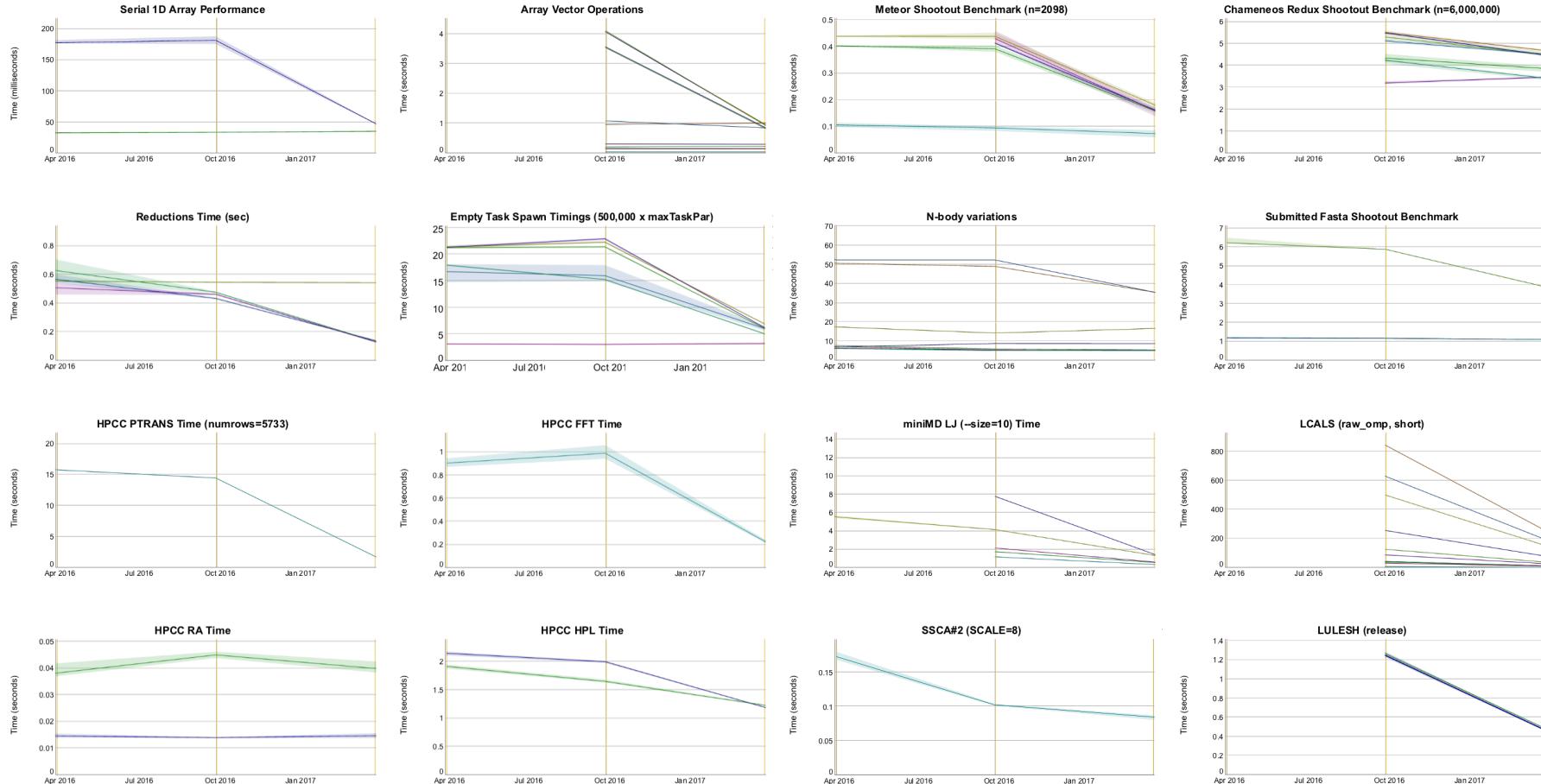
| ANALYZE

Single-Locale Performance: the past year

better



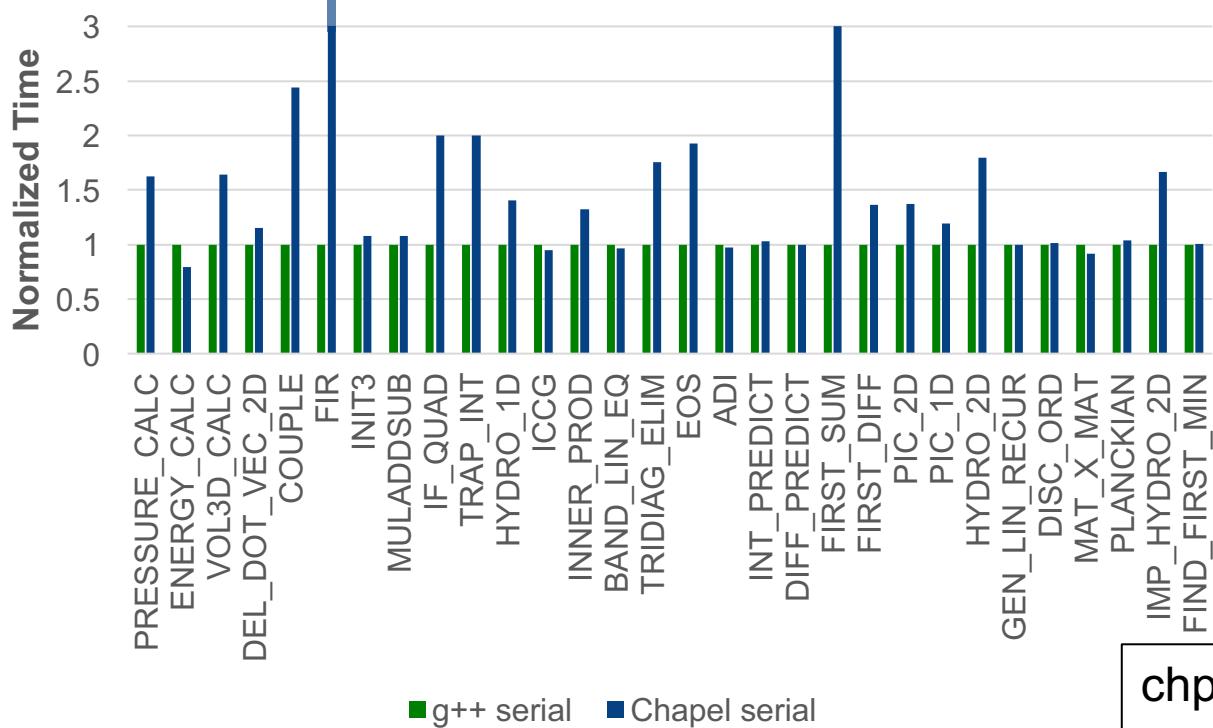
- Overall, single-locale performance improved dramatically



