# Chapel:
## Productive Parallel Programming at Scale

**Michael Ferguson, Chapel Team, Cray Inc.**

**Guest Lecture at CMSC 441**

**December 7th, 2017**

# 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.

# What You Will Learn

- **Concepts of distributed parallel programming models**
  - MPI, SPMD
  - CUDA, SIMT
  - PGAS
- **What is the Chapel programming language?**
  - Learn enough to get started writing programs
- **How does the Chapel language fit in to the landscape of parallel programming tools?**

# About Me



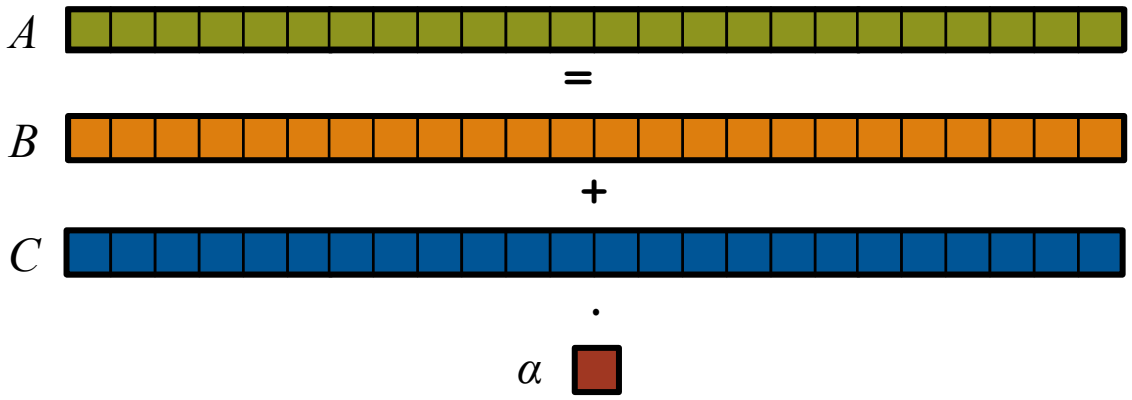
# High Performance Computing (HPC) Programming Models by Example

# 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:**

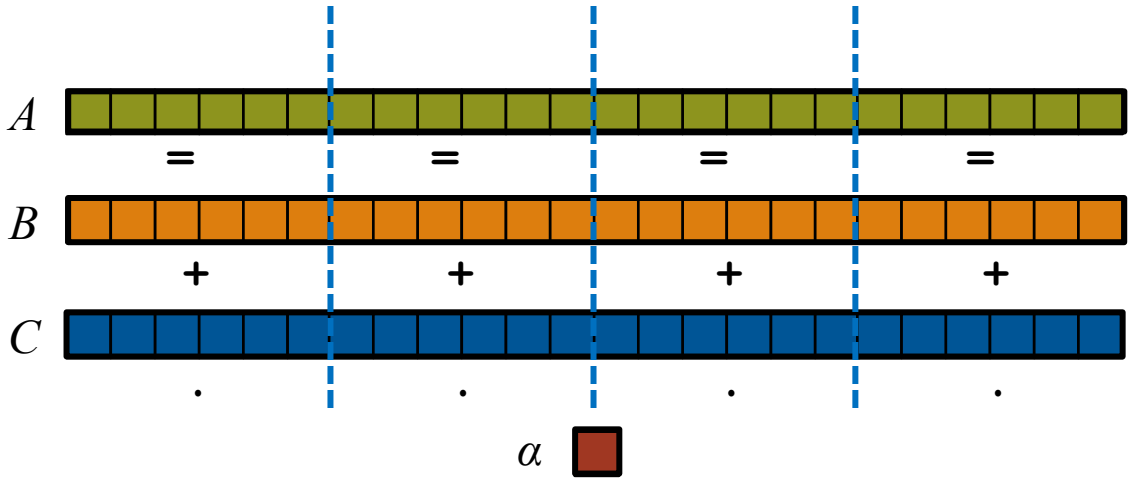


# STREAM Triad: a trivial parallel computation

**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:**

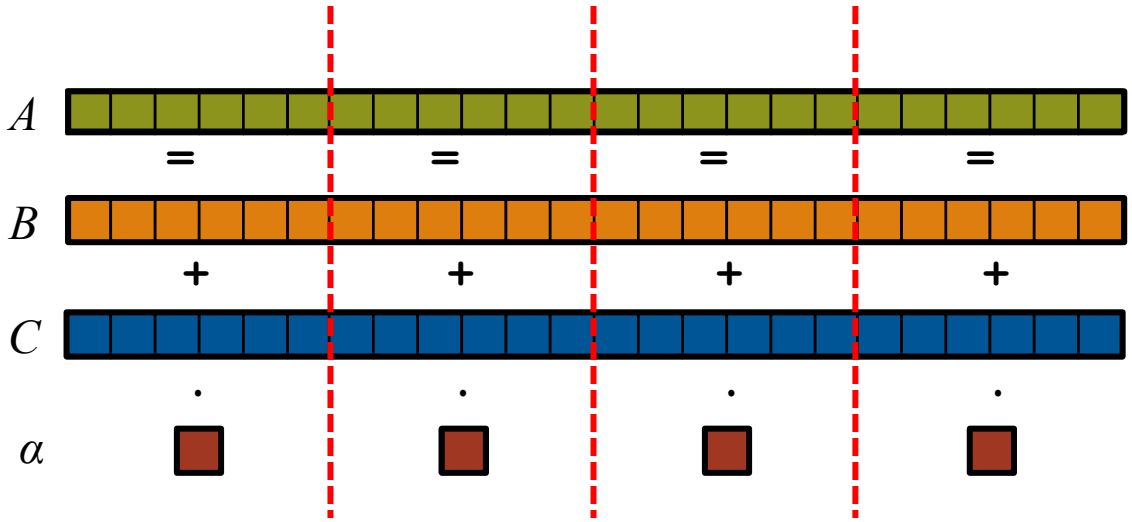


# STREAM Triad: a trivial parallel computation

**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):**

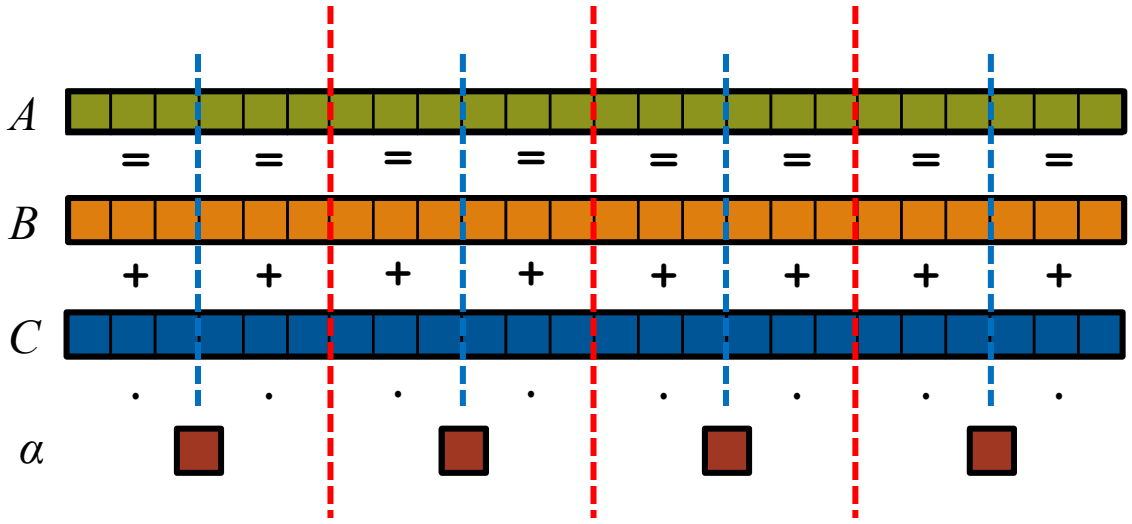


# STREAM Triad: a trivial parallel computation

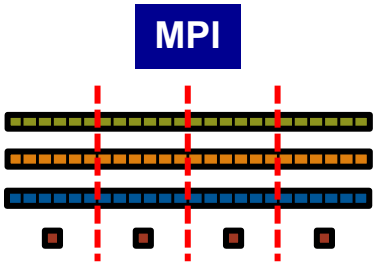
**Given:** $m$-element vectors $A$, $B$, $C$

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

**In pictures, in parallel (distributed memory multicore):**



# STREAM Triad: MPI



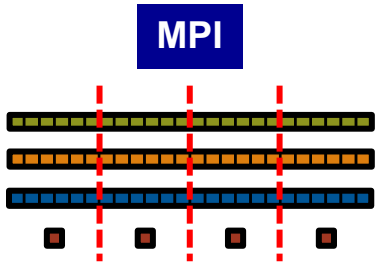

```
#include <hpcc.h>

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;
}
```

```
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);
```

# STREAM Triad: MPI




"SPMD": Single Program Multiple Data

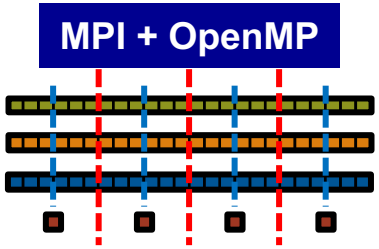
```
#include <hpcc.h>

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 |
  if (c)
  if (b)
  if (a)
  if (do
    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);
```

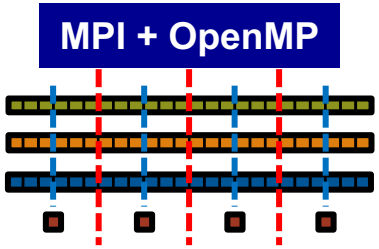
# STREAM Triad: MPI+OpenMP




```
#include <hpcc.h>
#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;
  }

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

  scalar = 3.0;

#ifdef _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

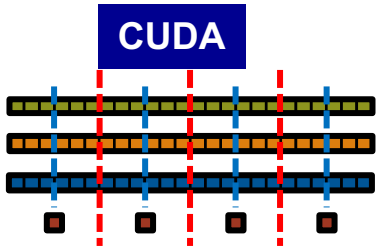
"Hybrid" multiple models of parallelism




```
#include <hpcc.h>
#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;
  }

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

  scalar = 3.0;

#ifdef _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);
```

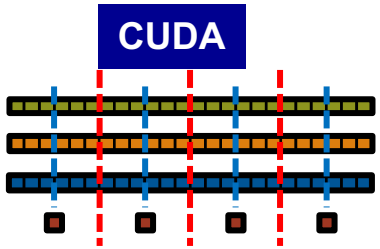
shared memory parallelism

# STREAM Triad: MPI+OpenMP vs. 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];
}
```

# STREAM Triad: MPI+OpenMP vs. CUDA



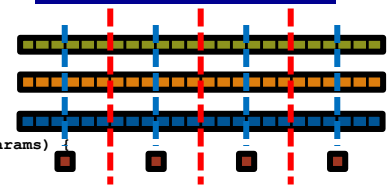

```
#define N       2000000

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];
}
```

