Chapel Background and Motivation
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.
Chapel’s Origins: HPCS

DARPA HPCS: High Productivity Computing Systems

- **Goal**: improve productivity by a factor of 10x
- **Timeframe**: summer 2002 – fall 2012
- Cray developed a new system architecture, network, software, ...
  - this became the very successful Cray XC30™ Supercomputer Series

...and a new programming language: Chapel
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:**

\[ A = B + \alpha \cdot C \]
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):**

\[
\begin{aligned}
A &= \ldots \\
B &= \ldots \\
C &= \ldots \\
\alpha &= \ldots \\
\end{aligned}
\]
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):

![Diagram showing the computation of $A_i = B_i + \alpha \cdot C_i$ in parallel with vectors $A$, $B$, and $C$. The diagram illustrates the iterative process of adding each element of $B$ and scaling it by $\alpha$ to each corresponding element of $C$ to produce $A$.](image)
STREAM Triad: MPI

```c
#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+OpenMP

```c
#include <hpcc.h>

#define _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 vs. CUDA

```
#include <hpcc.h>
#include <omp.h>

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 );

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);
}
```

HPC suffers from too many distinct notations for expressing parallelism and locality
Why so many programming models?

HPC has traditionally given users…
...low-level, *control-centric* programming models
...ones that are closely tied to the underlying hardware
...ones that support only a single type of parallelism

<table>
<thead>
<tr>
<th>Type of HW Parallelism</th>
<th>Programming Model</th>
<th>Unit of Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inter-node</td>
<td>MPI</td>
<td>executable</td>
</tr>
<tr>
<td>Intra-node/multicore</td>
<td>OpenMP/pthreads</td>
<td>iteration/task</td>
</tr>
<tr>
<td>Instruction-level vectors/threads</td>
<td>pragmas</td>
<td>iteration</td>
</tr>
<tr>
<td>GPU/accelerator</td>
<td>CUDA/OpenCL/OpenACC</td>
<td>SIMD function/task</td>
</tr>
</tbody>
</table>

**benefits:** lots of control; decent generality; easy to implement
**downsides:** lots of user-managed detail; brittle to changes
Rewinding a few slides...

HPC suffers from too many distinct notations for expressing parallelism and locality.

---

```c
#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_StarStream( params, 0 == myRank);
    MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM, 0, comm );
    return errCount;
}

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

```c
#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.x+=1;
    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: Chapel

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;
```

Philosophy: Good language design can tease details of locality and parallelism away from an algorithm, permitting the compiler, runtime, applied scientist, and HPC expert to each focus on their strengths.
Motivating Chapel Themes

1) General Parallel Programming
2) Global-View Abstractions
3) Multiresolution Design
4) Control over Locality/Affinity
5) Reduce HPC ↔ Mainstream Language Gap
1) General Parallel Programming

With a unified set of concepts...

...express any parallelism desired in a user’s program
- **Styles**: data-parallel, task-parallel, concurrency, nested, ...
- **Levels**: model, function, loop, statement, expression

...target any parallelism available in the hardware
- **Types**: machines, nodes, cores, instructions

<table>
<thead>
<tr>
<th>Type of HW Parallelism</th>
<th>Programming Model</th>
<th>Unit of Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inter-node</td>
<td>Chapel</td>
<td>executable/task</td>
</tr>
<tr>
<td>Intra-node/multicore</td>
<td>Chapel</td>
<td>iteration/task</td>
</tr>
<tr>
<td>Instruction-level vectors/threads</td>
<td>Chapel</td>
<td>iteration</td>
</tr>
<tr>
<td>GPU/accelerator</td>
<td>Chapel</td>
<td>SIMD function/task</td>
</tr>
</tbody>
</table>
2) Global-View Abstractions

In pictures: “Apply a 3-Point Stencil to a vector”

\[
\text{Global-View} \quad \left( \frac{\text{Red} + \text{Pink}}{2} \right) \times 2
\]

\[
\text{Local-View}
\]

Copyright 2014 Cray Inc.
2) Global-View Abstractions

In pictures: “Apply a 3-Point Stencil to a vector”

Global-View

\[
\begin{align*}
&\left[ \begin{array}{cccc}
\text{R} & \text{R} & \text{R} & \text{R} \\
\text{R} & \text{R} & \text{R} & \text{R} \\
\text{R} & \text{R} & \text{R} & \text{R} \\
\text{R} & \text{R} & \text{R} & \text{R} \\
\end{array} \right]/2 \\
+ & \left[ \begin{array}{cccc}
\text{R} & \text{R} & \text{R} & \text{R} \\
\text{R} & \text{R} & \text{R} & \text{R} \\
\text{R} & \text{R} & \text{R} & \text{R} \\
\text{R} & \text{R} & \text{R} & \text{R} \\
\end{array} \right]/2 \\
= & \left[ \begin{array}{cccc}
\text{R} & \text{R} & \text{R} & \text{R} \\
\text{R} & \text{R} & \text{R} & \text{R} \\
\text{R} & \text{R} & \text{R} & \text{R} \\
\text{R} & \text{R} & \text{R} & \text{R} \\
\end{array} \right]
\end{align*}
\]

Local-View

\[
\begin{align*}
&\left[ \begin{array}{cccc}
\text{R} & \text{R} & \text{R} & \text{R} \\
\text{R} & \text{R} & \text{R} & \text{R} \\
\text{R} & \text{R} & \text{R} & \text{R} \\
\text{R} & \text{R} & \text{R} & \text{R} \\
\end{array} \right]/2 \\
+ & \left[ \begin{array}{cccc}
\text{R} & \text{R} & \text{R} & \text{R} \\
\text{R} & \text{R} & \text{R} & \text{R} \\
\text{R} & \text{R} & \text{R} & \text{R} \\
\text{R} & \text{R} & \text{R} & \text{R} \\
\end{array} \right]/2 \\
= & \left[ \begin{array}{cccc}
\text{R} & \text{R} & \text{R} & \text{R} \\
\text{R} & \text{R} & \text{R} & \text{R} \\
\text{R} & \text{R} & \text{R} & \text{R} \\
\text{R} & \text{R} & \text{R} & \text{R} \\
\end{array} \right]
\end{align*}
\]
2) Global-View Abstractions

In code: “Apply a 3-Point Stencil to a vector”

**Global-View**