LCALS: Serial Timings, Chapel 1.13.0

Long problem size

(Similar results for medium and short problem sizes)



chpl --fast
--no-ieee-float

g++ -Ofast -fopenmp

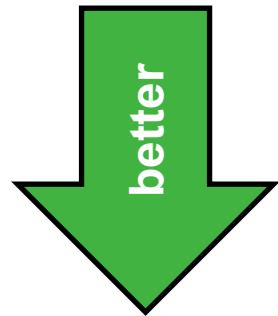
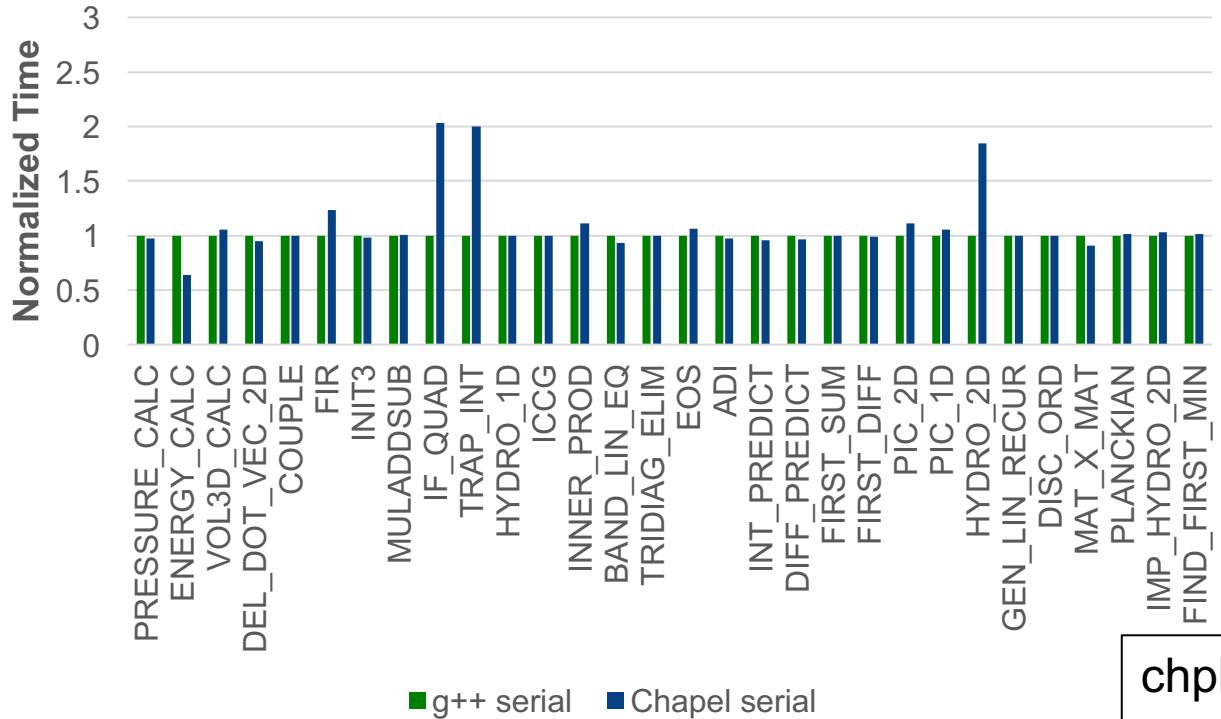
LCALS: Serial Timings, Chapel 1.14.0

Long problem size

(Similar results for medium and short problem sizes)