"SIMT"
Single Instruction Multiple Threads

kernels

# STREAM Triad: MPI+OpenMP vs. CUDA

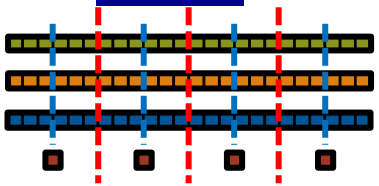
## MPI + OpenMP



```
#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;
  }

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

  scalar = 3.0;

#ifdef _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;
}
```

*Why do we need so much code?*

## 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)+( N % dimBlock.x != 0 ));

  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];
}
```

*Why so many programming models?*

# Why so many programming models?

**HPC tends to approach programming models bottom-up:**

Given a system and its core capabilities…

…provide features that can access the available performance.

- portability? generality? programmability? …not strictly required.

| Type of HW Parallelism | Programming Model | Unit of Parallelism |
|---|---|---|
| Inter-node | MPI | executable |
| Intra-node/multicore | OpenMP / pthreads | iteration/task |
| Instruction-level vectors/threads | pragmas, intrinsics | iteration |
| GPU/accelerator | CUDA / Open[CL\|MP\|ACC] | iteration |

benefits: lots of control; decent generality; easy to implement

downsides: lots of user-managed detail; brittle to changes

# Motivation for Chapel

**Q: Can a single language be…**

…as productive as Python?
…as fast as Fortran?
…as portable as C?
…as scalable as MPI?
…as fun as <your favorite language here>?

**A: We believe so.**

# 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 at scale far more productive

# What does "Productivity" mean to you?

**Recent Graduates:**

"something similar to what I used in school: Python, Matlab, Java, …"

**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:**

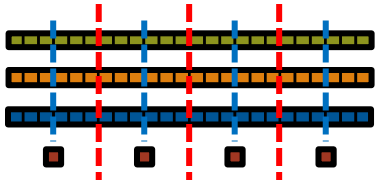
"something that lets me express my parallel computations without having to wrestle with architecture-specific details"

**Chapel Team:**

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

# Rewinding a few slides…

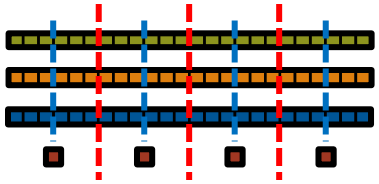
## MPI + OpenMP



```
#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;
  }

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

  scalar = 3.0;

#ifdef _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;
}
```

*Why do we need so much code?*

## 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)+( !(N % dimBlock.x == 0) ? 1 : 0));
  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];
}
```

*Why so many programming models?*

# STREAM Triad: Chapel

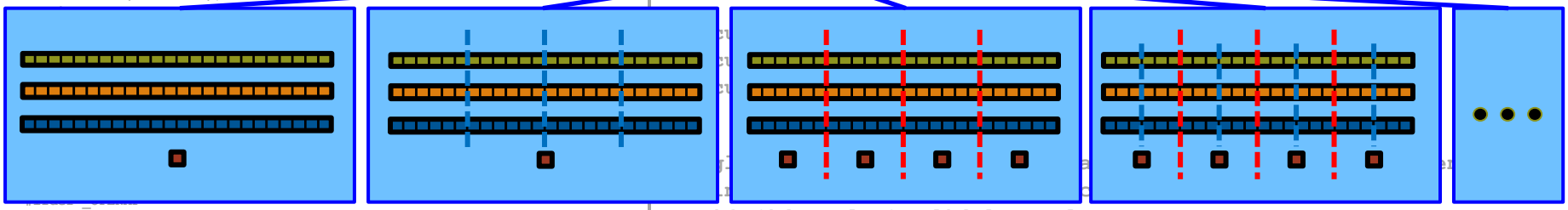
MPI + OpenMP

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

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

int HPCC_StarStream(HPCC_Params *pa
  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 == myR
  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, VectorSi
  b = HPCC_XMALLOC( double, VectorSi
  c = HPCC_XMALLOC( double, VectorSi

  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"
      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 = 3.0;

A = B + alpha * C;
```

the special sauce



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

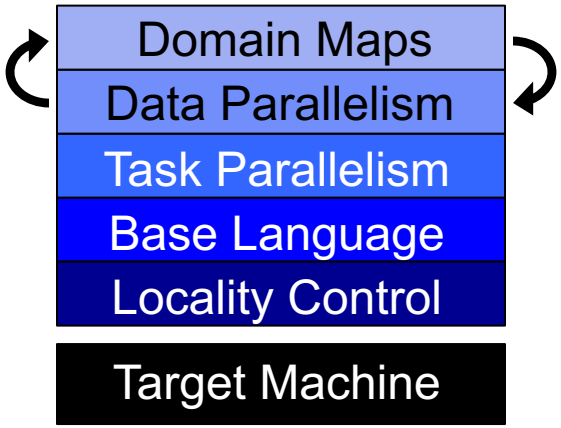
- ✓ Motivation for Chapel
- ➢ Survey of Chapel Concepts
- Chapel Project and Characterizations
- Chapel Resources

# Chapel's Multiresolution Philosophy

***Multiresolution Design:*** **Support multiple tiers of features**

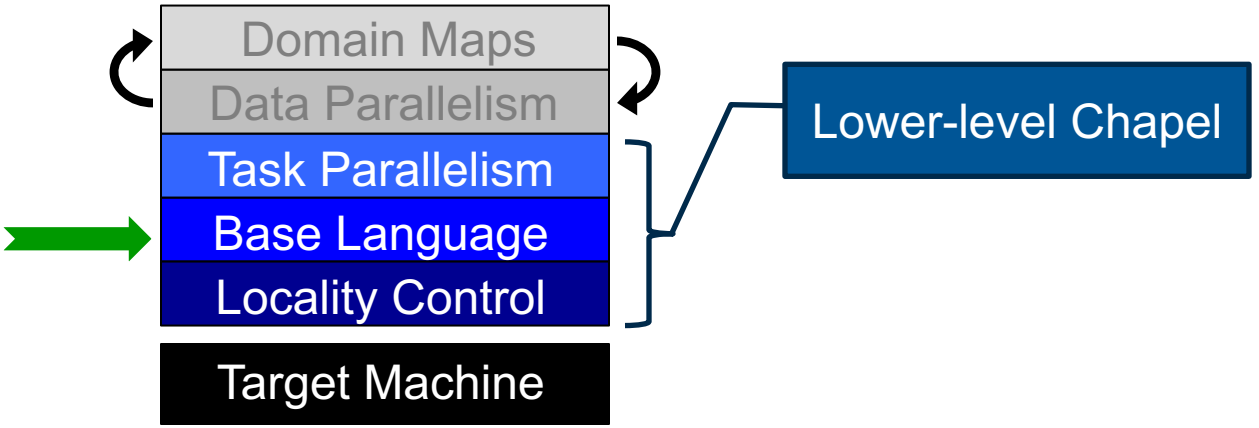
- higher levels for programmability, productivity
- lower levels for greater degrees of control

*Chapel language concepts*




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

# Base Language



# Base Language Features, by example

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

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

```
config const n = 10;

for f in fib(n) do
  writeln(f);
```

```
0
1
1
2
3
5
8
...
```

# Base Language Features, by example

CLU-style iterators

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

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

```
config const n = 10;

for f in fib(n) do
  writeln(f);
```

```
0
1
1
2
3
5
8
...
```

# Base Language Features, by example

Configuration declarations
(to avoid command-line argument parsing)
`./a.out –n=1000000`

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

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

```
config const n = 10;

for f in fib(n) do
  writeln(f);
```

```
0
1
1
2
3
5
8
...
```

# Base Language Features, by 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;
  }
}
```

```
config const n = 10;

for f in fib(n) do
  writeln(f);
```

```
0
1
1
2
3
5
8
...
```

# Base Language Features, by example

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

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

Zippered iteration

```
config const n = 10;

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, by example

Range types and operators

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

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

```
config const n = 10;

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, by example

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

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

tuples

```
config const n = 10;

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, by example

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

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

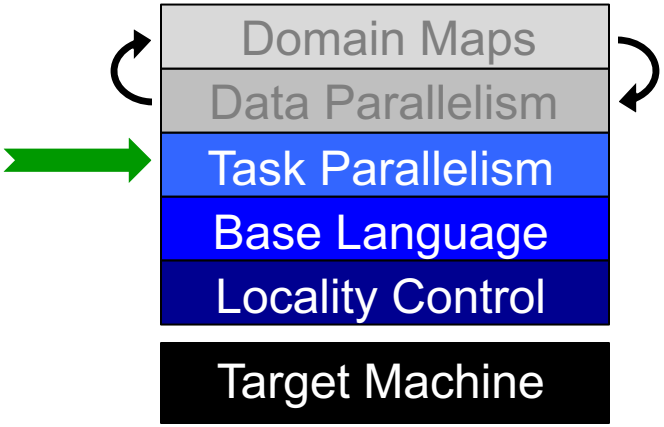
```
config const n = 10;

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
...
```

# Other Base Language Features

- **interoperability features**
- **OOP** (value- and reference-based)
- **overloading, where clauses**
- **argument intents, default values, match-by-name**
- **compile-time features for meta-programming**
  - e.g., compile-time functions to compute types, values; reflection
- **modules** (for namespace management)
- **rank-independent programming features**
- **…**

# Task Parallelism



# Task Parallelism: Begin Statements

```
// create a fire-and-forget task for a statement
begin writeln("hello world");
writeln("goodbye");
```

## Possible outputs:

```
hello world
goodbye
```

```
goodbye
hello world
```

# Task Parallelism: Coforall Loops

```
// create a task per iteration
coforall t in 0..#numTasks {
  writeln("Hello from task ", t, " of ", numTasks);
} // implicit join of the numTasks tasks here

writeln("All tasks done");
```

## Sample output:

```
Hello from task 2 of 4
Hello from task 0 of 4
Hello from task 3 of 4
Hello from task 1 of 4
All tasks done
```

# Task Parallelism: Data-Driven Synchronization

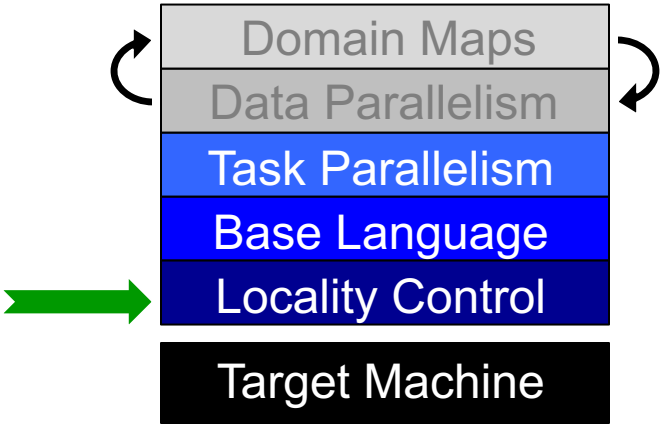
- **atomic variables:** support atomic operations
  - e.g., compare-and-swap; atomic sum, multiply, etc.
  - similar to C/C++

- **sync variables:** store full-empty state along with value
  - by default, reads/writes block until full/empty, leave in opposite state

# Other Task Parallel Concepts

- **cobegins:** create tasks using compound statements
- **single variables:** like sync variables, but write-once
- **sync statements:** join unstructured tasks
- **serial statements:** conditionally squash parallelism

# Locality Control



# The Locale Type

**Definition:**

- Abstract unit of target architecture
- Supports reasoning about locality
  - defines “here vs. there” / “local vs. remote”
- Capable of running tasks and storing variables
  - i.e., has processors and memory

**Typically:** A compute node (multicore processor or SMP)

# Getting started with locales

- Specify # of locales when running Chapel programs