```chapel
proc main() {
    var n = 1000;
    var A, B: [1..n] real;

    forall i in 2..n-1 do
        B[i] = (A[i-1] + A[i+1])/2;
}
```

**Local-View (SPMD)**

```chapel
proc main() {
    var n = 1000;
    var A, B: [0..myN+1] real;

    if (me < p-1) {
        send(me+1, A[myN]);
        recv(me+1, A[myN+1]);
    }
    if (me > 0) {
        send(me-1, A[0]);
        recv(me-1, A[1]);
    }

    forall i in 1..myN do
        B[i] = (A[i-1] + A[i+1])/2;
}
```

Bug: Refers to uninitialized values at ends of A
2) Global-View Abstractions

In code: “Apply a 3-Point Stencil to a vector”

Global-View

```
proc main() {
  var n = 1000;
  var A, B: [1..n] real;
  forall i in 2..n-1 do
    B[i] = (A[i-1] + A[i+1])/2;
}
```

Local-View (SPMD)

```
proc main() {
  var n = 1000;
  var p = numProcs(),
       me = myProc(),
       myN = n/p,
       myLo = 1,
       myHi = myN;
  var A, B: [0..myN+1] real;

  if (me < p-1) {
    send(me+1, A[myN]);
    recv(me+1, A[myN+1]);
  } else
    myHi = myN-1;

  if (me > 0) {
    send(me-1, A[1]);
    recv(me-1, A[0]);
  } else
    myLo = 2;

  forall i in myLo..myHi do
    B[i] = (A[i-1] + A[i+1])/2;
}
```

Assumes p divides n

Communication becomes geometrically more complex for higher-dimensional arrays
2) Global-View Programming: A Final Note

A language may support both global- and local-view programming — in particular, Chapel does

```chapel
proc main() {
    coforall loc in Locales do
        on loc do
            MySPMDProgram(loc.id, Locales.numElements);

    proc MySPMDProgram(myImageID, numImages) {
        ...
    }
}
```
3) Multiresolution Design: Motivation

“Why is everything so tedious/difficult?”
“Why don’t my programs port trivially?”

“Why don’t I have more control?”
3) Multiresolution Design: Concept

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

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

*Chapel language concepts*

- Domain Maps
- Data Parallelism
- Task Parallelism
- Base Language
- Locality Control
- Target Machine

- build the higher-level concepts in terms of the lower
- permit the user to intermix layers arbitrarily
4) Control over Locality/Affinity

Consider:
- Scalable architectures package memory near processors
- Remote accesses take longer than local accesses

Therefore:
- Placement of data relative to tasks affects scalability
- Give programmers control of data and task placement

Note:
- Increasingly, locality matters more and more within a compute node as well
Partitioned Global Address Space Languages

(Or perhaps: partitioned global namespace languages)

abstract concept:

- support a shared namespace on distributed memory
  - permit any parallel task to access any lexically visible variable
  - doesn’t matter if it’s local or remote
Partitioned Global Address Space Languages

(Or perhaps: partitioned global namespace languages)

abstract concept:

- support a shared namespace on distributed memory
  - permit any parallel task to access any lexically visible variable
  - doesn’t matter if it’s local or remote
- establish a strong sense of ownership
  - every variable has a well-defined location
  - local variables are cheaper to access than remote ones
Chapel and PGAS

- Chapel is a PGAS language...
  - but unlike most, it’s not restricted to SPMD
  - ⇒ never think in terms of “the other copies of the program”

Locales (think: “compute nodes”)
5) Reduce HPC ↔ Mainstream Language Gap

Consider:

- Students graduate with training in Java, Matlab, Python, etc.
- Yet HPC programming is dominated by Fortran, C/C++, MPI

We’d like to narrow this gulf with Chapel:

- to leverage advances in modern language design
- to better utilize the skills of the entry-level workforce...
- ...while not alienating the traditional HPC programmer
  - e.g., support object-oriented programming, but make it optional
Questions about Context and Motivation?
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, URIKA, and YARCDATA. 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.

Copyright 2014 Cray Inc.