Serial Chapel vs g++

Normalized time – serial reference is 1.0

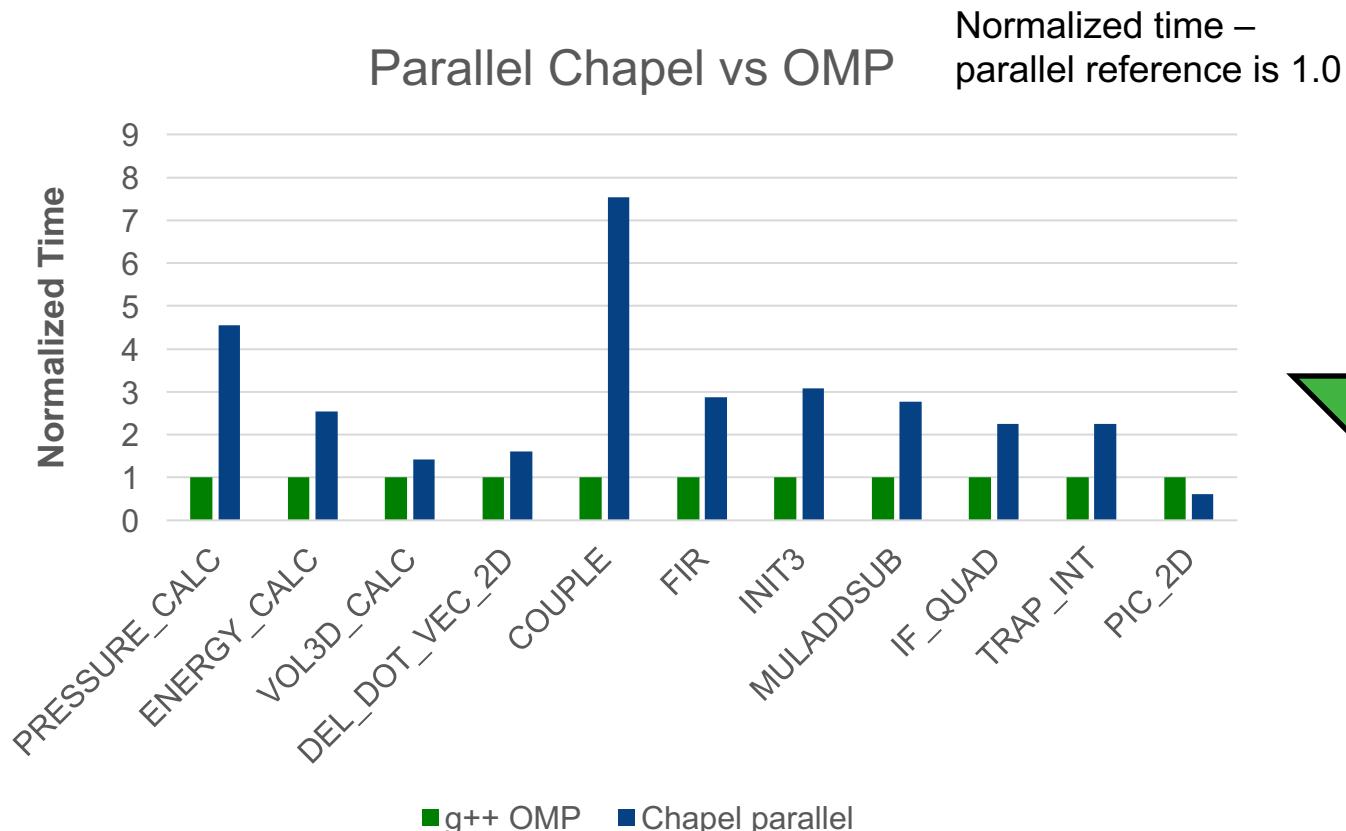


chpl --fast
--no-ieee-float

g++ -Ofast -fopenmp

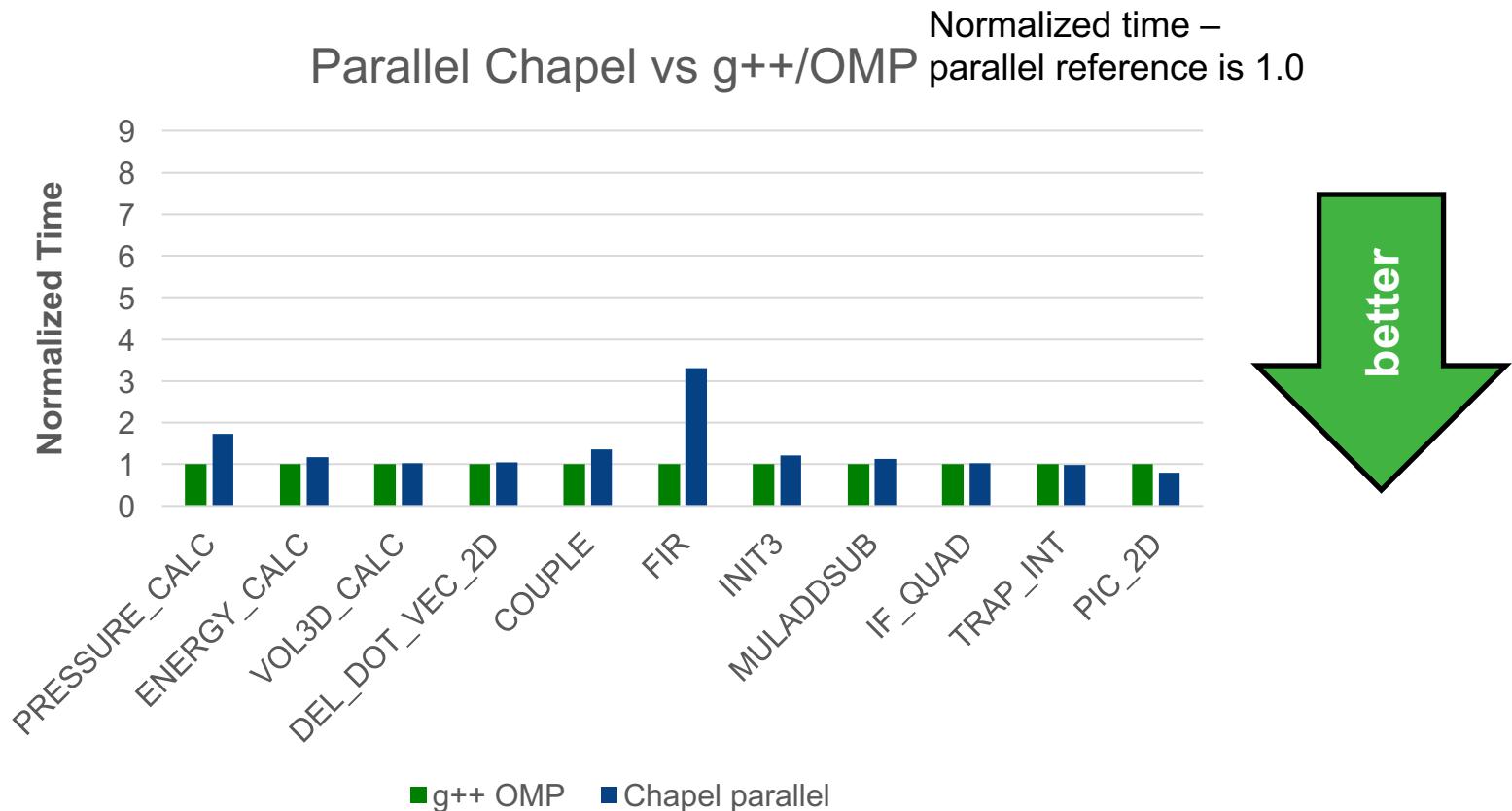
LCALS: Parallel Timings, Chapel 1.14.0

- Parallel variants still lagged behind the reference in 1.14