```
% a.out --numLocales=8
```

```
% a.out -nl 8
```

- Chapel provides built-in locale variables

```
config const numLocales: int = …;
const Locales: [0..#numLocales] locale = …;
```


- `main()` starts execution as a task on locale #0

# Locale Operations

- **Locale methods support queries about the target system:**

```
proc locale.physicalMemory(…) { … }
proc locale.numCores { … }
proc locale.id { … }
proc locale.name { … }
```

- ***On-clauses* support placement of computations:**

```
writeln("on locale 0");

on Locales[1] do
  writeln("now on locale 1");

writeln("on locale 0 again");
```

```
on A[i,j] do
  bigComputation(A);

on node.left do
  search(node.left);
```

# 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!");
```

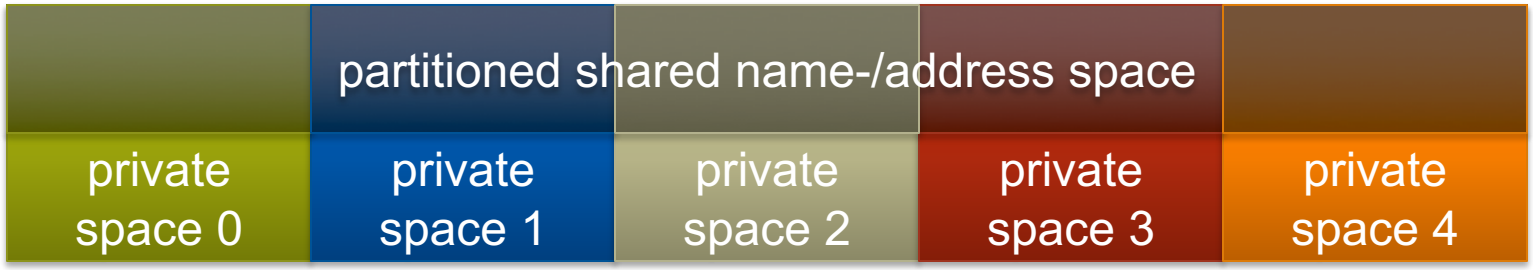
- This is a **distributed parallel** program:

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

# Partitioned Global Address Space (PGAS) Languages

## (Or perhaps: partitioned global namespace languages)

- **abstract concept:**
  - support a shared namespace on distributed memory
    - permit parallel tasks to access remote variables by naming them
  - establish a strong sense of ownership
    - every variable has a well-defined location
    - local variables are cheaper to access than remote ones
- **traditional PGAS languages have been SPMD in nature**
  - best-known examples: Fortran Co-Arrays, UPC



# SPMD PGAS Languages (using a pseudo-language, not Chapel)

```
shared int i(*);    // declare a shared variable i
```

# SPMD PGAS Languages (using a pseudo-language, not Chapel)

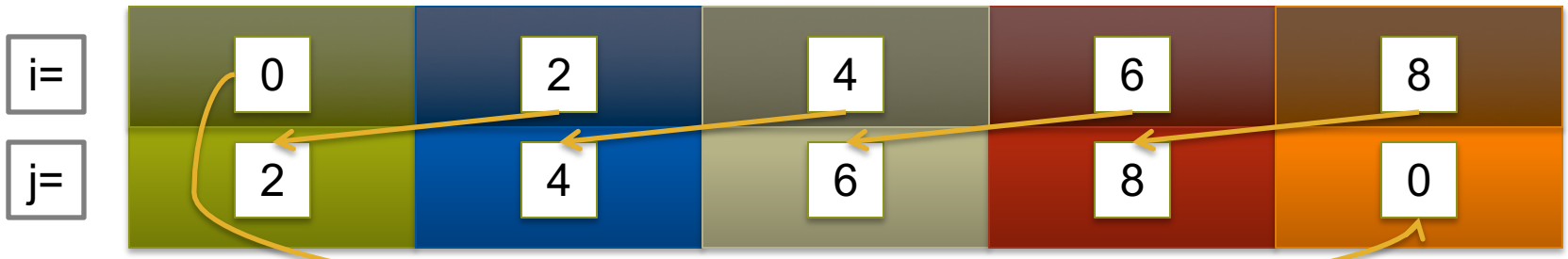
```
shared int i(*);    // declare a shared variable i
function main() {
  i = 2*this_image();  // each image initializes its copy
```

# SPMD PGAS Languages (using a pseudo-language, not Chapel)

```
shared int i(*);    // declare a shared variable i
function main() {
  i = 2*this_image();  // each image initializes its copy

  private int j;       // declare a private variable j
```

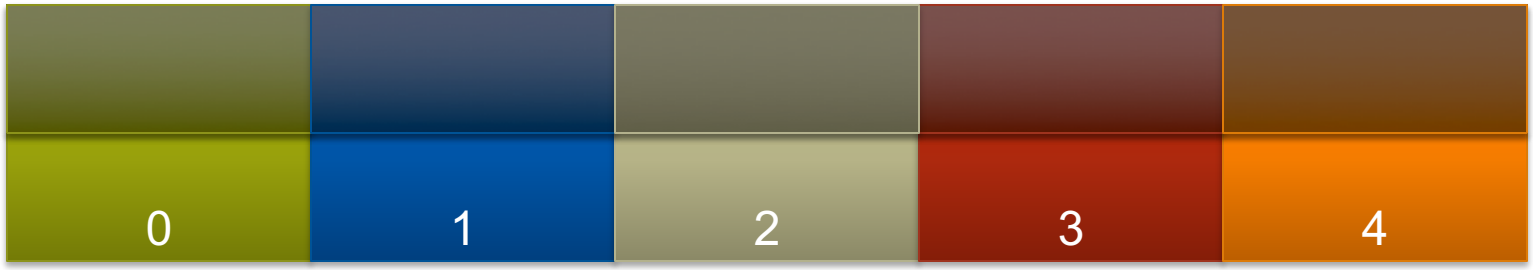
# SPMD PGAS Languages (using a pseudo-language, not Chapel)

```
shared int i(*);    // declare a shared variable i
function main() {
  i = 2*this_image();  // each image initializes its copy
  barrier();
  private int j;       // declare a private variable j
  j = i( (this_image()+1) % num_images() );
    // ^^ access our neighbor's copy of i
    // communication implemented by compiler + runtime
    // How did we know our neighbor had an i?
    // Because it's SPMD - we're all running the same
    // program.  (Simple, but restrictive)
```



# Chapel and PGAS

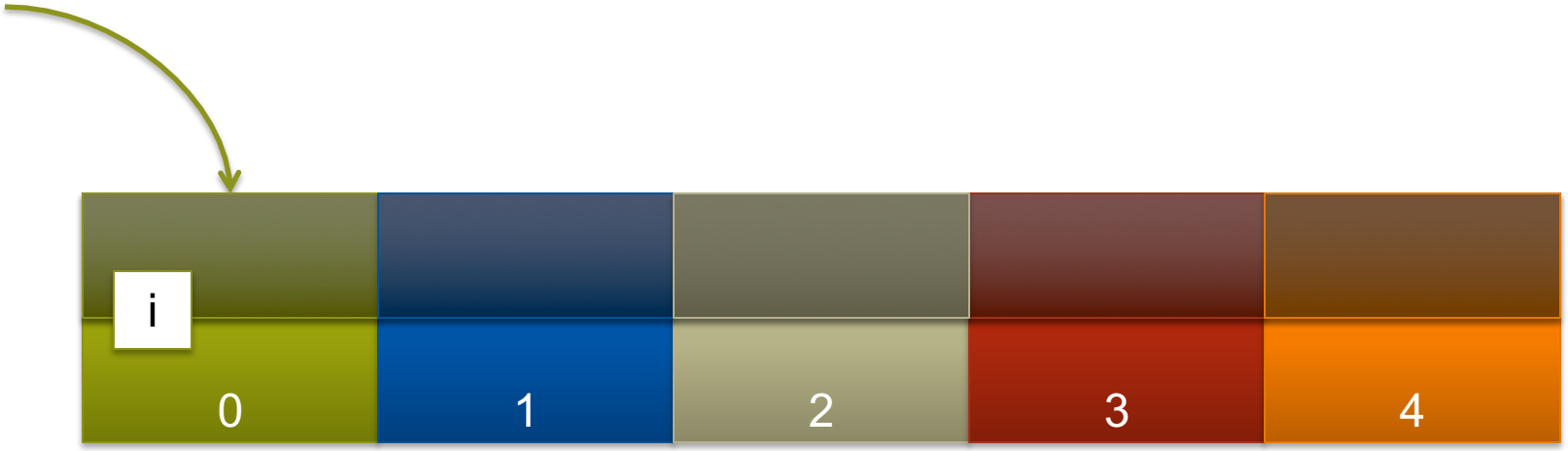
- **Chapel is PGAS, but unlike most, it's not inherently SPMD**
  - never think about "the other copies of the program"
  - "global name/address space" comes from lexical scoping
    - as in traditional languages, each declaration yields one variable
    - variables are stored on the locale where the task declaring it is executing




*Locales* (think: "compute nodes")

# Chapel: Scoping and Locality

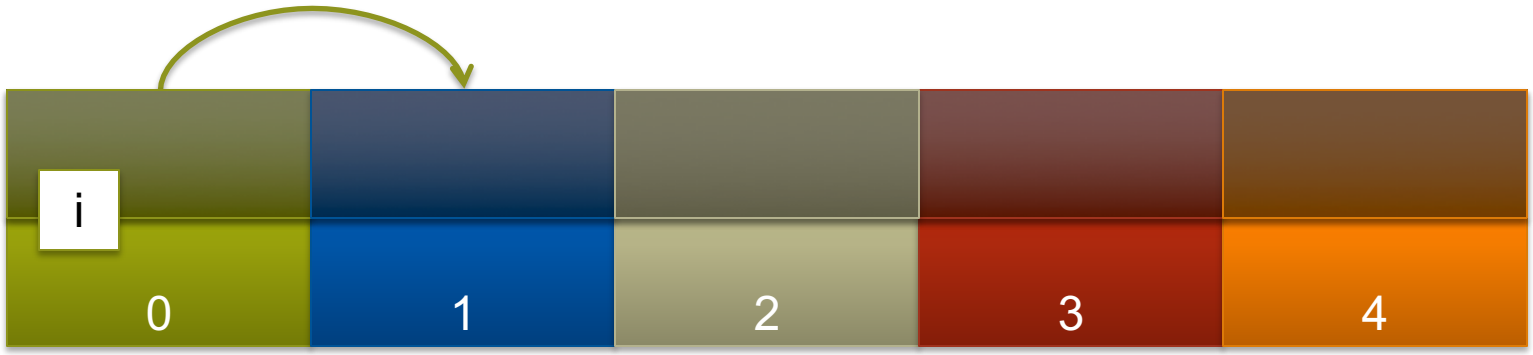
```
var i: int;
```




*Locales* (think: “compute nodes”)

