Chapel: Background
Sustained Performance Milestones

1 GF – 1988: Cray Y-MP; 8 Processors
- Static finite element analysis

1 TF – 1998: Cray T3E; 1,024 Processors
- Modeling of metallic magnet atoms

1 PF – 2008: Cray XT5; 150,000 Processors
- Superconductive materials

1 EF – ~2018: Cray ____; ~10,000,000 Processors
- TBD
Sustained Performance Milestones

1 GF – 1988: Cray Y-MP; 8 Processors
- Static finite element analysis
- Fortran77 + Cray autotasking + vectorization

1 TF – 1998: Cray T3E; 1,024 Processors
- Modeling of metallic magnet atoms
- Fortran + MPI (?)

1 PF – 2008: Cray XT5; 150,000 Processors
- Superconducting materials
- C++/Fortran + MPI + vectorization

1 EF – ~2018: Cray ____; ~10,000,000 Processors
- TBD
- TBD: C/C++/Fortran + MPI + CUDA/OpenCL/OpenMP/??? or ???
HPC has traditionally given users...

...low-level, *control-centric* programming models
...ones that are closely tied to the underlying hardware

**benefits:** lots of control; decent generality; easy to implement
**downsides:** lots of user-managed detail; brittle to changes
Given: \( m \)-element vectors \( A, B, C \)

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

Pictorially:
Introduction to STREAM Triad

Given: \( m \)-element vectors \( A, B, C \)

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

Pictorially (in parallel):

\[
\begin{align*}
A & \quad = & \quad = & \quad = & \quad = & \quad = \\
B & \quad + & \quad + & \quad + & \quad + & \quad + \\
C & \quad * & \quad * & \quad * & \quad * & \quad * \\
\alpha & \quad & & & & \\
\end{align*}
\]
A Few Versions of STREAM Triad

```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);

    return 0;
}
```
A Few Versions of STREAM Triad

```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_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;
}
```
A Few Versions of STREAM Triad

```
#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 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;
    }
    double scalar; // Initialize scalar
    #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;
}
```

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.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);
}
```

HPC suffers from too many distinct notations for expressing parallelism and locality
A Few Versions of STREAM Triad

### 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);
}
```

### Chapel

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

const ProbSpace = [1..m] dmapped ...