 - between 1.5X and 8X slower for long problem size



LCALS: Parallel Timings, Chapel 1.15.0

- Chapel 1.15 closed the gap significantly
 - ~3-4x speedup: on par or very close to reference for most kernels



Multi-Locale Performance



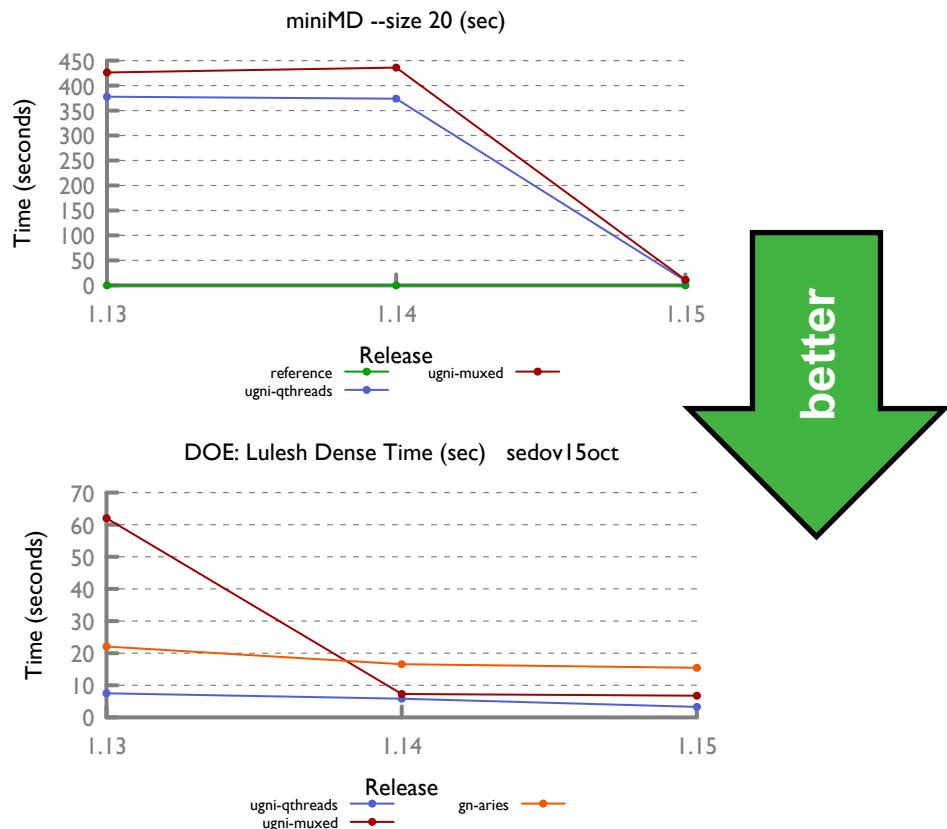
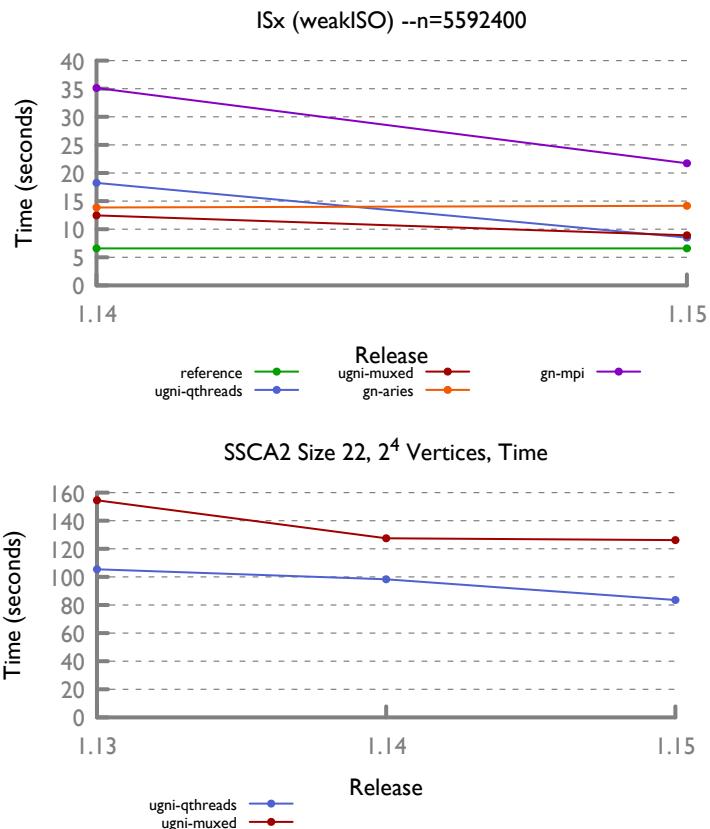
COMPUTE