# Chapel: Scoping and Locality

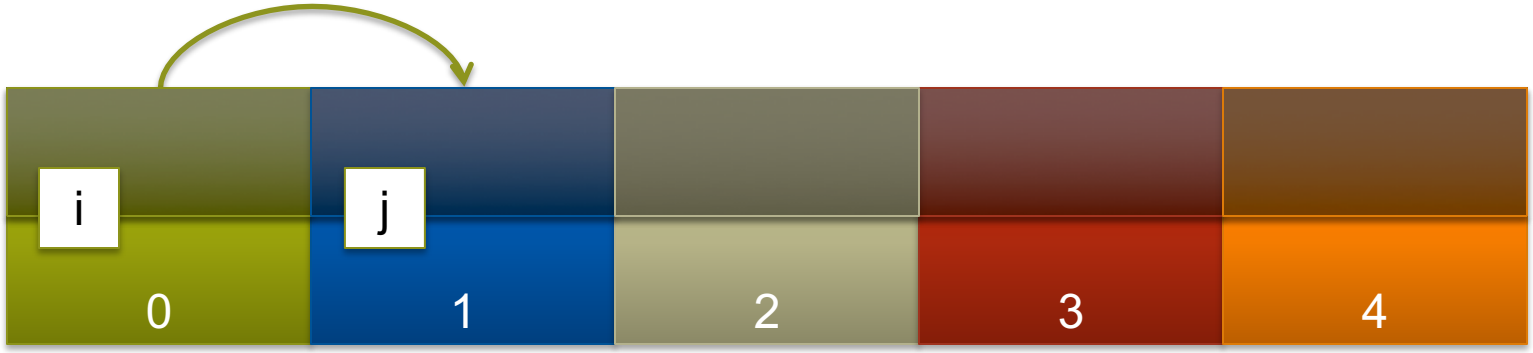
```
var i: int;
on Locales[1] {
```




*Locales* (think: “compute nodes”)

# Chapel: Scoping and Locality

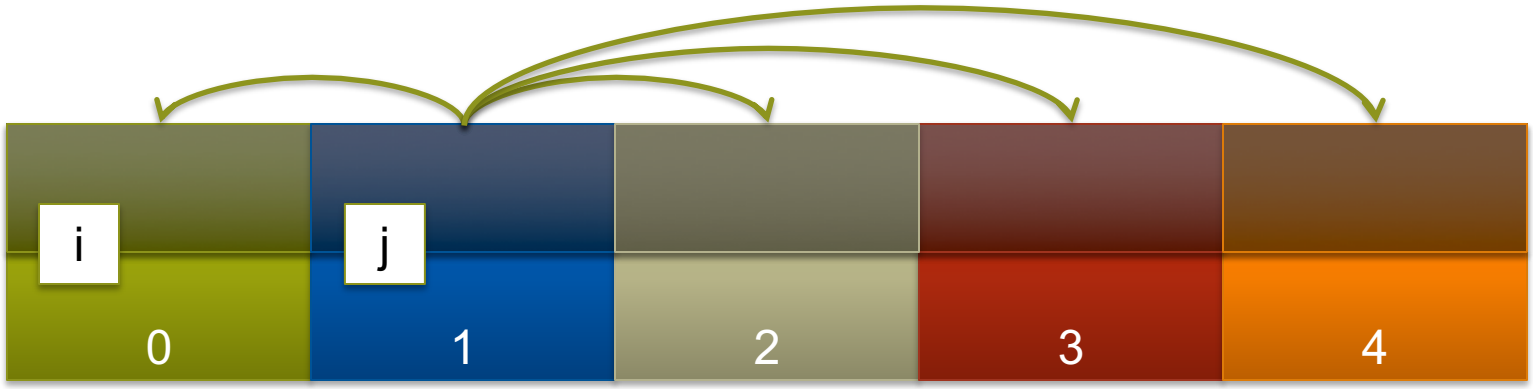
```
var i: int;
on Locales[1] {
  var j: int;
```




*Locales* (think: “compute nodes”)

# Chapel: Scoping and Locality

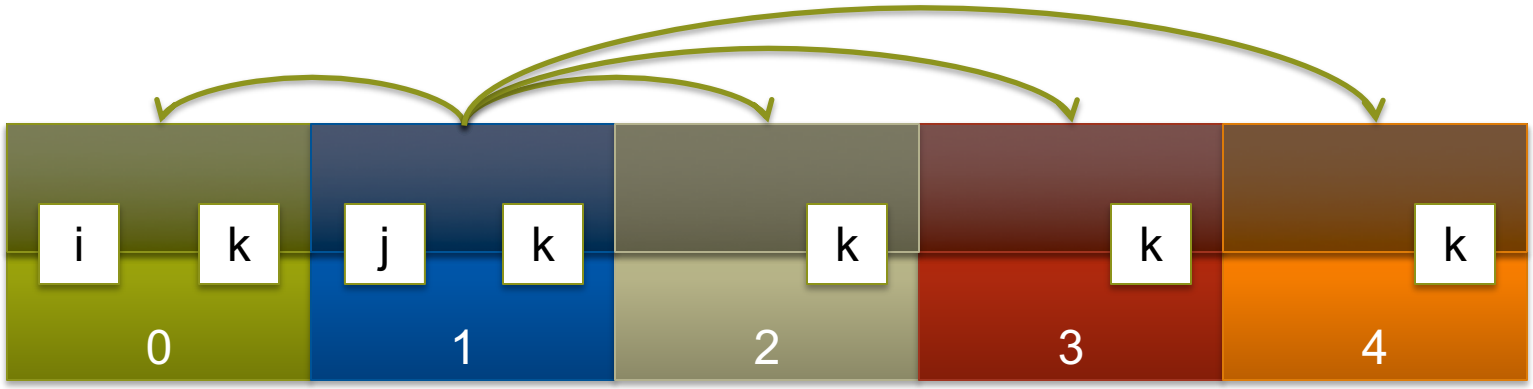
```
var i: int;
on Locales[1] {
  var j: int;
  coforall loc in Locales {
    on loc {
```




*Locales* (think: “compute nodes”)

# Chapel: Scoping and Locality

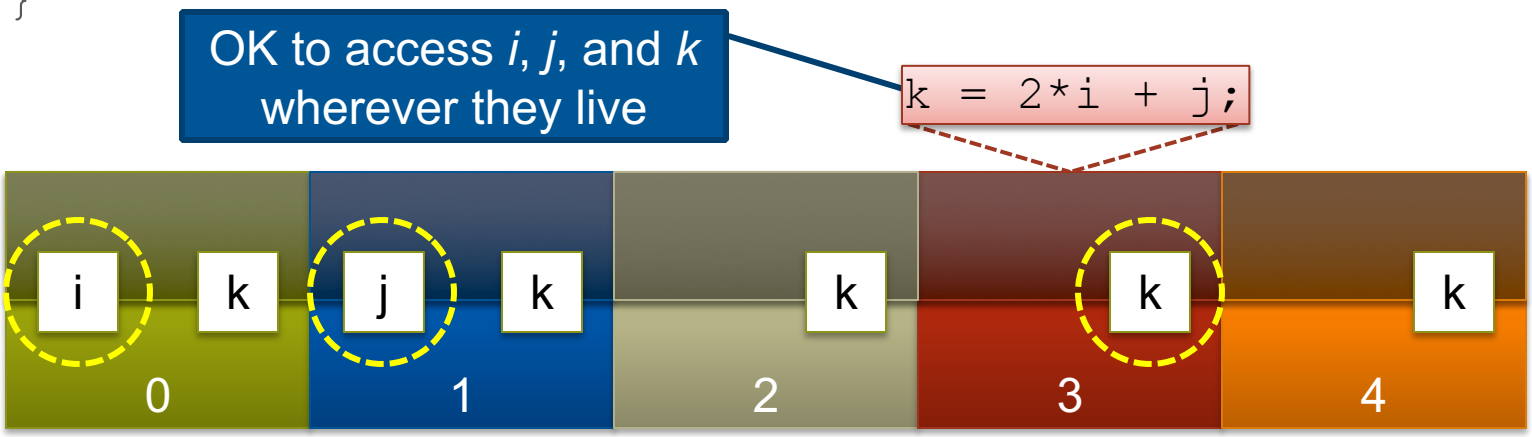
```
var i: int;
on Locales[1] {
  var j: int;
  coforall loc in Locales {
    on loc {
      var k: int;
      …
    }
  }
}
```




*Locales* (think: “compute nodes”)

# Chapel: Scoping and Locality

```
var i: int;
on Locales[1] {
  var j: int;
  coforall loc in Locales {
    on loc {
      var k: int;
      k = 2*i + j;
    }
  }
}
```

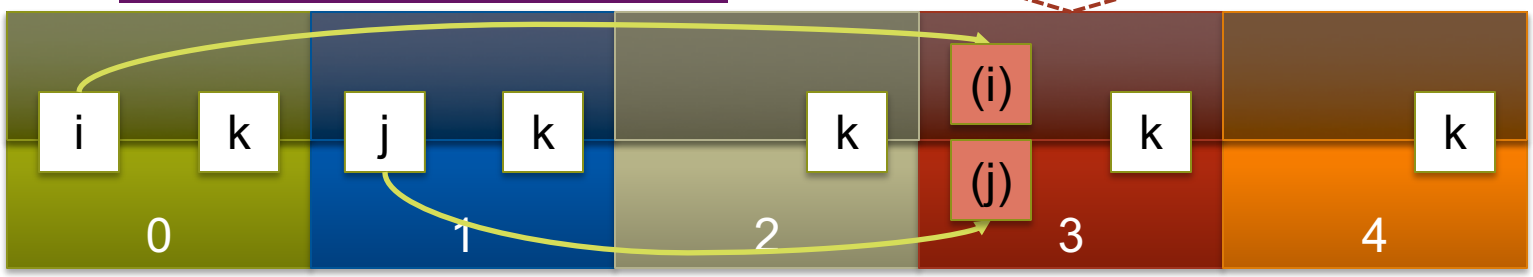



*Locales* (think: "compute nodes")

# Chapel: Scoping and Locality

```
var i: int;
on Locales[1] {
  var j: int;
  coforall loc in Locales {
    on loc {
      var k: int;
      k = 2*i + j;
    }
  }
}
```

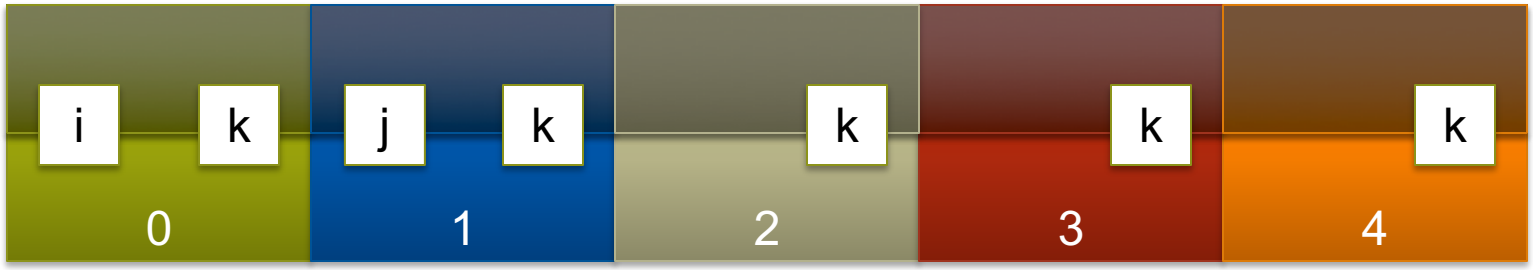
here, *i* and *j* are remote, so the compiler + runtime will transfer their values