var A, B, C: [ProbSpace] real;
B = ...;
C = ...;
A = B + alpha * C;
```

### MPI + OpenMP

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

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

int HPCC_StartStream(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;
}
```

```
hpcc.h
__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 Do HPC Programming Models Change?

HPC has traditionally given users...
...low-level, control-centric programming models
...ones that are closely tied to the underlying hardware

benefits: lots of control; decent generality; easy to implement
downsides: lots of user-managed detail; brittle to changes

one characterization of Chapel’s goals:

• Raise the level of abstraction to insulate parallel algorithms from underlying hardware when possible/practical
• Yet permit control over such details using appropriate abstraction and separation of concerns
e.g., OpenMP, pthreads

+ support dynamic, fine-grain parallelism
+ considered simpler, more like traditional programming
  - “if you want to access something, simply name it”
  - no support for expressing locality/affinity; limits scalability
  - bugs can be subtle, difficult to track down (race conditions)
  - tend to require complex memory consistency models
e.g., MPI

+ a more constrained model; can only access local data
+ run on most large-scale parallel platforms
  - and for many of them, can achieve near-optimal performance
+ are relatively easy to implement
+ can serve as a strong foundation for higher-level models
+ users are able to get real work done with them
e.g., MPI

- communication must be used to get copies of remote data
  - and tends to reveal too much about *how* to transfer data, not simply *what*
- only supports “cooperating executable”-level parallelism
- couples data transfer and synchronization
- has frustrating classes of bugs of its own
  - e.g., mismatches between sends/recvs, buffer overflows, etc.
Hybrid Programming Models

e.g., MPI+OpenMP, MPI+pthreads, MPI+CUDA, ...

+ support a division of labor: each handles what it does best
+ permit overheads to be amortized across processor cores
  - require multiple distinct notations to express a single logical parallel algorithm, each with its own distinct semantics
e.g., Co-Array Fortran (CAF), Unified Parallel C (UPC)

+ support a shared namespace, like shared-memory
+ support a strong sense of ownership and locality
  • each variable is stored in a particular memory segment
  • tasks can access any visible variable, local or remote
  • local variables are cheaper to access than remote ones
+ implicit communication eases user burden; permits compiler use best mechanisms available
e.g., Co-Array Fortran (CAF), Unified Parallel C (UPC)

- retain many of the downsides of shared-memory
  - error cases, memory consistency models
- restricted to SPMD programming and execution models
- data structures not as flexible/rich as one might like
Chapel: A Next-Generation PGAS Language

- a PGAS language, but non-traditional:
  - more general/dynamic/multithreaded parallelism
  - concepts for composable data and task parallelism
  - distinct concepts for locality vs. parallelism
    - e.g., *locale* type represents architectural locality
  - productivity features
    - type inference, iterator functions, rich array types, OOP, ...
Chapel's Origins

• **HPCS**: High Productivity Computing Systems
  - Overall goal: Raise high-end user productivity by 10x
    \[ \text{Productivity} = \text{Performance} + \text{Programmability} + \text{Portability} + \text{Robustness} \]

• **Phase II**: Cray, IBM, Sun (July 2003 – June 2006)
  - Goal: Propose new productive system architectures
  - Each vendor created a new programming language
    - **Cray**: Chapel
    - **IBM**: X10
    - **Sun**: Fortress

• **Phase III**: Cray, IBM (July 2006 – )
  - Goal: Develop the systems proposed in phase II
  - Each vendor implemented a compiler for their language
    - Sun also continued their Fortress effort without HPCS funding
Chapel's Productivity Goals

• Vastly improve **programmability** over current languages
  • Writing parallel programs
  • Reading, modifying, porting, tuning, maintaining them

• Support **performance** at least as good as MPI
  • Competitive with MPI on generic clusters
  • Better than MPI on more capable architectures

• Improve **portability** over current languages
  • As ubiquitous as MPI but more abstract
  • More portable than OpenMP, UPC, and CAF are thought to be

• Improve **robustness** via improved semantics
  • Eliminate common error cases
  • Provide better abstractions to help avoid other errors
Outline

- Chapel’s Context
- Chapel’s Motivating Themes
  1. General parallel programming
  2. *Global-view* abstractions
  3. *Multiresolution* design
  4. Control over locality/affinity
  5. Reduce gap between mainstream & HPC languages
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 all parallelism available in the hardware

- **Systems**: multicore desktops, clusters, HPC systems, ...
- **Levels**: machines, nodes, cores, instructions
In pictures: “Apply a 3-Point Stencil to a vector”

\[
\text{Global-View}:
\begin{align*}
( & \begin{array}{cccc}
\text{pink} & \text{pink} & \text{purple} & \text{pink} \\