| STORE

| ANALYZE

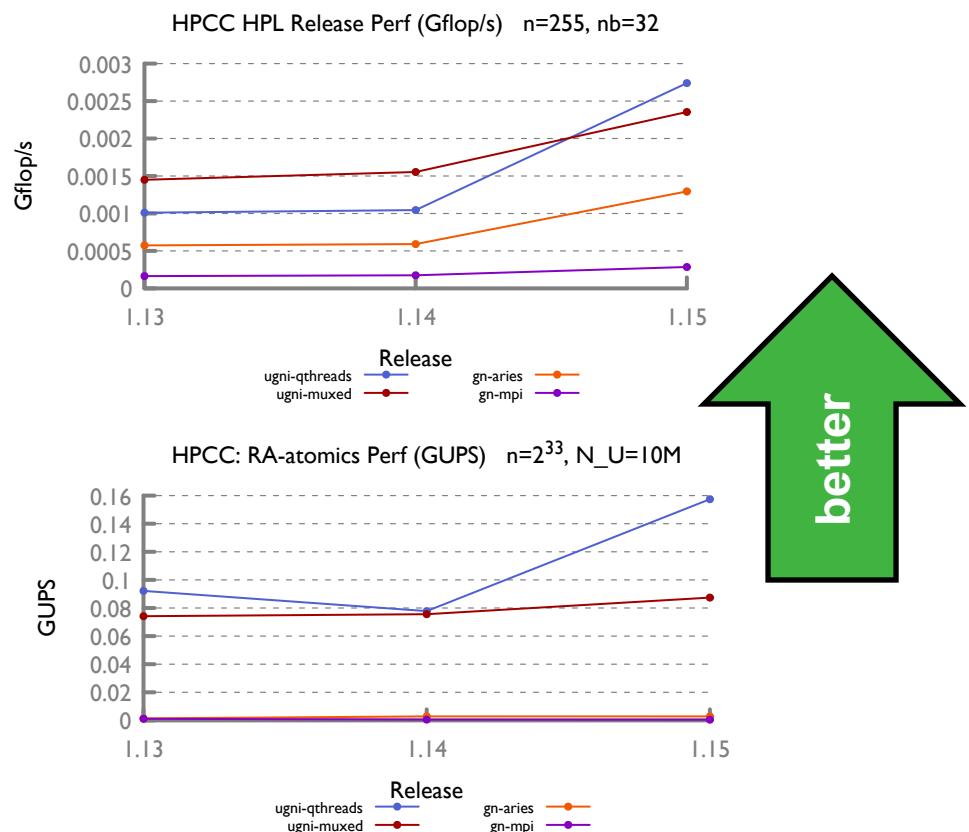
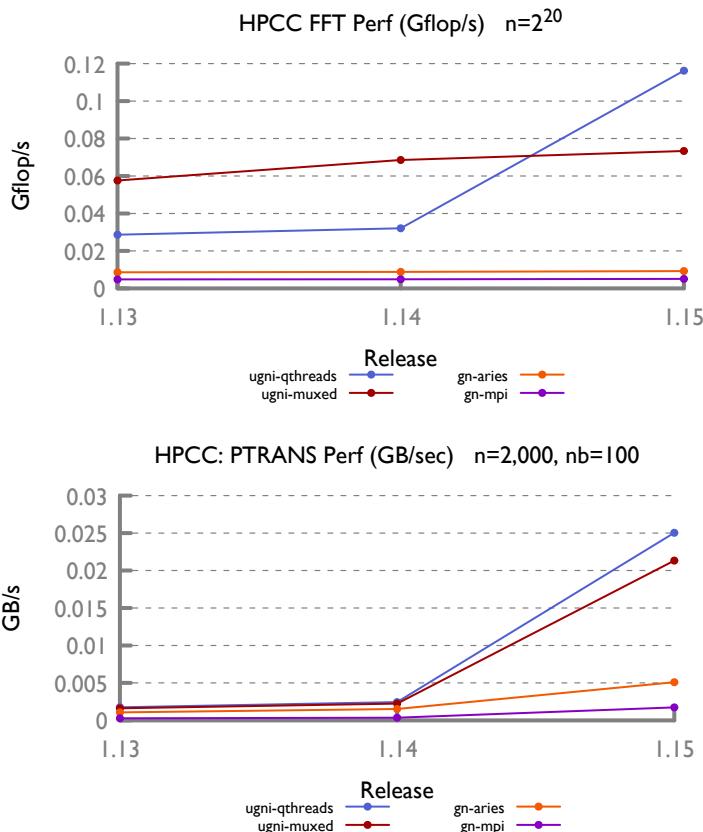
Multi-locale Performance