*Locales* (think: "compute nodes")

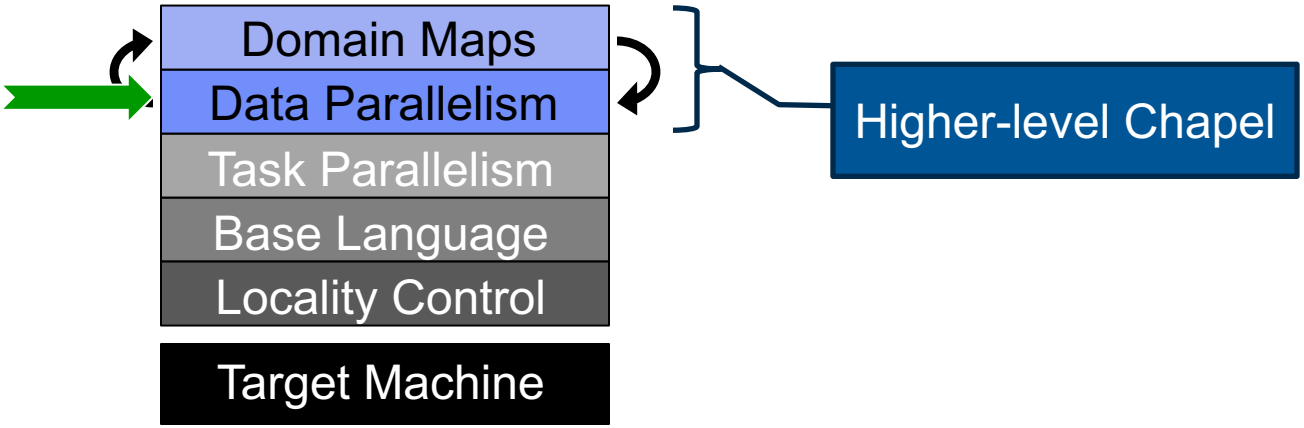
# Chapel: Locality queries

```
var i: int;
on Locales[1] {
  var j: int;
  coforall loc in Locales {
    on loc {
      var k: int;

      …here…          // query the locale on which this task is running
      …j.locale…      // query the locale on which j is stored
    }
  }
}
```



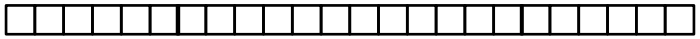

*Locales* (think: “compute nodes”)

# Data Parallelism

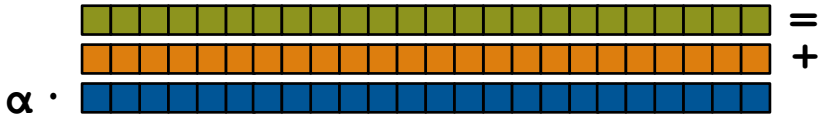


# Data Parallelism By Example: STREAM Triad

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



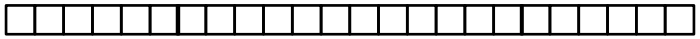
```
var A, B, C: [ProblemSpace] real;
```



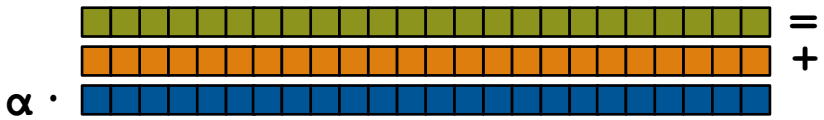

```
forall (a,b,c) in zip(A,B,C) do
  a = b + alpha*c;
```

# Data Parallelism By Example: STREAM Triad

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



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




```
A = B + alpha * C;  // equivalent to the zippered forall
```

# Other Data Parallel Features

- **Rich Domain/Array Types:**
  - multidimensional
  - strided
  - sparse
  - associative
- **Slicing:** Refer to subarrays using ranges/domains

```
… A[2..n-1, lo..#b] …
… A[ElementsOfInterest] …
```

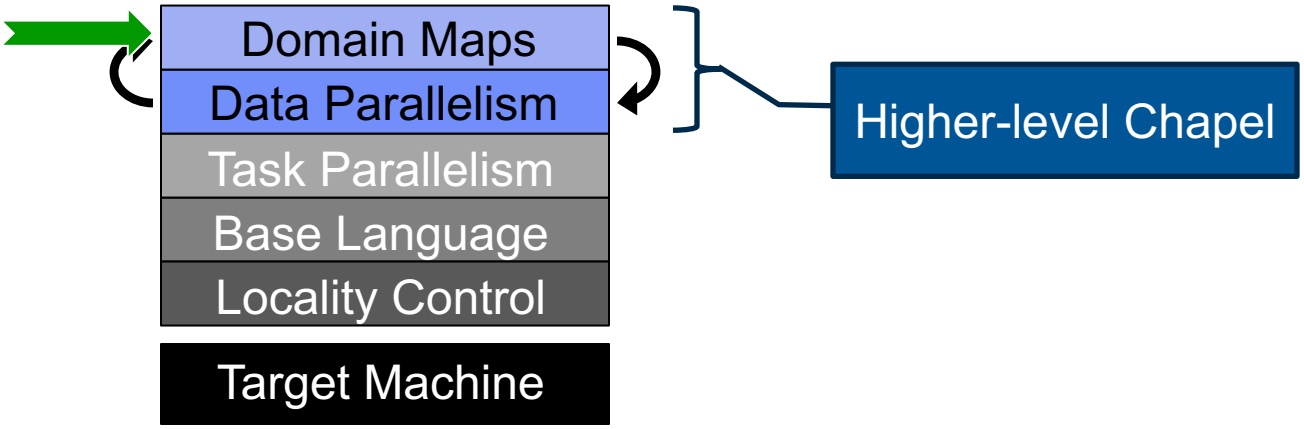
- **Promotion:** Call scalar functions with array arguments

```
… pow(A, B)…   // equivalent to: forall (a,b) in zip(A,B) do pow(a,b)
```

- **Reductions/Scans:** Apply operations across collections

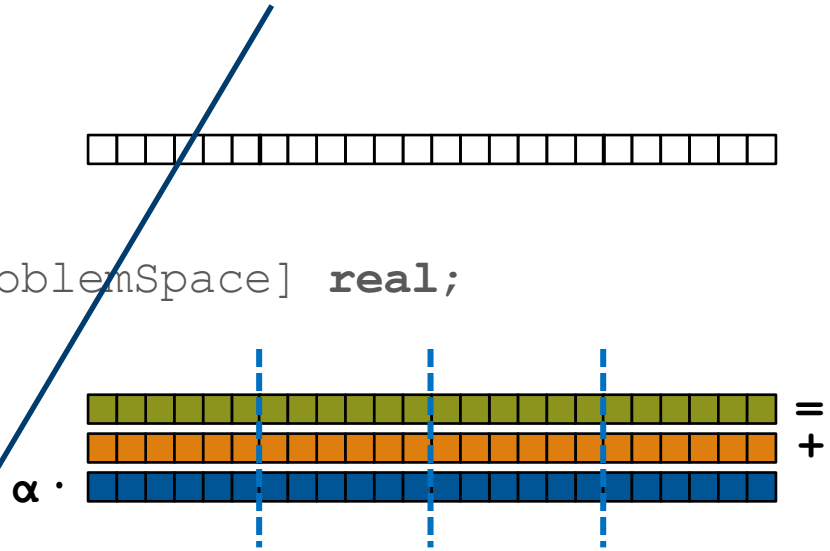
```
… + reduce A …
… myReduceOp reduce A …
```

# Domain Maps



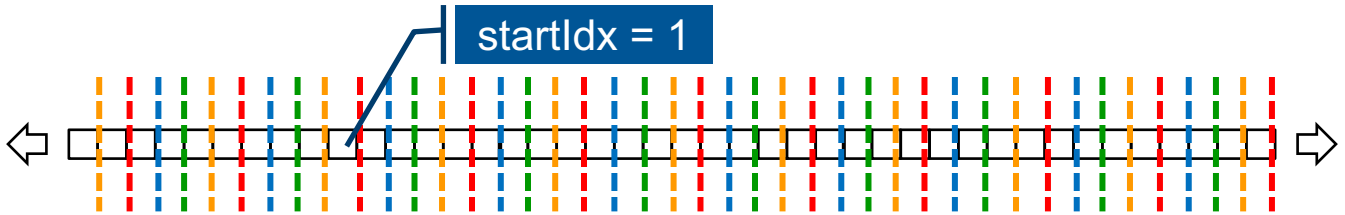
# STREAM Triad: Chapel (multicore)

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

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

A = B + alpha * C;
```

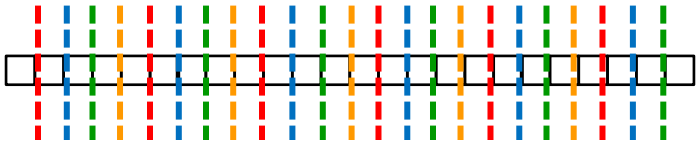



No domain map specified ⇒ use default layout
- current locale owns all domain indices and array values
- computation will execute using local processors only

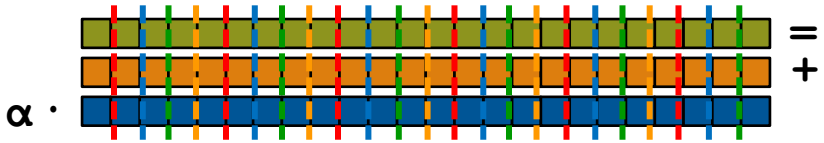
# STREAM Triad: Chapel (multilocale, cyclic)




```
const ProblemSpace = {1..m}
                       dmapped Cyclic(startIdx=1);
```

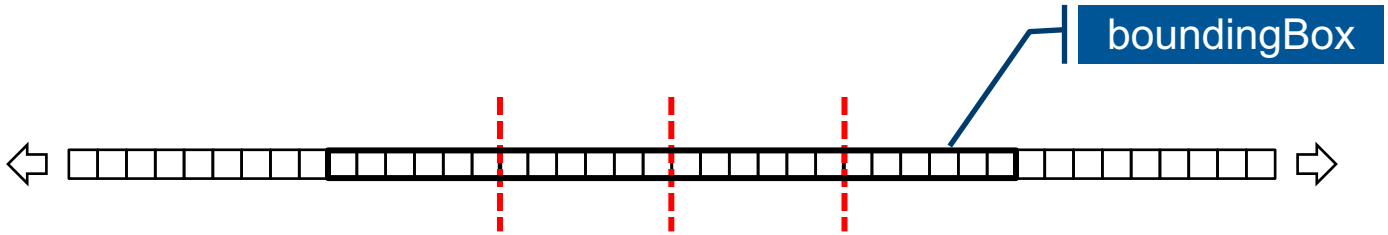