\text{yellow} & \text{yellow} & \text{yellow} & \text{yellow}
\end{array} \\
+ & \begin{array}{cccc}
\text{pink} & \text{pink} & \text{purple} & \text{pink}
\end{array} \\
= & \begin{array}{cccc}
\text{purple} & \text{purple} & \text{purple} & \text{purple}
\end{array}
\end{align*}
\]

\[
\text{Local-View}:
\begin{align*}
\text{pink} & \begin{array}{cccc}
\text{pink} & \text{pink} & \text{purple} & \text{pink}
\end{array} \\
\text{yellow} & \begin{array}{cccc}
\text{yellow} & \text{yellow} & \text{yellow} & \text{yellow}
\end{array} \\
\text{pink} & \begin{array}{cccc}
\text{pink} & \text{pink} & \text{purple} & \text{pink}
\end{array}
\end{align*}
\]
2) Global-View Abstractions

In pictures: “Apply a 3-Point Stencil to a vector”
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 p = numProcs(),
        me = myProc(),
        myN = n/p,
    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[1]);
        recv(me-1, A[0]);
    }
    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) \textit{rprj3} Stencil from NAS MG

\[
\begin{align*}
&\begin{array}{c}
\text{Input} \\
\end{array} \\
&\begin{array}{c}
\text{Output} \\
\end{array} \\
&\begin{array}{c}
\text{Output} \\
\end{array} \\
&\begin{array}{c}
\text{Output} \\
\end{array}
\end{align*}
\]
2) *rprj3* Stencil from NAS MG in Fortran + MPI
Our previous work in ZPL demonstrated that such compact codes can result in better performance than Fortran + MPI while also supporting more flexibility at runtime*

*e.g., the Fortran + MPI *rprj3* code shown previously not only assumes \( p \) divides \( n \), it also assumes that \( p \) and \( n \) are specified at compile-time and powers of two.
## 2) Classifying Current Programming Models

<table>
<thead>
<tr>
<th>System</th>
<th>Data Model</th>
<th>Control Model</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Communication Libraries</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI/MPI-2</td>
<td>Local-View</td>
<td>Local-View</td>
</tr>
<tr>
<td>SHMEM, ARMCI, GASNet</td>
<td>Local-View</td>
<td>SPMD</td>
</tr>
<tr>
<td><strong>Shared Memory</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenMP, Pthreads</td>
<td>Global-View (trivially)</td>
<td>Global-View (trivially)</td>
</tr>
<tr>
<td><strong>PGAS Languages</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Co-Array Fortran</td>
<td>Local-View</td>
<td>SPMD</td>
</tr>
<tr>
<td>UPC</td>
<td>Global-View</td>
<td>SPMD</td>
</tr>
<tr>
<td>Titanium</td>
<td>Local-View</td>
<td>SPMD</td>
</tr>
<tr>
<td><strong>PGAS Libraries</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Global Arrays</td>
<td>Global-View</td>
<td>SPMD</td>
</tr>
</tbody>
</table>
### 2) Classifying Current Programming Models

<table>
<thead>
<tr>
<th>System</th>
<th>Data Model</th>
<th>Control Model</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Communication Libraries</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI/MPI-2</td>
<td>Local-View</td>
<td>Local-View</td>
</tr>
<tr>
<td>SHMEM, ARMCI, GASNet</td>
<td>Local-View</td>
<td>SPMD</td>
</tr>
<tr>
<td><strong>Shared Memory</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenMP, Pthreads</td>
<td>Global-View (trivially)</td>
<td>Global-View (trivially)</td>
</tr>
<tr>
<td><strong>PGAS Languages</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Co-Array Fortran</td>
<td>Local-View</td>
<td>SPMD</td>
</tr>
<tr>
<td>UPC</td>
<td>Global-View</td>
<td>SPMD</td>
</tr>
<tr>
<td>Titanium</td>
<td>Local-View</td>
<td>SPMD</td>
</tr>
<tr>
<td><strong>PGAS Libraries</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Global Arrays</td>
<td>Global-View</td>
<td>SPMD</td>
</tr>
<tr>
<td><strong>HPCS Languages</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chapel</td>
<td>Global-View</td>
<td>Global-View</td>
</tr>
<tr>
<td>X10 (IBM)</td>
<td>Global-View</td>
<td>Global-View</td>
</tr>
<tr>
<td>Fortress (Sun)</td>
<td>Global-View</td>
<td>Global-View</td>
</tr>
</tbody>
</table>
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(me, p) {
  ...
}
```
3) Multiresolution Language Design: Motivation

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

“Why don’t I have more control?”
**Multiresolution Design**: Support multiple tiers of features

- higher levels for programmability, productivity
- lower levels for performance, control
- build the higher-level concepts in terms of the lower

*Chapel language concepts*

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

- separate concerns appropriately for clean design
Consider:

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

Therefore:

- Placement of data relative to computation affects scalability
- Give programmers control of data and task placement

Note:

- As core counts grow, locality will matter more on desktops
- GPUs and accelerators already expose node-level locality
Consider:

- Students graduate with training in Java, Matlab, Perl, Python
- 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 ostracizing the traditional HPC programmer
  - e.g., support object-oriented programming, but make it optional
Questions?

- Chapel’s Context
- Chapel’s Motivating Themes
  1. General parallel programming
  2. Global-view abstractions
  3. Multiresolution design
  4. Control over locality/affinity
  5. Reduce gap between mainstream & HPC languages