- Significant multi-locale performance improvements

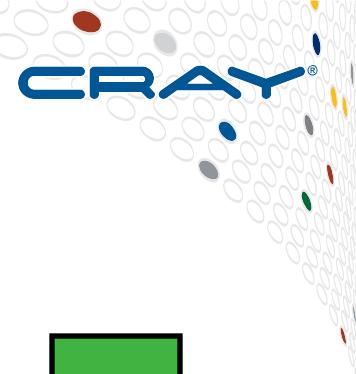


Multi-locale Performance

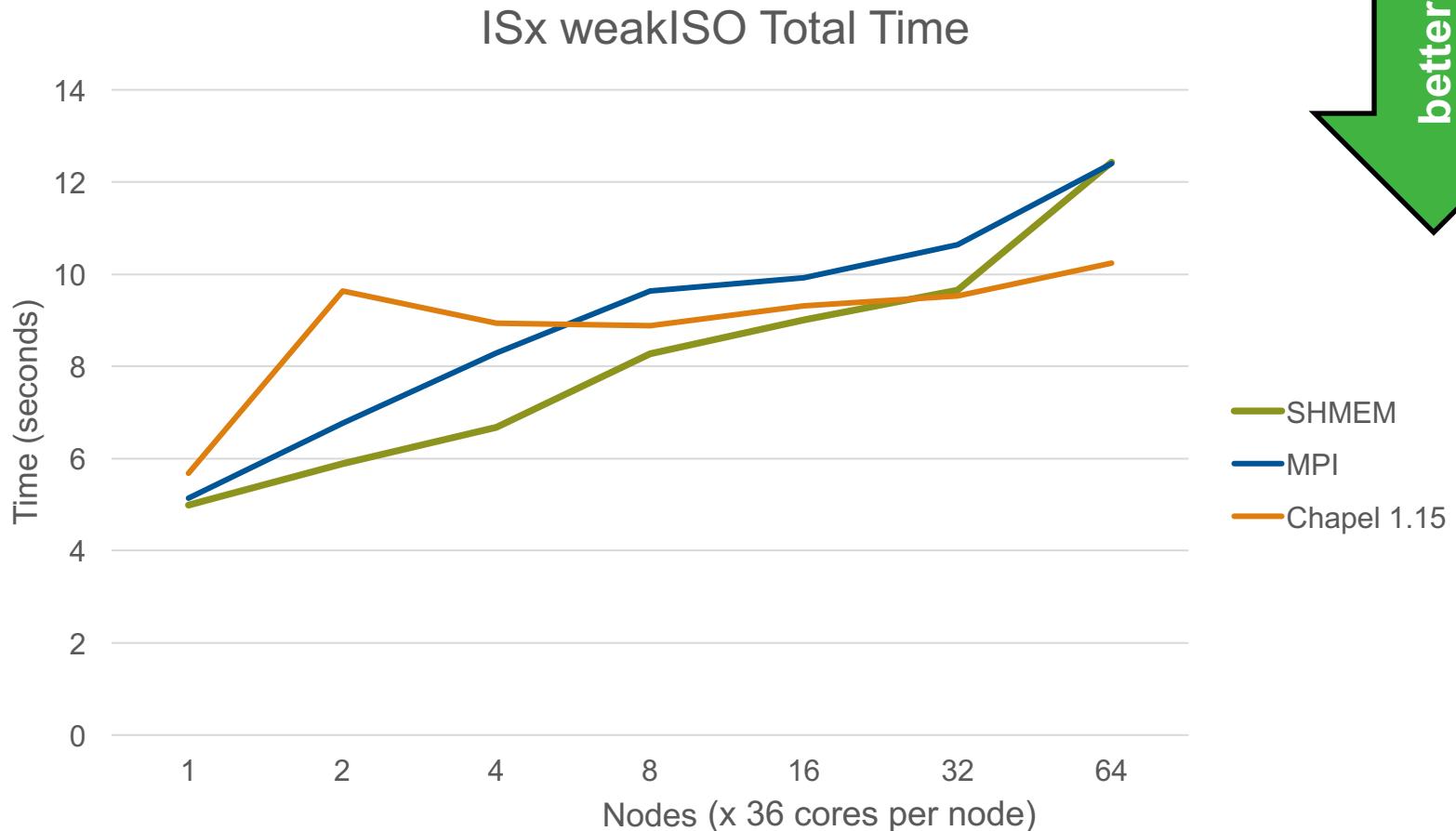
- Significant multi-locale performance improvements