```
var A, B, C: [ProblemSpace] real;
```

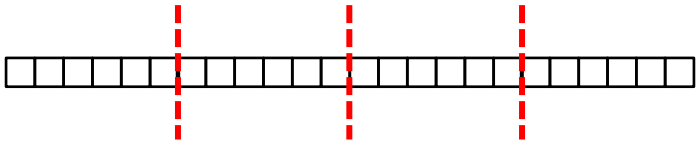



```
A = B + alpha * C;
```

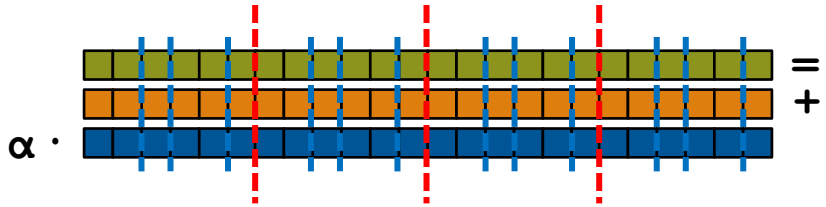
# STREAM Triad: Chapel (multilocale, blocked)




```
const ProblemSpace = {1..m}
                       dmapped Block(boundingBox={1..m});
```



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




```
A = B + alpha * C;
```

# STREAM Triad: Chapel

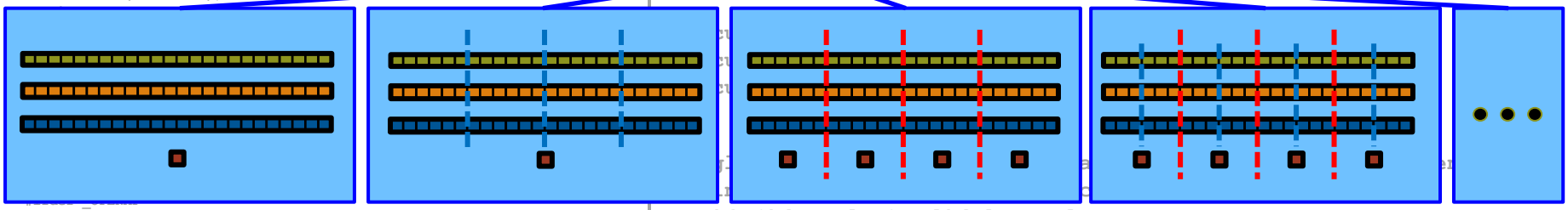
MPI + OpenMP

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

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

int HPCC_StarStream(HPCC_Params *pa
  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 == myR
  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, VectorSi
  b = HPCC_XMALLOC( double, VectorSi
  c = HPCC_XMALLOC( double, VectorSi

  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"
      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 = 3.0;

A = B + alpha * C;
```

the special sauce



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.

# Language Summary

***Parallel programmers deserve better programming models***

***Higher-level programming models can help insulate algorithms from parallel implementation details***

- yet, without necessarily abdicating control
- Chapel does this via its multiresolution design

***We believe Chapel can greatly improve productivity***

…for current and emerging parallel architectures

…for HPC users as well as mainstream uses of parallelism

# Outline

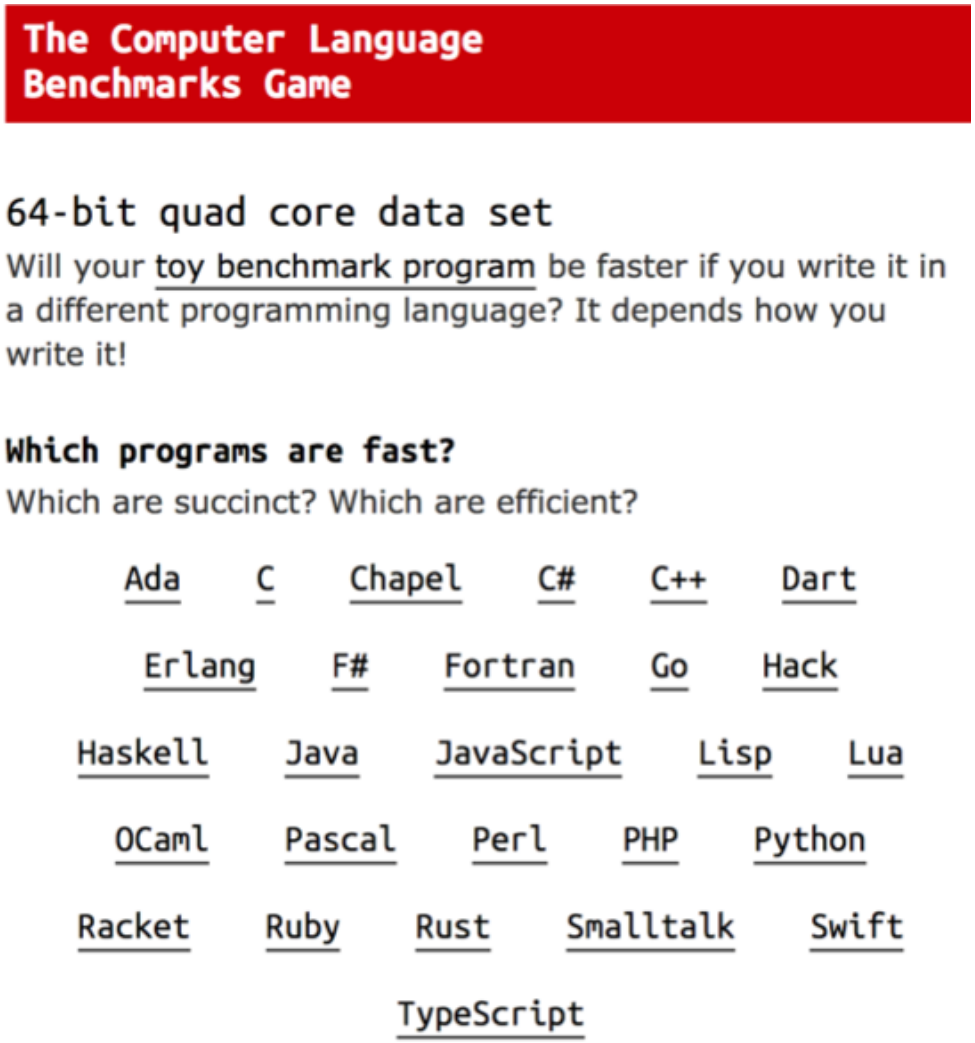
- ✓ Motivation for Chapel
- ✓ Survey of Chapel Concepts
- ➢ Chapel Project and Characterizations
- ● Chapel Resources

# Computer Language Benchmarks Game

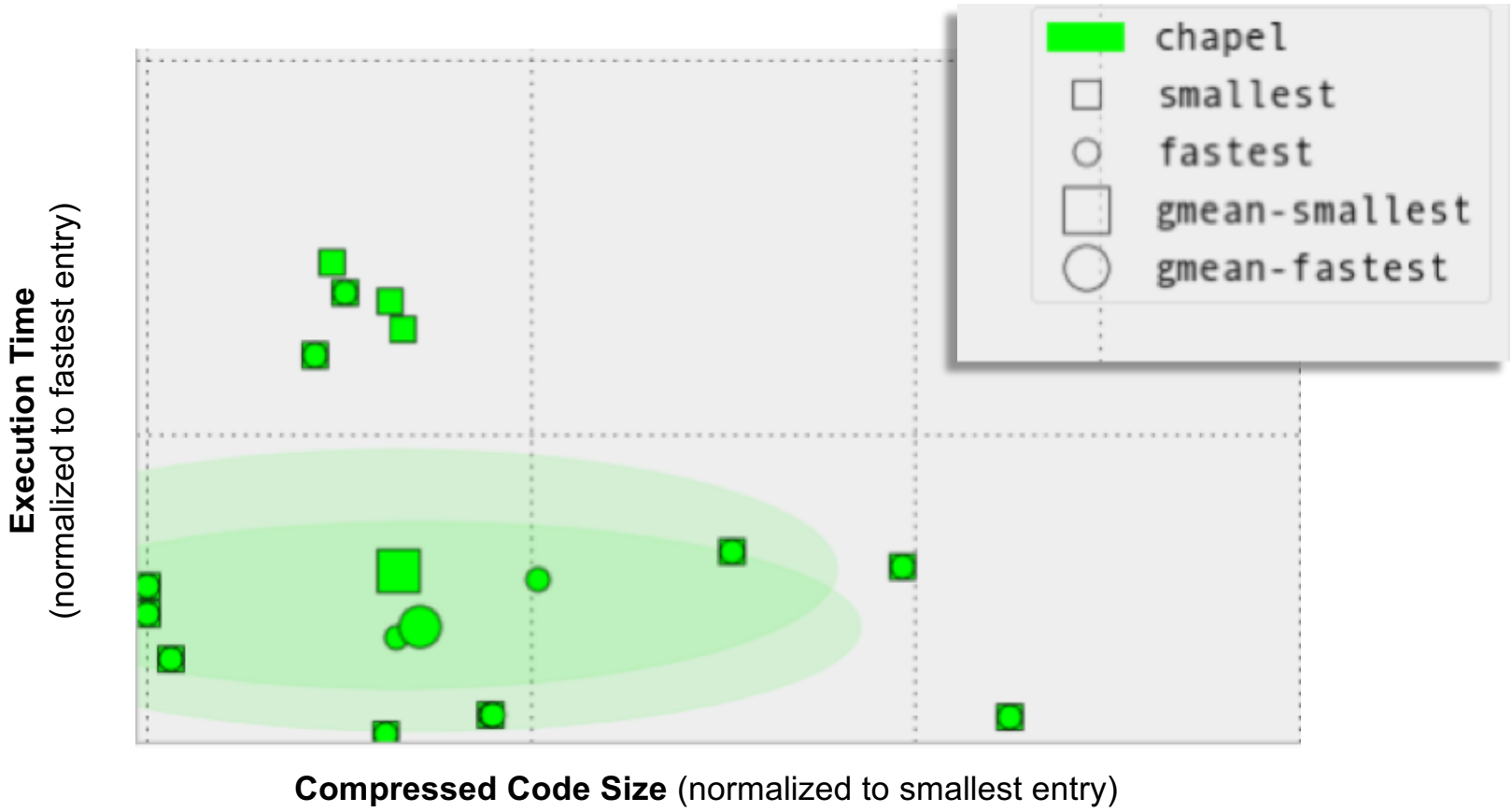
## Chapel was recently added to the game:

## As of December 2017:

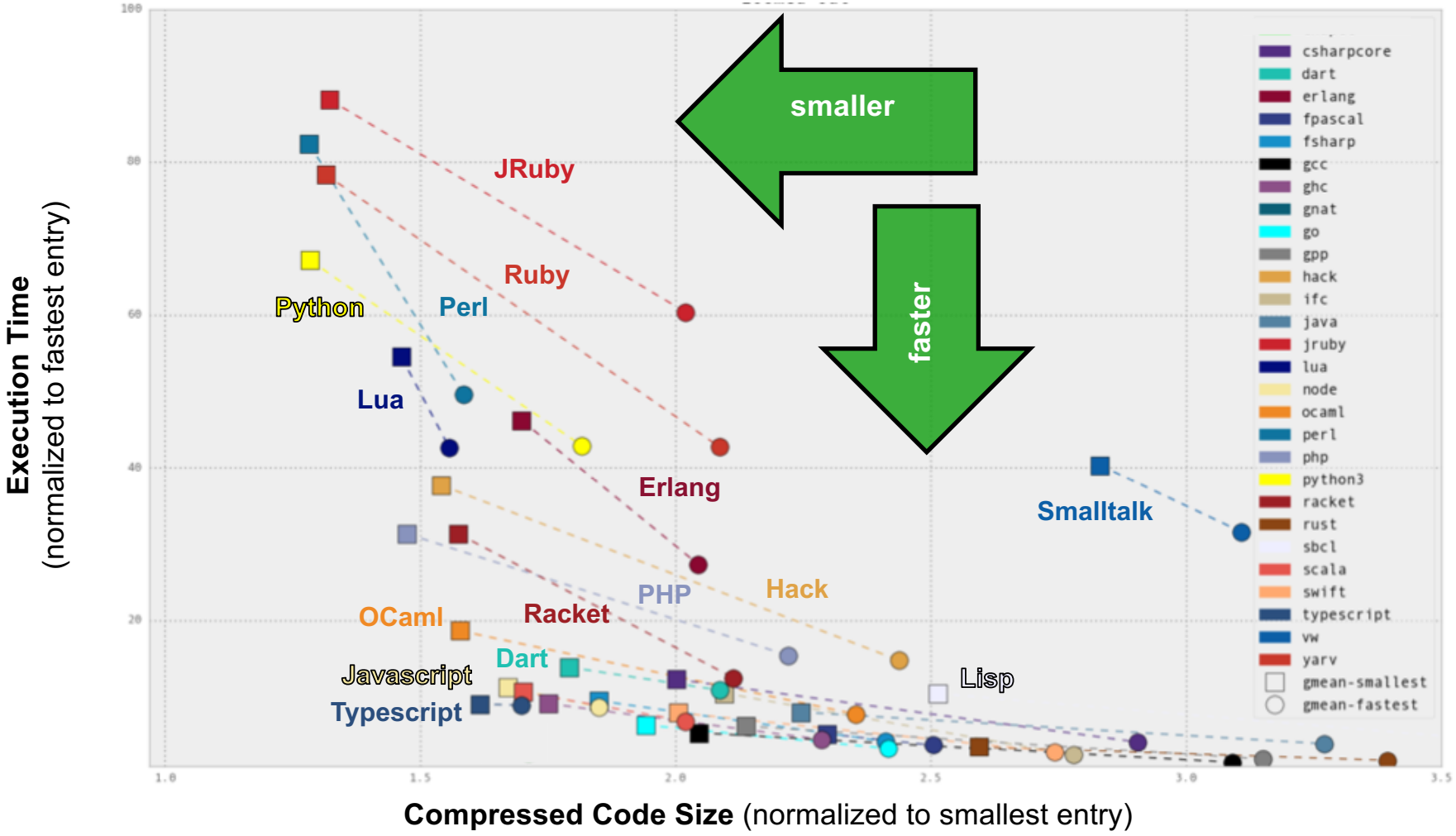
- **for performance:**
  - **1 top entries:** pidigits
  - **4 top-5 entries:** fannkuch-redux, chameneos-redux, meteor, thread-ring
  - **2 top-10 entries:**, mandelbrot, fasta
  - **3 top-20 entries:** spectral-norm, regex-redux
- **for code compactness:**
  - **2 top entries:** n-body, thread-ring
  - **1 top-5 entries:** spectral-norm
  - **2 top-10 entry:** pidigits, regex-redux
  - **3 top-20 entries:** mandelbrot, chameneos-redux, meteor



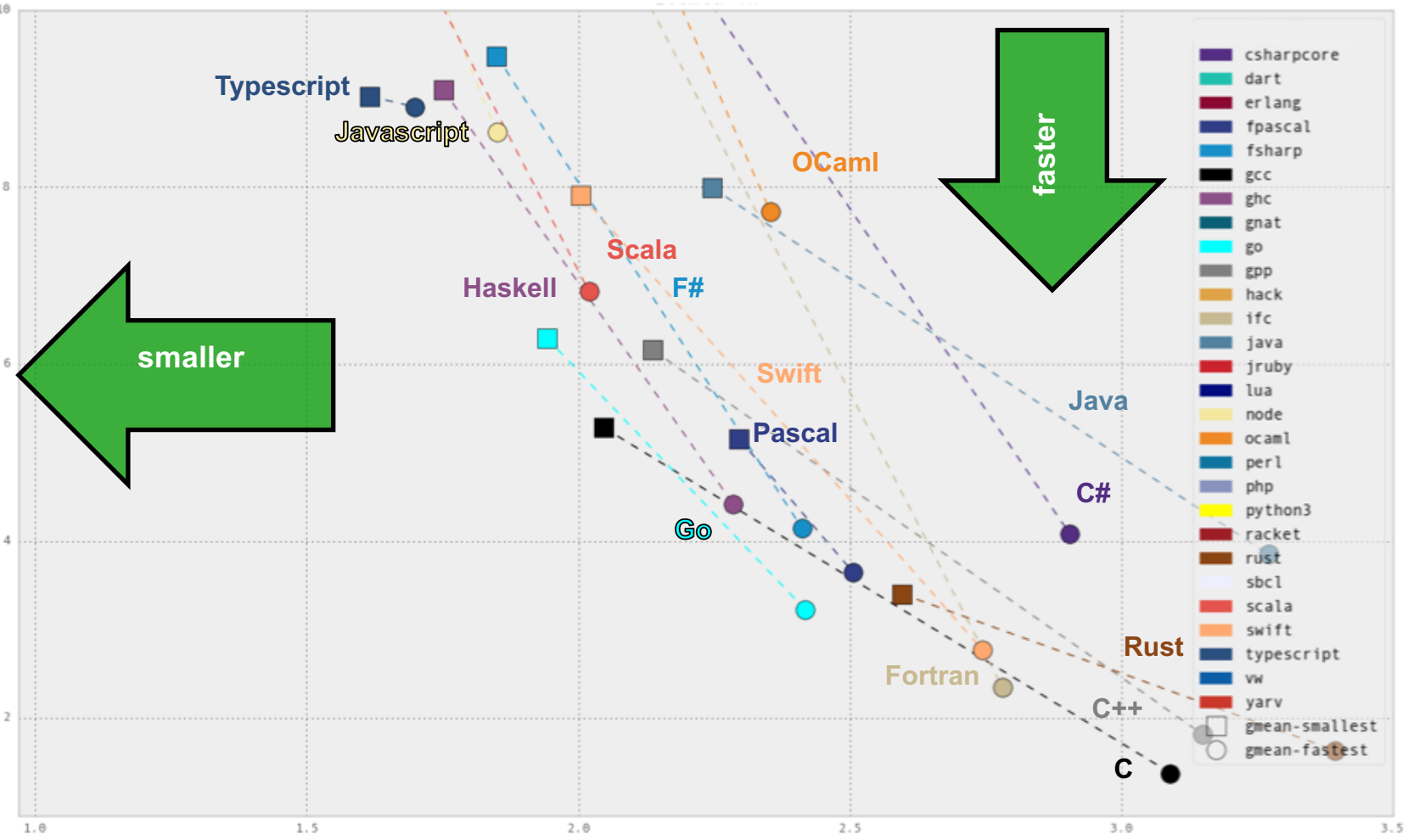
# Scatter plots of CLBG code size x speed



# CLBG Cross-Language Summary
(Oct 2017 standings)



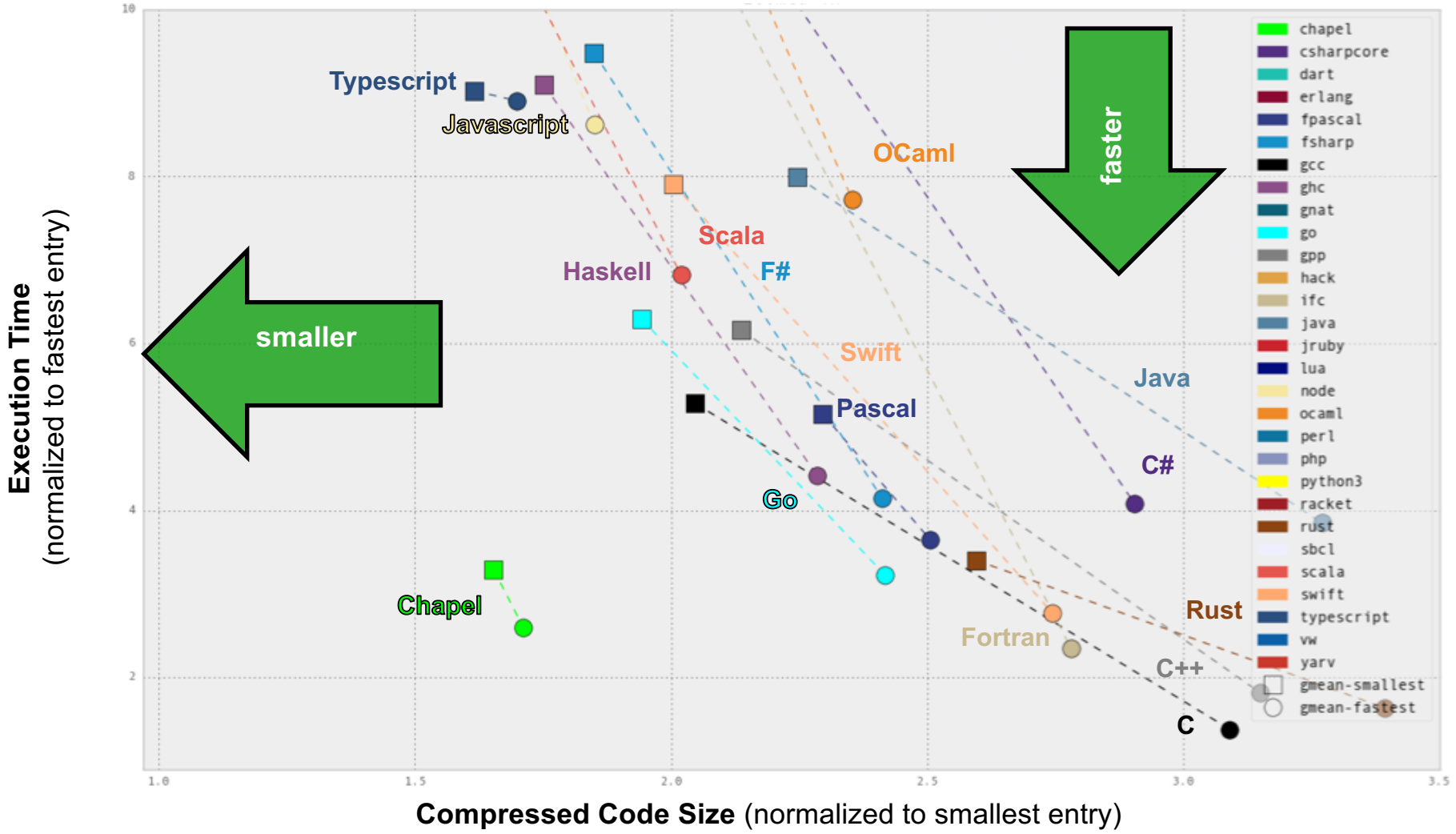
# CLBG Cross-Language Summary
(Oct 2017 standings)