ISx Execution Time: MPI, SHMEM, Chapel 1.15



- 64 nodes on Cray XC



COMPUTE

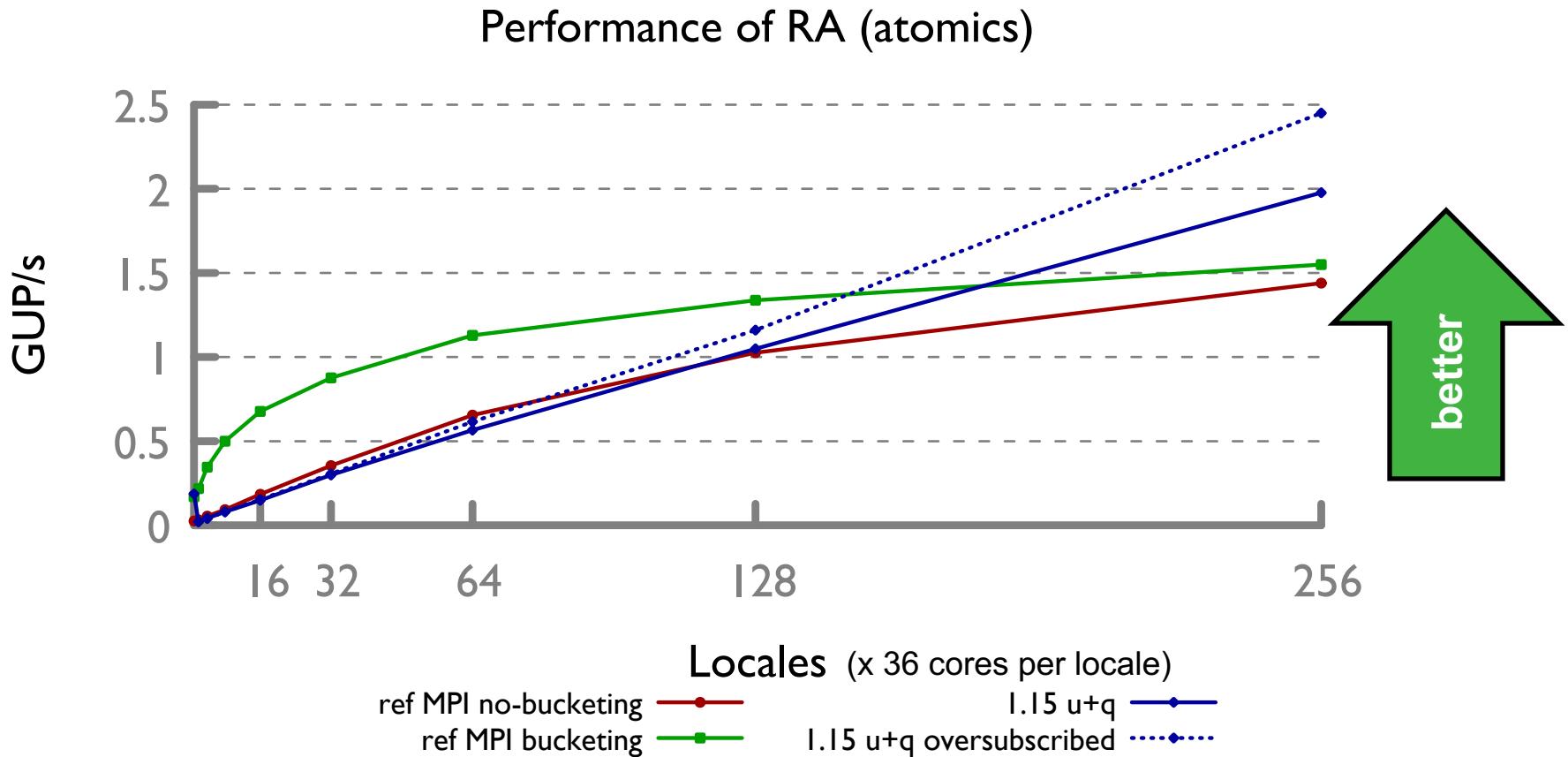
|

STORE

|

ANALYZE

RA Performance: Chapel vs. MPI



Performance: Summary

Summary:

- performance has been dramatically improving
- shared memory performance is competitive with C+OpenMP
- some distributed memory applications now competitive with C+MPI
 - many applications need more work

Next steps:

- **Multi-locale:**
 - benchmark-driven performance and scalability improvements
 - particularly for DOE proxy apps, stencils codes, PRK benchmarks, and others
- **Single-locale:**
 - vectorization

Our #1 Challenge

- How to grow the user and developer communities?
- How to encourage people to look at Chapel again?
 - overcome impressions made in our young, awkward years...

'Scientific computing communities are very wary of new technologies (it took 10+ years for Python to start getting any traction), with the usual, self-fulfilling, fear being "what if it goes away?"'

- Jonathan Dursi, from *Should I Use Chapel or Julia for my next project?*

This Talk's Takeaways

- **Chapel is a modern and productive parallel language**
 - Well suited for current and emerging HPC and commodity systems
 - Ready for early adopters now
 - Performance and stability are increasing with each release

'There are other research projects in this area - productive, performant, parallel computing languages for distributed-memory scientific computing.

But Chapel, especially now with 1.15, is a mature product.

It is crossing the barrier of Fast Enough'

- Jonathan Dursi, from *Chapel's Home in the New Landscape of Scientific Frameworks*

Outline

- ✓ Chapel Motivation and Background
- ✓ Chapel in a Nutshell
- ✓ Chapel Project: Status
- Chapel Resources



Chapel Websites

Project page: <http://chapel.cray.com>

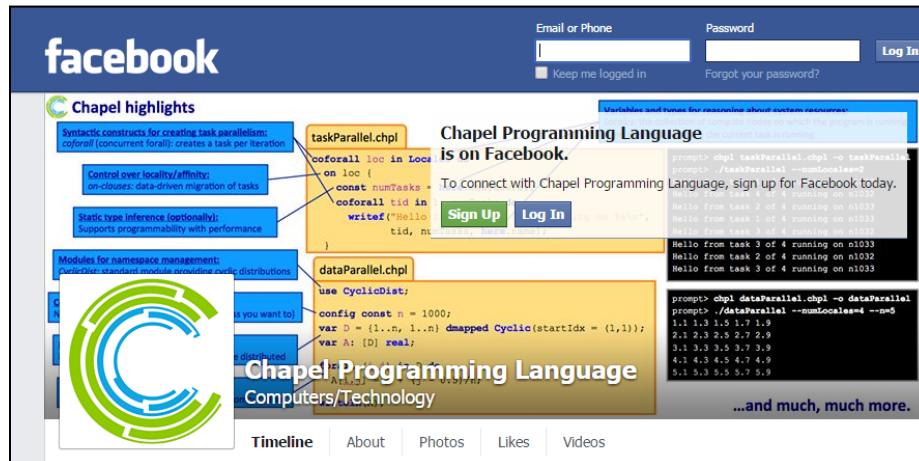
- overview, papers, presentations, language spec, ...

GitHub: <https://github.com/chapel-lang>

- download Chapel; browse source repository; contribute code

Facebook: <https://www.facebook.com/ChapelLanguage>

Twitter: <https://twitter.com/ChapelLanguage>



The screenshot shows the Chapel Programming Language Facebook page. It features a large green and blue 'C' logo. The page highlights include:

- Syntactic constructs for creating task parallelism: `coforall` (concurrent forall), creates a task per iteration.
- Control over locality/affinity: on-edges, data-driven migration of tasks.
- Static type inference (optionally), supports programmability with performance.
- Modules for namespace management: provides a clean way to manage dependencies and distributions.
- Task parallelism: what you want to do.
- Computers/Technology: Chapel Programming Language.

A central box displays the message: "Chapel Programming Language is on Facebook. To connect with Chapel Programming Language, sign up for Facebook today." Below this are snippets of Chapel code and their corresponding outputs:

```

taskParallel.chpl
coforall loc in Locs
on loc {
    const numTasks = 4;
    coforall tid in writef("Hello
tid, n = %d", tid, n);
}

dataParallel.chpl
use CyclicDist;
config const n = 1000;
var D = (1..n, 1..n) dmapped Cyclic(startIdx = (1,1));
var A: [1..n] real;

Hello from task 0 of 4 running on n1033
Hello from task 1 of 4 running on n1032
Hello from task 2 of 4 running on n1031
Hello from task 3 of 4 running on n1030

```

...and much, much more.



The screenshot shows the Chapel Language Twitter account (@ChapelLanguage). The profile picture is the same green and blue 'C' logo. The bio reads: "Chapel is a productive parallel programming language designed for large-scale computing whose development is being led by @cray_inc". The account has 4 tweets, 1 following, and 19 followers. A tweet from March 8, 2018, states: "Two weeks ago, we switched Chapel's memory allocator to jemalloc. Performance wins ensued: chapel.sourceforge.net/perf/chapcs/?s...". Below the tweet is a line graph titled "Binary Trees Shootout Benchmark (n=20)" showing time in seconds for three methods: binary-tees_xy, binarytrees, and binary-trees-newWhiskering.



COMPUTE

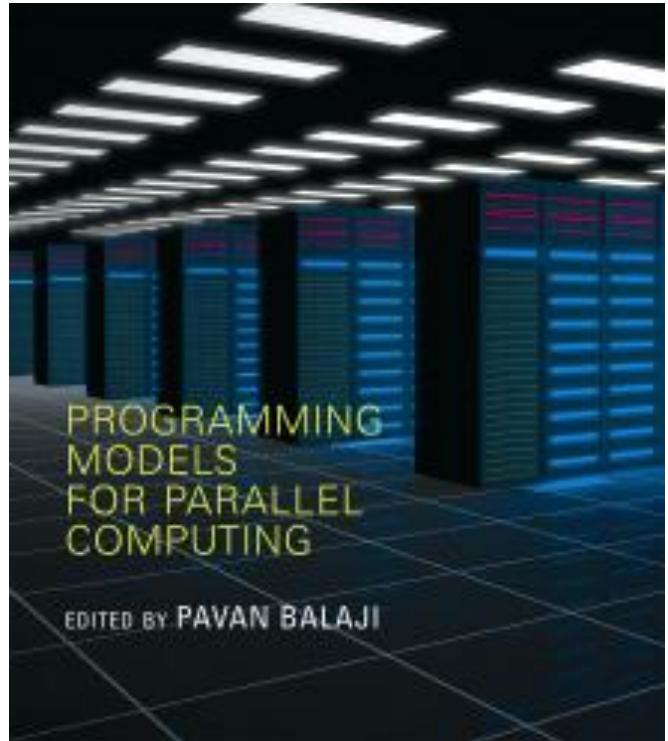
STORE

ANALYZE

Suggested Reading

Chapel chapter from *Programming Models for Parallel Computing*

- a detailed overview of Chapel's history, motivating themes, features
- edited by Pavan Balaji, published by MIT Press, November 2015
- chapter is now also available online



Other Chapel papers/publications available at <http://chapel.cray.com/papers.html>

Chapel Blog Articles

[Chapel: Productive Parallel Programming](#), Cray Blog, May 2013.

- *a short-and-sweet introduction to Chapel*

[Chapel Springs into a Summer of Code](#), Cray Blog, April 2016.

- *a run-down of some current events*

[Six Ways to Say “Hello” in Chapel](#) (parts [1](#), [2](#), [3](#)), Cray Blog, Sep-Oct 2015.

- *a series of articles illustrating the basics of parallelism and locality in Chapel*

[Why Chapel?](#) (parts [1](#), [2](#), [3](#)), Cray Blog, Jun-Oct 2014.

- *a series of articles answering common questions about why we are pursuing Chapel in spite of the inherent challenges*

[\[Ten\] Myths About Scalable Programming Languages](#), IEEE TCSC Blog (index available on [chapel.cray.com](#) “blog articles” page), Apr-Nov 2012.

- *a series of technical opinion pieces designed to argue against standard reasons given for not developing high-level parallel languages*

Mailing Lists

low-traffic (read-only):

chapel-announce@lists.sourceforge.net: announcements about Chapel

community lists:

chapel-users@lists.sourceforge.net: user-oriented discussion list

chapel-developers@lists.sourceforge.net: developer discussions

chapel-education@lists.sourceforge.net: educator discussions

(subscribe at SourceForge: <http://sourceforge.net/p/chapel/mailman/>)

To contact the Cray team:

chapel_info@cray.com: contact the team at Cray

chapel_bugs@cray.com: for reporting non-public bugs

Other Community Resources

IRC channels ([freenode.net](#)):

#chapel: user-oriented discussions

#chapel-developers: developer discussions

Stack Overflow

[stackoverflow.com](#): [chapel] tag monitored by core team

GitHub Issues:

[github.com/chapel-lang/chapel/issues](#): bug reports & feature requests



Chapel: Productive Parallel Programming at Scale

Questions?



PARIS 2017

COMPUTE

| STORE

| ANALYZE

Legal Disclaimer

Information in this document is provided in connection with Cray Inc. products. No license, express or implied, to any intellectual property rights is granted by this document.

Cray Inc. may make changes to specifications and product descriptions at any time, without notice.

All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

Cray hardware and software products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Cray uses codenames internally to identify products that are in development and not yet publically announced for release. Customers and other third parties are not authorized by Cray Inc. to use codenames in advertising, promotion or marketing and any use of Cray Inc. internal codenames is at the sole risk of the user.

Performance tests and ratings are measured using specific systems and/or components and reflect the approximate performance of Cray Inc. products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, and URIKA. The following are trademarks of Cray Inc.: ACE, APPRENTICE2, CHAPEL, CLUSTER CONNECT, CRAYPAT, CRAYPORT, ECOPHLEX, LIBSCI, NODEKARE, THREADSTORM. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT, and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used in this document are the property of their respective owners.





CRAY
THE SUPERCOMPUTER COMPANY