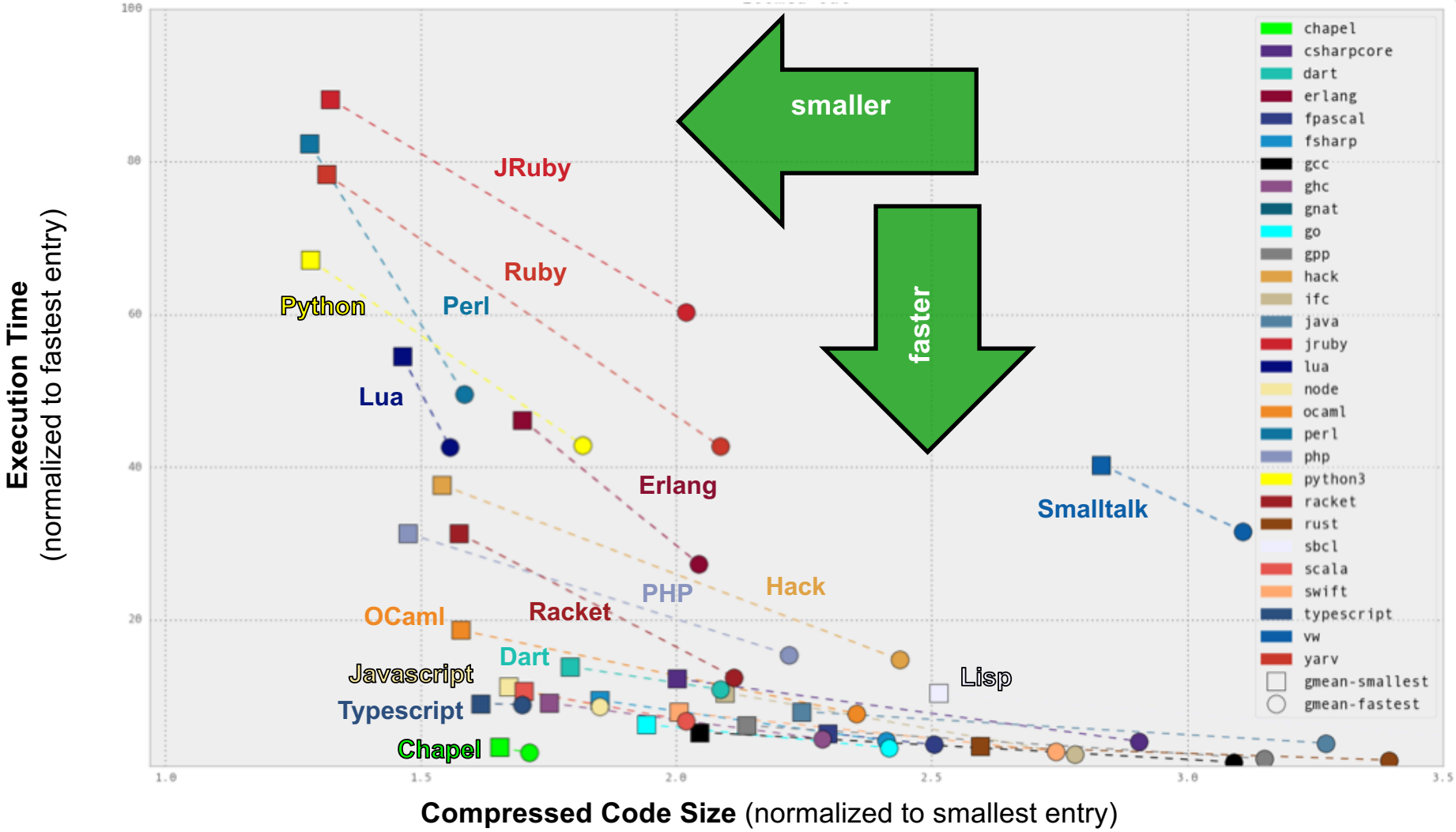

**Execution Time** (normalized to fastest entry)

**Compressed Code Size** (normalized to smallest entry)

# CLBG Cross-Language Summary
(Oct 2017 standings)



# CLBG Cross-Language Summary
(Oct 2017 standings)



# Chapel is Portable

- **Chapel is designed to be hardware-independent**

- **The current release requires:**
  - a C/C++ compiler
  - a *NIX environment (Linux, OS X, BSD, Cygwin, …)
  - POSIX threads
  - RDMA, MPI, or UDP (for distributed memory execution)

- **Chapel can run on…**

  …laptops and workstations
  …commodity clusters
  …the cloud
  …HPC systems from Cray and other vendors
  …modern processors like Intel Xeon Phi, GPUs*, etc.

  * = academic work only; not yet supported in the official release

# Chapel is Open-Source

- **Chapel's development is hosted at GitHub**
  - https://github.com/chapel-lang
- **Chapel is licensed as Apache v2.0 software**
- **Instructions for download + install are online**
  - see http://chapel.cray.com/download.html to get started

# The Chapel Team at Cray (Summer 2016)

# Chapel is a Collaborative Effort









Lawrence Berkeley
National Laboratory





(and several others…)

https://chapel-lang.org/collaborations.html

# Chapel Status

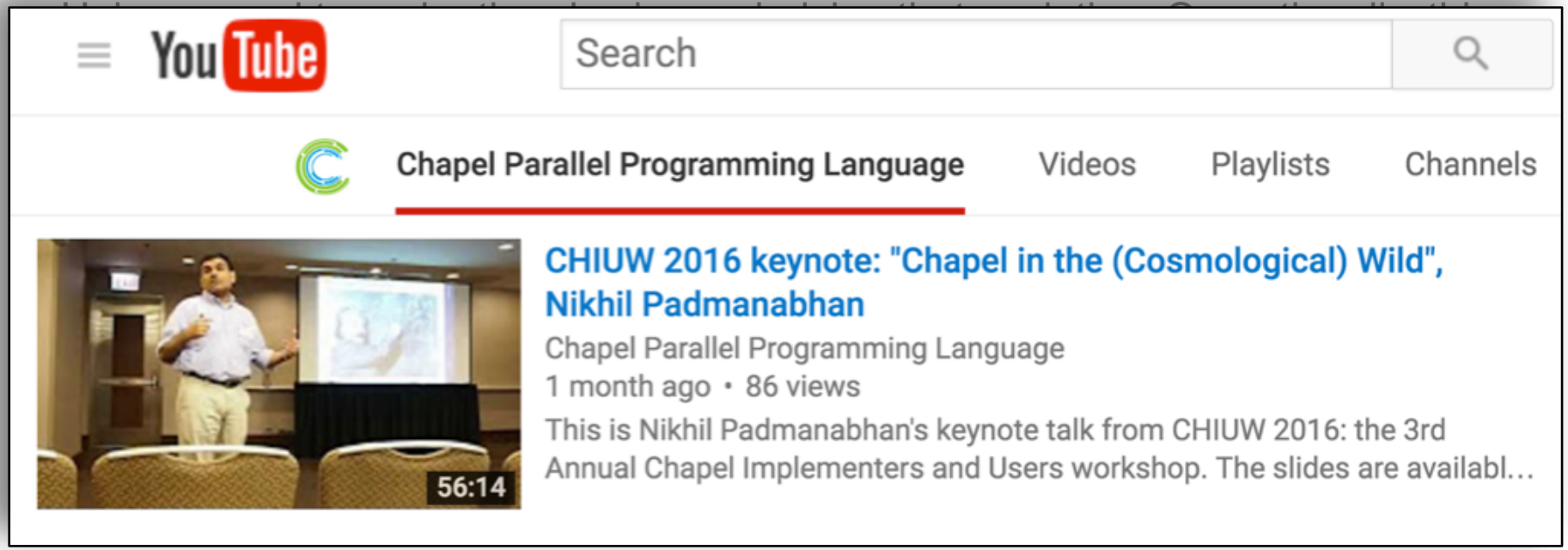
- **Currently being picked up by early adopters**
  - over two releases, 3000+ downloads per year
  - Users who try it generally like what they see
- **Most current features are functional and working well**
- **Performance is compelling in many cases**

# A notable early adopter

## Chapel in the (Cosmological) Wild

**Nikhil Padmanabhan**, *Yale University Professor, Physics & Astronomy*

**Abstract:** This talk aims to present my personal experiences using Chapel in my research. My research interests are in observational cosmology; more specifically, I use large surveys of galaxies to constrain the evolution of the



# Outline

- ✓ Motivation for Chapel
- ✓ Survey of Chapel Concepts
- ✓ Chapel Project and Characterizations
- ➢ **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

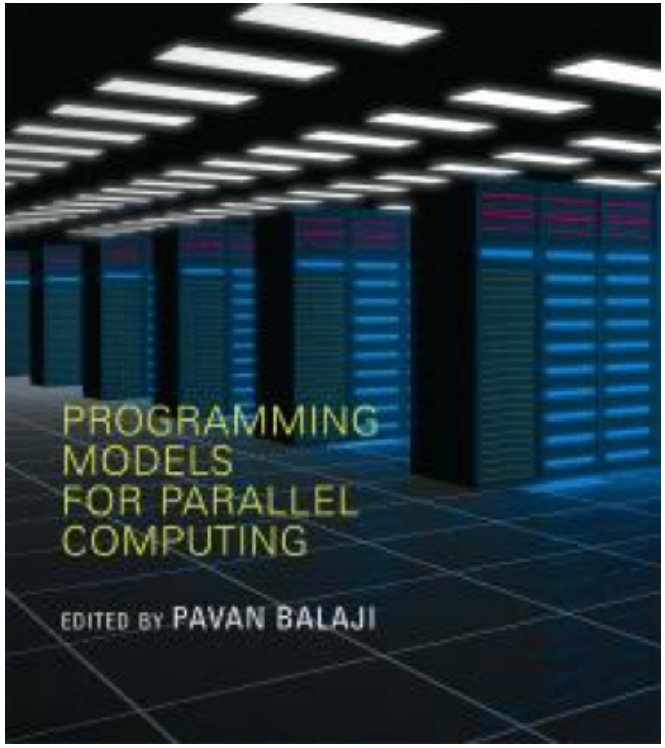
**Facebook:** https://www.facebook.com/ChapelLanguage

**Twitter:** https://twitter.com/ChapelLanguage


# Suggested Reading

Chapel chapter from ***Programming Models for Parallel Computing***

- a detailed overview of Chapel's history, motivating themes, features
- published by MIT Press, November 2015
- edited by Pavan Balaji (Argonne)
- 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.

- *coverage of recent 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*

# Chapel 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
chapel-bugs@lists.sourceforge.net: public bug forum

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

## To mail the Cray team:

chapel_info@cray.com: contact the team at Cray
chapel_bugs@cray.com: for reporting non-public bugs

## or use IRC (#chapel on chat.freenode.net) or StackOverflow

# Chapel:
## Productive Parallel Programming at Scale

## Questions?